



EBook Gratis

APRENDIZAJE algorithm

Free unaffiliated eBook created from
Stack Overflow contributors.

#algorithm

Tabla de contenido

Acerca de.....	1
Capítulo 1: Empezando con el algoritmo.....	2
Observaciones.....	2
Introducción a los algoritmos.....	2
Examples.....	2
Un problema algorítmico de muestra.....	2
Comenzando con el algoritmo de zumbido simple de Fizz en Swift.....	3
Capítulo 2: A * Algoritmo de búsqueda de rutas.....	6
Introducción.....	6
Examples.....	6
Ejemplo simple de A * Pathfinding: un laberinto sin obstáculos.....	6
Capítulo 3: A * Pathfinding.....	14
Examples.....	14
Introducción a A *.....	14
Resolviendo un problema de 8 rompecabezas usando el algoritmo A *.....	14
A * Recorrer un laberinto sin obstáculos.....	16
Capítulo 4: Algo: - Imprimir matriz $m \times n$ en forma de cuadrado.....	24
Introducción.....	24
Examples.....	24
Ejemplo de muestra.....	24
Escribe el código genérico.....	24
Capítulo 5: Algoritmo de Bellman-Ford.....	26
Observaciones.....	26
Examples.....	26
Algoritmo de ruta más corta de una sola fuente (dado que hay un ciclo negativo en una gráf.....	26
¿Por qué necesitamos relajar todos los bordes como máximo $(V-1)$ veces?.....	31
Detectando ciclo negativo en una gráfica.....	33
Capítulo 6: Algoritmo de Floyd-Warshall.....	36
Examples.....	36
Algoritmo de ruta más corta de todos los pares.....	36

Capítulo 7: Algoritmo de Knuth Morris Pratt (KMP)	39
Introducción.....	39
Examples.....	39
Ejemplo de KMP.....	39
Capítulo 8: Algoritmo de linea	41
Introducción.....	41
Examples.....	41
Algoritmo de dibujo lineal de Bresenham.....	41
Capítulo 9: Algoritmo de partición entero	45
Examples.....	45
Información básica del algoritmo de partición entero.....	45
Implementación del algoritmo de partición Interger en C #.....	46
Capítulo 10: Algoritmo de Prim	48
Examples.....	48
Introducción al algoritmo de Prim.....	48
Capítulo 11: Algoritmo de subarray máximo	56
Examples.....	56
Información básica del algoritmo de subarray máximo.....	56
Implementación de C #.....	57
Capítulo 12: Algoritmo de suma de ruta máxima	59
Examples.....	59
Información básica de la suma máxima de ruta.....	59
Implementación de C #.....	60
Capítulo 13: Algoritmo de ventana deslizante	62
Examples.....	62
Algoritmo de ventana deslizante Información básica.....	62
Implementación del algoritmo de ventana deslizante en C #.....	63
Capítulo 14: Algoritmo delimitado por tiempo polinómico para la cobertura mínima de vértic	65
Introducción.....	65
Parámetros.....	65
Observaciones.....	65

Examples.....	65
Algoritmo Pseudo Código.....	65
Algoritmo PMinVertexCover (gráfico G).....	65
Entrada conectada grafo G.....	65
Conjunto de cubierta de vértice mínimo de salida C.....	65
Capítulo 15: Algoritmo Numérico Catalán.....	67
Examples.....	67
Algoritmo Numérico Catalán Información Básica.....	67
Implementación de C #.....	68
Capítulo 16: Algoritmos codiciosos.....	69
Observaciones.....	69
Examples.....	69
Problema continuo de la mochila.....	69
Codificación Huffman.....	69
Problema de cambio.....	73
Problema de selección de actividad.....	75
El problema.....	75
Análisis.....	75
La solución.....	77
Capítulo 17: Algoritmos en línea.....	78
Observaciones.....	78
Teoría.....	78
Fuentes.....	80
Material básico.....	80
Otras lecturas.....	80
Código fuente.....	80
Examples.....	80
Paginación (almacenamiento en caché en línea).....	80
Prefacio.....	80
Paginacion.....	80
Enfoque sin conexión.....	81
Enfoque en línea.....	82

Algoritmos de marcado.....	83
Capítulo 18: Algoritmos multihilo.....	86
Introducción.....	86
Sintaxis.....	86
Examples.....	86
Multiplexación de matriz cuadrada multihilo.....	86
Matriz de multiplicación vector multihilo.....	86
fusionar y ordenar multiproceso.....	86
Capítulo 19: Aplicaciones de la técnica codiciosa.....	88
Observaciones.....	88
Fuentes.....	88
Examples.....	88
Ticket automático.....	88
Programación de intervalos.....	91
Minimizando la latitud.....	94
Offline Caching.....	98
Ejemplo (FIFO).....	98
Ejemplo (LFD).....	99
FIFO.....	101
LIFO.....	102
LRU.....	103
LFU.....	105
LFD.....	106
Algoritmo vs Realidad.....	107
Capítulo 20: Aplicaciones de Programación Dinámica.....	108
Introducción.....	108
Observaciones.....	108
Definiciones.....	108
Examples.....	108
Números de Fibonacci.....	108
Notas.....	111
Capítulo 21: Arboles.....	112

Observaciones.....	112
Examples.....	112
Introducción.....	112
Representación típica del árbol del árbol.....	113
Para comprobar si dos árboles binarios son iguales o no.....	114
Capítulo 22: Árboles binarios de búsqueda.....	116
Introducción.....	116
Examples.....	116
Árbol de búsqueda binario - Inserción (Python).....	116
Árbol de búsqueda binario - Eliminación (C ++).	118
El antepasado común más bajo en un BST.....	120
Árbol binario de búsqueda - Python.....	121
Capítulo 23: buscando.....	123
Examples.....	123
Búsqueda binaria.....	123
Introducción.....	123
Ejemplo de pregunta.....	123
Explicación de ejemplo.....	123
Búsqueda binaria: en números ordenados.....	124
Busqueda lineal.....	125
Rabin Karp.....	126
Análisis de búsqueda lineal (peor, promedio y mejores casos).....	127
Capítulo 24: Búsqueda de amplitud.....	130
Examples.....	130
Encontrar el camino más corto desde la fuente a otros nodos.....	130
Encontrar la ruta más corta desde la fuente en un gráfico 2D.....	137
Componentes conectados de un gráfico no dirigido utilizando BFS.....	138
Capítulo 25: Búsqueda de subcadena.....	143
Examples.....	143
Algoritmo KMP en C.....	143
Introducción al algoritmo de Rabin-Karp.....	145

Introducción al algoritmo de Knuth-Morris-Pratt (KMP).....	148
Implementación Python del algoritmo KMP.....	152
Capítulo 26: Clasificación.....	154
Parámetros.....	154
Examples.....	154
Estabilidad en la clasificación.....	154
Capítulo 27: Combinar clasificación.....	156
Examples.....	156
Fundamentos de clasificación de fusión.....	156
Implementación de Merge Sort en C & C #.....	157
Implementación de Merge Sort en Java.....	159
Fusionar la implementación de orden en Python.....	160
Implementación de Java de abajo hacia arriba.....	160
Fusionar la implementación de ordenación en Go.....	161
Capítulo 28: Complejidad de algoritmos.....	163
Observaciones.....	163
Trabajo.....	164
Lapso.....	164
Examples.....	165
Notación Big-Theta.....	165
Notación Big-Omega.....	166
Definición formal.....	166
Notas.....	166
Referencias.....	167
Comparación de las notaciones asintóticas.....	167
Campo de golf.....	168
Capítulo 29: Compruebe que dos cadenas son anagramas.....	170
Introducción.....	170
Examples.....	170
Muestra de entrada y salida.....	170
Código genérico para anagramas.....	171

Capítulo 30: Compruebe si un árbol es BST o no	173
Examples	173
Si un árbol de entrada dado sigue una propiedad del árbol de búsqueda binaria o no	173
Algoritmo para verificar si un árbol binario dado es BST	173
Capítulo 31: Editar distancia del algoritmo dinámico	175
Examples	175
Ediciones mínimas requeridas para convertir la cadena 1 a la cadena 2	175
Capítulo 32: El algoritmo de Dijkstra	178
Examples	178
Algoritmo de la ruta más corta de Dijkstra	178
Capítulo 33: El algoritmo de Kruskal	183
Observaciones	183
Examples	183
Implementación simple, más detallada	183
Implementación simple, basada en conjuntos disjuntos	183
Implementación óptima, basada en conjuntos disjuntos	184
Implementación simple y de alto nivel	185
Capítulo 34: El ancestro común más bajo de un árbol binario	186
Introducción	186
Examples	186
Encontrar el antepasado común más bajo	186
Capítulo 35: El problema más corto de la supersecuencia	187
Examples	187
Información básica sobre el problema de la supersecuencia más corta	187
Implementación del problema más corto de supersecuencia en C #	188
Capítulo 36: Exposición de matrices	190
Examples	190
Exposición de matrices para resolver problemas de ejemplo	190
Capítulo 37: Funciones hash	195
Examples	195
Introducción a las funciones hash	195

Métodos hash	195
Tabla de picadillo	195
Ejemplos	196
Campo de golf	197
Códigos hash para tipos comunes en C #	197
Booleano	197
Byte , UInt16 , Int32 , UInt32 , Single	197
SByte	197
Carbonizarse	197
Int16	198
Int64 , doble	198
UInt64 , DateTime , TimeSpan	198
Decimal	198
Objeto	198
Cuerda	198
Tipo de valor	198
Nullable <T>	199
Formación	199
Referencias	199
Capítulo 38: Grafico	200
Introducción	200
Observaciones	200
Examples	200
Clasificación topológica	200
Ejemplo de problema y su solución	201
Algoritmo de Thorup	202
Detectando un ciclo en un gráfico dirigido usando Depth First Traversal	202
Introducción a la teoría de grafos	204
Almacenando Gráficos (Matriz de Adyacencia)	209
Almacenamiento de gráficos (lista de adyacencia)	213
Capítulo 39: Gráficos de travesías	216

Examples.....	216
Profundidad de la primera búsqueda de la función transversal.....	216
Capítulo 40: Heap Sort.....	217
Examples.....	217
Heap Sort Información Básica.....	217
Implementación de C #.....	218
Capítulo 41: La subsecuencia cada vez mayor.....	219
Examples.....	219
La información básica de la subsecuencia cada vez más creciente.....	219
Implementación de C #.....	221
Capítulo 42: La subsecuencia común más larga.....	223
Examples.....	223
Explicación de la subsecuencia común más larga.....	223
Capítulo 43: Notación Big-O.....	229
Observaciones.....	229
Examples.....	230
Un bucle simple.....	230
Un bucle anidado.....	231
Un ejemplo de $O(\log n)$	232
Introducción.....	232
Enfoque ingenuo.....	232
Dicotomía.....	232
Explicación.....	233
Conclusión.....	233
$O(\log n)$ tipos de algoritmos.....	233
Capítulo 44: Orden de conteo.....	236
Examples.....	236
Información básica de orden de conteo.....	236
Implementacion Psuedocode.....	236
Implementación de C #.....	237
Capítulo 45: Ordenación rápida.....	238

Observaciones.....	238
Examples.....	238
Fundamentos de Quicksort.....	238
Implementación de C #.....	240
Implementación Haskell.....	241
Lomuto partición java implementacion.....	241
Quicksort en Python.....	241
Impresiones "[1, 1, 2, 3, 6, 8, 10]".....	242
Capítulo 46: Ordenamiento de burbuja.....	243
Parámetros.....	243
Examples.....	243
Ordenamiento de burbuja.....	243
Implementación en Javascript.....	244
Implementación en C #.....	244
Implementación en C & C ++.....	245
Implementación en Java.....	246
Implementación de Python.....	247
Capítulo 47: Primera búsqueda de profundidad.....	248
Examples.....	248
Introducción a la búsqueda en profundidad primero.....	248
Capítulo 48: Problema de mochila.....	254
Observaciones.....	254
Examples.....	254
Fundamentos del problema de la mochila.....	254
Solución implementada en C #.....	255
Capítulo 49: Programación dinámica.....	256
Introducción.....	256
Observaciones.....	256
Examples.....	256
Problema de mochila.....	256
Ejemplo de C ++:.....	257
Python (2.7.11) Ejemplo:.....	257

Algoritmo de programación de trabajo ponderado.....	258
Editar distancia.....	262
La subsecuencia común más larga.....	263
Número de Fibonacci.....	264
La subcadena común más larga.....	265
Capítulo 50: Pseudocódigo.....	266
Observaciones.....	266
Examples.....	266
Afectaciones variables.....	266
Mecanografiado.....	266
Sin tipo.....	266
Funciones.....	266
Capítulo 51: Radix Sort.....	268
Examples.....	268
Radix Sort Información Básica.....	268
Capítulo 52: Resolución de ecuaciones.....	270
Examples.....	270
Ecuación lineal.....	270
Ecuación no lineal.....	272
Capítulo 53: Selección de selección.....	276
Examples.....	276
Selección Ordenar Información Básica.....	276
Implementación del ordenamiento de selección en C #.....	278
Implementación de Elixir.....	278
Capítulo 54: Shell Sort.....	280
Examples.....	280
Información básica de Shell Sort.....	280
Implementación de C #.....	282
Capítulo 55: Time Warping dinámico.....	283
Examples.....	283
Introducción a la distorsión de tiempo dinámico.....	283
Capítulo 56: Tipo de casillero.....	288

Examples.....	288
Pigeonhole Sort Información Básica.....	288
Implementación de C #.....	289
Capítulo 57: Tipo de ciclo.....	291
Examples.....	291
Ciclo de Información Básica.....	291
Implementación de pseudocódigo.....	291
Implementación de C #.....	292
Capítulo 58: Tipo de cubo.....	293
Examples.....	293
Bucket Sort Información Básica.....	293
Implementación de C #.....	293
Capítulo 59: Tipo de inserción.....	295
Observaciones.....	295
Media de intercambio.....	295
Comparación de promedios.....	295
Examples.....	296
Fundamentos del algoritmo.....	296
Implementación de C #.....	298
Implementación Haskell.....	298
Capítulo 60: Tipo de panqueque.....	299
Examples.....	299
Información básica sobre el tipo de panqueque.....	299
Implementación de C #.....	300
Capítulo 61: Tipo impar-par.....	302
Examples.....	302
Información básica de orden impar.....	302
Capítulo 62: Transformada rápida de Fourier.....	305
Introducción.....	305
Examples.....	305
Radix 2 FFT.....	305
Radix 2 Inverse FFT.....	309

Capítulo 63: Travesías de árboles binarios	312
Introducción.....	312
Examples.....	312
Pre-order, Inorder y Post Order transversal de un árbol binario.....	312
Desplazamiento de la orden de nivel - Implementación.....	312
Capítulo 64: Triángulo de Pascal	315
Examples.....	315
Pascal's Triagle Información Básica.....	315
Implementación del Triángulo de Pascal en C #.....	316
Triángulo de Pascal en C.....	316
Capítulo 65: Vendedor ambulante	318
Observaciones.....	318
Examples.....	318
Algoritmo de fuerza bruta.....	318
Algoritmo de programación dinámico.....	319
Creditos	321

Acerca de

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.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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

Capítulo 1: Empezando con el algoritmo

Observaciones

Introducción a los algoritmos

Los algoritmos son omnipresentes en Ciencias de la Computación e Ingeniería de Software. La selección de algoritmos y estructuras de datos adecuados mejora la eficiencia de nuestro programa en costos y tiempo.

¿Qué es un algoritmo? Informalmente, un algoritmo es un procedimiento para realizar una tarea específica. [Skiena: 2008: ADM: 1410219] Específicamente, un algoritmo es un procedimiento computacional *bien definido*, que toma algún valor (o conjunto de valores) como **entrada** y produce algún valor, o un conjunto de valores, como **salida**. Un algoritmo es, por lo tanto, una secuencia de pasos computacionales que transforman la entrada en la salida. Cormen et. Alabama. no comenta explícitamente que un algoritmo no necesariamente requiere una entrada. [Cormen: 2001: IA: 580470]

Formalmente, un algoritmo debe satisfacer cinco características: [Knuth: 1997: ACP: 260999]

1. *Finitud* Un algoritmo siempre debe terminar después de un número finito de pasos.
2. *La definición* Cada paso de un algoritmo debe ser definido con precisión; Las acciones a realizar deben ser especificadas rigurosamente. Es esta cualidad a la que [Cormen: 2001: IA: 580470] se refiere con el término "bien definido".
3. *Entrada* Un algoritmo tiene cero o más *entradas*. Estas son cantidades que se le dan al algoritmo inicialmente antes de que comience o dinámicamente mientras se ejecuta.
4. *Salida*. Un algoritmo tiene una o más *salidas*. Estas son cantidades que tienen una relación específica con las entradas. Esperamos que un algoritmo produzca la misma salida cuando se le dé la misma entrada una y otra vez.
5. *Efectividad* También se espera que un algoritmo sea *efectivo*. Sus operaciones deben ser lo suficientemente básicas como para que se puedan hacer exactamente en principio y en un tiempo finito por alguien que use lápiz y papel.

Un procedimiento que carece de finitud pero que satisface todas las demás características de un algoritmo puede denominarse *método computacional*. [Knuth: 1997: ACP: 260999]

Examples

Un problema algorítmico de muestra.

Un problema algorítmico se especifica al describir el conjunto completo de *instancias en las que* debe trabajar y su salida después de ejecutarse en una de estas instancias. Esta distinción, entre un problema y una instancia de un problema, es fundamental. El *problema* algorítmico conocido como *clasificación* se define de la siguiente manera: [Skiena: 2008: ADM: 1410219]

- Problema: la clasificación
- Entrada: Una secuencia de n teclas, a_1, a_2, \dots, a_n .
- Salida: la reordenación de la secuencia de entrada tal que $a'_1 \leq a'_2 \leq \dots \leq a'_{n-1} \leq a'_n$

Una *instancia* de clasificación podría ser una matriz de cadenas, como `{ Haskell, Emacs }` o una secuencia de números como `{ 154, 245, 1337 }`.

Comenzando con el algoritmo de zumbido simple de Fizz en Swift

Para aquellos de ustedes que son nuevos en la programación en Swift y aquellos que vienen de diferentes bases de programación, como Python o Java, este artículo debería ser muy útil. En este post, discutiremos una solución simple para implementar algoritmos swift.

Fizz Buzz

Es posible que haya visto a Fizz Buzz escrito como Fizz Buzz, FizzBuzz o Fizz-Buzz; Todos se refieren a lo mismo. Esa "cosa" es el tema principal de discusión hoy. En primer lugar, ¿qué es FizzBuzz?

Esta es una pregunta común que surge en las entrevistas de trabajo.

Imagina una serie de un número del 1 al 10.

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

Fizz y Buzz se refieren a cualquier número que sea múltiplo de 3 y 5 respectivamente. En otras palabras, si un número es divisible por 3, se sustituye por fizz; si un número es divisible por 5, se sustituye por un zumbido. Si un número es simultáneamente un múltiplo de 3 y 5, el número se reemplaza con "zumbido de fizz". En esencia, emula el famoso juego infantil "fizz buzz".

Para solucionar este problema, abra Xcode para crear un nuevo campo de juego e inicie una matriz como la siguiente:

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

Para encontrar todo el fizz y el zumbido, debemos recorrer la matriz y comprobar qué números son fizz y cuáles son el zumbido. Para hacer esto, cree un bucle for para iterar a través de la matriz que hemos inicializado:

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

Después de esto, simplemente podemos usar la condición "si no" y el operador del módulo en swift, es decir `-%` para localizar el fizz y el zumbido

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

¡Genial! Puedes ir a la consola de depuración en el área de juegos de Xcode para ver el resultado. Encontrarás que los "fizzes" se han resuelto en tu matriz.

Para la parte Buzz, usaremos la misma técnica. Inténtelo antes de desplazarse por el artículo: puede comparar sus resultados con este artículo una vez que haya terminado de hacer esto.

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

Compruebe la salida!

Es bastante sencillo: dividiste el número entre 3, efervescencia y dividiste el número entre 5, zumbido. Ahora, aumenta los números en la matriz

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

Aumentamos el rango de números de 1-10 a 1-15 para demostrar el concepto de "zumbido de fizz". Dado que 15 es un múltiplo de 3 y 5, el número debe reemplazarse con "zumbido de chispa". Pruébalo tú mismo y comprueba la respuesta!

Aquí está la solución:

```
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)
  }
}
```

Espera ... ¡aún no ha terminado! Todo el propósito del algoritmo es personalizar el tiempo de ejecución correctamente. Imagínese si el rango aumenta de 1-15 a 1-100. El compilador verificará cada número para determinar si es divisible entre 3 o 5. Luego volverá a revisar los números para verificar si los números son divisibles entre 3 y 5. El código esencialmente tendría que pasar por

cada número en la matriz dos veces - tendría que ejecutar los números por 3 primero y luego por 5. Para acelerar el proceso, simplemente podemos decirle a nuestro código que divida los números por 15 directamente.

Aquí está el código final:

```
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)
  }
}
```

Tan simple como eso, puede usar cualquier idioma de su elección y comenzar

Disfrutar de la codificación

Lea [Empezando con el algoritmo en línea](https://riptutorial.com/es/algorithm/topic/757/empezando-con-el-algoritmo):

<https://riptutorial.com/es/algorithm/topic/757/empezando-con-el-algoritmo>

Capítulo 2: A * Algoritmo de búsqueda de rutas

Introducción

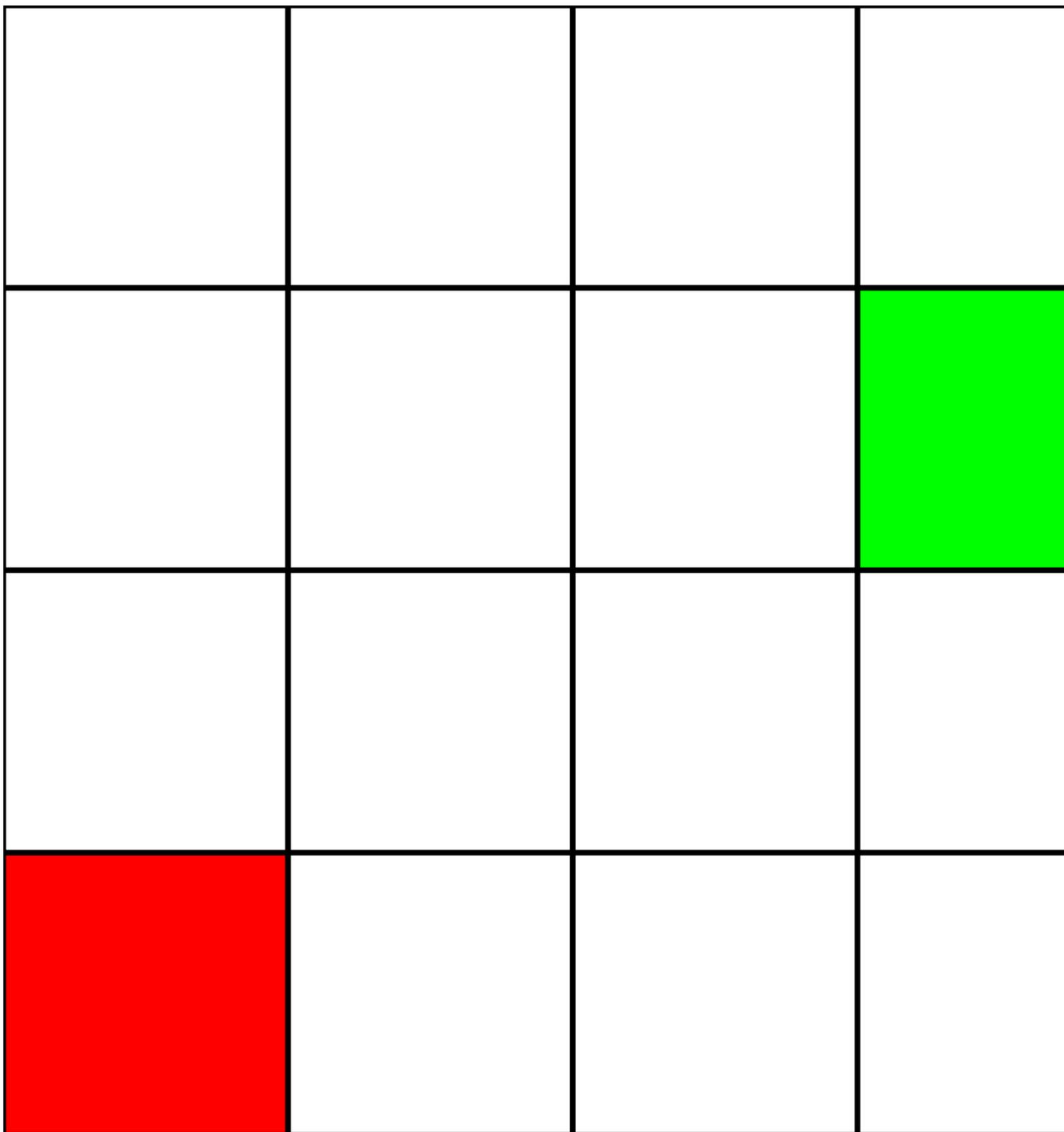
Este tema se centrará en el algoritmo de búsqueda de rutas A *, cómo se usa y por qué funciona.

Nota para futuros colaboradores: He agregado un ejemplo para A * Pathfinding sin ningún obstáculo, en una cuadrícula de 4x4. Todavía se necesita un ejemplo con obstáculos.

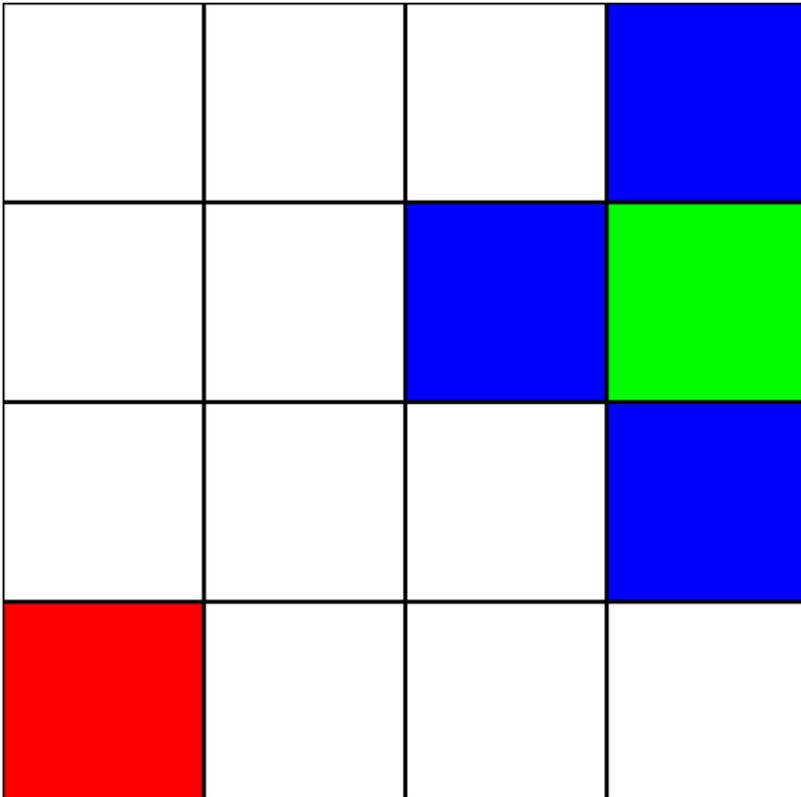
Examples

Ejemplo simple de A * Pathfinding: un laberinto sin obstáculos

Digamos que tenemos la siguiente cuadrícula de 4 por 4:



Supongamos que esto es un *laberinto* . Sin embargo, no hay paredes / obstáculos. Solo tenemos un punto de inicio (el cuadrado verde) y un punto de finalización (el cuadrado rojo). Supongamos también que para pasar de verde a rojo, no podemos movernos en diagonal. Entonces, comenzando desde el cuadrado verde, veamos a qué cuadrados podemos movernos y resáltalos en azul:



Con el fin de elegir a qué cuadrado pasar, debemos tener en cuenta 2 heurísticas:

1. El valor "g": esta es la distancia a la que se encuentra este nodo del cuadrado verde.
2. El valor "h": esta es la distancia a la que se encuentra este nodo del cuadrado rojo.
3. El valor "f" - Esta es la suma del valor "g" y el valor "h". Este es el número final que nos dice a qué nodo movernos.

Para calcular estas heurísticas, esta es la fórmula que usaremos: $distance = abs(from.x - to.x) + abs(from.y - to.y)$

Esto se conoce como la fórmula "[Manhattan Distance](#)".

Calculemos el valor "g" para el cuadrado azul inmediatamente a la izquierda del cuadrado verde:

$$abs(3 - 2) + abs(2 - 2) = 1$$

¡Genial! Tenemos el valor: 1. Ahora, intentemos calcular el valor "h": $abs(2 - 0) + abs(2 - 0) = 4$

Perfecto. Ahora, obtengamos el valor "f": $1 + 4 = 5$

Entonces, el valor final para este nodo es "5".

Hagamos lo mismo para todos los otros cuadrados azules. El número grande en el centro de cada cuadrado es el valor "f", mientras que el número en la parte superior izquierda es el valor "g", y el número en la parte superior derecha es el valor "h":

			1 6 7
		1 4 5	
			1 4 5

Hemos calculado los valores de g, h y f para todos los nodos azules. Ahora, ¿cuál recogemos?

El que tenga el valor f más bajo.

Sin embargo, en este caso, tenemos 2 nodos con el mismo valor de f, 5. ¿Cómo seleccionamos entre ellos?

Simplemente, elija uno al azar o establezca una prioridad. Generalmente prefiero tener una prioridad como esta: "Derecha> Arriba> Abajo> Izquierda"

Uno de los nodos con el valor f de 5 nos lleva en la dirección "Abajo", y el otro nos lleva a "Izquierda". Como Down tiene una prioridad más alta que Left, elegimos el cuadrado que nos lleva "Down".

Ahora marca los nodos para los que calculamos las heurísticas, pero no nos movimos a, como naranja, y el nodo que elegimos como cian:

			1 6 7
		1 4 5	
			1 4 5

Bien, ahora calculemos las mismas heurísticas para los nodos alrededor del nodo cian:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
			2 3 5

Nuevamente, elegimos el nodo que baja desde el nodo cian, ya que todas las opciones tienen el mismo valor f:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
			2 3 5

Calculemos las heurísticas para el único vecino que tiene el nodo cian:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
		3 2 5	2 3 5

Muy bien, ya que seguiremos el mismo patrón que hemos estado siguiendo:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
		3 2 5	2 3 5

Una vez más, calculemos las heurísticas para el vecino del nodo:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
	4 1 5	3 2 5	2 3 5

Vayamos allí:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
	4 1 5	3 2 5	2 3 5

Finalmente, podemos ver que tenemos una *casilla ganadora* a nuestro lado, por lo que nos movemos allí y hemos terminado.

Lea A * Algoritmo de búsqueda de rutas en línea: <https://riptutorial.com/es/algorithm/topic/8787/a--algoritmo-de-busqueda-de-rutas>

Capítulo 3: A * Pathfinding

Examples

Introducción a A *

Un * (una estrella) es un algoritmo de búsqueda que se utiliza para encontrar la ruta de un nodo a otro. Por lo tanto, se puede comparar con la [búsqueda en primer lugar](#) o [el algoritmo de Dijkstra](#), o la [búsqueda en profundidad](#) o la primera búsqueda. Un algoritmo * se usa ampliamente en la búsqueda de gráficos para ser mejor en eficiencia y precisión, donde el procesamiento previo de gráficos no es una opción.

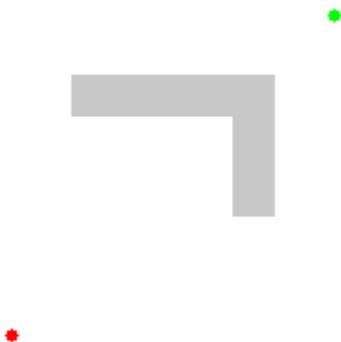
A * es una especialización de Best First Search, en la cual la función de evaluación f se define de una manera particular.

$f(n) = g(n) + h(n)$ es el costo mínimo desde el nodo inicial hasta los objetivos condicionados para ir al nodo de pensamiento n .

$g(n)$ es el costo mínimo desde el nodo inicial hasta n .

$h(n)$ es el costo mínimo de n al objetivo más cercano a n

A * es un algoritmo de búsqueda informado y siempre garantiza encontrar la ruta más pequeña (ruta con un costo mínimo) en el menor tiempo posible (si se usa [heurística admisible](#)). Así que es tanto *completa* como *óptima*. La siguiente animación muestra A * search-



Resolviendo un problema de 8 rompecabezas usando el algoritmo A *

Definición del problema :

Un rompecabezas 8 es un juego simple que consiste en una cuadrícula de 3 x 3 (que contiene 9 cuadrados). Uno de los cuadrados está vacío. El objetivo es moverse a cuadrados alrededor en diferentes posiciones y tener los números mostrados en el "estado objetivo".

1	2	3
8		4
7	6	5

Dado un estado inicial de juego de 8 rompecabezas y un estado final a alcanzar, encuentre el camino más rentable para alcanzar el estado final desde el estado inicial.

Estado inicial :

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

Estado final :

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

Heurística a asumir :

Consideremos la distancia de Manhattan entre el estado actual y el estado final como la heurística para esta declaración de problema.

```
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
```

Función de costo total :

Entonces, la función de costo total $f(n)$ viene dada por,

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

Solución al problema de ejemplo :

Primero encontramos el valor heurístico requerido para alcanzar el estado final desde el estado inicial. La función de costo, $g(n) = 0$, ya que estamos en el estado inicial

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

Se obtiene el valor anterior, ya que 1 en el estado actual está a 1 distancia horizontal de distancia que 1 en el estado final. Lo mismo ocurre con 2, 5, 6. _ está a 2 distancias horizontales y 2 distancias verticales. Entonces, el valor total para $h(n)$ es $1 + 1 + 1 + 1 + 2 + 2 = 8$. La función de costo total $f(n)$ es igual a $8 + 0 = 8$.

Ahora, se encuentran los posibles estados a los que se puede llegar desde el estado inicial y sucede que podemos mover _ hacia la derecha o hacia abajo.

Así que los estados obtenidos después de mover esos movimientos son:

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

Nuevamente, la función de costo total se calcula para estos estados utilizando el método descrito anteriormente y resulta ser 6 y 7 respectivamente. Elegimos el estado con el costo mínimo que es el estado (1). Los siguientes movimientos posibles pueden ser Izquierda, Derecha o Abajo. No nos moveremos a la izquierda como estábamos anteriormente en ese estado. Por lo tanto, podemos movernos hacia la derecha o hacia abajo.

Nuevamente encontramos los estados obtenidos de (1).

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

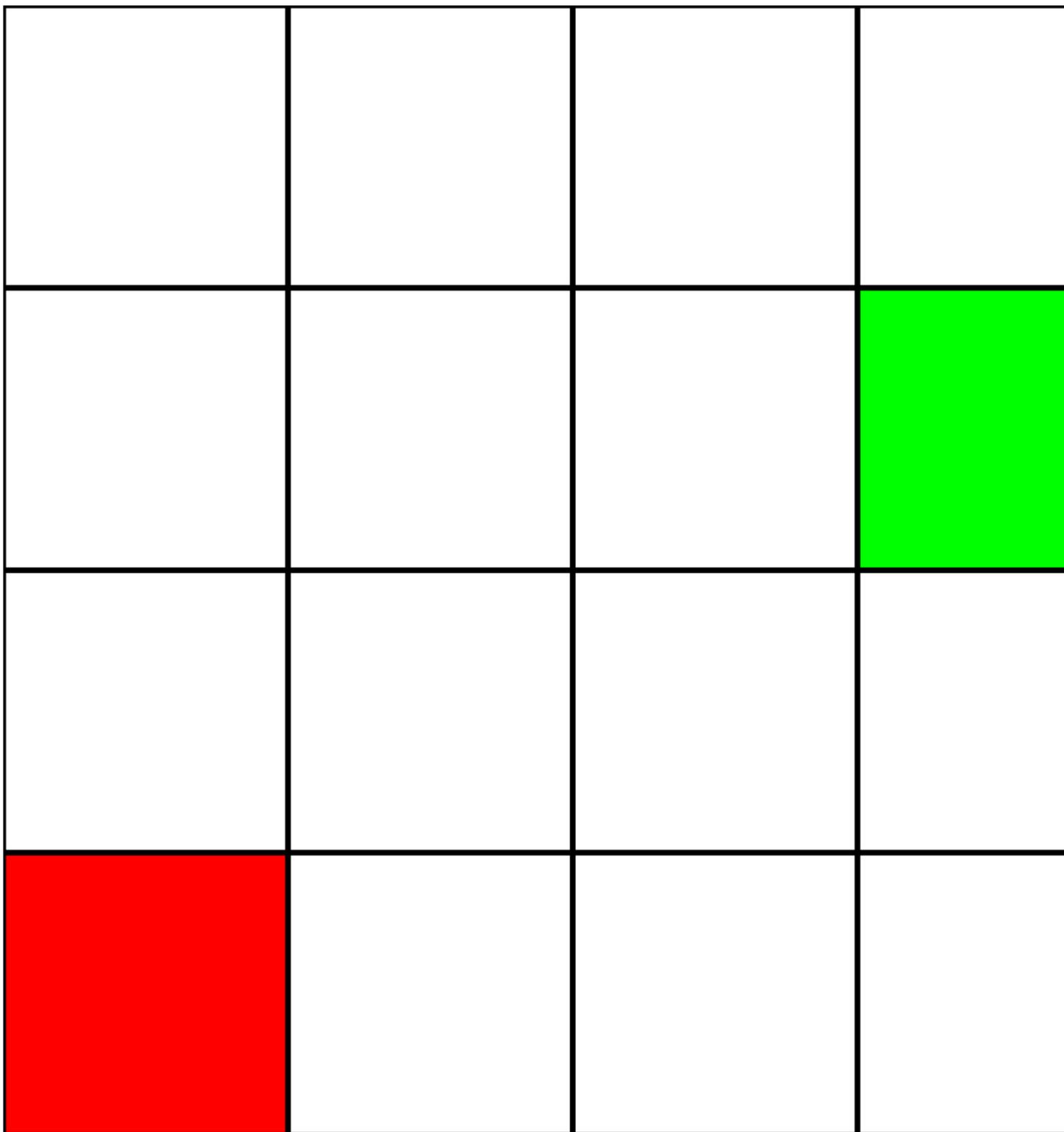
(3) lleva a una función de costo igual a 6 y (4) conduce a 4. Además, consideraremos (2) el resultado anterior que tiene una función de costo igual a 7. Elegir el mínimo de ellos conduce a (4). Los siguientes movimientos posibles pueden ser Izquierda o Derecha o Abajo. Obtenemos estados:

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)

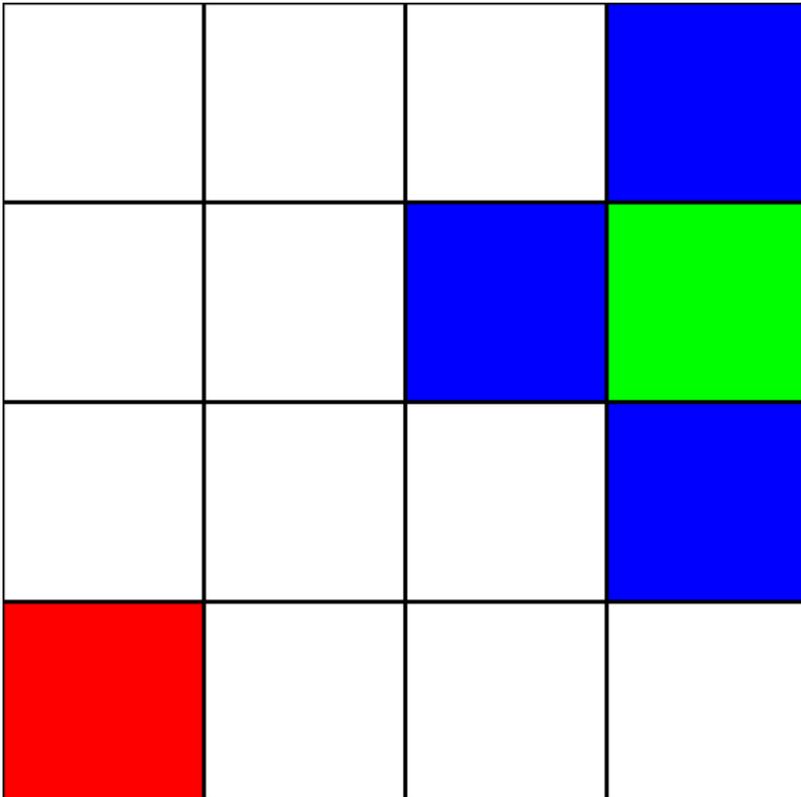
Obtenemos costos iguales a 5, 2 y 4 para (5), (6) y (7) respectivamente. Además, tenemos estados anteriores (3) y (2) con 6 y 7 respectivamente. Elegimos el estado de costo mínimo que es (6). Los siguientes movimientos posibles son Arriba, y Abajo y claramente Abajo nos llevarán al estado final que lleva a un valor de función heurística igual a 0.

A * Recorrer un laberinto sin obstáculos.

Digamos que tenemos la siguiente cuadrícula de 4 por 4:



Supongamos que esto es un *laberinto* . Sin embargo, no hay paredes / obstáculos. Solo tenemos un punto de inicio (el cuadrado verde) y un punto de finalización (el cuadrado rojo). Supongamos también que para pasar de verde a rojo, no podemos movernos en diagonal. Entonces, comenzando desde el cuadrado verde, veamos a qué cuadrados podemos movernos y resáltalos en azul:



Con el fin de elegir a qué cuadrado pasar, debemos tener en cuenta 2 heurísticas:

1. El valor "g": esta es la distancia a la que se encuentra este nodo del cuadrado verde.
2. El valor "h": esta es la distancia a la que se encuentra este nodo del cuadrado rojo.
3. El valor "f" - Esta es la suma del valor "g" y el valor "h". Este es el número final que nos dice a qué nodo movernos.

Para calcular estas heurísticas, esta es la fórmula que usaremos: $distance = abs(from.x - to.x) + abs(from.y - to.y)$

Esto se conoce como la fórmula "[Manhattan Distance](#)".

Calculemos el valor "g" para el cuadrado azul inmediatamente a la izquierda del cuadrado verde:

$$abs(3 - 2) + abs(2 - 2) = 1$$

¡Genial! Tenemos el valor: 1. Ahora, intentemos calcular el valor "h": $abs(2 - 0) + abs(2 - 0) = 4$

Perfecto. Ahora, obtengamos el valor "f": $1 + 4 = 5$

Entonces, el valor final para este nodo es "5".

Hagamos lo mismo para todos los otros cuadrados azules. El número grande en el centro de cada cuadrado es el valor "f", mientras que el número en la parte superior izquierda es el valor "g", y el número en la parte superior derecha es el valor "h":

			1 6 7
		1 4 5	
			1 4 5

Hemos calculado los valores de g, h y f para todos los nodos azules. Ahora, ¿cuál recogemos?

El que tenga el valor f más bajo.

Sin embargo, en este caso, tenemos 2 nodos con el mismo valor de f, 5. ¿Cómo seleccionamos entre ellos?

Simplemente, elija uno al azar o establezca una prioridad. Generalmente prefiero tener una prioridad como esta: "Derecha> Arriba> Abajo> Izquierda"

Uno de los nodos con el valor f de 5 nos lleva en la dirección "Abajo", y el otro nos lleva a "Izquierda". Como Down tiene una prioridad más alta que Left, elegimos el cuadrado que nos lleva "Down".

Ahora marca los nodos para los que calculamos las heurísticas, pero no nos movimos a, como naranja, y el nodo que elegimos como cian:

			1 6 7
		1 4 5	
			1 4 5

Bien, ahora calculemos las mismas heurísticas para los nodos alrededor del nodo cian:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
			2 3 5

Nuevamente, elegimos el nodo que baja desde el nodo cian, ya que todas las opciones tienen el mismo valor f:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
			2 3 5

Calculemos las heurísticas para el único vecino que tiene el nodo cian:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
		3 2 5	2 3 5

Muy bien, ya que seguiremos el mismo patrón que hemos estado siguiendo:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
		3 2 5	2 3 5

Una vez más, calculemos las heurísticas para el vecino del nodo:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
	4 1 5	3 2 5	2 3 5

Vayamos allí:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
	4 1 5	3 2 5	2 3 5

Finalmente, podemos ver que tenemos una *casilla ganadora* a nuestro lado, por lo que nos movemos allí y hemos terminado.

Lea A * Pathfinding en línea: <https://riptutorial.com/es/algorithm/topic/7439/a---pathfinding>

Capítulo 4: Algo: - Imprimir matriz am * n en forma de cuadrado

Introducción

Compruebe la entrada de muestra y la salida a continuación.

Examples

Ejemplo de muestra

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
```

Escribe el código genérico

```
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+''+1);
    }
    for(var x = k; x > i; x--) {
        console.log(x+''+1);
    }
}
}
squarePrint(6,4);
```

Lea Algo: - Imprimir matriz $m \times n$ en forma de cuadrado en línea:

<https://riptutorial.com/es/algorithm/topic/9968/algo---imprimir-matriz-am---n-en-forma-de-cuadrado>

Capítulo 5: Algoritmo de Bellman-Ford

Observaciones

Dado un gráfico G dirigido, a menudo queremos encontrar la distancia más corta desde un nodo A dado al resto de los nodos en el gráfico. El algoritmo de **Dijkstra** es el algoritmo más famoso para encontrar la ruta más corta, sin embargo, funciona solo si los pesos de borde del gráfico dado no son negativos. Sin embargo, **Bellman-Ford** pretende encontrar la ruta más corta desde un nodo determinado (si existe), incluso si algunas de las ponderaciones son negativas. Tenga en cuenta que la distancia más corta puede no existir si hay un ciclo negativo en el gráfico (en cuyo caso podemos recorrer el ciclo y generar una distancia total infinitamente pequeña). **Bellman-Ford**, además, nos permite determinar la presencia de dicho ciclo.

La complejidad total del algoritmo es $O(V * E)$, donde V - es el número de vértices y E número de bordes

Examples

Algoritmo de ruta más corta de una sola fuente (dado que hay un ciclo negativo en una gráfica)

Antes de leer este ejemplo, se requiere tener una breve idea sobre la relajación del borde. Puedes aprenderlo desde [aquí](#).

El algoritmo de **Bellman-Ford** calcula las rutas más cortas desde un solo vértice fuente a todos los otros vértices en un dígrafo ponderado. Aunque es más lento que el algoritmo de **Dijkstra**, funciona en los casos en que el peso del borde es negativo y también encuentra un ciclo de peso negativo en la gráfica. El problema con el algoritmo de Dijkstra es que, si hay un ciclo negativo, sigues repitiendo el ciclo una y otra vez y continúas reduciendo la distancia entre dos vértices.

La idea de este algoritmo es recorrer todos los bordes de este gráfico uno por uno en un orden aleatorio. Puede ser cualquier orden aleatorio. Pero debe asegurarse de que si uv (uno de los vértices de u y v son dos vértices en una gráfica) es uno de sus pedidos, entonces debe haber un margen de u a v . Por lo general, se toma directamente del orden de la entrada dada. De nuevo, cualquier orden aleatorio funcionará.

Después de seleccionar el orden, *relajaremos* los bordes según la fórmula de relajación. Para un borde dado uv que va de u a v , la fórmula de relajación es:

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

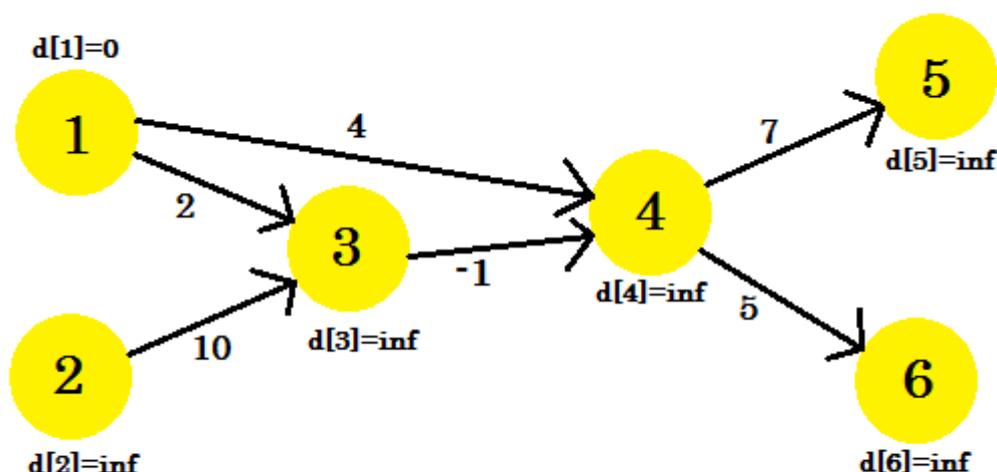
Es decir, si la distancia de la **fuente** a cualquier vértice u + el peso del **borde** uv es menor que la distancia de la **fuente** a otro vértice v , actualizamos la distancia de la **fuente** a v . Necesitamos *relajar* los bordes como máximo $(V-1)$ donde V es el número de bordes en el gráfico. ¿Por qué $(V-1)$?

1) preguntas? Lo explicaremos en otro ejemplo. También vamos a hacer un seguimiento del vértice padre de cualquier vértice, es decir, cuando relajamos un borde, estableceremos:

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

Significa que hemos encontrado otro camino más corto para alcanzar **v** a través de **u**. Necesitaremos esto más adelante para imprimir la ruta más corta desde la **fuentes** hasta el vértice destinado.

Veamos un ejemplo. Tenemos un gráfico:



Hemos seleccionado **1** como el vértice de **origen**. Queremos encontrar la ruta más corta desde la **fuentes** a todos los demás vértices.

Al principio, $d[1] = 0$ porque es la fuente. Y el descanso es *infinito*, porque aún no sabemos su distancia.

Vamos a relajar los bordes en esta secuencia:

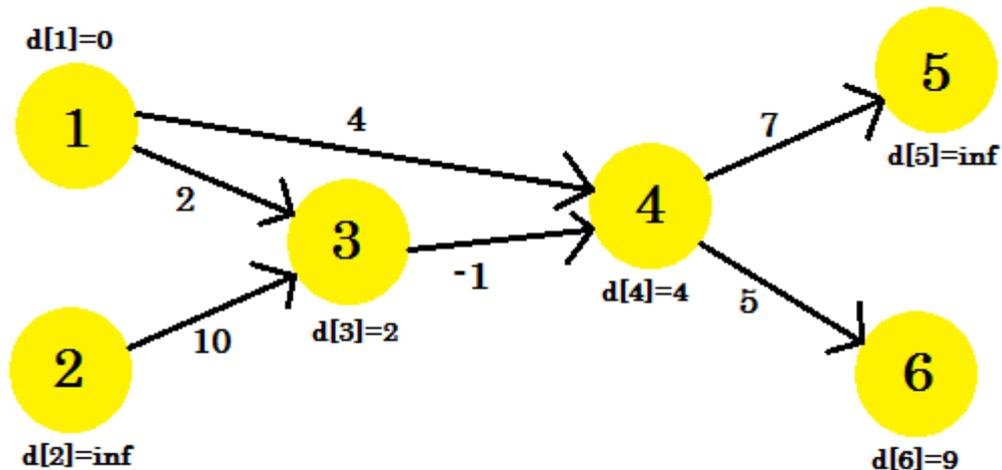
Serial	1	2	3	4	5	6
Edge	4->5	3->4	1->3	1->4	4->6	2->3

Puedes tomar cualquier secuencia que quieras. Si *relajamos* los bordes una vez, ¿qué obtenemos? Obtenemos la distancia desde la **fuentes** a todos los otros vértices de la ruta que utiliza a lo sumo 1 borde. Ahora relajemos los bordes y actualicemos los valores de $d[]$. Obtenemos:

- $d[4] + \text{costo}[4][5] = \text{infinito} + 7 = \text{infinito}$. No podemos actualizar este.
- $d[2] + \text{costo}[3][4] = \text{infinito}$. No podemos actualizar este.
- $d[1] + \text{costo}[1][2] = 0 + 2 = 2 < d[2]$. Entonces $d[2] = 2$. También **padre[2] = 1**.

4. $d[1] + \text{costo}[1][4] = 4$. Entonces $d[4] = 4 < d[4]$. **padre[4] = 1** .
5. $d[4] + \text{costo}[4][6] = 9$. $d[6] = 9 < d[6]$. **padre[6] = 4** .
6. $d[2] + \text{costo}[2][2] = \text{infinito}$. No podemos actualizar este.

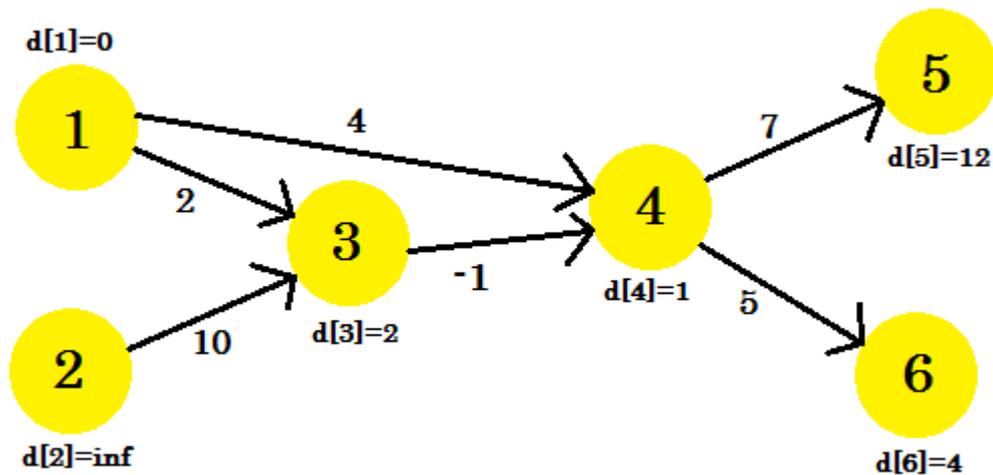
No pudimos actualizar algunos vértices, porque la condición $d[u] + \text{costo}[u][v] < d[v]$ no coincidió. Como hemos dicho anteriormente, encontramos las rutas desde la **fuentes** a otros nodos utilizando un máximo de 1 borde.



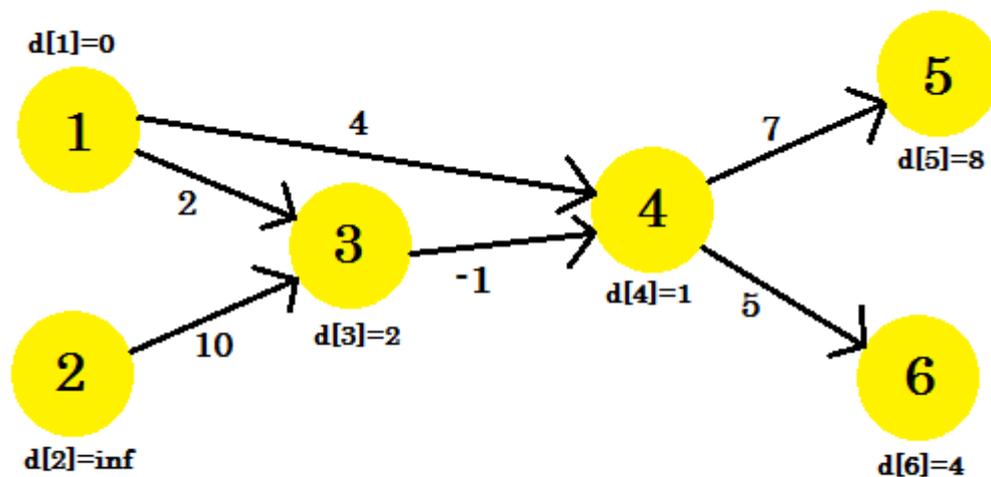
Nuestra segunda iteración nos proporcionará la ruta utilizando 2 nodos. Obtenemos:

1. $d[4] + \text{costo}[4][5] = 12 < d[5]$. $d[5] = 12$. **padre[5] = 4** .
2. $d[3] + \text{costo}[3][4] = 1 < d[4]$. $d[4] = 1$. **padre[4] = 3** .
3. $d[3]$ permanece sin cambios.
4. $d[4]$ permanece sin cambios.
5. $d[4] + \text{costo}[4][6] = 6 < d[6]$. $d[6] = 6$. **padre[6] = 4** .
6. $d[3]$ permanece sin cambios.

Nuestro gráfico se verá como:



Nuestra tercera iteración solo actualizará el **vértice 5** , donde **d [5]** será **8** . Nuestro gráfico se



verá como:

Después de esto, sin importar cuántas iteraciones hagamos, tendremos las mismas distancias. Así que mantendremos una bandera que verifique si alguna actualización tiene lugar o no. Si no es así, simplemente romperemos el bucle. Nuestro pseudo-código será:

```

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

```

Para realizar un seguimiento del ciclo negativo, podemos modificar nuestro código mediante el procedimiento descrito [aquí](#) . Nuestro pseudo-código completado será:

```

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

```

Ruta de impresión:

Para imprimir la ruta más corta a un vértice, volveremos a iterar a su padre hasta que encontremos **NULL** e imprimamos los vértices. El pseudocódigo será:

```

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

```

Complejidad:

Como necesitamos relajar los tiempos máximos de bordes (**V-1**) , la complejidad del tiempo de este algoritmo será igual a **O (V * E)** donde **E** denota el número de bordes, si usamos la `adjacency`

list para representar el gráfico. Sin embargo, si se usa una adjacency matrix para representar el gráfico, la complejidad del tiempo será $O(V^3)$. La razón es que podemos iterar a través de todos los bordes en el tiempo $O(E)$ cuando se usa la adjacency list, pero toma el tiempo $O(V^2)$ cuando se usa la adjacency matrix.

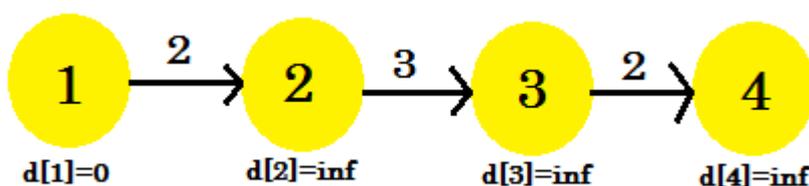
¿Por qué necesitamos relajar todos los bordes como máximo (V-1) veces?

Para comprender este ejemplo, se recomienda tener una breve idea del algoritmo de ruta más corta de fuente única de Bellman-Ford que se puede encontrar [aquí](#).

En el algoritmo de Bellman-Ford, para descubrir la ruta más corta, necesitamos relajar todos los bordes del gráfico. Este proceso se repite como máximo (V-1) veces, donde V es el número de vértices en el gráfico.

El número de iteraciones necesarias para encontrar la ruta más corta desde la fuente a todos los demás vértices depende del orden que seleccionamos para relajar los bordes.

Echemos un vistazo a un ejemplo:



Aquí, el vértice de origen es 1. Descubriremos la distancia más corta entre el origen y todos los demás vértices. Podemos ver claramente que, para alcanzar el vértice 4, en el peor de los casos, tomará los bordes (V-1). Ahora, dependiendo del orden en que se descubren los bordes, puede tomar (V-1) veces descubrir el vértice 4. ¿No lo conseguiste? Usemos el algoritmo de Bellman-Ford para encontrar el camino más corto aquí:

Vamos a utilizar esta secuencia:

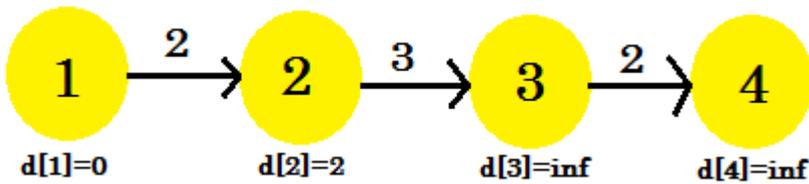
```

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

Para nuestra primera iteración:

1. $d[3] + \text{costo}[3][4] = \text{infinito}$. No cambiará nada.
2. $d[2] + \text{costo}[2][3] = \text{infinito}$. No cambiará nada.
3. $d[1] + \text{costo}[1][2] = 2 < d[2]$. $d[2] = 2$. padre [2] = 1.

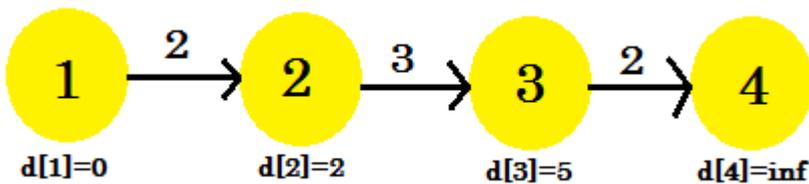
Podemos ver que nuestro proceso de relajación solo cambió $d[2]$. Nuestro gráfico se verá como:



Segunda iteración:

1. $d[3] + \text{costo}[3][4] = \text{infinito}$. No cambiará nada.
2. $d[2] + \text{costo}[2][3] = 5 < d[3]$. $d[3] = 5$. **padre[3] = 2**.
3. No será cambiado.

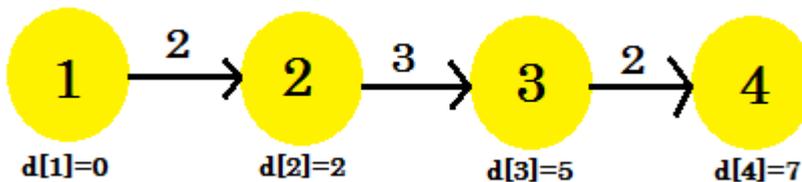
Esta vez el proceso de *relajación* cambió $d[3]$. Nuestro gráfico se verá como:



Tercera iteración:

1. $d[3] + \text{costo}[3][4] = 7 < d[4]$. $d[4] = 7$. **padre[4] = 3** .
2. No será cambiado.
3. No será cambiado.

Nuestra tercera iteración finalmente descubrió el camino más corto a 4 de 1 . Nuestro gráfico se



verá como:

Entonces, se necesitaron **3** iteraciones para descubrir el camino más corto. Después de este, no importa cuántas veces *relajemos* los bordes, los valores en $d[]$ seguirán siendo los mismos.

Ahora, si consideramos otra secuencia:

Serial	1	2	3
Edge	1->2	2->3	3->4

Nosotros obtendríamos

1. $d[1] + \text{costo}[1][2] = 2 < d[2]$. $d[2] = 2$.
2. $d[2] + \text{costo}[2][3] = 5 < d[3]$. $d[3] = 5$.

$$3. d[3] + \text{costo}[3][4] = 7 < d[4] . d[4] = 5 .$$

Nuestra primera iteración ha encontrado la ruta más corta desde la **fuentes** a todos los demás nodos. Otra secuencia 1-> 2 , 3-> 4 , 2-> 3 es posible, lo que nos dará el camino más corto después de 2 iteraciones. Podemos llegar a la decisión de que, sin importar cómo ordenemos la secuencia, no se necesitarán más de 3 iteraciones para encontrar la ruta más corta desde la **fuentes** en este ejemplo.

Podemos concluir que, en el mejor de los casos, tomará 1 iteración para descubrir la ruta más corta desde la **fuentes** . En el peor de los casos, tomará iteraciones $(V-1)$, por lo que repetimos el proceso de *relajación* $(V-1)$.

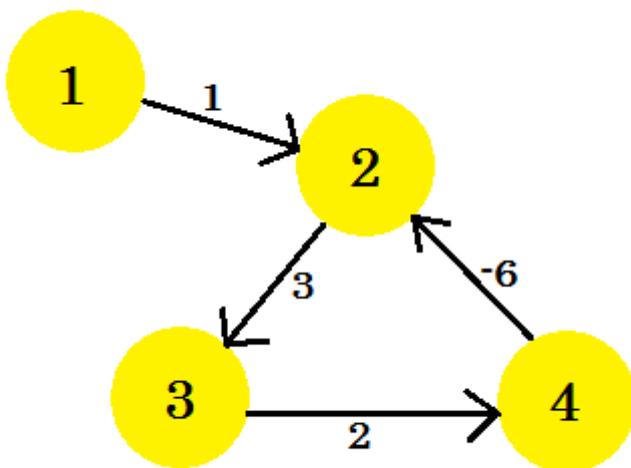
Detectando ciclo negativo en una gráfica

Para comprender este ejemplo, se recomienda tener una breve idea sobre el algoritmo de Bellman-Ford que se puede encontrar [aquí](#).

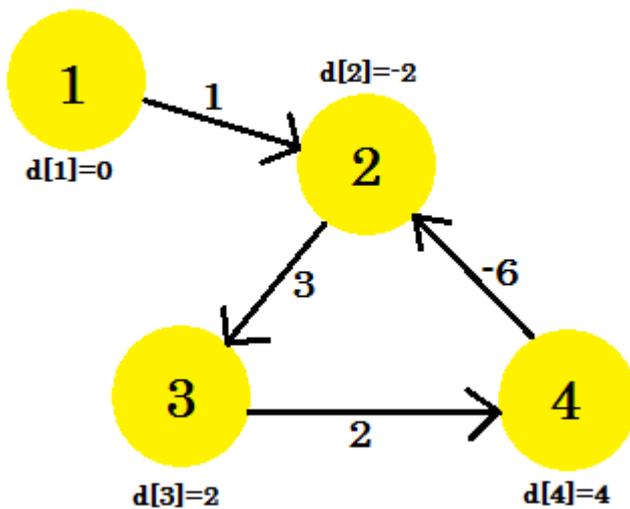
Usando el algoritmo de Bellman-Ford, podemos detectar si hay un ciclo negativo en nuestra gráfica. Sabemos que, para descubrir la ruta más corta, necesitamos *relajar* todos los bordes del gráfico $(V-1)$, donde V es el número de vértices en un gráfico. Ya hemos visto que en este [ejemplo](#) , después de las iteraciones $(V-1)$, no podemos actualizar $d[]$, no importa cuántas iteraciones hagamos. ¿O podemos?

Si hay un ciclo negativo en una gráfica, incluso después de las iteraciones $(V-1)$, podemos actualizar $d[]$. Esto sucede porque para cada iteración, atravesar el ciclo negativo siempre disminuye el costo de la ruta más corta. Esta es la razón por la que el algoritmo de Bellman-Ford limita el número de iteraciones a $(V-1)$. Si usáramos el [algoritmo de Dijkstra](#) aquí, estaríamos atrapados en un bucle sin fin. Sin embargo, concentrémonos en encontrar el ciclo negativo.

Supongamos que tenemos un gráfico:



Vamos a elegir el **vértice 1** como la **fuentes** . Después de aplicar el algoritmo de ruta más corta de una sola fuente de Bellman-Ford al gráfico, descubriremos las distancias desde la **fuentes** a todos los otros vértices.



Así es como se ve la gráfica después de $(V-1) = 3$ iteraciones. Debería ser el resultado ya que hay 4 bordes, necesitamos como máximo 3 iteraciones para encontrar el camino más corto. Entonces, o esta es la respuesta, o hay un ciclo de peso negativo en la gráfica. Para encontrar eso, después de las iteraciones $(V-1)$, hacemos una iteración final más y si la distancia continúa disminuyendo, significa que definitivamente hay un ciclo de peso negativo en la gráfica.

Para este ejemplo: si marcamos 2-3, $d[2] + \text{costo}[2][3]$ nos dará 1 que es menor que $d[3]$. Entonces podemos concluir que hay un ciclo negativo en nuestra gráfica.

Entonces, ¿cómo averiguamos el ciclo negativo? Hacemos una pequeña modificación al procedimiento de 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"
  
```

Así es como descubrimos si hay un ciclo negativo en una gráfica. También podemos modificar el algoritmo de Bellman-Ford para realizar un seguimiento de los ciclos negativos.

Lea Algoritmo de Bellman-Ford en línea: <https://riptutorial.com/es/algorithm/topic/4791/algoritmo-de-bellman-ford>

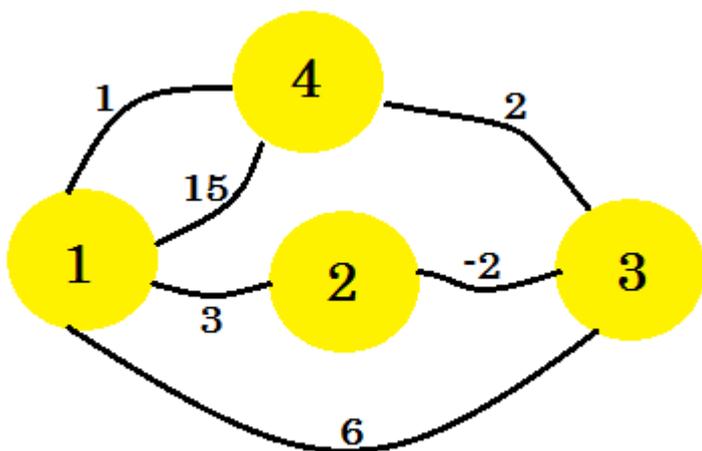
Capítulo 6: Algoritmo de Floyd-Warshall

Examples

Algoritmo de ruta más corta de todos los pares

El algoritmo de Floyd-Warshall es para encontrar rutas más cortas en un gráfico ponderado con ponderaciones de borde positivas o negativas. Una sola ejecución del algoritmo encontrará las longitudes (pesos sumados) de las rutas más cortas entre todos los pares de vértices. Con una pequeña variación, puede imprimir la ruta más corta y puede detectar ciclos negativos en un gráfico. Floyd-Warshall es un algoritmo de programación dinámica.

Veamos un ejemplo. Vamos a aplicar el algoritmo de Floyd-Warshall en este gráfico:



Lo primero que hacemos es tomar dos matrices 2D. Estas son [matrices de adyacencia](#). El tamaño de las matrices será el número total de vértices. Para nuestra gráfica, tomaremos $4 * 4$ matrices. La **Matriz de distancia** almacenará la distancia mínima encontrada hasta ahora entre dos vértices. Al principio, para los bordes, si hay un borde entre uv y la distancia / peso es w , almacenaremos: $distance[u][v] = w$. Por todos los bordes que no existen, vamos a poner el *infinito*. La **Matriz de ruta** es para regenerar la ruta de distancia mínima entre dos vértices. Entonces, inicialmente, si hay una ruta entre u y v , vamos a poner la $path[u][v] = u$. Esto significa que la mejor manera de llegar al **vértice-v** del **vértice-u** es usar el borde que conecta v con u . Si no hay una ruta entre dos vértices, vamos a poner **N** allí para indicar que no hay una ruta disponible ahora. Las dos tablas para nuestra gráfica se verán como:

	1	2	3	4
1	0	3	6	15
2	inf	0	-2	inf
3	inf	inf	0	2
4	1	inf	inf	0

	1	2	3	4
1	1	N	1	1
2	2	N	N	2
3	3	N	N	3
4	4	N	N	N

```

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

```

Como no hay bucle, las diagonales se configuran en **N**. Y la distancia desde el propio vértice es **0**.

Para aplicar el algoritmo Floyd-Warshall, vamos a seleccionar un vértice medio **k**. Luego, para cada vértice **i**, vamos a verificar si podemos ir de **i** a **k** y luego **k** a **j**, donde **j** es otro vértice y minimizar el costo de ir de **i** a **j**. Si la **distancia** actual **[i][j]** es mayor que la **distancia [i][k] + distancia [k][j]**, vamos a poner la **distancia [i][j]** a la suma de esas dos distancias. Y la **ruta [i][j]** se establecerá en la **ruta [k][j]**, ya que es mejor ir de **i** a **k**, y luego **k** a **j**. Todos los vértices serán seleccionados como **k**. Tendremos 3 bucles anidados: para **k** yendo de 1 a 4, **i** va de 1 a 4 y **j** va de 1 a 4. Vamos cheque:

```

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

```

Entonces, lo que básicamente estamos comprobando es, *para cada par de vértices, ¿obtenemos una distancia más corta al atravesar otro vértice?* El número total de operaciones para nuestro gráfico será $4 * 4 * 4 = 64$. Eso significa que vamos a hacer este control **64** veces. Veamos algunos de ellos:

Cuando **k = 1**, **i = 2** y **j = 3**, la **distancia [i][j]** es **-2**, que no es mayor que la **distancia [i][k] + distancia [k][j] = -2 + 0 = -2**. Así se mantendrá sin cambios. De nuevo, cuando **k = 1**, **i = 4** y **j = 2**, **distancia [i][j] = infinito**, que es mayor que la **distancia [i][k] + distancia [k][j] = 1 + 3 = 4**. Entonces colocamos la **distancia [i][j] = 4**, y colocamos la **ruta [i][j] = ruta [k][j] = 1**. Lo que esto significa es que, para pasar del **vértice 4** al **vértice 2**, la ruta **4->1->2** es más corta que la ruta existente. Así es como poblamos ambas matrices. El cálculo para cada paso se muestra [aquí](#). Después de hacer los cambios necesarios, nuestras matrices se verán como:

```

+-----+-----+-----+-----+-----+
|      | 1 | 2 | 3 | 4 |
+-----+-----+-----+-----+
| 1 | 0 | 3 | 1 | 3 |
+-----+-----+-----+-----+
| 2 | 1 | 0 | -2 | 0 |
+-----+-----+-----+-----+
| 3 | 3 | 6 | 0 | 2 |
+-----+-----+-----+-----+
| 4 | 1 | 4 | 2 | 0 |
+-----+-----+-----+-----+
                        distance
+-----+-----+-----+-----+-----+
|      | 1 | 2 | 3 | 4 |
+-----+-----+-----+-----+
| 1 | N | 1 | 2 | 3 |
+-----+-----+-----+-----+
| 2 | 4 | N | 2 | 3 |
+-----+-----+-----+-----+
| 3 | 4 | 1 | N | 3 |
+-----+-----+-----+-----+
| 4 | 4 | 1 | 2 | N |
+-----+-----+-----+-----+
                        path

```

Esta es nuestra matriz de distancias más corta. Por ejemplo, la distancia más corta de **1** a **4** es **3** y la distancia más corta entre **4** a **3** es **2**. Nuestro pseudo-código será:

```

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

```

Imprimiendo el camino:

Para imprimir la ruta, revisaremos la matriz de **ruta** . Para imprimir la ruta de **u** a **v** , comenzaremos desde la **ruta [u] [v]** . Estableceremos seguir cambiando **v = ruta [u] [v]** hasta que encontremos la **ruta [u] [v] = u** y empujemos todos los valores de la **ruta [u] [v]** en una pila. Después de encontrar **u**, vamos a imprimir **u** y empiezan a saltar los elementos de la pila e imprimirlas. Esto funciona porque la matriz de **ruta** almacena el valor del vértice que comparte la ruta más corta a **v** desde cualquier otro nodo. El pseudocódigo será:

```

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

```

Encontrando el Ciclo del Borde Negativo:

Para saber si hay un ciclo de borde negativo, tendremos que verificar la diagonal principal de la matriz de **distancia** . Si algún valor en la diagonal es negativo, eso significa que hay un ciclo negativo en la gráfica.

Complejidad:

La complejidad del algoritmo Floyd-Warshall es **O (V³)** y la complejidad del espacio es: **O (V²)** .

Lea Algoritmo de Floyd-Warshall en línea: <https://riptutorial.com/es/algorithm/topic/7193/algoritmo-de-floyd-warshall>

Capítulo 7: Algoritmo de Knuth Morris Pratt (KMP)

Introducción

El KMP es un algoritmo de coincidencia de patrones que busca las apariciones de una "palabra" **W** dentro de una "cadena de texto" principal **S** empleando la observación de que cuando se produce una discrepancia, tenemos la información suficiente para determinar dónde podría comenzar la siguiente coincidencia. aproveche esta información para evitar que coincidan los caracteres que sabemos que de todos modos coincidirán. La complejidad del peor de los casos para buscar un patrón se reduce a $O(n)$.

Examples

Ejemplo de KMP

Algoritmo

Este algoritmo es un proceso de dos pasos. Primero creamos una matriz auxiliar `lps []` y luego usamos esta matriz para buscar el patrón.

Preprocesamiento

1. Preprocesamos el patrón y creamos una matriz auxiliar `lps []` que se usa para omitir caracteres al hacer coincidir.
2. Aquí `lps []` indica el prefijo apropiado más largo que también es sufijo. Un prefijo apropiado es el prefijo en el que no se incluye la cadena completa. Por ejemplo, los prefijos de la cadena **ABC** son "", "A", "AB" y "ABC". Los prefijos apropiados son "", "A" y "AB". Los sufijos de la cadena son "", "C", "BC" y "ABC".

buscando

1. Seguimos combinando los caracteres `txt [i]` y `pat [j]` y seguimos incrementando `i` y `j` mientras `pat [j]` y `txt [i]` siguen coincidiendo.
2. Cuando vemos una discrepancia, sabemos que los caracteres `pat [0..j-1]` coinciden con `txt [i-j + 1... i-1]`. También sabemos que `lps [j-1]` es el recuento de caracteres de `pat [0... j-1]` que son el prefijo y el sufijo propios. De esto podemos concluir que no necesitamos hacer coincidir estos caracteres `lps [j-1]` con `txt [ij ... i-1]` porque sabemos que estos caracteres coincidirá de todos modos.

Implementación en 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;
}
}

```

Lea Algoritmo de Knuth Morris Pratt (KMP) en línea:

<https://riptutorial.com/es/algorithm/topic/10811/algoritmo-de-knuth-morris-pratt--kmp->

Capítulo 8: Algoritmo de línea

Introducción

El dibujo lineal se realiza calculando posiciones intermedias a lo largo de la trayectoria de línea entre dos posiciones de punto final especificadas. Luego se dirige un dispositivo de salida para completar estas posiciones entre los puntos finales.

Examples

Algoritmo de dibujo lineal de Bresenham

Antecedentes de la teoría: el algoritmo de dibujo de líneas de Bresenham es un algoritmo de generación de líneas de trama eficiente y preciso desarrollado por Bresenham. Implica solo el cálculo de enteros para que sea preciso y rápido. También se puede ampliar para mostrar círculos de otras curvas.

En el algoritmo de dibujo de líneas de Bresenham:

Para Pendiente $|m| < 1$:

Cualquiera de los valores de x aumenta

O tanto x como y se incrementan usando el parámetro de decisión.

Para Pendiente $|m| > 1$:

Cualquiera de los valores de y se incrementa

O tanto x como y se incrementan usando el parámetro de decisión.

Algoritmo para pendiente $|m| < 1$:

1. Introduzca dos puntos finales (x_1, y_1) y (x_2, y_2) de la línea.
2. Grafica el primer punto (x_1, y_1) .
3. Calcular
 $Delx = |x_2 - x_1|$
 $Dely = |y_2 - y_1|$
4. Obtener el parámetro de decisión inicial como
 $P = 2 * dely - delx$
5. Para $l = 0$ para $delx$ en el paso de 1

Si $p < 0$ entonces

$X1 = x_1 + 1$

Olla (x_1, y_1)

$P = p + 2 * dely$

Más

$X1 = x1 + 1$

$Y1 = y1 + 1$

Parcela (x1, y1)

$P = p + 2 \text{ dely} - 2 * \text{delx}$

Terminara si

Fin de

6. FIN

Código fuente:

```
/* 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;
}
```

```
}
```

Algoritmo para pendiente $|m| > 1$:

1. Introduzca dos puntos finales (x_1, y_1) y (x_2, y_2) de la línea.
2. Grafica el primer punto (x_1, y_1) .
3. Calcular
$$\text{Delx} = |x_2 - x_1|$$
$$\text{Dely} = |y_2 - y_1|$$
4. Obtener el parámetro de decisión inicial como
$$P = 2 * \text{delx} - \text{dely}$$
5. Para $i = 0$ para dely en el paso de 1
Si $p < 0$ entonces
$$y_1 = y_1 + 1$$
Olla (x_1, y_1)
$$P = p + 2 \text{delx}$$

Más

$$X1 = x1 + 1$$
$$Y1 = y1 + 1$$

Parcela (x_1, y_1)

$$P = p + 2 \text{delx} - 2 * \text{dely}$$

Terminara si

Fin de

6. FIN

Código fuente:

```
/* 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;
}
```

Lea Algoritmo de linea en línea: <https://riptutorial.com/es/algorithm/topic/7596/algoritmo-de-linea>

Capítulo 9: Algoritmo de partición entero

Examples

Información básica del algoritmo de partición entero

La [partición de un entero](#) es una forma de escribirlo como una suma de enteros positivos. Por ejemplo, las particiones del número 5 son:

- 5
- 4 + 1
- 3 + 2
- 2 + 2 + 1
- 2 + 1 + 1 + 1
- 1 + 1 + 1 + 1 + 1

Tenga en cuenta que cambiar el orden de los sumandos no creará una partición diferente.

La función de partición es intrínsecamente recursiva ya que los resultados de números más pequeños aparecen como componentes en el resultado de un número más grande. Sea $p(n, m)$ el número de particiones de n que usan solo números enteros positivos que son menores o iguales que m . Se puede ver que $p(n) = p(n, n)$, y también $p(n, m) = p(n, n) = p(n)$ para $m > n$.

$$p(n, m) = \sum_{k=1}^m p(n - k, k)$$

Ejemplo de algoritmo de partición entero:


```
        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);
}
}
```

Lea Algoritmo de partición entero en línea:

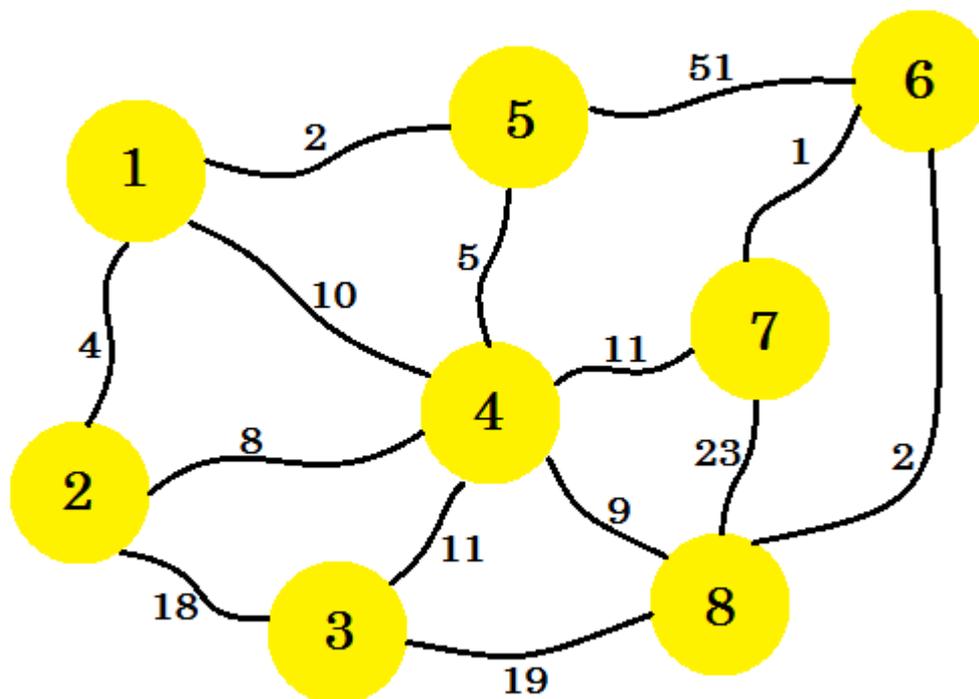
<https://riptutorial.com/es/algorithm/topic/7424/algoritmo-de-particion-entero>

Capítulo 10: Algoritmo de Prim

Examples

Introducción al algoritmo de Prim

Digamos que tenemos **8** casas. Queremos configurar líneas telefónicas entre estas casas. El borde entre las casas representa el costo de establecer la línea entre dos casas.



Nuestra tarea es configurar líneas de tal manera que todas las casas estén conectadas y el costo de configurar toda la conexión sea mínimo. Ahora, ¿cómo averiguamos eso? Podemos usar el **algoritmo de Prim** .

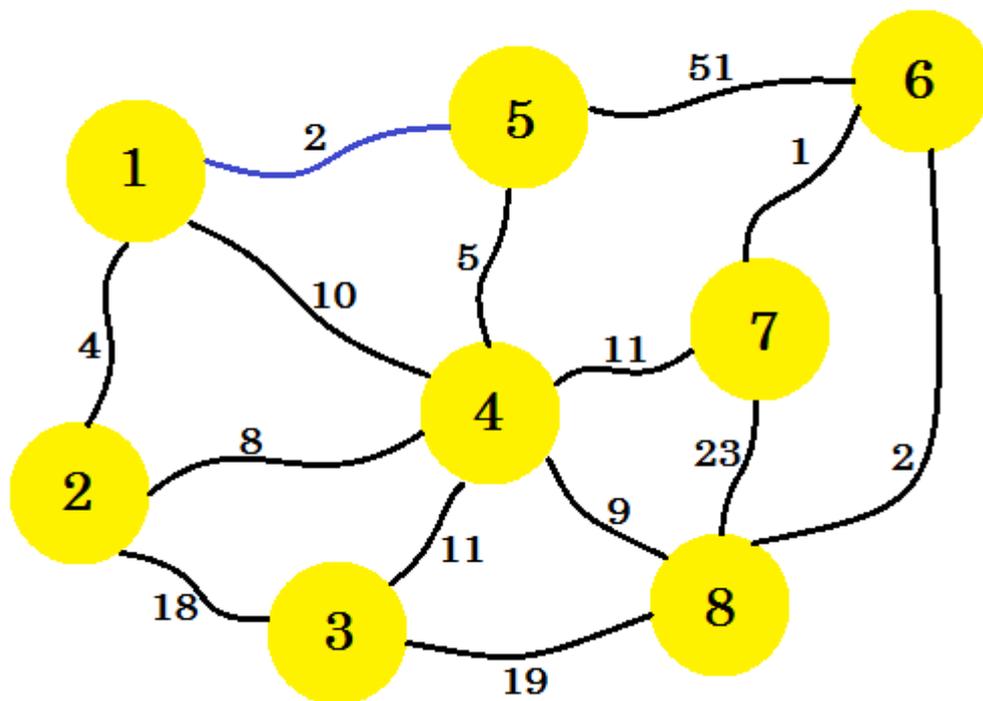
El **algoritmo de Prim** es un algoritmo codicioso que encuentra un árbol de expansión mínimo para un gráfico no dirigido ponderado. Esto significa que encuentra un subconjunto de los bordes que forma un árbol que incluye todos los nodos, donde se minimiza el peso total de todos los bordes del árbol. El algoritmo fue desarrollado en 1930 por el matemático checo [Vojtěch Jarník](#) y luego redescubierto y publicado por el científico informático [Robert Clay Prim](#) en 1957 y [Edsger Wybe Dijkstra](#) en 1959. También se conoce como **algoritmo DJP** , **algoritmo de Jarnik**, **algoritmo de Prim-Jarnik** o **algoritmo de Prim-Dijkstra**. **algoritmo**

Ahora veamos los términos técnicos primero. Si creamos un gráfico, **S** utilizando algunos nodos y bordes de un gráfico **G** no dirigido, entonces **S** se llama un **subgrafo** del gráfico **G**. Ahora **S** se llamará **árbol de expansión** si y solo si:

- Contiene todos los nodos de **G**.
- Es un árbol, eso significa que no hay ciclo y todos los nodos están conectados.
- Hay **(n-1)** bordes en el árbol, donde **n** es el número de nodos en **G**.

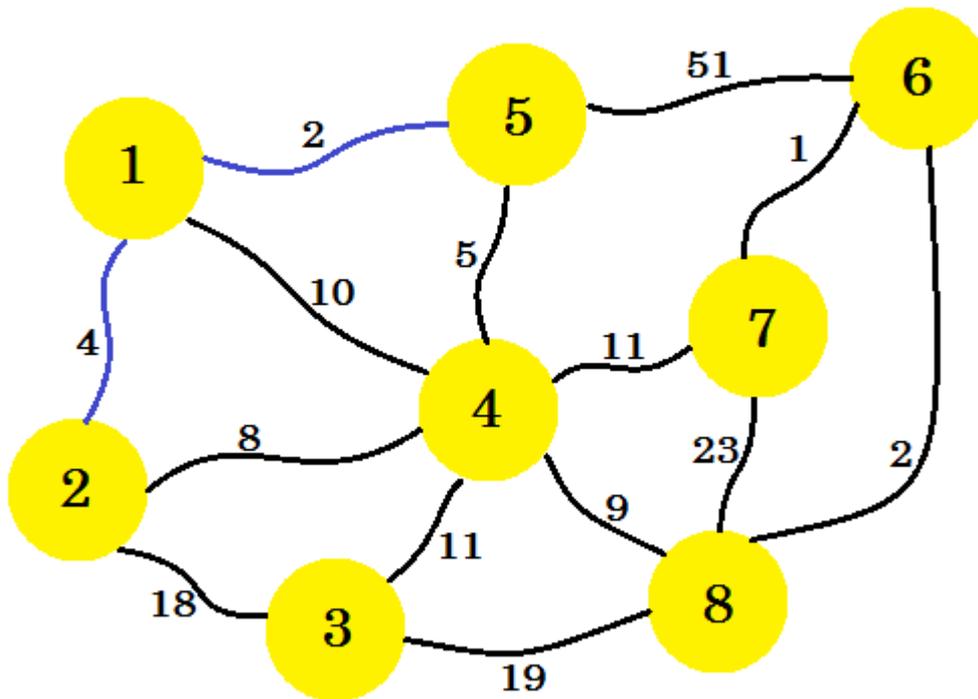
Puede haber muchos **árboles de expansión** de un gráfico. El **árbol de expansión mínima** de un gráfico no dirigido ponderado es un árbol, por lo que la suma del peso de los bordes es mínima. Ahora usaremos **el algoritmo de Prim** para averiguar el árbol de expansión mínima, que es cómo configurar las líneas telefónicas en nuestro gráfico de ejemplo de tal manera que el costo de configuración sea mínimo.

Al principio seleccionaremos un nodo **fuentes** . Digamos que el **nodo-1** es nuestra **fuentes** . Ahora agregaremos el borde del **nodo-1** que tiene el costo mínimo para nuestro subgrafo. Aquí marcamos los bordes que están en el subgrafo usando el color **azul** . Aquí **1-5** es nuestro borde

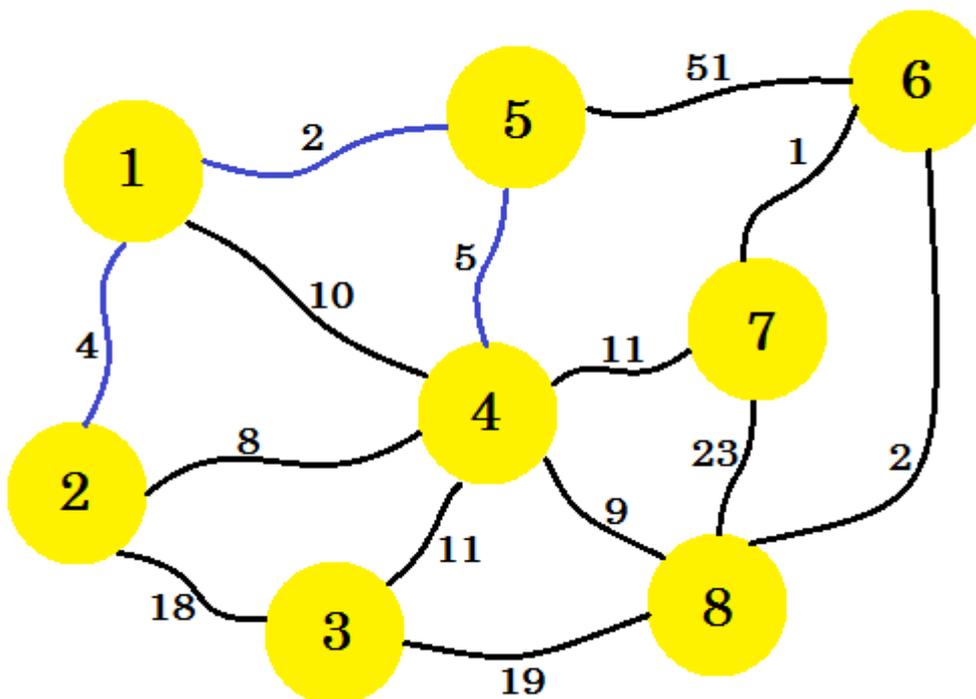


deseado.

Ahora consideramos todos los bordes del **nodo 1** y el **nodo 5** y tomamos el mínimo. Como **1-5** ya está marcado, tomamos **1-2** .

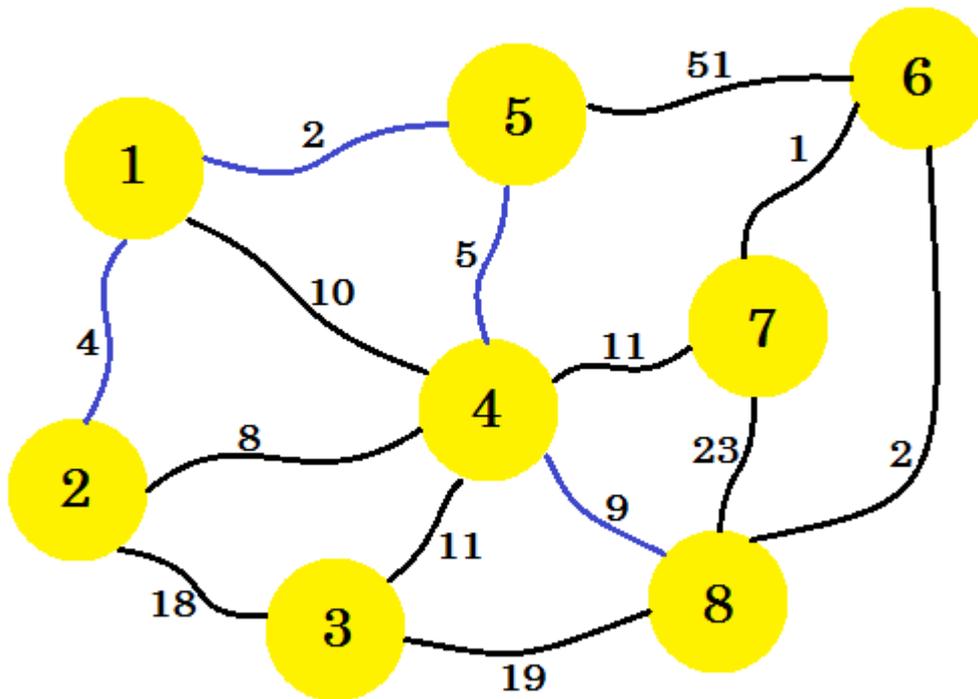


Esta vez, consideramos los **nodos-1** , **nodos-2** y **nodos-5** y tomamos el borde mínimo que es **5-**

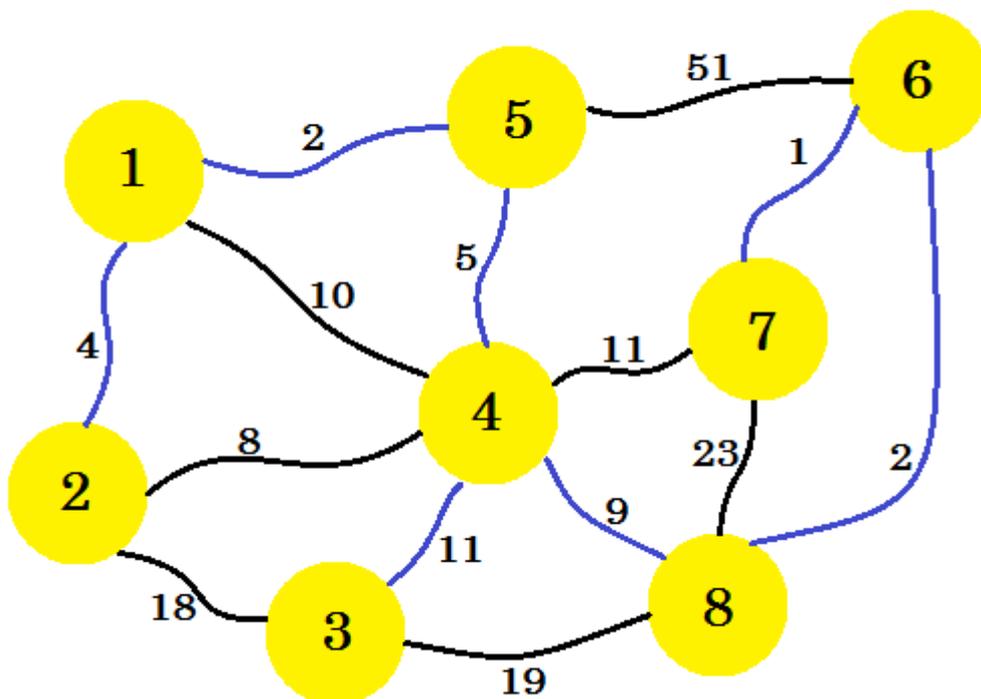


4 .

El siguiente paso es importante. Desde el **nodo 1** , el **nodo 2** , el **nodo 5** y el **nodo 4** , el límite mínimo es **2-4** . Pero si seleccionamos ese, creará un ciclo en nuestro subgrafo. Esto se debe a que **node-2** y **node-4** ya están en nuestro subgrafo. Así que tomar ventaja **2-4** no nos beneficia. *Seleccionaremos los bordes de tal manera que agregue un nuevo nodo en nuestro subgrafo* . Así que seleccionamos el borde **4-8** .

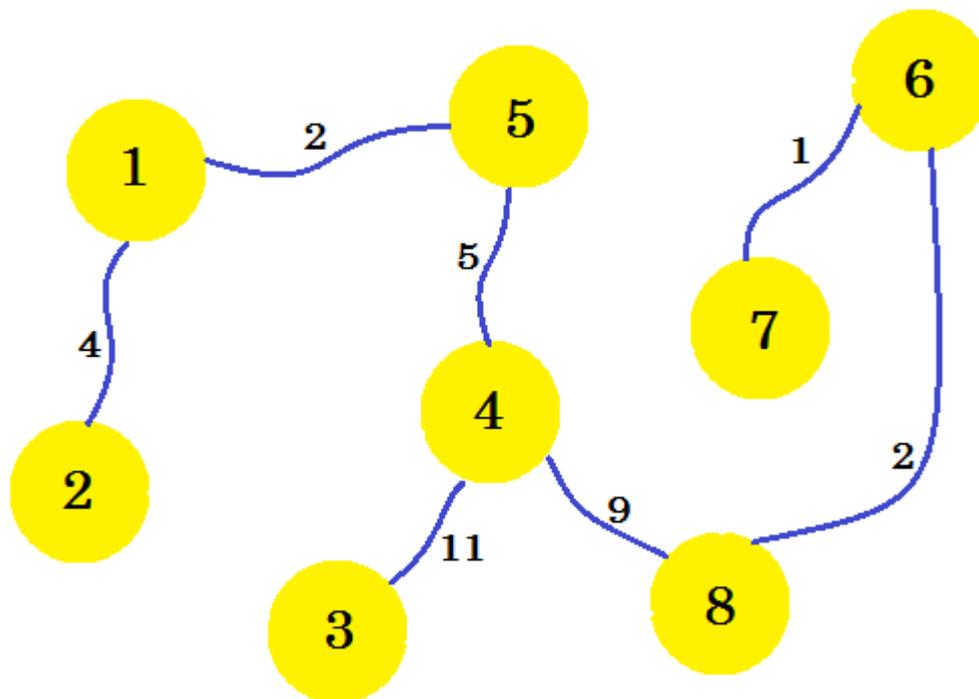


Si continuamos de esta manera, seleccionaremos el borde 8-6 , 6-7 y 4-3 . Nuestro subgrafo se



verá como:

Este es nuestro subgrafo deseado, que nos dará el árbol de expansión mínimo. Si eliminamos los bordes que no seleccionamos, obtendremos:



Este es nuestro **árbol de expansión mínima (MST)**. Entonces, el costo de configurar las conexiones telefónicas es: $4 + 2 + 5 + 11 + 9 + 2 + 1 = 34$. Y el conjunto de casas y sus conexiones se muestran en el gráfico. Puede haber múltiples **MST** de un gráfico. Depende del nodo **fuente** que elijamos.

El pseudocódigo del algoritmo se da a continuación:

```

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
  
```

Complejidad:

La complejidad temporal del enfoque ingenuo anterior es $O(V^2)$. Utiliza matriz de adyacencia. Podemos reducir la complejidad usando la cola de prioridad. Cuando agregamos un nuevo nodo a **Vnew**, podemos agregar sus bordes adyacentes en la cola de prioridad. Luego saca el borde mínimo ponderado de él. Entonces la complejidad será: $O(E \log E)$, donde **E** es el número de bordes. Nuevamente, se puede construir un montón binario para reducir la complejidad a $O(E \log V)$.

El pseudocódigo que utiliza la cola de prioridad se muestra a continuación:

```

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 u
        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

```

Aquí la **clave []** almacena el costo mínimo de atravesar **node-v** . **parent []** se utiliza para almacenar el nodo padre. Es útil para atravesar e imprimir el árbol.

A continuación se muestra un programa simple en Java:

```

import java.util.*;

public class Graph
{
    private static int infinite = 99999999;
    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();
}
}

```

Compila el código anterior usando `javac Graph.java`

Salida:

```

$ 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

```

Lea Algoritmo de Prim en línea: <https://riptutorial.com/es/algorithm/topic/7285/algoritmo-de-prim>

Capítulo 11: Algoritmo de subarray máximo

Examples

Información básica del algoritmo de subarray máximo

El [problema de subarray máximo](#) es el método para encontrar el subarray contiguo dentro de una matriz unidimensional de números que tiene la suma más grande.

El problema fue propuesto originalmente por [Ulf Grenander](#) de la Brown University en 1977, como un modelo simplificado para la estimación de máxima probabilidad de patrones en imágenes digitalizadas.

Podemos resolver este problema, consideremos una lista de varios enteros. Podríamos estar interesados en cuál subconjunto completamente adyacente tendrá la mayor suma. Por ejemplo, si tenemos la matriz `[0, 1, 2, -2, 3, 2]`, el subarreglo máximo es `[3, 2]`, con una suma de 5.

Enfoque de fuerza bruta para la solución:

Este método es el más ineficiente para encontrar la solución. En esto, terminaremos repasando cada posible subarreglo, y luego encontraremos la suma de todos ellos. Por fin, compare todos los valores y averigüe el subarray máximo.

Pseudo código para el enfoque de fuerza bruta:

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

La complejidad del tiempo para el método de fuerza bruta es $O(n^2)$. Así que vamos a pasar a dividir y conquistar el enfoque.

Divide y conquista el enfoque para la solución:

Encuentra la suma de los subarrays en el lado izquierdo, los subarrays en el lado derecho. Luego, eche un vistazo a todos los que cruzan la división del centro y finalmente devuelva la suma máxima. Debido a que este es un algoritmo de dividir y conquistar, necesitamos tener dos funciones diferentes.

Lo primero es dividir paso,

```
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))

```

En la segunda parte, separe las diferentes partes que se crean en la primera parte.

```

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

```

La complejidad del tiempo para el método de Dividir y Conquistar es $O(n \log n)$. Así que vamos a pasar al enfoque de la programación dinámica.

Enfoque de programación dinámica:

Esta solución también se conoce como algoritmo de Kadane. Es un algoritmo de tiempo lineal. Esta solución es dada por [Joseph B. Kadane](#) a finales de los 70.

Este algoritmo solo pasa por el bucle, cambiando continuamente la suma máxima actual. Curiosamente, este es un ejemplo muy simple de un algoritmo de programación dinámica, ya que toma un problema de superposición y lo reduce para que podamos encontrar una solución más eficiente.

Pseudo código del algoritmo de Kadane:

```

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

```

La complejidad del tiempo para el algoritmo de Kadane es $O(n)$.

Implementación de 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);
    }
}
```

Lea Algoritmo de subarray máximo en línea:

<https://riptutorial.com/es/algorithm/topic/7578/algoritmo-de-subarray-maximo>

Capítulo 12: Algoritmo de suma de ruta máxima

Examples

Información básica de la suma máxima de ruta

Maximum Path Sum es un algoritmo para encontrar una ruta tal que la suma del elemento (nodo) de esa ruta sea mayor que cualquier otra ruta.

Por ejemplo, tengamos un triángulo, como se muestra a continuación.

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

En el triángulo anterior, encuentra el camino máximo que tiene la suma máxima. La respuesta es, $3 + 7 + 4 + 9 = 23$

Para encontrar la solución, como siempre, tenemos una idea del método de Fuerza Bruta. El método de fuerza bruta es bueno para este triángulo de 4 filas, pero piensa en un triángulo con 100 o más de 100 filas. Por lo tanto, no podemos utilizar el método de fuerza bruta para resolver este problema.

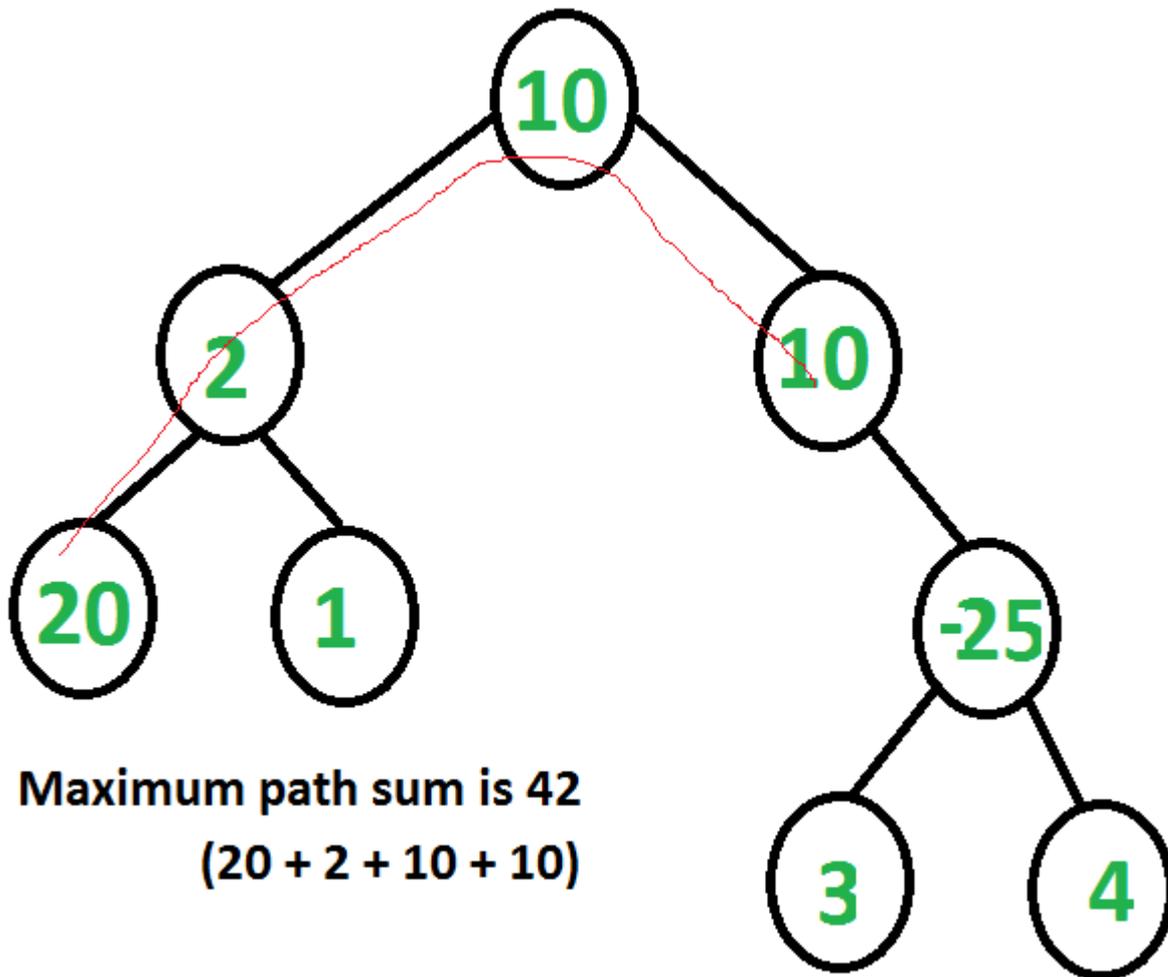
Vayamos al enfoque de la programación dinámica.

Algoritmo:

Para todos y cada uno de los nodos en un triángulo o en un árbol binario, puede haber cuatro formas en que la ruta máxima pasa por el nodo.

1. Solo nodo
2. Ruta máxima a través del hijo izquierdo + nodo
3. Ruta máxima a través del hijo derecho + nodo
4. Ruta máxima a través del hijo izquierdo + Nodo + Ruta máxima a través del niño derecho.

Ejemplo de algoritmo de suma de ruta máxima:



Espacio auxiliar: $O(n)$

Complejidad del tiempo: $O(n)$

Implementación de 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();
}
}

```

Lea Algoritmo de suma de ruta máxima en línea:

<https://riptutorial.com/es/algorithm/topic/7570/algoritmo-de-suma-de-ruta-maxima>

Capítulo 13: Algoritmo de ventana deslizante

Examples

Algoritmo de ventana deslizante Información básica

El algoritmo de la ventana deslizante se utiliza para realizar la operación requerida en un tamaño de ventana específico de un gran búfer o conjunto dado. La ventana comienza desde el primer elemento y sigue desplazándose hacia la derecha un elemento. El objetivo es encontrar los k números mínimos presentes en cada ventana. Esto se conoce comúnmente como problema de ventana deslizante o algoritmo.

Por ejemplo, para encontrar el elemento máximo o mínimo de cada elemento n en una matriz dada, se utiliza el algoritmo de ventana deslizante.

Ejemplo:

Matriz de entrada: [1 3 -1 -3 5 3 6 7]

Tamaño de la ventana: 3

Elemento **máximo** de cada elemento 3 de la matriz de entrada:

```
+-----+
|      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      |
+-----+-----+-----+-----+-----+
```

Elemento **mínimo** de cada 3 elementos de la matriz de entrada:

```
+-----+
|      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 |
+---+---+---+---+-----+---+---+

```

Métodos para encontrar la suma de 3 elementos:

Método 1:

La primera forma es usar la clasificación rápida, cuando el pivote está en la posición Kth, todos los elementos en el lado derecho son mayores que el pivote, por lo tanto, todos los elementos en el lado izquierdo se convierten automáticamente en los elementos más pequeños de la matriz dada.

Método 2:

Mantenga una matriz de elementos K, llénela con los primeros K elementos de la matriz de entrada dada. Ahora, desde el elemento K + 1, verifique si el elemento actual es menor que el elemento máximo en la matriz auxiliar, en caso afirmativo, agregue este elemento a la matriz. El único problema con la solución anterior es que debemos realizar un seguimiento del elemento máximo. Aún es factible. ¿Cómo podemos realizar un seguimiento del elemento máximo en conjunto de entero? Piense montón. Piensa en el montón de Max.

Método 3:

¡Genial! En $O(1)$ obtendríamos el elemento máximo entre los elementos K ya elegidos como los elementos K más pequeños. Si el máximo en el conjunto actual es mayor que el elemento recientemente considerado, debemos eliminar el máximo e introducir un nuevo elemento en el conjunto del K elemento más pequeño. Heapify nuevamente para mantener la propiedad del montón. Ahora podemos obtener fácilmente K elementos mínimos en la matriz de N.

Espacio auxiliar: $O(n)$

Complejidad del tiempo: $O(n)$

Implementación del algoritmo de ventana deslizante en 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);
}
}
```

Lea Algoritmo de ventana deslizante en línea:

<https://riptutorial.com/es/algorithm/topic/7622/algoritmo-de-ventana-deslizante>

Capítulo 14: Algoritmo delimitado por tiempo polinómico para la cobertura mínima de vértices

Introducción

Este es un algoritmo polinomial para obtener la cobertura mínima de vértice de un gráfico no dirigido conectado. La complejidad temporal de este algoritmo es $O(n^2)$.

Parámetros

Variable	Sentido
sol	Entrada conectada grafica no dirigida
X	Conjunto de vértices
do	Conjunto final de vértices

Observaciones

Lo primero que debe hacer en este algoritmo para obtener todos los vértices de la gráfica ordenados en orden descendente según su grado.

Después de eso, hay que iterar en ellos y agregar cada uno al conjunto de vértices finales que no tienen ningún vértice adyacente en este conjunto.

En la etapa final, itere en el conjunto de vértices finales y elimine todos los vértices que tengan uno de sus vértices adyacentes en este conjunto.

Examples

Algoritmo Pseudo Código

Algoritmo PMinVertexCover (gráfico G)

Entrada conectada grafo G

Conjunto de cubierta de vértice mínimo de salida C

```

Set C <- new Set<Vertex>()

Set X <- new Set<Vertex>()

X <- G.getAllVerticesArrangedDescendinglyByDegree()

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 es la cubierta de vértice mínima del gráfico G

podemos usar la clasificación de cubos para clasificar los vértices según su grado porque el valor máximo de grados es $(n-1)$ donde n es el número de vértices, entonces la complejidad del tiempo de clasificación será $O(n)$

Lea [Algoritmo delimitado por tiempo polinómico para la cobertura mínima de vértices en línea](https://riptutorial.com/es/algorithm/topic/9535/algorithm-delimitado-por-tiempo-polinomico-para-la-cobertura-minima-de-vertices):
<https://riptutorial.com/es/algorithm/topic/9535/algorithm-delimitado-por-tiempo-polinomico-para-la-cobertura-minima-de-vertices>

Capítulo 15: Algoritmo Numérico Catalán

Examples

Algoritmo Numérico Catalán Información Básica

El algoritmo de números catalán es el algoritmo de programación dinámica.

En las matemáticas combinatorias, los **números catalanes** forman una secuencia de números naturales que ocurren en varios problemas de conteo, que a menudo involucran objetos definidos recursivamente. Los números catalanes en enteros no negativos n son un conjunto de números que surgen en los problemas de enumeración de árboles del tipo, '¿De cuántas maneras se puede dividir un n -gon regular en $n-2$ triángulos si las diferentes orientaciones se cuentan por separado?'

Aplicación del algoritmo numérico catalán:

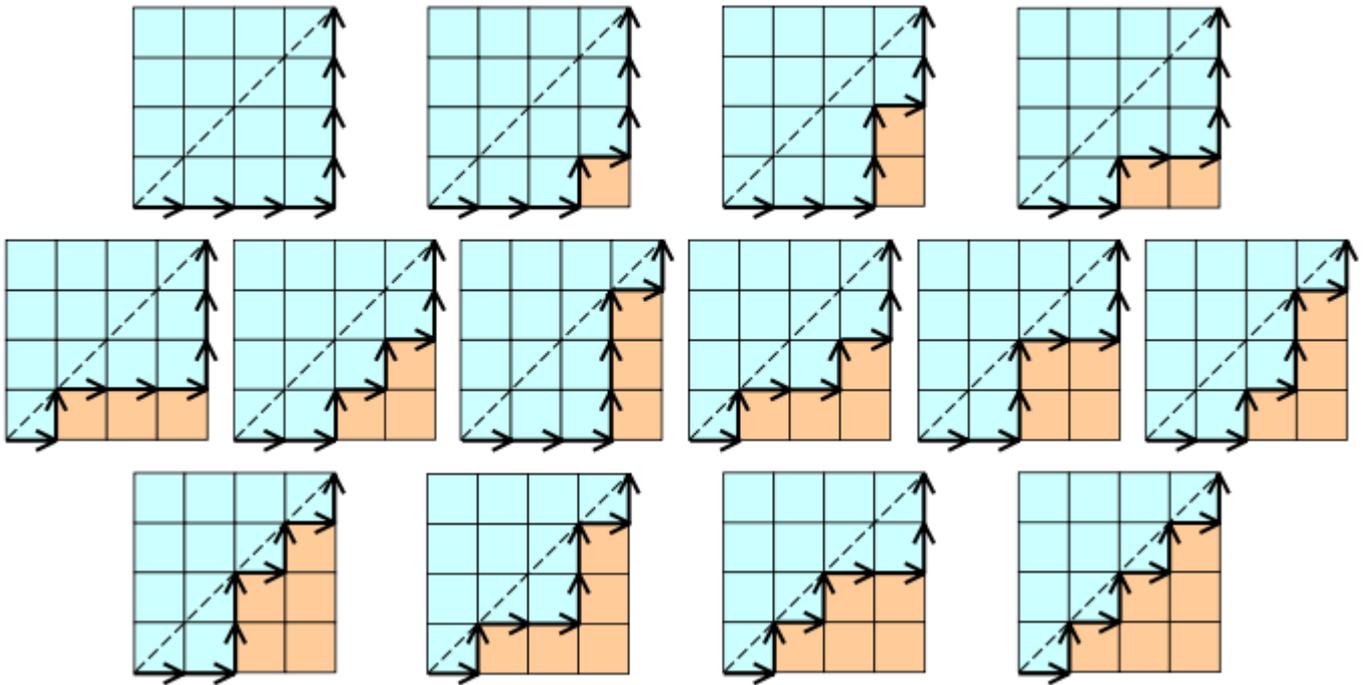
1. El número de formas de apilar monedas en una fila inferior que consta de n monedas consecutivas en un plano, de manera que no se permite colocar monedas en los dos lados de las monedas inferiores y cada moneda adicional debe estar por encima de otras dos monedas, es El número n catalán.
2. La cantidad de formas de agrupar una serie de n pares de paréntesis, de manera que cada paréntesis abierto tenga un paréntesis cerrado coincidente, es el número n catalán.
3. La cantidad de formas de cortar un polígono convexo de $n + 2$ lados en un plano en triángulos conectando vértices con líneas rectas que no se intersectan es el número n catalán. Esta es la aplicación en la que Euler estaba interesado.

Usando una numeración basada en cero, el número n catalán se da directamente en términos de coeficientes binomiales mediante la siguiente ecuación.

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

Ejemplo de número catalán:

Aquí el valor de $n = 4$. (Mejor ejemplo: de Wikipedia)



Espacio auxiliar: $O(n)$

Complejidad del tiempo: $O(n^2)$

Implementación de 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;
    }
}
```

Lea Algoritmo Numérico Catalán en línea: <https://riptutorial.com/es/algorithm/topic/7406/algoritmo-numerico-catalan>

Capítulo 16: Algoritmos codiciosos

Observaciones

Un algoritmo codicioso es un algoritmo en el que en cada paso elegimos la opción más beneficiosa en cada paso sin mirar hacia el futuro. La elección depende solo del beneficio actual.

El enfoque codicioso suele ser un buen enfoque cuando cada ganancia puede recogerse en cada paso, por lo que ninguna opción bloquea otra.

Examples

Problema continuo de la mochila.

Dados los artículos como $(value, weight)$ debemos colocarlos en una mochila (contenedor) de una capacidad k . ¡Nota! ¡Podemos romper artículos para maximizar el valor!

Ejemplo de entrada:

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

Rendimiento esperado:

```
maximumValueOfItemsInK = 20;
```

Algoritmo:

```
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;
```

Codificación Huffman

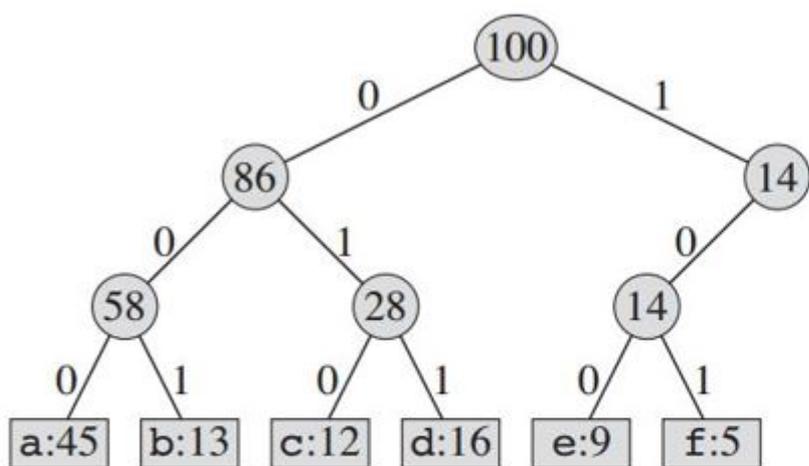
El [código de Huffman](#) es un tipo particular de código de prefijo óptimo que se usa comúnmente para la compresión de datos sin pérdida. Comprime los datos de manera muy efectiva, ahorrando

de 20% a 90% de memoria, dependiendo de las características de los datos comprimidos. Consideramos que los datos son una secuencia de caracteres. El codicioso algoritmo de Huffman utiliza una tabla que proporciona la frecuencia con la que aparece cada carácter (es decir, su frecuencia) para crear una forma óptima de representar cada carácter como una cadena binaria. El código de Huffman fue propuesto por [David A. Huffman](#) en 1951.

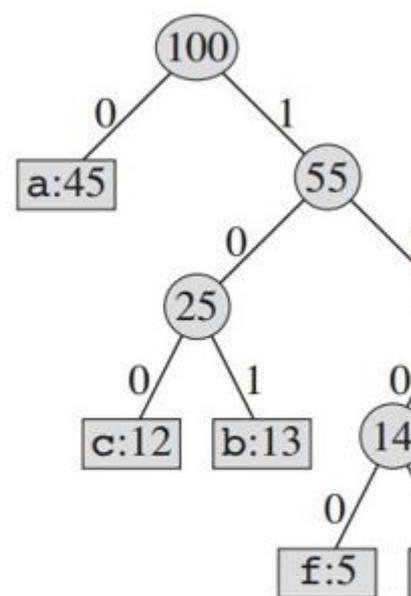
Supongamos que tenemos un archivo de datos de 100.000 caracteres que deseamos almacenar de forma compacta. Suponemos que solo hay 6 caracteres diferentes en ese archivo. La frecuencia de los personajes viene dada por:

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

Tenemos muchas opciones sobre cómo representar tal archivo de información. Aquí, consideramos el problema de diseñar un *código de carácter binario* en el que cada carácter está representado por una cadena binaria única, que llamamos una **palabra de código**.



Fixed-length Codeword



Variable-length Codeword

El árbol construido nos proporcionará:

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

Si usamos un **código de longitud fija**, necesitamos tres bits para representar 6 caracteres. Este

método requiere 300,000 bits para codificar el archivo completo. Ahora la pregunta es, ¿podemos hacerlo mejor?

Un **código de longitud variable** puede ser considerablemente mejor que un código de longitud fija, al dar caracteres frecuentes palabras en clave cortas y caracteres poco frecuentes palabras en código largas. Este código requiere: $(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1000 = 224000$ bits para representar el archivo, lo que ahorra aproximadamente el 25% de la memoria.

Una cosa para recordar, consideramos aquí solo los códigos en los que ninguna palabra de código es también un prefijo de alguna otra palabra de código. Estos son llamados *códigos de prefijo*. Para la codificación de longitud variable, codificamos el archivo *abc* de 3 caracteres como 0.101.100 = 0101100, donde "." Denota la concatenación.

Los códigos de prefijo son deseables porque simplifican la decodificación. Dado que ninguna palabra de código es un prefijo de otro, la palabra de código que comienza un archivo codificado es inequívoca. Podemos simplemente identificar la palabra de código inicial, traducirla de nuevo al carácter original y repetir el proceso de decodificación en el resto del archivo codificado. Por ejemplo, 001011101 analiza únicamente como 0.0.101.1101, que se decodifica a *aabe*. En resumen, todas las combinaciones de representaciones binarias son únicas. Digamos, por ejemplo, si una letra se denota por 110, ninguna otra se denotará por 1101 o 1100. Esto se debe a que podría tener confusión sobre si seleccionar 110 o continuar concatenando el siguiente bit y seleccionando ese.

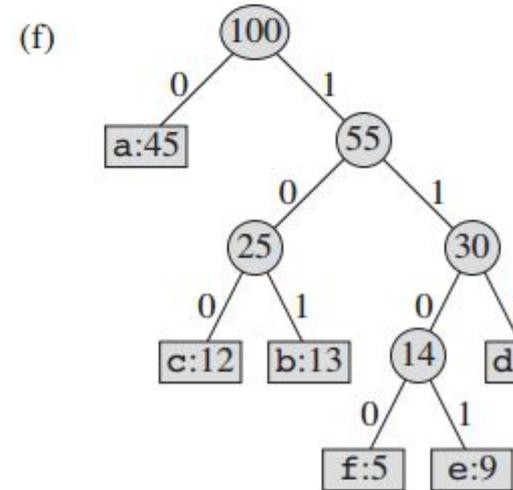
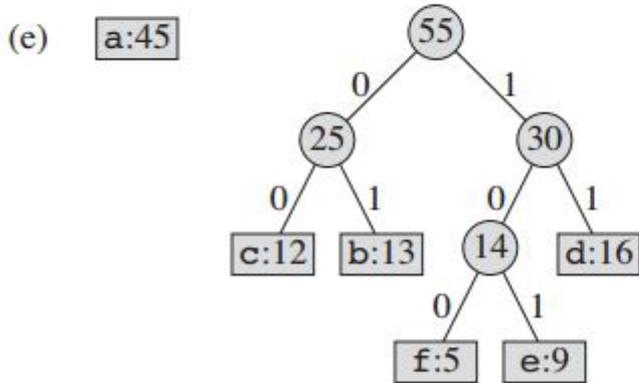
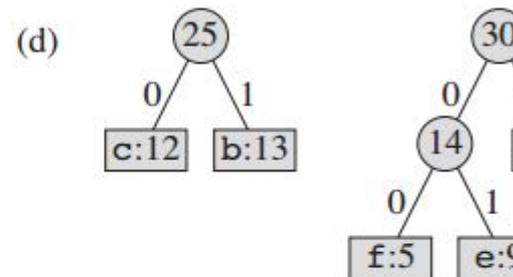
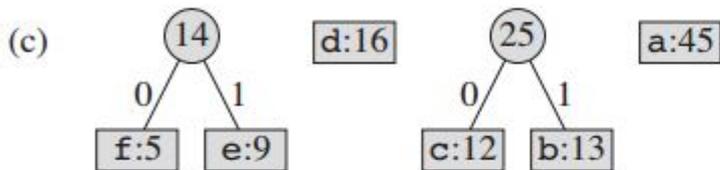
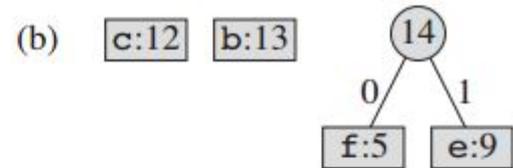
Técnica de compresión:

La técnica funciona creando un *árbol binario* de nodos. Estos pueden almacenarse en una matriz regular, cuyo tamaño depende del número de símbolos, n . Un nodo puede ser un *nodo de hoja* o un *nodo interno*. Inicialmente, todos los nodos son nodos de hoja, que contienen el símbolo mismo, su frecuencia y, opcionalmente, un enlace a sus nodos secundarios. Como convención, el bit '0' representa el hijo izquierdo y el bit '1' representa el niño derecho. *La cola de prioridad* se utiliza para almacenar los nodos, lo que proporciona al nodo la frecuencia más baja cuando se abre. El proceso se describe a continuación:

1. Cree un nodo de hoja para cada símbolo y agréguelo a la cola de prioridad.
2. Si bien hay más de un nodo en la cola:
 1. Eliminar los dos nodos de mayor prioridad de la cola.
 2. Cree un nuevo nodo interno con estos dos nodos como hijos y con una frecuencia igual a la suma de la frecuencia de los dos nodos.
 3. Agregue el nuevo nodo a la cola.
3. El nodo restante es el nodo raíz y el árbol de Huffman está completo.

Para nuestro ejemplo:

(a) f:5 e:9 c:12 b:13 d:16 a:45



El pseudocódigo parece:

```
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
```

A pesar de que el tiempo lineal recibe una entrada ordenada, en general en los casos de una entrada arbitraria, el uso de este algoritmo requiere una clasificación previa. Por lo tanto, dado

que la clasificación lleva tiempo **O (n log n)** en casos generales, ambos métodos tienen la misma complejidad.

Como **n** aquí es el número de símbolos en el alfabeto, que suele ser un número muy pequeño (en comparación con la longitud del mensaje a codificar), la complejidad del tiempo no es muy importante en la elección de este algoritmo.

Técnica de descompresión:

El proceso de descompresión es simplemente una cuestión de traducir la secuencia de códigos de prefijo a un valor de byte individual, generalmente atravesando el nodo del árbol de Huffman por nodo a medida que se lee cada bit de la secuencia de entrada. Alcanzar un nodo hoja termina necesariamente la búsqueda de ese valor de byte en particular. El valor de la hoja representa el carácter deseado. Por lo general, el árbol Huffman se construye utilizando datos ajustados estadísticamente en cada ciclo de compresión, por lo que la reconstrucción es bastante simple. De lo contrario, la información para reconstruir el árbol debe enviarse por separado. El pseudocódigo:

```
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
```

Explicación codiciosa:

La codificación de Huffman examina la aparición de cada carácter y lo almacena como una cadena binaria de una manera óptima. La idea es asignar códigos de longitud variable a los caracteres de entrada, la longitud de los códigos asignados se basa en las frecuencias de los caracteres correspondientes. Creamos un árbol binario y lo operamos de manera ascendente para que los menos dos caracteres frecuentes estén lo más lejos posible de la raíz. De esta manera, el carácter más frecuente obtiene el código más pequeño y el carácter menos frecuente obtiene el código más grande.

Referencias:

- Introducción a los algoritmos: Charles E. Leiserson, Clifford Stein, Ronald Rivest y Thomas H. Cormen
- [Codificación Huffman](#) - Wikipedia
- Matemáticas discretas y sus aplicaciones - Kenneth H. Rosen

Problema de cambio

Dado un sistema monetario, ¿es posible dar una cantidad de monedas y cómo encontrar un conjunto mínimo de monedas correspondiente a esta cantidad?

Sistemas monetarios canónicos. Para algunos sistemas monetarios, como los que usamos en la vida real, la solución "intuitiva" funciona perfectamente. Por ejemplo, si las diferentes monedas y billetes en euros (excluyendo centavos) son 1 €, 2 €, 5 €, 10 €, lo que otorga la moneda o la factura más alta hasta que alcancemos la cantidad y la repetición de este procedimiento dará lugar al conjunto mínimo de monedas. .

Podemos hacer eso recursivamente con 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
```

Estos sistemas están hechos para que el cambio sea fácil. El problema se vuelve más difícil cuando se trata de un sistema monetario arbitrario.

Caso general. ¿Cómo dar 99 € con monedas de 10 €, 7 € y 5 €? Aquí, dar monedas de 10 € hasta que nos queden con 9 € conduce obviamente a ninguna solución. Peor que una solución puede no existir. Este problema es de hecho np-hard, pero existen soluciones aceptables que mezclan la **codicia** y la **memoria** . La idea es explorar todas las posibilidades y elegir la que tenga el número mínimo de monedas.

Para dar una cantidad $X > 0$, elegimos una pieza P en el sistema monetario y luego resolvemos el subproblema correspondiente a XP . Intentamos esto para todas las piezas del sistema. La solución, si existe, es la ruta más pequeña que llevó a 0.

Aquí una función recursiva OCaml correspondiente a este método. Devuelve Ninguno, si no existe solución.

```
(* 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

```

Nota : Podemos observar que este procedimiento puede calcular varias veces el cambio establecido para el mismo valor. En la práctica, el uso de la memoria para evitar estas repeticiones conduce a resultados más rápidos (mucho más rápidos).

Problema de selección de actividad

El problema

Tienes un conjunto de cosas que hacer (actividades). Cada actividad tiene una hora de inicio y una hora de finalización. No se le permite realizar más de una actividad a la vez. Su tarea es encontrar una manera de realizar el número máximo de actividades.

Por ejemplo, suponga que tiene una selección de clases para elegir.

Actividad No.	hora de inicio	hora de finalización
1	10.20 AM	11.00AM
2	10.30 AM	11.30AM
3	11.00 AM	12.00 AM
4	10.00 AM	11.30AM
5	9.00 AM	11.00AM

Recuerda, no puedes tomar dos clases al mismo tiempo. Eso significa que no puede tomar clases 1 y 2 porque comparten un horario común de 10.30 a 11.00 AM. Sin embargo, puede tomar clases 1 y 3 porque no comparten una hora común. Por lo tanto, su tarea es tomar el máximo número de clases posible sin ninguna superposición. Como puedes hacer eso?

Análisis

Pensemos en la solución mediante un enfoque codicioso. En primer lugar, elegimos aleatoriamente algún enfoque y comprobamos que funcionará o no.

- **ordene la actividad por hora de inicio, lo** que significa que la actividad comienza primero,

la tomaremos primero. luego tome primero al último de la lista ordenada y verifique que se intersecará con la actividad tomada anteriormente o no. Si la actividad actual no se interseca con la actividad tomada anteriormente, realizaremos la actividad de lo contrario no realizaremos. Este enfoque funcionará para algunos casos como

Actividad No.	hora de inicio	hora de finalización
1	11.00 AM	13:30
2	11.30 AM	12.00 p.m.
3	1.30 PM	2:00 p.m.
4	10.00 AM	11.00AM

el orden de clasificación será 4 -> 1 -> 2 -> 3. La actividad 4 -> 1 -> 3 se realizará y la actividad 2 se omitirá. Se realizará la actividad máxima de 3. Funciona para este tipo de casos. pero fallará para algunos casos. Vamos a aplicar este enfoque para el caso.

Actividad No.	hora de inicio	hora de finalización
1	11.00 AM	13:30
2	11.30 AM	12.00 p.m.
3	1.30 PM	2:00 p.m.
4	10.00 AM	3.00PM

El orden de clasificación será 4 -> 1 -> 2 -> 3 y solo se realizará la actividad 4, pero la respuesta puede ser la actividad 1 -> 3 o 2 -> 3. Así que nuestro enfoque no funcionará para el caso anterior. Probemos otro enfoque

- **Ordene la actividad por duración de tiempo, lo** que significa realizar primero la actividad más corta. Eso puede solucionar el problema anterior. Aunque el problema no está completamente resuelto. Todavía hay algunos casos que pueden fallar la solución. Aplicar este enfoque en el caso abajo.

Actividad No.	hora de inicio	hora de finalización
1	6.00 AM	11.40AM
2	11.30 AM	12.00 p.m.
3	11.40 PM	2:00 p.m.

Si ordenamos la actividad por tiempo, el orden será 2 -> 3 ---> 1. y si realizamos la actividad No. 2 primero, entonces no se puede realizar ninguna otra actividad. Pero la respuesta será realizar la

actividad 1 y luego realizar 3. Por lo tanto, podemos realizar una actividad máxima de 2. Entonces, esto no puede ser una solución para este problema. Debemos intentar un enfoque diferente.

La solución

- **Ordene la Actividad por hora de finalización, lo** que significa que la actividad termina primero que viene primero. el algoritmo se da a continuación
 1. Ordenar las actividades por sus tiempos finales.
 2. Si la actividad a realizar no comparte un tiempo común con las actividades que se realizaron anteriormente, realice la actividad.

Analícemos el primer ejemplo.

Actividad No.	hora de inicio	hora de finalización
1	10.20 AM	11.00AM
2	10.30 AM	11.30AM
3	11.00 AM	12.00 AM
4	10.00 AM	11.30AM
5	9.00 AM	11.00AM

ordene la actividad por sus tiempos finales, así que el orden será 1 -> 5 -> 2 -> 4 -> 3 .. la respuesta es 1 -> 3 estas dos actividades se llevarán a cabo. y esa es la respuesta. Aquí está el código sudo.

1. ordenar: actividades
2. realizar la primera actividad de la lista ordenada de actividades.
3. Conjunto: Actividad_actual: = primera actividad
4. set: end_time: = end_time de la actividad actual
5. Vaya a la siguiente actividad si existe, si no existe, finalice.
6. if start_time of current activity <= end_time: realice la actividad y vaya a 4
7. otra cosa: llegó a 5.

Consulte aquí para obtener ayuda sobre la codificación <http://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/>

Lea Algoritmos codiciosos en línea: <https://riptutorial.com/es/algorithm/topic/3140/algoritmos-codiciosos>

Capítulo 17: Algoritmos en línea

Observaciones

Teoría

Definición 1: Un **problema de optimización** Π consiste en un conjunto de **instancias** Σ . Para cada instancia $\sigma \in \Sigma$ hay un conjunto Z_σ de **soluciones** y una **función objetivo** $f_\sigma : Z_\sigma \rightarrow \mathbb{R}_{\geq 0}$ que asigna un valor real positivo a cada solución.

Decimos que $\text{OPT}(\sigma)$ es el valor de una solución óptima, $A(\sigma)$ es la solución de un algoritmo A para el problema Π y $w_A(\sigma) = f_\sigma(A(\sigma))$ su valor.

Definición 2: Un algoritmo en línea A para un problema de minimización Π tiene una **relación competitiva** de $r \geq 1$ si hay una constante $\tau \in \mathbb{R}$ con

$$w_A(\sigma) = f_\sigma(A(\sigma)) \leq r \cdot \text{OPT}(\sigma) + \tau$$

para todas las instancias $\sigma \in \Sigma$. A se llama un algoritmo en línea **competitivo**. Incluso

$$w_A(\sigma) \leq r \cdot \text{OPT}(\sigma)$$

para todas las instancias $\sigma \in \Sigma$, entonces A se denomina algoritmo en línea **estrictamente competitivo**.

Proposición 1.3: **LRU** y **FWF** son algoritmo de marcado.

Prueba: al comienzo de cada fase (a excepción de la primera), **FWF** pierde el caché y lo borra. Eso significa que tenemos k páginas vacías. En cada fase se solicitan k páginas máximas diferentes, por lo que ahora habrá desalojo durante la fase. Así que **FWF** es un algoritmo de marcado.

Supongamos que **LRU** no es un algoritmo de marcado. Luego hay una instancia σ donde **LRU** una página marcada x en la fase i desalojada. Sea σ_{t^*} la solicitud en la fase i donde se desaloja x . Dado que x está marcado, debe haber una solicitud anterior σ_{t^*} para x en la misma fase, entonces $t^* < t$. Después de que σ_{t^*} es la página más nueva de las memorias caché, para que se desaloje en t la secuencia $\sigma_{t^*+1}, \dots, \sigma_t$ debe solicitar al menos k de x páginas diferentes. Eso implica que la fase i ha solicitado al menos $k+1$ páginas diferentes, lo que es contradictorio con la definición de la fase. Así que **LRU** tiene que ser un algoritmo de marcado.

Proposición 1.4: Cada algoritmo de marcado es **estrictamente competitivo**.

Prueba: Sea σ una instancia para el problema de paginación y l el número de fases para σ . Es $l = 1$, entonces, cada algoritmo de marcado es óptimo y el algoritmo fuera de línea óptimo no puede ser mejor.

Suponemos que $l \geq 2$. el costo de cada algoritmo de marcado, por ejemplo σ , se limita desde arriba con $l \cdot k$ porque en cada fase un algoritmo de marcado no puede expulsar más de k páginas sin

desalojar una página marcada.

Ahora intentamos mostrar que el algoritmo fuera de línea óptimo desaloja al menos $k + l - 2$ páginas para σ , k en la primera fase y al menos una para cada fase siguiente, excepto la última. Para la prueba vamos a definir $l - 2$ subsecuencias disyuntivas de σ . La subsecuencia $i \in \{1, \dots, l - 2\}$ comienza en la segunda posición de la fase $i + 1$ y termina con la primera posición de la fase $i + 2$.

Sea x la primera página de la fase $i + 1$. Al comienzo de la subsecuencia i hay una página x y, como máximo, $k - 1$ páginas diferentes en el caché de algoritmos fuera de línea óptimo. En la subsecuencia, la solicitud de la página k es diferente de x , por lo que el algoritmo fuera de línea óptimo debe desalojar al menos una página para cada subsecuencia. Dado que en el inicio de la fase 1, el caché aún está vacío, el algoritmo óptimo fuera de línea causa k desalojos durante la primera fase. Eso demuestra que

$$w_A(\sigma) \leq l \cdot k \leq (k + l - 2) k \leq \text{OPT}(\sigma) \cdot k$$

Corolario 1.5: LRU y FWF son estrictamente competitivos .

¿No hay una r constante para la cual un algoritmo en línea A es r -competitivo, llamamos A **no competitivo** ?

Proposición 1.6: LFU y LIFO no son competitivos .

Prueba: Deje $l \geq 2$ una constante, $k \geq 2$ el tamaño del caché. Las diferentes páginas de caché están nubladas $1, \dots, k + 1$. Nos fijamos en la siguiente secuencia:

La primera página 1 se solicita más veces que la página 2 y por lo tanto una. Al final hay $(l - 1)$ solicitudes alternas para las páginas k y $k + 1$.

LFU y **LIFO** llenan su caché con las páginas $1 - k$. Cuando se solicita la página $k + 1$, la página k se desaloja y viceversa. Eso significa que cada solicitud de subsecuencia $(k, k + 1)^{l - 1}$ desaloja una página. Además, su caché $k - 1$ se pierde por primera vez en el uso de las páginas $1 - (k - 1)$. Así que **LFU** y **LIFO** desalojan páginas exactas $k - 1 + 2(l - 1)$.

Ahora debemos mostrar que para cada constante $\tau \in \mathbb{R}$ y cada constante $r \leq 1$ existe una l para que

que es igual a

Para satisfacer esta desigualdad solo tienes que elegir lo suficientemente grande. Así que **LFU** y **LIFO** no son competitivos.

Proposición 1.7: No hay un algoritmo en línea determinista r -competetive para paginar con $r < k$

Fuentes

Material básico

1. Script Online Algorithms (alemán), Heiko Roeglin, Universidad de Bonn
2. [Algoritmo de reemplazo de página](#)

Otras lecturas

1. [Computación en línea y análisis competitivo](#) por Allan Borodin y Ran El-Yaniv

Código fuente

1. Código fuente para el [almacenamiento en caché sin conexión](#)
2. Código fuente para [juego adversario](#)

Examples

Paginación (almacenamiento en caché en línea)

Prefacio

En lugar de comenzar con una definición formal, el objetivo es abordar este tema a través de una fila de ejemplos, introduciendo definiciones en el camino. La sección de comentarios La **teoría** consistirá en todas las definiciones, teoremas y proposiciones para proporcionarle toda la información para buscar más rápidamente aspectos específicos.

Las fuentes de la sección de comentarios consisten en el material de base utilizado para este tema e información adicional para lecturas adicionales. Además, encontrará los códigos fuente completos para los ejemplos allí. Preste atención a que para que el código fuente de los ejemplos sea más legible y más corto, se abstenga de cosas como el manejo de errores, etc. También transmite algunas características específicas del lenguaje que podrían ocultar la claridad del ejemplo, como el uso extensivo de bibliotecas avanzadas, etc.

Paginacion

El problema de la paginación surge de la limitación del espacio finito. Supongamos que nuestro caché c tiene k páginas. Ahora queremos procesar una secuencia de m solicitudes de página que se deben haber colocado en el caché antes de que se procesen. Por supuesto, si $m \leq k$, simplemente colocamos todos los elementos en el caché y funcionará, pero generalmente es $m \gg k$

Decimos que una solicitud es un **acierto de caché**, cuando la página ya está en caché, de lo

contrario, se denomina **falta de caché** . En ese caso, debemos traer la página solicitada al caché y desalojar a otro, suponiendo que el caché esté lleno. El objetivo es un programa de desalojo que **minimiza el número de desalojos** .

Existen numerosas estrategias para este problema, veamos algunas:

1. **Primero en entrar, primero en salir (FIFO)** : la página más antigua se desaloja
2. **Último en entrar, primero en salir (LIFO)** : la página más nueva se desaloja
3. **Usado menos recientemente (LRU)** : página de desalojo cuyo acceso más reciente fue el más antiguo
4. **Uso menos frecuente (LFU)** : página de desalojo solicitada con menor frecuencia
5. **Distancia de avance más larga (LFD)** : Desaloja la página en el caché que no se solicita hasta el momento más lejano.
6. **Vaciar cuando esté lleno (FWF)** : borre la memoria caché completa tan pronto como se produzca una falla de caché

Hay dos formas de abordar este problema:

1. **fuera de línea** : la secuencia de solicitudes de página se conoce de antemano
2. **en línea** : la secuencia de solicitudes de página no se conoce de antemano

Enfoque sin conexión

Para el primer enfoque mira el tema [Aplicaciones de la técnica codiciosa](#) . Es el tercer ejemplo. El **almacenamiento en línea sin conexión** considera las primeras cinco estrategias desde arriba y le brinda un buen punto de entrada para lo siguiente.

El programa de ejemplo se amplió con la estrategia **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;
}
}
};

```

El código fuente completo está disponible [aquí](#) . Si reutilizamos el ejemplo del tema, obtenemos el siguiente resultado:

```

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

```

Aunque **LFD** es óptimo, **FWF** tiene menos errores de caché. Pero el objetivo principal era minimizar el número de desalojos y para **FWF** cinco fallos significan 15 desalojos, lo que la convierte en la opción más pobre para este ejemplo.

Enfoque en línea

Ahora queremos abordar el problema en línea de la paginación. Pero primero necesitamos un entendimiento de cómo hacerlo. Obviamente, un algoritmo en línea no puede ser mejor que el algoritmo sin conexión óptimo. ¿Pero cuánto peor es? Necesitamos definiciones formales para responder a esa pregunta:

Definición 1.1: Un **problema de optimización** Π consiste en un conjunto de **instancias** Σ_{Π} . Para cada instancia $\sigma \in \Sigma_{\Pi}$ hay un conjunto Z_{σ} de **soluciones** y una **función objetivo** $f_{\sigma} : Z_{\sigma} \rightarrow \mathbb{R}_{\geq 0}$ que asigna un valor real positivo a cada solución.

Decimos que $\text{OPT}(\sigma)$ es el valor de una solución óptima, $A(\sigma)$ es la solución de un algoritmo A para el problema Π y $w_A(\sigma) = f_{\sigma}(A(\sigma))$ su valor.

Definición 1.2: Un algoritmo en línea A para un problema de minimización Π tiene una **relación competitiva** de $r \geq 1$ si hay una constante $\tau \in \mathbb{R}$ con

$$w_A(\sigma) = f_\sigma(A(\sigma)) \leq r \cdot \text{OPT}(\sigma) + \tau$$

para todas las instancias $\sigma \in \Sigma_\Pi$. A se llama un algoritmo en línea **competitivo**. Incluso

$$w_A(\sigma) \leq r \cdot \text{OPT}(\sigma)$$

para todas las instancias $\sigma \in \Sigma_\Pi$, entonces A se denomina algoritmo en línea **estrictamente competitivo**.

Entonces, la pregunta es cuán **competitivo** es nuestro algoritmo en línea en comparación con un algoritmo sin conexión óptimo. En su famoso [libro](#) Allan Borodin y Ran El-Yaniv usaron otro escenario para describir la situación de la paginación en línea:

Hay un **adversario malvado** que conoce su algoritmo y el algoritmo óptimo fuera de línea. En cada paso, intenta solicitar una página que sea peor para usted y, al mismo tiempo, mejor para el algoritmo sin conexión. el **factor competitivo** de su algoritmo es el factor de qué tan mal lo hizo su algoritmo contra el algoritmo fuera de línea óptimo del adversario. Si quieres tratar de ser el adversario, puedes probar el [Juego Adversario](#) (intenta superar las estrategias de paginación).

Algoritmos de mercado

En lugar de analizar cada algoritmo por separado, veamos una familia especial de algoritmos en línea para el problema de paginación llamado **algoritmos de mercado**.

Deje que $\sigma = (\sigma_1, \dots, \sigma_p)$ sea una instancia de nuestro problema y k nuestro tamaño de caché, ya que σ se puede dividir en fases:

- La fase 1 es la subsecuencia máxima de σ desde el inicio hasta el máximo k se solicitan diferentes páginas
- La fase $i \geq 2$ es la subsecuencia máxima de σ desde el final del pase $i-1$ hasta que se solicitan k páginas máximas diferentes.

Por ejemplo con $k = 3$:

Un algoritmo de mercado (implícito o explícito) mantiene si una página está marcada o no. Al comienzo de cada fase se encuentran todas las páginas sin marcar. Es una página solicitada durante una fase que se marca. Un algoritmo es un algoritmo de mercado **si** nunca desaloja una página marcada de la caché. Eso significa que las páginas que se utilizan durante una fase no serán desalojadas.

Proposición 1.3: LRU y FWF son algoritmo de marcado.

Prueba: al comienzo de cada fase (a excepción de la primera), **FWF** pierde el caché y lo borra. Eso significa que tenemos k páginas vacías. En cada fase se solicitan k páginas máximas diferentes, por lo que ahora habrá desalojo durante la fase. Así que **FWF** es un algoritmo de marcado.

Supongamos que **LRU** no es un algoritmo de marcado. Luego hay una instancia σ donde **LRU** una página marcada x en la fase i desalojada. Sea σ_t la solicitud en la fase i donde se desaloja x . Dado que x está marcado, debe haber una solicitud anterior σ_{t^*} para x en la misma fase, entonces $t^* < t$. Después de que $t^* x$ es la página más nueva de las memorias caché, para que se desaloje en t la secuencia $\sigma_{t^*+1}, \dots, \sigma_t$ debe solicitar al menos k de x páginas diferentes. Eso implica que la fase i ha solicitado al menos $k + 1$ páginas diferentes, lo que es contradictorio con la definición de la fase. Así que **LRU** tiene que ser un algoritmo de marcado.

Proposición 1.4: Cada algoritmo de marcado es estrictamente competitivo .

Prueba: Sea σ una instancia para el problema de paginación y l el número de fases para σ . Es $l = 1$, entonces, cada algoritmo de marcado es óptimo y el algoritmo fuera de línea óptimo no puede ser mejor.

Suponemos que $l \geq 2$. el costo de cada algoritmo de marcado, por ejemplo, σ está limitado desde arriba con $l \cdot k$ porque en cada fase un algoritmo de marcado no puede expulsar más de k páginas sin desalojar una página marcada.

Ahora intentamos mostrar que el algoritmo fuera de línea óptimo desaloja al menos $k + l - 2$ páginas para σ , k en la primera fase y al menos una para cada fase siguiente, excepto la última. Para la prueba vamos a definir $l - 2$ subsecuencias disyuntivas de σ . La subsecuencia $i \in \{1, \dots, l - 2\}$ comienza en la segunda posición de la fase $i + 1$ y termina con la primera posición de la fase $i + 2$.

Sea x la primera página de la fase $i + 1$. Al comienzo de la subsecuencia i hay una página x y, como máximo, $k - 1$ páginas diferentes en el caché de algoritmos fuera de línea óptimo. En la subsecuencia, la solicitud de la página k es diferente de x , por lo que el algoritmo fuera de línea óptimo debe desalojar al menos una página para cada subsecuencia. Dado que en el inicio de la fase 1, el caché aún está vacío, el algoritmo óptimo fuera de línea causa k desalojos durante la primera fase. Eso demuestra que

$$w_A(\sigma) \leq l \cdot k \leq (k + l - 2) k \leq \text{OPT}(\sigma) \cdot k$$

Corolario 1.5: LRU y FWF son estrictamente competitivos .

Ejercicio: Mostrar que **FIFO** no es un algoritmo de marcado, sino estrictamente **k-competitivo** .

¿No hay una r constante para la cual un algoritmo en línea A es r -competitivo, llamamos A **no competitivo**?

Proposición 1.6: LFU y LIFO no son competitivos .

Prueba: Deje $l \geq 2$ una constante, $k \geq 2$ el tamaño del caché. Las diferentes páginas de caché están nublasadas $1, \dots, k + 1$. Nos fijamos en la siguiente secuencia:

La primera página 1 se solicita más veces que la página 2, y por lo tanto una. Al final, hay $(l-1)$ solicitudes alternas para las páginas k y $k + 1$.

LFU y **LIFO** llenan su caché con las páginas $1-k$. Cuando se solicita la página $k + 1$, la página k se desaloja y viceversa. Eso significa que cada solicitud de subsecuencia $(k, k + 1)^{l-1}$ desaloja una página. Además, su caché de $k-1$ se pierde por primera vez en el uso de las páginas $1-(k-1)$. Así que **LFU** y **LIFO** desalojan páginas exactas $k-1 + 2(l-1)$.

Ahora debemos mostrar que para cada constante $\tau \in \mathbb{R}$ y cada constante $r \leq 1$ existe una l para que

que es igual a

Para satisfacer esta desigualdad solo tienes que elegir lo suficientemente grande. Entonces, **LFU** y **LIFO** no son competitivos.

Proposición 1.7: No hay un algoritmo en línea determinista **r-competitive** para paginar con $r < k$.

La prueba de esta última proposición es bastante larga y se basa en la afirmación de que **LFD** es un algoritmo sin conexión óptimo. El lector interesado puede buscarlo en el libro de Borodin y El-Yaniv (consulte las fuentes a continuación).

La pregunta es si podríamos hacerlo mejor. Para eso, tenemos que dejar atrás el enfoque determinista y comenzar a aleatorizar nuestro algoritmo. Claramente, es mucho más difícil para el adversario castigar tu algoritmo si es aleatorio.

La paginación aleatoria se tratará en uno de los siguientes ejemplos ...

Lea Algoritmos en línea en línea: <https://riptutorial.com/es/algorithm/topic/8022/algoritmos-en-linea>

Capítulo 18: Algoritmos multihilo

Introducción

Ejemplos para algunos algoritmos multiproceso.

Sintaxis

- **paralelo** antes de un bucle significa que cada iteración del bucle son independientes entre sí y se pueden ejecutar en paralelo.
- **spawn** es para indicar la creación de un nuevo hilo.
- **sincronizar** es sincronizar todos los hilos creados.
- Las matrices / matrices se indexan de 1 a n en los ejemplos.

Examples

Multiplexación de matriz cuadrada multihilo

```
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
```

Matriz de multiplicación vector multihilo

```
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
```

fusionar y ordenar multiproceso

A es una matriz y los índices p y q de la matriz, como por ejemplo, la ordenación de la sub-matriz $A [p..r]$. B es una sub-matriz que será poblada por la ordenación.

Una llamada a p -merge-sort (A, p, r, B, s) ordena los elementos de $A [p..r]$ y los pone en $B [s..s + rp]$.

```

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)

```

Aquí está la función auxiliar que realiza la fusión en paralelo.

p-merge asume que las dos subarreglas para fusionar están en la misma matriz pero no asume que son adyacentes a la matriz. Es por eso que necesitamos *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

```

Y aquí está la función auxiliar de búsqueda dicotómica.

x es la clave a buscar en la sub-matriz 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

```

Lea Algoritmos multihilo en línea: <https://riptutorial.com/es/algorithm/topic/9965/algoritmos-multihilo>

Capítulo 19: Aplicaciones de la técnica codiciosa.

Observaciones

Fuentes

1. Los ejemplos anteriores son de apuntes de una conferencia que se impartió en 2008 en Bonn, Alemania. En términos generales, se basan en el libro [Algorithm Design](#) de Jon Kleinberg y Eva Tardos:

Examples

Ticket automático

Primer ejemplo simple:

Usted tiene un ticket automático que ofrece el intercambio en monedas con valores 1, 2, 5, 10 y 20. La disposición del cambio se puede ver como una serie de monedas que caen hasta que se entrega el valor correcto. Decimos que una dispensación es **óptima** cuando su **cuenta de monedas es mínima** por su valor.

Sea M en $[1, 50]$ el precio del boleto T y P en $[1, 50]$ el dinero que alguien pagó por T , con $P \geq M$. Sea $D = P - M$. Definimos el **beneficio** de un paso como la diferencia entre D y D_c con c la moneda que el autómata dispensa en este paso.

La **técnica codiciosa** para el intercambio es el siguiente enfoque pseudo algorítmico:

Paso 1: mientras que $D > 20$ dispensa una moneda de 20 y establece $D = D - 20$

Paso 2: mientras que $D > 10$ dispensa una moneda de 10 y establece $D = D - 10$

Paso 3: mientras que $D > 5$ dispensa una moneda de 5 y establece $D = D - 5$

Paso 4: mientras que $D > 2$ dispensa una moneda de 2 y establece $D = D - 2$

Paso 5: mientras que $D > 1$ dispensa una moneda 1 y establece $D = D - 1$

Después, la suma de todas las monedas es claramente D . Es un **algoritmo codicioso** porque después de cada paso y después de cada repetición de un paso, el beneficio se maximiza. No podemos dispensar otra moneda con un beneficio mayor.

Ahora el ticket automático como programa (en C ++):

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;
```

```

// read some coin values, sort them descending,
// purge copies and guarantee 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;
}

```

Tenga en cuenta que ahora hay verificación de entrada para mantener el ejemplo simple. Un ejemplo de salida:

```

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

```

Mientras d esté en los valores de la moneda, ahora, el algoritmo terminará, porque:

- d disminuye estrictamente con cada paso.
- d nunca es >0 y más pequeño que la moneda más pequeña 1 al mismo tiempo

Pero el algoritmo tiene dos trampas:

1. Sea c el mayor valor de la moneda. El tiempo de ejecución es solo polinomial mientras D/c sea polinomial, porque la representación de D utiliza solo los bits de $\log D$ y el tiempo de ejecución es al menos lineal en D/c
2. En cada paso nuestro algoritmo elige el óptimo local. Pero esto no es suficiente para decir que el algoritmo encuentra la solución óptima global (consulte más información [aquí](#) o en el Libro de [Korte y Vygen](#)).

Un simple ejemplo de contador: las monedas son $1, 3, 4$ y $D=6$. La solución óptima es claramente dos monedas de valor 3 pero codiciosa elige 4 en el primer paso, por lo que debe elegir 1 en el paso dos y tres. Así que no da una solución óptima. Un posible algoritmo óptimo para este ejemplo se basa en **la programación dinámica** .

Programación de intervalos

Tenemos un conjunto de trabajos $J=\{a,b,c,d,e,f,g\}$. Sea $j \in J$ un trabajo que su comienzo en s_j y termina en f_j . Dos trabajos son compatibles si no se superponen. Una imagen como ejemplo: El objetivo es encontrar el **subconjunto máximo de trabajos compatibles entre sí** . Hay varios enfoques codiciosos para este problema:

1. **Hora de inicio más temprana** : considerar trabajos en orden ascendente de s_j
2. **Tiempo de finalización más temprano** : considerar trabajos en orden ascendente de f_j
3. **Intervalo más corto** : considere trabajos en orden ascendente de f_j-s_j
4. **Menos conflictos** : para cada trabajo j , cuente el número de trabajos en conflicto c_j

La pregunta ahora es, qué enfoque es realmente exitoso. **Hora de inicio** anticipado definitivamente no, aquí hay un ejemplo contrario. **El intervalo más corto** tampoco es óptimo y el **menor número de conflictos** puede parecer realmente óptimo, pero aquí hay un caso de problema para este enfoque: Lo que nos deja con el **tiempo de llegada más temprano** . El pseudo código es bastante simple:

1. Ordena los trabajos por tiempo de finalización para que $f_1 \leq f_2 \leq \dots \leq f_n$
2. Sea A un conjunto vacío.
3. para $j=1$ a n si j es compatible con **todos los** trabajos en A set $A=A+\{j\}$
4. A es un **subconjunto máximo de trabajos compatibles entre sí**.

O como programa 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;
}

```

La salida para este ejemplo es: Compatible: (1,3) (4,5) (6,8) (9,10)

La implementación del algoritmo está claramente en $\Theta(n^2)$. Hay una implementación de $\Theta(n \log n)$ y el lector interesado puede continuar leyendo a continuación (Ejemplo de Java).

Ahora tenemos un algoritmo codicioso para el problema de la programación de intervalos, pero ¿es óptimo?

Proposición: El algoritmo codicioso, el **tiempo de llegada más temprano**, es óptimo.

Prueba: (por contradicción)

Supongamos que la codicia no es óptima y i_1, i_2, \dots, i_k denota el conjunto de trabajos seleccionados por codicia. Dejemos que j_1, j_2, \dots, j_m denote el conjunto de trabajos en una solución **óptima** con $i_1=j_1, i_2=j_2, \dots, i_r=j_r$ para el **mayor valor posible** de r .

El trabajo $i_{(r+1)}$ existe y finaliza antes de $j_{(r+1)}$ (primer final). Pero que es $j_1, j_2, \dots, j_r, i_{(r+1)}, j_{(r+2)}, \dots, j_m$ también es una solución **óptima** y para todo k en $[1, (r+1)]$ es $j_k=i_k$. eso es una **contradicción** a la maximalidad de r . Esto concluye la prueba.

Este segundo ejemplo demuestra que usualmente hay muchas estrategias codiciosas posibles pero solo algunas o incluso ninguna puede encontrar la solución óptima en cada instancia.

A continuación se muestra un programa Java que se ejecuta en $\Theta(n \log n)$

```
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));
}
}

```

Y la salida esperada es:

```
Optimal profit is 250
```

Minimizando la latitud

Existen numerosos problemas para minimizar la tardanza, aquí tenemos un recurso único que solo puede procesar un trabajo a la vez. El trabajo j requiere t_j unidades de tiempo de procesamiento y se debe entregar en el momento d_j . si j comienza en el tiempo s_j , terminará en el tiempo $f_j = s_j + t_j$. Definimos la tardanza $L = \max\{0, f_j - d_j\}$ para todos j . El objetivo es minimizar la **tardanza máxima L**.

	1	2	3	4	5	6
t_j	3	2	1	4	3	2

	1	2	3	4	5	6									
d _j	6	8	9	9	10	11									
Trabajo	3	2	2	5	5	5	4	4	4	4	1	1	1	6	6
Hora	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L _j		-8		-5						1				7	4

La solución $L=7$ obviamente no es óptima. Veamos algunas estrategias codiciosas:

1. **Tiempo de procesamiento más corto primero** : programe trabajos en orden ascendente og tiempo de procesamiento j
2. **Primera fecha límite más temprana** : programe los trabajos en orden ascendente de la fecha límite d_j
3. **Menor holgura** : programe trabajos en orden ascendente de holgura $d_j - t_j$

Es fácil ver que el **tiempo de procesamiento más corto primero** no es óptimo, un buen ejemplo de contador es

	1	2
t _j	1	5
d _j	10	5

La solución de **pila más pequeña** tiene problemas similares.

	1	2
t _j	1	5
d _j	3	5

La última estrategia parece válida, así que comenzamos con un pseudo código:

1. Ordene n trabajos a su debido tiempo para que $d_1 \leq d_2 \leq \dots \leq d_n$
2. Establecer $t=0$
3. para $j=1$ a n
 - Asignar trabajo j a intervalo $[t, t+t_j]$
 - establece $s_j=t$ y $f_j=t+t_j$
 - establecer $t=t+t_j$
4. intervalos de retorno $[s_1, f_1], [s_2, f_2], \dots, [s_n, f_n]$

Y como implementación en 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;
}

```

Y la salida para este programa es:

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

El tiempo de ejecución del algoritmo es obviamente $\Theta(n \log n)$ porque la ordenación es la operación dominante de este algoritmo. Ahora tenemos que demostrar que es óptimo. Claramente un horario óptimo no tiene **tiempo de inactividad**. el **primer** horario del **primer plazo** tampoco tiene tiempo de inactividad.

Supongamos que los trabajos están numerados de modo que $d_1 \leq d_2 \leq \dots \leq d_n$. Decimos que una **inversión** de un cronograma es un par de trabajos i y j para que $i < j$ pero j esté programado antes de i . Debido a su definición, el **primer** calendario de la **primera fecha límite** no tiene inversiones. Por supuesto, si un programa tiene una inversión, tiene uno con un par de trabajos invertidos programados consecutivamente.

Proposición: El intercambio de dos trabajos invertidos adyacentes reduce el número de inversiones en **uno** y **no aumenta** la tardanza máxima.

Prueba: Sea L la tardanza antes del cambio y M la tardanza después. Debido a que el intercambio de dos trabajos adyacentes no mueve los otros trabajos de su posición, es $L_k = M_k$ para todo $k \neq i, j$.

Está claro que es $M_i \leq L_i$, porque la petición i he programado anteriormente. Si el trabajo j se retrasa, entonces se sigue de la definición:

$$\begin{aligned} M_j &= f_i - d_j && \text{(definition)} \\ &\leq f_i - d_i && \text{(since } i \text{ and } j \text{ are exchanged)} \\ &\leq L_i \end{aligned}$$

Eso significa que la tardanza después del intercambio es menor o igual que antes. Esto concluye la prueba.

Proposición: El **primer** horario de la **primera fecha límite** s es óptimo.

Prueba: (por contradicción)

Supongamos que s^* es un programa óptimo con el **menor** número **posible** de inversiones. Podemos asumir que s^* no tiene tiempo de inactividad. Si s^* no tiene inversiones, entonces $s = s^*$ y hemos terminado. Si s^* tiene una inversión, entonces tiene una inversión adyacente. La última

Proposición declara que podemos intercambiar la inversión adyacente sin aumentar el retraso, pero disminuyendo el número de inversiones. Esto contradice la definición de s^* .

El problema de la minimización de la demora y su problema de **makepan mínimo** relacionado cercano, donde se hace la pregunta para un horario mínimo, tiene muchas aplicaciones en el mundo real. Pero, por lo general, no tiene una sola máquina, sino muchas, y manejan la misma tarea a diferentes velocidades. Estos problemas consiguen NP-completo muy rápido.

Otra pregunta interesante surge si no observamos el problema **fuera de línea**, donde tenemos todas las tareas y los datos a la mano, pero en la variante en **línea**, donde aparecen las tareas durante la ejecución.

Offline Caching

El problema del almacenamiento en caché surge de la limitación del espacio finito. Supongamos que nuestro caché c tiene k páginas. Ahora queremos procesar una secuencia de m solicitudes de elementos que deben haber sido colocadas en el caché antes de que sean procesadas. Por supuesto, si $m \leq k$, simplemente colocamos todos los elementos en el caché y funcionará, pero generalmente es $m \gg k$.

Decimos que una solicitud es un **acierto de caché**, cuando el elemento ya está en caché, de lo contrario se denomina **falta de caché**. En ese caso, debemos traer el elemento solicitado a la memoria caché y desalojar a otro, asumiendo que la memoria caché está llena. El objetivo es un programa de desalojo que **minimiza el número de desalojos**.

Existen numerosas estrategias codiciosas para este problema, veamos algunas:

1. **Primero en entrar, primero en salir (FIFO)**: la página más antigua se desaloja
2. **Último en entrar, primero en salir (LIFO)**: la página más nueva se desaloja
3. **Última salida reciente (LRU)**: página de desalojo cuyo acceso más reciente fue el más antiguo
4. **Solicitud menos frecuente (LFU)**: página de desalojo que se solicitó con menor frecuencia
5. **Distancia de avance más larga (LFD)**: Desaloja la página en el caché que no se solicita hasta el momento más lejano.

Atención: En los siguientes ejemplos, desalojamos la página con el índice más pequeño, si se puede desalojar más de una página.

Ejemplo (FIFO)

Sea el tamaño del caché $k=3$ el caché inicial a, b, c y la solicitud $a, a, d, e, b, b, a, c, f, d, e, a, f, b, e, c$:

Solicitud	una	una	re	mi	segundo	segundo	una	do	F	re
cache 1	una	una	re	re	re	re	una	una	una	re
cache 2	segundo	segundo	segundo	mi	mi	mi	mi	do	do	do

Solicitud	una	una	re	mi	segundo	segundo	una	do	F	re
cache 3	do	do	do	do	segundo	segundo	segundo	segundo	F	F
señorita caché			X	X	X		X	X	X	X

Trece fallos de caché por dieciséis solicitudes no suenan muy óptimos, probemos el mismo ejemplo con otra estrategia:

Ejemplo (LFD)

Sea el tamaño del caché $k=3$ el caché inicial a, b, c y la solicitud $a, a, d, e, b, b, a, c, f, d, e, a, f, b, e, c$:

Solicitud	una	una	re	mi	segundo	segundo	una	do	F	re
cache 1	una	una	re	mi	mi	mi	mi	mi	mi	mi
cache 2	segundo	segundo	segundo	segundo	segundo	segundo	una	una	una	una
cache 3	do	do	do	do	do	do	do	do	F	re
señorita caché			X	X			X		X	X

Ocho errores de caché es mucho mejor.

Prueba inmediata : haga el ejemplo para LIFO, LFU, RFU y vea lo que sucedió.

El siguiente programa de ejemplo (escrito en C++) consta de dos partes:

El esqueleto es una aplicación que resuelve el problema dependiendo de la estrategia codiciosa elegida:

```
#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];
}
```

La idea básica es simple: para cada solicitud tengo dos llamadas dos mi estrategia:

1. **aplicar** : la estrategia tiene que decirle a la persona que llama qué página usar
2. **actualización** : después de que la persona que llama usa el lugar, le dice a la estrategia si fue una falla o no. Entonces la estrategia puede actualizar sus datos internos. La estrategia **LFU**, por ejemplo, tiene que actualizar la frecuencia de aciertos para las páginas de caché, mientras que la estrategia **LFD** tiene que recalcular las distancias para las páginas de caché.

Ahora veamos ejemplos de implementaciones para nuestras cinco estrategias:

FIFO

```
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 solo necesita la información sobre el tiempo que una página está en el caché (y, por supuesto, solo en relación con las otras páginas). Así que lo único que hay que hacer es esperar a que se pierda y luego hacer las páginas, que no fueron desalojadas. Para nuestro ejemplo anterior, la solución del programa es:

```
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
```

Eso es exacto la solución de arriba.

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];
};

```

La implementación de **LIFO** es más o menos la misma que en **FIFO**, pero desalojamos a los más jóvenes y no a los más antiguos. Los resultados del programa son:

```

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];
};

```

En el caso de **LRU**, la estrategia es independiente de lo que está en la página de caché, su único interés es el último uso. Los resultados del programa son:

```

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

LFU

```
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 desaloja la página con menos frecuencia. Así que la estrategia de actualización es solo contar cada acceso. Por supuesto, después de una falta, la cuenta se reinicia. Los resultados del programa son:

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];
};
```

La estrategia de **LFD** es diferente de todos antes. Es la única estrategia que utiliza las futuras solicitudes para su decisión a quién desalojar. La implementación utiliza la función `calcNextUse` para obtener la página que el siguiente uso está más lejos en el futuro. La solución del programa es igual a la solución a mano desde arriba:

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

La estrategia codiciosa **LFD** es, de hecho, la única estrategia óptima de las cinco presentadas. La prueba es bastante larga y se puede encontrar [aquí](#) o en el libro de Jon Kleinberg y Eva Tardos (consulte las fuentes en los comentarios a continuación).

Algoritmo vs Realidad

La estrategia de **LFD** es óptima, pero hay un gran problema. Es una solución óptima **fuera de línea**. En la práctica, el almacenamiento en caché suele ser un problema en **línea**, lo que significa que la estrategia es inútil porque no podemos ahora la próxima vez que necesitemos un elemento en particular. Las otras cuatro estrategias son también estrategias en **línea**. Para problemas en línea necesitamos un enfoque general diferente.

Lea [Aplicaciones de la técnica codiciosa. en línea](#):

<https://riptutorial.com/es/algorithm/topic/7993/aplicaciones-de-la-tecnica-codiciosa->

Capítulo 20: Aplicaciones de Programación Dinámica.

Introducción

La idea básica detrás de la programación dinámica es dividir un problema complejo en varios problemas pequeños y simples que se repiten. Si puede identificar un subproblema simple que se calcula repetidamente, es probable que exista un enfoque de programación dinámica para el problema.

Como este tema se titula *Aplicaciones de programación dinámica*, se centrará más en las aplicaciones que en el proceso de creación de algoritmos de programación dinámica.

Observaciones

Definiciones

Memorización: una técnica de optimización utilizada principalmente para acelerar los programas informáticos al almacenar los resultados de llamadas a funciones costosas y devolver el resultado almacenado en caché cuando vuelven a ocurrir las mismas entradas.

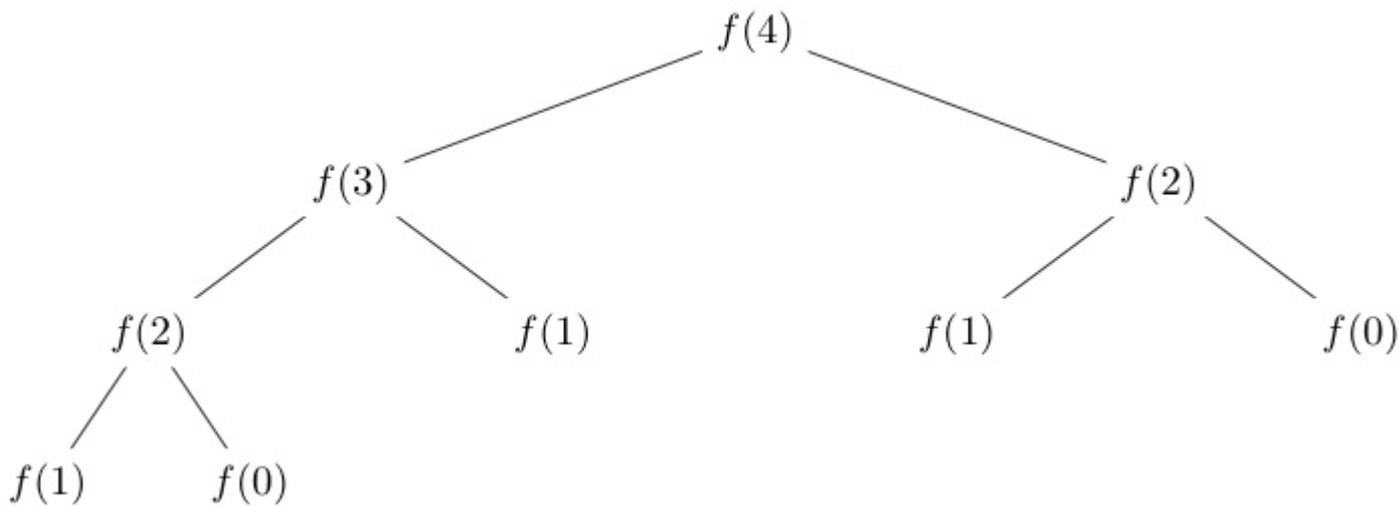
Programación dinámica: un método para resolver un problema complejo dividiéndolo en una colección de subproblemas más simples, resolviendo cada uno de esos subproblemas solo una vez y almacenando sus soluciones.

Examples

Números de Fibonacci

Los **números de Fibonacci** son un tema primordial para la programación dinámica, ya que el enfoque recursivo tradicional realiza muchos cálculos repetidos. En estos ejemplos usaré el caso base de $f(0) = f(1) = 1$.

Aquí hay un ejemplo de árbol recursivo para `fibonacci(4)`, note los cálculos repetidos:



Programación no dinámica $O(2^n)$ Complejidad en tiempo de ejecución, $O(n)$ Complejidad de pila

```
def fibonacci(n):
    if n < 2:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

Esta es la forma más intuitiva de escribir el problema. A lo sumo, el espacio de la pila será $O(n)$ medida que descienda la primera rama recursiva haciendo llamadas a `fibonacci(n-1)` hasta que llegue al caso base $n < 2$.

La $O(2^n)$ prueba de complejidad en tiempo de ejecución que se puede ver aquí: [Complejidad computacional de la secuencia de Fibonacci](#). El punto principal a tener en cuenta es que el tiempo de ejecución es exponencial, lo que significa que el tiempo de ejecución para esto se duplicará para cada término subsiguiente, `fibonacci(15)` tomará el doble de tiempo que `fibonacci(14)`.

Memoized $O(n)$ Complejidad de tiempo de ejecución, $O(n)$ Complejidad de espacio, $O(n)$ Complejidad de pila

```
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
```

Con el enfoque memorizado, introducimos una matriz que puede considerarse como todas las llamadas de función anteriores. La `memo[n]` ubicación `memo[n]` es el resultado de la función llamada

`fibonacci(n)` . Esto nos permite intercambiar la complejidad de espacio de $O(n)$ por un tiempo de ejecución $O(n)$ ya que ya no necesitamos calcular llamadas de función duplicadas.

Programación dinámica iterativa $O(n)$ Complejidad en tiempo de ejecución, $O(n)$ Complejidad de espacio, No pila recursiva

```
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]
```

Si dividimos el problema en sus elementos centrales, notará que para calcular `fibonacci(n)` necesitamos `fibonacci(n-1)` y `fibonacci(n-2)` . También podemos notar que nuestro caso base aparecerá al final de ese árbol recursivo como se ve arriba.

Con esta información, ahora tiene sentido calcular la solución hacia atrás, comenzando en los casos base y trabajando hacia arriba. Ahora, para calcular la `fibonacci(n)` , primero calculamos **todos** los números de fibonacci hasta `n` .

Este beneficio principal aquí es que ahora hemos eliminado la pila recursiva mientras mantenemos el tiempo de ejecución $O(n)$. Desafortunadamente, todavía tenemos una complejidad de espacio $O(n)$, pero eso también se puede cambiar.

Programación dinámica iterativa avanzada $O(1)$ Complejidad en tiempo de ejecución, $O(1)$ Complejidad de espacio, No pila recursiva

```
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]
```

Como se señaló anteriormente, el enfoque de programación dinámica iterativa comienza desde los casos base y funciona hasta el resultado final. La observación clave a realizar para llegar a la complejidad del espacio a $O(1)$ (constante) es la misma observación que hicimos para la pila recursiva: solo necesitamos `fibonacci(n-1)` y `fibonacci(n-2)` para construir `fibonacci(n)` . Esto significa que solo necesitamos guardar los resultados de `fibonacci(n-1)` y `fibonacci(n-2)` en cualquier punto de nuestra iteración.

Para almacenar estos últimos 2 resultados, utilizo una matriz de tamaño 2 y simplemente invierto el índice al que estoy asignando utilizando `i % 2` que alternará así: 0, 1, 0, 1, 0, 1, ..., `i % 2` .

Agrego ambos índices de la matriz juntos porque sabemos que la adición es conmutativa (`5 + 6 = 11` y `6 + 5 == 11`). El resultado se asigna al más antiguo de los dos puntos (indicado por `i % 2`). El

resultado final se almacena en la posición $n\%2$

Notas

- Es importante tener en cuenta que a veces puede ser mejor encontrar una solución memorizada iterativa para funciones que realizan cálculos grandes repetidamente, ya que acumulará un caché de la respuesta a las llamadas de función y las llamadas subsiguientes pueden ser $O(1)$ si ya ha sido computado

Lea [Aplicaciones de Programación Dinámica](#). en línea:

<https://riptutorial.com/es/algorithm/topic/10596/aplicaciones-de-programacion-dinamica->

Capítulo 21: Árboles

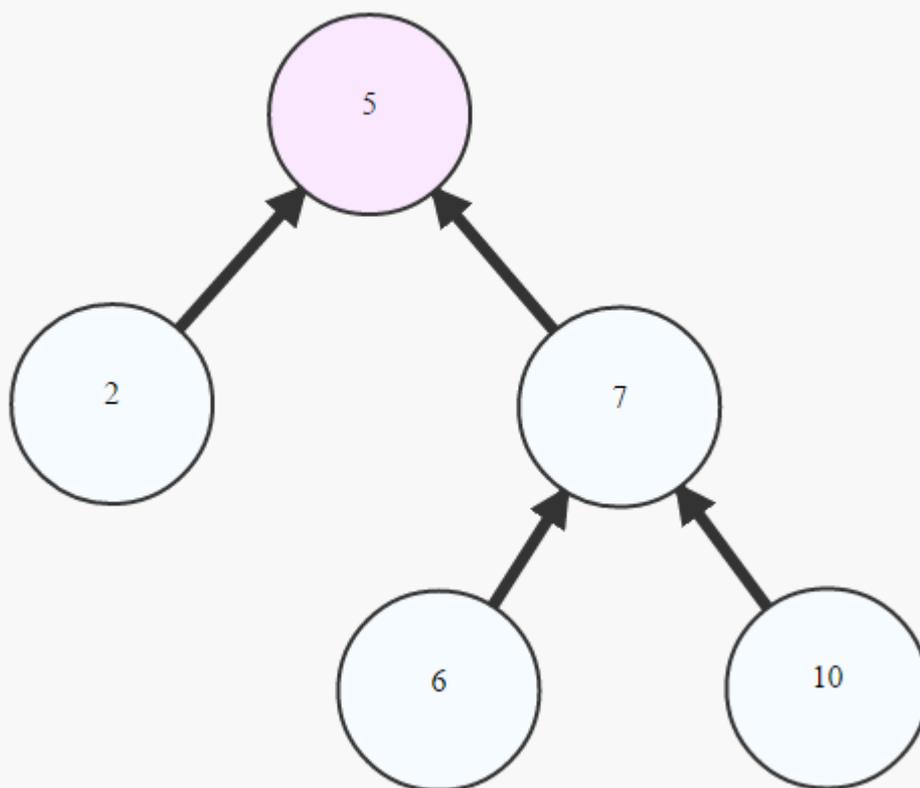
Observaciones

Los árboles son una subcategoría o subtipo de gráficos de borde de nodo. Son omnipresentes dentro de la informática debido a su prevalencia como modelo para muchas estructuras algorítmicas diferentes que, a su vez, se aplican en muchos algoritmos diferentes.

Examples

Introducción

Los árboles son un subtipo de la estructura de datos más general del gráfico de borde de nodo.



Para ser un árbol, una gráfica debe satisfacer dos requisitos:

- **Es acíclico.** No contiene ciclos (o "bucles").
- **Esta conectado.** Para cualquier nodo dado en el gráfico, cada nodo es accesible. Todos los nodos son accesibles a través de una ruta en el gráfico.

La estructura de datos del árbol es bastante común dentro de la informática. Los árboles se

utilizan para modelar muchas estructuras de datos algorítmicos diferentes, como árboles binarios ordinarios, árboles rojo-negros, árboles B, árboles AB, árboles 23, Heap, y tries.

es común referirse a un árbol como un `Rooted Tree` por:

```
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)
```

símbolo común para los árboles: T

Representación típica del árbol del árbol.

Normalmente, representamos un árbol de anarios (uno con hijos potencialmente ilimitados por nodo) como un árbol binario (uno de ellos con exactamente dos hijos por nodo). El "siguiente" niño es considerado como un hermano. Tenga en cuenta que si un árbol es binario, esta representación crea nodos adicionales.

Luego iteramos sobre los hermanos y asistimos a los niños. Como la mayoría de los árboles son relativamente poco profundos: muchos niños, pero solo unos pocos niveles de jerarquía, dan lugar a un código eficiente. Tenga en cuenta que las genealogías humanas son una excepción (muchos niveles de ancestros, solo unos pocos niños por nivel).

Si es necesario, se pueden guardar punteros hacia atrás para permitir que el árbol ascienda. Estos son más difíciles de mantener.

Tenga en cuenta que es típico tener una función para llamar a la raíz y una función recursiva con parámetros adicionales, en este caso, la profundidad del árbol.

```
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)
{
    printtree_r(root, 0);
}

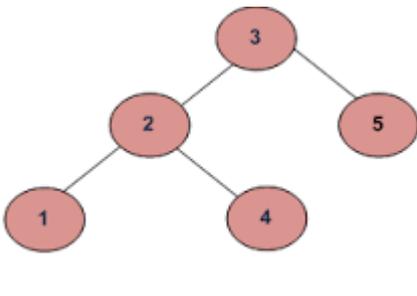
```

Para comprobar si dos árboles binarios son iguales o no.

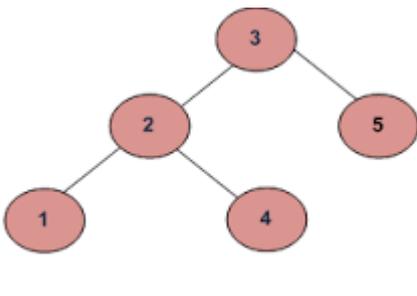
1. Por ejemplo si las entradas son:

Ejemplo 1

una)



segundo)

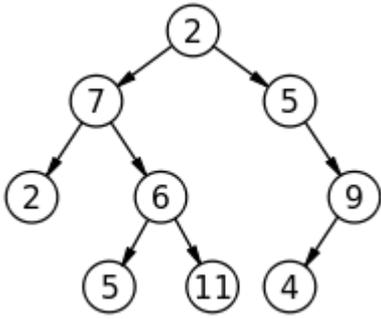


La salida debe ser verdadera.

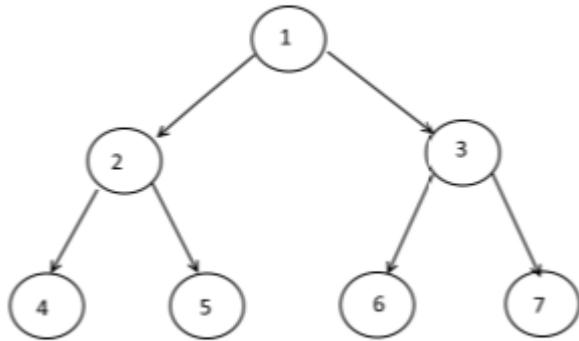
Ejemplo: 2

Si las entradas son:

una)



segundo)



La salida debe ser falsa.

Seudo código para el mismo:

```

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;

}
  
```

Lea Arboles en línea: <https://riptutorial.com/es/algorithm/topic/5737/arboles>

Capítulo 22: Árboles binarios de búsqueda

Introducción

El árbol binario es un árbol en el que cada nodo tiene un máximo de dos hijos. El árbol de búsqueda binaria (BST) es un árbol binario en el que sus elementos se colocan en un orden especial. En cada BST, todos los valores (es decir, la clave) en el subárbol izquierdo son menores que los valores en el subárbol derecho.

Examples

Árbol de búsqueda binario - Inserción (Python)

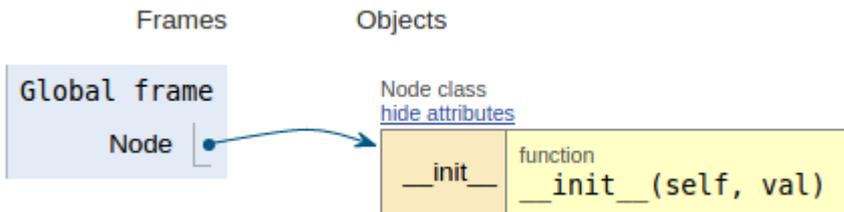
Esta es una implementación simple de la inserción de árbol de búsqueda binaria utilizando Python.

A continuación se muestra un ejemplo:

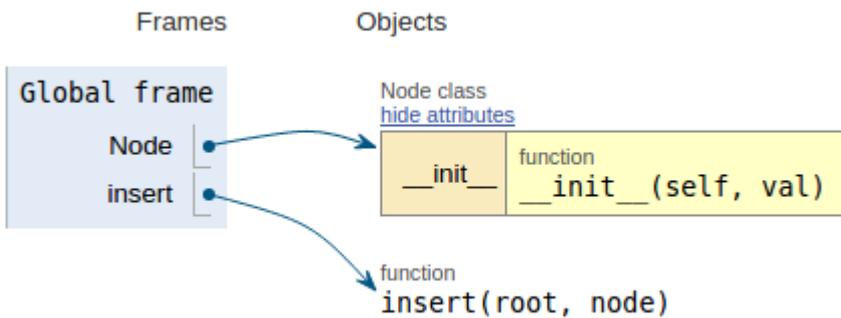
www.penjee.com

Siguiendo el fragmento de código, cada imagen muestra la visualización de ejecución, lo que facilita la visualización de cómo funciona este código.

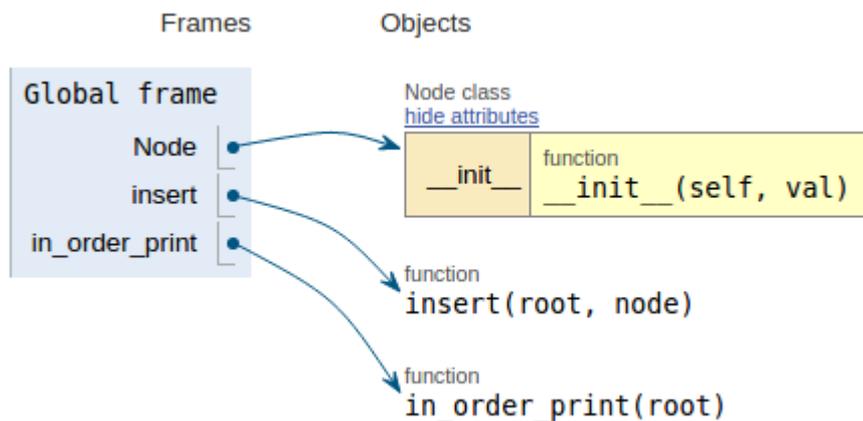
```
class Node:
    def __init__(self, val):
        self.l_child = None
        self.r_child = None
        self.data = val
```



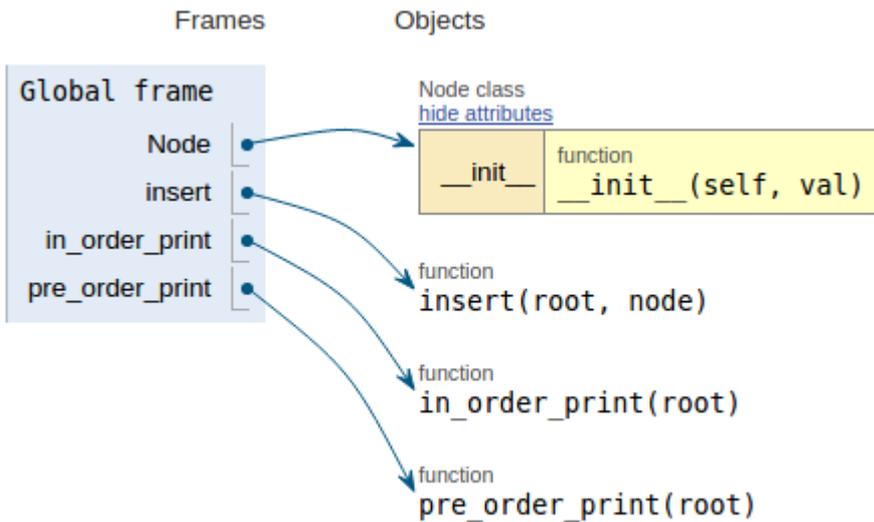
```
def insert(root, node):
    if root is None:
        root = node
    else:
        if root.data > node.data:
            if root.l_child is None:
                root.l_child = node
            else:
                insert(root.l_child, node)
        else:
            if root.r_child is None:
  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



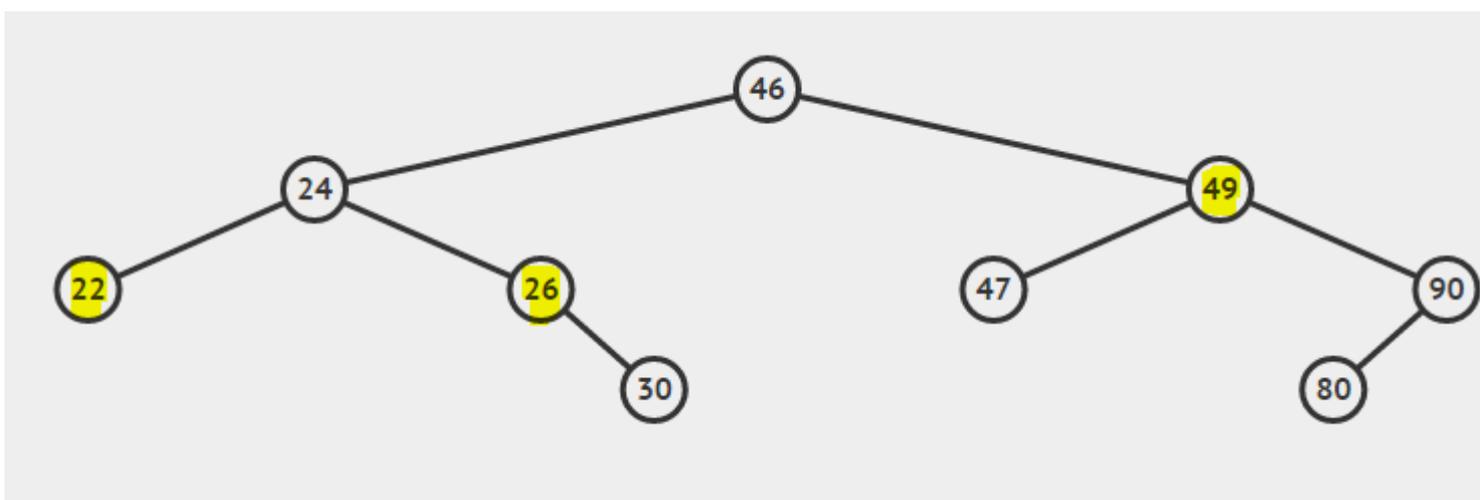
```
def pre_order_print(root):
    if not root:
        return
    print root.data
    pre_order_print(root.l_child)
    pre_order_print(root.r_child)
```



Árbol de búsqueda binario - Eliminación (C ++)

Antes de comenzar con la eliminación, solo quiero poner algunas luces en lo que es un árbol de búsqueda binaria (BST). Cada nodo en una BST puede tener un máximo de dos nodos (hijo izquierdo y derecho). El subárbol izquierdo de un nodo tiene una clave menor o igual que la clave de su nodo principal. El subárbol derecho de un nodo tiene una clave mayor que la clave de su nodo principal.

Eliminar un nodo en un árbol mientras se mantiene su **propiedad de árbol de búsqueda binaria**.



Hay tres casos que deben considerarse al eliminar un nodo.

- Caso 1: el nodo que se eliminará es el nodo de hoja (nodo con valor 22).
- Caso 2: el nodo que se va a eliminar tiene un hijo (nodo con valor 26).

- Caso 3: El nodo a eliminar tiene ambos hijos (Nodo con valor 49).

Explicación de los casos:

1. Cuando el nodo que se va a eliminar es un nodo de hoja, simplemente elimine el nodo y pase `nullptr` a su nodo principal.
2. Cuando un nodo que se va a eliminar tiene solo un hijo, copie el valor hijo en el valor del nodo y elimine el hijo **(convertido al caso 1)**
3. Cuando un nodo que se va a eliminar tiene dos hijos, el mínimo de su subárbol derecho se puede copiar al nodo y el valor mínimo se puede eliminar del subárbol derecho del nodo **(Convertido al Caso 2)**

Nota: El mínimo en el subárbol derecho puede tener un máximo de un hijo y ese hijo también correcto si tiene el hijo izquierdo significa que no es el valor mínimo o que no está siguiendo la propiedad BST.

La estructura de un nodo en un árbol y el código para la eliminación:

```
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;

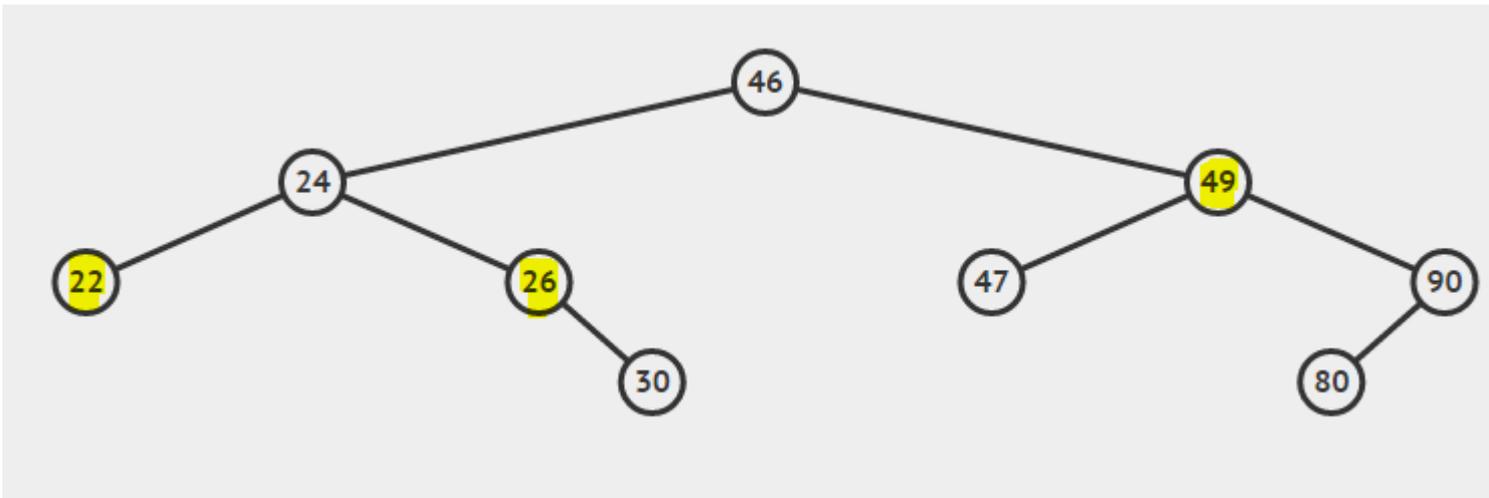
            root->data = temp->data;
            root->right = delete_node(root->right, temp->data);
        }
    }
}
```

```
    }  
  }  
  return root;  
}
```

La complejidad de tiempo del código anterior es $O(h)$, donde h es la altura del árbol.

El antepasado común más bajo en un BST

Considere el BST:



El antepasado común más bajo de 22 y 26 es 24.

El antepasado común más bajo de 26 y 49 es 46.

El antepasado común más bajo de 22 y 24 es 24.

La propiedad del árbol de búsqueda binario se puede usar para encontrar los nodos del antepasado más bajo

Código Psuedo:

```
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);
}
}

```

Árbol binario de búsqueda - 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

```

""" "Crear un nodo diferente e insertar datos en él" """

```

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))
```

Lea **Árboles binarios de búsqueda en línea**: <https://riptutorial.com/es/algorithm/topic/5735/arboles-binarios-de-busqueda>

Capítulo 23: buscando

Examples

Búsqueda binaria

Introducción

Binary Search es un algoritmo de búsqueda Dividir y Conquistar. Utiliza $O(\log n)$ tiempo para encontrar la ubicación de un elemento en un espacio de búsqueda donde n es el tamaño del espacio de búsqueda.

La búsqueda binaria funciona dividiendo a la mitad el espacio de búsqueda en cada iteración después de comparar el valor objetivo con el valor medio del espacio de búsqueda.

Para utilizar la búsqueda binaria, el espacio de búsqueda debe ordenarse (ordenarse) de alguna manera. Las entradas duplicadas (las que se comparan como iguales según la función de comparación) no se pueden distinguir, aunque no violan la propiedad de búsqueda binaria.

Convencionalmente, usamos menos de ($<$) como función de comparación. Si $a < b$, devolverá verdadero. si a no es menor que b y b no es menor que a , a y b son iguales.

Ejemplo de pregunta

Eres un economista, aunque bastante malo. Se le asigna la tarea de encontrar el precio de equilibrio (es decir, el precio donde la oferta = demanda) para el arroz.

Recuerde que cuanto mayor sea el precio, mayor será la oferta y menor la demanda.

Como su compañía es muy eficiente en el cálculo de las fuerzas del mercado, puede obtener instantáneamente la oferta y la demanda en unidades de arroz cuando el precio del arroz se establece a un precio determinado p .

Su jefe desea el precio de equilibrio CUANTO ANTES, pero le dice que el precio de equilibrio puede ser un número entero positivo que sea a lo sumo 10^{17} y se garantiza que exista exactamente una solución de número entero positivo en el rango. ¡Así que sigue con tu trabajo antes de que lo pierdas!

Se le permite llamar a las funciones `getSupply(k)` y `getDemand(k)`, que harán exactamente lo que se indica en el problema.

Explicación de ejemplo

Aquí nuestro espacio de búsqueda es de 1 a 10^{17} . Así, una búsqueda lineal es inviable.

Sin embargo, observe que a medida que k aumenta, $\text{getSupply}(k)$ aumenta y $\text{getDemand}(k)$ disminuye. Por lo tanto, para cualquier $x > y$, $\text{getSupply}(x) - \text{getDemand}(x) > \text{getSupply}(y) - \text{getDemand}(y)$. Por lo tanto, este espacio de búsqueda es monotónico y podemos utilizar la búsqueda binaria.

El siguiente pseudocódigo demuestra el uso de la búsqueda binaria:

```
high = 1000000000000000000    <- 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
```

Este algoritmo se ejecuta en tiempo $\sim O(\log 10^{17})$. Esto se puede generalizar a $\sim O(\log S)$ de tiempo en la que S es el tamaño del espacio de búsqueda, ya que en cada iteración del `while` de bucle, que reduce a la mitad el espacio de búsqueda (*de [baja: alta] a cualquiera [baja: mediados] o [medio: alto]*).

C Implementación de búsqueda binaria con recursión.

```
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);
  }
}
```

Búsqueda binaria: en números ordenados

Es más fácil mostrar una búsqueda binaria en números usando pseudocódigo

```
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

```

No intente regresar temprano comparando `array [mid]` con `x` para la igualdad. La comparación adicional solo puede ralentizar el código. Tenga en cuenta que debe agregar uno a bajo para evitar quedar atrapado por la división de enteros siempre redondeando hacia abajo.

Curiosamente, la versión anterior de la búsqueda binaria le permite encontrar la aparición más pequeña de `x` en la matriz. Si la matriz contiene duplicados de `x`, el algoritmo se puede modificar ligeramente para que devuelva la mayor aparición de `x` simplemente agregando a la condicional de condicional:

```

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;
}

```

Tenga en cuenta que en lugar de hacer `mid = (low + high) / 2`, también puede ser una buena idea probar `mid = low + ((high - low) / 2)` para implementaciones como las implementaciones de Java para reducir el riesgo de obtener un Desbordamiento para insumos realmente grandes.

Busqueda lineal

La búsqueda lineal es un algoritmo simple. Recorre los elementos hasta que se encuentra la consulta, lo que lo convierte en un algoritmo lineal: la complejidad es $O(n)$, donde n es el número de elementos que se deben revisar.

¿Por qué $O(n)$? En el peor de los casos, tienes que pasar por todos los n elementos.

Puede compararse con la búsqueda de un libro en una pila de libros: recorre todos ellos hasta encontrar el que desea.

A continuación se muestra una implementación de 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

```

Rabin Karp

El algoritmo de Rabin-Karp o el algoritmo de Karp-Rabin es un algoritmo de búsqueda de cadenas que utiliza el hash para encontrar cualquiera de un conjunto de cadenas de patrones en un texto. Su tiempo promedio y el mejor de los casos es $O(n + m)$ en el espacio $O(p)$, pero su tiempo en el peor de los casos es $O(nm)$ donde n es la longitud del texto y m es la longitud del patrón.

Implementación de algoritmos en java para la coincidencia de cadenas

```

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;
        }
    }
}
}

```

Al calcular el valor hash, lo estamos dividiendo por un número primo para evitar la colisión.

Después de dividir por el número primo, las posibilidades de colisión serán menores, pero aún existe la posibilidad de que el valor hash pueda ser el mismo para dos cadenas, por lo que conseguimos una coincidencia que tenemos que verificar personaje por personaje para asegurarnos de que obtenemos una coincidencia adecuada.

```
t = (d * (t - text.charAt (i) * h) + text.charAt (i + m))% q;
```

Esto es para recalcular el valor de hash para el patrón, primero eliminando el carácter más a la izquierda y luego agregando el nuevo carácter del texto.

Análisis de búsqueda lineal (peor, promedio y mejores casos)

Podemos tener tres casos para analizar un algoritmo:

1. Peor de los casos
2. Caso medio
3. Mejor caso

```
#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;
}
```

/ Programa controlador para probar las funciones anteriores */*

```
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;
}
```

Análisis de los peores casos (generalmente hecho)

En el peor análisis de caso, calculamos el límite superior del tiempo de ejecución de un algoritmo. Debemos conocer el caso que provoca la máxima cantidad de operaciones a ejecutar. Para la

búsqueda lineal, el caso más desfavorable ocurre cuando el elemento a buscar (x en el código anterior) no está presente en la matriz. Cuando x no está presente, las funciones de búsqueda () lo comparan con todos los elementos de arr [] uno por uno. Por lo tanto, la complejidad en el peor de los casos de la búsqueda lineal sería $\Theta(n)$

Análisis de casos promedio (a veces hecho)

En el análisis de casos promedio, tomamos todas las entradas posibles y calculamos el tiempo de cálculo para todas las entradas. Sumamos todos los valores calculados y dividimos la suma por el número total de entradas. Debemos conocer (o predecir) la distribución de los casos. Para el problema de búsqueda lineal, supongamos que todos los casos se distribuyen uniformemente (incluido el caso de que x no esté presente en la matriz). Entonces sumamos todos los casos y dividimos la suma por (n + 1). A continuación se muestra el valor de la complejidad media de tiempo de caso.

$$\begin{aligned} \text{Average Case Time} &= \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)} \\ &= \frac{\theta((n+1)*(n+2)/2)}{(n+1)} \\ &= \theta(n) \end{aligned}$$

Mejor análisis de caso (Bogus)

En el mejor análisis de casos, calculamos el límite inferior del tiempo de ejecución de un algoritmo. Debemos conocer el caso que causa la cantidad mínima de operaciones a ejecutar. En el problema de búsqueda lineal, el mejor caso ocurre cuando x está presente en la primera ubicación. El número de operaciones en el mejor de los casos es constante (no depende de n). Entonces, la complejidad del tiempo en el mejor de los casos sería $\Theta(1)$ La mayoría de las veces, hacemos el análisis del peor de los casos para analizar algoritmos. En el peor de los análisis, garantizamos un límite superior en el tiempo de ejecución de un algoritmo que es buena información. El análisis de casos promedio no es fácil de hacer en la mayoría de los casos prácticos y rara vez se realiza. En el análisis de casos promedio, debemos conocer (o predecir) la distribución matemática de todas las entradas posibles. El mejor análisis de casos es falso. Garantizar un límite inferior en un algoritmo no proporciona ninguna información, como en el peor de los casos, un algoritmo puede tardar años en ejecutarse.

Para algunos algoritmos, todos los casos son asintóticamente iguales, es decir, no hay peores ni mejores casos. Por ejemplo, Merge Sort. Merge Sort realiza operaciones $\Theta(n \log n)$ en todos los casos. La mayoría de los otros algoritmos de clasificación tienen peores y mejores casos. Por ejemplo, en la implementación típica de Ordenación rápida (donde el pivote se elige como elemento de esquina), lo peor ocurre cuando la matriz de entrada ya está ordenada y lo mejor ocurre cuando los elementos pivote siempre dividen la matriz en dos mitades. Para la ordenación

por inserción, el peor de los casos ocurre cuando la matriz se ordena en orden inverso y el mejor caso ocurre cuando la matriz se ordena en el mismo orden que la salida.

Lea buscando en línea: <https://riptutorial.com/es/algorithm/topic/4471/buscando>

Capítulo 24: Búsqueda de amplitud

Examples

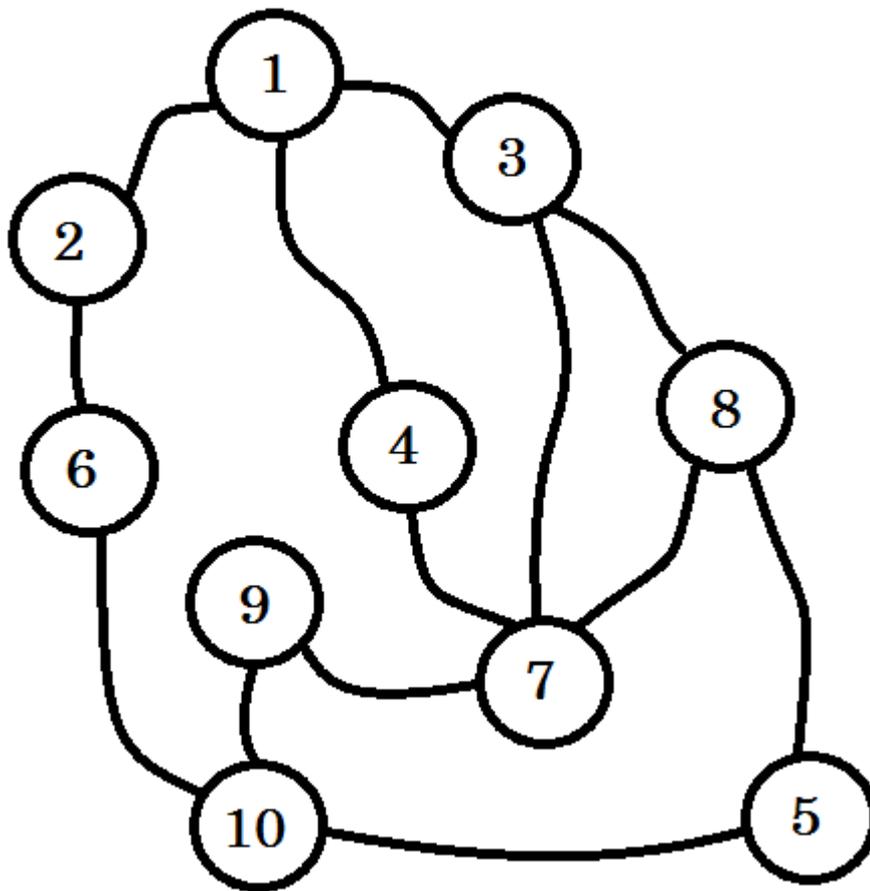
Encontrar el camino más corto desde la fuente a otros nodos

La **búsqueda en primer lugar** (BFS) es un algoritmo para atravesar o buscar estructuras de datos de árboles o gráficos. Comienza en la raíz del árbol (o en algún nodo arbitrario de un gráfico, a veces denominado "clave de búsqueda") y explora los nodos vecinos primero, antes de pasar al siguiente nivel de vecinos. BFS fue inventado a fines de la década de 1950 por [Edward Forrest Moore](#), quien lo usó para encontrar el camino más corto para salir de un laberinto y fue descubierto por CY Lee como un algoritmo de enrutamiento de cables en 1961.

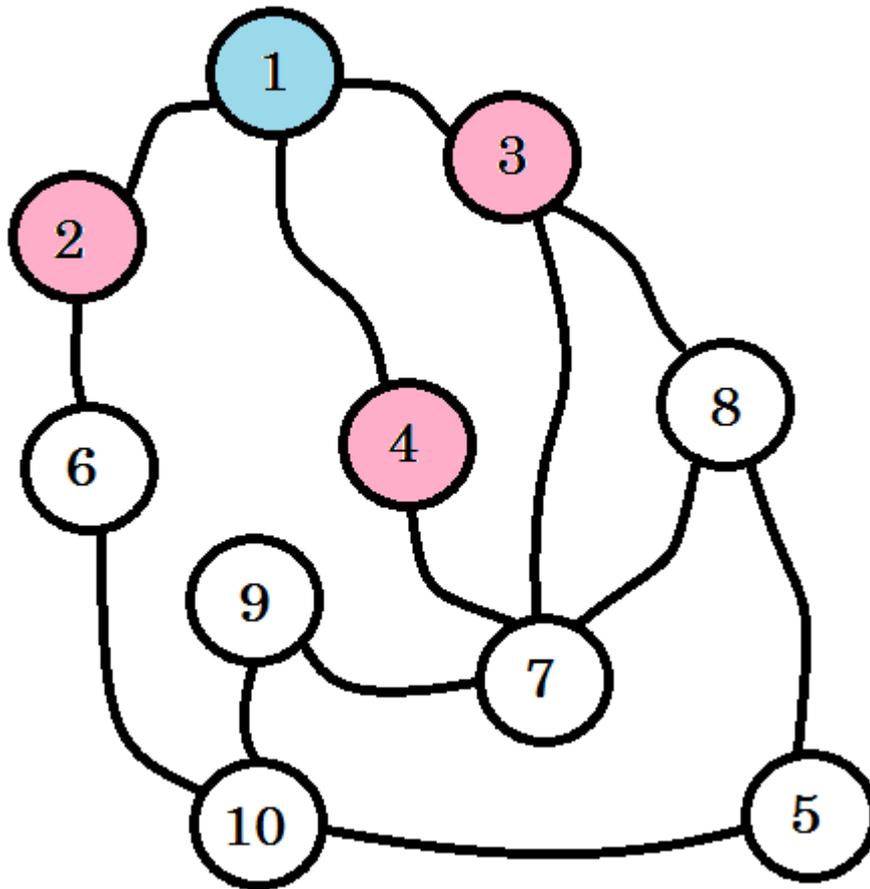
Los procesos del algoritmo BFS funcionan bajo estas suposiciones:

1. No atravesaremos ningún nodo más de una vez.
2. El nodo de origen o el nodo desde el que comenzamos se encuentra en el nivel 0.
3. Los nodos a los que podemos acceder directamente desde el nodo fuente son los nodos del nivel 1, los nodos a los que podemos acceder directamente desde los nodos del nivel 1 son los nodos del nivel 2 y así sucesivamente.
4. El nivel denota la distancia del camino más corto desde la fuente.

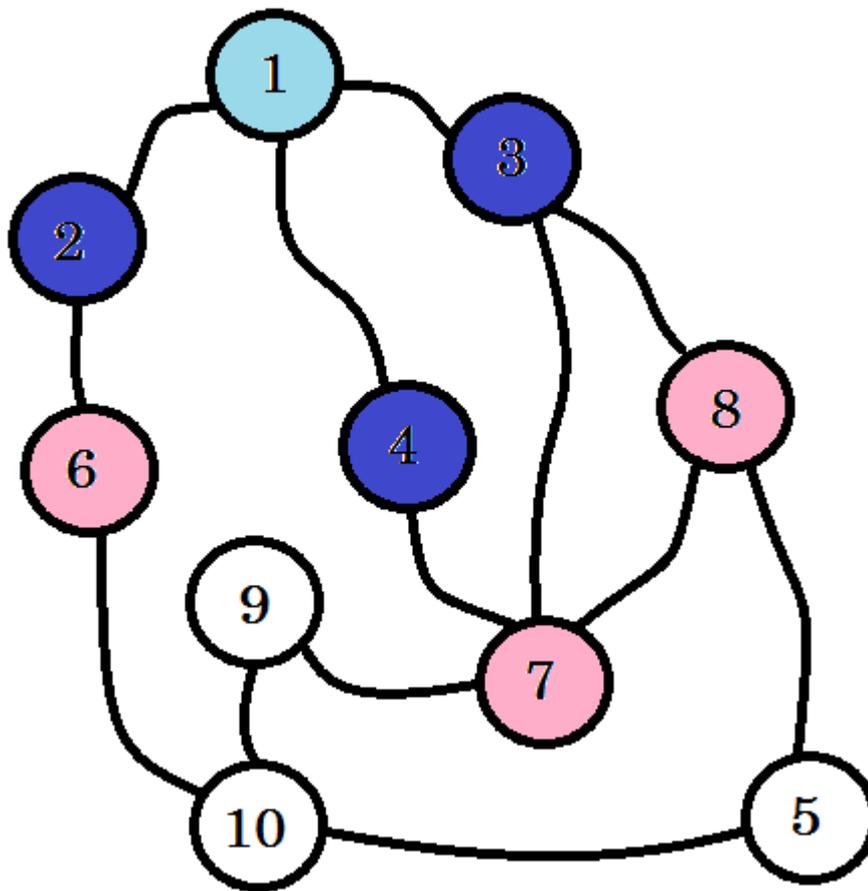
Veamos un ejemplo:



Supongamos que este gráfico representa la conexión entre varias ciudades, donde cada nodo denota una ciudad y un borde entre dos nodos indica que hay una carretera que los une. Queremos ir del **nodo 1** al **nodo 10**. Entonces el **nodo 1** es nuestra **fuentes**, que es el **nivel 0**. Marcamos el **nodo 1** como visitado. Podemos ir al **nodo 2**, **nodo 3** y **nodo 4** desde aquí. Así que estarán a **nivel (0 + 1) = nodos de nivel 1**. Ahora los marcaremos como visitados y trabajaremos con ellos.

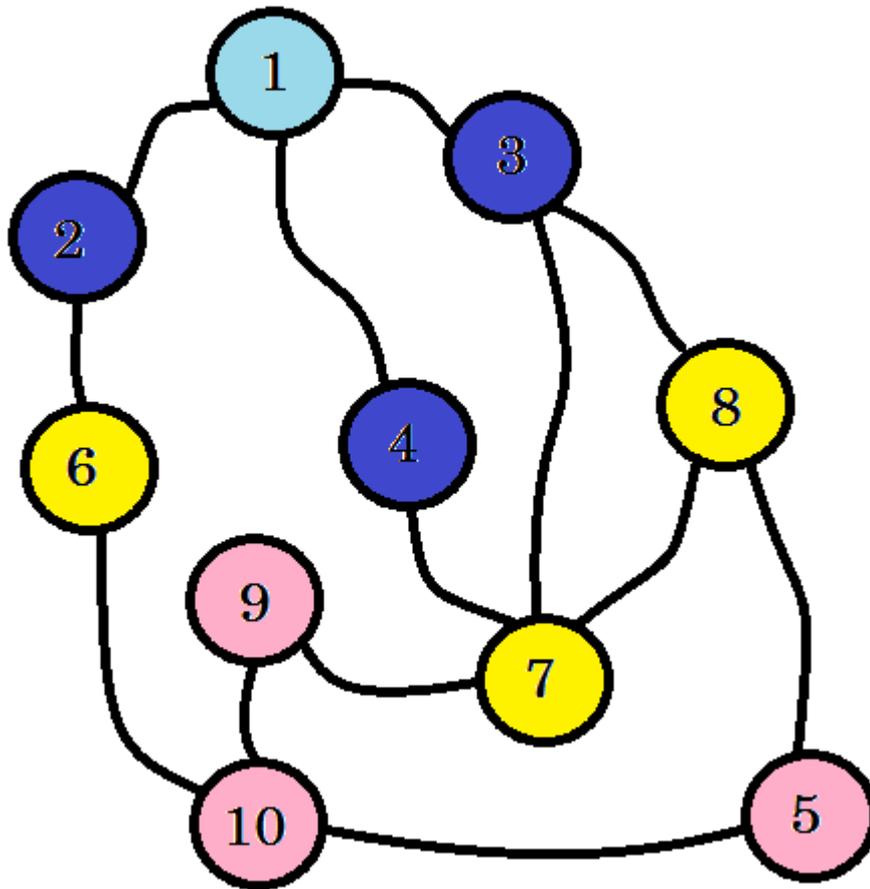


Se visitan los nodos de colores. Los nodos con los que estamos trabajando actualmente se marcarán con rosa. No visitaremos el mismo nodo dos veces. Desde el **nodo 2** , el **nodo 3** y el **nodo 4** , podemos ir al **nodo 6**, **nodo 7** y **nodo 8** . Vamos a marcarlos como visitados. El nivel de estos nodos será **nivel (1 + 1) = nivel 2** .

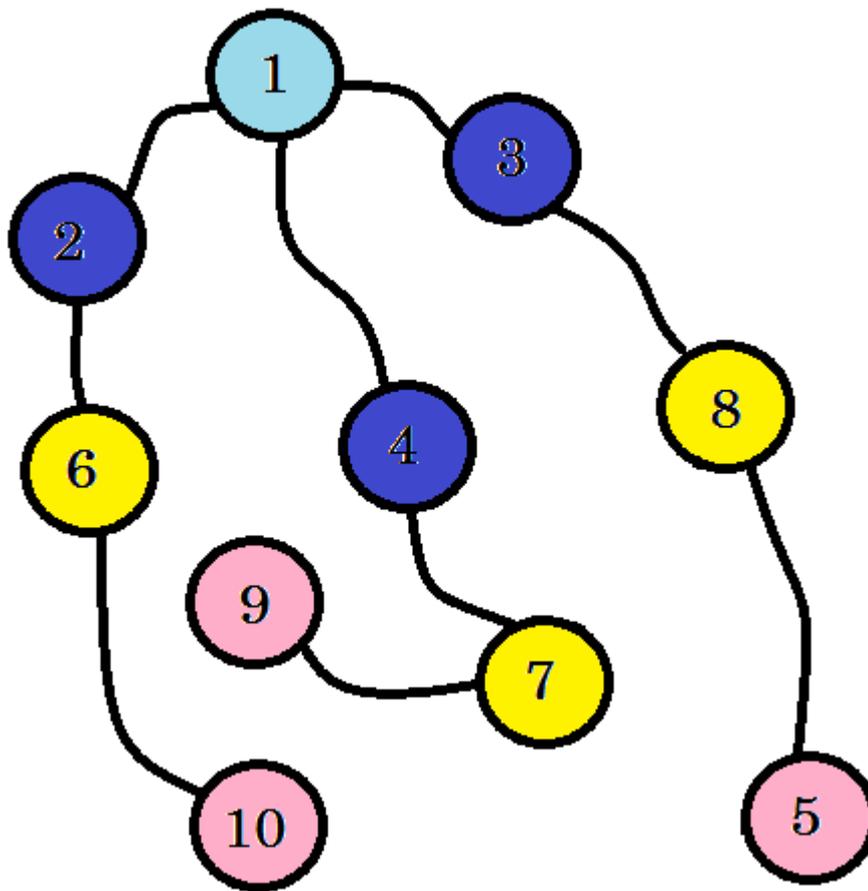


Si no lo ha notado, el nivel de los nodos simplemente denota la distancia de ruta más corta desde la **fuentes** . Por ejemplo: hemos encontrado el **nodo 8** en el **nivel 2** . Entonces la distancia desde la **fuentes** al **nodo 8** es **2** .

Aún no llegamos a nuestro nodo objetivo, ese es el **nodo 10** . Así que visitemos los próximos nodos. Podemos ir directamente desde el **nodo 6** , **nodo 7** y **nodo 8** .



Podemos ver que encontramos el **nodo 10** en el **nivel 3** . Así que la ruta más corta desde la **fuelle** hasta el **nodo 10** es **3**. Buscamos el nivel del gráfico por nivel y encontramos la ruta más corta. Ahora vamos a borrar los bordes que no usamos:



Después de eliminar los bordes que no usamos, obtenemos un árbol llamado árbol BFS. Este árbol muestra la ruta más corta desde la **fuentes** a todos los demás nodos.

Así que nuestra tarea será, ir desde la **fuentes** hasta los nodos del **nivel 1** . Luego, desde los nodos del **nivel 1** al **nivel 2**, etc., hasta que llegemos a nuestro destino. Podemos usar la *cola* para almacenar los nodos que vamos a procesar. Es decir, para cada nodo con el que vamos a trabajar, empujaremos a todos los demás nodos que puedan atravesarse directamente y que aún no estén en la cola.

La simulación de nuestro ejemplo:

Primero empujamos la fuente en la cola. Nuestra cola se verá como:

```

front
+-----+
|  1  |
+-----+
  
```

El nivel del **nodo 1** será 0. **nivel [1] = 0** . Ahora empezamos nuestro BFS. Al principio, sacamos un nodo de nuestra cola. Conseguimos el **nodo 1** . Podemos ir al **nodo 4** , **nodo 3** y **nodo 2** desde este. Hemos llegado a estos nodos desde el **nodo 1** . Entonces **nivel [4] = nivel [3] = nivel [2] = nivel [1] + 1 = 1** . Ahora los marcamos como visitados y los empujamos en la cola.

```

                                front
+-----+ +-----+ +-----+
|  2  | |  3  | |  4  |
+-----+ +-----+ +-----+

```

Ahora abrimos el **nodo 4** y trabajamos con él. Podemos ir al **nodo 7** desde el **nodo 4** . **nivel [7] = nivel [4] + 1 = 2** . Marcamos el **nodo 7** como visitado y lo colocamos en la cola.

```

                                front
+-----+ +-----+ +-----+
|  7  | |  2  | |  3  |
+-----+ +-----+ +-----+

```

Desde el **nodo 3** , podemos ir al **nodo 7** y al **nodo 8** . Como ya hemos marcado el **nodo 7** como visitado, marcamos el **nodo 8** como visitado, cambiamos el **nivel [8] = nivel [3] + 1 = 2** . Empujamos el **nodo 8** en la cola.

```

                                front
+-----+ +-----+ +-----+
|  6  | |  7  | |  2  |
+-----+ +-----+ +-----+

```

Este proceso continuará hasta que llegemos a nuestro destino o la cola se vacíe. La matriz de **nivel** nos proporcionará la distancia de la ruta más corta desde la **fuentes** . Podemos inicializar la matriz de **nivel** con un valor *infinito* , lo que marcará que los nodos aún no se han visitado. Nuestro pseudo-código será:

```

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

```

Al iterar a través de la matriz de **niveles** , podemos averiguar la distancia de cada nodo desde la fuente. Por ejemplo: la distancia del **nodo 10** a la **fuentes** se almacenará en el **nivel [10]** .

A veces es posible que tengamos que imprimir no solo la distancia más corta, sino también la ruta a través de la cual podemos ir a nuestro nodo destinado desde la **fuentes** . Para esto necesitamos mantener una matriz **padre** . **padre [fuente]** será NULL. Para cada actualización en la matriz de **nivel** , simplemente agregaremos `parent[v] := u` en nuestro pseudo código dentro del bucle for. Después de finalizar BFS, para encontrar la ruta, volveremos a recorrer la matriz **principal** hasta que llegemos a la **fuentes**, que se denotará con el valor NULL. El pseudocódigo será:

```

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

```

Complejidad:

Hemos visitado cada nodo una vez y todos los bordes una vez. Entonces, la complejidad será $O(V + E)$ donde V es el número de nodos y E es el número de bordes.

Encontrar la ruta más corta desde la fuente en un gráfico 2D

La mayoría de las veces, necesitaremos encontrar la ruta más corta desde una sola fuente a todos los demás nodos o un nodo específico en un gráfico 2D. Por ejemplo, queremos saber cuántos movimientos se requieren para que un caballero alcance un cierto cuadrado en un tablero de ajedrez, o tenemos una matriz donde algunas celdas están bloqueadas, tenemos que encontrar el camino más corto de una celda a otra. Podemos movernos solo horizontal y verticalmente. Incluso los movimientos diagonales pueden ser posibles también. Para estos casos, podemos convertir los cuadrados o celdas en nodos y resolver estos problemas fácilmente usando BFS. Ahora nuestro **visitado**, **padre** y **nivel** serán matrices 2D. Para cada nodo, consideraremos todos los movimientos posibles. Para encontrar la distancia a un nodo específico, también verificaremos si hemos llegado a nuestro destino.

Habrá una cosa adicional llamada matriz de dirección. Esto simplemente almacenará todas las combinaciones posibles de direcciones a las que podemos ir. Digamos, para movimientos horizontales y verticales, nuestras matrices de dirección serán:

```

+---+---+---+---+---+
| dx | 1 | -1 | 0 | 0 |
+---+---+---+---+---+
| dy | 0 | 0 | 1 | -1 |
+---+---+---+---+---+

```

Aquí dx representa el movimiento en el eje x y dy representa el movimiento en el eje y . Nuevamente esta parte es opcional. También puedes escribir todas las combinaciones posibles por separado. Pero es más fácil manejarlo usando la matriz de dirección. Puede haber más e incluso diferentes combinaciones para movimientos diagonales o movimientos de caballero.

La parte adicional que debemos tener en cuenta es:

- Si alguna de las celdas está bloqueada, para cada movimiento posible, verificaremos si la celda está bloqueada o no.
- También comprobaremos si nos hemos salido de los límites, es decir, hemos cruzado los límites de la matriz.
- Se dará el número de filas y columnas.

Nuestro pseudo-código será:

```
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
```

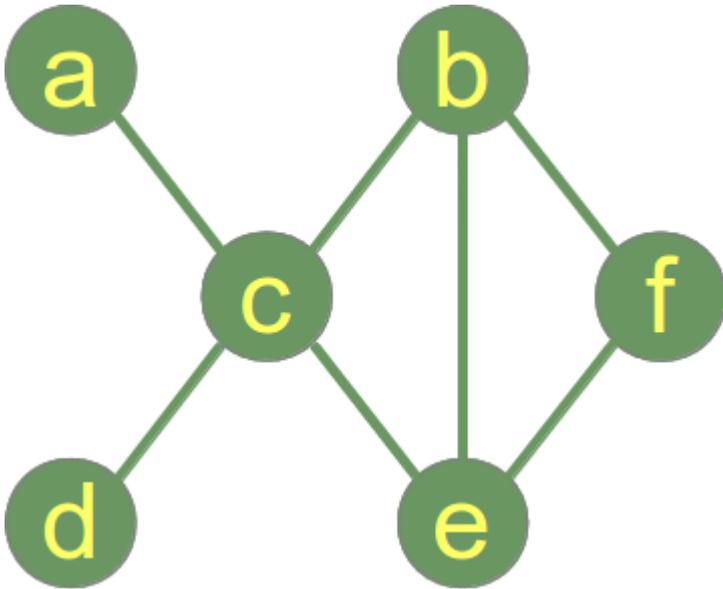
Como hemos discutido anteriormente, BFS solo funciona para gráficos no ponderados. Para gráficos ponderados, necesitaremos el algoritmo de [Dijkstra](#) . Para los ciclos de borde negativo, necesitamos el algoritmo de [Bellman-Ford](#) . Nuevamente este algoritmo es el algoritmo de ruta más corta de una sola fuente. Si necesitamos encontrar la distancia de cada nodo a todos los demás nodos, necesitaremos [el algoritmo de Floyd-Warshall](#) .

Componentes conectados de un gráfico no dirigido utilizando BFS.

BFS se puede usar para encontrar los componentes conectados de un [gráfico no dirigido](#) . También podemos encontrar si el gráfico dado está conectado o no. Nuestra discusión posterior asume que estamos tratando con gráficos no dirigidos. La definición de un gráfico conectado es:

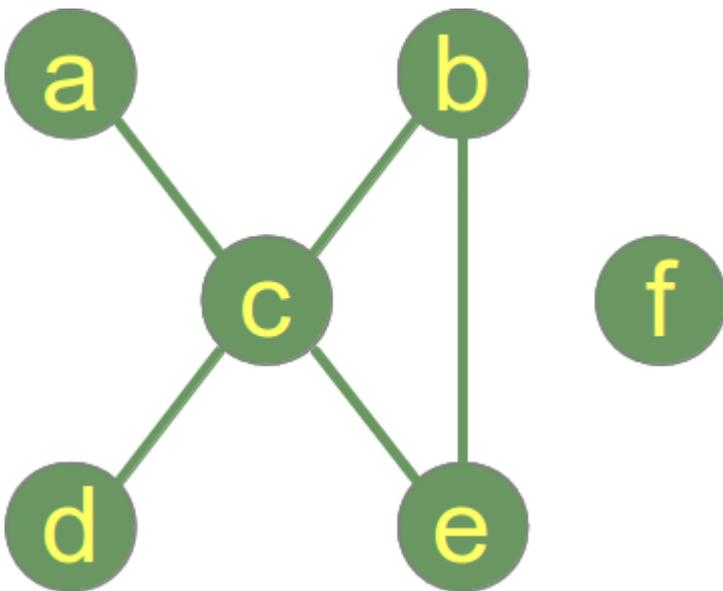
Un gráfico está conectado si hay una ruta entre cada par de vértices.

A continuación se muestra un **gráfico conectado** .



El siguiente gráfico **no** está **conectado** y tiene 2 componentes conectados:

1. Componente conectado 1: {a, b, c, d, e}
2. Componente conectado 2: {f}



BFS es un algoritmo de recorrido gráfico. Entonces, a partir de un nodo de origen aleatorio, si al finalizar el algoritmo, se visitan todos los nodos, entonces el gráfico está conectado, de lo contrario no está conectado.

PseudoCódigo para el algoritmo.

```

boolean isConnected(Graph g)
{
  BFS(v)//v is a random source node.
  if(allVisited(g))
  {
    return true;
  }
  else return false;
}
  
```

Implementación de C para encontrar si un gráfico no dirigido está conectado o no:

```
#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';
            }
    }
}

```

Para encontrar todos los componentes conectados de un gráfico no dirigido, solo necesitamos agregar 2 líneas de código a la función BFS. La idea es llamar a la función BFS hasta que se visiten todos los vértices.

Las líneas a añadir son:

```

printf("\nConnected component %d\n",++count);
//count is a global variable initialized to 0
//add this as first line to BFS function

```

Y

```

printf("%d ",vertex+1);
add this as first line of while loop in BFS

```

Y definimos la siguiente función:

```

void listConnectedComponents(char **graph,int noOfVertices)
{
    int i;
    for(i = 0;i < noOfVertices;++i)
    {
        if(visited[i] == 'N')
            BFS(graph,i,noOfVertices);
    }
}

```

Lea **Búsqueda de amplitud en línea**: <https://riptutorial.com/es/algorithm/topic/7215/busqueda-de-amplitud>

Capítulo 25: Búsqueda de subcadena

Examples

Algoritmo KMP en C

Dado un texto *txt* y un patrón *pat* , el objetivo de este programa será imprimir toda la ocurrencia de *pat* en *txt* .

Ejemplos:

Entrada:

```
txt[] = "THIS IS A TEST TEXT"  
pat[] = "TEST"
```

salida:

```
Pattern found at index 10
```

Entrada:

```
txt[] = "AABAACAADAABAAABAA"  
pat[] = "AABA"
```

salida:

```
Pattern found at index 0  
Pattern found at index 9  
Pattern found at index 13
```

Implementación del lenguaje C:

```
// 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;
}

```

Salida:

```
Found pattern at index 10
```

Referencia:

<http://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/>

Introducción al algoritmo de Rabin-Karp

El [algoritmo de Rabin-Karp](#) es un algoritmo de búsqueda de cadenas creado por [Richard M. Karp](#) y [Michael O. Rabin](#) que utiliza el hash para encontrar cualquiera de un conjunto de cadenas de patrones en un texto.

Una subcadena de una cadena es otra cadena que aparece en. Por ejemplo, *ver* es una subcadena de *stackoverflow*. No debe confundirse con la subsecuencia porque *cover* es una subsecuencia de la misma cadena. En otras palabras, cualquier subconjunto de letras consecutivas en una cadena es una subcadena de la cadena dada.

En el algoritmo de Rabin-Karp, generaremos un hash de nuestro *patrón* que estamos buscando y verificaremos si el hash de nuestro *texto* coincide con el *patrón* o no. Si no coincide, podemos garantizar que el *patrón* **no existe** en el *texto*. Sin embargo, si coincide, el *patrón* **puede** estar presente en el *texto*. Veamos un ejemplo:

Digamos que tenemos un texto: **yeminsajid** y queremos averiguar si el patrón **nsa** existe en el texto. Para calcular el hash y el hash rodante, necesitaremos usar un número primo. Este puede ser cualquier número primo. Tomemos **prime = 11** para este ejemplo. Determinaremos el valor de hash usando esta fórmula:

$$(1st\ letter) \times (prime) + (2nd\ letter) \times (prime)^1 + (3rd\ letter) \times (prime)^2 \times + \dots$$

Vamos a denotar:

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
```

El valor de hash de **nsa** será:

$$14 \times 11^0 + 19 \times 11^1 + 1 \times 11^2 = 344$$

Ahora encontramos el hash de nuestro texto. Si el hash variable coincide con el valor de hash de nuestro patrón, verificaremos si las cadenas coinciden o no. Ya que nuestro patrón tiene **3** letras, tomaremos las primeras **3** letras **yem** de nuestro texto y calcularemos el valor de hash.

Obtenemos:

$$25 \times 11^0 + 5 \times 11^1 + 13 \times 11^2 = 1653$$

Este valor no coincide con el valor hash de nuestro patrón. Así que la cadena no existe aquí. Ahora tenemos que considerar el siguiente paso. Para calcular el valor hash de nuestra siguiente cadena **emi**. Podemos calcular esto utilizando nuestra fórmula. Pero eso sería bastante trivial y nos costaría más. En su lugar, utilizamos otra técnica.

- Restamos el valor de la **Primera letra de la cadena anterior** de nuestro valor de hash actual. En este caso, **y**. Obtenemos, $1653 - 25 = 1628$.
- Dividimos la diferencia con nuestro **principal**, que es **11** para este ejemplo. Lo obtenemos, $1628 / 11 = 148$.
- **Añadimos la nueva letra X (primo)**⁻¹, donde **m** es la longitud del patrón, con el cociente, que es **i = 9**. Obtenemos, $148 + 9 \times 11^2 = 1237$.

El nuevo valor de hash no es igual a nuestro valor de hash de patrones. Continuando, para **n** obtenemos:

```
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 11^2 = 1806
```

No coincide. Después de eso, por **s**, obtenemos:

```
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 11^2 = 2462
```

No coincide. A continuación, para **a**, obtenemos:

```
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
```

¡Es un partido! Ahora comparamos nuestro patrón con la cadena actual. Como ambas cadenas coinciden, la subcadena existe en esta cadena. Y volvemos a la posición inicial de nuestra subcadena.

El pseudocódigo será:

Cálculo de hash:

```
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])m-1
end for
Return hash
```

Recálculo de hash:

```
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])m-1
Return Hash
```

Partido de cuerda

```
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
```

Rabin-Karp:

```
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
```

Si el algoritmo no encuentra ninguna coincidencia, simplemente devuelve **-1** .

Este algoritmo se utiliza en la detección de plagio. Dado el material de origen, el algoritmo puede buscar rápidamente en un papel ejemplos de oraciones del material de origen, ignorando detalles como el caso y la puntuación. Debido a la abundancia de las cadenas buscadas, los algoritmos de búsqueda de una sola cadena no son prácticos aquí. De nuevo, el **algoritmo Knuth-Morris-Pratt** o el **algoritmo de búsqueda de cadenas de Boyer-Moore** es un **algoritmo de búsqueda de cadenas de un solo patrón** más rápido que **Rabin-Karp** . Sin embargo, es un algoritmo de elección para la búsqueda de múltiples patrones. Si queremos encontrar alguno de los números grandes, digamos k , patrones de longitud fija en un texto, podemos crear una variante simple del algoritmo de Rabin-Karp.

Para el texto de longitud N y P patrones de longitud combinada m , su promedio y mejor de los casos tiempo de ejecución es $O(n + m)$ en O espacio (p), pero su tiempo del peor caso es $O(nm)$.

Introducción al algoritmo de Knuth-Morris-Pratt (KMP)

Supongamos que tenemos un *texto* y un *patrón* . Necesitamos determinar si el patrón existe en el texto o no. Por ejemplo:

```
+-----+-----+-----+-----+-----+-----+
| 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 |
+-----+-----+-----+-----+
```

Este *patrón* existe en el *texto* . Por lo tanto, nuestra búsqueda de subcadenas debe devolver **3** , el índice de la posición a partir de la cual comienza este *patrón* . Entonces, ¿cómo funciona nuestro procedimiento de búsqueda de subcadenas de fuerza bruta?

Lo que normalmente hacemos es: comenzamos desde el índice **0** del *texto* y el índice **0** de nuestro patrón * y comparamos el **Texto [0]** con el **Patrón [0]** . Como no coinciden, pasamos al siguiente índice de nuestro *texto* y comparamos el **Texto [1]** con el **Patrón [0]** . Como se trata de una coincidencia, incrementamos el índice de nuestro *patrón* y el índice del *Texto* también. Comparamos el **texto [2]** con el **patrón [1]** . También son un partido. Siguiendo el mismo procedimiento establecido anteriormente, ahora comparamos el **Texto [3]** con el **Patrón [2]** . Como no coinciden, comenzamos desde la siguiente posición donde comenzamos a encontrar el partido. Ese es el índice **2** del *texto* . Comparamos el **texto [2]** con el **patrón [0]** . Ellos no coinciden. Luego, incrementando el índice del *texto* , comparamos el **texto [3]** con el **patrón [0]** . Ellos coinciden. Nuevamente coinciden el **texto [4]** y el **patrón [1]** , coinciden el **texto [5]** y el **patrón [2]** y coinciden el **texto [6]** y el **patrón [3]** . Desde que llegamos al final de nuestro *patrón*

, ahora devolvemos el índice desde el que comenzó nuestra coincidencia, que es **3** . Si nuestro *patrón* era: `bcgll` , eso significa que si el *patrón* no existía en nuestro *texto* , nuestra búsqueda debería devolver una excepción o **-1** o cualquier otro valor predefinido. Podemos ver claramente que, en el peor de los casos, este algoritmo tomaría $O(mn)$ tiempo donde **m** es la longitud del *Texto* y **n** es la longitud del *Patrón* . ¿Cómo reducimos esta complejidad del tiempo? Aquí es donde KMP Substring Search Algorithm entra en la imagen.

El Algoritmo de Búsqueda de Cadenas **Knuth-Morris-Pratt** o el **Algoritmo KMP** busca las ocurrencias de un "Patrón" dentro de un "Texto" principal mediante el uso de la observación de que cuando se produce una falta de coincidencia, la palabra en sí incluye información suficiente para determinar dónde podría comenzar la próxima coincidencia , evitando así el reexamen de personajes previamente emparejados. El algoritmo fue concebido en 1970 por **Donald Knuth** y **Vaughan Pratt** e independientemente por **James H. Morris** . El trío lo publicó conjuntamente en 1977.

Extendamos nuestro ejemplo *Texto* y *patrón* para una mejor comprensión:

```

+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 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| |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

Al principio, nuestro *texto* y *patrón* coinciden hasta el índice **2** . **El texto [3]** y el **patrón [3]** no coinciden. Por lo tanto, nuestro objetivo es no retroceder en este *Texto* , es decir, en caso de una falta de coincidencia, no queremos que nuestra coincidencia comience nuevamente desde la posición con la que comenzamos a coincidir. Para lograrlo, buscaremos un **sufijo** en nuestro *patrón* justo antes de que se produzca la discrepancia (subcadena **abc**), que también es un **prefijo** de la subcadena de nuestro *patrón* . Para nuestro ejemplo, ya que todos los caracteres son únicos, no hay sufijo, es el prefijo de nuestra subcadena coincidente. Entonces, lo que eso significa es que nuestra próxima comparación comenzará a partir del índice **0** . Espera un momento, entenderás por qué hicimos esto. A continuación, comparamos el **Texto [3]** con el **Patrón [0]** y no coincide. Después de eso, para el *texto* del índice **4** al índice **9** y para el *patrón* del índice **0** al índice **5** , encontramos una coincidencia. Encontramos una falta de coincidencia en **Texto [10]** y **Patrón [6]** . Así que tomamos la subcadena del *patrón* justo antes del punto donde se produce la discrepancia (subcadena **abcdabc**), verificamos si hay un sufijo, que también es un prefijo de esta subcadena. Podemos ver aquí que **ab** es tanto el sufijo como el prefijo de esta subcadena. Lo que eso significa es que, desde que hemos hecho coincidir hasta **Texto [10]** , los caracteres justo antes de la falta de coincidencia son **ab** . Lo que podemos inferir de esto es que dado que **ab** también es un prefijo de la subcadena que tomamos, no tenemos que marcar **ab** nuevamente y la próxima verificación puede comenzar desde **Texto [10]** y **Patrón [2]** . No tuvimos que volver a mirar el *texto* completo, podemos comenzar directamente desde donde ocurrió nuestra falta de coincidencia. Ahora verificamos **Texto [10]** y **Patrón [2]** , ya que es una

falta de coincidencia, y la subcadena antes de la falta de coincidencia (**abc**) no contiene un sufijo que también es un prefijo, verificamos **Texto [10]** y **Patrón [0]** , no coinciden Después, para *Texto* del índice **11** al índice **17** y para *Patrón* del índice **0** al índice **6** . Encontramos una falta de coincidencia en **Texto [18]** y **Patrón [7]** . De nuevo, comprobamos la subcadena antes de la discrepancia (subcadena **abcdabc**) y encontramos que **abc** es tanto el sufijo como el prefijo. Entonces, ya que hemos emparejado hasta el **Patrón [7]** , **abc** debe estar antes del **Texto [18]** . Eso significa que no necesitamos comparar hasta **Texto [17]** y nuestra comparación comenzará a partir de **Texto [18]** y **Patrón [3]** . Por lo tanto, encontraremos una coincidencia y devolveremos **15**, que es nuestro índice de inicio de la partida. Así es como funciona nuestra búsqueda de subcadenas KMP usando la información de sufijo y prefijo.

Ahora, ¿cómo calculamos de manera eficiente si el sufijo es igual al prefijo y en qué punto comenzar la comprobación si hay una falta de correspondencia de caracteres entre el *texto* y el *patrón* ? Echemos un vistazo a un ejemplo:

```

+-----+-----+-----+-----+-----+-----+-----+
| Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-----+-----+-----+-----+-----+-----+-----+
| Pattern| a | b | c | d | a | b | c | a |
+-----+-----+-----+-----+-----+-----+-----+

```

Generaremos una matriz que contenga la información requerida. Llamemos a la matriz **S**. El tamaño de la matriz será igual a la longitud del patrón. Como la primera letra del *patrón* no puede ser el sufijo de ningún prefijo, pondremos **S [0] = 0** . Tomamos **i = 1** y **j = 0** al principio. En cada paso, comparamos el **Patrón [i]** y el **Patrón [j]** y el incremento **i** . Si hay una coincidencia, ponemos **S [i] = j + 1** e incrementamos **j** , si hay una discrepancia, verificamos la posición del valor anterior de **j** (si está disponible) y establecemos **j = S [j-1]** (si **j** no es igual a **0**), seguimos haciendo esto hasta que **S [j]** no coincida con **S [i]** o **j** no se convierta en **0** . Para el último, ponemos **S [i] = 0** . Para nuestro ejemplo:

```

          j   i
+-----+-----+-----+-----+-----+-----+-----+
| Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-----+-----+-----+-----+-----+-----+-----+
| Pattern| a | b | c | d | a | b | c | a |
+-----+-----+-----+-----+-----+-----+-----+

```

El **patrón [j]** y el **patrón [i]** no coinciden, por lo que incrementamos **i** y como **j** es **0** , no verificamos el valor anterior y colocamos el **patrón [i] = 0** . Si seguimos incrementando **i** , para **i = 4** , obtendremos una coincidencia, por lo que colocamos **S [i] = S [4] = j + 1 = 0 + 1 = 1** e incrementamos **j** e **i** . Nuestra matriz se verá como:

```

          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 |   |   |   |
+-----+-----+-----+-----+-----+-----+-----+

```

Como el **Patrón [1]** y el **Patrón [5]** son una coincidencia, ponemos $S[i] = S[5] = j + 1 = 1 + 1 = 2$. Si continuamos, encontraremos una falta de coincidencia para $j = 3$ y $i = 7$. Como j no es igual a 0 , ponemos $j = S[j-1]$. Y compararemos los caracteres en i y j son iguales o no, ya que son iguales, pondremos $S[i] = j + 1$. Nuestra matriz completa se verá así:

```
+-----+-----+-----+-----+-----+-----+-----+
|  S     | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 1 |
+-----+-----+-----+-----+-----+-----+-----+
```

Esta es nuestra matriz requerida. Aquí, un valor distinto de cero de $S[i]$ significa que hay un sufijo de longitud $S[i]$ igual al prefijo en esa subcadena (subcadena de 0 a i) y la próxima comparación comenzará desde la posición $S[i] + 1$ del *Patrón*. Nuestro algoritmo para generar la matriz se vería así:

```
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
```

La complejidad del tiempo para construir esta matriz es $O(n)$ y la complejidad del espacio también es $O(n)$. Para asegurarse de que ha entendido completamente el algoritmo, intente generar una matriz para el patrón `aabaabaa` y verifique si el resultado coincide con [este](#).

Ahora hagamos una búsqueda de subcadenas usando el siguiente ejemplo:

```
+-----+-----+-----+-----+-----+-----+-----+
| 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 |
+-----+-----+-----+-----+-----+-----+
```

Tenemos un *texto*, un *patrón* y una matriz precalculada S usando nuestra lógica definido antes.

Comparamos **Texto [0]** y **Patrón [0]** y son iguales. El **texto [1]** y el **patrón [1]** son iguales. El **texto [2]** y el **patrón [2]** no son iguales. Verificamos el valor en la posición justo antes de la falta de coincidencia. Como **S [1]** es **0** , no hay sufijo que sea el mismo que el prefijo en nuestra subcadena y nuestra comparación comienza en la posición **S [1]** , que es **0** . Así que el **Patrón [0]** no es el mismo que el **Texto [2]** , así que seguimos adelante. El **texto [3]** es el mismo que el **patrón [0]** y hay una coincidencia hasta el **texto [8]** y el **patrón [5]** . Retrocedemos un paso en la matriz **S** y encontramos **2** . Entonces, esto significa que hay un prefijo de longitud **2** que también es el sufijo de esta subcadena (**abcab**) que es **ab** . Eso también significa que hay un **ab** antes del **Texto [8]** . Así que podemos ignorar con seguridad el **Patrón [0]** y el **Patrón [1]** y comenzar nuestra próxima comparación desde el **Patrón [2]** y el **Texto [8]** . Si continuamos, encontraremos el *patrón* en el *texto* . Nuestro procedimiento se verá como:

```

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

```

La complejidad temporal de este algoritmo, aparte del cálculo de la matriz de sufijo, es $O(m)$. Como *GenerateSuffixArray* toma $O(n)$, la complejidad del tiempo total del algoritmo KMP es: $O(m+n)$.

PD: Si desea encontrar varias apariciones de *Patrón* en el *Texto* , en lugar de devolver el valor, imprímalo / guárdelo y configure `j := S[j-1]` . También mantenga una `flag` para rastrear si ha encontrado alguna ocurrencia o no y manejarla en consecuencia.

Implementación Python del algoritmo KMP.

Pajar : La cadena en la que se debe buscar el patrón dado.

Aguja : El patrón a buscar.

Complejidad en el tiempo : la parte de búsqueda (método `strstr`) tiene la complejidad $O(n)$ donde n es la longitud del pajar, pero como la aguja también se analiza para construir la tabla de prefijos $O(m)$ se requiere para construir la tabla de prefijos donde m es la longitud de la aguja Por lo tanto, la complejidad de tiempo global para KMP es **$O(n + m)$**

Complejidad de espacio : **$O(m)$** debido a la tabla de prefijos en la aguja.

Nota: La siguiente implementación devuelve la posición de inicio de la coincidencia en el pajar (si hay una coincidencia) o devuelve -1, para los casos de borde, como si la aguja / el pajar es una cadena vacía o la aguja no se encuentra en el pajar.

```
def get_prefix_table(needle):
    prefix_set = set()
    n = len(needle)
    prefix_table = [0]*n
    delimiter = 1
    while(delimiter<n):
        prefix_set.add(needle[:delimiter])
        j = 1
        while(j<delimiter+1):
            if needle[j:delimiter+1] in prefix_set:
                prefix_table[delimiter] = delimiter - j + 1
                break
            j += 1
        delimiter += 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)
```

Lea Búsqueda de subcadena en línea: <https://riptutorial.com/es/algorithm/topic/7118/busqueda-de-subcadena>

Capítulo 26: Clasificación

Parámetros

Parámetro	Descripción
Estabilidad	Un algoritmo de clasificación es estable si conserva el orden relativo de elementos iguales después de la clasificación.
En su lugar	Un algoritmo de clasificación está en su lugar si se ordena utilizando solo la memoria auxiliar $O(1)$ (sin contar la matriz que se necesita clasificar).
La mejor complejidad del caso	Un algoritmo de clasificación tiene una complejidad en el mejor de los casos de $O(T(n))$ si su tiempo de ejecución es al menos $T(n)$ para todas las entradas posibles.
Complejidad media del caso	Un algoritmo de clasificación tiene una complejidad de tiempo de caso promedio de $O(T(n))$ si su tiempo de ejecución, promediado en todas las entradas posibles , es $T(n)$.
Peor complejidad del caso	Un algoritmo de clasificación tiene una complejidad en el peor de los casos de $O(T(n))$ si su tiempo de ejecución es a lo sumo $T(n)$.

Examples

Estabilidad en la clasificación

La estabilidad en la clasificación significa si un algoritmo de clasificación mantiene el orden relativo de las claves iguales de la entrada original en la salida del resultado.

Por lo tanto, se dice que un algoritmo de clasificación es estable si dos objetos con claves iguales aparecen en el mismo orden en la salida ordenada que aparecen en la matriz de entrada sin clasificar.

Considere una lista de pares:

```
(1, 2) (9, 7) (3, 4) (8, 6) (9, 3)
```

Ahora vamos a ordenar la lista utilizando el primer elemento de cada par.

Una **clasificación estable** de esta lista dará como resultado la siguiente lista:

```
(1, 2) (3, 4) (8, 6) (9, 7) (9, 3)
```

Porque $(9, 3)$ aparece después de $(9, 7)$ en la lista original también.

Una **clasificación inestable** generará la siguiente lista:

```
(1, 2) (3, 4) (8, 6) (9, 3) (9, 7)
```

La clasificación inestable puede generar la misma salida que la clasificación estable pero no siempre.

Clases estables bien conocidas:

- [Fusionar orden](#)
- [Tipo de inserción](#)
- [Tipo radix](#)
- Tim tipo
- [Ordenamiento de burbuja](#)

Clases inestables bien conocidas:

- [Tipo de pila](#)
- [Ordenación rápida](#)

Lea [Clasificación en línea](https://riptutorial.com/es/algorithm/topic/821/clasificacion): <https://riptutorial.com/es/algorithm/topic/821/clasificacion>

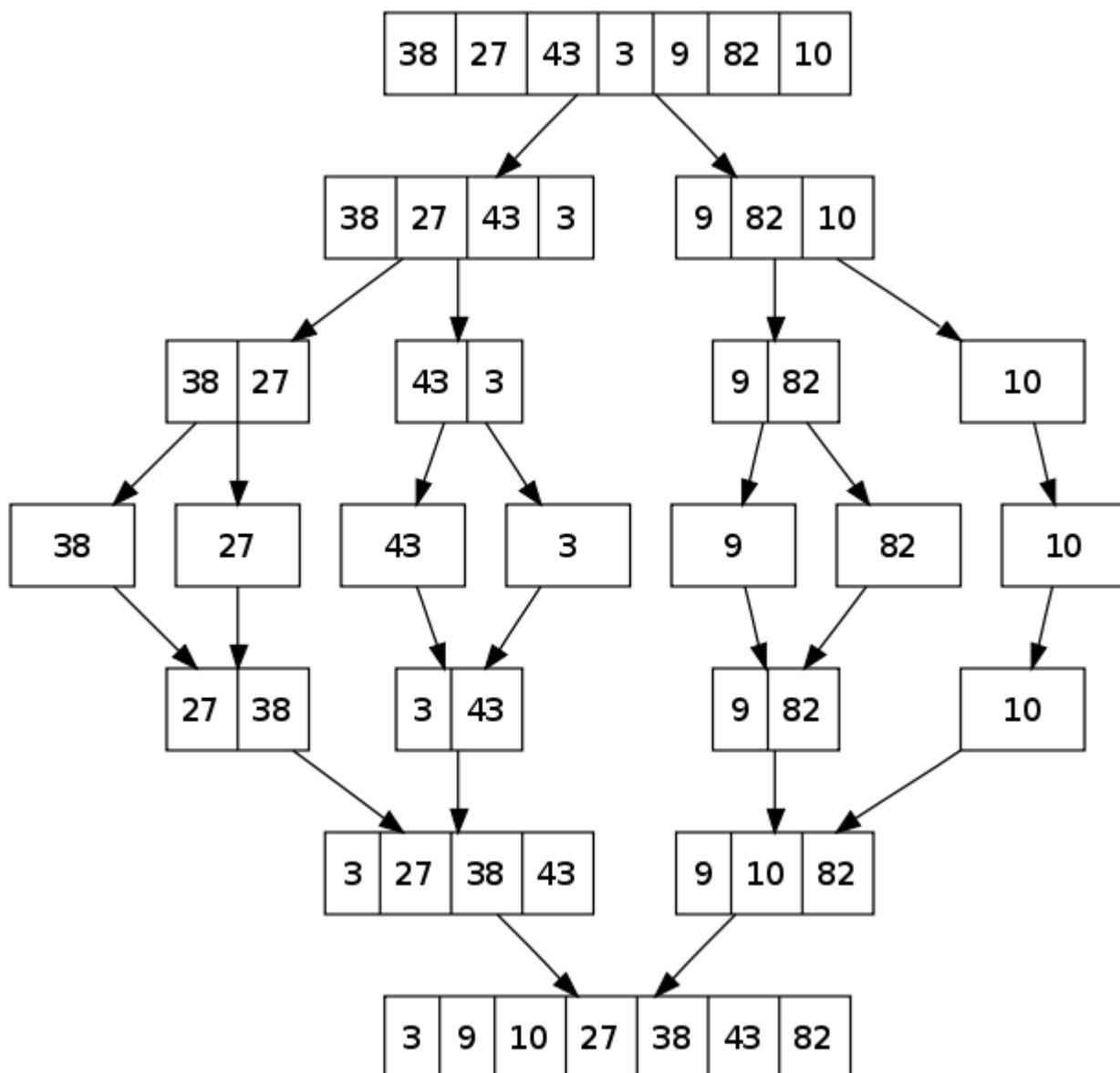
Capítulo 27: Combinar clasificación

Examples

Fundamentos de clasificación de fusión

Merge Sort es un algoritmo de dividir y conquistar. Divide la lista de entrada de longitud n por la mitad sucesivamente hasta que hay n listas de tamaño 1. Luego, las parejas de listas se combinan con el primer elemento más pequeño entre las dos listas que se agregan en cada paso. Mediante la fusión sucesiva y la comparación de los primeros elementos, se crea la lista ordenada.

Un ejemplo:



Complejidad de tiempo : $T(n) = 2T(n/2) + \Theta(n)$

La recurrencia anterior se puede resolver utilizando el método del árbol de recurrencia o el

método maestro. Cae en el caso II del Método Maestro y la solución de la recurrencia es $\Theta(n \log n)$. La complejidad temporal de la *clasificación de* $\Theta(n \log n)$ es $\Theta(n \log n)$ en los 3 casos (*peor, promedio y mejor*) ya que la clasificación de fusión siempre divide la matriz en dos mitades y toma un tiempo lineal para fusionar dos mitades.

Espacio auxiliar : $O(n)$

Paradigma algorítmico : divide y vencerás

Clasificación en el lugar : no en una implementación típica

Estable : si

Implementación de Merge Sort en C & C

C Combinar Ordenar

```
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 # Merge Sort

```
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;
}
}

```

Implementación de Merge Sort en Java

A continuación se muestra la implementación en Java utilizando un enfoque genérico. Es el mismo algoritmo, que se presenta arriba.

```

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];
}}

```

Fusionar la implementación de orden en 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)

```

Implementación de Java de abajo hacia arriba

```

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);
}
}

```

Fusionar la implementación de ordenación en 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))
}

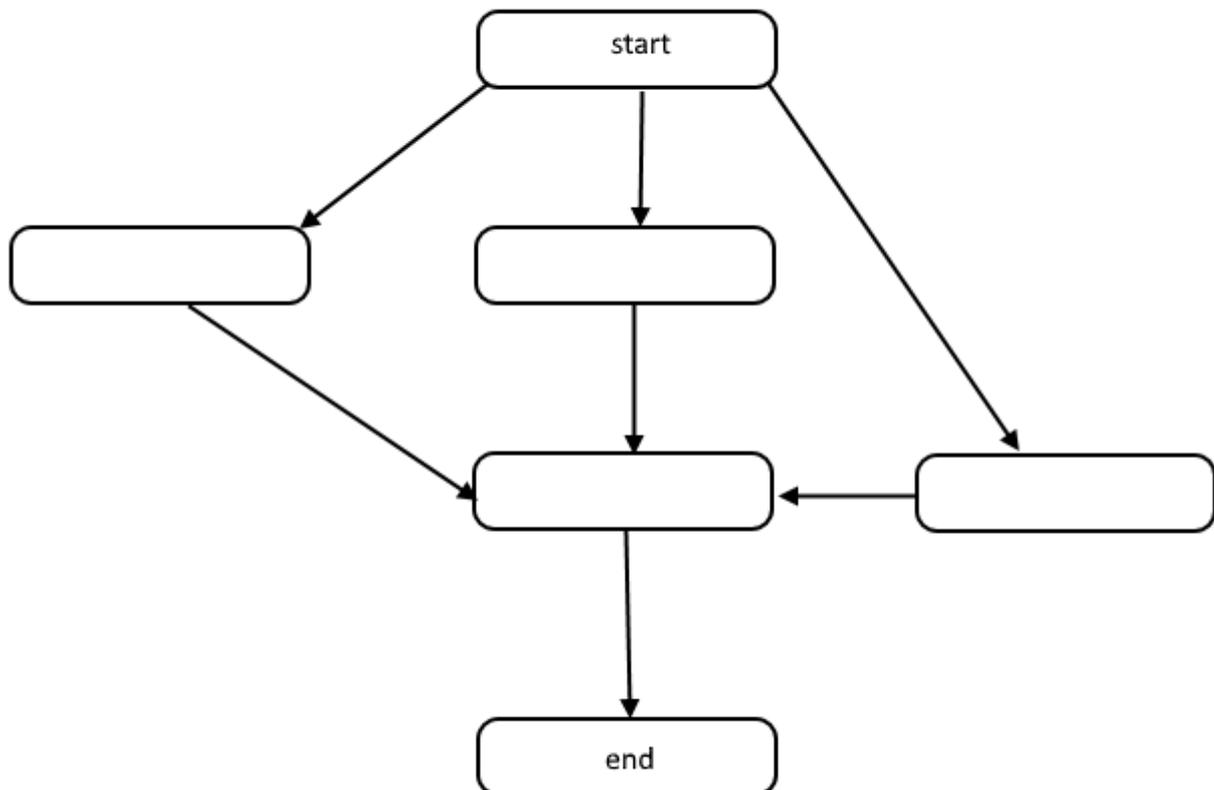
```

Lea Combinar clasificación en línea: <https://riptutorial.com/es/algorithm/topic/5732/combinar-clasificacion>

Capítulo 28: Complejidad de algoritmos

Observaciones

Todos los algoritmos son una lista de pasos para resolver un problema. Cada paso tiene dependencias en algún conjunto de pasos anteriores, o el inicio del algoritmo. Un pequeño problema podría parecerse al siguiente:

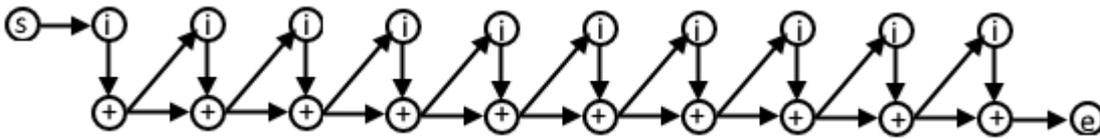


Esta estructura se llama un gráfico acíclico dirigido, o DAG para abreviar. Los enlaces entre cada nodo en el gráfico representan dependencias en el orden de las operaciones, y no hay ciclos en el gráfico.

¿Cómo ocurren las dependencias? Tomemos por ejemplo el siguiente código:

```
total = 0
for(i = 1; i < 10; i++)
    total = total + i
```

En este pseudocódigo, cada iteración del bucle for depende del resultado de la iteración anterior porque estamos usando el valor calculado en la iteración anterior en la siguiente iteración. El DAG para este código podría tener este aspecto:



Si comprende esta representación de algoritmos, puede usarla para comprender la complejidad de los algoritmos en términos de trabajo y espacio.

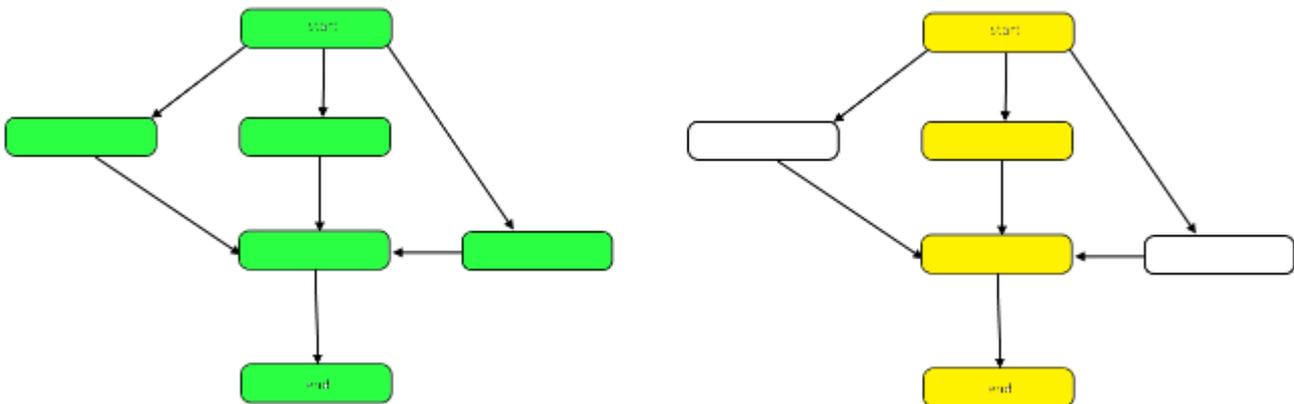
Trabajo

Trabajo es el número real de operaciones que deben ejecutarse para lograr el objetivo del algoritmo para un tamaño de entrada dado n .

Lapso

En ocasiones, se hace referencia al intervalo como la ruta crítica, y es el menor número de pasos que debe realizar un algoritmo para alcanzar el objetivo del algoritmo.

La siguiente imagen resalta el gráfico para mostrar las diferencias entre el trabajo y el intervalo en nuestro DAG de muestra.



El trabajo es el número de nodos en el gráfico en su conjunto. Esto está representado por el gráfico de la izquierda arriba. El tramo es la ruta crítica, o la ruta más larga desde el principio hasta el final. Cuando se puede trabajar en paralelo, los nodos resaltados en amarillo a la derecha representan el intervalo, el menor número de pasos necesarios. Cuando el trabajo debe realizarse en serie, el intervalo es el mismo que el trabajo.

Tanto el trabajo como el lapso pueden evaluarse independientemente en términos de análisis. La velocidad de un algoritmo está determinada por el intervalo. La cantidad de potencia computacional requerida está determinada por el trabajo.

Examples

Notación Big-Theta

A diferencia de la notación Big-O, que representa solo el límite superior del tiempo de ejecución de algunos algoritmos, Big-Theta es un límite ajustado; Ambos límites superior e inferior. La unión estrecha es más precisa, pero también más difícil de calcular.

La notación Big-Theta es simétrica: $f(x) = \Theta(g(x)) \Leftrightarrow g(x) = \Theta(f(x))$

Una forma intuitiva de comprenderlo es que $f(x) = \Theta(g(x))$ significa que los gráficos de $f(x)$ y $g(x)$ crecen en la misma tasa, o que los gráficos se "comportan" de manera similar para grandes suficientes valores de x .

La expresión matemática completa de la notación Big-Theta es la siguiente:

$\Theta(f(x)) = \{g: N_0 \rightarrow R \text{ y } c_1, c_2, n_0 > 0, \text{ donde } c_1 \leq \text{abs}(g(n)/f(n)), \text{ para cada } n > n_0 \text{ y abs es el valor absoluto}\}$

Un ejemplo

Si el algoritmo para la entrada n necesita $42n^2 + 25n + 4$ operaciones para finalizar, decimos que es $O(n^2)$, pero también es $O(n^3)$ y $O(n^{100})$. Sin embargo, es $\Theta(n^2)$ y no es $\Theta(n^3)$, $\Theta(n^4)$ etc. El algoritmo que es $\Theta(f(n))$ también es $O(f(n))$, pero ¡no viceversa!

Definición matemática formal

$\Theta(g(x))$ es un conjunto de funciones.

$\Theta(g(x)) = \{f(x) \text{ such that there exist positive constants } c_1, c_2, N \text{ such that } 0 \leq c_1 \cdot g(x) \leq f(x) \leq c_2 \cdot g(x) \text{ for all } x > N\}$

Como $\Theta(g(x))$ es un conjunto, podríamos escribir $f(x) \in \Theta(g(x))$ para indicar que $f(x)$ es un miembro de $\Theta(g(x))$. En su lugar, normalmente escribiremos $f(x) = \Theta(g(x))$ para expresar la misma noción, esa es la forma común.

Cuando aparece $\Theta(g(x))$ en una fórmula, interpretamos que representa una función anónima que no nos interesa nombrar. Por ejemplo, la ecuación $T(n) = T(n/2) + \Theta(n)$, significa $T(n) = T(n/2) + f(n)$ donde $f(n)$ es una función en el conjunto $\Theta(n)$.

Sean f y g dos funciones definidas en algún subconjunto de números reales. Escribimos $f(x) = \Theta(g(x))$ como $x \rightarrow \text{infinity}$ si y solo si hay constantes positivas K y L y un número real x_0 tal que se mantenga:

$K|g(x)| \leq f(x) \leq L|g(x)|$ para todos $x \geq x_0$.

La definición es igual a:

$f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$

Un método que utiliza límites.

si el $\lim_{x \rightarrow \infty} f(x)/g(x) = c \in (0, \infty)$ es decir, el límite existe y es positivo, entonces $f(x) = \Theta(g(x))$

Clases de complejidad común

Nombre	Notación	n = 10	n = 100
Constante	$\Theta(1)$	1	1
Logarítmica	$\Theta(\log(n))$	3	7
Lineal	$\Theta(n)$	10	100
Linealista	$\Theta(n * \log(n))$	30	700
Cuadrático	$\Theta(n^2)$	100	10 000
Exponencial	$\Theta(2^n)$	1 024	1.267650e + 30
Factorial	$\Theta(n!)$	3 628 800	9.332622e + 157

Notación Big-Omega

La notación se utiliza para el límite inferior asintótico.

Definición formal

Sean $f(n)$ $g(n)$ dos funciones definidas en el conjunto de los números reales positivos. Escribimos $f(n) = \Omega(g(n))$ si hay constantes positivas c y n_0 tales que:

$$0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0.$$

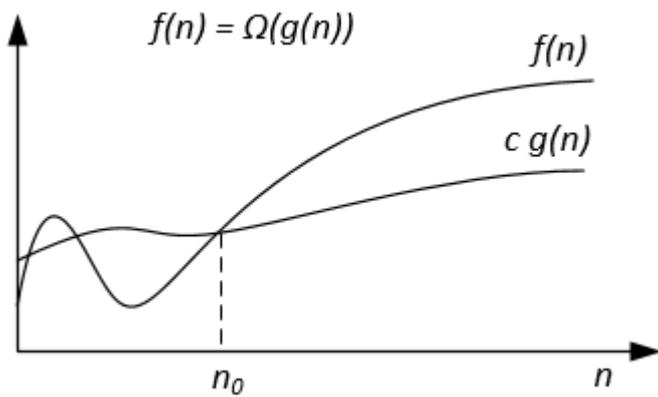
Notas

$f(n) = \Omega(g(n))$ significa que $f(n)$ crece asintóticamente no más lento que $g(n)$. También podemos decir acerca de $\Omega(g(n))$ cuando el análisis de algoritmos no es suficiente para la declaración sobre $\Theta(g(n))$ y $O(g(n))$.

De las definiciones de notaciones sigue el teorema:

Para dos funciones de $f(n)$ $g(n)$ tenemos $f(n) = \Theta(g(n))$ si y solo si $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Gráficamente la notación Ω se puede representar de la siguiente manera:



Por ejemplo, vamos a tener $f(n) = 3n^2 + 5n - 4$. Entonces $f(n) = \Omega(n^2)$. También es correcto $f(n) = \Omega(n)$, o incluso $f(n) = \Omega(1)$.

Otro ejemplo para resolver el algoritmo de coincidencia perfecta: si el número de vértices es impar, emita "Sin coincidencia perfecta"; de lo contrario, intente todas las coincidencias posibles.

Nos gustaría decir que el algoritmo requiere tiempo exponencial, pero en realidad no puede probar un límite inferior de $\Omega(n^2)$ usando la definición habitual de Ω ya que el algoritmo se ejecuta en tiempo lineal para n impar. En su lugar, deberíamos definir $f(n) = \Omega(g(n))$ diciendo que para alguna constante $c > 0$, $f(n) \geq cg(n)$ para infinitamente muchos n . Esto proporciona una buena correspondencia entre los límites superior e inferior: $f(n) = \Omega(g(n))$ iff $f(n) \neq o(g(n))$.

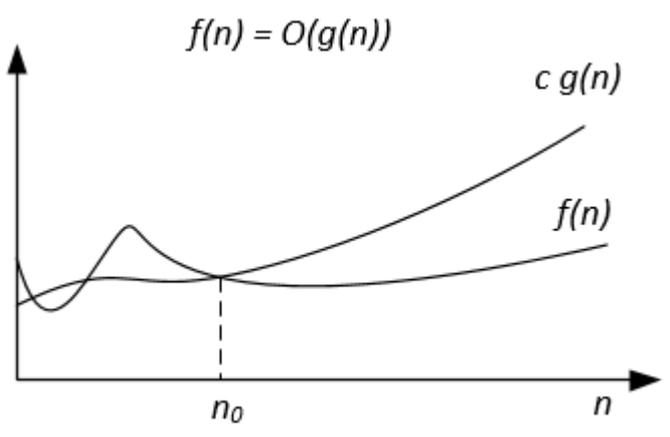
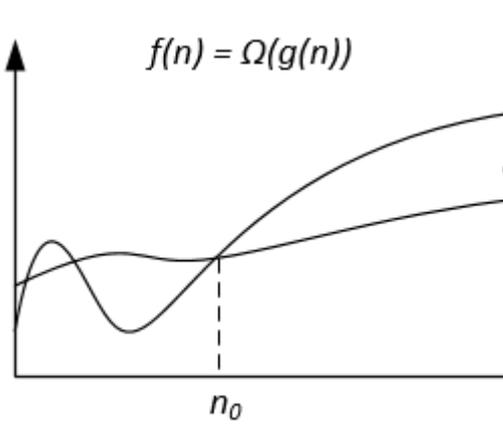
Referencias

La definición formal y el teorema están tomados del libro "Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introducción a los algoritmos".

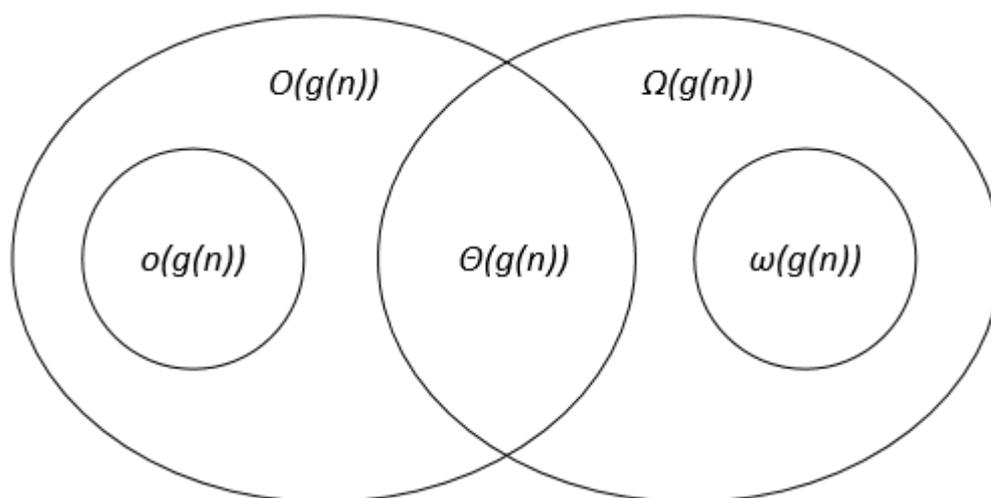
Comparación de las notaciones asintóticas.

Sean $f(n)$, $g(n)$ dos funciones definidas en el conjunto de los números reales positivos, c , c_1 , c_2 , n_0 son constantes reales positivas.

Notación	$f(n) = O(g(n))$	$f(n) = \Omega(g(n))$
Definición formal	$\exists c > 0, \exists n_0 > 0 : \forall n \geq n_0, 0 \leq f(n) \leq cg(n)$	$\exists c > 0, \exists n_0 > 0 : \forall n \geq n_0, 0 \leq cg(n) \leq f(n)$

Notación	$f(n) = O(g(n))$	$f(n) = \Omega(g(n))$
Analogía entre la comparación asintótica de f , g y números reales a , b	$a \leq b$	$a \geq b$
Ejemplo	$7n + 10 = O(n^2 + n - 9)$	$n^3 - 34 = \Omega(10n^2 - 7n + 1)$
Interpretación grafica	 <p>The graph shows a coordinate system with a vertical axis and a horizontal axis labeled n. A curve $f(n)$ is plotted. A straight line $c g(n)$ is also plotted, starting from a point n_0 on the horizontal axis. For $n > n_0$, the curve $f(n)$ remains below the line $c g(n)$. The title above the graph is $f(n) = O(g(n))$.</p>	 <p>The graph shows a coordinate system with a vertical axis and a horizontal axis labeled n. A curve $f(n)$ is plotted. A straight line $c g(n)$ is also plotted, starting from a point n_0 on the horizontal axis. For $n > n_0$, the curve $f(n)$ remains above the line $c g(n)$. The title above the graph is $f(n) = \Omega(g(n))$.</p>

Las notaciones asintóticas se pueden representar en un diagrama de Venn de la siguiente



manera:

Campo de golf

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introducción a los algoritmos.

Lea Complejidad de algoritmos en línea:

<https://riptutorial.com/es/algorithm/topic/1529/complejidad-de-algoritmos>

Capítulo 29: Compruebe que dos cadenas son anagramas

Introducción

Dos cadenas con el mismo conjunto de caracteres se llaman anagramas. He utilizado javascript aquí.

Crearemos un hash de str1 y aumentaremos la cuenta +1. Haremos un bucle en la segunda cadena y comprobaremos que todos los caracteres están en hash y disminuirémos el valor de la clave hash. Compruebe que todo el valor de la clave hash es cero será anagrama.

Examples

Muestra de entrada y salida

Ex1: -

```
let str1 = 'stackoverflow';
let str2 = 'flowerovstack';
```

Estas cuerdas son anagramas.

// Crear Hash desde str1 y aumentar una cuenta.

```
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
}
```

Puede ver que la hashKey 'o' contiene el valor 2 porque o es 2 veces en una cadena.

Ahora pase sobre str2 y verifique que cada carácter esté presente en hashMap, si es así, reduzca el valor de la clave hashMap, de lo contrario, devuelva false (lo que indica que no es un anagrama).

```
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
}

```

Ahora, haga un bucle sobre el objeto hashMap y verifique que todos los valores sean cero en la clave de hashMap.

En nuestro caso, todos los valores son cero, por lo que es un anagrama.

Código genérico para anagramas

```

(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

})();

```

Complejidad del tiempo: - $3n$ es decir $O(n)$.

Lea [Compruebe que dos cadenas son anagramas en línea](https://riptutorial.com/es/algorithm/topic/9970/compruebe-que-dos-cadenas-son-anagramas):

<https://riptutorial.com/es/algorithm/topic/9970/compruebe-que-dos-cadenas-son-anagramas>

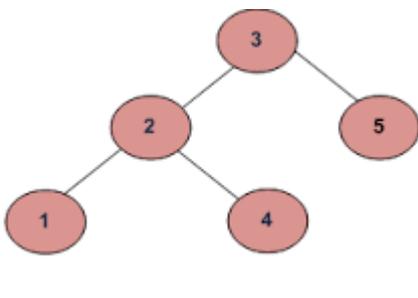
Capítulo 30: Compruebe si un árbol es BST o no

Examples

Si un árbol de entrada dado sigue una propiedad del árbol de búsqueda binaria o no

Por ejemplo

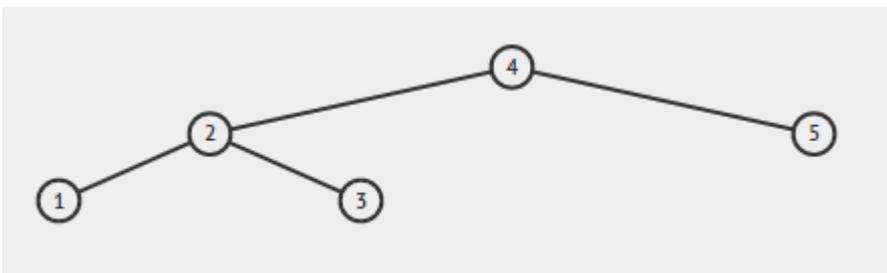
si la entrada es:



La salida debe ser falsa:

Como 4 en el subárbol izquierdo es mayor que el valor raíz (3)

Si la entrada es:



La salida debe ser verdadera

Algoritmo para verificar si un árbol binario dado es BST

Un árbol binario es BST si satisface cualquiera de las siguientes condiciones:

1. Esta vacío
2. No tiene subárboles.
3. Para cada nodo x en el árbol, todas las claves (si las hay) en el subárbol izquierdo deben ser menores que la clave (x) y todas las claves (si las hay) en el subárbol derecho deben ser mayores que la clave (x).

Entonces, un algoritmo recursivo directo sería:

```
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)
```

El algoritmo recursivo anterior es correcto pero ineficiente, ya que atraviesa cada nodo varios tiempos.

Otro enfoque para minimizar las visitas múltiples de cada nodo es recordar los valores mínimos y máximos posibles de las claves en el subárbol que estamos visitando. Deje que el valor mínimo posible de cualquier clave sea K_{MIN} y el valor máximo sea K_{MAX} . Cuando comenzamos desde la raíz del árbol, el rango de valores en el árbol es $[K_{MIN}, K_{MAX}]$. Deje que la clave del nodo raíz sea x . Luego, el rango de valores en el subárbol izquierdo es $[K_{MIN}, x)$ y el rango de valores en el subárbol derecho es $(x, K_{MAX}]$. Usaremos esta idea para desarrollar un algoritmo más eficiente.

```
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)
```

Se llamará inicialmente como:

```
is_BST(my_tree_root, KEY_MIN, KEY_MAX)
```

Otro enfoque será hacer un recorrido ordenado del árbol binario. Si el recorrido inorder produce una secuencia ordenada de claves, entonces el árbol dado es un BST. Para verificar si la secuencia inorder está ordenada, recuerde el valor del nodo visitado anteriormente y compárelo con el nodo actual.

Lea [Compruebe si un árbol es BST o no en línea](https://riptutorial.com/es/algorithm/topic/8840/compruebe-si-un-arbol-es-bst-o-no):

<https://riptutorial.com/es/algorithm/topic/8840/compruebe-si-un-arbol-es-bst-o-no>

Capítulo 31: Editar distancia del algoritmo dinámico

Examples

Ediciones mínimas requeridas para convertir la cadena 1 a la cadena 2

La declaración del problema es como si nos dieran dos cadenas `str1` y `str2`, entonces, ¿cuántos números mínimos de operaciones se pueden realizar en el `str1` que se convierte en `str2`. Las operaciones pueden ser:

1. Insertar
2. retirar
3. Reemplazar

Por ejemplo

```
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
```

Para resolver este problema usaremos una matriz 2D $dp[n + 1][m + 1]$ donde n es la longitud de la primera cadena y m es la longitud de la segunda cadena. Para nuestro ejemplo, si `str1` es **azcef** y `str2` es **abcdef**, nuestra matriz será $dp[6][7]$ y nuestra respuesta final se almacenará en $dp[5][6]$.

	(a)	(b)	(c)	(d)	(e)	(f)
(a)	1					
(z)	2					
(c)	3					
(e)	4					
(f)	5					

Para $dp[1][1]$ tenemos que comprobar lo que podemos hacer para convertir **una** en **una**. Es **0**. Para $dp[1][2]$ tenemos que comprobar qué podemos hacer para convertir **una** en **ab**. Es **1** porque tenemos que **insertar b**. Entonces, después de la primera iteración, nuestra matriz se verá como

	(a)	(b)	(c)	(d)	(e)	(f)
	0	1	2	3	4	5
(a)	1	0	1	2	3	4
(z)	2					
(c)	3					
(e)	4					
(f)	5					

Para iteración 2

Para **dp [2] [1]** tenemos que verificar que para convertir **az** a **a** debemos eliminar **z** , por lo tanto, **dp [2] [1]** será **1**. Al igual que para **dp [2] [2]** necesitamos reemplazar **z** con **b** , por lo tanto, **dp [2] [2]** será **1**. **Por lo tanto**, después de la segunda iteración, nuestra matriz **dp []** se verá así.

	(a)	(b)	(c)	(d)	(e)	(f)
	0	1	2	3	4	5
(a)	1	0	1	2	3	4
(z)	2	1	1	2	3	4
(c)	3					
(e)	4					
(f)	5					

Entonces nuestra **fórmula** se verá como

```

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])

```

Después de la última iteración, nuestra matriz **dp []** se verá como

	(a)	(b)	(c)	(d)	(e)	(f)
	0	1	2	3	4	5
(a)	1	0	1	2	3	4
(z)	2	1	1	2	3	4
(c)	3	2	2	1	2	3
(e)	4	3	3	2	2	3

```

+---+---+---+---+---+---+---+
(f) | 5 | 4 | 4 | 2 | 3 | 3 | 3 |
+---+---+---+---+---+---+---+

```

Implementación en 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()];
}

```

Complejidad del tiempo

$O(n^2)$

Lea Editar distancia del algoritmo dinámico en línea:

<https://riptutorial.com/es/algorithm/topic/10500/editar-distancia-del-algoritmo-dinamico>

Capítulo 32: El algoritmo de Dijkstra

Examples

Algoritmo de la ruta más corta de Dijkstra

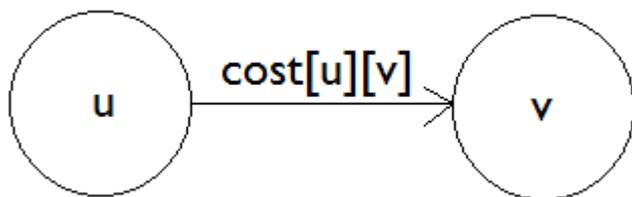
Antes de continuar, se recomienda tener una breve idea sobre la matriz de adyacencia y BFS

El algoritmo de Dijkstra se conoce como algoritmo de ruta más corta de una sola fuente. Se utiliza para encontrar las rutas más cortas entre nodos en un gráfico, que pueden representar, por ejemplo, redes de carreteras. Fue concebido por Edsger W. Dijkstra en 1956 y publicado tres años después.

Podemos encontrar la ruta más corta utilizando el algoritmo de búsqueda de búsqueda de amplitud (BFS). Este algoritmo funciona bien, pero el problema es que asume que el costo de atravesar cada ruta es el mismo, lo que significa que el costo de cada borde es el mismo. El algoritmo de Dijkstra nos ayuda a encontrar la ruta más corta donde el costo de cada ruta no es el mismo.

Al principio veremos cómo modificar BFS para escribir el algoritmo de Dijkstra, luego agregaremos una cola de prioridad para convertirlo en un algoritmo completo de Dijkstra.

Digamos que la distancia de cada nodo desde la fuente se mantiene en la matriz d . Como en, $d[3]$ representa que el tiempo $d[3]$ se toma para llegar al **nodo 3** desde la **fuentes**. Si no conocemos la distancia, almacenaremos el *infinito* en $d[3]$. Además, deje que el **costo $[u][v]$** represente el costo de uv . Eso significa que **cuesta $[u][v]$** pasar de u nodo a v nodo.



Necesitamos entender la relajación del borde. Digamos que desde su casa, esa es la **fuentes**, se necesitan 10 minutos para ir al lugar **A**. Y se tarda 25 minutos en ir al lugar **B**. Tenemos,

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

Ahora digamos que se necesitan 7 minutos para ir desde el lugar **A** al lugar **B**, eso significa:

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

Luego podemos ir al lugar **B** desde la **fuentes** yendo al lugar **A** desde la **fuentes** y luego desde el lugar **A**, yendo al lugar **B**, que tomará $10 + 7 = 17$ minutos, en lugar de 25 minutos. Así que,

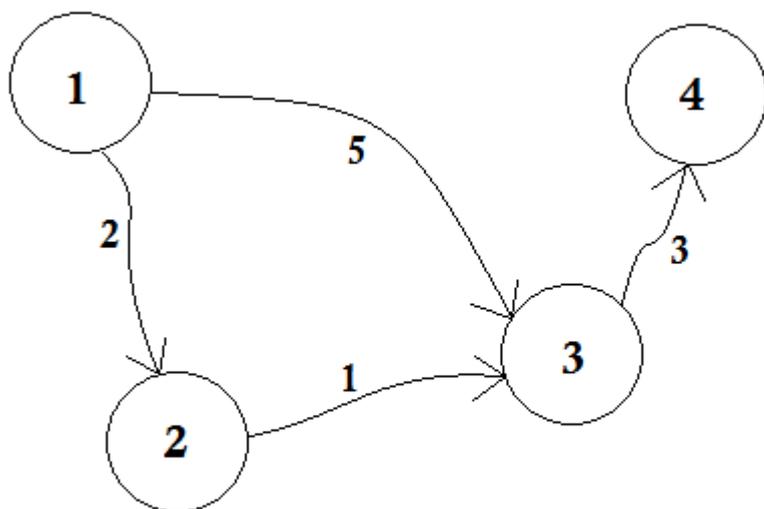
$$d[A] + \text{cost}[A][B] < d[B]$$

Luego actualizamos,

$$d[B] = d[A] + \text{cost}[A][B]$$

Esto se llama relajación. Iremos del nodo u al nodo v y si $d[u] + \text{costo}[u][v] < d[v]$ luego actualizaremos $d[v] = d[u] + \text{costo}[u][v]$.

En BFS, no necesitamos visitar ningún nodo dos veces. Solo verificamos si un nodo es visitado o no. Si no fue visitado, pusimos el nodo en la cola, lo marcamos como visitado e incrementamos la distancia en 1. En Dijkstra, podemos empujar un nodo en la cola y, en lugar de actualizarlo, visitamos, *relajamos* o actualizamos el nuevo borde. Veamos un ejemplo:



Supongamos que el **Nodo 1** es la **Fuente**. Entonces,

$$\begin{aligned} d[1] &= 0 \\ d[2] &= d[3] = d[4] = \text{infinity (or a large value)} \end{aligned}$$

Configuramos, $d[2]$, $d[3]$ y $d[4]$ al *infinito* porque todavía no sabemos la distancia. Y la distancia de la **fuentes** es, por supuesto, 0 . Ahora, vamos a otros nodos desde la **fuentes** y si podemos actualizarlos, los pondremos en la cola. Digamos por ejemplo, vamos a atravesar el **borde 1-2**. Como $d[1] + 2 < d[2]$, lo que hará que $d[2] = 2$. De manera similar, atravesaremos el **borde 1-3**, lo que hace que $d[3] = 5$.

Podemos ver claramente que 5 no es la distancia más corta que podemos cruzar para ir al **nodo 3**. Así que atravesar un nodo solo una vez, como BFS, no funciona aquí. Si vamos del **nodo 2** al **nodo 3** usando el **borde 2-3**, podemos actualizar $d[3] = d[2] + 1 = 3$. Entonces podemos ver

que un nodo puede ser actualizado muchas veces. Cuantas veces preguntas El número máximo de veces que se puede actualizar un nodo es el número de grados de un nodo.

Veamos el pseudo-código para visitar cualquier nodo varias veces. Simplemente modificaremos 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
```

Esto se puede usar para encontrar la ruta más corta de todos los nodos desde la fuente. La complejidad de este código no es tan buena. Este es el por qué,

En BFS, cuando pasamos del **nodo 1** a todos los demás nodos, seguimos el método de *primer orden de llegada* . Por ejemplo, fuimos al **nodo 3** desde la **fuentes** antes de procesar el **nodo 2** . Si vamos al **nodo 3** desde la **fuentes** , actualizamos el **nodo 4** como $5 + 3 = 8$. Cuando volvamos a actualizar el **nodo 3** desde el **nodo 2** , necesitamos actualizar el **nodo 4** como $3 + 3 = 6$ nuevamente. Así que el **nodo 4** se actualiza dos veces.

Dijkstra propuso, en lugar de ir por el método *Primero en llegar, primero en servir* , si actualizamos los nodos más cercanos primero, entonces tomará menos actualizaciones. Si hubiéramos procesado el **nodo 2** antes, entonces el **nodo 3** se habría actualizado antes, y después de actualizar el **nodo 4** en consecuencia, ¡obtendríamos fácilmente la distancia más corta! La idea es elegir de la cola, el nodo, que está más cerca de la **fuentes** . Así que usaremos la *cola de prioridad* aquí para que cuando abramos la cola, nos traiga el nodo más cercano **u** desde la **fuentes** . ¿Cómo va a hacer eso? Verificará el valor de **d [u]** con él.

Veamos el pseudo-código:

```
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
```

```
end while
Return distance
```

El pseudocódigo devuelve la distancia de todos los demás nodos desde la **fuentes** . Si queremos conocer la distancia de un solo nodo **v** , simplemente podemos devolver el valor cuando **v** se extrae de la cola.

Ahora, ¿el algoritmo de Dijkstra funciona cuando hay una ventaja negativa? Si hay un ciclo negativo, entonces ocurrirá un bucle infinito, ya que seguirá reduciendo el costo cada vez. Incluso si hay una ventaja negativa, Dijkstra no funcionará, a menos que regresemos justo después de que se haga estallar el objetivo. Pero entonces, no será un algoritmo de Dijkstra. Necesitaremos el algoritmo de **Bellman-Ford** para procesar el borde / ciclo negativo.

Complejidad:

La complejidad de BFS es **$O(\log(V + E))$** donde **V** es el número de nodos y **E** es el número de bordes. Para Dijkstra, la complejidad es similar, pero la clasificación de la *cola de prioridad* toma **$O(\log V)$** . Entonces la complejidad total es: **$O(V \log(V) + E)$**

A continuación se muestra un ejemplo de Java para resolver el algoritmo de ruta más corta de Dijkstra usando la matriz de adyacencia

```
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);
}
}

```

La salida esperada del programa es

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

Lea El algoritmo de Dijkstra en línea: <https://riptutorial.com/es/algorithm/topic/7151/el-algoritmo-de-dijkstra>

Capítulo 33: El algoritmo de Kruskal

Observaciones

El algoritmo de Kruskal es un algoritmo **codicioso que se** utiliza para encontrar el **árbol de expansión mínimo (MST)** de un gráfico. Un árbol de expansión mínimo es un árbol que conecta todos los vértices de la gráfica y tiene el peso total mínimo del borde.

El algoritmo de Kruskal lo hace al seleccionar repetidamente bordes con **un peso mínimo** (que aún no están en el MST) y agregarlos al resultado final si los dos vértices conectados por ese borde aún no están conectados en el MST, de lo contrario, se salta ese borde. Unión: la estructura de datos de búsqueda se puede usar para verificar si dos vértices ya están conectados en el MST o no. Algunas propiedades de MST son las siguientes:

1. Un MST de una gráfica con n vértices tendrá exactamente $n-1$ aristas.
2. Existe una ruta única de cada vértice a cada otro vértice.

Examples

Implementación simple, más detallada.

Para manejar de manera eficiente la detección de ciclos, consideramos cada nodo como parte de un árbol. Al agregar un borde, verificamos si sus dos nodos componentes forman parte de árboles distintos. Inicialmente, cada nodo forma un árbol de un nodo.

```
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
```

Implementación simple, basada en conjuntos disjuntos

La metodología forestal anterior es en realidad una estructura de datos de conjuntos disjuntos, que involucra tres operaciones principales:

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

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

```

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

    uRoot.parent = vRoot

algoritmo 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)

```

Esta implementación ingenua lleva al tiempo $O(n \log n)$ para administrar la estructura de datos del conjunto disjunto, lo que lleva al tiempo $O(m \cdot n \log n)$ para todo el algoritmo de Kruskal.

Implementación óptima, basada en conjuntos disjuntos

Podemos hacer dos cosas para mejorar los subalgoritmos de conjuntos disjuntos simples y subóptimos:

1. **Heurística de compresión de ruta** : `findSet` no necesita manejar un árbol con una altura mayor a 2 . Si termina iterando tal árbol, puede vincular los nodos inferiores directamente a la raíz, optimizando futuros recorridos;

```

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

```

2. **Heurística de fusión basada en altura** : para cada nodo, almacene la altura de su subárbol. Al fusionar, convierta al árbol más alto en el padre del más pequeño, por lo que no aumentará la altura de nadie.

```

subalgoritmo 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

```

Esto lleva al tiempo de $O(\alpha(n))$ para cada operación, donde α es el inverso de la función de Ackermann de rápido crecimiento, por lo que su crecimiento es muy lento y puede considerarse $O(1)$ por motivos prácticos.

Esto hace que todo el algoritmo de Kruskal $O(m \log m + m) = O(m \log m)$, debido a la clasificación inicial.

Nota

La compresión del camino puede reducir la altura del árbol, por lo tanto, comparar las alturas de los árboles durante la operación de unión podría no ser una tarea trivial. Por lo tanto, para evitar la complejidad de almacenar y calcular la altura de los árboles, el padre resultante se puede seleccionar al azar:

```
subalgoritmo 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
```

En la práctica, este algoritmo aleatorio junto con la compresión de ruta para la operación `findSet` dará como resultado un rendimiento comparable, pero mucho más sencillo de implementar.

Implementación simple y de alto nivel.

Ordene los bordes por valor y agregue cada uno al MST en orden ordenado, si no crea un ciclo.

```
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
```

Lea El algoritmo de Kruskal en línea: <https://riptutorial.com/es/algorithm/topic/2583/el-algoritmo-de-kruskal>

Capítulo 34: El ancestro común más bajo de un árbol binario

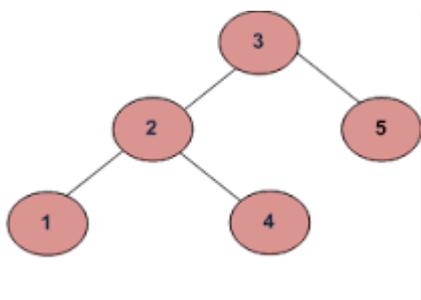
Introducción

El antepasado común más bajo entre dos nodos n_1 y n_2 se define como el nodo más bajo en el árbol que tiene n_1 y n_2 como descendientes.

Examples

Encontrar el antepasado común más bajo

Considera el árbol:



El antecesor común más bajo de los nodos con valor 1 y 4 es 2

El antecesor común más bajo de los nodos con valor 1 y 5 es 3

El antecesor común más bajo de los nodos con valor 2 y 4 es 2

El antecesor común más bajo de los nodos con valor 1 y 2 es 2

Lea [El ancestro común más bajo de un árbol binario en línea](https://riptutorial.com/es/algorithm/topic/8848/el-ancestro-comun-mas-bajo-de-un-arbol-binario):

<https://riptutorial.com/es/algorithm/topic/8848/el-ancestro-comun-mas-bajo-de-un-arbol-binario>

Capítulo 35: El problema más corto de la supersecuencia

Examples

Información básica sobre el problema de la supersecuencia más corta

La **súper secuencia más corta** es un problema estrechamente relacionado con la subsecuencia común más larga, que puede utilizar como una función externa para esta tarea. El problema de súper secuencia común más corto es un problema estrechamente relacionado con el problema de subsecuencia común más largo.

Una supersecuencia más corta (scs) es una supersecuencia común de longitud mínima. En el problema más corto de supersecuencia común, se dan las dos secuencias x e y , y la tarea es encontrar una súper secuencia común lo más corta posible de estas secuencias. En general, un scs no es único.

Dadas dos secuencias $x = \langle x_1, \dots, x_m \rangle$ e $y = \langle y_1, \dots, y_n \rangle$, una secuencia $u = \langle u_1, \dots, u_k \rangle$ es una super secuencia común de x e y si u es una super secuencia de x e y . En otras palabras, una secuencia de súper común más corto de cadenas x y y es una cadena más corta z de tal manera que ambos x y y son subsecuencias de z .

Para dos secuencias de entrada, un scs puede formarse a partir de una subsecuencia común más larga (lcs) fácilmente. Por ejemplo, si $X[1..m]=abcbabdab$ e $Y[1..n]=bdcaba$, el lcs es $Z[1..r]=bcba$. Al insertar los símbolos no-lcs mientras se conserva el orden de los símbolos, obtenemos los SCS: $U[1..t]=abdcabdab$.

Es bastante claro que $r+t=m+n$ para dos secuencias de entrada. Sin embargo, para tres o más secuencias de entrada esto no es válido. Tenga en cuenta también que los problemas de lcs y scs no son problemas duales.

Para el problema más general de encontrar una cadena, s , que es una supercadena de un conjunto de cadenas s_1, s_2, \dots, s_l , el problema es NP-Completo. Además, se pueden encontrar buenas aproximaciones para el caso promedio pero no para el peor de los casos.

Ejemplo de problema de supersecuencia más corto:

	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

Complejidad de tiempo: $O(\max(m, n))$

Implementación del problema más corto de supersecuencia en 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);
}
```

Lea [El problema más corto de la supersecuencia en línea](https://riptutorial.com/es/algorithm/topic/7604/el-problema-mas-corto-de-la-supersecuencia):

<https://riptutorial.com/es/algorithm/topic/7604/el-problema-mas-corto-de-la-supersecuencia>

Capítulo 36: Exposición de matrices

Examples

Exposición de matrices para resolver problemas de ejemplo

Encuentre $f(n)$: n *ésimo* número de Fibonacci. El problema es bastante fácil cuando n es relativamente pequeño. Podemos usar la recursión simple, $f(n) = f(n-1) + f(n-2)$, o podemos usar el enfoque de programación dinámica para evitar el cálculo de la misma función una y otra vez. Pero, ¿qué harás si el problema dice: **Dado $0 < n < 10^9$, encuentra $f(n) \bmod 999983$** ? La programación dinámica fallará, entonces, ¿cómo abordamos este problema?

Primero veamos cómo la exponenciación matricial puede ayudar a representar una relación recursiva.

Requisitos previos:

- Dadas dos matrices, saber encontrar su producto. Además, dada la matriz de producto de dos matrices, y una de ellas, sabe cómo encontrar la otra matriz.
- Dada una matriz de tamaño $d \times d$, saber cómo encontrar su n *ésima* potencia en $O(d^3 \log(n))$.

Patrones:

Al principio necesitamos una relación recursiva y queremos encontrar una matriz M que pueda llevarnos al estado deseado desde un conjunto de estados ya conocidos. Supongamos que, conocemos los k estados de una relación de recurrencia dada y queremos encontrar el estado $(k + 1)^{th}$. Sea M una matriz $k \times k$, y construimos una matriz $A: [k \times 1]$ a partir de los estados conocidos de la relación de recurrencia, ahora queremos obtener una matriz $B: [k \times 1]$ que representará el conjunto de estados siguientes, es decir, $MA = B$ como se muestra a continuación:

$$M \times \begin{bmatrix} f(n) \\ f(n-1) \\ f(n-2) \\ \dots \\ f(n-k) \end{bmatrix} = \begin{bmatrix} f(n+1) \\ f(n) \\ f(n-1) \\ \dots \\ f(n-k+1) \end{bmatrix}$$

Entonces, si podemos diseñar M en consecuencia, ¡nuestro trabajo estará listo! La matriz se utilizará para representar la relación de recurrencia.

Tipo 1:

Comencemos con el más simple, $f(n) = f(n-1) + f(n-2)$

Obtenemos, $f(n+1) = f(n) + f(n-1)$.

Supongamos que sabemos $f(n)$ y $f(n-1)$; Queremos averiguar $f(n+1)$.

A partir de la situación mencionada anteriormente, la matriz A y la matriz B se pueden formar como se muestra a continuación:

Matrix A Matrix B

$$\begin{array}{|c|} \hline f(n) \\ \hline f(n-1) \\ \hline \end{array} \quad \begin{array}{|c|} \hline f(n+1) \\ \hline f(n) \\ \hline \end{array}$$

[Nota: la matriz **A** siempre se diseñará de tal manera que, todos los estados de los que depende $f(n+1)$, estarán presentes]

Ahora, necesitamos diseñar una matriz **M 2X2** tal que satisfaga $MXA = B$ como se indicó anteriormente.

El primer elemento de **B** es $f(n+1)$ que en realidad es $f(n) + f(n-1)$. Para obtener esto, de la matriz **A**, necesitamos $1 \times f(n)$ y $1 \times f(n-1)$. Así que la primera fila de **M** será $[1 \ 1]$.

$$\begin{array}{|c|c|} \hline 1 & 1 \\ \hline \end{array} \times \begin{array}{|c|} \hline f(n) \\ \hline f(n-1) \\ \hline \end{array} = \begin{array}{|c|} \hline f(n+1) \\ \hline \end{array}$$

[Nota: ----- significa que no estamos preocupados por este valor.]

De manera similar, el segundo elemento de **B** es $f(n)$ que puede obtenerse simplemente tomando $1 \times f(n)$ de **A**, por lo que la segunda fila de **M** es $[1 \ 0]$.

$$\begin{array}{|c|c|} \hline ----- & \\ \hline 1 & 0 \\ \hline \end{array} \times \begin{array}{|c|} \hline f(n) \\ \hline f(n-1) \\ \hline \end{array} = \begin{array}{|c|} \hline f(n) \\ \hline \end{array}$$

Entonces conseguimos nuestra deseada **2 X 2 M** de la matriz.

$$\begin{array}{|c|c|} \hline 1 & 1 \\ \hline 1 & 0 \\ \hline \end{array} \times \begin{array}{|c|} \hline f(n) \\ \hline f(n-1) \\ \hline \end{array} = \begin{array}{|c|} \hline f(n+1) \\ \hline f(n) \\ \hline \end{array}$$

Estas matrices se derivan simplemente utilizando la multiplicación de matrices.

Tipo 2:

Hagámoslo un poco complejo: encuentre $f(n) = a \times f(n-1) + b \times f(n-2)$, donde **a** y **b** son constantes.

Esto nos dice que $f(n+1) = a \times f(n) + b \times f(n-1)$.

Hasta este punto, esto debe quedar claro que la dimensión de las matrices será igual al número de dependencias, es decir, en este ejemplo particular, nuevamente 2. Entonces, para **A** y **B**, podemos construir dos matrices de tamaño **2 X 1**:

$$\begin{array}{|c|} \hline f(n) \\ \hline f(n-1) \\ \hline \end{array} \quad \begin{array}{|c|} \hline f(n+1) \\ \hline f(n) \\ \hline \end{array}$$

Ahora para $f(n+1) = a \times f(n) + b \times f(n-1)$, necesitamos $[a, b]$ en la primera fila de la matriz objetivo **M**. Y para el segundo elemento en **B**, es decir, $f(n)$ ya tenemos eso en la matriz **A**, así que simplemente tomamos eso, lo que lleva a la segunda fila de la matriz **M** a $[1 \ 0]$. Esta vez tenemos:

$$\begin{array}{|c|c|} \hline a & b \\ \hline 1 & 0 \\ \hline \end{array} \times \begin{array}{|c|} \hline f(n) \\ \hline f(n-1) \\ \hline \end{array} = \begin{array}{|c|} \hline f(n+1) \\ \hline f(n) \\ \hline \end{array}$$

Bastante simple, ¿eh?

Tipo 3:

Si ha sobrevivido hasta esta etapa, ha envejecido mucho, ahora enfrentemos una relación un poco compleja: encontrar $f(n) = a \times f(n-1) + c \times f(n-3)$?

Ups! Hace unos minutos, todo lo que vimos fueron estados contiguos, pero aquí falta el estado $f(n-2)$. ¿Ahora?

En realidad esto ya no es un problema, podemos convertir la relación de la siguiente manera: $f(n) = a \times f(n-1) + 0 \times f(n-2) + c \times f(n-3)$, deduciendo $f(n+1) = a \times f(n) + 0 \times f(n-1) + c \times f(n-2)$. Ahora, vemos que esto es en realidad una forma descrita en el Tipo 2. Así que aquí la matriz objetivo M será 3×3 , y los elementos son:

$$\begin{array}{ccc|ccc} | a & 0 & c & | & f(n) & | & | f(n+1) & | \\ | 1 & 0 & 0 & | & X & | & f(n-1) & | = & | f(n) & | \\ | 0 & 1 & 0 & | & & | & f(n-2) & | & | f(n-1) & | \end{array}$$

Estos se calculan de la misma manera que en el tipo 2, si le resulta difícil, pruébelo con lápiz y papel.

Tipo 4:

La vida se está volviendo compleja, y el Sr. Problema ahora te pide que encuentres $f(n) = f(n-1) + f(n-2) + c$ donde c es cualquier constante.

Ahora, este es uno nuevo y todo lo que hemos visto en el pasado, después de la multiplicación, cada estado en A se transforma en su próximo estado en B .

$$\begin{array}{l} f(n) = f(n-1) + f(n-2) + c \\ f(n+1) = f(n) + f(n-1) + c \\ f(n+2) = f(n+1) + f(n) + c \\ \dots\dots\dots \text{so on} \end{array}$$

Entonces, normalmente no podemos hacerlo a través de la moda anterior, pero ¿qué tal si agregamos c como un estado:

$$\begin{array}{ccc|ccc} & | f(n) & | & | f(n+1) & | \\ M \times & | f(n-1) & | = & | f(n) & | \\ & | c & | & | c & | \end{array}$$

Ahora, no es mucho difícil diseñar M . Así es como se hace, pero no te olvides de verificar:

$$\begin{array}{ccc|ccc} | 1 & 1 & 1 & | & f(n) & | & | f(n+1) & | \\ | 1 & 0 & 0 & | & X & | & f(n-1) & | = & | f(n) & | \\ | 0 & 0 & 1 & | & & | & c & | & | c & | \end{array}$$

Tipo 5:

Pongámoslo del todo: encuentre $f(n) = a \times f(n-1) + c \times f(n-3) + d \times f(n-4) + e$. Vamos a dejarlo como un ejercicio para ti. Primero intenta averiguar los estados y la matriz M . Y

comprueba si coincide con tu solución. También encuentra la matriz **A** y **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 |
```

Tipo 6:

A veces la recurrencia se da así:

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

En breve:

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

Aquí, podemos dividir las funciones en la base de pares impares y mantener 2 matrices diferentes para ambas y calcularlas por separado.

Tipo 7:

¿Te sientes un poco demasiado seguro? Bien por usted. A veces es posible que tengamos que mantener más de una recurrencia, cuando estén interesados. Por ejemplo, deje que una recurrencia re; atopm sea:

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

Aquí, la recurrencia $g(n)$ depende de $f(n)$ y esto se puede calcular en la misma matriz pero de dimensiones aumentadas. A partir de estos, primero diseñemos las matrices **A** y **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)

Aquí, $g(n+1) = 2g(n-1) + f(n+1)$ and $f(n+2) = 2f(n+1) + 2f(n)$. Ahora, utilizando los procesos mencionados anteriormente, podemos encontrar que la matriz objetivo **M** es:

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

Por lo tanto, estas son las categorías básicas de relaciones de recurrencia que se utilizan para resolver con esta técnica simple.

Lea Exposición de matrices en línea: <https://riptutorial.com/es/algorithm/topic/7290/exposicion-de-matrices>

Capítulo 37: Funciones hash

Examples

Introducción a las funciones hash.

La función hash $h()$ es una función arbitraria que mapea los datos $x \in X$ de tamaño arbitrario para valorar $y \in Y$ de tamaño fijo: $y = h(x)$. Las buenas funciones de hash tienen restricciones siguientes:

- Las funciones hash se comportan como distribución uniforme
- Las funciones hash son deterministas. $h(x)$ siempre debe devolver el mismo valor para una x dada
- cálculo rápido (tiene tiempo de ejecución $O(1)$)

En general, el tamaño de la función hash es menor que el tamaño de los datos de entrada: $|Y| < |X|$. Las funciones de hash no son reversibles o, en otras palabras, pueden ser colisiones: $\exists x_1, x_2 \in X, x_1 \neq x_2: h(x_1) = h(x_2)$. X puede ser un conjunto finito o infinito y Y es un conjunto finito.

Las funciones de hash se utilizan en muchas partes de la informática, por ejemplo, en ingeniería de software, criptografía, bases de datos, redes, aprendizaje automático, etc. Hay muchos tipos diferentes de funciones hash, con diferentes propiedades específicas de dominio.

A menudo, hash es un valor entero. Existen métodos especiales en los lenguajes de programación para el cálculo de hash. Por ejemplo, en C# método `GetHashCode()` C# para todos los tipos, se devuelve el valor `Int32` (número entero de 32 bits). En Java cada clase proporciona el método `hashCode()` que devuelve `int`. Cada tipo de datos tiene implementaciones propias o definidas por el usuario.

Métodos hash

Hay varios enfoques para determinar la función hash. Sin pérdida de generalidad, sea $x \in X = \{z \in \mathbb{Z}: z \geq 0\}$ son números enteros positivos. A menudo m es primo (no demasiado cerca de una potencia exacta de 2).

Método	Función hash
Método de división	$h(x) = x \bmod m$
Método de multiplicación	$h(x) = \lfloor m (xA \bmod 1) \rfloor, A \in \{z \in \mathbb{R}: 0 < z < 1\}$

Tabla de picadillo

Funciones de hash usadas en tablas hash para calcular el índice en una matriz de ranuras. La tabla de hash es una estructura de datos para implementar diccionarios (estructura clave-valor). Las buenas tablas hash implementadas tienen $O(1)$ tiempo para las siguientes operaciones: insertar, buscar y eliminar datos por clave. Más de una de las teclas pueden ser hash en la misma ranura. Hay dos formas de resolver la colisión:

1. Encadenamiento: la lista enlazada se usa para almacenar elementos con el mismo valor hash en la ranura
2. Direccionamiento abierto: cero o un elemento se almacena en cada ranura

Los siguientes métodos se utilizan para calcular las secuencias de sondeo requeridas para el direccionamiento abierto

Método	Fórmula
Sondeo lineal	$h(x, i) = (h'(x) + i) \bmod m$
Sondeo cuadrático	$h(x, i) = (h'(x) + c_1*i + c_2*i^2) \bmod m$
Doble hashing	$h(x, i) = (h_1(x) + i*h_2(x)) \bmod m$

Donde $i \in \{0, 1, \dots, m-1\}$, $h'(x)$, $h_1(x)$, $h_2(x)$ son funciones hash auxiliares, c_1 , c_2 son constantes auxiliares positivas.

Ejemplos

Permite $x \in U\{1, 1000\}$, $h = x \bmod m$. La siguiente tabla muestra los valores de hash en caso de que no sean primos y primos. El texto en negrita indica los mismos valores hash.

X	m = 100 (no primo)	m = 101 (prime)
723	23	dieciséis
103	3	2
738	38	31
292	92	90
61	61	61
87	87	87
995	95	86

X	m = 100 (no primo)	m = 101 (prime)
549	49	44
991	91	82
757	57	50
920	20	11
626	26	20
557	57	52
831	31	23
619	19	13

Campo de golf

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introducción a los algoritmos.
- [Resumen de las tablas de hash](#)
- [Wolfram MathWorld - Hash Function](#)

Códigos hash para tipos comunes en C

Los códigos hash producidos por el método `GetHashCode()` para los tipos C # [incorporados](#) y comunes del espacio de nombres del `System` se muestran a continuación.

Booleano

1 si el valor es verdadero, 0 de lo contrario.

Byte , UInt16 , Int32 , UInt32 , Single

Valor (si es necesario casteado a Int32).

SByte

```
((int)m_value ^ (int)m_value << 8);
```

Carbonizarse

```
(int)m_value ^ ((int)m_value << 16);
```

Int16

```
((int)((ushort)m_value) ^ (((int)m_value) << 16));
```

Int64 , doble

Xor entre los 32 bits inferior y superior del número de 64 bits

```
(unchecked((int)((long)m_value)) ^ (int)(m_value >> 32));
```

UInt64 , DateTime , TimeSpan

```
((int)m_value) ^ (int)(m_value >> 32);
```

Decimal

```
((int *)&dbl)[0] & 0xFFFFFFFF ^ ((int *)&dbl)[1];
```

Objeto

```
RuntimeHelpers.GetHashCode(this);
```

La implementación por defecto se utiliza el [índice de bloque de sincronización](#) .

Cuerda

El cálculo del código hash depende del tipo de plataforma (Win32 o Win64), la característica de usar hashing de cadenas aleatorias, el modo Debug / Release. En caso de plataforma 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);
```

Tipo de valor

El primer campo no estático es buscar y obtener su código hash. Si el tipo no tiene campos no estáticos, se devuelve el código hash del tipo. El código hash de un miembro estático no se

puede tomar porque si ese miembro es del mismo tipo que el tipo original, el cálculo termina en un bucle infinito.

Nullable <T>

```
return hasValue ? value.GetHashCode() : 0;
```

Formación

```
int ret = 0;
for (int i = (Length >= 8 ? Length - 8 : 0); i < Length; i++)
{
    ret = ((ret << 5) + ret) ^ comparer.GetHashCode(GetValue(i));
}
```

Referencias

- [GitHub .Net Core CLR](#)

Lea Funciones hash en línea: <https://riptutorial.com/es/algorithm/topic/6204/funciones-hash>

Capítulo 38: Grafico

Introducción

Un gráfico es una colección de puntos y líneas que conectan algunos subconjuntos (posiblemente vacíos) de ellos. Los puntos de un gráfico se denominan vértices del gráfico, "nodos" o simplemente "puntos". De manera similar, las líneas que conectan los vértices de un gráfico se denominan bordes de gráfico, "arcos" o "líneas".

Un gráfico G se puede definir como un par (V, E) , donde V es un conjunto de vértices y E es un conjunto de bordes entre los vértices $E \subseteq \{(u, v) \mid u, v \in V\}$.

Observaciones

Los gráficos son una estructura matemática que modela conjuntos de objetos que pueden o no estar conectados con miembros de conjuntos de bordes o enlaces.

Una gráfica se puede describir a través de dos conjuntos diferentes de objetos matemáticos:

- Un conjunto de **vértices** .
- Un conjunto de **aristas** que conectan pares de vértices.

Las gráficas pueden ser dirigidas o no dirigidas.

- **Los gráficos dirigidos** contienen bordes que se "conectan" de una sola manera.
- **Los gráficos no dirigidos** contienen solo bordes que conectan automáticamente dos vértices juntos en ambas direcciones.

Examples

Clasificación topológica

Un ordenamiento topológico, o una ordenación topológica, ordena los vértices en un gráfico acíclico dirigido en una línea, es decir, en una lista, de manera que todos los bordes dirigidos van de izquierda a derecha. Tal ordenamiento no puede existir si la gráfica contiene un ciclo dirigido porque no hay manera de que pueda continuar en una línea y aún regresar a donde comenzó.

Formalmente, en una gráfica $G = (V, E)$, entonces un ordenamiento lineal de todos sus vértices es tal que si G contiene un borde $(u, v) \in E$ desde el vértice u al vértice v entonces u precede a v en el ordenamiento.

Es importante tener en cuenta que cada DAG tiene *al menos un* ordenamiento topológico.

Hay algoritmos conocidos para construir un ordenamiento topológico de cualquier DAG en tiempo lineal, un ejemplo es:

1. Llame a `depth_first_search(G)` para calcular los tiempos de finalización v_f para cada vértice v
2. Cuando haya terminado cada vértice, insértelo en el frente de una lista vinculada
3. La lista enlazada de vértices, ya que ahora está ordenada.

Se puede realizar una ordenación topológica en el tiempo $\mathcal{O}(V + E)$, ya que el algoritmo de búsqueda primero en profundidad toma el tiempo $\mathcal{O}(V + E)$ y toma $\mathcal{O}(1)$ (tiempo constante) para insertar cada uno de $|V|$ vértices en el frente de una lista vinculada.

Muchas aplicaciones utilizan gráficos acíclicos dirigidos para indicar las precedencias entre los eventos. Utilizamos la ordenación topológica para obtener un orden para procesar cada vértice antes de cualquiera de sus sucesores.

Los vértices en una gráfica pueden representar tareas a realizar y los bordes pueden representar restricciones que una tarea debe realizar antes que otra; un ordenamiento topológico es una secuencia válida para realizar el conjunto de tareas de las tareas descritas en v

Ejemplo de problema y su solución

Deje que un vértice v describa una `Task(hours_to_complete: int)`, es decir, la `Task(4)` describe una `Task` que toma 4 horas para completar, y un borde e describe un `Cooldown(hours: int)` tal que `Cooldown(3)` describe una duración de tiempo para enfriarse después de una tarea completada.

Que nuestro gráfico se llame `dag` (ya que es un gráfico acíclico dirigido), y que contenga 5 vértices:

```
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));
```

donde conectamos los vértices con bordes dirigidos de modo que el gráfico sea acíclico,

```
// 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));
```

entonces hay tres posibles ordenamientos topológicos entre A y E ,

1. $A \rightarrow B \rightarrow D \rightarrow E$
2. $A \rightarrow C \rightarrow D \rightarrow E$
3. $A \rightarrow C \rightarrow E$

Algoritmo de Thorup

El algoritmo de Thorup para la ruta más corta de una sola fuente para el gráfico no dirigido tiene la complejidad de tiempo $O(m)$, más baja que Dijkstra.

Las ideas básicas son las siguientes. (Lo siento, no intenté implementarlo todavía, así que podría faltar algunos detalles menores. Y el papel original está bloqueado, así que intenté reconstruirlo a partir de otras fuentes que lo mencionan. Por favor, elimine este comentario si lo pudo verificar).

- Hay formas de encontrar el árbol de expansión en $O(m)$ (no se describe aquí). Debe "hacer crecer" el árbol de expansión desde el borde más corto hasta el más largo, y sería un bosque con varios componentes conectados antes de que crezca completamente.
- Seleccione un número entero b ($b \geq 2$) y solo considere los bosques extensivos con el límite de longitud b^k . Combine los componentes que son exactamente iguales pero con k diferente, y llame al mínimo k el nivel del componente. Luego lógicamente hacer componentes en un árbol. u es el padre de v iff u es el componente más pequeño distinto de v que contiene completamente v . La raíz es la gráfica completa y las hojas son vértices simples en la gráfica original (con el nivel de infinito negativo). El árbol todavía tiene solo nodos $O(n)$.
- Mantenga la distancia de cada componente a la fuente (como en el algoritmo de Dijkstra). La distancia de un componente con más de un vértice es la distancia mínima de sus hijos no expandidos. Establezca la distancia del vértice de origen a 0 y actualice los ancestros en consecuencia.
- Considere las distancias en base b . Cuando visite un nodo en el nivel k la primera vez, coloque a sus hijos en cubos compartidos por todos los nodos del nivel k (como en la clasificación de cubos, reemplazando el montón en el algoritmo de Dijkstra) por el dígito k y mayor de su distancia. Cada vez que visite un nodo, considere solo sus primeros b cubos, visite y elimine cada uno de ellos, actualice la distancia del nodo actual y vuelva a vincular el nodo actual a su propio padre utilizando la nueva distancia y espere la próxima visita para la siguiente visita cubos
- Cuando se visita una hoja, la distancia actual es la distancia final del vértice. Expanda todos los bordes en el gráfico original y actualice las distancias según corresponda.
- Visite el nodo raíz (gráfico completo) varias veces hasta llegar al destino.

Se basa en el hecho de que no hay un borde con una longitud menor que l entre dos componentes conectados del bosque de expansión con una limitación de longitud l , por lo que, a partir de la distancia x , puede enfocarse solo en un componente conectado hasta llegar a la distancia $x + l$. Visitará algunos vértices antes de que se vean todos los vértices con distancias más cortas, pero eso no importa porque se sabe que no habrá un camino más corto hasta aquí desde esos vértices. Otras partes funcionan como la clasificación de cubo / MSD radix, y por supuesto, requiere el árbol de expansión $O(m)$.

Detectando un ciclo en un gráfico dirigido usando Depth First Traversal

Existe un ciclo en un gráfico dirigido si se descubre un borde posterior durante un DFS. Un borde posterior es un borde de un nodo a sí mismo o uno de los ancestros en un árbol DFS. Para un gráfico desconectado, obtenemos un bosque DFS, por lo que debe recorrer todos los vértices en

el gráfico para encontrar árboles DFS separados.

Implementación de C ++:

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

using namespace std;

#define NUM_V 4

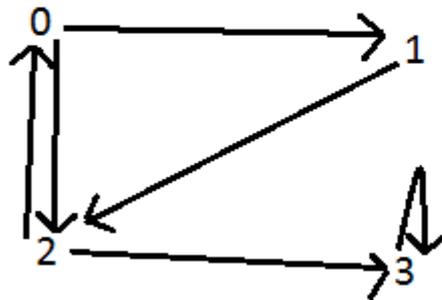
bool helper(list<int> *graph, int u, bool* visited, bool* recStack)
{
    visited[u]=true;
    recStack[u]=true;
    list<int>::iterator i;
    for(i = graph[u].begin();i!=graph[u].end();++i)
    {
        if(recStack[*i]) //if vertice v is found in recursion stack of this DFS traversal
            return 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;
}
```

Resultado: Como se muestra a continuación, hay tres bordes posteriores en el gráfico. Uno entre vértice 0 y 2; entre el vértice 0, 1 y 2; y vértice 3. La complejidad del tiempo de búsqueda es $O(V + E)$ donde V es el número de vértices y E es el número de bordes.



Introducción a la teoría de grafos

La [Teoría de los Gráficos](#) es el estudio de los gráficos, que son estructuras matemáticas utilizadas para modelar relaciones de pares entre objetos.

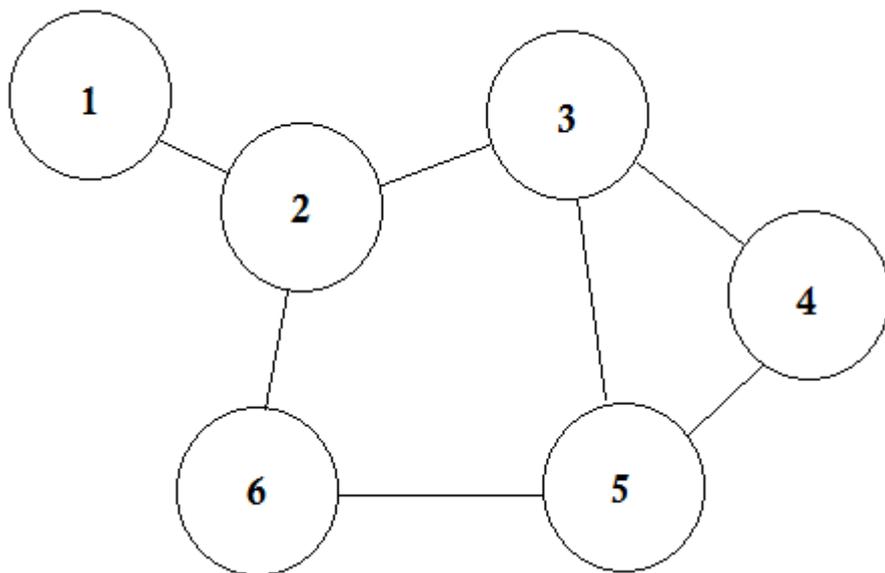
¿Sabías que casi todos los problemas del planeta Tierra se pueden convertir en problemas de Caminos y Ciudades, y se pueden resolver? La teoría de los gráficos se inventó hace muchos años, incluso antes de la invención de la computadora. [Leonhard Euler](#) escribió un artículo sobre los [Siete Puentes de Königsberg](#), que se considera el primer artículo de Graph Theory. Desde entonces, las personas se han dado cuenta de que si podemos convertir cualquier problema en este problema de la ciudad-carretera, podemos resolverlo fácilmente mediante la Teoría de grafos.

La teoría de los gráficos tiene muchas aplicaciones. Una de las aplicaciones más comunes es encontrar la distancia más corta entre una ciudad y otra. Todos sabemos que para llegar a su PC, esta página web tenía que viajar muchos enrutadores desde el servidor. La Teoría de Gráficos le ayuda a descubrir los enrutadores que necesitaban ser cruzados. Durante la guerra, qué calle necesita ser bombardeada para desconectar la ciudad capital de otras, eso también se puede descubrir usando la Teoría de Gráficos.

Primero aprendamos algunas definiciones básicas sobre la teoría de grafos.

Grafico:

Digamos que tenemos 6 ciudades. Los marcamos como 1, 2, 3, 4, 5, 6. Ahora conectamos las ciudades que tienen caminos entre sí.



Este es un gráfico simple donde se muestran algunas ciudades con las carreteras que las conectan. En Graph Theory, llamamos a cada una de estas ciudades **Node** o **Vertex** y las carreteras se llaman **Edge**. El gráfico es simplemente una conexión de estos nodos y bordes.

Un **nodo** puede representar muchas cosas. En algunos gráficos, los nodos representan ciudades, algunos representan aeropuertos, algunos representan un cuadrado en un tablero de ajedrez. **Edge** representa la relación entre cada uno de los nodos. Esa relación puede ser el momento de ir de un aeropuerto a otro, los movimientos de un caballero de una casilla a todas las demás plazas, etc.

Camino del caballero en un tablero de ajedrez

En palabras simples, un **nodo** representa cualquier objeto y **Edge** representa la relación entre dos objetos.

Nodo adyacente:

Si un nodo **A** comparte un borde con el nodo **B**, entonces se considera que **B** es adyacente a **A**. En otras palabras, si dos nodos están conectados directamente, se llaman nodos adyacentes. Un nodo puede tener múltiples nodos adyacentes.

Gráfico dirigido y no dirigido:

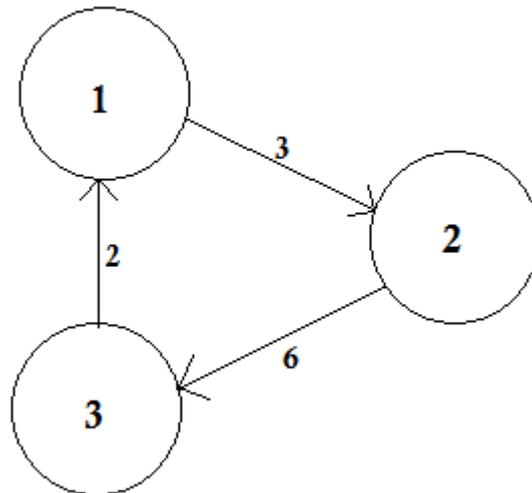
En los gráficos dirigidos, los bordes tienen signos de dirección en un lado, lo que significa que los bordes son *unidireccionales*. Por otro lado, los bordes de los gráficos no dirigidos tienen señales de dirección en ambos lados, lo que significa que son *bidireccionales*. Por lo general, los gráficos no dirigidos se representan sin signos en ninguno de los lados de los bordes.

Supongamos que hay una fiesta en marcha. Las personas en el grupo están representadas por nodos y hay una ventaja entre dos personas si se dan la mano. Entonces, esta gráfica no está dirigida porque cualquier persona **A** le da la mano a la persona **B** si y solo si **B** también le da la mano a **A**. Por el contrario, si los bordes de una persona de **A** a otra persona **B** corresponde a **A**'s admirando **B**, entonces este gráfico se dirige, debido a la admiración no es necesariamente un movimiento alternativo. El primer tipo de gráfico se denomina *gráfico no dirigido* y los bordes se

denominan *bordes no dirigidos*, mientras que el último tipo de gráfico se denomina *gráfico dirigido* y los bordes se denominan *bordes directos*

Gráfico ponderado y no ponderado:

Un gráfico ponderado es un gráfico en el que se asigna un número (el peso) a cada borde. Dichos pesos podrían representar, por ejemplo, costos, longitudes o capacidades, dependiendo del

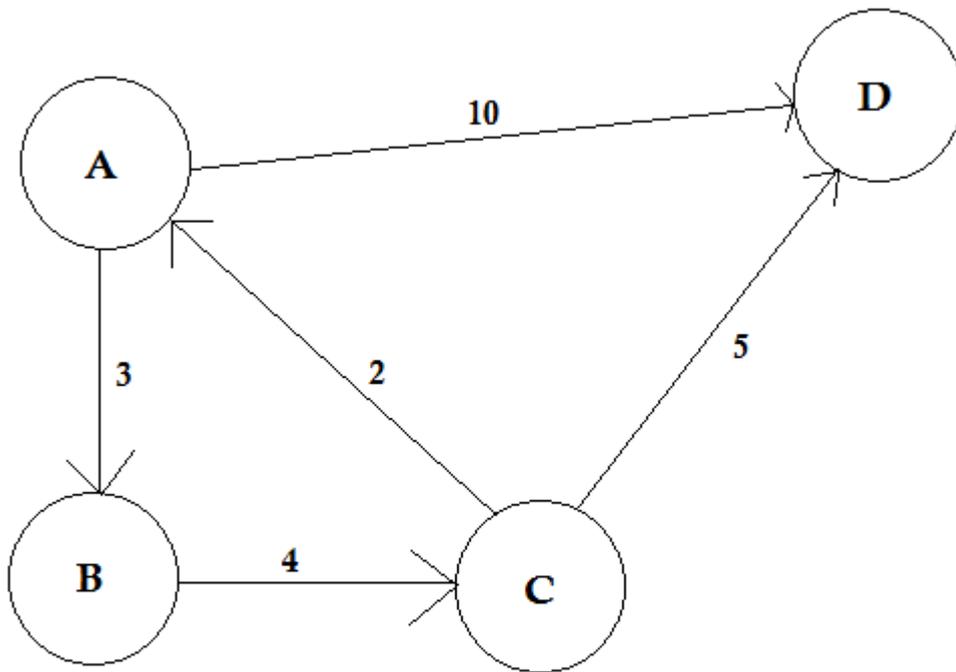


problema en cuestión.

Un gráfico no ponderado es simplemente lo contrario. Asumimos que, el peso de todos los bordes es el mismo (presumiblemente 1).

Camino:

Una ruta representa una forma de ir de un nodo a otro. Se compone de secuencia de aristas. Puede haber múltiples rutas entre dos nodos.



En el ejemplo anterior, hay dos caminos de **A** a **D**. **A**-> **B**, **B**-> **C**, **C**-> **D** es una ruta. El costo de este camino es $3 + 4 + 2 = 9$. Una vez más, hay otro camino **A**-> **D**. El costo de este camino es **10**. La ruta que cuesta lo más bajo se llama *ruta más corta*.

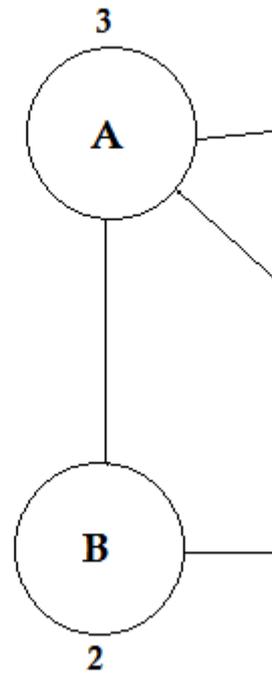
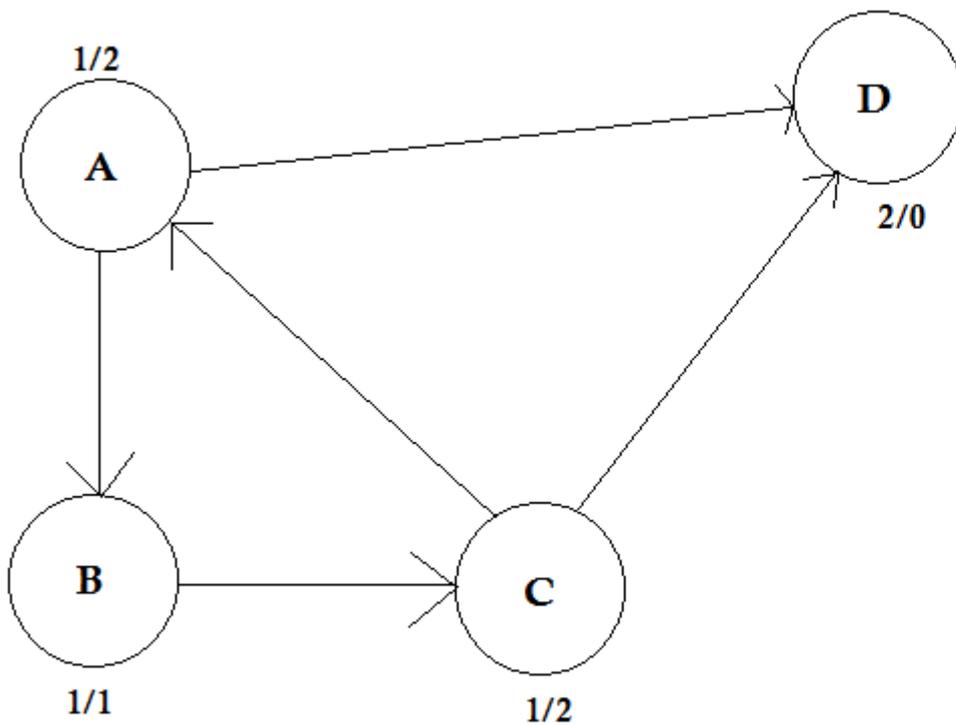
La licenciatura:

El grado de un vértice es el número de bordes que están conectados a él. Si hay algún borde que se conecta al vértice en ambos extremos (un bucle) se cuenta dos veces.

En los gráficos dirigidos, los nodos tienen dos tipos de grados:

- En grado: el número de bordes que apuntan al nodo.
- Out-degree: el número de bordes que apuntan desde el nodo a otros nodos.

Para las gráficas no dirigidas, simplemente se llaman grados.



Algunos algoritmos relacionados con la teoría de grafos

- Algoritmo de Bellman-Ford
- Algoritmo de Dijkstra
- Algoritmo Ford – Fulkerson
- Algoritmo de Kruskal
- Algoritmo vecino más cercano
- Algoritmo de prim
- Búsqueda en profundidad primero
- Búsqueda de amplitud

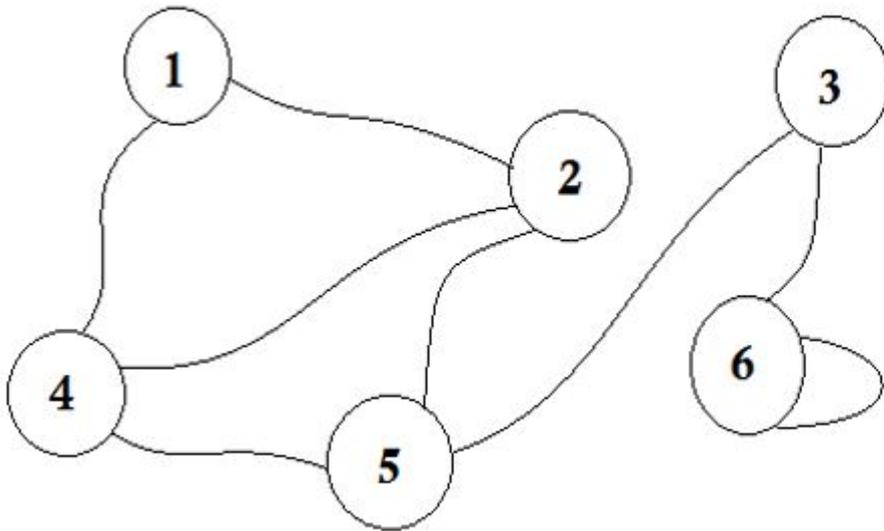
Almacenando Gráficos (Matriz de Adyacencia)

Para almacenar una gráfica, dos métodos son comunes:

- Matriz de adyacencia
- Lista de adyacencia

Una **matriz de adyacencia** es una matriz cuadrada que se utiliza para representar un gráfico finito. Los elementos de la matriz indican si los pares de vértices son adyacentes o no en el gráfico.

Adyacente significa 'al lado o contiguo a otra cosa' o estar al lado de algo. Por ejemplo, tus vecinos están adyacentes a ti. En la teoría de gráficos, si podemos ir al **nodo B** desde el **nodo A**, podemos decir que el **nodo B** es adyacente al **nodo A**. Ahora aprenderemos cómo almacenar qué nodos son adyacentes a cada uno a través de la Matriz de Adyacencia. Esto significa que vamos a representar qué nodos comparten el borde entre ellos. Aquí matriz significa matriz 2D.

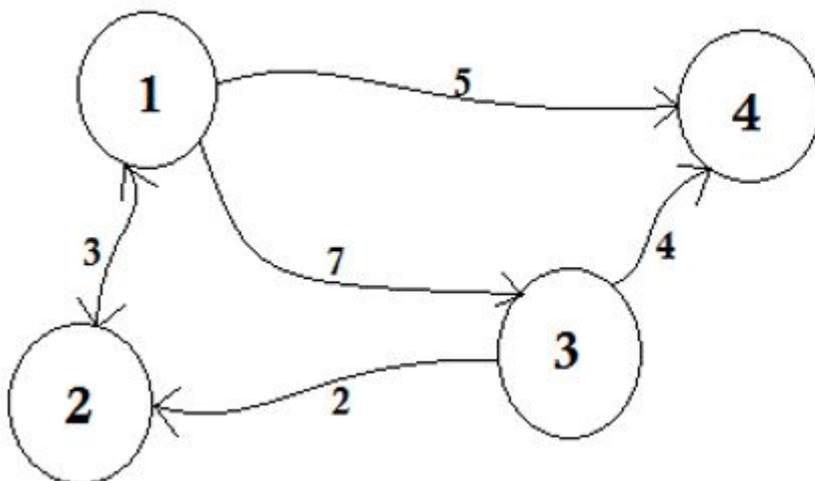


Node	1	2	3
1	0	1	0
2	1	0	0
3	0	0	0
4	1	1	0
5	0	1	1
6	0	0	1

Aquí puede ver una tabla al lado del gráfico, esta es nuestra matriz de adyacencia. Aquí la **matriz** $[i][j] = 1$ representa que hay un borde entre i y j . Si no hay ventaja, simplemente ponemos **Matrix** $[i][j] = 0$.

Estos bordes pueden ser pesados, al igual que puede representar la distancia entre dos ciudades. Luego pondremos el valor en **Matrix** $[i][j]$ en lugar de poner 1.

El gráfico descrito anteriormente es *bidireccional* o no *dirigido*, lo que significa que si podemos ir al **nodo 1** desde el **nodo 2**, también podemos ir al **nodo 2** desde el **nodo 1**. Si la gráfica fuera *dirigida*, entonces habría un signo de flecha en un lado de la gráfica. Incluso entonces, podríamos representarlo usando una matriz de adyacencia.



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

Representamos los nodos que no comparten borde por *infinito*. Una cosa a tener en cuenta es que, si la gráfica no está dirigida, la matriz se vuelve *simétrica*.

El pseudocódigo para crear la matriz:

```
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

```

También podemos rellenar la matriz utilizando esta forma común:

```

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

```

Para gráficos dirigidos, podemos eliminar **Matrix [n2] [n1] = línea de costo** .

Los inconvenientes de usar la matriz de adyacencia:

La memoria es un gran problema. No importa cuántos bordes haya, siempre necesitaremos una matriz de tamaño $N * N$ donde N es el número de nodos. Si hay 10000 nodos, el tamaño de la matriz será $4 * 10000 * 10000$ alrededor de 381 megabytes. Esto es un gran desperdicio de memoria si consideramos gráficos que tienen algunos bordes.

Supongamos que queremos averiguar a qué nodo podemos ir desde un nodo u . Tendremos que revisar toda la fila de u , lo que cuesta mucho tiempo.

El único beneficio es que podemos encontrar fácilmente la conexión entre los nodos UV y su costo utilizando la Matriz de Adyacencia.

Código Java implementado utilizando el pseudo-código anterior:

```

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("Something went wrong");
    }
}

```

```
        sc.close();
    }
}
```

Ejecutando el código: Guarde el archivo y compile usando `javac Represent_Graph_Adjacency_Matrix.java`

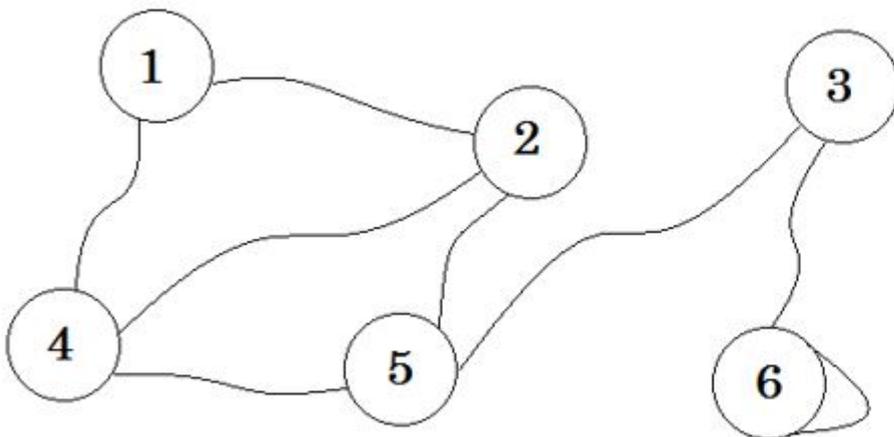
Ejemplo:

```
$ 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
```

Almacenamiento de gráficos (lista de adyacencia)

La **lista de adyacencia** es una colección de listas desordenadas que se utilizan para representar un gráfico finito. Cada lista describe el conjunto de vecinos de un vértice en un gráfico. Se necesita menos memoria para almacenar gráficos.

Veamos un gráfico, y su matriz de adyacencia:



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

Ahora creamos una lista usando estos valores.



Esto se llama lista de adyacencia. Muestra qué nodos están conectados a qué nodos. Podemos almacenar esta información utilizando una matriz 2D. Pero nos costará la misma memoria que la Matriz de Adyacencia. En su lugar, vamos a utilizar memoria asignada dinámicamente para almacenar esta.

Muchos idiomas admiten **Vector** o **Lista** que podemos usar para almacenar la lista de adyacencia. Para estos, no necesitamos especificar el tamaño de la **Lista** . Solo necesitamos especificar el número máximo de nodos.

El pseudocódigo será:

```

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
  
```

Dado que este es un gráfico no dirigido, hay un borde de **x** a **y** , también hay un borde de **y** a **x** . Si fuera un gráfico dirigido, omitiríamos el segundo. Para los gráficos ponderados, necesitamos almacenar el costo también. Crearemos otro **vector** o **lista** llamado **costo []** para almacenar estos. El pseudocódigo:

```

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
```

A partir de este, podemos averiguar fácilmente el número total de nodos conectados a cualquier nodo y qué son estos nodos. Lleva menos tiempo que la Matriz de Adyacencia. Pero si tuviéramos que averiguar si hay una ventaja entre u y v , habría sido más fácil si hubiéramos mantenido una matriz de adyacencia.

Lea Grafico en línea: <https://riptutorial.com/es/algorithm/topic/2299/grafico>

Capítulo 39: Gráficos de travesías

Examples

Profundidad de la primera búsqueda de la función transversal

La función toma el argumento del índice de nodo actual, la lista de adyacencia (almacenada en el vector de vectores en este ejemplo) y el vector de booleano para realizar un seguimiento de qué nodo ha sido visitado.

```
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);
}
```

Lea Gráficos de travesías en línea: <https://riptutorial.com/es/algorithm/topic/9493/graficos-de-travesias>

Capítulo 40: Heap Sort

Examples

Heap Sort Información Básica

La ordenación de pila es una técnica de clasificación basada en comparación en la estructura de datos de pila binaria. Es similar al tipo de selección en el que primero encontramos el elemento máximo y lo ponemos al final de la estructura de datos. Luego repita el mismo proceso para los elementos restantes.

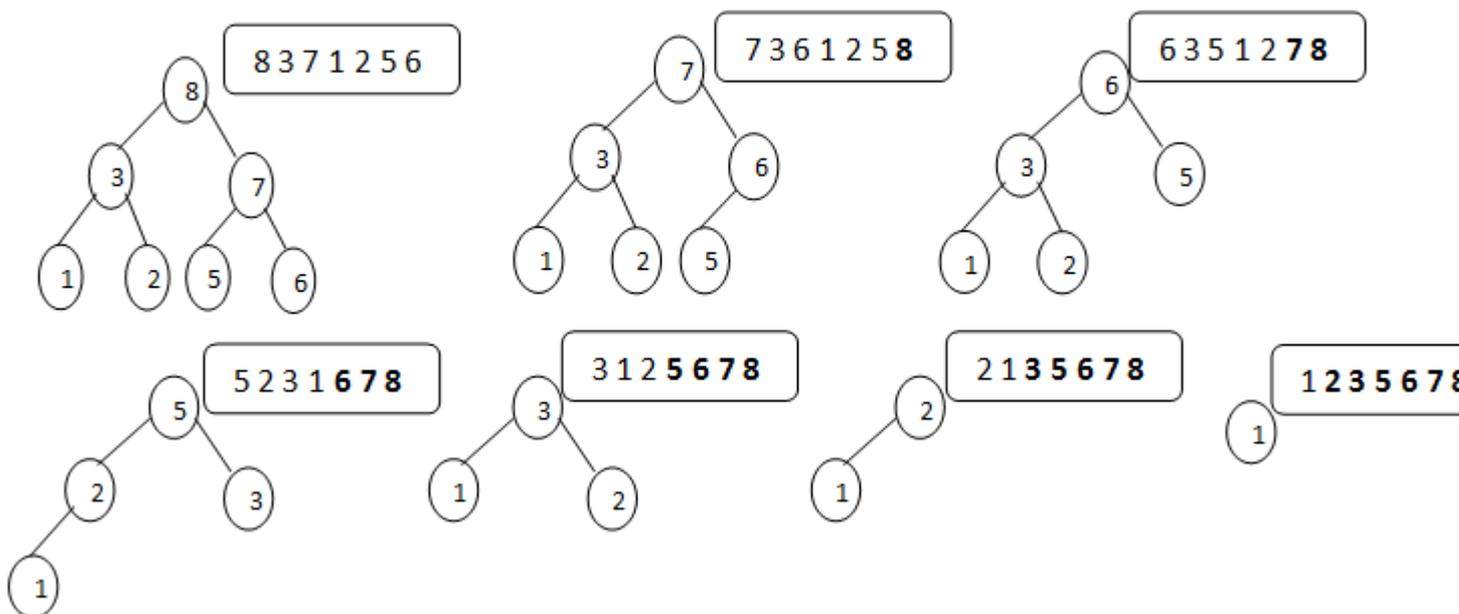
Pseudo código para Heap Sort:

```
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)
  end

function heapify(a, count)
  start <- parent(count - 1)
  while start >= 0 do
    restore(a, start, count - 1)
    start <- start - 1
  end
```

Ejemplo de clasificación del montón:

Example:- The fig. shows steps of heap-sort for list (2 3 7 1 8 5 6)



Espacio auxiliar: $O(1)$

Complejidad del tiempo: $O(n \log n)$

Implementación de 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;
    }
}
```

Lea Heap Sort en línea: <https://riptutorial.com/es/algorithm/topic/7281/heap-sort>

Capítulo 41: La subsecuencia cada vez mayor

Examples

La información básica de la subsecuencia cada vez más creciente

El problema de la [Subsecuencia Cada vez mayor más larga](#) es encontrar la subsecuencia de la secuencia de entrada de ingreso en la que los elementos de la subsecuencia se ordenan de menor a mayor orden. Todas las subsecuencias no son contiguas o únicas.

Aplicación de la subsecuencia creciente más larga:

Algoritmos como la Subsecuenciación Más Larga, la Subsecuencia Común Más Larga se utilizan en los sistemas de control de versiones como Git y etc.

Forma simple de algoritmo:

1. Encuentra líneas únicas que son comunes a ambos documentos.
2. Tome todas las líneas del primer documento y ordénelas de acuerdo con su aspecto en el segundo documento.
3. Calcule el LIS de la secuencia resultante (haciendo una [Clasificación de Paciencia](#)), obteniendo la secuencia de líneas más larga, una correspondencia entre las líneas de dos documentos.
4. Recurse el algoritmo en cada rango de líneas entre los ya emparejados.

Ahora consideremos un ejemplo más simple del problema LCS. Aquí, la entrada es solo una secuencia de enteros distintos a_1, a_2, \dots, a_n , y queremos encontrar la subsecuencia creciente más larga en ella. Por ejemplo, si la entrada es **7,3,8,4,2,6**, la subsecuencia de mayor **crecimiento** es **3,4,6**.

El enfoque más sencillo es ordenar los elementos de entrada en orden creciente y aplicar el algoritmo LCS a las secuencias originales y ordenadas. Sin embargo, si observa la matriz resultante, notará que muchos valores son iguales, y la matriz parece muy repetitiva. Esto sugiere que el problema de LIS (la subsecuencia en aumento más larga) se puede hacer con el algoritmo de programación dinámica utilizando solo una matriz unidimensional.

Pseudo Código:

1. Describe una matriz de valores que queremos calcular.
Para $1 \leq i \leq n$, sea **A (i)** la longitud de una secuencia de entrada cada vez más larga.
Tenga en cuenta que la longitud que nos interesa en última instancia es la $\max\{A(i) \mid 1 \leq i \leq n\}$.
2. Dar una recurrencia.
Para $1 \leq i \leq n$, $A(i) = 1 + \max\{A(j) \mid 1 \leq j < i \text{ y } \text{input}(j) < \text{input}(i)\}$.
3. Calcula los valores de A.
4. Encuentra la solución óptima.

El siguiente programa usa A para calcular una solución óptima. La primera parte calcula un valor m tal que **A (m)** es la longitud de una subsecuencia de entrada en aumento óptima. La segunda parte calcula una subsecuencia creciente óptima, pero por conveniencia lo imprimimos en orden inverso. Este programa se ejecuta en el tiempo $O(n)$, por lo que todo el algoritmo se ejecuta en el tiempo $O(n^2)$.

Parte 1:

```
m ← 1
for i : 2..n
  if A(i) > A(m) then
    m ← i
  end if
end for
```

Parte 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
```

Solución recursiva:

Enfoque 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
```

La complejidad del tiempo en el Enfoque 1: $O(n \cdot 2^n)$

Enfoque 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], ∞)
```

Complejidad de tiempo en el enfoque 2: $O(n^2)$

Enfoque 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])
```

Complejidad de tiempo en el enfoque 3: $O(n^2)$

Algoritmo iterativo:

Calcula los valores de forma iterativa de manera ascendente.

```
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
```

La complejidad del tiempo en el enfoque iterativo: $O(n^2)$

Espacio auxiliar: $O(n)$

Permite tomar **{0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15}** como entrada. Por lo tanto, la Subsecuencia creciente más larga para la entrada dada es **{0, 2, 6, 9, 11, 15}** .

Implementación de 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);
}
}
```

Lea La subsecuencia cada vez mayor en línea: <https://riptutorial.com/es/algorithm/topic/7537/la-subsecuencia-cada-vez-mayor>

Capítulo 42: La subsecuencia común más larga

Examples

Explicación de la subsecuencia común más larga

Una de las implementaciones más importantes de la Programación Dinámica es encontrar la [Subsecuencia Común Más Larga](#) . Vamos a definir algunas de las terminologías básicas primero.

Subsecuencia

Una subsecuencia es una secuencia que puede derivarse de otra secuencia al eliminar algunos elementos sin cambiar el orden de los elementos restantes. Digamos que tenemos una cadena **ABC** . Si borramos cero o uno o más de un carácter de esta cadena obtenemos la subsecuencia de esta cadena. Así que las subsecuencias de la cadena **ABC** serán { "A" , "B" , "C" , "AB" , "AC" , "BC" , "ABC" , "" }. Incluso si eliminamos todos los caracteres, la cadena vacía también será una subsecuencia. Para averiguar la subsecuencia, para cada carácter de una cadena, tenemos dos opciones: o tomamos el carácter o no. Entonces, si la longitud de la cadena es n , hay 2^n subsecuencias de esa cadena.

Subsecuencia más larga:

Como el nombre sugiere, de todas las subsecuencias comunes entre dos cadenas, la subsecuencia común más larga (LCS) es la que tiene la longitud máxima. Por ejemplo: las subsecuencias comunes entre "HELLOM" y "HMLD" son "H" , "HL" , "HM", etc. Aquí "HLL" es la subsecuencia común más larga que tiene la longitud 3.

Método de fuerza bruta:

Podemos generar todas las subsecuencias de dos cadenas usando *backtracking* . Luego podemos compararlas para descubrir las subsecuencias comunes. Después tendremos que averiguar el que tiene la longitud máxima. Ya hemos visto que hay 2^n subsecuencias de una cadena de longitud n . Tomaría años resolver el problema si nuestra n cruza **20-25** .

Método de programación dinámica:

Abordemos nuestro método con un ejemplo. Supongamos que tenemos dos cadenas **abcdaf** y **acbcf** . Vamos a denotar estos con **s1** y **s2** . Por lo tanto, la subsecuencia común más larga de estas dos cadenas será "abcf" , que tiene una longitud 4. De nuevo, les recuerdo que las subsecuencias no tienen que ser continuas en la cadena. Para construir "abcf" , ignoramos "da" en **s1** y "c" en **s2** . ¿Cómo podemos descubrir esto utilizando la programación dinámica?

Comenzaremos con una tabla (una matriz 2D) que tiene todos los caracteres de **s1** en una fila y todos los caracteres de **s2** en la columna. Aquí la tabla tiene un índice de 0 y colocamos los caracteres de 1 en adelante. Recorreremos la tabla de izquierda a derecha para cada fila. Nuestra

mesa se verá como:

	0	1	2	3	4	5	6
0		a	b	c	d	a	f
1	a						
2	c						
3	b						
4	c						
5	f						

Aquí, cada fila y columna representan la longitud de la subsecuencia común más larga entre dos cadenas si tomamos los caracteres de esa fila y columna y agregamos el prefijo anterior. Por ejemplo: la **Tabla [2] [3]** representa la longitud de la subsecuencia común más larga entre "ac" y "abc" .

La columna 0 representa la subsecuencia vacía de **s1** . De manera similar, la fila 0 representa la subsecuencia vacía de **s2** . Si tomamos una subsecuencia vacía de una cadena y tratamos de hacerla coincidir con otra, no importa cuán larga sea la longitud de la segunda subcadena, la subsecuencia común tendrá una longitud de 0. Entonces podemos llenar las filas 0-th y las columnas 0-th con 0's. Obtenemos:

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	a	0					
2	c	0					
3	b	0					
4	c	0					
5	f	0					

Vamos a empezar. Cuando llenemos la **Tabla [1] [1]** , nos preguntamos, si tuviéramos una cadena **a** y otra cadena **a** y nada más, ¿cuál será la subsecuencia común más larga aquí? La longitud del LCS aquí será 1. Ahora veamos la **Tabla [1] [2]** . Tenemos la cadena **ab** y la cadena **a** . La longitud del LCS será 1. Como puede ver, el resto de los valores también serán 1 para la primera fila, ya que considera solo la cadena **a** con **abcd** , **abcda** , **abcdaf** . Así nuestra mesa se verá como:

	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
2	c	0					
3	b	0					
4	c	0					
5	f	0					

Para la fila 2, que ahora incluirá **c** . Para la **Tabla [2] [1]** tenemos **ac** en un lado y **a** en el otro lado. Entonces, la longitud del LCS es 1. ¿De dónde obtuvimos este 1? Desde la parte superior, que denota el LCS **a** entre dos subcadenas. Entonces, lo que estamos diciendo es que si **s1 [2]** y **s2 [1]** no son lo mismo, entonces la longitud del LCS será la máxima de la longitud del LCS en la **parte superior** o en la **izquierda** . Tomar la longitud del LCS en la parte superior indica que no tomamos el carácter actual de **s2** . De manera similar, tomar la longitud del LCS a la izquierda denota eso, no tomamos el carácter actual de **s1** para crear el LCS. Obtenemos:

	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
2	c	0	1				
3	b	0					
4	c	0					
5	f	0					

Así que nuestra primera fórmula será:

```
if s2[i] is not equal to s1[j]
    Table[i][j] = max(Table[i-1][j], Table[i][j-1])
endif
```

Continuando, para la **Tabla [2] [2]** tenemos la cadena **ab** y **ac** . Como **c** y **b** no son iguales, ponemos el máximo de la parte superior o izquierda aquí. En este caso, es nuevamente 1. Después de eso, para la **Tabla [2] [3]** tenemos la cadena **abc** y **ac** . Esta vez los valores actuales de la fila y la columna son los mismos. Ahora la longitud de la LCS será igual a la longitud máxima de la LCS hasta ahora + 1. ¿Cómo obtenemos la longitud máxima de la LCS hasta ahora? Verificamos el valor diagonal, que representa la mejor coincidencia entre **ab** y **a** . Desde este

estado, para los valores actuales, agregamos un carácter más a **s1** y **s2** que resultó ser el mismo. Por lo tanto, la duración de la LCS por supuesto aumentará. Pondremos $1 + 1 = 2$ en la **Tabla [2] [3]** . Obtenemos,

	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
2	c	0	1	1	2		
3	b	0					
4	c	0					
5	f	0					

Así que nuestra segunda fórmula será:

```
if s2[i] equals to s1[j]
    Table[i][j] = Table[i-1][j-1] + 1
endif
```

Hemos definido ambos casos. Usando estas dos fórmulas, podemos rellenar toda la tabla. Después de llenar la mesa, se verá así:

	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
2	c	0	1	1	2	2	2
3	b	0	1	2	2	2	2
4	c	0	1	2	3	3	3
5	f	0	1	2	3	3	4

La longitud del LCS entre **s1** y **s2** será la **Tabla [5] [6] = 4** . Aquí, 5 y 6 son la longitud de **s2** y **s1** respectivamente. Nuestro pseudo-código será:

```
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]

```

La complejidad del tiempo para este algoritmo es: **O (mn)** donde **m** y **n** denota la longitud de cada cadena.

¿Cómo encontramos la subsecuencia común más larga? Comenzaremos desde la esquina inferior derecha. Comprobaremos de donde viene el valor. Si el valor proviene de la diagonal, es decir, si la **Tabla [i-1][j-1]** es igual a la **Tabla [i][j] - 1**, presionamos **s2 [i]** o **s1 [j]** (ambos son lo mismo) y se mueven en diagonal. Si el valor viene de arriba, eso significa que si la **Tabla [i-1][j]** es igual a la **Tabla [i][j]**, nos movemos hacia la parte superior. Si el valor viene de la izquierda, eso significa que si la **Tabla [i][j-1]** es igual a la **Tabla [i][j]**, nos movemos hacia la izquierda. Cuando llegamos a la columna de la izquierda o de la parte superior, nuestra búsqueda termina. Luego sacamos los valores de la pila y los imprimimos. El pseudocódigo:

```

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

```

Puntos a destacar: si tanto la **Tabla [i-1][j]** como la **Tabla [i][j-1]** son iguales a la **Tabla [i][j]** y la **Tabla [i-1][j-1]** no es igual a la **Tabla [i][j] - 1**, puede haber dos LCS para ese momento. Este pseudocódigo no considera esta situación. Tendrás que resolver esto recursivamente para encontrar múltiples LCS.

La complejidad del tiempo para este algoritmo es: **O (máx (m, n))**.

Lea La subsecuencia común más larga en línea: <https://riptutorial.com/es/algorithm/topic/7517/la->

Capítulo 43: Notación Big-O

Observaciones

Definición

La notación Big-O es en su corazón una notación matemática, utilizada para comparar la tasa de convergencia de funciones. Sean $n \rightarrow f(n)$ y $n \rightarrow g(n)$ funciones definidas sobre los números naturales. Luego decimos que $f = O(g)$ si y solo si $f(n)/g(n)$ está limitada cuando n se acerca al infinito. En otras palabras, $f = O(g)$ si y solo si existe una constante A , tal que para todos n , $f(n)/g(n) \leq A$.

En realidad, el alcance de la notación Big-O es un poco más amplio en matemáticas, pero por simplicidad lo he reducido a lo que se usa en el análisis de complejidad de algoritmos: funciones definidas en los naturales, que tienen valores distintos de cero, y el caso de n creciente hasta el infinito.

Qué significa eso ?

Tomemos el caso de $f(n) = 100n^2 + 10n + 1$ y $g(n) = n^2$. Es bastante claro que ambas funciones tienden a infinito ya que n tiende a infinito. Pero a veces, saber el límite no es suficiente, y también queremos saber la *velocidad* a la que las funciones se acercan a su límite. Nociones como Big-O ayudan a comparar y clasificar funciones por su velocidad de convergencia.

Averigüemos si $f = O(g)$ aplicando la definición. Tenemos $f(n)/g(n) = 100 + 10/n + 1/n^2$. Desde $10/n$ es 10 cuando n es 1 y está disminuyendo, y desde $1/n^2$ es 1 cuando n es 1 y también está disminuyendo, hemos $f(n)/g(n) \leq 100 + 10 + 1 = 111$. La definición se cumple porque hemos encontrado un límite de $f(n)/g(n)$ (111) y por tanto $f = O(g)$ (decimos que f es un Big-O de n^2).

Esto significa que f tiende a infinito aproximadamente a la misma velocidad que g . Ahora bien, esto puede parecer algo extraño de decir, porque lo que hemos encontrado es que f es a lo sumo 111 veces más grande que g , o en otras palabras, cuando g crece en 1, f crece como máximo en 111. Puede parecer que en crecimiento 111 veces más rápido no es "aproximadamente la misma velocidad". Y, de hecho, la notación Big-O no es una forma muy precisa de clasificar la velocidad de convergencia de la función, por lo que en matemáticas usamos la [relación de equivalencia](#) cuando queremos una estimación precisa de la velocidad. Pero a los efectos de separar algoritmos en clases de gran velocidad, Big-O es suficiente. No necesitamos separar las funciones que crecen un número fijo de veces más rápido que las otras, sino solo las funciones que crecen *infinitamente* más rápido que las otras. Por ejemplo, si tomamos $h(n) = n^2 \log(n)$, vemos que $h(n)/g(n) = \log(n)$ que tiende a infinito con n , por lo que h *no* es $O(n^2)$, porque h crece *infinitamente* más rápido que n^2 .

Ahora necesito hacer una nota al margen: es posible que haya notado que si $f = O(g)$ y $g = O(h)$, entonces $f = O(h)$. Por ejemplo, en nuestro caso, tenemos $f = O(n^3)$, y $f = O(n^4)$... En el análisis de complejidad de algoritmos, frecuentemente decimos que $f = O(g)$ significa que $f = O(g)$ y $g = O(f)$, que puede entenderse como "g es el Big-O más pequeño para f". En matemáticas decimos que tales funciones son Big-Theta entre sí.

Cómo se usa ?

Al comparar el rendimiento del algoritmo, nos interesa la cantidad de operaciones que realiza un algoritmo. Esto se llama *complejidad del tiempo* . En este modelo, consideramos que cada operación básica (suma, multiplicación, comparación, asignación, etc.) toma una cantidad de tiempo fija, y contamos el número de dichas operaciones. Por lo general, podemos expresar este número en función del tamaño de la entrada, que llamamos n . Y lamentablemente, este número generalmente crece hasta el infinito con n (si no lo hace, decimos que el algoritmo es $O(1)$). Separamos nuestros algoritmos en clases de gran velocidad definidas por Big-O: cuando hablamos de un "algoritmo $O(n^2)$ ", queremos decir que la cantidad de operaciones que realiza, expresada como una función de n , es una $O(n^2)$. Esto dice que nuestro algoritmo es aproximadamente tan rápido como un algoritmo que haría una cantidad de operaciones igual al cuadrado del tamaño de su entrada, *o más rápido* . La parte "o más rápida" está ahí porque usé Big-O en lugar de Big-Theta, pero generalmente la gente dirá que Big-O quiere decir Big-Theta.

Cuando contamos las operaciones, generalmente consideramos el peor de los casos: por ejemplo, si tenemos un bucle que puede ejecutarse como máximo n veces y que contiene 5 operaciones, el número de operaciones que contamos es $5n$. También es posible considerar la complejidad del caso promedio.

Nota rápida: un algoritmo rápido es uno que realiza pocas operaciones, por lo que si el número de operaciones crece infinitamente *más rápido* , entonces el algoritmo es *más lento* : $O(n)$ es mejor que $O(n^2)$.

También a veces estamos interesados en la *complejidad espacial* de nuestro algoritmo. Para esto, consideramos el número de bytes en la memoria ocupada por el algoritmo como una función del tamaño de la entrada, y usamos Big-O de la misma manera.

Examples

Un bucle simple

La siguiente función encuentra el elemento máximo en una matriz:

```
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;
}
```

El tamaño de entrada es el tamaño de la matriz, a la que llamé `len` en el código.

Contemos las operaciones.

```
int max = INT_MIN;
```

```
size_t i = 0;
```

Estas dos tareas se realizan solo una vez, por lo que son 2 operaciones. Las operaciones que se realizan en bucle son:

```
if (max < array[i])  
i++;  
max = array[i]
```

Como hay 3 operaciones en el bucle, y el bucle se realiza n veces, agregamos $3n$ a nuestras 2 operaciones ya existentes para obtener $3n + 2$. Así que nuestra función toma $3n + 2$ operaciones para encontrar el máximo (su complejidad es $3n + 2$). Este es un polinomio donde el término de más rápido crecimiento es un factor de n , por lo que es $O(n)$.

Probablemente habrás notado que la "operación" no está muy bien definida. Por ejemplo, dije que `if (max < array[i])` era una operación, pero dependiendo de la arquitectura, esta declaración puede compilar, por ejemplo, tres instrucciones: una lectura de memoria, una comparación y una rama. También he considerado todas las operaciones como iguales, aunque, por ejemplo, las operaciones de memoria serán más lentas que las demás y su rendimiento variará enormemente debido, por ejemplo, a los efectos de caché. También he ignorado por completo la declaración de devolución, el hecho de que se creará un marco para la función, etc. Al final, no importa el análisis de complejidad, porque de cualquier forma que elija para contar las operaciones, solo cambiará el coeficiente del factor n y la constante, por lo que el resultado seguirá siendo $O(n)$. La complejidad muestra cómo el algoritmo se escala con el tamaño de la entrada, pero no es el único aspecto del rendimiento!

Un bucle anidado

La siguiente función comprueba si una matriz tiene algún duplicado tomando cada elemento, luego iterando sobre toda la matriz para ver si el elemento está ahí

```
_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;  
}
```

El bucle interno realiza en cada iteración una serie de operaciones que son constantes con n . El bucle externo también realiza algunas operaciones constantes y ejecuta el bucle interno n veces. El propio bucle externo se ejecuta n veces. Así que las operaciones dentro del bucle interno se ejecutan n^2 veces, las operaciones en el bucle externo se ejecutan n veces, y la asignación a i se realiza una vez. Por lo tanto, la complejidad será algo así como $an^2 + bn + c$, y como el término más alto es n^2 , la notación $O(n^2)$ es $O(n^2)$.

Como habrá notado, podemos mejorar el algoritmo evitando hacer las mismas comparaciones varias veces. Podemos comenzar desde $i + 1$ en el bucle interno, porque todos los elementos antes de que ya se hayan verificado con todos los elementos de la matriz, incluido el del índice $i + 1$. Esto nos permite soltar el cheque $i == j$.

```
_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;
}
```

Obviamente, esta segunda versión hace menos operaciones y por lo tanto es más eficiente. ¿Cómo se traduce eso a la notación Big-O? Bueno, ahora el cuerpo del bucle interno se ejecuta $1 + 2 + \dots + n - 1 = n(n-1)/2$ veces. Este es *todavía* un polinomio de segundo grado, y por lo tanto es solo $O(n^2)$. Claramente, hemos reducido la complejidad, ya que dividimos aproximadamente por 2 el número de operaciones que estamos realizando, pero aún estamos en la misma *clase de complejidad* definida por Big-O. Para reducir la complejidad a una clase más baja, tendríamos que dividir el número de operaciones por algo que *tiende a infinito* con n .

Un ejemplo de $O(\log n)$

Introducción

Considere el siguiente problema:

L es una lista ordenada que contiene n enteros con signo (siendo n suficientemente grande), por ejemplo $[-5, -2, -1, 0, 1, 2, 4]$ (aquí, n tiene un valor de 7). Si se sabe que L contiene el entero 0, ¿cómo puedes encontrar el índice de 0?

Enfoque ingenuo

Lo primero que viene a la mente es leer cada índice hasta que se encuentre 0. En el peor de los casos, el número de operaciones es n , por lo que la complejidad es $O(n)$.

Esto funciona bien para valores pequeños de n , pero ¿hay una forma más eficiente?

Dicotomía

Considere el siguiente algoritmo (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 y b son los índices entre los que 0 es que se encuentran. Cada vez que entramos en el bucle, utilizamos un índice entre a y b y lo utilizan para reducir el área de búsqueda.

En el peor de los casos, tenemos que esperar hasta que a y b sean iguales. ¿Pero cuántas operaciones lleva eso? No n , porque cada vez que entramos en el bucle, dividimos la distancia entre a y b en alrededor de dos. Más bien, la complejidad es $O(\log n)$.

Explicación

Nota: Cuando escribimos "log", nos referimos al logaritmo binario, o log base 2 (que escribiremos "log₂"). Como $O(\log_2 n) = O(\log n)$ (puedes hacer los cálculos) usaremos "log" en lugar de "log₂".

Llamemos a x el número de operaciones: sabemos que $1 = n / (2^x)$.

Entonces $2^x = n$, entonces $x = \log n$

Conclusión

Cuando se enfrente a divisiones sucesivas (ya sea por dos o por cualquier número), recuerde que la complejidad es logarítmica.

$O(\log n)$ tipos de algoritmos

Digamos que tenemos un problema de talla n . Ahora, para cada paso de nuestro algoritmo (que necesitamos escribir), nuestro problema original se convierte en la mitad de su tamaño anterior ($n / 2$).

Así que en cada paso, nuestro problema se convierte en medio.

Paso	Problema
1	$n / 2$
2	$n / 4$
3	$n / 8$
4	$n / 16$

Cuando el espacio del problema se reduce (es decir, se resuelve por completo), no se puede

reducir más (n se vuelve igual a 1) después de salir de la condición de verificación.

1. Digamos en el paso k o número de operaciones:

$$\text{problema-tamaño} = 1$$

2. Pero sabemos que en el paso k, nuestro tamaño del problema debe ser:

$$\text{tamaño del problema} = n / 2^k$$

3. De 1 y 2:

$$n / 2^k = 1 \text{ o}$$

$$n = 2^k$$

4. Tomar registro en ambos lados

$$\log_e n = k \log_e 2$$

o

$$k = \log_e n / \log_e 2$$

5. Usando fórmula $\log_x m / \log_x n = \log_n m$

$$k = \log_2 n$$

o simplemente $k = \log n$

Ahora sabemos que nuestro algoritmo puede ejecutarse al máximo hasta el registro n, por lo que la complejidad del tiempo se presenta como $O(\log n)$

Un ejemplo muy simple en código para soportar el texto anterior es:

```
for(int i=1; i<=n; i=i*2)
{
    // perform some operation
}
```

Entonces, si alguien le pregunta si n es 256 cuántos pasos se ejecutarán en bucle (o cualquier otro algoritmo que reduzca su tamaño del problema a la mitad), puede calcularlo muy fácilmente.

$$k = \log_2 256$$

$$k = \log_2 2^8 \text{ (}\Rightarrow \log_a a = 1\text{)}$$

$$k = 8$$

Otro muy buen ejemplo para un caso similar es el **algoritmo de búsqueda binaria**.

```
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
}
```

Lea Notación Big-O en línea: <https://riptutorial.com/es/algorithm/topic/4770/notacion-big-o>

Capítulo 44: Orden de conteo

Examples

Información básica de orden de conteo

La **ordenación de conteo** es un algoritmo de clasificación de enteros para una colección de objetos que se ordenan según las claves de los objetos.

Pasos

1. Construya una matriz de trabajo C que tenga un tamaño igual al rango de la matriz de entrada A .
2. Iterar a través de A , asignando $C[x]$ en función del número de veces que x apareció en A .
3. Transforme C en una matriz donde $C[x]$ se refiere al número de valores $\leq x$ mediante la iteración a través de la matriz, asignando a cada $C[x]$ la suma de su valor anterior y todos los valores en C que le preceden.
4. Iterar hacia atrás a través de A , colocando cada valor en una nueva matriz ordenada B en el índice registrado en C . Esto se hace para un $A[x]$ dado asignando $B[C[A[x]]]$ a $A[x]$, y disminuyendo $C[A[x]]$ en caso de que hubiera valores duplicados en la matriz original sin clasificar.

Ejemplo de clasificación de conteo

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3
	0	1	2	3	4	5		
C	2	0	2	3	0	1		

(a)

	0	1	2	3	4	5
C	2	2	4	7	7	8

(b)

	1	2	3	4	5	6	7	8
B							3	
	0	1	2	3	4	5		
C	2	2	4	6	7	8		

(c)

	1	2	3	4	5	6	7	8
B	0						3	
	0	1	2	3	4	5		
C	1	2	4	6	7	8		

(d)

	1	2	3	4	5	6	7	8
B	0				3	3		
	0	1	2	3	4	5		
C	1	2	4	5	7	8		

(e)

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

(f)

Espacio auxiliar: $O(n+k)$

Complejidad temporal: Peor caso: $O(n+k)$, Mejor caso: $O(n)$, Caso promedio $O(n+k)$

Implementacion Psuedocode

Restricciones:

1. Entrada (una matriz a ordenar)
2. Número de elemento en la entrada (n)

3. Teclas en el rango de $0..k-1$ (k)
4. Cuenta (una matriz de números)

Pseudocódigo

```
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
```

Implementación de 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;
    }
}
```

Lea Orden de conteo en línea: <https://riptutorial.com/es/algorithm/topic/7251/orden-de-conteo>

Capítulo 45: Ordenación rápida

Observaciones

A veces, Quicksort también se conoce como partición de intercambio.

Espacio auxiliar: $O(n)$

Complejidad del tiempo: peor $O(n^2)$, mejor $O(n \log n)$

Examples

Fundamentos de Quicksort

Quicksort es un algoritmo de clasificación que selecciona un elemento ("el pivote") y reordena la matriz formando dos particiones de tal manera que todos los elementos menos que el pivote vienen antes y todos los elementos más grandes vienen después. El algoritmo se aplica recursivamente a las particiones hasta que se ordena la lista.

1. Mecanismo de esquema de partición de Lomuto:

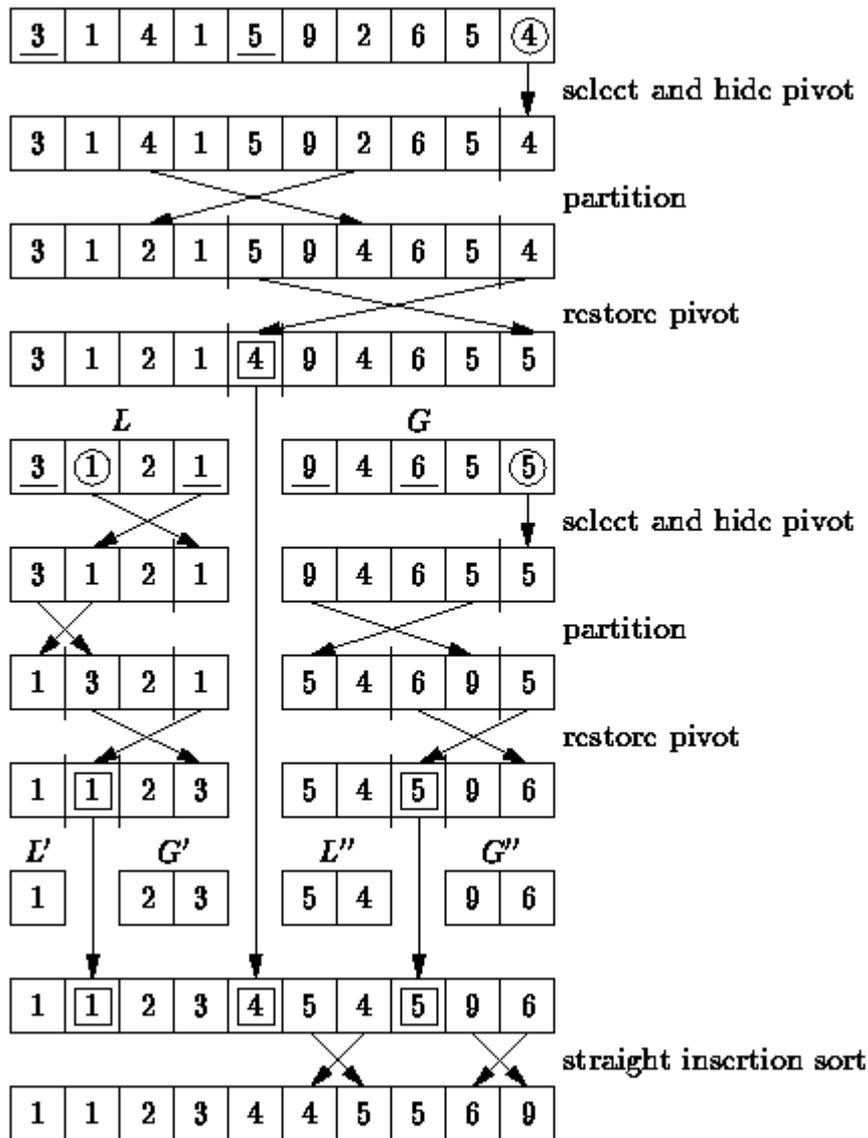
Este esquema elige un pivote que suele ser el último elemento de la matriz. El algoritmo mantiene el índice para poner el pivote en la variable i y cada vez que encuentra un elemento menor o igual que el pivote, este índice se incrementa y ese elemento se coloca antes del pivote.

```
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
```

Mecanismo de clasificación rápida:

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

Ejemplo de clasificación rápida:



2. Esquema de partici3n Hoare:

Utiliza dos 3ndices que comienzan en los extremos de la matriz que se est1 dividiendo, luego se mueven uno hacia el otro, hasta que detectan una inversi3n: un par de elementos, uno mayor o igual que el pivote, uno menor o igual, que est1n en el lugar equivocado orden relativa entre s3. Los elementos invertidos se intercambian. Cuando los 3ndices se encuentran, el algoritmo se detiene y devuelve el 3ndice final. El esquema de Hoare es m1s eficiente que el esquema de partici3n de Lomuto porque hace tres veces menos intercambios en promedio, y crea particiones eficientes incluso cuando todos los valores son iguales.

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

Partici3n

```
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]

```

Implementación de 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;
    }
}
```

Implementación Haskell

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

Lomuto partición java implementacion

```
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;
    }
}
```

Quicksort en 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])
```

Impresiones "[1, 1, 2, 3, 6, 8, 10]"

Lea Ordenación rápida en línea: <https://riptutorial.com/es/algorithm/topic/7232/ordenacion-rapida>

Capítulo 46: Ordenamiento de burbuja

Parámetros

Parámetro	Descripción
Estable	Sí
En su lugar	Sí
La mejor complejidad del caso	En)
Complejidad media del caso	$O(n^2)$
Peor complejidad del caso	$O(n^2)$
Complejidad del espacio	$O(1)$

Examples

Ordenamiento de burbuja

El `BubbleSort` compara cada par de elementos sucesivos en una lista desordenada e invierte los elementos si no están en orden.

El siguiente ejemplo ilustra el ordenamiento de las burbujas en la lista `{6, 5, 3, 1, 8, 7, 2, 4}` (los pares que se compararon en cada paso están encapsulados en `''`):

```
{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
```

Después de una iteración a través de la lista, tenemos `{5, 3, 1, 6, 7, 2, 4, 8}`. Tenga en cuenta que el mayor valor sin clasificar de la matriz (8 en este caso) siempre alcanzará su posición final. Por lo tanto, para asegurarnos de que la lista esté ordenada, debemos iterar $n-1$ veces para las listas de longitud n .

Gráfico:

6 5 3 1 8 7 2 4

Implementación en Javascript

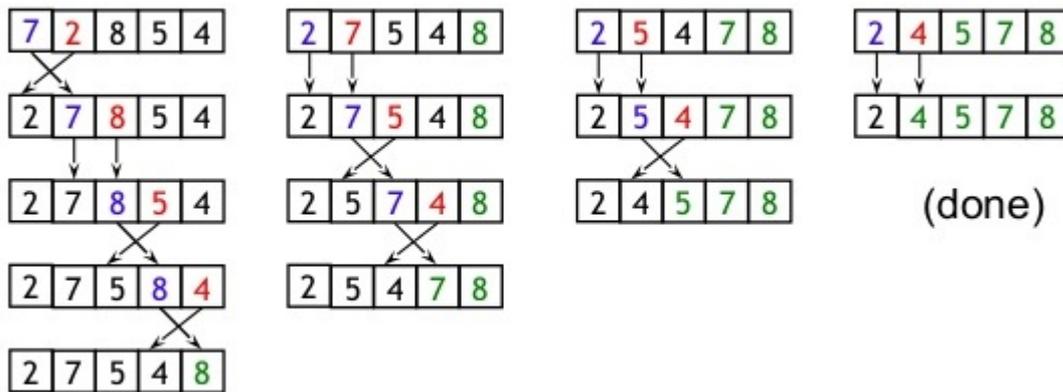
```
function bubbleSort(a)
{
    var swapped;
    do {
        swapped = false;
        for (var i=0; i < a.length-1; i++) {
            if (a[i] > a[i+1]) {
                var temp = a[i];
                a[i] = a[i+1];
                a[i+1] = temp;
                swapped = true;
            }
        }
    } 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 ]
```

Implementación en C

El orden de burbuja también se conoce como **Sinking Sort** . Es un algoritmo de clasificación simple que recorre repetidamente la lista para ser ordenado, compara cada par de elementos adyacentes y los intercambia si están en el orden incorrecto.

Ejemplo de clasificación de burbuja



Implementación de Bubble Sort

Usé el lenguaje C # para implementar el algoritmo de clasificación de burbujas

```
public class BubbleSort
{
    public static void SortBubble(int[] input)
    {
        for (var i = input.Length - 1; i >= 0; i--)
        {
            for (var j = input.Length - 1 - i; 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;
    }
}
```

Implementación en C & C ++.

Una implementación de ejemplo de BubbleSort en 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));
            }
        }
    }
}
```

Implementación 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 Sort con puntero

```
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;
            }
        }
    }
}
```

Implementación en 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);
}
}
}

```

Implementación de 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

```

Lea Ordenamiento de burbuja en línea:

<https://riptutorial.com/es/algorithm/topic/1478/ordenamiento-de-burbuja>

Capítulo 47: Primera búsqueda de profundidad

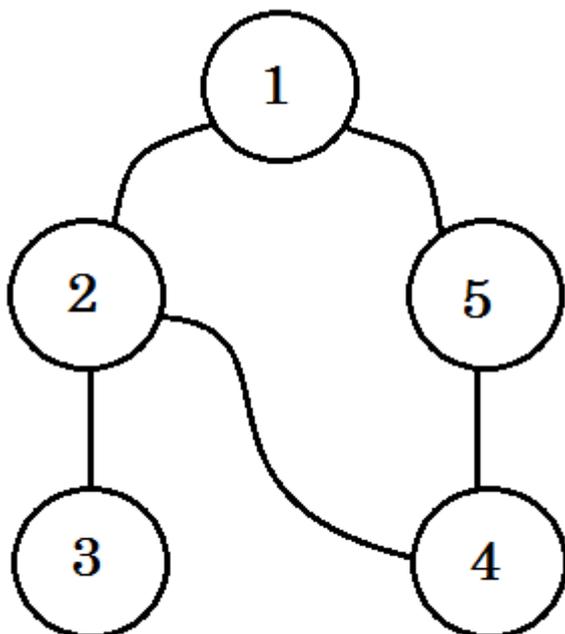
Examples

Introducción a la búsqueda en profundidad primero

La **búsqueda en profundidad** es un algoritmo para atravesar o buscar estructuras de datos de árboles o gráficos. Uno comienza en la raíz y explora en la medida de lo posible a lo largo de cada rama antes de retroceder. Se investigó una versión de la búsqueda en profundidad en el matemático francés del siglo XIX Charles Pierre Trémaux como estrategia para resolver laberintos.

La búsqueda en profundidad es una forma sistemática de encontrar todos los vértices accesibles desde un vértice fuente. Al igual que la búsqueda de amplitud, el DFS atraviesa un componente conectado de un gráfico dado y define un árbol de expansión. La idea básica de la búsqueda en profundidad es explorar metódicamente cada borde. Comenzamos de nuevo desde vértices diferentes según sea necesario. Tan pronto como descubrimos un vértice, DFS comienza a explorar desde él (a diferencia de BFS, que pone un vértice en una cola para que explore más adelante).

Veamos un ejemplo. Atravesaremos este gráfico:

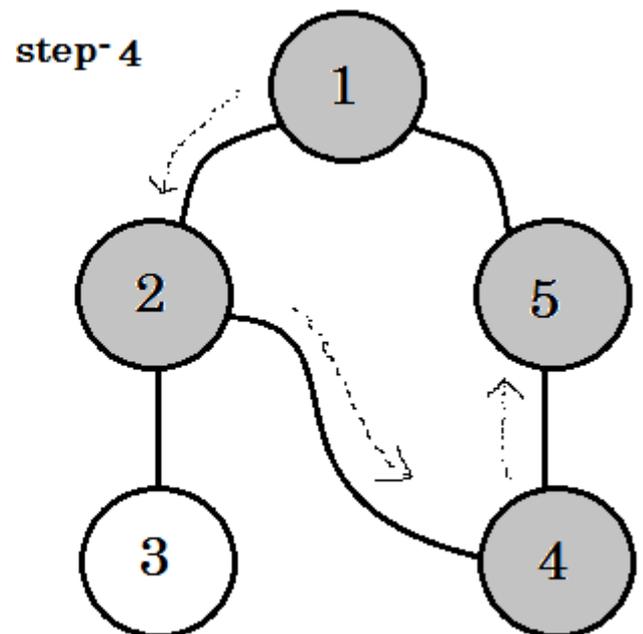
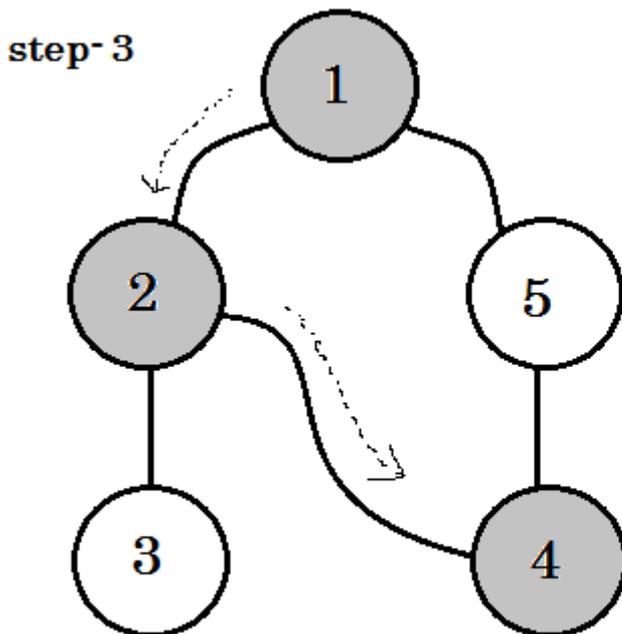
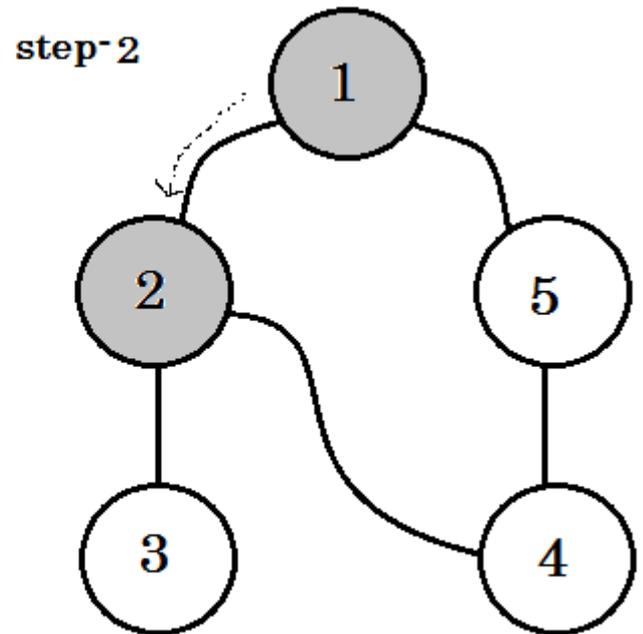
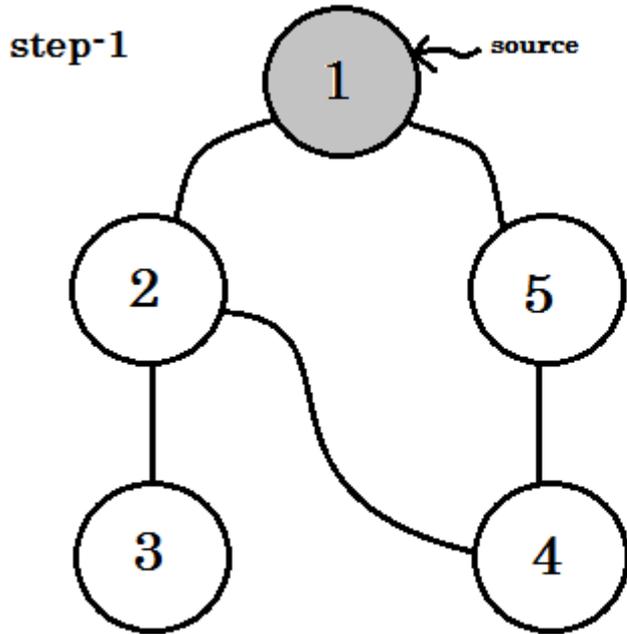


Atravesaremos la gráfica siguiendo estas reglas:

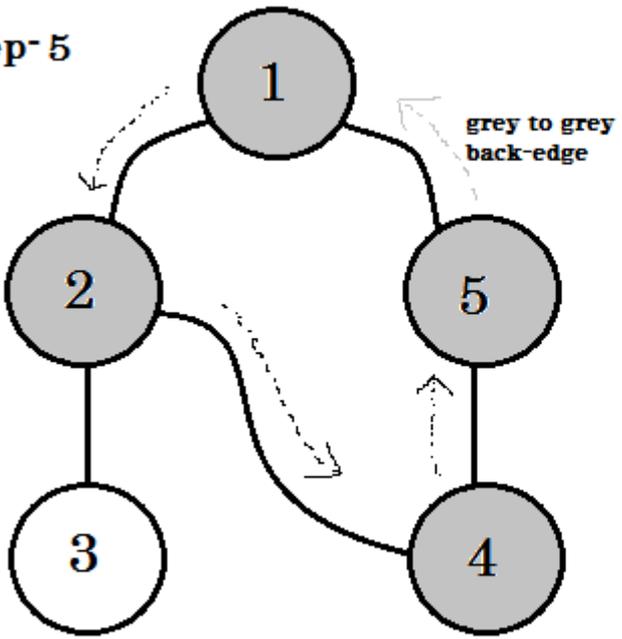
- Empezaremos desde la fuente.
- Ningún nodo será visitado dos veces.

- Los nodos que aún no visitamos, serán de color blanco.
- El nodo que visitamos, pero no visitamos todos sus nodos secundarios, será de color gris.
- Los nodos completamente atravesados serán de color negro.

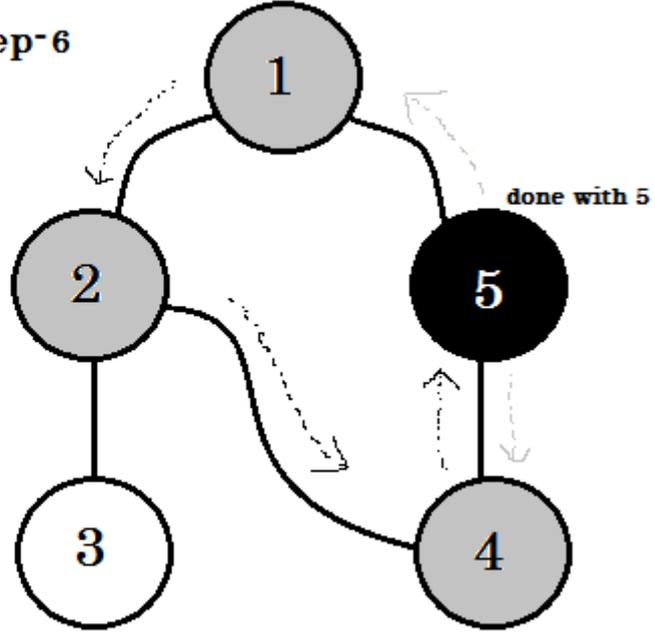
Vamos a verlo paso a paso:



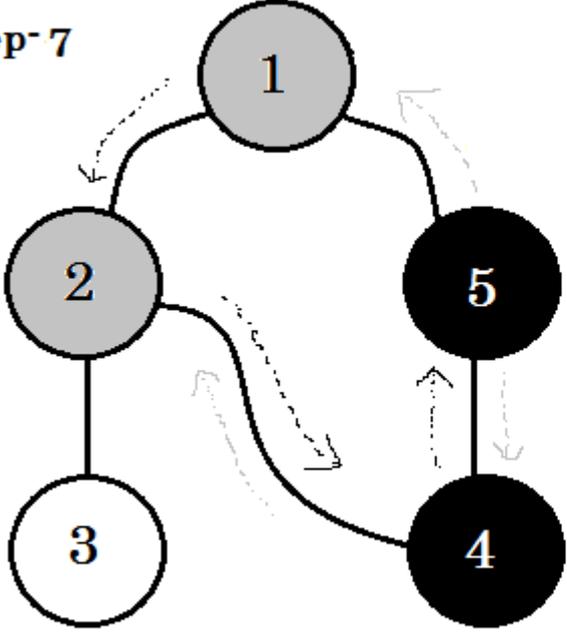
step-5



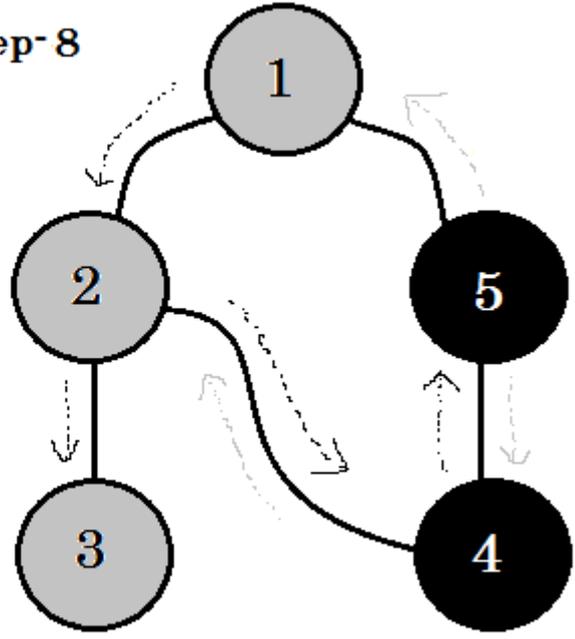
step-6

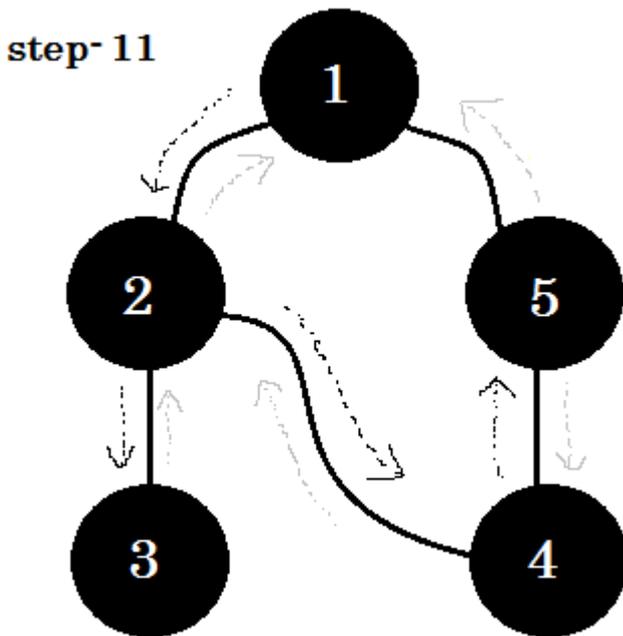
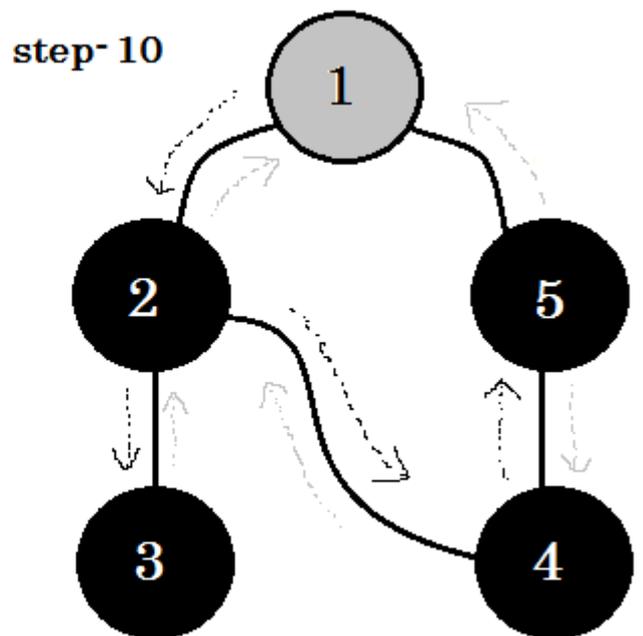
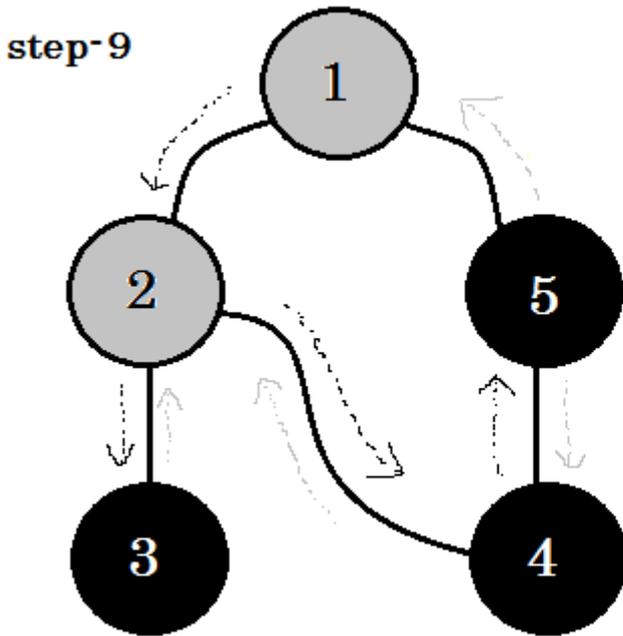


step-7



step-8





Podemos ver una palabra clave importante. Eso es **backedge** . Puedes ver. **5-1** se llama backedge. Esto se debe a que aún no hemos terminado con el **nodo 1** , por lo que pasar de otro nodo al **nodo 1** significa que hay un ciclo en el gráfico. En DFS, si podemos pasar de un nodo gris a otro, podemos estar seguros de que la gráfica tiene un ciclo. Esta es una de las formas de detectar ciclos en una gráfica. Según el nodo de **origen** y el orden de los nodos que visitemos, podemos descubrir cualquier borde en un ciclo como **backedge** . Por ejemplo: si fuimos a **5** de **1** primero, habríamos descubierto **2-1** como backedge.

El borde que tomamos para ir del nodo gris al nodo blanco se llama **borde del árbol** . Si solo mantenemos el **borde del árbol** y eliminamos otros, obtendremos el **árbol DFS** .

En un gráfico no dirigido, si podemos visitar un nodo ya visitado, ese debe ser un **backedge** .

Pero para los gráficos dirigidos, hay que comprobar los colores. *Si y solo si podemos pasar de un nodo gris a otro nodo gris, eso se llama backedge* .

En DFS, también podemos mantener las marcas de tiempo para cada nodo, que se pueden usar de muchas maneras (por ejemplo: Ordenación topológica).

1. Cuando un nodo v cambia de blanco a gris, el tiempo se registra en $d[v]$.
2. Cuando un nodo v cambia de gris a negro, el tiempo se registra en $f[v]$.

Aquí $d[]$ significa *tiempo de descubrimiento* y $f[]$ significa *tiempo de finalización* . Nuestro pseudo-código se verá así:

```
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
```

Complejidad:

Cada nodos y aristas se visitan una vez. Entonces, la complejidad de DFS es $O(V + E)$, donde V denota el número de nodos y E denota el número de bordes.

Aplicaciones de Profundidad Primera Búsqueda:

- Encontrar la ruta más corta de todos los pares en un gráfico no dirigido.
- Ciclo de detección en una gráfica.
- Búsqueda de caminos.
- Clasificación topológica.
- Comprobando si una gráfica es bipartita.
- Encontrar el componente fuertemente conectado.
- Resolviendo puzzles con una solución.

[Lea Primera búsqueda de profundidad en línea:](#)

Capítulo 48: Problema de mochila

Observaciones

El problema de la mochila surge principalmente en los mecanismos de asignación de recursos. El nombre "Mochila" fue introducido por primera vez por [Tobias Dantzig](#).

Espacio auxiliar: $O(nw)$

Complejidad del tiempo $O(nw)$

Examples

Fundamentos del problema de la mochila

El problema : dado un conjunto de elementos en los que cada elemento contiene un peso y un valor, determine el número de cada uno para incluir en una colección de modo que el peso total sea menor o igual a un límite dado y el valor total sea lo más grande posible .

Pseudo código para problema de mochila

Dado:

1. Valores (array v)
2. Pesos (matriz w)
3. Número de elementos distintos (n)
4. Capacidad (W)

```
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])
```

Una implementación simple del pseudo código anterior usando 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))

```

Ejecutando el código: Guarde esto en un archivo llamado knapSack.py

```

$ python knapSack.py
220

```

Complejidad temporal del código anterior: $O(nW)$ donde n es el número de elementos y W es la capacidad de la mochila.

Solución implementada en 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);
    }
}

```

Lea Problema de mochila en línea: <https://riptutorial.com/es/algorithm/topic/7250/problema-de-mochila>

Capítulo 49: Programación dinámica

Introducción

La programación dinámica es un concepto ampliamente utilizado y se utiliza a menudo para la optimización. Se refiere a simplificar un problema complicado dividiéndolo en subproblemas más simples de manera recursiva, por lo general con un enfoque ascendente. Hay dos atributos clave que debe tener un problema para que la programación dinámica sea aplicable "Subestructura óptima" y "Subproblemas superpuestos". Para lograr su optimización, la programación de Dinámica utiliza un concepto llamado Memorización.

Observaciones

La programación dinámica es una mejora de Brute Force; vea [este ejemplo](#) para comprender cómo se puede obtener una solución de programación dinámica de Brute Force.

Una solución de programación dinámica tiene 2 requisitos principales:

1. Problemas superpuestos
2. Subestructura óptima

La superposición de subproblemas significa que los resultados de versiones más pequeñas del problema se reutilizan varias veces para llegar a la solución del problema original.

Subestructura óptima significa que existe un método para calcular un problema a partir de sus subproblemas.

Una solución de programación dinámica tiene 2 componentes principales, el **estado** y la **transición**

El **Estado se** refiere a un subproblema del problema original.

La **transición** es el método para resolver un problema basado en sus subproblemas

El tiempo que tarda una solución de programación dinámica se puede calcular como el $\text{No. of States} * \text{Transition Time}$. Por lo tanto, si una solución tiene N^2 estados y la transición es $O(N)$, entonces la solución tomaría aproximadamente $O(N^3)$ tiempo.

Examples

Problema de mochila

0-1 Mochila

El problema de la mochila es un problema cuando se le asigna un conjunto de artículos, cada uno con un peso, un valor y **exactamente 1 copia** , determina qué artículos deben incluirse en una colección para que el peso total sea menor o igual que un determinado Límite y el valor total es lo más grande posible.

Ejemplo de C ++:

Implementación :

```
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] = -1000000000;
    }
    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];
}
```

Prueba :

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

Salida :

```
3
```

Eso significa que el valor máximo que se puede alcanzar es 3, que se logra al elegir (2,1) y (3,2).

Mochila sin límites

El problema de la mochila sin límites es un problema que, dado un conjunto de elementos, cada uno con un peso, un valor y **copias infinitas** , determina el número de cada artículo que se incluirá en una colección de modo que el peso total sea menor o igual a un límite dado y el valor total es lo más grande posible.

Python (2.7.11) Ejemplo:

Implementación :

```
def unbounded_knapsack(w, v, c): # weight, value and capacity
    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

```

La complejidad de esa implementación es $O(nc)$, que n es el número de elementos.

Prueba :

```

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

print unbounded_knapsack(w, v, 13)

```

Salida :

20

Eso significa que el valor máximo que se puede alcanzar es 20, que se logra al elegir (5, 8), (5, 8) y (3, 4).

Algoritmo de programación de trabajo ponderado

El algoritmo ponderado de programación de trabajos también se puede denotar como algoritmo ponderado de selección de actividades.

El problema es que, dados ciertos trabajos con su hora de inicio y finalización, y el beneficio que obtiene cuando termina el trabajo, ¿cuál es el beneficio máximo que puede obtener dado que no se pueden ejecutar dos trabajos en paralelo?

Este se parece a la Selección de Actividad usando el Algoritmo Greedy, pero hay un giro adicional. Es decir, en lugar de maximizar el número de trabajos finalizados, nos centramos en obtener el máximo beneficio. La cantidad de trabajos realizados no importa aquí.

Veamos un ejemplo:

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

Los trabajos se indican con un nombre, su hora de inicio y finalización y las ganancias. Después de algunas iteraciones, podemos averiguar si realizamos **Job-A** y **Job-E** , podemos obtener el máximo beneficio de 17. ¿Cómo descubrir esto utilizando un algoritmo?

Lo primero que hacemos es clasificar los trabajos por su tiempo de finalización en orden no decreciente. ¿Por qué hacemos esto? Es porque si seleccionamos un trabajo que demora menos tiempo en terminar, dejamos la mayor cantidad de tiempo para elegir otros trabajos. Tenemos:

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

Tendremos una matriz temporal adicional **Acc_Prof** de tamaño **n** (Aquí, **n** indica el número total de trabajos). Esto contendrá el beneficio acumulado máximo de realizar los trabajos. ¿No lo entiendes? Espera y mira. Inicializaremos los valores de la matriz con el beneficio de cada trabajo. Eso significa que **Acc_Prof [i]** al principio tendrá el beneficio de realizar el trabajo **i-th** .

Acc_Prof	5	6	5	4	11	2
----------	---	---	---	---	----	---

Ahora denotemos la **posición 2** con **i** , y la **posición 1** se denotará con **j** . Nuestra estrategia será iterar **j** de **1** a **i-1** y después de cada iteración, incrementaremos **i** en **1**, hasta que **i** se convierta en **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

Verificamos si la **Tarea [i]** y la **Tarea [j]** se superponen, es decir, si la **hora de finalización** de la **Tarea [j]** es mayor que la hora de inicio de la **Tarea [i]** , entonces estas dos **tareas** no se pueden hacer juntas. Sin embargo, si no se superponen, verificaremos si **Acc_Prof [j] + Profit [i] > Acc_Prof [i]** . Si este es el caso, actualizaremos **Acc_Prof [i] = Acc_Prof [j] + Profit [i]** . Es decir:

```

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

```

Aquí, **Acc_Prof [j] + Profit [i]** representa el beneficio acumulado de hacer estos dos trabajos juntos. Vamos a comprobarlo para nuestro ejemplo:

Aquí el **trabajo [j]** se superpone con el **trabajo [i]** . Así que estos no se pueden hacer juntos. Como nuestra **j** es igual a **i-1** , incrementamos el valor de **i** a **i + 1** que es **3** . Y hacemos **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

Ahora **Job [j]** y **Job [i]** no se superponen. La cantidad total de ganancias que podemos obtener al elegir estos dos trabajos es: **Acc_Prof [j] + Profit [i] = 5 + 5 = 10**, que es mayor que **Acc_Prof [i]** . Así que actualizamos **Acc_Prof [i] = 10** . También incrementamos **j** en 1. Obtenemos,

	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

Aquí, la **tarea [j]** se superpone con la **tarea [i]** y **j** también es igual a **i-1** . Entonces incrementamos **i** por 1, y hacemos **j = 1** . Obtenemos,

	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

Ahora, **Job [j]** y **Job [i]** no se superponen, obtenemos el beneficio acumulado **5 + 4 = 9** , que es mayor que **Acc_Prof [i]** . Actualizamos **Acc_Prof [i] = 9** e incrementamos **j** en 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

De nuevo, **Job [j]** y **Job [i]** no se superponen. El beneficio acumulado es: $6 + 4 = 10$, que es mayor que **Acc_Prof [i]**. Nuevamente actualizamos **Acc_Prof [i] = 10**. Incrementamos **j** en 1. Obtenemos:

	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

Si continuamos este proceso, después de recorrer toda la tabla usando **i**, nuestra tabla finalmente se verá así:

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

* Se han saltado algunos pasos para acortar el documento.

¡Si iteramos a través de la matriz **Acc_Prof**, podemos encontrar la ganancia máxima de **17**! El pseudocódigo:

```

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]
  endif
endfor

```

```
return maxProfit
```

La complejidad de **llenar la matriz Acc_Prof** es **$O(n^2)$** . El recorrido transversal de la matriz toma **$O(n)$** . Entonces, la complejidad total de este algoritmo es **$O(n^2)$** .

Ahora, si queremos averiguar qué trabajos se realizaron para obtener el máximo beneficio, debemos atravesar la matriz en orden inverso y si **Acc_Prof** coincide con **maxProfit**, empujaremos el **nombre** del trabajo en una **pila** y restaremos **Ganancia** de ese trabajo de **maxProfit**. Haremos esto hasta que nuestro **maxProfit > 0** o alcancemos el punto de inicio de la matriz **Acc_Prof**. El pseudocódigo se verá así:

```
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
```

La complejidad de este procedimiento es: **$O(n)$** .

Una cosa para recordar, si hay varios horarios de trabajo que pueden darnos el máximo beneficio, solo podemos encontrar un programa de trabajo a través de este procedimiento.

Editar distancia

La declaración del problema es como si nos dieran dos cadenas str1 y str2, entonces, ¿cuántas operaciones mínimas se pueden realizar en el str1 que se convierte en str2?

Implementación en 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()];
}

```

```

}

```

Salida

3

La subsecuencia común más larga

Si nos dan con las dos cadenas, tenemos que encontrar la subsecuencia común más larga presente en ambas.

Ejemplo

LCS para secuencias de entrada "ABCDGH" y "AEDFHR" es "ADH" de longitud 3.

LCS para secuencias de entrada "AGGTAB" y "GXTXAYB" es "GTAB" de longitud 4.

Implementación en 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()];
}
}

```

Salida

4

Número de Fibonacci

Enfoque de abajo hacia arriba para imprimir el número n de Fibonacci utilizando la programación dinámica.

Árbol recursivo

```

                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)

```

Sub-problemas superpuestos

En este caso, fib (0), fib (1) y fib (3) son los subproblemas superpuestos. Fib (0) se repite 3 veces, fib (1) se repite 5 veces y fib (3) se repite 2 veces.

Implementación

```

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];
}

```

Complejidad del tiempo

En)

La subcadena común más larga

Teniendo en cuenta 2 cadenas str1 y str2, tenemos que encontrar la longitud de la subcadena común más larga entre ellas.

Ejemplos

Entrada: X = "abcdxyz", y = "xyzabcd" Salida: 4

La subcadena común más larga es "abcd" y tiene una longitud de 4.

Entrada: X = "zxabcdzey", y = "yzabcdez" Salida: 6

La subcadena común más larga es "abcdez" y tiene una longitud de 6.

Implementación en 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;
}
```

Complejidad del tiempo

$O(m * n)$

Lea Programación dinámica en línea: <https://riptutorial.com/es/algorithm/topic/2345/programacion-dinamica>

Capítulo 50: Pseudocódigo

Observaciones

El pseudocódigo es por definición informal. El objetivo de este tema es describir formas de traducir el código específico del idioma en algo que todos los que tienen conocimientos de programación pueden entender.

El pseudocódigo es una manera importante de describir un algoritmo y es más neutral que dar una implementación específica de lenguaje. Wikipedia a menudo usa algún tipo de pseudocódigo cuando describe un algoritmo

Algunas cosas, como las condiciones de tipo if-else son bastante fáciles de escribir de manera informal. Pero otras cosas, por ejemplo, las devoluciones de llamada de estilo js, pueden ser difíciles de convertir en pseudocódigo para algunas personas.

Es por esto que estos ejemplos pueden resultar útiles.

Examples

Afectaciones variables

Podrías describir la afectación variable de diferentes maneras.

Mecanografiado

```
int a = 1
int a := 1
let int a = 1
int a <- 1
```

Sin tipo

```
a = 1
a := 1
let a = 1
a <- 1
```

Funciones

Siempre que el nombre de la función, la declaración de retorno y los parámetros estén claros, está bien.

```
def incr n
  return n + 1
```

0

```
let incr(n) = n + 1
```

0

```
function incr (n)  
  return n + 1
```

son muy claros, así que puedes usarlos. Intenta no ser ambiguo con una afectación variable.

Lea Pseudocódigo en línea: <https://riptutorial.com/es/algorithm/topic/7393/pseudocodigo>

Capítulo 51: Radix Sort

Examples

Radix Sort Información Básica

Radix Sort es un algoritmo basado en comparación de límite inferior. Es un algoritmo de ordenación de enteros no comparativo que ordena los datos con claves de enteros agrupando las claves por dígitos individuales que comparten alguna posición y valor significativos. La clasificación Radix es un algoritmo de clasificación de tiempo lineal que se ordena en tiempo $O(n + k)$ cuando los elementos están en el rango de 1 a k . La idea de Radix Sort es hacer una clasificación de dígito por dígito desde el dígito menos significativo hasta el dígito más significativo. La ordenación de radix usa la ordenación de conteo como una subrutina para ordenar. La clasificación de radix es una generalización de la clasificación de cubo.

Pseudo código para Bucket Sort:

1. Crea una matriz de $[0..n-1]$ elementos.
2. Llame a Bucket Sort repetidamente en el dígito de menor a más significativo de cada elemento como clave.
3. Devuelve la matriz ordenada.

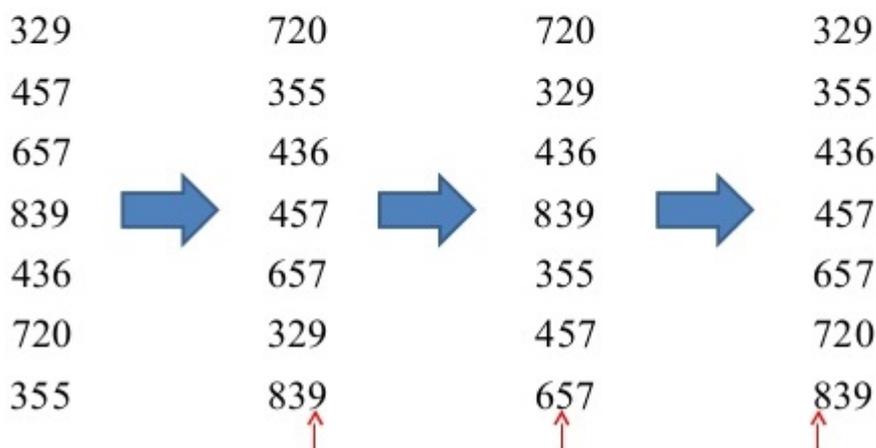
Ejemplo de clasificación de radix:

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 ;



Espacio auxiliar: $O(n)$

Complejidad del tiempo: $O(n)$

Lea Radix Sort en línea: <https://riptutorial.com/es/algorithm/topic/7314/radix-sort>

Capítulo 52: Resolución de ecuaciones

Examples

Ecuación lineal

Hay dos clases de métodos para resolver ecuaciones lineales:

1. **Métodos directos** : las características comunes de los métodos directos son que transforman la ecuación original en ecuaciones equivalentes que se pueden resolver más fácilmente, lo que significa que podemos resolver directamente de una ecuación.
2. **Método iterativo** : Métodos iterativos o indirectos, comience con una conjetura de la solución y luego afine repetidamente la solución hasta que se alcance un cierto criterio de convergencia. Los métodos iterativos son generalmente menos eficientes que los métodos directos porque requieren un gran número de operaciones. Ejemplo: Método de iteración de Jacobi, Método de iteración de Gauss-Seidal.

Implementación en 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;
}

```

Ecuación no lineal

Una ecuación del tipo $f(x)=0$ es algebraica o trascendental. Estos tipos de ecuaciones se pueden resolver utilizando dos tipos de métodos:

1. **Método directo** : este método proporciona el valor exacto de todas las raíces directamente en un número finito de pasos.
2. **Método indirecto o iterativo** : los métodos iterativos son los más adecuados para que los programas de computadora resuelvan una ecuación. Se basa en el concepto de aproximación sucesiva. En el método iterativo hay dos formas de resolver una ecuación:
 - **Método de horquillado** : Tomamos dos puntos iniciales donde la raíz se encuentra entre ellos. Ejemplo: Método de bisección, Método de posición falsa.
 - **Método de final abierto** : Tomamos uno o dos valores iniciales donde la raíz puede estar en cualquier lugar. Ejemplo: método de Newton-Raphson, método de aproximación sucesiva, método secante.

Implementación en 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;
}

```

Lea Resolución de ecuaciones en línea: <https://riptutorial.com/es/algorithm/topic/7379/resolucion-de-ecuaciones>

Capítulo 53: Selección de selección

Examples

Selección Ordenar Información Básica

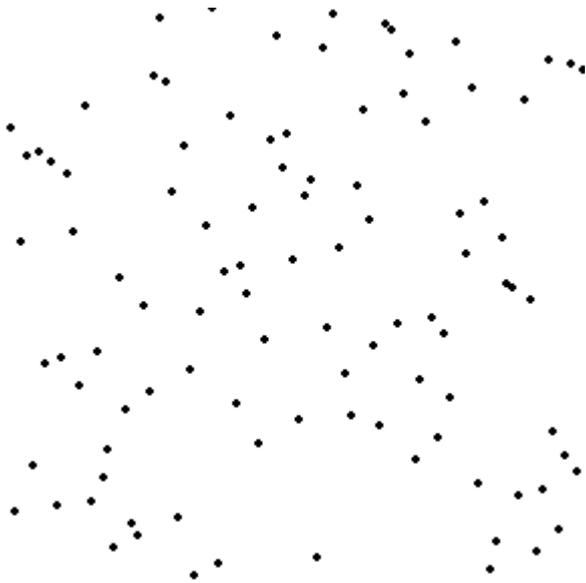
La **selección de selección** es un algoritmo de clasificación, específicamente una clasificación de comparación in situ. Tiene una complejidad de tiempo $O(n^2)$, lo que la hace ineficiente en listas grandes y, en general, tiene un rendimiento peor que el orden de inserción similar. El orden de selección se destaca por su simplicidad y tiene ventajas de rendimiento sobre algoritmos más complicados en ciertas situaciones, particularmente cuando la memoria auxiliar es limitada.

El algoritmo divide la lista de entrada en dos partes: la lista secundaria de elementos ya ordenados, que se construye de izquierda a derecha en la parte delantera (izquierda) de la lista, y la lista secundaria de elementos pendientes de clasificación que ocupan el resto de la lista. Inicialmente, la lista secundaria ordenada está vacía y la lista secundaria sin clasificar es la lista de entrada completa. El algoritmo procede al encontrar el elemento más pequeño (o más grande, según el orden de clasificación) en la lista secundaria sin clasificar, intercambiarlo (intercambiarlo) con el elemento sin clasificar más a la izquierda (ordenándolo) y mover los límites de la lista secundaria un elemento a la derecha .

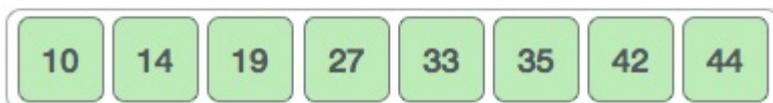
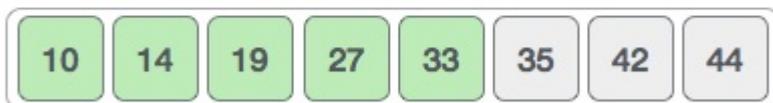
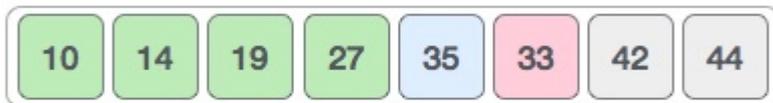
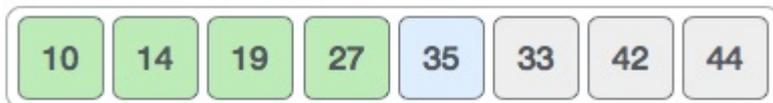
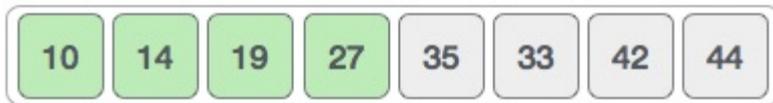
Pseudo código para selección de selección:

```
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]
```

Visualización del orden de selección:



Ejemplo de selección por selección:



Espacio auxiliar: $O(n)$

Complejidad del tiempo: $O(n^2)$

Implementación del ordenamiento de selección en C

Utilicé el lenguaje C # para implementar el algoritmo de selección de selección.

```
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;
    }
}
```

Implementación de Elixir

```
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
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
```

Lea Selección de selección en línea: <https://riptutorial.com/es/algorithm/topic/7473/seleccion-de-seleccion>

Capítulo 54: Shell Sort

Examples

Información básica de Shell Sort

La [clasificación de shell](#) , también conocida como la clasificación de incremento decreciente, es uno de los algoritmos de clasificación más antiguos, que lleva el nombre de su inventor [Donald. L. Shell](#) (1959). Es rápido, fácil de entender y fácil de implementar. Sin embargo, su análisis de complejidad es un poco más sofisticado.

La idea de la ordenación de Shell es la siguiente:

1. Organizar la secuencia de datos en una matriz bidimensional.
2. Ordenar las columnas de la matriz.

La ordenación de la carcasa mejora la ordenación por inserción. Comienza comparando elementos muy separados, luego elementos menos separados, y finalmente comparando elementos adyacentes (de hecho, una clasificación por inserción).

El efecto es que la secuencia de datos está parcialmente ordenada. El proceso anterior se repite, pero cada vez con una matriz más estrecha, es decir, con un número menor de columnas. En el último paso, la matriz consta de una sola columna.

Ejemplo de ordenación de Shell:

Shellsort example



Sort every 5th element:



Sort every 3rd element:



Final a normal insertion sort:



Notice that by the time we do this last insertion sort elements don't have a long way to go before being ins

Pseudo código para Shell Sort:

```
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
  }
}
```

Espacio auxiliar: $O(n)$ total, $O(1)$ auxiliary

Complejidad del tiempo: $O(n \log n)$

Implementación de 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;
    }
}
```

Lea Shell Sort en línea: <https://riptutorial.com/es/algorithm/topic/7454/shell-sort>

Capítulo 55: Time Warping dinámico

Examples

Introducción a la distorsión de tiempo dinámico

Dynamic Time Warping (DTW) es un algoritmo para medir la similitud entre dos secuencias temporales que pueden variar en velocidad. Por ejemplo, las similitudes en la marcha se pueden detectar utilizando DTW, incluso si una persona caminaba más rápido que la otra, o si hubo aceleraciones y desaceleraciones durante el curso de una observación. Se puede usar para hacer coincidir un comando de voz de muestra con el comando de otros, incluso si la persona habla más rápido o más lento que la voz de muestra pregrabada. DTW puede aplicarse a secuencias temporales de datos de video, audio y gráficos; de hecho, cualquier información que pueda convertirse en una secuencia lineal puede analizarse con DTW.

En general, DTW es un método que calcula una coincidencia óptima entre dos secuencias dadas con ciertas restricciones. Pero sigamos con los puntos más simples aquí. Digamos, tenemos dos secuencias de voces **Muestra y Prueba**, y queremos comprobar si estas dos secuencias coinciden o no. Aquí la secuencia de voz se refiere a la señal digital convertida de su voz. Podría ser la amplitud o frecuencia de su voz que denota las palabras que dice. Asumamos:

```
Sample = {1, 2, 3, 5, 5, 5, 6}
Test   = {1, 1, 2, 2, 3, 5}
```

Queremos encontrar la coincidencia óptima entre estas dos secuencias.

Al principio, definimos la distancia entre dos puntos, $d(x, y)$ donde x e y representan los dos puntos. Dejar,

```
d(x, y) = |x - y| //absolute difference
```

Vamos a crear una **tabla de** matriz 2D utilizando estas dos secuencias. Calcularemos las distancias entre cada punto de **muestra** con cada punto de **prueba** y encontraremos la coincidencia óptima entre ellos.

		0	1	1	2	2	3	5
0								
1								
2								
3								
5								

5								
5								
6								

Aquí, la **Tabla [i][j]** representa la distancia óptima entre dos secuencias si consideramos la secuencia hasta la **Muestra [i]** y la **Prueba [j]**, considerando todas las distancias óptimas que observamos antes.

Para la primera fila, si no tomamos valores de **Muestra**, la distancia entre esto y **Prueba** será *infinita*. Así que ponemos *infinito* en la primera fila. Lo mismo ocurre con la primera columna. Si no tomamos valores de **Test**, la distancia entre éste y **Sample** también será infinita. Y la distancia entre **0** y **0** será simplemente **0**. Obtenemos,

	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						

Ahora, para cada paso, consideraremos la distancia entre cada punto en cuestión y la agregaremos con la distancia mínima que encontramos hasta ahora. Esto nos dará la distancia óptima de dos secuencias hasta esa posición. Nuestra fórmula será,

$$\text{Table}[i][j] := d(i, j) + \min(\text{Table}[i-1][j], \text{Table}[i-1][j-1], \text{Table}[i][j-1])$$

Para el primero, $d(1, 1) = 0$, la **Tabla [0][0]** representa el mínimo. Entonces el valor de la **Tabla [1][1]** será $0 + 0 = 0$. Para el segundo, $d(1, 2) = 0$. La **tabla [1][1]** representa el mínimo. El valor será: **Tabla [1][2] = 0 + 0 = 0**. Si continuamos de esta manera, después de terminar, la tabla se verá así:

	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	
+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+

El valor en la **Tabla [7] [6]** representa la distancia máxima entre estas dos secuencias dadas. Aquí **1** representa la distancia máxima entre la **muestra** y la **prueba** es **1** .

Ahora, si retrocedemos desde el último punto, hacia el punto de inicio **(0, 0)** , obtendremos una línea larga que se mueve horizontal, vertical y diagonalmente. Nuestro procedimiento de backtracking será:

```

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

```

Continuaremos esto hasta llegar a **(0, 0)** . Cada movimiento tiene su propio significado:

- Un movimiento horizontal representa la eliminación. Eso significa que nuestra secuencia de **prueba** se aceleró durante este intervalo.
- Un movimiento vertical representa la inserción. Eso significa que la secuencia de **prueba** se desaceleró durante este intervalo.
- Un movimiento diagonal representa un partido. Durante este periodo la **prueba** y la **muestra** fueron iguales.

	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

Nuestro pseudo-código será:

```

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]

```

También podemos agregar una restricción de localidad. Es decir, requerimos que si la `Sample[i]` coincide con la `Test[j]`, entonces $|i - j|$ no es más grande que w , un parámetro de ventana.

Complejidad:

La complejidad de calcular el DTW es $O(m * n)$ donde m y n representan la longitud de cada secuencia. Las técnicas más rápidas para computar DTW incluyen PrunedDTW, SparseDTW y FastDTW.

Aplicaciones:

- Reconocimiento de palabras habladas
- Análisis de poder de correlación

Lea Time Warping dinámico en línea: <https://riptutorial.com/es/algorithm/topic/7584/time-warping-dinamico>

Capítulo 56: Tipo de casillero

Examples

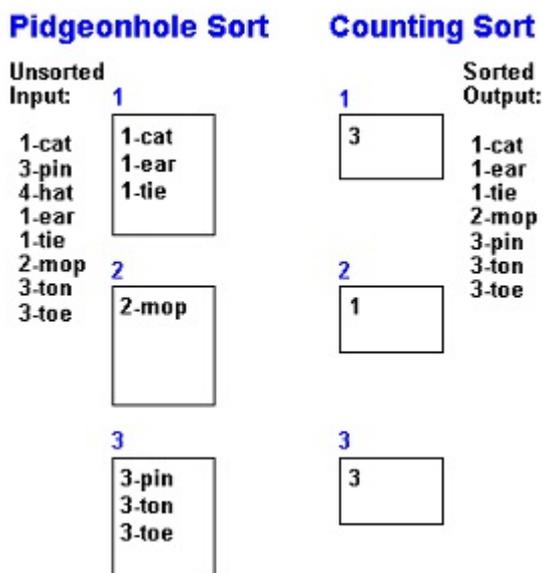
Pigeonhole Sort Información Básica

Pigeonhole Sort es un algoritmo de clasificación que es adecuado para ordenar listas de elementos donde el número de elementos (n) y el número de valores clave posibles (N) son aproximadamente los mismos. Requiere un tiempo $O(n + \text{Rango})$ donde n es el número de elementos en la matriz de entrada y 'Rango' es el número de valores posibles en la matriz.

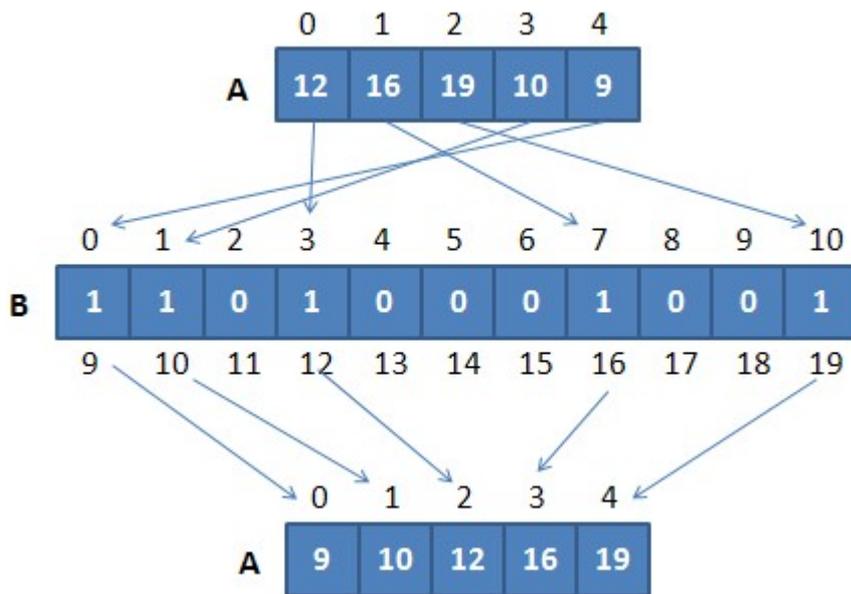
Trabajo (pseudo código) para la clasificación de casilleros:

1. Encuentra valores mínimos y máximos en matriz. Deje que los valores mínimo y máximo sean 'min' y 'max' respectivamente. También encuentra el rango como 'max-min-1'.
2. Configure una matriz de "casilleros" inicialmente vacíos del mismo tamaño que el rango.
3. Visita cada elemento de la matriz y luego coloca cada elemento en su casillero. Una entrada de elemento $[i]$ se coloca en el orificio en la entrada de índice $[i] - \text{mín}$.
4. Inicie el bucle en toda la matriz de casilleros en orden y vuelva a colocar los elementos de los orificios no vacíos en la matriz original.

La clasificación de Pigeonhole es similar a la clasificación de recuento, por lo que aquí hay una comparación entre la Clasificación de Pigeonhole y la clasificación de recuento.



Ejemplo de clasificación de casillero:



Espacio auxiliar: $O(n)$

Complejidad de tiempo: $O(n + N)$

Implementación de 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;
    }
}
```

```
}
```

Lea Tipo de casillero en línea: <https://riptutorial.com/es/algorithm/topic/7310/tipo-de-casillero>

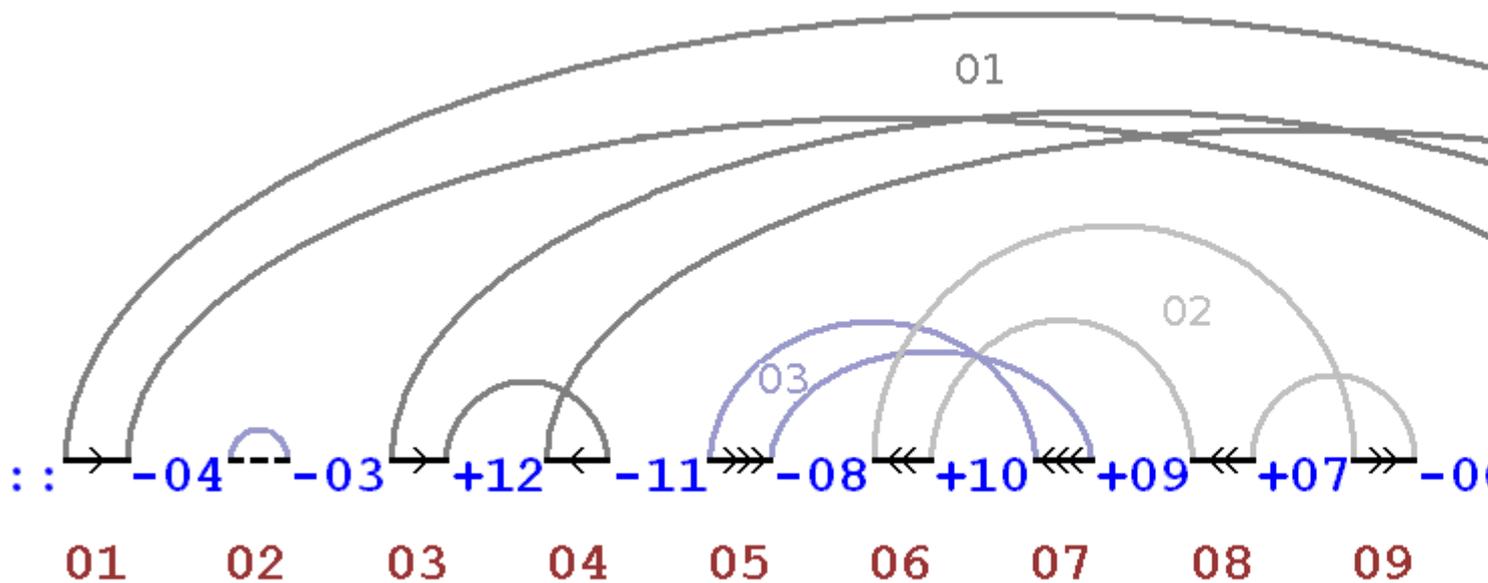
Capítulo 57: Tipo de ciclo

Examples

Ciclo de Información Básica

Cycle Sort es un algoritmo de clasificación que utiliza una **clasificación de comparación** que es teóricamente óptima en términos del número total de escrituras en la matriz original, a diferencia de cualquier otro algoritmo de clasificación en el lugar. La clasificación del ciclo es un algoritmo de clasificación inestable. Se basa en la idea de permutación en la que las permutaciones se dividen en ciclos, que rotan individualmente y devuelven una salida ordenada.

Ejemplo de clasificación del ciclo:



Espacio auxiliar: $O(1)$

Complejidad del tiempo: $O(n^2)$

Implementación de pseudocódigo

```
(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

```

Implementación de 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;
    }
}

```

Lea Tipo de ciclo en línea: <https://riptutorial.com/es/algorithm/topic/7252/tipo-de-ciclo>

Capítulo 58: Tipo de cubo

Examples

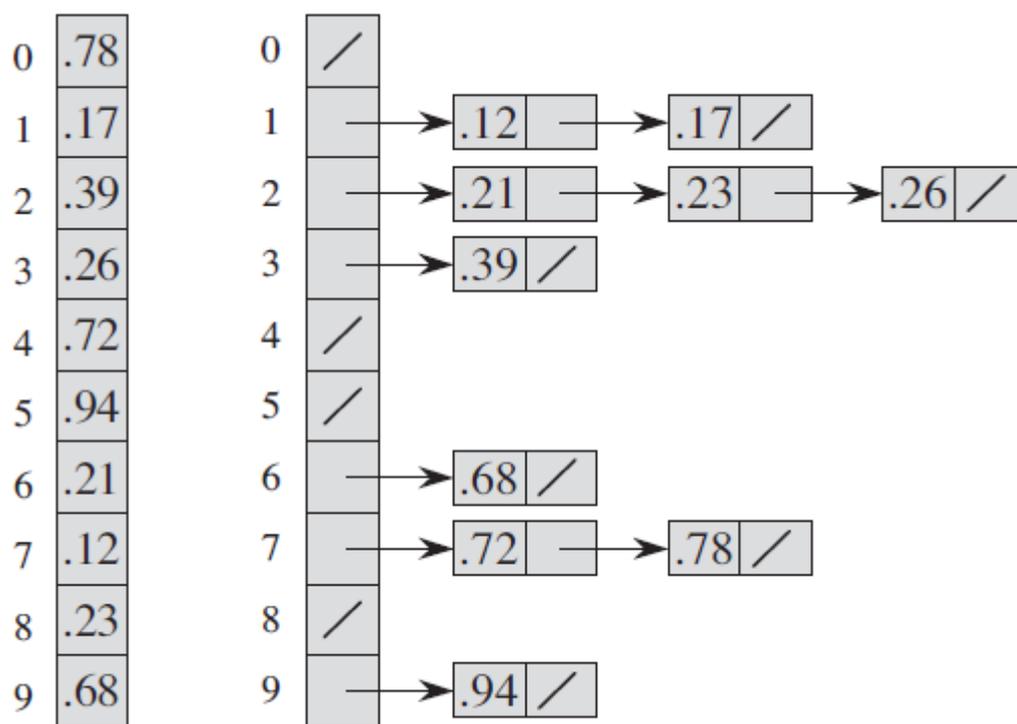
Bucket Sort Información Básica

La **clasificación de cubo** es un algoritmo de clasificación en el que los elementos de la matriz de entrada se distribuyen en grupos. Después de distribuir todos los elementos, los grupos se ordenan individualmente por otro algoritmo de clasificación. A veces también se ordena por método recursivo.

Pseudo código para Bucket Sort

1. Sea n la longitud de la lista de entrada L ;
2. Para cada elemento i de L
3. Si $B[i]$ no está vacío
4. Pon $A[i]$ en $B[i]$;
5. Else $B[i] := A[i]$
6. devuelva el Concat $B[i .. n]$ en una lista ordenada;

Ejemplo de clasificación de cubo:



Input array

Buckets created for input array

En su mayoría, la gente usa el paradigma de inserción para un poco de optimización.

Espacio auxiliar: $O\{n\}$

Implementación de 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;
    }
}

```

Lea Tipo de cubo en línea: <https://riptutorial.com/es/algorithm/topic/7230/tipo-de-cubo>

Capítulo 59: Tipo de inserción

Observaciones

Cuando analizamos el rendimiento del algoritmo de clasificación, nos interesamos principalmente en el número de comparación e intercambio.

Media de intercambio

Sea E_n el número promedio total de intercambio para ordenar la matriz del elemento N . $E_1 = 0$ (no necesitamos ningún intercambio para la matriz con un elemento). El número promedio de intercambio para ordenar la matriz de elementos N es la suma del número promedio de intercambio para ordenar la matriz de elementos $N-1$ con el intercambio promedio para insertar el último elemento en la matriz de elementos $N-1$.

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

Simplificar la suma (serie aritmética).

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

Expande el termino

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

Simplifica la suma de nuevo (series aritméticas)

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

Comparación de promedios

Sea C_n el número promedio total de comparación para ordenar la matriz del elemento N . $C_1 = 0$ (no necesitamos ninguna comparación en una matriz de elementos). El número promedio de comparación para ordenar la matriz de elementos N es la suma del número promedio de comparación para ordenar la matriz de elementos $N-1$ con la comparación promedio para insertar el último elemento en la matriz de elementos $N-1$. Si el último elemento es el elemento más grande, solo necesitamos una comparación, si el último elemento es el segundo elemento más pequeño, necesitamos la comparación $N-1$. Sin embargo, si el último elemento es el elemento más pequeño, no necesitamos la comparación N . Todavía solo necesitamos la comparación $N-1$. Es por eso que eliminamos $1/N$ en la siguiente ecuación.

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

Simplificar la suma (serie aritmética).

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

Ampliar el término

$$C_n = \left(\frac{n+1}{2} + \dots + \frac{3}{2}\right) - \left(\frac{1}{n} + \dots + \frac{1}{2}\right)$$

Simplifique nuevamente la suma (serie aritmética y número de armónicos)

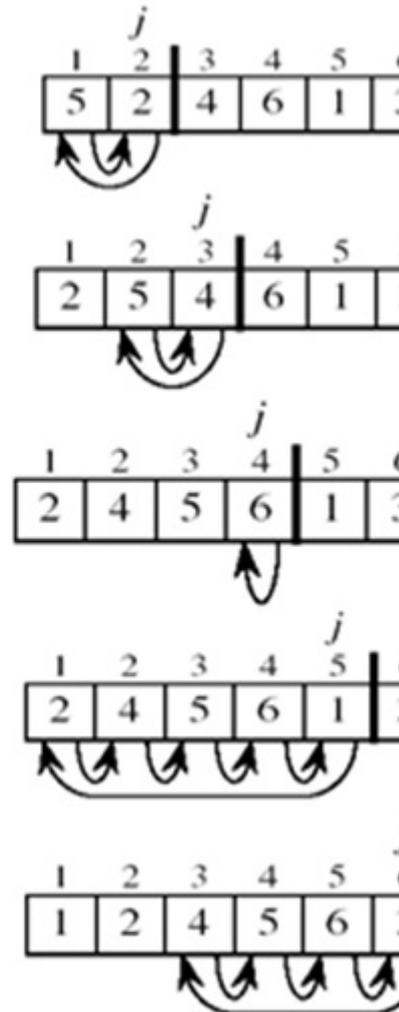
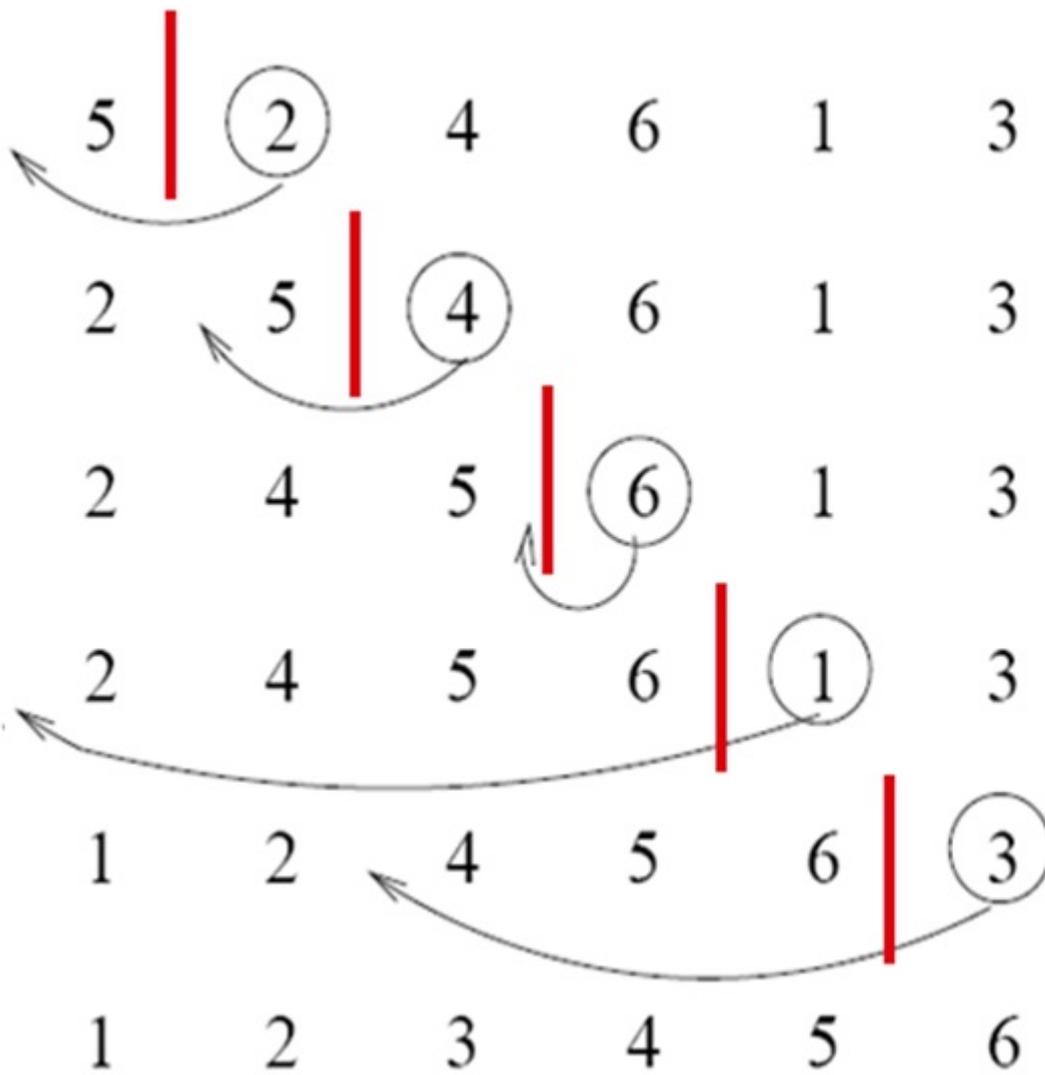
$$C_n = \frac{(n+1)(n+2)}{4} - H_n - 0.5$$

Examples

Fundamentos del algoritmo

La ordenación por inserción es un algoritmo de clasificación en sitio, muy simple y estable. Se desempeña bien en secuencias pequeñas, pero es mucho menos eficiente en listas grandes. En cada paso, los algoritmos consideran el elemento i -th de la secuencia dada, moviéndolo hacia la izquierda hasta que esté en la posición correcta.

Ilustración gráfica



Done !

Pseudocódigo

```

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

```

Ejemplo

Considere la siguiente lista de enteros:

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

El algoritmo realizará los siguientes pasos:

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]

Implementación de 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;
    }
}
```

Espacio auxiliar: $O(1)$

Complejidad del tiempo: $O(n)$

Implementación 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
```

Lea Tipo de inserción en línea: <https://riptutorial.com/es/algorithm/topic/5738/tipo-de-insercion>

Capítulo 60: Tipo de panqueque

Examples

Información básica sobre el tipo de panqueque

Pancake Sort es un término coloquial para el problema matemático de clasificar una pila desordenada de pancakes en orden de tamaño cuando se puede insertar una espátula en cualquier punto de la pila y se usa para voltear todas las pancakes por encima. Un número de panqueques es el número mínimo de tirones requeridos para un número dado de panqueques.

A diferencia de un algoritmo de clasificación tradicional, que intenta clasificar con la menor cantidad de comparaciones posible, el objetivo es clasificar la secuencia en la menor cantidad de inversiones posible.

La idea es hacer algo similar a la selección por selección. Colocamos el elemento máximo uno por uno al final y reducimos el tamaño de la matriz actual en uno.

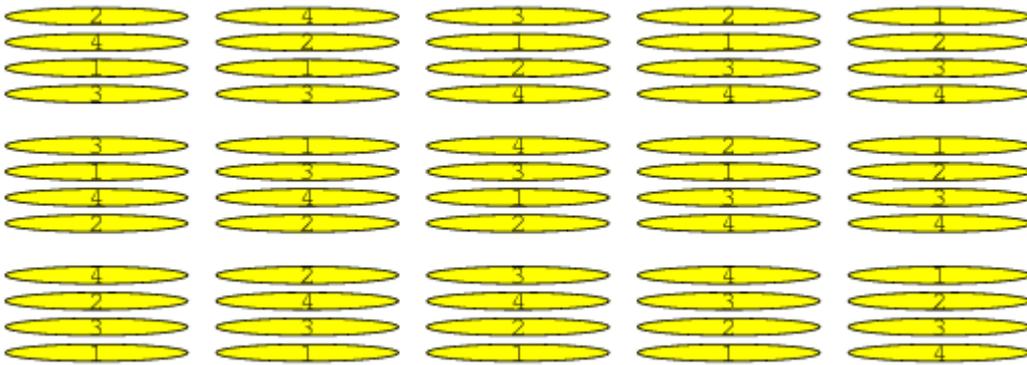
Disecionando el problema:

1. Si necesita ordenar los panqueques de la más pequeña (superior) a la mayor (inferior), la pila de inicio se puede organizar en cualquier orden.
2. Solo puedo realizar voltear todo el stack.
3. Para voltear un panqueque específico a la parte inferior de la pila, primero debemos voltearlo a la parte superior (luego voltearlo nuevamente a la parte inferior).
4. Para ordenar cada panqueque se requerirá un giro hacia arriba y otro hacia abajo hasta su ubicación final.

Algoritmo intuitivo:

1. Encuentre el panqueque más grande que esté fuera de orden y colóquelo en la parte inferior (es posible que primero tenga que voltearlo hacia la parte superior de la pila).
2. Repita el paso uno hasta que se ordene la pila.
3. Eso es todo, un algoritmo de dos pasos funcionará.

Ejemplo de algoritmo de clasificación de panqueques:



Espacio auxiliar: $O(1)$

Complejidad del tiempo: $O(n^2)$

Implementación de 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;
    }
}
```

```
}
```

Lea Tipo de panqueque en línea: <https://riptutorial.com/es/algorithm/topic/7501/tipo-de-panqueque>

Capítulo 61: Tipo impar-par

Examples

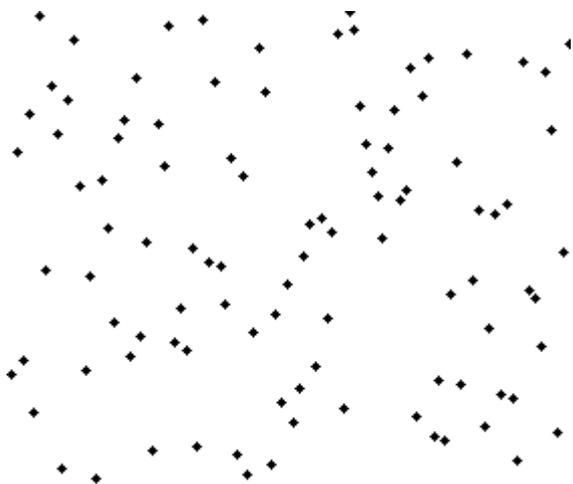
Información básica de orden impar

Una [clasificación impar-par](#) o de ladrillo es un algoritmo de clasificación simple, que se desarrolla para su uso en procesadores paralelos con interconexión local. Funciona comparando todos los pares impares / pares de elementos adyacentes en la lista y, si un par está en el orden incorrecto, los elementos se cambian. El siguiente paso repite esto para pares indexados pares / impares. Luego se alterna entre pasos impares / pares e pares / impares hasta que se ordena la lista.

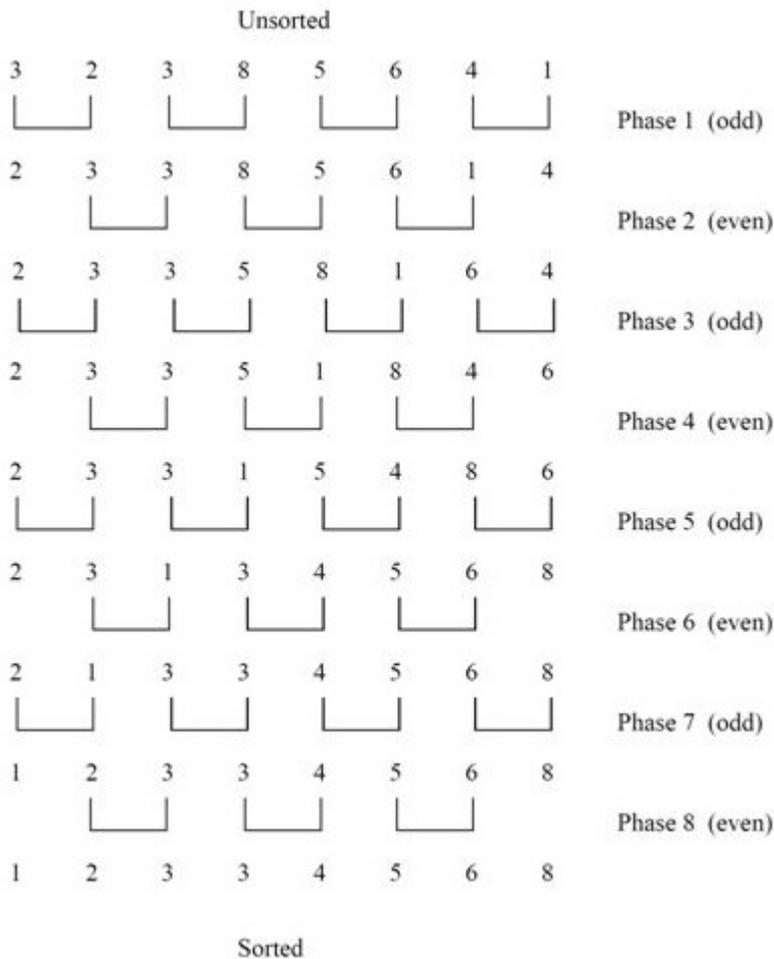
Pseudo código para Odd-Even Sort:

```
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]
```

Wikipedia tiene mejor ilustración del tipo Odd-Even:



Ejemplo de tipo impar-par:



Implementación:

Utilicé el lenguaje C # para implementar el algoritmo de ordenación impar.

```
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;
}
}
```

Espacio auxiliar: $O(n)$

Complejidad del tiempo: $O(n)$

Lea Tipo impar-par en línea: <https://riptutorial.com/es/algorithm/topic/7386/tipo-impar-par>

Capítulo 62: Transformada rápida de Fourier

Introducción

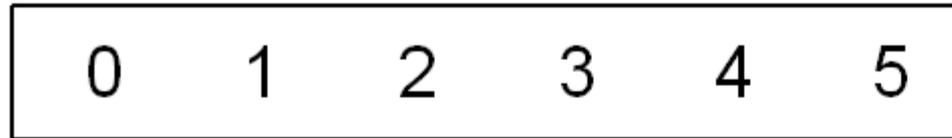
La forma real y compleja de DFT (**D**iscrete **F**ourier **T**ransforms) se puede utilizar para realizar análisis de frecuencia o síntesis para cualquier señal discreta y periódica. La FFT (**F**ast **F**ORREO **T**ransform) es una implementación de la DFT que puede realizarse de forma rápida en las CPU modernas.

Examples

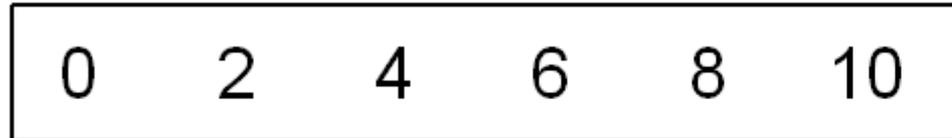
Radix 2 FFT

El método más simple y quizás el más conocido para calcular la FFT es el algoritmo de diezmado en el tiempo Radix-2. El Radix-2 FFT funciona al descomponer una señal de dominio de tiempo de punto N en N señales de dominio de tiempo cada una compuesta de un solo punto

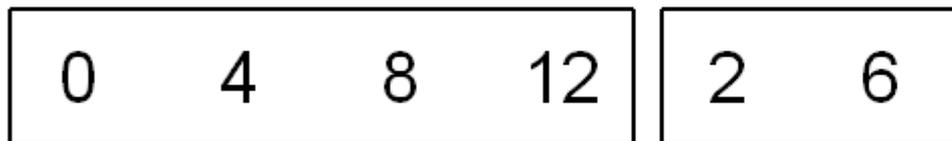
1 signal of
16 points



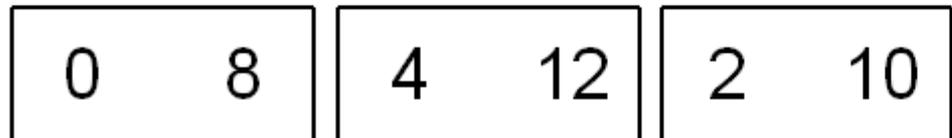
2 signals of
8 points



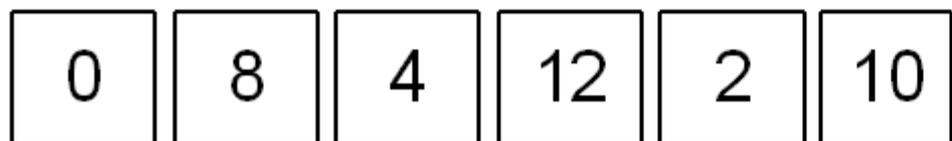
4 signals of
4 points



8 signals of
2 points



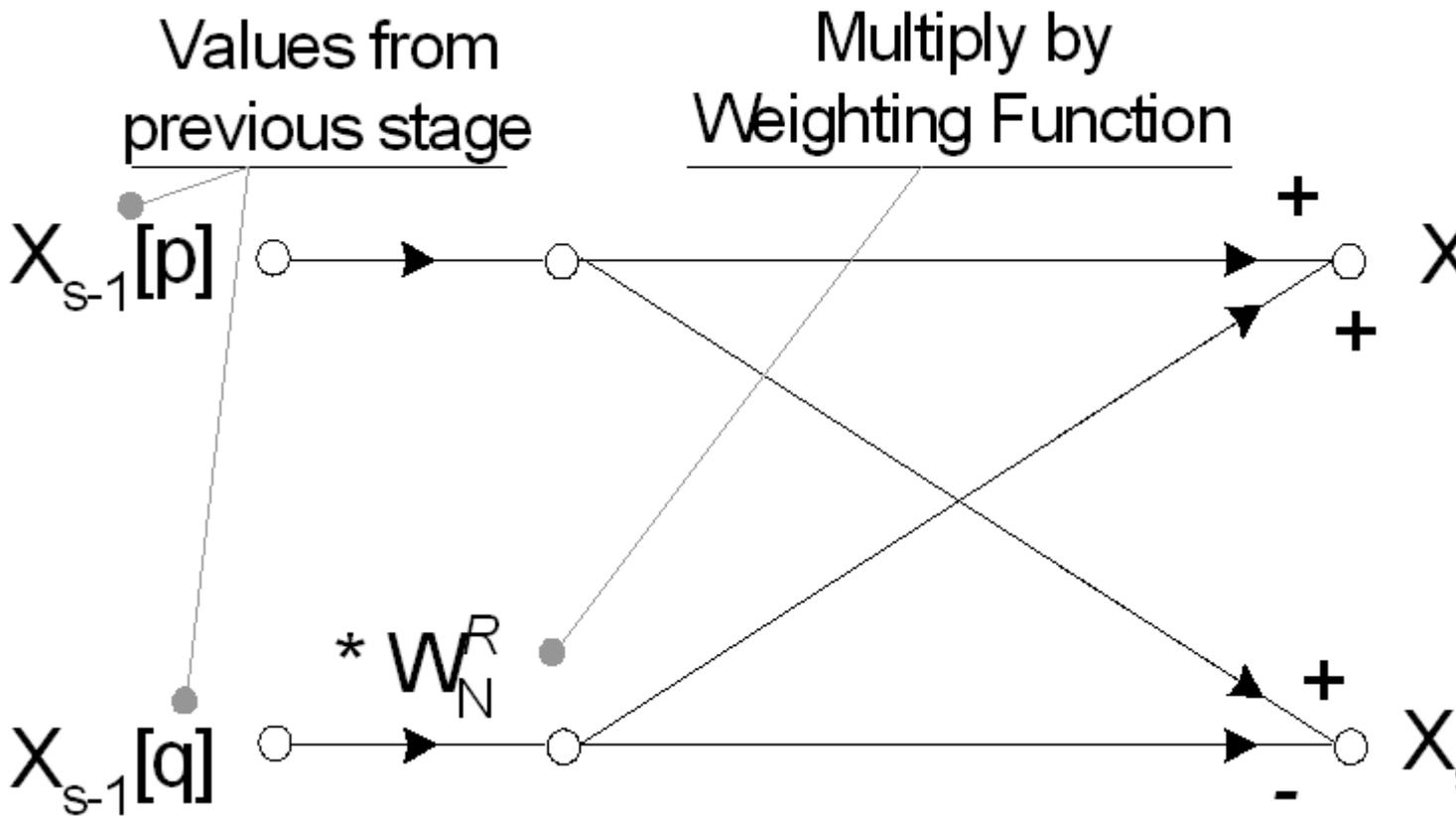
16 signals of
1 point



La descomposición de la señal, o 'decimación en el tiempo' se logra invirtiendo los índices para la matriz de datos del dominio del tiempo. Así, para una señal de dieciséis puntos, la muestra 1 (0001 binario) se intercambia con la muestra 8 (1000), la muestra 2 (0010) se intercambia con 4 (0100) y así sucesivamente. El intercambio de muestras con la técnica de inversión de bits se puede lograr simplemente en el software, pero limita el uso de Radix 2 FFT a señales de longitud

$$N = 2^M.$$

El valor de una señal de 1 punto en el dominio del tiempo es igual a su valor en el dominio de la frecuencia, por lo que esta matriz de puntos del dominio del tiempo único descompuestos no requiere transformación para convertirse en una matriz de puntos del dominio de la frecuencia. Los N puntos individuales; sin embargo, deben ser reconstruidos en un espectro de frecuencia de N puntos. La reconstrucción óptima del espectro de frecuencia completo se realiza mediante cálculos de mariposa. Cada etapa de reconstrucción en el Radix-2 FFT realiza una serie de mariposas de dos puntos, utilizando un conjunto similar de funciones de ponderación exponencial, W_N^R .



La FFT elimina los cálculos redundantes en la Transformada Discreta de Fourier explotando la periodicidad de W_N^R . La reconstrucción espectral se completa en las etapas $\log_2(N)$ de los cálculos de mariposa que dan $X[K]$; Los datos del dominio de frecuencia real e imaginario en forma rectangular. Para convertir a magnitud y fase (coordenadas polares) se requiere encontrar el valor absoluto, $\sqrt{(\text{Re}^2 + \text{Im}^2)}$, y el argumento, $\tan^{-1}(\text{Im} / \text{Re})$.

Exponential Weighting Factor:

W

N:

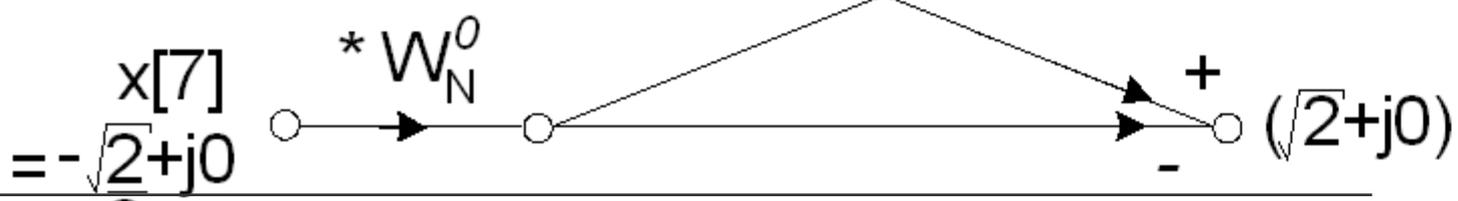
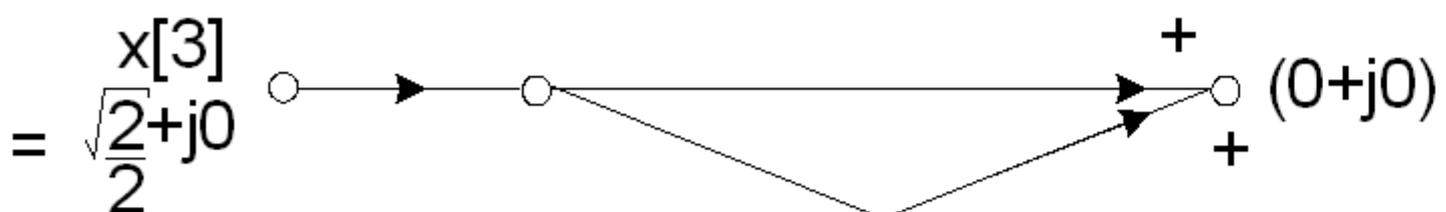
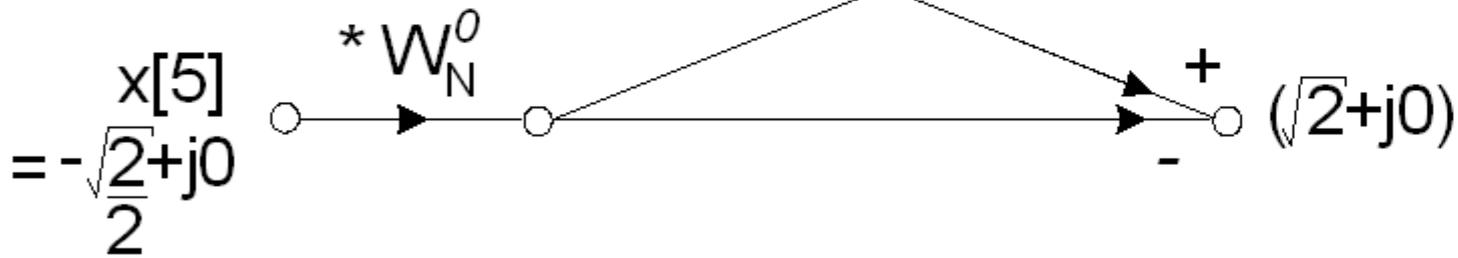
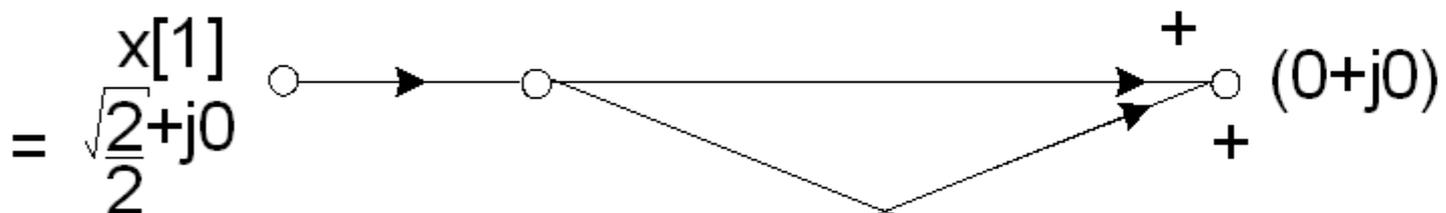
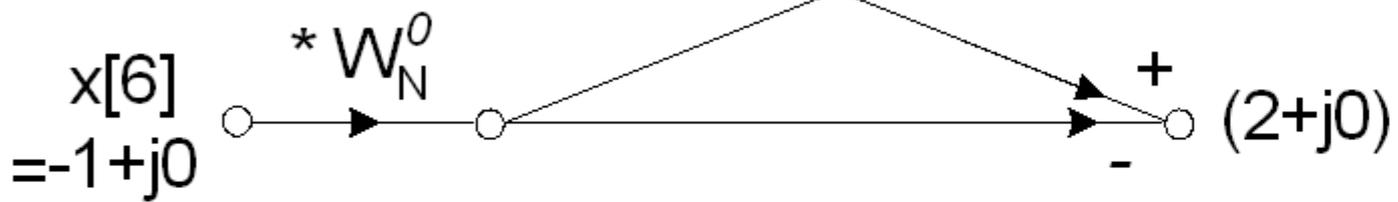
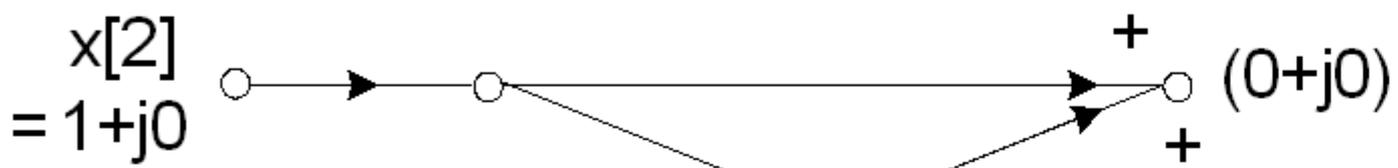
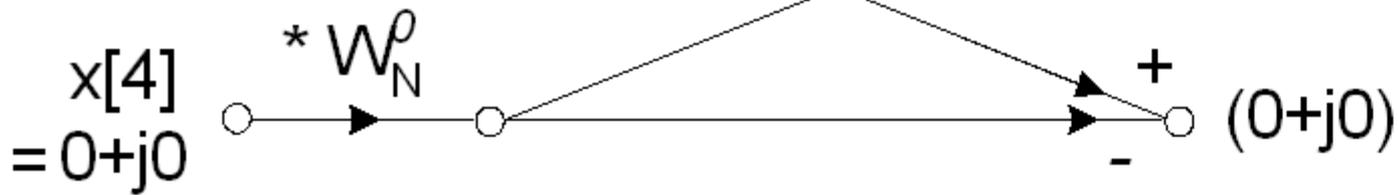
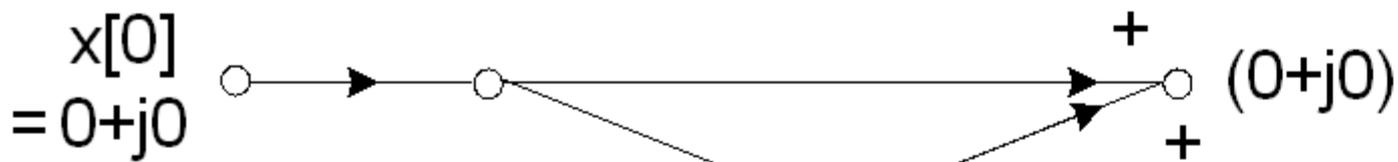
Nu

R:

Cu

sep

El diagrama de flujo completo de la mariposa para un Radix 2 FFT de ocho puntos se muestra a continuación. Tenga en cuenta que las señales de entrada se han reordenado previamente de acuerdo con el procedimiento de reducción de tiempo descrito anteriormente.



2. Realice la FFT directa en los datos del dominio de frecuencia conjugada.
3. Divida cada salida del resultado de esta FFT por N para obtener el valor real del dominio de tiempo.
4. Encuentre el complejo conjugado de la salida invirtiendo el componente imaginario de los datos del dominio de tiempo para todas las instancias de n.

Nota : los datos de dominio de frecuencia y tiempo son variables complejas. Normalmente, el componente imaginario de la señal de dominio de tiempo después de una FFT inversa es cero o se ignora como error de redondeo. El aumento de la precisión de las variables de flotación de 32 bits a doble de 64 bits, o doble de 128 bits de longitud reduce significativamente los errores de redondeo producidos por varias operaciones FFT consecutivas.

Ejemplo de código (C / C ++)

```
#include <math.h>

#define PI      3.1415926535897932384626433832795    // PI for sine/cos calculations
#define TWOPI   6.283185307179586476925286766559    // 2*PI for sine/cos calculations
#define Deg2Rad 0.017453292519943295769236907684886 // Degrees to Radians factor
#define Rad2Deg 57.295779513082320876798154814105   // Radians to Degrees factor
#define log10_2 0.30102999566398119521373889472449 // Log10 of 2
#define log10_2_INV 3.3219280948873623478703194294948 // 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
  }
}
```

Lea Transformada rápida de Fourier en línea:

<https://riptutorial.com/es/algorithm/topic/8683/transformada-rapida-de-fourier>

Capítulo 63: Travesías de árboles binarios

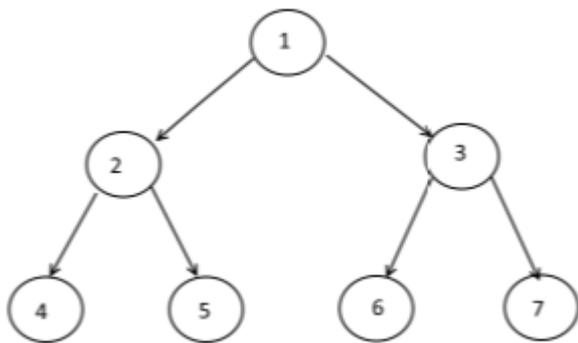
Introducción

Visitar un nodo de un árbol binario en algún orden en particular se llama travesías.

Examples

Pre-order, Inorder y Post Order transversal de un árbol binario

Considere el árbol binario:



El recorrido previo al pedido (raíz) es atravesar el nodo, luego el subárbol izquierdo del nodo y luego el subárbol derecho del nodo.

Por lo tanto, el recorrido previo al pedido del árbol anterior será:

1 2 4 5 3 6 7

En orden (transversal) está atravesando el subárbol izquierdo del nodo, luego el nodo y luego el subárbol derecho del nodo.

Así que el recorrido en orden del árbol anterior será:

4 2 5 1 6 3 7

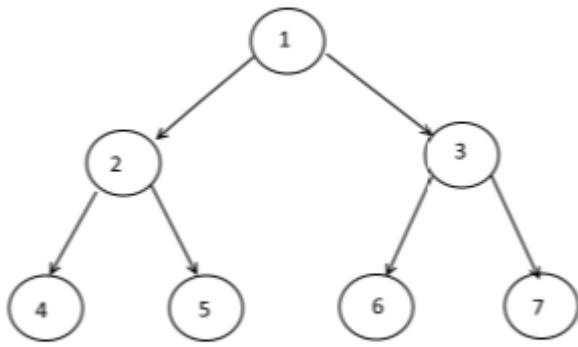
El recorrido posterior al pedido (raíz) recorre el subárbol izquierdo del nodo, luego el subárbol derecho y luego el nodo.

Por lo tanto, el recorrido posterior al pedido del árbol anterior será:

4 5 2 6 7 3 1

Desplazamiento de la orden de nivel - Implementación

Por ejemplo, si el árbol dado es:



El orden de nivel será transversal.

1 2 3 4 5 6 7

Impresión del nivel de datos del nodo por nivel.

Código:

```

#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;  
  
}
```

La estructura de datos de la cola se utiliza para lograr el objetivo anterior.

Lea Travesías de árboles binarios en línea:

<https://riptutorial.com/es/algorithm/topic/8844/travesias-de-arboles-binarios>

Capítulo 64: Triángulo de Pascal

Examples

Pascal's Triagle Información Básica

Uno de los patrones numéricos más interesantes es [el triángulo de Pascal](#) . El nombre "Triángulo de Pascal" lleva el nombre de [Blaise Pascal](#) , un famoso matemático y filósofo francés.

En Matemáticas, el Triángulo de Pascal es una matriz triangular de coeficientes binomiales. Las filas del triángulo de Pascal se enumeran convencionalmente comenzando con la fila $n = 0$ en la parte superior (la fila 0). Las entradas en cada fila están numeradas desde la izquierda comenzando con $k = 0$ y generalmente están escalonadas en relación con los números en las filas adyacentes.

El triángulo se construye de la siguiente manera:

- En la fila superior, hay una entrada única que no es cero 1.
- Cada entrada de cada fila subsiguiente se construye agregando el número arriba y a la izquierda con el número arriba y a la derecha, tratando las entradas en blanco como 0.

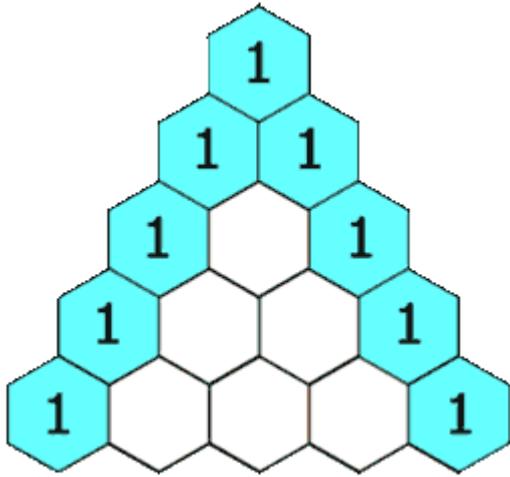
Por ejemplo, el número inicial en la primera (o cualquier otra fila) es 1 (la suma de 0 y 1), mientras que los números 1 y 3 en la tercera fila se suman para producir el número 4 en la cuarta fila.

Ecuación para generar cada entrada en el triángulo de Pascal:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

para cualquier entero no negativo n y cualquier entero k entre 0 y n , ambos inclusive. Esta recurrencia de los coeficientes binomiales se conoce como [regla de Pascal](#) . El triángulo de Pascal tiene generalizaciones dimensionales superiores. La versión tridimensional se llama pirámide de Pascal o tetraedro de Pascal, mientras que las versiones generales se llaman simplices de Pascal.

Ejemplo del Triángulo de Pascal:



Implementación del Triángulo de Pascal en 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;
    }
}
```

Triángulo de Pascal en 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");  
}
```

Salida

```
    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
```

Lea Triángulo de Pascal en línea: <https://riptutorial.com/es/algorithm/topic/7590/triangulo-de-pascal>

Capítulo 65: Vendedor ambulante

Observaciones

El problema del vendedor ambulante es el problema de encontrar el costo mínimo de viajar a través de N vértices exactamente una vez por vértice. Hay un costo $cost[i][j]$ para viajar del vértice i al vértice j .

Hay 2 tipos de algoritmos para resolver este problema: *Algoritmos exactos* y *Algoritmos de aproximación*

Algoritmos exactos

1. Algoritmo de fuerza bruta
2. Algoritmo de programación dinámica

Algoritmos de aproximación

Para ser agregado

Examples

Algoritmo de fuerza bruta

Una ruta a través de cada vértice exactamente una vez es lo mismo que ordenar el vértice de alguna manera. Por lo tanto, para calcular el costo mínimo de viajar a través de cada vértice exactamente una vez, ¿podemos aplicar fuerza bruta a cada uno de los $N!$ Permutaciones de los números del 1 al 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
```

Complejidad del tiempo

Hay $N!$ Las permutaciones a recorrer y el costo de cada ruta se calculan en $O(N)$, por lo tanto, este algoritmo toma tiempo $O(N * N!)$ para dar la respuesta exacta.

Algoritmo de programación dinámica

Observe que si consideramos el camino (en orden):

```
(1, 2, 3, 4, 6, 0, 5, 7)
```

y el camino

```
(1, 2, 3, 5, 0, 6, 7, 4)
```

El costo de ir del vértice 1 al vértice 2 al vértice 3 sigue siendo el mismo, entonces ¿por qué debe recalcularse? Este resultado se puede guardar para su uso posterior.

Sea $dp[bitmask][vertex]$ el costo mínimo de viajar a través de todos los vértices cuyo bit correspondiente en la $bitmask$ de $bitmask$ se establece en 1 termina en el $vertex$. Por ejemplo:

```
dp[12][2]
12    =   1 1 0 0
        ^ ^
vertices: 3 2 1 0
```

Dado que 12 representa 1100 en binario, $dp[12][2]$ representa ir a través de los vértices 2 y 3 en la gráfica con la ruta que termina en el vértice 2.

Así podemos tener el siguiente algoritmo (implementación de 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)
```

Esta línea puede ser un poco confusa, así que avancemos lentamente:

```
cost = min(cost, TSP(bitmask | (1 << i), i) + cost[pos][i]);
```

Aquí, la `bitmask | (1 << i)` establece el bit `i` de la `bitmask` de `bitmask` en 1, lo que representa que se ha visitado el vértice `ith`. La `i` después de la coma representa la nueva `pos` en esa llamada de función, que representa el nuevo "último" vértice. `cost[pos][i]` es sumar el costo de viajar del vértice `pos` al vértice `i`.

Por lo tanto, esta línea es para actualizar el valor del `cost` al valor mínimo posible de viajar a cualquier otro vértice que aún no se haya visitado.

Complejidad del tiempo

La función `TSP(bitmask, pos)` tiene valores de 2^N para la `bitmask` de `bitmask` y valores de N para `pos`. Cada función tarda $O(N)$ en ejecutarse (el bucle `for`). Por lo tanto, esta implementación toma tiempo $O(N^2 * 2^N)$ para dar la respuesta exacta.

Lea **Vendedor ambulante en línea**: <https://riptutorial.com/es/algorithm/topic/6631/vendedor-ambulante>

Creditos

S. No	Capítulos	Contributors
1	Empezando con el algoritmo	Abdul Karim , Austin Conlon , C L K Kissane , Community , EsmaeelE , Filip Allberg , Hesham Attia , Jonathan Landrum , msohng , Sayakiss , user2314737
2	A * Algoritmo de búsqueda de rutas	TajyMany
3	A * Pathfinding	kiner_shah , Minhas Kamal , mnoronha , Roberto Fernandez , TajyMany
4	Algo: - Imprimir matriz $m \times n$ en forma de cuadrado	Creative John
5	Algoritmo de Bellman-Ford	Bakhtiar Hasan , Sumeet Singh , user2314737 , Yerken
6	Algoritmo de Floyd-Warshall	Bakhtiar Hasan , Sayakiss
7	Algoritmo de Knuth Morris Pratt (KMP)	Vishwas
8	Algoritmo de linea	Dipesh Poudel , Martin Frank
9	Algoritmo de partición entero	Keyur Ramoliya
10	Algoritmo de Prim	Bakhtiar Hasan , Tejus Prasad
11	Algoritmo de subarray máximo	Keyur Ramoliya , mnoronha
12	Algoritmo de suma de ruta máxima	Keyur Ramoliya , mnoronha
13	Algoritmo de ventana deslizante	Keyur Ramoliya
14	Algoritmo delimitado por tiempo polinómico para la cobertura mínima de	Alber Tadrous

	vértices	
15	Algoritmo Numérico Catalán	Keyur Ramoliya , mnoronha
16	Algoritmos codiciosos	Bakhtiar Hasan , Cameron , Community , ghilesZ , M S Hossain , theJollySin , xenteros
17	Algoritmos en línea	Andrii Artamonov , goeddek
18	Algoritmos multihilo	Julien Rousé
19	Aplicaciones de la técnica codiciosa.	EsmaeelE , goeddek , Tejus Prasad , user2314737
20	Aplicaciones de Programación Dinámica.	Chris , user2314737
21	Arboles	Isha Agarwal , Malcolm McLean , mnoronha , VermillionAzure , yd1
22	Árboles binarios de búsqueda	a13ph , EsmaeelE , greatwolf , Isha Agarwal , Ishit Mehta , Mehedi Hasan , nbro , RamenChef , Rashik Hasnat , Tejus Prasad
23	buscando	Anagh Hegde , Benson Lin , brijs , Community , EsmaeelE , Iwan , Khaled.K , Malcolm McLean , Miljen Mikic , msohng , RamenChef , ShreePool , Timothy G. , umop apisdn , xenteros
24	Búsqueda de amplitud	Bakhtiar Hasan , mnoronha , Sumeet Singh , Zubayet Zaman Zico
25	Búsqueda de subcadena	AnukuL , Bakhtiar Hasan , mnoronha , Rashik Hasnat
26	Clasificación	Ahmad Faiyaz , Carlton , Filip Allberg , Frank , ganesshkumar , IVlad , Iwan , Kedar Mhaswade , Miljen Mikic , mok , Patrick87 , RamenChef , Rob Fagen , Set
27	Combinar clasificación	EsmaeelE , Iwan , Juxhin Metaj , Keyur Ramoliya , Luv Agarwal , mnoronha , Santiago Gil , SHARMA
28	Complejidad de algoritmos	A. Raza , Daniel Nugent , Didgeridoo , EsmaeelE , fgb , Juxhin Metaj , Miljen Mikic , Nick Larsen , Peter K , Sayakiss , Tejus Prasad , user23013 , user2314737 , VermillionAzure , xenteros , Yair Twito
29	Compruebe que dos cadenas son anagramas	Creative John

30	Compruebe si un árbol es BST o no	Isha Agarwal , Janaky Murthy
31	Editar distancia del algoritmo dinámico	Vishwas
32	El algoritmo de Dijkstra	Bakhtiar Hasan , Tejus Prasad
33	El algoritmo de Kruskal	IvIad , Shubham , Yerken
34	El ancestro común más bajo de un árbol binario	Isha Agarwal
35	El problema más corto de la supersecuencia	Keyur Ramoliya
36	Exposición de matrices	Bakhtiar Hasan , mnoronha
37	Funciones hash	afeldspar , Didgeridoo , mnoronha
38	Grafico	Ahmed Mazher , Bakhtiar Hasan , EsmaeelE , Filip Allberg , hurricane , JJTO , John Odom , Idog , Sayakiss , Tejus Prasad , user23013 , VermillionAzure
39	Gráficos de travesías	Dian Bakti
40	Heap Sort	Keyur Ramoliya , mnoronha
41	La subsecuencia cada vez mayor	Keyur Ramoliya , mnoronha
42	La subsecuencia común más larga	Bakhtiar Hasan , Keyur Ramoliya
43	Notación Big-O	Community , EsmaeelE , mnoronha , msohng , Nick the coder , Samuel Peter , user2314737 , WitVault
44	Orden de conteo	Bakhtiar Hasan , Keyur Ramoliya , mnoronha
45	Ordenación rápida	Bakhtiar Hasan , Keyur Ramoliya , Malav , mnoronha , optimistanoop
46	Ordenamiento de burbuja	Anagh Hegde , Deepak , EsmaeelE , Ijaz Khan , Keyur Ramoliya , mnoronha , optimistanoop , samgak , xenteros , YoungHobbit

47	Primera búsqueda de profundidad	Bakhtiar Hasan
48	Problema de mochila	Bakhtiar Hasan , dtt , Keyur Ramoliya , mnoronha , Tejus Prasad , user2314737
49	Programación dinámica	Bakhtiar Hasan , Benson Lin , kraskevich , Muyide Ibukun , nbro , RamenChef , Razik , Sayakiss , Vishwas
50	Pseudocódigo	Community
51	Radix Sort	Keyur Ramoliya , mnoronha , Zopesconk
52	Resolución de ecuaciones	Minhas Kamal
53	Selección de selección	Keyur Ramoliya , lambda , mnoronha , Teodor Kurtev , user2314737
54	Shell Sort	Keyur Ramoliya , mnoronha
55	Time Warping dinámico	Bakhtiar Hasan , mnoronha , Zubayet Zaman Zico
56	Tipo de casillero	Keyur Ramoliya , mnoronha
57	Tipo de ciclo	Keyur Ramoliya , mnoronha
58	Tipo de cubo	Keyur Ramoliya , mnoronha
59	Tipo de inserción	Bakhtiar Hasan , invisal , Keyur Ramoliya , Lymphatus , mnoronha , RamenChef
60	Tipo de panqueque	Keyur Ramoliya , mnoronha
61	Tipo impar-par	Keyur Ramoliya
62	Transformada rápida de Fourier	Dr. ABT , EsmaeelE
63	Travesías de árboles binarios	Isha Agarwal
64	Triángulo de Pascal	EsmaeelE , Keyur Ramoliya
65	Vendedor ambulante	Benson Lin