



Programación dinámica

IIC2133

Selección de tareas con ganancias

Tenemos n tareas,

... cada una con una hora de inicio s_i y una hora de fin f_i

- definen el intervalo de tiempo $[s_i, f_i)$ de la tarea

Para realizar las tareas tenemos una única máquina

... que sólo puede realizar una tarea a la vez

- si los intervalos de tiempo de dos tareas se traslapan, entonces solo se puede realizar una de ellas

Además ...

Cada tarea produce una ganancia v_i si es realizada

El problema es ...

¿Cuáles tareas realizar de manera de maximizar la suma de las ganancias de las tareas realizadas?

P.ej., $[s_i, f_i), v_i$

$i=1$ $[0, 5), 2$

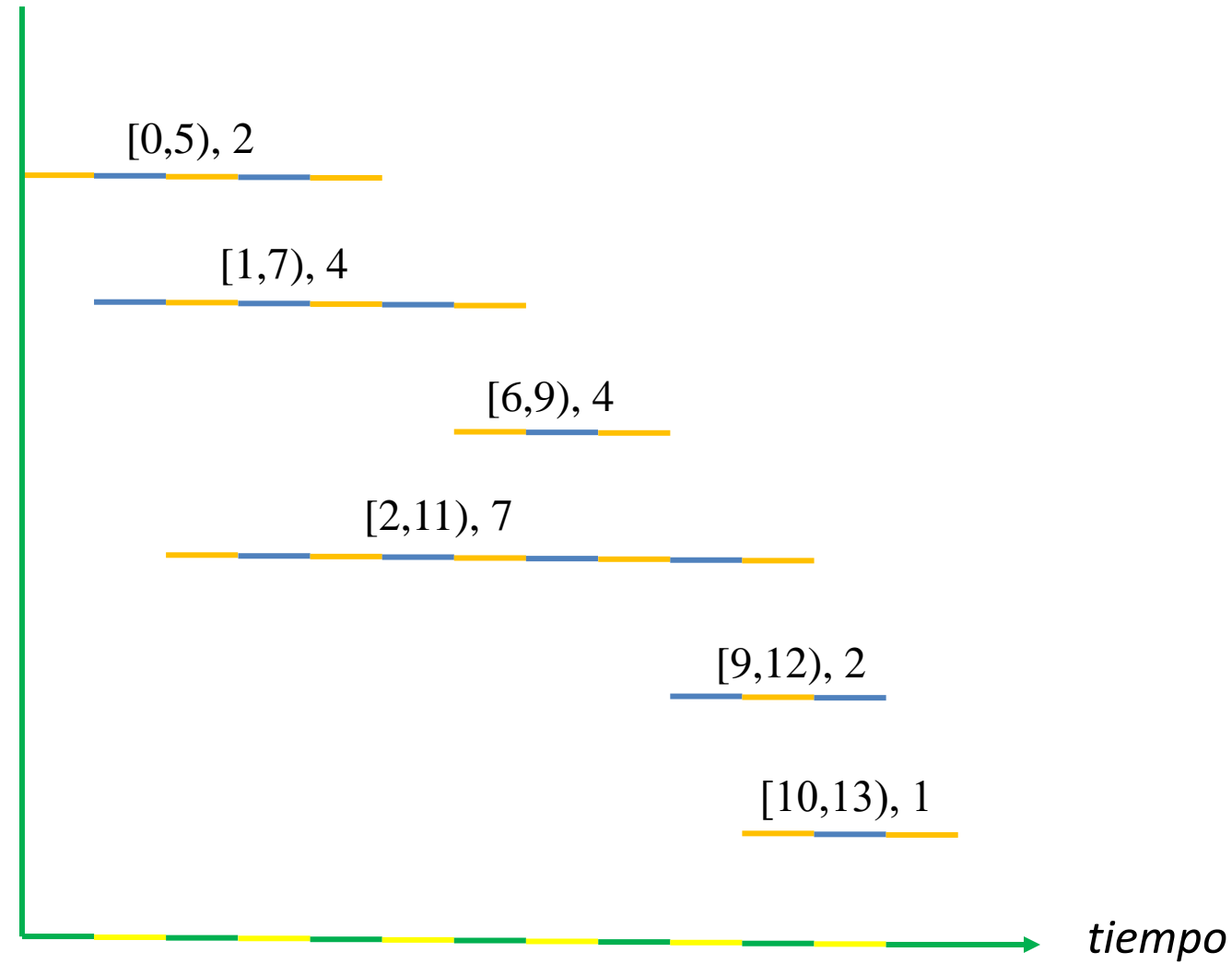
$i=2$ $[1, 7), 4$

$i=3$ $[6, 9), 4$

$i=4$ $[2, 11), 7$

$i=5$ $[9, 12), 2$

$i=6$ $[10, 13), 1$



Veamos primero la versión en que cada tarea produce una ganancia de 1 si es realizada —todas valen lo mismo

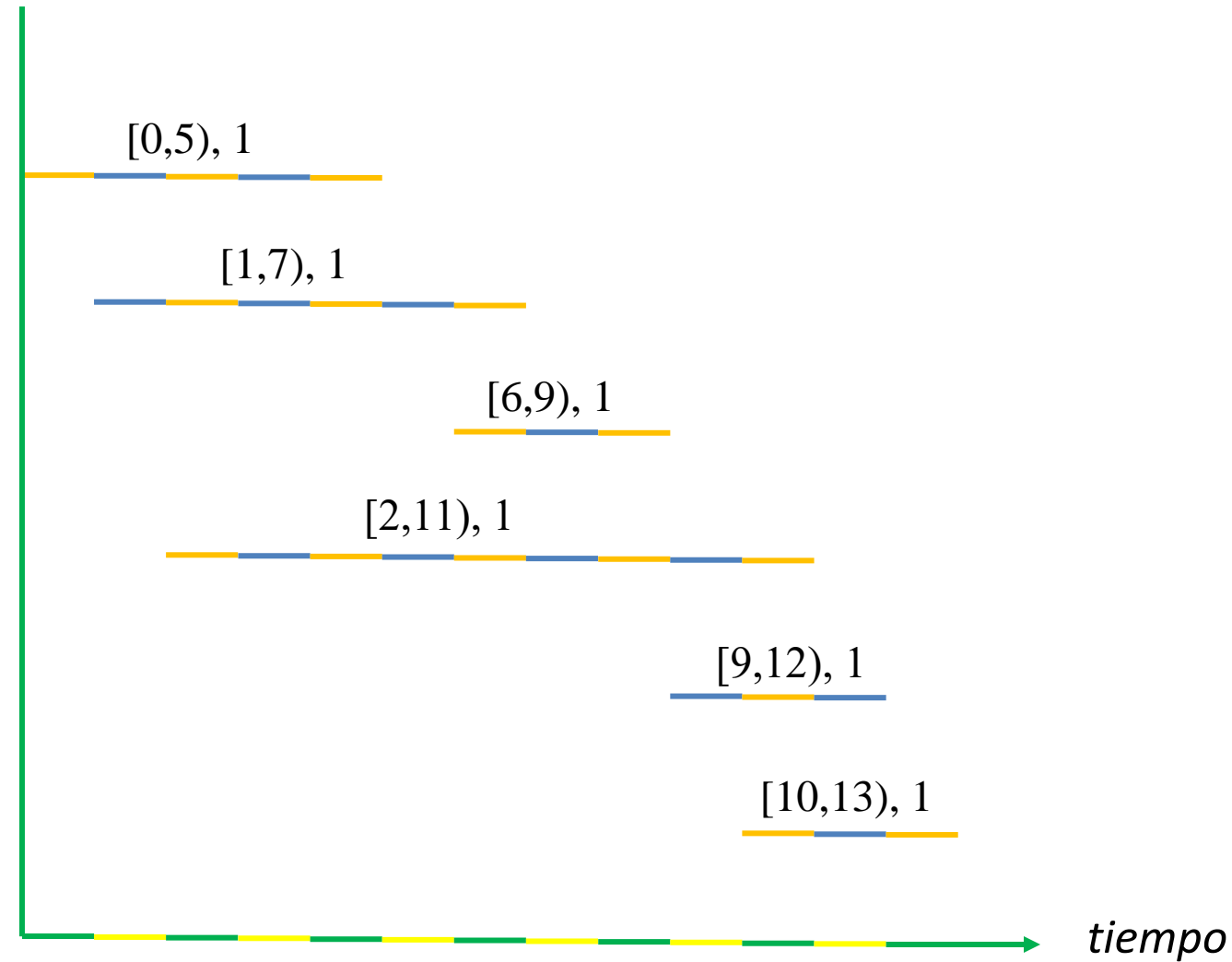
¿Cuáles tareas realizar de manera de maximizar la suma de las ganancias de las tareas realizadas?

La ganancia total va a ser igual al número de tareas realizadas:

- el problema consiste entonces en seleccionar un subconjunto de tamaño máximo de tareas mutuamente compatibles

P.ej., $[s_i, f_i), v_i$

| | |
|-------|---------------|
| $i=1$ | $[0, 5), 1$ |
| $i=2$ | $[1, 7), 1$ |
| $i=3$ | $[6, 9), 1$ |
| $i=4$ | $[2, 11), 1$ |
| $i=5$ | $[9, 12), 1$ |
| $i=6$ | $[10, 13), 1$ |



La próxima diapositiva muestra tres casos en que diferentes estrategias *codiciosas* **no** producen una solución óptima

(cada segmento de línea es el intervalo de tiempo de una tarea y el tiempo transcurre de izquierda a derecha):

- a) elegir primero la tarea que empieza más temprano
- b) elegir primero la tarea más corta
- c) elegir primero la tarea que tiene menos incompatibilidades con otras tareas

a)

b)

c)

Sin embargo, el problema sí puede resolverse mediante una *estrategia codiciosa* (cuando todas las tareas valen lo mismo):

elegir primero la tarea que termina más temprano

En el ej. de la diap. # 5, las tareas elegidas son las tareas 1, 3 y 5, y el valor de la solución es 3:

- la tarea 1 es la que termina más temprano, en $t = 5$
- (la tarea 2 es la segunda tarea que termina más temprano, pero es incompatible con la tarea 1 => la descartamos)
- la tarea 3 es la tercera tarea que termina más temprano, en $t = 9$, y es compatible con la tarea 1
- (la tarea 4 es incompatible con las tareas 1 y 3)
- la tarea 5 es compatible con las tareas 1 y 3
- (la tarea 6 es incompatible con la tarea 5)

Volvamos a la versión original: cada tarea produce una ganancia v_i si es realizada

¿Cuáles tareas realizar de manera de maximizar la suma de las ganancias de las tareas realizadas?

- ahora no importa el número de tareas realizadas

P.ej., $[s_i, f_i), v_i$

$i=1$ $[0, 5), 1$

$i=2$ $[1, 7), 1$

$i=3$ $[6, 9), 1$

$i=4$ $[2, 11), 1$

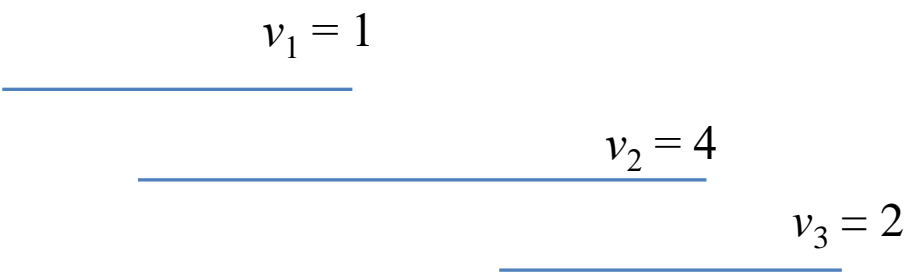
$i=5$ $[9, 12), 1$

$i=6$ $[10, 13), 1$

En este caso, más general

... ni siquiera la estrategia de elegir primero la tarea que termina más temprano produce garantizadamente una solución óptima:

- el ej. de la próxima diapositiva muestra que la estrategia codiciosa elegiría las tareas 1 y 3, con un valor total de 3
- ... mientras que la elección de sólo la tarea 2 tiene un valor de 4



$v_1 = 1$

$v_2 = 4$

$v_3 = 2$

En estos casos, en que las estrategias algorítmicas más atractivas —dividir para conquistar, elección codiciosa, recorridos de grafos— no funcionan,

... recurrimos a la **programación dinámica**, técnica de aplicabilidad muy amplia

Suponemos que las tareas están ordenadas por hora de término:

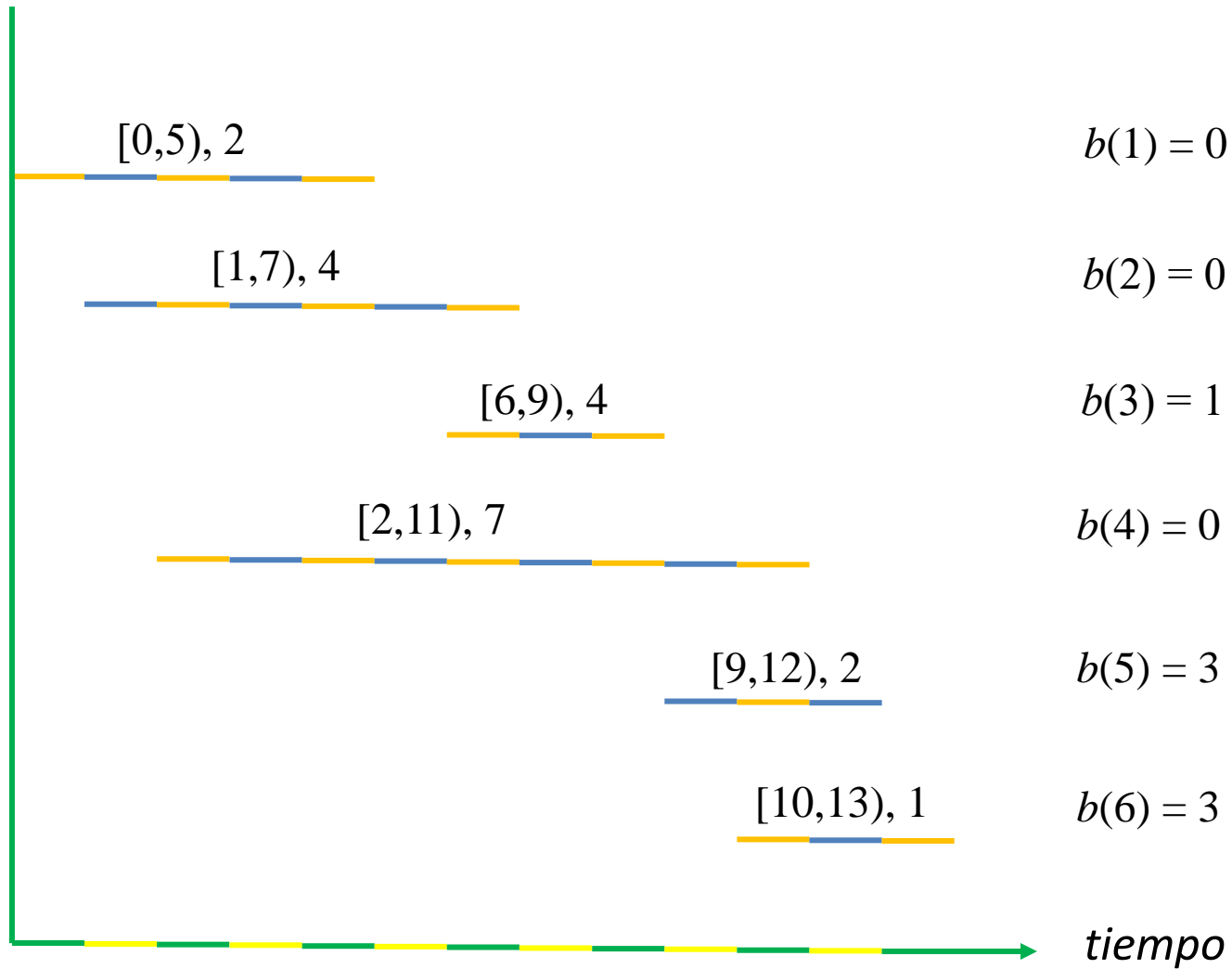
- $f_1 \leq f_2 \leq \dots \leq f_n$

Para cada tarea k definimos ...

$b(k)$ = la tarea i que termina más tarde antes del inicio de k :

- $f_i \leq s_k$ tal que para todo $r > i$, $f_r > s_k$
- $b(k) = 0$ si ninguna tarea $i < k$ satisface la condición anterior

| | | |
|--------|---------------|------------|
| P.ej., | $[0, 5), 2$ | $b(1) = 0$ |
| | $[1, 7), 4$ | $b(2) = 0$ |
| | $[6, 9), 4$ | $b(3) = 1$ |
| | $[2, 11), 7$ | $b(4) = 0$ |
| | $[9, 12), 2$ | $b(5) = 3$ |
| | $[10, 13), 1$ | $b(6) = 3$ |



(Aquí —con el “supongamos”— empieza propiamente la aplicación de la técnica de programación dinámica:

- la definición de $b(k)$ es principalmente una conveniencia para nuestros propósitos en este problema particular

)

Supongamos que tenemos una solución óptima Ω

Obviamente, con respecto a la presencia de la tarea n —la que termina más tarde entre todas las tareas— en esta solución óptima hay sólo dos posibilidades:

- la tarea n pertenece a Ω
- ... o bien la tarea n no pertenece a Ω

No sabemos cuál de las dos posibilidades es la que finalmente se va a dar en la solución óptima => hay que analizar ambas ...

Si la tarea n **no pertenece** a Ω ,

... entonces Ω es igual a la solución óptima para las tareas $1, \dots, n-1$:

- un problema del mismo tipo, pero más pequeño: no incluye la tarea n

...

...

En cambio, si la tarea n **pertenece** a Ω ,

... entonces ninguna tarea r , $b(n) < r < n$, puede pertenecer a Ω

... y Ω debe incluir,

... además de la tarea n ,

... una solución óptima para las tareas $1, \dots, b(n)^{[*]}$:

- nuevamente, un problema del mismo tipo, pero más pequeño

[*] esta afirmación se puede demostrar por contradicción

Es decir, en ambos casos, encontrar la solución óptima para las tareas 1, ..., n involucra encontrar las soluciones óptimas a problemas más pequeños del mismo tipo:

- esta característica es clave en los problemas que se pueden resolver usando programación dinámica
- ... y, como veremos, da origen a una posible formulación recursiva de la solución del problema

Así, dado que la solución óptima al problema original involucra encontrar soluciones óptimas a problemas más pequeños del mismo tipo,
... y aplicando este mismo razonamiento a estas soluciones a los problemas más pequeños, podemos afirmar lo siguiente:

1) Sea Ω_j la solución óptima al problema de las tareas 1, ..., j

... y sea $\text{opt}(j)$ su valor

... entonces buscamos Ω_n con valor $\text{opt}(n)$

...

...

2) Además, generalizando de las diaps. anteriores, para cada Ω_j podemos decir lo siguiente con respecto a la tarea j :

- si j pertenece a Ω_j , entonces $\text{opt}(j) = v_j + \text{opt}(b(j))$
- si j no pertenece a Ω_j , entonces $\text{opt}(j) = \text{opt}(j-1)$

3) Por lo tanto,

$$\text{opt}(j) = \max\{ v_j + \text{opt}(b(j)) , \text{opt}(j-1) \} \quad [*]$$

A partir de 3), escribimos un algoritmo recursivo para calcular $\text{opt}(n)$:

```
opt(j):  
    if  $j = 0$ :  
        return 0  
    else:  
        return  $\max\{v_j + \text{opt}(b(j)) , \text{opt}(j-1)\}$ 
```

- supone que las tareas están ordenadas por hora de término y que tenemos calculados los $b(j)$ para cada j
- $\text{opt}(0) = 0$, basado en la convención de que éste es el óptimo para un conjunto vacío de tareas

La corrección del algoritmo se puede demostrar por inducción

El problema de **opt** es su tiempo de ejecución en el peor caso:

- cada llamada a **opt** da origen a otras dos llamadas a **opt**
- esto es, *tiempo exponencial*
- p.ej., la próxima diapositiva muestra las llamadas recursivas que se producen como consecuencia de llamar inicialmente a `opt(6)`

La llamada `opt(6)` da origen a las llamadas `opt(5)`^[*] y **`opt(3)`**^[**] :

^[*] `opt(5)` da origen a las llamadas `opt(4)` y **`opt(3)`** :

- `opt(4)` da origen a la llamada **`opt(3)`**

- `opt(3)` da origen a ...

^[**] `opt(3)` da origen a las llamadas `opt(2)` y `opt(1)` :

- `opt(2)` da origen a la llamada `opt(1)`

Pero solo está resolviendo $n+1$ subproblemas diferentes:

- **opt(0), opt(1), ..., opt(n)**
- la razón por la que toma tiempo exponencial es el *número de veces* que resuelve cada subproblema

... p.ej., en la diapositiva anterior, se producen tres llamadas separadas a **opt(3)**, todas las cuales resuelven el mismo subproblema y, por supuesto, obtienen el mismo resultado

- esta es otra característica clave en la aplicación de programación dinámica

Podemos guardar el valor de **opt(j)** en un arreglo global la primera vez que lo calculamos

... y luego usar este valor ya calculado en lugar de todas las futuras llamadas recursivas a **opt(j)**

```
rec-opt(j):  
    if j = 0:  
        return 0  
    else:  
        if m[j] no está vacía:  
            return m[j]  
        else:  
            m[j] = max{  $v_j + \text{rec-opt}(b(j))$  ,  $\text{rec-opt}(j-1)$  }  
            return m[j]
```

rec-opt(n) es $O(n)$:

- ¿por qué?

Por supuesto, además de calcular el valor de la solución óptima,
... necesitamos saber cuál es esta solución

La clave es el arreglo **m**:

- usamos el valor de soluciones óptimas a los subproblemas sobre las tareas 1, 2, ..., j para cada j
- ... y usa (*) para definir el valor de $m[j]$ basado en los valores que aparecen antes (es decir, en índices menores que j) en **m**

Cuando llenamos **m**, el problema está resuelto:

- $m[n]$ contiene el valor de la solución óptima
- ... y podemos usar **m** para reconstruir la solución propiamente tal

$$(*) \text{ opt}(j) = \max\{ v_j + \text{opt}(b(j)) , \text{opt}(j-1) \}$$

También podemos calcular los valores en m iterativamente:

- $m[0] = 0$ y vamos incrementando j
- cada vez que necesitamos calcular un valor $m[j]$, usamos (*)

it-opt:

$m[0] = 0$

for $j = 1, 2, \dots, n$:

$m[j] = \max\{ v_j + m[b(j)] , m[j-1] \}$

```
it-opt:
  m[0] = 0
  for  $j = 1, 2, \dots, n$  :
     $m[j] = \max\{ v_j + m[b(j)] , m[j-1] \}$ 
```

Podemos demostrar por inducción que **it-opt** asigna a $m[j]$ el valor $\text{opt}(j)$

it-opt es claramente $O(n)$

El ej. de la selección de tareas con ganancias es representativo de los problemas que pueden ser resueltos eficientemente mediante programación dinámica

Generalizando, para desarrollar un algoritmo de programación dinámica

... necesitamos una colección de subproblemas, derivados del problema original, que cumplan ciertas propiedades:

- próx. diapo.

- i) el número de subproblemas es (idealmente) polinomial
 - ii) la solución al problema original puede calcularse a partir de las soluciones a los subproblemas
 - iii) hay un orden natural de los subproblemas, desde “el más pequeño” hasta “el más grande”
- ... y una recurrencia (ojalá) fácil de calcular (tal como [*] en diap. # 21)
- ... que permite calcular la solución a un subproblema a partir de las soluciones a subproblemas más pequeños

Multiplicación de matrices

Queremos multiplicar una secuencia de matrices

$$A_1 \times A_2 \times \cdots \times A_{n-1} \times A_n$$

La matriz A_i tiene dimensiones $p_{i-1} \times p_i$

¿Cómo minimizamos el número de multiplicaciones escalares ...
o **costo de la multiplicación de las matrices?**

Recordemos:

Multiplicar una matriz de dimensiones $a \times b$ por una de dimensiones $b \times c$ cuesta (es decir, implica realizar) $a \cdot b \cdot c$ multiplicaciones escalares:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{bmatrix} \times \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \\ y_{31} & y_{32} \end{bmatrix} = \begin{bmatrix} z_{11} & z_{12} \\ z_{21} & z_{22} \end{bmatrix}$$

... y el resultado es una matriz de $a \times c$

P.ej., la multiplicación de matrices $A \times B \times C$, con dimensiones:

- A de 10×100
- B de 100×5
- C de 5×50

... puede hacerse asociando las matrices de dos maneras —la multiplicación de matrices es asociativa:

$$(A \times B) \times C \quad \text{o bien} \quad A \times (B \times C)$$

En el primer caso, el número de multiplicaciones escalares —o costo— es 7,500

... en el segundo, ¡ 75,000 ! (produce la misma matriz resultado)

Por otra parte, a medida que aumenta el número de matrices en la secuencia

... el número de posibles formas de asociarlas para multiplicarlas también crece ... *exponencialmente*

P.ej., hay dos formas de asociar una secuencia de tres matrices

... y hay cinco formas de asociar una de cuatro matrices:

- $A_1 \times (A_2 \times (A_3 \times A_4))$
- $A_1 \times ((A_2 \times A_3) \times A_4)$
- $(A_1 \times A_2) \times (A_3 \times A_4)$
- $(A_1 \times (A_2 \times A_3)) \times A_4$
- $((A_1 \times A_2) \times A_3) \times A_4$

la matriz resultado es la misma, pero el número de multiplicaciones escalares (costo de la multiplicación) puede variar mucho

Así, nuestro problema es determinar la forma de asociar las matrices para realizar la multiplicación

$$A_1 \times A_2 \times \cdots \times A_{n-1} \times A_n$$

... tal que se *minimice el costo de la multiplicación* (es decir, el número de multiplicaciones escalares)

¿Cómo lo resolvemos?

En particular, ¿podemos resolverlo aplicando *programación dinámica*?

... es decir, ¿podemos encontrar una colección de subproblemas, derivados del problema original, que cumplan las propiedades de la diap. # 33?

En general, sea $w_{i,j}$ el costo mínimo para multiplicar $A_i \times \cdots \times A_j$

Claramente, alguna de las multiplicaciones de matrices será la última, por lo que la multiplicación anterior puede verse como

$$(A_i \times A_{i+1} \times \cdots A_k) \times (A_{k+1} \times \cdots \times A_{j-1} \times A_j)$$

... para algún k entre i y j

¿Cuál k ?

El que se obtenga al resolver la recurrencia:

$$w_{i,j} = \min_{i \leq k < j} \{w_{i,k} + p_{i-1} \cdot p_k \cdot p_j + w_{k+1,j}\}$$

... con condición de borde $w_{i,j} = 0$ si $i = j$

¿Por qué la recurrencia

$$w_{i,j} = \min_{i \leq k < j} \{w_{i,k} + p_{i-1} \cdot p_k \cdot p_j + w_{k+1,j}\}$$

... con condición de borde $w_{i,j} = 0$ si $i = j$?

Si la última multiplicación que produce el costo mínimo $w_{i,j}$ es

$$(A_i \times A_{i+1} \times \cdots A_k) \times (A_{k+1} \times \cdots \times A_{j-1} \times A_j)$$

... entonces los costos de las multiplicaciones $(A_i \times A_{i+1} \times \cdots A_k)$ y $(A_{k+1} \times \cdots \times A_{j-1} \times A_j)$ deben ser también los respectivos cos-tos mínimos, $w_{i,k}$ y $w_{k+1,j}$ —demostrable por contradicción

... además, $p_{i-1} \cdot p_k \cdot p_j$ es el costo de la última multiplicación

Este enfoque cumple las propiedades de la diap. # 33

La recurrencia muestra que la solución del problema, $w_{i,j}$, puede calcularse a partir de las soluciones de subproblemas, $w_{i,k}$ y $w_{k+1,j}$, del mismo tipo pero más pequeños

El número de subproblemas diferentes es sólo $O(n^2)$:

- uno por cada elección de i y j tal que $1 \leq i \leq j \leq n \rightarrow \binom{n}{2} + n$

Además, a partir de la recurrencia, podemos plantear un algoritmo recursivo que resuelve el problema (próx. diap.):

- sólo que este algoritmo toma tiempo exponencial —cada llamada a $w_{i,j}$ hace dos llamadas recursivas
- (similarmente al algoritmo en la diap. # 22, esto se debe a que el algoritmo resuelve un mismo problema varias veces)

$w_{i,j}$:

if $i = j$:

return 0

$w_{i,j} \leftarrow \infty$

for $k \leftarrow i \dots j - 1$:

$q \leftarrow w_{i,k} + p_{i-1} \cdot p_k \cdot p_j + w_{k+1,j}$

if $q < w_{i,j}$:

$w_{i,j} \leftarrow q$

return $w_{i,j}$

Similarmente a la estrategia de la diap. # 27, una posibilidad es usar un arreglo **w** en donde guardamos el valor de $w_{i,j}$ la primera vez que lo calculamos:

- sólo que en este caso, necesitamos un arreglo bidimensional —cada subproblema tiene dos índices

Y similarmente a la estrategia de la diap. # 30, podemos calcular los valores de **w** iterativamente (en lugar de recursivamente):

- primero, calculamos y guardamos los valores que no dependen de otros —los valores de las secuencias de largo uno (las matrices tomadas individualmente), $w_{i,j} = 0$, y los valores de las secuencias de largo dos, $A_1 \times A_2, A_2 \times A_3, \dots, A_{n-2} \times A_{n-1}, A_{n-1} \times A_n$
- luego, calculamos los valores que sólo dependen de los valores que ya tenemos guardados —los valores de las secuencias de largo tres— y los guardamos
- ... y así sucesivamente

```

matrices( $p$ ):
  for  $i \leftarrow 1 \dots n$ :
     $w_{i,i} \leftarrow 0$ 
    for  $t \leftarrow 2 \dots n$ :
      for  $i \leftarrow 1 \dots n - t + 1$ :
         $j \leftarrow i + t - 1$ 
         $w_{i,j} \leftarrow \infty$ 
        for  $k \leftarrow i \dots j - 1$ :
           $q \leftarrow w_{i,k} + p_{i-1} \cdot p_k \cdot p_j + w_{k+1,j}$ 
          if  $q < w_{i,j}$ ,  $w_{i,j} \leftarrow q$ 
  return  $w_{1,n}$ 

```

El problema de dar vuelto

Queremos dar vuelto por un total de S pesos *minimizando el número de monedas entregados*

Las monedas tienen los valores v_1, v_2, \dots, v_n , y tenemos un número suficiente de cada una

¿Cómo se puede resolver el problema si los valores son 1, 2, 5 y 10?

¿Y si los valores son 1, 4 y 6?

Solución para el caso $n = 4$ y $\{v_1, v_2, v_3, v_4\} = \{1, 2, 5, 10\}$

Proponemos el siguiente algoritmo codicioso:

- ordenemos las monedas en orden decreciente de valor $\rightarrow 10, 5, 2, 1$
- considerando las monedas en orden decreciente de valor
... para cada valor de moneda tomamos tantas monedas de ese valor como sea posible

Podemos demostrar que este algoritmo codicioso efectivamente produce el menor número de monedas —es óptimo—

... si las monedas tienen los valores indicados

Solución para el caso $n = 3$ y $\{v_1, v_2, v_3\} = \{1, 4, 6\}$

Es fácil ver que el algoritmo codicioso anterior ahora no funciona:

- p.ej., si $S = 8$, el algoritmo codicioso requiere tres monedas de valores 6, 1 y 1
... pero la solución óptima requiere sólo dos, de valores 4 y 4

Es decir, el algoritmo codicioso anterior no es óptimo para un conjunto cualquiera de (valores de) monedas

¿Cómo podemos resolver el caso más general?

Para que el problema tenga solución, vamos a considerar que siempre $v_1 = 1$

Seguimos (más o menos) la idea general aplicada en el problema de la selección de tareas, y sin tener que ordenar las monedas

La moneda v_n puede o no ser parte de la solución óptima, que denotamos por $z(S, n)$

... no sabemos, pero sí sabemos lo siguiente:

- si v_n no es parte de la solución óptima, entonces $z(S, n) = z(S, n-1)$
- si v_n es parte de la solución óptima, entonces $z(S, n) = z(S-v_n, n) + 1$
- es decir, $z(S, n) = \min \{ z(S, n-1), z(S-v_n, n) + 1 \}$

Tanto $z(S, n-1)$ como $z(S-v_n, n) + 1$ son problemas del mismo tipo que el problema original, sólo que “más pequeños”:

- $z(S, n-1)$ es el problema de dar el vuelto de S pesos, pero sólo con las monedas v_1, \dots, v_{n-1} (“más pequeño” \rightarrow menos monedas)
- $z(S-v_n, n) + 1$ es el problema de dar vuelto de $S-v_n$ pesos con las monedas v_1, \dots, v_n (y agregar una moneda de valor v_n) (“más pequeño” \rightarrow el monto del vuelto es menor)

Por lo tanto, para resolverlos ocupamos recursivamente la misma estrategia, generalizada

La solución óptima al problema de dar vuelto de T pesos ($T \leq S$) con las monedas de valores v_1, \dots, v_k ($0 \leq k \leq n$) se calcula así:

$$z(T, k) = \text{mín} \{ z(T, k-1), z(T-v_k, k) + 1 \}$$

Para que la recurrencia sea útil, hay que inicializarla (o establecer sus condiciones de borde) \rightarrow

como los valores de k y T van decreciendo, hay que considerar los casos $k = 0$ y $T \leq 0$:

- $z(T, 0) = +\infty$ para $T > 0$
- $z(0, k) = 0$
- $z(T, k) = +\infty$ para $T < 0$

Como vimos antes, en general no es una buena idea implementar directamente el algoritmo recursivo:

- va a resolver muchos subproblemas varias veces cada uno, aumentando artificialmente el tiempo de ejecución

La idea es ir almacenando en una tabla los valores $z(T, k)$ a medida que los vamos calculando —una tabla de $n \times S$

... e implementar el algoritmo ya sea recursiva o iterativamente

... pero mirando la tabla cada vez que aparece un subproblema para ver si ya lo resolvimos (y así evitar resolverlo de nuevo):

- mirar la tabla ocurre naturalmente en la versión iterativa

La complejidad es $O(n \times S)$

```
for  $T \leftarrow 1 \dots S$ :  
     $z[T, 0] \leftarrow +\infty$   
for  $k \leftarrow 1 \dots n$ :  
     $z[0, k] \leftarrow 0$   
for  $k \leftarrow 1 \dots n$ :  
    for  $T \leftarrow 1 \dots S$ :  
         $z[T, k] \leftarrow z[T, k-1]$   
        if  $T - v[k] \geq 0$ :  
             $z[T, k] \leftarrow \min\{z[T, k], z[T-v[k], k]+1\}$ 
```

La mochila con objetos 0/1^[*]

Tenemos n objetos **no fraccionables**^[*], cada uno con un valor v_k y un peso w_k ,

... y queremos ponerlos en una mochila con capacidad W :

- $W < \sum w_k$, es decir, no podemos poner todos los objetos en la mochila

... de manera de *maximizar la suma de los valores*, pero *sin exceder la capacidad de la mochila*

[*]cada objeto va completo o, simplemente, no va en la mochila: no se puede poner sólo una parte del objeto en la mochila

Si $knap(p, q, \omega)$ representa el problema de maximizar

$$\sum_{k=p}^q v_k x_k$$

... sujeto a

$$\sum_{k=p}^q w_k x_k \leq \omega \text{ y } x_k = \{0,1\}$$

... entonces nuestro problema es

$$knap(1, n, W)$$

Sea y_1, y_2, \dots, y_n una selección óptima de valores 0/1 para x_1, x_2, \dots, x_n :

- es decir, cada y_i vale 0, si el objeto j no está en la solución óptima, y vale 1 si el objeto j está en la solución óptima

En particular ...

y_1 puede ser 0 o 1 (no sabemos cuál es ... pero —de los ejemplos anteriores— sabemos qué implica cada posibilidad)

Si $y_1 = 0$ (es decir, el objeto 1 no está en la solución)

... entonces y_2, y_3, \dots, y_n debe ser una selección óptima para $\text{knap}(2, n, W)$:

- de lo contrario, no sería una selección óptima para $\text{knap}(1, n, W)$

Si $y_1 = 1$

... entonces y_2, y_3, \dots, y_n debe ser una selección óptima para $\text{knap}(2, n, W-w_1)$:

- de lo contrario, habría otra selección z_2, z_3, \dots, z_n de valores 0/1 tal que

$$\dots \sum w_k z_k \leq W-w_1 \text{ y } \sum v_k z_k > \sum v_k y_k, 2 \leq k \leq n$$

... por lo que la selección $y_1, z_2, z_3, \dots, z_n$ sería una selección para $\text{knap}(1, n, W)$ con mayor valor

Es decir, el problema se puede resolver a partir de las soluciones óptimas a subproblemas (más pequeños) del mismo tipo

Sea $g_k(\omega)$ el valor de una solución óptima a $\text{knap}(k+1, n, \omega)$:

- $g_0(W)$ es el valor de una solución óptima a $\text{knap}(1, n, W)$ —el problema original
- las decisiones posibles para x_1 son 0 y 1
- de las diapos. anteriores se deduce que

$$g_0(W) = \max\{ g_1(W) , g_1(W-w_1) + v_1 \}$$

Más aún,

... si y_1, y_2, \dots, y_n es una solución óptima a $\text{knap}(1, n, W)$,

... entonces para cada j , $1 \leq j \leq n$

$$y_1, \dots, y_j, y_{j+1}, \dots, y_n$$

... deben ser soluciones óptimas a¹

$$\text{knap}(1, j, \sum w_k y_k), 1 \leq k \leq j$$

$$\text{knap}(j+1, n, W - \sum w_k y_k), 1 \leq k \leq j$$

Por lo tanto²,

$$g_k(\omega) = \max\{ g_{k+1}(\omega) , g_{k+1}(\omega - w_{k+1}) + v_{k+1} \}$$

... en que $g_n(\omega) = 0$ si $\omega = 0$ y $g_n(\omega) = -\infty$ si $\omega < 0$

¹ significa que la solución a un subproblema puede calcularse a partir de las soluciones a subproblemas del mismo tipo más pequeños

² significa que hay una recurrencia (fácil) de calcular

P.ej., si $n = 3$, $(w_1, w_2, w_3) = (2, 3, 4)$, $(v_1, v_2, v_3) = (1, 2, 5)$, y $W = 6$

... tenemos que calcular

$$g_0(6) = \max\{ g_1(6), g_1(4)+1 \}$$

$$g_1(6) = \max\{ g_2(6), g_2(3)+2 \} = \max\{5, 2\} = 5, \text{ ya que}$$

$$g_2(6) = \max\{ g_3(6), g_3(2)+5 \} = \max\{0, 5\} = 5$$

$$g_2(3) = \max\{ g_3(3), g_3(-1)+5 \} = \max\{0, -\infty\} = 0$$

$$g_1(4) = \max\{ g_2(4), g_2(1)+2 \} = \max\{5, 2\} = 5, \text{ ya que}$$

$$g_2(4) = \max\{ g_3(4), g_3(0)+5 \} = \max\{0, 5\} = 5$$

$$g_2(1) = \max\{ g_3(1), g_3(-3)+5 \} = \max\{0, -\infty\} = 0$$

$$\text{Luego, } g_0(6) = \max\{5, 5 + 1\} = 6$$

Rutas más cortas entre todos los pares de vértices

Podemos ejecutar $|V|$ veces un algoritmo para rutas más cortas desde un vértice, una vez para cada vértice en el rol de s :

- si los costos de las aristas son no negativos, podemos usar el algoritmo de Dijkstra
... el tiempo de ejecución sería $O(VE \log V)$

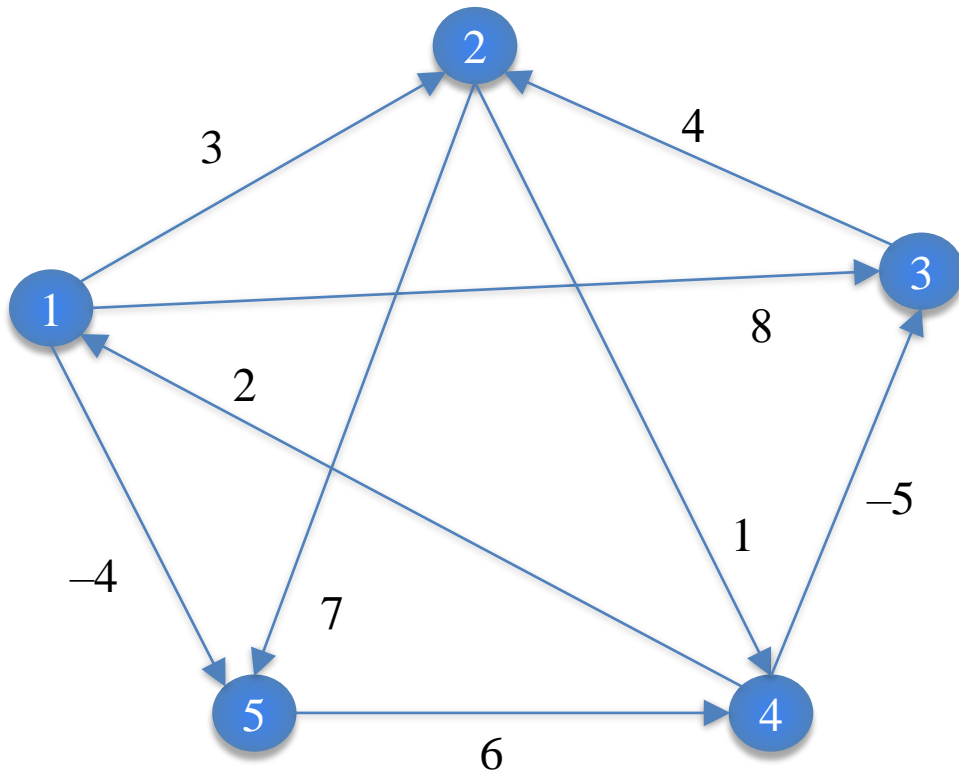
- si las aristas pueden tener costos negativos, debemos usar el algoritmo de Bellman-Ford
... el tiempo de ejecución sería $O(V^2E)$, que para grafos densos es $O(V^4)$

Podemos mejorar este último desempeño

Representaremos G por su *matriz de adyacencias* (en vez de las listas de adyacencias, que hemos usado mayoritariamente)

Si los vértices están numerados $1, 2, \dots, n$ (o sea, $|V| = n$),

... el input es una matriz W que representa los costos de las aristas



$W =$

| | | | | |
|----------|----------|----------|----------|----------|
| 0 | 3 | 8 | ∞ | -4 |
| ∞ | 0 | ∞ | 1 | 7 |
| ∞ | 4 | 0 | ∞ | ∞ |
| 2 | ∞ | -5 | 0 | ∞ |
| ∞ | ∞ | ∞ | 6 | 0 |

$W = (\omega_{ij})$, en que

$$\omega_{ij} = 0 \quad \text{si } i = j$$

= costo de la arista direccional (i, j) si $i \neq j$ y $(i, j) \in E$

$$= \infty \quad \text{si } i \neq j \text{ y } (i, j) \notin E$$

Suponemos que G **no contiene ciclos de costo negativo**

El algoritmo de Floyd-Warshall

El algoritmo considera los vértices intermedios de una ruta más corta

Si los vértices de G son $V = \{1, 2, \dots, n\}$, consideremos el subconjunto $\{1, 2, \dots, k\}$, para algún k

Para cualquier par de vértices $i, j \in V$,

... consideremos todas las rutas de i a j cuyos vértices intermedios están todos tomados del conjunto $\{1, 2, \dots, k\}$

... y sea p una ruta más corta entre ellas

k puede ser o no un vértice (intermedio) de p

Si k **no es** un vértice de p ,

... entonces todos los vértices (intermedios) de p están en el conjunto $\{1, 2, \dots, k-1\}$

\Rightarrow una ruta más corta de i a j con todos los vértices intermedios en $\{1, 2, \dots, k-1\}$

... es también una ruta más corta de i a j con todos los vértices intermedios en $\{1, 2, \dots, k\}$

Si k es un vértice de p , entonces podemos dividir p en dos tramos:

el tramo p_1 de i a k

... y el tramo p_2 de k a j

⇒ por el principio de optimalidad, p_1 es una ruta más corta de i a k con todos los vértices intermedios en $\{1, 2, \dots, k-1\}$

... y p_2 es una ruta más corta de k a j con todos los vértices intermedios en $\{1, 2, \dots, k-1\}$

Sea $d_{ij}^{(k)}$ el costo de una ruta más corta de i a j , tal que todos los vértices intermedios están en el conjunto $\{1, 2, \dots, k\}$

Cuando $k = 0$,

... una ruta de i a j sin vértices intermedios con número mayor que 0

... simplemente no tiene vértices intermedios,

... y por lo tanto tiene a lo más una arista $\Rightarrow d_{ij}^{(0)} = \omega_{ij}$

Definimos $d_{ij}^{(k)}$ recursivamente por

$$\begin{aligned} d_{ij}^{(k)} &= \omega_{ij} && \text{si } k = 0 \\ &= \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) && \text{si } k \geq 1 \end{aligned}$$

La matriz $D^{(n)} = (d_{ij}^{(n)})$ da la respuesta final:

$$d_{ij}^{(n)} = \delta(i, j) \text{ para todo } i, j \in V$$

El algoritmo de Floyd-Warshall, *bottom-up*, toma tiempo $O(V^3)$

$D^{(0)} = W$

for $k = 1 \dots n$:

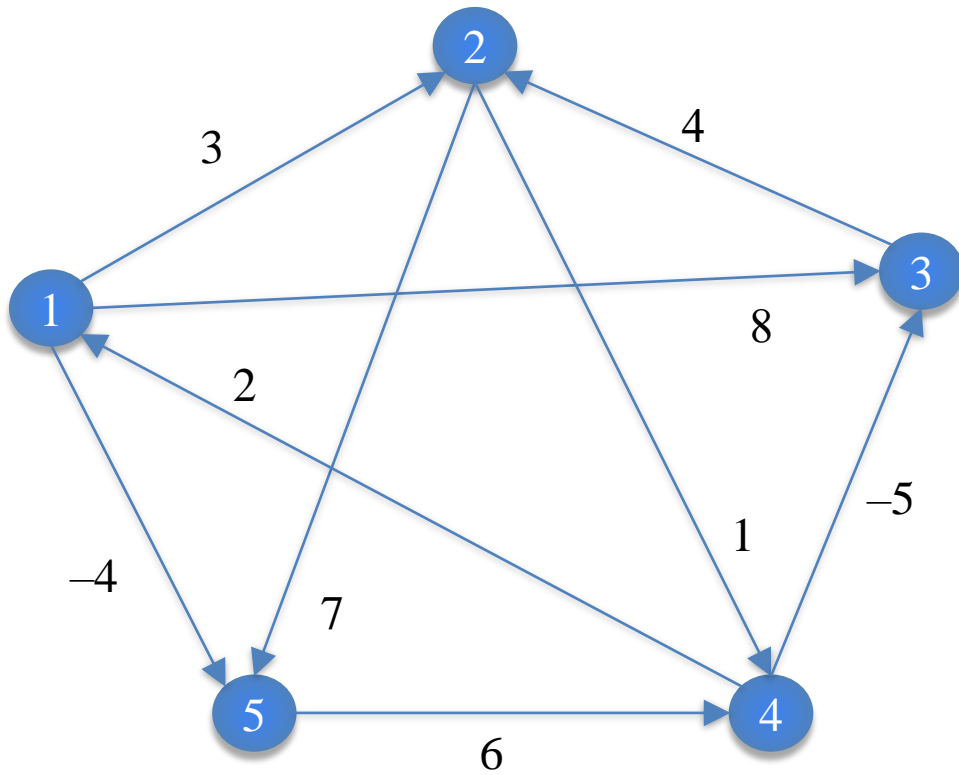
 sea $D^{(k)} = (d_{ij}^{(k)})$ una nueva matriz

 for $i = 1 \dots n$:

 for $j = 1 \dots n$:

$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

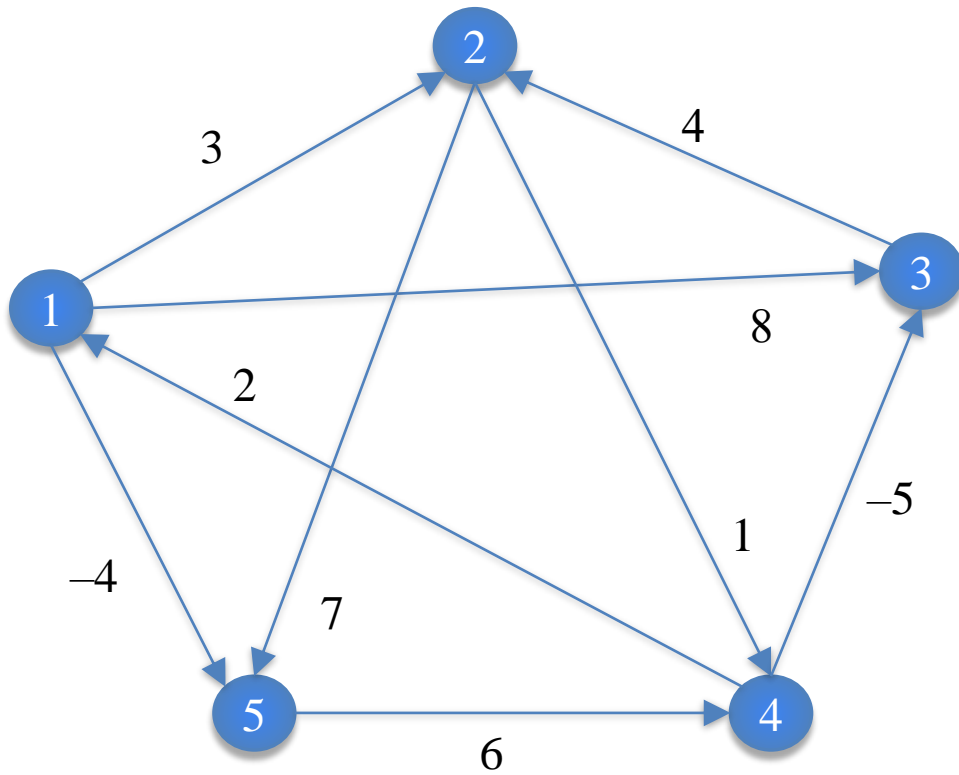
return $D^{(n)}$



$$D^{(0)} = \begin{matrix} & \begin{matrix} 0 & 3 & 8 & \infty & -4 \end{matrix} \\ \begin{matrix} \infty & 0 & \infty & 1 & 7 \end{matrix} & \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$

$$D^{(1)} = \begin{matrix} & \begin{matrix} 0 & 3 & 8 & \infty & -4 \end{matrix} \\ \begin{matrix} \infty & 0 & \infty & 1 & 7 \end{matrix} & \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \color{red}{5} & -5 & 0 & \color{red}{-2} \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$

$$D^{(2)} = \begin{matrix} & \begin{matrix} 0 & 3 & 8 & \color{red}{4} & -4 \end{matrix} \\ \begin{matrix} \infty & 0 & \infty & 1 & 7 \end{matrix} & \\ \infty & 4 & 0 & \color{red}{5} & \color{red}{11} \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$



$$D^{(3)} = \begin{matrix} & \begin{matrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix} \end{matrix}$$

$$D^{(4)} = \begin{matrix} & \begin{matrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{matrix} \end{matrix}$$

$$D^{(5)} = \begin{matrix} & \begin{matrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{matrix} \end{matrix}$$

Problema propuesto: Ubicación de propaganda en la ruta

Quieres poner letreros con propaganda de tu negocio en la ruta de Santiago a Puerto Montt (que la suponemos una línea recta)

Los puntos en que puedes poner los letreros son x_1, x_2, \dots, x_n , que representan las distancias desde Santiago

Además, tienes una predicción confiable que dice que si pones un letrero en el punto x_k , entonces obtienes una ganancia r_k

Por último, la autoridad vial exige que no puede haber dos letreros de un mismo negocio a menos de 5 km entre ellos

¿Dónde te conviene poner los letreros de manera de maximizar tu ganancia?

P.ej., si los puntos son $\{x_1, x_2, x_3, x_4\} = \{6, 7, 12, 14\}$ y las ganancias son $\{r_1, r_2, r_3, r_4\} = \{5, 6, 5, 1\}$, entonces te conviene poner los letreros en x_1 y x_3 para ganar 10