Бесплатная электронная книга

УЧУСЬ

algorithm

Free unaffiliated eBook created from
Stack Overflow contributors.

#algorithm

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: algorithm

It is an unofficial and free algorithm ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official algorithm.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с алгоритмом

замечания

Введение в алгоритмы

Алгоритмы повсеместно распространены в области компьютерных наук и разработки программного обеспечения. Выбор соответствующих алгоритмов и структур данных повышает эффективность нашей программы в стоимости и времени.

Что такое алгоритм? Неформально алгоритм является процедурой для выполнения конкретной задачи. [Skiena: 2008: ADM: 1410219] В частности, алгоритм представляет собой *четко определенную* вычислительную процедуру, которая принимает некоторое значение (или набор значений) в качестве **входных данных** и выдает некоторое значение или набор значений в качестве **вывода** . Таким образом, алгоритм представляет собой последовательность этапов вычисления, которые преобразуют входной сигнал в выходной сигнал. Cormen et. и др. явно не замечает, что алгоритм не обязательно требует ввода. [Кормен: 2001: IA: 580470]

Формально алгоритм должен удовлетворять пяти особенностям: [Knuth: 1997: ACP: 260999]

1. *Конечность* . Алгоритм должен всегда заканчиваться после конечного числа шагов.
2. *Определенность* . Каждый шаг алгоритма должен быть точно определен; действия, которые необходимо выполнить, должны быть строго определены. Именно это качество [Cormen: 2001: IA: 580470] относится к термину «четко определенный».
3. *Вход* . Алгоритм имеет ноль или более *входных данных* . Это величины, которые задаются алгоритму первоначально до его начала или динамически по мере его запуска.
4. *Выход* . Алгоритм имеет один или несколько *выходов* . Это величины, которые имеют определенное отношение к входам. Мы ожидаем, что алгоритм выдаст один и тот же результат при повторении одного и того же ввода.
5. *Эффективность* . Как правило, ожидается, что алгоритм будет *эффективным* . Его операции должны быть достаточно базовыми, чтобы их можно было выполнять в принципе и за конечный промежуток времени кем-то, использующим карандаш и бумагу.

Процедура, которая не имеет конечности, но удовлетворяющая всем остальным характеристикам алгоритма, может быть названа *вычислительным методом* . [Кнут: 1997: ACP: 260999]

Examples

Пример алгоритмической задачи

Алгоритмическая проблема задается путем описания полного набора *экземпляров,* над которыми он должен работать, и их вывода после запуска в одном из этих экземпляров. Это различие между проблемой и экземпляром проблемы является основополагающим. Алгоритмическая *проблема,* известная как *сортировка* , определяется следующим образом: [Skiena: 2008: ADM: 1410219]

- Проблема: Сортировка
- Вход: последовательность из *n* ключей, `a_1, a_2, ..., a_n` .
- Выход: переупорядочение входной последовательности так, что `a'_1 <= a'_2 <= ... <= a'_{n-1} <= a'_n`

Экземпляром сортировки может быть массив строк, например `{ Haskell, Emacs }` или последовательность чисел, например `{ 154, 245, 1337 }` .

Начало работы с Simple Fizz Buzz Algorithm in Swift

Для тех из вас, кто новичок в программировании в Swift, и для тех из вас, которые исходят из разных баз программирования, таких как Python или Java, эта статья должна быть весьма полезной. В этом посте мы обсудим простое решение для реализации быстрых алгоритмов.

Fizz Buzz

Возможно, вы видели Fizz Buzz, написанную как Fizz Buzz, FizzBuzz или Fizz-Buzz; все они относятся к одному и тому же. Эта «вещь» является главной темой сегодняшнего обсуждения. Во-первых, что такое FizzBuzz?

Это общий вопрос, который возникает в собеседовании.

Представьте себе серию чисел от 1 до 10.

```
1 2 3 4 5 6 7 8 9 10
```

Fizz и Buzz относятся к любому числу, которое кратно 3 и 5 соответственно. Другими словами, если число делится на 3, оно заменяется на fizz; если число делится на 5, оно заменяется шумом. Если число одновременно кратно 3 И 5, число заменяется на «fizz buzz». По сути, он эмулирует знаменитую игру детей «fizz buzz».

Чтобы решить эту проблему, откройте Xcode, чтобы создать новую игровую площадку и инициализировать массив, как показано ниже:

```
// for example
let number  = [1,2,3,4,5]
// here 3 is fizz and 5 is buzz
```

Чтобы найти все fizz и buzz, мы должны перебирать массив и проверять, какие числа являются fizz, а какие - шум. Для этого создайте цикл for для итерации через инициализированный массив:

```
for num in number {
  // Body and calculation goes here
}
```

После этого мы можем просто использовать условие «if else» и оператор модуля в swift ie - %, чтобы найти fizz и buzz

```
for num in number {
  if num % 3 == 0 {
    print("\(num) fizz")
  } else {
    print(num)
  }
}
```

Большой! Вы можете перейти на консоль отладки на игровой площадке Xcode, чтобы увидеть выход. Вы обнаружите, что «шипы» были отсортированы в вашем массиве.

Для части Buzz мы будем использовать ту же технику. Попробуем попробовать, прежде чем прокручивать статью - вы можете проверить свои результаты против этой статьи, как только закончите это делать.

```
for num in number {
  if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
    print(num)
  }
}
```

Проверьте выход!

Это довольно прямолинейно - вы разделили число на 3, fizz и разделили число на 5, гул. Теперь увеличьте числа в массиве

```
let number = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

Мы увеличили диапазон чисел от 1-10 до 1-15, чтобы продемонстрировать концепцию «fizz buzz». Поскольку 15 кратно и 3, и 5, число должно быть заменено на «fizz buzz». Попробуйте сами и проверьте ответ!

Вот решение:

```
for num in number {
  if num % 3 == 0 && num % 5 == 0 {
    print("\(num) fizz buzz")
  } else if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
    print(num)
  }
}
```

Подожди ... это еще не конец! Вся цель алгоритма заключается в правильной настройке среды выполнения. Представьте, если диапазон увеличивается от 1-15 до 1-100. Компилятор проверяет каждый номер, чтобы определить, делится ли он на 3 или 5. Затем он снова пробегает числа, чтобы проверить, делятся ли числа на 3 и 5. Код, по существу, должен будет проходить через каждое число в массиве дважды - сначала нужно будет выполнить число на 3, а затем запустить его на 5. Чтобы ускорить процесс, мы можем просто сказать, что наш код будет делить числа на 15 напрямую.

Вот окончательный код:

```
for num in number {
  if num % 15 == 0 {
    print("\(num) fizz buzz")
  } else if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
    print(num)
  }
}
```

Как простой, вы можете использовать любой язык по вашему выбору и приступить к работе

Наслаждайтесь кодированием

Прочитайте Начало работы с алгоритмом онлайн: https://riptutorial.com/ru/algorithm/topic/757/начало-работы-с-алгоритмом

глава 2: A * Pathfinding

Examples

Введение в A *

A * (звезда) - это алгоритм поиска, который используется для поиска пути от одного узла к другому. Таким образом, его можно сравнить с Breadth First Search , или с алгоритмом Дейкстры , или с глубиной первого поиска , или с помощью Best First Search. Алгоритм A * широко используется в поиске графов для повышения эффективности и точности, где предварительная обработка графа не является вариантом.

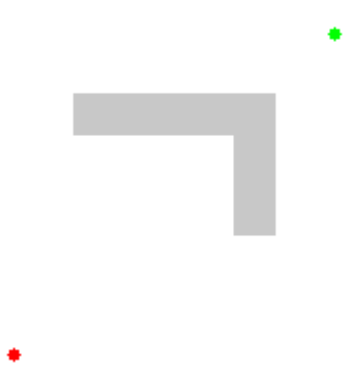
A * является специализацией Best First Search, в которой функция оценки *f* определяется определенным образом.

f (n) = g (n) + h (n) - минимальная стоимость с начального узла до целей, обусловленных движением мыслительного узла *n* .

g (n) - минимальная стоимость от начального узла до *n* .

h (n) - минимальная стоимость от *n* до ближайшей цели до *n*

A * - это информированный алгоритм поиска, и он всегда гарантирует наименьший путь (путь с минимальной стоимостью) в наименьшее возможное время (если используется допустимая эвристика). Таким образом, он является *полным* и *оптимальным* . Следующая анимация демонстрирует A * search-

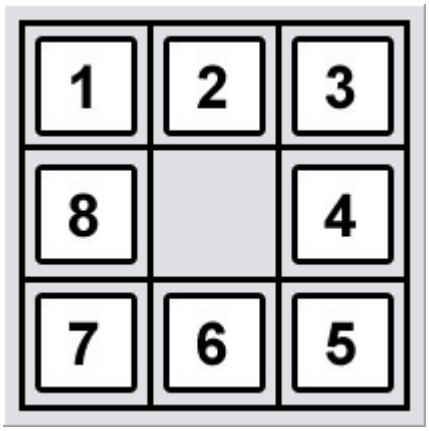


Решение 8-головоломок с использованием алгоритма A *

Определение проблемы :

8-головоломка - простая игра, состоящая из сетки 3 x 3 (содержащей 9 квадратов). Один из квадратов пуст. Объект состоит в том, чтобы перемещаться в квадраты в разные

положения и иметь номера, отображаемые в «состоянии цели».



Учитывая начальное состояние игры с 8 играми и конечное состояние, которое нужно достичь, найдите наиболее рентабельный путь для достижения конечного состояния из исходного состояния.

Исходное состояние :

```
_ 1 3
4 2 5
7 8 6
```

Конечное состояние :

```
1 2 3
4 5 6
7 8 _
```

Предполагается эвристика :

Рассмотрим Манхэттенское расстояние между текущим и конечным состоянием как эвристику для этой постановки задачи.

```
h(n) = | x - p | + | y - q |
where x and y are cell co-ordinates in the current state
      p and q are cell co-ordinates in the final state
```

Общая функция затрат :

Таким образом, общая функция стоимости `f(n)` задается,

```
f(n) = g(n) + h(n), where g(n) is the cost required to reach the current state from given
initial state
```

Решение проблемы :

Сначала мы находим эвристическое значение, необходимое для достижения конечного

состояния из начального состояния. Функция стоимости, g (n) = 0, поскольку мы находимся в начальном состоянии

```
h(n) = 8
```

Вышеуказанное значение получается, так как `1` в текущем состоянии равно 1 горизонтальному расстоянию, чем `1` в конечном состоянии. То же самое касается `2` , `5` , `6` . _ 2 горизонтальных расстояния и 2 вертикальных расстояния. Таким образом, общее значение для `h(n)` равно 1 + 1 + 1 + 1 + 2 + 2 = 8. Общая стоимость `f(n)` равна 8 + 0 = 8.

Теперь, возможные состояния , которые могут быть достигнуты из начального состояния встречаются и случается , что мы можем либо двигаться _ вправо или вниз.

Таким образом, состояния, полученные после перемещения этих ходов, следующие:

```
1 _ 3    4 1 3
4 2 5    _ 2 5
7 8 6    7 8 6
 (1)      (2)
```

Опять же, общая функция стоимости вычисляется для этих состояний с использованием метода, описанного выше, и получается, что она равна 6 и 7 соответственно. Мы выбрали состояние с минимальной стоимостью, которая является состоянием (1). Следующие возможные шаги могут быть влево, вправо или влево. Мы не будем двигаться влево, как мы были ранее в этом состоянии. Таким образом, мы можем двигаться вправо или влево.

Снова мы находим состояния, полученные из (1).

```
1 3 _    1 2 3
4 2 5    4 _ 5
7 8 6    7 8 6
 (3)      (4)
```

(3) приводит к функции стоимости, равной 6, и (4) приводит к 4. Кроме того, мы рассмотрим (2), полученную до того, какая функция стоимости равна 7. Выбор минимального из них приводит к (4). Следующие возможные перемещения могут быть влево или вправо или влево. Мы получаем состояния:

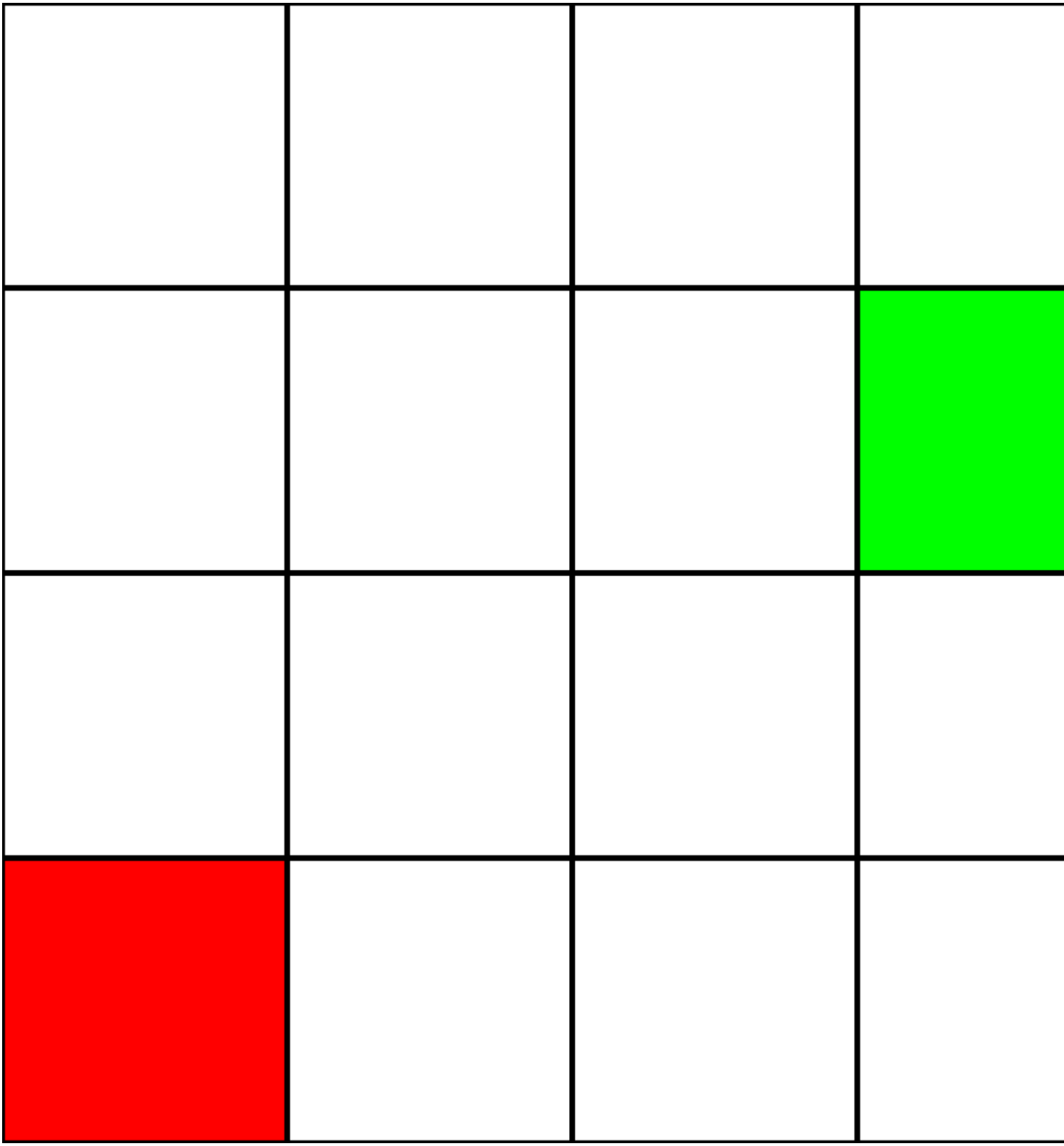
```
1 2 3    1 2 3    1 2 3
_ 4 5    4 5 _    4 8 5
7 8 6    7 8 6    7 _ 6
 (5)      (6)      (7)
```

Мы получаем затраты, равные 5, 2 и 4 для (5), (6) и (7) соответственно. Кроме того, мы имеем предыдущие состояния (3) и (2) с 6 и 7 соответственно. Мы выбрали минимальное состояние затрат, которое равно (6). Следующие возможные шаги: Вверх, Вниз и ясно вниз приводят нас к окончательному состоянию, приводящему к значению эвристической

функции, равному 0.

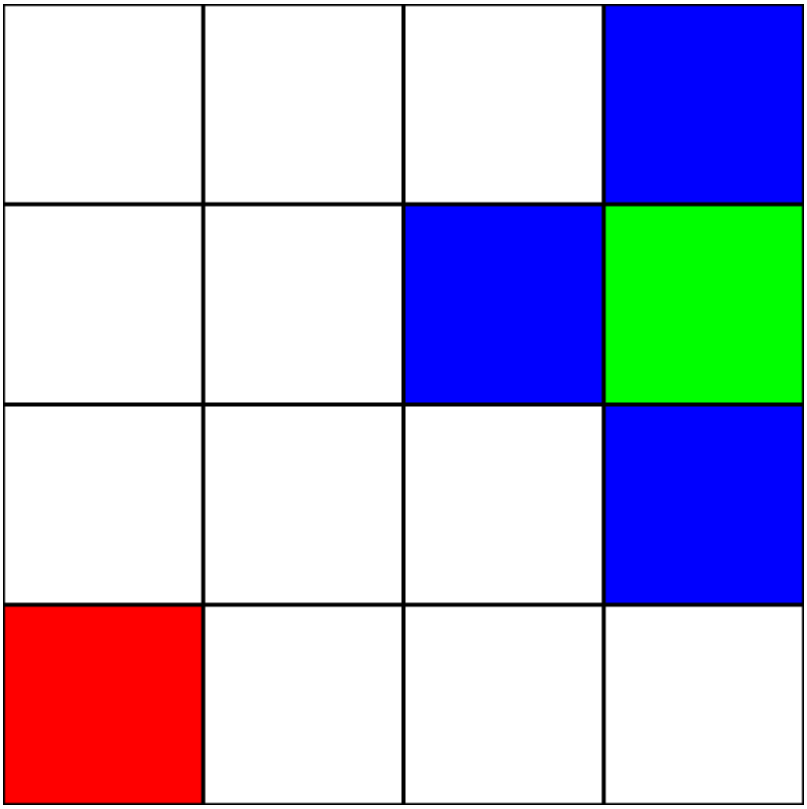
A * Прослеживание через лабиринт без препятствий

Предположим, у нас есть следующая сетка 4 на 4:



Предположим, что это *лабиринт* . Однако никаких препятствий нет. У нас есть только начальная точка (зеленый квадрат) и конечная точка (красный квадрат). Предположим

также, что для перехода от зеленого к красному мы не можем двигаться по диагонали. Итак, начиная с зеленой площади, давайте посмотрим, на какие квадраты мы можем переместиться, и выделим их синим цветом:



Чтобы выбрать квадрат для перехода к следующему, нам нужно учитывать 2 эвристики:

1. Значение «g» - это то, как далеко этот узел находится от зеленого квадрата.
2. Значение «h» - это то, как далеко этот узел находится от красного квадрата.
3. Значение «f» - это сумма значения «g» и «h». Это окончательное число, которое указывает нам, к какому узлу нужно перейти.

Чтобы вычислить эти эвристики, это формула, которую мы будем использовать: `distance = abs(from.x - to.x) + abs(from.y - to.y)`

Это называется формулой «Манхэттен Дистанция» .

Вычислим значение «g» для синего квадрата сразу слева от зеленого квадрата: `abs(3 - 2) + abs(2 - 2) = 1`

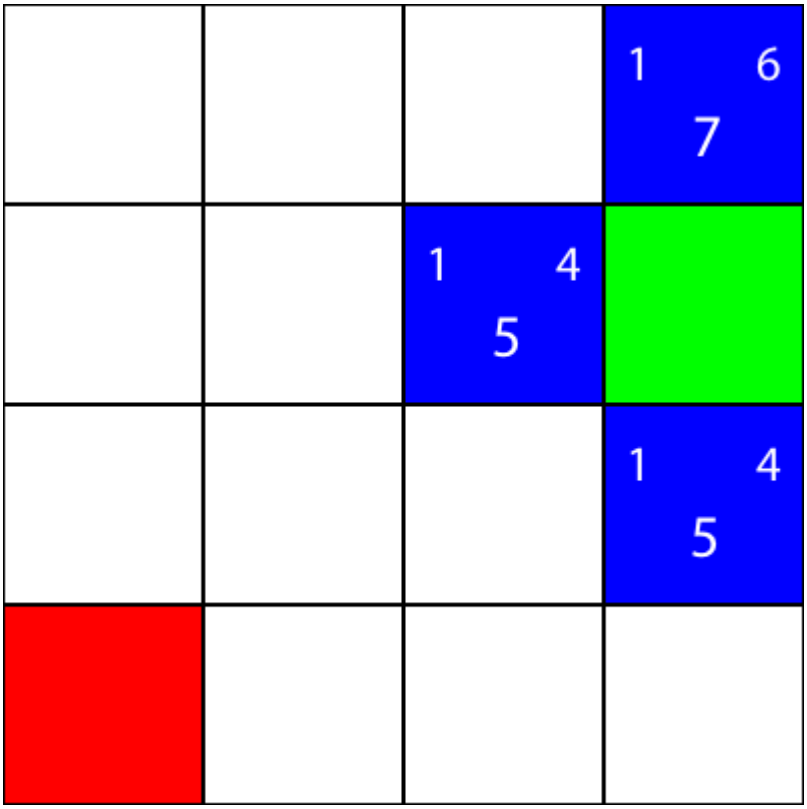
Большой! У нас есть значение: 1. Теперь давайте попробуем вычислить значение «h»: `abs(2 - 0) + abs(2 - 0) = 4`

Отлично. Теперь давайте получим значение «f»: `1 + 4 = 5`

Итак, окончательное значение для этого узла равно «5».

Давайте сделаем то же самое для всех остальных синих квадратов. Большое число в центре каждого квадрата - это значение «f», а число в левом верхнем углу - это значение «

g», а число в правом верхнем углу - это значение «h»:



Мы вычислили значения g, h и f для всех синих узлов. Теперь, что мы выбираем?

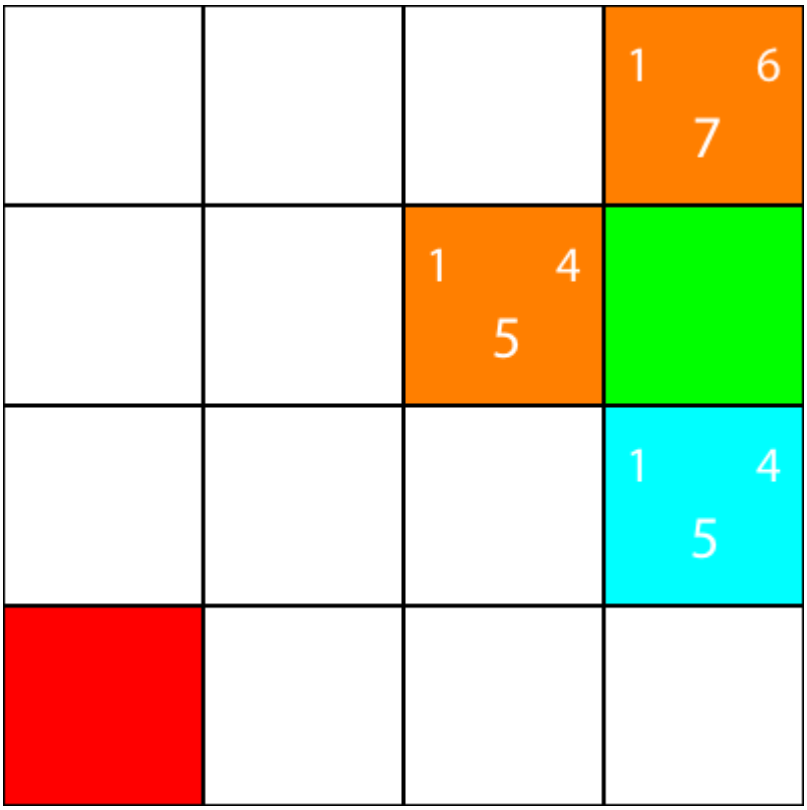
Какое бы ни было наименьшее значение f.

Однако в этом случае мы имеем 2 узла с одинаковым значением f, 5. Как мы можем выбрать между ними?

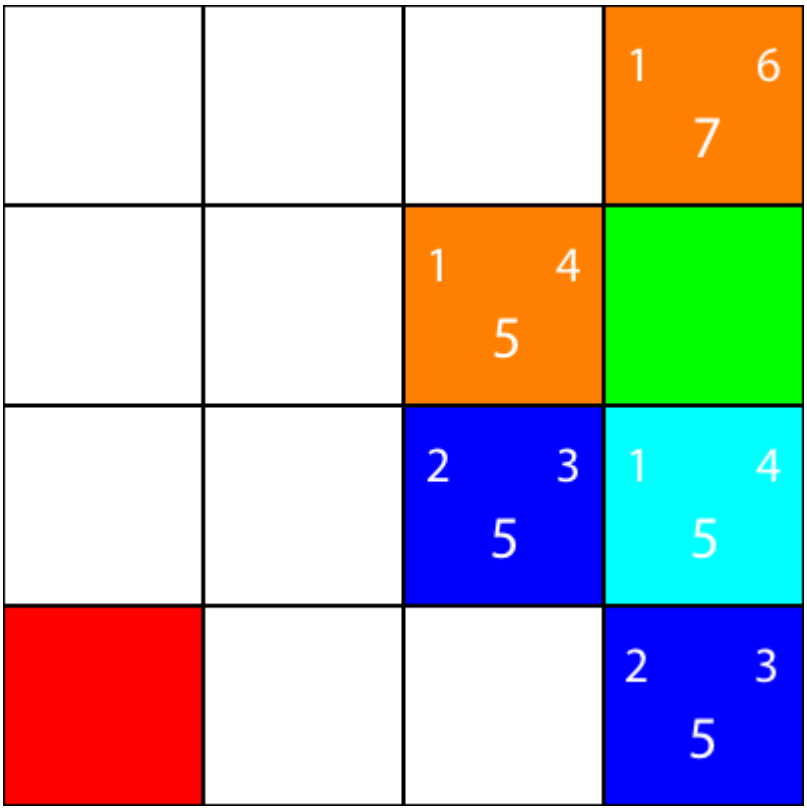
Просто выберите либо наугад, либо установите приоритет. Обычно я предпочитаю иметь такой приоритет: «Вправо> Вверх> Вниз> Влево»

Один из узлов с значением f 5 принимает нас в направлении «Вниз», а другой берет нас «влево». Поскольку Down имеет более высокий приоритет, чем Left, мы выбираем квадрат, который берет нас «вниз».

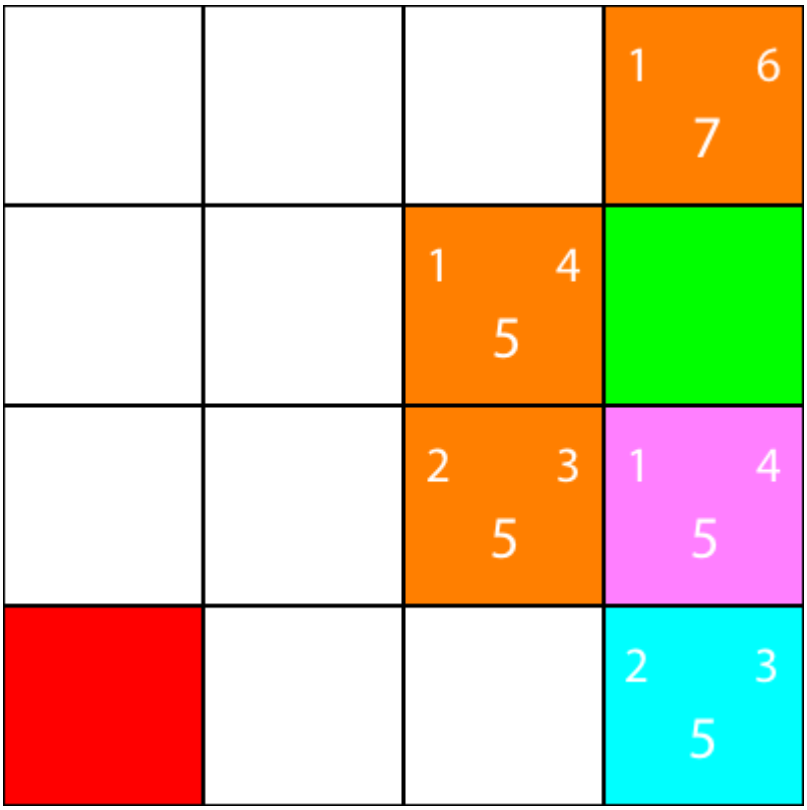
Теперь я отмечаю узлы, для которых мы вычислили эвристику, но не переместились в оранжевый цвет и узел, выбранный нами как голубой:



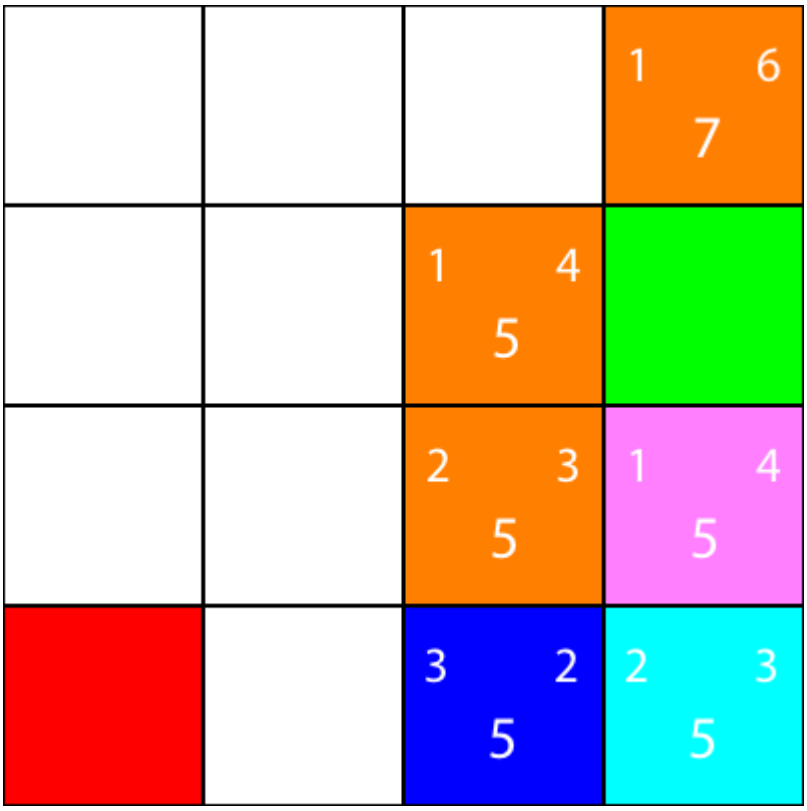
Хорошо, теперь давайте рассчитаем ту же эвристику для узлов вокруг голубого узла:



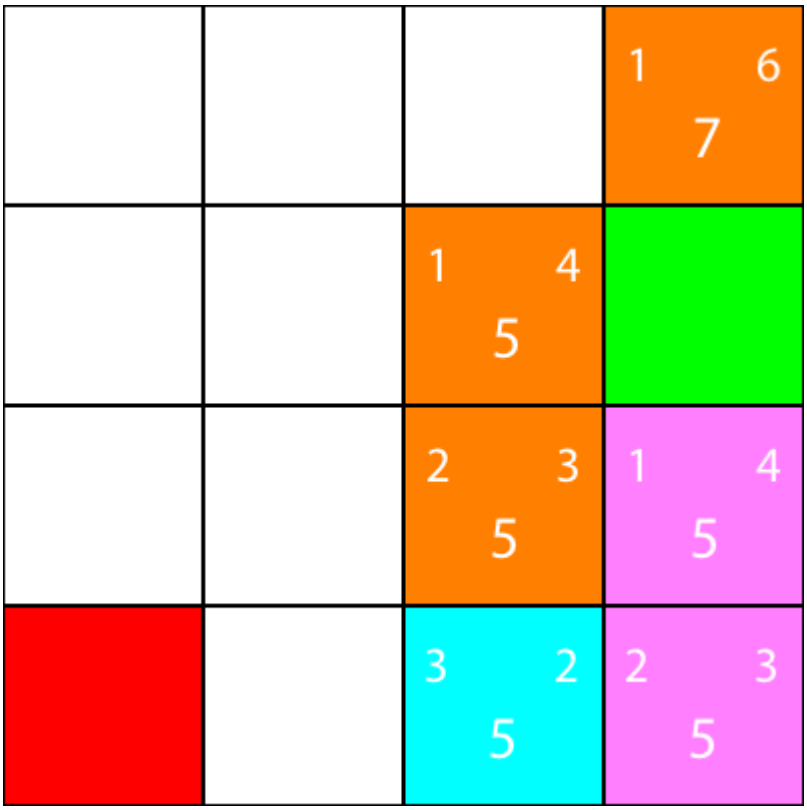
Опять же, мы выбираем узел, идущий вниз от голубого узла, так как все параметры имеют одно и то же значение f:



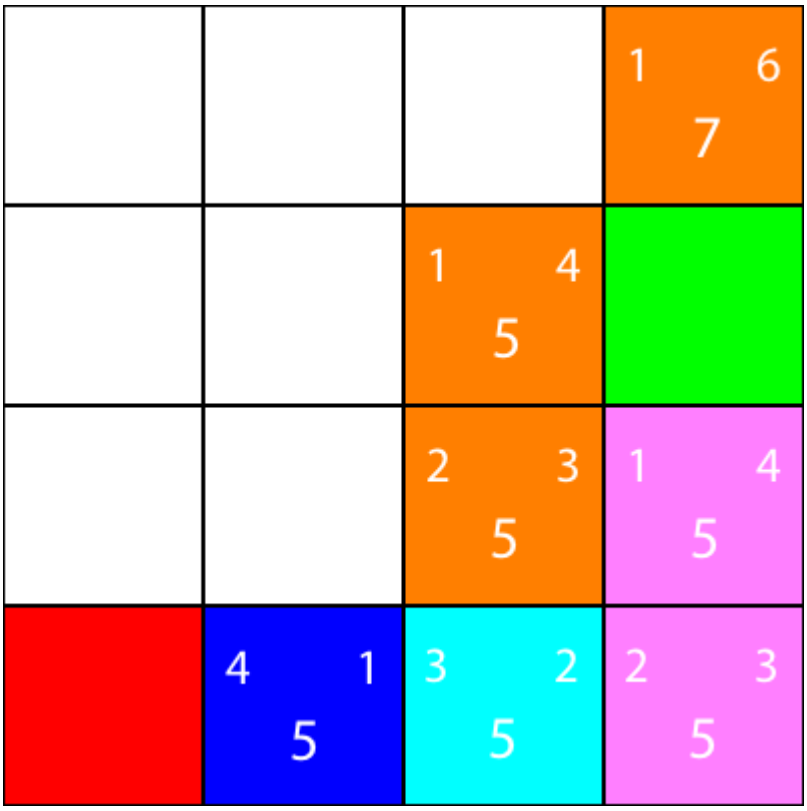
Давайте вычислим эвристику для единственного соседа, который имеет голубой узел:



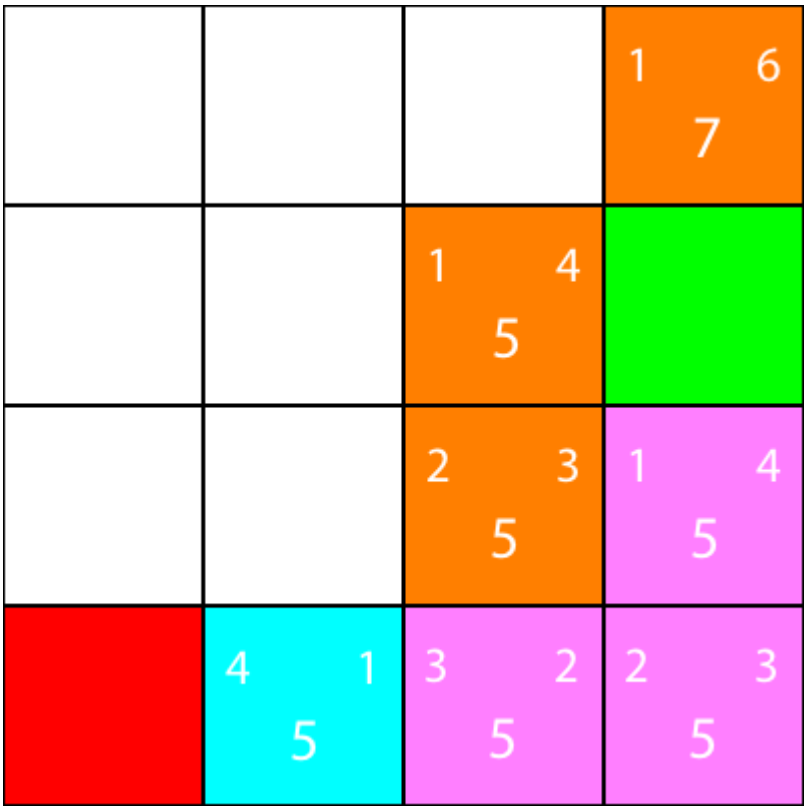
Хорошо, поскольку мы будем следовать той же схеме, которую мы придерживаемся:



Еще раз, давайте вычислим эвристику для соседа узла:



Перейдем туда:



Наконец, мы видим, что у нас есть *квадрат победы* рядом с нами, поэтому мы переезжаем туда, и мы закончили.

Прочитайте A * Pathfinding онлайн: https://riptutorial.com/ru/algorithm/topic/7439/a---pathfinding

глава 3: A * Алгоритм поиска пути

Вступление

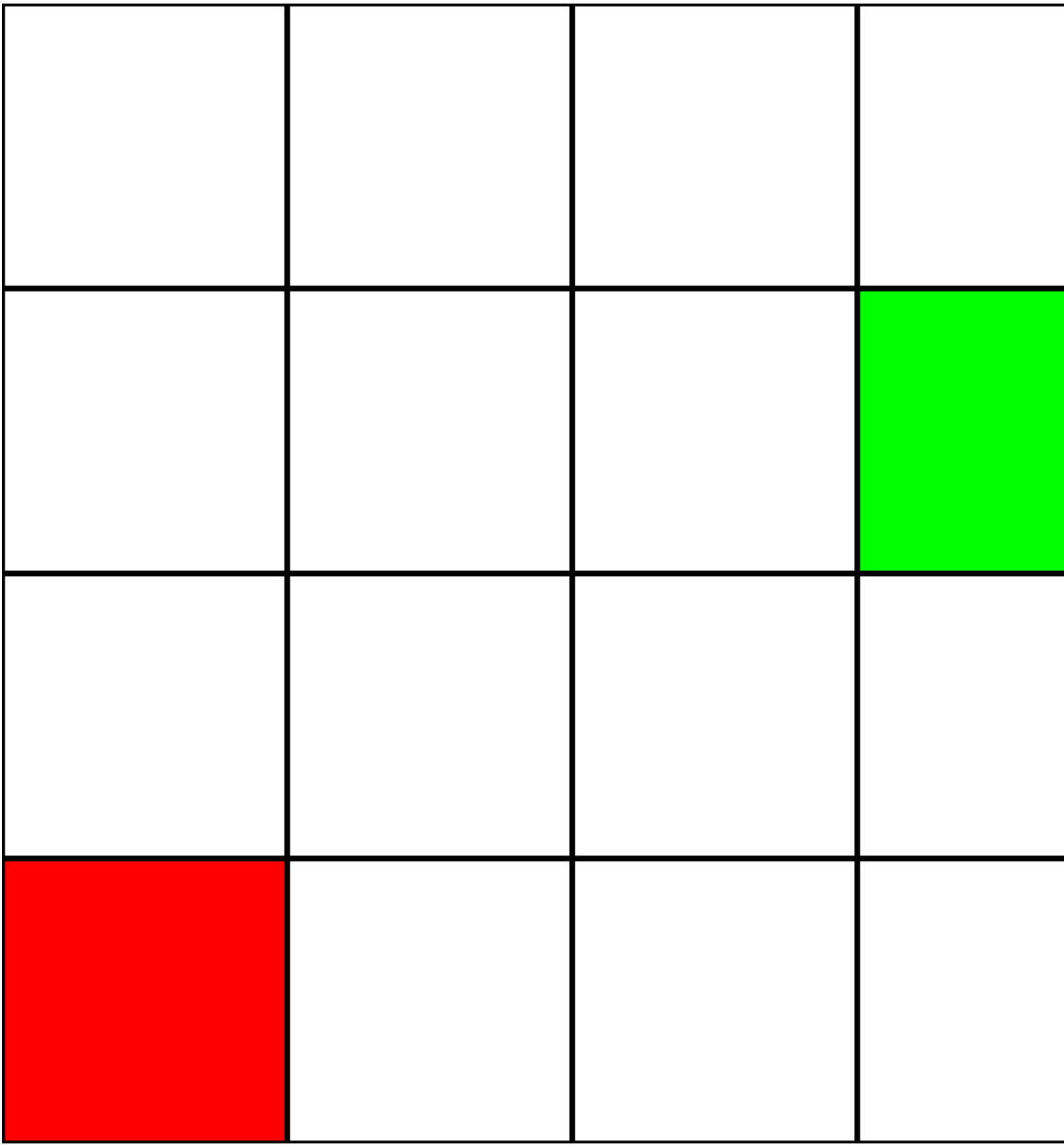
В этом разделе будет рассмотрен алгоритм A * Pathfinding, как он используется и почему он работает.

Обратите внимание на будущих авторов: я добавил пример для A * Pathfinding без каких-либо препятствий на сетке 4x4. Пример с препятствиями по-прежнему необходим.

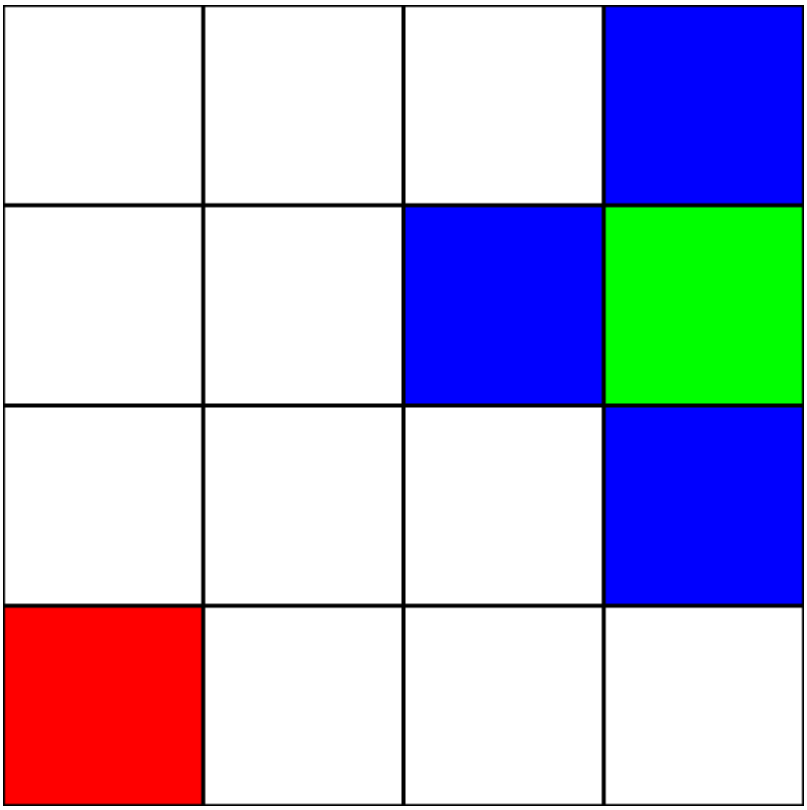
Examples

Простой пример A * Pathfinding: лабиринт без препятствий

Предположим, у нас есть следующая сетка 4 на 4:



Предположим, что это *лабиринт* . Однако никаких препятствий нет. У нас есть только начальная точка (зеленый квадрат) и конечная точка (красный квадрат). Предположим также, что для перехода от зеленого к красному мы не можем двигаться по диагонали. Итак, начиная с зеленой площади, давайте посмотрим, на какие квадраты мы можем переместиться, и выделим их синим цветом:



Чтобы выбрать квадрат для перехода к следующему, нам нужно учитывать 2 эвристики:

1. Значение «g» - это то, как далеко этот узел находится от зеленого квадрата.
2. Значение «h» - это то, как далеко этот узел находится от красного квадрата.
3. Значение «f» - это сумма значения «g» и «h». Это окончательное число, которое указывает нам, к какому узлу нужно перейти.

Чтобы вычислить эти эвристики, это формула, которую мы будем использовать: `distance = abs(from.x - to.x) + abs(from.y - to.y)`

Это называется формулой «Манхэттен Дистанция» .

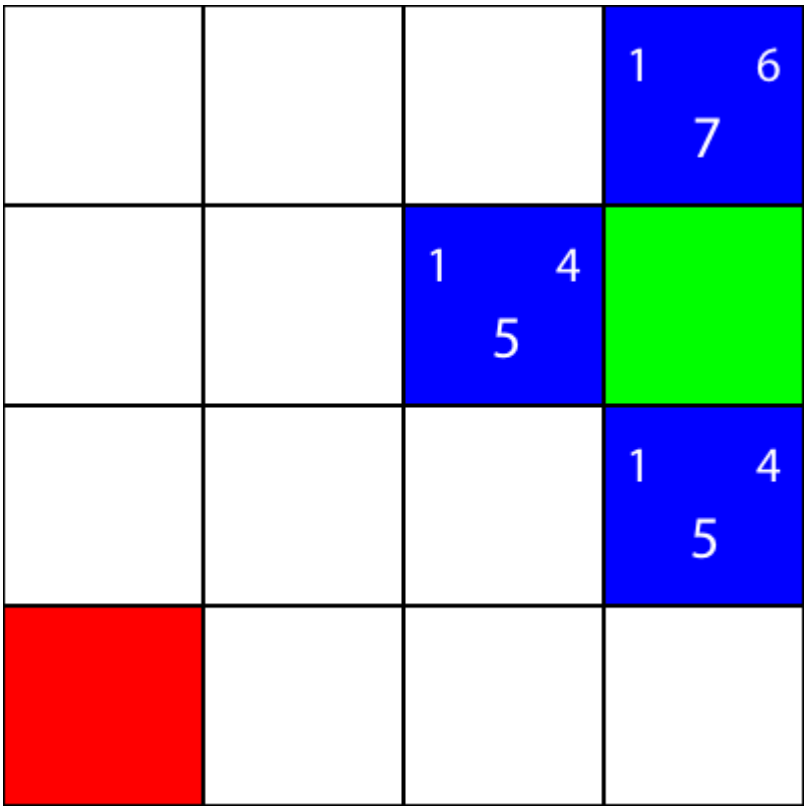
Вычислим значение «g» для синего квадрата сразу слева от зеленого квадрата: `abs(3 - 2) + abs(2 - 2) = 1`

Большой! У нас есть значение: 1. Теперь давайте попробуем вычислить значение «h»: `abs(2 - 0) + abs(2 - 0) = 4`

Отлично. Теперь давайте получим значение «f»: `1 + 4 = 5`

Итак, окончательное значение для этого узла равно «5».

Давайте сделаем то же самое для всех остальных синих квадратов. Большое число в центре каждого квадрата - это значение «f», а число в левом верхнем углу - это значение « g», а число в правом верхнем углу - это значение «h»:



Мы вычислили значения g, h и f для всех синих узлов. Теперь, что мы выбираем?

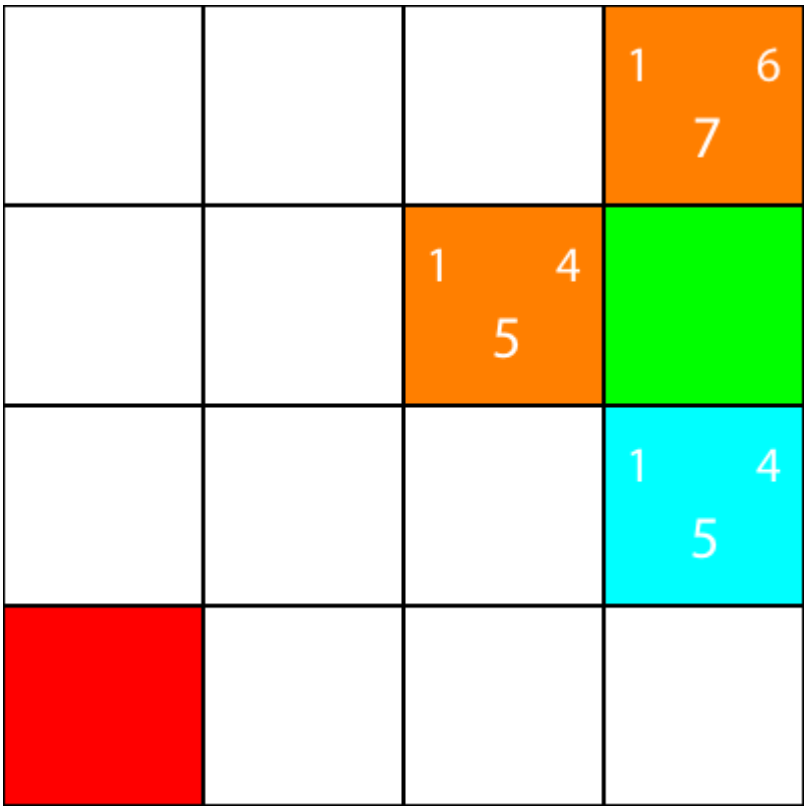
Какое бы ни было наименьшее значение f.

Однако в этом случае мы имеем 2 узла с одинаковым значением f, 5. Как мы можем выбрать между ними?

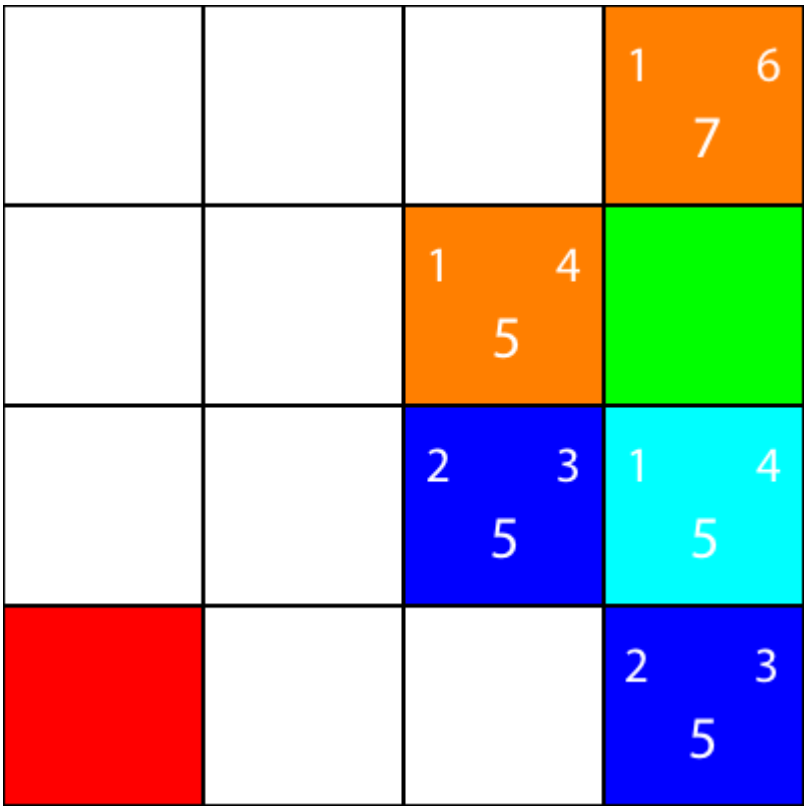
Просто выберите либо наугад, либо установите приоритет. Обычно я предпочитаю иметь такой приоритет: «Вправо> Вверх> Вниз> Влево»

Один из узлов с значением f 5 принимает нас в направлении «Вниз», а другой берет нас «влево». Поскольку Down имеет более высокий приоритет, чем Left, мы выбираем квадрат, который берет нас «вниз».

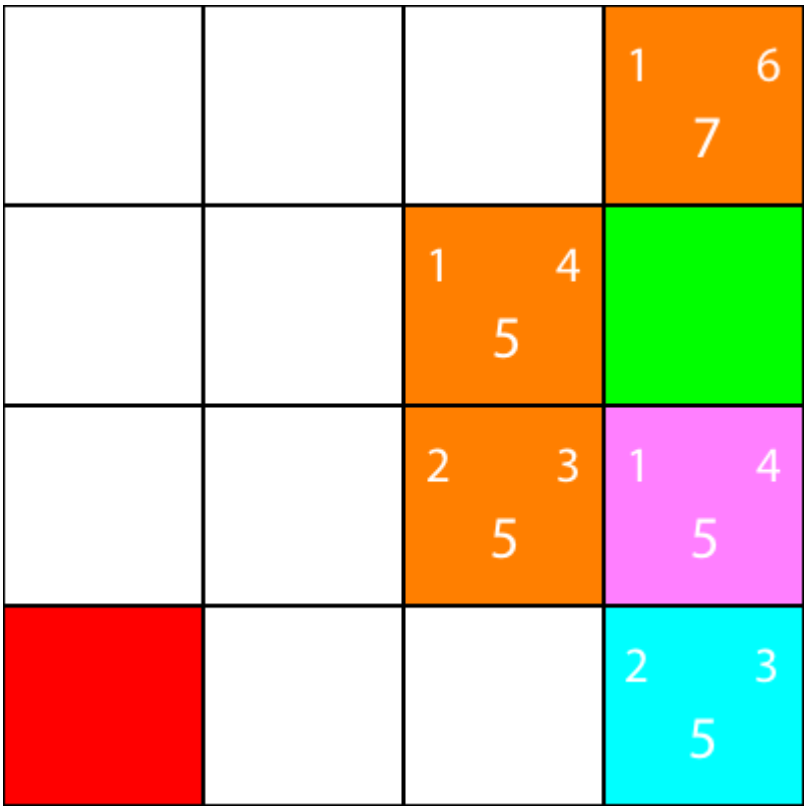
Теперь я отмечаю узлы, для которых мы вычислили эвристику, но не переместились в оранжевый цвет и узел, выбранный нами как голубой:



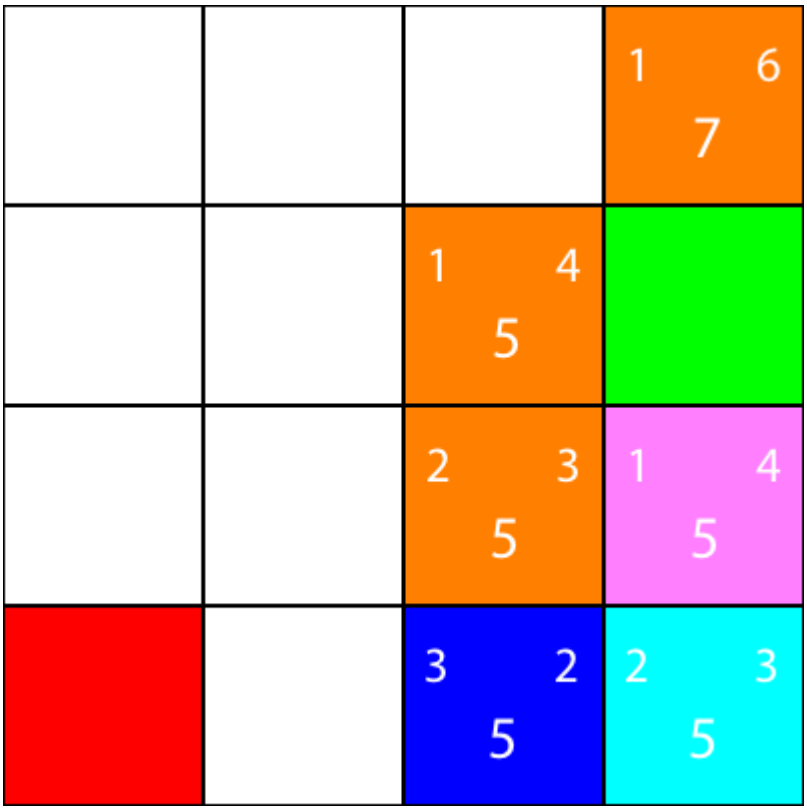
Хорошо, теперь давайте рассчитаем ту же эвристику для узлов вокруг голубого узла:



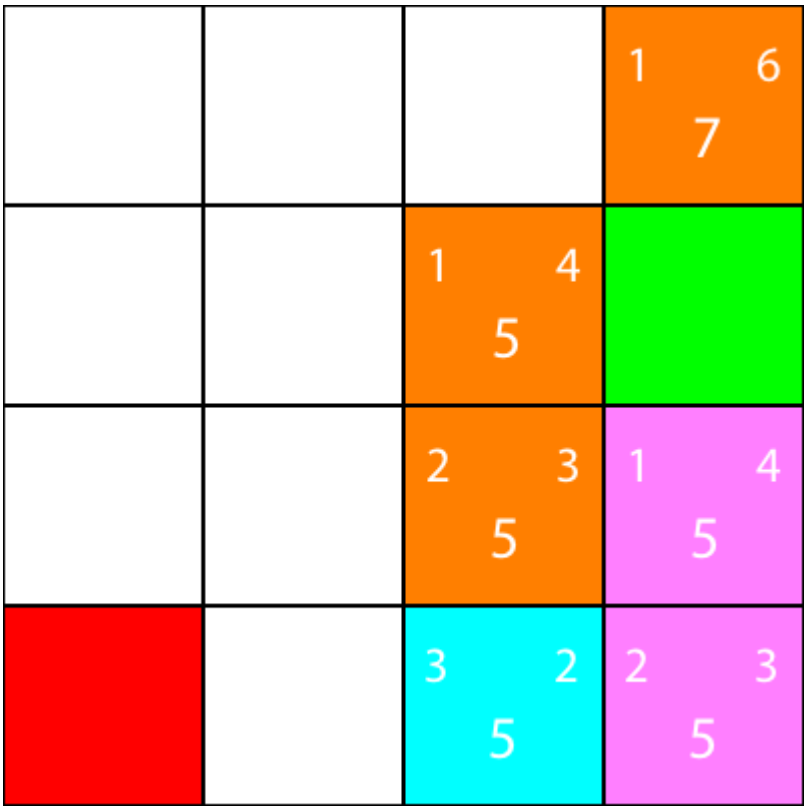
Опять же, мы выбираем узел, идущий вниз от голубого узла, так как все параметры имеют одно и то же значение f:



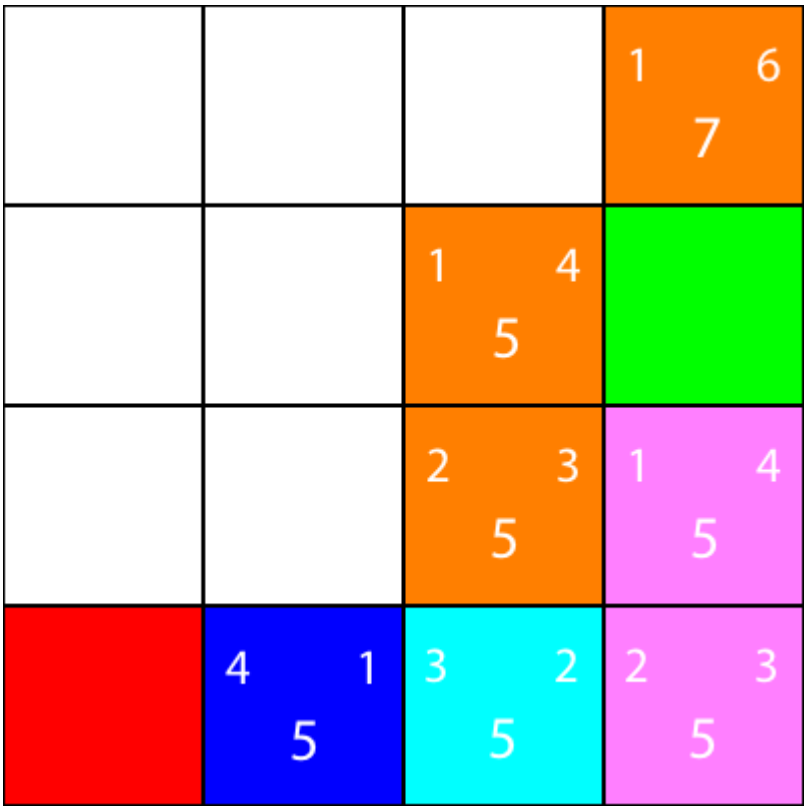
Давайте вычислим эвристику для единственного соседа, который имеет голубой узел:



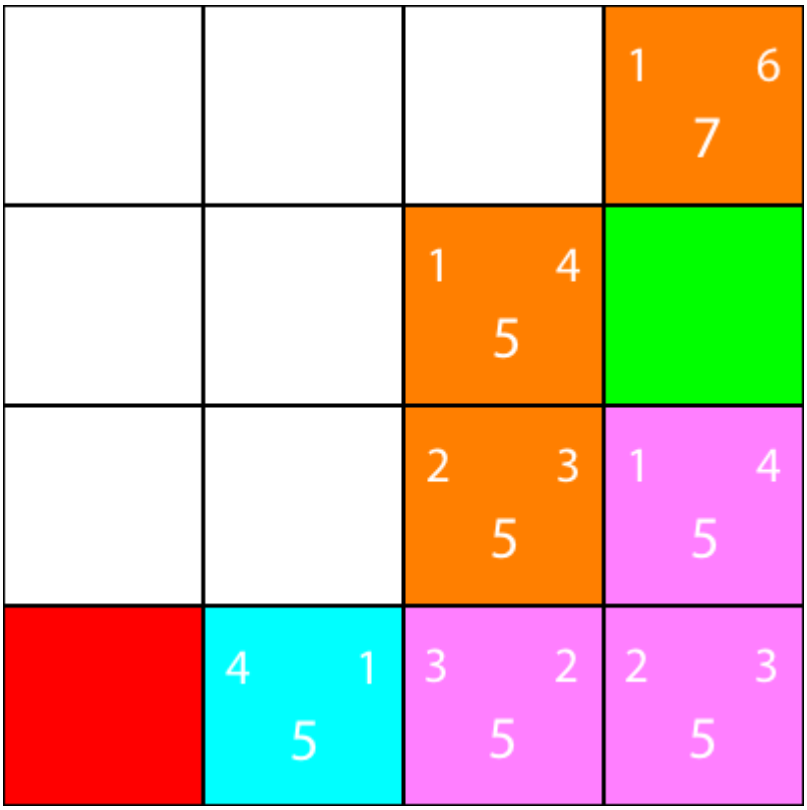
Хорошо, поскольку мы будем следовать той же схеме, которую мы придерживаемся:



Еще раз, давайте вычислим эвристику для соседа узла:



Перейдем туда:



Наконец, мы видим, что у нас есть *квадрат победы* рядом с нами, поэтому мы переезжаем туда, и мы закончили.

Прочитайте A * Алгоритм поиска пути онлайн: https://riptutorial.com/ru/algorithm/topic/8787/a--алгоритм-поиска-пути

глава 4: Algo: - Печатать am * n матрицу в квадратной форме

Вступление

Проверьте ввод и вывод образцов ниже.

Examples

Пример примера

```
Input :-

14 15 16 17 18 21
19 10 20 11 54 36
64 55 44 23 80 39
91 92 93 94 95 42

Output:-
print value in index
14 15 16 17 18 21 36 39 42 95 94 93 92 91 64 19 10 20 11 54 80 23 44 55

or print index
00 01 02 03 04 05 15 25 35 34 33 32 31 30 20 10 11 12 13 14 24 23 22 21
```

Напишите общий код

```
function noOfLooping(m,n) {
    if(m > n) {
        smallestValue = n;
    } else {
        smallestValue = m;
    }

    if(smallestValue % 2 == 0) {
            return smallestValue/2;
    } else {
            return (smallestValue+1)/2;
    }
}

function squarePrint(m,n) {
    var looping = noOfLooping(m,n);
    for(var i = 0; i < looping; i++) {
        for(var j = i; j < m - 1 - i; j++) {
                console.log(i+''+j);
        }
        for(var k = i; k < n - 1 - i; k++) {
                console.log(k+''+j);
        }
```

```
        for(var l = j; l > i; l--) {
                console.log(k+''+l);
        }
        for(var x = k; x > i; x--) {
                console.log(x+''+l);
        }
    }
}

squarePrint(6,4);
```

Прочитайте Algo: - Печатать am * n матрицу в квадратной форме онлайн: https://riptutorial.com/ru/algorithm/topic/9968/algo----печатать-am---n-матрицу-в-квадратной-форме

глава 5: Odd-Even Сортировать

Examples

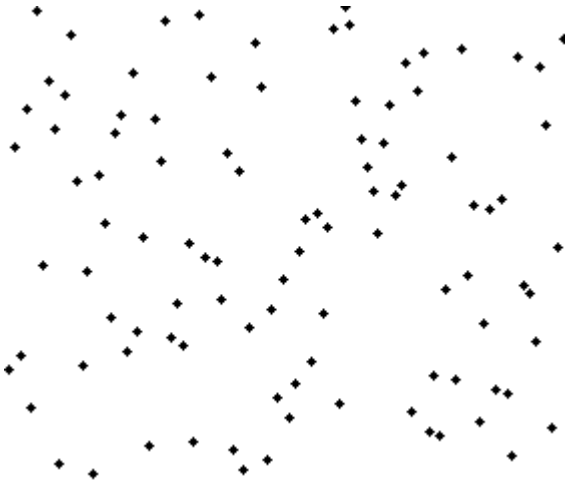
Нечетная-четная сортировка

Сорт Odd-Even Sort или Brick - это простой алгоритм сортировки, который разработан для использования на параллельных процессорах с локальной связью. Он работает, сравнивая все нечетные / четные индексы пары смежных элементов в списке и, если пара находится в неправильном порядке, элементы переключаются. Следующий шаг повторяет это для четных / нечетных индексированных пар. Затем он чередуется между нечетными / четными и четными / нечетными шагами, пока список не будет отсортирован.

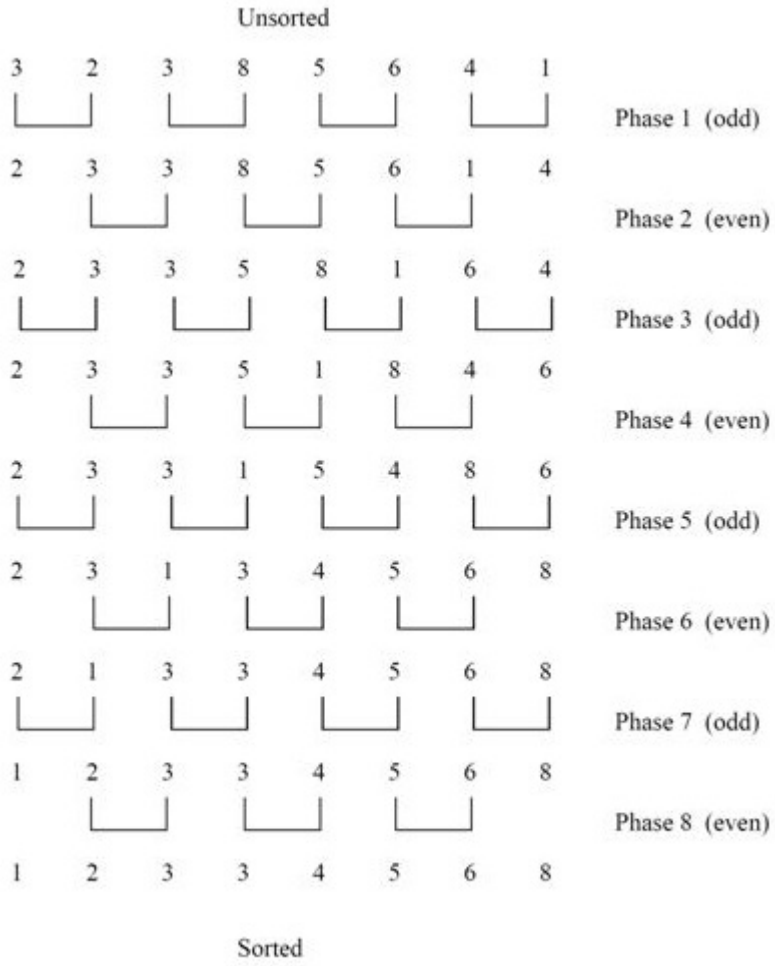
Псевдокод для Odd-Even Сортировать:

```
if n>2 then
    1. apply odd-even merge(n/2) recursively to the even subsequence a0, a2, ..., an-2 and to
the odd subsequence a1, a3, , ..., an-1
    2. comparison [i : i+1] for all i element {1, 3, 5, 7, ..., n-3}
else
    comparison [0 : 1]
```

В Википедии есть лучшая иллюстрация типа Odd-Even:



Пример Odd-Even Сортировать:



Реализация:

Я использовал язык C # для реализации Odd-Even Sort Algorithm.

```
public class OddEvenSort
{
    private static void SortOddEven(int[] input, int n)
    {
        var sort = false;

        while (!sort)
        {
            sort = true;
            for (var i = 1; i < n - 1; i += 2)
            {
                if (input[i] <= input[i + 1]) continue;
                var temp = input[i];
                input[i] = input[i + 1];
                input[i + 1] = temp;
                sort = false;
            }
            for (var i = 0; i < n - 1; i += 2)
            {
                if (input[i] <= input[i + 1]) continue;
                var temp = input[i];
                input[i] = input[i + 1];
                input[i + 1] = temp;
```

```
                sort = false;
            }
        }
    }

    public static int[] Main(int[] input)
    {
        SortOddEven(input, input.Length);
        return input;
    }
}
```

Вспомогательное пространство: `O(n)`
Сложность времени: `O(n)`

Прочитайте Odd-Even Сортировать онлайн: https://riptutorial.com/ru/algorithm/topic/7386/odd-even-сортировать

глава 6: Quicksort

замечания

Иногда Quicksort также известен как сортировка разделов-Exchange.
Вспомогательное пространство: `O(n)`
Сложность времени: худший `O(n²)` , лучший `O(nlogn)`

Examples

Основы Quicksort

Quicksort - это алгоритм сортировки, который выбирает элемент («стержень») и переупорядочивает массив, образующий два раздела, так что перед ним появляются все элементы, меньшие, чем точка поворота, и все последующие элементы. Затем алгоритм применяется рекурсивно к разделам, пока список не будет отсортирован.

1. Механизм схемы разделения Ломуто:

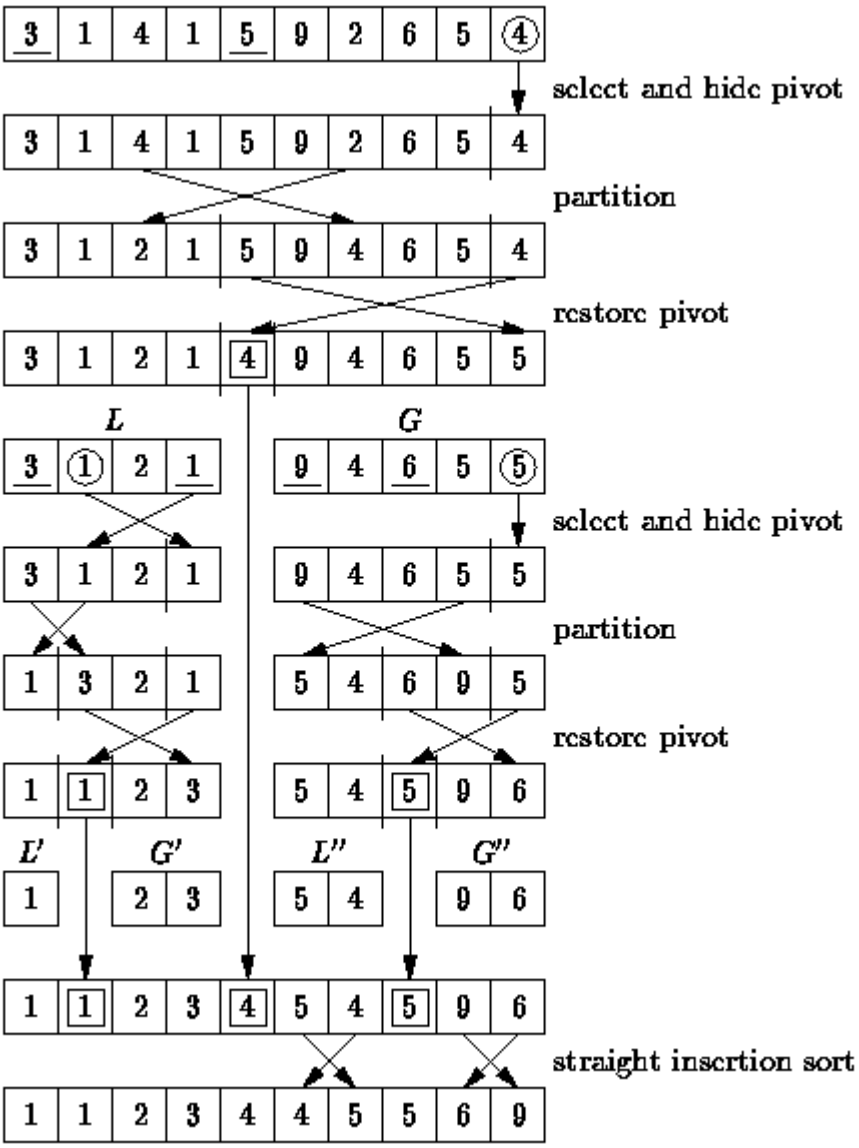
Эта схема выбирает стержень, который обычно является последним элементом в массиве. Алгоритм поддерживает индекс, чтобы поместить ось вращения в переменную i, и каждый раз, когда он находит элемент, который меньше или равен оси вращения, этот индекс увеличивается и этот элемент будет помещен перед точкой поворота.

```
partition(A, low, high) is
pivot := A[high]
i := low
for j := low to high - 1 do
    if A[j] ≤ pivot then
        swap A[i] with A[j]
        i := i + 1
swap A[i] with A[high]
return i
```

Механизм быстрой сортировки:

```
quicksort(A, low, high) is
if low < high then
    p := partition(A, low, high)
    quicksort(A, low, p - 1)
    quicksort(A, p + 1, high)
```

Пример быстрого сортировки:



2. Схема разбиения Хоара:

Он использует два индекса, которые начинаются на концах массива, которые разбиваются на разделы, а затем перемещаются друг к другу, пока они не обнаружат инверсию: пара элементов, одна большая или равная, чем точка опоры, одна меньшая или равная, которые ошибаются порядка друг относительно друга. Перевернутые элементы затем меняются местами. Когда индексы встречаются, алгоритм останавливается и возвращает окончательный индекс. Схема Хора более эффективна, чем схема разделения Ломуто, поскольку она в среднем делает в три раза меньше свопов и создает эффективные разделы, даже если все значения равны.

```
quicksort(A, lo, hi) is
if lo < hi then
    p := partition(A, lo, hi)
    quicksort(A, lo, p)
    quicksort(A, p + 1, hi)
```

Раздел:

```
partition(A, lo, hi) is
pivot := A[lo]
i := lo - 1
j := hi + 1
loop forever
    do:
        i := i + 1
    while A[i] < pivot do

    do:
        j := j - 1
    while A[j] > pivot do

    if i >= j then
        return j

    swap A[i] with A[j]
```

Реализация C

```
public class QuickSort
{
    private static int Partition(int[] input, int low, int high)
    {
        var pivot = input[high];
        var i = low - 1;
        for (var j = low; j <= high - 1; j++)
        {
            if (input[j] <= pivot)
            {
                i++;
                var temp = input[i];
                input[i] = input[j];
                input[j] = temp;
            }
        }
        var tmp = input[i + 1];
        input[i + 1] = input[high];
        input[high] = tmp;
        return (i + 1);
    }

    private static void SortQuick(int[] input, int low, int high)
    {
        while (true)
        {
            if (low < high)
            {
                var pi = Partition(input, low, high);
                SortQuick(input, low, pi - 1);
                low = pi + 1;
                continue;
            }
            break;
        }
    }

    public static int[] Main(int[] input)
    {
```

```
        SortQuick(input, 0, input.Length - 1);
        return input;
    }
}
```

Реализация Haskell

```
quickSort :: Ord a => [a] -> [a]
quickSort [] = []
quickSort (x:xs) = quickSort [ y | y <- xs, y <= x ]
                   ++ [x]
                   ++ quickSort [ z | z <- xs, z > x ]
```

Реализация jomoto для раздела Lomuto

```
public class Solution {

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    int[] ar = new int[n];
    for(int i=0; i<n; i++)
      ar[i] = sc.nextInt();
     quickSort(ar, 0, ar.length-1);
}

public static void quickSort(int[] ar, int low, int high)
 {
    if(low<high)
    {
        int p = partition(ar, low, high);
        quickSort(ar, 0 , p-1);
        quickSort(ar, p+1, high);
    }
 }
public static int partition(int[] ar, int l, int r)
 {
    int pivot = ar[r];
    int i =l;
    for(int j=l; j<r; j++)
     {
        if(ar[j] <= pivot)
         {
            int t = ar[j];
            ar[j] = ar[i];
            ar[i] = t;
            i++;
        }
    }
    int t = ar[i];
    ar[i] = ar[r];
    ar[r] = t;

    return i;
}
```

Быстрое сортирование в Python

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) / 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

print quicksort([3,6,8,10,1,2,1])
```

Печать "[1, 1, 2, 3, 6, 8, 10]"

Прочитайте Quicksort онлайн: https://riptutorial.com/ru/algorithm/topic/7232/quicksort

глава 7: Radix Sort

Examples

Основная информация о Radix

Radix Sort - это алгоритм, основанный на сравнении нижних границ. Это не сравнительный целочисленный алгоритм сортировки, который сортирует данные с помощью целых ключей, группируя ключи по отдельным цифрам, которые имеют значительную позицию и значение. Radix sort - это линейный алгоритм сортировки по времени, который сортирует по времени O (n + k), когда элементы находятся в диапазоне от 1 до k. Идея Radix Sort состоит в том, чтобы выполнять сортировку по цифрам, начиная с наименьшей значащей цифры и заканчивая самой значительной цифрой. Сортировка Radix использует сортировку подсчета в качестве подпрограммы для сортировки. Сорт Radix - это обобщение сортировки ковша.

Псевдокод для Bucket Сортировка:

1. Создайте массив из элементов [0..n-1].
2. Call Bucket Сортировать по меньшей мере до самой значимой цифры каждого элемента в качестве ключа.
3. Верните отсортированный массив.

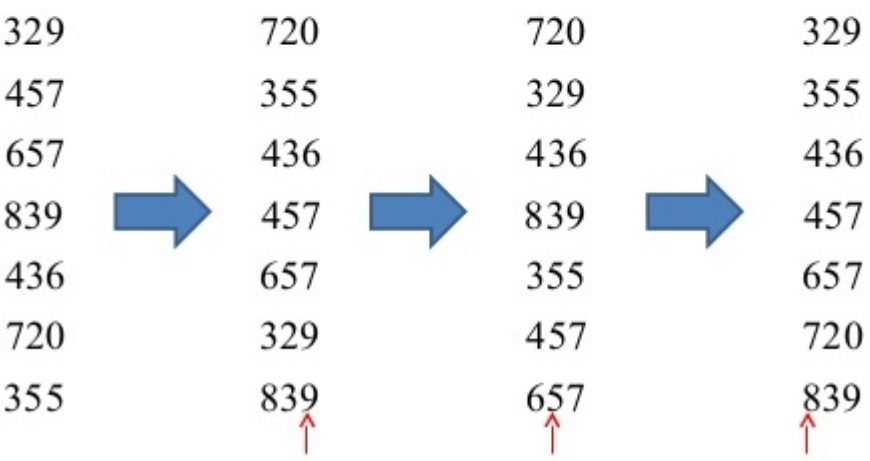
Пример сортировки Radix:

In input array A, each element is a number of d digit.

Radix - Sort(A, d)

for $i \leftarrow 1$ to d

do "use a stable sort to sort array A on digit i;



Вспомогательный объем: `O(n)`
Сложность времени: `O(n)`

Прочитайте Radix Sort онлайн: https://riptutorial.com/ru/algorithm/topic/7314/radix-sort

глава 8: Алгоритм Беллмана-Форда

замечания

Учитывая ориентированный граф `G` , мы часто хотим найти кратчайшее расстояние от данного узла `A` до остальной части узлов на графике. Алгоритм **Дейкстры** является самым известным алгоритмом для нахождения кратчайшего пути, однако он работает только в том случае, если веса ребер данного графа неотрицательны. **Однако Bellman-Ford** стремится найти самый короткий путь от данного узла (если он существует), даже если некоторые из весов являются отрицательными. Заметим, что кратчайшее расстояние может не существовать, если на графике присутствует отрицательный цикл (в этом случае мы можем обойти цикл, приводящий к бесконечно малым общим расстоянием). **Bellman-Ford** дополнительно позволяет определить наличие такого цикла.

Общая сложность алгоритма `O(V*E)` , где V - число вершин и `E` число ребер

Examples

Алгоритм наименьшего пути одиночного источника (учитывая, что на графике есть отрицательный цикл)

Прежде чем читать этот пример, требуется краткое представление о рекреационной релаксации. Вы можете узнать это здесь

Алгоритм Беллмана-Форда вычисляет кратчайшие пути из одной вершины источника ко всем остальным вершинам взвешенного орграфа. Несмотря на то, что он медленнее Алгоритма Дейкстры , он работает в случаях, когда вес края отрицательный, и он также находит отрицательный весовой цикл на графике. Проблема с алгоритмом Дейкстры заключается в том, что если есть отрицательный цикл, вы продолжаете проходить цикл снова и снова и продолжаете уменьшать расстояние между двумя вершинами.

Идея этого алгоритма состоит в том, чтобы пройти через все ребра этого графа один за другим в некотором случайном порядке. Это может быть любой случайный порядок. Но вы должны убедиться, что если **uv** (где **u** и **v** - две вершины в графе) является одним из ваших ордеров, тогда должно быть ребро от **u** до **v** . Обычно это берется непосредственно из порядка введенного ввода. Опять же, любой случайный порядок будет работать.

Выбрав порядок, мы *расслабим* ребра по формуле релаксации. Для данного ребра **uv,** идущего от **u** до **v,** релаксационная формула:

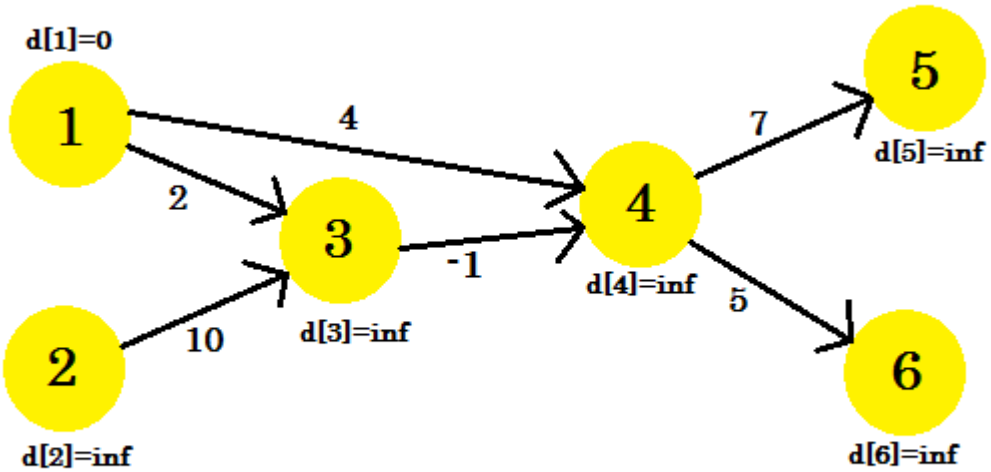
```
if distance[u] + cost[u][v] < d[v]
    d[v] = d[u] + cost[u][v]
```

То есть, если расстояние от **источника** до любой вершины **u** + вес **ребра uv** меньше расстояния от **источника** к другой вершине **v** , мы обновляем расстояние от **источника** до **v** . Нам нужно максимально *релаксировать* ребра **(V-1),** где **V** - количество ребер в графе. Почему **(V-1)** вы спрашиваете? Мы объясним это в другом примере. Также мы будем отслеживать родительскую вершину любой вершины, то есть когда мы расслабляем ребро, мы будем устанавливать:

```
parent[v] = u
```

Это означает, что мы нашли еще один более короткий путь для достижения **v** через **u** . Нам понадобится это позже, чтобы напечатать кратчайший путь от **источника** до назначенной вершины.

Давайте посмотрим на пример. У нас есть график:



Мы выбрали **1** в качестве **исходной** вершины. Мы хотим узнать кратчайший путь от **источника** ко всем другим вершинам.

Сначала **d [1] = 0,** потому что это источник. И отдых - *бесконечность* , потому что мы еще не знаем их расстояния.

Мы разделим ребра в этой последовательности:

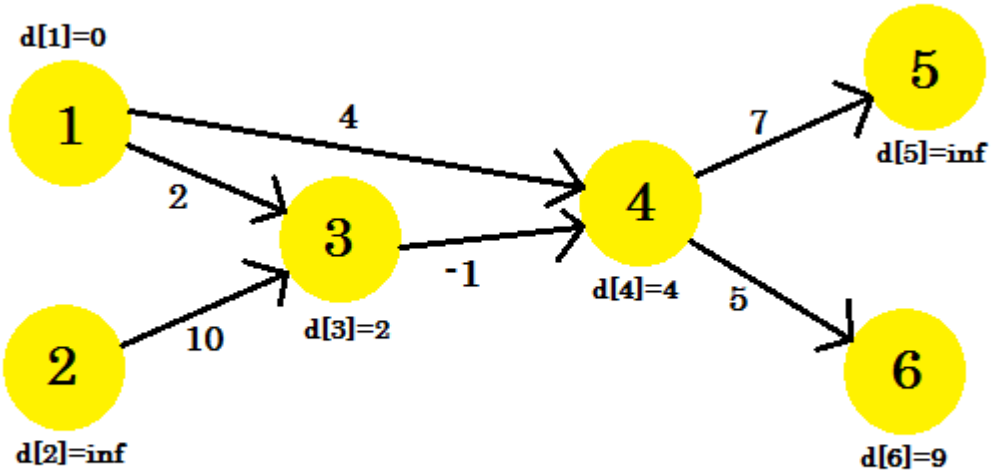
```
+--------+--------+--------+--------+--------+--------+--------+
| Serial |    1   |    2   |    3   |    4   |    5   |    6   |
+--------+--------+--------+--------+--------+--------+--------+
|  Edge  |  4->5  |  3->4  |  1->3  |  1->4  |  4->6  |  2->3  |
+--------+--------+--------+--------+--------+--------+--------+
```

Вы можете выполнить любую последовательность, которую вы хотите. Если *разделить* края один раз, что мы получим? Мы получаем расстояние от **источника** ко всем другим

вершинам пути, использующему не более 1 ребра. Теперь давайте расслабляем края и обновляем значения **d []** . Мы получаем:

1. **d [4]** + **стоимость [4] [5]** = **бесконечность** + **7** = **бесконечность** . Мы не можем обновить это.
2. **d [2]** + **стоимость [3] [4]** = **бесконечность** . Мы не можем обновить это.
3. **d [1]** + **стоимость [1] [2]** = **0** + **2** = **2** < **d [2]** . Итак, **d [2]** = **2** . Также **parent [2] = 1** .
4. **d [1]** + **стоимость [1] [4]** = **4** . Итак, **d [4] = 4** < **d [4]** . **parent [4]** = **1** .
5. **d [4]** + **стоимость [4] [6]** = **9** . **d [6]** = **9** < **d [6]** . **parent [6]** = **4** .
6. **d [2]** + **стоимость [2] [2]** = **бесконечность** . Мы не можем обновить это.

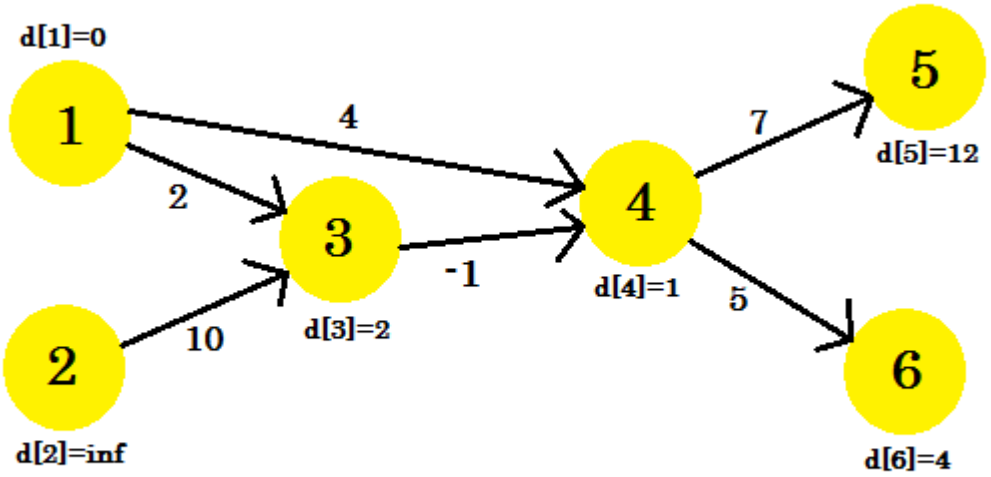
Мы не смогли обновить некоторые вершины, потому что условие `d[u] + cost[u][v] < d[v]` не совпало. Как мы уже говорили, мы нашли пути от **источника** к другим узлам с использованием максимального 1 ребра.



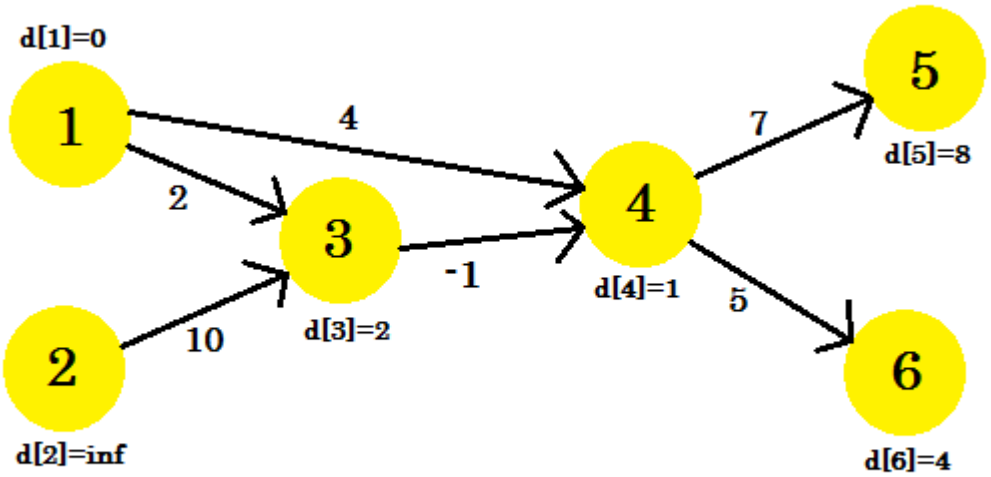
Наша вторая итерация предоставит нам путь с использованием двух узлов. Мы получаем:

1. **d [4]** + **стоимость [4] [5]** = **12** < **d [5]** . **d [5]** = **12** . **parent [5]** = **4** .
2. **d [3]** + **стоимость [3] [4]** = **1** < **d [4]** . **d [4]** = **1** . **parent [4]** = **3** .
3. **d [3]** остается неизменным.
4. **d [4]** остается неизменным.
5. **d [4]** + **стоимость [4] [6]** = **6** < **d [6]** . **d [6]** = **6** . **parent [6]** = **4** .
6. **d [3]** остается неизменным.

Наш график будет выглядеть так:



Наша третья итерация будет обновлять только **вершину 5** , где **d [5]** будет **8** . Наш график будет выглядеть так:



После этого, независимо от того, сколько итераций мы делаем, мы будем иметь одинаковые расстояния. Поэтому мы будем держать флаг, который проверяет, имеет место какое-либо обновление или нет. Если это не так, мы просто сломаем цикл. Наш псевдокод будет:

```
Procedure Bellman-Ford(Graph, source):
n := number of vertices in Graph
for i from 1 to n
    d[i] := infinity
    parent[i] := NULL
end for
d[source] := 0
for i from 1 to n-1
```

```
    flag := false
    for all edges from (u,v) in Graph
        if d[u] + cost[u][v] < d[v]
            d[v] := d[u] + cost[u][v]
            parent[v] := u
            flag := true
        end if
    end for
    if flag == false
        break
end for
Return d
```

Чтобы отслеживать отрицательный цикл, мы можем изменить наш код, используя описанную здесь процедуру. Наш законченный псевдокод будет:

```
Procedure Bellman-Ford-With-Negative-Cycle-Detection(Graph, source):
n := number of vertices in Graph
for i from 1 to n
    d[i] := infinity
    parent[i] := NULL
end for
d[source] := 0
for i from 1 to n-1
    flag := false
    for all edges from (u,v) in Graph
        if d[u] + cost[u][v] < d[v]
            d[v] := d[u] + cost[u][v]
            parent[v] := u
            flag := true
        end if
    end for
    if flag == false
        break
end for
for all edges from (u,v) in Graph
    if d[u] + cost[u][v] < d[v]
        Return "Negative Cycle Detected"
    end if
end for
Return d
```

Путь печати:

Чтобы напечатать кратчайший путь к вершине, мы перейдем к его родительскому элементу, пока не найдем **NULL,** а затем напечатаем вершины. Псевдокод будет:

```
Procedure PathPrinting(u)
v := parent[u]
if v == NULL
    return
PathPrinting(v)
print -> u
```

Сложность:

Так как нам нужно уменьшить максимум краев **(V-1)** , временная сложность этого алгоритма будет равна **O (V * E),** где **E** обозначает количество ребер, если мы используем `adjacency list` для представления графика. Однако, если `adjacency matrix` используется для представления графика, временной сложностью будет **O (V ^ 3)** . Причина состоит в том, что мы можем перебирать все ребра в **O (E),** когда используется `adjacency list` , но требуется время **O (V ^ 2),** когда используется `adjacency matrix` .

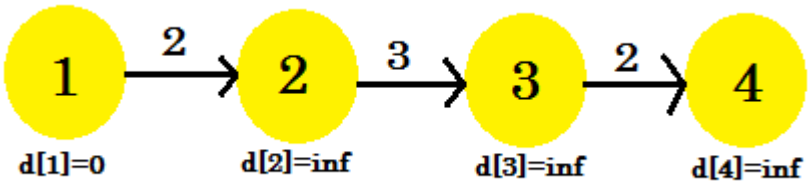
Почему мы должны максимально расслаблять все ребра (V-1) раз

Чтобы понять этот пример, рекомендуется дать краткое представление об алгоритме кратчайшего пути одиночного источника Bellman-Ford, который можно найти здесь

В алгоритме Беллмана-Форда, чтобы узнать кратчайший путь, нам нужно *ослабить* все ребра графа. Этот процесс повторяется максимум **(V-1)** раз, где **V** - количество вершин в графе.

Количество итераций, необходимых для выяснения кратчайшего пути от **источника** ко всем другим вершинам, зависит от порядка, который мы выбираем для *релаксации* ребер.

Давайте рассмотрим пример:



Здесь **исходная** вершина равна 1. Мы найдем кратчайшее расстояние между **источником** и всеми другими вершинами. Мы можем ясно видеть, что для достижения **вершины 4** , в худшем случае, это займет **(V-1)** ребра. Теперь, в зависимости от порядка, в котором обнаружены края, может потребоваться **(V-1)** раз, чтобы обнаружить **вершину 4** . Не понял? Давайте используем алгоритм Беллмана-Форда, чтобы узнать кратчайший путь здесь:

Мы будем использовать эту последовательность:

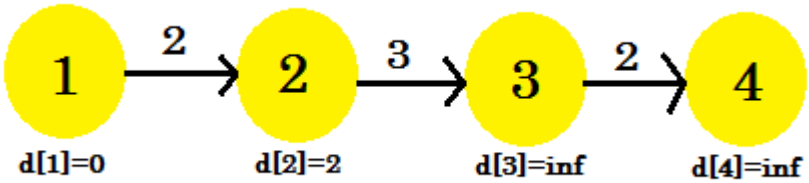
```
+--------+--------+--------+--------+
| Serial |    1   |    2   |    3   |
+--------+--------+--------+--------+
|  Edge  |  3->4  |  2->3  |  1->2  |
+--------+--------+--------+--------+
```

Для нашей первой итерации:

1. **d [3]** + **стоимость [3] [4]** = **бесконечность** . Это ничего не изменит.
2. **d [2]** + **стоимость [2] [3]** = **бесконечность** . Это ничего не изменит.

3. **d [1]** + **стоимость [1] [2]** = **2** < **d [2]** . **d [2]** = **2** . **parent [2]** = **1** .

Мы видим, что наш *релаксационный* процесс только изменился **d [2]** . Наш график будет

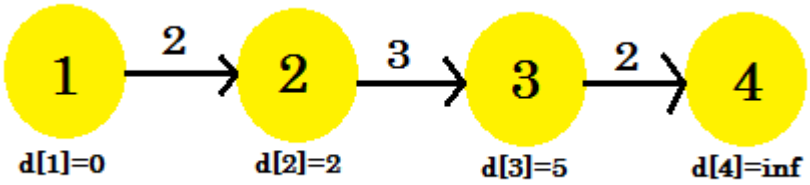


выглядеть так:

Вторая итерация:

1. **d [3]** + **стоимость [3] [4]** = **бесконечность** . Это ничего не изменит.
2. **d [2]** + **стоимость [2] [3]** = **5** < **d [3]** . **d [3]** = **5** . **parent [3]** = **2.**
3. Он не будет изменен.

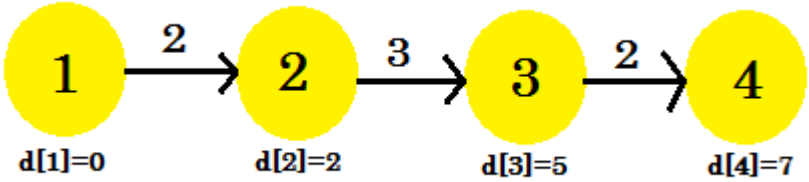
На этот раз процесс *релаксации* изменился **d [3]** . Наш график будет выглядеть так:



Третья итерация:

1. **d [3]** + **стоимость [3] [4]** = **7** < **d [4]** . **d [4]** = **7** . **parent [4]** = **3** .
2. Он не будет изменен.
3. Он не будет изменен.

Наша третья итерация наконец обнаружила кратчайший путь к **4** из **1** . Наш график будет



выглядеть так:

Итак, для выяснения кратчайшего пути потребовалось **3** итерации. После этого, независимо от того, сколько раз мы *расслабляем* ребра, значения в **d []** останутся неизменными. Теперь, если мы рассмотрим другую последовательность:

```
+--------+--------+--------+--------+
| Serial |    1   |    2   |    3   |
+--------+--------+--------+--------+
|  Edge  |  1->2  |  2->3  |  3->4  |
+--------+--------+--------+--------+
```

Мы получим:

1. **d [1]** + **стоимость [1] [2]** = **2** < **d [2]** . **d [2]** = **2** .
2. **d [2]** + **стоимость [2] [3]** = **5** < **d [3]** . **d [3]** = **5** .
3. **d [3]** + **стоимость [3] [4]** = **7** < **d [4]** . **d [4]** = **5** .

Наша самая первая итерация нашла кратчайший путь от **источника** ко всем остальным узлам. Возможна другая последовательность **1-> 2** , **3-> 4** , **2-> 3** , которая даст нам кратчайший путь после **2** итераций. Мы можем прийти к решению о том, что независимо от того, как мы упорядочим последовательность, в этом примере не будет более **трех** итераций, чтобы узнать кратчайший путь от **источника** .

Мы можем заключить, что для наилучшего случая потребуется **1** итерация, чтобы узнать кратчайший путь из **источника** . В худшем случае это потребует **(V-1)** итераций, поэтому мы повторяем процесс *релаксации* **(V-1)** раз.

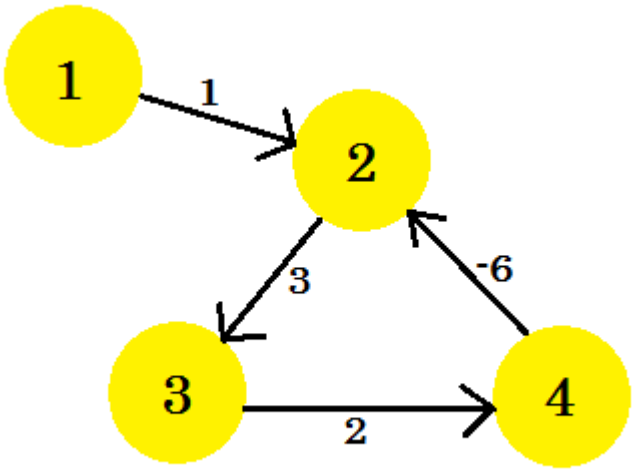
Обнаружение отрицательного цикла на графике

Чтобы понять этот пример, рекомендуется иметь краткое представление о алгоритме Беллмана-Форда, который можно найти здесь

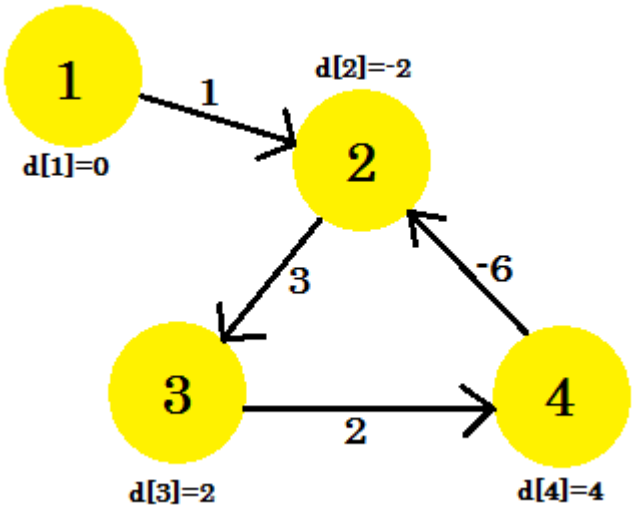
Используя алгоритм Беллмана-Форда, мы можем определить, есть ли отрицательный цикл на нашем графике. Мы знаем, что для выяснения кратчайшего пути нам необходимо *ослабить* все ребра графа **(V-1)** раз, где **V** - число вершин в графе. Мы уже видели, что в этом примере после **(V-1)** итераций мы не можем обновить **d []** , независимо от того, сколько итераций мы делаем. Или мы можем?

Если на графике есть отрицательный цикл, даже после **(V-1)** итераций мы можем обновить **d []** . Это происходит потому, что для каждой итерации, проходя через отрицательный цикл, всегда уменьшается стоимость кратчайшего пути. Вот почему алгоритм Беллмана-Форда ограничивает количество итераций **(V-1)** . Если бы мы использовали Алгоритм Дейкстры , мы бы застряли в бесконечном цикле. Однако давайте сосредоточимся на поиске отрицательного цикла.

Предположим, у нас есть график:



Выберем **вершину 1** как **источник** . После применения алгоритма алгоритма одиночного источника Беллмана-Форда к графику мы выясним расстояния от **источника** до всех остальных вершин.



Вот как выглядит график после **(V-1)** = **3** итераций. Это должно быть результатом, так как существует **4** ребра, нам нужно не более **трех** итераций, чтобы узнать кратчайший путь. Таким образом, либо это ответ, либо график отрицательного веса на графике. Чтобы найти, что после **(V-1)** итераций мы делаем еще одну заключительную итерацию, и если расстояние продолжает уменьшаться, это означает, что на графике определенно отрицательный весовой цикл.

В этом примере: если мы проверим **2-3** , **d [2]** + **стоимость [2] [3]** даст нам **1,** что меньше, чем **d [3]** . Поэтому мы можем заключить, что на нашем графике есть отрицательный цикл.

Итак, как мы узнаем отрицательный цикл? Мы немного модифицируем процедуру Bellman-Ford:

```
Procedure NegativeCycleDetector(Graph, source):
```

```
n := number of vertices in Graph
for i from 1 to n
    d[i] := infinity
end for
d[source] := 0
for i from 1 to n-1
    flag := false
    for all edges from (u,v) in Graph
        if d[u] + cost[u][v] < d[v]
            d[v] := d[u] + cost[u][v]
            flag := true
        end if
    end for
    if flag == false
        break
end for
for all edges from (u,v) in Graph
    if d[u] + cost[u][v] < d[v]
        Return "Negative Cycle Detected"
    end if
end for
Return "No Negative Cycle"
```

Вот как мы узнаем, есть ли отрицательный цикл на графике. Мы также можем модифицировать алгоритм Bellman-Ford для отслеживания отрицательных циклов.

Прочитайте Алгоритм Беллмана-Форда онлайн: https://riptutorial.com/ru/algorithm/topic/4791/алгоритм-беллмана-форда

глава 9: Алгоритм Дейкстры

Examples

Алгоритм кратчайшего пути Дейкстры

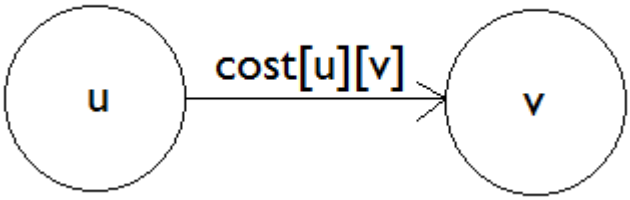
Прежде чем продолжить, рекомендуется иметь краткое представление о матрице смежности и BFS

Алгоритм Дейкстры известен как алгоритм кратчайшего пути с одним источником. Он используется для поиска кратчайших путей между узлами в графе, которые могут представлять, например, дорожные сети. Он был задуман Эдсгером В. Дейкстре в 1956 году и опубликован три года спустя.

Мы можем найти кратчайший путь, используя алгоритм поиска Breadth First Search (BFS). Этот алгоритм работает отлично, но проблема в том, что он предполагает, что стоимость прохождения каждого пути одинакова, что означает, что стоимость каждого ребра одинакова. Алгоритм Дейкстры помогает найти кратчайший путь, где стоимость каждого пути не совпадает.

Сначала мы увидим, как изменить BFS для написания алгоритма Дейкстры, тогда мы добавим очередь приоритетов, чтобы сделать ее полным алгоритмом Дейкстры.

Скажем, расстояние каждого узла от источника сохраняется в массиве **d []** . Как и в, **d [3]** означает, что **d [3]** время для достижения **узла 3** из **источника** . Если мы не знаем расстояния, мы сохраним *бесконечность* в **d [3]** . Кроме того, пусть **стоимость [u] [v]** отражает стоимость **uv** . Это означает, что для перехода от узла **u** к **v** требуется **стоимость [u] [v]** .



Нам нужно понять релаксацию края. Скажем, из вашего дома, **источника** , требуется *10* минут, чтобы пойти на место **A.** И это займет *25* минут, чтобы пойти на место **B.** У нас есть,

```
d[A] = 10
d[B] = 25
```

Теперь предположим, что от места **A** до места **B** требуется *7* минут, это означает:

```
cost[A][B] = 7
```

Затем мы можем поместить **B** из **источника** , перейдя на место **A** из **источника,** а затем из места **A** , идя на место **B** , которое займет *10 + 7 = 17* минут, вместо *25* минут. Так,

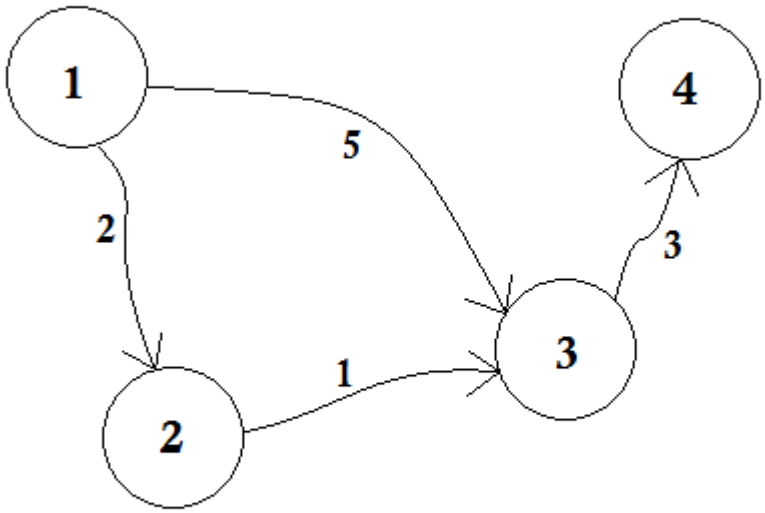
```
d[A] + cost[A][B] < d[B]
```

Затем мы обновляем,

```
d[B] = d[A] + cost[A][B]
```

Это называется расслаблением. Мы перейдем от узла **u** к узлу **v,** а если **d [u] + cost [u] [v] <d [v],** то мы обновим **d [v] = d [u] + cost [u] [v]** .

В BFS нам не нужно было посещать какой-либо узел дважды. Мы только проверяли, посетил ли узел или нет. Если он не был посещен, мы переместили узел в очередь, пометили его как посетили и увеличили расстояние на 1. В Dijkstra мы можем нажать узел в очереди и вместо обновления его посещением, мы *расслабляем* или обновляем новый край. Давайте рассмотрим один пример:



Предположим, что **Node 1** является **источником** . Затем,

```
d[1] = 0
d[2] = d[3] = d[4] = infinity (or a large value)
```

Мы установили **d [2], d [3]** и **d [4]** на *бесконечность,* потому что мы еще не знаем расстояния. И расстояние от **источника,** конечно, *0* . Теперь мы переходим к другим узлам

из **исходного кода,** и если мы можем их обновить, мы будем выталкивать их в очередь. Скажем, например, мы пройдем к **краю 1-2** . Как **d [1] + 2 <d [2],** что сделает **d [2] = 2** . Точно так же мы будем пересекать **край 1-3,** что делает **d [3] = 5** .

Мы можем ясно видеть, что *5* - это не кратчайшее расстояние, которое мы можем пересечь, чтобы перейти к **узлу 3** . Поэтому перемещение узла только один раз, как BFS, здесь не работает. Если перейти от **узла 2** к **узлу 3** с использованием **ребра 2-3** , мы можем обновить **d [3] = d [2] + 1 = 3** . Таким образом, мы видим, что один узел может обновляться много раз. Сколько раз вы спрашиваете? Максимальное количество раз, когда узел может быть обновлен, - это количество встроенных узлов.

Давайте посмотрим на псевдокод для посещения любого узла несколько раз. Мы просто изменим BFS:

```
procedure BFSmodified(G, source):
Q = queue()
distance[] = infinity
Q.enqueue(source)
distance[source]=0
while Q is not empty
    u <- Q.pop()
    for all edges from u to v in G.adjacentEdges(v) do
        if distance[u] + cost[u][v] < distance[v]
            distance[v] = distance[u] + cost[u][v]
        end if
    end for
end while
Return distance
```

Это можно использовать для поиска кратчайшего пути всего узла из источника. Сложность этого кода не так хороша. Вот почему,

В BFS, когда мы переходим от **узла 1** ко всем другим узлам, мы следуем *первому* методу *first serve* . Например, мы перешли к **узлу 3** из **источника** перед обработкой **узла 2** . Если мы перейдем к **узлу 3** из **источника** , мы обновим **узел 4** как **5 + 3 = 8** . Когда мы снова обновляем **узел 3** из **узла 2** , нам нужно снова обновить **узел 4** как **3 + 3 = 6** ! Поэтому **узел 4** обновляется дважды.

Дейкстра предложил вместо того, чтобы идти *первым, первым подавать* метод, если сначала обновить ближайшие узлы, тогда это займет меньше обновлений. Если бы мы обработали **узел 2** раньше, то **узел 3** был бы обновлен до этого, и после обновления **узла 4** соответственно мы легко получили бы кратчайшее расстояние! Идея состоит в том, чтобы выбрать из очереди, ближайший к **источнику** узел. Поэтому мы будем использовать *Priority Queue* здесь, чтобы, когда мы выходим в очередь, это принесет нам ближайший узел **u** из **источника** . Как он это сделает? Он проверит значение **d [u]** с ним.

Давайте посмотрим на псевдокод:

```
procedure dijkstra(G, source):
Q = priority_queue()
distance[] = infinity
Q.enqueue(source)
distance[source] = 0
while Q is not empty
    u <- nodes in Q with minimum distance[]
    remove u from the Q
    for all edges from u to v in G.adjacentEdges(v) do
        if distance[u] + cost[u][v] < distance[v]
            distance[v] = distance[u] + cost[u][v]
            Q.enqueue(v)
        end if
    end for
end while
Return distance
```

Псевдокод возвращает расстояние от всех остальных узлов от **источника** . Если мы хотим знать расстояние от одного узла **v** , мы можем просто вернуть значение, когда **v** выставляется из очереди.

Теперь, работает ли алгоритм Дейкстры, когда есть отрицательное ребро? Если есть отрицательный цикл, тогда будет цикл бесконечности, так как он будет продолжать уменьшать стоимость каждый раз. Даже если есть отрицательный фронт, Dijkstra не будет работать, если мы не вернемся сразу после того, как цель выскочит. Но тогда это не будет алгоритм Дейкстры. Нам понадобится алгоритм Bellman-Ford для обработки отрицательного края / цикла.

Сложность:

Сложность BFS - **O (log (V + E)),** где **V** - число узлов, а **E** - количество ребер. Для Dijkstra сложность аналогична, но сортировка *очереди приоритетов* принимает **O (logV)** . Таким образом, общая сложность: **O (Vlog (V) + E)**

Ниже приведен пример Java для решения алгоритма кратчайшего пути Дейкстры с использованием матрицы смежности

```
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{
    static final int V=9;
    int minDistance(int dist[], Boolean sptSet[])
    {
        int min = Integer.MAX_VALUE, min_index=-1;

        for (int v = 0; v < V; v++)
            if (sptSet[v] == false && dist[v] <= min)
            {
                min = dist[v];
                min_index = v;
            }
```

```
        return min_index;
    }

    void printSolution(int dist[], int n)
    {
        System.out.println("Vertex Distance from Source");
        for (int i = 0; i < V; i++)
            System.out.println(i+" \t\t "+dist[i]);
    }

    void dijkstra(int graph[][], int src)
    {

        Boolean sptSet[] = new Boolean[V];

        for (int i = 0; i < V; i++)
        {
            dist[i] = Integer.MAX_VALUE;
            sptSet[i] = false;
        }

        dist[src] = 0;

        for (int count = 0; count < V-1; count++)
        {
            int u = minDistance(dist, sptSet);

            sptSet[u] = true;

            for (int v = 0; v < V; v++)

                if (!sptSet[v] && graph[u][v]!=0 &&
                        dist[u] != Integer.MAX_VALUE &&
                        dist[u]+graph[u][v] < dist[v])
                    dist[v] = dist[u] + graph[u][v];
        }

        printSolution(dist, V);
    }

    public static void main (String[] args)
    {
    int graph[][] = new int[][]{{0, 4, 0, 0, 0, 0, 0, 8, 0},
                                {4, 0, 8, 0, 0, 0, 0, 11, 0},
                                {0, 8, 0, 7, 0, 4, 0, 0, 2},
                                {0, 0, 7, 0, 9, 14, 0, 0, 0},
                                {0, 0, 0, 9, 0, 10, 0, 0, 0},
                                {0, 0, 4, 14, 10, 0, 2, 0, 0},
                                {0, 0, 0, 0, 0, 2, 0, 1, 6},
                                {8, 11, 0, 0, 0, 0, 1, 0, 7},
                                {0, 0, 2, 0, 0, 0, 6, 7, 0}
                                };
        ShortestPath t = new ShortestPath();
        t.dijkstra(graph, 0);
    }
}
```

Ожидаемый результат программы

```
Vertex   Distance from Source
0                0
1                4
2                12
3                19
4                21
5                11
6                9
7                8
8                14
```

Прочитайте Алгоритм Дейкстры онлайн: https://riptutorial.com/ru/algorithm/topic/7151/алгоритм-дейкстры

глава 10: Алгоритм каталонского номера

Examples

Алгоритм каталонского номера Основная информация

Алгоритм каталитических чисел - алгоритм динамического программирования.

В комбинаторной математике каталонские числа образуют последовательность натуральных чисел, которые встречаются в различных задачах подсчета, часто включающие рекурсивно определенные объекты. Каталонские числа на неотрицательных целых числах представляют собой набор чисел, возникающих в задачах нумерации деревьев типа: «Сколько способов можно разделить обычный n-угольник на n-2 треугольники, если разные ориентации подсчитываются отдельно?»

Применение алгоритма каталонского номера:

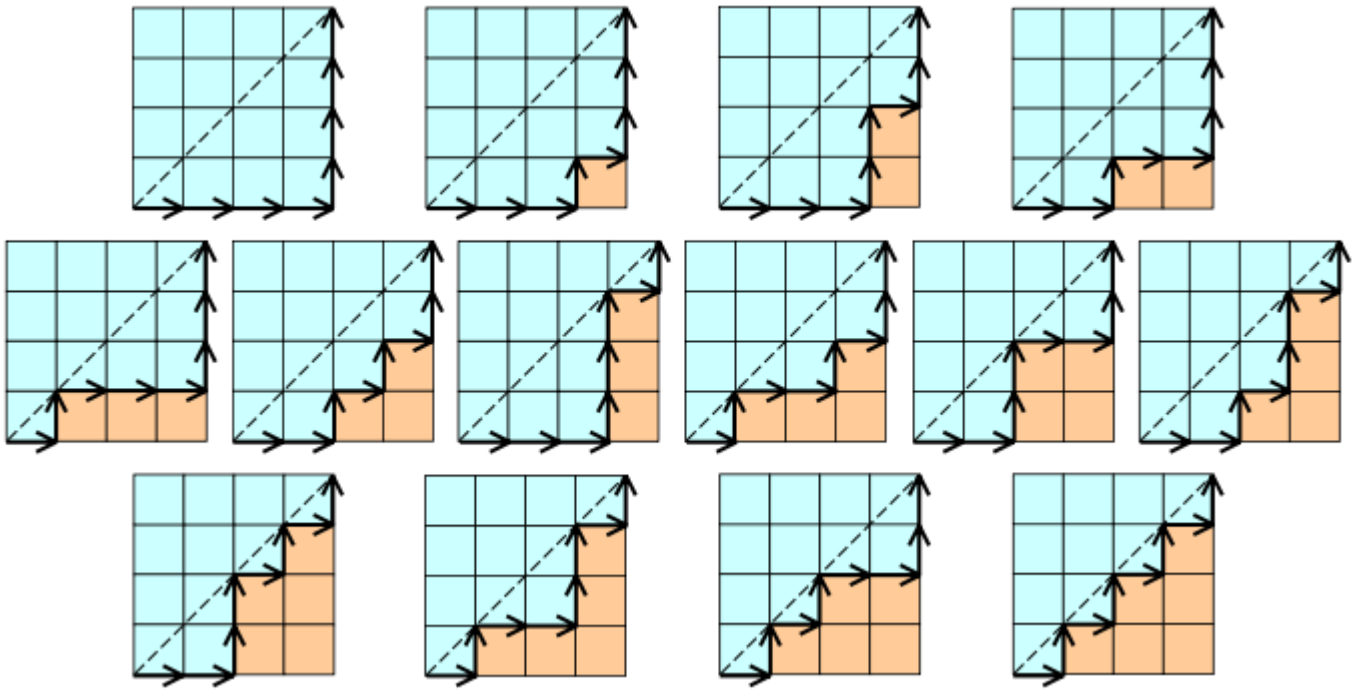
1. Количество способов укладки монет в нижнем ряду, состоящее из n последовательных монет в плоскости, так что никакие монеты не могут быть помещены на две стороны нижних монет, и каждая дополнительная монета должна быть выше двух других монет, n-е каталонское число.
2. Количество способов группировки строки из n пар круглых скобок, так что каждая открытая скобка имеет соответствующую закрытую скобку, является n-м каталонским числом.
3. Количество способов разрезания n + 2-стороннего выпуклого многоугольника в плоскости на треугольники путем соединения вершин с прямыми непересекающимися линиями является n-м каталонским числом. Это приложение, в котором интересовался Эйлер.

Используя нулевую нумерацию, *n*- е каталонское число задается непосредственно в терминах биномиальных коэффициентов следующим уравнением.

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n}\frac{n+k}{k} \qquad \text{for } n \geq 0.$$

Пример каталонского номера:

Здесь значение n = 4. (Лучший пример - из Википедии)



Вспомогательное пространство: `O(n)`
Сложность времени: `O(n^2)`

Реализация C

```
public class CatalanNumber
{
    public static int Main(int number)
    {
        int result = 0;
        if (number <= 1) return 1;
        for (int i = 0; i < number; i++)
        {
            result += Main(i)*Main(number - i - 1);
        }
        return result;
    }
}
```

Прочитайте Алгоритм каталонского номера онлайн:
https://riptutorial.com/ru/algorithm/topic/7406/алгоритм-каталонского-номера

глава 11: Алгоритм Кнут Моррис Пратт (KMP)

Вступление

KMP - это алгоритм сопоставления с образцом, который ищет появление «слова» **W** в основной «текстовой строке» **S** , используя наблюдение, что, когда происходит несоответствие, мы располагаем достаточной информацией, чтобы определить, где можно начать следующее совпадение. Мы используйте эту информацию, чтобы избежать совпадения символов, которые, как мы знаем, будут в любом случае соответствовать. Самая сложная сложность поиска шаблона сводится к **O (n)** .

Examples

KMP-пример

Алгоритм

Этот алгоритм является двухэтапным процессом. Сначала мы создаем вспомогательный массив lps [], а затем используем этот массив для поиска шаблона.

Предварительная обработка :

1. Мы предварительно обрабатываем шаблон и создаем вспомогательный массив lps [], который используется для пропуска символов при совпадении.
2. Здесь lps [] указывает на самый длинный правильный префикс, который также является суффиксом. Правильный префикс является префиксом, в котором целая строка не включена. Например, префиксы строки **ABC** - это **«»** , **«A»** , **«AB»** и **«ABC»** . Правильными префиксами являются **«»** , **«A»** и **«AB»** . Суффиксами строки являются **«»** , **«C»** , **«BC»** и **«ABC»** .

поиск

1. Мы сохраняем соответствующие символы **txt [i]** и **pat [j]** и продолжаем увеличивать i и j, пока **pat [j]** и **txt [i]** сохраняют соответствие.

2. Когда мы видим несоответствие, мы знаем, что символы **pat [0..j-1]** соответствуют **txt [i-j + 1 ... i-1].** Мы также знаем, что lps [j-1] - количество символов **pat [0 ... j-1],** которые являются как правильным префиксом, так и суффиксом. Из этого можно заключить, что нам не нужно сопоставлять эти **lps [j-1]** символы с **txt [ij ... i-1],** потому что мы знаем, что эти символы будет соответствовать в любом случае.

Реализация в Java

```
public class KMP {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        String str = "abcabdabc";
        String pattern = "abc";
        KMP obj = new KMP();
        System.out.println(obj.patternExistKMP(str.toCharArray(), pattern.toCharArray()));
    }

    public int[] computeLPS(char[] str){
        int lps[] = new int[str.length];

        lps[0] = 0;
        int j = 0;
        for(int i =1;i<str.length;i++){
            if(str[j] == str[i]){
                lps[i] = j+1;
                j++;
                i++;
            }else{
                if(j!=0){
                    j = lps[j-1];
                }else{
                    lps[i] = j+1;
                    i++;
                }
            }

        }

        return lps;
    }

    public boolean patternExistKMP(char[] text,char[] pat){
        int[] lps = computeLPS(pat);
        int i=0,j=0;
        while(i<text.length && j<pat.length){
            if(text[i] == pat[j]){
                i++;
                j++;
            }else{
                if(j!=0){
                    j = lps[j-1];
                }else{
                    i++;
                }
            }
        }

        if(j==pat.length)
            return true;
        return false;
    }

}
```

Прочитайте Алгоритм Кнут Моррис Пратт (KMP) онлайн:

https://riptutorial.com/ru/algorithm/topic/10811/алгоритм-кнут-моррис-пратт--kmp-

глава 12: Алгоритм Крускаля

замечания

Алгоритм Крускала - это **жадный** алгоритм, используемый для поиска **минимального связующего дерева (MST)** графа. Минимальное остовное дерево - это дерево, которое соединяет все вершины графа и имеет минимальный общий вес края.

Алгоритм Крускаля делает это путем многократного выделения краев с **минимальным весом** (которые еще не находятся в MST) и добавляет их к окончательному результату, если две вершины, связанные этим ребром, еще не соединены в MST, в противном случае он пропускает это ребро. Union. Поиск структуры данных может использоваться для проверки того, что две вершины уже подключены в MST или нет. Несколько свойств MST заключаются в следующем:

1. MST графа с `n` вершинами будет иметь ровно `n-1` ребер.
2. Существует единственный путь от каждой вершины до любой другой вершины.

Examples

Простая, более подробная реализация

Чтобы эффективно обрабатывать обнаружение циклов, мы рассматриваем каждый узел как часть дерева. При добавлении ребра мы проверяем, являются ли его два компонентных узла частью различных деревьев. Первоначально каждый узел составляет одноузловое дерево.

```
algorithm kruskalMST'(G: a graph)
    sort G's edges by their value
    MST = a forest of trees, initially each tree is a node in the graph
    for each edge e in G:
        if the root of the tree that e.first belongs to is not the same
        as the root of the tree that e.second belongs to:
            connect one of the roots to the other, thus merging two trees

    return MST, which now a single-tree forest
```

Простая реализация на основе несвязанного набора

Вышеупомянутая методология леса фактически представляет собой структуру данных, не связанных между собой, которая включает в себя три основные операции:

```
subalgo makeSet(v: a node):
    v.parent = v    <- make a new tree rooted at v
```

```
subalgo findSet(v: a node):
    if v.parent == v:
        return v
    return findSet(v.parent)

subalgo unionSet(v, u: nodes):
    vRoot = findSet(v)
    uRoot = findSet(u)

    uRoot.parent = vRoot

algorithm kruskalMST''(G: a graph):
    sort G's edges by their value
    for each node n in G:
        makeSet(n)
    for each edge e in G:
        if findSet(e.first) != findSet(e.second):
            unionSet(e.first, e.second)
```

Эта наивная реализация приводит к времени `O(n log n)` для управления структурой данных с несвязанными наборами, приводящей к `O(m*n log n)` времени для всего алгоритма Крускала.

Оптимальная реализация на основе непересекающихся множеств

Мы можем сделать две вещи, чтобы улучшить простые и субоптимальные неагрессивные суб-алгоритмы:

1. **Эвристика сжатия** `findSet` : `findSet` не требует когда-либо обрабатывать дерево с высотой больше `2` . Если он завершает итерацию такого дерева, он может связать нижние узлы напрямую с корнем, оптимизируя будущие обходы;

   ```
   subalgo findSet(v: a node):
       if v.parent != v
           v.parent = findSet(v.parent)
       return v.parent
   ```

2. **Эвристическая слияния на основе высоты** : для каждого узла сохраняйте высоту своего поддерева. При слиянии сделайте более высокое дерево родителем меньшего, тем самым не увеличивая рост человека.

   ```
   subalgo unionSet(u, v: nodes):
       vRoot = findSet(v)
       uRoot = findSet(u)

       if vRoot == uRoot:
           return

       if vRoot.height < uRoot.height:
           vRoot.parent = uRoot
       else if vRoot.height > uRoot.height:
           uRoot.parent = vRoot
   ```

```
    else:
        uRoot.parent = vRoot
        uRoot.height =  uRoot.height + 1
```

Это приводит к `O(alpha(n))` времени для каждой операции, где `alpha` является обратной для быстрорастущей функции Аккермана, поэтому она очень медленно растет и может быть рассмотрена `O(1)` для практических целей.

Это делает весь алгоритм Крускала `O(m log m + m) = O(m log m)` из-за начальной сортировки.

Заметка

Сжатие пути может уменьшить высоту дерева, поэтому сравнение высот деревьев во время операции объединения может не быть тривиальной задачей. Следовательно, чтобы избежать сложности хранения и вычисления высоты деревьев, результирующий родитель может быть выбран случайным образом:

```
    subalgo unionSet(u, v: nodes):
        vRoot = findSet(v)
        uRoot = findSet(u)

        if vRoot == uRoot:
            return
        if random() % 2 == 0:
            vRoot.parent = uRoot
        else:
            uRoot.parent = vRoot
```

На практике этот рандомизированный алгоритм вместе с сжатием пути для операции `findSet` приведет к сопоставимой производительности, но намного проще реализовать.

Простая реализация на высоком уровне

Сортируйте ребра по значению и добавьте каждый в MST в отсортированном порядке, если он не создает цикл.

```
algorithm kruskalMST(G: a graph)
    sort G's edges by their value
    MST = an empty graph
    for each edge e in G:
        if adding e to MST does not create a cycle:
            add e to MST

    return MST
```

Прочитайте Алгоритм Крускаля онлайн: https://riptutorial.com/ru/algorithm/topic/2583/алгоритм-крускаля

глава 13: Алгоритм максимального субаруса

Examples

Алгоритм максимальной субары

Проблема с максимальным субаравием - это метод поиска смежного подмассива в одномерном массиве чисел с наибольшей суммой.

Первоначально проблема была предложена Ульфом Гренандером Университета Брауна в 1977 году как упрощенная модель для оценки максимального правдоподобия моделей в оцифрованных изображениях.

Мы можем решить эту проблему, рассмотрим список различных целых чисел. Нам может быть интересно, в какой сумме будет располагаться полностью смежное подмножество. Например, если у нас есть массив `[0, 1, 2, -2, 3, 2]` , максимальный подмассив равен `[3, 2]` с суммой `5` .

Подход Brute-Force для решения:

Этот метод является наиболее неэффективным, чтобы найти решение. В этом мы закончим тем, что пройдем через каждый возможный подмассив, а затем найдем сумму всех из них. Наконец, сравните все значения и найдите максимальный субарах.

Псевдокод для подхода Brute-Force:

```
MaxSubarray(array)
  maximum = 0
  for i in input
    current = 0
    for j in input
       current += array[j]
       if current > maximum
         maximum = current
  return maximum
```

Сложность времени для метода Brute-Force равна `O(n^2)` . Итак, давайте двигаться, чтобы разделить и победить подход.

Разделить и покорить подход к решению:

Найдите сумму подмассивов с левой стороны, подмассивы справа. Затем просмотрите все те, которые пересекают центральный деление и, наконец, вернут максимальную сумму. Поскольку это алгоритм разделения и покорения, нам нужно иметь две разные функции.

Сначала - шаг разделения,

```
maxSubarray(array)
  if start = end
    return array[start]
  else
    middle = (start + end) / 2
    return max(maxSubarray(array(From start to middle)), maxSubarray(array(From middle + 1 to
end)), maxCrossover(array))
```

Во второй части отделите другую часть, созданную в первой части.

```
maxCrossover(array)
  currentLeftSum = 0
      leftSum = 0
  currentRightSum = 0
      rightSum = 0
  for i in array
    currentLeftSum += array[i]
    if currentLeftSum > leftSum
      leftSum = currentLeftSum
  for i in array
    currentRightSum += array[i]
    if currentRightSum > rightSum
      rightSum = currentRightSum
  return leftSum + rightSum
```

Сложность времени для метода Divide и Conquer - `O(nlogn)` . Итак, перейдем к подходу к динамическому программированию.

Подход к динамическому программированию:

Это решение также известно как алгоритм Кадана. Это линейный алгоритм времени. Это решение дано Джозефом Б. Кадане в конце 70-х годов.

Этот алгоритм просто проходит цикл, непрерывно изменяя текущую максимальную сумму. Интересно, что это очень простой пример алгоритма динамического программирования, поскольку он выполняет проблему перекрытия и уменьшает его, поэтому мы можем найти более эффективное решение.

Псевдокод алгоритма Кадане:

```
MaxSubArray(array)
  max = 0
  currentMax = 0
  for i in array
    currentMax += array[i]
    if currentMax < 0
      currentMax = 0
    if max < currentMax
      max = currentMax
  return max
```

Сложность времени для алгоритма Кадане - `O(n)` .

Реализация C

```
public class MaximumSubarray
{
    private static int Max(int a, int b)
    {
        return a > b ? a : b;
    }

    static int MaxSubArray(int[] input, int n)
    {
        int max = input[0];
        int currentMax = input[0];
        for (int i = 1; i < n; i++)
        {
            currentMax = Max(input[i], currentMax + input[i]);
            max = Max(max, currentMax);
        }
        return max;
    }

    public static int Main(int[] input)
    {
        int n = input.Length;
        return MaxSubArray(input, n);
    }
}
```

Прочитайте Алгоритм максимального субаруса онлайн:
https://riptutorial.com/ru/algorithm/topic/7578/алгоритм-максимального-субаруса

глава 14: Алгоритм максимальной длины пути

Examples

Maximum Path Sum Основная информация

Maximum Path Sum - это алгоритм для определения пути, так что сумма элемента (узла) этого пути больше любого другого пути.

Например, у нас есть треугольник, как показано ниже.

```
        3
      7   4
    2   4   6
  8   5   9   3
```

В вышеприведенном треугольнике найдите максимальный путь, который имеет максимальную сумму. Ответ: `3 + 7 + 4 + 9 = 23`

Чтобы найти решение, как всегда мы получаем представление о методе Брута-Силы. Метод Brute-Force хорош для этого четырехстрочного треугольника, но подумайте о треугольнике со 100 или более 100 рядами. Итак, мы не можем использовать метод Brute-Force для решения этой проблемы.

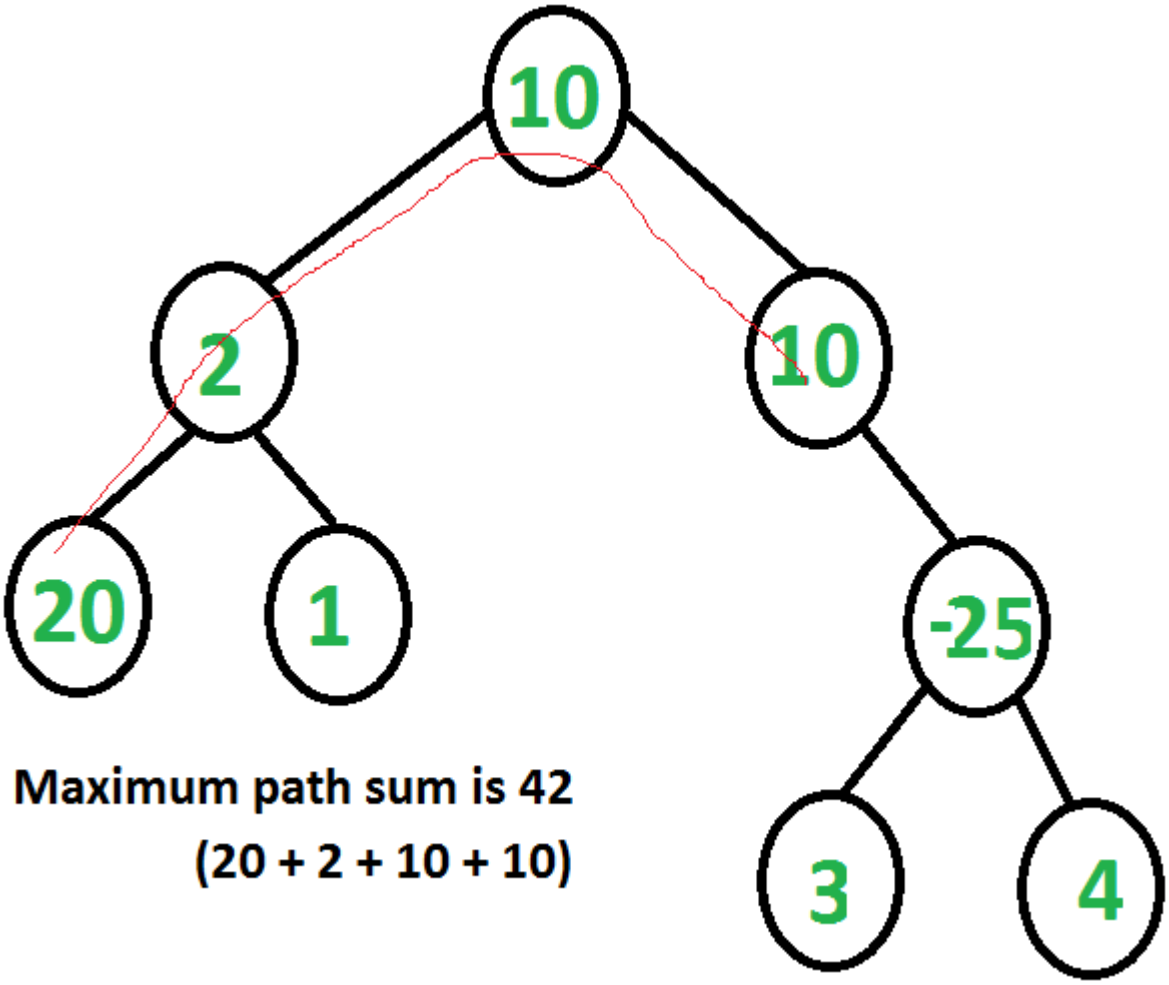
Перейдем к подходу к динамическому программированию.

Алгоритм:

Для каждого узла в треугольнике или в двоичном дереве может быть четыре пути, по которым максимальный путь проходит через узел.

1. Только узел
2. Максимальный путь через левый ребенок + узел
3. Максимальный путь через правый дочерний узел + узел
4. Максимальный путь через левый ребенок + узел + максимальный путь через правый ребенок.

Пример алгоритма максимальной длины пути:



Вспомогательный объем: `O(n)`
Сложность времени: `O(n)`

Реализация C

```
public class Node
{

    public int Value;
    public Node Left, Right;

    public Node(int item)
    {
        Value = item;
        Left = Right = null;
    }
}

class Res
{
    public int Val;
}

public class MaximumPathSum
{
    Node _root;
    int FindMaxUtil(Node node, Res res)
```

```
    {
        if (node == null) return 0;
        int l = FindMaxUtil(node.Left, res);
        int r = FindMaxUtil(node.Right, res);
        int maxSingle = Math.Max(Math.Max(l, r) + node.Value, node.Value);
        int maxTop = Math.Max(maxSingle, l + r + node.Value);
        res.Val = Math.Max(res.Val, maxTop);
        return maxSingle;
    }

    int FindMaxSum()
    {
        return FindMaxSum(_root);
    }

    int FindMaxSum(Node node)
    {
        Res res = new Res {Val = Int32.MinValue};
        FindMaxUtil(node, res);
        return res.Val;
    }

    public static int Main()
    {
        MaximumPathSum tree = new MaximumPathSum
        {
            _root = new Node(10)
            {
                Left = new Node(2),
                Right = new Node(10)
            }
        };
        tree._root.Left.Left = new Node(20);
        tree._root.Left.Right = new Node(1);
        tree._root.Right.Right = new Node(-25)
        {
            Left = new Node(3),
            Right = new Node(4)
        };
        return tree.FindMaxSum();
    }
}
```

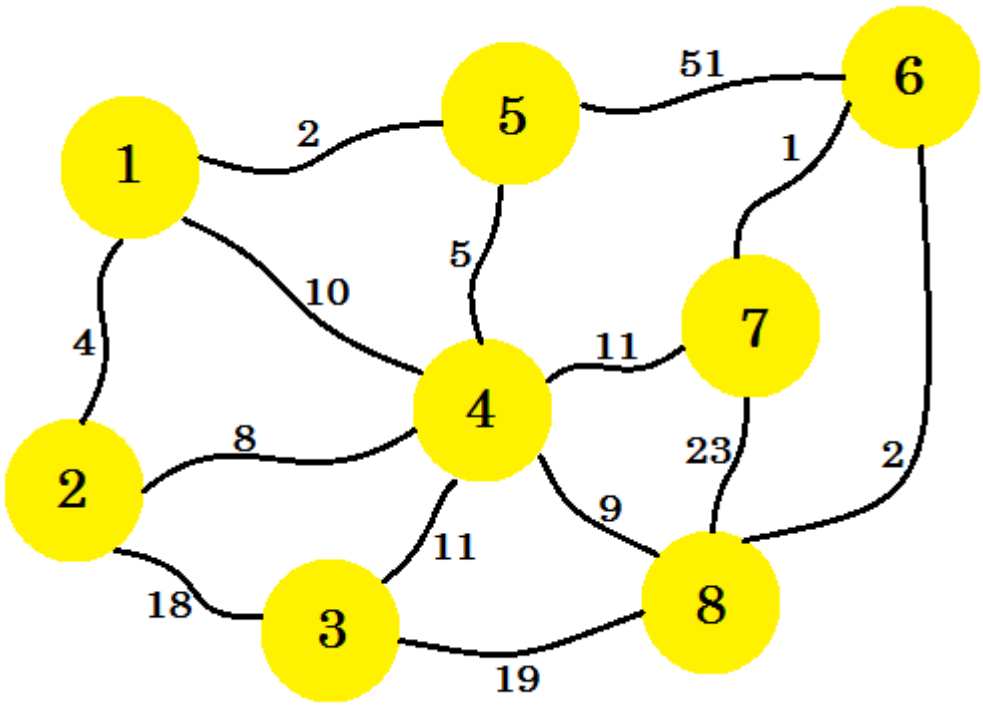
Прочитайте Алгоритм максимальной длины пути онлайн:
https://riptutorial.com/ru/algorithm/topic/7570/алгоритм-максимальной-длины-пути

глава 15: Алгоритм Прима

Examples

Введение в алгоритм Прима

Допустим, у нас **8** домов. Мы хотим установить телефонные линии между этими домами. Краем между домами является стоимость установки линии между двумя домами.



Наша задача - настроить линии таким образом, чтобы все дома были подключены, а стоимость настройки всего соединения минимальна. Теперь, как мы это выясним? Мы можем использовать **алгоритм Прима** .

Алгоритм Прима - жадный алгоритм, который находит минимальное связующее дерево для взвешенного неориентированного графа. Это означает, что он находит подмножество ребер, которые образуют дерево, которое включает в себя каждый узел, где общий вес всех ребер в дереве минимизируется. Алгоритм был разработан в 1930 году чешским математиком Войтехом Ярником, а затем вновь открыт и переиздан компьютерным ученым Робертом Клеем Примом в 1957 году и Эдсгером Вибе Дикстром в 1959 году. Он также известен как **алгоритм DJP, алгоритм Ярника, алгоритм Прима-Ярника** или **Прим-Дижсктра алгоритм** .

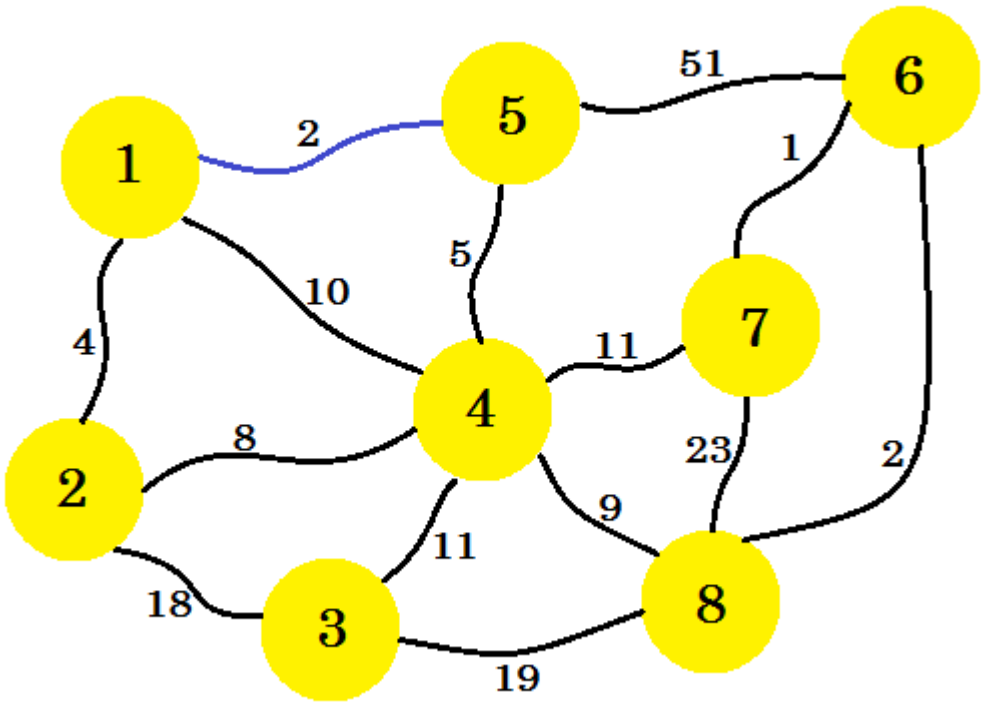
Теперь давайте сначала рассмотрим технические термины. Если мы создадим граф, **S** используя некоторые узлы и ребра неориентированного графа **G** , то **S** называется

подграфом графа **G.** Теперь **S** будет называться **Spanning Tree** тогда и только тогда, когда:

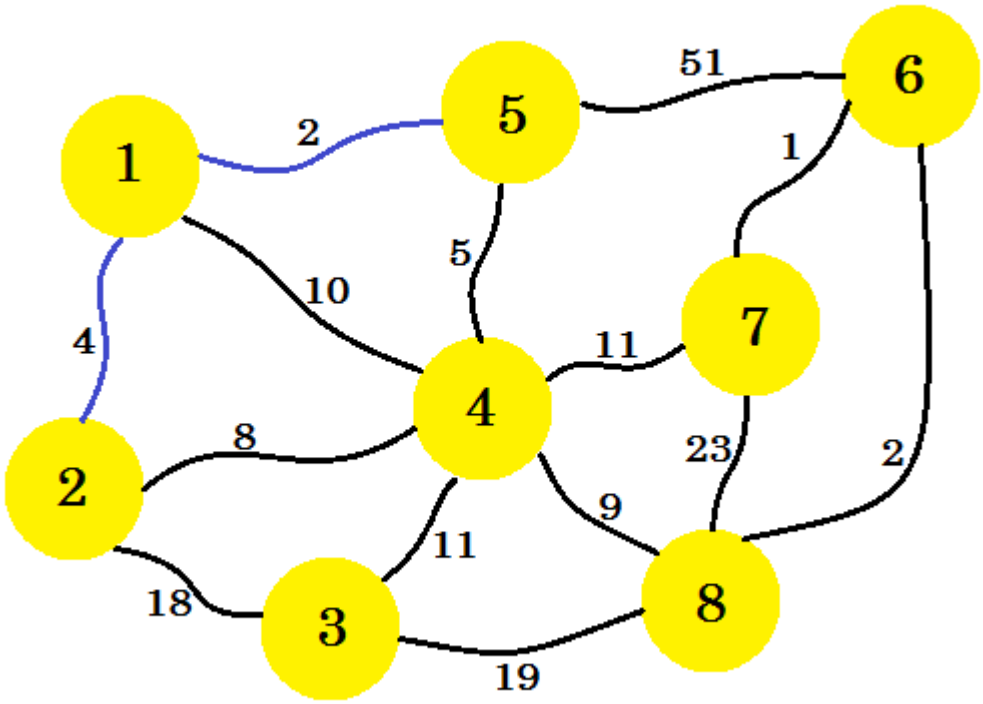
- Он содержит все узлы группы **G.**
- Это дерево, это означает, что нет цикла, и все узлы связаны.
- В дереве есть **(n-1)** ребра, где **n** - число узлов в **G.**

На графике может быть много **Spanning Tree** . **Минимальное связующее дерево** взвешенного неориентированного графа - это дерево, так что сумма веса ребер минимальна. Теперь мы будем использовать **алгоритм Prim,** чтобы узнать минимальное связующее дерево, то есть как настроить телефонные линии на нашем примере графика таким образом, чтобы стоимость установки была минимальной.

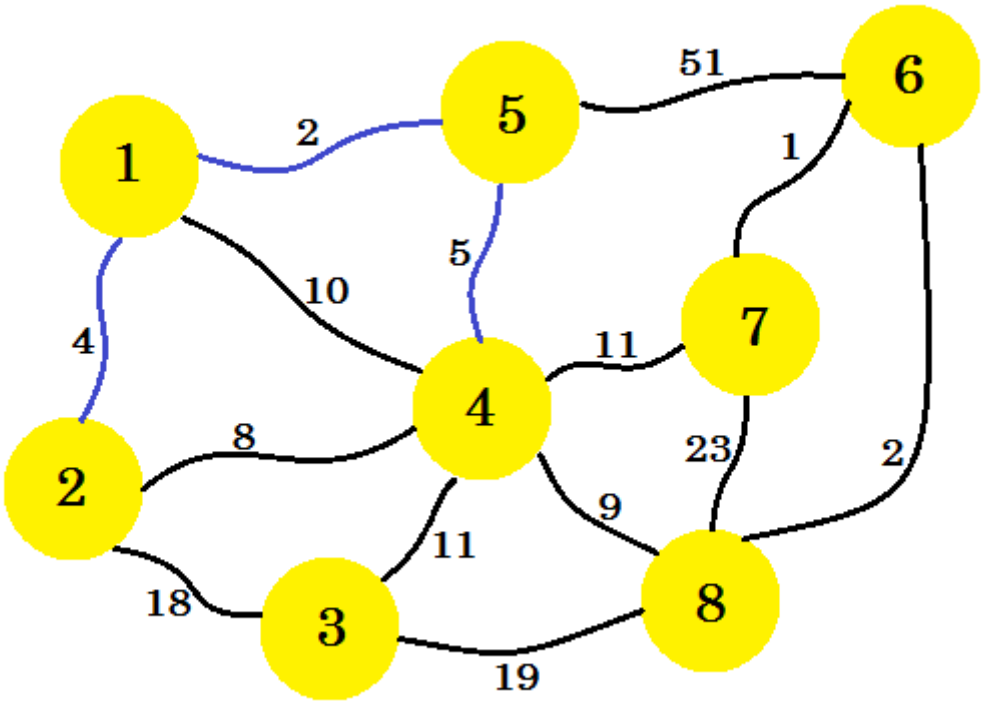
Сначала мы выберем **исходный** узел. Скажем, **узел-1** - наш **источник** . Теперь мы добавим край из **узла-1,** который имеет минимальную стоимость для нашего подграфа. Здесь мы отмечаем края, которые находятся в подграфе, используя **синий** цвет. Здесь **1-5** - наше искомое ребро.



Теперь рассмотрим все ребра из **узла-1** и **узла-5** и возьмем минимум. Поскольку **1-5** уже отмечено, мы принимаем **1-2** .



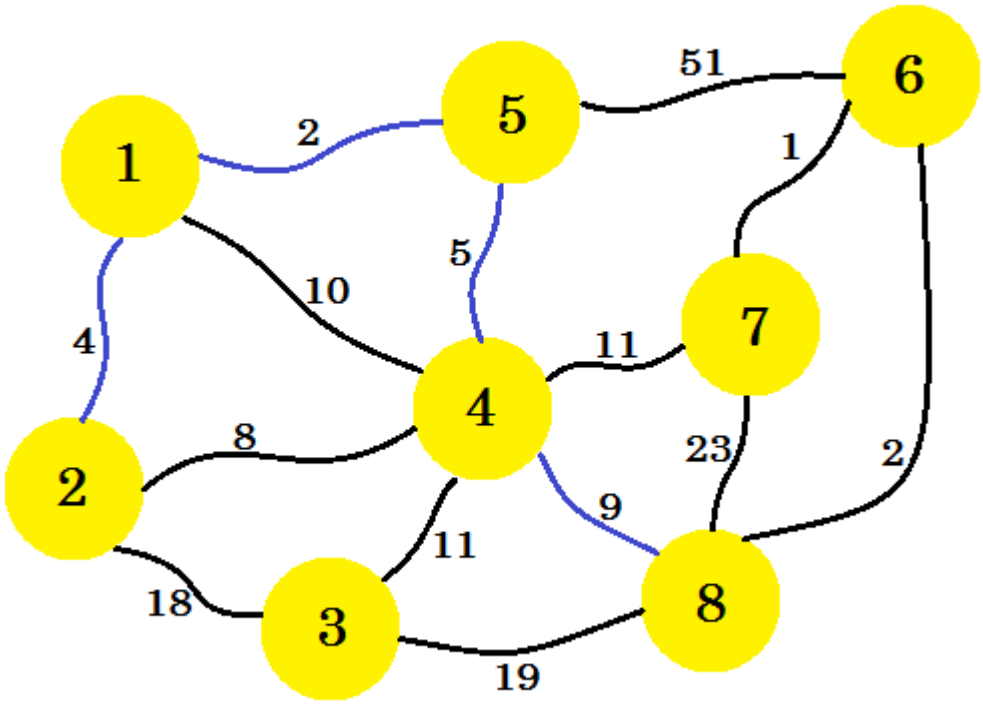
На этот раз мы рассмотрим **узел-1** , **узел-2** и **узел-5** и возьмем минимальный край, который



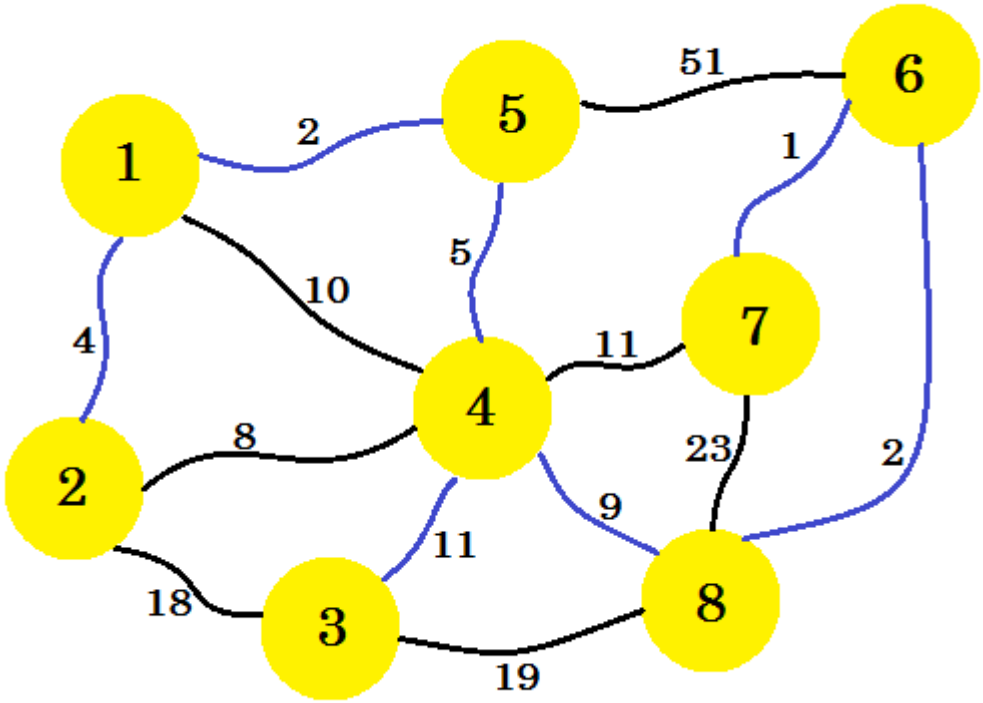
равен **5-4** .

Следующий шаг важен. От **узла-1** , **узла-2** , **узла-5** и **узла-4** минимальный край равен **2-4** . Но если мы выберем тот, он создаст цикл в нашем подграфе. Это связано с тем, что **узел-2** и **узел-4** уже находятся в нашем подграфе. Поэтому преимущество **2-4** не приносит нам пользы. *Мы будем выбирать ребра таким образом, чтобы он добавлял новый узел в наш*

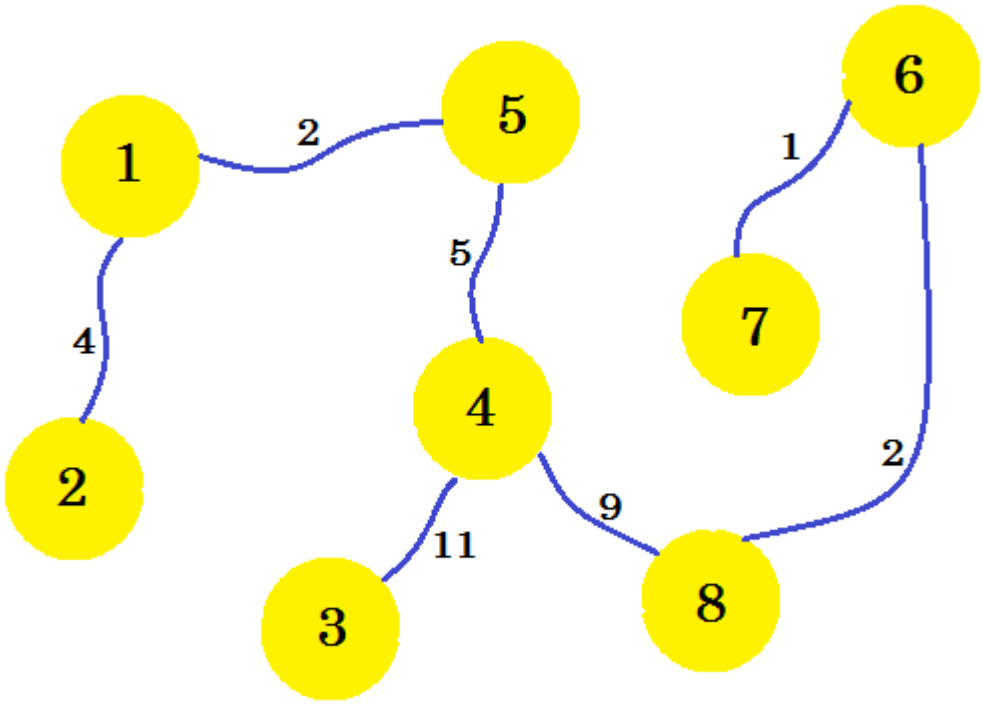
подграф . Поэтому мы выбираем край **4-8** .



Если мы продолжим этот путь, мы выберем края **8-6** , **6-7** и **4-3** . Наш подграф будет выглядеть так:



Это наш желаемый подграф, который даст нам минимальное остовное дерево. Если мы удалим ребра, которые мы не выбрали, мы получим:



Это наше **минимальное связующее дерево** (MST). Таким образом, стоимость настройки телефонных соединений составляет: **4** + **2** + **5** + **11** + **9** + **2** + **1** = **34** . И набор домов и их соединений показаны на графике. Может быть несколько **MST** графика. Это зависит от **выбранного** узла- **источника** .

Псевдокод алгоритма приведен ниже:

```
Procedure PrimsMST(Graph):      // here Graph is a non-empty connected weighted graph
Vnew[] = {x}                    // New subgraph Vnew with source node x
Enew[] = {}
while Vnew is not equal to V
    u -> a node from Vnew
    v -> a node that is not in Vnew such that edge u-v has the minimum cost
                                // if two nodes have same weight, pick any of them
    add v to Vnew
    add edge (u, v) to Enew
end while
Return Vnew and Enew
```

Сложность:

Сложность времени наивысшего подхода - **O (V²)** . Он использует матрицу смежности. Мы можем уменьшить сложность с помощью очереди приоритетов. Когда мы добавляем новый узел в **Vnew** , мы можем добавить его соседние ребра в очереди приоритетов. Затем вытащите минимальный взвешенный край из него. Тогда сложность будет: **O (ElogE)** , где **E** - количество ребер. Снова можно построить **двоичную кучу,** чтобы уменьшить сложность до **O (ElogV)** .

Псевдокод с использованием очереди приоритетов приведен ниже:

```
Procedure MSTPrim(Graph, source):
for each u in V
    key[u] := inf
    parent[u] := NULL
end for
key[source] := 0
Q = Priority_Queue()
Q = V
while Q is not empty
    u -> Q.pop
    for each v adjacent to i
        if v belongs to Q and Edge(u,v) < key[v]    // here Edge(u, v) represents
                                                    // cost of edge(u, v)
            parent[v] := u
            key[v] := Edge(u, v)
        end if
    end for
end while
```

Здесь **ключ []** хранит минимальную стоимость перемещения **узла-v** . **parent []** используется для хранения родительского узла. Это полезно для перемещения и печати дерева.

Ниже приведена простая программа на Java:

```
import java.util.*;

public class Graph
{
   private static int infinite = 9999999;
   int[][]  LinkCost;
   int      NNodes;
   Graph(int[][] mat)
   {
      int i, j;
      NNodes = mat.length;
      LinkCost = new int[NNodes][NNodes];
      for ( i=0; i < NNodes; i++)
      {
         for ( j=0; j < NNodes; j++)
         {
            LinkCost[i][j] = mat[i][j];
            if ( LinkCost[i][j] == 0 )
               LinkCost[i][j] = infinite;
         }
      }
      for ( i=0; i < NNodes; i++)
      {
         for ( j=0; j < NNodes; j++)
            if ( LinkCost[i][j] < infinite )
               System.out.print( " " + LinkCost[i][j] + " " );
            else
               System.out.print(" * " );
         System.out.println();
      }
   }
   public int unReached(boolean[] r)
```

```
  {
     boolean done = true;
     for ( int i = 0; i < r.length; i++ )
        if ( r[i] == false )
           return i;
     return -1;
  }
  public void Prim( )
  {
     int i, j, k, x, y;
     boolean[] Reached = new boolean[NNodes];
     int[] predNode = new int[NNodes];
     Reached[0] = true;
     for ( k = 1; k < NNodes; k++ )
     {
        Reached[k] = false;
     }
     predNode[0] = 0;
     printReachSet( Reached );
     for (k = 1; k < NNodes; k++)
     {
        x = y = 0;
        for ( i = 0; i < NNodes; i++ )
           for ( j = 0; j < NNodes; j++ )
           {
               if ( Reached[i] && !Reached[j] &&
                    LinkCost[i][j] < LinkCost[x][y] )
               {
                  x = i;
                  y = j;
               }
           }
        System.out.println("Min cost edge: (" +
                               + x + "," +
                               + y + ")" +
                               "cost = " + LinkCost[x][y]);
        predNode[y] = x;
        Reached[y] = true;
        printReachSet( Reached );
        System.out.println();
     }
     int[] a= predNode;
  for ( i = 0; i < NNodes; i++ )
         System.out.println( a[i] + " --> " + i );
  }
  void printReachSet(boolean[] Reached )
  {
     System.out.print("ReachSet = ");
     for (int i = 0; i < Reached.length; i++ )
        if ( Reached[i] )
          System.out.print( i + " ");
     //System.out.println();
  }
public static void main(String[] args)
  {
     int[][] conn = {{0,3,0,2,0,0,0,0,4},  // 0
                     {3,0,0,0,0,0,0,4,0},  // 1
                     {0,0,0,6,0,1,0,2,0},  // 2
                     {2,0,6,0,1,0,0,0,0},  // 3
                     {0,0,0,1,0,0,0,0,8},  // 4
                     {0,0,1,0,0,0,8,0,0},  // 5
```

```
                        {0,0,0,0,0,8,0,0,0},   // 6
                        {0,4,2,0,0,0,0,0,0},   // 7
                        {4,0,0,0,8,0,0,0,0}    // 8
                       };
      Graph G = new Graph(conn);
      G.Prim();
   }
}
```

Скомпилируйте приведенный выше код с помощью `javac Graph.java`

Выход:

```
$ java Graph
 *  3  *  2  *  *  *  *  4
 3  *  *  *  *  *  *  4  *
 *  *  *  6  *  1  *  2  *
 2  *  6  *  1  *  *  *  *
 *  *  *  1  *  *  *  *  8
 *  *  1  *  *  *  8  *  *
 *  *  *  *  *  8  *  *  *
 *  4  2  *  *  *  *  *  *
 4  *  *  *  8  *  *  *  *
ReachSet = 0 Min cost edge: (0,3)cost = 2
ReachSet = 0 3
Min cost edge: (3,4)cost = 1
ReachSet = 0 3 4
Min cost edge: (0,1)cost = 3
ReachSet = 0 1 3 4
Min cost edge: (0,8)cost = 4
ReachSet = 0 1 3 4 8
Min cost edge: (1,7)cost = 4
ReachSet = 0 1 3 4 7 8
Min cost edge: (7,2)cost = 2
ReachSet = 0 1 2 3 4 7 8
Min cost edge: (2,5)cost = 1
ReachSet = 0 1 2 3 4 5 7 8
Min cost edge: (5,6)cost = 8
ReachSet = 0 1 2 3 4 5 6 7 8
0 --> 0
0 --> 1
7 --> 2
0 --> 3
3 --> 4
2 --> 5
5 --> 6
1 --> 7
0 --> 8
```

Прочитайте Алгоритм Прима онлайн: https://riptutorial.com/ru/algorithm/topic/7285/алгоритм-прима

глава 16: Алгоритм раздвижного окна

Examples

Алгоритм скользящего окна Основная информация

Алгоритм скользящего окна используется для выполнения требуемой операции для конкретного размера окна заданного большого буфера или массива. Окно начинается с 1-го элемента и смещается справа на один элемент. Цель состоит в том, чтобы найти минимальные k чисел, присутствующих в каждом окне. Это обычно известно как проблема или алгоритм раздвижного окна.

Например, чтобы найти максимальный или минимальный элемент из каждого `n` элемента в заданном массиве, используется алгоритм скользящего окна.

Пример:

Входной массив: [1 3 -1 -3 5 3 6 7]

Размер окна: 3

Максимальный элемент из каждых 3 элементов входного массива:

```
+---------------------------------+---------+
|      Windows Position           |   Max   |
+-----------+----+---+---+---+---+---------+
|[1   3   -1]| -3 | 5 | 3 | 6 | 7 |    3    |
+-----------+----+---+---+---+---+---------+
| 1 |[3   -1   -3]| 5 | 3 | 6 | 7 |    3    |
+---+-------------+---+---+---+---+---------+
| 1 | 3 |[-1   -3   5]| 3 | 6 | 7 |    5    |
+---+---+-------------+---+---+---+---------+
| 1 | 3 | -1 |[-3   5   3]| 6 | 7 |    5    |
+---+---+----+------------+---+---+---------+
| 1 | 3 | -1 | -3 |[5   3   6]| 7 |    6    |
+---+---+----+----+-----------+---+---------+
| 1 | 3 | -1 | -3 | 5 |[3   6   7]|    7    |
+---+---+----+----+---+-----------+---------+
```

Минимальный элемент из каждых 3 элементов входного массива:

```
+---------------------------------+---------+
|      Windows Position           |   Min   |
+-----------+----+---+---+---+---+---------+
|[1   3   -1]| -3 | 5 | 3 | 6 | 7 |   -1    |
+-----------+----+---+---+---+---+---------+
| 1 |[3   -1   -3]| 5 | 3 | 6 | 7 |   -3    |
+---+-------------+---+---+---+---+---------+
| 1 | 3 |[-1   -3   5]| 3 | 6 | 7 |   -3    |
+---+---+-------------+---+---+---+---------+
```

```
| 1 | 3 | -1 |[-3   5   3]| 6 | 7 |   -3    |
+---+---+----+------------+---+---+---------+
| 1 | 3 | -1 | -3 |[5   3   6]| 7 |    3    |
+---+---+----+----+-----------+---+---------+
| 1 | 3 | -1 | -3 | 5 |[3   6   7]|    3    |
+---+---+----+----+---+-----------+---------+
```

Методы для нахождения суммы 3 элемента:

Способ 1:

Первый способ - использовать быструю сортировку, когда точка поворота находится в позиции Kth, все элементы с правой стороны больше, чем точка поворота, поэтому все элементы с левой стороны автоматически становятся K наименьшими элементами заданного массива.

Способ 2:

Храните массив элементов K, заполните его с помощью первых элементов K данного массива ввода. Теперь из элемента K + 1 проверьте, является ли текущий элемент меньше максимального элемента во вспомогательном массиве, если да, добавьте этот элемент в массив. Единственная проблема с вышеупомянутым решением заключается в том, что нам нужно отслеживать максимальный элемент. Все еще работоспособно. Как мы можем отслеживать максимальный элемент в наборе integer? Подумайте, куча. Подумайте, Макс куча.

Способ 3:

Большой! В O (1) мы получили бы максимальный элемент среди K элементов, которые были выбраны как самые маленькие K-элементы. Если max в текущем наборе больше, чем новый элемент, нам нужно удалить max и ввести новый элемент в множество наименьшего элемента K. Heapify снова, чтобы сохранить свойство кучи. Теперь мы можем легко получить K минимальных элементов в массиве N.

Вспомогательный объем: `O(n)`

Сложность времени: `O(n)`

Реализация алгоритма раздвижного окна в C

```
public class SlidingWindow
{
    public static int[] MaxSlidingWindow(int[] input, int k)
    {
        int[] result = new int[input.Length - k + 1];
        for (int i = 0; i <= input.Length - k; i++)
        {
            var max = input[i];
            for (int j = 1; j < k; j++)
```

```
            {
                if (input[i + j] > max) max = input[i + j];
            }
            result[i] = max;
        }
        return result;
    }

    public static int[] MinSlidingWindow(int[] input, int k)
    {
        int[] result = new int[input.Length - k + 1];
        for (int i = 0; i <= input.Length - k; i++)
        {
            var min = input[i];
            for (int j = 1; j < k; j++)
            {
                if (input[i + j] < min) min = input[i + j];
            }
            result[i] = min;
        }
        return result;
    }

    public static int[] MainMax(int[] input, int n)
    {
        return MaxSlidingWindow(input, n);
    }

    public static int[] MainMin(int[] input, int n)
    {
        return MinSlidingWindow(input, n);
    }
}
```

Прочитайте Алгоритм раздвижного окна онлайн: https://riptutorial.com/ru/algorithm/topic/7622/алгоритм-раздвижного-окна

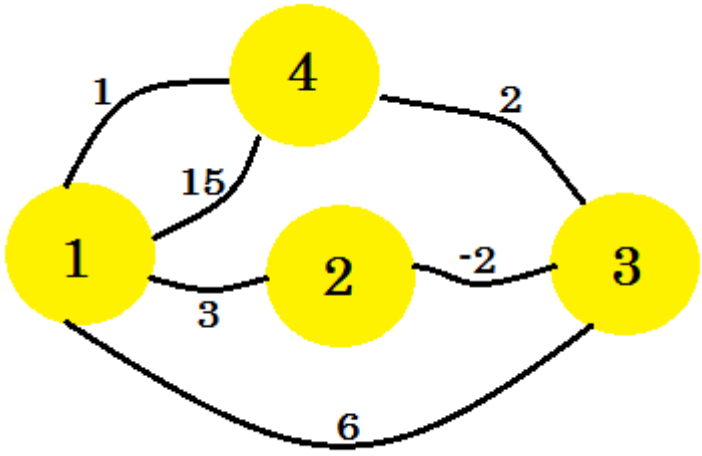
глава 17: Алгоритм Флойда-Варшалла

Examples

Алгоритм всех парных кратчайших путей

Алгоритм Флойда-Варшалла заключается в поиске кратчайших путей в взвешенном графе с положительными или отрицательными весами ребер. Однократное выполнение алгоритма найдет длины (суммированные веса) кратчайших путей между всеми парами вершин. С небольшим изменением он может печатать кратчайший путь и может обнаруживать отрицательные циклы в графике. Floyd-Warshall - это алгоритм динамического программирования.

Давайте посмотрим на пример. Мы будем применять алгоритм Флойда-Варшалла на этом



графике:

Первое, что мы делаем, - взять две двумерные матрицы. Это матрицы смежности . Размер матриц будет полным числом вершин. Для нашего графика возьмем **4 * 4** матрицы. **Матрица** расстояний будет хранить минимальное расстояние, найденное до сих пор между двумя вершинами. Сначала для ребер, если есть ребро между **uv** и расстояние / вес **w** , мы будем хранить: `distance[u][v] = w` . Для всех ребер, которые не существуют, мы собираемся положить *бесконечность* . **Матрица пути** предназначена для регенерации минимального расстояния между двумя вершинами. Поэтому, если есть путь между **u** и **v** , мы собираемся поместить `path[u][v] = u` . Это означает, что лучший способ прийти к **вершине-v** из **вершины-u** - использовать ребро, соединяющее **v** с **u** . Если между двумя вершинами нет пути, мы собираемся указать **N,** указав, что пути нет. Две таблицы для нашего графика будут выглядеть так:

```
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|     |  1  |  2  |  3  |  4  |          |     |  1  |  2  |  3  |  4  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  1  |  0  |  3  |  6  |  15 |          |  1  |  N  |  1  |  1  |  1  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
```

```
|  2  | inf |  0  | -2  | inf |          |  2  |  N  |  N  |  2  |  N  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  3  | inf | inf |  0  |  2  |          |  3  |  N  |  N  |  N  |  3  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  4  |  1  | inf | inf |  0  |          |  4  |  4  |  N  |  N  |  N  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
           distance                                 path
```

Поскольку нет петли, диагонали устанавливаются **N.** И расстояние от самой вершины равно **0** .

Чтобы применить алгоритм Флойда-Варшалла, мы будем выбирать среднюю вершину **k** . Тогда для каждой вершины **i** мы проверим, можем ли мы перейти от **i** к **k,** а затем **к** к **j** , где **j** - другая вершина и минимизирующая стоимость перехода от **i** к **j** . Если текущее **расстояние [i] [j]** больше **расстояния [i] [k]** + **расстояние [k] [j]** , мы собираемся положить **расстояние [i] [j]** равным суммированию этих двух расстояний , И **путь [i] [j]** будет установлен в **путь [k] [j]** , так как лучше перейти от **i** к **k** , а затем **к** к **j** . Все вершины будут выбраны как **k** . У нас будет 3 вложенных цикла: для **k,** идущих от 1 до 4, **я** иду от 1 до 4 и **j** идет от 1 до 4. Мы собираемся проверить:

```
if distance[i][j] > distance[i][k] + distance[k][j]
    distance[i][j] := distance[i][k] + distance[k][j]
    path[i][j] := path[k][j]
end if
```

Итак, что мы в основном проверяем, *для каждой пары вершин мы получаем более короткое расстояние, пройдя через другую вершину?* Общее количество операций для нашего графика будет **4 * 4 * 4** = **64** . Это означает, что мы будем делать эту проверку **64** раза. Давайте посмотрим на некоторые из них:

Когда **k** = **1** , **i** = **2** и **j** = **3** , **расстояние [i] [j]** равно **-2** , что не больше **расстояния [i] [k]** + **расстояние [k] [j]** = **-2** + **0** = **-2** . Таким образом, он останется неизменным. Опять же, когда **k** = **1** , **i** = **4** и **j** = **2** , **расстояние [i] [j]** = **бесконечность** , что больше **расстояния [i] [k]** + **расстояние [k] [j]** = **1** + **3** = **4** , Таким образом, мы положили **расстояние [i] [j]** = **4** и положили **путь [i] [j]** = **path [k] [j]** = **1** . Это означает, что для перехода от **вершины-4** к **вершине-2** путь **4-> 1-> 2** короче существующего пути. Так мы заполняем обе матрицы. Расчет для каждого шага показан здесь . После внесения необходимых изменений наши матрицы будут выглядеть так:

```
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|     |  1  |  2  |  3  |  4  |          |     |  1  |  2  |  3  |  4  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  1  |  0  |  3  |  1  |  3  |          |  1  |  N  |  1  |  2  |  3  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  2  |  1  |  0  | -2  |  0  |          |  2  |  4  |  N  |  2  |  3  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  3  |  3  |  6  |  0  |  2  |          |  3  |  4  |  1  |  N  |  3  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  4  |  1  |  4  |  2  |  0  |          |  4  |  4  |  1  |  2  |  N  |
```

```
+-----+-----+-----+-----+-----+           +-----+-----+-----+-----+-----+
           distance                                 path
```

Это наша кратчайшая матрица расстояния. Например, кратчайшее расстояние от **1** до **4** равно **3,** а кратчайшее расстояние между **4** и **3** равно **2** . Наш псевдокод будет:

```
Procedure Floyd-Warshall(Graph):
for k from 1 to V     // V denotes the number of vertex
    for i from 1 to V
       for j from 1 to V
           if distance[i][j] > distance[i][k] + distance[k][j]
               distance[i][j] := distance[i][k] + distance[k][j]
               path[i][j] := path[k][j]
           end if
       end for
    end for
end for
```

Печать пути:

Чтобы напечатать путь, мы проверим матрицу **Path** . Чтобы напечатать путь от **u** до **v** , мы начнем с **пути [u] [v]** . Мы установим изменение **v** = **path [u] [v],** пока не найдем **путь [u] [v]** = **u** и не выталкиваем все значения **пути [u] [v]** в стеке. После нахождения **u** , мы напечатаем **u** и начнем выскакивать элементы из стека и печатать их. Это работает, потому что в матрице **пути** хранится значение вершины, которое имеет самый короткий путь к **v** из любого другого узла. Псевдокод будет:

```
Procedure PrintPath(source, destination):
s = Stack()
S.push(destination)
while Path[source][destination] is not equal to source
    S.push(Path[source][destination])
    destination := Path[source][destination]
end while
print -> source
while S is not empty
    print -> S.pop
end while
```

Поиск отрицательного краевого цикла:

Чтобы узнать, есть ли отрицательный цикл ребер, нам нужно проверить основную диагональ матрицы **расстояния** . Если какое-либо значение на диагонали отрицательное, это означает, что на графике есть отрицательный цикл.

Сложность:

Сложность алгоритма Флойда-Варшалла равна **O (V³),** а сложность пространства: **O (V²)** .

Прочитайте Алгоритм Флойда-Варшалла онлайн:
https://riptutorial.com/ru/algorithm/topic/7193/алгоритм-флойда-варшалла

глава 18: Быстрое преобразование Фурье

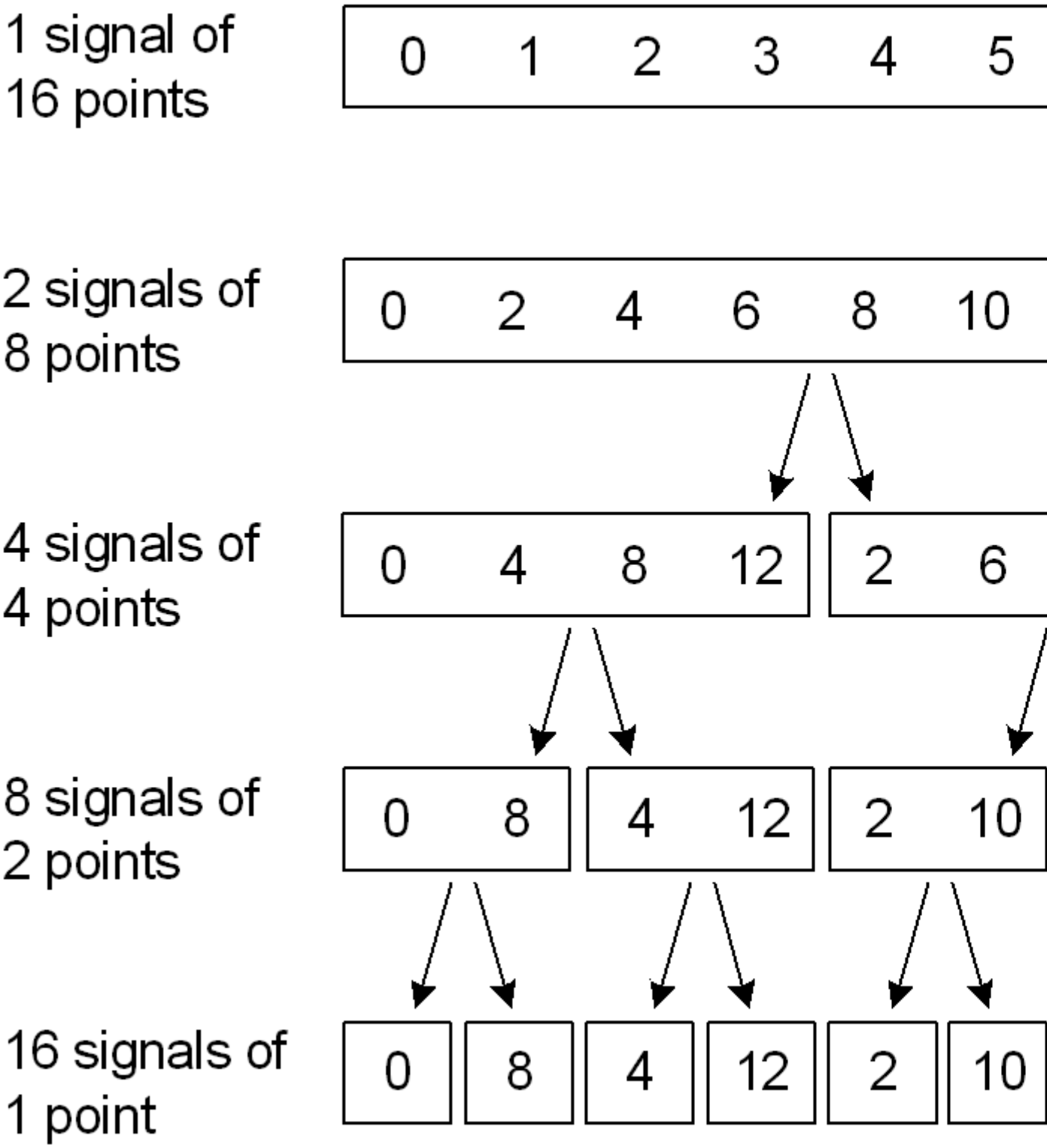
Вступление

Вещественная и комплексная форма ДПФА **(D** iscrete **F** ourier **T** ransforms) может быть использована для выполнения частотного анализа или синтеза для любых дискретных и периодических сигналов. FFT (**F** ast **F** ourier **T** ransform) представляет собой реализацию DFT, которая может быть быстро выполнена на современных процессорах.

Examples

Radix 2 FFT

Простейшим и, возможно, самым известным методом вычисления БПФ является алгоритм Decix-2 Decimation in Time. Radix-2 FFT работает, разлагая N-точечный сигнал во временной области в N сигналов временной области, каждый из которых состоит из одной точки

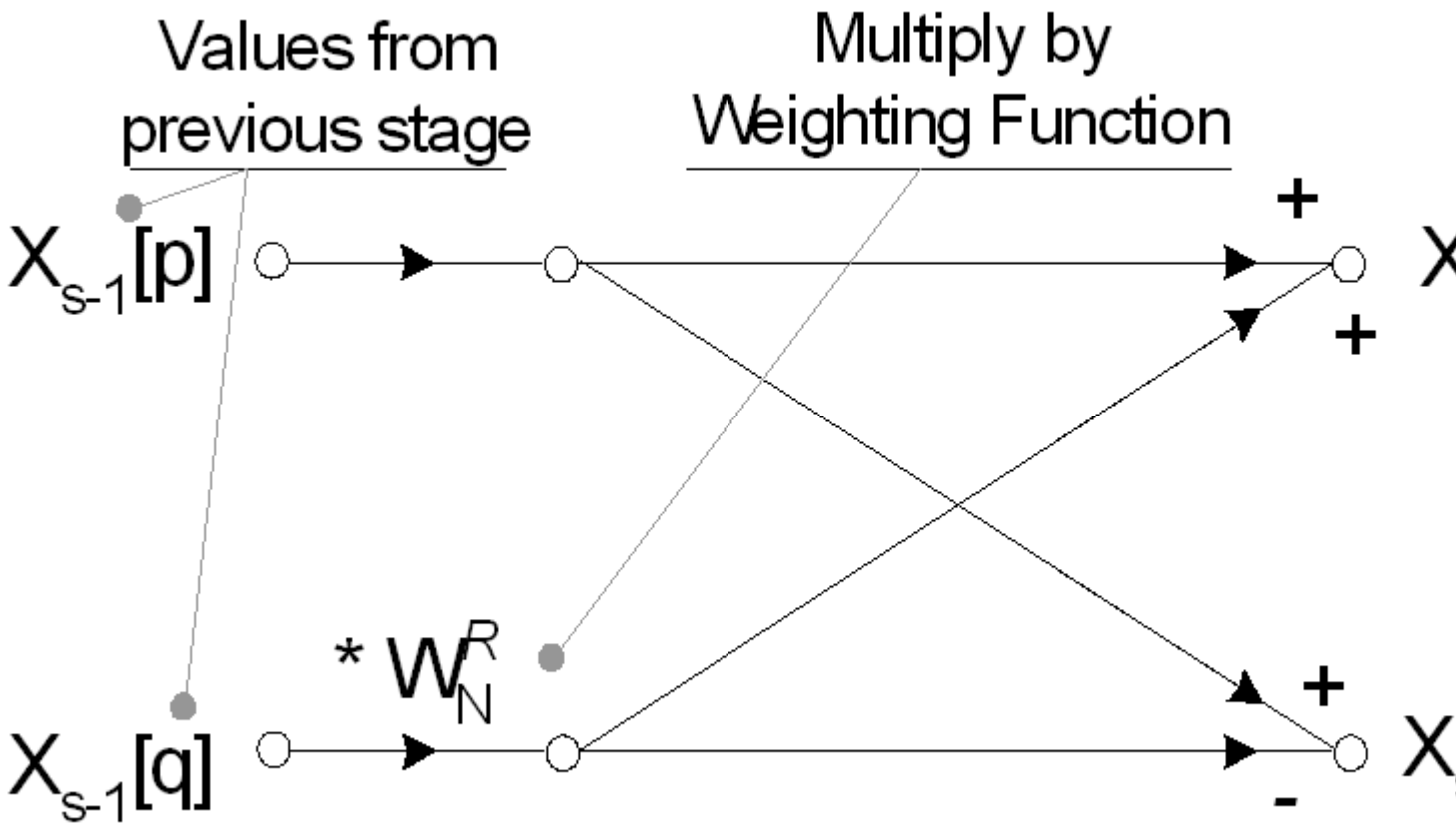


,

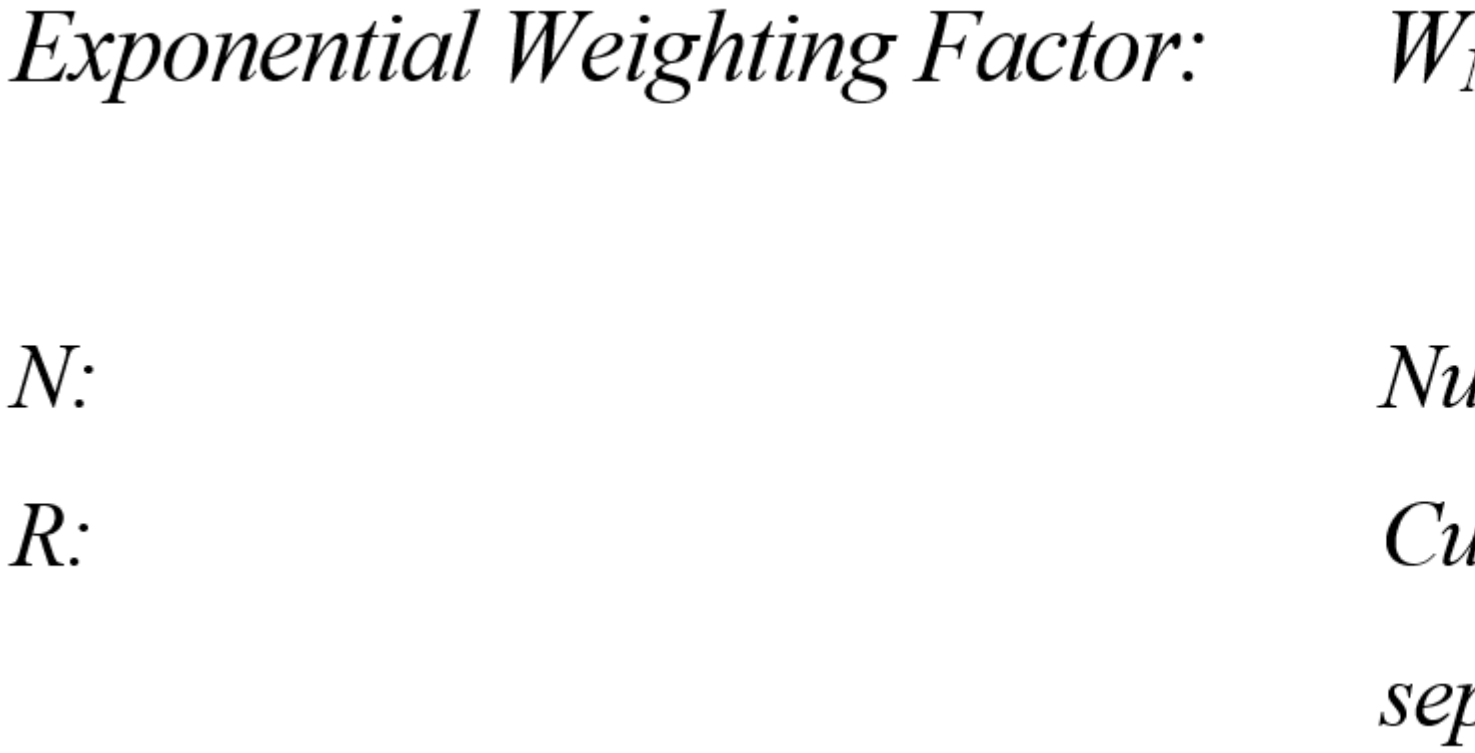
Разложение сигнала или «прореживание во времени» достигается путем бина, изменяющего индексы для массива данных временной области. Таким образом, для шестнадцатиточечного сигнала образец 1 (двоичный 0001) заменяется образцом 8 (1000), образец 2 (0010) заменяется на 4 (0100) и так далее. Образцовая свопинг с использованием метода обратного бита может быть достигнута просто в программном

обеспечении, но ограничивает использование Radix 2 FFT сигналами длиной N = 2 ^ M.

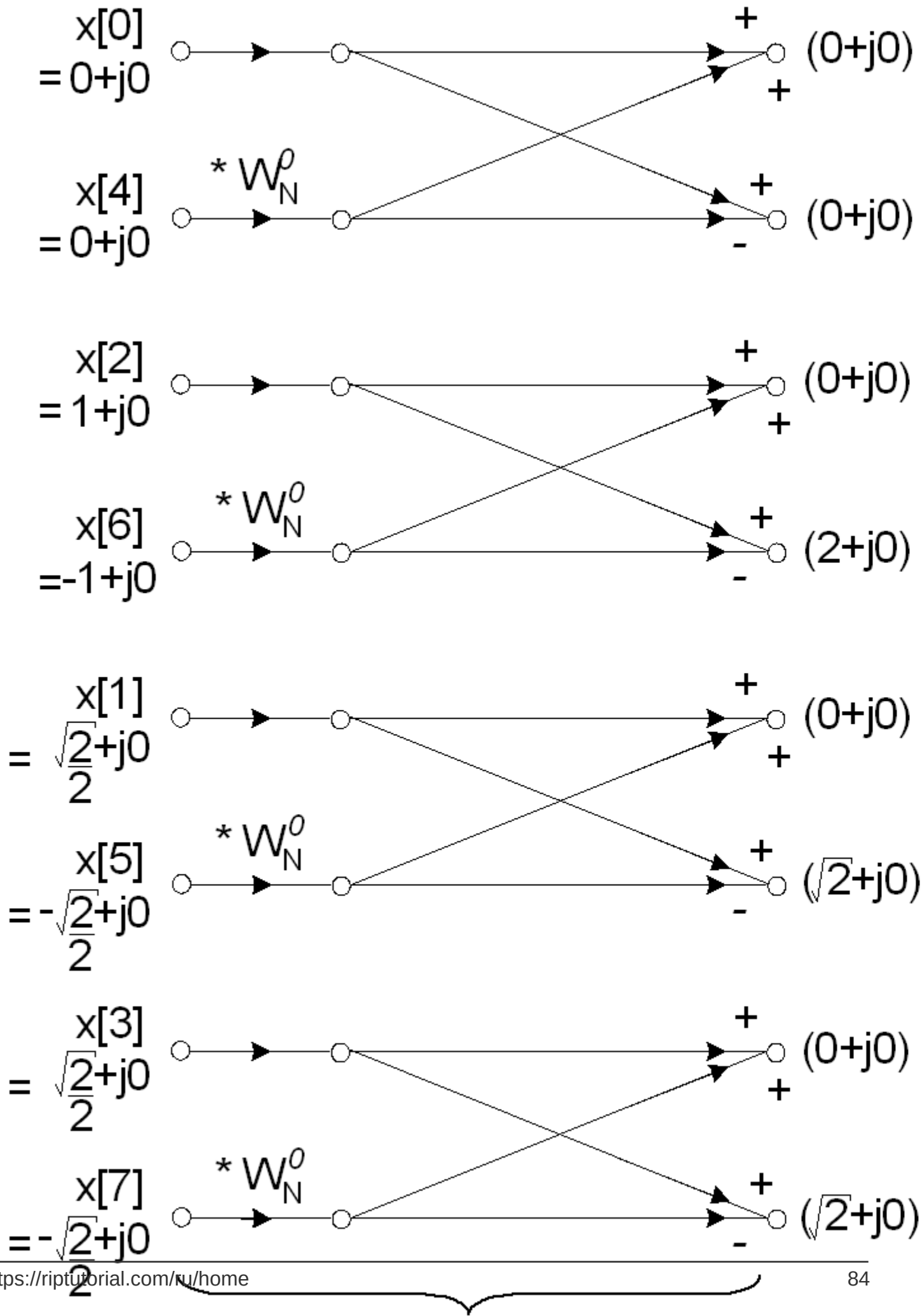
Значение одноточечного сигнала во временной области равно его значению в частотной области, поэтому этот массив разложенных единичных точек временной области не требует преобразования, чтобы стать массивом точек частотной области. N отдельных точек; однако необходимо перестроить в один N-точечный частотный спектр. Оптимальная реконструкция полного частотного спектра выполняется с использованием расчетов бабочки. Каждый этап реконструкции в Fix Radix-2 выполняет множество двухточечных бабочек, используя аналогичный набор экспоненциальных весовых функций Wn ^ R.



БПФ удаляет избыточные вычисления в Дискретном преобразовании Фурье, используя периодичность Wn ^ R. Спектральная реконструкция завершена в log2 (N) этапах расчетов бабочек, дающих X [K]; реальные и мнимые данные частотной области в прямоугольной форме. Для преобразования в амплитуду и фазу (полярные координаты) требуется найти абсолютное значение, √ (Re2 + Im2) и аргумент tan-1 (Im / Re).



Ниже показана полная диаграмма потока бабочек для восьмиступенчатого Fix Radix 2. Обратите внимание, что входные сигналы ранее были переупорядочены в соответствии с процедурой прореживания во времени, описанной ранее.


x[0]
= 0+j0
x[4]
= 0+j0
* W_N^0
+
+
+
-
(0+j0)
(0+j0)
x[2]
= 1+j0
x[6]
=-1+j0
* W_N^0
(0+j0)
(2+j0)
x[1]
= $\frac{\sqrt{2}}{2}$+j0
x[5]
= -$\frac{\sqrt{2}}{2}$+j0
* W_N^0
(0+j0)
($\sqrt{2}$+j0)
x[3]
= $\frac{\sqrt{2}}{2}$+j0
x[7]
= -$\frac{\sqrt{2}}{2}$+j0
* W_N^0
(0+j0)
($\sqrt{2}$+j0)

экспоненциальная весовая мощность) определяется текущим этапом в спектральной реконструкции и вычислением тока в конкретной бабочке.

Пример кода (C / C ++)

Пример кода AC / C ++ для вычисления Radix 2 FFT можно найти ниже. Это простая реализация, которая работает для любого размера N, где N - мощность 2. Она примерно на 3 раза медленнее, чем самая быстрая реализация FFTw, но все же является хорошей основой для будущей оптимизации или для изучения того, как работает этот алгоритм.

```
#include <math.h>

#define PI       3.14159265358979323846264338327795    // PI for sine/cos calculations
#define TWOPI    6.28318530717958647692528676655900     // 2*PI for sine/cos calculations
#define Deg2Rad  0.01745329251994329576923690768488612  // Degrees to Radians factor
#define Rad2Deg  57.29577951308232087679815481410517    // Radians to Degrees factor
#define log10_2  0.30102999566398119521373889472449   // Log10 of 2
#define log10_2_INV 3.32192809488736234787031942949483 // 1/Log10(2)

// complex variable structure (double precision)
struct complex
{
public:
    double  Re, Im;        // Not so complicated after all
};

// Returns true if N is a power of 2
bool isPwrTwo(int N, int *M)
{
    *M = (int)ceil(log10((double)N) * log10_2_INV);// M is number of stages to perform. 2^M =
N
    int NN = (int)pow(2.0, *M);
    if ((NN != N) || (NN == 0)) // Check N is a power of 2.
        return false;

    return true;
}

void rad2FFT(int N, complex *x, complex *DFT)
{
    int M = 0;

    // Check if power of two. If not, exit
    if (!isPwrTwo(N, &M))
        throw "Rad2FFT(): N must be a power of 2 for Radix FFT";

    // Integer Variables

    int BSep;                  // BSep is memory spacing between butterflies
    int BWidth;                // BWidth is memory spacing of opposite ends of the butterfly
    int P;                     // P is number of similar Wn's to be used in that stage
    int j;                     // j is used in a loop to perform all calculations in each
stage
    int stage = 1;             // stage is the stage number of the FFT. There are M stages in
total (1 to M).
    int HiIndex;               // HiIndex is the index of the DFT array for the top value of
each butterfly calc
    unsigned int iaddr;        // bitmask for bit reversal
```

```
    int ii;                    // Integer bitfield for bit reversal (Decimation in Time)
    int MM1 = M - 1;

    unsigned int i;
    int l;
    unsigned int nMax = (unsigned int)N;

    // Double Precision Variables
    double TwoPi_N = TWOPI / (double)N;    // constant to save computational time.  = 2*PI / N
    double TwoPi_NP;

    // complex Variables (See 'struct complex')
    complex WN;               // Wn is the exponential weighting function in the form a + jb
    complex TEMP;             // TEMP is used to save computation in the butterfly calc
    complex *pDFT = DFT;      // Pointer to first elements in DFT array
    complex *pLo;             // Pointer for lo / hi value of butterfly calcs
    complex *pHi;
    complex *pX;              // Pointer to x[n]


    // Decimation In Time - x[n] sample sorting
    for (i = 0; i < nMax; i++, DFT++)
    {
        pX = x + i;             // Calculate current x[n] from base address *x and index i.
        ii = 0;                 // Reset new address for DFT[n]
        iaddr = i;              // Copy i for manipulations
        for (l = 0; l < M; l++) // Bit reverse i and store in ii...
        {
            if (iaddr & 0x01)     // Detemine least significant bit
                ii += (1 << (MM1 - l));    // Increment ii by 2^(M-1-l) if lsb was 1
            iaddr >>= 1;                // right shift iaddr to test next bit. Use logical
operations for speed increase
            if (!iaddr)
                break;
        }
        DFT = pDFT + ii;        // Calculate current DFT[n] from base address *pDFT and bit
reversed index ii
        DFT->Re = pX->Re;       // Update the complex array with address sorted time domain
signal x[n]
        DFT->Im = pX->Im;       // NB: Imaginary is always zero
    }

    // FFT Computation by butterfly calculation
    for (stage = 1; stage <= M; stage++) // Loop for M stages, where 2^M = N
    {
        BSep = (int)(pow(2, stage)); // Separation between butterflies = 2^stage
        P = N / BSep;             // Similar Wn's in this stage = N/Bsep
        BWidth = BSep / 2;     // Butterfly width (spacing between opposite points) =
Separation / 2.

        TwoPi_NP = TwoPi_N*P;

        for (j = 0; j < BWidth; j++) // Loop for j calculations per butterfly
        {
            if (j != 0)              // Save on calculation if R = 0, as WN^0 = (1 + j0)
            {
                //WN.Re = cos(TwoPi_NP*j)
                WN.Re = cos(TwoPi_N*P*j);     // Calculate Wn (Real and Imaginary)
                WN.Im = -sin(TwoPi_N*P*j);
            }
```

```
            for (HiIndex = j; HiIndex < N; HiIndex += BSep) // Loop for HiIndex Step BSep
butterflies per stage
            {
                pHi = pDFT + HiIndex;                   // Point to higher value
                pLo = pHi + BWidth;                     // Point to lower value (Note VC++
adjusts for spacing between elements)

                if (j != 0)                             // If exponential power is not zero...
                {
                    //CMult(pLo, &WN, &TEMP);           // Perform complex multiplication of
Lovalue with Wn
                    TEMP.Re = (pLo->Re * WN.Re) - (pLo->Im * WN.Im);
                    TEMP.Im = (pLo->Re * WN.Im) + (pLo->Im * WN.Re);

                    //CSub (pHi, &TEMP, pLo);
                    pLo->Re = pHi->Re - TEMP.Re;        // Find new Lovalue (complex
subtraction)
                    pLo->Im = pHi->Im - TEMP.Im;

                    //CAdd (pHi, &TEMP, pHi);           // Find new Hivalue (complex addition)
                    pHi->Re = (pHi->Re + TEMP.Re);
                    pHi->Im = (pHi->Im + TEMP.Im);
                }
                else
                {
                    TEMP.Re = pLo->Re;
                    TEMP.Im = pLo->Im;

                    //CSub (pHi, &TEMP, pLo);
                    pLo->Re = pHi->Re - TEMP.Re;        // Find new Lovalue (complex
subtraction)
                    pLo->Im = pHi->Im - TEMP.Im;

                    //CAdd (pHi, &TEMP, pHi);           // Find new Hivalue (complex addition)
                    pHi->Re = (pHi->Re + TEMP.Re);
                    pHi->Im = (pHi->Im + TEMP.Im);
                }
            }
        }
    }


    pLo = 0;    // Null all pointers
    pHi = 0;
    pDFT = 0;
    DFT = 0;
    pX = 0;
}
```

Радикс 2 Обратный БПФ

Из-за сильной двойственности преобразования Фурье, настройка выхода прямого преобразования может привести к обратному БПФ. Данные в частотной области могут быть преобразованы во временную область следующим образом:

1. Найти комплексное сопряжение данных частотной области путем инвертирования мнимой составляющей для всех экземпляров K.

2. Выполнение прямого БПФ на данных сопряженной частотной области.
3. Разделите каждый вывод результата этого FFT на N, чтобы указать значение истинной временной области.
4. Найти комплексное сопряжение вывода путем инвертирования мнимой составляющей данных во временной области для всех экземпляров n.

***Примечание** : данные частоты и временной области являются комплексными переменными. Обычно мнимая составляющая сигнала временной области после обратного БПФ либо равна нулю, либо игнорируется как ошибка округления. Повышение точности переменных с 32-битного поплавка до 64-битного двойного или 128-битного удвоения значительно снижает ошибки округления, возникающие в результате нескольких последовательных операций БПФ.*

Пример кода (C / C ++)

```
#include <math.h>

#define PI       3.14159265358979323846264338327795    // PI for sine/cos calculations
#define TWOPI    6.28318530717958647692528676655900    // 2*PI for sine/cos calculations
#define Deg2Rad  0.01745329251994329576923690768488612  // Degrees to Radians factor
#define Rad2Deg  57.29577951308232087679815481410517    // Radians to Degrees factor
#define log10_2  0.30102999566398119521373889472449   // Log10 of 2
#define log10_2_INV 3.32192809488736234787031942948939 // 1/Log10(2)

// complex variable structure (double precision)
struct complex
{
public:
    double  Re, Im;        // Not so complicated after all
};

void rad2InverseFFT(int N, complex *x, complex *DFT)
{
    // M is number of stages to perform. 2^M = N
    double Mx = (log10((double)N) / log10((double)2));
    int a = (int)(ceil(pow(2.0, Mx)));
    int status = 0;
    if (a != N) // Check N is a power of 2
    {
        x = 0;
        DFT = 0;
        throw "rad2InverseFFT(): N must be a power of 2 for Radix 2 Inverse FFT";
    }

    complex *pDFT = DFT;        // Reset vector for DFT pointers
    complex *pX = x;            // Reset vector for x[n] pointer
    double NN = 1 / (double)N;  // Scaling factor for the inverse FFT

    for (int i = 0; i < N; i++, DFT++)
        DFT->Im *= -1;          // Find the complex conjugate of the Frequency Spectrum

    DFT = pDFT;                 // Reset Freq Domain Pointer
    rad2FFT(N, DFT, x); // Calculate the forward FFT with variables switched (time & freq)

    int i;
    complex* x;
```

```
    for ( i = 0, x = pX; i < N; i++, x++){
        x->Re *= NN;    // Divide time domain by N for correct amplitude scaling
        x->Im *= -1;    // Change the sign of ImX
    }
}
```

Прочитайте Быстрое преобразование Фурье онлайн:
https://riptutorial.com/ru/algorithm/topic/8683/быстрое-преобразование-фурье

глава 19: Вставка Сортировка

замечания

Когда мы анализируем производительность алгоритма сортировки, мы в основном интересуемся количеством сравнения и обмена.

Средняя биржа

Пусть E $_n$ - общее среднее число обменов для сортировки массива из элемента N. E $_1$ = 0 (нам не нужен обмен для массива с одним элементом). Среднее количество обменов для сортировки массива элементов N представляет собой сумму среднего числа номеров для сортировки массива элементов N-1 со средним обменом для вставки последнего элемента в массив элементов N-1.

$$E_n = E_{n-1} + \frac{1}{n}\sum_{0\le i<n} i$$

Упрощение суммирования (арифметическая серия)

$$E_n = E_{n-1} + \frac{n-1}{2}$$

Расширяет термин

$$E_n = \frac{n-1}{2} + \frac{n-2}{2} + \ldots + \frac{1}{2}$$

Упростите суммирование снова (арифметическая серия)

$$E_n = \frac{n(n-1)}{4}$$

Среднее сравнение

Пусть C $_n$ - общее среднее число сравнения для сортировки массива из элемента N. C $_1$ = 0 (нам не нужно сравнивать ни один массив элементов). Среднее количество сравнений для сортировки массива элементов N представляет собой сумму среднего числа сравнения для сортировки массива элементов N-1 со средним сравнением для вставки последнего элемента в массив элементов N-1. Если последний элемент является самым большим элементом, нам нужно только одно сравнение, если последний элемент является вторым наименьшим элементом, нам нужно сравнение N-1. Однако, если последний элемент является наименьшим элементом, нам не нужно сравнивать N. Нам все еще нужно только сравнение N-1. Вот почему мы удаляем 1 / N в следующем уравнении.

$$C_n = C_{n-1} - \frac{1}{n} + \frac{n+1}{2}$$

Упрощение суммирования (арифметическая серия)

$$C_n = C_{n-1} - \frac{1}{n} + \frac{n+1}{2}$$

Развернуть термин

$$C_n = (\frac{n+1}{2} + ... + \frac{3}{2}) - (\frac{1}{n} + ... + \frac{1}{2})$$

Упростите суммирование снова (арифметические ряды и гармоническое число)

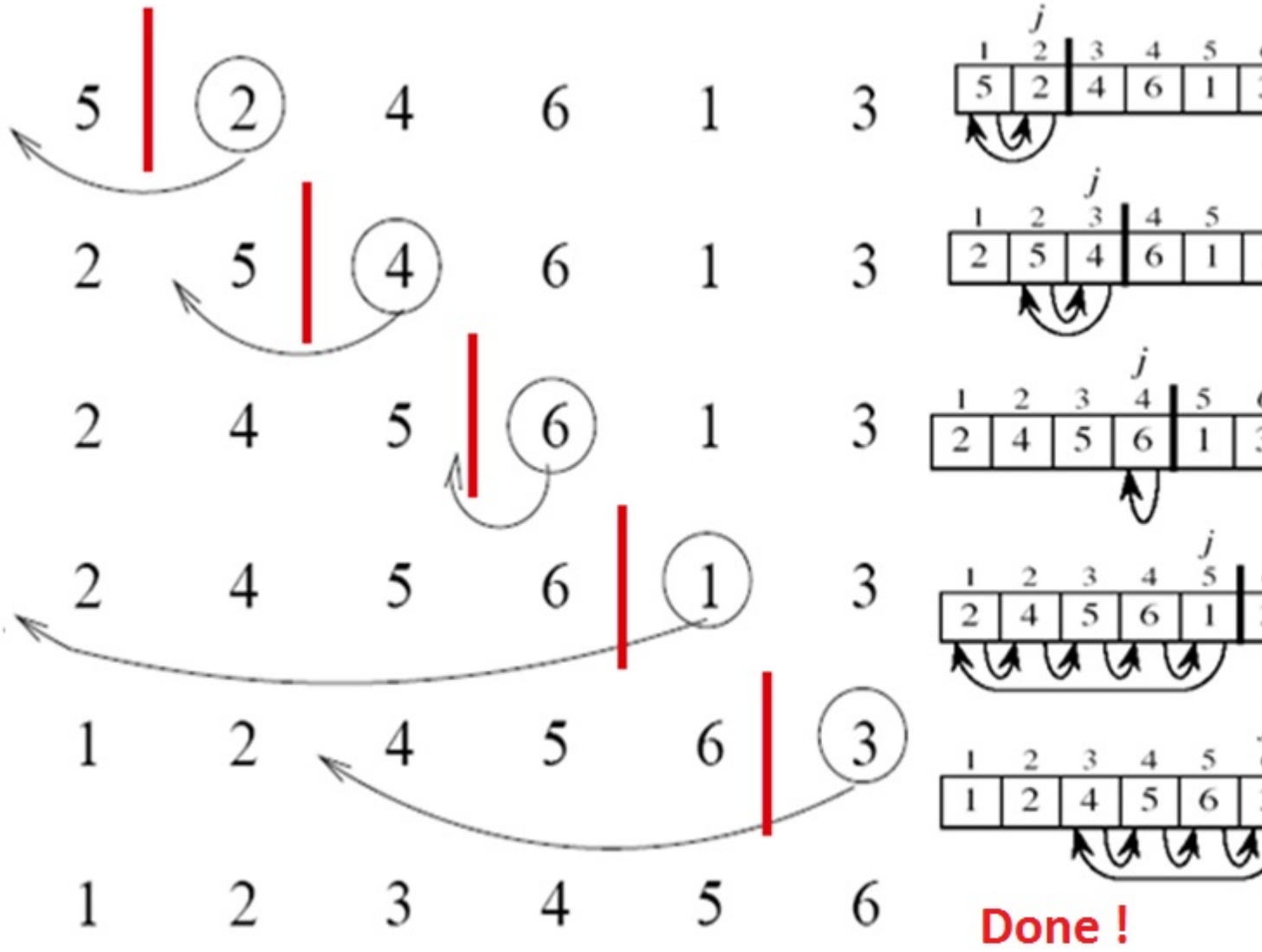
$$C_n = \frac{(n+1)(n+2)}{4} - H_n - 0.5$$

Examples

Основы алгоритма

Сортировка вставки - очень простой, стабильный алгоритм сортировки на месте. Он хорошо работает на небольших последовательностях, но он намного менее эффективен для больших списков. На каждом шаге алгоритмы рассматривают i-й элемент данной последовательности, перемещая ее влево, пока она не окажется в правильном положении.

Графическая иллюстрация



ПСЕВДОКОД

```
for j = 1 to length(A)
    n = A[j]
    i = j - 1
    while j > 0 and A[i] > n
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = n
```

пример

Рассмотрим следующий список целых чисел:

```
[5, 2, 4, 6, 1, 3]
```

Алгоритм выполнит следующие шаги:

1. `[5, 2, 4, 6, 1, 3]`
2. `[2, 5, 4, 6, 1, 3]`
3. `[2, 4, 5, 6, 1, 3]`

4. `[2, 4, 5, 6, 1, 3]`
5. `[1, 2, 4, 5, 6, 3]`
6. `[1, 2, 3, 4, 5, 6]`

Реализация C

```
public class InsertionSort
{
    public static void SortInsertion(int[] input, int n)
    {
        for (int i = 0; i < n; i++)
        {
            int x = input[i];
            int j = i - 1;
            while (j >= 0 && input[j] > x)
            {
                input[j + 1] = input[j];
                j = j - 1;
            }
            input[j + 1] = x;
        }
    }

    public static int[] Main(int[] input)
    {
        SortInsertion(input, input.Length);
        return input;
    }
}
```

Вспомогательное пространство: `O(1)`
Сложность времени: `O(n)`

Реализация Haskell

```
insertSort :: Ord a => [a] -> [a]
insertSort [] = []
insertSort (x:xs) = insert x (insertSort xs)

insert :: Ord a => a-> [a] -> [a]
insert n [] = [n]
insert n (x:xs) | n <= x    = (n:x:xs)
                | otherwise = x:insert n xs
```

Прочитайте Вставка Сортировка онлайн: https://riptutorial.com/ru/algorithm/topic/5738/вставка-сортировка

глава 20: Выбор Сортировка

Examples

Выбор Сортировка Основная информация

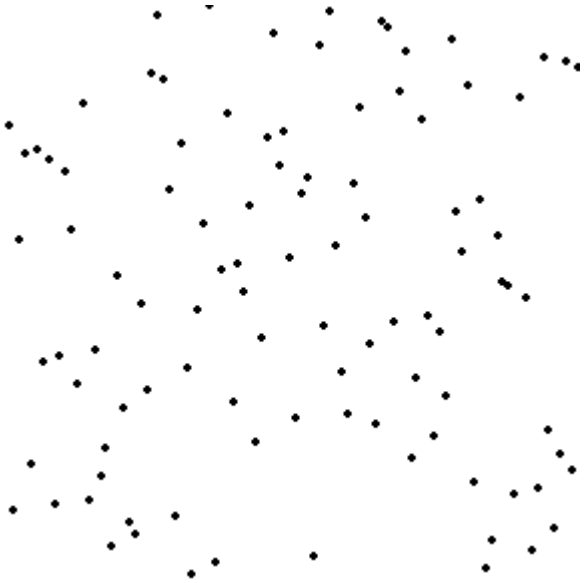
Сортировка сортировки - это алгоритм сортировки, в частности, сортировка на месте. Он имеет временную сложность O (n2), что делает его неэффективным в больших списках и, как правило, хуже, чем аналогичная сортировка вставки. Сортировка выбора отличается своей простотой и имеет преимущества производительности по сравнению с более сложными алгоритмами в определенных ситуациях, особенно там, где ограничена вспомогательная память.

Алгоритм делит входной список на две части: подсписку отсортированных элементов, которая создается слева направо в начале (слева) списка, и подсписку оставшихся элементов, которые будут отсортированы, которые занимают остальную часть список. Первоначально отсортированный подсписок пуст, а несортированный подсписок - это весь список ввода. Алгоритм продолжается путем нахождения наименьшего (или самого большого, в зависимости от порядка сортировки) элемента в несортированном подсписке, обменивая (свопинг) его с самым левым несортированным элементом (помещая его в отсортированный порядок) и перемещая границы субблица один элемент вправо ,

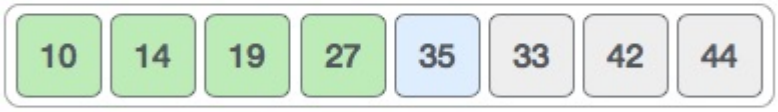
Псевдокод для сортировки сортировки:

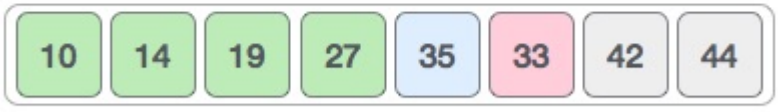
```
function select(list[1..n], k)
 for i from 1 to k
     minIndex = i
     minValue = list[i]
     for j from i+1 to n
         if list[j] < minValue
             minIndex = j
             minValue = list[j]
     swap list[i] and list[minIndex]
 return list[k]
```

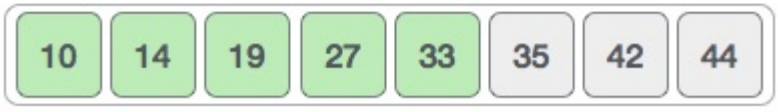
Визуализация сортировки выбора:

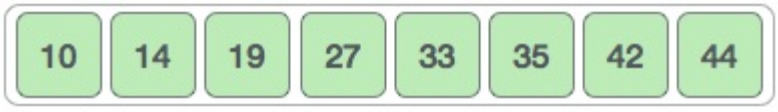


Пример сортировки выбора:









Вспомогательное пространство: `O(n)`
Сложность времени: `O(n^2)`

Реализация сортировки в C

Я использовал язык C # для реализации алгоритма сортировки выборок.

```
public class SelectionSort
{
    private static void SortSelection(int[] input, int n)
    {
        for (int i = 0; i < n - 1; i++)
        {
            var minId = i;
            int j;
            for (j = i + 1; j < n; j++)
            {
                if (input[j] < input[minId]) minId = j;
            }
            var temp = input[minId];
            input[minId] = input[i];
            input[i] = temp;
        }
    }

    public static int[] Main(int[] input)
    {
        SortSelection(input, input.Length);
        return input;
    }
}
```

Реализация эликсира

```
defmodule Selection do

  def sort(list) when is_list(list) do
    do_selection(list, [])
  end

  def do_selection([head|[]], acc) do
    acc ++ [head]
  end

  def do_selection(list, acc) do
    min = min(list)
    do_selection(:lists.delete(min, list), acc ++ [min])
  end

  defp min([first|[second|[]]]) do
    smaller(first, second)
  end

  defp min([first|[second|tail]]) do
    min([smaller(first, second)|tail])
  end
```

```
  defp smaller(e1, e2) do
    if e1 <= e2 do
      e1
    else
      e2
    end
  end
end

Selection.sort([100,4,10,6,9,3])
|> IO.inspect
```

Прочитайте Выбор Сортировка онлайн: https://riptutorial.com/ru/algorithm/topic/7473/выбор-сортировка

глава 21: Вычисление матрицы

Examples

Выравнивание матрицы для решения проблем

Найти f (n): n- й номер Фибоначчи. Проблема довольно проста, когда **n** относительно мало. Мы можем использовать простую рекурсию, `f(n) = f(n-1) + f(n-2)` , или мы можем использовать метод динамического программирования, чтобы избежать повторения одной и той же функции снова и снова. Но что вы будете делать, если проблема говорит: **учитывая 0 <n <10⁹, найдите f (n) mod 999983?** Динамическое программирование завершится неудачно, и как мы справимся с этой проблемой?

Сначала давайте посмотрим, как матричное возведение в степень может помочь представить рекурсивное отношение.

Предпосылки:

- Учитывая две матрицы, вы знаете, как найти их продукт. Далее, учитывая матрицу произведения двух матриц и одну из них, знаем, как найти другую матрицу.
- Учитывая матрицу размера **d X d** , знаете, как найти ее n- ю степень в **O (d 3 log (n))** .

Шаблоны:

Сначала нам нужно рекурсивное соотношение, и мы хотим найти матрицу **M,** которая может привести нас к желаемому состоянию из набора уже известных состояний. Предположим, что мы знаем **k** состояний данного рекуррентного отношения и хотим найти **(k + 1) -e** состояние. Пусть **M** - матрица **k X k** , и мы строим матрицу **A: [k X 1]** из известных состояний рекуррентного отношения, теперь мы хотим получить матрицу **B: [k X 1],** которая будет представлять множество следующих состояний, то есть **MXA = B,** как показано ниже:

```
      |  f(n)   |      | f(n+1) |
      | f(n-1)  |      |  f(n)  |
  M X | f(n-2)  |  =   | f(n-1) |
      | ......  |      | ...... |
      | f(n-k)  |      |f(n-k+1)|
```

Итак, если мы сможем спроектировать **M** соответственно, наша работа будет выполнена! Затем матрица будет использоваться для представления рекуррентного отношения.

Тип 1:
Начнем с простейшего, `f(n) = f(n-1) + f(n-2)`
Получаем, `f(n+1) = f(n) + f(n-1)` .

Предположим, что мы знаем `f(n)` и `f(n-1)` ; Мы хотим узнать `f(n+1)` .
Из изложенной выше ситуации матрица **A** и матрица **B** могут быть сформированы, как показано ниже:

```
 Matrix A          Matrix B

|  f(n)  |        | f(n+1) |
| f(n-1) |        |  f(n)  |
```

[Примечание: матрица **A** будет всегда сконструирована таким образом, чтобы каждое состояние, от которого зависит `f(n+1)` , будет присутствовать]
Теперь нам нужно создать матрицу **2X2 M** такую, что она удовлетворяет **MXA = B,** как указано выше.
Первым элементом **B** является `f(n+1)` который фактически является `f(n) + f(n-1)` . Чтобы получить это, из матрицы **A** нам понадобится **1 X f (n)** и **1 X f (n-1)** . Таким образом, первая строка **M** будет **[1 1]** .

```
| 1   1 |  X  |  f(n)  |  =  | f(n+1) |
| ----- |     | f(n-1) |     | ------ |
```

[Примечание: ----- означает, что нас это не касается.]
Аналогично, второй элемент **B** является `f(n)` который можно получить, просто взяв **1 X f (n)** из **A** , поэтому вторая строка **M** равна [1 0].

```
| ----- |  X  |  f(n)  |  =  | ------ |
| 1   0 |     | f(n-1) |     |  f(n)  |
```

Тогда мы получим желаемые **2 X 2** матрицы **M.**

```
| 1   1 |  X  |  f(n)  |  =  | f(n+1) |
| 1   0 |     | f(n-1) |     |  f(n)  |
```

Эти матрицы просто выведены с использованием матричного умножения.

Тип 2:

Сделаем его несколько сложным: найдем `f(n) = a X f(n-1) + b X f(n-2)` , где **a** и **b** - постоянные.
Это говорит нам, что `f(n+1) = a X f(n) + b X f(n-1)` .
До сих пор должно быть ясно, что размерность матриц будет равна числу зависимостей, то есть в этом конкретном примере, снова 2. Итак, для **A** и **B** мы можем построить две матрицы размера **2 X 1** :

```
Matrix A            Matrix B
|  f(n)  |         | f(n+1) |
| f(n-1) |         |  f(n)  |
```

Теперь для `f(n+1) = a X f(n) + b X f(n-1)` нам нужно [a, b] в первой строке объектной матрицы **M.** А для второго элемента в **B** , т. `f(n)` мы уже имеем это в матрице **A** , поэтому просто возьмем то, что приводит, вторая строка матрицы M к [1 0]. На этот раз мы получим:

```
| a   b |  X  |  f(n)  |  =  | f(n+1) |
| 1   0 |     | f(n-1) |     |  f(n)  |
```

Довольно просто, а?

Тип 3:

Если вы дожили до этого этапа, вы стали намного старше, теперь давайте рассмотрим несколько сложное соотношение: find `f(n) = a X f(n-1) + c X f(n-3)` ?
По электронной почте Ой! Несколько минут назад все, что мы видели, были смежными состояниями, но здесь состояние **f (n-2)** отсутствует. Сейчас?

На самом деле это уже не проблема, мы можем преобразовать соотношение следующим образом: `f(n) = a X f(n-1) + 0 X f(n-2) + c X f(n-3)` , выводя `f(n+1) = a X f(n) + 0 X f(n-1) + c X f(n-2)` . Теперь мы видим, что это на самом деле форма, описанная в типе 2. Таким образом, объективная матрица **M** будет **3 X 3** , а элементы:

```
| a 0 c |     |  f(n)  |     | f(n+1) |
| 1 0 0 |  X  | f(n-1) |  =  |  f(n)  |
| 0 1 0 |     | f(n-2) |     | f(n-1) |
```

Они вычисляются так же, как тип 2, если вам сложно, попробуйте его на ручке и бумаге.

Тип 4:

Теперь жизнь становится сложной, и г-н, Проблема теперь просит вас найти `f(n) = f(n-1) + f(n-2) + c` где **c** - любая константа.
Теперь это новый и все, что мы видели в прошлом, после умножения каждое состояние в **A** переходит в следующее состояние в **B.**

```
f(n) = f(n-1) + f(n-2) + c
f(n+1) = f(n) + f(n-1) + c
f(n+2) = f(n+1) + f(n) + c
.................. so on
```

Таким образом, обычно мы не можем получить это через предыдущий способ, но как насчет добавления **c** как состояния:

```
     |  f(n)  |    | f(n+1) |
  M X | f(n-1) | = |  f(n)  |
     |    c   |    |    c   |
```

Теперь его не так сложно создать **M.** Вот как это делается, но не забудьте проверить:

```
  | 1 1 1 |     |  f(n)  |     | f(n+1) |
  | 1 0 0 |  X  | f(n-1) |  =  |  f(n)  |
  | 0 0 1 |     |    c   |     |    c   |
```

Тип 5:

Положим все: find `f(n) = a X f(n-1) + c X f(n-3) + d X f(n-4) + e` . Давайте оставим это как упражнение для вас. Сначала попробуйте выяснить состояния и матрицу **M.** И проверьте, соответствует ли это вашему решению. Также найдите матрицы **A** и **B.**

```
| a 0 c d 1 |
| 1 0 0 0 0 |
| 0 1 0 0 0 |
| 0 0 1 0 0 |
| 0 0 0 0 1 |
```

Тип 6:

Иногда повторение дается так:

```
f(n) = f(n-1)   -> if n is odd
f(n) = f(n-2)   -> if n is even
```

Короче:

```
f(n) = (n&1) X f(n-1) + (!(n&1)) X f(n-2)
```

Здесь мы можем разбить функции на основе нечетного четного и сохранить для них две разные матрицы и рассчитать их отдельно.

Тип 7:

Чувствовать себя слишком уверенно? Повезло тебе. Иногда нам может потребоваться поддерживать более одного повторения, где они заинтересованы. Например, пусть повторение re; atopm будет:

```
g(n) = 2g(n-1) + 2g(n-2) + f(n)
```

Здесь рекурсия *g (n)* зависит от `f(n)` и это можно вычислить в той же матрице, но с увеличенными размерами. Из них сначала создадим матрицы **A** и **B.**

```
 Matrix A            Matrix B
|  g(n)  |          | g(n+1) |
| g(n-1) |          |  g(n)  |
| f(n+1) |          | f(n+2) |
|  f(n)  |          | f(n+1) |
```

Здесь `g(n+1) = 2g(n-1) + f(n+1) and f(n+2) = 2f(n+1) + 2f(n)` . Теперь, используя описанные

выше процессы, мы можем найти объективную матрицу **M** :

```
| 2 2 1 0 |
| 1 0 0 0 |
| 0 0 2 2 |
| 0 0 1 0 |
```

Итак, это основные категории рекуррентных отношений, которые используются для решения этой простой методики.

Прочитайте Вычисление матрицы онлайн: https://riptutorial.com/ru/algorithm/topic/7290/вычисление-матрицы

глава 22: Глубина первого поиска

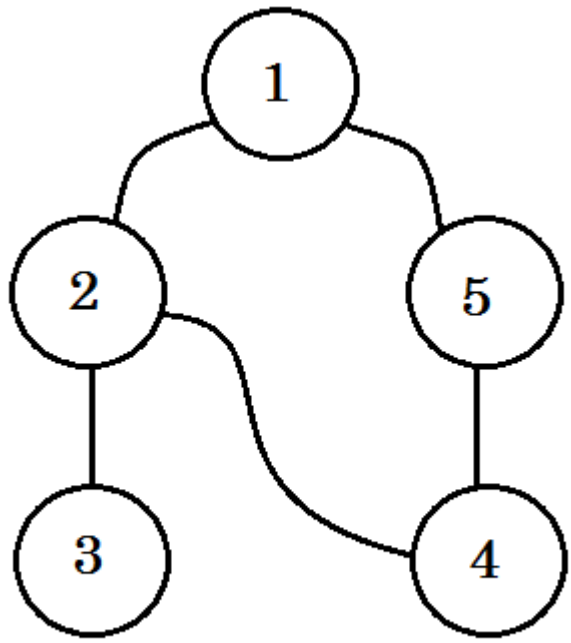
Examples

Введение в поиск по глубине

Поиск по глубине - это алгоритм для перемещения или поиска структур данных дерева или графика. Один начинается с корня и исследует, насколько это возможно, вдоль каждой ветви перед отступлением. Версия поиска по глубине была исследована в 19-м веке французским математиком Чарльзом Пьером Тремо в качестве стратегии решения лабиринтов.

Поиск по глубине - это систематический способ найти все вершины, достижимые из исходной вершины. Как и поиск по ширине, DFS пересекает связанный компонент заданного графа и определяет остовное дерево. Основная идея поиска по глубине - это методическое изучение каждого края. При необходимости мы начинаем с разных вершин. Как только мы обнаруживаем вершину, DFS начинает изучать ее (в отличие от BFS, которая помещает вершину в очередь, чтобы потом исследовать ее).

Давайте посмотрим на пример. Мы проведем этот график:

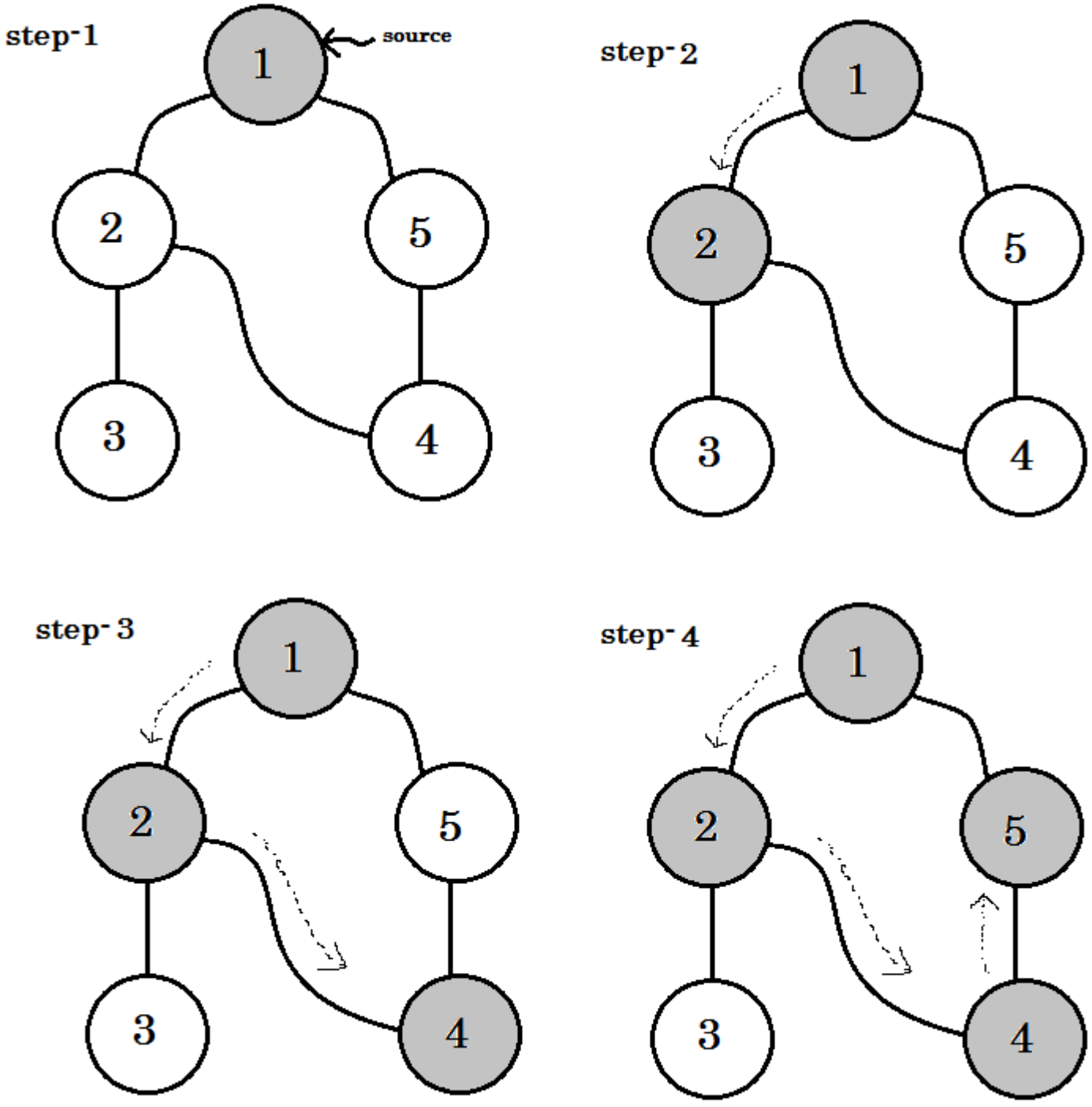


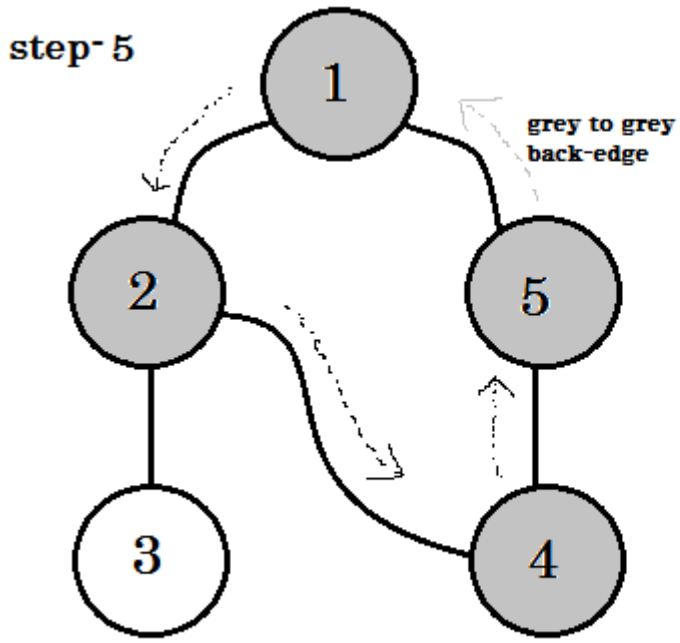
Мы пройдем по графику, следуя этим правилам:

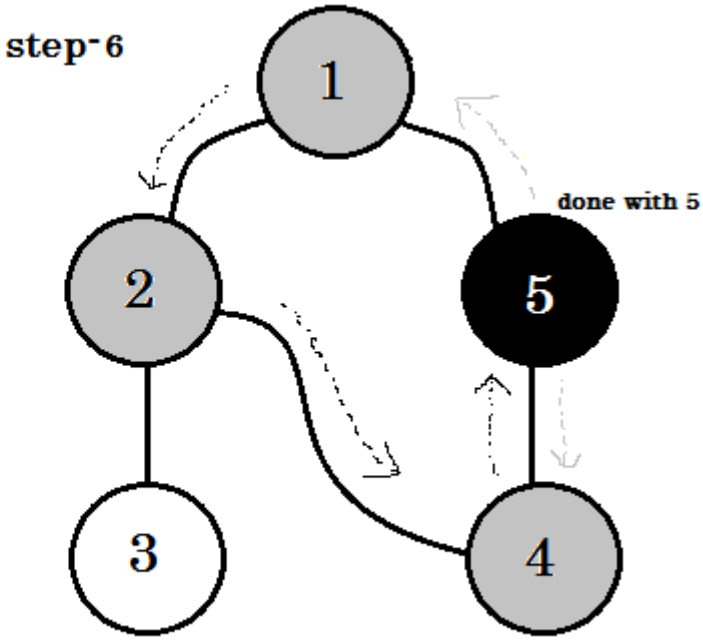
- Мы начнем с источника.
- Ни один узел не будет посещаться дважды.
- Узлы, которые мы еще не посетили, будут окрашены в белый цвет.

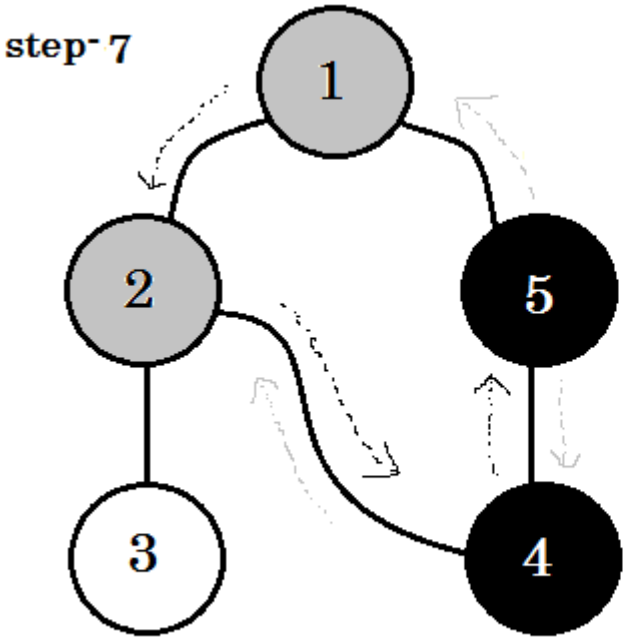
- Узел, который мы посетили, но не посетил все его дочерние узлы, будет окрашен в серый цвет.
- Полностью пройденные узлы будут окрашены в черный цвет.

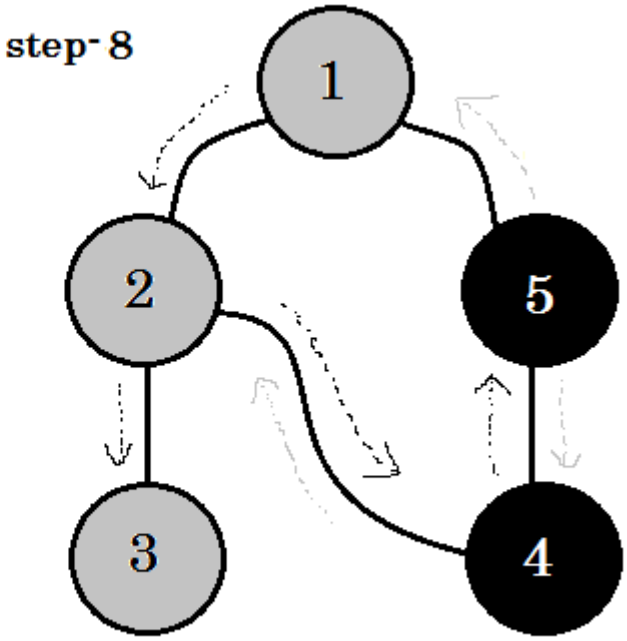
Давайте посмотрим на это шаг за шагом:

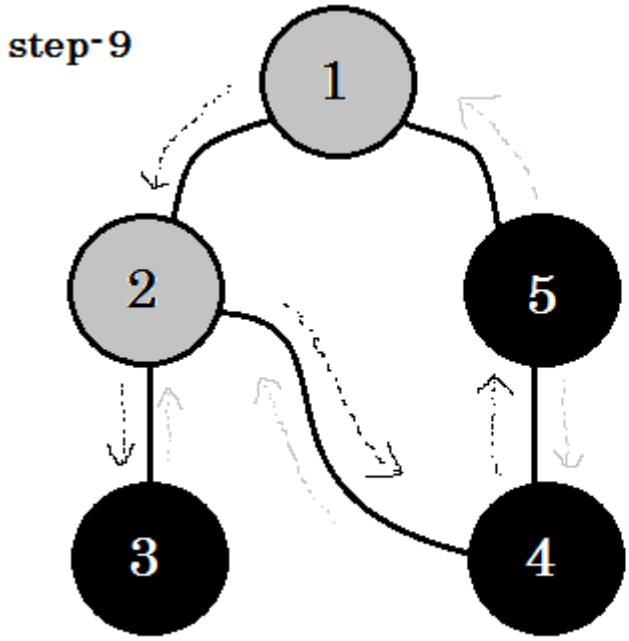


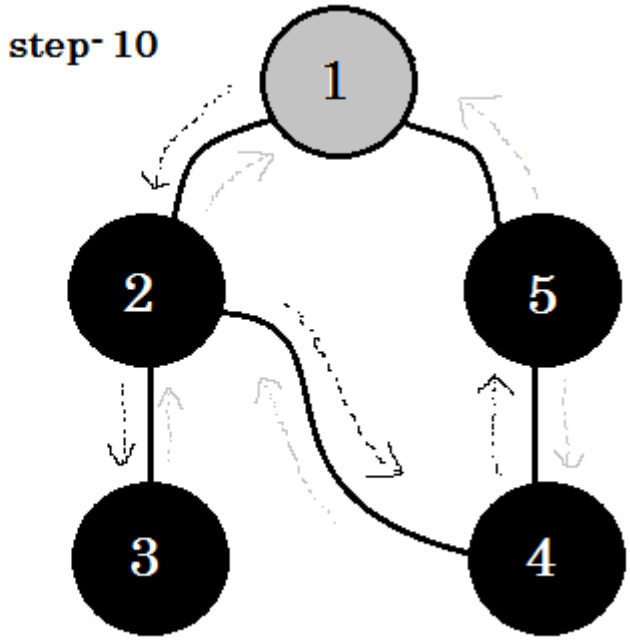

step- 5
1
2
5
3
4
grey to grey
back-edge

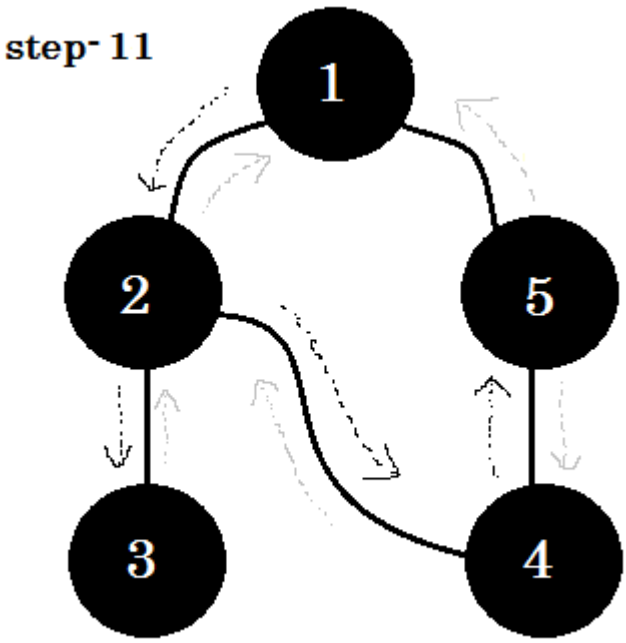

step- 6
1
2
5
3
4
done with 5


step- 7
1
2
5
3
4


step- 8
1
2
5
3
4







Мы можем увидеть одно важное ключевое слово. Это **поддержка** . Ты можешь видеть. **5-1** называется backedge. Это связано с тем, что мы еще не сделали это с **узлом-1** , поэтому переход от другого узла к **узлу-1** означает, что на графике есть цикл. В DFS, если мы можем перейти от одного серого узла к другому, мы можем быть уверены, что график имеет цикл. Это один из способов обнаружения цикла на графике. В зависимости от **исходного** узла и порядка узлов, которые мы посещаем, мы можем найти любое ребро в цикле в качестве **поддержки** . Например: если мы сначала отправимся в **5** из **1** , мы бы обнаружили **2-1 в** качестве поддержки.

Край, который мы берем, чтобы перейти от серого узла к белому узлу, называется **краем дерева** . Если мы сохраним только **ребро дерева** и удалим другие, мы получим **дерево**

DFS .

В неориентированном графике, если мы можем посетить уже посещенный узел, это должно быть **поддержкой** . Но для ориентированных графов мы должны проверить цвета. *Если и только если мы можем перейти от одного серого узла к другому серому узлу, который называется поддержкой* .

В DFS мы также можем сохранять временные метки для каждого узла, которые могут использоваться многими способами (например: Топологическая сортировка).

1. Когда узел **v** изменяется с белого на серый, время записывается в **d [v]** .
2. Когда узел **v** изменяется с серого на черный, время записывается в **f [v]** .

Здесь **d []** означает *время обнаружения,* а **f []** означает *время окончания* . Наш пес-код будет выглядеть так:

```
Procedure DFS(G):
for each node u in V[G]
    color[u] := white
    parent[u] := NULL
end for
time := 0
for each node u in V[G]
    if color[u] == white
        DFS-Visit(u)
    end if
end for

Procedure DFS-Visit(u):
color[u] := gray
time := time + 1
d[u] := time
for each node v adjacent to u
    if color[v] == white
        parent[v] := u
        DFS-Visit(v)
    end if
end for
color[u] := black
time := time + 1
f[u] := time
```

Сложность:

Каждый узел и ребра посещаются один раз. Таким образом, сложность DFS равна **O (V + E)** , где **V** обозначает количество узлов, а **E** обозначает количество ребер.

Приложения глубины Первый поиск:

- Поиск кратчайшего пути пары в неориентированном графе.
- Обнаружение цикла на графике.
- Найти путь.

- Топологическая сортировка.
- Тестирование, если граф двудольный.
- Поиск сильно подключенного компонента.
- Решение головоломок с одним решением.

Прочитайте Глубина первого поиска онлайн: https://riptutorial.com/ru/algorithm/topic/7247/глубина-первого-поиска

глава 23: график

Вступление

Граф - это набор точек и линий, соединяющих некоторые (возможно, пустые) их подмножества. Точки графа называются вершинами графа, «узлами» или просто «точками». Точно так же линии, соединяющие вершины графа, называются графами графа, «дугами» или «строками».

Граф G можно определить как пару (V, E), где V - множество вершин, а E - множество ребер между вершинами $E \subseteq \{(u, v) \mid u, v \in V\}$.

замечания

Графики представляют собой математическую структуру, которая представляет собой набор моделей объектов, которые могут или не могут быть связаны с элементами из наборов ребер или ссылок.

Граф можно описать двумя различными наборами математических объектов:

- Набор **вершин** .
- Набор **ребер** , соединяющих пары вершин.

Графики могут быть либо направленными, либо неориентированными.

- **Направленные графики** содержат ребра, которые «соединяются» только одним способом.
- **Непрямые графы** содержат только ребра, которые автоматически соединяют две вершины вместе в обоих направлениях.

Examples

Топологическая сортировка

Топологическое упорядочение или топологическая сортировка упорядочивает вершины в ориентированном ациклическом графе на линии, т. Е. В списке, так что все направленные ребра идут слева направо. Такое упорядочение не может существовать, если граф содержит направленный цикл, потому что нет способа, которым вы можете продолжать идти прямо по линии и все еще возвращаться туда, откуда вы начали.

Формально в графе `G = (V, E)` линейное упорядочение всех его вершин таково, что если `G` содержит ребро `(u, v) ∈ E` от вершины `u` до вершины `v` то `u` предшествует `v` в упорядочении.

Важно отметить, что каждая группа DAG имеет *по крайней мере одну* топологическую сортировку.

Известны алгоритмы построения топологического упорядочения любой DAG в линейном времени, например:

1. Вызовите `depth_first_search(G)` чтобы вычислить время окончания `vf` для каждой вершины `v`
2. Когда каждая вершина закончена, вставьте ее в переднюю часть связанного списка
3. связанный список вершин, так как теперь он отсортирован.

Топологическая сортировка может быть выполнена в `(V + E)` время, поскольку алгоритм поиска , глубина-сначала берет `(V + E)` время и он принимает `(1)` (постоянное время) для вставки каждого из `|V|` вершины в списке связанного списка.

Во многих приложениях используются ориентированные ациклические графики, указывающие приоритеты среди событий. Мы используем топологическую сортировку, чтобы получить упорядочение для обработки каждой вершины перед любым ее преемником.

Вершины в графе могут представлять задачи, которые должны выполняться, и ребра могут представлять собой ограничения, которые одна задача должна выполнять перед другой; топологическое упорядочение является допустимой последовательностью для выполнения задач, заданных задачами, описанными в `V`

Экземпляр проблемы и ее решение

Пусть вершина `v` описывает `Task(hours_to_complete: int)` , то есть `Task(4)` описывает `Task` которая занимает `4` часа, а ребро `e` описывает время `Cooldown(hours: int)` , так что время `Cooldown(3)` описывает продолжительность чтобы остыть после выполнения заданной задачи.

Пусть наш граф называется `dag` (так как он является ациклическим графом) и пусть он содержит 5 вершин:

```
A <- dag.add_vertex(Task(4));
B <- dag.add_vertex(Task(5));
C <- dag.add_vertex(Task(3));
D <- dag.add_vertex(Task(2));
E <- dag.add_vertex(Task(7));
```

где мы соединяем вершины с направленными ребрами, так что граф ацикличен,

```
// A ---> C ----+
// |      |     |
```

```
// v      v     v
// B ---> D --> E
dag.add_edge(A, B, Cooldown(2));
dag.add_edge(A, C, Cooldown(2));
dag.add_edge(B, D, Cooldown(1));
dag.add_edge(C, D, Cooldown(1));
dag.add_edge(C, E, Cooldown(1));
dag.add_edge(D, E, Cooldown(3));
```

то существует три возможных топологических порядка между `A` и `E` ,

1. `A -> B -> D -> E`
2. `A -> C -> D -> E`
3. `A -> C -> E`

Алгоритм Торапа

Алгоритм Торапа для кратчайшего пути одного источника для неориентированного графа имеет временную сложность O (m), ниже Dijkstra.

Основные идеи заключаются в следующем. (Извините, я еще не пытался его реализовать, поэтому я мог бы пропустить некоторые мелкие детали. И исходная бумага была оплачена, поэтому я попытался восстановить ее из других источников, ссылающихся на нее. Удалите этот комментарий, если сможете проверить.)

- Есть способы найти остовное дерево в O (m) (не описано здесь). Вам нужно «развить» остовное дерево от самого короткого края до самого длинного, и это будет лес с несколькими компонентами, которые будут соединены до полного роста.
- Выберите целое число b (b> = 2) и рассмотрите только покрывающие леса с ограничением длины b ^ k. Объедините компоненты, которые являются точно такими же, но с разными k, и назовите минимум k уровнем компонента. Затем логически вставляйте компоненты в дерево. u является родительским элементом v, если u - наименьший компонент, отличный от v, который полностью содержит v. Корень - это весь граф, а листья - единственные вершины в исходном графе (с уровнем отрицательной бесконечности). Дерево все еще имеет только O (n) узлы.
- Поддерживайте расстояние каждого компонента до источника (как в алгоритме Дейкстры). Расстояние компонента с более чем одной вершиной - это минимальное расстояние его нерасширенных детей. Установите расстояние от исходной вершины до 0 и соответствующим образом обновите предков.
- Рассмотрим расстояния в базе b. При первом посещении узла на уровне k поместите его детей в ведра, разделяемые всеми узлами уровня k (как в сортировке ведра, заменяя кучу в алгоритме Дейкстры) цифрой k и выше ее расстояния. Каждый раз, посещая узел, учитывайте только его первые b ведра, посещайте и удаляйте каждый из них, обновляйте расстояние текущего узла и переставляйте текущий узел на свой собственный родитель с использованием нового расстояния и дожидайтесь следующего посещения для следующего ковши.
- Когда лист посещается, текущее расстояние является конечным расстоянием от

вершины. Разверните все ребра из него в исходном графике и соответствующим образом обновите расстояния.
- Несколько раз перейдите на корневой узел (целый график) до достижения цели.

Он основан на том факте, что не существует края длиной менее l между двумя компонентами связующего леса с ограничением длины l, поэтому, начиная с расстояния x, вы можете сосредоточиться только на одном подключенном компоненте, пока не достигнете расстояние x + 1. Вы увидите некоторые вершины до того, как все вершины с более коротким расстоянием будут посещены, но это не имеет значения, поскольку известно, что здесь не будет более короткого пути от этих вершин. Другие части работают как сортировка ковша / MSD radix, и, конечно же, для этого требуется O (m) spanning tree.

Обнаружение цикла в ориентированном графе с использованием метода «Глубина первого обхода»

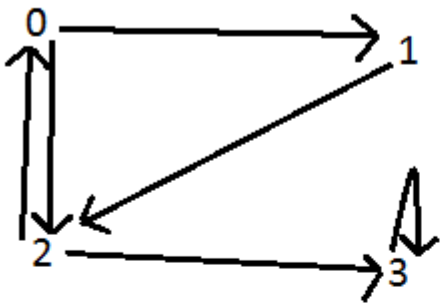
Цикл в ориентированном графе существует, если во время DFS есть задний край. Задний край - это край от узла к себе или к одному из предков в дереве DFS. Для несвязанного графа мы получаем лес DFS, поэтому вам нужно перебирать все вершины графа, чтобы найти непересекающиеся деревья DFS.

Реализация C ++:

```
    #include <iostream>
     #include <list>

    using namespace std;

    #define NUM_V   4

    bool helper(list<int> *graph, int u, bool* visited, bool* recStack)
    {
        visited[u]=true;
        recStack[u]=true;
        list<int>::iterator i;
        for(i = graph[u].begin();i!=graph[u].end();++i)
        {
            if(recStack[*i]) //if vertice v is found in recursion stack of this DFS traversal
                return true;
            else if(*i==u) //if there's an edge from the vertex to itself
                return true;
            else if(!visited[*i])
            {   if(helper(graph, *i, visited, recStack))
                   return true;
            }
        }
        recStack[u]=false;
        return false;
    }
    /*
    /The wrapper function calls helper function on each vertices which have not been visited.
Helper function returns true if it detects a back edge in the subgraph(tree) or false.
*/
    bool isCyclic(list<int> *graph, int V)
```

```
    {
      bool visited[V];  //array to track vertices already visited
      bool recStack[V]; //array to track vertices in recursion stack of the traversal.

      for(int i = 0;i<V;i++)
       visited[i]=false, recStack[i]=false;  //initialize all vertices as not visited and not
recursed

      for(int u = 0; u < V; u++) //Iteratively checks if every vertices have been visited
      {   if(visited[u]==false)
          {  if(helper(graph, u, visited, recStack)) //checks if the DFS tree from the vertex
contains a cycle
                return true;
          }
      }
       return false;
    }
    /*
    Driver function
    */
    int main()
    {
        list<int>* graph = new list<int>[NUM_V];
        graph[0].push_back(1);
        graph[0].push_back(2);
        graph[1].push_back(2);
        graph[2].push_back(0);
        graph[2].push_back(3);
        graph[3].push_back(3);
        bool res = isCyclic(graph, NUM_V);
        cout<<res<<endl;
    }
```

Результат: Как показано ниже, на графике есть три задних края. Один между вершинами 0 и 2; между вершинами 0, 1 и 2; и вершина 3. Сложность времени поиска - O (V + E), где V - число вершин, E - число ребер.



Введение в теорию графов

Теория графов - это исследование графов, которые являются математическими структурами, используемыми для моделирования парных отношений между объектами.

Знаете ли вы, что почти все проблемы планеты Земля могут быть преобразованы в проблемы дорог и городов и решены? Теория графа была изобретена много лет назад, еще до изобретения компьютера. Леонард Эйлер написал статью о семи мостах Кенигсберга, которая рассматривается как первая статья теории графов. С тех пор люди поняли, что если мы сможем преобразовать любую проблему в эту проблему City-Road, мы сможем легко ее решить с помощью теории графов.

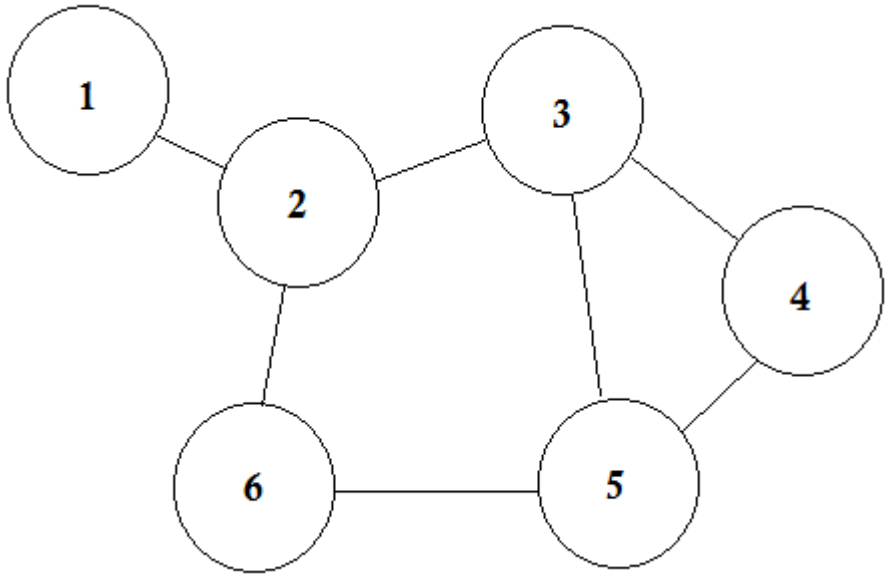
Теория диаграмм имеет много приложений. Одним из наиболее распространенных приложений является поиск кратчайшего расстояния между одним городом в другой. Мы

все знаем, что для доступа к вашему компьютеру на этой веб-странице нужно было перемещать много маршрутизаторов с сервера. Теория графов помогает ему узнать маршрутизаторы, которые необходимо пересечь. Во время войны, на какую улицу нужно бомбардировать, чтобы отключить столицу от других, это также можно обнаружить с помощью теории графов.

Давайте сначала изучим некоторые базовые определения теории графов.

График:

Скажем, у нас 6 городов. Мы отмечаем их как 1, 2, 3, 4, 5, 6. Теперь мы соединяем города, которые имеют дороги между собой.



Это простой график, где показаны некоторые города с дорогими, которые соединяют их. В теории графов мы называем каждый из этих городов **узлами** или **вершинами,** а дороги называются **Edge.** График - это просто соединение этих узлов и ребер.

Узел может представлять много вещей. На некоторых графиках узлы представляют города, некоторые представляют аэропорты, а некоторые представляют собой квадрат в шахматной доске. **Edge** представляет собой отношение между каждым узлом. Это отношение может быть временем перехода из одного аэропорта в другой, движения рыцаря с одного квадрата на все остальные квадраты и т. Д.

Путь рыцаря в шахматной доске

Простыми словами, **узел** представляет любой объект, а **Edge** представляет связь между двумя объектами.

Прилегающий узел:

Если узел **A** разделяет ребро с узлом **B** , то **B** считается смежным с **A.** Другими словами, если два узла напрямую связаны, они называются соседними узлами. Один узел может иметь несколько соседних узлов.

График Directed и Undirected:

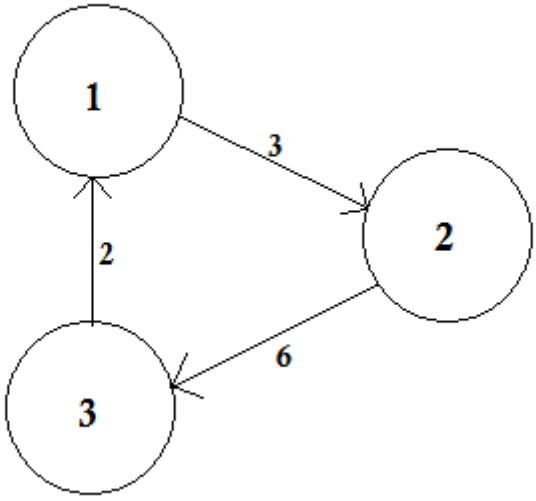
В ориентированных графах ребра имеют знаки направления с одной стороны, что означает, что ребра являются *однонаправленными* . С другой стороны, края неориентированных графов имеют знаки направления с обеих сторон, что означает, что они *двунаправленные* . Обычно неориентированные графы представлены без знаков по обе стороны от ребер.

Предположим, что вечеринка продолжается. Люди в партии представлены узлами, и между двумя людьми есть край, если они пожимают друг другу руки. Тогда этот граф неориентированный , потому что любой человек **A** пожать друг другу руки с человеком **B**

тогда и только тогда , когда **B** также рукопожатием с. Напротив, если ребра от человека **A** другому человеку **B** соответствуют **A** , любующему **B** , тогда этот график направлен, потому что восхищение не обязательно взаимно. Первый тип графа называется *неориентированным графом,* а ребра называются *неориентированными ребрами,* а последний тип графа называется *ориентированным графом,* а ребра называются *направленными ребрами.*

Взвешенный и невзвешенный график:

Весовой график представляет собой график, в котором каждому ребру присваивается число (вес). Такие веса могут представлять собой, например, затраты, длины или

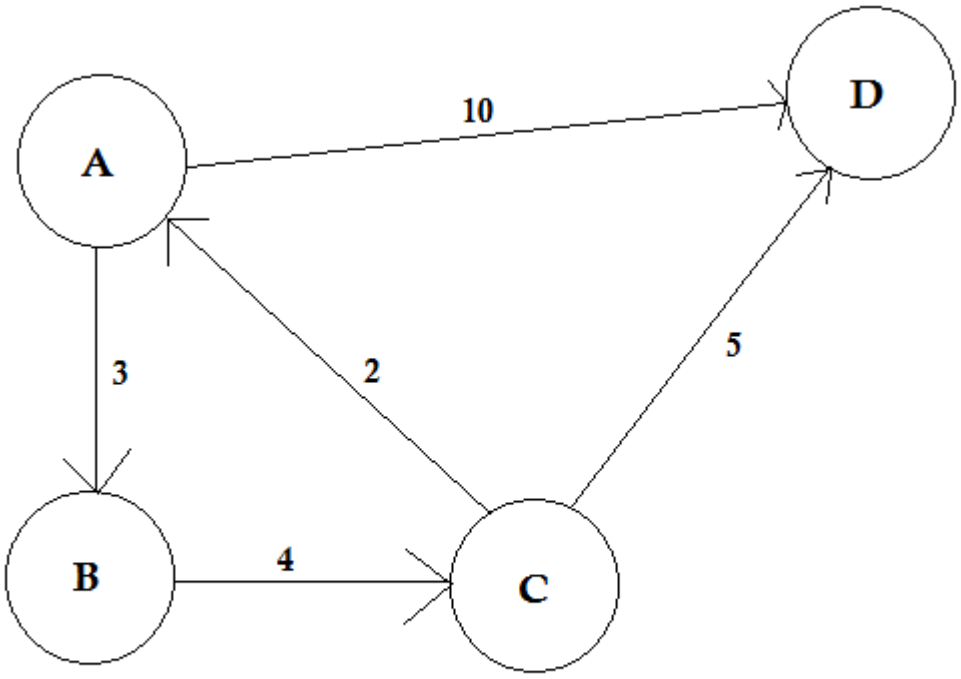


мощности, в зависимости от проблемы.

Невзвешенный график просто противоположный. Предположим, что вес всех ребер одинаковый (предположительно 1).

Дорожка:

Путь представляет собой способ перехода от одного узла к другому. Он состоит из последовательности ребер. Между двумя узлами может быть несколько путей.



В приведенном выше примере есть два пути от **A** до **D. A-> B, B-> C, C-> D** - один путь. Стоимость этого пути равна **3 + 4 + 2 = 9** . Опять же, есть еще один путь **A-> D.** Стоимость этого пути равна **10** . Путь, который стоит наименьший, называется *кратчайшим путем* .

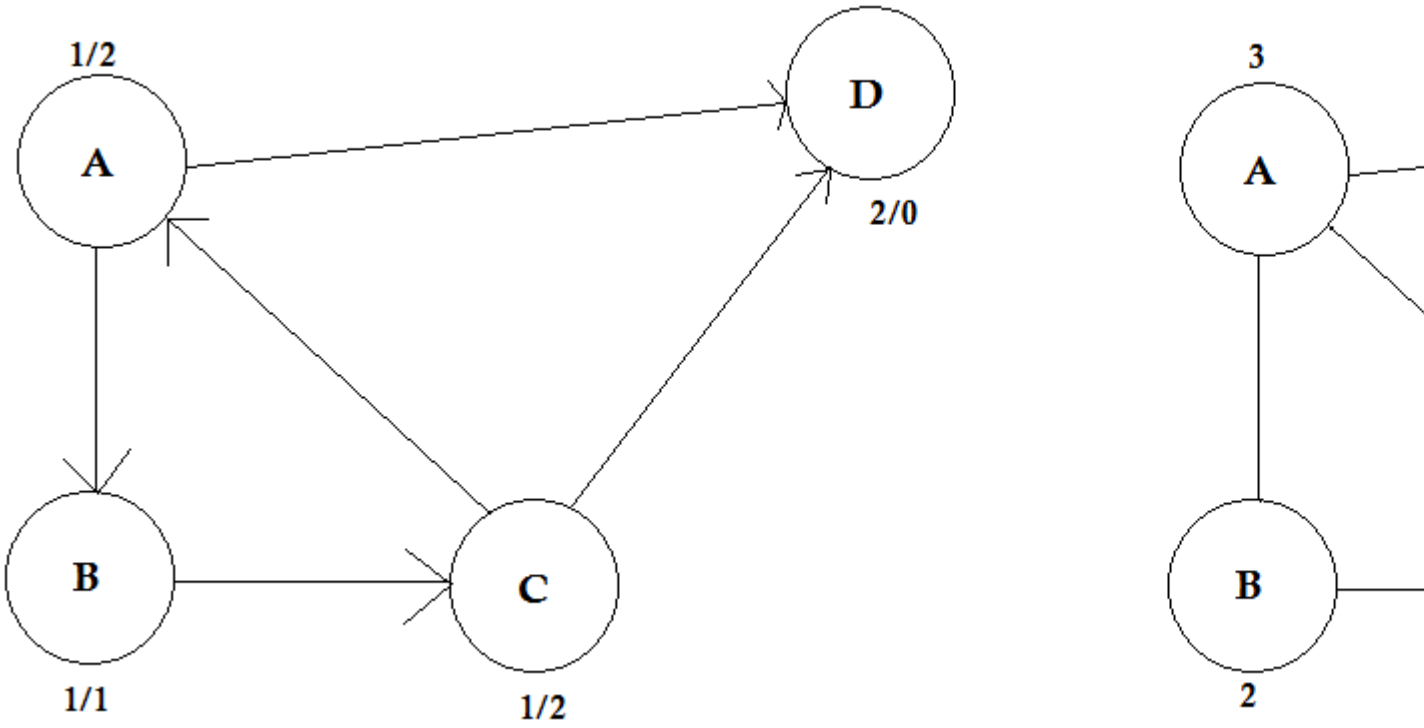
Степень:

Степень вершины - это число ребер, связанных с ней. Если есть какое-либо ребро, которое соединяется с вершиной на обоих концах (цикл), подсчитывается дважды.

В ориентированных графах узлы имеют два типа степеней:

- In-degree: количество ребер, указывающих на узел.
- Out-degree: количество ребер, которые указывают от узла к другим узлам.

Для неориентированных графов их просто называют степенью.



Некоторые алгоритмы, связанные с теорией графов

- Алгоритм Беллмана-Форда
- Алгоритм Дейкстры
- Алгоритм Форда-Фулкерсона
- Алгоритм Крускаля
- Алгоритм ближайшего соседа
- Алгоритм Прима
- Поиск по глубине
- Поиск по ширине

Сохранение графиков (матрица смежности)

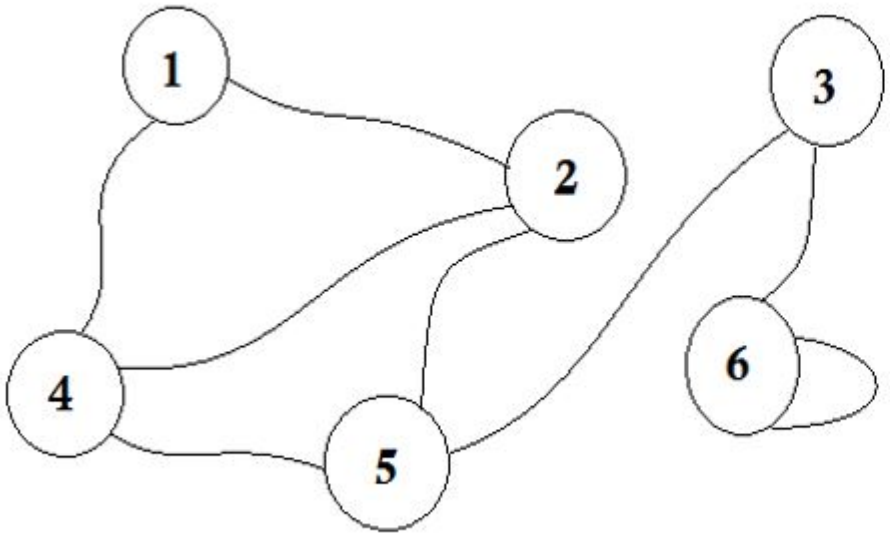
Для хранения графика используются два метода:

- Матрица смежности
- Список прилавков

Матрица смежности представляет собой квадратную матрицу, используемую для представления конечного графа. Элементы матрицы указывают, смежны ли пары вершин или нет в графе.

Смежные средства «рядом или примыкают к чему-то другому» или быть рядом с чем-то. Например, ваши соседи находятся рядом с вами. В теории графов, если мы можем перейти к **узлу B** из **узла A** , можно сказать, что **узел B** смежна с **узлом A.** Теперь мы узнаем о том, как хранить, какие узлы примыкают к одному из них через матрицу Adjacency. Это означает, что мы будем представлять, какие узлы разделяют границу между ними. Здесь

матрица означает 2D-массив.

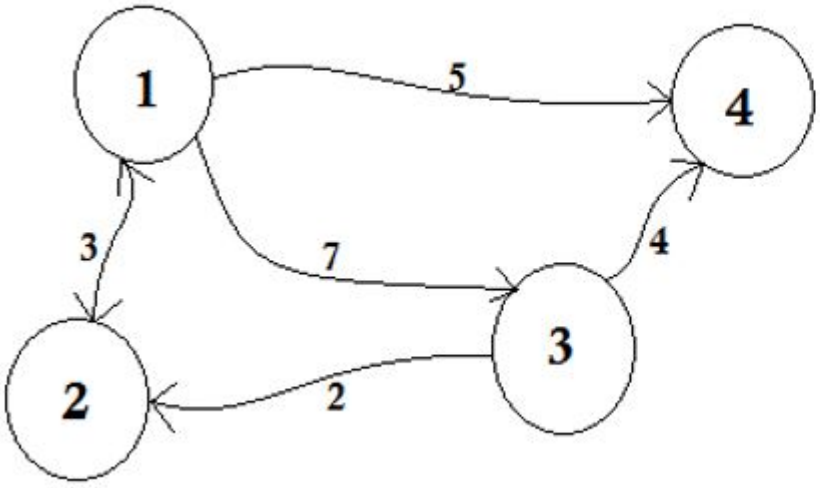


Node	1	2	3
1	0	1	0
2	1	0	0
3	0	0	0
4	1	1	0
5	0	1	1
6	0	0	1

Здесь вы можете увидеть таблицу рядом с графиком, это наша матрица смежности. Здесь **Matrix [i] [j] = 1** представляет собой ребро между **i** и **j** . Если нет ребра, мы просто ставим **Матрицу [i] [j] = 0** .

Эти края могут быть взвешены, так как они могут представлять расстояние между двумя городами. Затем мы помещаем значение в **Матрицу [i] [j]** вместо того, чтобы положить 1.

График, описанный выше, является *двунаправленным* или *ненаправленным* , что означает, что если мы сможем перейти к **узлу 1** из **узла 2** , мы также перейдем к **узлу 2** из **узла 1** . Если график был *направлен* , тогда на одной стороне графика был бы указатель стрелки. Даже тогда мы могли бы представлять его с помощью матрицы смежности.



Node	1	2
1	Inf	3
2	3	Inf
3	Inf	2
4	Inf	Inf

Мы представляем узлы, которые не разделяют ребро на *бесконечность* . Следует заметить, что если граф неориентирован, матрица становится *симметричной* .

Псевдокод для создания матрицы:

```
Procedure AdjacencyMatrix(N):    //N represents the number of nodes
Matrix[N][N]
for i from 1 to N
    for j from 1 to N
        Take input -> Matrix[i][j]
    endfor
endfor
```

Мы также можем заполнить матрицу следующим образом:

```
Procedure AdjacencyMatrix(N, E):    // N -> number of nodes
Matrix[N][E]                        // E -> number of edges
for i from 1 to E
    input -> n1, n2, cost
    Matrix[n1][n2] = cost
    Matrix[n2][n1] = cost
endfor
```

Для ориентированных графов мы можем удалить строку **Matrix [n2] [n1] = cost** .

Недостатки использования матрицы смещения:

Память - огромная проблема. Независимо от того, сколько ребер существует, нам всегда понадобится матрица размера N * N, где N - количество узлов. Если есть 10000 узлов, размер матрицы будет 4 * 10000 * 10000 около 381 мегабайт. Это огромная потеря памяти, если мы рассмотрим графики с несколькими ребрами.

Предположим, мы хотим выяснить, к какому узлу мы можем перейти от узла **u** . Нам нужно будет проверить всю строку **u** , которая стоит много времени.

Единственное преимущество в том, что мы можем легко найти связь между **uv-** узлами и их стоимостью с помощью Adjacency Matrix.

Java-код, реализованный с использованием вышеуказанного псевдокода:

```
import java.util.Scanner;

public class Represent_Graph_Adjacency_Matrix
{
    private final int vertices;
    private int[][] adjacency_matrix;

    public Represent_Graph_Adjacency_Matrix(int v)
    {
        vertices = v;
        adjacency_matrix = new int[vertices + 1][vertices + 1];
    }

    public void makeEdge(int to, int from, int edge)
    {
        try
```

```
        {
            adjacency_matrix[to][from] = edge;
        }
        catch (ArrayIndexOutOfBoundsException index)
        {
            System.out.println("The vertices does not exists");
        }
    }

    public int getEdge(int to, int from)
    {
        try
        {
            return adjacency_matrix[to][from];
        }
        catch (ArrayIndexOutOfBoundsException index)
        {
            System.out.println("The vertices does not exists");
        }
        return -1;
    }

    public static void main(String args[])
    {
        int v, e, count = 1, to = 0, from = 0;
        Scanner sc = new Scanner(System.in);
        Represent_Graph_Adjacency_Matrix graph;
        try
        {
            System.out.println("Enter the number of vertices: ");
            v = sc.nextInt();
            System.out.println("Enter the number of edges: ");
            e = sc.nextInt();

            graph = new Represent_Graph_Adjacency_Matrix(v);

            System.out.println("Enter the edges: <to> <from>");
            while (count <= e)
            {
                to = sc.nextInt();
                from = sc.nextInt();

                graph.makeEdge(to, from, 1);
                count++;
            }

            System.out.println("The adjacency matrix for the given graph is: ");
            System.out.print("  ");
            for (int i = 1; i <= v; i++)
                System.out.print(i + " ");
            System.out.println();

            for (int i = 1; i <= v; i++)
            {
                System.out.print(i + " ");
                for (int j = 1; j <= v; j++)
                    System.out.print(graph.getEdge(i, j) + " ");
                System.out.println();
            }

        }
```

```
        catch (Exception E)
        {
            System.out.println("Somthing went wrong");
        }

        sc.close();
    }
}
```

Запуск кода: Сохраните файл и скомпилируйте его с помощью `javac Represent_Graph_Adjacency_Matrix.java`

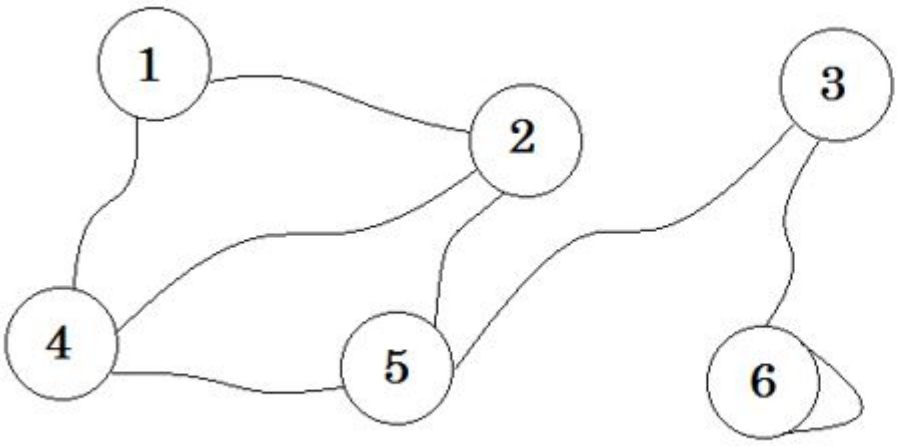
Пример:

```
$ java Represent_Graph_Adjacency_Matrix
Enter the number of vertices:
4
Enter the number of edges:
6
Enter the edges: <to> <from>
1 1
3 4
2 3
1 4
2 4
1 2
The adjacency matrix for the given graph is:
  1 2 3 4
1 1 1 0 1
2 0 0 1 1
3 0 0 0 1
4 0 0 0 0
```

Сохранение графиков (список адресов)

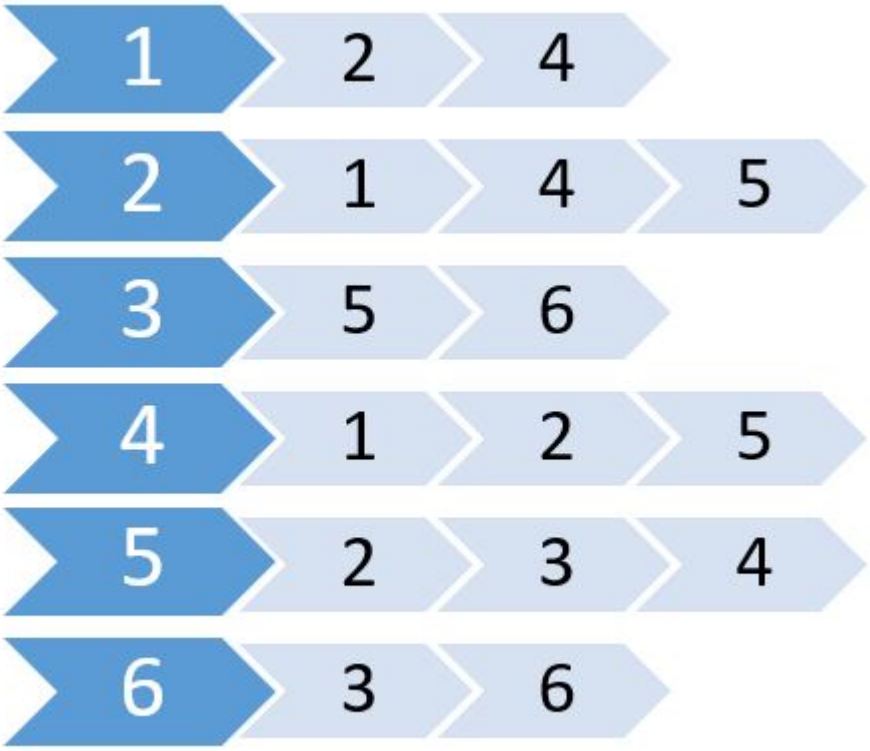
Список смежности представляет собой набор неупорядоченных списков, используемых для представления конечного графа. Каждый список описывает множество соседей вершины в графе. Для хранения графиков требуется меньше памяти.

Посмотрим на график и его матрицу смежности:



Node	1	2	3
1	0	1	0
2	1	0	0
3	0	0	0
4	1	1	0
5	0	1	1
6	0	0	1

Теперь мы создаем список, используя эти значения.



Это называется списком смежности. Он показывает, какие узлы связаны с узлами. Мы можем хранить эту информацию с помощью 2D-массива. Но это будет стоить нам той же памяти, что и матрица смежности. Вместо этого мы будем использовать динамически выделенную память для ее хранения.

Многие языки поддерживают **Vector** или **List,** которые мы можем использовать для хранения списка смежности. Для этого нам не нужно указывать размер **списка** . Нам нужно только указать максимальное количество узлов.

Псевдокод будет:

```
Procedure Adjacency-List(maxN, E):       // maxN denotes the maximum number of nodes
edge[maxN] = Vector()                    // E denotes the number of edges
for i from 1 to E
    input -> x, y                        // Here x, y denotes there is an edge between x, y
    edge[x].push(y)
    edge[y].push(x)
end for
Return edge
```

Так как это - неориентированный граф, то есть ребро от **x** до **y** , также существует ребро от **y** до **x** . Если бы это был ориентированный граф, мы бы опустили второй. Для взвешенных графиков нам нужно также сохранить стоимость. Мы создадим другой **вектор** или **список c** именем **cost [],** чтобы сохранить их. Псевдокод:

```
Procedure Adjacency-List(maxN, E):
edge[maxN] = Vector()
cost[maxN] = Vector()
for i from 1 to E
    input -> x, y, w
    edge[x].push(y)
    cost[x].push(w)
end for
Return edge, cost
```

Из этого можно легко узнать общее количество узлов, подключенных к любому узлу, и каковы эти узлы. Это занимает меньше времени, чем матрица смежности. Но если нам нужно выяснить, есть ли граница между **u** и **v** , было бы проще, если бы мы сохранили матрицу смежности.

Прочитайте график онлайн: https://riptutorial.com/ru/algorithm/topic/2299/график

глава 24: Графические обходы

Examples

Функция поиска глубины первого поиска

Функция принимает аргумент текущего индекса узла, списка смежности (сохраненного в векторе векторов в этом примере) и вектора булевых, чтобы отслеживать, какой узел был посещен.

```
void dfs(int node, vector<vector<int>>* graph, vector<bool>* visited) {
    // check whether node has been visited before
    if((*visited)[node])
        return;

    // set as visited to avoid visiting the same node twice
    (*visited)[node] = true;

    // perform some action here
    cout << node;

    // traverse to the adjacent nodes in depth-first manner
    for(int i = 0; i < (*graph)[node].size(); ++i)
        dfs((*graph)[node][i], graph, visited);
}
```

Прочитайте Графические обходы онлайн: https://riptutorial.com/ru/algorithm/topic/9493/графические-обходы

глава 25: деревья

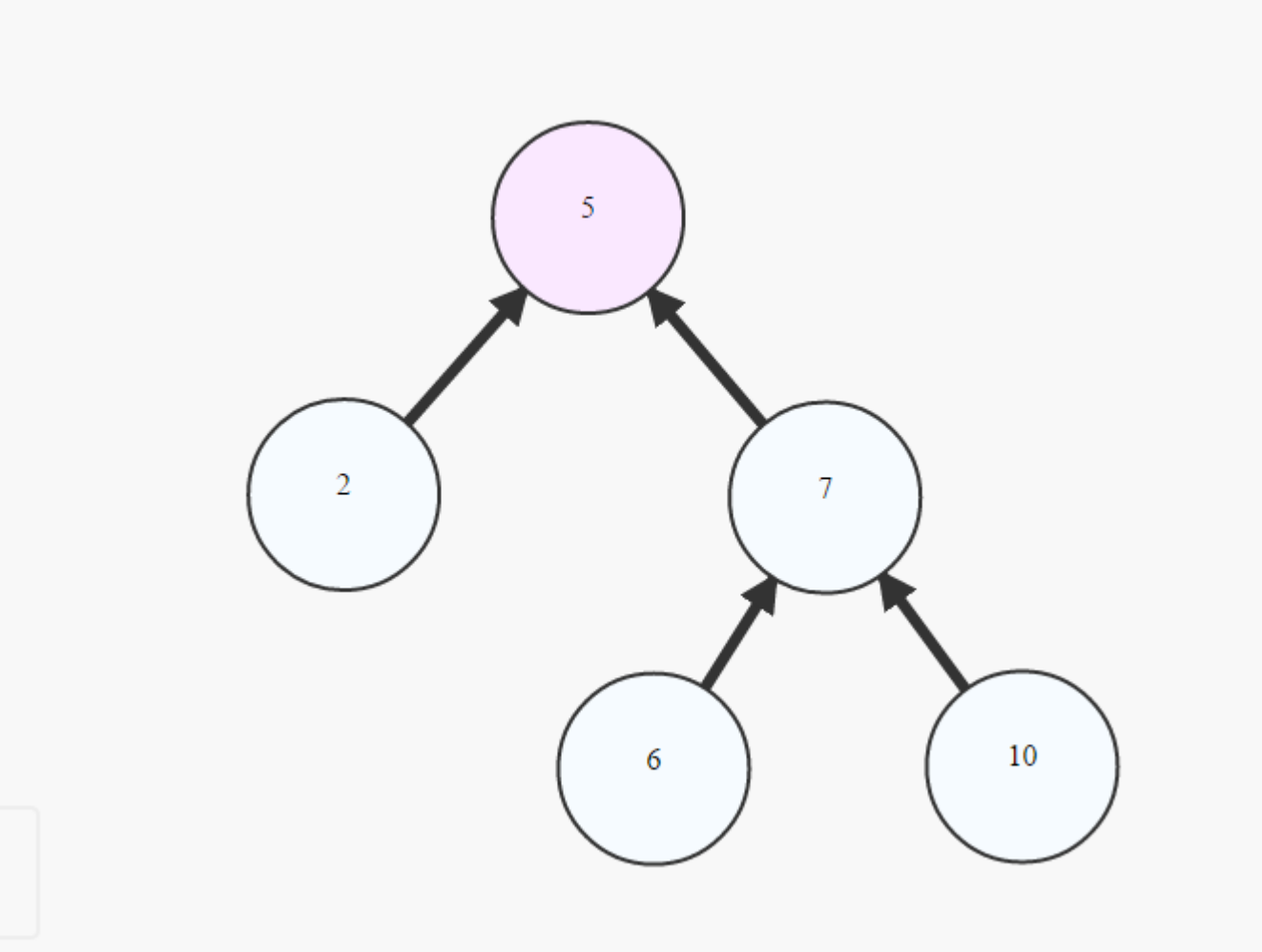
замечания

Деревья - это подкатегория или подтип графов узлов. Они повсеместны в информатике из-за их распространенности как модели для многих различных алгоритмических структур, которые, в свою очередь, применяются во многих разных алгоритмах

Examples

Вступление

Деревья являются подтипом более общей структуры данных графа узлов.



Чтобы быть деревом, граф должен удовлетворять двум требованиям:

- **Он ацикличен.** Он не содержит циклов (или «циклов»).
- **Это связано.** Для любого заданного узла в графе каждый узел доступен. Все узлы доступны по одному пути на графике.

Структура данных дерева довольно распространена в информатике. Деревья используются для моделирования многих различных алгоритмических структур данных, таких как обычные двоичные деревья, красно-черные деревья, B-деревья, AB-деревья, 23-деревья, кучи и попытки.

обычно упоминается Дерево как `Rooted Tree` :

```
choosing 1 cell to be called `Root`
painting the `Root` at the top
creating lower layer for each cell in the graph depending on their distance from the root –the
bigger the distance, the lower the cells (example above)
```

общий символ для деревьев: `T`

Типичное представление о древесном дереве

Обычно мы представляем анарное дерево (одно с потенциально неограниченными дочерними элементами на узел) как бинарное дерево (одно с ровно двумя детьми на узел). «Следующий» ребенок считается братом. Обратите внимание: если дерево двоично, это представление создает дополнительные узлы.

Затем мы перебираем братьев и сестер и рекурсируем детей. Поскольку большинство деревьев относительно мелкие - много детей, но только несколько уровней иерархии, это приводит к эффективному коду. Обратите внимание, что генеалогия человека является исключением (множество уровней предков, всего несколько детей на уровень).

При необходимости можно сохранить обратные указатели, чтобы дерево могло быть поднято. Их более сложно поддерживать.

Обратите внимание, что типично иметь одну функцию для вызова корня и рекурсивную функцию с дополнительными параметрами, в данном случае глубиной дерева.

```
struct node
{
   struct node *next;
   struct node *child;
   std::string data;
}

void printtree_r(struct node *node, int depth)
{
   int i;

   while(node)
   {
       if(node->child)
       {
          for(i=0;i<depth*3;i++)
              printf(" ");
          printf("{\n"):
          printtree_r(node->child, depth +1);
```

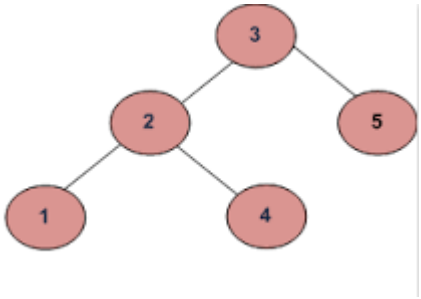
```
            for(i=0;i<depth*3;i++)
                printf(" ");
            printf("{\n"):

            for(i=0;i<depth*3;i++)
               printf(" ");
             printf("%s\n", node->data.c_str());

             node = node->next;
          }
      }
  }

  void printtree(node *root)
  {
     printree_r(root, 0);
  }
```

Чтобы проверить, являются ли два двоичных дерева одинаковыми или нет

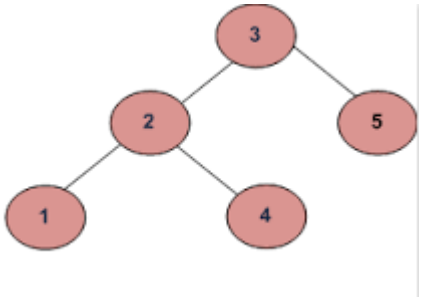
1. Например, если входы:

Пример: 1

а)



б)

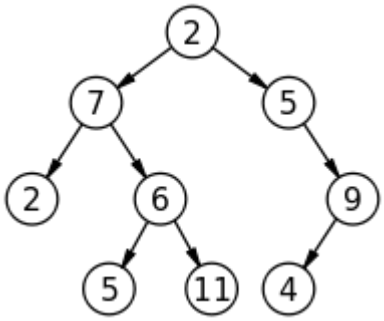


Результат должен быть правдой.

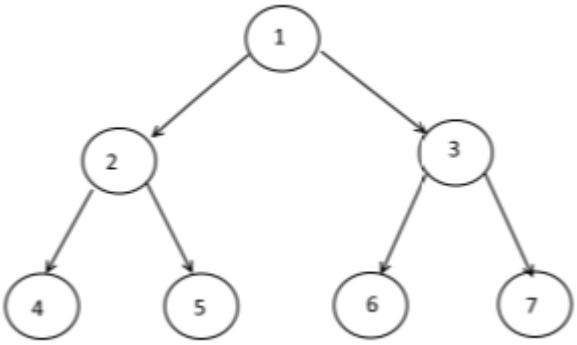
Пример: 2

Если входы:

а)



б)



Результат должен быть ложным.

Псевдокод для этого же:

```
boolean sameTree(node root1, node root2){

if(root1 == NULL && root2 == NULL)
return true;

if(root1 == NULL || root2 == NULL)
return false;

if(root1->data == root2->data
     && sameTree(root1->left,root2->left)
        && sameTree(root1->right, root2->right))
return true;

}
```

Прочитайте деревья онлайн: https://riptutorial.com/ru/algorithm/topic/5737/деревья

глава 26: Деревья двоичного поиска

Вступление

Двоичное дерево - это дерево, в котором каждый узел имеет максимум двух детей. Двоичное дерево поиска (BST) - это двоичное дерево, элементы которого расположены в специальном порядке. В каждом BST все значения (т.е. ключ) в левом вспомогательном дереве меньше значений в правом поддереве.

Examples

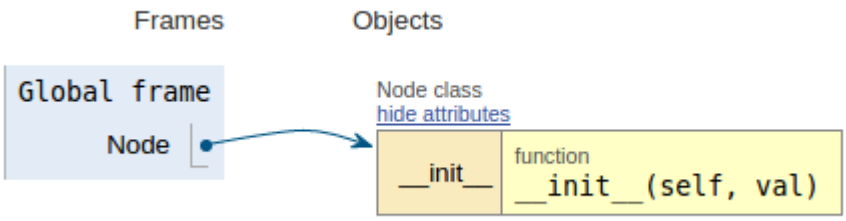
Дерево двоичного поиска - Вставка (Python)

Это простая реализация вставки двоичного дерева поиска с использованием Python.

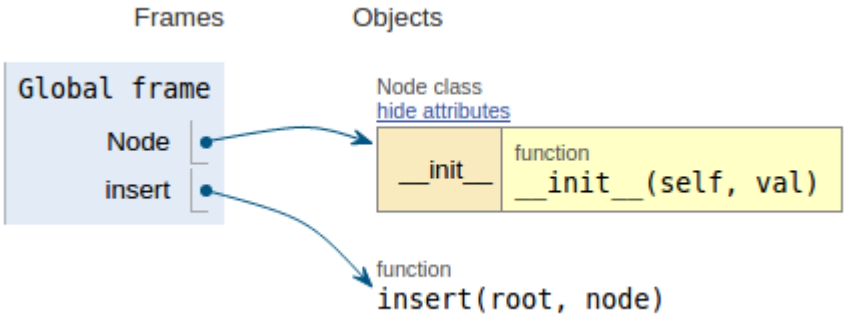
Пример показан ниже:

Следуя фрагменту кода, на каждом изображении отображается визуализация выполнения, что упрощает визуализацию работы этого кода.

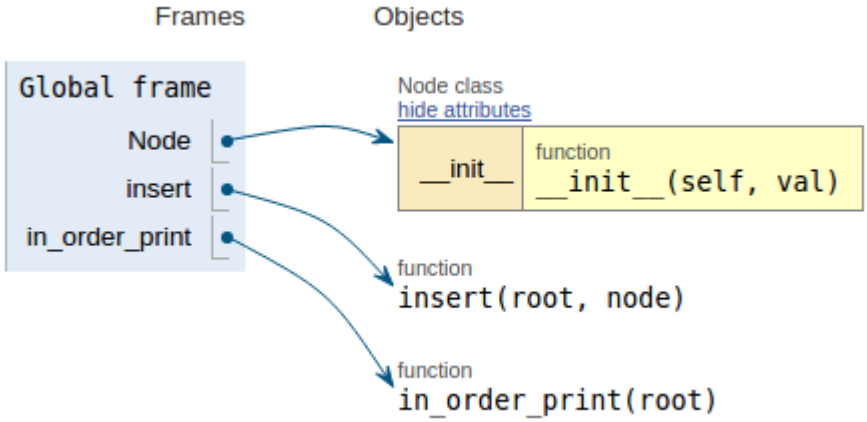
```
class Node:
    def __init__(self, val):
        self.l_child = None
        self.r_child = None
        self.data = val
```



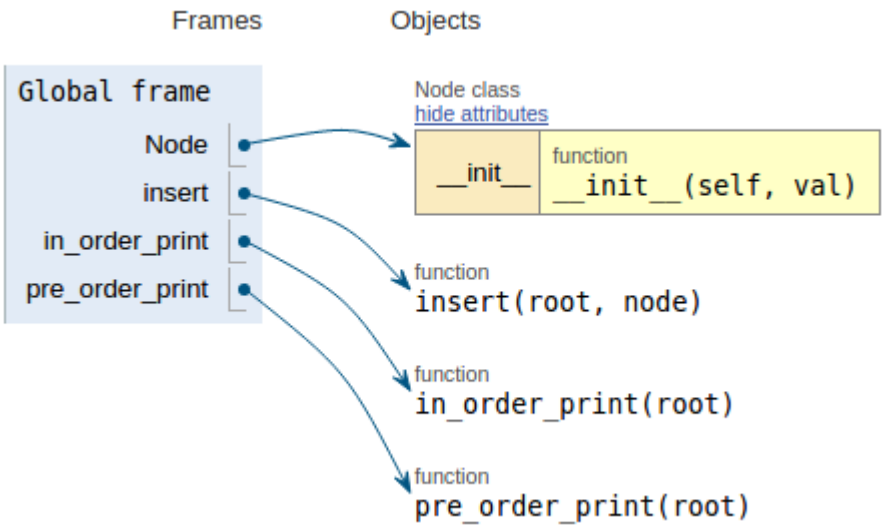

```
def insert(root, node):
    if root is None:
        root = node
    else:
        if root.data > node.data:
            if root.l_child is None:
                root.l_child = node
            else:
                insert(root.l_child, node)
        else:
            if root.r_child is None:
                root.r_child = node
            else:
                insert(root.r_child, node)
```




```
def in_order_print(root):
    if not root:
        return
    in_order_print(root.l_child)
    print root.data
    in_order_print(root.r_child)
```



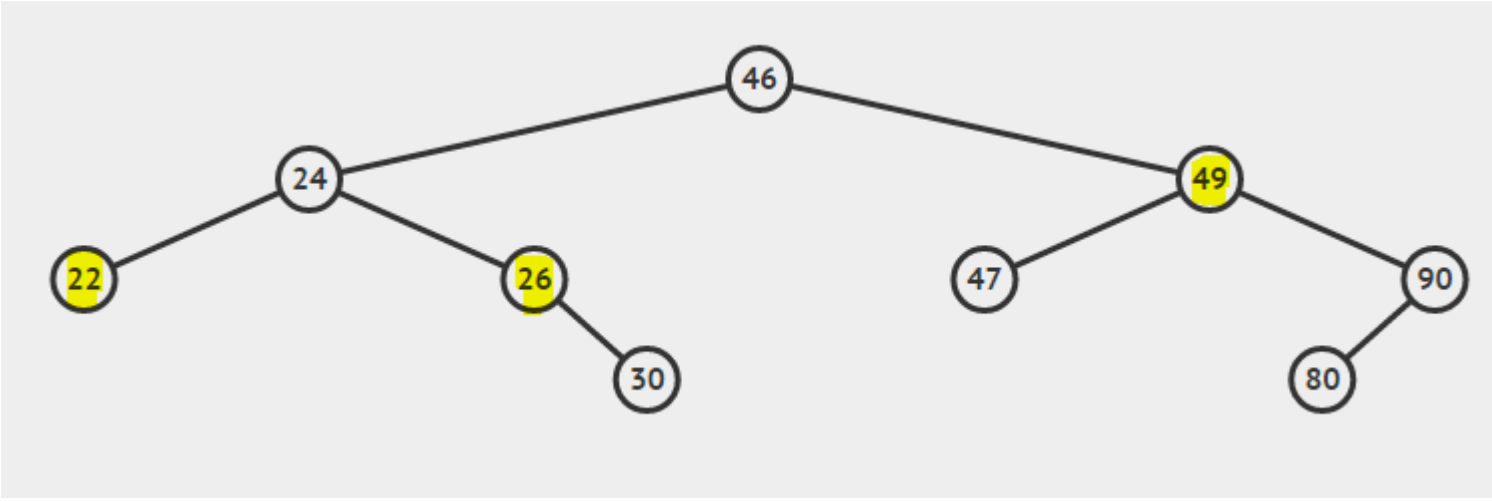
```
def pre_order_print(root):
    if not root:
        return
    print root.data
    pre_order_print(root.l_child)
    pre_order_print(root.r_child)
```



Двоичное дерево поиска - удаление (C ++)

Прежде чем начать с удаления, я просто хочу поместить некоторые огни на то, что является бинарным деревом поиска (BST). Каждый узел в BST может иметь максимум два узла (левый и правый дочерние элементы). Левое поддерево узла имеет ключ меньше или равен ключу его родительского узла. В правом поддереве узла есть ключ, который больше, чем ключ его родительского узла.

Удаление узла в дереве при сохранении его **свойства двоичного дерева поиска.**



При удалении узла необходимо учитывать три случая.

- Случай 1: Узел, подлежащий удалению, является листовым узлом (Узел со значением 22).

- Случай 2: Узел, который нужно удалить, имеет один ребенок. (Узел со значением 26).
- Случай 3: Узел, который нужно удалить, имеет обоих детей. (Узел со значением 49).

Объяснение случаев:

1. Когда удаляемый узел является листовым узлом, просто удалите узел и передайте `nullptr` в его родительский узел.
2. Когда удаляемый узел имеет только один дочерний элемент, затем копируйте дочернее значение в значение узла и удалите дочерний **элемент (преобразованный в случай 1)**
3. Когда узел, который должен быть удален, имеет два дочерних элемента, то минимальный из его правого поддерева может быть скопирован в узел, а затем минимальное значение может быть удалено из правого поддерева узла **(преобразовано в случай 2)**

Примечание . Минимальное значение в правом подэлементе может содержать максимум один ребенок, а слишком правильный ребенок, если он имеет левый дочерний элемент, означает, что это не минимальное значение или оно не соответствует свойству BST.

Структура узла в дереве и код для удаления:

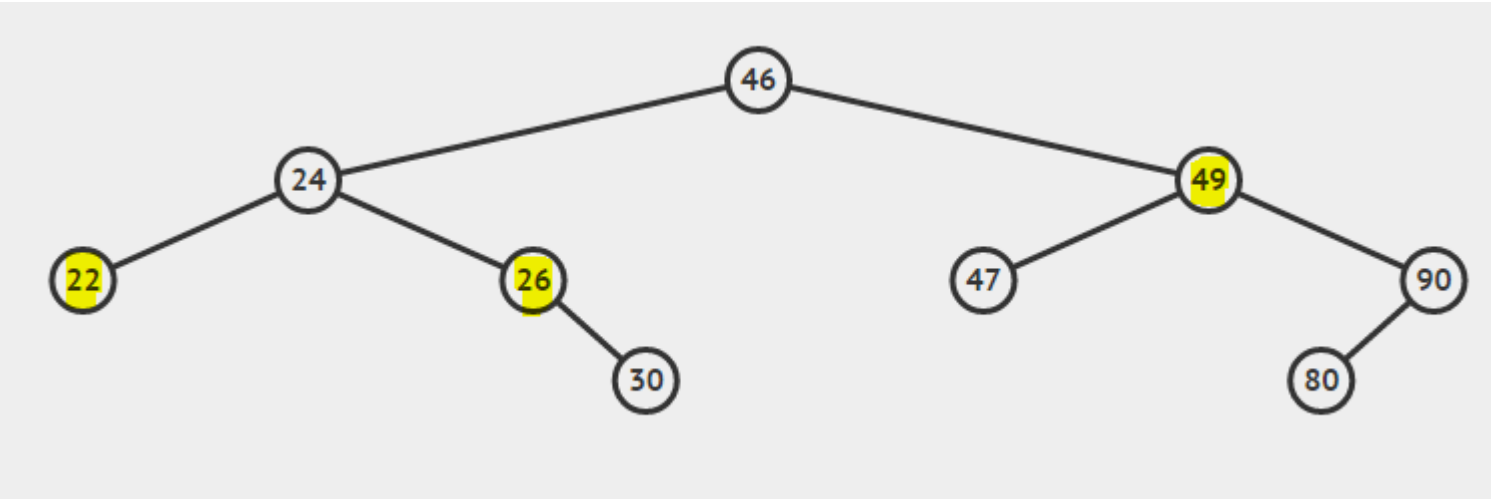
```
struct node
{
    int data;
    node *left, *right;
};

node* delete_node(node *root, int data)
{
  if(root == nullptr) return root;
  else if(data < root->data) root->left  = delete_node(root->left, data);
  else if(data > root->data) root->right = delete_node(root->right, data);

  else
  {
    if(root->left == nullptr && root->right == nullptr) // Case 1
    {
      free(root);
      root = nullptr;
    }
    else if(root->left == nullptr)       // Case 2
    {
       node* temp = root;
       root= root->right;
       free(temp);
    }
    else if(root->right == nullptr)      // Case 2
    {
       node* temp = root;
       root = root->left;
       free(temp);
    }
    else                                 // Case 3
    {
```

```
        node* temp = root->right;

        while(temp->left != nullptr) temp = temp->left;

        root->data = temp->data;
        root->right = delete_node(root->right, temp->data);
    }
  }
  return root;
}
```

Сложность времени выше кода равна O (*h*), где *h* - высота дерева.

Самый низкий общий предок в BST

Рассмотрим BST:



Самый низкий общий предок 22 и 26 составляет 24

Самый низкий общий предок 26 и 49 составляет 46

Самый низкий общий предок 22 и 24 составляет 24

Свойство дерева двоичного поиска может использоваться для нахождения узлов нижнего предка

Код Псевдоэ:

```
lowestCommonAncestor(root,node1, node2){

if(root == NULL)
return NULL;



 else if(node1->data == root->data || node2->data== root->data)
    return root;

    else if((node1->data <= root->data && node2->data > root->data)
```

```
            || (node2->data <= root->data && node1->data > root->data)){

             return root;
    }

    else if(root->data > max(node1->data,node2->data)){
            return lowestCommonAncestor(root->left, node1, node2);
    }

        else {
            return lowestCommonAncestor(root->right, node1, node2);
      }
        }
```

Дерево двоичного поиска - Python

```
class Node(object):
    def __init__(self, val):
        self.l_child = None
        self.r_child = None
        self.val = val


class BinarySearchTree(object):
    def insert(self, root, node):
        if root is None:
            return node

        if root.val < node.val:
            root.r_child = self.insert(root.r_child, node)
        else:
            root.l_child = self.insert(root.l_child, node)

        return root


    def in_order_place(self, root):
        if not root:
            return None
        else:
            self.in_order_place(root.l_child)
            print root.val
            self.in_order_place(root.r_child)

    def pre_order_place(self, root):
        if not root:
            return None
        else:
            print root.val
            self.pre_order_place(root.l_child)
            self.pre_order_place(root.r_child)

    def post_order_place(self, root):
        if not root:
            return None
        else:
            self.post_order_place(root.l_child)
            self.post_order_place(root.r_child)
            print root.val
```

"" Создать другой узел и вставить в него данные "" "

```
r = Node(3)
node = BinarySearchTree()
nodeList = [1, 8, 5, 12, 14, 6, 15, 7, 16, 8]

for nd in nodeList:
    node.insert(r, Node(nd))

print "------In order ---------"
print (node.in_order_place(r))
print "------Pre order ---------"
print (node.pre_order_place(r))
print "------Post order ---------"
print (node.post_order_place(r))
```

Прочитайте Деревья двоичного поиска онлайн: https://riptutorial.com/ru/algorithm/topic/5735/деревья-двоичного-поиска

глава 27: Динамическое программирование

Вступление

Динамическое программирование является широко используемой концепцией и часто используется для оптимизации. Это относится к упрощению сложной проблемы, разбивая ее на более простые подзадачи рекурсивным образом, как правило, подход Bottom up. Существует два ключевых атрибута, которые должны иметь проблемы для динамического программирования: «Оптимальная субструктура» и «Перекрывающиеся подзадачи». Для достижения своей оптимизации в программе Dynamics используется концепция «Запоминание»

замечания

Динамическое программирование - это усовершенствование Brute Force, см. Этот пример, чтобы понять, как можно получить решение динамического программирования от Brute Force.

Решение Dynamic Programming Solution имеет два основных требования:

1. Проблемы с перекрытием
2. Оптимальная основа

Перекрывающиеся подзадачи означают, что результаты меньших версий проблемы повторно используются несколько раз, чтобы прийти к решению исходной проблемы

Оптимальная подструктура означает, что существует метод вычисления проблемы из ее подзадач.

Решение динамического программирования имеет 2 основных компонента: ***состояние*** и ***переход***

Государство ссылается на подзадачу исходной проблемы.

Переход - это метод решения проблемы на основе ее подзадач

Время, затраченное на решение динамического программирования, можно рассчитать как число `No. of States * Transition Time` . Таким образом, если решение имеет `N^2` состояния, а переход - `O(N)` , то решение займет примерно `O(N^3)` раз.

Examples

Проблема с рюкзаком

0-1 Рюкзак

Проблема с рюкзаком представляет собой проблему, когда задан набор элементов, каждый с весом, значением и **ровно 1 копией** , определяет, какие элементы (ы) включать в коллекцию, чтобы общий вес был меньше или равен заданному предел и общее значение как можно больше.

Пример C ++:

Реализация :

```
int knapsack(vector<int> &value, vector<int> &weight, int N, int C){
    int dp[C+1];
    for (int i = 1; i <= C; ++i){
        dp[i] = -100000000;
    }
    dp[0] = 0;
    for (int i = 0; i < N; ++i){
        for (int j = C; j >= weight[i]; --j){
            dp[j] = max(dp[j],dp[j-weight[i]]+value[i]);
        }
    }
    return dp[C];
}
```

Тест :

```
3 5
5 2
2 1
3 2
```

Выход :

```
3
```

Это означает, что максимальное значение может быть достигнуто 3, что достигается выбором (2,1) и (3,2).

Неограниченный рюкзак

Проблема с неограниченным рюкзаком представляет собой проблему, которая задает набор элементов, каждый с весом, значением и **бесконечными копиями** , определяет

количество каждого элемента, который должен быть включен в коллекцию, чтобы общий вес был меньше или равен заданному пределу и общее значение максимально возможно.

Python (2.7.11) Пример:

Реализация :

```
def unbounded_knapsack(w, v, c): # weight, value and capactiy
    m = [0]
    for r in range(1, c+1):
        val = m[r-1]
        for i, wi in enumerate(w):
            if wi > r:
                continue
            val = max(val, v[i] + m[r-wi])
        m.append(val)
    return m[c] # return the maximum value can be achieved
```

Сложностью этой реализации является `O(nC)` , где n - количество элементов.

Тест :

```
w = [2, 3, 4, 5, 6]
v = [2, 4, 6, 8, 9]

print unbounded_knapsack(w, v, 13)
```

Выход :

```
20
```

Это означает, что максимальное значение может быть достигнуто равным 20, что достигается выбором (5, 8), (5, 8) и (3, 4).

Алгоритм взвешенного планирования работы

Алгоритм взвешенного планирования работы также можно обозначить как алгоритм выбора взвешенной активности.

Проблема в том, что с заданными заданиями с указанием времени начала и окончания времени и прибыль, которую вы делаете, когда вы заканчиваете работу, какова максимальная прибыль, которую вы можете сделать, если нет двух заданий, может выполняться параллельно?

Это выглядит как «Выбор действия», используя «Жадный алгоритм», но есть еще один поворот. То есть вместо максимизации количества завершенных заданий мы сосредоточимся на получении максимальной прибыли. Количество выполненных заданий здесь не имеет значения.

Давайте посмотрим на пример:

Name	A	B	C	D	E	F
(Start Time, Finish Time)	(2,5)	(6,7)	(7,9)	(1,3)	(5,8)	(4,6)
Profit	6	4	2	5	11	5

Работы обозначаются именем, временем начала и окончания и прибылью. После нескольких итераций мы можем узнать, выполняем ли мы **Job-A** и **Job-E** максимальную прибыль 17. Теперь, как найти это, используя алгоритм?

Первое, что мы делаем, - сортировать задания по времени окончания в неубывающем порядке. Почему мы это делаем? Это потому, что, если мы выберем задание, которое занимает меньше времени, то мы оставляем самое большое количество времени для выбора других заданий. У нас есть:

Name	D	A	F	B	E	C
(Start Time, Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2

У нас будет дополнительный временный массив **Acc_Prof** размером **n** (здесь **n** обозначает общее количество заданий). Это будет содержать максимальную накопленную прибыль от выполнения заданий. Не понял? Подождите и смотрите. Мы будем инициализировать значения массива с прибылью каждого задания. Это означает, что **Acc_Prof [i]** сначала получит прибыль от выполнения **i-й** работы.

Acc_Prof	5	6	5	4	11	2

Теперь давайте обозначим **позицию 2** с **i** , а **положение 1** обозначим через **j** . Наша стратегия будет состоять в повторении **j** от **1** до **i-1,** и после каждой итерации мы увеличим **i** на 1, пока **i не** станет **n + 1** .

```
                          j       i
```

Name	D	A	F	B	E	C
(Start Time, Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2
Acc_Prof	5	6	5	4	11	2

```
+------------------------+---------+---------+---------+---------+---------+---------+
```

Мы проверяем, перекрываются ли **задания [i]** и **Job [j]** , то есть, если время **окончания работы [j]** больше, чем время запуска **Job [i]** , то эти два задания не могут быть выполнены вместе. Однако, если они не перекрываются, мы проверим, есть ли **Acc_Prof [j] + Profit [i]> Acc_Prof [i]** . Если это так, мы обновим **Acc_Prof [i] = Acc_Prof [j] + Profit [i]** . То есть:

```
if Job[j].finish_time <= Job[i].start_time
    if Acc_Prof[j] + Profit[i] > Acc_Prof[i]
        Acc_Prof[i] = Acc_Prof[j] + Profit[i]
    endif
endif
```

Здесь **Acc_Prof [j] + Profit [i]** представляет собой накопленную прибыль от выполнения этих двух заданий. Давайте проверим это для нашего примера:

Здесь **Job [j]** перекрывается с **Job [i]** . Таким образом, это невозможно сделать вместе. Так как наш **j** равен **i-1** , мы увеличиваем значение **i** до **i + 1** , равное **3** . И сделаем **j = 1** .

```
                            j                   i

+------------------------+---------+---------+---------+---------+---------+---------+
|          Name          |    D    |    A    |    F    |    B    |    E    |    C    |
+------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+------------------------+---------+---------+---------+---------+---------+---------+
|         Profit         |    5    |    6    |    5    |    4    |    11   |    2    |
+------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof       |    5    |    6    |    5    |    4    |    11   |    2    |
+------------------------+---------+---------+---------+---------+---------+---------+
```

Теперь **Job [j]** и **Job [i]** не перекрываются. Общая сумма прибыли, которую мы можем сделать, выбрав эти два задания: **Acc_Prof [j] + Profit [i] = 5 + 5 = 10,** что больше, чем **Acc_Prof [i]** . Поэтому мы обновляем **Acc_Prof [i] = 10** . Мы также увеличиваем **j** на 1. Получаем,

```
                                    j       i

+------------------------+---------+---------+---------+---------+---------+---------+
|          Name          |    D    |    A    |    F    |    B    |    E    |    C    |
+------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+------------------------+---------+---------+---------+---------+---------+---------+
|         Profit         |    5    |    6    |    5    |    4    |    11   |    2    |
+------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof       |    5    |    6    |    10   |    4    |    11   |    2    |
+------------------------+---------+---------+---------+---------+---------+---------+
```

Здесь **Job [j]** перекрывается с **Job [i],** а **j** также равно **i-1** . Поэтому мы увеличиваем **i** на 1 и делаем **j = 1** . Мы получаем,

```
                            j                          i

+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    D    |    A    |    F    |    B    |    E    |    C    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    5    |    6    |    5    |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof        |    5    |    6    |    10   |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

Теперь **Job [j]** и **Job [i]** не перекрываются, мы получаем накопленную прибыль **5 + 4 = 9** , которая больше **Acc_Prof [i]** . Мы обновляем **Acc_Prof [i] = 9** и увеличиваем **j** на 1.

```
                                  j                  i

+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    D    |    A    |    F    |    B    |    E    |    C    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    5    |    6    |    5    |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof        |    5    |    6    |    10   |    9    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

Опять **Job [j]** и **Job [i]** не перекрываются. Накопленная прибыль равна: **6 + 4 = 10** , что больше, чем **Acc_Prof [i]** . Мы снова обновляем **Acc_Prof [i] = 10** . Мы увеличиваем **j** на 1. Получаем:

```
                                        j         i

+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    D    |    A    |    F    |    B    |    E    |    C    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    5    |    6    |    5    |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof        |    5    |    6    |    10   |    10   |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

Если мы продолжим этот процесс, после повторения всей таблицы с помощью **i** наша таблица будет выглядеть следующим образом:

```
+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    D    |    A    |    F    |    B    |    E    |    C    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    5    |    6    |    5    |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof        |    5    |    6    |    10   |    14   |    17   |    8    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

* Несколько шагов были пропущены, чтобы сделать документ короче.

Если мы итерации через массив **Acc_Prof** , мы можем узнать, что максимальная прибыль равна **17** ! Псевдокод:

```
Procedure WeightedJobScheduling(Job)
sort Job according to finish time in non-decreasing order
for i -> 2 to n
    for j -> 1 to i-1
        if Job[j].finish_time <= Job[i].start_time
            if Acc_Prof[j] + Profit[i] > Acc_Prof[i]
                Acc_Prof[i] = Acc_Prof[j] + Profit[i]
            endif
        endif
    endfor
endfor

maxProfit = 0
for i -> 1 to n
    if maxProfit < Acc_Prof[i]
        maxProfit = Acc_Prof[i]
return maxProfit
```

Сложность **заполнения** массива **Acc_Prof** равна **O (n ^{2}).** Обход массива принимает **O (n)** . Таким образом, общая сложность этого алгоритма равна **O (n ^{2}).**

Теперь, если мы хотим узнать, какие задания были выполнены для получения максимальной прибыли, нам нужно пройти массив в обратном порядке, и если **Acc_Prof** соответствует **maxProfit** , мы будем **вызывать имя** задания в **стеке** и вычитать **прибыль** эта работа от **maxProfit** . Мы будем делать это до нашего **maxProfit> 0** или мы достигнем начальной точки массива **Acc_Prof** . Псевдокод будет выглядеть так:

```
Procedure FindingPerformedJobs(Job, Acc_Prof, maxProfit):
S = stack()
for i -> n down to 0 and maxProfit > 0
    if maxProfit is equal to Acc_Prof[i]
        S.push(Job[i].name
        maxProfit = maxProfit - Job[i].profit
    endif
endfor
```

Сложность этой процедуры: **O (n)** .

Помните, что если есть несколько графиков работы, которые могут дать нам максимальную прибыль, мы можем найти только один график работы с помощью этой процедуры.

Изменить расстояние

Оператор проблемы похож, если нам даны две строки str1 и str2, то сколько минимального количества операций может быть выполнено на str1, которое оно преобразует в str2.

Реализация на Java

```
public class EditDistance {

public static void main(String[] args) {
    // TODO Auto-generated method stub
    String str1 = "march";
    String str2 = "cart";

    EditDistance ed = new EditDistance();
    System.out.println(ed.getMinConversions(str1, str2));
}

public int getMinConversions(String str1, String str2){
    int dp[][] = new int[str1.length()+1][str2.length()+1];
    for(int i=0;i<=str1.length();i++){
        for(int j=0;j<=str2.length();j++){
            if(i==0)
                dp[i][j] = j;
            else if(j==0)
                dp[i][j] = i;
            else if(str1.charAt(i-1) == str2.charAt(j-1))
                dp[i][j] = dp[i-1][j-1];
            else{
                dp[i][j] = 1 + Math.min(dp[i-1][j], Math.min(dp[i][j-1], dp[i-1][j-1]));
            }
        }
    }
    return dp[str1.length()][str2.length()];
}
```

}

Выход

```
3
```

Самая длинная общая подпоследовательность

Если нам заданы две строки, мы должны найти самую длинную общую подпоследовательность, присутствующую в обеих из них.

пример

LCS для входных последовательностей «ABCDGH» и «AEDFHR» - это «ADH» длины 3.

LCS для входных последовательностей «AGGTAB» и «GXTXAYB» - это «GTAB» длины 4.

Реализация на Java

```
public class LCS {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        String str1 = "AGGTAB";
```

```
        String str2 = "GXTXAYB";
        LCS obj = new LCS();
        System.out.println(obj.lcs(str1, str2, str1.length(), str2.length()));
        System.out.println(obj.lcs2(str1, str2));
    }

    //Recursive function
    public int lcs(String str1, String str2, int m, int n){
        if(m==0 || n==0)
            return 0;
        if(str1.charAt(m-1) == str2.charAt(n-1))
            return 1 + lcs(str1, str2, m-1, n-1);
        else
            return Math.max(lcs(str1, str2, m-1, n), lcs(str1, str2, m, n-1));
    }

    //Iterative function
    public int lcs2(String str1, String str2){
        int lcs[][] = new int[str1.length()+1][str2.length()+1];

        for(int i=0;i<=str1.length();i++){
            for(int j=0;j<=str2.length();j++){
                if(i==0 || j== 0){
                    lcs[i][j] = 0;
                }
                else if(str1.charAt(i-1) == str2.charAt(j-1)){
                    lcs[i][j] = 1 + lcs[i-1][j-1];
                }else{
                    lcs[i][j] = Math.max(lcs[i-1][j], lcs[i][j-1]);
                }
            }
        }

        return lcs[str1.length()][str2.length()];
    }

}
```

Выход

```
4
```

Число Фибоначчи

Нижний подход для печати n-го числа Фибоначчи с использованием динамического программирования.

Рекурсивное дерево

```
                        fib(5)
                     /         \
                fib(4)           fib(3)
              /      \           /    \
         fib(3)      fib(2)    fib(2)  fib(1)
        /    \       /   \     /   \
  fib(2)  fib(1) fib(1) fib(0) fib(1) fib(0)
```

```
  /    \
fib(1) fib(0)
```

Перекрывающиеся суб-проблемы

Здесь fib (0), fib (1) и fib (3) являются перекрывающимися подзадачами. Fib (0) повторяется 3 раза, fib (1) повторяется 5 раз, а fib (3) повторяется 2 раз.

Реализация

```
public int fib(int n){
        int f[] = new int[n+1];
        f[0]=0;f[1]=1;
        for(int i=2;i<=n;i++){
            f[i]=f[i-1]+f[i-2];
        }
        return f[n];
    }
```

Сложность времени

На)

Самая длинная общая подстрока

Учитывая 2 строки str1 и str2, мы должны найти длину самой длинной общей подстроки между ними.

Примеры

Вход: X = "abcdxyz", y = "xyzabcd" Выход: 4

Самая длинная общая подстрока - «abcd» и имеет длину 4.

Вход: X = "zxabcdezy", y = "yzabcdezx" Выход: 6

Самая длинная общая подстрока - «abcdez» и имеет длину 6.

Реализация на Java

```
public int getLongestCommonSubstring(String str1,String str2){
        int arr[][] = new int[str2.length()+1][str1.length()+1];
        int max = Integer.MIN_VALUE;
        for(int i=1;i<=str2.length();i++){
            for(int j=1;j<=str1.length();j++){
                if(str1.charAt(j-1) == str2.charAt(i-1)){
                    arr[i][j] = arr[i-1][j-1]+1;
                    if(arr[i][j]>max)
                        max = arr[i][j];
                }
                else
                    arr[i][j] = 0;
```

```
            }
        }
        return max;
    }
```

Сложность времени

O (т * п)

Прочитайте Динамическое программирование онлайн:
https://riptutorial.com/ru/algorithm/topic/2345/динамическое-программирование

глава 28: Жадные алгоритмы

замечания

Жадный алгоритм - это алгоритм, в котором на каждом шаге мы выбираем наиболее выгодный вариант на каждом шаге, не заглядывая в будущее. Выбор зависит только от текущей прибыли.

Жадный подход, как правило, является хорошим подходом, когда каждая прибыль может быть поднята на каждом шагу, поэтому ни один выбор не блокирует другой.

Examples

Проблема с непрерывным рюкзаком

Приведенные предметы как `(value, weight)` нам нужно поместить их в рюкзак (контейнер) емкости `k` . Заметка! Мы можем сломать элементы, чтобы максимизировать ценность!

Пример ввода:

```
values[] = [1, 4, 5, 2, 10]
weights[] = [3, 2, 1, 2, 4]
k = 8
```

Ожидаемый результат:

```
maximumValueOfItemsInK = 20;
```

Алгоритм:

```
1) Sort values and weights by value/weight.
   values[] = [5, 10, 4, 2, 1]
   weights[] = [1, 4, 2, 2, 3]
2) currentWeight = 0; currentValue = 0;
3) FOR i = 0; currentWeight < k && i < values.length; i++ DO:
       IF k - currentWeight < weights[i] DO
           currentValue = currentValue + values[i];
           currentWeight = currentWeight + weights[i];
       ELSE
           currentValue = currentValue + values[i]*(k - currentWeight)/weights[i]
           currentWeight = currentWeight + weights[i]*(k - currentWeight)/weights[i]
       END_IF
   END_FOR
   PRINT "maximumValueOfItemsInK = " + currentValue;
```

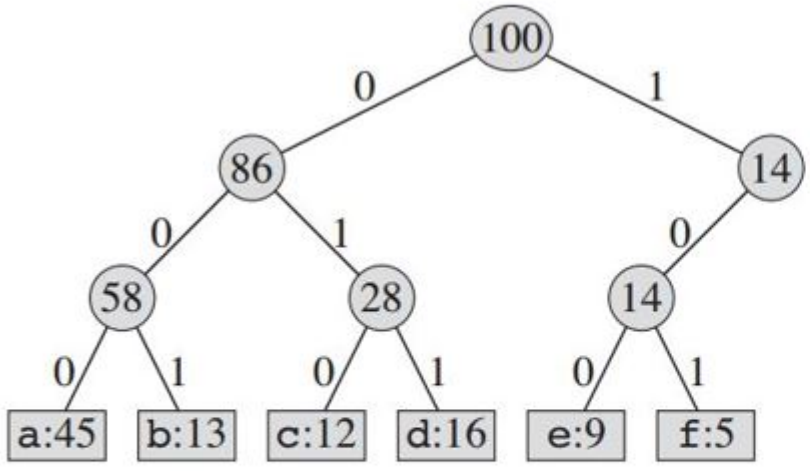
Кодирование Хаффмана

Код Хаффмана - это особый тип оптимального префиксного кода, который обычно используется для сжатия данных без потерь. Он сжимает данные, очень эффективно экономя от 20% до 90% памяти, в зависимости от характеристик сжатых данных. Мы рассматриваем данные как последовательность символов. Жадный алгоритм Хаффмана использует таблицу, показывающую, как часто каждый символ (т.е. его частота) генерирует оптимальный способ представления каждого символа в виде двоичной строки. Код Хаффмана был предложен Дэвидом А. Хаффманом в 1951 году.

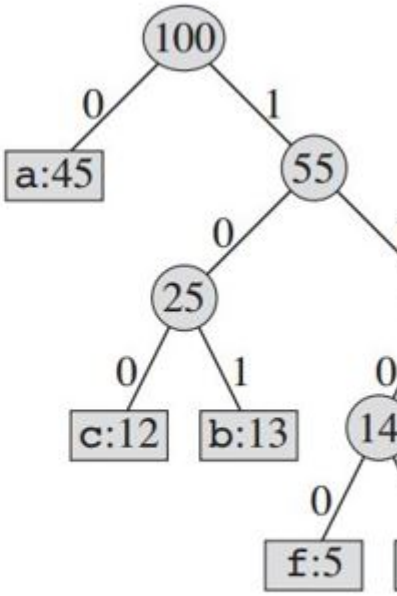
Предположим, у нас есть файл данных размером 100 000 символов, который мы хотим сохранить компактно. Мы предполагаем, что в этом файле всего 6 разных символов. Частота символов задается:

Character	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

У нас есть много вариантов, как представлять такой файл информации. Здесь мы рассмотрим проблему проектирования *двоичного символьного кода,* в котором каждый символ представлен уникальной двоичной строкой, которую мы называем **кодовым словом** .



Fixed-length Codeword



Variable-length Codev

Построенное дерево предоставит нам:

Character	a	b	c	d	e	f
Fixed-length Codeword	000	001	010	011	100	101

```
+------------------------+-----+-----+-----+-----+-----+-----+
|Variable-length Codeword|  0  | 101 | 100 | 111 | 1101| 1100|
+------------------------+-----+-----+-----+-----+-----+-----+
```

Если мы используем **код фиксированной длины** , нам нужны три бита для представления 6 символов. Этот метод требует 300 000 бит для кодирования всего файла. Теперь вопрос в том, можем ли мы сделать лучше?

Код переменной длины может значительно превосходить код фиксированной длины, предоставляя частые символы коротких кодовых слов и нечетные символы длинными кодовыми словами. Этот код требует: (45 X 1 + 13 X 3 + 12 X 3 + 16 X 3 + 9 X 4 + 5 X 4) X 1000 = 224000 бит для представления файла, что экономит примерно 25% памяти.

Одна вещь, которую следует помнить, мы рассматриваем здесь только коды, в которых кодовое слово не является также префиксом некоторого другого кодового слова. Они называются *префиксными кодами.* Для кодирования с переменной длиной мы кодируем 3-символьный файл *abc* как 0.101.100 = 0101100, где «.» обозначает конкатенацию.

Префиксные коды желательны, потому что они упрощают декодирование. Так как никакое кодовое слово не является префиксом для любого другого, кодовое слово, которое начинает кодированный файл, недвусмысленно. Мы можем просто определить исходное кодовое слово, перевести его обратно на исходный символ и повторить процесс декодирования в остальной части закодированного файла. Например, 001011101 анализирует однозначно как 0.0.101.1101, который декодирует до *aabe* . Короче говоря, все комбинации двоичных представлений уникальны. Скажем, например, если одна буква обозначена символом 110, никакая другая буква не будет обозначена 1101 или 1100. Это связано с тем, что вы можете столкнуться с путаницей в том, следует ли выбирать 110 или продолжать конкатенировать следующий бит и выбирать его.

Техника сжатия:

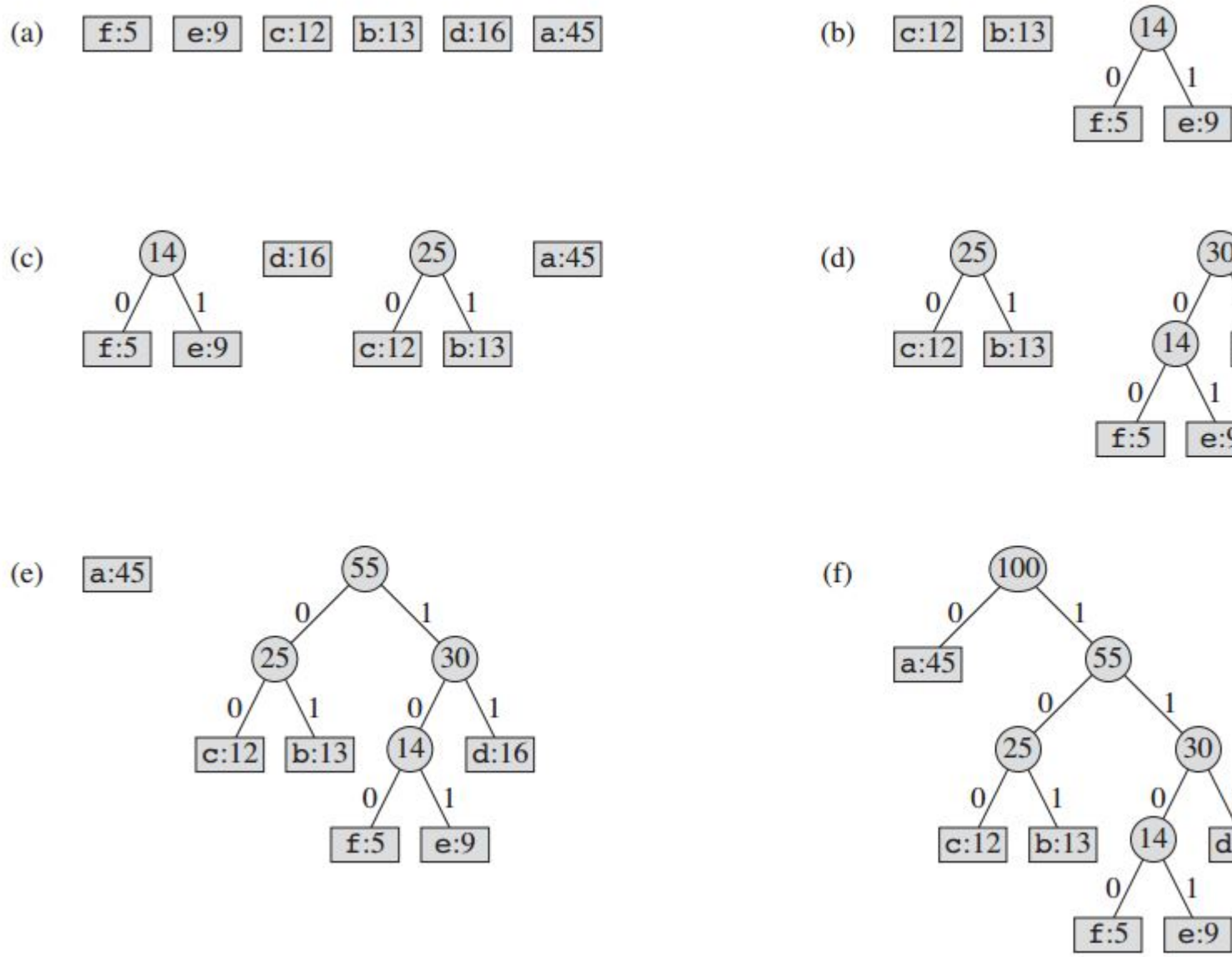
Эта технология работает, создавая *двоичное дерево* узлов. Они могут храниться в регулярном массиве, размер которого зависит от количества символов, **n** . Узел может быть либо *листовым,* либо *внутренним узлом* . Первоначально все узлы являются листовыми узлами, которые содержат сам символ, его частоту и, необязательно, ссылку на его дочерние узлы. В качестве условного обозначения бит «0» представляет собой левый дочерний элемент, а бит «1» представляет собой правильный ребенок. *Приоритетная очередь* используется для хранения узлов, которая обеспечивает узел с наименьшей частотой при выталкивании. Процесс описан ниже:

1. Создайте листовой узел для каждого символа и добавьте его в очередь приоритетов.
2. Хотя в очереди находится более одного узла:
    1. Удалите из очереди два узла с наивысшим приоритетом.
    2. Создайте новый внутренний узел с этими двумя узлами как дочерние и с частотой, равной сумме частоты двух узлов.

3. Добавьте новый узел в очередь.

3. Остальным узлом является корневой узел, а дерево Хаффмана завершено.

Для нашего примера:



Псевдокод выглядит так:

```
Procedure Huffman(C):     // C is the set of n characters and related information
n = C.size
Q = priority_queue()
for i = 1 to n
    n = node(C[i])
    Q.push(n)
end for
while Q.size() is not equal to 1
    Z = new node()
    Z.left = x = Q.pop
    Z.right = y = Q.pop
    Z.frequency = x.frequency + y.frequency
    Q.push(Z)
end while
```

```
Return Q
```

Хотя заданный для линейного времени отсортированный вход, в общем случае произвольного ввода, с использованием этого алгоритма требует предварительной сортировки. Таким образом, поскольку сортировка занимает **O (nlogn)** время в общих случаях, оба метода имеют одинаковую сложность.

Поскольку **n** здесь - количество символов в алфавите, которое обычно является очень маленьким числом (по сравнению с длиной сообщения, которое должно быть закодировано), сложность времени не очень важна при выборе этого алгоритма.

Техника декомпрессии:

Процесс декомпрессии - это просто перевод текста префиксных кодов на индивидуальное значение байта, обычно путем перемещения узла дерева Хаффмана по узлу, поскольку каждый бит считывается из входного потока. Достижение листового узла обязательно завершает поиск этого конкретного байтового значения. Значение листа представляет желаемый символ. Обычно дерево Хаффмана построено с использованием статистически скорректированных данных по каждому циклу сжатия, поэтому восстановление довольно просто. В противном случае информация для восстановления дерева должна быть отправлена отдельно. Псевдокод:

```
Procedure HuffmanDecompression(root, S):   // root represents the root of Huffman Tree
n := S.length                              // S refers to bit-stream to be decompressed
for i := 1 to n
    current = root
    while current.left != NULL and current.right != NULL
        if S[i] is equal to '0'
            current := current.left
        else
            current := current.right
        endif
        i := i+1
    endwhile
    print current.symbol
endfor
```

Жадный Пояснение:
Кодирование Хаффмана рассматривает появление каждого символа и сохраняет его как двоичную строку оптимальным образом. Идея состоит в том, чтобы назначать коды переменной длины для ввода входных символов, длина назначенных кодов основана на частотах соответствующих символов. Мы создаем двоичное дерево и работаем на нем снизу вверх, чтобы наименьшие два частых символа были как можно дальше от корня. Таким образом, самый частый персонаж получает наименьший код, а наименее частый персонаж получает самый большой код.

Рекомендации:

- Введение в алгоритмы - Чарльз Э. Лейзерсон, Клиффорд Штайн, Рональд Ривест и

Томас Х. Кормен
- Кодирование Хаффмана - Википедия
- Дискретная математика и ее приложения - Кеннет Х. Розен

Проблема изменения

Учитывая денежную систему, можно ли предоставить количество монет и как найти минимальный набор монет, соответствующий этой сумме.

Канонические денежные системы. Для некоторой денежной системы, такой как те, которые мы используем в реальной жизни, «интуитивное» решение работает отлично. Например, если различные монеты евро и счета (за исключением центов) составляют 1 €, 2 €, 5 €, 10 €, давая самую высокую монету или счет, пока мы не достигнем суммы, и повторение этой процедуры приведет к минимальному набору монет ,

Мы можем сделать это рекурсивно с OCaml:

```
(* assuming the money system is sorted in decreasing order *)
let change_make money_system amount =
  let rec loop given amount =
    if amount = 0 then given
    else
      (* we find the first value smaller or equal to the remaining amount *)
      let coin = List.find ((>=) amount) money_system in
      loop (coin::given) (amount - coin)
  in loop [] amount
```

Эти системы созданы таким образом, что изменение легко. Проблема становится сложнее, когда речь идет о произвольной денежной системе.

Общий случай. Как дать 99 € с монетами 10 €, 7 € и 5 €? Здесь, давая монеты в 10 €, пока мы не останемся с 9 €, очевидно, не будет решения. Хуже того, что решение может не существовать. Эта проблема на самом деле является np-hard, но приемлемые решения, связанные с **жадностью** и **памятью,** существуют. Идея состоит в том, чтобы исследовать все возможности и выбрать тот, у которого минимальное количество монет.

Чтобы дать сумму X> 0, мы выберем часть P в денежной системе, а затем решим подзапрос, соответствующий XP. Мы пробуем это для всех частей системы. Решение, если оно существует, является наименьшим путем, который привел к 0.

Здесь рекурсивная функция OCaml, соответствующая этому методу. Он возвращает None, если решение не существует.

```
(* option utilities *)
let optmin x y =
  match x,y with
  | None,a | a,None -> a
  | Some x, Some y-> Some (min x y)
```

```
let optsucc = function
  | Some x -> Some (x+1)
  | None -> None

(* Change-making problem*)
let change_make money_system amount =
  let rec loop n =
    let onepiece acc piece =
      match n - piece with
      | 0 -> (*problem solved with one coin*)
             Some 1
      | x -> if x < 0 then
               (*we don't reach 0, we discard this solution*)
               None
             else
               (*we search the smallest path different to None with the remaining pieces*)
               optmin (optsucc (loop x)) acc
    in
    (*we call onepiece forall the pieces*)
    List.fold_left onepiece None money_system
  in loop amount
```

Примечание . Можно заметить, что эта процедура может вычислить несколько раз набор изменений для того же значения. На практике использование memoization во избежание повторений приводит к более быстрым (более быстрым) результатам.

Проблема выбора действия

Эта проблема

У вас есть набор действий (действий). Каждое действие имеет время начала и время окончания. Вы не можете выполнять более одного действия одновременно. Ваша задача - найти способ выполнения максимального количества действий.

Например, предположим, что у вас есть выбор классов на выбор.

№ мероприятия	начальное время	время окончания
1	10.20.	11:00 утра
2	10.30.	11:30 утра
3	11.00.	12:00 утра
4	10.00	11:30 утра
5	9.00.	11:00 утра

Помните, что вы не можете брать два класса одновременно. Это означает, что вы не можете брать классы 1 и 2, потому что они имеют общее время с 10.30 до 11.00. Однако вы

можете взять класс 1 и 3, потому что они не имеют общего времени. Поэтому ваша задача - максимально использовать максимальное количество классов без каких-либо совпадений. Как вы можете это сделать?

Анализ

Давайте подумаем о решении жадным подходом. Прежде всего мы случайно выбрали какой-то подход и проверим, что будет работать или нет.

- **сортируйте активность по времени начала,** что означает, что деятельность начинается сначала, мы сначала возьмем их. затем возьмите сначала последний из отсортированного списка и проверьте, будет ли он пересекаться с предыдущим занятым или нет. Если текущая активность не пересекается с ранее принятой деятельностью, мы будем выполнять эту операцию, иначе мы не выполним ее. этот подход будет работать для некоторых случаев, таких как

№ мероприятия	начальное время	время окончания
1	11.00.	1:30 вечера
2	11.30	12:00 вечера
3	13:30	2:00 вечера
4	10.00	11:00 утра

порядок сортировки будет 4 -> 1 -> 2 -> 3. Будет выполняться активность 4 -> 1 -> 3, и активность 2 будет пропущена. будет выполнено максимум 3 активности. Он работает для подобных случаев. но в некоторых случаях это не удастся. Давайте применим этот подход для случая

№ мероприятия	начальное время	время окончания
1	11.00.	1:30 вечера
2	11.30	12:00 вечера
3	13:30	2:00 вечера
4	10.00	3:00 вечера

Порядок сортировки будет 4 -> 1 -> 2 -> 3, и будет выполнено только действие 4, но ответ может быть активен 1 -> 3 или 2 -> 3. Поэтому наш подход не будет работать для вышеуказанного случая. Попробуем другой подход

- **Сортируйте активность по длительности,** которая означает самую короткую деятельность. что может решить предыдущую проблему. Хотя проблема не полностью решена. Есть еще некоторые случаи, которые могут привести к отказу в решении. примените этот подход на примере ниже.

№ мероприятия	начальное время	время окончания
1	6.00 утра	11:40 утра
2	11.30	12:00 вечера
3	11.40 вечера	2:00 вечера

если сортировать активность по времени, порядок сортировки будет 2 -> 3 ---> 1. и если мы сначала выполняем деятельность № 2, тогда никакая другая деятельность не может быть выполнена. Но ответ будет выполнять активность 1, а затем выполнить 3. Таким образом, мы можем выполнять максимум 2 действия. Таким образом, это не может быть решением этой проблемы. Мы должны попробовать другой подход.

Решение

- **Сортировка активности по окончанию времени,** что означает, что сначала завершается действие, которое начинается первым. алгоритм приведен ниже

    1. Сортируйте действия по времени окончания.
    2. Если выполняемая деятельность не имеет общего времени с ранее выполненными действиями, выполните эту операцию.

Давайте проанализируем первый пример

№ мероприятия	начальное время	время окончания
1	10.20.	11:00 утра
2	10.30.	11:30 утра
3	11.00.	12:00 утра
4	10.00	11:30 утра
5	9.00.	11:00 утра

Сортировка активности по времени ее окончания. Таким образом, порядок сортировки будет 1 -> 5 -> 2 -> 4 -> 3 .. ответ будет 1 -> 3, эти два действия будут выполнены. ans - это

ответ. вот код sudo.

1. сортировать: деятельность
2. выполнить первую операцию из отсортированного списка видов деятельности.
3. Set: Current_activity: = первая активность
4. set: end_time: = end_time текущей активности
5. перейти к следующему действию, если оно существует, если оно не существует.
6. если start_time текущей активности <= end_time: выполните действие и перейдите к 4
7. еще: добрался до 5.

см. здесь для справки по кодированию http://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/

Прочитайте Жадные алгоритмы онлайн: https://riptutorial.com/ru/algorithm/topic/3140/жадные-алгоритмы

глава 29: Изменение динамического алгоритма расстояния

Examples

Минимальные изменения, необходимые для преобразования строки 1 в строку 2

Оператор проблемы похож, если нам даны две строки str1 и str2, а затем сколько минимального количества операций может быть выполнено на str1, которое оно преобразует в str2.The Операции могут быть:

1. **Вставить**
2. **Удалить**
3. **замещать**

Например

```
Input: str1 = "geek", str2 = "gesek"
Output: 1
We only need to insert s in first string

Input: str1 = "march", str2 = "cart"
Output: 3
We need to replace m with c and remove character c and then replace h with t
```

Для решения этой проблемы мы будем использовать 2D-массив dp [n + 1] [m + 1], где n - длина первой строки, а m - длина второй строки. Для нашего примера, если str1 является **azcef,** а str2 является **abcdef,** тогда наш массив будет dp [6] [7], и наш окончательный ответ будет сохранен в dp [5] [6].

```
          (a)  (b)  (c)  (d)  (e)  (f)
     +---+---+---+---+---+---+---+
     | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
     +---+---+---+---+---+---+---+
(a)  | 1 |   |   |   |   |   |   |
     +---+---+---+---+---+---+---+
(z)  | 2 |   |   |   |   |   |   |
     +---+---+---+---+---+---+---+
(c)  | 3 |   |   |   |   |   |   |
     +---+---+---+---+---+---+---+
(e)  | 4 |   |   |   |   |   |   |
     +---+---+---+---+---+---+---+
(f)  | 5 |   |   |   |   |   |   |
     +---+---+---+---+---+---+---+
```

Для **dp [1] [1]** мы должны проверить, что мы можем сделать, чтобы преобразовать **a** в **a.**

Это будет **0.** Для **dp [1] [2]** мы должны проверить, что мы можем сделать, чтобы преобразовать **a** в **ab** .It будет **1,** потому что мы должны **вставить b.** После первой итерации наш массив будет выглядеть так:

```
          (a) (b) (c) (d) (e) (f)
     +---+---+---+---+---+---+---+
     | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
     +---+---+---+---+---+---+---+
  (a)| 1 | 0 | 1 | 2 | 3 | 4 | 5 |
     +---+---+---+---+---+---+---+
  (z)| 2 |   |   |   |   |   |   |
     +---+---+---+---+---+---+---+
  (c)| 3 |   |   |   |   |   |   |
     +---+---+---+---+---+---+---+
  (e)| 4 |   |   |   |   |   |   |
     +---+---+---+---+---+---+---+
  (f)| 5 |   |   |   |   |   |   |
     +---+---+---+---+---+---+---+
```

Для итерации 2

Для **dp [2] [1]** мы должны проверить, что для преобразования **az** в **a** нам нужно удалить **z** , поэтому **dp [2] [1]** будет **1.** Симологично для **dp [2] [2]** нам нужно заменить **z** с **b** , поэтому **dp [2] [2]** будет **1.** После второй итерации будет выглядеть массив **dp []** .

```
          (a) (b) (c) (d) (e) (f)
     +---+---+---+---+---+---+---+
     | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
     +---+---+---+---+---+---+---+
  (a)| 1 | 0 | 1 | 2 | 3 | 4 | 5 |
     +---+---+---+---+---+---+---+
  (z)| 2 | 1 | 1 | 2 | 3 | 4 | 5 |
     +---+---+---+---+---+---+---+
  (c)| 3 |   |   |   |   |   |   |
     +---+---+---+---+---+---+---+
  (e)| 4 |   |   |   |   |   |   |
     +---+---+---+---+---+---+---+
  (f)| 5 |   |   |   |   |   |   |
     +---+---+---+---+---+---+---+
```

Итак, наша **формула** будет выглядеть так:

```
if characters are same
    dp[i][j] = dp[i-1][j-1];
else
    dp[i][j] = 1 + Min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])
```

После последней итерации наш массив dp [] будет выглядеть так:

```
          (a) (b) (c) (d) (e) (f)
     +---+---+---+---+---+---+---+
     | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
     +---+---+---+---+---+---+---+
  (a)| 1 | 0 | 1 | 2 | 3 | 4 | 5 |
```

```
     +---+---+---+---+---+---+---+
  (z)| 2 | 1 | 1 | 2 | 3 | 4 | 5 |
     +---+---+---+---+---+---+---+
  (c)| 3 | 2 | 2 | 1 | 2 | 3 | 4 |
     +---+---+---+---+---+---+---+
  (e)| 4 | 3 | 3 | 2 | 2 | 2 | 3 |
     +---+---+---+---+---+---+---+
  (f)| 5 | 4 | 4 | 2 | 3 | 3 | 3 |
     +---+---+---+---+---+---+---+
```

Реализация на Java

```
public int getMinConversions(String str1, String str2){
    int dp[][] = new int[str1.length()+1][str2.length()+1];
    for(int i=0;i<=str1.length();i++){
        for(int j=0;j<=str2.length();j++){
            if(i==0)
                dp[i][j] = j;
            else if(j==0)
                dp[i][j] = i;
            else if(str1.charAt(i-1) == str2.charAt(j-1))
                dp[i][j] = dp[i-1][j-1];
            else{
                dp[i][j] = 1 + Math.min(dp[i-1][j], Math.min(dp[i][j-1], dp[i-1][j-1]));
            }
        }
    }
    return dp[str1.length()][str2.length()];
}
```

Сложность времени

```
O(n^2)
```

Прочитайте Изменение динамического алгоритма расстояния онлайн: https://riptutorial.com/ru/algorithm/topic/10500/изменение-динамического-алгоритма-расстояния

глава 30: Интернет-алгоритмы

замечания

теория

Определение 1: Задача оптимизации П состоит из множества **экземпляров** Σ_{Π} . Для каждого экземпляра $\sigma \in \Sigma_{\Pi}$ существует множество Z_{σ} **решений** и **целевая функция** $f_{\sigma} : Z_{\sigma} \rightarrow \Re_{\geq 0}$, которая присваивает апо- зитивное вещественное значение каждому решению. Скажем, OPT (σ) является значением оптимального решения, A (σ) является решением Алгоритма A для задачи П и $w_A(\sigma) = f_{\sigma}(A(\sigma))$ его значение.

Определение 2: Онлайн-алгоритм A для задачи минимизации П имеет **конкурентное отношение** $r \geq 1$, если существует постоянная $\tau \in \Re$ с

$$w_A(\sigma) = f_{\sigma}(A(\sigma)) \leq r \cdot \mathrm{OPT}(\&\ \mathrm{sigma}) + \tau$$

для всех экземпляров $\sigma \in \Sigma_{\Pi}$. A называется **r-конкурентным** онлайн-алгоритмом. Даже

$$w_A(\sigma) \leq r \cdot \mathrm{OPT}(\&\ \text{сигма})$$

для всех экземпляров $\sigma \in \Sigma_{\Pi}$, тогда A называется **строго r-конкурентным** онлайн-алгоритмом.

Предложение 1.3: LRU и **FWF** - это алгоритм маркировки.

Доказательство. В начале каждой фазы (кроме первой) **FWF** имеет **пропущенную** кэш и очищает кеш. это означает, что у нас есть пустые страницы. В каждой фазе запрашивается максимальное количество k различных страниц, поэтому на этапе будет выселение. Таким образом, **FWF** является алгоритмом маркировки.
Предположим, что **LRU** не является алгоритмом маркировки. Тогда есть экземпляр σ, где **LRU** - отмеченная страница x на фазе i, выселенная. Пусть σ_t запрос в фазе i, где x выселяется. Так как x отмечен, в той же фазе должен быть более ранний запрос σ_{t^*} для x, поэтому $t^* < t$. После того, как t^* x является последней страницей кэшей, поэтому для выселения в t последовательность $\sigma_{t^*+1}, \ldots, \sigma_t$ должна запрашивать не менее k из x разных страниц. Это означает, что фаза i запросила не менее k + 1 разных страниц, что противоречит определению фазы. Поэтому **LRU** должен быть алгоритмом маркировки.

Предложение 1.4: Каждый маркерный алгоритм **строго k-конкурентный** .

Доказательство. Пусть σ - пример для проблемы поискового вызова и l число фаз для σ. L = 1, то каждый алгоритм маркировки оптимален, и оптимальный автономный алгоритм не может быть лучше.

Предположим, что l ≥ 2. стоимость каждого алгоритма маркировки, например, σ, ограничена сверху $l \cdot k$, так как в каждой фазе алгоритм маркировки не может выселить более k страниц, не вытесняя одну отмеченную страницу.
Теперь мы попытаемся показать, что оптимальный автономный алгоритм выдает по крайней мере k + l-2 страницы для σ, k в первой фазе и по крайней мере один для каждой следующей фазы, за исключением последней. Для доказательства можно определить l-2 дизъюнктных подпоследовательностей σ. Подпоследовательность i ∈ {1, ..., l-2} начинается со второго положения фазы i + 1 и заканчивается первой позицией фазы i + 2.
Пусть x - первая страница фазы i + 1. В начале подпоследовательности i есть страница x и не более k-1 разных страниц в кеше оптимальных автономных алгоритмов. В подпоследовательности i есть запрос страницы k, отличный от x, поэтому оптимальный автономный алгоритм должен вытеснять по крайней мере одну страницу для каждой подпоследовательности. Поскольку на этапе 1 кеш все еще пуст, оптимальный автономный алгоритм вызывает k выселений в течение первой фазы. Это показывает, что

$$w_A(\sigma) \leq l \cdot k \leq (k + l\text{-}2)\, k \leq \mathrm{OPT}(\sigma) \cdot k$$

Следствие 1.5: LRU и **FWF** являются **строго k-конкурентными** .

Нет ли константы r, для которой онлайн-алгоритм A является r-конкурентным, мы называем A **неконкурентоспособным** .

Предложение 1.6: LFU и **LIFO не** являются **конкурентоспособными** .

Доказательство. Пусть l ≥ 2 константа, k ≥ 2 размер кеша. Различные страницы кеша имеют нулевое значение 1, ..., k + 1. Мы рассмотрим следующую последовательность:

Первая страница 1 запрашивается l раз, чем страница 2, и так одна. В конце есть (l-1) чередующиеся запросы для страниц k и k + 1.
LFU и **LIFO** заполняют свой кеш страницами 1-k. Когда страница k + 1 запрашивается, страница k вызывается и наоборот. Это означает, что каждый запрос подпоследовательности $(k, k + 1)^{l-1}$ высылает одну страницу. Кроме того, они пропускают кеш-ке в первый раз для страниц 1- (k-1). Таким образом, **LFU** и **LIFO** выселяют точные k-1 + 2 (l-1) страницы.
Теперь мы должны показать, что для любой константы $\tau \in \Re$ и любого константа r ≤ 1 существует l, так что

которая равна

Чтобы удовлетворить этому неравенству, вам просто нужно выбрать I достаточно большой. Таким образом, **LFU** и **LIFO** не являются конкурентными.

Предложение 1.7: Не **существует r-конкурентного** детерминированного онлайн-алгоритма для пейджинга с **r <k** .

источники

Основной материал

1. Скриптовые онлайн-алгоритмы (немецкий язык), Хейко Роглин, Университет Бонн
2. Алгоритм замены страницы

Дальнейшее чтение

1. Онлайн-расчет и конкурентный анализ Аллан Бородин и Ран Эль-Янов

Исходный код

1. Исходный код для автономного кэширования
2. Исходный код для игры противников

Examples

Пейджинг (онлайн-кэширование)

Предисловие

Вместо того, чтобы начинать с формального определения, цель состоит в том, чтобы подходить к этой теме через ряд примеров, вводя определения на этом пути. Раздел замечания **Теория** будет состоять из всех определений, теорем и предложений , чтобы дать вам всю информацию , чтобы быстрее искать конкретные аспекты.

Источники разделов примечаний состоят из базового материала, используемого для этой темы, и дополнительной информации для дальнейшего чтения. Кроме того, вы найдете полные исходные коды для примеров. Пожалуйста, обратите внимание, что чтобы исходный код для примеров был более читабельным и короче, он воздерживается от таких вещей, как обработка ошибок и т. Д. Он также передает некоторые специфические языковые функции, которые скрывали бы ясность примера, например, широкое использование передовых библиотек и т. Д.

Paging

Проблема поискового вызова возникает из-за ограничения конечного пространства. Предположим, что наш кеш `C` имеет `k` страниц. Теперь мы хотим обработать последовательность из `m` запросов страниц, которые должны были быть помещены в кеш, прежде чем они будут обработаны. Конечно, если `m<=k` мы просто поместим все элементы в кеш, и он будет работать, но обычно это `m>>k` .

Мы говорим, что запрос является **удалением кеша** , когда страница уже находится в кеше, иначе ее называют **пропуском кеша** . В этом случае мы должны привести запрошенную страницу в кеш и выселить другую, если кеш будет заполнен. Цель - график выселения, который **минимизирует количество выселений** .

Существует множество стратегий для этой проблемы, давайте рассмотрим некоторые:

1. **Первое, сначала (FIFO)** : старейшая страница выселяется
2. **Последний, первый (LIFO)** : самая новая страница выселяется
3. **Наименее недавно использованная (LRU)** : страница с выездом, последний доступ которой был самым ранним
4. **Наименее часто используемая (LFU)** : страница с наименьшим количеством запросов
5. **Самое длинное прямое расстояние (LFD)** : вырезать страницу в кеше, которая не запрашивается дольше в будущем.
6. **Сброс при заполнении (FWF)** : очистите кеш, как только **произойдет сбой** кеша

Существует два способа решения этой проблемы:

1. **offline** : последовательность запросов страницы известна заранее
2. **онлайн** : последовательность запросов страницы не известна заранее

Автономный подход

Для первого подхода рассмотрите тему « Применения жадности» . Это третий пример. **Оффлайн Кэширование** рассматривает первые пять стратегий сверху и дает вам хорошую точку входа для следующего.

Пример программы был расширен с **помощью** стратегии **FWF** :

```
class FWF : public Strategy {
public:
    FWF() : Strategy("FWF")
    {
    }

    int apply(int requestIndex) override
    {
        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
```

```
                return i;

            // after first empty page all others have to be empty
            else if(cache[i] == emptyPage)
                return i;
        }

        // no free pages
        return 0;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {

        // no pages free -> miss -> clear cache
        if(cacheMiss && cachePos == 0)
        {
            for(int i = 1; i < cacheSize; ++i)
                cache[i] = emptyPage;
        }
    }
};
```

Полный исходный код доступен здесь . Если мы повторно используем пример из этой темы, мы получаем следующий результат:

```
Strategy: FWF

Cache initial: (a,b,c)

Request cache 0 cache 1 cache 2 cache miss
  a       a       b       c
  a       a       b       c
  d       d       X       X       x
  e       d       e       X
  b       d       e       b
  b       d       e       b
  a       a       X       X       x
  c       a       c       X
  f       a       c       f
  d       d       X       X       x
  e       d       e       X
  a       d       e       a
  f       f       X       X       x
  b       f       b       X
  e       f       b       e
  c       c       X       X       x

Total cache misses: 5
```

Несмотря на то, что **LFD** является оптимальным, **FWF** имеет меньше промахов в кэше. Но главная цель заключалась в том, чтобы свести к минимуму количество выселений, а для **FWF** пять промахов означают 15 выселений, что делает его самым бедным выбором для этого примера.

Интернет-подход

Теперь мы хотим подойти к онлайн-проблеме пейджинга. Но сначала нам нужно понять, как это сделать. Очевидно, онлайн-алгоритм не может быть лучше, чем оптимальный автономный алгоритм. Но насколько это хуже? Нам нужны формальные определения для ответа на этот вопрос:

Определение 1.1. Задача оптимизации П состоит из множества **экземпляров** $\Sigma_{П}$. Для каждого экземпляра $\sigma \in \Sigma_{П}$ существует множество Z_{σ} **решений** и **целевая функция** $f_{\sigma} : Z_{\sigma} \to \Re_{\geq 0}$, которая присваивает апо- зитивное вещественное значение каждому решению. Скажем, OPT (σ) является значением оптимального решения, A (σ) является решением Алгоритма A для задачи П и $w_A(\sigma) = f_{\sigma}(A(\sigma))$ его значение.

Определение 1.2. Онлайн-алгоритм A для задачи минимизации П имеет **конкурентное отношение** $r \geq 1$, если существует постоянная $\tau \in \Re$ с

$$w_A(\sigma) = f_{\sigma}(A(\sigma)) \leq r \cdot OPT(\sigma) + \tau$$

для всех экземпляров $\sigma \in \Sigma_{П}$. A называется **r-конкурентным** онлайн-алгоритмом. Даже

$$w_A(\sigma) \leq r \cdot OPT(\sigma)$$

для всех экземпляров $\sigma \in \Sigma_{П,}$ тогда A называется **строго r-конкурентным** онлайн-алгоритмом.

Поэтому вопрос заключается в том, насколько **конкурентоспособным** является наш онлайн-алгоритм по сравнению с оптимальным автономным алгоритмом. В своей знаменитой книге Аллан Бородин и Ран Эль-Янив использовали другой сценарий для описания ситуации онлайн-пейджинга:

Существует **злой противник,** который знает ваш алгоритм и оптимальный автономный алгоритм. На каждом шагу он пытается запросить страницу, которая для вас хуже всего, и одновременно лучше подходит для автономного алгоритма. **конкурентный фактор** вашего алгоритма - это фактор того, насколько плохо ваш алгоритм против оптимального автономного алгоритма противника. Если вы хотите попытаться стать противником, вы можете попробовать Противную игру (попытайтесь победить стратегии поискового вызова).

Алгоритмы маркировки

Вместо того, чтобы анализировать каждый алгоритм отдельно, давайте посмотрим на специальное семейство онлайн-алгоритмов для проблемы поискового вызова, называемой **алгоритмами маркировки** .

Пусть $\sigma = (\sigma_1, ..., \sigma_p)$ экземпляр для нашей задачи и k размер кеша, чем σ можно разделить на фазы:

- Фаза 1 - это максимальная подпоследовательность σ от начала до максимального k различных страниц
- Фаза i ≥ 2 - максимальная подпоследовательность σ от конца pase i-1 до максимального k различных страниц

Например, при k = 3:

Алгоритм маркировки (неявно или явно) поддерживает, отмечена ли страница или нет. В начале каждого этапа все страницы не отмечены. Является ли страница запрошенной в течение фазы, она становится отмеченной. Алгоритм является алгоритмом маркировки, **если** он никогда не высекает отмеченную страницу из кеша. Это означает, что страницы, которые используются во время фазы, не будут выселены.

Предложение 1.3: LRU и **FWF** - это алгоритм маркировки.

Доказательство. В начале каждой фазы (кроме первой) **FWF** имеет **пропущенную** кэш и очищает кеш. это означает, что у нас есть пустые страницы. В каждой фазе запрашивается максимальное количество k различных страниц, поэтому на этапе будет выселение. Таким образом, **FWF** является алгоритмом маркировки.
Предположим, что **LRU** не является алгоритмом маркировки. Тогда есть экземпляр σ, где **LRU** - отмеченная страница x на фазе i, выселенная. Пусть σ_t запрос в фазе i, где x выселяется. Так как x отмечен, в той же фазе должен быть более ранний запрос σ_{t^*} для x, поэтому t * <t. После того, как t * x является последней страницей кэшей, поэтому для выселения в t последовательность σ_{t^*+1} , ..., σ_t должна запрашивать не менее k из x разных страниц. Это означает, что фаза i запросила не менее k + 1 разных страниц, что противоречит определению фазы. Поэтому **LRU** должен быть алгоритмом маркировки.

Предложение 1.4: Каждый маркерный алгоритм **строго k-конкурентный** .

Доказательство. Пусть σ - пример для проблемы поискового вызова и l число фаз для σ. L = 1, то каждый алгоритм маркировки оптимален, и оптимальный автономный алгоритм не может быть лучше.
Предположим, что l ≥ 2. стоимость каждого алгоритма маркировки, например, σ ограничена сверху с $l \cdot k$, поскольку на каждой фазе алгоритм маркировки не может выселить более k страниц, не вытесняя одну отмеченную страницу.
Теперь мы попытаемся показать, что оптимальный автономный алгоритм выдает по крайней мере k + l-2 страницы для σ, k в первой фазе и по крайней мере один для каждой следующей фазы, за исключением последней. Для доказательства можно определить l-2 дизъюнктных подпоследовательностей σ. Подпоследовательность i ∈ {1, ..., l-2} начинается

со второго положения фазы i + 1 и заканчивается первой позицией фазы i + 2.
Пусть x - первая страница фазы i + 1. В начале подпоследовательности i есть страница x и не более k-1 разных страниц в кеше оптимальных автономных алгоритмов. В подпоследовательности i есть запрос страницы k, отличный от x, поэтому оптимальный автономный алгоритм должен вытеснять по крайней мере одну страницу для каждой подпоследовательности. Поскольку на этапе 1 кеш все еще пуст, оптимальный автономный алгоритм вызывает k выселений в течение первой фазы. Это показывает, что

$$w_A(\sigma) \leq l \cdot k \leq (k + l\text{-}2)\, k \leq \mathrm{OPT}(\sigma) \cdot k$$

Следствие 1.5: LRU и **FWF** являются **строго k-конкурентными** .

Упражнение: покажите, что **FIFO не** является алгоритмом маркировки, но **строго k-конкурентным** .

Нет ли константы r, для которой онлайн-алгоритм A является r-конкурентным, мы называем A **неконкурентоспособным**

Предложение 1.6: LFU и **LIFO не** являются **конкурентоспособными** .

Доказательство. Пусть l ≥ 2 константа, k ≥ 2 размер кеша. Различные страницы кеша имеют нулевое значение 1, ..., k + 1. Мы рассмотрим следующую последовательность:

Первая страница 1 запрашивается l раз, чем страница 2, и поэтому одна. В конце есть (l-1) чередующиеся запросы для страниц k и k + 1.
LFU и **LIFO** заполняют свой кеш страницами 1-k. Когда страница k + 1 запрашивается, страница k вызывается и наоборот. Это означает, что каждый запрос подпоследовательности $(k, k + 1)^{l-1}$ высылает одну страницу. Кроме того, их кешируют кэширование k-1 для использования в первый раз страниц 1- (k-1). Таким образом, **LFU** и **LIFO** выселяют точные k-1 + 2 (l-1) страницы.
Теперь мы должны показать, что для любой константы $\tau \in \Re$ и любой константы r ≤ 1 существует l, так что

которая равна

Чтобы удовлетворить этому неравенству, вам просто нужно выбрать l достаточно большой. Таким образом, **LFU** и **LIFO** не являются конкурентоспособными.

Предложение 1.7: Не **существует r-конкурентного** детерминированного онлайн-алгоритма для пейджинга с **r <k** .

Доказательство этого последнего предложения довольно длинное и основано на утверждении, что **LFD** является оптимальным автономным алгоритмом. Заинтересованный читатель может найти это в книге Бородина и Эль-Янива (см. Источники ниже).

Вопрос в том, можем ли мы сделать лучше. Для этого нам нужно оставить под собой детерминированный подход и начать рандомизировать наш алгоритм. Ясно, что гораздо труднее для противника наказать ваш алгоритм, если он рандомизирован.

Рандомизированный пейджинг будет обсуждаться в одном из следующих примеров ...

Прочитайте Интернет-алгоритмы онлайн: https://riptutorial.com/ru/algorithm/topic/8022/интернет-алгоритмы

глава 31: Кратчайшая общая проблема суперсимметрии

Examples

Кратчайшая общая проблема суперсимметрии Основная информация

Самая короткая общая суперпоследовательность - проблема, тесно связанная с самой длинной общей подпоследовательью, которую вы можете использовать в качестве внешней функции для этой задачи. Кратчайшая общая проблема суперпоследовательности - проблема, тесно связанная с самой длинной общей проблемой подпоследовательности.

Кратчайшая общая суперсимметрия (scs) является общей суперсимметрией минимальной длины. В кратчайшей общей проблеме суперсиммментации заданы две последовательности `X` и `Y` , и задача состоит в том, чтобы найти кратчайшую общую суперсимметрию этих последовательностей. В общем, scs не уникален.

Для двух последовательностей `X = <x1,...,xm>` и `Y = <y1,...,yn>` последовательность `U = <u1,...,uk>` является общей суперпоследовательностью `X` и `Y` если `U` - суперпоследовательность как `X` и `Y` Другими словами, кратчайшая общая суперпоследовательность строк `x` и `y` является кратчайшей строкой `z` такой, что `x` и `y` являются подпоследовательностями `z` .

Для двух входных последовательностей scs можно легко сформировать из самой длинной общей подпоследовательности (lcs). Например, если `X[1..m]=abcbdab` и `Y[1..n]=bdcaba` , `Y[1..n]=bdcaba` - `Z[1..r]=bcba` . Вставляя символы не-lcs при сохранении порядка символов, получаем scs: `U[1..t]=abdcabdab` .

Совершенно очевидно, что `r+t=m+n` для двух входных последовательностей. Однако для трех или более входных последовательностей это не выполняется. Заметим также, что проблемы lcs и scs не являются двойственными проблемами.

Для более общей проблемы нахождения строки `S` являющейся суперструной множества строк `S1,S2,...,Sl` , проблема NP-Complete. Также можно найти хорошие приближения для среднего случая, но не для наихудшего случая.

Пример кратчайшей общей проблемы суперсимметрии:

		Y---	G	X	T	X	A	Y	B
X	0	1	2	3	4	5	6	7
A	1	2	3	4	5	5	6	7
G	2	2	3	4	5	6	7	8
G	3	3	4	5	6	7	8	9
T	4	4	5	5	6	7	8	9
A	5	5	6	6	7	7	8	9
B	6	6	7	7	8	8	9	9

Сложность времени: `O(max(m,n))`

Реализация самой короткой общей проблемы суперсимметрии в C

```
public class ShortestCommonSupersequence
{
    private static int Max(int a, int b)
    {
        return a > b ? a : b;
    }

    private static int Lcs(string x, string y, int m, int n)
    {
        var l = new int[m + 1, n + 1];
        for (var i = 0; i <= m; i++)
        {
            for (var j = 0; j <= n; j++)
            {
                if (i == 0 || j == 0) l[i, j] = 0;
                else if (x[i - 1] == y[j - 1]) l[i, j] = l[i - 1, j - 1] + 1;
                else l[i, j] = Max(l[i - 1, j], l[i, j - 1]);
            }
        }
        return l[m, n];
    }

    private static int Scs(string x, string y)
    {
        int m = x.Length, n = y.Length;
        int l = Lcs(x, y, m, n);
        return m + n - l;
    }
```

```
    public static int Main(string x, string y)
    {
        return Scs(x, y);
    }
}
```

Прочитайте Кратчайшая общая проблема суперсимметрии онлайн:
https://riptutorial.com/ru/algorithm/topic/7604/кратчайшая-общая-проблема-суперсимметрии

глава 32: Куча сортировки

Examples

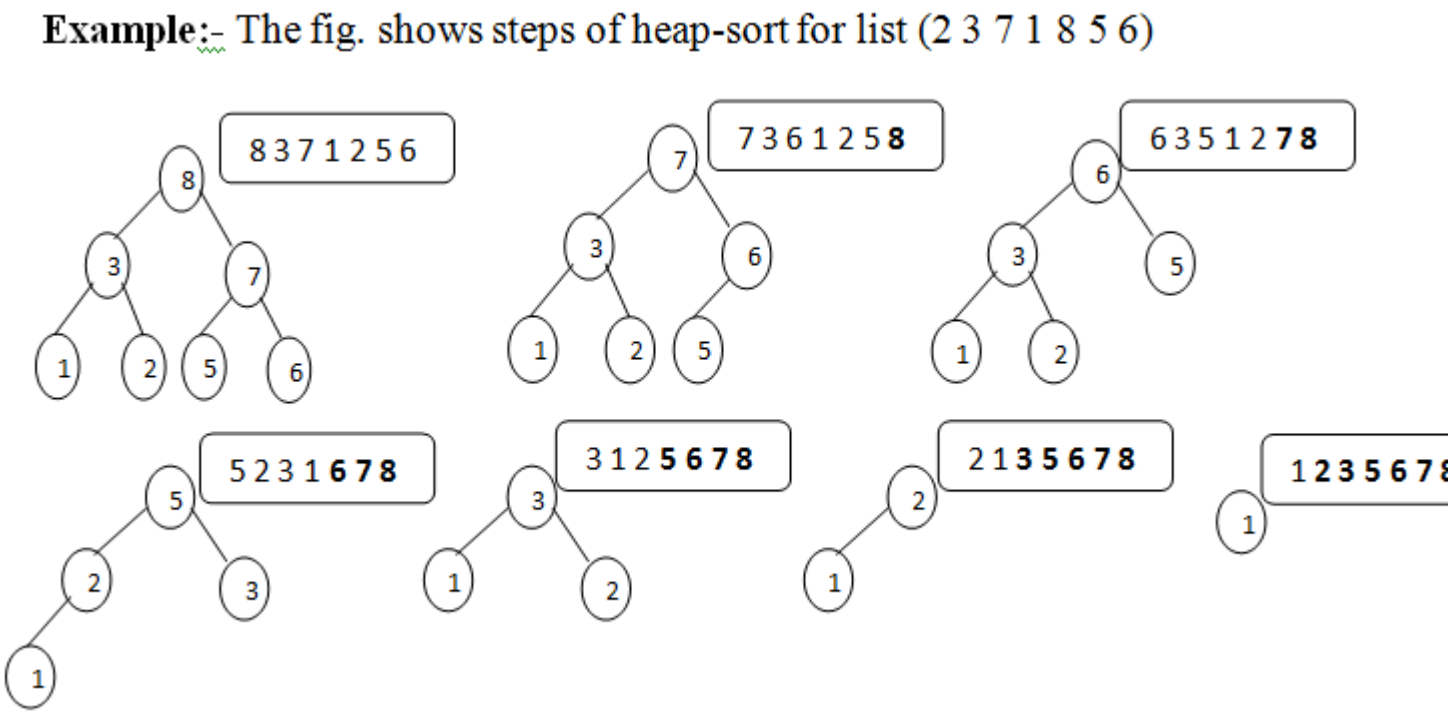
Heap Sort Базовая информация

Heap sort - это метод сортировки, основанный на сравнении, в структуре данных двоичной кучи. Он похож на сортировку выбора, в которой мы сначала находим максимальный элемент и помещаем его в конец структуры данных. Затем повторите тот же процесс для остальных предметов.

Псевдокод для кучи Сортировать:

```
function heapsort(input, count)
    heapify(a,count)
    end <- count - 1
    while end -> 0 do
    swap(a[end],a[0])
    end<-end-1
    restore(a, 0, end)

function heapify(a, count)
    start <- parent(count - 1)
    while start >= 0 do
        restore(a, start, count - 1)
        start <- start - 1
```

Пример сортировки кучи:



Вспомогательное пространство: `O(1)`
Сложность времени: `O(nlogn)`

Реализация C

```
public class HeapSort
{
    public static void Heapify(int[] input, int n, int i)
    {
        int largest = i;
        int l = i + 1;
        int r = i + 2;

        if (l < n && input[l] > input[largest])
            largest = l;

        if (r < n && input[r] > input[largest])
            largest = r;

        if (largest != i)
        {
            var temp = input[i];
            input[i] = input[largest];
            input[largest] = temp;
            Heapify(input, n, largest);
        }
    }

    public static void SortHeap(int[] input, int n)
    {
        for (var i = n - 1; i >= 0; i--)
        {
            Heapify(input, n, i);
        }
        for (int j = n - 1; j >= 0; j--)
        {
            var temp = input[0];
            input[0] = input[j];
            input[j] = temp;
            Heapify(input, j, 0);
        }
    }

    public static int[] Main(int[] input)
    {
        SortHeap(input, input.Length);
        return input;
    }
}
```

Прочитайте Куча сортировки онлайн: https://riptutorial.com/ru/algorithm/topic/7281/куча-сортировки

глава 33: Линейный алгоритм

Вступление

Линейный чертеж выполняется путем вычисления промежуточных положений вдоль пути линии между двумя заданными положениями конечных точек. Затем устройство вывода направляется для заполнения этих позиций между конечными точками.

Examples

Алгоритм рисования линии Bresenham

Теория фона: алгоритм линейного рисования Брешенема - эффективный и точный алгоритм генерации растровой линии, разработанный Брешенемом. Он включает только целочисленный расчет, поэтому он является точным и быстрым. Он также может быть расширен для отображения кругов других кривых.

В алгоритме рисования линии Bresenham:

Для Slope | m | <1:
Увеличивается либо значение x
ИЛИ оба x и y увеличены с использованием параметра решения.

Для склона | m |> 1:
Любое значение y увеличивается
ИЛИ оба x и y увеличены с использованием параметра решения.

Алгоритм для наклона | m | <1:

1. Введите две конечные точки (x1, y1) и (x2, y2) линии.

2. Выделите первую точку (x1, y1).

3. подсчитывать
   Delx = | x2 - x1 |
   Dely = | y2 - y1 |

4. Получить начальный параметр принятия решения как
   P = 2 * dely - delx

5. Для I = 0 до delx на шаге 1

   Если p <0, то
   X1 = x1 + 1

Пот (x1, y1)
P = p + 2 *dely*

еще
X1 = x1 + 1
Y1 = y1 + 1
Участок (x1, y1)
P = p + 2 dely - 2 * delx

Конец, если

Конец для

6. КОНЕЦ

Исходный код:

```
/* A C program to implement Bresenham line drawing algorithm for |m|<1 */
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>

int main()
{
 int gdriver=DETECT,gmode;
 int x1,y1,x2,y2,delx,dely,p,i;
 initgraph(&gdriver,&gmode,"c:\\TC\\BGI");

printf("Enter the intial points: ");
scanf("%d",&x1);
scanf("%d",&y1);
printf("Enter the end points: ");
scanf("%d",&x2);
scanf("%d",&y2);

putpixel(x1,y1,RED);

delx=fabs(x2-x1);
dely=fabs(y2-y1);
p=(2*dely)-delx;
for(i=0;i<delx;i++){
if(p<0)
{
 x1=x1+1;
 putpixel(x1,y1,RED);
 p=p+(2*dely);
}
else
{
 x1=x1+1;
 y1=y1+1;
 putpixel(x1,y1,RED);
 p=p+(2*dely)-(2*delx);
}
}
```

```
 getch();
 closegraph();
 return 0;
}
```

Алгоритм для наклона | m |> 1:

1. Введите две конечные точки (x1, y1) и (x2, y2) линии.
2. Выделите первую точку (x1, y1).
3. подсчитывать
   Delx = | x2 - x1 |
   Dely = | y2 - y1 |
4. Получить начальный параметр принятия решения как
   P = 2 * delx - dely
5. Для I = 0 к выполнению этапа 1
   Если p <0, то
   y1 = y1 + 1
   Пот (x1, y1)
   P = p + 2 *delx*

   еще
   X1 = x1 + 1
   Y1 = y1 + 1
   Участок (x1, y1)
   P = p + 2 delx - 2 * dely
   Конец, если
   Конец для
6. КОНЕЦ

Исходный код:

```
/* A C program to implement Bresenham line drawing algorithm for |m|>1 */
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>
int main()
{
int gdriver=DETECT,gmode;
int x1,y1,x2,y2,delx,dely,p,i;
initgraph(&gdriver,&gmode,"c:\\TC\\BGI");
printf("Enter the intial points: ");
scanf("%d",&x1);
scanf("%d",&y1);
printf("Enter the end points: ");
scanf("%d",&x2);
scanf("%d",&y2);
putpixel(x1,y1,RED);
delx=fabs(x2-x1);
dely=fabs(y2-y1);
p=(2*delx)-dely;
```

```
for(i=0;i<delx;i++){
if(p<0)
{
y1=y1+1;
putpixel(x1,y1,RED);
p=p+(2*delx);
}
else
{
x1=x1+1;
y1=y1+1;
putpixel(x1,y1,RED);
p=p+(2*delx)-(2*dely);
}
}
getch();
closegraph();
 return 0;
}
```

Прочитайте Линейный алгоритм онлайн: https://riptutorial.com/ru/algorithm/topic/7596/линейный-алгоритм

глава 34: Масленица

Examples

Основная информация о Pancake

Масленица - это разговорный термин для математической проблемы сортировки неупорядоченной стопки блинов в порядке размера, когда шпатель может быть вставлен в любую точку стека и используется, чтобы перевернуть все блины над ним. Число блинов - это минимальное количество флип, необходимых для данного количества блинов.

В отличие от традиционного алгоритма сортировки, который пытается сортировать с наименьшим количеством сравнений, цель состоит в том, чтобы отсортировать последовательность как можно меньше разворотов.

Идея состоит в том, чтобы сделать что-то похожее на Selection Sort. Мы по одному помещаем максимальный элемент в конце и уменьшаем размер текущего массива на единицу.

Расселение проблемы:

1. Необходимо заказать блины от самых маленьких (сверху) до самых больших (внизу), стартовый стек можно организовать в любом порядке.
2. Я могу выполнять флип, переворачивая весь стек.
3. Чтобы перевернуть конкретный блины на дно стека, сначала нужно перевернуть его вверх (затем перевернуть его на дно).
4. Чтобы заказать каждый блин, потребуется один флип вверх и один флип вниз до его окончательного местоположения.

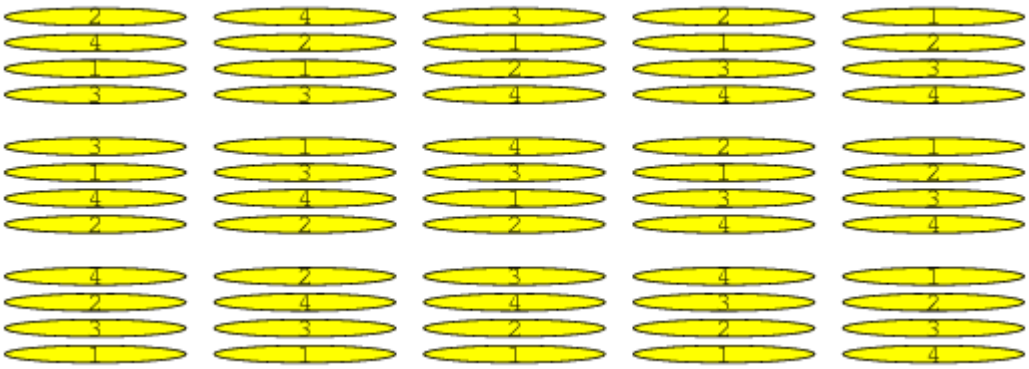
Интуитивный алгоритм:

1. Найдите самый крупный блайнд и переверните его на дно (вам может потребоваться сначала перевернуть его в верхнюю часть стека).

2. Повторите шаг 1 до тех пор, пока не будет упорядочен стек.

3. Вот и все, двухэтапный алгоритм будет работать.

Пример алгоритма сортировки блинов:



Вспомогательное пространство: `O(1)`
Сложность времени: `O(n^2)`

Реализация C

```
public class PancakeSort
{
    private static void SortPancake(int[] input, int n)
    {
        for (var bottom = n - 1; bottom > 0; bottom--)
        {
            var index = bottom;
            var maxIndex = input[bottom];
            int i;
            for (i = bottom - 1; i >= 0; i--)
            {
                if (input[i] > maxIndex)
                {
                    maxIndex = input[i];
                    index = i;
                }
            }
            if (index == bottom) continue;
            var temp = new int[n];
            var j = 0;
            for (i = bottom; i > index; i--,j++)
            {
                temp[j] = input[i];
            }
            for (i = 0; i < index; i++, j++)
            {
                temp[j] = input[i];
            }
            if (temp.Length > j) temp[j] = input[index];
            for (i = 0; i <= bottom; i++)
            {
                input[i] = temp[i];
            }
        }
    }

    public static int[] Main(int[] input)
    {
        SortPancake(input, input.Length);
        return input;
    }
```

```
}
```

Прочитайте Масленица онлайн: https://riptutorial.com/ru/algorithm/topic/7501/масленица

глава 35: Многопоточные алгоритмы

Вступление

Примеры некоторых многопоточных алгоритмов.

Синтаксис

- **параллельно** перед циклом каждая итерация цикла независима друг от друга и может выполняться параллельно.
- **spawn** указывает на создание нового потока.
- **sync** - синхронизировать все созданные потоки.
- Массивы / матрица индексируются от 1 до n в примерах.

Examples

Квадратное умножение матрицы многопоточность

```
multiply-square-matrix-parallel(A, B)
    n = A.lines
    C = Matrix(n,n) //create a new matrix n*n
    parallel for i = 1 to n
        parallel for j = 1 to n
            C[i][j] = 0
            pour k = 1 to n
                C[i][j] = C[i][j] + A[i][k]*B[k][j]
    return C
```

Мультипликационный матричный векторный многопоточный

```
matrix-vector(A,x)
    n = A.lines
    y = Vector(n) //create a new vector of length n
    parallel for i = 1 to n
        y[i] = 0
    parallel for i = 1 to n
        for j = 1 to n
            y[i] = y[i] + A[i][j]*x[j]
    return y
```

многопоточная слияния

A - это массив и *p* и *q* индексы массива, такие как сортировка *подматрицы A [p..r]* . *B* - это подматрица, которая будет заполнена сортировкой.

Вызов *p-merge-sort (A, p, r, B, s)* сортирует элементы из *A [p..r]* и помещает их в *B [s..s + rp]* .

```
p-merge-sort(A,p,r,B,s)
    n = r-p+1
    if n==1
        B[s] = A[p]
    else
        T = new Array(n) //create a new array T of size n
        q = floor((p+r)/2))
        q_prime = q-p+1
        spawn p-merge-sort(A,p,q,T,1)
        p-merge-sort(A,q+1,r,T,q_prime+1)
        sync
        p-merge(T,1,q_prime,q_prime+1,n,B,s)
```

Вот вспомогательная функция, которая выполняет слияние параллельно.
p-merge предполагает, что две подмассивы для слияния находятся в одном массиве, но не предполагают, что они смежны в массиве. Вот почему нам нужны *p1, r1, p2, r2* .

```
p-merge(T,p1,r1,p2,r2,A,p3)
    n1 = r1-p1+1
    n2 = r2-p2+1
    if n1<n2     //check if n1>=n2
        permute p1 and p2
        permute r1 and r2
        permute n1 and n2
    if n1==0     //both empty?
        return
    else
        q1 = floor((p1+r1)/2)
        q2 = dichotomic-search(T[q1],T,p2,r2)
        q3 = p3 + (q1-p1) + (q2-p2)
        A[q3] = T[q1]
        spawn p-merge(T,p1,q1-1,p2,q2-1,A,p3)
        p-merge(T,q1+1,r1,q2,r2,A,q3+1)
        sync
```

А вот вспомогательная функция дихотомического поиска.

x - это ключ к поиску в подматрице T [p..r].

```
dichotomic-search(x,T,p,r)
    inf = p
    sup = max(p,r+1)
    while inf<sup
        half = floor((inf+sup)/2)
        if x<=T[half]
            sup = half
        else
            inf = half+1
    return sup
```

Прочитайте Многопоточные алгоритмы онлайн: https://riptutorial.com/ru/algorithm/topic/9965/многопоточные-алгоритмы

глава 36: Обозначение Big-O

замечания

Определение

Обозначение Big-O в основе своей представляет собой математическую нотацию, используемую для сравнения скорости сходимости функций. Пусть `n -> f(n)` и `n -> g(n)` - функции, определенные над натуральными числами. Тогда будем говорить, что `f = O(g)` тогда и только тогда, когда `f(n)/g(n)` ограничена, когда n стремится к бесконечности. Другими словами, `f = O(g)` тогда и только тогда, когда существует константа A, такая, что для всех n, `f(n)/g(n) <= A`

На самом деле масштаб нотации Big-O немного шире в математике, но для простоты я сузил его до того, что используется в анализе сложности алгоритма: функции, определенные на естественных языках, которые имеют ненулевые значения, а случай роста n до бесконечности.

Что это значит ?

Возьмем случай `f(n) = 100n^2 + 10n + 1` и `g(n) = n^2` . Совершенно очевидно, что обе эти функции стремятся к бесконечности, поскольку n стремится к бесконечности. Но иногда знание предела не достаточно, и мы также хотим знать *скорость,* с которой функции приближаются к их пределу. Такие понятия, как Big-O, помогают сравнивать и классифицировать функции по скорости их конвергенции.

Давайте выясним, если `f = O(g)` , применяя определение. Мы имеем `f(n)/g(n) = 100 + 10/n + 1/n^2` . Так как `10/n` равно 10 , когда n равно 1 , и уменьшается, и так как `1/n^2` равен 1 , когда n равно 1 , а также уменьшается, мы F `f(n)/g(n) <= 100 + 10 + 1 = 111` . Определение выполняется, потому что мы нашли оценку `f(n)/g(n)` (111) и, следовательно, `f = O(g)` (мы говорим, что f является Big-O of `n^2`).

Это означает, что f стремится к бесконечности примерно с той же скоростью, что и g. Теперь это может показаться странным, потому что то, что мы обнаружили, состоит в том, что f в 111 раз больше, чем g, или, другими словами, когда g растет на 1, f растет не более чем на 111. Может показаться, что рост B 111 раз быстрее не «примерно такая же скорость». И действительно, нотация Big-O - не очень точный способ классификации скорости конвергенции функций, поэтому в математике мы используем соотношение эквивалентности, когда хотим точную оценку скорости. Но для разделения алгоритмов на больших скоростных классах Big-O достаточно. Нам не нужно выделять функции, которые растут фиксированное число раз быстрее друг друга, а только функции, которые растут *бесконечно* быстрее друг друга. Например, если взять `h(n) = n^2*log(n)` , мы видим, что `h(n)/g(n) = log(n)` который стремится к бесконечности с n, так что h *не* является O (n ^ 2),

так как h растет *бесконечно* быстрее, чем n ^ 2.

Теперь мне нужно сделать замечание: вы могли заметить, что если `f = O(g)` и `g = O(h)` , то `f = O(h)` . Например, в нашем случае мы имеем `f = O(n^3)` и `f = O(n^4)` ... В анализе сложности алгоритма мы часто говорим, что `f = O(g)` означает, что `f = O(g)` *и* `g = O(f)` , что можно понимать как «g - наименьший Big-O для f». В математике мы говорим, что такие функции являются биг-тетами друг друга.

Как он используется?

При сравнении производительности алгоритма нас интересует количество операций, выполняемых алгоритмом. Это называется *временной сложностью* . В этой модели мы считаем, что каждая базовая операция (сложение, умножение, сравнение, присвоение и т. Д.) Занимает фиксированный промежуток времени, и мы подсчитываем количество таких операций. Обычно мы можем выразить это число как функцию размера ввода, которую мы называем n. И, к сожалению, это число обычно растет до бесконечности с n (если это не так, мы говорим, что алгоритм O (1)). Мы разделяем наши алгоритмы на больших скоростных классах, определенных Big-O: когда мы говорим о «алгоритме O (n ^ 2)», мы имеем в виду, что число выполняемых им операций, выраженное как функция n, равно O (n ^ 2). Это говорит о том, что наш алгоритм примерно такой же быстрый, как алгоритм, который будет выполнять ряд операций, равных квадрату размера его ввода, *или быстрее* . «Скоростная» часть есть, потому что я использовал Big-O вместо Big-Theta, но обычно люди говорят, что Big-O означает Big-Theta.

При подсчете операций мы обычно рассматриваем наихудший случай: например, если у нас есть цикл, который может работать не более n раз и содержит 5 операций, количество операций, которые мы считаем, составляет 5n. Также можно рассмотреть сложность среднего случая.

Быстрое замечание: быстрый алгоритм - это тот, который выполняет несколько операций, поэтому, если число операций возрастает до бесконечности *быстрее* , то алгоритм работает *медленнее* : O (n) лучше, чем O (n ^ 2).

Иногда нас также интересует *пространственная сложность* нашего алгоритма. Для этого рассмотрим количество байтов в памяти, занимаемых алгоритмом, как функцию размера ввода, и используйте Big-O таким же образом.

Examples

Простая петля

Следующая функция находит максимальный элемент в массиве:

```
int find_max(const int *array, size_t len) {
    int max = INT_MIN;
```

```
    for (size_t i = 0; i < len; i++) {
        if (max < array[i]) {
            max = array[i];
        }
    }
    return max;
}
```

Размер ввода - это размер массива, который я назвал `len` в коде.

Давайте подсчитаем операции.

```
int max = INT_MIN;
size_t i = 0;
```

Эти два назначения выполняются только один раз, так что это две операции. Операциями, которые являются петлями, являются:

```
if (max < array[i])
i++;
max = array[i]
```

Поскольку в цикле есть три операции, и цикл выполняется n раз, мы добавляем `3n` в наши уже существующие 2 операции, чтобы получить `3n + 2` . Таким образом, наша функция выполняет `3n + 2` операции, чтобы найти max (его сложность `3n + 2`). Это многочлен, где наиболее быстро растущий член является фактором n, поэтому O (n).

Вероятно, вы заметили, что «операция» не очень четко определена. Например, я сказал, что `if (max < array[i])` была одна операция, но в зависимости от архитектуры этот оператор может скомпилировать, например, три команды: одно чтение памяти, одно сравнение и одну ветвь. Я также рассматривал все операции как одно и то же, хотя, например, операции с памятью будут медленнее, чем другие, и их производительность будет сильно различаться из-за, например, эффекта кеширования. Я также полностью проигнорировал оператор return, тот факт, что будет создан кадр для функции и т. Д. В конце концов, это не имеет значения для анализа сложности, потому что, как бы я ни выбрал подсчет операций, он изменит только коэффициент n-фактора и константы, поэтому результат все равно будет O (n). Сложность показывает, как алгоритм масштабируется с размером ввода, но это не единственный аспект производительности!

Вложенная петля

Следующая функция проверяет, имеет ли массив какие-либо дубликаты, беря каждый элемент, а затем итерируя по всему массиву, чтобы увидеть, существует ли этот элемент

```
_Bool contains_duplicates(const int *array, size_t len) {
    for (int i = 0; i < len - 1; i++) {
        for (int j = 0; j < len; j++) {
            if (i != j && array[i] == array[j]) {
```

```
                return 1;
            }
        }
    }
    return 0;
}
```

Внутренний цикл выполняет на каждой итерации ряд операций, которые являются постоянными с `n` . Внешний цикл также выполняет несколько постоянных операций и запускает внутренний цикл `n` раз. Сам внешний цикл запускается `n` раз. Таким образом, операции внутри внутреннего цикла выполняются `n^2` раза, операции во внешнем цикле выполняются `n` раз, а назначение `i` выполняется один раз. Таким образом, сложность будет чем-то вроде `an^2 + bn + c` , а так как старший член `n^2` , обозначение `O(n^2)` равно `O(n^2)` .

Как вы, возможно, заметили, мы можем улучшить алгоритм, избегая повторения одних и тех же сравнений несколько раз. Мы можем начинать с `i + 1` во внутреннем цикле, потому что все элементы перед ним уже были проверены на все элементы массива, в том числе на индекс `i + 1` . Это позволяет нам отказаться от проверки `i == j` .

```
_Bool faster_contains_duplicates(const int *array, size_t len) {
    for (int i = 0; i < len - 1; i++) {
        for (int j = i + 1; j < len; j++) {
            if (array[i] == array[j]) {
                return 1;
            }
        }
    }
    return 0;
}
```

Очевидно, что эта вторая версия делает меньше операций и, следовательно, более эффективна. Как это переводится в нотацию Big-O? Итак, теперь тело внутреннего цикла запускается `1 + 2 + ... + n - 1 = n(n-1)/2` раза. Это *все еще* многочлен второй степени, и поэтому остается только `O(n^2)` . Мы явно снизили сложность, так как мы грубо разделили на 2 числа операций, которые мы делаем, но мы все еще находимся в том же *классе* сложности, что и Big-O. Чтобы снизить сложность до более низкого класса, нам нужно было бы разделить количество операций на то, что *стремится к бесконечности* с `n` .

Пример O (log n)

Вступление

Рассмотрим следующую проблему:

`L` - отсортированный список, содержащий `n` значащих целых чисел (`n` достаточно большой), например `[-5, -2, -1, 0, 1, 2, 4]` (здесь `n` имеет значение 7). Если `L` как известно, содержит целое число 0, как вы можете найти индекс 0?

Наивный подход

Первое, что приходит на ум, - это просто прочитать каждый индекс до тех пор, пока не будет найдено 0. В худшем случае число операций равно `n` , поэтому сложность O (n).

Это отлично работает при малых значениях `n` , но есть ли более эффективный способ?

Дихотомия

Рассмотрим следующий алгоритм (Python3):

```
a = 0
b = n-1
while True:
  h = (a+b)//2 ## // is the integer division, so h is an integer
  if L[h] == 0:
    return h
  elif L[h] > 0:
    b = h
  elif L[h] < 0:
    a = h
```

`a` и `b` - индексы, между которыми должно быть найдено 0. Каждый раз, когда мы вводим цикл, мы используем индекс между `a` и `b` и используем его для сужения области поиска.

В худшем случае нам нужно подождать, пока `a` и `b` равны. Но сколько операций это делает? Не n, потому что каждый раз, когда мы вводим цикл, мы делим расстояние между `a` и `b` примерно на два. Скорее, сложность O (log n).

объяснение

Примечание. Когда мы записываем «журнал», мы имеем в виду двоичный логарифм или базу данных 2 (которую мы будем писать «log_2»). Поскольку O (log_2 n) = O (log n) (вы можете выполнить математику), мы будем использовать «log» вместо «log_2».

Назовем x числом операций: мы знаем, что 1 = n / (2 ^ x).

Итак, 2 ^ x = n, то x = log n

Заключение

Когда вы сталкиваетесь с последовательными делениями (будь то по два или по любому числу), помните, что сложность логарифмическая.

O (log n) типов алгоритмов

Допустим, у нас проблема размера n. Теперь для каждого шага нашего алгоритма (который нам нужно написать) наша исходная проблема становится половиной ее предыдущего размера (n / 2).

Поэтому на каждом шаге наша проблема становится вдвойне.

шаг	проблема
1	n / 2
2	n / 4
3	n / 8
4	n / 16

Когда проблемное пространство сокращается (т. Е. Решается полностью), оно не может быть уменьшено до конца (n становится равным 1) после выхода из условия проверки.

1. Скажем, на k-м шаге или количестве операций:

   размер проблемы = 1

2. Но мы знаем на k-м шаге, наш размер проблемы должен быть:

   размер проблемы = $n / 2^k$

3. От 1 до 2:

   $n / 2^k = 1$ или

   $n = 2^k$

4. Взять журнал с обеих сторон

   $\log_e n = k \log_e 2$

   или же

   $k = \log_e n / \log_e 2$

5. Используя формулу $\log_x m / \log_x n = \log_n m$

   $k = \log_2 n$

   или просто $k = \log n$

   Теперь мы знаем, что наш алгоритм может работать максимум до log n, поэтому временная сложность возникает как

O (log n)

Очень простой пример в коде для поддержки вышеприведенного текста:

```
for(int i=1; i<=n; i=i*2)
{
    // perform some operation
}
```

Итак, если кто-то спросит вас, будет ли n 256, сколько шагов этот цикл (или любой другой алгоритм, который сократит размер проблемы до половины), запустится, вы можете очень легко вычислить.

$k = \log_2 256$

$k = \log_2 2^8$ (=> $\log_a a = 1$)

$k = 8$

Другим очень хорошим примером для подобного случая является **алгоритм двоичного поиска** .

```
int bSearch(int arr[],int size,int item){
 int low=0;
 int high=size-1;

 while(low<=high){
     mid=low+(high-low)/2;
     if(arr[mid]==item)
         return mid;
     else if(arr[mid]<item)
         low=mid+1;
     else  high=mid-1;
     }
  return -1;// Unsuccessful result
}
```

Прочитайте Обозначение Big-O онлайн: https://riptutorial.com/ru/algorithm/topic/4770/обозначение-big-o

глава 37: Оболочка

Examples

Основная информация о Shell

Оболочка Shell , также известная как уменьшающаяся сортировка приращения, является одним из старейших алгоритмов сортировки, названным в честь его изобретателя Дональда. L. Shell (1959). Это быстро, легко понять и легко реализовать. Однако его анализ сложности немного сложнее.

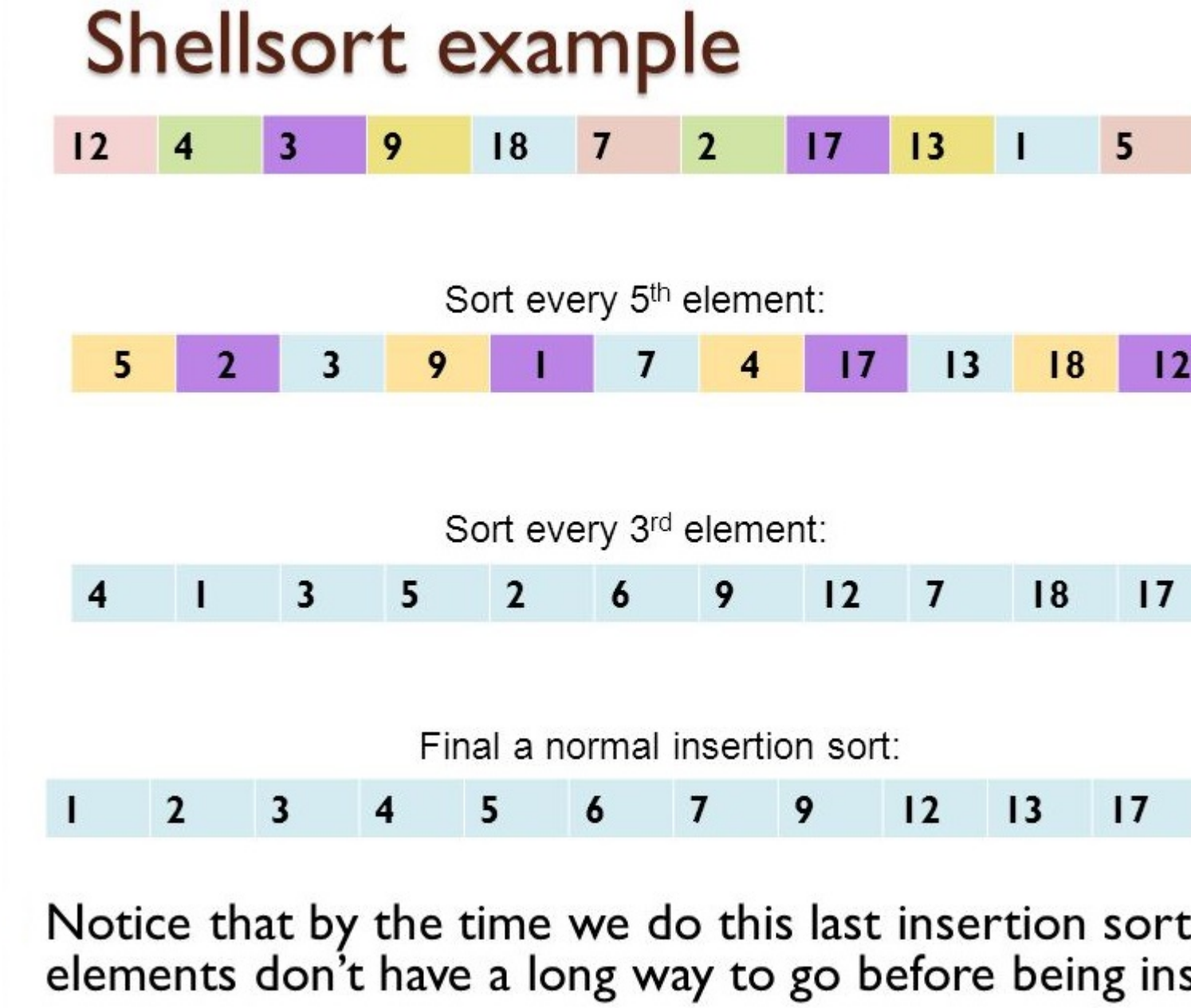
Идея сортировки Shell заключается в следующем:

1. Упорядочить последовательность данных в двумерном массиве
2. Сортировка столбцов массива

Shell sort улучшает сортировку вставки. Он начинается с сравнения элементов, находящихся далеко друг от друга, а затем элементов, находящихся далеко друг от друга, и, наконец, сравнения смежных элементов (фактически сортировки вставки).

Эффект заключается в том, что последовательность данных частично сортируется. Процесс выше повторяется, но каждый раз с более узким массивом, т. Е. С меньшим количеством столбцов. На последнем этапе массив состоит только из одного столбца.

Пример сортировки Shell:



Псевдокод для Shell Сортировка:

```
input
foreach element in input
{
    for(i = gap; i < n; i++)
    {
        temp = a[i]
        for (j = i; j >= gap and a[j - gap] > temp; j -= gap)
        {
            a[j] = a[j - gap]
        }
        a[j] = temp
    }
}
```

Вспомогательное пространство: `O(n) total, O(1) auxiliary`
Сложность времени: `O(nlogn)`

Реализация C

```
public class ShellSort
{
    static void SortShell(int[] input, int n)
    {
        var inc = 3;
        while (inc > 0)
        {
            int i;
            for (i = 0; i < n; i++)
            {
                var j = i;
                var temp = input[i];
                while ((j >= inc) && (input[j - inc] > temp))
                {
                    input[j] = input[j - inc];
                    j = j - inc;
                }
                input[j] = temp;
            }
            if (inc / 2 != 0)
                inc = inc / 2;
            else if (inc == 1)
                inc = 0;
            else
                inc = 1;
        }
    }

    public static int[] Main(int[] input)
    {
        SortShell(input, input.Length);
        return input;
    }
}
```

Прочитайте Оболочка онлайн: https://riptutorial.com/ru/algorithm/topic/7454/оболочка

глава 38: Подстрочный поиск

Examples

Алгоритм KMP в C

Учитывая текст *TXT* и шаблон *погладить,* цель этой программы будет печатать все вхождение *Пата* в *TXT.*

Примеры:

Входные данные **:**

```
  txt[] =  "THIS IS A TEST TEXT"
  pat[] = "TEST"
```

выход:

```
Pattern found at index 10
```

Входные данные **:**

```
  txt[] =  "AABAACAADAABAAABAA"
  pat[] = "AABA"
```

выход:

```
   Pattern found at index 0
   Pattern found at index 9
   Pattern found at index 13
```

C Язык:

```
// C program for implementation of KMP pattern searching
// algorithm
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

void computeLPSArray(char *pat, int M, int *lps);

void KMPSearch(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    // create lps[] that will hold the longest prefix suffix
    // values for pattern
    int *lps = (int *)malloc(sizeof(int)*M);
```

```
    int j  = 0;  // index for pat[]

    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps);

    int i = 0;  // index for txt[]
    while (i < N)
    {
      if (pat[j] == txt[i])
      {
        j++;
        i++;
      }

      if (j == M)
      {
        printf("Found pattern at index %d \n", i-j);
        j = lps[j-1];
      }

      // mismatch after j matches
      else if (i < N && pat[j] != txt[i])
      {
        // Do not match lps[0..lps[j-1]] characters,
        // they will match anyway
        if (j != 0)
         j = lps[j-1];
        else
         i = i+1;
      }
    }
    free(lps); // to avoid memory leak
}

void computeLPSArray(char *pat, int M, int *lps)
{
    int len = 0;  // length of the previous longest prefix suffix
    int i;

    lps[0] = 0; // lps[0] is always 0
    i = 1;

    // the loop calculates lps[i] for i = 1 to M-1
    while (i < M)
    {
       if (pat[i] == pat[len])
       {
         len++;
         lps[i] = len;
         i++;
       }
       else // (pat[i] != pat[len])
       {
         if (len != 0)
         {
           // This is tricky. Consider the example
           // AAACAAAA and i = 7.
           len = lps[len-1];

           // Also, note that we do not increment i here
         }
```

```
          else // if (len == 0)
          {
            lps[i] = 0;
            i++;
          }
        }
    }
}

// Driver program to test above function
int main()
{
   char *txt = "ABABDABACDABABCABAB";
   char *pat = "ABABCABAB";
   KMPSearch(pat, txt);
   return 0;
}
```

Выход:

```
Found pattern at index 10
```

Ссылка:

http://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/

Введение в алгоритм Рабина-Карпа

Алгоритм Рабина-Карпа - это алгоритм поиска строк, созданный Ричардом М. Карпом и Майклом О. Рабином, который использует хэширование, чтобы найти какой-либо из набора строк шаблонов в тексте.

Подстрокой строки является другая строка, которая встречается в. Например, *ver* является подстрокой *stackoverflow* . Не следует путать с подпоследовательностью, потому что *обложка* является подпоследовательностью одной и той же строки. Другими словами, любое подмножество последовательных букв в строке является подстрокой данной строки.

В алгоритме Рабина-Карпа мы создадим хэш нашего *шаблона,* который мы ищем, и проверьте, соответствует ли скользящий хэш нашего *текста шаблону* или нет. Если это не соответствует, мы можем гарантировать, что *шаблон* **не существует** в *тексте* . Однако, если он соответствует, *шаблон* **может** присутствовать в *тексте* . Давайте посмотрим на пример:

Допустим, у нас есть текст: **yeminsajid,** и мы хотим узнать, существует ли в тексте шаблон **nsa** . Чтобы вычислить хеш-хэш и хэш-лист, нам нужно использовать простое число. Это может быть любое простое число. Для этого примера возьмем **prime** = **11** . Мы определим хэш-значение, используя эту формулу:

```
(1st letter) X (prime) + (2nd letter) X (prime)¹ + (3rd letter) X (prime)² X + ......
```

Мы будем обозначать:

```
a -> 1    g -> 7    m -> 13   s -> 19   y -> 25
b -> 2    h -> 8    n -> 14   t -> 20   z -> 26
c -> 3    i -> 9    o -> 15   u -> 21
d -> 4    j -> 10   p -> 16   v -> 22
e -> 5    k -> 11   q -> 17   w -> 23
f -> 6    l -> 12   r -> 18   x -> 24
```

Хэш-значение **nsa** будет:

```
 14 X 11⁰ + 19 X 11¹ + 1 X 11² = 344
```

Теперь мы находим скользящий хэш нашего текста. Если скользящий хеш совпадает с хэш-значением нашего шаблона, мы проверим соответствие строк или нет. Поскольку наш шаблон имеет **3** буквы, мы возьмем 1-й **3** буквы **yem** из нашего текста и вычислим значение хэша. Мы получаем:

```
25 X 11⁰ + 5 X 11¹ + 13 X 11² = 1653
```

Это значение не совпадает с хэш-значением нашего шаблона. Поэтому строка здесь не существует. Теперь нам нужно рассмотреть следующий шаг. Чтобы вычислить хэш-значение нашей следующей строки **emi** . Мы можем вычислить это, используя нашу формулу. Но это было бы довольно тривиально и стоило бы нам больше. Вместо этого мы используем другую технику.

- Мы вычитаем значение **Первого письма предыдущей строки** из нашего текущего значения хэш-функции. В этом случае **y** . Получаем, `1653 - 25 = 1628` .
- Мы разделим разницу на наше **простое** , что для этого примера равно **11** . Получаем, `1628 / 11 = 148` .
- Добавим **новую букву X (простое)** $^{-1}$, где **m** - длина шаблона, с частным, что равно **i = 9** . Получаем, `148 + 9 X 11² = 1237` .

Новое значение хэша не равно нашему хэш-значению паттерна. Двигаясь дальше, для **n** получаем:

```
Previous String: emi
First Letter of Previous String: e(5)
New Letter: n(14)
New String: "min"
1237 - 5 = 1232
1232 / 11 = 112
112 + 14 X 11² = 1806
```

Это не соответствует. После этого для **s** получаем:

```
Previous String: min
First Letter of Previous String: m(13)
New Letter: s(19)
New String: "ins"
1806 - 13 = 1793
1793 / 11 = 163
163 + 19 X 11² = 2462
```

Это не соответствует. Далее, для **a** получаем:

```
Previous String: ins
First Letter of Previous String: i(9)
New Letter: a(1)
New String: "nsa"
2462 - 9 = 2453
2453 / 11 = 223
223 + 1 X 11² = 344
```

Это матч! Теперь мы сравниваем наш шаблон с текущей строкой. Поскольку обе строки совпадают, подстрока существует в этой строке. И мы возвращаем исходную позицию нашей подстроки.

Псевдокод будет:

Расчет хеширования:

```
Procedure Calculate-Hash(String, Prime, x):
hash := 0                                   // Here x denotes the length to be considered
for m from 1 to x                           // to find the hash value
    hash := hash + (Value of String[m])□^-1
end for
Return hash
```

Пересчет хешей:

```
Procedure Recalculate-Hash(String, Curr, Prime, Hash):
Hash := Hash - Value of String[Curr]  //here Curr denotes First Letter of Previous String
Hash := Hash / Prime
m := String.length
New := Curr + m - 1
Hash := Hash + (Value of String[New])□^-1
Return Hash
```

Строковое совпадение:

```
Procedure String-Match(Text, Pattern, m):
for i from m to Pattern-length + m - 1
    if Text[i] is not equal to Pattern[i]
        Return false
    end if
end for
Return true
```

Рабина-Карпа:

```
Procedure Rabin-Karp(Text, Pattern, Prime):
m := Pattern.Length
HashValue := Calculate-Hash(Pattern, Prime, m)
CurrValue := Calculate-Hash(Pattern, Prime, m)
for i from 1 to Text.length - m
    if HashValue == CurrValue and String-Match(Text, Pattern, i) is true
        Return i
    end if
    CurrValue := Recalculate-Hash(String, i+1, Prime, CurrValue)
end for
Return -1
```

Если алгоритм не находит никакого совпадения, он просто возвращает **-1** .

Этот алгоритм используется при обнаружении плагиата. Учитывая исходный материал, алгоритм может быстро искать бумагу для экземпляров предложений из исходного материала, игнорируя такие данные, как случай и пунктуация. Из-за обилия искомых строк однолинейные поисковые алгоритмы здесь нецелесообразны. Опять же, **алгоритм Кнута-Морриса-Пратта** или **алгоритм поиска по методу Boyer-Moore - это алгоритм поиска алгоритма с** одним шаблоном, чем **Rabin-Karp** . Тем не менее, это алгоритм выбора для множественного поиска по шаблону. Если мы хотим найти любое из большого числа, скажем k, шаблонов фиксированной длины в тексте, мы можем создать простой вариант алгоритма Рабина-Карпа.

Для текста длины **n** и **p-** образных изображений с комбинированной длиной **m** его среднее время и наилучшее время пробега **O (n + m)** в пространстве **O (p)** , но в худшем случае это **O (nm)** .

Введение в алгоритм Кнут-Моррис-Пратт (КМП)

Предположим, что у нас есть *текст* и *шаблон* . Нам нужно определить, существует ли шаблон в тексте или нет. Например:

```
+-------+---+---+---+---+---+---+---+---+
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-------+---+---+---+---+---+---+---+---+
|  Text | a | b | c | b | c | g | l | x |
+-------+---+---+---+---+---+---+---+---+

+---------+---+---+---+---+
| Index   | 0 | 1 | 2 | 3 |
+---------+---+---+---+---+
| Pattern | b | c | g | l |
+---------+---+---+---+---+
```

Этот *шаблон* существует в *тексте* . Поэтому наш поиск подстроки должен вернуть **3** , индекс позиции, с которой начинается этот *шаблон* . Итак, как работает наша процедура поиска подстроки грубой силы?

Обычно мы делаем это: мы начинаем с **0-го** индекса *текста* и **0-го** индекса нашего * шаблона, и мы сравниваем **Text [0]** с **Pattern [0]** . Поскольку они не соответствуют, мы переходим к следующему индексу нашего *текста,* и мы сравниваем **текст [1]** с **Pattern [0]** . Поскольку это совпадение, мы увеличиваем индекс нашего *шаблона* и индекс *текста* . Мы сравниваем **текст [2]** с **рисунком [1]** . Они также подходят. Следуя той же самой процедуре, описанной ранее, мы теперь сравниваем **текст [3]** с **рисунком [2]** . Поскольку они не совпадают, мы начинаем со следующей позиции, где мы начали находить матч. Это индекс **2** *текста* . Мы сравниваем **текст [2]** с **рисунком [0]** . Они не совпадают. Затем увеличивая индекс *текста* , мы сравниваем **Text [3]** с **Pattern [0]** . Они совпадают. Снова **текст [4]** и **Pattern [1]** соответствуют, **текст [5]** и **шаблон [2]** соответствуют, а **текст [6]** и **шаблон [3]** соответствуют. Поскольку мы достигли конца нашего *шаблона* , мы теперь возвращаем индекс, с которого начинался наш матч, т. Е. **3** . Если наш *шаблон* был: `bcgll` , это означает, что если *шаблон* не существует в нашем *тексте* , наш поиск должен возвращать исключение или **-1** или любое другое предопределенное значение. Мы можем ясно видеть, что в худшем случае этот алгоритм займет время `O(mn)` где **m** - длина *текста,* а **n** - длина *шаблона* . Как мы сокращаем эту сложность во времени? Именно здесь на картинке появляется алгоритм поиска подстроки KMP.

Алгоритм поиска Knuth-Morris-Pratt String или алгоритм KMP ищет появление «шаблона» в основном «тексте», используя наблюдение, что, когда происходит несоответствие, само слово содержит достаточную информацию, чтобы определить, где может начаться следующий матч , тем самым минуя повторное рассмотрение ранее согласованных символов. Алгоритм был задуман в 1970 году Донульдом Кнутом и Воганом Праттом и независимо от Джеймса Морриса . Трио опубликовало его совместно в 1977 году.

Давайте расширим наш пример *Text* и *Pattern* для лучшего понимания:

```
+-------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| Index |0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |10|11|12|13|14|15|16|17|18|19|20|21|22|
+-------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  Text |a |b |c |x |a |b |c |d |a |b |x |a |b |c |d |a |b |c |d |a |b |c |y |
+-------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | y |
+---------+---+---+---+---+---+---+---+---+
```

Сначала наш *текст* и *шаблон* соответствуют индексу **2** . **Текст [3]** и **шаблон [3]** не совпадают. Поэтому наша цель - не возвращаться назад в этом *тексте* , то есть в случае несоответствия мы не хотим, чтобы наше совпадение начиналось с позиции, с которой мы начали сопоставляться. Для этого мы будем искать **суффикс** в нашем *шаблоне* прямо перед нашим несоответствием (подстрока **abc**), который также является **префиксом** подстроки нашего *шаблона* . В нашем примере, поскольку все символы уникальны, суффиксов нет, это префикс нашей подобранной подстроки. Итак, что это значит, наше

следующее сравнение начнется с индекса **0** . Подождите немного, вы поймете, почему мы это сделали. Затем мы сравниваем **Text [3]** с **Pattern [0],** и это не соответствует. После этого для *текста* из индекса **4** в индекс **9** и для *шаблона* от индекса **0** до индекса **5** мы найдем совпадение. Мы находим несоответствие в **Text [10]** и **Pattern [6]** . Поэтому мы берем подстроку из *шаблона* прямо до точки, где происходит несоответствие (подстрока **abcdabc**), мы проверяем суффикс, который также является префиксом этой подстроки. Мы можем видеть, что **ab** является суффиксом и префиксом этой подстроки. Это означает, что, поскольку мы сопоставлены до **Text [10]** , символы перед ошибкой являются **ab** . Из этого можно сделать вывод, что поскольку **ab** также является префиксом подстроки, которую мы взяли, нам не нужно снова проверять **ab,** и следующая проверка может начинаться с **Text [10]** и **Pattern [2]** . Нам не пришлось оглядываться на весь *текст* , мы можем начать прямо с того места, где произошло наше несоответствие. Теперь мы проверяем **Text [10]** и **Pattern [2]** , так как это несоответствие, а подстрока перед несоответствием (**abc**) не содержит суффикса, который также является префиксом, мы проверяем **Text [10]** и **Pattern [0]** , они не совпадают. После этого для *текста* из индекса **11** в индекс **17** и для *шаблона* из индекса **0** в индекс **6** . Мы находим несоответствие в **Text [18]** и **Pattern [7]** . Таким образом, мы снова проверяем подстроку перед несоответствием (substring **abcdabc**), a find **abc** - как суффикс, так и префикс. Поэтому, поскольку мы сопоставлялись с **Pattern [7]** , **abc** должен быть до **Text [18]** . Это означает, что нам не нужно сравнивать до тех пор, пока **текст [17]** и наше сравнение не начнутся с **текста [18]** и **шаблона [3]** . Таким образом, мы найдем матч, и мы вернем **15,** что является нашим начальным индексом матча. Так наш поиск подстроки KMP работает с использованием суффикса и префикса.

Теперь, как мы можем эффективно вычислить, является ли суффикс таким же, как префикс, и в какой момент начать проверку, существует ли несоответствие символа между *Text* и *Pattern* . Давайте рассмотрим пример:

```
+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | a |
+---------+---+---+---+---+---+---+---+---+
```

Мы будем генерировать массив, содержащий требуемую информацию. Назовем массив **S.** Размер массива будет таким же, как длина рисунка. Поскольку первая буква *шаблона* не может быть суффиксом какого-либо префикса, мы поместим **S [0]** = **0** . Сначала возьмем **i** = **1** и **j** = **0** . На каждом шаге мы сравниваем **Pattern [i]** и **Pattern [j]** и приращение **i** . Если есть совпадение, мы помещаем **S [i]** = **j** + **1** и приращение **j** , если есть несоответствие, мы проверяем предыдущую позицию значения **j** (если доступно) и устанавливаем **j** = **S [j-1]** (если **j** не равен **0**), мы продолжаем делать это до тех пор, пока **S [j]** не будет соответствовать **S [i]** или **j** не станет **0** . Для более позднего мы положим **S [i]** = **0** . Для нашего примера:

```
            j   i
+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | a |
+---------+---+---+---+---+---+---+---+---+
```

Шаблон [j] и **Pattern [i]** не совпадают, поэтому мы увеличиваем **i,** а так как **j** равно **0** , мы не проверяем предыдущее значение и ставим **Pattern [i]** = **0** . Если мы будем продолжать увеличивать **i** , то для **i** = **4** мы получим соответствие, поэтому поместим **S [i]** = **S [4]** = **j** + **1** = **0** + **1** = **1** и приращение **j** и **i** . Наш массив будет выглядеть так:

```
                j               i
+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | a |
+---------+---+---+---+---+---+---+---+---+
|    S    | 0 | 0 | 0 | 0 | 1 |   |   |   |
+---------+---+---+---+---+---+---+---+---+
```

Так как **Pattern [1]** и **Pattern [5]** являются совпадением, мы помещаем **S [i]** = **S [5]** = **j** + **1** = **1** + **1** = **2** . Если продолжить, мы найдем несоответствие для **j** = **3** и **i** = **7** . Так как **j** не равно **0** , положим **j** = **S [j-1]** . И мы сравним символы в **i** и **j,** такие же или нет, поскольку они одинаковы, мы поместим **S [i]** = **j** + 1. Наш завершенный массив будет выглядеть так:

```
+---------+---+---+---+---+---+---+---+---+
|    S    | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 1 |
+---------+---+---+---+---+---+---+---+---+
```

Это наш необходимый массив. Здесь ненулевое значение **S [i]** означает, что существует суффикс длины **S [i],** такой же, как префикс в этой подстроке (подстрока от **0** до **i**), а следующее сравнение начнется с **S [i]** + **1** позиции *Образец* . Наш алгоритм генерации массива будет выглядеть так:

```
Procedure GenerateSuffixArray(Pattern):
i := 1
j := 0
n := Pattern.length
while i is less than n
    if Pattern[i] is equal to Pattern[j]
        S[i] := j + 1
        j := j + 1
        i := i + 1
    else
        if j is not equal to 0
            j := S[j-1]
        else
            S[i] := 0
            i := i + 1
        end if
    end if
end while
```

Сложность времени для создания этого массива - `O(n)` а сложность пространства также `O(n)` . Чтобы убедиться, что вы полностью поняли алгоритм, попробуйте создать массив для шаблона `aabaabaa` и проверить, соответствует ли результат этому .

Теперь давайте проведем поиск подстроки, используя следующий пример:

```
+---------+---+---+---+---+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |
+---------+---+---+---+---+---+---+---+---+---+---+---+---+
|   Text  | a | b | x | a | b | c | a | b | c | a | b | y |
+---------+---+---+---+---+---+---+---+---+---+---+---+---+

+---------+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 |
+---------+---+---+---+---+---+---+
| Pattern | a | b | c | a | b | y |
+---------+---+---+---+---+---+---+
|    S    | 0 | 0 | 0 | 1 | 2 | 0 |
+---------+---+---+---+---+---+---+
```

У нас есть *текст* , *шаблон* и предварительно вычисленный массив *S,* используя нашу логику, определенную ранее. Мы сравниваем **Text [0]** и **Pattern [0],** и они одинаковы. **Текст [1]** и **шаблон [1]** совпадают. **Текст [2]** и **шаблон [2]** не совпадают. Мы проверяем значение в позиции прямо перед несоответствием. Так как **S [1]** равно **0** , то суффикса не будет такого же, как префикс в нашей подстроке, и наше сравнение начинается в позиции **S [1]** , которая равна **0** . Итак, **Pattern [0]** не такой, как **Text [2]** , поэтому мы движемся дальше. **Текст [3]** совпадает с **шаблоном [0],** и существует совпадение с **текстом [8]** и **паттерном [5]** . Мы делаем один шаг назад в массиве **S** и находим **2** . Таким образом, это означает, что существует префикс длины **2,** который также является суффиксом этой подстроки (**abcab),** которая является **ab** . Это также означает, что есть **текст ab** перед **текстом [8]** . Поэтому мы можем смело игнорировать **Pattern [0]** и **Pattern [1]** и начать наше следующее сравнение с **Pattern [2]** и **Text [8]** . Если мы продолжим, мы найдем *шаблон* в *тексте* . Наша процедура будет выглядеть так:

```
Procedure KMP(Text, Pattern)
GenerateSuffixArray(Pattern)
m := Text.Length
n := Pattern.Length
i := 0
j := 0
while i is less than m
    if Pattern[j] is equal to Text[i]
        j := j + 1
        i := i + 1
    if j is equal to n
        Return (j-i)
    else if i < m and Pattern[j] is not equal t Text[i]
        if j is not equal to 0
            j = S[j-1]
        else
            i := i + 1
        end if
```

```
    end if
end while
Return -1
```

Сложность времени этого алгоритма, кроме вычисления суффиксного массива, равна `O(m)` . Поскольку *GenerateSuffixArray* принимает `O(n)` , общая временная сложность алгоритма KMP: `O(m+n)` .

PS: Если вы хотите найти несколько экземпляров *Pattern* в *тексте* , вместо того, чтобы возвращать значение, распечатайте его / сохраните и установите `j := S[j-1]` . Также сохраните `flag` чтобы отслеживать, находили ли вы какое-либо событие или нет и обрабатываете его соответствующим образом.

Python Реализация алгоритма KMP.

Haystack : строка, в которой данный шаблон нужно искать.
Игла : шаблон для поиска.

Сложность времени : часть поиска (метод strstr) имеет сложность O (n), где `n` - длина стога сена, но поскольку игла также предварительно анализируется для таблицы префикса здания O (m), требуется для построения таблицы префикса, где `m` - длина игла. Поэтому общая временная сложность для KMP равна **O (n + m)**
Сложность пространства : **O (m)** из-за таблицы префиксов на игле.

Примечание. После выполнения возвращается исходная позиция совпадения в стоге сена (если есть совпадение) else возвращает -1, для краевых случаев, например, если игла / стог сена - пустая строка или игла не найдена в стоге сена.

```
def get_prefix_table(needle):
    prefix_set = set()
    n = len(needle)
    prefix_table = [0]*n
    delimeter = 1
    while(delimeter<n):
        prefix_set.add(needle[:delimeter])
        j = 1
        while(j<delimeter+1):
            if needle[j:delimeter+1] in prefix_set:
                prefix_table[delimeter] = delimeter - j + 1
                break
            j += 1
        delimeter += 1
    return prefix_table

def strstr(haystack, needle):
    # m: denoting the position within S where the prospective match for W begins
    # i: denoting the index of the currently considered character in W.
    haystack_len = len(haystack)
    needle_len = len(needle)
    if (needle_len > haystack_len) or (not haystack_len) or (not needle_len):
        return -1
    prefix_table = get_prefix_table(needle)
```

```
    m = i = 0
    while((i<needle_len) and (m<haystack_len)):
        if haystack[m] == needle[i]:
            i += 1
            m += 1
        else:
            if i != 0:
                i = prefix_table[i-1]
            else:
                m += 1
    if i==needle_len and haystack[m-1] == needle[i-1]:
        return m - needle_len
    else:
        return -1

if __name__ == '__main__':
    needle = 'abcaby'
    haystack = 'abxabcabcaby'
    print strstr(haystack, needle)
```

Прочитайте Подстрочный поиск онлайн: https://riptutorial.com/ru/algorithm/topic/7118/подстрочный-поиск

глава 39: Подсчет сортировки

Examples

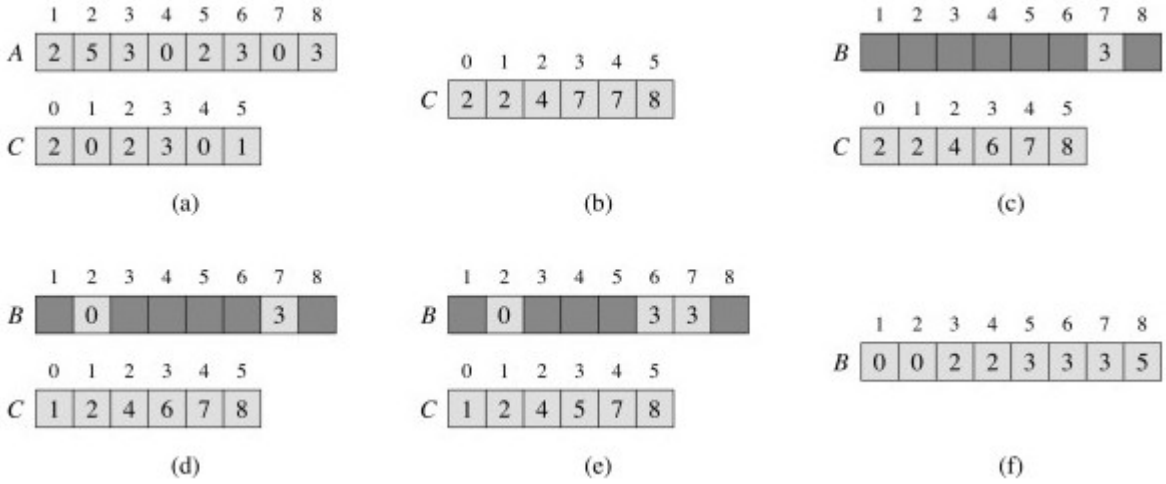
Подсчет сортировки

Сортировка сортировки - это целочисленный алгоритм сортировки для коллекции объектов, сортируемых в соответствии с ключами объектов.

меры

1. Построить рабочий массив *C* , размер которого равен диапазону входного массива *A*.
2. Перейдем через *A* , назначив *C* [x] в зависимости от количества раз x, появившегося в *A*.
3. Преобразуйте *C* в массив, где *C* [x] ссылается на число значений ≤ x, итерируя через массив, присваивая каждому *C* [x] сумму своего предыдущего значения и все значения в *C*, которые выходят перед ним.
4. Перемещайтесь назад через *A* , помещая каждое значение в новый отсортированный массив *B* по индексу, записанному на *C*. Это делается для заданного *A* [x], назначая *B* [*C* [*A* [x]]] *A* [x] и уменьшая *C* [*A* [x]] в случае, если в исходном несортированном массиве были повторяющиеся значения.

Пример подсчета сортировки



Вспомогательное пространство: `O(n+k)`
Сложность времени: худший случай: `O(n+k)` , наилучший случай: `O(n)` , средний случай `O(n+k)`

Внедрение Psuedocode

Ограничения:

1. Вход (массив, который нужно отсортировать)
2. Количество элементов ввода (n)
3. Клавиши в диапазоне *0..k-1* (k)
4. Count (массив чисел)

псевдокод:

```
for x in input:
    count[key(x)] += 1
total = 0
for i in range(k):
    oldCount = count[i]
    count[i] = total
    total += oldCount
for x in input:
    output[count[key(x)]] = x
    count[key(x)] += 1
return output
```

Реализация C

```
public class CountingSort
{
    public static void SortCounting(int[] input, int min, int max)
    {
        var count = new int[max - min + 1];
        var z = 0;

        for (var i = 0; i < count.Length; i++)
            count[i] = 0;

        foreach (int i in input)
            count[i - min]++;

        for (var i = min; i <= max; i++)
        {
            while (count[i - min]-- > 0)
            {
                input[z] = i;
                ++z;
            }
        }
    }

    public static int[] Main(int[] input)
    {
        SortCounting(input, input.Min(), input.Max());
        return input;
    }
}
```

Прочитайте Подсчет сортировки онлайн: https://riptutorial.com/ru/algorithm/topic/7251/подсчет-сортировки

глава 40: поиск

Examples

Двоичный поиск

Вступление

Бинарный поиск - это алгоритм поиска Divide и Conquer. Он использует время `O(log n)` чтобы найти местоположение элемента в пространстве поиска, где `n` - размер пространства поиска.

Двоичный поиск работает, уменьшая вдвое пространство поиска на каждой итерации после сравнения целевого значения с средним значением пространства поиска.

Чтобы использовать двоичный поиск, пространство поиска должно быть упорядочено (отсортировано) каким-то образом. Дублирующие записи (те, которые сравниваются как равные по сравнению с функцией сравнения) не могут быть различимы, хотя они не нарушают свойство двоичного поиска.

Обычно мы используем меньше (<) в качестве функции сравнения. Если a <b, оно вернет true. если a не меньше b и b не меньше a, a и b равны.

Пример

Вы экономист, но очень плохой. Вам задана задача найти равновесную цену (то есть цену, где спрос = спрос) на рис.

Помните, что чем выше цена, тем больше предложение и тем меньше спрос

Поскольку ваша компания очень эффективна при расчете рыночных сил, вы можете мгновенно получить спрос и предложение в единицах риса, когда цена на рис устанавливается по определенной цене. `p`

Ваш босс хочет равновесную цену как можно скорее, но говорит вам, что равновесная цена может быть положительным целым числом, которое составляет не более `10^17` и гарантируется, что в диапазоне будет ровно 1 положительное целочисленное решение. Так что идите с вашей работой, прежде чем потерять ее!

Вам разрешено вызывать функции `getSupply(k)` и `getDemand(k)` , которые будут выполнять именно то, что указано в задаче.

Пример Объяснение

Здесь наше пространство поиска от `1` до `10^17` . Таким образом, линейный поиск недостижим.

Однако заметьте, что при увеличении `k` увеличивается значение `getSupply(k)` и `getDemand(k)` . Таким образом, для любого `x > y getSupply(x) - getDemand(x) > getSupply(y) - getDemand(y)` . Поэтому это пространство поиска является монотонным, и мы можем использовать двоичный поиск.

Следующий псевдокод демонстрирует использование двоичного поиска:

```
high = 100000000000000000     <- Upper bound of search space
low = 1                       <- Lower bound of search space
while high - low > 1
    mid = (high + low) / 2    <- Take the middle value
    supply = getSupply(mid)
    demand = getDemand(mid)
    if supply > demand
        high = mid             <- Solution is in lower half of search space
    else if demand > supply
        low = mid              <- Solution is in upper half of search space
    else                       <- supply==demand condition
        return mid             <- Found solution
```

Этот алгоритм работает в `~O(log 10^17)` времени. Это может быть обобщена на `~O(log S)` времени , где S является размер пространства поиска , поскольку на каждой итерации в `while` цикла, мы вдвое поисковое пространство *(из [низкого: высокая] либо [низкий: средний] или [mid: high]*).

C Реализация двоичного поиска с рекурсией

```
int binsearch(int a[], int x, int low, int high) {
    int mid;

    if (low > high)
      return -1;

    mid = (low + high) / 2;

    if (x == a[mid]) {
        return (mid);
    } else
    if (x < a[mid]) {
        binsearch(a, x, low, mid - 1);
    } else {
        binsearch(a, x, mid + 1, high);
    }
}
```

Двоичный поиск: по отсортированным номерам

Проще всего показать двоичный поиск чисел, используя псевдокод

```
int array[1000] = { sorted list of numbers };
int N = 100;  // number of entries in search space;
int high, low, mid; // our temporaries
int x; // value to search for

low = 0;
high = N -1;
while(low < high)
{
    mid = (low + high)/2;
    if(array[mid] < x)
        low = mid + 1;
    else
        high = mid;
}
if(array[low] == x)
    // found, index is low
else
    // not found
```

Не пытайтесь вернуться раньше, сравнивая массив [mid] с x для равенства. Дополнительное сравнение может только замедлить работу кода. Обратите внимание, что вам нужно добавить один к минимуму, чтобы избежать попадания в ловушку целым делением, всегда округляющимся вниз.

Интересно, что приведенная выше версия бинарного поиска позволяет найти наименьшее количество x в массиве. Если массив содержит дубликаты x, алгоритм может быть слегка изменен, чтобы он мог вернуть наибольшее вхождение x, просто добавив к условию if:

```
while(low < high)
    {
        mid = low + ((high - low) / 2);
        if(array[mid] < x || (array[mid] == x && array[mid + 1] == x))
            low = mid + 1;
        else
            high = mid;
    }
```

Обратите внимание, что вместо выполнения `mid = (low + high) / 2` также может быть хорошей идеей попробовать `mid = low + ((high - low) / 2)` для реализаций, таких как реализации Java, чтобы снизить риск получения переполнение для действительно больших входов.

Линейный поиск

Линейный поиск - простой алгоритм. Он перебирает элементы до тех пор, пока запрос не будет найден, что делает его линейным алгоритмом - сложность O (n), где n - количество элементов, которые нужно пройти.

Почему O (n)? В худшем случае вам нужно пройти все n пунктов.

Его можно сравнить с поиском книги в стопке книг - вы просматриваете их все, пока не найдете тот, который вам нужен.

Ниже приведена реализация Python:

```
def linear_search(searchable_list, query):
    for x in searchable_list:
        if query == x:
            return True
    return False

linear_search(['apple', 'banana', 'carrot', 'fig', 'garlic'], 'fig') #returns True
```

Рабин Карп

Алгоритм Рабина-Карпа или алгоритм Карпа-Рабина представляет собой строковый алгоритм поиска, который использует хеширование для поиска любого из набора строк шаблонов в тексте. Среднее и лучшее время выполнения - O (n + m) в пространстве O (p), но в худшем случае это O (nm), где n - длина текста, а m - длина рисунка.

Реализация алгоритма в java для сопоставления строк

```
void RabinfindPattern(String text,String pattern){
    /*
    q a prime number
    p hash value for pattern
    t hash value for text
    d is the number of unique characters in input alphabet
    */
    int d=128;
    int q=100;
    int n=text.length();
    int m=pattern.length();
    int t=0,p=0;
    int h=1;
    int i,j;
//hash value calculating function
    for (i=0;i<m-1;i++)
            h = (h*d)%q;
        for (i=0;i<m;i++){
        p = (d*p + pattern.charAt(i))%q;
        t = (d*t + text.charAt(i))%q;
        }
//search for the pattern
    for(i=0;i<end-m;i++){
        if(p==t){
//if the hash value matches match them character by character
            for(j=0;j<m;j++)
                if(text.charAt(j+i)!=pattern.charAt(j))
                break;
            if(j==m && i>=start)
                System.out.println("Pattern match found at index "+i);
        }
        if(i<end-m){
            t =(d*(t - text.charAt(i)*h) + text.charAt(i+m))%q;
```

```
            if(t<0)
                t=t+q;
        }
    }
}
```

При вычислении хэш-значения мы делим его на простое число, чтобы избежать столкновения. После деления на простое число шансов столкновения будет меньше, но все же существует вероятность того, что значение хэша может быть одинаковым для двух строк, поэтому, когда мы получаем совпадение, мы должны проверить его персонаж по персонажу, чтобы убедиться, что мы получили правильное совпадение.

t = (d * (t - text.charAt (i) * h) + text.charAt (i + m))% q;

Это необходимо для пересчета хэш-значения для шаблона, сначала путем удаления самого левого символа, а затем добавления нового символа из текста.

Анализ линейного поиска (худшие, средние и лучшие случаи)

Мы можем иметь три случая для анализа алгоритма:

1. Худший случай
2. Средний случай
3. Лучший случай

```
#include <stdio.h>

// Linearly search x in arr[].  If x is present then return the index,

// otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
    {
        if (arr[i] == x)
         return i;
    }

    return -1;
}
```

/ * Программа драйвера для тестирования вышеперечисленных функций * /

```
int main()
{
    int arr[] = {1, 10, 30, 15};
    int x = 30;
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("%d is present at index %d", x, search(arr, n, x));
```

```
    getchar();
    return 0;
}
```

Худший анализ ситуации (обычно делается)

В худшем случае мы вычисляем верхнюю границу времени работы алгоритма. Мы должны знать случай, который вызывает максимальное количество операций. Для линейного поиска худший случай возникает, когда элемент, который нужно искать (x в приведенном выше коде), отсутствует в массиве. Когда x отсутствует, функции search () сравнивают его со всеми элементами arr [] поочередно. Поэтому наихудшая временная сложность линейного поиска будет равна Θ (n)

Средний анализ ситуации (иногда делается)

При анализе среднего случая мы берем все возможные входы и вычисляем время вычислений для всех входов. Суммируйте все рассчитанные значения и разделите сумму на общее количество входов. Мы должны знать (или предсказывать) распределение дел. Для задачи линейного поиска допустим, что все случаи равномерно распределены (в том числе случай, когда x не присутствует в массиве). Итак, мы суммируем все случаи и делим сумму на (n + 1). Ниже приведено значение средней сложности времени случая.

$$\text{Average Case Time} = \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$$

$$= \frac{\theta((n+1)*(n+2)/2)}{(n+1)}$$

$$= \Theta(n)$$

Лучший анализ случая (Bogus)

В лучшем случае анализа мы вычисляем нижнюю границу времени работы алгоритма. Мы должны знать случай, который вызывает минимальное количество операций. В задаче линейного поиска наилучший случай возникает, когда x присутствует в первом местоположении. Количество операций в лучшем случае является постоянным (не зависящим от n). Таким образом, сложность времени в лучшем случае будет равна Θ (1). В большинстве случаев мы делаем анализ худшего случая для анализа алгоритмов. В худшем анализе мы гарантируем верхнюю границу времени работы алгоритма, который является хорошей информацией. Средний случайный анализ непросто сделать в большинстве практических случаев, и это редко делается. В анализе среднего случая мы должны знать (или прогнозировать) математическое распределение всех возможных входов. Анализ

наилучшего случая - фиктивный. Гарантия нижней границы алгоритма не дает никакой информации, как в худшем случае, для выполнения алгоритма могут потребоваться годы.

Для некоторых алгоритмов все случаи асимптотически одинаковы, т. Е. Нет худших и лучших случаев. Например, Merge Sort. Merge Sort выполняет операции Θ (nLogn) во всех случаях. Большинство других алгоритмов сортировки имеют худшие и лучшие случаи. Например, в типичной реализации Quick Sort (где точка поворота выбрана как элемент угла) наихудшее происходит, когда входной массив уже отсортирован, и лучше всего возникает, когда опорные элементы всегда делят массив на две половины. Для сортировки вставки худший случай возникает, когда массив сортируется по обратным ссылкам, и лучший случай возникает, когда массив сортируется в том же порядке, что и вывод.

Прочитайте поиск онлайн: https://riptutorial.com/ru/algorithm/topic/4471/поиск

глава 41: Поиск по ширине

Examples

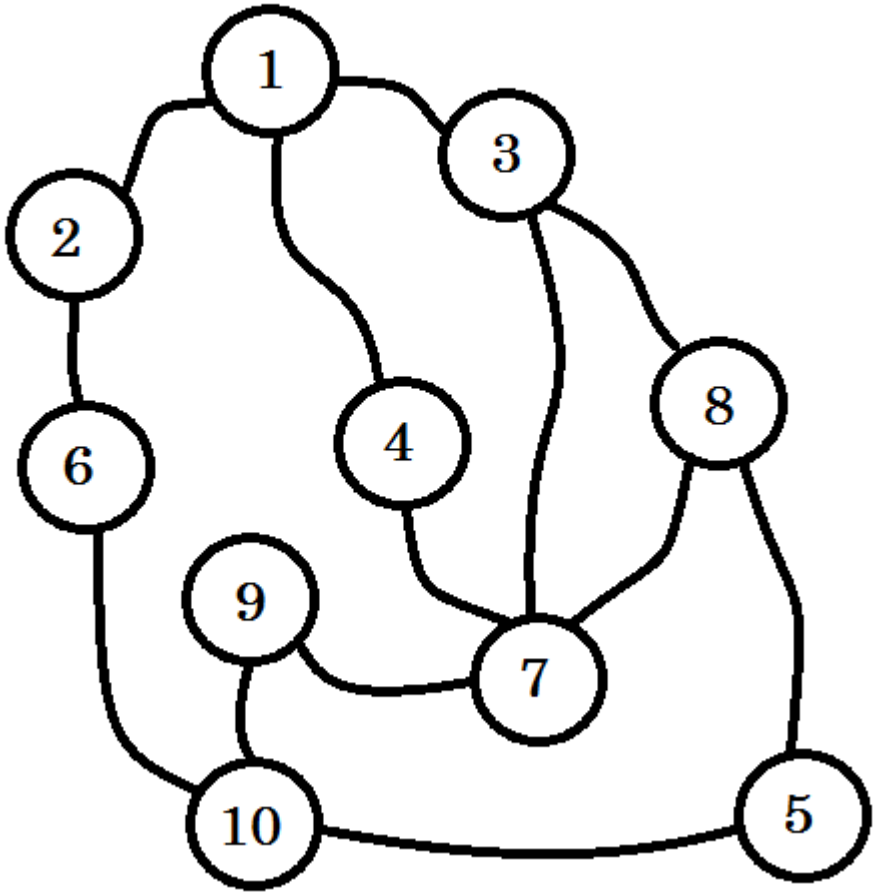
Поиск кратчайшего пути от источника к другим узлам

Breadth-first-search (BFS) - это алгоритм для перемещения или поиска структур данных дерева или графика. Он начинается с корня дерева (или какого-либо произвольного узла графа, иногда называемого «ключом поиска») и сначала исследует соседние узлы, а затем переходит к соседним соседним уровням. BFS был изобретен в конце 1950-х годов Эдвардом Форрестом Муром , который использовал его для поиска кратчайшего пути из лабиринта и независимо открыл CY Lee как алгоритм маршрутизации проводов в 1961 году.

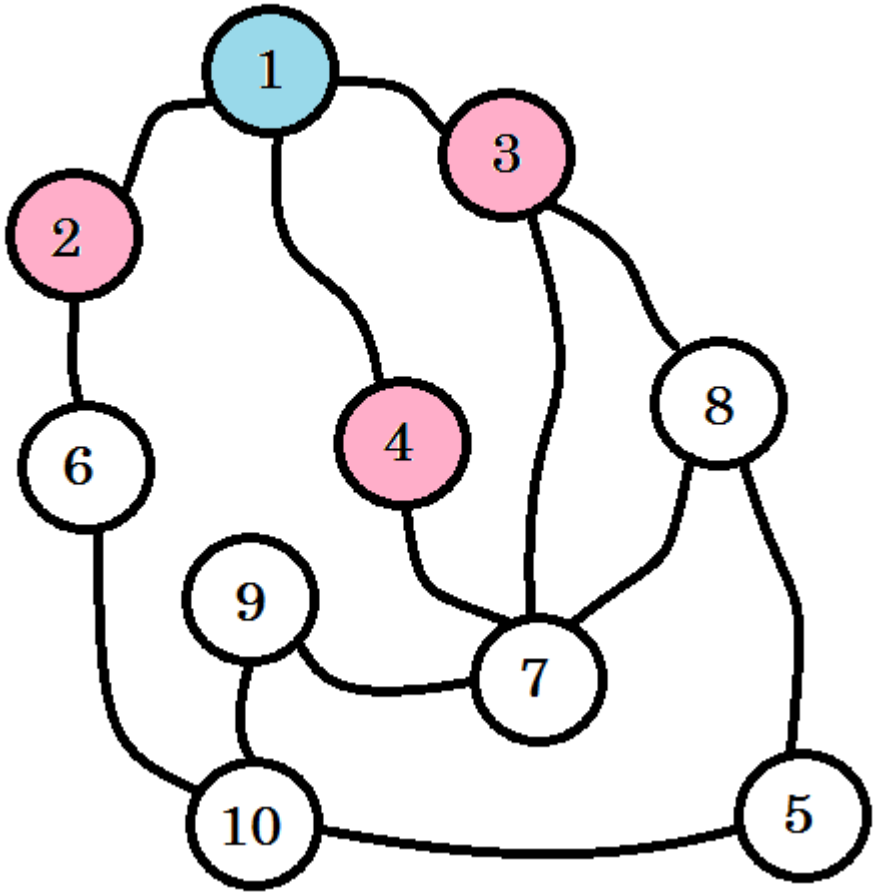
Процессы алгоритма BFS работают в этих предположениях:

1. Мы не будем проходить ни одного узла более одного раза.
2. Исходный узел или узел, с которого мы начинаем, находится на уровне 0.
3. Узлами, которые мы можем напрямую достичь от исходного узла, являются узлы уровня 1, узлы, с которыми мы можем напрямую связаться с узлами уровня 1, являются узлами уровня 2 и т. Д.
4. Уровень означает расстояние от кратчайшего пути от источника.

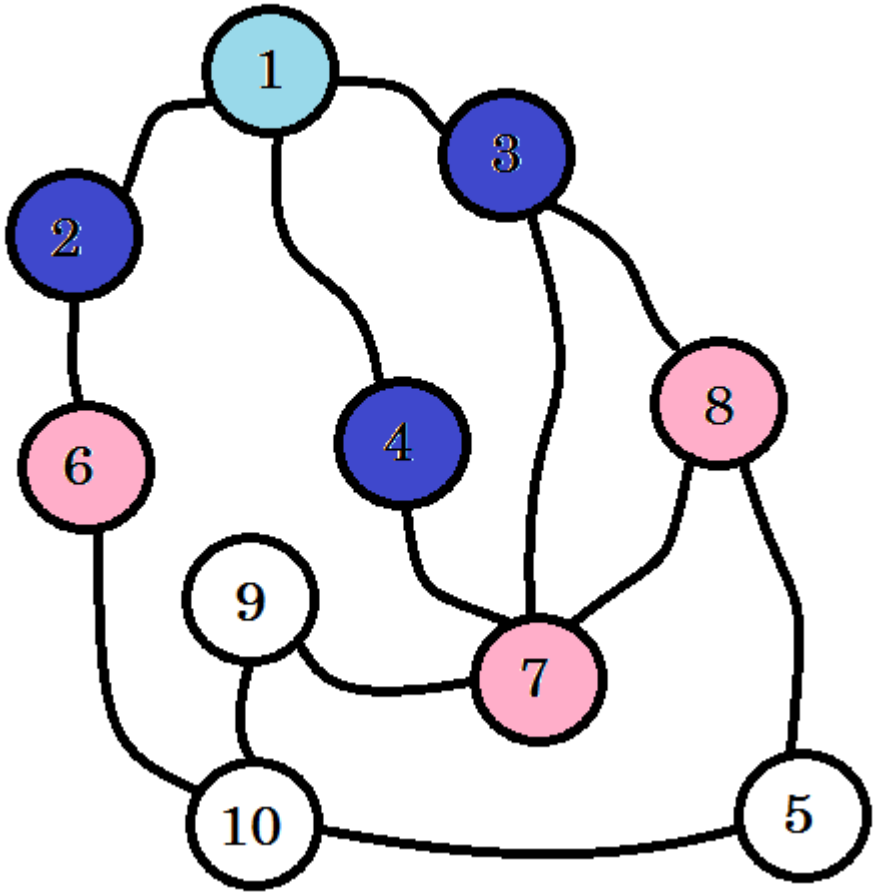
Давайте посмотрим пример:



Предположим, что этот график представляет собой соединение между несколькими городами, где каждый узел обозначает город, а край между двумя узлами означает, что существует дорога, соединяющая их. Мы хотим перейти от **узла 1** к **узлу 10** . Итак, **узел 1** является нашим **источником** , который является **уровнем 0** . Мы отмечаем **узел 1** как посещенный. Мы можем перейти к **узлу 2** , **узлу 3** и **узлу 4** отсюда. Таким образом, они будут **уровнями (0 + 1)** = **уровня 1** . Теперь мы помечаем их как посетили и работаем с ними.

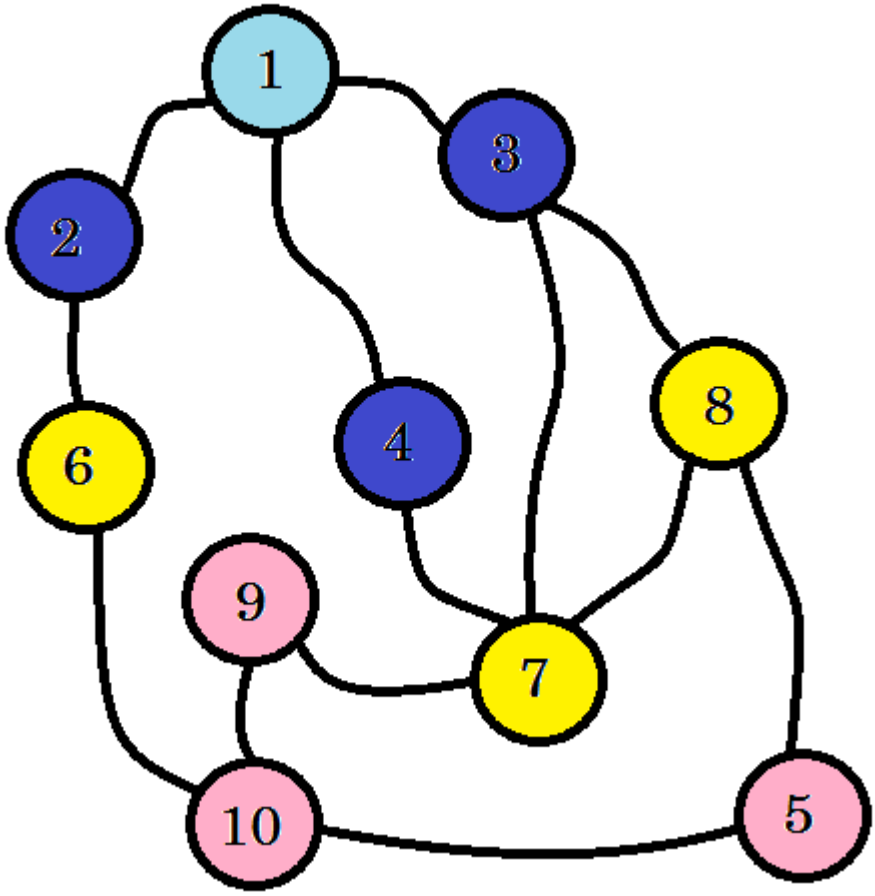


Посещаются цветные узлы. Узлы, с которыми мы сейчас работаем, будут отмечены розовым. Мы не будем посещать один и тот же узел дважды. Из **узла 2** , **узла 3** и **узла 4** мы можем перейти к **узлу 6, узлу 7** и **узлу 8** . Отметьте их как посетили. Уровень этих узлов будет **уровнем (1 + 1)** = **уровень 2** .

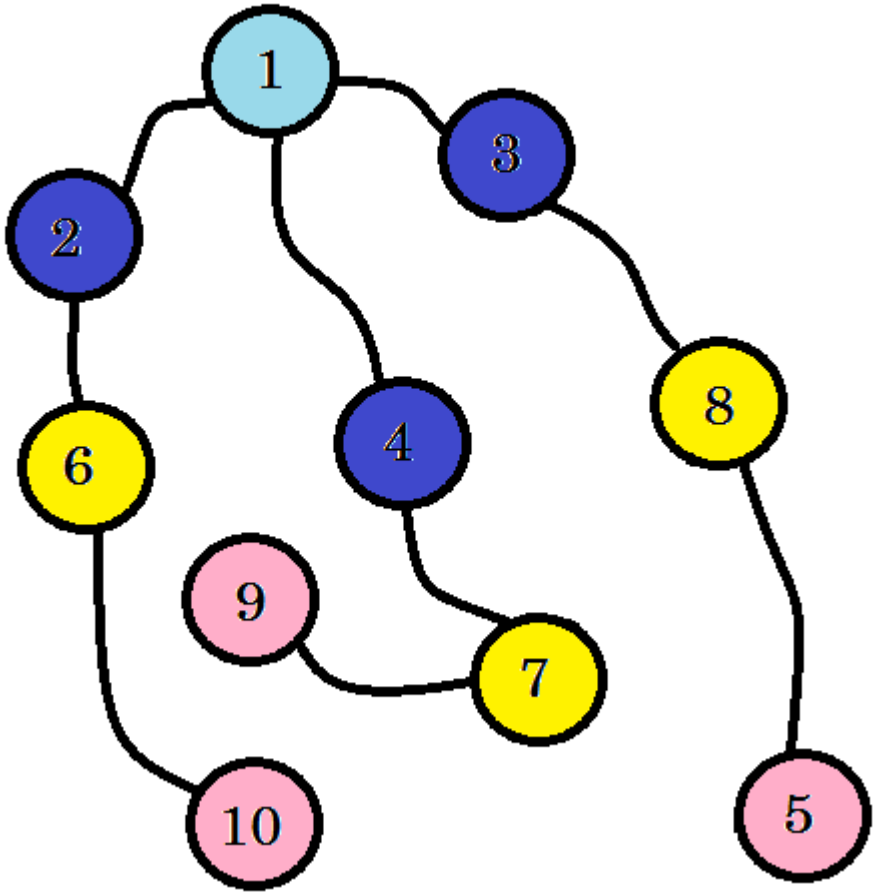


Если вы не заметили, уровень узлов просто обозначает кратчайшее расстояние пути от **источника** . Например: мы нашли **узел 8** на **уровне 2** . Таким образом, расстояние от **источника** до **узла 8** равно **2** .

Мы еще не достигли нашего целевого узла, то есть **узла 10** . Итак, давайте посмотрим на следующие узлы. мы можем напрямую перейти от **узла 6** , **узла 7** и **узла 8** .



Мы видим, что мы нашли **узел 10** на **уровне 3** . Таким образом, кратчайший путь от **источника** к **узлу 10** равен **3.** Мы искали уровень графа по уровню и находили кратчайший путь. Теперь давайте удалим те грани, которые мы не использовали:



После удаления краев, которые мы не использовали, мы получаем дерево, называемое деревом BFS. Это дерево показывает кратчайший путь от **источника** ко всем другим узлам.

Поэтому наша задача будет состоять в том, чтобы перейти от **источника** к узлам **уровня 1** . Затем от узлов **уровня 1** до **уровня 2** и так далее, пока мы не достигнем нашего пункта назначения. Мы можем использовать *очередь* для хранения узлов, которые мы будем обрабатывать. То есть для каждого узла, с которым мы собираемся работать, мы будем выталкивать все остальные узлы, которые могут быть напрямую перемещены и еще не пройдены в очереди.

Моделирование нашего примера:

Сначала мы подталкиваем источник в очередь. Наша очередь будет выглядеть так:

```
 front
+-----+
|  1  |
+-----+
```

Уровень **узла 1** будет равен 0. **level [1]** = **0** . Теперь мы начинаем нашу BFS. Сначала мы выставляем узел из нашей очереди. Мы получаем **узел 1** . Мы можем перейти к **узлу 4** ,

узлу 3 и **узлу 2** из этого. Мы достигли этих узлов из **узла 1** . Таким образом, **уровень [4]** = **уровень [3]** = **уровень [2]** = **уровень [1]** + **1** = **1** . Теперь мы отмечаем их как посещаемые и помещаем их в очередь.

```
                  front
+-----+  +-----+  +-----+
|  2  |  |  3  |  |  4  |
+-----+  +-----+  +-----+
```

Теперь мы поп- **узел 4** и работаем с ним. Мы можем перейти к **узлу 7** из **узла 4** . **уровень [7]** = **уровень [4]** + **1** = **2** . Мы отмечаем **узел 7** как посещенный и нажимаем его в очереди.

```
                  front
+-----+  +-----+  +-----+
|  7  |  |  2  |  |  3  |
+-----+  +-----+  +-----+
```

Из **узла 3** мы можем перейти к **узлу 7** и **узлу 8** . Поскольку мы уже пометили **узел 7** как посещенный, мы помечаем **узел 8** как посещенный, мы меняем **уровень [8]** = **уровень [3]** + **1** = **2** . Мы вставляем **узел 8** в очередь.

```
                  front
+-----+  +-----+  +-----+
|  6  |  |  7  |  |  2  |
+-----+  +-----+  +-----+
```

Этот процесс будет продолжаться до тех пор, пока мы не достигнем места назначения, или очередь не станет пустой. Массив **уровня** предоставит нам расстояние от кратчайшего пути от **источника** . Мы можем инициализировать массив **уровней** с *бесконечным* значением, что будет означать, что узлы еще не посещены. Наш псевдокод будет:

```
Procedure BFS(Graph, source):
Q = queue();
level[] = infinity
level[source] := 0
Q.push(source)
while Q is not empty
    u -> Q.pop()
    for all edges from u to v in Adjacency list
        if level[v] == infinity
            level[v] := level[u] + 1
            Q.push(v)
        end if
    end for
end while
Return level
```

Итерируя через массив **уровней** , мы можем узнать расстояние каждого узла от источника. Например: расстояние **узла 10** от **источника** будет сохранено на **уровне [10]** .

Иногда нам может потребоваться напечатать не только кратчайшее расстояние, но и путь,

по которому мы можем перейти к нашему узлу, который находится у вас из **источника** . Для этого нам нужно сохранить **родительский** массив. **parent [source]** будет NULL. Для каждого обновления в массиве **уровней** мы просто добавим `parent[v] := u` в наш псевдокод внутри цикла for. По завершении BFS, чтобы найти путь, мы перейдем назад к **родительскому** массиву, пока не достигнем **источника,** который будет обозначен значением NULL. Псевдокод будет:

```
Procedure PrintPath(u):  //recursive    |   Procedure PrintPath(u):   //iterative
if parent[u] is not equal to null       |   S =  Stack()
    PrintPath(parent[u])                |   while parent[u] is not equal to null
end if                                  |       S.push(u)
print -> u                              |       u := parent[u]
                                        |   end while
                                        |   while S is not empty
                                        |       print -> S.pop
                                        |   end while
```

Сложность:

Мы посетили каждый узел один раз и каждый ребро один раз. Таким образом, сложность будет **O (V + E),** где **V** - количество узлов, а **E** - количество ребер.

Поиск кратчайшего пути из источника в двумерном графике

В большинстве случаев нам нужно будет найти кратчайший путь от одного источника ко всем другим узлам или конкретному узлу в двумерном графике. Скажем, например: мы хотим узнать, сколько ходов требуется для того, чтобы рыцарь достиг определенного квадрата в шахматной доске, или у нас есть массив, в котором некоторые ячейки заблокированы, нам нужно выяснить кратчайший путь из одной ячейки в другую , Мы можем перемещаться только по горизонтали и по вертикали. Также возможны диагональные движения. Для этих случаев мы можем преобразовать квадраты или ячейки в узлы и легко решить эти проблемы с помощью BFS. Теперь нашими **посетителями** , **родителями** и **уровнями** будут 2D массивы. Для каждого узла мы рассмотрим все возможные шаги. Чтобы найти расстояние до определенного узла, мы также проверим, дошли ли мы до пункта назначения.

Будет еще одна вещь, называемая массивом направлений. Это просто сохранит все возможные комбинации направлений, на которые мы можем пойти. Скажем, для горизонтальных и вертикальных движений наши массивы направлений будут:

```
+----+-----+-----+-----+-----+
| dx |  1  |  -1 |  0  |  0  |
+----+-----+-----+-----+-----+
| dy |  0  |   0 |  1  |  -1 |
+----+-----+-----+-----+-----+
```

Здесь *dx* представляет перемещение по оси x, а *dy* представляет перемещение по оси y. Опять же эта часть является необязательной. Вы также можете написать все возможные

комбинации отдельно. Но с ним проще справиться с использованием массива направлений. Могут быть больше и даже разные комбинации для диагональных движений или движения рыцарей.

Дополнительную часть, о которой мы должны помнить, это:

- Если какая-либо ячейка заблокирована, для всех возможных ходов мы проверим, заблокирована ли ячейка или нет.
- Мы также проверим, не вышли ли мы за границы, то есть пересекли границы массива.
- Будет указано количество строк и столбцов.

Наш псевдокод будет:

```
Procedure BFS2D(Graph, blocksign, row, column):
for i from 1 to row
    for j from 1 to column
        visited[i][j] := false
    end for
end for
visited[source.x][source.y] := true
level[source.x][source.y] := 0
Q = queue()
Q.push(source)
m := dx.size
while Q is not empty
    top := Q.pop
    for i from 1 to m
        temp.x := top.x + dx[i]
        temp.y := top.y + dy[i]
        if temp is inside the row and column and top doesn't equal to blocksign
            visited[temp.x][temp.y] := true
            level[temp.x][temp.y] := level[top.x][top.y] + 1
            Q.push(temp)
        end if
    end for
end while
Return level
```

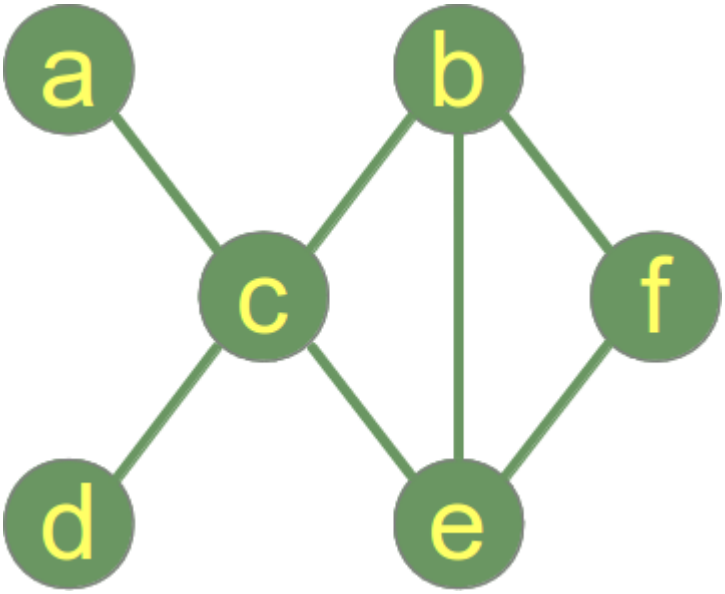
Как мы уже говорили ранее, BFS работает только для невзвешенных графиков. Для взвешенных графов нам понадобится алгоритм Дейкстры . Для отрицательных краевых циклов нам нужен алгоритм Беллмана-Форда . Снова этот алгоритм является алгоритмом кратчайшего пути с одним источником. Если нам нужно выяснить расстояние от каждого узла до всех других узлов, нам понадобится алгоритм Флойда-Варшалла .

Связанные компоненты ненаправленного графика с использованием BFS.

BFS можно использовать для поиска связанных компонентов неориентированного графа . Мы также можем найти, связан ли данный граф или нет. Наше последующее обсуждение предполагает, что мы имеем дело с неориентированными графами. Определение связного графа:

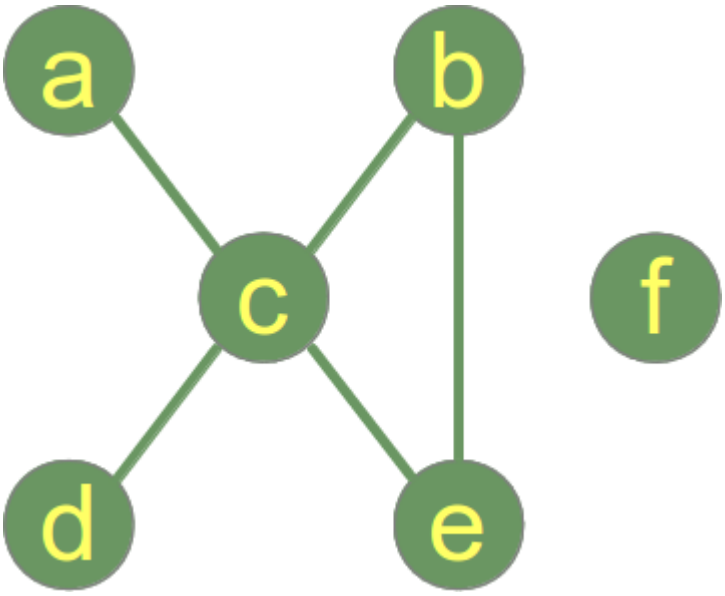
График связан, если есть путь между каждой парой вершин.

Ниже приводится **связанный граф** .



Следующий график **не подключен** и имеет 2 подключенных компонента:

1. Подключенный компонент 1: {a, b, c, d, e}
2. Подключенный компонент 2: {f}



BFS - алгоритм обхода графика. Итак, начиная с случайного узла источника, если по окончании алгоритма все узлы посещаются, то граф подключен, иначе он не подключен.

PseudoCode для алгоритма.

```
boolean isConnected(Graph g)
{
 BFS(v)//v is a random source node.
 if(allVisited(g))
 {
```

```
  return true;
 }
 else return false;
}
```

C для определения того, подключен ли неориентированный граф или нет:

```
#include<stdio.h>
#include<stdlib.h>
#define MAXVERTICES 100

void enqueue(int);
int deque();
int isConnected(char **graph,int noOfVertices);
void BFS(char **graph,int vertex,int noOfVertices);
int count = 0;
//Queue node depicts a single Queue element
//It is NOT a graph node.
struct node
{
    int v;
    struct node *next;
};

typedef struct node Node;
typedef struct node *Nodeptr;

Nodeptr Qfront = NULL;
Nodeptr Qrear = NULL;
char *visited;//array that keeps track of visited vertices.

int main()
{
    int n,e;//n is number of vertices, e is number of edges.
    int i,j;
    char **graph;//adjacency matrix

    printf("Enter number of vertices:");
    scanf("%d",&n);

    if(n < 0 || n > MAXVERTICES)
    {
     fprintf(stderr, "Please enter a valid positive integer from 1 to %d",MAXVERTICES);
     return -1;
    }

    graph = malloc(n * sizeof(char *));
    visited = malloc(n*sizeof(char));

    for(i = 0;i < n;++i)
    {
        graph[i] = malloc(n*sizeof(int));
        visited[i] = 'N';//initially all vertices are not visited.
        for(j = 0;j < n;++j)
            graph[i][j] = 0;
    }

    printf("enter number of edges and then enter them in pairs:");
    scanf("%d",&e);
```

```
    for(i = 0;i < e;++i)
    {
        int u,v;
        scanf("%d%d",&u,&v);
        graph[u-1][v-1] = 1;
        graph[v-1][u-1] = 1;
    }

    if(isConnected(graph,n))
        printf("The graph is connected");
    else printf("The graph is NOT connected\n");
}

void enqueue(int vertex)
{
    if(Qfront == NULL)
    {
        Qfront = malloc(sizeof(Node));
        Qfront->v = vertex;
        Qfront->next = NULL;
        Qrear = Qfront;
    }
    else
    {
        Nodeptr newNode = malloc(sizeof(Node));
        newNode->v = vertex;
        newNode->next = NULL;
        Qrear->next = newNode;
        Qrear = newNode;
    }
}

int deque()
{
    if(Qfront == NULL)
    {
        printf("Q is empty , returning -1\n");
        return -1;
    }
    else
    {
        int v = Qfront->v;
        Nodeptr temp= Qfront;
        if(Qfront == Qrear)
        {
            Qfront = Qfront->next;
            Qrear = NULL;
        }
        else
            Qfront = Qfront->next;

        free(temp);
        return v;
    }
}

int isConnected(char **graph,int noOfVertices)
{
    int i;

    //let random source vertex be vertex 0;
```

```
    BFS(graph,0,noOfVertices);

    for(i = 0;i < noOfVertices;++i)
        if(visited[i] == 'N')
         return 0;//0 implies false;

    return 1;//1 implies true;
}

void BFS(char **graph,int v,int noOfVertices)
{
    int i,vertex;
    visited[v] = 'Y';
    enqueue(v);
    while((vertex = deque()) != -1)
    {
        for(i = 0;i < noOfVertices;++i)
            if(graph[vertex][i] == 1 && visited[i] == 'N')
            {
                enqueue(i);
                visited[i] = 'Y';
            }
    }
}
```

Для поиска всех подключенных компонентов неориентированного графа нам нужно только добавить 2 строки кода в функцию BFS. Идея состоит в том, чтобы вызвать функцию BFS до тех пор, пока не будут посещены все вершины.

Строки, которые необходимо добавить:

```
printf("\nConnected component %d\n",++count);
//count is a global variable initialized to 0
//add this as first line to BFS function
```

А ТАКЖЕ

```
printf("%d ",vertex+1);
add this as first line of while loop in BFS
```

и мы определяем следующую функцию:

```
void listConnectedComponents(char **graph,int noOfVertices)
{
    int i;
    for(i = 0;i < noOfVertices;++i)
    {
        if(visited[i] == 'N')
            BFS(graph,i,noOfVertices);

    }
}
```

Прочитайте Поиск по ширине онлайн: https://riptutorial.com/ru/algorithm/topic/7215/поиск-по-

ширине

глава 42: полиномиально ограниченный алгоритм для минимальной вершины

Вступление

Это полиномиальный алгоритм для получения минимального вершинного покрытия связанного неориентированного графа. Сложность времени этого алгоритма O (n2)

параметры

переменная	Имея в виду
г	Ввод подключенного ненаправленного графика
Икс	Набор вершин
С	Конечный набор вершин

замечания

Первое, что вам нужно сделать в этом алгоритме, чтобы получить все вершины графа, отсортированные по убыванию в зависимости от степени.

После этого вы выполняете итерацию по ним и добавляете каждый из них в конечные вершины, которые не имеют соседних вершин в этом наборе.

В завершающей стадии итерации на конечном наборе вершин и удалим все вершины, которые имеют одну из своих смежных вершин в этом множестве.

Examples

Алгоритм Псевдокод

Алгоритм PMinVertexCover (граф G)

Ввод связанного графа G

Минимальный набор вершинной вершины C

```
Set C <- new Set<Vertex>()

Set X <- new Set<Vertex>()

X <- G.getAllVerticiesArrangedDescendinglyByDegree()

for v in X do
    List<Vertex> adjacentVertices1 <- G.getAdjacent(v)

    if !C contains any of adjacentVertices1 then

        C.add(v)

for vertex in C do

    List<vertex> adjacentVertices2 <- G.adjacentVertecies(vertex)

    if C contains any of adjacentVertices2 then

        C.remove(vertex)


return C
```

C - минимальное вершинное накрытие графа G

мы можем использовать сортировку ведра для сортировки вершин в соответствии с его степенью, поскольку максимальное значение степеней (n-1), где n - количество вершин, тогда сложность времени сортировки будет равна O (n)

Прочитайте полиномиально ограниченный алгоритм для минимальной вершины онлайн: https://riptutorial.com/ru/algorithm/topic/9535/полиномиально-ограниченный-алгоритм-для-минимальной-вершины

глава 43: Приложения динамического программирования

Вступление

Основная идея динамического программирования - преодоление сложной проблемы до нескольких небольших и простых задач, которые повторяются. Если вы можете идентифицировать простую подзадачу, которая многократно вычисляется, существует вероятность того, что проблема будет связана с динамическим программированием.

Поскольку этот раздел называется « *Приложения динамического программирования»* , он будет больше ориентироваться на приложения, а не на процесс создания динамических алгоритмов программирования.

замечания

Определения

Memoization - метод оптимизации, используемый прежде всего для ускорения компьютерных программ путем хранения результатов дорогих вызовов функций и возврата результата кэширования при повторных входах.

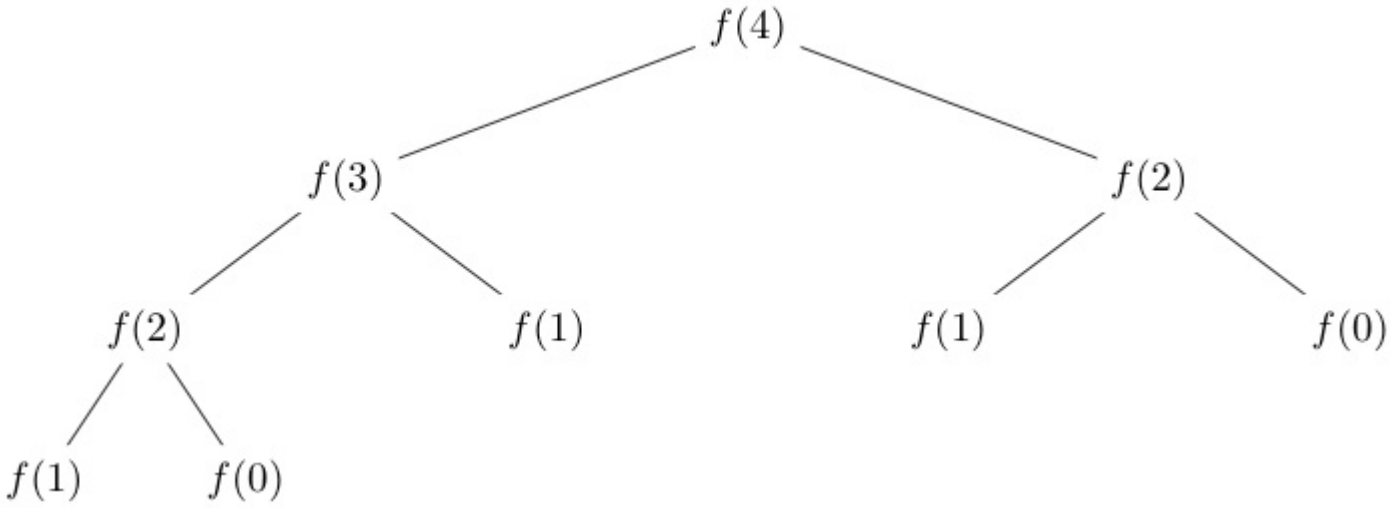
Динамическое программирование - метод решения сложной задачи, разбивая его на набор простых подзадач, решая каждую из этих подзадач только один раз и сохраняя их решения.

Examples

Числа Фибоначчи

Числа Фибоначчи являются основным объектом динамического программирования, поскольку традиционный рекурсивный подход делает много повторных вычислений. В этих примерах я буду использовать базовый случай `f(0) = f(1) = 1` .

Вот пример рекурсивного дерева для `fibonacci(4)` , обратите внимание на повторные вычисления:



Нединамическое программирование `O(2^n)` Сложность выполнения, `O(n)` Степень стека

```
def fibonacci(n):
    if n < 2:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

Это самый интуитивный способ написать проблему. В лучшем случае пространство стека будет `O(n)` когда вы опускаете первую рекурсивную ветвь, делающую вызовы `fibonacci(n-1)` пока не нажмете базовый случай `n < 2` .

Доказательство сложности выполнения `O(2^n)` которое можно увидеть здесь: Вычислительная сложность последовательности Фибоначчи . Основной момент следует отметить, что время выполнения является экспоненциальным, что означает, что время выполнения для этого будет удвоено для каждого последующего термина, `fibonacci(15)` будут в два раза длиннее `fibonacci(14)` .

Memoized `O(n)` Сложность выполнения, `O(n)` Сложность пространства, `O(n)` Сложность стека

```
memo = []
memo.append(1) # f(1) = 1
memo.append(1) # f(2) = 1

def fibonacci(n):
    if len(memo) > n:
        return memo[n]

    result = fibonacci(n-1) + fibonacci(n-2)
    memo.append(result) # f(n) = f(n-1) + f(n-2)
    return result
```

С мемуаризованным подходом мы вводим массив, который можно рассматривать как все предыдущие вызовы функций. Местоположение `memo[n]` является результатом функции

`fibonacci(n)` . Это позволяет нам торговать пространственной сложностью `O(n)` для времени выполнения `O(n)` поскольку нам больше не нужно вычислять дублированные вызовы функций.

Итеративное динамическое программирование `O(n)` Сложность выполнения, `O(n)` Сложность пространства, Нет рекурсивного стека

```
def fibonacci(n):
    memo = [1,1] # f(0) = 1, f(1) = 1

    for i in range(2, n+1):
         memo.append(memo[i-1] + memo[i-2])

    return memo[n]
```

Если мы разложим проблему на ее основные элементы, вы заметите, что для вычисления `fibonacci(n)` нам нужны `fibonacci(n-1)` и `fibonacci(n-2)` . Также мы можем заметить, что наш базовый случай появится в конце этого рекурсивного дерева, как показано выше.

С этой информацией теперь имеет смысл вычислить решение в обратном направлении, начиная с базовых корпусов и работать вверх. Теперь, чтобы вычислить `fibonacci(n)` мы сначала вычислим **все** числа фибоначчи с точностью до и через `n` .

Это основное преимущество здесь в том, что теперь мы исключили рекурсивный стек, сохранив время выполнения `O(n)` . К сожалению, мы все еще имеем сложность `O(n)` но это также можно изменить.

Расширенное итеративное динамическое программирование `O(n)` Сложность выполнения, `O(1)` Сложность пространства, Без рекурсивного стека

```
def fibonacci(n):
    memo = [1,1] # f(1) = 1, f(2) = 1

    for i in range (2, n):
        memo[i%2] = memo[0] + memo[1]

    return memo[n%2]
```

Как отмечалось выше, итеративный подход к динамическому программированию начинается с базовых случаев и работает до конечного результата. Главное наблюдение, которое нужно сделать для того, чтобы добраться до сложности пространства до `O(1)` (константа), - это то же самое наблюдение, которое мы сделали для рекурсивного стека - нам нужны только `fibonacci(n-1)` и `fibonacci(n-2)` для построения `fibonacci(n)` . Это означает, что нам нужно сохранить результаты для `fibonacci(n-1)` и `fibonacci(n-2)` в какой бы то ни было точке нашей итерации.

Чтобы сохранить эти последние 2 результата, я использую массив размером 2 и просто

переворачиваю индекс, который я назначаю, используя `i % 2` который будет чередоваться следующим образом: `0, 1, 0, 1, 0, 1, ..., i % 2` .

Я добавляю оба индекса массива вместе, потому что мы знаем, что сложение является коммутативным (`5 + 6 = 11` и `6 + 5 == 11`). Затем результат присваивается старшему из двух точек (обозначается `i % 2`). Окончательный результат затем сохраняется в позиции `n%2`

Заметки

- Важно отметить, что иногда бывает лучше придумать итеративное memoized решение для функций, которые многократно выполняют большие вычисления, поскольку вы создадите кеш ответа на вызовы функций, а последующие вызовы могут быть `O(1)` if он уже был вычислен.

Прочитайте Приложения динамического программирования онлайн:
https://riptutorial.com/ru/algorithm/topic/10596/приложения-динамического-программирования

глава 44: Применение жадности

замечания

источники

1. Приведенные выше примеры взяты из лекций из лекции, которая преподавалась в 2008 году в Бонне, Германия. Они в перспективе основаны на книге « Алгоритм дизайна » Джона Клейнберга и Евы Тардос:

Examples

Автомат для продажи билетов

Первый простой пример:

У вас есть автомат для билетов, который дает обмен в монетах со значениями 1, 2, 5, 10 и 20. Разделение обмена можно рассматривать как серию капель монет до тех пор, пока не будет отменено правильное значение. Мы говорим, что распределение является **оптимальным,** когда его **количество монет минимально** для его значения.

Пусть `M` в `[1,50]` будет ценой для билета `T` и `P` в `[1,50]` деньгами, которые кто-то заплатил за `T` , с `P >= M` Пусть `D=PM` . Определим **преимущество** шага как разница между `D` и `Dc` с `c` в монете автомат раздаточным в этом шаге.

Метод **жадности** для обмена - это следующий псевдо-алгоритмический подход:

Шаг 1: в то время как `D > 20` выдают 20 монет и устанавливают `D = D - 20`
Шаг 2: в то время как `D > 10` распределяет 10 монет и устанавливает `D = D - 10`
Шаг 3: в то время как `D > 5` выдают 5 монет и устанавливают `D = D - 5`
Шаг 4: в то время как `D > 2` выдают 2 монеты и устанавливают `D = D - 2`
Шаг 5: пока `D > 1` раздаст 1 монету и установите `D = D - 1`

Впоследствии сумма всех монет явно равна `D` Его **жадный алгоритм,** потому что после каждого шага и после каждого повторения шага преимущество максимизируется. Мы не можем отказаться от другой монеты с более высокой выгодой.

Теперь автоматика билетов как программа (в C ++):

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
```

```
using namespace std;

// read some coin values, sort them descending,
// purge copies and guaratee the 1 coin is in it
std::vector<unsigned int> readInCoinValues();

int main()
{
    std::vector<unsigned int> coinValues;   // Array of coin values ascending
    int ticketPrice;                        // M in example
    int paidMoney;                          // P in example

    // generate coin values
    coinValues = readInCoinValues();

    cout << "ticket price: ";
    cin >> ticketPrice;

    cout << "money paid: ";
    cin >> paidMoney;

    if(paidMoney <= ticketPrice)
    {
        cout << "No exchange money" << endl;
        return 1;
    }

    int diffValue = paidMoney - ticketPrice;

    // Here starts greedy

    // we save how many coins we have to give out
    std::vector<unsigned int> coinCount;

    for(auto coinValue  = coinValues.begin();
             coinValue != coinValues.end(); ++coinValue)
    {
        int countCoins = 0;

        while (diffValue >= *coinValue)
        {
            diffValue -= *coinValue;
            countCoins++;
        }

        coinCount.push_back(countCoins);
    }

    // print out result
    cout << "the difference " << paidMoney - ticketPrice
         << " is paid with: " << endl;

    for(unsigned int i=0; i < coinValues.size(); ++i)
    {
        if(coinCount[i] > 0)
            cout << coinCount[i] << " coins with value "
                 << coinValues[i] << endl;
    }

    return 0;
}
```

```
std::vector<unsigned int> readInCoinValues()
{
    // coin values
    std::vector<unsigned int> coinValues;

    // make sure 1 is in vectore
    coinValues.push_back(1);

    // read in coin values (attention: error handling is omitted)
    while(true)
    {
        int coinValue;

        cout << "Coin value (<1 to stop): ";
        cin >> coinValue;

        if(coinValue > 0)
            coinValues.push_back(coinValue);

        else
            break;
    }

    // sort values
    sort(coinValues.begin(), coinValues.end(), std::greater<int>());

    // erase copies of same value
    auto last = std::unique(coinValues.begin(), coinValues.end());
    coinValues.erase(last, coinValues.end());

    // print array
    cout << "Coin values: ";

    for(auto i : coinValues)
        cout << i << " ";

    cout << endl;

    return coinValues;
}
```

Имейте в виду, что теперь есть проверка ввода, чтобы простой пример. Один пример:

```
Coin value (<1 to stop): 2
Coin value (<1 to stop): 4
Coin value (<1 to stop): 7
Coin value (<1 to stop): 9
Coin value (<1 to stop): 14
Coin value (<1 to stop): 4
Coin value (<1 to stop): 0
Coin values: 14 9 7 4 2 1
ticket price: 34
money paid: 67
the difference 33 is paid with:
2 coins with value 14
1 coins with value 4
1 coins with value 1
```

Пока `1` находится в значениях монеты, мы теперь, что алгоритм закончится, потому что:

- `D` строго уменьшается с каждым шагом
- `D` никогда не `>0` и меньше, чем самая маленькая монета `1` в то же время

Но алгоритм имеет две ошибки:

1. Пусть `C` - наибольшее значение монеты. Время выполнения является только полиномиальным, если `D/C` является полиномиальным, поскольку представление `D` использует только биты `log D` а время выполнения по крайней мере линейно в `D/C`
2. На каждом шаге наш алгоритм выбирает локальный оптимум. Но этого недостаточно, чтобы сказать, что алгоритм находит глобальное оптимальное решение (см. Больше информации здесь или в Книге Корте и Vygen).

Простой пример счетчика: монеты составляют `1,3,4` и `D=6` . Оптимальным решением является, очевидно, две монеты значения `3` но жадный выбор `4` на первом этапе, поэтому ему нужно выбрать `1` на шаге два и три. Поэтому он не дает оптимального суждения. Возможный оптимальный алгоритм для этого примера основан на **динамическом программировании** .

Планирование интервалов

У нас есть набор заданий `J={a,b,c,d,e,f,g}` . Пусть `j in J` - задание, чем его начало в `sj` и заканчивается в `fj` . Два задания совместимы, если они не перекрываются. Рисунок как пример: Цель состоит в том, чтобы найти **максимальное подмножество взаимно совместимых заданий** . Есть несколько жадных подходов к этой проблеме:

1. **Самое раннее время начала** : Рассмотрите задания в порядке возрастания `sj`
2. **Самое раннее время окончания** : рассмотрите задания в порядке возрастания `fj`
3. **Самый короткий интервал** : рассмотрите задания в порядке возрастания `fj-sj`
4. **Наименьшие конфликты** : для каждого задания `j` подсчитывайте количество противоречивых заданий `cj`

Вопрос в том, какой подход действительно успешный. **Раннее время начала,** конечно, нет, вот примерный пример **Самый короткий интервал** не является оптимальным и **наименьшее количество конфликтов** может действительно оказаться оптимальным, но здесь есть проблема для такого подхода: Что оставляет нас с **самым ранним временем окончания** . Псевдокод довольно простой:

1. Сортировка заданий по времени окончания, так что `f1<=f2<=...<=fn`
2. Пусть `A` - пустое множество
3. для `j=1` до `n` , если `j` совместима со **всех** рабочих мест в `A` множества `A=A+{j}`
4. `A` - **максимальное подмножество взаимно совместимых заданий**

Или как программа на C ++:

```
#include <iostream>
#include <utility>
#include <tuple>
#include <vector>
#include <algorithm>

const int jobCnt = 10;

// Job start times
const int startTimes[] = { 2, 3, 1, 4, 3, 2, 6, 7, 8, 9};

// Job end times
const int endTimes[]   = { 4, 4, 3, 5, 5, 5, 8, 9, 9, 10};

using namespace std;

int main()
{
    vector<pair<int,int>> jobs;

    for(int i=0; i<jobCnt; ++i)
        jobs.push_back(make_pair(startTimes[i], endTimes[i]));

    // step 1: sort
    sort(jobs.begin(), jobs.end(),[](pair<int,int> p1, pair<int,int> p2)
                                    { return p1.second < p2.second; });

    // step 2: empty set A
    vector<int> A;

    // step 3:
    for(int i=0; i<jobCnt; ++i)
    {
        auto job = jobs[i];
        bool isCompatible = true;

        for(auto jobIndex : A)
        {
            // test whether the actual job and the job from A are incompatible
            if(job.second >= jobs[jobIndex].first &&
               job.first  <= jobs[jobIndex].second)
            {
                isCompatible = false;
                break;
            }
        }

        if(isCompatible)
            A.push_back(i);
    }

    //step 4: print A
    cout << "Compatible: ";

    for(auto i : A)
        cout << "(" << jobs[i].first << "," << jobs[i].second << ") ";
    cout << endl;

    return 0;
}
```

Выход для этого примера: `Compatible: (1,3) (4,5) (6,8) (9,10)`

Реализация алгоритма явно находится в Θ (n ^ 2). Существует реализация Θ (n log n), и заинтересованный читатель может продолжить чтение ниже (пример Java).

Теперь у нас есть жадный алгоритм для задачи планирования интервалов, но оптимален ли он?

Предложение: Самый жалкий алгоритм - **самое раннее время окончания** .

Доказательство: *(от противного)*

Предположим, что жадный не является оптимальным, а `i1,i2,...,ik` обозначают множество заданий, выбранных жадным. Пусть `j1,j2,...,jm` обозначает множество заданий в **оптимальном** решении с `i1=j1,i2=j2,...,ir=jr` для **максимально возможного** значения `r` .

Работа `i(r+1)` существует и заканчивается до `j(r+1)` (самое раннее завершение). Но, чем `j1,j2,...,jr,i(r+1),j(r+2),...,jm` также **оптимальное** решение и для всех `k` из `[1,(r+1)] jk=ik` . `r` . е. **противоречие** с максимальностью г. Это завершает доказательство.

Этот второй пример демонстрирует, что обычно существует много возможных жадных стратегий, но только некоторые или даже не могут найти оптимальное решение в каждом случае.

Ниже приведена программа Java, которая работает в Θ (n log n)

```
import java.util.Arrays;
import java.util.Comparator;

class Job
{
    int start, finish, profit;

    Job(int start, int finish, int profit)
    {
        this.start = start;
        this.finish = finish;
        this.profit = profit;
    }
}


class JobComparator implements Comparator<Job>
{
    public int compare(Job a, Job b)
    {
        return a.finish < b.finish ? -1 : a.finish == b.finish ? 0 : 1;
    }
}

public class WeightedIntervalScheduling
{
```

```
    static public int binarySearch(Job jobs[], int index)
    {
        int lo = 0, hi = index - 1;

        while (lo <= hi)
        {
            int mid = (lo + hi) / 2;
            if (jobs[mid].finish <= jobs[index].start)
            {
                if (jobs[mid + 1].finish <= jobs[index].start)
                    lo = mid + 1;
                else
                    return mid;
            }
            else
                hi = mid - 1;
        }

        return -1;
    }

    static public int schedule(Job jobs[])
    {
        Arrays.sort(jobs, new JobComparator());

        int n = jobs.length;
        int table[] = new int[n];
        table[0] = jobs[0].profit;

        for (int i=1; i<n; i++)
        {
            int inclProf = jobs[i].profit;
            int l = binarySearch(jobs, i);
            if (l != -1)
                inclProf += table[l];

            table[i] = Math.max(inclProf, table[i-1]);
        }

        return table[n-1];
    }

    public static void main(String[] args)
    {
        Job jobs[] = {new Job(1, 2, 50), new Job(3, 5, 20),
                      new Job(6, 19, 100), new Job(2, 100, 200)};

        System.out.println("Optimal profit is " + schedule(jobs));
    }
}
```

И ожидаемый результат:

```
Optimal profit is 250
```

Минимизация задержек

Существует множество проблем, минимизирующих опоздание, здесь у нас есть

единственный ресурс, который может обрабатывать только одно задание за раз. Задание `j` требует `tj` единиц времени обработки и ожидается в момент `dj` . если `j` начинается со временем `sj` оно будет завершено во время `fj=sj+tj` . Определим латентность `L=max{0,fj-dh}` для всех `j` . Цель состоит в том, чтобы свести к минимуму **максимальную задержку** L.

	1	2	3	4	5	6
tj	3	2	1	4	3	2
dj	6	8	9	9	10	11

работа	3	2	2	5	5	5	4	4	4	4	1	1	1	6	6
Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Lj	-8		-5			-4				1			**7**		4

Решение `L=7` , очевидно, не является оптимальным. Давайте посмотрим на некоторые жадные стратегии:

1. Самое короткое время **обработки** : расписание заданий в порядке возрастания og время обработки j`
2. **Самый ранний крайний срок** : расписание заданий в порядке возрастания `dj`
3. **Наименьший спад** : **задание** расписания в порядке возрастания slack `dj-tj`

Легко видеть, что **кратчайшее время обработки вначале** не является оптимальным, хороший пример счетчика

	1	2
tj	1	5
dj	10	5

самое **маленькое** решение для **стека** имеет проблемы с simillar

	1	2
tj	1	5
dj	3	5

последняя стратегия выглядит действительной, поэтому мы начинаем с некоторого псевдокода:

1. Сортируйте `n` заданий по времени, чтобы `d1<=d2<=...<=dn`

2. Положим `t=0`
3. для `j=1` - `n`
    - Назначьте задание `j` интервалу `[t,t+tj]`
    - множество `sj=t` и `fj=t+tj`
    - множество `t=t+tj`
4. интервалы возврата `[s1,f1],[s2,f2],...,[sn,fn]`

И как реализация в C ++:

```
#include <iostream>
#include <utility>
#include <tuple>
#include <vector>
#include <algorithm>

const int jobCnt = 10;

// Job start times
const int processTimes[] = { 2, 3, 1, 4, 3, 2, 3, 5, 2, 1};

// Job end times
const int dueTimes[]     = { 4, 7, 9, 13, 8, 17, 9, 11, 22, 25};

using namespace std;

int main()
{
    vector<pair<int,int>> jobs;

    for(int i=0; i<jobCnt; ++i)
        jobs.push_back(make_pair(processTimes[i], dueTimes[i]));

    // step 1: sort
    sort(jobs.begin(), jobs.end(),[](pair<int,int> p1, pair<int,int> p2)
                                    { return p1.second < p2.second; });

    // step 2: set t=0
    int t = 0;

    // step 3:
    vector<pair<int,int>> jobIntervals;

    for(int i=0; i<jobCnt; ++i)
    {
        jobIntervals.push_back(make_pair(t,t+jobs[i].first));
        t += jobs[i].first;
    }

    //step 4: print intervals
    cout << "Intervals:\n" << endl;

    int lateness = 0;

    for(int i=0; i<jobCnt; ++i)
    {
        auto pair = jobIntervals[i];

        lateness = max(lateness, pair.second-jobs[i].second);
```

```
        cout << "(" << pair.first << "," << pair.second << ") "
             << "Lateness: " << pair.second-jobs[i].second << std::endl;
    }

    cout << "\nmaximal lateness is " << lateness << endl;

    return 0;
}
```

И выход для этой программы:

```
Intervals:

(0,2)   Lateness:-2
(2,5)   Lateness:-2
(5,8)   Lateness: 0
(8,9)   Lateness: 0
(9,12)  Lateness: 3
(12,17) Lateness: 6
(17,21) Lateness: 8
(21,23) Lateness: 6
(23,25) Lateness: 3
(25,26) Lateness: 1

maximal lateness is 8
```

Время выполнения алгоритма, очевидно, Θ (n log n), поскольку сортировка является доминирующей операцией этого алгоритма. Теперь нам нужно показать, что оно оптимально. Очевидно, что оптимальный график не имеет **времени простоя** . в самом **раннем срочном** графике также нет простоя.

Предположим, что задания нумеруются так, что `d1<=d2<=...<=dn` . Мы говорим, что **инверсия** графика - это пара заданий `i` и `j` так что `i<j` но j запланировано до `i` . Из-за его определения самый **ранний крайний срок для первого** графика не имеет инверсий. Конечно, если в расписании есть инверсия, у него есть одна с двумя инвертированными заданиями, запланированными последовательно.

Предложение: замена двух смежных, перевернутых заданий уменьшает количество инверсий на **единицу** и **не увеличивает** максимальную задержку.

Доказательство. Пусть `L` - задержка перед свопом, а `M` - позднее. Поскольку обмен двумя соседними заданиями не перемещает другие задания из их положения, это `Lk=Mk` для всех `k != i,j` .

Ясно, что `Mi<=Li` , так как работы `i` получил запланировано ранее. если задание `j` опаздывает, это следует из определения:

```
                        Mj = fi-dj    (definition)
                          <= fi-di    (since i and j are exchanged)
                          <= Li
```

Это означает, что задержка после смены меньше или равна, чем раньше. Это завершает доказательство.

Предложение: самый ранний крайний срок первого графика `S` является оптимальным.

Доказательство: *(от противного)*

Предположим, что `S*` - оптимальное расписание с **наименьшим** числом инверсий. мы можем предположить, что `S*` не имеет простоя. Если `S*` не имеет инверсий, то `S=S*` и мы закончили. Если `S*` имеет инверсию, то она имеет смежную инверсию. В последнем предложении говорится, что мы можем поменять соседнюю инверсию без увеличения задержки, но с уменьшением числа инверсий. Это противоречит определению `S*` .

Сведение к минимуму проблемы с **задержкой** и связанной с ней **минимальной** проблемой **разрешения** проблемы, когда задан вопрос о минимальном расписании, есть множество приложений в реальном мире. Но обычно у вас не только одна машина, но и многие, и они выполняют одну и ту же задачу с разной скоростью. Эти проблемы очень быстрые.

Еще один интересный вопрос возникает, если мы не смотрим на **автономную** проблему, где у нас есть все задачи и данные под рукой, но в **онлайн-** варианте, где задачи появляются во время выполнения.

Автономное кэширование

Проблема кэширования возникает из-за ограничения конечного пространства. Допустим, что наш кеш `C` имеет `k` страниц. Теперь мы хотим обработать последовательность `m` запросов элементов, которые должны были быть помещены в кеш, прежде чем они будут обработаны. Конечно, если `m<=k` мы просто поместим все элементы в кеш, и он будет работать, но обычно это `m>>k` .

Мы говорим, что запрос - это **кэш** , когда элемент уже находится в кеше, иначе его называют **пропуском кеша** . В этом случае мы должны привести запрошенный элемент в кеш и выселить другого, считая, что кеш заполнен. Цель - график выселения, который **минимизирует количество выселений** .

Есть много жадных стратегий для этой проблемы, давайте посмотрим на некоторые:

1. **Первое, сначала (FIFO)** : старейшая страница выселяется
2. **Последний, первый (LIFO)** : самая новая страница выселяется
3. **Последняя недавняя** страница **(LRU)** : страница с изъятием, последний доступ которой был самым ранним
4. **Наименее часто запрашиваемая (LFU)** : страница с изъятием, которая была запрошена наименее часто
5. **Самое длинное прямое расстояние (LFD)** : вырезать страницу в кеше, которая не

запрашивается дольше в будущем.

Внимание: В следующих примерах мы выселяем страницу с наименьшим индексом, если может быть выселено более одной страницы.

Пример (FIFO)

Пусть размер кеша `k=3` - начальный кеш `a,b,c` и запрос `a,a,d,e,b,b,a,c,f,d,e,a,f,b,e,c` :

Запрос			d	e	б	б		c	e	d	e		e	б	e	c
cache 1			d	d	d	d				d	d	d	e	e	e	c
cache 2	б	б	б	e	e	e	e	c	c	c	e	e	e	б	б	б
cache 3	c	c	c	c	б	б	б	б	e	e	e				e	e
пропустить кеш			Икс	Икс	Икс		Икс	Икс	Икс	Икс	Икс	Икс	Икс	Икс	Икс	Ик

Тринадцать промахов кэша на шестнадцать запросов не очень оптимальны, попробуйте один и тот же пример с другой стратегией:

Пример (LFD)

Пусть размер кеша `k=3` - начальный кеш `a,b,c` и запрос `a,a,d,e,b,b,a,c,f,d,e,a,f,b,e,c` :

Запрос			d	e	б	б		c	e	d	e		e	б	e	c
cache 1			d	e	e	e	e	e	e	e	e	e	e	e	e	c
cache 2	б	б	б	б	б	б							e	e	e	e
cache 3	c	c	c	c	c	c	c	c	e	d	d	d	d	б	б	б
пропустить кеш			Икс	Икс			Икс		Икс	Икс			Икс	Икс		Икс

Восемь недостатков кэша намного лучше.

Selftest : Сделайте пример для LIFO, LFU, RFU и посмотрите, что произошло.

Следующая примерная программа (написанная на C ++) состоит из двух частей:

Скелет - это приложение, которое решает проблему, зависящую от выбранной жадной стратегии:

```
#include <iostream>
#include <memory>

using namespace std;

const int cacheSize     = 3;
const int requestLength = 16;

const char request[]    = {'a','a','d','e','b','b','a','c','f','d','e','a','f','b','e','c'};
char cache[]            = {'a','b','c'};

// for reset
char originalCache[]    = {'a','b','c'};


class Strategy {

public:
    Strategy(std::string name) : strategyName(name) {}
    virtual ~Strategy() = default;

    // calculate which cache place should be used
    virtual int apply(int requestIndex)                                         = 0;

    // updates information the strategy needs
    virtual void update(int cachePlace, int requestIndex, bool cacheMiss)    = 0;

    const std::string strategyName;
};

bool updateCache(int requestIndex, Strategy* strategy)
{
    // calculate where to put request
    int cachePlace = strategy->apply(requestIndex);

    // proof whether its a cache hit or a cache miss
    bool isMiss = request[requestIndex] != cache[cachePlace];

    // update strategy (for example recount distances)
    strategy->update(cachePlace, requestIndex, isMiss);

    // write to cache
    cache[cachePlace] = request[requestIndex];

    return isMiss;
}

int main()
{
    Strategy* selectedStrategy[] = { new FIFO, new LIFO, new LRU, new LFU, new LFD };

    for (int strat=0; strat < 5; ++strat)
    {
        // reset cache
        for (int i=0; i < cacheSize; ++i) cache[i] = originalCache[i];

        cout <<"\nStrategy: " << selectedStrategy[strat]->strategyName << endl;

        cout << "\nCache initial: (";
        for (int i=0; i < cacheSize-1; ++i) cout << cache[i] << ",";
        cout << cache[cacheSize-1] << ")\n\n";
```

```
        cout << "Request\t";
        for (int i=0; i < cacheSize; ++i) cout << "cache " << i << "\t";
        cout << "cache miss" << endl;

        int cntMisses = 0;

        for(int i=0; i<requestLength; ++i)
        {
            bool isMiss = updateCache(i, selectedStrategy[strat]);
            if (isMiss) ++cntMisses;

            cout << "  " << request[i] << "\t";
            for (int l=0; l < cacheSize; ++l) cout << "  " << cache[l] << "\t";
            cout << (isMiss ? "x" : "") << endl;
        }

        cout<< "\nTotal cache misses: " << cntMisses << endl;
    }

    for(int i=0; i<5; ++i) delete selectedStrategy[i];
}
```

Основная идея проста: для каждого запроса у меня есть два вызова две моей стратегии:

1. **применить** : стратегия должна сообщить вызывающему абоненту, какая страница использовать
2. **update** : После того, как вызывающий абонент использует это место, он сообщает стратегии, было ли это пропуском или нет. Затем стратегия может обновить свои внутренние данные. Например, стратегия **LFU** должна обновлять частоту попадания для страниц кеша, тогда как стратегия **LFD** должна пересчитывать расстояния для страниц кеша.

Теперь рассмотрим примеры реализации для наших пяти стратегий:

ФИФО

```
class FIFO : public Strategy {
public:
    FIFO() : Strategy("FIFO")
    {
        for (int i=0; i<cacheSize; ++i) age[i] = 0;
    }

    int apply(int requestIndex) override
    {
        int oldest = 0;

        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(age[i] > age[oldest])
                oldest = i;
```

```
        }

        return oldest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        // nothing changed we dont need to update the ages
        if(!cacheMiss)
            return;

        // all old pages get older, the new one get 0
        for(int i=0; i<cacheSize; ++i)
        {
            if(i != cachePos)
                age[i]++;

            else
                age[i] = 0;
        }
    }

private:
    int age[cacheSize];
};
```

FIFO просто нуждается в информации о том, как долго страница находится в кеше (и, конечно, только по отношению к другим страницам). Так что единственное, что нужно сделать - это подождать промаха, а затем сделать страницы, где бы не выселили. Для нашего примера выше решение программы:

```
Strategy: FIFO

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          d          b          c          x
  e          d          e          c          x
  b          d          e          b          x
  b          d          e          b
  a          a          e          b          x
  c          a          c          b          x
  f          a          c          f          x
  d          d          c          f          x
  e          d          e          f          x
  a          d          e          a          x
  f          f          e          a          x
  b          f          b          a          x
  e          f          b          e          x
  c          c          b          e          x

Total cache misses: 13
```

Точное решение сверху.

LIFO

```
class LIFO : public Strategy {
public:
    LIFO() : Strategy("LIFO")
    {
        for (int i=0; i<cacheSize; ++i) age[i] = 0;
    }

    int apply(int requestIndex) override
    {
        int newest = 0;

        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(age[i] < age[newest])
                newest = i;
        }

        return newest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        // nothing changed we dont need to update the ages
        if(!cacheMiss)
            return;

        // all old pages get older, the new one get 0
        for(int i=0; i<cacheSize; ++i)
        {
            if(i != cachePos)
                age[i]++;

            else
                age[i] = 0;
        }
    }

private:
    int age[cacheSize];
};
```

Реализация **LIFO** более или менее такая же, как и **FIFO,** но мы выселяем самую молодую не самую старую страницу. Результаты программы:

```
Strategy: LIFO

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          d          b          c          x
  e          e          b          c          x
```

```
  b          e          b          c
  b          e          b          c
  a          a          b          c          x
  c          a          b          c
  f          f          b          c          x
  d          d          b          c          x
  e          e          b          c          x
  a          a          b          c          x
  f          f          b          c          x
  b          f          b          c
  e          e          b          c          x
  c          e          b          c

Total cache misses: 9
```

LRU

```
class LRU : public Strategy {
public:
    LRU() : Strategy("LRU")
    {
        for (int i=0; i<cacheSize; ++i) age[i] = 0;
    }

    // here oldest mean not used the longest
    int apply(int requestIndex) override
    {
        int oldest = 0;

        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(age[i] > age[oldest])
                oldest = i;
        }

        return oldest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        // all old pages get older, the used one get 0
        for(int i=0; i<cacheSize; ++i)
        {
            if(i != cachePos)
                age[i]++;

            else
                age[i] = 0;
        }
    }

private:
    int age[cacheSize];
};
```

В случае **LRU** стратегия не зависит от того, что находится на странице кэша, но ее

единственным интересом является последнее использование. Результаты программы:

```
Strategy: LRU

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          a          d          c          x
  e          a          d          e          x
  b          b          d          e          x
  b          b          d          e
  a          b          a          e          x
  c          b          a          c          x
  f          f          a          c          x
  d          f          d          c          x
  e          f          d          e          x
  a          a          d          e          x
  f          a          f          e          x
  b          a          f          b          x
  e          e          f          b          x
  c          e          c          b          x

Total cache misses: 13
```

УКП

```
class LFU : public Strategy {
public:
    LFU() : Strategy("LFU")
    {
        for (int i=0; i<cacheSize; ++i) requestFrequency[i] = 0;
    }

    int apply(int requestIndex) override
    {
        int least = 0;

        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(requestFrequency[i] < requestFrequency[least])
                least = i;
        }

        return least;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        if(cacheMiss)
            requestFrequency[cachePos] = 1;

        else
            ++requestFrequency[cachePos];
    }
```

```
private:

    // how frequently was the page used
    int requestFrequency[cacheSize];
};
```

LFU выселяет страницу, используя ее наименее часто. Поэтому стратегия обновления - это всего лишь подсчет каждого доступа. Конечно, после промаха счет сбрасывается. Результаты программы:

```
Strategy: LFU

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          a          d          c          x
  e          a          d          e          x
  b          a          b          e          x
  b          a          b          e
  a          a          b          e
  c          a          b          c          x
  f          a          b          f          x
  d          a          b          d          x
  e          a          b          e          x
  a          a          b          e
  f          a          b          f          x
  b          a          b          f
  e          a          b          e          x
  c          a          b          c          x

Total cache misses: 10
```

LFD

```
class LFD : public Strategy {
public:
    LFD() : Strategy("LFD")
    {
        // precalc next usage before starting to fullfill requests
        for (int i=0; i<cacheSize; ++i) nextUse[i] = calcNextUse(-1, cache[i]);
    }

    int apply(int requestIndex) override
    {
        int latest = 0;

        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(nextUse[i] > nextUse[latest])
                latest = i;
        }
```

```
        return latest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
            nextUse[cachePos] = calcNextUse(requestIndex, cache[cachePos]);
    }

private:

    int calcNextUse(int requestPosition, char pageItem)
    {
        for(int i = requestPosition+1; i < requestLength; ++i)
        {
            if (request[i] == pageItem)
                return i;
        }

        return requestLength + 1;
    }

    // next usage of page
    int nextUse[cacheSize];
};
```

Стратегия **LFD** отличается от всех раньше. Это единственная стратегия, которая использует будущие просьбы о ее освобождении, которые выселяют. Реализация использует функцию `calcNextUse` чтобы получить страницу, которая будет использоваться дальше в будущем. Решение программы равно решению сверху сверху:

```
Strategy: LFD

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          a          b          d          x
  e          a          b          e          x
  b          a          b          e
  b          a          b          e
  a          a          b          e
  c          a          c          e          x
  f          a          f          e          x
  d          a          d          e          x
  e          a          d          e
  a          a          d          e
  f          f          d          e          x
  b          b          d          e          x
  e          b          d          e
  c          c          d          e          x

Total cache misses: 8
```

Жадная стратегия **LFD** - действительно единственная оптимальная стратегия из пяти представленных. Доказательство довольно длинное и можно найти здесь или в книге

Джона Клейнберга и Евы Тардос (см. Источники в примечаниях ниже).

Алгоритм против реальности

Стратегия **LFD** оптимальна, но есть большая проблема. Это оптимальное **автономное** решение. В практическом кэшировании обычно проблема **онлайн** , это означает, что стратегия бесполезна, потому что мы не можем сейчас в следующий раз нам нужен конкретный элемент. Другие четыре стратегии также являются **онлайн-** стратегиями. Для онлайн-проблем нам нужен общий подход.

Прочитайте Применение жадности онлайн: https://riptutorial.com/ru/algorithm/topic/7993/применение-жадности

глава 45: Проблема с рюкзаком

замечания

Проблема Knapsack в основном возникает в механизмах распределения ресурсов. Название «Рюкзак» было впервые представлено **Тобиасом Данцигом** .

Вспомогательное пространство: `O(nw)`
Сложность времени `O(nw)`

Examples

Основы работы с рюкзаком

Проблема . Учитывая набор элементов, в которых каждый элемент содержит вес и значение, определите количество каждого из них для включения в коллекцию, чтобы общий вес был меньше или равным заданному пределу, а общее значение максимально велико ,

Псевдокод для проблемы Knapsack

Дано:

1. Значения (массив v)
2. Массы (массив w)
3. Количество отдельных элементов (n)
4. Мощность (Вт)

```
for j from 0 to W do:
    m[0, j] := 0
for i from 1 to n do:
    for j from 0 to W do:
        if w[i] > j then:
            m[i, j] := m[i-1, j]
        else:
            m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
```

Простая реализация вышеуказанного псевдокода с использованием Python:

```
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W+1)] for x in range(n+1)]
    for i in range(n+1):
        for w in range(W+1):
            if i==0 or w==0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w])
```

```
            else:
                K[i][w] = K[i-1][w]
    return K[n][W]
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print(knapSack(W, wt, val, n))
```

Запуск кода: Сохраните это в файле с именем knapSack.py

```
$ python knapSack.py
220
```

Сложность времени для вышеуказанного кода: `O(nW)` где n - количество элементов, а W - емкость рюкзака.

Решение, реализованное на C

```
public class KnapsackProblem
{
    private static int Knapsack(int w, int[] weight, int[] value, int n)
    {
        int i;
        int[,] k = new int[n + 1, w + 1];
        for (i = 0; i <= n; i++)
        {
            int b;
            for (b = 0; b <= w; b++)
            {
                if (i==0 || b==0)
                {
                    k[i, b] = 0;
                }
                else if (weight[i - 1] <= b)
                {
                    k[i, b] = Math.Max(value[i - 1] + k[i - 1, b - weight[i - 1]], k[i - 1,
b]);
                }
                else
                {
                    k[i, b] = k[i - 1, b];
                }
            }
        }
        return k[n, w];
    }

    public static int Main(int nItems, int[] weights, int[] values)
    {
        int n = values.Length;
        return Knapsack(nItems, weights, values, n);
    }
}
```

Прочитайте Проблема с рюкзаком онлайн: https://riptutorial.com/ru/algorithm/topic/7250/

проблема-с-рюкзаком

глава 46: Проверьте две строки: анаграммы

Вступление

Две строки с одинаковым набором символов называются анаграммами. Я использовал javascript здесь.

Мы создадим хэш str1 и увеличим счет +1. Мы будем зацикливаться на 2-й строке и проверять наличие всех символов в хэше и уменьшить значение хэш-ключа. Проверьте, что все значение хэш-клавиши равно нулю, будет анаграммой.

Examples

Пример ввода и вывода

Ex1: -

```
let str1 = 'stackoverflow';
let str2 = 'flowerovstack';
```

Эти строки являются анаграммами.

// Создаем хэш из str1 и увеличиваем один счетчик.

```
hashMap = {
    s : 1,
    t : 1,
    a : 1,
    c : 1,
    k : 1,
    o : 2,
    v : 1,
    e : 1,
    r : 1,
    f : 1,
    l : 1,
    w : 1
}
```

Вы можете видеть, что hashKey 'o' содержит значение 2, потому что o 2 раза в строке.

Теперь loop over str2 и проверка каждого символа присутствуют в hashMap, если да, уменьшите значение hashMap Key, иначе верните false (что указывает, что это не анаграмма).

```
hashMap = {
    s : 0,
    t : 0,
    a : 0,
    c : 0,
    k : 0,
    o : 0,
    v : 0,
    e : 0,
    r : 0,
    f : 0,
    l : 0,
    w : 0
}
```

Теперь перетащите объект hashMap и проверьте, что все значения равны нулю в ключе hashMap.

В нашем случае все значения равны нулю, поэтому его анаграмма.

Общий код для анаграмм

```
(function(){

    var hashMap = {};

    function isAnagram (str1, str2) {

        if(str1.length !== str2.length){
            return false;
        }

        // Create hash map of str1 character and increase value one (+1).
        createStr1HashMap(str1);

        // Check str2 character are key in hash map and decrease value by one(-1);
        var valueExist = createStr2HashMap(str2);

        // Check all value of hashMap keys are zero, so it will be anagram.
        return isStringsAnagram(valueExist);
    }

    function createStr1HashMap (str1) {
        [].map.call(str1, function(value, index, array){
            hashMap[value] = value in hashMap ?  (hashMap[value] + 1) : 1;
            return value;
        });
    }

    function createStr2HashMap (str2) {
        var valueExist = [].every.call(str2, function(value, index, array){
            if(value in hashMap) {
                hashMap[value] = hashMap[value] - 1;
            }
            return value in hashMap;
        });
        return valueExist;
    }
```

```
    function isStringsAnagram (valueExist) {
        if(!valueExist) {
            return valueExist;
        } else {
            var isAnagram;
            for(var i in hashMap) {
                if(hashMap[i] !== 0) {
                    isAnagram = false;
                    break;
                } else {
                    isAnagram = true;
                }
            }

            return isAnagram;
        }
    }

    isAnagram('stackoverflow', 'flowerovstack'); // true
    isAnagram('stackoverflow', 'flowervvstack'); // false

})();
```

Сложность времени: - 3n, т.е. O (n).

Прочитайте Проверьте две строки: анаграммы онлайн:
https://riptutorial.com/ru/algorithm/topic/9970/проверьте-две-строки--анаграммы

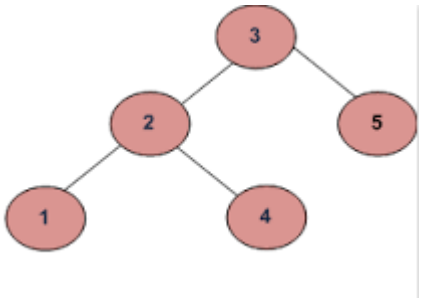
глава 47: Проверьте, нет ли дерева BST или нет.

Examples

Если заданное дерево ввода следует за свойством дерева двоичного поиска или нет

Например

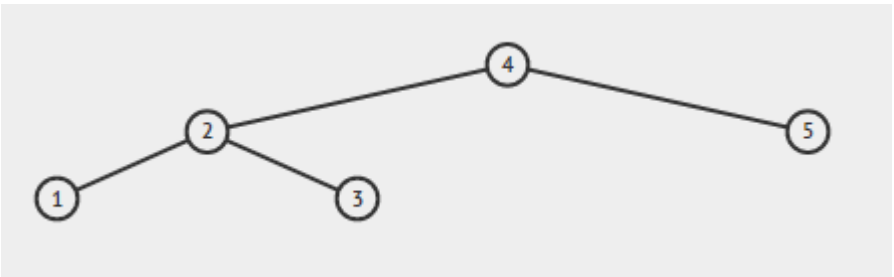
если вход:



Выход должен быть ложным:

Поскольку 4 в левом поддереве больше корневого значения (3)

Если вход:



Выход должен быть правдой

Алгоритм проверки, является ли данное двоичное дерево BST

Бинарное дерево BST, если оно удовлетворяет любому из следующих условий:

1. Он пуст
2. У него нет поддеревьев
3. Для каждого узла x в дереве все ключи (если есть) в левом вспомогательном дереве должны быть меньше ключа (x), а все ключи (если есть) в правом поддереве должны

быть больше, чем ключ (x).

Таким образом, простой рекурсивный алгоритм будет:

```
is_BST(root):
  if root == NULL:
   return true

  // Check values in left subtree
  if root->left != NULL:
    max_key_in_left = find_max_key(root->left)
    if max_key_in_left > root->key:
        return false

  // Check values in right subtree
  if root->right != NULL:
    min_key_in_right = find_min_key(root->right)
    if min_key_in_right < root->key:
        return false

  return is_BST(root->left) && is_BST(root->right)
```

Вышеуказанный рекурсивный алгоритм является правильным, но неэффективным, поскольку он пересекает каждый узел несколько раз.

Другим подходом к минимизации множественных посещений каждого узла является запоминание минимальных и максимальных значений ключей в поддереве, который мы посещаем. Пусть минимально возможное значение любого ключа будет `K_MIN` а максимальное значение - `K_MAX` . Когда мы начинаем с корня дерева, диапазон значений в дереве `[K_MIN,K_MAX]` . Пусть ключ корневого узла равен `x` . Тогда диапазон значений в левом поддереве равен `[K_MIN,x)` а диапазон значений в правом поддереве `(x,K_MAX]` . Мы будем использовать эту идею для разработки более эффективного алгоритма.

```
is_BST(root, min, max):
    if root == NULL:
        return true

    // is the current node key out of range?
    if root->key < min || root->key > max:
        return false

    // check if left and right subtree is BST
    return is_BST(root->left,min,root->key-1) && is_BST(root->right,root->key+1,max)
```

Сначала он будет называться:

```
is_BST(my_tree_root,KEY_MIN,KEY_MAX)
```

Другим подходом будет сделать обход объекта двоичного дерева. Если обход ордера создает отсортированную последовательность ключей, то данное дерево является BST. Чтобы проверить, упорядочена ли последовательность порядка, помните значение ранее посещаемого узла и сравнивайте его с текущим узлом.

Прочитайте Проверьте, нет ли дерева BST или нет. онлайн:
https://riptutorial.com/ru/algorithm/topic/8840/проверьте--нет-ли-дерева-bst-или-нет-

глава 48: Прохождение двоичных деревьев

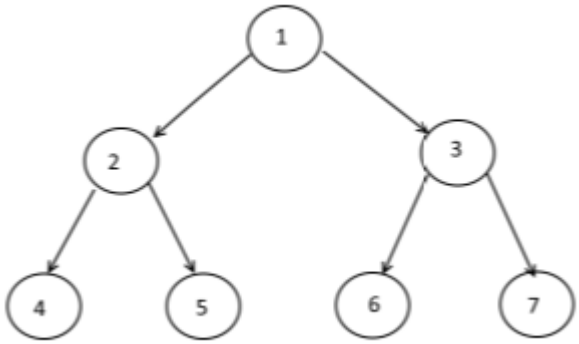
Вступление

Посещение узла двоичного дерева в определенном порядке называется обходами.

Examples

Предпросмотр, порядок и порядок заказа двоичного дерева

Рассмотрим двоичное дерево:



Проход по порядку (корень) проходит по узлу, затем левому поддереву узла, а затем правому поддереву узла.

Таким образом, предварительный обход предыдущего дерева будет:

1 2 4 5 3 6 7

В обходном пути (корень) проходит левое поддерево узла, а затем узел, а затем правый поддерево узла.

Таким образом, обход предыдущего дерева будет выглядеть следующим образом:

4 2 5 1 6 3 7

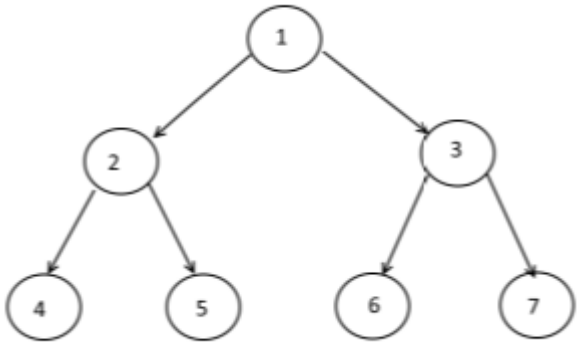
Постоперационный обход (корень) проходит по левому поддереву узла, а затем по правому поддереву, а затем по узлу.

Таким образом, послепорядок, пройденный выше, будет:

4 5 2 6 7 3 1

Прохождение обхода уровня - реализация

Например, если данное дерево:



Прохождение обхода уровня будет

1 2 3 4 5 6 7

Уровень данных уровня печати по уровню.

Код:

```
#include<iostream>
#include<queue>
#include<malloc.h>

using namespace std;

struct node{

    int data;
    node *left;
    node *right;
};

void levelOrder(struct node *root){

        if(root == NULL)    return;

        queue<node *> Q;
        Q.push(root);

        while(!Q.empty()){
        struct    node* curr = Q.front();
            cout<< curr->data <<" ";
            if(curr->left != NULL) Q.push(curr-> left);
                if(curr->right != NULL) Q.push(curr-> right);

                Q.pop();


        }
}
struct node* newNode(int data)
```

```
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

int main(){

    struct node *root = newNode(1);
    root->left        = newNode(2);
    root->right       = newNode(3);
    root->left->left  = newNode(4);
    root->left->right = newNode(5);
    root->right->left  = newNode(6);
    root->right->right = newNode(7);


    printf("Level Order traversal of binary tree is \n");
    levelOrder(root);

    return 0;


}
```

Для достижения вышеуказанной цели используется структура данных очереди.

Прочитайте Прохождение двоичных деревьев онлайн:
https://riptutorial.com/ru/algorithm/topic/8844/прохождение-двоичных-деревьев

глава 49: ПСЕВДОКОД

замечания

Псевдокод по определению является неформальным. Эта тема предназначена для описания способов перевода языкового кода в нечто, что может понять каждый, у кого есть фон программирования.

Псевдокод - это важный способ описать алгоритм и более нейтрален, чем предоставление специфической реализации. Википедия часто использует некоторую форму псевдокода при описании алгоритма

Некоторые вещи, такие как условия типа if-else, довольно легко записывать неформально. Но, к примеру, например, обратные вызовы js-стиля могут быть трудными для псевдокода для некоторых людей.

Вот почему эти примеры могут оказаться полезными

Examples

Переменные аффектации

Вы можете описать переменную аффектацию по-разному.

набранный

```
int a = 1
int a := 1
let int a = 1
int a <- 1
```

Нет типа

```
a = 1
a := 1
let a = 1
a <- 1
```

функции

Пока имя функции, оператор возврата и параметры ясны, вы в порядке.

```
def incr n
```

```
    return n + 1
```

или же

```
let incr(n) = n + 1
```

или же

```
function incr (n)
    return n + 1
```

все достаточно ясны, поэтому вы можете их использовать. Старайтесь не быть двусмысленным с переменным аффектацией

Прочитайте ПСЕВДОКОД онлайн: https://riptutorial.com/ru/algorithm/topic/7393/псевдокод

глава 50: Путешественник

замечания

Задача Traveling Salesman - проблема нахождения минимальной стоимости перемещения по `N` вершинам ровно один раз на вершину. Существует стоимость `cost[i][j]` для перехода от вершины `i` к вершине `j` .

Для решения этой проблемы существует два типа алгоритмов: *точные алгоритмы* и *алгоритмы аппроксимации*

Точные алгоритмы

1. Алгоритм грубой силы
2. Алгоритм динамического программирования

Аппроксимационные алгоритмы

Быть добавленным

Examples

Алгоритм грубой силы

Путь через каждую вершину ровно один раз совпадает с порядком верстки вершины. Таким образом, чтобы вычислить минимальную стоимость прохода через каждую вершину ровно один раз, мы можем переборщить каждую единственную из `N!` перестановки чисел от `1` до `N`

Psuedocode

```
minimum = INF
for all permutations P

    current = 0

    for i from 0 to N-2
        current = current + cost[P[i]][P[i+1]]  <- Add the cost of going from 1 vertex to the
next

    current = current + cost[P[N-1]][P[0]]      <- Add the cost of going from last vertex to
the first

    if current < minimum                        <- Update minimum if necessary
        minimum = current

output minimum
```

Сложность времени

Есть `N!` перестановки, и стоимость каждого пути вычисляется в `O(N)` , поэтому этот алгоритм принимает время `O(N * N!)` для вывода точного ответа.

Алгоритм динамического программирования

Обратите внимание, что если мы рассмотрим путь (по порядку):

```
(1,2,3,4,6,0,5,7)
```

и путь

```
(1,2,3,5,0,6,7,4)
```

Стоимость перехода от вершины `1` к вершине `2` к вершине `3` остается неизменной, так зачем ее пересчитывать? Этот результат можно сохранить для последующего использования.

Пусть `dp[bitmask][vertex]` представляет собой минимальную стоимость перемещения по всем вершинам, соответствующий бит в `bitmask` установлен в `1` заканчивающийся в `vertex` . Например:

```
dp[12][2]

   12   =   1 1 0 0
            ^ ^
vertices:   3 2 1 0
```

Так как `12` представляет `1100` в двоичном формате, `dp[12][2]` представляет перемещение по вершинам `2` и `3` в графе с концом, заканчивающимся в вершине 2.

Таким образом, мы можем иметь следующий алгоритм (реализация C ++):

```
int cost[N][N];                         //Adjust the value of N if needed
int memo[1 << N][N];                    //Set everything here to -1
int TSP(int bitmask, int pos){
    int cost = INF;
    if (bitmask == ((1 << N) - 1)){      //All vertices have been explored
        return cost[pos][0];             //Cost to go back
    }
    if (memo[bitmask][pos] != -1){       //If this has already been computed
        return memo[bitmask][pos];       //Just return the value, no need to recompute
    }
    for (int i = 0; i < N; ++i){         //For every vertex
        if ((bitmask & (1 << i)) == 0){  //If the vertex has not been visited
            cost = min(cost,TSP(bitmask | (1 << i) , i) + cost[pos][i]);   //Visit the vertex
        }
    }
    memo[bitmask][pos] = cost;           //Save the result
    return cost;
}
//Call TSP(1,0)
```

Эта строка может быть немного запутанной, поэтому давайте медленно ее протекаем:

```
cost = min(cost,TSP(bitmask | (1 << i) , i) + cost[pos][i]);
```

Здесь `bitmask | (1 << i)` устанавливает i-й бит `bitmask` в 1, что означает, что i-я вершина была посещена. `i` после запятой представляет новый `pos` в вызове функции, который представляет новую «последнюю» вершину. `cost[pos][i]` заключается в добавлении стоимости перехода от вершины `pos` к вершине `i` .

Таким образом, эта строка заключается в обновлении стоимости `cost` до минимально возможного значения перемещения в любую другую вершину, которая еще не была посещена.

Сложность времени

Функция `TSP(bitmask,pos)` имеет значения `2^N` для `bitmask` и значения `N` для `pos` . Каждая функция выполняет время `O(N)` для запуска (цикл `for`). Таким образом, для реализации точного ответа эта реализация принимает время `O(N^2 * 2^N)` .

Прочитайте Путешественник онлайн: https://riptutorial.com/ru/algorithm/topic/6631/путешественник

глава 51: Решение уравнений

Examples

Линейное уравнение

Существует два класса методов решения линейных уравнений:

1. **Прямые методы** . Общими характеристиками прямых методов являются то, что они преобразуют исходное уравнение в эквивалентные уравнения, которые могут быть решены более легко, означает, что мы получаем решение непосредственно из уравнения.

2. **Итеративный метод** : Итеративные или косвенные методы, начните с догадки решения, а затем повторно уточните решение до тех пор, пока не будет достигнут определенный критерий конвергенции. Итерационные методы обычно менее эффективны, чем прямые методы, потому что требуется большое количество операций. Пример - метод итерации Якоби, метод ионизации Гаусса-Сейдаля.

Реализация в C-

```
//Implementation of Jacobi's Method
void JacobisMethod(int n, double x[n], double b[n], double a[n][n]){
    double Nx[n]; //modified form of variables
    int rootFound=0; //flag

    int i, j;
    while(!rootFound){
        for(i=0; i<n; i++){              //calculation
            Nx[i]=b[i];

            for(j=0; j<n; j++){
                if(i!=j) Nx[i] = Nx[i]-a[i][j]*x[j];
            }
            Nx[i] = Nx[i] / a[i][i];
        }

        rootFound=1;                    //verification
        for(i=0; i<n; i++){
            if(!( (Nx[i]-x[i])/x[i] > -0.000001 && (Nx[i]-x[i])/x[i] < 0.000001 )){
                rootFound=0;
                break;
            }
        }

        for(i=0; i<n; i++){             //evaluation
            x[i]=Nx[i];
        }
    }

    return ;
```

```
}

//Implementation of Gauss-Seidal Method
void GaussSeidalMethod(int n, double x[n], double b[n], double a[n][n]){
    double Nx[n]; //modified form of variables
    int rootFound=0; //flag

    int i, j;
    for(i=0; i<n; i++){                  //initialization
        Nx[i]=x[i];
    }

    while(!rootFound){
        for(i=0; i<n; i++){              //calculation
            Nx[i]=b[i];

            for(j=0; j<n; j++){
                if(i!=j) Nx[i] = Nx[i]-a[i][j]*Nx[j];
            }
            Nx[i] = Nx[i] / a[i][i];
        }

        rootFound=1;                     //verification
        for(i=0; i<n; i++){
            if(!( (Nx[i]-x[i])/x[i] > -0.000001 && (Nx[i]-x[i])/x[i] < 0.000001 )){
                rootFound=0;
                break;
            }
        }

        for(i=0; i<n; i++){              //evaluation
            x[i]=Nx[i];
        }
    }

    return ;
}

//Print array with comma separation
void print(int n, double x[n]){
    int i;
    for(i=0; i<n; i++){
        printf("%lf, ", x[i]);
    }
    printf("\n\n");

    return ;
}

int main(){
    //equation initialization
    int n=3;    //number of variables

    double x[n];    //variables

    double b[n],    //constants
        a[n][n];    //arguments

    //assign values
    a[0][0]=8; a[0][1]=2; a[0][2]=-2; b[0]=8;    //8x1+2x2-2x3+8=0
    a[1][0]=1; a[1][1]=-8; a[1][2]=3; b[1]=-4;   //x1-8x2+3x3-4=0
```

```
    a[2][0]=2; a[2][1]=1; a[2][2]=9; b[2]=12;    //2x1+x2+9x3+12=0


    int i;

    for(i=0; i<n; i++){                          //initialization
        x[i]=0;
    }
    JacobisMethod(n, x, b, a);
    print(n, x);


    for(i=0; i<n; i++){                          //initialization
        x[i]=0;
    }
    GaussSeidalMethod(n, x, b, a);
    print(n, x);

    return 0;
}
```

Нелинейное уравнение

Уравнение типа `f(x)=0` является либо алгебраическим, либо трансцендентным. Эти типы уравнений могут быть решены с использованием двух типов методов -

1. **Прямой метод** : этот метод дает точное значение всех корней непосредственно в конечном числе шагов.

2. **Косвенный или итеративный метод** : Итерационные методы лучше всего подходят для компьютерных программ для решения уравнения. Он основан на концепции последовательного приближения. В Итеративном методе существует два способа решения уравнения -

    - **Метод брекетинга** : мы берем две исходные точки, где корень лежит между ними. Пример - метод бисекции, метод ложной позиции.

    - **Метод открытого конца** : мы берем одно или два начальных значения, где корень может быть любым. Пример - метод Ньютона-Рафсона, метод последовательной аппроксимации, метод секущей.

Реализация в C-

```
/// Here define different functions to work with
#define f(x) ( ((x)*(x)*(x)) - (x) - 2 )
#define f2(x) ( (3*(x)*(x)) - 1 )
#define g(x) ( cbrt( (x) + 2 ) )


/**
* Takes two initial values and shortens the distance by both side.
**/
double BisectionMethod(){
```

```
    double root=0;

    double a=1, b=2;
    double c=0;

    int loopCounter=0;
    if(f(a)*f(b) < 0){
        while(1){
            loopCounter++;
            c=(a+b)/2;

            if(f(c)<0.00001 && f(c)>-0.00001){
                root=c;
                break;
            }

            if((f(a))*(f(c)) < 0){
                b=c;
            }else{
                a=c;
            }

        }
    }
    printf("It took %d loops.\n", loopCounter);

    return root;
}

/**
* Takes two initial values and shortens the distance by single side.
**/
double FalsePosition(){
    double root=0;

    double a=1, b=2;
    double c=0;

    int loopCounter=0;
    if(f(a)*f(b) < 0){
        while(1){
            loopCounter++;

            c=(a*f(b) - b*f(a)) / (f(b) - f(a));

            /*/printf("%lf\t %lf \n", c, f(c));/**////test
            if(f(c)<0.00001 && f(c)>-0.00001){
                root=c;
                break;
            }

            if((f(a))*(f(c)) < 0){
                b=c;
            }else{
                a=c;
            }
        }
    }
    printf("It took %d loops.\n", loopCounter);

    return root;
```

```
}

/**
* Uses one initial value and gradually takes that value near to the real one.
**/
double NewtonRaphson(){
    double root=0;

    double x1=1;
    double x2=0;

    int loopCounter=0;
    while(1){
        loopCounter++;

        x2 = x1 - (f(x1)/f2(x1));
        /*/printf("%lf \t %lf \n", x2, f(x2));/**////test

        if(f(x2)<0.00001 && f(x2)>-0.00001){
            root=x2;
            break;
        }

        x1=x2;
    }
    printf("It took %d loops.\n", loopCounter);

    return root;
}

/**
* Uses one initial value and gradually takes that value near to the real one.
**/
double FixedPoint(){
    double root=0;
    double x=1;

    int loopCounter=0;
    while(1){
        loopCounter++;

        if( (x-g(x)) <0.00001 && (x-g(x)) >-0.00001){
            root = x;
            break;
        }

        /*/printf("%lf \t %lf \n", g(x), x-(g(x)));/**////test

        x=g(x);
    }
    printf("It took %d loops.\n", loopCounter);

    return root;
}

/**
* uses two initial values & both value approaches to the root.
**/
double Secant(){
    double root=0;
```

```
    double x0=1;
    double x1=2;
    double x2=0;

    int loopCounter=0;
    while(1){
        loopCounter++;

        /*/printf("%lf \t %lf \t %lf \n", x0, x1, f(x1));/**////test

        if(f(x1)<0.00001 && f(x1)>-0.00001){
            root=x1;
            break;
        }

        x2 = ((x0*f(x1))-(x1*f(x0))) / (f(x1)-f(x0));

        x0=x1;
        x1=x2;
    }
    printf("It took %d loops.\n", loopCounter);

    return root;
}


int main(){
    double root;

    root = BisectionMethod();
    printf("Using Bisection Method the root is: %lf \n\n", root);

    root = FalsePosition();
    printf("Using False Position Method the root is: %lf \n\n", root);

    root = NewtonRaphson();
    printf("Using Newton-Raphson Method the root is: %lf \n\n", root);

    root = FixedPoint();
    printf("Using Fixed Point Method the root is: %lf \n\n", root);

    root = Secant();
    printf("Using Secant Method the root is: %lf \n\n", root);

    return 0;
}
```

Прочитайте Решение уравнений онлайн: https://riptutorial.com/ru/algorithm/topic/7379/решение-уравнений

глава 52: Самая длинная общая подпоследовательность

Examples

Наибольшая общая подпоследовательность Объяснение

Одна из наиболее важных реализаций динамического программирования - поиск самой длинной общей подпоследовательности . Сначала определим некоторые основные термины.

подпоследовательности:

Подпоследовательность представляет собой последовательность, которая может быть получена из другой последовательности путем удаления некоторых элементов без изменения порядка остальных элементов. Допустим, у нас есть строка **ABC** . Если мы стираем нуль или один или несколько символов из этой строки, мы получаем подпоследовательность этой строки. Таким образом, подпоследовательности строки **ABC** будут { **"A"** , **"B"** , **"C"** , **"AB"** , **"AC"** , **"BC"** , **"ABC"** , **""** }. Даже если мы удалим все символы, пустая строка также будет подпоследовательностью. Чтобы узнать подпоследовательность, для каждого символа в строке мы имеем два варианта: либо мы берем символ, либо нет. Поэтому, если длина строки равна **n** , существует 2^n подпоследовательностей этой строки.

Самая длинная общая подпоследовательность:

Как следует из названия, из всех общих подпоследовательностей между двумя строками самая длинная общая подпоследовательность (LCS) - это та, которая имеет максимальную длину. Например: общие подпоследовательности между **«HELLOM»** и **«HMLD»** - это **«H»** , **«HL»** , **«HM»** и т. Д. Здесь **«HLL»** является самой длинной общей подпоследовательностью, длина которой 3.

Метод грубой силы:

Мы можем сгенерировать все подпоследовательности двух строк с использованием *обратного отсчета* . Затем мы можем сравнить их, чтобы выяснить общие подпоследовательности. После того, как нам понадобится узнать ту, которая имеет максимальную длину. Мы уже видели, что существует 2^n подпоследовательностей строки длины **n** . Потребуются годы, чтобы решить проблему, если наши **русские** кресты **20-25** .

Метод динамического программирования:

Давайте подходим к нашему методу с примером. Предположим, что мы имеем две строки

abcdaf и **acbcf** . Обозначим их с **s1** и **s2** . Таким образом, самая длинная общая подпоследовательность этих двух строк будет **«abcf»** , которая имеет длину 4. Снова напоминаю вам, подпоследовательности не обязательно должны быть непрерывными в строке. Чтобы построить **«abcf»** , мы игнорировали **«da»** в **s1** и **«c»** в **s2** . Как это узнать, используя динамическое программирование?

Мы начнем с таблицы (2D-массив), содержащей все символы **s1** в строке и все символы **s2** в столбце. Здесь таблица 0-индексируется, и мы помещаем символы от 1 до и далее. Мы будем перемещать таблицу слева направо для каждой строки. Наша таблица будет выглядеть так:

```
          0     1     2     3     4     5     6
    +-----+-----+-----+-----+-----+-----+-----+-----+
    | ch  |     |  a  |  b  |  c  |  d  |  a  |  f  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
0   |     |     |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
1   |  a  |     |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
2   |  c  |     |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
3   |  b  |     |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
4   |  c  |     |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
5   |  f  |     |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
```

Здесь каждая строка и столбец представляют длину самой длинной общей подпоследовательности между двумя строками, если мы берем символы этой строки и столбца и добавляем к префиксу перед ним. Например: **Таблица [2] [3]** представляет длину самой длинной общей подпоследовательности между **«ac»** и **«abc»** .

0-й столбец представляет собой пустую подпоследовательность **s1** . Точно так же 0-я строка представляет собой пустую подпоследовательность **s2** . Если взять пустую подпоследовательность строки и попытаться сопоставить ее с другой строкой, независимо от того, как долго длина второй подстроки, то общая подпоследовательность будет иметь длину 0. Таким образом, мы можем заполнить 0-й ряд и 0-й столбцы нулями. Мы получаем:

```
          0     1     2     3     4     5     6
    +-----+-----+-----+-----+-----+-----+-----+-----+
    | ch  |     |  a  |  b  |  c  |  d  |  a  |  f  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
0   |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
1   |  a  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
2   |  c  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
3   |  b  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
4   |  c  |  0  |     |     |     |     |     |     |
```

```
    +-----+-----+-----+-----+-----+-----+-----+-----+
5   |  f  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
```

Давай начнем. Когда мы заполняем **таблицу [1] [1]** , мы спрашиваем себя, если бы у нас была строка **a** и другая строка **a** и ничего больше, какая будет самая длинная общая подпоследовательность здесь? Длина LCS здесь будет 1. Теперь рассмотрим **таблицу [1] [2]** . У нас есть строка **ab** и строка **a** . Длина LCS будет равна 1. Как вы можете видеть, остальные значения будут также 1 для первой строки, так как он рассматривает только строку **a** с **abcd** , **abcda** , **abcdaf** . Итак, наша таблица будет выглядеть так:

```
            0     1     2     3     4     5     6
    +-----+-----+-----+-----+-----+-----+-----+-----+
    | ch□ |     |  a  |  b  |  c  |  d  |  a  |  f  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
0   |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
1   |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
2   |  c  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
3   |  b  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
4   |  c  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
5   |  f  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
```

Для строки 2, которая теперь будет включать **c** . Для **таблицы [2] [1]** мы имеем **ac** с одной стороны, а **c** другой стороны. Таким образом, длина LCS равна 1. Откуда у нас это 1? С вершины, обозначающей LCS **a** между двумя подстроками. Итак, мы говорим, что если **s1 [2]** и **s2 [1]** не совпадают, то длина LCS будет максимальной длины LCS **сверху** или **слева** . Взяв длину LCS сверху, это означает, что мы не берем текущий символ из **s2** . Аналогично, взяв длину LCS слева, мы не берем текущий символ из **s1** для создания LCS. Мы получаем:

```
            0     1     2     3     4     5     6
    +-----+-----+-----+-----+-----+-----+-----+-----+
    | ch□ |     |  a  |  b  |  c  |  d  |  a  |  f  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
0   |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
1   |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
2   |  c  |  0  |  1  |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
3   |  b  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
4   |  c  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
5   |  f  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
```

Итак, наша первая формула будет:

```
if s2[i] is not equal to s1[j]
    Table[i][j] = max(Table[i-1][j], Table[i][j-1]
endif
```

Двигаясь дальше, для **таблицы [2] [2]** имеем строку **ab** и **ac** . Поскольку **c** и **b** не являются одинаковыми, мы помещаем максимум сверху или слева здесь. В этом случае это снова 1. После этого для **таблицы [2] [3]** имеем строку **abc** и **ac** . На этот раз текущие значения обеих строк и столбцов совпадают. Теперь длина LCS будет равна максимальной длине LCS до + 1. Как мы получаем максимальную длину LCS до сих пор? Мы проверяем диагональное значение, которое представляет наилучшее совпадение между **ab** и **a** . Из этого состояния для текущих значений мы добавили еще один символ к **s1** и **s2,** который оказался одним и тем же. Таким образом, длина LCS, разумеется, возрастет. Мы поместим **1 + 1 = 2** в **таблицу [2] [3]** . Мы получаем,

```
          0     1     2     3     4     5     6
    +-----+-----+-----+-----+-----+-----+-----+-----+
    | ch  |     |  a  |  b  |  c  |  d  |  a  |  f  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 1  |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 2  |  c  |  0  |  1  |  1  |  2  |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 3  |  b  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 4  |  c  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 5  |  f  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
```

Итак, наша вторая формула будет:

```
if s2[i] equals to s1[j]
    Table[i][j] = Table[i-1][j-1] + 1
endif
```

Мы определили оба случая. Используя эти две формулы, мы можем заполнить всю таблицу. После заполнения таблицы это будет выглядеть так:

```
          0     1     2     3     4     5     6
    +-----+-----+-----+-----+-----+-----+-----+-----+
    | ch  |     |  a  |  b  |  c  |  d  |  a  |  f  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 1  |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 2  |  c  |  0  |  1  |  1  |  2  |  2  |  2  |  2  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 3  |  b  |  0  |  1  |  2  |  2  |  2  |  2  |  2  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 4  |  c  |  0  |  1  |  2  |  3  |  3  |  3  |  3  |
```

```
     +-----+-----+-----+-----+-----+-----+-----+-----+
  5  |  f  |  0  |  1  |  2  |  3  |  3  |  3  |  4  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
```

Длина LCS между **s1** и **s2** будет равна **таблице [5] [6] = 4** . Здесь 5 и 6 - длина **s2** и **s1** соответственно. Наш псевдокод будет:

```
Procedure LCSlength(s1, s2):
Table[0][0] = 0
for i from 1 to s1.length
    Table[0][i] = 0
endfor
for i from 1 to s2.length
    Table[i][0] = 0
endfor
for i from 1 to s2.length
    for j from 1 to s1.length
        if s2[i] equals to s1[j]
            Table[i][j] = Table[i-1][j-1] + 1
        else
            Table[i][j] = max(Table[i-1][j], Table[i][j-1])
        endif
    endfor
endfor
Return Table[s2.length][s1.length]
```

Сложность времени для этого алгоритма: **O (mn),** где **m** и **n** обозначает длину каждой строки.

Как узнать самую длинную общую подпоследовательность? Мы начнем с нижнего правого угла. Мы проверим, откуда это значение. Если значение идет по диагонали, то есть, если **таблица [i-1] [j-1]** равна **таблице [i] [j] - 1** , мы нажимаем либо **s2 [i],** либо **s1 [j]** (оба одинаковы) и двигаться по диагонали. Если значение идет сверху, это означает, что если **таблица [i-1] [j]** равна **таблице [i] [j]** , мы переходим к вершине. Если значение идет слева, это означает, что если **таблица [i] [j-1]** равна **таблице [i] [j]** , мы перемещаемся влево. Когда мы достигнем самого левого или верхнего столбца, наш поиск заканчивается. Затем мы выставляем значения из стека и печатаем их. Псевдокод:

```
Procedure PrintLCS(LCSlength, s1, s2)
temp := LCSlength
S = stack()
i := s2.length
j := s1.length
while i is not equal to 0 and j is not equal to 0
     if Table[i-1][j-1] == Table[i][j] - 1 and s1[j]==s2[i]
        S.push(s1[j])   //or S.push(s2[i])
        i := i - 1
        j := j - 1
    else if Table[i-1][j] == Table[i][j]
        i := i-1
    else
        j := j-1
    endif
endwhile
```

```
while S is not empty
    print(S.pop)
endwhile
```

Следует отметить: если обе **таблицы [i-1] [j]** и **таблица [i] [j-1]** равны **таблице [i] [j],** а **таблица [i-1] [j-1]** не является равной **таблице [i] [j] - 1** , для этого момента может быть два LCS. Этот псевдокод не рассматривает эту ситуацию. Вам придется решить эту проблему, чтобы найти несколько LCS.

Сложность времени для этого алгоритма: **O (max (m, n))** .

Прочитайте Самая длинная общая подпоследовательность онлайн:
https://riptutorial.com/ru/algorithm/topic/7517/самая-длинная-общая-подпоследовательность

глава 53: Самая продолжительная подпоследовательность

Examples

Самая длинная нарастающая базовая информация

Задача « Самая длинная нарастающая подпоследовательность» заключается в том, чтобы найти подпоследовательность из входной последовательности give, в которой элементы подпоследовательности отсортированы в младшем и высоком порядке. Все подпоследовательности не являются непрерывными или уникальными.

Применение наиболее долговременной последовательности:

Алгоритмы, такие как Longest Increasing Subsequence, Longest Common Subsequence, используются в системах контроля версий, таких как Git и т. Д.

Простая форма алгоритма:

1. Найдите уникальные строки, общие для обоих документов.
2. Возьмите все такие строки из первого документа и закажите их в соответствии со своим внешним видом во втором документе.
3. Вычислите LIS результирующей последовательности (сделав терпение Сортировка), получив самую длинную совпадающую последовательность строк, соответствие между строками двух документов.
4. Учтите алгоритм для каждого диапазона строк между уже согласованными.

Теперь рассмотрим более простой пример проблемы LCS. Здесь ввод представляет собой только одну последовательность различных целых чисел `a1,a2,...,an.` , и мы хотим найти в нем самую длинную возрастающую подпоследовательность. Например, если вход **7,3,8,4,2,6,** то самая длинная возрастающая подпоследовательность составляет **3,4,6** .

Самый простой способ - сортировать входные элементы в порядке возрастания и применять алгоритм LCS к исходным и отсортированным последовательностям. Однако, если вы посмотрите на результирующий массив, вы заметите, что многие значения одинаковы, и массив выглядит очень повторяющимся. Это говорит о том, что проблема LIS (самая длинная возрастающая подпоследовательность) может быть выполнена с помощью алгоритма динамического программирования, использующего только одномерный массив.

Псевдокод:

1. Опишите массив значений, которые мы хотим вычислить.
   Для `1 <= i <= n` пусть **A (i)** - длина самой длинной возрастающей последовательности

ввода. Заметим, что длина, в которой мы в конечном счете заинтересованы, равна `max{A(i)|1 ≤ i ≤ n}`.

2. Дайте повторение.
   Для `1 <= i <= n A(i) = 1 + max{A(j)|1 ≤ j < i` **И** `input(j) < input(i)}`.
3. Вычислить значения A.
4. Найти оптимальное решение.

Следующая программа использует A для вычисления оптимального решения. Первая часть вычисляет значение m такое, что **A (m)** - длина оптимальной возрастающей подпоследовательности ввода. Вторая часть вычисляет оптимальную возрастающую подпоследовательность, но для удобства мы ее распечатываем в обратном порядке. Эта программа выполняется во времени O (n), поэтому весь алгоритм работает во времени O (n ^ 2).

Часть 1:

```
m ← 1
for i : 2..n
    if A(i) > A(m) then
        m ← i
    end if
end for
```

Часть 2:

```
put a
while A(m) > 1 do
    i ← m-1
    while not(ai < am and A(i) = A(m)-1) do
        i ← i-1
    end while
    m ← i
    put a
 end while
```

Рекурсивное решение:

Подход 1:

```
LIS(A[1..n]):
    if (n = 0) then return 0
    m = LIS(A[1..(n - 1)])
    B is subsequence of A[1..(n - 1)] with only elements less than a[n]
    (* let h be size of B, h ≤ n-1 *)
    m = max(m, 1 + LIS(B[1..h]))
    Output m
```

Сложность времени в подходе 1: `O(n*2^n)`

Подход 2:

```
LIS(A[1..n], x):
    if (n = 0) then return 0
    m = LIS(A[1..(n - 1)], x)
    if (A[n] < x) then
        m = max(m, 1 + LIS(A[1..(n - 1)], A[n]))
    Output m

MAIN(A[1..n]):
    return LIS(A[1..n], ∞)
```

Сложность времени в подходе 2: `O(n^2)`

Подход 3:

```
LIS(A[1..n]):
    if (n = 0) return 0
    m = 1
    for i = 1 to n - 1 do
        if (A[i] < A[n]) then
            m = max(m, 1 + LIS(A[1..i]))
    return m

MAIN(A[1..n]):
    return LIS(A[1..i])
```

Сложность времени в подходе 3: `O(n^2)`

Итеративный алгоритм:

Вычисляет значения итеративно снизу вверх.

```
LIS(A[1..n]):
    Array L[1..n]
    (* L[i] = value of LIS ending(A[1..i]) *)
    for i = 1 to n do
        L[i] = 1
        for j = 1 to i - 1 do
            if (A[j] < A[i]) do
                L[i] = max(L[i], 1 + L[j])
    return L

MAIN(A[1..n]):
    L = LIS(A[1..n])
        return the maximum value in L
```

Сложность времени в итеративном подходе: `O(n^2)`

Вспомогательное пространство: `O(n)`

Позволяет принимать **{0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15} в** качестве входных данных. Таким образом, наибольшая возрастающая подпоследовательность для данного ввода - **{0, 2, 6, 9, 11, 15}** .

Реализация C

```
public class LongestIncreasingSubsequence
{
    private static int Lis(int[] input, int n)
    {
        int[] lis = new int[n];
        int max = 0;
        for(int i = 0; i < n; i++)
        {
            lis[i] = 1;
        }
        for (int i = 1; i < n; i++)
        {
            for (int j = 0; j < i; j++)
            {
                if (input[i] > input[j] && lis[i] < lis[j] + 1)
                    lis[i] = lis[j] + 1;
            }
        }
        for (int i = 0; i < n; i++)
        {
            if (max < lis[i])
                max = lis[i];
        }
        return max;
    }

    public static int Main(int[] input)
    {
        int n = input.Length;
        return Lis(input, n);
    }
}
```

Прочитайте Самая продолжительная подпоследовательность онлайн:
https://riptutorial.com/ru/algorithm/topic/7537/самая-продолжительная-подпоследовательность

глава 54: Самый низкий общий предок двоичного дерева

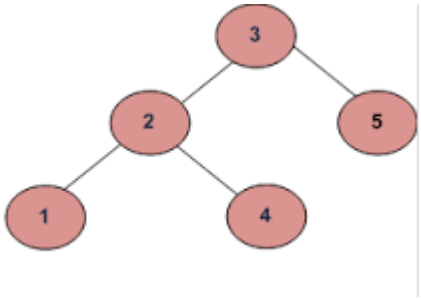
Вступление

Самый низкий общий предк между двумя узлами n1 и n2 определяется как самый низкий узел в дереве, который имеет как n1, так и n2 как потомки.

Examples

Поиск самого низкого общего предка

Рассмотрим дерево:



Самый низкий общий предок узлов со значениями 1 и 4 равен 2

Самый низкий общий предок узлов со значениями 1 и 5 равен 3

Самый низкий общий предок узлов со значениями 2 и 4 равен 4

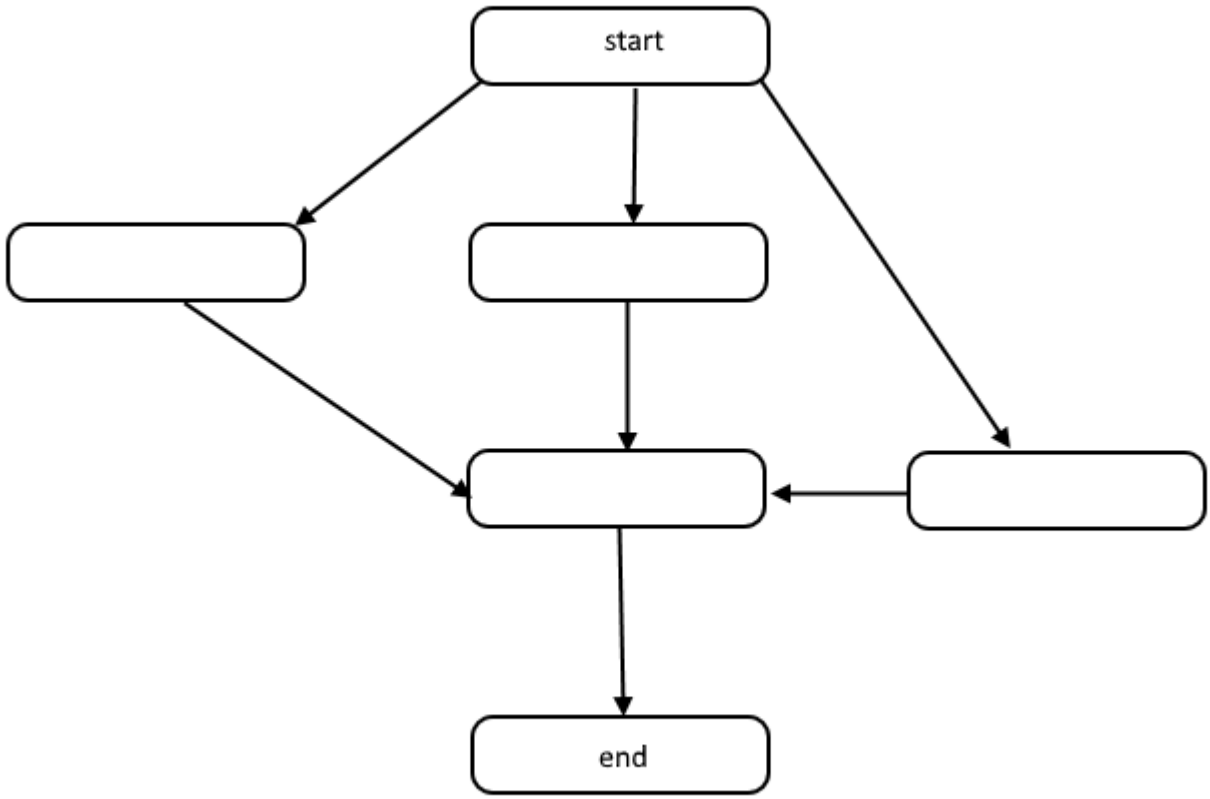
Самый низкий общий предок узлов со значениями 1 и 2 равен 2

Прочитайте Самый низкий общий предок двоичного дерева онлайн:
https://riptutorial.com/ru/algorithm/topic/8848/самый-низкий-общий-предок-двоичного-дерева

глава 55: Сложность алгоритма

замечания

Все алгоритмы - это список шагов для решения проблемы. Каждый шаг имеет зависимости от некоторого набора предыдущих шагов или начала алгоритма. Небольшая проблема может выглядеть следующим образом:



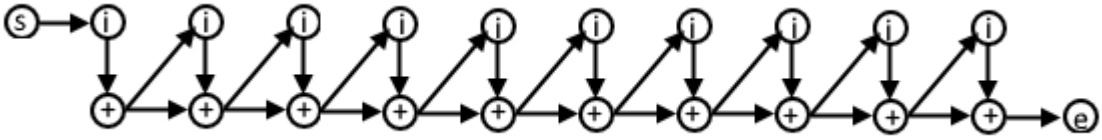
Эта структура называется направленным ациклическим графом или DAG для краткости. Связи между каждым узлом в графике представляют зависимости в порядке операций, а на графике нет циклов.

Как происходят зависимости? Возьмем, к примеру, следующий код:

```
total = 0
for(i = 1; i < 10; i++)
    total = total + i
```

В этом psuedocode каждая итерация цикла for зависит от результата предыдущей итерации, потому что мы используем значение, вычисленное в предыдущей итерации в

этой следующей итерации. DAG для этого кода может выглядеть так:



Если вы понимаете это представление алгоритмов, вы можете использовать его для понимания сложности алгоритма с точки зрения работы и диапазона.

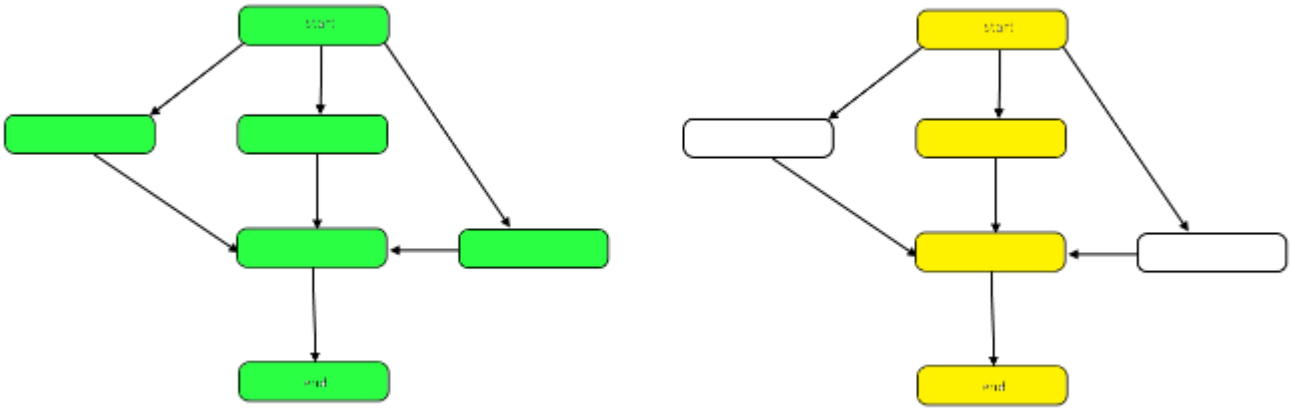
Работа

Работа - это фактическое количество операций, которые необходимо выполнить для достижения цели алгоритма для заданного размера ввода `n` .

пядь

Span иногда называют критическим путем и представляет собой наименьшее количество шагов, которые должен выполнить алгоритм для достижения цели алгоритма.

На следующем рисунке показан график, показывающий различия между работой и интервалом в нашем образце DAG.



Работа - это количество узлов на графике в целом. Это представлено графиком слева выше. Пролет - это критический путь или самый длинный путь от начала до конца. Когда работа может быть выполнена параллельно, желтые выделенные узлы справа представляют диапазон, минимальное количество шагов требуется. Когда работа должна выполняться серийно, пролет такой же, как и работа.

Как работу, так и диапазон можно оценивать независимо с точки зрения анализа.

Скорость алгоритма определяется диапазоном. Объем требуемой вычислительной мощности определяется работой.

Examples

Обозначение Big-Theta

В отличие от нотации Big-O, которая представляет собой только верхнюю границу времени работы для некоторого алгоритма, Big-Theta является плотной границей; как верхняя, так и нижняя граница. Плотная привязка точнее, но сложнее вычислить.

Обозначение Big-Theta симметрично: `f(x) = Θ(g(x)) <=> g(x) = Θ(f(x))`

Интуитивно понятный способ понять это состоит в том, что `f(x) = Θ(g(x))` означает, что графики f (x) и g (x) растут с той же скоростью или что графики «ведут себя» аналогично для больших достаточно значений x.

Полное математическое выражение нотации Большого тета выглядит следующим образом: Θ (f (x)) = {g: N_0 -> R и c_1 , c_2 , n_0 > 0, где c_1 <abs (g (n) / f (n)), для любого n> n_0, а абс - абсолютное значение}

Пример

Если алгоритм для ввода `n` 42n^2 + 25n + 4 операций, мы говорим, что `O(n^2)` , но также `O(n^3)` и `O(n^100)` . Однако это `Θ(n^2)` и это не `Θ(n^3)` , `Θ(n^4)` и т. Д. Алгоритм `Θ(f(n))` также `O(f(n))` , но не наоборот!

Формальное математическое определение

`Θ(g(x))` - множество функций.

```
Θ(g(x)) = {f(x) such that there exist positive constants c1, c2, N such that 0 <= c1*g(x) <= f(x) <= c2*g(x) for all x > N}
```

Поскольку `Θ(g(x))` является множеством, мы могли бы написать `f(x) ∈ Θ(g(x))` чтобы указать, что `f(x)` является членом `Θ(g(x))` . Вместо этого мы обычно будем писать `f(x) = Θ(g(x))` чтобы выразить одно и то же понятие - это общий путь.

Всякий раз, когда `Θ(g(x))` появляется в формуле, мы интерпретируем ее как стоящую за какую-то анонимную функцию, которую мы не хотим называть. Например, уравнение `T(n) = T(n/2) + Θ(n)` означает `T(n) = T(n/2) + f(n)` где `f(n)` - функция из множества `Θ(n)` .

Пусть `f` и `g` - две функции, определенные на некотором подмножестве вещественных чисел. Запишем `f(x) = Θ(g(x))` как `x->infinity` тогда и только тогда, когда существуют положительные константы `K` и `L` и вещественное число `x0` такое, что выполнено:

`K|g(x)| <= f(x) <= L|g(x)|` для всех `x >= x0` .

Определение равно:

`f(x) = O(g(x)) and f(x) = Ω(g(x))`

Метод, который использует ограничения

если `limit(x->infinity) f(x)/g(x) = c ∈ (0,∞)` т. е. предел существует и положителен, то `f(x) = Θ(g(x))`

Общие классы сложности

название	нотация	n = 10	n = 100
постоянная	Θ (1)	1	1
логарифмический	Θ (журнал (п))	3	7
линейный	Θ (п)	10	100
Linearithmic	Θ (п * журнала (п))	30	700
квадратный	Θ (п ^ 2)	100	10 000
экспоненциальный	Θ (2 ^ п)	1 024	1.267650e + 30
Факториал	Θ (п!)	3 628 800	9.332622e + 157

Обозначение Big-Omega

Ω-обозначение используется для асимптотической нижней границы.

Формальное определение

Пусть `f(n)` и `g(n)` - две функции, определенные на множестве положительных вещественных чисел. Запишем `f(n) = Ω(g(n))` если существуют положительные константы `c` и `n0` такие, что:

`0 ≤ cg(n) ≤ f(n) for all n ≥ n0` .

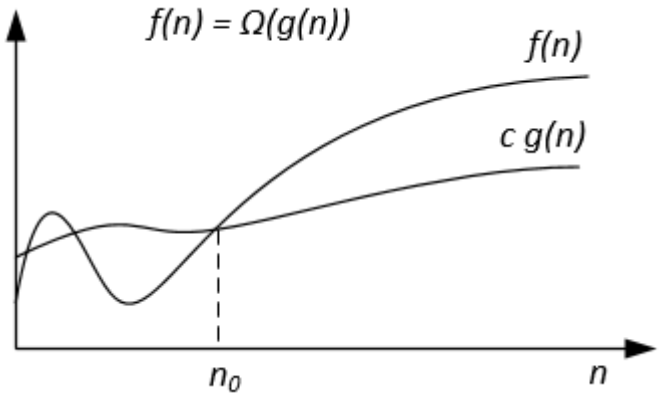
Заметки

`f(n) = Ω(g(n))` означает, что `f(n)` растет асимптотически не медленнее `g(n)` . Также можно сказать о `Ω(g(n))` когда анализа алгоритма недостаточно для утверждения о `Θ(g(n))` или / и `O(g(n))` .

Из определений обозначений следует теорема:

Для двух любых функций `f(n)` и `g(n)` имеем `f(n) = Θ(g(n))` тогда и только тогда, когда `f(n) = O(g(n)) and f(n) = Ω(g(n))` .

Графически Ω-обозначение может быть представлено следующим образом:



Например, пусть `f(n) = 3n^2 + 5n - 4` . Тогда `f(n) = Ω(n^2)` . Это также верно `f(n) = Ω(n)` или даже `f(n) = Ω(1)` .

Другой пример для решения алгоритма идеального совпадения: если число вершин нечетное, тогда выведите «Нет идеального соответствия», в противном случае попробуйте все возможные сопоставления.

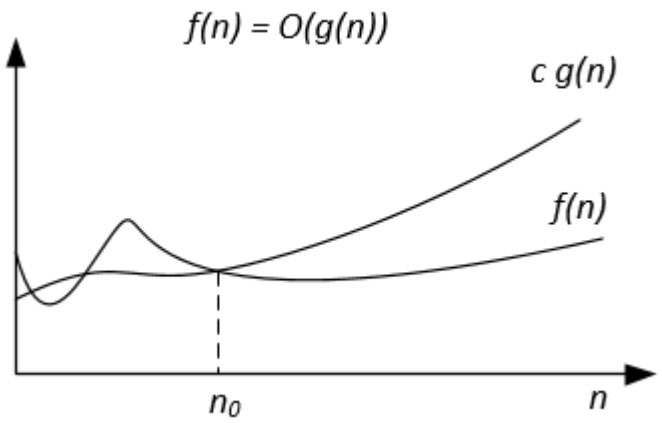
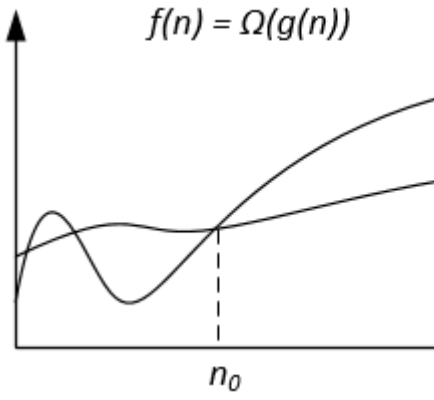
Мы хотели бы сказать, что алгоритм требует экспоненциального времени, но на самом деле вы не можете доказать нижнюю границу `Ω(n^2)` используя обычное определение `Ω` поскольку алгоритм работает в линейном времени для n нечетных. Мы должны вместо этого определить `f(n)=Ω(g(n))` , сказав для некоторой константы `c>0` , `f(n)≥ cg(n)` для бесконечного числа `n` . Это дает хорошее соответствие между верхней и нижней границами: `f(n)=Ω(g(n))` если `f(n) != o(g(n))` .

Рекомендации

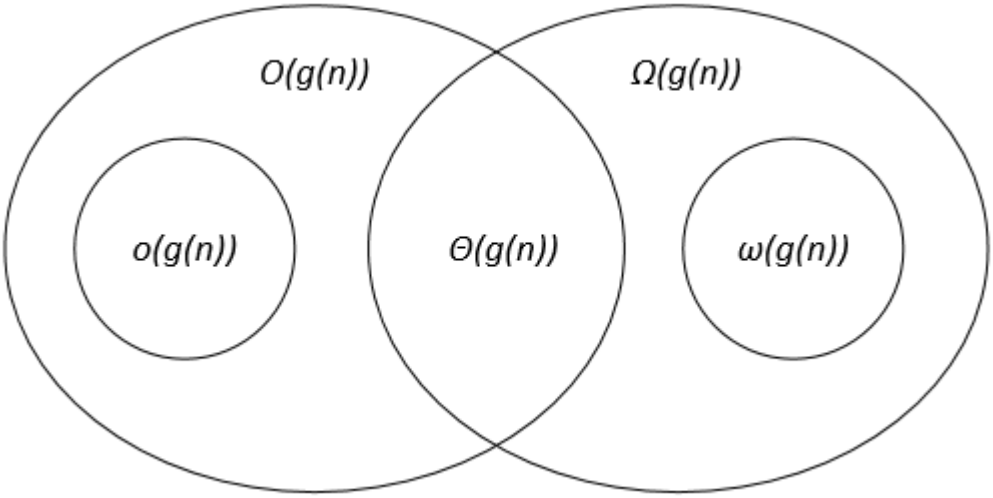
Формальное определение и теорема взяты из книги «Томас Х. Кормен, Чарльз Э. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. Введение в алгоритмы».

Сравнение асимптотических обозначений

Пусть `f(n)` и `g(n)` - две функции, определенные на множестве положительных вещественных чисел, `c, c1, c2, n0` - положительные вещественные константы.

нотация	f (n) = O (g (n))	f (n) = Ω (g (n))
Формальное определение	`∃ c > 0, ∃ n0 > 0 : ∀ n ≥ n0, 0 ≤ f(n) ≤ cg(n)`	`∃ c > 0, ∃ n0 > 0 : ∀ n ≥ n0, 0`
Аналогия между асимптотическим сравнением `f, g` и действительных чисел `a, b`	`a ≤ b`	`a ≥ b`
пример	`7n + 10 = O(n^2 + n - 9)`	`n^3 - 34 = Ω(10n^2 - 7n + 1)`
Графическая интерпретация		

Асимптотические обозначения могут быть представлены на диаграмме Венна следующим образом:



связи

Томас Х. Кормен, Чарльз Э. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. Введение в алгоритмы.

Прочитайте Сложность алгоритма онлайн: https://riptutorial.com/ru/algorithm/topic/1529/сложность-алгоритма

глава 56: Сортировать по

Examples

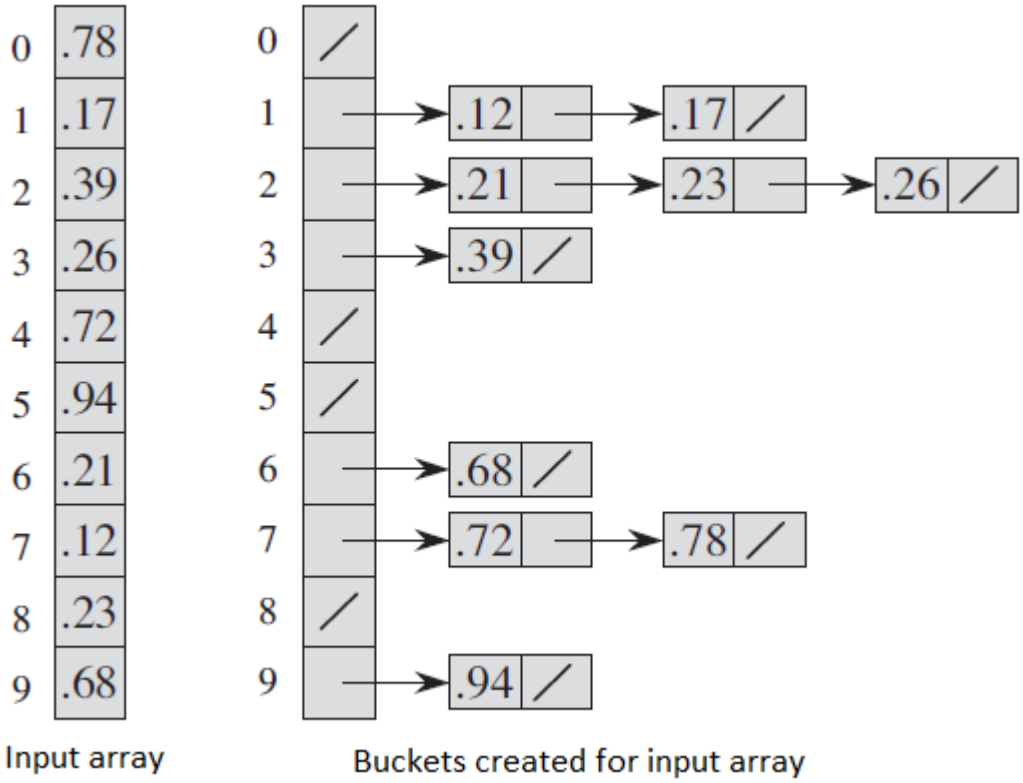
Базовая сортировка

Bucket Sort - это алгоритм сортировки, в котором элементы входного массива распределяются в ведрах. После распределения всех элементов ведра сортируются по отдельности другим алгоритмом сортировки. Иногда он также сортируется рекурсивным методом.

Псевдокод для сортировки ковша

1. Пусть n - длина входного списка L;
2. Для каждого элемента i из L
3. Если B [i] не пусто
4. Положите A [i] в B [i];
5. Else B [i]: = A [i]
6. return Concat B [i .. n] в один отсортированный список;

Пример сортировки ковша:



В основном люди используют парадигму ввода для небольшой оптимизации.
Вспомогательное пространство: `O{n}`

Реализация C

```
public class BucketSort
{
    public static void SortBucket(ref int[] input)
    {
        int minValue = input[0];
        int maxValue = input[0];
        int k = 0;

        for (int i = input.Length - 1; i >= 1; i--)
        {
            if (input[i] > maxValue) maxValue = input[i];
            if (input[i] < minValue) minValue = input[i];
        }

        List<int>[] bucket = new List<int>[maxValue - minValue + 1];

        for (int i = bucket.Length - 1; i >= 0; i--)
        {
            bucket[i] = new List<int>();
        }

        foreach (int i in input)
        {
            bucket[i - minValue].Add(i);
        }

        foreach (List<int> b in bucket)
        {
            if (b.Count > 0)
            {
                foreach (int t in b)
                {
                    input[k] = t;
                    k++;
                }
            }
        }
    }

    public static int[] Main(int[] input)
    {
        SortBucket(ref input);
        return input;
    }
}
```

Прочитайте Сортировать по онлайн: https://riptutorial.com/ru/algorithm/topic/7230/сортировать-по

глава 57: Сортировка

параметры

параметр	Описание
стабильность	Алгоритм сортировки является **стабильным,** если он сохраняет относительный порядок равных элементов после сортировки.
На месте	Алгоритм сортировки **на месте,** если он сортирует, используя только `O(1)` вспомогательную память (не считая массив, который нужно сортировать).
Наилучшая сложность случая	Алгоритм сортировки имеет наилучшую временную сложность `O(T(n))` если ее время работы **не менее** `T(n)` для всех возможных входов.
Средняя сложность случая	Алгоритм сортировки имеет среднюю временную сложность `O(T(n))` если ее время работы, **усредненное по всем возможным входам** , равно `T(n)` .
Худшая сложность случая	Алгоритм сортировки имеет худшую временную сложность `O(T(n))` если ее время работы **не превышает** `T(n)` .

Examples

Стабильность при сортировке

Стабильность в сортировке означает, поддерживает ли алгоритм сортировки относительный порядок равных ключей исходного ввода в выходном результате.

Таким образом, алгоритм сортировки считается стабильным, если два объекта с равными ключами отображаются в том же порядке в отсортированном виде, что и во входном несортированном массиве.

Рассмотрим список пар:

```
(1, 2) (9, 7) (3, 4) (8, 6) (9, 3)
```

Теперь мы отсортируем список, используя первый элемент каждой пары.

При **стабильной сортировке** этого списка будет выводиться следующий список:

```
(1, 2) (3, 4) (8, 6) (9, 7) (9, 3)
```

Потому что `(9, 3)` появляется после `(9, 7)` в исходном списке.

Нестабильная сортировка выведет следующий список:

```
(1, 2) (3, 4) (8, 6) (9, 3) (9, 7)
```

Нестабильная сортировка может генерировать тот же результат, что и стабильный, но не всегда.

Известные стабильные сорта:

- Сортировка слиянием
- Сортировка вставки
- Radix sort
- Тим сортировать
- Сортировка пузырьков

Известные неустойчивые сорта:

- Куча сортировки
- Быстрая сортировка

Прочитайте Сортировка онлайн: https://riptutorial.com/ru/algorithm/topic/821/сортировка

глава 58: Сортировка Pigeonhole

Examples

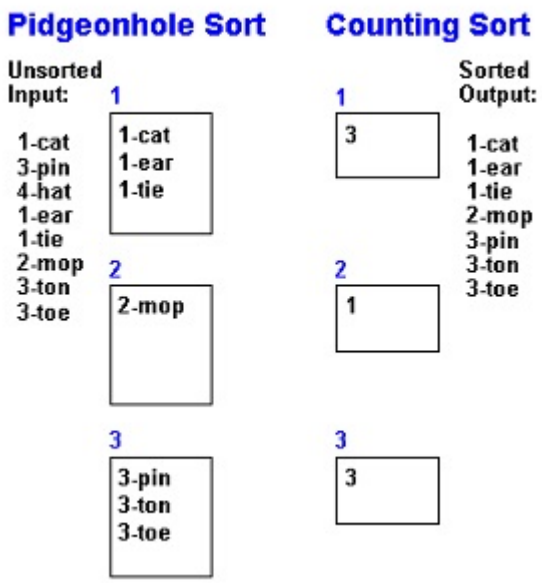
Pigeonhole Сортировать базовую информацию

Pigeonhole Sort - это алгоритм сортировки, который подходит для сортировки списков элементов, где число элементов (n) и количество возможных значений ключа (N) примерно одинаковы. Для этого требуется время O (n + Range), где n - количество элементов в входном массиве, а «Range» - количество возможных значений в массиве.

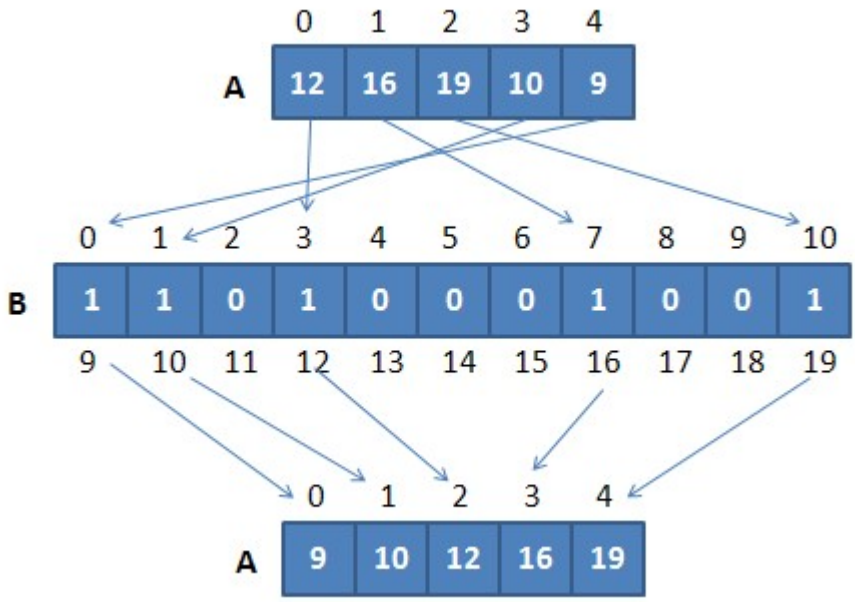
Рабочий (псевдо-код) для Pigeonhole Сортировать:

1. Найти минимальное и максимальное значения в массиве. Пусть минимальное и максимальное значения равны «min» и «max» соответственно. Также найдите диапазон как «max-min-1».
2. Настройте массив изначально пустых «голубинок» того же размера, что и диапазон.
3. Посетите каждый элемент массива, а затем поместите каждый элемент в его яму. Элементный вход [i] помещается в отверстие на входе индекса [i] - мин.
4. Начните цикл по всей решетке пигмента и поместите элементы из непустых отверстий обратно в исходный массив.

Тип Pigeonhole аналогичен сортировке сортировки, поэтому здесь приведено сравнение Сортировки Pigeonhole Sort и counting sort.



Пример Pigeonhole Сортировать:



Вспомогательный объем: `O(n)`
Сложность времени: `O(n + N)`

Реализация C

```
public class PigeonholeSort
{
    private static void SortPigeonhole(int[] input, int n)
    {
        int min = input[0], max = input[n];
        for (int i = 1; i < n; i++)
        {
            if (input[i] < min) min = input[i];
            if (input[i] > max) max = input[i];
        }
        int range = max - min + 1;
        int[] holes = new int[range];

        for (int i = 0; i < n; i++)
        {
            holes[input[i] - min] = input[i];
        }
        int index = 0;

        for (int i = 0; i < range; i++)
        {
            foreach (var value in holes)
            {
                input[index++] = value;
            }
        }
    }

    public static int[] Main(int[] input)
    {
        SortPigeonhole(input, input.Length);
        return input;
    }
```

```
}
```

Прочитайте Сортировка Pigeonhole онлайн: https://riptutorial.com/ru/algorithm/topic/7310/сортировка-pigeonhole

глава 59: Сортировка пузырьков

параметры

параметр	Описание
стабильный	да
На месте	да
Наилучшая сложность случая	На)
Средняя сложность случая	O (N ^ 2)
Худшая сложность случая	O (N ^ 2)
Сложность пространства	O (1)

Examples

Сортировка пузырьков

`BubbleSort` сравнивает каждую последующую пару элементов в неупорядоченном списке и инвертирует элементы, если они не в порядке.

Следующий пример иллюстрирует сортировку пузырьков в списке `{6,5,3,1,8,7,2,4}` (пары, которые были сопоставлены на каждом шаге, инкапсулированы в «**»):

```
{6,5,3,1,8,7,2,4}
{**5,6**,3,1,8,7,2,4} -- 5 < 6 -> swap
{5,**3,6**,1,8,7,2,4} -- 3 < 6 -> swap
{5,3,**1,6**,8,7,2,4} -- 1 < 6 -> swap
{5,3,1,**6,8**,7,2,4} -- 8 > 6 -> no swap
{5,3,1,6,**7,8**,2,4} -- 7 < 8 -> swap
{5,3,1,6,7,**2,8**,4} -- 2 < 8 -> swap
{5,3,1,6,7,2,**4,8**} -- 4 < 8 -> swap
```

Через одну итерацию по списку у нас есть `{5,3,1,6,7,2,4,8}` . Обратите внимание, что наибольшее несортированное значение в массиве (8 в этом случае) всегда достигнет конечной позиции. Таким образом, чтобы отсортировать список, мы должны повторить n-1 раз для списков длины n.

Графика:

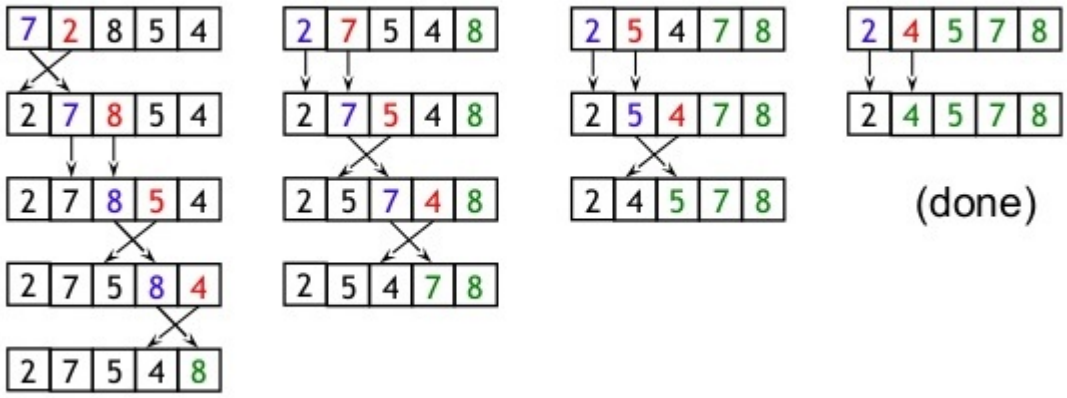
Реализация в Javascript

```
        function bubbleSort(a)
          {
                var swapped;
                do {
                    swapped = false;
                    for (var i=0; i < a.length-1; i++) {
                        if (a[i] > a[i+1]) {
                            var temp = a[i];
                            a[i] = a[i+1];
                            a[i+1] = temp;
                            swapped = true;
                        }
                    }
                } while (swapped);
            }

    var a = [3, 203, 34, 746, 200, 984, 198, 764, 9];
    bubbleSort(a);
    console.log(a); //logs [ 3, 9, 34, 198, 200, 203, 746, 764, 984 ]
```

Реализация в C

Сорт Bubble также известен как **Sinking Sort** . Это простой алгоритм сортировки, который многократно выполняет сортировку списка, сравнивает каждую пару соседних элементов и меняет их, если они находятся в неправильном порядке.

Пример сортировки пузырьков



Реализация сортировки пузырьков

Я использовал язык C # для реализации алгоритма сортировки пузырьков

```
public class BubbleSort
{
    public static void SortBubble(int[] input)
    {
        for (var i = input.Length - 1; i >= 0; i--)
        {
            for (var j = input.Length - 1 - 1; j >= 0; j--)
            {
                if (input[j] <= input[j + 1]) continue;
                var temp = input[j + 1];
                input[j + 1] = input[j];
                input[j] = temp;
            }
        }
    }

    public static int[] Main(int[] input)
    {
        SortBubble(input);
        return input;
    }
}
```

Внедрение в C & C ++

Пример реализации `BubbleSort` в `C++` :

```
   void bubbleSort(vector<int>numbers)
   {
       for(int i = numbers.size() - 1; i >= 0; i--) {
           for(int j = 1; j <= i; j++) {
               if(numbers[j-1] > numbers[j]) {
                   swap(numbers[j-1],numbers(j));
               }
           }
       }
```

```
   }
```

Внедрение C

```
void bubble_sort(long list[], long n)
{
  long c, d, t;

  for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (list[d] > list[d+1])
      {
        /* Swapping */

        t         = list[d];
        list[d]   = list[d+1];
        list[d+1] = t;
      }
    }
  }
}
```

Bubble Сортировать по указателю

```
void pointer_bubble_sort(long * list, long n)
{
  long c, d, t;

  for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if ( * (list + d ) > *(list+d+1))
      {
        /* Swapping */

        t         = * (list + d );
        * (list + d )   = * (list + d + 1 );
        * (list + d + 1) = t;
      }
    }
  }
}
```

Реализация на Java

```
public class MyBubbleSort {

    public static void bubble_srt(int array[]) {//main logic
        int n = array.length;
        int k;
        for (int m = n; m >= 0; m--) {
            for (int i = 0; i < n - 1; i++) {
                k = i + 1;
```

```
                if (array[i] > array[k]) {
                    swapNumbers(i, k, array);
                }
            }
            printNumbers(array);
        }
    }

    private static void swapNumbers(int i, int j, int[] array) {

        int temp;
        temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    private static void printNumbers(int[] input) {

        for (int i = 0; i < input.length; i++) {
            System.out.print(input[i] + ", ");
        }
        System.out.println("\n");
    }

    public static void main(String[] args) {
        int[] input = { 4, 2, 9, 6, 23, 12, 34, 0, 1 };
        bubble_srt(input);

    }
}
```

Реализация Python

```
#!/usr/bin/python

input_list = [10,1,2,11]

for i in range(len(input_list)):
  for j in range(i):
    if int(input_list[j]) > int(input_list[j+1]):
      input_list[j],input_list[j+1] = input_list[j+1],input_list[j]

print input_list
```

Прочитайте Сортировка пузырьков онлайн: https://riptutorial.com/ru/algorithm/topic/1478/сортировка-пузырьков

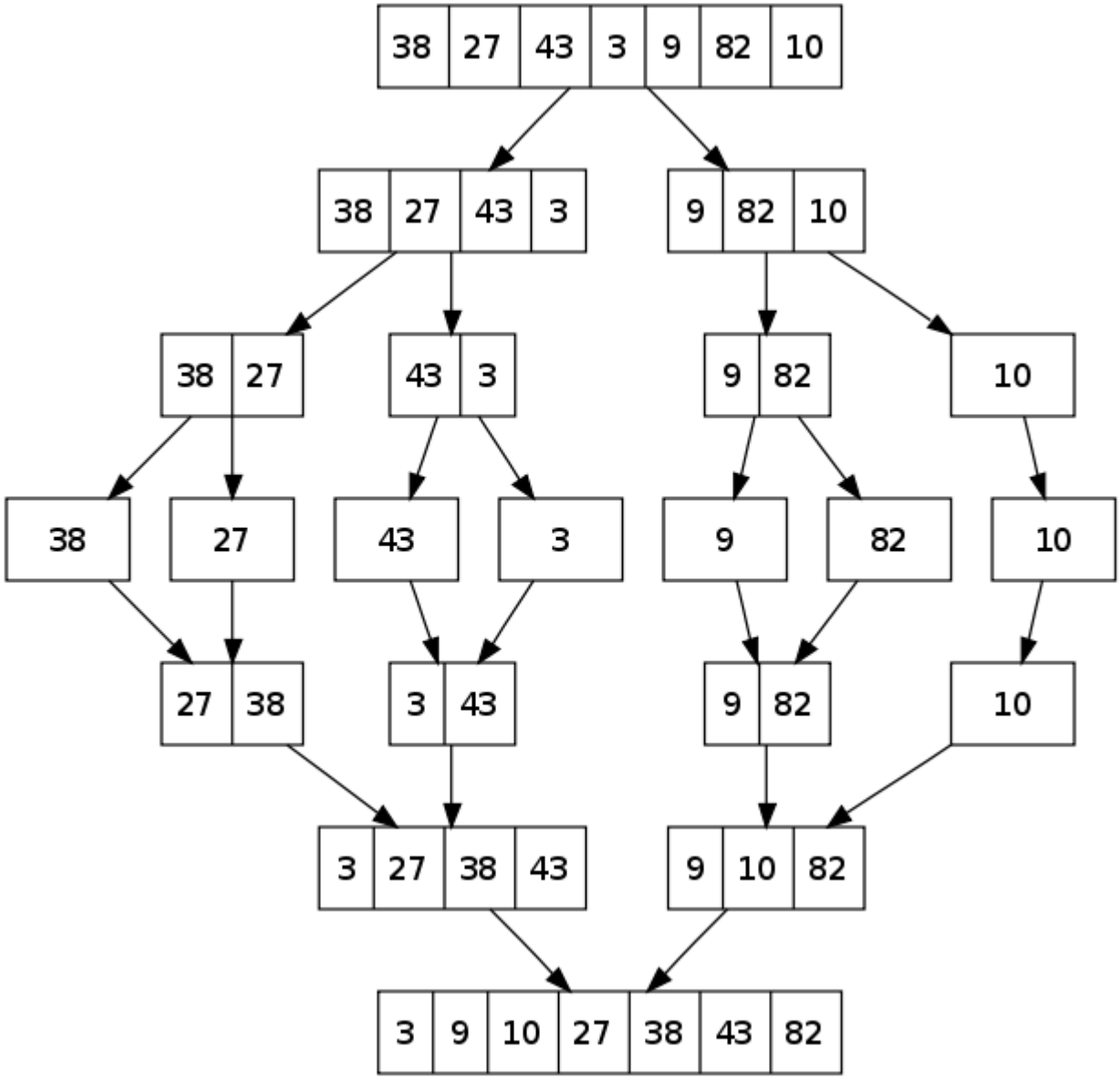
глава 60: Сортировка слиянием

Examples

Объединить сортировки

Merge Sort - алгоритм разделения и покоя. Он делит входной список длины n пополам, пока не будет n списков размера 1. Затем пары списков объединяются вместе с меньшим первым элементом среди пары списков, добавляемых на каждом шаге. После последовательного слияния и сравнения первых элементов создается отсортированный список.

Пример:



Сложность времени : `T(n) = 2T(n/2) + Θ(n)`

Вышеупомянутое повторение может быть решено либо с использованием метода рекурсивного дерева, либо с помощью метода Master. Это относится к случаю II метода Мастера и решению повторения `Θ(nLogn)` . Временная сложность *Merge Sort* - это `Θ(nLogn)` во всех трех случаях (*худший, средний и лучший*), поскольку сортировка слияния всегда делит массив на две половины и принимает линейное время для слияния двух половинок.

Вспомогательное пространство : `O(n)`

Алгоритмическая парадигма : разделить и покорить

Сортировка на месте : не в типичной реализации

Стабильный : Да

Реализация сортировки слияния в C & C

C Сортировка слияния

```
int merge(int arr[],int l,int m,int h)
{
  int arr1[10],arr2[10];  // Two temporary arrays to
  hold the two arrays to be merged
  int n1,n2,i,j,k;
  n1=m-l+1;
  n2=h-m;

  for(i=0; i<n1; i++)
    arr1[i]=arr[l+i];
  for(j=0; j<n2; j++)
    arr2[j]=arr[m+j+1];

  arr1[i]=9999;  // To mark the end of each temporary array
  arr2[j]=9999;

  i=0;
  j=0;
  for(k=l; k<=h; k++) { //process of combining two sorted arrays
    if(arr1[i]<=arr2[j])
      arr[k]=arr1[i++];
    else
      arr[k]=arr2[j++];
  }

  return 0;
}

int merge_sort(int arr[],int low,int high)
{
  int mid;
  if(low<high) {
    mid=(low+high)/2;
    // Divide and Conquer
    merge_sort(arr,low,mid);
    merge_sort(arr,mid+1,high);
    // Combine
    merge(arr,low,mid,high);
```

```
  }

  return 0;
}
```

C # Слияние Сортировка

```
public class MergeSort
    {
        static void Merge(int[] input, int l, int m, int r)
        {
            int i, j;
            var n1 = m - l + 1;
            var n2 = r - m;

            var left = new int[n1];
            var right = new int[n2];

            for (i = 0; i < n1; i++)
            {
                left[i] = input[l + i];
            }

            for (j = 0; j < n2; j++)
            {
                right[j] = input[m + j + 1];
            }

            i = 0;
            j = 0;
            var k = l;

            while (i < n1 && j < n2)
            {
                if (left[i] <= right[j])
                {
                    input[k] = left[i];
                    i++;
                }
                else
                {
                    input[k] = right[j];
                    j++;
                }
                k++;
            }

            while (i < n1)
            {
                input[k] = left[i];
                i++;
                k++;
            }

            while (j < n2)
            {
                input[k] = right[j];
                j++;
                k++;
            }
```

```
        }

        static void SortMerge(int[] input, int l, int r)
        {
            if (l < r)
            {
                int m = l + (r - l) / 2;
                SortMerge(input, l, m);
                SortMerge(input, m + 1, r);
                Merge(input, l, m, r);
            }
        }

        public static int[] Main(int[] input)
        {
            SortMerge(input, 0, input.Length - 1);
            return input;
        }
    }
```

Реализация сортировки Merge в Java

Ниже приведена реализация на Java с использованием подхода generics. Это тот же алгоритм, который представлен выше.

```
public interface InPlaceSort<T extends Comparable<T>> {
void sort(final T[] elements); }


public class MergeSort < T extends Comparable < T >> implements InPlaceSort < T > {


@Override
public void sort(T[] elements) {
    T[] arr = (T[]) new Comparable[elements.length];
    sort(elements, arr, 0, elements.length - 1);
}

// We check both our sides and then merge them
private void sort(T[] elements, T[] arr, int low, int high) {
    if (low >= high) return;
    int mid = low + (high - low) / 2;
    sort(elements, arr, low, mid);
    sort(elements, arr, mid + 1, high);
    merge(elements, arr, low, high, mid);
}


private void merge(T[] a, T[] b, int low, int high, int mid) {
    int i = low;
    int j = mid + 1;

    // We select the smallest element of the two. And then we put it into b
    for (int k = low; k <= high; k++) {

        if (i <= mid && j <= high) {
            if (a[i].compareTo(a[j]) >= 0) {
                b[k] = a[j++];
```

```
            } else {
                b[k] = a[i++];
            }
        } else if (j > high && i <= mid) {
            b[k] = a[i++];
        } else if (i > mid && j <= high) {
            b[k] = a[j++];
        }
    }

    for (int n = low; n <= high; n++) {
        a[n] = b[n];
    }}}
```

Реализация сортировки Merge в Python

```
def merge(X, Y):
    " merge two sorted lists "
    p1 = p2 = 0
    out = []
    while p1 < len(X) and p2 < len(Y):
        if X[p1] < Y[p2]:
            out.append(X[p1])
            p1 += 1
        else:
            out.append(Y[p2])
            p2 += 1
    out += X[p1:] + Y[p2:]
    return out

def mergeSort(A):
    if len(A) <= 1:
        return A
    if len(A) == 2:
        return sorted(A)

    mid = len(A) / 2
    return merge(mergeSort(A[:mid]), mergeSort(A[mid:]))

if __name__ == "__main__":
    # Generate 20 random numbers and sort them
    A = [randint(1, 100) for i in xrange(20)]
    print mergeSort(A)
```

Внедрение Java-реализации

```
public class MergeSortBU {
    private static Integer[] array = { 4, 3, 1, 8, 9, 15, 20, 2, 5, 6, 30, 70,
60,80,0,9,67,54,51,52,24,54,7 };

    public MergeSortBU() {
    }

    private static void merge(Comparable[] arrayToSort, Comparable[] aux, int lo,int mid, int
hi) {
```

```
        for (int index = 0; index < arrayToSort.length; index++) {
            aux[index] = arrayToSort[index];
        }

        int i = lo;
        int j = mid + 1;
        for (int k = lo; k <= hi; k++) {
            if (i > mid)
                arrayToSort[k] = aux[j++];
            else if (j > hi)
                arrayToSort[k] = aux[i++];
            else if (isLess(aux[i], aux[j])) {
                arrayToSort[k] = aux[i++];
            } else {
                arrayToSort[k] = aux[j++];
            }

        }
    }

    public static void sort(Comparable[] arrayToSort, Comparable[] aux, int lo, int hi) {
        int N = arrayToSort.length;
        for (int sz = 1; sz < N; sz = sz + sz) {
            for (int low = 0; low < N; low = low + sz + sz) {
                System.out.println("Size:"+ sz);
                merge(arrayToSort, aux, low, low + sz -1 ,Math.min(low + sz + sz - 1, N - 1));
                print(arrayToSort);
            }
        }

    }

    public static boolean isLess(Comparable a, Comparable b) {
        return a.compareTo(b) <= 0;
    }

    private static void print(Comparable[] array)
{http://stackoverflow.com/documentation/algorithm/5732/merge-sort#
        StringBuffer buffer = new
StringBuffer();http://stackoverflow.com/documentation/algorithm/5732/merge-sort#
        for (Comparable value : array) {
            buffer.append(value);
            buffer.append(' ');
        }
        System.out.println(buffer);
    }

    public static void main(String[] args) {
        Comparable[] aux = new Comparable[array.length];
        print(array);
        MergeSortBU.sort(array, aux, 0, array.length - 1);
    }
}
```

Реализация сортировки слияния в Go

```
package main

import "fmt"
```

```
func mergeSort(a []int) []int {
    if len(a) < 2 {
        return a
    }
    m := (len(a)) / 2

    f := mergeSort(a[:m])
    s := mergeSort(a[m:])

    return merge(f, s)
}

func merge(f []int, s []int) []int {
    var i, j int
    size := len(f) + len(s)

    a := make([]int, size, size)

    for z := 0; z < size; z++ {
        lenF := len(f)
        lenS := len(s)

        if i > lenF-1 && j <= lenS-1 {
            a[z] = s[j]
            j++
        } else if j > lenS-1 && i <= lenF-1 {
            a[z] = f[i]
            i++
        } else if f[i] < s[j] {
            a[z] = f[i]
            i++
        } else {
            a[z] = s[j]
            j++
        }
    }

    return a
}

func main() {
    a := []int{75, 12, 34, 45, 0, 123, 32, 56, 32, 99, 123, 11, 86, 33}
    fmt.Println(a)
    fmt.Println(mergeSort(a))
}
```

Прочитайте Сортировка слиянием онлайн: https://riptutorial.com/ru/algorithm/topic/5732/сортировка-слиянием

глава 61: Треугольник Паскаля

Examples

Основная информация о Pascal's Triagle

Одним из наиболее интересных шаблонов чисел является Треугольник Паскаля . Название «Треугольник Паскаля», названный в честь Блейза Паскаля , известного французского математика и философа.

В математике Треугольник Паскаля представляет собой треугольный массив биномиальных коэффициентов. Строки треугольника Паскаля обычно перечисляются, начиная с строки n = 0 в верхней части (0-я строка). Записи в каждой строке пронумерованы слева от k = 0 и обычно расположены в шахматном порядке относительно чисел в соседних строках.

Треугольник строится следующим образом:

- В самой верхней строке есть уникальная ненулевая запись 1.
- Каждая запись каждой следующей строки строится путем добавления числа выше и слева с номером выше и направо, обрабатывая пустые записи как 0.

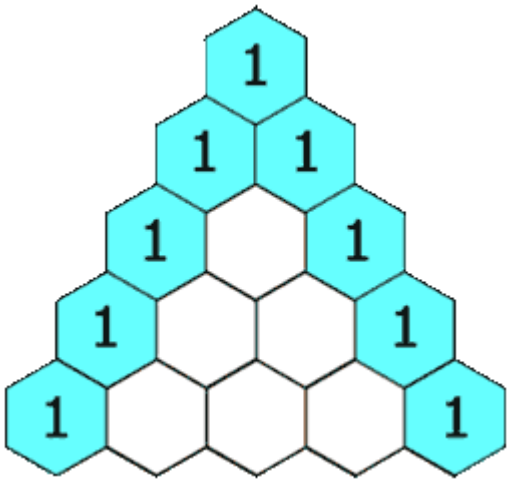
Например, начальное число в первой (или любой другой) строке равно 1 (сумма 0 и 1), тогда как числа 1 и 3 в третьей строке добавляются для создания числа 4 в четвертой строке.

Уравнение для генерации каждой записи в треугольнике Паскаля:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k},$$

для любого неотрицательного целого n и любого целого k между 0 и n включительно. Этот повтор для биномиальных коэффициентов известен как правило Паскаля . Треугольник Паскаля имеет более высокие размерные обобщения. Трехмерная версия называется пирамидой Паскаля или тетраэдром Паскаля, в то время как общие версии называются упрощениями Паскаля.

Пример треугольника Паскаля:



Реализация Треугольника Паскаля в C

```
public class PascalsTriangle
{
    static void PascalTriangle(int n)
    {
        for (int line = 1; line <= n; line++)
        {
            int c = 1;
            for (int i = 1; i <= line; i++)
            {
                Console.WriteLine(c);
                c = c * (line - i) / i;
            }
            Console.WriteLine("\n");
        }
    }

    public static int Main(int input)
    {
        PascalTriangle(input);
        return input;
    }
}
```

Треугольник Паскаля в C

```
int i, space, rows, k=0, count = 0, count1 = 0;
row=5;
for(i=1; i<=rows; ++i)
{
    for(space=1; space <= rows-i; ++space)
    {
        printf("  ");
        ++count;
    }

    while(k != 2*i-1)
    {
        if (count <= rows-1)
        {
            printf("%d ", i+k);
```

```
            ++count;
        }
        else
        {
            ++count1;
            printf("%d ", (i+k-2*count1));
        }
        ++k;
    }
    count1 = count = k = 0;

    printf("\n");
}
```

Выход

```
        1
      2 3 2
    3 4 5 4 3
  4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5
```

Прочитайте Треугольник Паскаля онлайн: https://riptutorial.com/ru/algorithm/topic/7590/треугольник-паскаля

глава 62: Ускорение динамического времени

Examples

Введение в динамическое деформирование времени

Dynamic Time Warping (DTW) - это алгоритм для измерения сходства между двумя временными последовательностями, которые могут варьироваться в зависимости от скорости. Например, сходство в ходьбе можно обнаружить с помощью DTW, даже если один человек идет быстрее, чем другой, или если в ходе наблюдения наблюдались ускорения и замедление. Его можно использовать для сопоставления с образцовой голосовой командой с другими командами, даже если человек говорит быстрее или медленнее, чем предварительно записанный образец голоса. DTW может применяться к временным последовательностям видео-, аудио- и графических данных - действительно, любые данные, которые могут быть преобразованы в линейную последовательность, могут быть проанализированы с помощью DTW.

В общем, DTW - это метод, который вычисляет оптимальное совпадение между двумя заданными последовательностями с определенными ограничениями. Но давайте придерживаться более простых моментов здесь. Предположим, у нас есть две голосовые последовательности **Sample** и **Test** , и мы хотим проверить, соответствуют ли эти две последовательности или нет. Здесь голосовая последовательность относится к преобразованному цифровому сигналу вашего голоса. Это может быть амплитуда или частота вашего голоса, который обозначает слова, которые вы говорите. Предположим:

```
Sample = {1, 2, 3, 5, 5, 5, 6}
Test   = {1, 1, 2, 2, 3, 5}
```

Мы хотим найти оптимальное совпадение между этими двумя последовательностями.

Сначала мы определяем расстояние между двумя точками, *d (x, y),* где **x** и **y** представляют две точки. Позволять,

```
d(x, y) = |x - y|     //absolute difference
```

Давайте создадим **таблицу** 2D-матрицы, используя эти две последовательности. Мы рассчитаем расстояния между каждой точкой **образца** с каждой точкой **теста** и найдем оптимальное совпадение между ними.

```
+------+------+------+------+------+------+------+------+
|      |   0  |   1  |   1  |   2  |   2  |   3  |   5  |
```

```
+------+------+------+------+------+------+------+------+
|   0  |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   1  |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   2  |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   3  |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   5  |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   5  |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   5  |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   6  |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
```

Здесь **таблица [i] [j]** представляет оптимальное расстояние между двумя последовательностями, если мы рассмотрим последовательность до **Sample [i]** и **Test [j]** , учитывая все оптимальные расстояния, которые мы наблюдали ранее.

Для первой строки, если мы не принимаем значения из **Sample** , расстояние между этим и **Test** будет *бесконечным* . Поэтому мы помещаем *бесконечность* в первую строку. То же самое касается первого столбца. Если мы не будем принимать значения из **теста** , расстояние между этим и **Sample** также будет бесконечным. И расстояние между **0** и **0** будет просто **0** . Мы получаем,

```
+------+------+------+------+------+------+------+------+
|      |   0  |   1  |   1  |   2  |   2  |   3  |   5  |
+------+------+------+------+------+------+------+------+
|   0  |   0  |  inf |  inf |  inf |  inf |  inf |  inf |
+------+------+------+------+------+------+------+------+
|   1  |  inf |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   2  |  inf |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   3  |  inf |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   5  |  inf |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   5  |  inf |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   5  |  inf |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   6  |  inf |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
```

Теперь для каждого шага мы рассмотрим расстояние между каждым интересующим вас пунктом и добавим его с минимальным расстоянием, которое мы обнаружили до сих пор. Это даст нам оптимальное расстояние двух последовательностей до этой позиции. Наша формула будет,

```
Table[i][j] := d(i, j) + min(Table[i-1][j], Table[i-1][j-1], Table[i][j-1])
```

Для первого, **d (1, 1)** = **0** , **таблица [0] [0]** представляет собой минимум. Таким образом, значение **таблицы [1] [1]** будет равно **0** + **0** = **0** . Для второго **d (1, 2)** = **0** . **Таблица [1] [1]** представляет собой минимум. Значение будет: **Таблица [1] [2]** = **0** + **0** = **0** . Если мы продолжим этот путь, после окончания таблицы таблица будет выглядеть так:

```
+------+------+------+------+------+------+------+------+
|      |   0  |   1  |   1  |   2  |   2  |   3  |   5  |
+------+------+------+------+------+------+------+------+
|   0  |   0  |  inf |  inf |  inf |  inf |  inf |  inf |
+------+------+------+------+------+------+------+------+
|   1  |  inf |   0  |   0  |   1  |   2  |   4  |   8  |
+------+------+------+------+------+------+------+------+
|   2  |  inf |   1  |   1  |   0  |   0  |   1  |   4  |
+------+------+------+------+------+------+------+------+
|   3  |  inf |   3  |   3  |   1  |   1  |   0  |   2  |
+------+------+------+------+------+------+------+------+
|   5  |  inf |   7  |   7  |   4  |   4  |   2  |   0  |
+------+------+------+------+------+------+------+------+
|   5  |  inf |  11  |  11  |   7  |   7  |   4  |   0  |
+------+------+------+------+------+------+------+------+
|   5  |  inf |  15  |  15  |  10  |  10  |   6  |   0  |
+------+------+------+------+------+------+------+------+
|   6  |  inf |  20  |  20  |  14  |  14  |   9  |   1  |
+------+------+------+------+------+------+------+------+
```

Значение в **таблице [7] [6]** представляет собой максимальное расстояние между этими двумя заданными последовательностями. Здесь **1** представляет максимальное расстояние между **образцом** и **тестом 1** .

Теперь, если мы отступим от последней точки, все пути назад к стартовой точке **(0, 0)** , мы получим длинную линию, которая перемещается горизонтально, вертикально и по диагонали. Наша процедура обратного отслеживания будет:

```
if Table[i-1][j-1] <= Table[i-1][j] and Table[i-1][j-1] <= Table[i][j-1]
    i := i - 1
    j := j - 1
else if Table[i-1][j] <= Table[i-1][j-1] and Table[i-1][j] <= Table[i][j-1]
    i := i - 1
else
    j := j - 1
end if
```

Мы продолжим это, пока не достигнем **(0, 0)** . Каждый ход имеет свое значение:

- Горизонтальное перемещение представляет собой удаление. Это означает, что наша **тестовая** последовательность ускорилась в течение этого интервала.
- Вертикальный ход представляет собой вставку. Это означает, что **тестовая** последовательность замедляется в течение этого интервала.
- Диагональный ход представляет собой совпадение. В течение этого периода **тест** и **образец** были **такими** же.

		0	1	1	2	2	3	5
0	0	inf	inf	inf	inf	inf	inf
1	inf	0	0	1	2	4	8
2	inf	1	1	0	0	1	4
3	inf	3	3	1	1	0	2
5	inf	7	7	4	4	2	0
5	inf	11	11	7	7	4	0
5	inf	15	15	10	10	6	0
6	inf	20	20	14	14	9	1

Наш псевдокод будет:

```
Procedure DTW(Sample, Test):
n := Sample.length
m := Test.length
Create Table[n + 1][m + 1]
for i from 1 to n
    Table[i][0] := infinity
end for
for i from 1 to m
    Table[0][i] := infinity
end for
Table[0][0] := 0
for i from 1 to n
    for j from 1 to m
        Table[i][j] := d(Sample[i], Test[j])
                       + minimum(Table[i-1][j-1],      //match
                                 Table[i][j-1],        //insertion
                                 Table[i-1][j])        //deletion
    end for
end for
Return Table[n + 1][m + 1]
```

Мы также можем добавить ограничение локальности. То есть, мы требуем, чтобы если `Sample[i]` соответствовал `Test[j]` , то `|i - j|` не больше **w** , параметр окна.

Сложность:

Сложность вычисления DTW равна **O (m * n),** где **m** и **n** представляют длину каждой последовательности. Более быстрые методы для вычисления DTW включают PrunedDTW, SparseDTW и FastDTW.

Приложения:

- Распознавание слов
- Анализ корреляции мощности

Прочитайте Ускорение динамического времени онлайн:
https://riptutorial.com/ru/algorithm/topic/7584/ускорение-динамического-времени

глава 63: Хэш-функции

Examples

Введение в хэш-функции

Хэш-функция `h()` - это произвольная функция, которая отображает данные `x ∈ X` произвольного размера в значение `y ∈ Y` фиксированного размера: `y = h(x)` . Хорошие хэш-функции имеют следующие ограничения:

- хеш-функции ведут себя одинаково
- хэш-функции детерминированы. `h(x)` всегда должно возвращать одно и то же значение для заданного `x`
- быстрый расчет (имеет время выполнения O (1))

В общем случае размер хеш-функции меньше размера входных данных: `|y| < |x|` , Хэш-функции не обратимы, иначе говоря, это может быть столкновение: `∃ x1, x2 ∈ X, x1 ≠ x2: h(x1) = h(x2)` . `X` может быть конечным или бесконечным множеством, а `Y` - конечным множеством.

Функции хэши используются во многих частях информатики, например, в разработке программного обеспечения, криптографии, базах данных, сетях, машинах и т. Д. Существует множество различных типов хеш-функций с различными свойствами домена.

Часто hash представляет собой целочисленное значение. Существуют специальные методы программирования языков для вычисления хешей. Например, в методе `C# GetHashCode()` для всех типов возвращается значение `Int32` (32-битное целочисленное число). В `Java` каждый класс предоставляет метод `hashCode()` который возвращает `int` . Каждый тип данных имеет собственные или определенные пользователем реализации.

Хэш-методы

Существует несколько подходов к детерминированной хэш-функции. Без ограничения общности, пусть `x ∈ X = {z ∈ ℤ: z ≥ 0}` - положительные целые числа. Часто `m` является простым (не слишком близко к точной мощности 2).

метод	Хэш-функция
Метод разделения	`h(x) = x mod m`
Метод умножения	`h(x) = ⌊m (xA mod 1)⌋, A ∈ {z ∈ ℝ: 0 < z < 1}`

Хеш-таблица

Хэш-функции, используемые в хэш-таблицах для вычисления индекса в массив слотов. Хэш-таблица - это структура данных для реализации словарей (структура ключевых значений). Хорошие реализованные хеш-таблицы имеют O (1) время для следующих операций: вставка, поиск и удаление данных по ключу. Более одного ключа могут иметь хэш в том же слоте. Существует два способа разрешения столкновения:

1. Цепочка: связанный список используется для хранения элементов с одинаковым значением хэша в слоте
2. Открытая адресация: нулевой или один элемент сохраняется в каждом слоте

Следующие методы используются для вычисления последовательностей зондов, необходимых для открытой адресации

метод	формула
Линейное зондирование	`h(x, i) = (h'(x) + i) mod m`
Квадратичное зондирование	`h(x, i) = (h'(x) + c1*i + c2*i^2) mod m`
Двойной хэш	`h(x, i) = (h1(x) + i*h2(x)) mod m`

Где `i ∈ {0, 1, ..., m-1}` , `h'(x), h1(x), h2(x)` - вспомогательные хэш-функции, `c1, c2` - положительные вспомогательные константы.

Примеры

Пусть `x ∈ U{1, 1000}, h = x mod m` . Следующая таблица показывает значения хэша в случае не простых и простых. Полужирный текст указывает те же значения хэша.

Икс	m = 100 (не простое)	m = 101 (простой)
+723	23	16
103	3	2
+738	38	31
292	92	90
61	61	61

Икс	m = 100 (не простое)	m = 101 (простой)
87	87	87
995	95	86
549	49	44
+991	91	82
757	**57**	50
920	20	11
626	26	20
557	**57**	52
831	31	23
619	19	13

СВЯЗИ

- Томас Х. Кормен, Чарльз Э. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. Введение в алгоритмы.
- Обзор таблиц Hash
- Wolfram MathWorld - Функция хэша

Хэш коды для обычных типов в C

Хэш-коды, созданные методом `GetHashCode()` для встроенных и общих типов C # из пространства имен `System` , показаны ниже.

логический

1, если значение истинно, 0 в противном случае.

Байт , UInt16 , Int32 , UInt32 , Single

Значение (если необходимо, заносится в Int32).

SByte

```
((int)m_value ^ (int)m_value << 8);
```

голец

```
(int)m_value ^ ((int)m_value << 16);
```

Int16

```
((int)((ushort)m_value) ^ (((int)m_value) << 16));
```

Int64 , Double

Xor между нижним и верхним 32 битами 64-разрядного номера

```
(unchecked((int)((long)m_value)) ^ (int)(m_value >> 32));
```

UInt64 , DateTime , TimeSpan

```
((int)m_value) ^ (int)(m_value >> 32);
```

Десятичный

```
((((int *)&dbl)[0]) & 0xFFFFFFF0) ^ ((int *)&dbl)[1];
```

объект

```
RuntimeHelpers.GetHashCode(this);
```

В реализации по умолчанию используется индекс блока синхронизации .

строка

Вычисление кода хэш-кода зависит от типа платформы (Win32 или Win64), особенности использования хеширования рандомизированных строк, режима отладки / выпуска. В случае платформы Win64:

```
int hash1 = 5381;
int hash2 = hash1;
int c;
char *s = src;
while ((c = s[0]) != 0) {
    hash1 = ((hash1 << 5) + hash1) ^ c;
    c = s[1];
    if (c == 0)
        break;
```

```
    hash2 = ((hash2 << 5) + hash2) ^ c;
    s += 2;
}
return hash1 + (hash2 * 1566083941);
```

Тип ценности

Первое нестатическое поле ищет и получает его hashcode. Если тип не имеет нестатических полей, возвращается хэш-код типа. Хэш-код статического члена нельзя принять, потому что если этот элемент имеет тот же тип, что и исходный тип, вычисление заканчивается в бесконечном цикле.

Nullable <T>

```
return hasValue ? value.GetHashCode() : 0;
```

массив

```
int ret = 0;
for (int i = (Length >= 8 ? Length - 8 : 0); i < Length; i++)
{
    ret = ((ret << 5) + ret) ^ comparer.GetHashCode(GetValue(i));
}
```

Рекомендации

- GitHub .Net Core CLR

Прочитайте Хэш-функции онлайн: https://riptutorial.com/ru/algorithm/topic/6204/хэш-функции

глава 64: Целочисленный алгоритм разделения

Examples

Основная информация алгоритма разделения целых чисел

Разделение целого числа является способом записи его в виде суммы положительных целых чисел. Например, разделы числа 5:

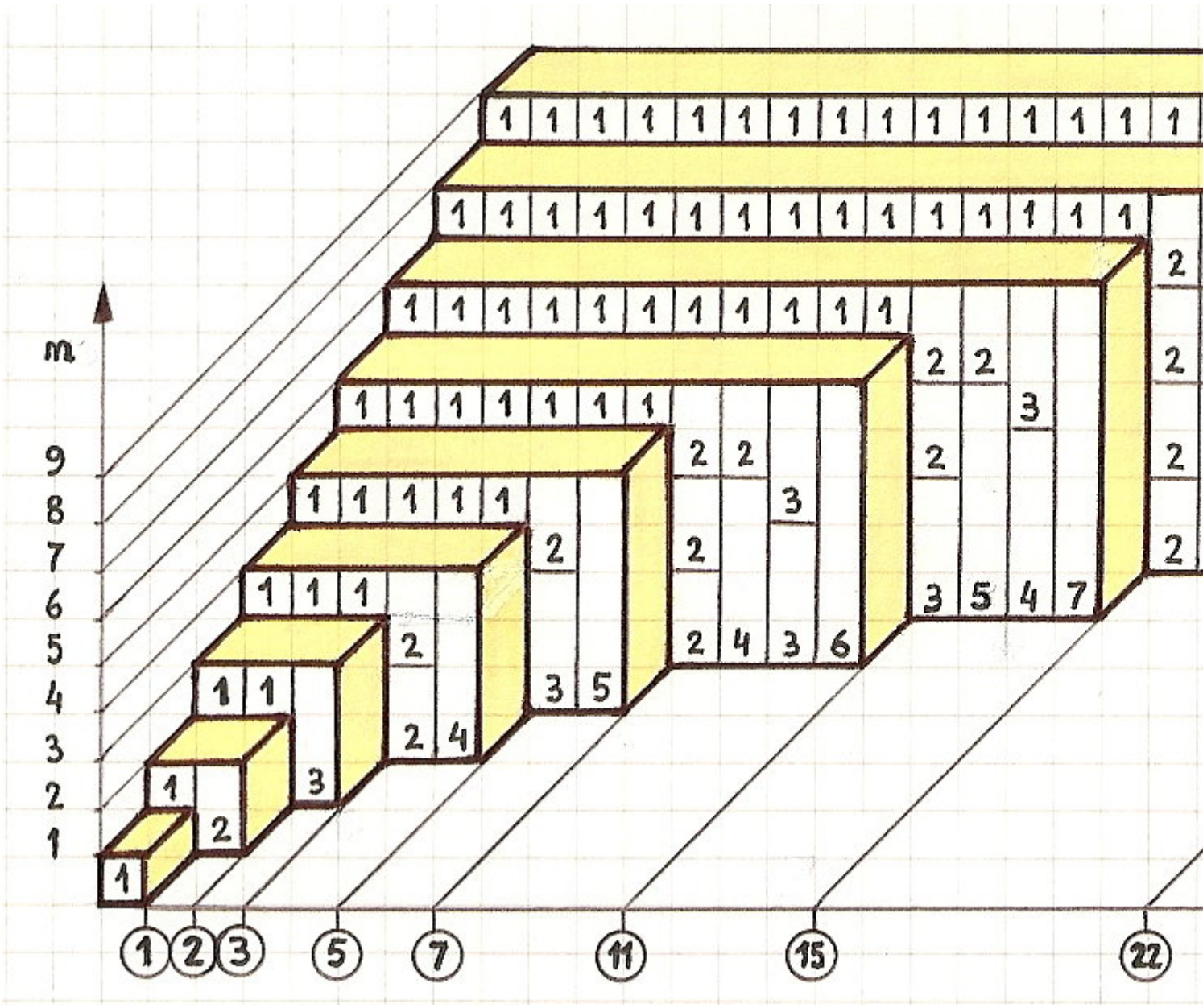
- 5
- 4 + 1
- 3 + 2
- 2 + 2 + 1
- 2 + 1 + 1 + 1
- 1 + 1 + 1 + 1 + 1

Обратите внимание, что изменение порядка слагаемых не создаст другой раздел.

Функция разбиения носит по своей природе рекурсивный характер, так как результаты меньших чисел появляются как компоненты в результате большего числа. Пусть *p (n, m)* - число разбиений *n,* используя только положительные целые числа, которые меньше или равны *m* . Можно видеть, что *p (n) = p (n, n)* , а также *p (n, m) = p (n, n) = p (n)* при *m > n* .

$$p(n,m) = \sum_{k=1}^{m} p(n-k,k)$$

Пример целочисленного алгоритма разбиения:



Вспомогательное пространство: `O(n^2)`
Сложность времени: `O(n(logn))`

Реализация алгоритма Interger Partition в C

```
 public class IntegerPartition
{
    public static int[,] Result = new int[100,100];

    private static int Partition(int targetNumber, int largestNumber)
    {
        for (int i = 1; i <= targetNumber; i++)
        {
            for (int j = 1; j <= largestNumber; j++)
            {
                if (i - j < 0)
                {
                    Result[i, j] = Result[i, j - 1];
                    continue;
```

```
                }
                Result[i, j] = Result[i, j - 1] + Result[i - j, j];
            }
        }
        return Result[targetNumber, largestNumber];
    }

    public static int Main(int number, int target)
    {
        int i;
        for (i = 0; i <= number; i++)
        {
            Result[i, 0] = 0;
        }
        for (i = 1; i <= target; i++)
        {
            Result[0, i] = 1;
        }
        return Partition(number, target);
    }
}
```

Прочитайте Целочисленный алгоритм разделения онлайн:
https://riptutorial.com/ru/algorithm/topic/7424/целочисленный-алгоритм-разделения

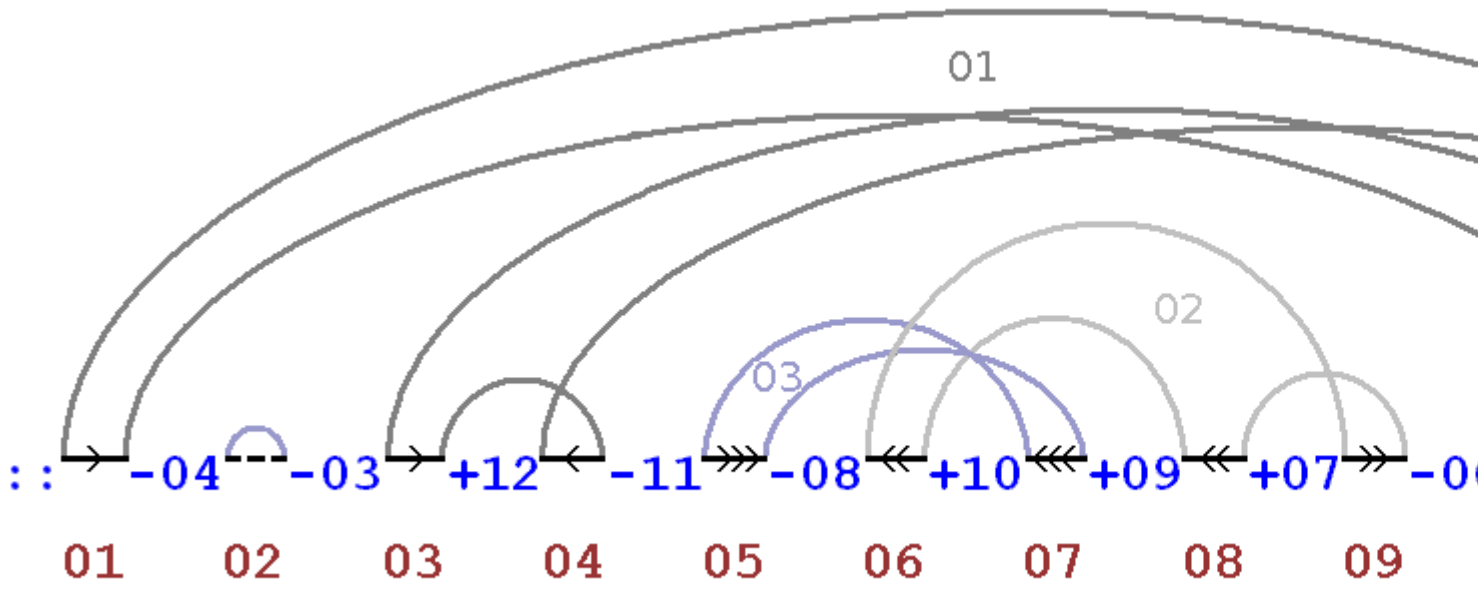
глава 65: Цикл сортировки

Examples

Цикл сортировки Основная информация

Cycle Sort - алгоритм сортировки, который использует сортировку сравнения, которая теоретически оптимальна с точки зрения общего количества записей в исходном массиве, в отличие от любого другого алгоритма сортировки на месте. Сортировка циклов - это неустойчивый алгоритм сортировки. Он основан на идее перестановки, в которой перестановки учитываются в циклах, которые индивидуально вращают и возвращают отсортированный результат.

Пример сортировки цикла:



Вспомогательное пространство: `O(1)`
Сложность времени: `O(n^2)`

Реализация псевдокода

```
(input)
output = 0
for cycleStart from 0 to length(array) - 2
    item = array[cycleStart]
    pos = cycleStart
    for i from cycleStart + 1 to length(array) - 1
        if array[i] < item:
            pos += 1
    if pos == cycleStart:
        continue
```

```
    while item == array[pos]:
        pos += 1
    array[pos], item = item, array[pos]
    writes += 1
    while pos != cycleStart:
        pos = cycleStart
        for i from cycleStart + 1 to length(array) - 1
            if array[i] < item:
                pos += 1
        while item == array[pos]:
            pos += 1
        array[pos], item = item, array[pos]
        writes += 1
return outout
```

Реализация C

```
public class CycleSort
{
    public static void SortCycle(int[] input)
    {
        for (var i = 0; i < input.Length; i++)
        {
            var item = input[i];
            var position = i;
            do
            {
                var k = input.Where((t, j) => position != j && t < item).Count();
                if (position == k) continue;
                while (position != k && item == input[k])
                {
                    k++;
                }
                var temp = input[k];
                input[k] = item;
                item = temp;
                position = k;
            } while (position != i);
        }
    }

    public static int[] Main(int[] input)
    {
        SortCycle(input);
        return input;
    }
}
```

Прочитайте Цикл сортировки онлайн: https://riptutorial.com/ru/algorithm/topic/7252/цикл-сортировки

кредиты

S. No	Главы	Contributors
1	Начало работы с алгоритмом	Abdul Karim, Austin Conlon, C L K Kissane, Community, EsmaeelE, Filip Allberg, Hesham Attia, Jonathan Landrum, msohng, Sayakiss, user2314737
2	A * Pathfinding	kiner_shah, Minhas Kamal, mnoronha, Roberto Fernandez, TajyMany
3	A * Алгоритм поиска пути	TajyMany
4	Algo: - Печатать am * n матрицу в квадратной форме	Creative John
5	Odd-Even Сортировать	Keyur Ramoliya
6	Quicksort	Bakhtiar Hasan, Keyur Ramoliya, Malav, mnoronha, optimistanoop
7	Radix Sort	Keyur Ramoliya, mnoronha, Zopesconk
8	Алгоритм Беллмана-Форда	Bakhtiar Hasan, Sumeet Singh, user2314737, Yerken
9	Алгоритм Дейкстры	Bakhtiar Hasan, Tejus Prasad
10	Алгоритм каталонского номера	Keyur Ramoliya, mnoronha
11	Алгоритм Кнут Моррис Пратт (KMP)	Vishwas
12	Алгоритм Крускаля	IVlad, Shubham, Yerken
13	Алгоритм максимального субаруса	Keyur Ramoliya, mnoronha
14	Алгоритм максимальной длины пути	Keyur Ramoliya, mnoronha

15	Алгоритм Прима	Bakhtiar Hasan, Tejus Prasad
16	Алгоритм раздвижного окна	Keyur Ramoliya
17	Алгоритм Флойда-Варшалла	Bakhtiar Hasan, Sayakiss
18	Быстрое преобразование Фурье	Dr. ABT, EsmaeelE
19	Вставка Сортировка	Bakhtiar Hasan, invisal, Keyur Ramoliya, Lymphatus, mnoronha, RamenChef
20	Выбор Сортировка	Keyur Ramoliya, lambda, mnoronha, Teodor Kurtev, user2314737
21	Вычисление матрицы	Bakhtiar Hasan, mnoronha
22	Глубина первого поиска	Bakhtiar Hasan
23	график	Ahmed Mazher, Bakhtiar Hasan, EsmaeelE, Filip Allberg, hurricane, JJTO, John Odom, ldog, Sayakiss, Tejus Prasad , user23013, VermillionAzure
24	Графические обходы	Dian Bakti
25	деревья	Isha Agarwal, Malcolm McLean, mnoronha, VermillionAzure, yd1
26	Деревья двоичного поиска	a13ph, EsmaeelE, greatwolf, Isha Agarwal, Ishit Mehta, Mehedi Hasan, nbro, RamenChef, Rashik Hasnat, Tejus Prasad
27	Динамическое программирование	Bakhtiar Hasan, Benson Lin, kraskevich, Muyide Ibukun, nbro, RamenChef, Razik, Sayakiss, Vishwas
28	Жадные алгоритмы	Bakhtiar Hasan, Cameron, Community, ghilesZ, M S Hossain, theJollySin, xenteros
29	Изменение динамического алгоритма расстояния	Vishwas
30	Интернет-алгоритмы	Andrii Artamonov, goeddek
31	Кратчайшая общая проблема суперсимметрии	Keyur Ramoliya

32	Куча сортировки	Keyur Ramoliya, mnoronha
33	Линейный алгоритм	Dipesh Poudel, Martin Frank
34	Масленица	Keyur Ramoliya, mnoronha
35	Многопоточные алгоритмы	Julien Rousé
36	Обозначение Big-O	Community, EsmaeelE, mnoronha, msohng, Nick the coder , Samuel Peter, user2314737, WitVault
37	Оболочка	Keyur Ramoliya, mnoronha
38	Подстрочный поиск	AnukuL, Bakhtiar Hasan, mnoronha, Rashik Hasnat
39	Подсчет сортировки	Bakhtiar Hasan, Keyur Ramoliya, mnoronha
40	поиск	Anagh Hegde, Benson Lin, brijs, Community, EsmaeelE, Iwan, Khaled.K, Malcolm McLean, Miljen Mikic, msohng, RamenChef, ShreePool, Timothy G., umop apisdn, xenteros
41	Поиск по ширине	Bakhtiar Hasan, mnoronha, Sumeet Singh, Zubayet Zaman Zico
42	полиномиально ограниченный алгоритм для минимальной вершины	Alber Tadrous
43	Приложения динамического программирования	Chris, user2314737
44	Применение жадности	EsmaeelE, goeddek, Tejus Prasad, user2314737
45	Проблема с рюкзаком	Bakhtiar Hasan, dtt, Keyur Ramoliya, mnoronha, Tejus Prasad, user2314737
46	Проверьте две строки: анаграммы	Creative John
47	Проверьте, нет ли дерева BST или нет.	Isha Agarwal, Janaky Murthy
48	Прохождение двоичных деревьев	Isha Agarwal

49	ПСЕВДОКОД	Community
50	Путешественник	Benson Lin
51	Решение уравнений	Minhas Kamal
52	Самая длинная общая подпоследовательность	Bakhtiar Hasan, Keyur Ramoliya
53	Самая продолжительная подпоследовательность	Keyur Ramoliya, mnoronha
54	Самый низкий общий предок двоичного дерева	Isha Agarwal
55	Сложность алгоритма	A. Raza, Daniel Nugent, Didgeridoo, EsmaeelE, fgb, Juxhin Metaj, Miljen Mikic, Nick Larsen, Peter K, Sayakiss, Tejus Prasad, user23013, user2314737, VermillionAzure, xenteros, Yair Twito
56	Сортировать по	Keyur Ramoliya, mnoronha
57	Сортировка	Ahmad Faiyaz, Carlton, Filip Allberg, Frank, ganesshkumar, IVlad, Iwan, Kedar Mhaswade, Miljen Mikic, mok, Patrick87, RamenChef, Rob Fagen, Set
58	Сортировка Pigeonhole	Keyur Ramoliya, mnoronha
59	Сортировка пузырьков	Anagh Hegde, Deepak, EsmaeelE, Ijaz Khan, Keyur Ramoliya, mnoronha, optimistanoop, samgak, xenteros, YoungHobbit
60	Сортировка слиянием	EsmaeelE, Iwan, Juxhin Metaj, Keyur Ramoliya, Luv Agarwal, mnoronha, Santiago Gil, SHARMA
61	Треугольник Паскаля	EsmaeelE, Keyur Ramoliya
62	Ускорение динамического времени	Bakhtiar Hasan, mnoronha, Zubayet Zaman Zico
63	Хэш-функции	afeldspar, Didgeridoo, mnoronha
64	Целочисленный алгоритм разделения	Keyur Ramoliya
65	Цикл сортировки	Keyur Ramoliya, mnoronha