EBook Gratis

APRENDIZAJE

dynamic-programming

Free unaffiliated eBook created from **Stack Overflow contributors.**

#dynamic-programming

Tabla de contenido

Acerca de 1

Capítulo 1: Empezando con la programación dinámica. 2

Observaciones 2

Examples 2

Introducción a la programación dinámica 2

Entendiendo el estado en la programación dinámica 5

Construyendo una solución de DP 6

Capítulo 2: Corte de varilla 12

Examples 12

Cortando la caña para obtener el máximo beneficio. 12

Capítulo 3: Matriz de multiplicación de la cadena 15

Examples 15

Solucion recursiva 15

Capítulo 4: Moneda que cambia el problema 18

Examples 18

Número de formas de obtener el total 18

Cantidad mínima de monedas para obtener el total 20

Capítulo 5: Problema de mochila 23

Observaciones 23

Examples 23

0-1 Problema de mochila 23

Capítulo 6: Resolviendo problemas de gráficas usando programación dinámica 28

Examples 28

Algoritmo de Floyd-Warshall 28

Cubierta de vértice mínimo 30

Capítulo 7: Selección de actividad ponderada 35

Examples 35

Algoritmo de programación de trabajo ponderado 35

Capítulo 8: Subsecuencias relacionadas con los algoritmos 40

Examples 40

La subsecuencia común más larga 40
La subsecuencia cada vez mayor 44
Secuencia palindrómica más larga 48
Capítulo 9: Time Warping dinámico 52
Examples 52
Introducción a la distorsión de tiempo dinámico 52
Creditos 57

Acerca de

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

It is an unofficial and free dynamic-programming 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 dynamic-programming.

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 la programación dinámica.

Observaciones

Esta sección proporciona una descripción general de qué es la programación dinámica y por qué un desarrollador puede querer usarla.

También debe mencionar cualquier tema grande dentro de la programación dinámica, y vincular a los temas relacionados. Dado que la Documentación para la programación dinámica es nueva, es posible que deba crear versiones iniciales de esos temas relacionados.

Examples

Introducción a la programación dinámica

La programación dinámica resuelve problemas al combinar las soluciones a los subproblemas. Puede ser análogo al método de dividir y conquistar, donde el problema se divide en subproblemas separados, los subproblemas se resuelven recursivamente y luego se combinan para encontrar la solución del problema original. En contraste, la programación dinámica se aplica cuando los subproblemas se superponen, es decir, cuando los subproblemas comparten subproblemas. En este contexto, un algoritmo de dividir y conquistar hace más trabajo del necesario, resolviendo repetidamente los subproyectos comunes. Un algoritmo de programación dinámica resuelve cada sub-problema solo una vez y luego guarda su respuesta en una tabla, evitando así el trabajo de volver a calcular la respuesta cada vez que resuelve cada sub-problema.

Veamos un ejemplo. El matemático italiano Leonardo Pisano Bigollo , a quien conocemos comúnmente como Fibonacci, descubrió una serie de números considerando el crecimiento idealizado de la población de conejos . La serie es:

```
1, 1, 2, 3, 5, 8, 13, 21, ......
```

Podemos notar que cada número después de los dos primeros es la suma de los dos números anteriores. Ahora, formulemos una función **F (n)** que nos devuelva el número n de Fibonacci, lo que significa que

```
F(n) = nth Fibonacci Number
```

Hasta ahora, hemos sabido que,

```
F(1) = 1
F(2) = 1
F(3) = F(2) + F(1) = 2
```

```
F(4) = F(3) + F(2) = 3
```

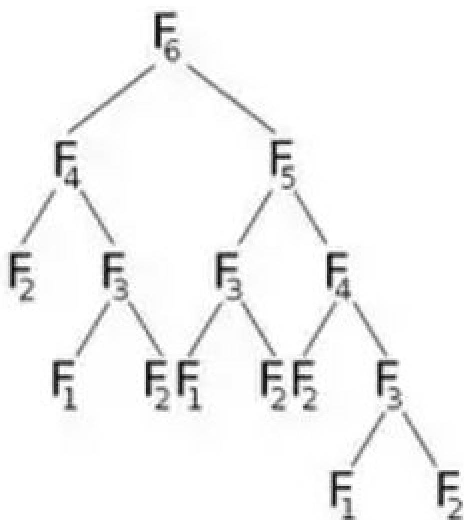
Podemos generalizarlo por:

```
F(1) = 1
F(2) = 1
F(n) = F(n-1) + F(n-2)
```

Ahora, si queremos escribirlo como una función recursiva, tenemos `F(1)` y `F(2)` como nuestro caso base. Así que nuestra función de Fibonacci sería:

```
Procedure F(n):                //A function to return nth Fibonacci Number
if n is equal to 1
    Return 1
else if n is equal to 2
    Return 1
end if
Return F(n-1) + F(n-2)
```

Ahora, si llamamos a `F(6)` , llamará a `F(5)` y `F(4)` , que llamará a algunos más. Vamos a representar esto gráficamente:



En la imagen, podemos ver que `F(6)` llamará a `F(5)` y `F(4)` . Ahora `F(5)` llamará a `F(4)` y `F(3)` . Después de calcular `F(5)` , seguramente podemos decir que todas las funciones que fueron llamadas por `F(5)` ya se han calculado. Eso significa que ya hemos calculado `F(4)` . Pero nuevamente estamos calculando `F(4)` como indica el hijo derecho de `F(6)` . ¿Realmente necesitamos recalcular? Lo que podemos hacer es que, una vez que hayamos calculado el valor de `F(4)` , lo almacenaremos en una matriz llamada **dp** y la reutilizaremos cuando sea necesario. Inicializaremos nuestra matriz **dp** con **-1** (o cualquier valor que no aparezca en nuestro cálculo). Luego llamaremos a **F (6)** donde se verá nuestra **F (n)** modificada:

```
Procedure F(n):
if n is equal to 1
    Return 1
else if n is equal to 2
    Return 1
```

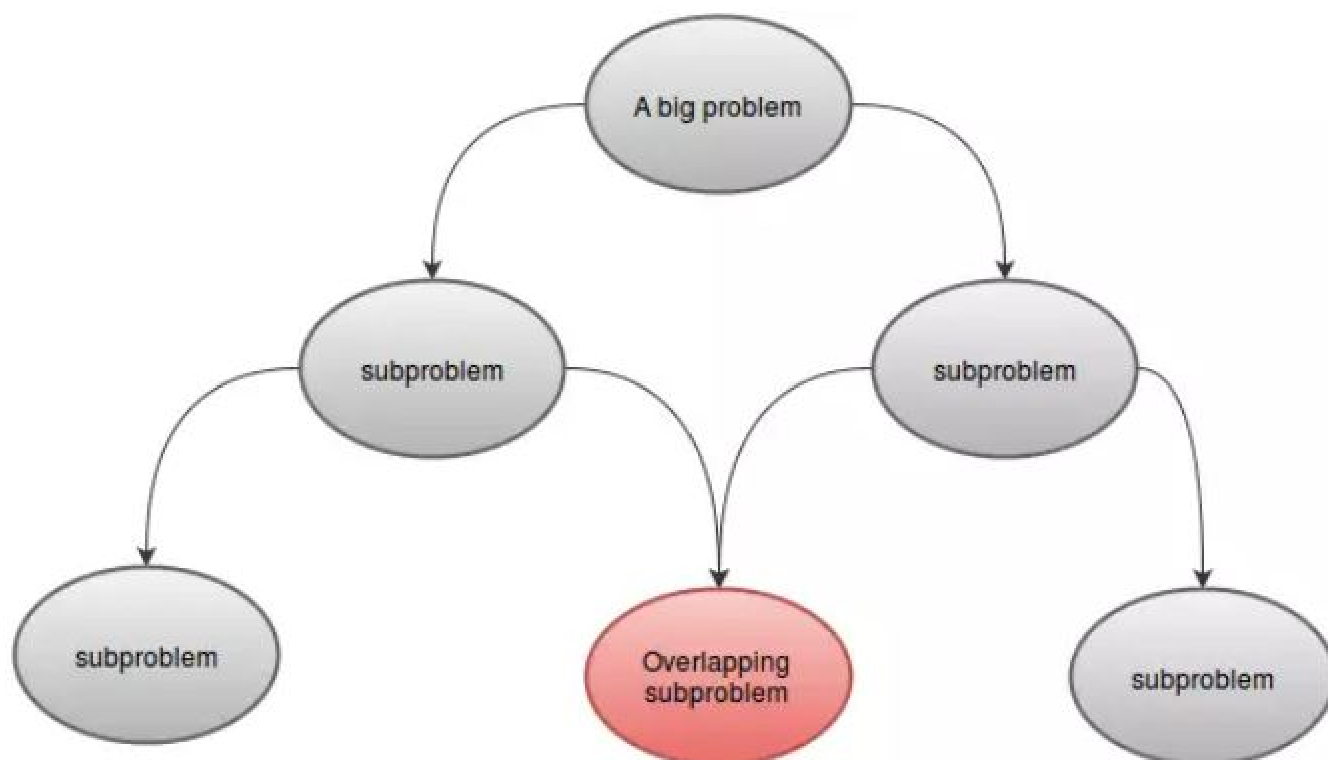
```
else if dp[n] is not equal to -1            //That means we have already calculated dp[n]
    Return dp[n]
else
    dp[n] = F(n-1) + F(n-2)
    Return dp[n]
end if
```

Hemos hecho la misma tarea que antes, pero con una optimización simple. Es decir, hemos utilizado la técnica de memoización . Al principio, todos los valores de **dp** array serán **-1** . Cuando se llama `F(4)` , verificamos si está vacío o no. Si almacena **-1** , calcularemos su valor y lo almacenaremos en **dp [4]** . Si almacena cualquier cosa menos **-1** , eso significa que ya hemos calculado su valor. Así que simplemente devolveremos el valor.

Esta simple optimización utilizando memoización se llama **Programación Dinámica** .

Un problema se puede resolver utilizando la Programación Dinámica si tiene algunas características. Estos son:

- **Subproblemas:**
  Un problema de DP se puede dividir en uno o más subproblemas. Por ejemplo: `F(4)` se puede dividir en subproblemas más pequeños `F(3)` y `F(2)` . Como los subproblemas son similares a nuestro problema principal, estos pueden resolverse utilizando la misma técnica.
- **Subproblemas superpuestos:**
  Un problema de DP debe tener subproblemas superpuestos. Eso significa que debe haber una parte común para la cual la misma función se llama más de una vez. Por ejemplo: `F(5)` y `F(6)` tiene `F(3)` y `F(4)` en común. Esta es la razón por la que almacenamos los valores en nuestra matriz.



- **Subestructura óptima:**

Digamos que se le pide que minimice la función `g(x)` . Usted sabe que el valor de `g(x)` depende de `g(y)` y `g(z)` . Ahora, si podemos minimizar `g(x)` minimizando tanto `g(y)` como `g(z)` , solo entonces podemos decir que el problema tiene una subestructura óptima. Si se minimiza `g(x)` minimizando solo `g(y)` y si minimiza o maximiza `g(z)` no tiene ningún efecto sobre `g(x)` , entonces este problema no tiene una subestructura óptima. En palabras simples, si la solución óptima de un problema puede encontrarse a partir de la solución óptima de su subproblema, entonces podemos decir que el problema tiene una propiedad de subestructura óptima.

Entendiendo el estado en la programación dinámica

Vamos a discutir con un ejemplo. De **n** artículos, ¿de cuántas maneras puede elegir **r** artículos? Usted sabe que se denota por ^nC_r . Ahora piensa en un solo artículo.

- Si no selecciona el elemento, después de eso tendrá que tomar **r** elementos de los elementos **n-1** restantes, lo cual es dado por $^{n-1}C_r$.
- Si selecciona el elemento, después de eso tendrá que tomar los elementos **r-1 de los** elementos **n-1** restantes, lo cual es dado por $^{n-1}C_{r-1}$.

La suma de estos dos nos da el número total de formas. Es decir:
$^nC_r = ^{n-1}C_r + ^{n-1}C_{r-1}$

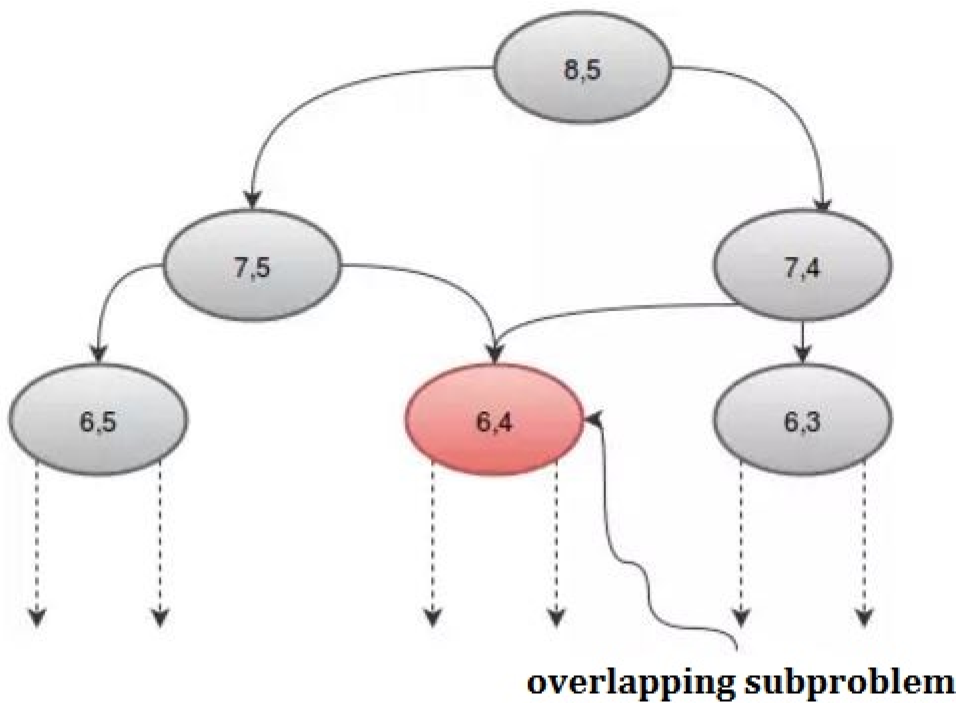
Si pensamos `nCr(n,r)` como una función que toma `n` y `r` como parámetro y determina ^nC_r , podemos escribir la relación mencionada anteriormente como:

```
nCr(n,r) = nCr(n-1,r) + nCr(n-1,r-1)
```

Esta es una relación recursiva. Para terminarlo, necesitamos determinar los casos base. Lo sabemos, $^nC_1 = n$ y $^nC_n = 1$. Usando estos dos como casos base, nuestro algoritmo para determinar ^nC_r estarán:

```
Procedure nCr(n, r):
if r is equal to 1
    Return n
else if n is equal to r
    Return 1
end if
Return nCr(n-1,r) + nCr(n-1,r-1)
```

Si observamos el procedimiento gráficamente, podemos ver que algunas recursiones se llaman más de una vez. Por ejemplo: si tomamos **n** = **8** y **r** = **5** , obtenemos:



Podemos evitar esta llamada repetida usando una matriz, **dp** . Como hay **2** parámetros, necesitaremos una matriz 2D. Inicializaremos la matriz **dp** con **-1** , donde **-1** denota que el valor aún no se ha calculado. Nuestro procedimiento será:

```
Procedure nCr(n,r):
if r is equal to 1
    Return n
else if n is equal to r
    Return 1
else if dp[n][r] is not equal to -1        //The value has been calculated
    Return dp[n][r]
end if
dp[n][r] := nCr(n-1,r) + nCr(n-1,r-1)
Return dp[n][r]
```

Para determinar ${}^nC_r$, necesitábamos dos parámetros **n** y **r** . Estos parámetros se denominan *estado* . Simplemente podemos deducir que el número de estados determina el número de dimensión de la matriz **dp** . El tamaño de la matriz cambiará de acuerdo con nuestra necesidad. Nuestros algoritmos de programación dinámica mantendrán el siguiente patrón general:

```
Procedure DP-Function(state_1, state_2, ...., state_n)
Return if reached any base case
Check array and Return if the value is already calculated.
Calculate the value recursively for this state
Save the value in the table and Return
```

Determinar el *estado* es una de las partes más cruciales de la programación dinámica. Consiste en la cantidad de parámetros que definen nuestro problema y optimizando su cálculo, podemos optimizar todo el problema.

Construyendo una solución de DP

No importa cuántos problemas resuelva utilizando la programación dinámica (DP), todavía puede sorprenderlo. Pero como todo lo demás en la vida, la práctica te hace mejor. Teniendo esto en cuenta, veremos el proceso de construcción de una solución para problemas de DP. Otros ejemplos sobre este tema lo ayudarán a comprender qué es DP y cómo funciona. En este ejemplo, trataremos de comprender cómo crear una solución de DP desde cero.

Iterativo vs recursivo:
Existen dos técnicas de construcción de la solución DP. Son:

- Iterativa (usando para ciclos)
- Recursivo (usando recursión)

Por ejemplo, el algoritmo para calcular la longitud de la Subsecuencia Común Más Larga de dos cadenas **s1** y **s2** se vería así:

```
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]
```

Esta es una solución iterativa. Hay algunas razones por las que se codifica de esta manera:

- La iteración es más rápida que la recursión.
- Determinar el tiempo y la complejidad del espacio es más fácil.
- El código fuente es corto y limpio

Mirando el código fuente, puede comprender fácilmente cómo y por qué funciona, pero es difícil entender cómo encontrar esta solución. Sin embargo, los dos enfoques mencionados anteriormente se traducen en dos pseudocódigos diferentes. Uno usa la iteración (Bottom-Up) y el otro utiliza el enfoque de recursión (Top-Down). La última también se conoce como técnica de memoización. Sin embargo, las dos soluciones son más o menos equivalentes y una puede transformarse fácilmente en otra. Para este ejemplo, mostraremos cómo encontrar una solución recursiva para un problema.

Problema de ejemplo:
Digamos que tiene **N** (**1, 2, 3, ..., N**) vinos colocados uno al lado del otro en un estante. El precio del **i-ésimo** vino es **p [i].** El precio de los vinos aumenta cada año. Supongamos que es el año **1** , en el año **y** el precio del vino **i** th será: año * precio del vino = **y** * **p [i]** . Quiere vender los vinos

que tiene, pero tiene que vender exactamente un vino por año, a partir de este año. Una vez más, en cada año, se le permite vender solo el vino más a la izquierda o más a la derecha y no puede reorganizar los vinos, lo que significa que deben permanecer en el mismo orden que al principio.

Por ejemplo: digamos que tiene **4** vinos en la estantería, y sus precios son (de izquierda a derecha):

```
+---+---+---+---+
| 1 | 4 | 2 | 3 |
+---+---+---+---+
```

La solución óptima sería vender los vinos en el orden **1** -> **4** -> **3** -> **2** , lo que nos dará un beneficio total de: $1*1+3*2+2*3+4*4=29$

Enfoque codicioso:
Después de una lluvia de ideas durante un tiempo, puede encontrar la solución para vender el vino caro lo más tarde posible. Entonces tu estrategia *codiciosa* será:

```
Every year, sell the cheaper of the two (leftmost and rightmost) available wines.
```

Aunque la estrategia no menciona qué hacer cuando los dos vinos cuestan lo mismo, la estrategia se siente un poco correcta. Pero desafortunadamente, no lo es. Si los precios de los vinos son:

```
+---+---+---+---+---+
| 2 | 3 | 5 | 1 | 4 |
+---+---+---+---+---+
```

La estrategia codiciosa los vendería en el orden **1** -> **2** -> **5** -> **4** -> **3** para un beneficio total de: $2*1+3*2+4*3+1*4+5*5=49$

Pero podemos hacerlo mejor si vendemos los vinos en el orden **1** -> **5** -> **4** -> **2** -> **3** para un beneficio total de: $2*1+4*2+1*3+3*4+5*5=50$

Este ejemplo debería convencerlo de que el problema no es tan fácil como parece a primera vista. Pero se puede resolver utilizando la Programación Dinámica.

Retroceso:
Crear una solución de memoria para un problema es útil para encontrar una solución de retroceso. La solución Backtrack evalúa todas las respuestas válidas para el problema y elige la mejor. Para la mayoría de los problemas, es más fácil encontrar tal solución. Puede haber tres estrategias a seguir para acercarse a una solución de retroceso:

1. debe ser una función que calcula la respuesta usando la recursión.
2. debe devolver la respuesta con declaración de *retorno* .
3. todas las variables no locales deben usarse como de solo lectura, es decir, la función puede modificar solo las variables locales y sus argumentos.

Para nuestro problema de ejemplo, trataremos de vender el vino más a la izquierda o más a la derecha y calcularemos recursivamente la respuesta y devolveremos la mejor. La solución de

retroceso se vería así:

```
// year represents the current year
// [begin, end] represents the interval of the unsold wines on the shelf
Procedure profitDetermination(year, begin, end):
if begin > end                  //there are no more wines left on the shelf
    Return 0
Return max(profitDetermination(year+1, begin+1, end) + year * p[begin], //selling the leftmost
item
           profitDetermination(year+1, begin, end+1) + year * p[end])   //selling the
rightmost item
```

Si llamamos al procedimiento utilizando la determinación de `profitDetermination(1, 0, n-1)` , donde **n** es el número total de vinos, devolverá la respuesta.

Esta solución simplemente prueba todas las posibles combinaciones válidas de venta de vinos. Si al principio hay **n** vinos, comprobará 2^n posibilidades Aunque ahora obtenemos la respuesta correcta, la complejidad del tiempo del algoritmo crece exponencialmente.

La función de retroceso correctamente escrita siempre debe representar una respuesta a una pregunta bien formulada. En nuestro ejemplo, el procedimiento `profitDetermination` representa una respuesta a la pregunta: *¿Cuál es el mejor beneficio que podemos obtener al vender los vinos con precios almacenados en la matriz p, cuando el año actual es el año y el intervalo de los vinos no vendidos se extiende hasta [comenzar , fin], inclusive?* Siempre debe intentar crear una pregunta de este tipo para su función de marcha atrás para ver si lo entendió bien y entender exactamente lo que hace.

Estado determinante:
Estado es el número de parámetros utilizados en la solución de DP. En este paso, debemos pensar en cuáles de los argumentos que pasa a la función son redundantes, es decir, podemos construirlos a partir de los otros argumentos o no los necesitamos en absoluto. Si existen tales argumentos, no necesitamos pasarlos a la función, los calcularemos dentro de la función.

En la función de ejemplo `profitDetermination` muestra arriba, el `year` del argumento es redundante. Es equivalente a la cantidad de vinos que ya hemos vendido más uno. Se puede determinar utilizando el número total de vinos desde el principio menos el número de vinos que no hemos vendido más uno. Si almacenamos el número total de vinos **n** como una variable global, podemos reescribir la función como:

```
Procedure profitDetermination(begin, end):
if begin > end
    Return 0
year := n - (end-begin+1) + 1        //as described above
Return max(profitDetermination(begin+1, end) + year * p[begin],
           profitDetermination(begin, end+1) + year * p[end])
```

También debemos pensar en el rango de valores posibles de los parámetros que se pueden obtener de una entrada válida. En nuestro ejemplo, cada uno de los parámetros `begin` y `end` puede contener valores de **0** a **n-1** . En una entrada válida, también esperaremos `begin <= end + 1` . Puede haber $O(n^2)$ diferentes argumentos con los que se puede llamar a nuestra función.

Memorización
Ya casi hemos terminado. Transformar la solución de retroceso con complejidad de tiempo. $O(2^n)$ en solución de memoria con complejidad de tiempo $O(n^2)$, usaremos un pequeño truco que no requiere mucho esfuerzo.

Como se señaló anteriormente, sólo hay $O(n^2)$ Diferentes parámetros nuestra función puede ser llamada con. En otras palabras, sólo hay $O(n^2)$ Diferentes cosas que realmente podemos calcular. Entonces donde hace $O(2^n)$ El tiempo viene de la complejidad y ¿qué se computa?

La respuesta es: la complejidad del tiempo exponencial proviene de la repetición recursiva y debido a eso, calcula los mismos valores una y otra vez. Si ejecuta el código mencionado anteriormente para un conjunto arbitrario de **n** = **20** vinos y calcula cuántas veces se llamó la función para los argumentos **begin** = **10** y **end** = **10** , obtendrá un número **92378** . Es una enorme pérdida de tiempo calcular la misma respuesta muchas veces. Lo que podríamos hacer para mejorar esto es almacenar los valores una vez que los hayamos computado y cada vez que la función solicite un valor ya calculado, no necesitamos ejecutar la recursión completa nuevamente. Tendremos una matriz **dp [N] [N]** , la inicializaremos con **-1** (o cualquier valor que no aparezca en nuestro cálculo), donde **-1** denota que el valor aún no se ha calculado. El tamaño de la matriz está determinado por el valor máximo posible de **inicio** y **finalización,** ya que almacenaremos los valores correspondientes de ciertos valores de **inicio** y **finalización** en nuestra matriz. Nuestro procedimiento se vería como:

```
Procedure profitDetermination(begin, end):
if begin > end
    Return 0
if dp[begin][end] is not equal to -1                //Already calculated
    Return dp[begin][end]
year := n - (end-begin+1) + 1
dp[begin][end] := max(profitDetermination(year+1, begin+1, end) + year * p[begin],
           profitDetermination(year+1, begin, end+1) + year * p[end])
Return dp[begin][end]
```

Esta es nuestra solución de DP requerida. Con nuestro truco muy simple, la solución se ejecuta. $O(n^2)$ tiempo, porque hay $O(n^2)$ diferentes parámetros con los que se puede llamar a nuestra función y para cada uno de ellos, la función se ejecuta solo una vez con $O(1)$ complejidad.

Veraniego:
Si puede identificar un problema que se puede resolver utilizando DP, siga los siguientes pasos para construir una solución DP:

- Cree una función de retroceso para proporcionar la respuesta correcta.
- Eliminar los argumentos redundantes.
- Estimar y minimizar el rango máximo de valores posibles de parámetros de función.
- Intente optimizar la complejidad del tiempo de una llamada de función.
- Almacena los valores para que no tengas que calcularlos dos veces.

La complejidad de una solución de DP es: **rango de valores posibles a los que se puede**

llamar la función con * **complejidad de tiempo de cada llamada** .

Lea Empezando con la programación dinámica. en línea: https://riptutorial.com/es/dynamic-programming/topic/7946/empezando-con-la-programacion-dinamica-

Capítulo 2: Corte de varilla

Examples

Cortando la caña para obtener el máximo beneficio.

Dada una vara de longitud **n** pulgadas y una serie de longitud **m** de precios que contiene precios de todas las piezas de tamaño más pequeño que n. Tenemos que encontrar el valor máximo que se puede obtener cortando la barra y vendiendo las piezas. Por ejemplo, si la longitud de la varilla es **8** y los valores de las diferentes piezas se indican a continuación, el valor máximo obtenible es **22** .

```
       +---+---+---+---+---+---+---+---+
(price)| 1 | 5 | 8 | 9 | 10| 17| 17| 20|
       +---+---+---+---+---+---+---+---+
```

Usaremos una matriz 2D **dp [m] [n + 1]** donde n es la longitud de la barra y m es la longitud de la matriz de precios. Para nuestro ejemplo, necesitaremos **dp [8] [9]** . Aquí **dp [i] [j]** indicará el precio máximo vendiendo la vara de longitud j. Podemos tener el valor máximo de longitud j en su totalidad o podríamos haber roto la longitud para maximizar el beneficio.

Al principio, para la columna 0, no aportará nada, por lo que marcará todos los valores como 0. Por lo tanto, todos los valores de la columna 0 serán 0. Para **dp [0] [1]** , ¿cuál es el valor máximo que podemos obtener? vendiendo varilla de longitud 1. Será 1. De manera similar para varilla de longitud 2 dp [0] [2] podemos tener 2 (1 + 1). Esto continúa hasta **dp [0] [8]** . Por lo tanto, después de la primera iteración nuestra matriz dp [] se verá como

```
      +---+---+---+---+---+---+---+---+---+
(price)| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
       +---+---+---+---+---+---+---+---+---+
 (1) 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9 |
       +---+---+---+---+---+---+---+---+---+
 (5) 2 | 0 |   |   |   |   |   |   |   |   |
       +---+---+---+---+---+---+---+---+---+
 (8) 3 | 0 |   |   |   |   |   |   |   |   |
       +---+---+---+---+---+---+---+---+---+
 (9) 4 | 0 |   |   |   |   |   |   |   |   |
       +---+---+---+---+---+---+---+---+---+
(10) 5 | 0 |   |   |   |   |   |   |   |   |
       +---+---+---+---+---+---+---+---+---+
(17) 6 | 0 |   |   |   |   |   |   |   |   |
       +---+---+---+---+---+---+---+---+---+
(17) 7 | 0 |   |   |   |   |   |   |   |   |
       +---+---+---+---+---+---+---+---+---+
(20) 8 | 0 |   |   |   |   |   |   |   |   |
       +---+---+---+---+---+---+---+---+---+
```

Para **dp [2] [2]** tenemos que preguntarnos qué es lo mejor que puedo obtener si rompo la varilla en dos piezas (1,1) o si tomo la varilla como un todo (longitud = 2). Podemos ver que si rompo la varilla en dos partes, el beneficio máximo que puedo obtener es 2 y si si tengo la varilla en su

totalidad, puedo venderla por 5. Después de la segunda iteración, la matriz dp [] se verá así:

```
      +---+---+---+---+---+---+---+---+---+
 (price)| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
        +---+---+---+---+---+---+---+---+---+
  (1) 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9 |
        +---+---+---+---+---+---+---+---+---+
  (5) 2 | 0 | 1 | 5 | 6 | 10| 11| 15| 16| 20|
        +---+---+---+---+---+---+---+---+---+
  (8) 3 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
  (9) 4 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
 (10) 5 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
 (17) 6 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
 (17) 7 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
 (20) 8 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
```

Entonces para calcular dp [i] [j] nuestra fórmula se verá así:

```
if j>=i
    dp[i][j] = Max(dp[i-1][j], price[i]+arr[i][j-i]);
else
    dp[i][j] = dp[i-1][j];
```

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

```
        +---+---+---+---+---+---+---+---+---+
 (price)| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
        +---+---+---+---+---+---+---+---+---+
  (1) 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9 |
        +---+---+---+---+---+---+---+---+---+
  (5) 2 | 0 | 1 | 5 | 6 | 10| 11| 15| 16| 20|
        +---+---+---+---+---+---+---+---+---+
  (8) 3 | 0 | 1 | 5 | 8 | 10| 13| 16| 18| 21|
        +---+---+---+---+---+---+---+---+---+
  (9) 4 | 0 | 1 | 5 | 8 | 10| 13| 16| 18| 21|
        +---+---+---+---+---+---+---+---+---+
 (10) 5 | 0 | 1 | 5 | 8 | 10| 13| 16| 18| 21|
        +---+---+---+---+---+---+---+---+---+
 (17) 6 | 0 | 1 | 5 | 8 | 10| 13| 17| 18| 22|
        +---+---+---+---+---+---+---+---+---+
 (17) 7 | 0 | 1 | 5 | 8 | 10| 13| 17| 18| 22|
        +---+---+---+---+---+---+---+---+---+
 (20) 8 | 0 | 1 | 5 | 8 | 10| 13| 17| 18| 22|
        +---+---+---+---+---+---+---+---+---+
```

Tendremos el resultado en **dp [n] [m + 1]** .

Implementación en Java

```
public int getMaximumPrice(int price[],int n){
```

```
        int arr[][] = new int[n][price.length+1];

        for(int i=0;i<n;i++){
            for(int j=0;j<price.length+1;j++){
                if(j==0 || i==0)
                    arr[i][j] = 0;
                else if(j>=i){
                    arr[i][j] = Math.max(arr[i-1][j], price[i-1]+arr[i][j-i]);
                }else{
                    arr[i][j] = arr[i-1][j];
                }
            }
        }
        return arr[n-1][price.length];
    }
```

Complejidad del tiempo

```
O(n^2)
```

Lea Corte de varilla en línea: https://riptutorial.com/es/dynamic-programming/topic/10486/corte-de-varilla

Capítulo 3: Matriz de multiplicación de la cadena

Examples

Solucion recursiva

La multiplicación de la cadena de matrices es un problema de optimización que se puede resolver utilizando la programación dinámica. Dada una secuencia de matrices, el objetivo es encontrar la forma más eficiente de multiplicar estas matrices. El problema no es realmente realizar las multiplicaciones, sino simplemente decidir la secuencia de las multiplicaciones matriciales involucradas.

Digamos que tenemos dos matrices A_1 y A_2 de dimensión $m * n$ y $p * q$. De las reglas de la multiplicación de matrices, sabemos que,

1. Podemos multiplicar A_1 y A_2 si y solo si $n = p$. Eso significa que el número de columnas de A_1 debe ser igual al número de filas de A_2.
2. Si la primera condición se cumple y multiplicamos A_1 y A_2, obtendremos una nueva matriz, llamémosla A_3 de dimensión $m * q$.
3. Para multiplicar A_1 y A_2, necesitamos hacer algunas multiplicaciones de escalador. El número total de multiplicaciones de escalador que debemos realizar es ($m * n * q$) o ($m * p * q$).
4. $A_1 * A_2$ no es igual a $A_2 * A_1$.

Si tenemos tres matrices A_1, A_2 y A_3 que tienen la dimensión $m * n$, $n * p$ y $p * q$ respectivamente, entonces $A_4 = A_1 * A_2 * A_3$ tendrá la dimensión de $m * q$. Ahora podemos realizar esta multiplicación de matrices de dos maneras:

- Primero multiplicamos A_1 y A_2, luego con el resultado multiplicamos A_3. Es decir: $(A_1 * A_2) * A_3$.
- Primero multiplicamos A_2 y A_3, luego con el resultado multiplicamos A_1. Es decir: $A_1 * (A_2 * A_3)$.

Puede observar que el orden de la multiplicación sigue siendo el mismo, es decir, no multiplicamos $(A_1 * A_3) * A_2$ porque podría no ser válido. Solo cambiamos el *paréntesis* para multiplicar un conjunto antes de multiplicarlo con el restante. Cómo colocamos estos paréntesis son importantes. ¿Por qué? Digamos, la dimensión de 3 matrices A_1, A_2 y A_3 son $10 * 100$, $100 * 5$, $5 * 50$. Entonces,

1. Para $(A_1 * A_2) * A_3$, el número total de multiplicaciones del escalador es: $(10 * 100 * 5) + (10 * 5 * 50) = 7500$ veces.
2. Para $A_1 * (A_2 * A_3)$, el número total de multiplicaciones del escalador es: $(100 * 5 * 50) + (10 * 100 * 50) = 75000$ veces.

Para el segundo tipo, ¡el número de la multiplicación del escalador es **10** veces el número del primer tipo! Entonces, si puede idear una manera de averiguar la orientación correcta del paréntesis necesaria para minimizar la multiplicación total del escalador, se reduciría el tiempo y la memoria necesarios para la multiplicación de matrices. Aquí es donde la multiplicación de la cadena de matriz es útil. Aquí, no nos ocuparemos de la multiplicación real de matrices, solo encontraremos el orden de paréntesis correcto para minimizar el número de multiplicaciones del escalador. Tendremos las matrices A_1 , A_2 , A_3 .... A_n y descubriremos el número mínimo de multiplicaciones de escalador necesarias para multiplicar estas. Asumiremos que las dimensiones dadas son válidas, es decir, satisfacen nuestro primer requisito para la multiplicación de matrices.

Usaremos el método de dividir y vencer para resolver este problema. La programación dinámica es necesaria debido a los subproblemas comunes. Por ejemplo: para **n** = **5** , tenemos **5** matrices A_1 , A_2 , A_3 , A_4 y A_5 . Queremos averiguar el número mínimo de multiplicaciones necesarias para realizar esta multiplicación de matrices $A_1 * A_2 * A_3 * A_4 * A_5$. De las muchas maneras, concentrémonos en una: ($A_1 * A_2$) * ($A_3 * A_4 * A_5$).

Para este, descubriremos $A_{izquierda} = A_1 * A_2$. $A_{derecha} = A_3 * A_4 * A_5$. Luego encontraremos **una** $_{respuesta} = A_{izquierda} * A_{derecha}$. El número total de multiplicaciones de escalador necesarias para encontrar **una** $_{respuesta}$: = El número total de multiplicaciones de escalador necesarias para determinar $A_{izquierda}$ + El número total de multiplicaciones de escalador necesarias para determinar $A_{derecha}$ + El número total de multiplicaciones de escalador necesarias para determinar $A_{izquierda}$ * **Un** $_{derecho}$

El último término, el número total de multiplicaciones de escalador necesarias para determinar $A_{izquierda} * A_{derecha}$ puede escribirse como: El número de filas en $A_{izquierda}$ * el número de columnas en $A_{izquierda}$ * el número de columnas en $A_{derecha}$. (De acuerdo con la 2a regla de la multiplicación de matrices)

Pero también podríamos poner el paréntesis de otras maneras. Por ejemplo:

- A_1 * ($A_2 * A_3 * A_4 * A_5$)
- ($A_1 * A_2 * A_3$) * ($A_4 * A_5$)
- ($A_1 * A_2 * A_3 * A_4$) * A_5, etc.

Para cada uno de los casos posibles, determinaremos el número de multiplicaciones de escalador necesarias para encontrar $A_{izquierda}$ y $A_{derecha}$, luego para $A_{izquierda} * A_{derecha}$. Si tiene una idea general sobre la recursión, ya ha comprendido cómo realizar esta tarea. Nuestro algoritmo será:

```
- Set parenthesis in all possible ways.
- Recursively solve for the smaller parts.
- Find out the total number of scaler multiplication by merging left and right.
- Of all possible ways, choose the best one.
```

¿Por qué este es un problema de programación dinámica? Para determinar ($A_1 * A_2 * A_3$), si ya ha calculado ($A_1 * A_2$), será útil en este caso.

Para determinar el estado de esta recursión, podemos ver que para resolver cada caso, necesitamos conocer el rango de matrices con las que estamos trabajando. Así que

necesitaremos un **comienzo** y un **final** . Como estamos utilizando divide y vencerás, nuestro caso base tendrá menos de **2** matrices (**inicio** > = **fin**), donde no necesitamos multiplicar en absoluto. Tendremos **2** matrices: **fila** y **columna** . **la fila [i]** y la **columna [i]** almacenarán el número de filas y columnas para la matriz A_i . Tendremos una matriz **dp [n] [n]** para almacenar los valores ya calculados e inicializarlos con **-1** , donde **-1** representa el valor que aún no se ha calculado. **dp [i] [j]** representa el número de multiplicaciones de escalador necesarias para multiplicar A_i , A_{i+1} , ..., A_j inclusive. El pseudocódigo se verá así:

```
Procedure matrixChain(begin, end):
if begin >= end
    Return 0
else if dp[begin][end] is not equal to -1
    Return dp[begin][end]
end if
answer := infinity
for mid from begin to end
    operation_for_left := matrixChain(begin, mid)
    operation_for_right := matrixChain(mid+1, right)
    operation_for_left_and_right := row[begin] * column[mid] * column[end]
    total := operation_for_left + operation_for_right + operation_for_left_and_right
    answer := min(total, answer)
end for
dp[begin][end] := answer
Return dp[begin][end]
```

Complejidad:

El valor de **inicio** y **final** puede variar de **1** a **n** . Hay n^2 estados diferentes. Para cada estado, el bucle interno se ejecutará **n** veces. Complejidad de tiempo total: `O(n³)` y complejidad de memoria: `O(n²)` .

Lea Matriz de multiplicación de la cadena en línea: https://riptutorial.com/es/dynamic-programming/topic/7996/matriz-de-multiplicacion-de-la-cadena

Capítulo 4: Moneda que cambia el problema

Examples

Número de formas de obtener el total

Dadas monedas de diferentes denominaciones y un total, ¿de cuántas maneras podemos combinar estas monedas para obtener el total? Digamos que tenemos `coins = {1, 2, 3}` y un `total = 5` , podemos obtener el total de **5** maneras:

- 1 1 1 1 1
- 1 1 1 2
- 1 1 3
- 1 2 2
- 2 3

El problema está estrechamente relacionado con el problema de la mochila. La única diferencia es que tenemos un suministro ilimitado de monedas. Vamos a utilizar la programación dinámica para resolver este problema.

Usaremos una matriz 2D **dp [n] [total + 1]** donde **n** es el número de diferentes denominaciones de monedas que tenemos. Para nuestro ejemplo, necesitaremos **dp [3] [6]** . Aquí **dp [i] [j]** indicará el número de formas en que podemos obtener **j** si tuviéramos monedas de **monedas [0]** hasta **monedas [i]** . Por ejemplo, **dp [1] [2]** almacenará si tuviéramos **monedas [0]** y **monedas [1]** , de cuántas maneras podríamos hacer **2** . Vamos a empezar:

Para **dp [0] [0]** , nos preguntamos si solo tenemos **1** denominación de moneda, es decir **monedas [0]** = **1** , ¿de cuántas maneras podemos obtener **0** ? La respuesta es de **1** manera, que es si no tomamos ninguna moneda en absoluto. Continuando, **dp [0] [1]** representará si solo tuviéramos **monedas [0]** , ¿de cuántas maneras podemos obtener **1** ? La respuesta es de nuevo **1** . De la misma manera, **dp [0] [2]** , **dp [0] [3]** , **dp [0] [4]** , **dp [0] [5]** será **1** . Nuestra matriz se verá como:

```
     +---+---+---+---+---+---+---+
(den)|   | 0 | 1 | 2 | 3 | 4 | 5 |
     +---+---+---+---+---+---+---+
 (1) | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
     +---+---+---+---+---+---+---+
 (2) | 1 |   |   |   |   |   |   |
     +---+---+---+---+---+---+---+
 (3) | 2 |   |   |   |   |   |   |
     +---+---+---+---+---+---+---+
```

Para **dp [1] [0]** , nos preguntamos si teníamos monedas de **2** denominaciones, es decir, si tuviéramos **monedas [0]** = **1** y **monedas [1]** = **2** , ¿de cuántas maneras podríamos hacer **0** ? La respuesta es **1** , que es no tomar ninguna moneda. Para **dp [1] [1]** , dado que las **monedas [1]** = **2** es mayor que nuestro total actual, **2** no contribuirá a obtener el total. Por lo tanto, excluiremos **2**

y contaremos el número de formas en que podemos obtener el total usando **monedas [0]** . ¡Pero este valor ya está almacenado en **dp [0] [1]** ! Así que vamos a tomar el valor de la parte superior. Nuestra primera fórmula:

```
if coins[i] > j
    dp[i][j] := dp[i-1][j]
end if
```

Para **dp [1] [2]** , ¿de cuántas maneras podemos obtener **2** , si tuviéramos monedas de denominación **1** y **2** ? Podemos hacer **2** usando monedas de denominación de **1** , que están representadas por **dp [0] [2]** , otra vez podemos tomar **1** denominación de **2** que está almacenada en **dp [1] [2-coins [i]]** , donde **i** = **1** . ¿Por qué? Será evidente si nos fijamos en el siguiente ejemplo. Para **dp [1] [3]** , ¿de cuántas maneras podemos obtener **3** , si tuviéramos monedas de denominación **1** y **2** ? Podemos hacer **3** usando monedas de denominación **1** en **1** forma, que se almacenan en **dp [0] [3]** . Ahora, si tomamos **1** denominación de **2** , necesitaremos **3** - **2** = **1** para obtener el total. El número de formas de obtener **1** usando las monedas de denominación **1** y **2** se almacena en **dp [1] [1]** , que se puede escribir como, **dp [i] [j-coins [i]]** , donde **i** = **1** . Por eso escribimos el valor anterior de esta manera. Nuestra segunda y última fórmula será:

```
if coins[i] <= j
    dp[i][j] := dp[i-1][j] + dp[i][j-coins[i]]
end if
```

Estas son las dos fórmulas necesarias para completar toda la matriz **dp** . Después de llenar la matriz se verá como:

```
     +---+---+---+---+---+---+---+
(den)|   | 0 | 1 | 2 | 3 | 4 | 5 |
     +---+---+---+---+---+---+---+
 (1) | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
     +---+---+---+---+---+---+---+
 (2) | 1 | 1 | 1 | 2 | 2 | 3 | 3 |
     +---+---+---+---+---+---+---+
 (3) | 2 | 1 | 1 | 2 | 3 | 4 | 5 |
     +---+---+---+---+---+---+---+
```

dp [2] [5] contendrá nuestra respuesta requerida. El algoritmo:

```
Procedure CoinChange(coins, total):
n := coins.length
dp[n][total + 1]
for i from 0 to n
    dp[i][0] := 1
end for
for i from 0 to n
    for j from 1 to (total + 1)
        if coins[i] > j
            dp[i][j] := dp[i-1][j]
        else
            dp[i][j] := dp[i-1][j] + dp[i][j-coins[i]]
        end if
```

```
    end for
end for
Return dp[n-1][total]
```

La complejidad temporal de este algoritmo es `O(n * total)` , donde **n** es el número de denominaciones de monedas que tenemos.

Cantidad mínima de monedas para obtener el total

Dadas las monedas de diferentes denominaciones y un total, ¿cuántas monedas necesitamos combinar para obtener el total si utilizamos el número mínimo de monedas? Digamos que tenemos `coins = {1, 5, 6, 8}` y un `total = 11` , podemos obtener el total usando **2** monedas que es `{5, 6}` . Este es de hecho el número mínimo de monedas requeridas para obtener **11** . También asumiremos que hay un suministro ilimitado de monedas. Vamos a utilizar la programación dinámica para resolver este problema.

Usaremos una matriz 2D **dp [n] [total + 1]** donde **n** es el número de diferentes denominaciones de monedas que tenemos. Para nuestro ejemplo, necesitaremos **dp [4] [12]** . Aquí **dp [i] [j]** indicará el número mínimo de monedas necesarias para obtener **j** si tuviéramos monedas de **monedas [0]** hasta **monedas [i]** . Por ejemplo, **dp [1] [2]** almacenará si tuviéramos **monedas [0]** y **monedas [1]** , ¿cuál es el número mínimo de monedas que podemos usar para hacer **2** ? Vamos a empezar:

Al principio, para la columna **0** , puede hacer **0** al no tomar ninguna moneda. Así que todos los valores de **0** º columna será **0.** Para **dp [0] [1]** , nos preguntamos si solo tenemos **1** denominación de moneda, es decir, **monedas [0]** = **1** , ¿cuál es el número mínimo de monedas necesarias para obtener **1** ? La respuesta es **1** . Para **dp [0] [2]** , si solo tuviéramos **1** , ¿cuál es el número mínimo de monedas necesarias para obtener **2** ? La respuesta es **2** . De manera similar, **dp [0] [3]** = **3** , **dp [0] [4]** = **4** y así sucesivamente. Una cosa a mencionar aquí es que, si no tenemos una moneda de denominación **1,** puede haber algunos casos en los que no se puede lograr la total usando solamente **1** moneda. Por simplicidad tomamos **1** en nuestro ejemplo. Después de la primera iteración nuestra matriz **dp** se verá así:

```
          +---+---+---+---+---+---+---+---+---+---+---+---+---+
 (denom)|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11|
          +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (1) | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11|
          +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (5) | 1 | 0 |   |   |   |   |   |   |   |   |   |   |   |
          +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (6) | 2 | 0 |   |   |   |   |   |   |   |   |   |   |   |
          +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (8) | 3 | 0 |   |   |   |   |   |   |   |   |   |   |   |
          +---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Continuando, para **dp [1] [1]** , nos preguntamos si teníamos **monedas [0]** = **1** y **monedas [1]** = **5** , ¿cuál es el número mínimo de monedas necesarias para obtener **1** ? Como las **monedas [1]** son mayores que nuestro total actual, no afectará nuestro cálculo. Tendremos que excluir **monedas [5]** y obtener **1** usando **monedas [0]** solamente. Este valor se almacena en **dp [0] [1]** . Así que tomamos el valor de la parte superior. Nuestra primera fórmula es:

```
if coins[i] > j
    dp[i][j] := dp[i-1][j]
```

Esta condición será verdadera hasta que nuestro total sea **5** en **dp [1] [5]** , para ese caso, podemos hacer **5** de dos maneras:

- Tomamos **5** denominaciones de **monedas [0]** , que se almacenan en **dp [0] [5]** (desde la parte superior).
- Tomamos **1** denominación de **monedas [1]** y (**5** - **5**) = **0** denominaciones de **monedas [0]** .

Elegiremos el mínimo de estos dos. Entonces **dp [1] [5]** = min (**dp [0] [5]** , **1** + **dp [1] [0]**) = **1** . ¿Por qué mencionamos **0** denominaciones de **monedas? [0]** aquí será evidente en nuestra próxima posición.

Para **dp [1] [6]** , podemos hacer **6** de dos maneras:

- Tomamos **6** denominaciones de **monedas [0]** , que se almacenan en la parte superior.
- Podemos tomar **1** denominación de **5** , necesitaremos **6** - **5** = **1** para obtener el total. El número mínimo de formas de obtener **1** utilizando las monedas de denominación **1** y **5** se almacena en **dp [1] [1]** , que se puede escribir como **dp [i] [j-coins [i]]** , donde **i** = **1** . Por eso escribimos el valor anterior de esa manera.

Tomaremos el mínimo de estas dos formas. Así que nuestra segunda fórmula será:

```
if coins[i] >= j
    dp[i][j] := min(dp[i-1][j], dp[i][j-coins[i]])
```

Usando estas dos fórmulas podemos completar toda la tabla. Nuestro resultado final será:

```
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
  (denom)|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11|
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (1) | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11|
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (5) | 1 | 0 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2 | 3 |
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (6) | 2 | 0 | 1 | 2 | 3 | 4 | 1 | 1 | 2 | 3 | 4 | 2 | 2 |
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (8) | 3 | 0 | 1 | 2 | 3 | 4 | 1 | 1 | 2 | 1 | 2 | 2 | 2 |
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Nuestro resultado requerido se almacenará en **dp [3] [11]** . El procedimiento será:

```
Procedure coinChange(coins, total):
n := coins.length
dp[n][total + 1]
for i from 0 to n
    dp[i][0] := 0
end for
for i from 1 to (total + 1)
    dp[0][i] := i
end for
for i from 1 to n
```

```
    for j from 1 to (total + 1)
        if coins[i] > j
            dp[i][j] := dp[i-1][j]
        else
            dp[i][j] := min(dp[i-1][j], dp[i][j-coins[i]])
        end if
    end for
end for
Return dp[n-1][total]
```

La complejidad del tiempo de ejecución de este algoritmo es: O (n * total) donde **n** es el número de denominaciones de monedas.

Para imprimir las monedas necesarias, tenemos que comprobar:

- Si el valor vino de arriba, entonces la moneda actual no está incluida.
- Si el valor vino de la izquierda, entonces se incluye la moneda actual.

El algoritmo sería:

```
Procedure printChange(coins, dp, total):
i := coins.length - 1
j := total
min := dp[i][j]
while j is not equal to 0
    if dp[i-1][j] is equal to min
        i := i - 1
    else
        Print(coins[i])
        j := j - coins[i]
    end if
end while
```

Para nuestro ejemplo, la dirección será:

```
        +---+---+---+---+---+---+---+---+---+---+---+---+---+
 (denom)|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11|
        +---+---+---+---+---+---+---+---+---+---+---+---+---+
    (1) | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11|
        +---+---+---+---+---+---+---+---+---+---+---+---+---+
    (5) | 1 | 0 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2 | 3 |
        +---+---+---+---+---+---+---+---+---+---+---+---+---+
    (6) | 2 | 0 | 1 | 2 | 3 | 4 | 1 | 1 | 2 | 3 | 4 | 2 | 2 |
        +---+---+---+---+---+---+---+---+---+---+---+---+---+
    (8) | 3 | 0 | 1 | 2 | 3 | 4 | 1 | 1 | 2 | 1 | 2 | 2 | 2 |
        +---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Los valores serán **6** , **5** .

Lea Moneda que cambia el problema en línea: https://riptutorial.com/es/dynamic-programming/topic/8013/moneda-que-cambia-el-problema

Capítulo 5: Problema de mochila

Observaciones

El problema de mochila o de mochila es un problema en la optimización combinatoria . Dado un conjunto de artículos, cada uno con un peso y un valor, 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 sea lo más grande posible. Deriva su nombre del problema al que se enfrenta alguien que está limitado por una mochila de tamaño fijo y debe llenarla con los elementos más valiosos.

El problema a menudo surge en la asignación de recursos donde existen limitaciones financieras y se estudia en campos tales como combinatoria , informática , teoría de la complejidad , criptografía , matemáticas aplicadas y deportes de fantasía diarios .

El problema de la mochila se ha estudiado durante más de un siglo, con trabajos anteriores que datan de 1897. El nombre "problema de la mochila" se remonta a los primeros trabajos del matemático Tobias Dantzig (1884-1956) y se refiere al problema común de Empaque sus artículos más valiosos o útiles sin sobrecargar su equipaje.

Examples

0-1 Problema de mochila

Supongamos que se le pregunta, dado el peso total que puede llevar en su mochila y algunos artículos con su peso y valores, ¿cómo puede tomar esos artículos de tal manera que la suma de sus valores sea máxima, pero la suma de sus pesos no no exceder el peso total que puede llevar? El **0-1** indica que eliges el ítem o no. También tenemos una cantidad de cada artículo. Significa que, no se puede dividir el elemento. Si no fue un **problema de mochila 0-1** , eso significa que si pudo dividir los artículos, hay una solución codiciosa, que se llama **problema de mochila fraccional** .

Por ahora, concentrémonos en nuestro problema actual. Por ejemplo, digamos que tenemos una capacidad de mochila de **7** . Tenemos **4** artículos. Sus pesos y valores son:

```
+----------+---+---+---+---+
|   Item   | 1 | 2 | 3 | 4 |
+----------+---+---+---+---+
|  Weight  | 1 | 3 | 4 | 5 |
+----------+---+---+---+---+
|   Value  | 1 | 4 | 5 | 7 |
+----------+---+---+---+---+
```

Un método de fuerza bruta sería tomar todas las combinaciones posibles de elementos. Luego, podemos calcular sus pesos totales y excluirlos que excedan la capacidad de nuestra mochila y descubrir el que nos da el máximo valor. Para **4** artículos, tendremos que verificar (**4!** - **1**) = **23** combinaciones posibles (excluyendo una sin artículos). Esto es bastante engorroso cuando

aumenta la cantidad de elementos. Aquí, algunos aspectos que podemos notar, es decir:

- Podemos tomar ítems menores y calcular el valor máximo que podemos obtener con esos ítems y combinarlos. Entonces nuestro problema se puede dividir en subproblemas.
- Si calculamos las combinaciones para el elemento **{1,2}** , podemos usarlo cuando calculamos **{1, 2, 3}** .
- Si minimizamos el peso y maximizamos el valor, podemos encontrar nuestra solución óptima.

Por estas razones, usaremos la programación dinámica para resolver nuestro problema. Nuestra estrategia será: cada vez que llegue un nuevo artículo, verificaremos si podemos elegir el artículo o no, y nuevamente seleccionaremos los artículos que nos dan el máximo valor. Ahora, si seleccionamos el artículo, nuestro valor será el valor del artículo, más el valor que podamos obtener al restar el valor de nuestra capacidad y el máximo que podemos obtener para el peso restante. Si no seleccionamos el artículo, seleccionaremos los artículos que nos den el valor máximo sin incluir el artículo. Intentemos entenderlo con nuestro ejemplo:

Tomaremos una **tabla de** matriz 2D, donde el número de columnas será el valor máximo que podemos obtener al tomar los elementos + 1. Y el número de filas será el mismo que el número de elementos. Nuestra matriz se verá como:

```
+-------+--------+---+---+---+---+---+---+---+---+
| Value | Weight | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-------+--------+---+---+---+---+---+---+---+---+
|   1   |    1   | 0 |   |   |   |   |   |   |   |
+-------+--------+---+---+---+---+---+---+---+---+
|   4   |    3   | 0 |   |   |   |   |   |   |   |
+-------+--------+---+---+---+---+---+---+---+---+
|   5   |    4   | 0 |   |   |   |   |   |   |   |
+-------+--------+---+---+---+---+---+---+---+---+
|   7   |    5   | 0 |   |   |   |   |   |   |   |
+-------+--------+---+---+---+---+---+---+---+---+
```

Hemos incorporado el peso y el valor de cada elemento a la matriz para nuestra conveniencia. Recuerde que estos no son parte de la matriz, son solo para fines de cálculo, no necesita almacenar estos valores en la matriz de **tabla** .

Nuestra primera columna está llena de **0** . Significa que si nuestra capacidad máxima es **0** , sin importar el artículo que tengamos, ya que no podemos elegir ningún artículo, nuestro valor máximo será **0** . Empecemos por la **tabla [0] [1]** . Cuando llenamos la **Tabla [1] [1]** , nos preguntamos si nuestra capacidad máxima era **1** y solo teníamos el primer elemento, ¿cuál sería nuestro valor máximo? Lo mejor que podemos hacer es **1** , eligiendo el artículo. Para la **Tabla [0] [2]** eso significa que si nuestra capacidad máxima es **2** y solo tenemos el primer elemento, el valor máximo que podemos obtener es **1** . Esto será igual para la **Tabla [0] [3]** , la **Tabla [0] [4]** , la **Tabla [0] [5]** , la **Tabla [0] [6]** y la **Tabla [0] [7]** . Esto se debe a que solo tenemos un elemento, lo que nos da valor **1** . Como solo tenemos **1** cantidad de cada artículo, no importa cómo aumentemos la capacidad de nuestra mochila, de un artículo, **1** es el mejor valor que podemos obtener. Así que nuestra matriz se verá como:

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

```
| Value | Weight | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-------+--------+---+---+---+---+---+---+---+---+
|   1   |    1   | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
+-------+--------+---+---+---+---+---+---+---+---+
|   4   |    3   | 0 |   |   |   |   |   |   |   |
+-------+--------+---+---+---+---+---+---+---+---+
|   5   |    4   | 0 |   |   |   |   |   |   |   |
+-------+--------+---+---+---+---+---+---+---+---+
|   7   |    5   | 0 |   |   |   |   |   |   |   |
+-------+--------+---+---+---+---+---+---+---+---+
```

Continuando, para la **Tabla [1] [1]** , nos preguntamos si teníamos los ítems **1** y **2** y si la capacidad máxima de nuestra mochila era **1** , ¿cuál es el valor máximo que podemos obtener? Si tomamos los ítems **1** y **2** , el peso total será **4** , lo que excederá nuestra capacidad de mochila actual. Así que el artículo **2** no puede ser seleccionado. Ahora, ¿qué es lo mejor que podemos hacer sin tomar el artículo **2** ? El valor de la parte superior, que es la **Tabla [0] [1],** que contiene el valor máximo que podemos obtener si tuviéramos la capacidad máxima **1** y no seleccionáramos el segundo elemento. Para la **Tabla [1] [2]** , dado que **2** es menor que el **peso [2]** , que es el peso del segundo artículo, no podemos tomar el artículo. Así que podemos establecer que, si el peso del artículo actual es mayor que nuestra capacidad máxima, no podemos tomar ese artículo. En este caso, simplemente tomaremos el valor de arriba, que representa el valor máximo que podemos tomar excluyendo el artículo.

```
if weight[i] > j
    Table[i][j] := Table[i-1][j]
end if
```

Ahora para la **Tabla [1] [3]** ya que nuestra capacidad máxima es igual a nuestro peso actual, tenemos dos opciones.

- Seleccionamos el artículo y agregamos su valor con el valor máximo que podemos obtener de los demás elementos restantes después de tomar este artículo.
- Podemos excluir este artículo.

Entre las dos opciones, elegiremos una de la que podamos obtener el máximo valor. Si seleccionamos el artículo, obtenemos: valor para este artículo + valor máximo del resto de los artículos después de tomar este artículo = **4 + 0 = 4** . Obtenemos **4** (valor del artículo) de nuestra matriz de **peso** y el **0** (valor máximo que podemos obtener del resto de los artículos después de tomar este artículo) yendo **1** paso arriba y **3** pasos atrás. Eso es la **tabla [0] [0]** . ¿Por qué? Si tomamos el artículo, nuestra capacidad restante será **3** - **3** = **0** y el artículo restante será el primer artículo. Bueno, si recuerdas la **Tabla [0] [0]** almacena el valor máximo que podemos obtener si nuestra capacidad fuera **0** y solo tuviéramos el primer elemento. Ahora, si no seleccionamos el artículo, el valor máximo que podemos obtener proviene del paso **1** anterior, que es **1** . Ahora tomamos el máximo de estos dos valores (**4** , **1**) y configuramos la **Tabla [1] [2]** = **4** . Para la **Tabla [1] [4]** , desde **4** , la capacidad máxima de mochila es mayor que **3** , el peso de nuestro artículo actual, nuevamente tenemos dos opciones. Tomamos max (**Peso [2]** + **Tabla [0] [4-Weight [2]]** , **Tabla [0] [4]**) = max (**Peso [2] + Tabla [0] [1]** , **Tabla [0] [4]**) = max (**4 + 1** , **1**) = **5** .

```
if weight[i] <= j
    w := weight[i]
    Table[i][j] := max(w + Table[i][j-w], Table[i-1][j])
end if
```

Usando estas dos fórmulas, podemos rellenar toda **la** matriz de la **tabla** . Nuestra matriz se verá como:

```
+-------+--------+---+---+---+---+---+---+---+---+
| Value | Weight | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-------+--------+---+---+---+---+---+---+---+---+
|   1   |    1   | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
+-------+--------+---+---+---+---+---+---+---+---+
|   4   |    3   | 0 | 1 | 1 | 4 | 5 | 5 | 5 | 5 |
+-------+--------+---+---+---+---+---+---+---+---+
|   5   |    4   | 0 | 1 | 1 | 4 | 5 | 6 | 6 | 9 |
+-------+--------+---+---+---+---+---+---+---+---+
|   7   |    5   | 0 | 1 | 1 | 4 | 5 | 7 | 8 | 9 |
+-------+--------+---+---+---+---+---+---+---+---+
```

Aquí, el último valor que insertamos en nuestra matriz, la **Tabla [3] [7]** contiene nuestro valor máximo requerido. Este es el valor máximo que podemos obtener si tuviéramos **4** artículos y nuestra capacidad máxima de mochila fuera de **7** .

Una cosa que debemos recordar es que, incluso en la primera fila, el peso puede ser mayor que la capacidad de la mochila. Necesitaremos mantener otra restricción para verificar el valor mientras se llena la primera fila. O simplemente podemos tomar otra fila y establecer todos los valores de la primera fila en **0** . El pseudocódigo se vería así:

```
Procedure Knapsack(Weight, Value, maxCapacity):
n := Item.size - 1
Table[n+1][maxCapacity+1]
for i from 0 to n
    Table[i][0] := 0
end for
for j from 1 to maxCapacity
    if j >= Weight[0]
        Table[0][j] := Weight[0]
    else
        Table[0][j] := 0
    end if
end for
for i from 1 to n
    for j from 1 to maxCapacity
        if Weight[i] >= j                                            //can't pick the item
            Table[i][j] := Table[i-1][j]
        else                                                        //can pick the item
            w := Weight[i]
            Table[i][j] := max(w + Table[i-1][j-w], Table[i-1][j])
        end if
    end for
end for
Return Table[n][maxCapacity]
```

La complejidad del tiempo de este algoritmo es `O(n*maxCapacity)` , donde **n** es el número de

elementos y `maxCapacity` es la capacidad máxima de nuestra mochila.

Hasta ahora, hemos encontrado el valor máximo que podemos obtener de nuestro ejemplo. Aún queda una pregunta. ¿Cuáles son los artículos reales? Volveremos sobre los valores en nuestra matriz de **tabla** para descubrir los elementos que hemos tomado. Seguiremos dos estrategias:

- Para cualquier elemento, si el valor proviene de la posición anterior, no tomamos el elemento actual. Vamos un paso más arriba.
- Si el valor no proviene de la posición anterior, tomamos el artículo. Así que vamos 1 paso arriba y x retrocede, donde x es el peso del artículo actual.

El pseudocódigo será:

```
Procedure printItems(Table, maxCapacity, Value):
i := Item.size - 1
j := maxCapacity
while j is not equal to 0
    if Table[i][j] is equal to Table[i-1][j]
        i := i - 1
    else
        Print: i
        j := j - weight[i]
        i := i - 1
    end if
end while
```

Si volvemos sobre nuestro ejemplo, obtendremos:

```
+-------+--------+---+---+---+---+---+---+---+---+
| Value | Weight | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-------+--------+---+---+---+---+---+---+---+---+
|   1   |    1   | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
+-------+--------+---+---+---+---+---+---+---+---+
|   4   |    3   | 0 | 1 | 1 | 4 | 5 | 5 | 5 | 5 |
+-------+--------+---+---+---+---+---+---+---+---+
|   5   |    4   | 0 | 1 | 1 | 4 | 5 | 6 | 6 | 9 |
+-------+--------+---+---+---+---+---+---+---+---+
|   7   |    5   | 0 | 1 | 1 | 4 | 5 | 7 | 8 | 9 |
+-------+--------+---+---+---+---+---+---+---+---+
```

A partir de esto, podemos decir que podemos tomar los ítems **2** y **3** para obtener el valor máximo.

Lea Problema de mochila en línea: https://riptutorial.com/es/dynamic-programming/topic/7972/problema-de-mochila

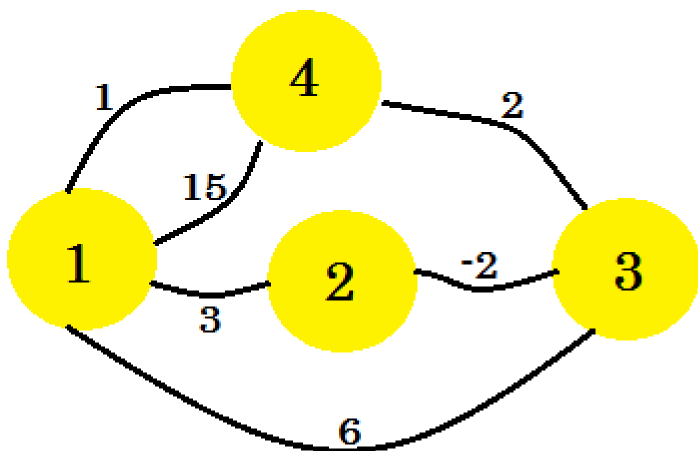
Capítulo 6: Resolviendo problemas de gráficas usando programación dinámica

Examples

Algoritmo de Floyd-Warshall

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

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

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

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:

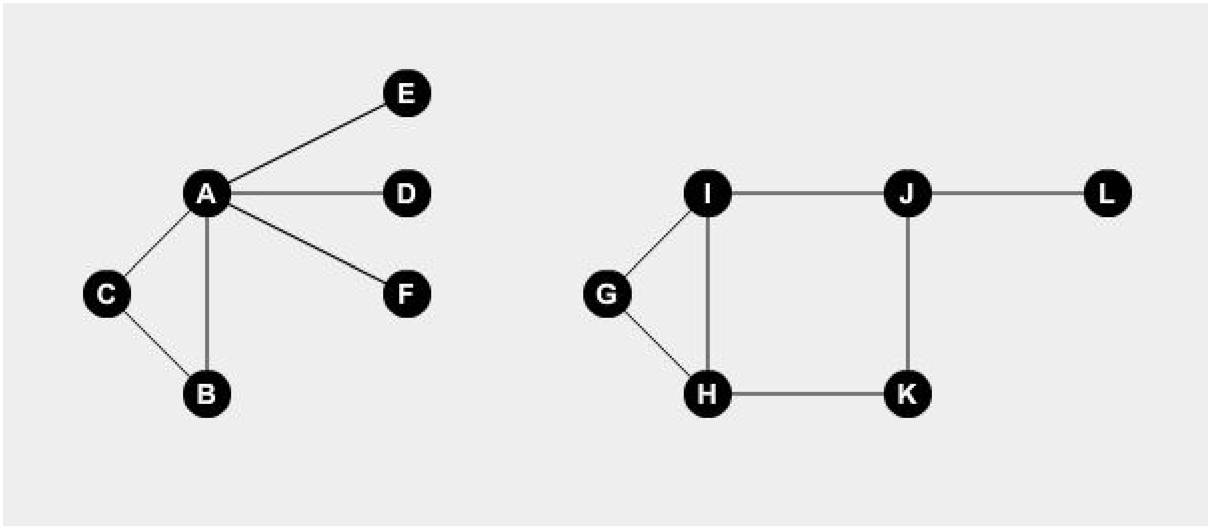
Para saber si hay un ciclo de borde negativo, tendremos que verificar la diagonal principal de la matriz de **distancia** . Si algún valor en la diagonal es negativo, eso significa que hay un ciclo negativo en la gráfica.

Complejidad:

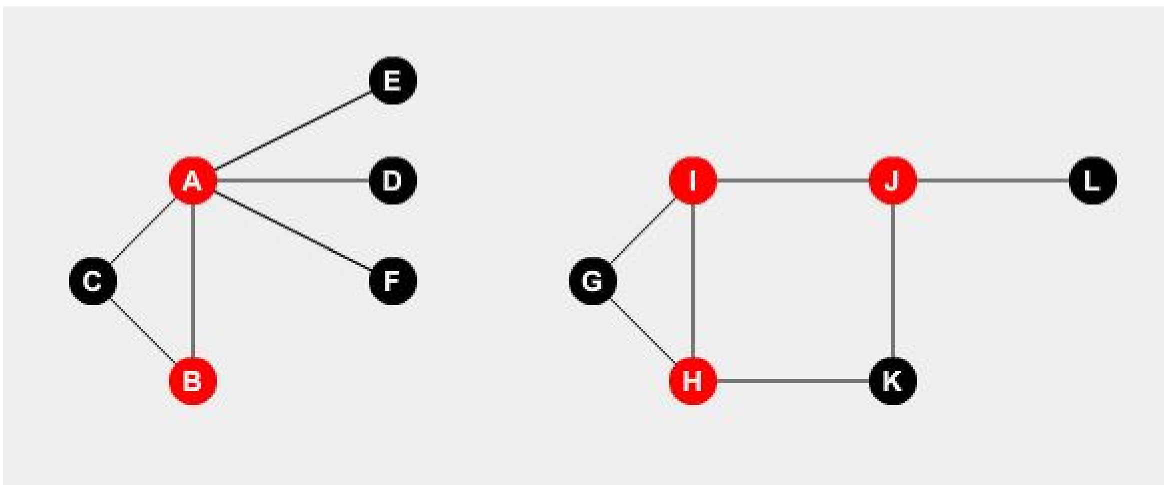
La complejidad del algoritmo Floyd-Warshall es **O (V^3)** y la complejidad del espacio es: **O (V^2)** .

Cubierta de vértice mínimo

La cubierta mínima de vértices es un problema de gráficos clásico. Digamos que en una ciudad tenemos algunas carreteras que conectan algunos puntos. Representemos los caminos usando bordes y los puntos usando nodos. Tomemos dos gráficos de ejemplo:

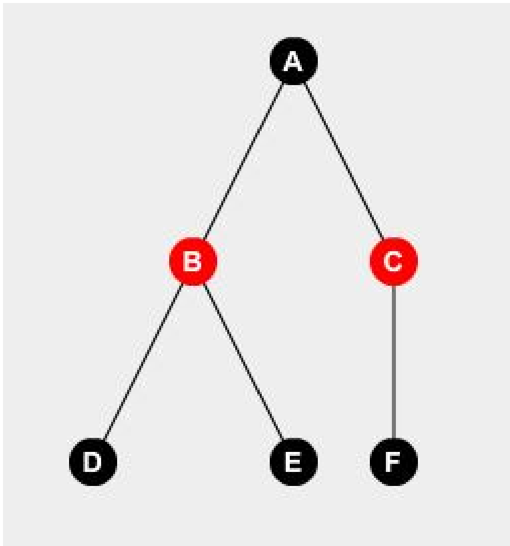


Queremos poner vigilantes en algunos puntos. Un vigilante puede proteger todos los caminos conectados al punto. El problema es, ¿cuál es el número mínimo de vigilantes necesarios para cubrir todas las carreteras? Si colocamos vigilantes en los nodos **A** , **B** , **H** , **I** y **J** , podemos cubrir todas las carreteras.



Esta es nuestra solución óptima. Necesitamos al menos **5** vigilantes para vigilar toda la ciudad. ¿Cómo determinar esto?

Al principio, debemos entender que este es un *problema NP-difícil* , es decir, el problema no tiene una solución de tiempo polinómico. Pero si el gráfico era un **árbol** , eso significa que si tenía nodos **(n-1)** donde **n** es el número de bordes y no hay un ciclo en el gráfico, podemos resolverlo mediante la programación dinámica.



Para construir una solución de DP, debemos seguir dos estrategias:

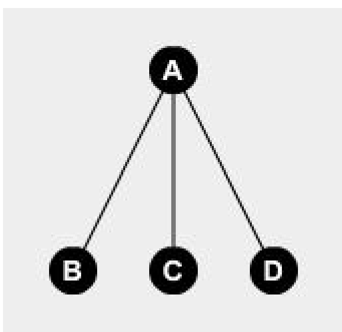
1. Si no hay un vigilante en un nodo, todos los nodos conectados a él deben tener un vigilante, o todas las carreteras no estarán cubiertas. Si **u** y **v** están conectados y **u** no tiene ningún vigilante, entonces **v** debe tener un vigilante.
2. Si hay un vigilante en un nodo, un nodo diferente conectado a él puede o no tener un vigilante. Eso significa que no es necesario tener un vigilante, pero puede ser beneficioso. Si **u** y **v** están conectados y **u** tiene un vigilante, verificaremos y encontraremos cuál es beneficioso para nosotros al:
   - Poniendo vigilante en **v** .
   - No poner vigilante en **v** .

Definamos una función recursiva con el estado que es el nodo actual en el que estamos y si tiene un vigilante o no. Aquí:

```
F(u,1) = Currently we're in 'u' node and there is a watchman in this node.
F(u,0) = Currently we're in 'u' node and there is no watchman in this node.
```

La función devolverá el número de vigilantes en los nodos restantes.

Tomemos un árbol de ejemplo:



Podemos decir fácilmente que si no ponemos al vigilante en el nodo **A** , tendremos que poner vigilantes en el nodo **B** , **C** y **D.** Podemos deducir:

```
F(A,0) = F(B,1) + F(C,1) + F(D,1) + 0
```

Nos devuelve el número de vigilantes necesarios si no colocamos al vigilante en el nodo **A.** Agregamos **0** al final porque no configuramos ningún vigilante en nuestro nodo actual.

Ahora `F(A,1)` significa que configuramos a watchman en el nodo **A.** Para eso, podemos establecer vigilantes en todos los nodos conectados o no. Tomaremos el que nos proporcione el número mínimo de vigilantes.

```
F(A,1) = min(F(B,0), F(B,1) + min(F(C,0), F(C,1)) + min(F(D,0), F(D,1)) + 1
```

Verificamos estableciendo y no configurando al vigilante en cada nodo y tomando el valor óptimo.

Una cosa que debemos tener cuidado es que, una vez que vamos al nodo secundario, nunca volveremos a mirar el nodo principal. Del ejemplo anterior, fuimos a **B** desde **A** , entonces el **padre [B]** = **A.** Así que no volveremos a la **A** de la **B.**

Para determinar el caso base, podemos notar que, si desde un nodo, no podemos ir a ningún otro nodo nuevo, devolveremos **1** si hay un vigilante en nuestro nodo actual, **0** si no hay un vigilante en nuestro actual nodo.

Es mejor tener una lista de adyacencia para nuestro árbol. Deja que la lista se denote por **borde** . Tendremos una matriz **dp [n] [2]** , donde **n** denota el número de nodos para almacenar los valores calculados e inicializarlos con **-1** . También tendremos una **matriz principal [n]** para denotar la relación principal y secundaria entre nodos. Nuestro pseudo-código se verá así:

```
Procedure f(u, isGuarded):
if edge[u].size is equal to 0                    //node doesn't have any new edge
    Return isGuarded
else if dp[u][isGuarded] is not equal to -1      //already calculated
    Return dp[u][isGuarded]
end if
sum := 0
for i from i to edge[u].size
    v := edge[u][i]
    if v is not equal to parent[u]               //not a parent
        parent[v] := u
        if isGuarded equals to 0                 //not guarded, must set a watchman
            sum := sum + f(v,1)
        else                                     //guarded, check both
            sum := sum + min(f(v,1), f(v,0)
        end if
    end if
end for
dp[u][isGuarded] := sum + isGuarded
Return dp[u][isGuarded]
```

Si denotamos el nodo **A** como raíz, llamaremos la función por: `min(f(A,1), f(A,0))` . Eso significa que también verificaremos si es óptimo establecer a watchman en el nodo raíz o no. Esta es nuestra solución de DP. Este problema también se puede resolver utilizando el algoritmo de coincidencia máxima o max-flow.

Lea Resolviendo problemas de gráficas usando programación dinámica en línea: https://riptutorial.com/es/dynamic-programming/topic/7979/resolviendo-problemas-de-graficas-usando-programacion-dinamica

Capítulo 7: Selección de actividad ponderada

Examples

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]
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.

Lea Selección de actividad ponderada en línea: https://riptutorial.com/es/dynamic-programming/topic/7999/seleccion-de-actividad-ponderada

Capítulo 8: Subsecuencias relacionadas con los algoritmos

Examples

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 compararlos 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
    +-----+-----+-----+-----+-----+-----+-----+-----+
    | ch  |     |  a  |  b  |  c  |  d  |  a  |  f  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
  0 |     |     |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
  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
    +-----+-----+-----+-----+-----+-----+-----+-----+
    | ch  |     |  a  |  b  |  c  |  d  |  a  |  f  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
  0 |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
  1 |  a  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
  2 |  c  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
  3 |  b  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
  4 |  c  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
  5 |  f  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
```

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  |  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  |  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  |  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  |  1  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  2  |  c  |  0  |  1  |  1  |  2  |  2  |  2  |  2  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  3  |  b  |  0  |  1  |  2  |  2  |  2  |  2  |  2  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  4  |  c  |  0  |  1  |  2  |  3  |  3  |  3  |  3  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  5  |  f  |  0  |  1  |  2  |  3  |  3  |  3  |  4  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
```

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))** .

La subsecuencia cada vez mayor

La tarea es encontrar la longitud de la subsecuencia más larga en una matriz dada de enteros, de modo que todos los elementos de la subsecuencia se clasifiquen en orden ascendente. Por ejemplo, la longitud de la subsecuencia creciente más larga (LIS) para **{15, 27, 14, 38, 26, 55, 46, 65, 85}** es **6** y la subsecuencia creciente más larga es **{15, 27, 38, 55, 65, 85}** . Nuevamente, para **{3, 4, -1, 0, 6, 2, 3} la** longitud de LIS es **4** y la subsecuencia es **{-1, 0, 2, 3}** . Usaremos la programación dinámica para resolver este problema.

Tomemos el segundo ejemplo: `Item = {3, 4, -1, 0, 6, 2, 3}` . Comenzaremos tomando un **dp** de matriz del mismo tamaño de nuestra secuencia. **dp [i]** representa la longitud del LIS si incluimos el elemento **i** th de nuestra secuencia original. Al principio, sabemos que, para cada uno de los elementos, al menos la subsecuencia de mayor crecimiento es de longitud **1** . Eso es considerando el elemento único en sí. Así que vamos a inicializar la matriz **dp** con **1** . Tendremos dos variables **i** y **j** . Inicialmente **i** será **1** y **j** a ser **0.** Nuestra matriz se verá como:

```
           3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  1  |  1  |  1  |  1  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
           j     i
```

El número sobre la matriz representa el elemento correspondiente de nuestra secuencia. Nuestra estrategia será:

```
if Item[i] > Item[j]
    dp[i] := dp[j] + 1
```

Eso significa que si el elemento en **i** es mayor que el elemento en **j** , la longitud del LIS que contiene el elemento en **j** , aumentará en longitud **1** si incluimos el elemento en **i** con él. En nuestro ejemplo, para **i** = **1** y **j** = **0** , el **elemento [i]** es mayor que el **elemento [j]** . Entonces **dp [i]** = **dp [j]** + **1** . Nuestra matriz se verá como:

```
           3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  1  |  1  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
           j     i
```

En cada paso, aumentaremos **j** hasta **i** y luego restableceremos **j** a **0** e incrementaremos **i** . Por ahora, **j** ha alcanzado **i** , por lo que incrementamos **i** a **2** .

```
           3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  1  |  1  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
           j           i
```

Para **i** = **2** , **j** = **0** , el **elemento [i]** no es mayor que el **elemento [j]** , por lo que no hacemos nada e incrementamos **j** .

```
           3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  1  |  1  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
                 j     i
```

Para **i** = **2** , **j** = **1** , el **elemento [i]** no es mayor que el **elemento [j]** , por lo que no hacemos nada y dado que **j** ha alcanzado su límite, incrementamos **i** y restablecemos **j** a **0** .

```
           3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  1  |  1  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
           j                 i
```

Para **i** = **3** , **j** = **0** , el **elemento [i]** no es mayor que el **elemento [j]** , por lo que no hacemos nada e incrementamos **j** .

```
           3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  1  |  1  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
                 j           i
```

Para **i** = **3** , **j** = **1** , el **elemento [i]** no es mayor que el **elemento [j]** , por lo que no hacemos nada e incrementamos **j** .

```
           3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  1  |  1  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
                       j     i
```

Para **i** = **3** , **j** = **2** , el **elemento [i]** es mayor que el **elemento [j]** , por lo que **dp [i]** = **dp [j] + 1** . Después de eso, ya que **j** ha alcanzado su límite, nuevamente restablecemos **j** a **0** e incrementamos **i** .

```
           3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  2  |  1  |  1  |  1  |
```

```
+-------+-----+-----+-----+-----+-----+-----+-----+
           j                       i
```

Para **i** = **4** y **j** = **0** , el **elemento [i]** es mayor que el **elemento [j]** , por lo que **dp [i]** = **dp [j] + 1** . Después de eso, incrementamos **j** .

```
           3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  2  |  2  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
                 j                 i
```

Para **i** = **4** y **j** = **1** , el **elemento [i]** es mayor que el **elemento [j]** . También podemos notar que **dp [i]** = **dp [j] + 1** nos proporcionará **3** , lo que significa que si tomamos el LIS para el **artículo [j]** y le agregamos el **artículo [i]** , obtendremos un mejor LIS { 3,4,6} que antes {3,6}. Entonces configuramos **dp [i]** = **dp [j]** + **1** . Luego incrementamos **j** .

```
           3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  2  |  3  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
                       j           i
```

Para **i** = **4** y **j** = **2** , el **elemento [i]** es mayor que el **elemento [j]** . Pero para este caso, si establecemos **dp [i]** = **dp [j] + 1** , obtendremos **2** , que es {-1,6} no es el mejor {3,4,6} que podemos hacer usando el **Elemento [i]** . Así que descartamos este. Agregaremos una condición a nuestra estrategia, que es:

```
if Item[i]>Item[j] and dp[i]<dp[j] + 1
    dp[i] := dp[j] + 1
```

Incrementamos **j** en **1** . Siguiendo esta estrategia, si completamos nuestra matriz **dp** , se verá así:

```
           3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  2  |  3  |  3  |  4  |
+-------+-----+-----+-----+-----+-----+-----+-----+
                                         j     i
```

Ahora recorreremos esta matriz y descubriremos el valor máximo, que será la longitud del LIS. Nuestro pseudo-código será:

```
Procedure LISLength(Item):
n := Item.length
dp[] := new Array(n)
for i from 0  to n
```

```
    dp[i] := 1
end for
for i from 1 to n
    for j from 0 to i
        if Item[i]>Item[j] and dp[i]<dp[j] + 1
            dp[i] := dp[j] + 1
        end if
    end for
end for
l := -1
for i from 0 to n
    l := max(l, dp[i])
end for
Return l
```

La complejidad del tiempo de este algoritmo es `O(n²)` donde **n** es la longitud de la secuencia.

Para descubrir la secuencia original, necesitamos iterar hacia atrás y hacerla coincidir con nuestra longitud. El procedimiento es:

```
Procedure LIS(Item, dp, maxLength):
i := Item.length
while dp[i] is not equal to maxLength
    i := i - 1
end while
s = new Stack()
s.push(Item[i])
maxLength := maxLength - 1
current := Item[i]
while maxLength is not equal to 0
    i := i-1
    if dp[i] := maxLength and Item[i] < current
        current := Item[i]
        s.push(current)
        maxLength := maxLength - 1
    end if
end while
while s is not empty
    x := s.pop
    Print(s)
end while
```

La complejidad temporal de este algoritmo es: `O(n)` .

Secuencia palindrómica más larga

Dada una cadena, ¿cuál es la subsecuencia palindrómica (LPS) más larga? Tomemos una cuerda **agbdba** . El LPS de esta cuerda es **abdba** de longitud **5** . Recuerde, ya que estamos buscando una *subsecuencia* , los caracteres no necesitan ser continuos en la cadena original. La *subcadena* palindrómica más larga de la secuencia sería **bdb** de longitud **3** . Pero nos concentraremos en la *subsecuencia* aquí. Vamos a utilizar la programación dinámica para resolver este problema.

Al principio, tomaremos una matriz 2D de la misma dimensión de nuestra secuencia original. Para nuestro ejemplo: `S = "agbdba"` , tomaremos **dp [6] [6]** array. Aquí, **dp [i] [j]** representa la longitud

del LPS que podemos hacer si consideramos los caracteres de **s [i]** a **s [j]** . Por ejemplo. si nuestra cadena fuera **aa** , **dp [0] [1]** almacenaría **2** . Ahora consideraremos diferentes longitudes de nuestra cadena y descubriremos la longitud más larga posible que podamos hacer con ella.

Longitud = 1 :
Aquí, estamos considerando solo **1** personaje a la vez. Entonces, si tuviéramos una cadena de longitud **1** , ¿cuál es el LPS que podemos tener? Por supuesto que la respuesta es **1** . ¿Cómo almacenarlo? **dp [i] [j]** donde **i** es igual a **j** representa una cadena de longitud **1** . Así que vamos a configurar **dp [0] [0]** , **dp [1] [1]** , **dp [2] [2]** , **dp [3] [3]** , **dp [4] [4]** , **dp [5] [5]** a **1** . Nuestra matriz se verá como:

```
+---+---+---+---+---+---+---+
|   | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
| 0 | 1 |   |   |   |   |   |
+---+---+---+---+---+---+---+
| 1 |   | 1 |   |   |   |   |
+---+---+---+---+---+---+---+
| 2 |   |   | 1 |   |   |   |
+---+---+---+---+---+---+---+
| 3 |   |   |   | 1 |   |   |
+---+---+---+---+---+---+---+
| 4 |   |   |   |   | 1 |   |
+---+---+---+---+---+---+---+
| 5 |   |   |   |   |   | 1 |
+---+---+---+---+---+---+---+
```

Longitud = 2 :
Esta vez consideraremos cadenas de longitud **2** . Ahora, considerando cadenas de longitud **2** , la longitud máxima de LPS puede ser **2** si y solo si los dos caracteres de la cadena son iguales. Entonces nuestra estrategia será:

```
j := i + Length - 1
if s[i] is equal to s[j]
    dp[i][j] := 2
else
    dp[i][j] := 1
```

Si llenamos nuestra matriz siguiendo la estrategia para **Length** = **2** , obtendremos:

```
+---+---+---+---+---+---+---+
|   | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
| 0 | 1 | 1 |   |   |   |   |
+---+---+---+---+---+---+---+
| 1 |   | 1 | 1 |   |   |   |
+---+---+---+---+---+---+---+
| 2 |   |   | 1 | 1 |   |   |
+---+---+---+---+---+---+---+
| 3 |   |   |   | 1 | 1 |   |
+---+---+---+---+---+---+---+
| 4 |   |   |   |   | 1 | 1 |
+---+---+---+---+---+---+---+
| 5 |   |   |   |   |   | 1 |
```

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

Longitud = 3 :
Ahora estamos viendo **3** caracteres a la vez para nuestra cadena original. A partir de ahora, el LPS que podemos hacer a partir de nuestra cadena estará determinado por:

- Si el primer y el último carácter coinciden, tendremos al menos **2** elementos de los que podemos hacer el LPS + si excluimos el primer y el último carácter, lo que sea mejor que podamos hacer de la cadena restante.
- Si el primer y el último carácter no coinciden, el LPS que podemos crear provendrá de la exclusión del primer carácter o del último, que ya hemos calculado.

Para veranear,

```
j := i + Length - 1
if s[i] is equal to s[j]
    dp[i][j] := 2 + dp[i+1][j-1]
else
    dp[i][j] := max(dp[i+1][j], dp[i][j-1])
end if
```

Si llenamos la matriz **dp** para **Longitud** = **3** a **Longitud** = **6** , obtendremos:

```
+---+---+---+---+---+---+---+
|   | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
| 0 | 1 | 1 | 1 | 1 | 3 | 5 |
+---+---+---+---+---+---+---+
| 1 |   | 1 | 1 | 1 | 3 | 3 |
+---+---+---+---+---+---+---+
| 2 |   |   | 1 | 1 | 3 | 3 |
+---+---+---+---+---+---+---+
| 3 |   |   |   | 1 | 1 | 1 |
+---+---+---+---+---+---+---+
| 4 |   |   |   |   | 1 | 1 |
+---+---+---+---+---+---+---+
| 5 |   |   |   |   |   | 1 |
+---+---+---+---+---+---+---+
```

Esta es nuestra matriz de **dp** requerida y **dp [0] [5]** contendrá la longitud del LPS. Nuestro procedimiento se verá como:

```
Procedure LPSLength(S):
n := S.length
dp[n][n]
for i from 0 to n
    dp[i][i] := 1
end for
for i from 0 to (n-2)
    if S[i] := S[i+1]
        dp[i][i+1] := 2
    else
        dp[i][i+1] := 1
    end if
```

```
end for
Length := 3
while Length <= n
    for i from 0 to (n - Length)
        j := i + Length - 1
        if S[i] is equal to s[j]
            dp[i][j] := 2 + dp[i+1][j-1]
        else
            dp[i][j] := max(dp[i+1][j], dp[i][j-1])
        end if
    end for
Length := Length + 1
end while
Return dp[0][n-1]
```

La complejidad del tiempo de este algoritmo es `O(n²)` , donde **n** es la longitud de nuestra cadena dada. El problema de la subsecuencia palindrómica más larga está estrechamente relacionado con la subsecuencia común más larga. Si tomamos la segunda cadena como la inversa de la primera y calculamos la longitud e imprimimos el resultado, esa será la subsecuencia palindrómica más larga de la cadena dada. La complejidad de ese algoritmo también es `O(n²)` .

Lea Subsecuencias relacionadas con los algoritmos en línea: https://riptutorial.com/es/dynamic-programming/topic/7969/subsecuencias-relacionadas-con-los-algoritmos

Capítulo 9: 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,

```
Table[i][j] := d(i, j) + min(Table[i-1][j], Table[i-1][j-1], 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/dynamic-programming/topic/7967/time-warping-dinamico

Creditos

S. No	Capítulos	Contributors
1	Empezando con la programación dinámica.	Bakhtiar Hasan, Community
2	Corte de varilla	metahost, Vishwas
3	Matriz de multiplicación de la cadena	Bakhtiar Hasan
4	Moneda que cambia el problema	Bakhtiar Hasan
5	Problema de mochila	Bakhtiar Hasan
6	Resolviendo problemas de gráficas usando programación dinámica	Bakhtiar Hasan
7	Selección de actividad ponderada	Bakhtiar Hasan
8	Subsecuencias relacionadas con los algoritmos	Bakhtiar Hasan
9	Time Warping dinámico	Bakhtiar Hasan