Programación Dinámica

Práctica de Algorítmica

Adrián Peláez Vegas
Alejandro Rodríguez Muñoz
Jose Antonio Ruiz Millán
Julio Antonio Fresneda García

Ejercicio 1: Producto máximo con suma fija.

Para resolver un problema con Programación Dinámica, necesitamos que éste tenga naturaleza n-etápica, que verifique el Principio de Optimalidad de Bellman. Además, se debe poder plantear una ecuación de recurrencia que represente la forma de ir logrando etapa por etapa la solución optimal, hasta encontrar la solución final.

Descripción

Dado *C*, un número no negativo, ¿cuál es el mayor valor que podemos obtener multiplicando *n* números naturales que sumen *C*?

Más técnicamente, buscamos:

Maximizar $x_1 \times x_2 \times \ldots \times x_k$

Sujeto a $x_1 + x_2 + \ldots + x_k = C.$

Vamos a desarrollar las características necesarias para resolver el problema por Programación Dinámica:

Naturaleza n-etápica

Sabemos que es un problema n-etápico, ya que tenemos:

- $MAX(k, C)$ como el máximo producto alcanzable.
- x como el valor de la primera subdivisión, es decir, del número que, al sumarse con las $(n-1)$ partes restantes, resulte C
- $(C - x)$ como el valor de la suma de las $(n-1)$ partes restantes

Verificación del POB

Vamos a verificar el POB mediante reducción al absurdo.

Sea $x_1, x_2, x_3, \ldots x_k$ enteros que suman C y cuyo producto es máximo (solución óptima).

Se ha obtenido x_1, entero óptimo, por lo que $x_2, x_3, \ldots x_k$ debe ser un subconjunto óptimo. De no ser óptimo:

- Sea $y_2, y_3, \ldots y_k$ una solución óptima, y por lo tanto, mejor que $x_2, x_3, \ldots x_k$
- Entonces $y_2 \times y_3 \times \ldots \times y_k$ > $x_2 \times x_3 \times \ldots \times x_k$
- Por lo tanto, $x_1 \times y_2 \times y_3 \times \ldots \times y_k$ > $x_1 \times x_2 \times x_3 \times \ldots \times x_k$

Esto es una contradicción, puesto que hemos partido de la hipótesis de que $x_1, x_2, x_3, \ldots x_k$ es solución óptima.

Por lo tanto, el POB se puede verificar y aplicar a este problema.

Solución

La ecuación que describe el problema es:

$$MAX(k,C) = Máximo_{1 \leq x \leq C}\{xMAX(k-1, C-x)\}, \text{ siendo } k \geq 1 \text{y } C > 0$$

Vamos a demostrar que esto es cierto (y calcular la x máxima entre 1 y C) mediante inducción.

Primero, sabemos que:

- $MAX(k,0) = 0 \quad \forall k \geq 0$
- $MAX(1,C) = C \quad si\ C > 0$ (caso más pequeño, k=1)

Vamos a resolver, como caso base, para k=2:

$$MAX(2,C) = Máximo_{1\leq x\leq C}\{xMAX(1,C-x)\}$$

Como sabemos que $MAX(1,C) = C$:

$$MAX(2,C) = Máximo_{1\leq x\leq C}\{x(C-x)\}$$

Vamos a sacar ese máximo derivando la función $f(x) = x(C-x)$ e igualándola a 0.

$$f'(x) = C - 2x$$
$$C - 2x = 0, \text{ por lo que } x = \frac{C}{2}$$

Sustituyendo, $MAX(2,C) = \frac{C}{2}MAX(1,\frac{C}{2})$, es decir,

$$MAX(2,C) = \frac{C}{2}\cdot\frac{C}{2} =$$

$$(\frac{C}{2})^2, \text{ con soluciones } \{\frac{C}{2},\frac{C}{2}\}$$

Vamos a probar con k=3:

$$MAX(3,C) = Máximo_{1<x<C}\{xMAX(2,C-x)\}$$

Como sabemos que $MAX(2,C) = (\frac{C}{2})^2$,

$$MAX(2,C-x) = (\frac{C-x}{2})^2.$$

Por lo que $MAX(3,C) = Máximo_{1\leq x\leq C}\{x(\frac{C-x}{2})^2\}$

Vamos a sacar ese máximo derivando la función $f(x) = x(\frac{C-x}{2})^2$ e igualándola a 0.

$$f'(x) = \frac{(C-x)^2}{2} - x(C-x)$$

$$\frac{(C-x)^2}{2} - x(C-x) = 0;\ C - x = 2x;\ \text{por lo que } x = \frac{C}{3}$$

Sustituyendo, $MAX(3,C) = \frac{C}{3} \cdot (\frac{C-\frac{C}{3}}{2})^2 = \frac{C}{3} \cdot \frac{2C}{3} = (\frac{C}{3})^3$

Las soluciones son $\{\frac{C}{3}, \frac{C}{3}, \frac{C}{3}\}$, ya que $MAX(3,C) = \frac{C}{3} \cdot MAX(2, C - \frac{C}{3})$, y $MAX(2, C - \frac{C}{3})$ es bi-etápico, con soluciones $\{\frac{C}{3}, \frac{C}{3}\}$

Los dos ejemplos anteriores nos llevan a pensar el máximo valor alcanzable del problema cuando k=i sería $MAX(i,C) = (C/i)^i$, con soluciones $\{\frac{C}{i}, \frac{C}{i}, \frac{C}{i}, ..., \frac{C}{i}\}$

Es decir, planteamos la siguiente ***hipótesis***:

$$MAX(i,C) = Máximo_{1 \leq x \leq C}\{xMAX(i-1, C-x)\} = Máximo_{1 \leq x \leq C}\{x.\frac{(C-x)^{i-1}}{i-1}\}$$

Derivando la función entre corchetes e igualando a 0, comprobamos que el *x* máximo es $\frac{C}{i}$

Para demostrar esto, vamos a usar inducción:

Suponiendo la anterior hipótesis como cierta, vamos a ver si se cumple para k=i+1:

De acuerdo con el POB, tenemos que:

$$MAX(i+1,C) = Máximo_{1 \leq x \leq C}\{xMAX(i, C-x)\} = Máximo_{1 \leq x \leq C}\{x \cdot \frac{(C-x)^i}{i}\}$$

Derivando e igualando a 0 la función entre corchetes, llegamos a que $x = \frac{C}{i+1}$, siendo el máximo valor del producto,

$$MAX(i+1,C) = (\frac{c}{i+1})^{i+1}$$

Vemos que el resultado tiene sentido, por lo que la hipótesis se puede tomar como cierta.

Una vez demostrado por inducción, podemos llegar a la conclusión de que si tomamos como soluciones $\{\frac{C}{n}, \frac{C}{n}, \frac{C}{n}, ..., \frac{C}{n}\}$ alcanzamos el valor máximo, el cual es $(\frac{C}{n})^n$.

Ecuación recurrente:

La ecuación que resolvería este problema, es:

$$MAX(n, C) = \frac{C}{n} \cdot MAX(n-1, C-\frac{C}{n})$$

Un ejemplo de uso: Producto máximo de tres números cuya suma resulte 9.

Anticipadamente, podemos saber el resultado con la fórmula $(\frac{C}{n})^n$: $(\frac{9}{3})^3 = 27$

Usando la ecuación recurrente:

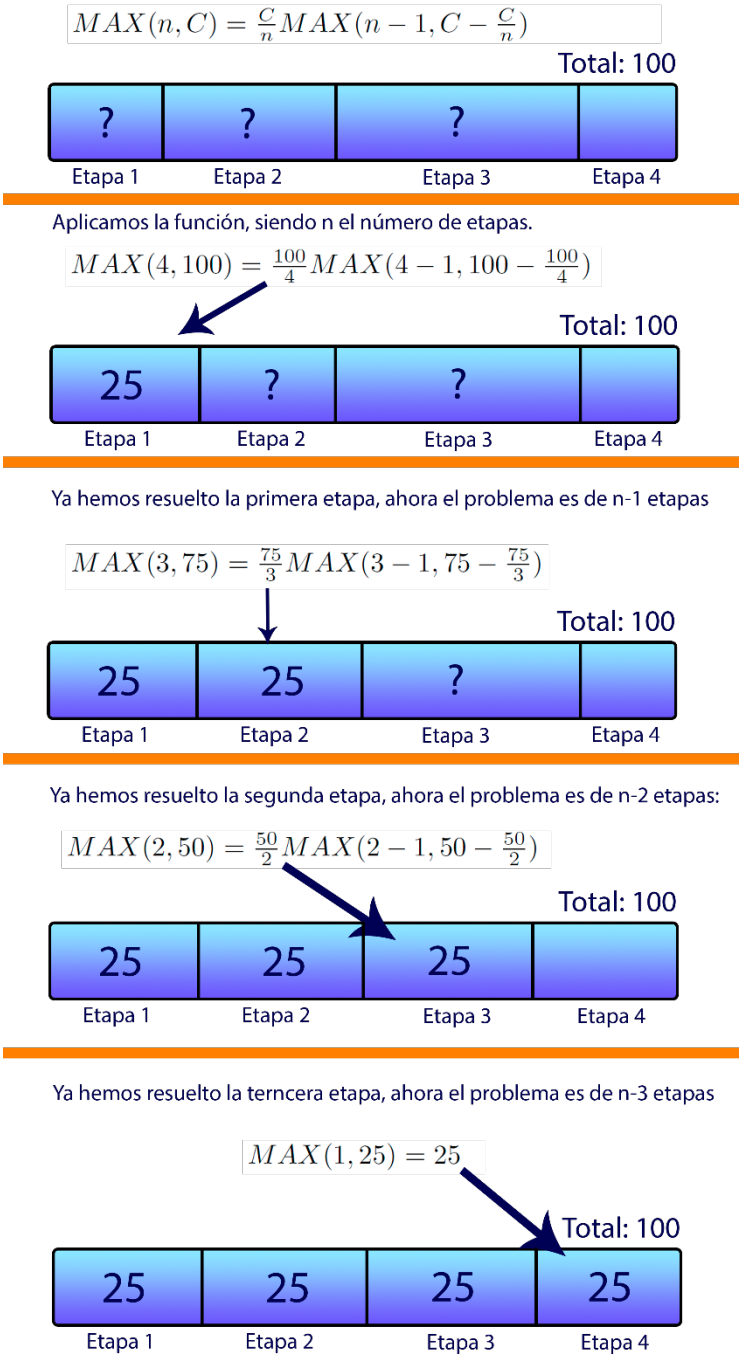
$$MAX(3,9) = \frac{9}{3} \cdot MAX(2,6) = \frac{9}{3} \cdot \frac{6}{2} \cdot MAX(1,3) = \frac{9}{3} \cdot \frac{6}{2} \cdot 3 = 27$$

Como vemos, el resultado es correcto.

Ejemplo visual

Vamos a comprobar visualmente cómo se resuelve el problema mediante PD para poder apreciar la naturaleza de “etapa a etapa”. En este ejemplo, buscaremos 4 números que sumen 100 y cuyo producto sea máximo. Como vemos, en cada recurrencia se resuelve una etapa, es decir, uno de los 4 números que son solución.

Se ve claramente como sigue el POB, en el que en un problema de n etapas, encuentra la solución óptima para una etapa y se vuelve a resolver el problema para las n-1 etapas restantes.



Como vemos, la solución es el producto de los cuatro números resultantes de resolver cada etapa: 25 x 25 x 25 x 25.

Pseudo-código del algoritmo (eficiencia de O(nM)

```
func MAX(n, M) return natural
  for k in 0..n loop T(k,0) ← 0
  for C in 1..M loop T(1,C) ← C
  for k in 2..n loop
    for C in 1..M loop
      T(k,C) ← 0
      for x in 1..C loop
        T(k,C) ← máximo{T(k,C), x × T(k−1,C−x)}
  return T(n,M)
```

Código implementado en C++

```cpp
#include <iostream>
using namespace std;


int MAX( int n, int C ){
  int T[n+1][C+1];
  for( int k=0; k<=n; k++ ){
    T[k][0] = 0;
  }
  for( int k=1; k<=C; k++ ){
    T[1][k] = k;
  }
  for( int k=2; k<=n; k++ ){
    for( int c=1; c<=C; c++ ){
      T[k][c] = 0;
      for( int x=1; x<=c; x++ ){
        if( x*T[k-1][c-x] > T[k][c] ) T[k][c] = x*T[k-1][c-x];
      }
    }
  }

  return T[n][C];
}
```

Ejecución del algoritmo implementado

Ponemos a prueba el código anterior con los siguientes parámetros:

Máximo producto de 3 números que sumen 9.

```
25  int main(){
26    int sol = MAX(3,9);
27    cout << "El producto máximo de 3 números que suman 9 es: " << sol << endl;
28  }
```

El resultado de la ejecución es la siguiente.

```
julioxxx@julio-pc:~/Desktop$ ./alg
El producto máximo de 3 números que suman 9 es: 27
```

Como vemos, el resultado es correcto.

Ejercicio 2: La conexión de “El Pedregal”.

Descripción:

- En un archipiélago, con multitud de pequeñas islas cercanas, hay puentes que unen ciertos pares de islas entre sí. Para cada puente (que puede ser de dirección única), además de saber la isla de origen y la isla de destino, se conoce su anchura (número entero mayor que 0).
- La anchura de un camino, formado por una sucesión de puentes, es la anchura mínima de las anchuras de todos los puentes que lo forman.
- Para cada par de islas se desea saber cuál es el camino de anchura máxima que las une (siempre que exista alguno).

Este ejercicio cumple las condiciones para poder solucionarse utilizando programación dinámica ya que cumple las siguientes características.

Naturaleza n-etápica:

Representaremos las diferentes ciudades y sus comunicaciones como un grafo *G* donde las ciudades serían los nodos/vértices y los puentes las aristas que unen los distintos nodos.

Para encontrar el camino más ancho desde un vértice *i* a otro *j* en un grafo *G,* veríamos que vértice debe ser el segundo, cual el tercero, etc. hasta alcanzar el *j*.

Una sucesión optimal de decisiones proporcionará entonces el camino de anchura máxima.

Por lo tanto, este problema tiene una clara naturaleza n-etápica.

Verificación del POB:

Sea $i, i_1, \ldots, i_k, j$ el camino con máxima anchura desde i hasta j, comenzando con el vértice inicial *i*, se ha tomado decisión de ir al vértice i_1.

Como resultado, ahora el estado del problema está definido por el vértice i_1, y lo que se necesita es encontrar un camino desde i_1 hasta *j*.

Está claro que la sucesión $i, i_1, i_2, \ldots, i_k, j$ debe constituir un camino con anchura máxima entre i_1 y *j.* Si no:

- Sea $i_1, r_1, \ldots, r_q, j$ un camino más corto entre i_1 y *j*, entonces $i, i_1, r_1, r_2, \ldots r_q, j$ es un camino entre *i* y *j* que es más corto que el camino $i, i_1, i_2, i_3, \ldots, i_k, j$.
- Como eso **es una contradicción, se verifica el POB** y también puede aplicarse a este problema.

Planteamiento de una recurrencia:

Denominaremos A(i, j) la anchura del puente que va de la isla *i* a la isla *j*. Si no hay puente en esa dirección entonces A(i, j) = ∞. Obsérvese que conviene A(i, i) = 0 puesto que para ir de una isla a ella misma no debe existir ninguna restricción.

Definimos, de manera recursiva, la siguiente función:

- Max A(i,j,k) = máxima anchura de los caminos que van desde la isla *i* hasta la isla *j* , pudiendo pasar por las islas {1, …, k}.
- Max A(i,j,0) = A(i,j).
- Max A(i,j,k) = Max{Max A(i,j,k-1), Min{Max A(i,k,k-1), Max A(k,j,k-1)}} si k >0.

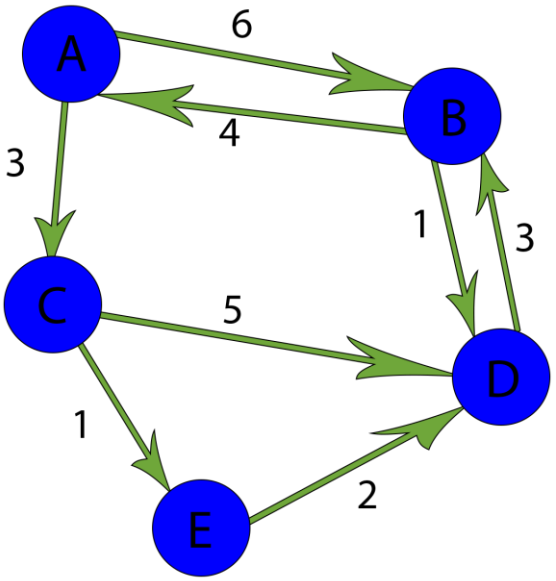
Así podemos diseñar un algoritmo como el siguiente, que calcula los valores de una tabla M(1..n, 1..n), de manera que al terminar: M(i,j) = Max A(i,j,n) ∀i, j.

```
proc MÁXIMA_ANCHURA(A, M)
  M(1..n, 1..n) ← A(1..n, 1..n)
  for k in 1..n loop
    for i in 1..n loop
      for j in 1..n loop
        if M(i,k) < M(k,j) then aux ← M(i,k)
        else aux ← M(k,j)
        if M(i,j) < aux then M(i,j) ← aux
```

Implementación C++:

```
#define N 5

int main(int argc, char const *argv[]) {

  int M[N][N];
  int aux;

  inicializarMatriz(M);

  for(int k = 0; k < N; k++){
    for(int i = 0; i < N; i++){
      for(int j = 0; j < N; j++){
        aux = (M[i][k] < M[k][j])?M[i][k]:M[k][j];
        if((M[i][j] < aux) && (i != j)) M[i][j] = aux;
      }
    }
  }

  for(int i = 0; i < N; i++){
    for(int j = 0; j < N; j++){
      cout << M[i][j] << " ";
    }
    cout << endl;
  }

  return 0;
}
```

Caso de ejemplo (enfoque adelantado):



Dado este grafo como ejemplo, vamos a ver el resultado que devuelve el programa para poder verificar los resultados.

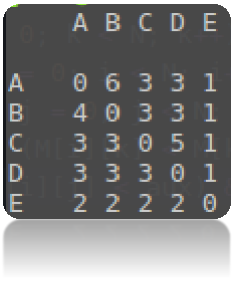
Matriz Inicial:

	A	B	C	D	E
A	0	6	3	∞	∞
B	4	0	∞	1	∞
C	∞	∞	0	5	1
D	∞	3	∞	0	∞
E	∞	∞	∞	2	0

Resultado programa:

	A	B	C	D	E
A	0	6	3	3	1
B	4	0	3	3	1
C	3	3	0	5	1
D	3	3	3	0	1
E	2	2	2	2	0

Podemos observar en el resultado que da lo esperado y efectivamente calcula los caminos más anchos entre todos los posibles caminos.



Ejercicio 3: Regalos por la fama.

Descripción:

Mª Gabriela y Eduardo Fernando han recibido n regalos por su estupendo trabajo en una serie de televisión de reconocida fama. Cada regalo viene en una caja destinada a ambos. Como no tienen suficiente tiempo para desempaquetar y mirar que es cada cosa, han decidido utilizar el siguiente criterio: Cada uno debe quedarse con la misma cantidad de peso, para ello cuentan los pesos de cada una de las cajas $P_1, \dots, P_n$(números enteros positivos). Al cabo de un rato aun no han conseguido hacer el reparto que desean.

Nuestro objetivo en este problema será diseñar un algoritmo que nos diga si dado el **número de regalos que tenemos (n)** y el **peso de cada uno de ellos($\mathbf{P_1}, \dots, \mathbf{P_n}$)**, nos diga si es posible dividir este conjunto en dos subconjuntos de igual peso, sin dejar ningún regalo fuera.

Naturaleza n-etápica:

Que un conjunto de regalos sea subdivisible en dos subconjuntos de igual peso dependerá del número de regalos que tengamos y del peso de estos. Estas características se verán reflejadas en una función booleana S(m,k), la cual nos dice si esta subdivisión es posible o no. En esta función la m representa el peso del conjunto de regalos que consideramos, y la k el número de regalos que consideramos. Concretamente, esta función nos dice si podemos hacer un subconjunto de peso m con los k regalos que tenemos. Por lo tanto, siendo **D** la suma del peso de los n regalos que tenemos, la solución consistirá en ver si se puede hacer un subconjunto de peso **D/2** teniendo en cuenta todos los regalos que tenemos, lo cual sería equivalente a averiguar el valor de **S(D/2 , n)**.

Dicho esto, podemos ver la solución al problema como una sucesión de decisiones, en la que en cada etapa se verá la factibilidad de crear un subconjunto con el peso y los regalos actuales, hasta llegar al objetivo.

Primero tomaríamos una decisión para el subconjunto de regalos de k = 1, con peso m = 1 (etapa 1), después tomaríamos otra decisión sobre el subconjunto de regalos de K = 1, con peso m = 2 (etapa 2). Seguiremos tomando decisiones incrementando en una unidad el peso del subconjunto que queremos realizar (m), hasta que lleguemos al peso de nuestro objetivo: m = D/2. Una vez tomada la decisión para este peso, incrementaríamos la k y volveríamos a tomar las decisiones para todo el abanico de pesos hasta que volvamos a llegar a nuestro peso límite. Este proceso seguiría hasta llegar a la toma de decisión del conjunto de n regalos para un peso de D/2, que es el objetivo al que intentamos llegar.

Por lo tanto, este problema tiene una clara naturaleza n-etápica.

Verificación del POB:

Partiremos del enunciado del principio de optimalidad de Bellman, el cual nos dice que para que un problema satisfaga dicho principio, si en una sucesión óptima de decisiones, cada subsucesión es a su vez óptima. Es decir, si miramos una solución parcial de la solución general, esta debe ser una solución óptima al subproblema asociado.

Entonces, si tenemos una solución óptima para nuestro problema (S(m,k) = true), las soluciones parciales a los subproblemas también tienen que ser óptimas para que se cumpla el principio de optimalidad. Concretamente en programación dinámica, para que una solución sea óptima la solución parcial al subproblema que le precede (subproblema de tamaño -1 unidad) también tiene que serlo.

Por lo tanto, para demostrar que se verifica el P.O.B., partiremos de una solución óptima al problema, cuya solución parcial al subproblema precedente no es óptima, lo cual no es admisible para el cumplimiento del principio, y demostraremos que dicha solución parcial nos conduce a una solución general **no óptima**, lo cual sería contradictorio y verificaría el P.O.B. para nuestro problema.

- ➔ Solución general óptima: $\boldsymbol{S(m,k) = true}$
- ➔ Solución parcial no óptima para el subproblema precedente: $\boldsymbol{S(m,k-1)\ V\ S(m-P_k,k-1) = false}$

A partir de esta solución parcial calculamos la solución para la siguiente etapa:

$$\boldsymbol{S(m,k) = S(m,k-1)\ V\ S(m-P_k,k-1) = false}$$

Llegamos así a la conclusión de que $\boldsymbol{S(m,k) = false}$, habiendo partido de que $\boldsymbol{S(m,k) = true}$, por lo tanto es contradictorio, lo cual demuestra que toda solución parcial de una solución óptima es óptima en nuestro problema, o lo que es lo mismo, **se cumple el principio de optimalidad de Bellman**.

Planteamiento de una recurrencia:

Como se dijo anteriormente, llamamos S(m,k) a la función que nos dice si es factible obtener un subconjunto de peso m eligiendo determinados regalos de los k que disponemos.

Ahora pasaremos a definir dicha función para posteriormente poder implementar un algoritmo basado Programación Dinámica que resuelva nuestro problema:

$$S(m,k) = \begin{cases} true & si\ m = 0 \\ false & si\ k = 0\ y\ m > 0 \\ S(m,k-1)\ V\ S(m-P_k,k-1) & si\ m \geq P_k\ y\ k > 0 \\ S(m,k-1) & si\ m < P_k\ y\ k > 0 \end{cases}$$

Una vez definida nuestra ecuación recurrente, podemos definir un algoritmo basándonos en ella.

El siguiente algoritmo encuentra el valor que buscamos. Irá guardando todos los valores obtenidos en una tabla con el mismo nombre que la función para cuando se precise recurrir a la recurrencia a la hora de calcular un nuevo valor, sacar este de la tabla en vez de volver a realizar la ejecución recurrente de la función, ya que este valor ya habrá sido previamente calculado. Además el algoritmo incluirá sentencias que dejen marcas que posteriormente permitan construir el conjunto de regalos de uno de ellos. Los regalos restantes formarán el otro subconjunto:

```
func HAY_REPARTO(P(1..n), S(0..D/2, 0..n)) return (Boolean × Conjunto)
  for k in 0..n loop S(0,k) ← True
  for m in 1..D/2 loop S(m,0) ← False
  for k in 1..n loop
    for m in 1..D/2 loop
      if m < P(k) then
        S(m,k) ← S(m,k-1)
        marca(m,k) ← False
      elsif S(m-P(k), k-1) then
        S(m,k) ← S(m-P(k), k-1)
        marca(m,k) ← True
      else
        S(m,k) ← S(m, k-1)
        marca(m,k) ← False
  end loop
  {Ahora calculamos la colección de regalos}
  if S(D/2, n) then
    i ← D/2
    j ← n
    C ← ∅
    while j > 0 loop
      if marca(i,j) then
        C ← C ∪ {j}
        i ← i - P(j)
        j ← j - 1
      else
        j ← j - 1
    else
      return (False, ∅)
```

El algoritmo es O(nD)

Implementación en c++:

```
P.push_back(2);
P.push_back(2);
P.push_back(3);
P.push_back(3);
//P.push_back(8);//FALSE
P.push_back(4);//TRUE

for(int k = 0; k <= n; k++){
  S[0][k] = true;
}
for(int m = 1; m < (D/2); m++){
  S[m][0] = false;
}
for(int k = 1; k <= n; k++){
  for(int m = 1; m <= (D/2); m++){
    if(m < P[k]){
      S[m][k] = S[m][k-1];
      marca[m][k] = false;
    }
    else if( ( S[m - P[k]][k-1] == true ) && ( ( (m - P[k]) >= 0) && ( (k-1) >= 0 ) ) ){
      S[m][k] = true;
      marca[m][k] = true;
    }
    else{
      S[m][k] = S[m][k-1];
    }
  }
}

if(S[D/2][n] == true) cout << "TRUE" << endl;
else cout << "FALSE" << endl;
```

Caso de ejemplo:

➔ Caso en el que tenemos un conjunto del cual no podemos hacer un reparto paritario:

Pesos de los regalos (P_i) :

2	2	3	3	8

$D = P_1 + P_2 + P_3 + P_4 + P_5 = 2 + 2 + 3 + 3 + 8 = 18$

Por lo tanto la solución a nuestro problema nos la dará S(D/2, n) -> S(9,5) .

La tabla resultante sería:

m \ K	0	1	2	3	4	5
0	1	1	1	1	1	1
1	0	0	0	0	0	0
2	0	1	1	1	1	1
3	0	0	0	1	1	1
4	0	0	1	1	1	1
5	0	0	0	1	1	1
6	0	0	0	0	1	1
7	0	0	0	1	1	1
8	0	0	0	0	1	1
9	0	0	0	0	0	0

S(9,5) = 0
El conjunto de regalos inicial no puede ser subdivido en dos subconjuntos paritarios.

Las celdas sombreadas son las celdas que el algoritma ha marcado para posteriormente definir los subconjuntos en el caso de que S(D/2, n) = 1.

➔ Caso en el que tenemos un conjunto del cual podemos hacer un reparto paritario:

Pesos de los regalos (P_i) :

2	2	3	3	4

$D = P_1 + P_2 + P_3 + P_4 + P_5 = 2 + 2 + 3 + 3 + 4 = 14$

Por lo tanto la solución a nuestro problema nos la dará S(D/2, n) -> S(7,5) .

La tabla resultante sería:

m \ K	0	1	2	3	4	5
0	1	1	1	1	1	1
1	0	0	0	0	0	0
2	0	1	1	1	1	1
3	0	0	0	1	1	1
4	0	0	1	1	1	1
5	0	0	0	1	1	1
6	0	0	0	0	1	1
7	0	0	0	1	1	1

S(7,5) = 1
El conjunto de regalos inicial puede ser subdivido en dos subconjuntos paritarios.

Una vez que sabemos que S(D/2, n) = 1, podemos afirmar que el conjunto inicial de regalos se puede dividir en dos del mismo peso. Ahora bastaría con seguir las marcas como el algoritmo indica y tendríamos también los regalos que componen dichos subconjuntos.

Ejercicio 4: El problema del turista en Manhattan

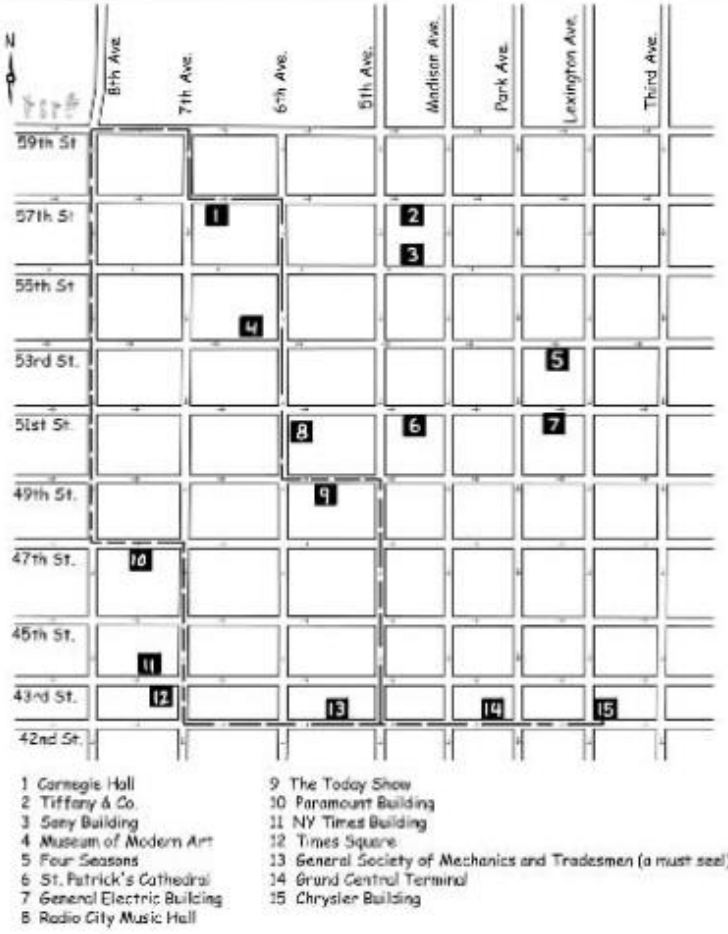
Descripción:

En la isla de Manhattan, cierto turista quiere aprovechar al máximo sus vacaciones visitando el mayor número de lugares turísticos, pero en dicho recorrido se encuentra con varias restricciones que lo limitan:

-Comienza en la parte noroeste del mapa (esquina superior izquierda).

-Solo se puede desplazar hacia el sur y hacia el este

Con estos requisitos podemos deducir que el final de su recorrido se haya en la parte más al sureste del mapa (esquina inferior derecha) y una vez en este punto el número de lugares visitados tiene que ser máximo.



Naturaleza n-etápica:

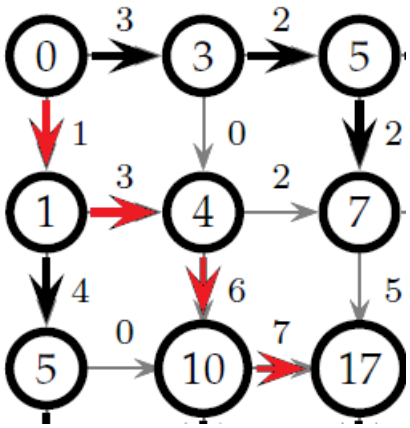
Para este problema el mapa de la ciudad se representará como un grafo donde cada arista representará una calle y cada vértice una intersección de calles. Cada una de las aristas tendrá un valor asociado que indicará el número de sitios turísticos que hay en esa calle. Cada vértice tendrá a su vez otro valor asociado que indicará el número de sitios turísticos visitados hasta el momento priorizando la ruta que maximice el número de estos.

En este problema se puede observar una clara naturaleza n-etápica. Ya que el cálculo de cada nuevo vértice debe hacerse una vez hallados sus predecesores. No se puede calcular un vértice aislado ni una solución sin antes haber pasado por cada una de las etapas previas para la construcción del grafo.

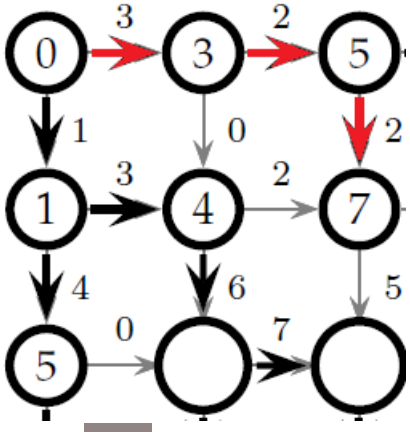
Verificación del POB:

El principio de optimalidad de Bellman se verifica si toda solución óptima a un problema está compuesta por soluciones óptimas de sus subproblemas. En este problema es fácil comprobar que no se cumple dicho principio con un ejemplo simple.

Tenemos un grafo 3X3 donde hemos hallado la solución óptima que es {0, 1, 4, 10, 17}



Pero si nos remontamos a una etapa anterior donde se estaba resolviendo un subproblema del problema actual podemos veremos que la solución cambia siendo {0, 3, 5, 7}



Por ello podemos decir que **este problema no verifica el POB**.

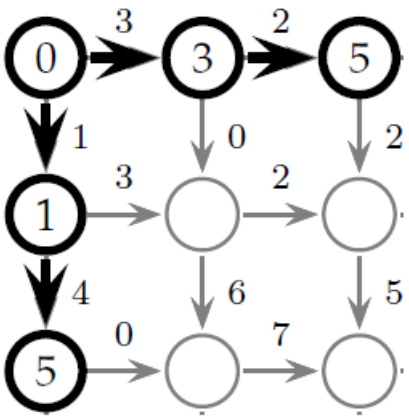
Planteamiento de una recurrencia:

En una primera fase del algoritmo se calcula los vértices en vertical y en horizontal del grafo. Después aplicamos la ecuación recurrente.

Llamaremos $S_{i,j}$ a un vértice que se haya en la posición i,j

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{Valor de la arista entre } (i-1,j) \text{ y } (i,j) \\ s_{i,j-1} + \text{Valor de la arista entre } (i,j-1) \text{ y } (i,j) \end{cases}$$

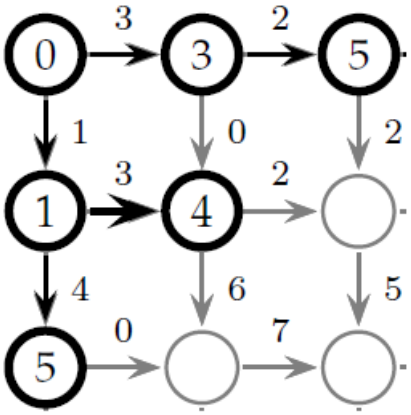
Para probar la ecuación podemos usar un ejemplo que determine el valor que tener el $S_{1,1}$ en el siguiente grafo (con la horizontal y vertical previamente calculada)



Utilizando la formula anterior tenemos

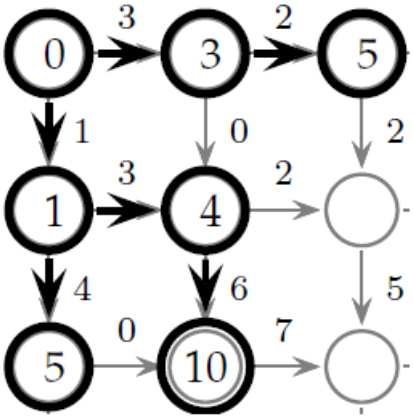
Etapa 1

$S_{1,1}$ = MAX((3+0) y (3+1))



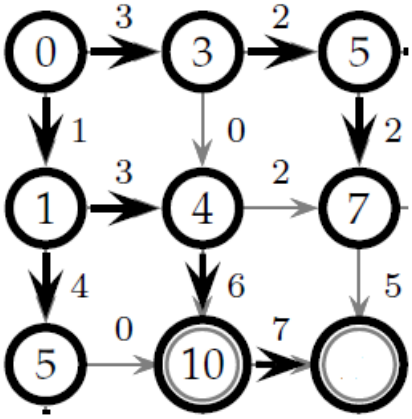
Etapa 2

$S_{2,1}$ = MAX((5+0) y (4+6))



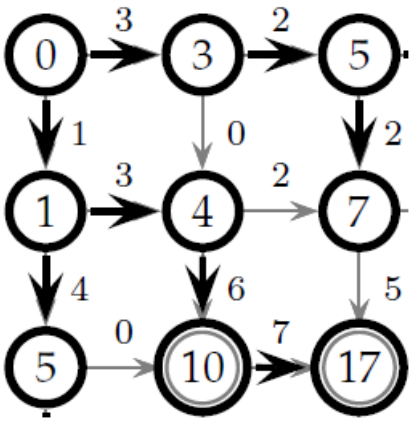
Etapa 3

$S_{1,2}$ = MAX((4+2) y (5+2))



Etapa 4

$S_{2,2}$ = MAX((10+7) y (7+5))



Código

Pseudocódigo

MANHATTANTOURIST($\downarrow w, \rightarrow w, n, m$)

```
1  s_{0,0} ← 0
2  for i ← 1 to n                                   } n
3      s_{i,0} ← s_{i-1,0} + ↓w_{i,0}
4  for j ← 1 to m                                   } m
5      s_{0,j} ← s_{0,j-1} + →w_{0,j}
6  for i ← 1 to n                                   } n*m
7      for j ← 1 to m
8          s_{i,j} ← max { s_{i-1,j} + ↓w_{i,j}
                         { s_{i,j-1} + →w_{i,j}
9  return s_{n,m}
```

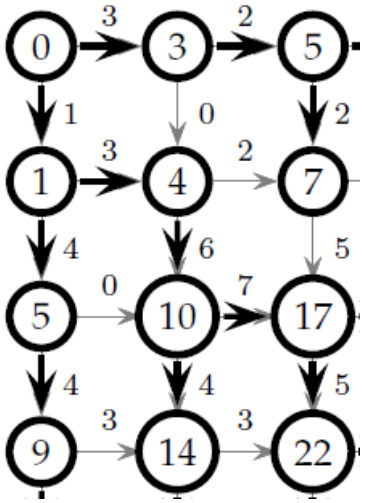
O(nm)

Implementación en Java

```
private static void turista(Vertice[][] ve, Arista[] a, int n, int m) {

    for (int i = 1; i < n; i++) {
        usadaI = devuelveArista(a, ve[0][i - 1], ve[0][i]);
        ve[0][i].setValor(ve[0][i - 1].getValor() + usadaI.getValor());
    }
    for (int j = 1; j < m; j++) {
        usadaJ = devuelveArista(a, ve[j - 1][0], ve[j][0]);
        ve[j][0].setValor(ve[j - 1][0].getValor() + usadaJ.getValor());
    }

    for (int i = 1; i < n; i++) {
        for (int j = 1; j < m; j++) {

            usadaI = devuelveArista(a, ve[j][i - 1], ve[j][i]);
            usadaJ = devuelveArista(a, ve[j - 1][i], ve[j][i]);
            if ((ve[j][i - 1].getValor() + usadaI.getValor()) > (ve[j - 1][i].getValor() + usadaJ.getValor())) {
                ve[j][i].setValor(ve[j][i - 1].getValor() + usadaI.getValor());
                anadeSolucion(ve[j][i], usadaI);

            } else {
                ve[j][i].setValor(ve[j - 1][i].getValor() + usadaJ.getValor());
               anadeSolucion(ve[j][i], usadaJ);
            }
        }
    }
}
```

Salida

Para el siguiente grafo



Esta sería la salida de nuestro algoritmo

```
Salida - TuristaManhattan (run)
run:
Desde 1,0 a 1,1 con valor 3 ---> Total: 4
Desde 1,1 a 2,1 con valor 6 ---> Total: 10
Desde 2,1 a 2,2 con valor 7 ---> Total: 17
Desde 2,2 a 3,2 con valor 5 ---> Total: 22
BUILD SUCCESSFUL (total time: 0 seconds)
```