

Problemes 3

- 3.1. Tenim un programa que permet la simulació d'un sistema físic de temps discret. Volem simular tants passos del sistema com sigui possible. El nostre laboratori té accés a dos supercomputadors (A i B) capaços de processar el treball. No obstant això, són màquines compartides i no sempre poden executar els nostres treballs amb la prioritat més alta. Tant A com B poden processar el nostre programa.

Suposem que sabem, pels següents n minuts, la potència de processament disponible a cada màquina. Al minut i , podem executar a_i passos de la simulació a A o bé b_i a B. La simulació es pot transferir d'una màquina a un altra però, per fer-ho, s'ha de salvar i restaurar l'estat i això té un cost d'un minut de temps en el què no es pot fer cap progrés en la simulació. Volem un pla d'execució pels n minuts següents. Aquest pla ha de indicar, per a cada minut, A o B o "mou", i ha de ser consistent amb les restriccions donades. A més, volem que maximitzi el nombre total de passos de simulació executats.

- (a) Demostreu que el següent algorisme no resol correctament el problema proposat.

```

1: procedure PLA D'EXEC( $a, b$ )
2:   if  $a[1] \geq b[1]$  then
3:      $s[1] = 'A'$ 
4:   else
5:      $s[1] = 'B'$ 
6:    $i = 2$ 
7:   while  $i \leq n$  do
8:     if  $s[i - 1] == 'A'$  then
9:       if  $b[i + 1] > a[i] + a[i + 1]$  then
10:         $s[i] = 'mou'; s[i + 1] = 'B'; i = i + 2$ 
11:      else
12:         $s[i] = 'A'; i = i + 1$ 
13:    else
14:      Com al cas previ canviant A/a per B/b

```

- (b) Doneu un algorisme eficient que, donats a_1, \dots, a_n i b_1, \dots, b_n , proporcioni un pla d'execució que permeti executar el màxim nombre de passos de simulació.

Una solució:

- (a) Per als vectors $a = \langle 2, 1 \rangle$ i $b = \langle 2, 20 \rangle$, suposant que l'accés fora de rang no falla, el programa retornaria la solució $\langle A, A \rangle$ que és incorrecta.
- (b) Para encontrar una solución analizamos la estructura de suboptimalidad de una solución óptima. Observemos que una solución óptima ejecutará pasos de simulación en el instante n , si no no sería óptima. Puede hacerlo en A o en B . Suponiendo que sea en A , el paso previo puede ser A o mou . En el primer caso la solución debe ser una solución óptima para $n - 1$ pasos ejecutando en A en el paso $n - 1$ y en el segundo una solución óptima para $n - 2$ pasos ejecutando en B en el paso $n - 2$.

Para establecer la recurrencia utilizaremos notación adicional, para $0 \leq k \leq n$:

$A(k)$ = el número máximo de pasos de simulación que podemos ejecutar en k pasos ejecutando la simulación en A en el paso k .

$B(k)$ = el número máximo de pasos de simulación que podemos ejecutar en k pasos ejecutando la simulación en B en el paso k .

Tenemos la recurrencia:

$$\begin{aligned}A(k) &= a(k) + \max(A(k-1), B(k-2)) \\B(k) &= b(k) + \max(B(k-1), A(k-2))\end{aligned}$$

para $k \geq 2$, y los casos base $A(0) = B(0) = 0$, $A(1) = a[1]$ y $B(1) = b[1]$.

El coste de la solución óptima que buscamos es $\max(A(n), B(n))$.

Como el número total de subproblemas es $O(n)$ podemos utilizar PD. Para ello basta con un recorrido en orden creciente de las dos tablas guardando un puntero con la información de la opción de dónde proviene el valor máximo.

El coste de calcular uno de los valores de la tabla A o B es constante y por ello el algoritmo necesita tiempo $O(n)$ incluido el paso de recuperación de la solución siguiendo la información de los punteros.

3.2. Tenim un graf no dirigit $G = (V, E)$. Com és habitual, d_u denota el grau del vèrtex u . Diem que una partició dels vèrtexs en V_1 i $\bar{V}_1 = V \setminus V_1$ és *equilibrada* quan $\sum_{u \in V_1} d_u = \sum_{v \notin V_1} d_v$.

Doneu un algorisme de programació dinàmica per a determinar si un graf donat té o no té una partició equilibrada.

Una solución

Sabemos que en un grafo $\sum_{u \in V} d_u = m$. El enunciado nos pide decir si es posible encontrar un conjunto $V_1 \subseteq V$ para el que $\sum_{u \in V_1} d_u = m$.

Supongamos que $V = \{v_1, \dots, v_n\}$ y que tenemos una solución V_1 al problema. Vamos a analizar la estructura de suboptimalidad de esta solución.

Con relación al vértice v_n , tenemos dos casos

- $v_n \in V_1$, en este caso tenemos que $\sum_{v \in V_1 - \{v_n\}} d_v = m - d_{v_n}$
- $v_n \notin V_1$, en este caso tenemos que $\sum_{v \in V_1 - \{v_n\}} d_v = m$

Usando esta caracterización podemos identificar un conjunto de subproblemas, $P[i, x]$ determinar si en $V_i = \{v_1, \dots, v_i\}$ se puede encontrar un subconjunto de vértices $V' \subseteq V_i$ tal que $\sum_{v \in V'} d_v = x$.

De acuerdo con el estudio anterior, tenemos caracterizadas la posibilidad de tener una solución que incluya el último vértice en el conjunto considerado o no. Esto nos lleva a la siguiente recurrencia.

For, $1 \leq i \leq n$ and $0 \leq x \leq m$

$$P[i, x] = \begin{cases} d_{v_1} = x & i = 1 \\ P[i - 1, x] & i > 1 \text{ and } x - d_{v_i} < 0 \\ P[i - 1, x - d_{v_i}] \text{ or } P[i - 1, x] & \text{otherwise} \end{cases}$$

El número total de subproblemas es nm y el coste por elemento es $O(1)$. Por lo tanto implementando la recurrencia con memoización o mediante un cálculo en tabla tendremos un algoritmo con coste $O(nm)$.

- 3.3. Us donen una cadena de n caràcters $s[1 \dots n]$, que pot ser un text on totes les separacions entre mots ha desaparegut, per exemple *aquestaesunafrasequepodiaserunexample*. Volem reconstruir el text original amb ajut d'un diccionari $D(\cdot)$, tal que per a tot mot possible w

$$D(w) = \begin{cases} \text{cert} & \text{si } w \text{ és un mot valid} \\ \text{fals} & \text{altrament} \end{cases}$$

- (a) Doneu un algorisme de programació dinàmica que determine si la cadena $s[\cdot]$ es pot reconstruir com a una seqüència de mots vàlids. Si assumim que les crides a D es poden fer en temps $\Theta(1)$. Quina és la complexitat del vostre algorisme?
- (b) Si la cadena és valida, fes que el teu algorisme escrigui la frase correcta, amb separacions entre mots.

```

procedure mots ( string S )
vector<bool> v (S.size() + 1, false)
v[0] = true
for (int i = 1; i < v.size(); i++)
    for (int j = 0; j < i and not v[i]; j++)
        if (v[j] and D(S.substring(j, i)))
            v[i] = true

if (v[S.size()])
    string frase = ""
    int i = S.size()
    int ini = 0
    while (i > 0)
        for (int j = i - 1; j >= 0; j--)
            if (v[j] and D(S.substring(j, i)))
                ini = j
        if (i == S.size())
            frase = S.substring(ini, i) + frase
        else
            frase = S.substring(ini, i) + " " + frase
        i = ini

return frase
return "Mots no vàlids"

```

Cost: $O(n^2)$

Sent n la mida de la cadena de caràcters

Exemple

"holacomestás"

"holacomestás"
 T F F F F F F F F F F
 0 1 2 3 4 5 6 7 8 9 10 11 12

j i
 $v[j] = \text{true}$ $D(v[j:i] = "h") = \text{false}$

"holacomestás"
 T F F F F F F F F F F
 0 1 2 3 4 5 6 7 8 9 10 11 12

j i
 $v[j] = \text{true}$ $D(v[j:i] = "ho") = \text{false}$
 $v[j] = \text{false}$ $D(v[j:i] = "o") = \text{false}$

"holacomestás"
 T F F F F F F F F F F
 0 1 2 3 4 5 6 7 8 9 10 11 12

j=0 i=3
 $v[j] = \text{true}$ $D(v[j:i] = "hol") = \text{false}$
 $v[j] = \text{false}$ $D(v[j:i] = "ho") = \text{false}$
 $2=j$ i=3
 $v[j] = \text{false}$ $D(v[j:i] = "o") = \text{false}$

"holacomestás"
 T F F F T F F F F F F
 0 1 2 3 4 5 6 7 8 9 10 11 12

j=0 i=4
 $v[j] = \text{true}$ $D(v[j:i] = "hol a") = \text{true}$

"holacomestás"
 T F F F T F F F F F F
 0 1 2 3 4 5 6 7 8 9 10 11 12

j=0 i=5
 $v[j] = \text{true}$ $D(v[j:i] = "holac") = \text{false}$

j=1 i=5
 $v[j] = \text{false}$ $D(v[j:i] = "olac") = \text{false}$

j=2 i=5
 $v[j] = \text{false}$ $D(v[j:i] = "lac") = \text{false}$

j=3 i=5
 $v[j] = \text{false}$ $D(v[j:i] = "ac") = \text{false}$

4=j i=5
 $v[j] = \text{true}$ $D(v[j:i] = "c") = \text{false}$

"holacomestás"
 T F F F T F F F F F F
 0 1 2 3 4 5 6 7 8 9 10 11 12

j=0 i=6
 $v[j] = \text{true}$ $D(v[j:i] = "holaco") = \text{false}$

j=1 i=6
 $v[j] = \text{false}$ $D(v[j:i] = "olaco") = \text{false}$

j=2 i=6
 $v[j] = \text{false}$ $D(v[j:i] = "laco") = \text{false}$

j=3 i=6
 $v[j] = \text{false}$ $D(v[j:i] = "aco") = \text{false}$

4=j i=6
 $v[j] = \text{true}$ $D(v[j:i] = "co") = \text{false}$

5=j i=6
 $v[j] = \text{false}$ $D(v[j:i] = "o") = \text{false}$

"holacomestás"
 T F F F T F F T F F F F
 0 1 2 3 4 5 6 7 8 9 10 11 12

j=0 i=7
 $v[j] = \text{true}$ $D(v[j:i] = "holacom") = \text{false}$

j=1 i=7
 $v[j] = \text{false}$ $D(v[j:i] = "olacom") = \text{false}$

j=2 i=7
 $v[j] = \text{false}$ $D(v[j:i] = "lacom") = \text{false}$

j=3 i=7
 $v[j] = \text{false}$ $D(v[j:i] = "acom") = \text{false}$

4=j i=7
 $v[j] = \text{true}$ $D(v[j:i] = "com") = \text{true}$

"holacomes está
 T F F F T F F T F F F F
 0 1 2 3 4 5 6 7 8 9 10 11 12
 j=0 i=8
 v[j] = true D("holacome") = false
 j=1 i=8
 v[j] = false
 j=2 i=8
 v[j] = false
 j=3 i=8
 v[j] = false
 j=4 i=8
 v[j] = true D("come") = false
 j=5 i=8
 v[j] = false
 j=6 i=8
 v[j] = false
 j=7 i=8
 v[j] = true D("e") = false

"holacomes está
 T F F F T F F T F F F F
 0 1 2 3 4 5 6 7 8 9 10 11 12
 j=0 i=9
 v[j] = true D("holacomes") = false
 j=1 i=9
 v[j] = false
 j=2 i=9
 v[j] = false
 j=3 i=9
 v[j] = false
 j=4 i=9
 v[j] = true D("comes") = false
 j=5 i=9
 v[j] = false
 j=6 i=9
 v[j] = false
 j=7 i=9
 v[j] = true D("es") = false
 8=j i=9
 v[j] = false

"holacomes está
 T F F F T F F T F F F F
 0 1 2 3 4 5 6 7 8 9 10 11 12
 j=0 i=10
 v[j] = true D("holacomoest") = false
 j=1 i=10
 v[j] = false
 j=2 i=10
 v[j] = false
 j=3 i=10
 v[j] = false
 j=4 i=10
 v[j] = true D("comest") = false
 j=5 i=10
 v[j] = false
 j=6 i=10
 v[j] = false
 j=7 i=10
 v[j] = true D("est") = false
 j=8 i=10
 v[j] = false
 9=j i=10
 v[j] = false

"holacomes está
 T F F F T F F T F F F T F
 0 1 2 3 4 5 6 7 8 9 10 11 12
 j=0 i=11
 v[j] = true D("holacomesta") = false
 j=1 i=11
 v[j] = false
 j=2 i=11
 v[j] = false
 j=3 i=11
 v[j] = false
 j=4 i=11
 v[j] = true D("comesta") = false
 j=5 i=11
 v[j] = false
 j=6 i=11
 v[j] = false
 j=7 i=11
 v[j] = true D("está") = true

Resultat del vector v:

"holacomestás"
T F F F T F F T F F F T T
0 1 2 3 4 5 6 7 8 9 10 11 12
j=0 i=12

v[j] = true
j=1
v[j] = false
j=2
v[j] = false
j=3
v[j] = false
j=4
v[j] = true
j=5
v[j] = false
j=6
v[j] = false
j=7
v[j] = true

D("holacomestás") = false
i=12
i=12
i=12
i=12
i=12
i=12
i=12
D("comestás") = false
i=12
i=12
i=12
D("estás") = true

"holacomestás"
T F F F T F F T F F F T T
0 1 2 3 4 5 6 7 8 9 10 11 12

Ara construim la frase de manera correcte

i=12 j=11
v[j] = true D("ás") = false
i=12 j=10
v[j] = true D("estás") = true $\Rightarrow \text{ini}=7$
i=12 j=9
v[j] = true D("comestás") = false
i=12 j=8
v[j] = true D("hola comestás") = false

frase = "estás"

i=7 j=4
v[j] = true D("com") = true $\Rightarrow \text{ini}=4$
i=7 j=0
v[j] = true D("hola com") = false

frase = "com estás"

i=4 j=0
v[j] = true D("hola!") = true $\Rightarrow \text{ini}=0$

frase = "hola com estás"

- 3.4. Considerem una graella de 4 files per n columnes, i un conjunt de $2n$ fitxes. Una fitxa es pot col·locar exactament a una casella de la graella. Definim un *patró legal columna* a una columna de la graella com la situació resultant de col·locar fitxes a la columna de manera que no dues fitxes estiguin en caselles adjacents. De la mateixa manera podem definir un *patró legal fila*. Una *configuració legal* és la situació resultant de situar fitxes a la graella, de manera que totes les columnes i files tinguin patrons legals. A cada casella de la graella hi ha escrit un enter. El *valor* de la configuració és la suma dels enters de les caselles ocupades.

Dos patrons a columnes adjacents són *compatibles* si formen una configuració legal a la matriu formada per les dos columnes.

(a) Determineu el nombre total de patrons legals que pot haver en una columna.

(b) Dissenyeu un algorisme $O(n)$ per calcular una configuració de valor màxim.

Ajut: Considereu subproblemes amb les primeres k columnnes ($k \leq n$) i un patró prefixat a la columna k .

3.3 - Patrons legals

Considerem una graella de 4 files per n columnes, i un conjunt de $2n$ fitxes. Una fitxa es pot col·locar exactament a una casella de la graella. Definim un *patró legal columna* a una columna de la graella com la situació resultant de col·locar fitxes a la columna de manera que no dues fitxes estiguin en caselles adjacents. De la mateixa manera podem definir un *patró legal fila*. Una *configuració legal* és la situació resultant de situar fitxes a la graella, de manera que totes les columnes i files tinguin patrons legals. A cada casella de la graella hi ha escrit un enter. El *valor* de la configuració és la suma dels enters de les caselles ocupades.

Dos patrons a columnes adjacents són *compatibles* si formen una configuració legal a la matriu formada per les dos columnes.

1. Determineu el nombre total de patrons legals que pot haver en una columna.
2. Dissenyeu un algorisme $O(n)$ per calcular una configuració de valor màxim.

Ajut: Considereu subproblemes amb les primeres k columnes ($k \leq n$) i un patró prefixat a la columna k .

Una solución (prof. Maria Serna)

1. En una columna podemos colocar como máximo dos fichas y como mínimo ninguna. Esto nos da 8 patrones (a los que identificaremos por un número de 0 a 8):

	x				x	x	
		x					x
			x		x		
				x		x	x

Una vez tengamos un tablero G podemos calcular $P[i, j]$, $0 \leq i < 8$, $0 \leq j < n$, la puntuación obtenida al colocar las fichas con el patrón i en la columna j , en tiempo $O(n)$.

2. Si miramos una solución óptima, en la columna 0 tenemos un patrón y en la columna 1 un patrón compatible con el de la columna anterior. Además la puntuación obtenida en las columnas 1 a n tiene que ser máxima con la única condición de que el patrón en la columna 1 sea el que tenemos fijado. Si no fuese así podríamos reemplazar la segunda parte de la solución y obtener mejor puntuación.

Esta estructura de suboptimalidad indica como subproblemas los que se indica en el enunciado, determinados por las columnas k a $n - 1$ y un patrón colocado en la columna k .

Introduzcamos notación, $T(i, k)$ será la puntuación máxima posible a las columnas k a $n - 1$ colocando patrón i en la columna k ($0 \leq i < 8$ y $0 \leq k < n$). Teniendo en cuenta el análisis de suboptimalidad tenemos la siguiente recurrencia para calcular la puntuación máxima en cada subproblema:

$$T(i, k) = \begin{cases} P[i, n - 1] & \text{si } k = n - 1 \\ P[i, k] + \max_{\ell \text{ comp } i} T(\ell, k + 1) & \text{si } k < n \end{cases}$$

Con esto la puntuación máxima que podemos obtener es $\max_{i \in [0..8]} T(i, 0)$.

Si además queremos obtener la solución necesitamos calcular $S(i, k)$ ($0 \leq i < 8$ y $0 \leq k < n - 1$) el patrón que nos da el máximo en la fila siguiente.

$$S(i, k) = \arg \max_{\ell \text{ comp } i} T(\ell, k + 1).$$

Utilizo aquí la notación arg para identificar el valor ℓ que proporciona el máximo.

A partir de esta recurrencia tenemos que

1. Calcular los valores $P[i, k]$
2. Calcular los valores $T(i, k)$ y $S(i, k)$.
3. Calcular la puntuación máxima $\max_{i \in [0..8]} T(i, 0)$ y el patrón en la columna 0, $i_0 = \arg \max_{i \in [0..8]} T(i, 0)$.
4. Calcular i_k , patrón en la columna k , $k > 1$ utilizando $i_k = S(i_{k-1}, k - 1)$.

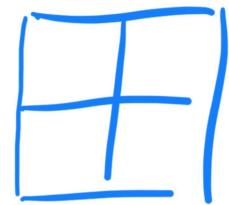
El paso más costoso es el 2 que se puede implementar con coste $O(1)$ por subproblema (cálculo en tabla o llamada recursiva con memoización) ya que solo hay 8 patrones posibles y n columnas.

El coste total es $O(n)$, ya que el número total de subproblemas es $8n$, el espacio adicional también es $O(n)$.

3.5. El Rector vol construir el monument a l'alumne sacrificat. Decideix construir al Campus Nord, el monòlit més alt possible de totxo vist. Aconsegueix n tipus diferents de totxos, i de cada tipus una quantitat suficientment gran. Cada totxo de tipus i es pot considerar com un ortoedre amb dimensions $\langle l_i, w_i, h_i \rangle$. Un totxo es pot col·locar en qualsevol de les tres posicions que mantenen les arestes paral·leles als tres eixos fixos. Per a construir el monòlit un totxo es pot col·locar a sobre d'un altre, sols si cada una de les dues arestes de la base del totxo de sobre té longitud estrictament menor que l'aresta que li és paral·lela del totxo de sota. A la base es col·loca un sol totxo. Dissenyeu un algorisme eficient per a determinar el monòlit més alt que el Rector pot construir. Quina és la seva complexitat?.

- 3.6. Imagina que ets el cap dels serveis informàtics de la FIB, on milers de persones accedeixen cada dia al servidor central. Suposem que tens una estimació (x_1, x_2, \dots, x_n) del nombre d'usuaris que accediran al servidor en els propers n dies. El software que controla el servidor no està ben dissenyat i el nombre d'usuaris per dia que pot gestionar decrementa cada dia, a partir del darrer dia en què es va fer *reboot*. Sigui s_i el nombre d'usuaris que el servidor pot gestionar l' i -èsim dia després de la darrera aturada, per tant $s_1 > s_2 > s_3 > \dots > s_n$. Assumim que el dia que es fa el reboot, no es pot donar servei a cap usuari. Donada una seqüència de carrega (x_1, \dots, x_n) i de limitacions (s_1, \dots, s_n) , dissenyeu un algorisme per a la planificació que especifique els dies òptims que s'han de fer els reboots de manera que es maximitza el nombre total de clients als quals el servidor dóna servei. Per exemple, si $n = 5$ i $s_1 = 16, s_2 = 8, s_3 = 4, s_4 = 2, s_5 = 1$. Quan $x_1 = 17, x_2 = 9, x_3 = 5, x_4 = 3, x_5 = 2$, la solució òptima és no rebotar i donar servei a 31 clients. Quan $x_1 = 17, x_2 = 9, x_3 = 17, x_4 = 3, x_5 = 17$ la solució òptima és rebotar el segon i quart dies, donant servei a un màxim de 48 clients.

$s \backslash x$	0	17	9	17	3	17
0	0	0	0	0	0	0
16	0	16	9	16	3	16
8	0	16	24			
4	0	16				
2	0	16				
1	0	16				48



3.7. El problema de la *partició lineal* es el següent: Donada una seqüència de n valors positius, (s_1, \dots, s_n) volem obtenir una seqüència de $r+1$ valors, $i_1, \dots, i_{r+1} \in \{1, \dots, n+1\}$ tal que $i_1 = 1$, $i_{r+1} = n+1$ i $i_j < i_{j+1}$, per $1 \leq j \leq r$. Aquesta successió divideix la seqüència inicial en r rangs. Per cada rang j , $1 \leq j \leq r$, definim $S_j = \sum_{i_j \leq k < i_{j+1}} s_k$. A cada seqüència i_1, \dots, i_{r+1} se li assigna el cost $S(i_1, \dots, i_{r+1}) = \max_{1 \leq j < r} S_j$. Volem obtenir seqüència que proporcioni r rangs amb cost mínim.

Per exemple, si els valors són:

100, 200, 300, 400, 500, 600, 700, 800, 900

i $r = 3$, una solució és 1, 4, 7, 10 que té cost 2400, i una solució óptima és 1, 6, 8, 10 amb cost 1700.

Però si els valors són:

1000, 250, 120, 40, 50, 160, 700, 180, 90

i $r = 3$, una solució és 1, 4, 7, 10 que té cost 1370, i una solució óptima és 1, 2, 7, 10 amb cost 1000.

Dissenyeu un algorisme basat en programació dinàmica per resoldre el problema. Analitza la complexitat temporal i espacial de l'algorisme proposat.

$C_{i,p}$ = coste de una partició óptima de la subsecuencia s_i, s_{i+1}, \dots, s_n en p parts

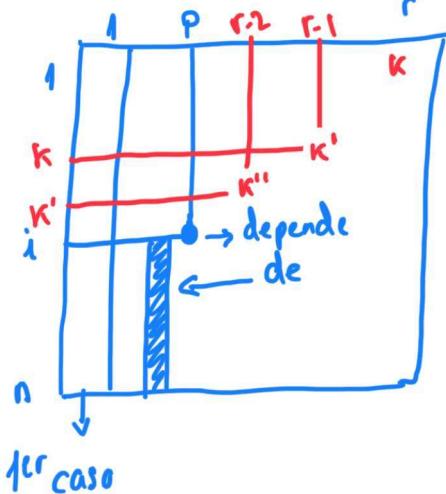
$C_{1,r}$ = valor que nos interesa

$$C_{i,p} = \begin{cases} \boxed{s_i + s_{i+1} + \dots + s_n} & \text{Si } p=1 \wedge i \leq n \\ \max\{s_i, \dots, s_n\} & \text{Si } p=n+1-i \\ +\infty & \text{Si } p > n+1-i \vee p \leq 0 \\ \min \left\{ \max \left\{ C_{k,p-1}, \boxed{s_i + \dots + s_{k-i}} \right\} \right\}_{1 \leq k \leq n} & \text{en otro caso} \end{cases}$$

$$\begin{aligned} SS[i] &= s_1 + \dots + s_i \\ ss[i] &= s_1 + \dots + s_n \end{aligned}$$

$O(n^2 r)$

$$C[i][p] \equiv C_{i,p} = \sum_{k=i}^n \Theta(i, r) = O(n^2 r)$$



$$\Theta(n+r)$$

$R_{i,p} = \begin{cases} 0 & \text{en caso base} \\ k+q \text{ minimiza } C_{i,p} \end{cases}$

$\Theta(r)$

$$\sum_{i=1}^n \sum_{p=1}^{n-i} \Theta(n-i+1) = \sum_{i=1}^n \Theta((n-i+1)r)$$

rellenamos \rightarrow

- 3.8. Un palíndrom és un mot $w_1w_2\dots w_k$ tal que $w_kw_{k-1}\dots w_1$ és el mateix mot. Per exemple *senentesnensisetnenes*. Donada una cadena $A = a_1a_2a_3\dots a_n$ direm que A té una partició en forma de palíndroms si cada subcadena de la partició és un palíndrom. Per exemple, si $A = ababbabbababa$ tenen particions $aba|b|bbabb|a|b|aba$ i $aba|bbb|abba|ba|ba$ en forma de palíndroms. Dissenyeu un algorisme amb complexitat $O(n^3)$ tal que donada una cadena A de talla n , produeixi la partició en forma de palíndroms amb el **mínim nombre de talls**. Podeu dissenyar un algorisme amb complexitat $O(n^2)$? Noteu que si A es un palíndrom no hi haurà cap partició.

$T_k = \text{mínim nombre de talls per dividir } w_k, \dots, w_n \text{ en forma de palíndrom}$

$$\begin{array}{ll} T_n = 1 & P_{ii} = 1 \\ T_i = ? & P_{i,i+1} = \begin{cases} 1 & \text{si } w_i = w_{i+1} \\ 0 & \text{si } w_i \neq w_{i+1} \end{cases} \end{array}$$

$$T_k = \min_{j: P[k, j-1] = 1} \{ 1 + T_j \}$$

- 3.9. (**Tallant ADN**). En termes computacionals, podem pensar que una cadena de ADN es un mot sobre l'alfabet $\{A, C, G, T\}$. El cost d'una cadena de ADN el definim com $\prod_{X \in \{A, C, G, T\}} (\text{freq}[X] + 1)$ on $\text{freq}[X]$ es el nombre d'ocurrències del símbol X . Per exemple, si la cadena es $ACGAC$ el seu cost es $3 \times 3 \times 2 \times 1 = 18$

Tenim una cadena de ADN de llargada n i volem dividir-la en k trossos (subcadenes no buides), de manera que es minimitze el cost màxim d'entre les k portions. Per exemple per $ACGAC$ i $k = 2$ tenim 4 formes de trencar la cadena en dos i una de les que ens dona el millor cost es AC i GAC (amb cost màxim 8). Proporcioneu un algorisme ho més eficient possible per a resoldre aquest problema, donats S i k .

3.10. **🔗(Estiu als jardins de la UPC).** Per tal de incrementar els ingressos, el Gerent de la UPC ha decidit llogar els jardins de Torre Girona (i part del Campus Nord) per realitzar diferents activitats d'esbarjo durant la temporada d'estiu. Després d'una crida a propostes d'activitats, la UPC ha rebut una gran quantitat de sol·licituds. Cada sol·licitud indica la data i l'hora, d'inici i de fi de l'activitat, i el tipus de recursos que es necessiten per fer-la. Per a cadascuna de les sol·licituds, el servei d'Economia de la UPC ha calculat el benefici que s'obtindria si es fés l'activitat. Per tal d'evitar problemes, en aquesta primera temporada, la UPC ha decidit llogar els espais en exclusiva a les activitats. Això implica que no es poden dur a terme més d'una activitat al mateix temps. UPCNet necessita un programa que permeti obtenir una selecció d'activitats que es puguen planificar en exclusiva i que proporcionin el màxim benefici possible. Proporcioneu un algorisme, tant eficient com us sigui possible, per a resoldre el problema de UPCNet.

Una solució El problema a resoldre és el problema de selecció d'activitats compatibles maximitzant el pes de les activitats seleccionades. El pes d'una activitat es el benefici que obtindria la UPC. Aquest problema està resolt amb programació dinàmica a les transparències de l'assignatura. L'algorisme té cost $O(n)$ i fa servir espai addicional $O(n)$.

3.11. (Rèpliques)

Tenim un sistema S format per la concatenació de n subsistemes S_1, S_2, \dots, S_n . Per a cada subsistema S_i es coneix la seva probabilitat de fallida ϕ_i . La probabilitat p_{corr} que el sistema funcioni correctament és la probabilitat que **tots** els subsistemes funcionin correctament, és a dir:

$$p_{\text{corr}} = \prod_{1 \leq i \leq n} (1 - \phi_i)$$

Ara bé, per tal d'augmentar la probabilitat que el sistema funcioni correctament podem replicar els subsistemes; així, si posem $x_i > 0$ rèpliques del subsistema S_i la probabilitat que falli passa a ser

$$\phi'_i = \phi_i^{x_i},$$

donat que només fallarà si les x_i rèpliques fallen. Desgraciadament tenim un pressupost limitat de $B \in N$ euros en total, el cost del subsistema S_i és $v_i \in N$ i només hi ha y_i unitats del subsistema S_i .

Es demana que plantegueu la resolució d'aquest problema mitjançant un algorisme de programació dinàmica (PD) que calculi el nombre de rèpliques x_1, \dots, x_n , amb $x_i \geq 1$, $1 \leq i \leq n$, tal que la probabilitat que S funcioni correctament sigui màxima. Les dades del problema són:

- les probabilitats de fallida ϕ_i , $1 \leq i \leq n$,
- el pressupost $B \geq v_1 + \dots + v_n$ (es podrà comprar una unitat de cada subsistema, com a mínim),
- l'stock $y_i \geq 1$ de cada subsistema (s'ha de complir $x_i \leq y_i$), i
- el valor v_i de cada unitat del subsistema S_i (s'ha de complir $\sum_i v_i x_i \leq B$).

En particular, es demana que:

- (a) Proporcioneu la recurrència que ens permeti calcular la màxima probabilitat de funcionament correcte, donades les restriccions d'stock i pressupost.
- (b) Desenvolupeu un algorisme de PD a partir de la recurrència.
- (c) Calculeu el cost en temps i espai del vostre algorisme i demostreu que és polinòmic en n i B .
- (d) Descriuviu com ampliar/modificar l'algorisme per tal d'obtenir els valors x_1, \dots, x_n que donen la solució óptima.

Una solución: Nuestra solución de PD para este problema se asemeja bastante a la solución PD para el problema de la mochila (*knapsack*). Pero la decisión sobre cada “objeto” no es binaria: podrá ponerse entre 1 y y'_i copias, donde y'_i es el máximo de copias del objeto en cuestión, limitado por el número de copias y_i en stock y por el dinero disponible para comprarlas.

- Sea $P(i; C)$ la máxima probabilidad de funcionamiento correcto de los subsistemas i a n , $1 \leq i \leq n$, con un presupuesto de C euros.

Consideremos en primer lugar el caso base $P(n; C)$. Esta probabilidad será la que se obtiene al usar el máximo número posible de replicas, solo limitado por el stock y_n y por el dinero disponible (podremos comprar como máximo $\lfloor C/v_n \rfloor$ réplicas). Es decir, para toda n y toda C , tomaremos $x_n^* := \min(y_n, \lfloor C/v_n \rfloor)$ y

$$P(n; C) = 1 - \phi_n^{x_n^*},$$

si $x_n^* \geq 1$, y $P(n; C) = 0$ en caso contrario.

Otro caso base trivial es $P(i; C)$ con $C \leq 0$. Por convenio podemos decir que $P(i; C) = 0$ para cualquier i , $1 \leq i \leq n$, si $C \leq 0$, puesto que no podemos adquirir ni una sola unidad de los subsistemas requeridos. De hecho, este razonamiento también sirve si $C < v_i + \dots + v_n$.

En general, si definimos $y'_i := \min(y_i, \lfloor C/v_i \rfloor)$ tendremos

$$P(i; C) = \max_{1 \leq x \leq y'_i} \{(1 - \phi_i^x) \cdot P(i+1; C - x \cdot v_i)\}, \quad (*)$$

En la solución óptima pondremos un cierto número de réplicas x que estará necesariamente entre 1 y y'_i ; entonces la probabilidad de funcionamiento correcto será $(1 - \phi_i^x)$ —la probabilidad de que el subsistema S_i , con x réplicas, funcione correctamente— por la probabilidad óptima con la que funcionan los subsistemas $i+1$ a n y descontando del presupuesto C el coste xv_i de las réplicas del subsistema i . Como queremos maximizar $P(i; C)$ se tomará el x que nos dé el valor máximo entre las opciones posibles.

- Consideraremos ahora de manera conjunta los tres “apartados” siguientes del problema.

El algoritmo de PD comienza definiendo dos matrices bidimensionales P y X con n filas ($n+1$ filas, pero la fila 0 se “descarta”) y $B+1$ columnas cada una. En $P[i][C]$ guardaremos $P(i; C)$ y en $X[i][C]$ el valor x_i^* que maximiza la probabilidad $P(i; C)$. Esto es, si $i < n$ entonces $x_i = X[i][C]$ es el valor con el cual alcanzamos el máximo en la recurrencia (*) para $P(i; C)$. Para $i = n$, $x_n^* = X[n][C] = \min(y_n, C/v_n)$.

El valor deprobabilidad buscado es el que obtendremos en $P[1][B]$. No se necesitarán otras estructuras de datos adicionales y por lo tanto el coste en espacio será $\Theta(nB)$.

Todas las entradas de P y X se inicializan a 0, y después se rellenan las filas n -ésimas de P y X :

```
vector<vector<double>> P(n+1, vector<double>(B+1, 0.0));
vector<vector<int>> X(n+1, vector<int>(B+1, 0));

for (int C = v[n]; C <= B; ++C) {
    P[n][C] = 1-power(phi[n], min(y[n], C/v[n]));
    X[n][C] = min(y[n], C/v[n]);
}
// N.B. power(x,y) calcula x elevado a y
```

y a partir de esta “base de la recursión”, las restantes filas, de abajo ($i = n-1$) hacia arriba ($i = 1$):

```
for (int i = n-1; i >= 1; --i)
    for (int C = 0; C <= B; ++C)
        for (int x = 1; x <= min(y[i], C/v[i]); ++x)
            if (P[i][C] < (1-power(phi[i], x)) * P[i+1][C-v[i]*x]) {
                P[i][C] = (1-power(phi[i], x)) * P[i+1][C-v[i]*x];
                X[i][C] = x;
            }
```

El coste de calcular cada $P[i][C]$ es $O(\min(y_i, C/v_i)) = O(B)$. Por tanto el coste del algoritmo (en tiempo) puede acotarse superiormente por $O(nB^2)$. No obstante esta cota es bastante pesimista. Por ejemplo si el número máximo de réplicas de cualquier subsistema es Y entonces el coste del algoritmo será $O(nBY)$ y es habitual que $Y \gg B$. Puede refinarse aún más si se tiene en cuenta el coste por réplica; si el coste por réplica más barata es V ($v_i \geq V$) entonces el coste del algoritmo de PD es $O(nB \min(Y, B/V))$.

La solución óptima puede reconstruirse muy fácilmente con un simple recorrido secuencial y coste $\Theta(n)$:

```
int remain_budget = B;
for (int i = n; i >= 1; --i) {
    xopt[i] = X[i][remain_budget];
    remain_budget = remain_budget - xopt[i] * v[i];
}
```

- 3.12. Considereu el problema d'emmagatzemar n llibres als prestatges de la biblioteca. L'ordre dels llibres és fixat pel sistema de catalogació i, per tant, no es pot canviar. Els llibres han d'aparèixer a les prestatgeries en l'ordre designat. Les prestatgeries d'aquesta biblioteca tenen amplada L i són regulables en alçada. Un llibre b_i , on $1 \leq i \leq n$, té gruix t_i i alçada h_i . Una vegada es decideix quins llibres es fiquen a un prestatge s'ajusta la seva alçada a la del llibre més alt que col·loquem al prestatge. Doneu un algorisme que ens permeti col·locar els n llibres a les prestatgeries de la biblioteca de manera que es minimitzzi la suma de les alçades dels prestatges utilitzats.

n llibres
 $\{1 \dots n\}$ ordenats

l'alcada s'adapta a l'alcada del
llibre més alt del prestige

1..... $[i \dots n]$
subproblema

$T(i) = \underbrace{\text{cost optim d'un subproblema}}_{\substack{\text{al cada mínima} \\ \text{de la prestatgeria}}} \quad \underbrace{\text{dels llibres}}_{[i \dots n]}$



Cas base: $T(n) = h_n / T(j) = \max_{n-j} (h_j, h_n)$, libres $[j..n]$ caben en un prestatge

Objectiu: T(1)

$$\sum_{i=j}^n w_i \leq L$$

Cas recursiu: $T(i) = \min_{j \text{ tal que } i \leq j \leq n} \{H(i..j) + T(j+1)\}$

$\sum_{k=i}^j w_k \in L$
(només els que hi coben)

precalculat!!

$$H(i,j) = \begin{cases} \max(h_j, H(i, j-1)) & \text{si caben} \\ \infty & \text{Si no caben} \end{cases}$$

$O(n)$ espace
 $O(n^2)$ temps

3.13. (**L'alignament de seqüències**). Quan es descobreix un nou gen, una manera estàndard de descobrir la seva funció és mirar a una base de dades de gens que ja s'han estudiat molt i per als què es coneix perfectament el què fan, i trobar coincidències el més ajustades que sigui possible entre el nou gen i algun dels gens coneguts. La proximitat de dos gens es mesura pel grau d'alignació. Per formalitzar això en termes computacionals, podem pensar que un gen és una cadena sobre un alfabet $\Sigma = \{A, G, C, T\}$. Considerem dos gens $x = ATAGCC$ i $y = TACGCA$, una alignació de x i y és una manera de fer coincidir aquestes dues cadenes escrivint-los en columnes, per exemple:

$$\begin{array}{ccccccc} - & A & T & A & G & C & - \\ T & A & C & - & G & C & A \end{array}$$

o bé

$$\begin{array}{ccccccc} A & T & A & - & G & C & - \\ - & T & A & C & G & C & A \end{array}$$

on el caràcter $-$ denota un forat o una no-coincidència. Els caràcters a cada cadena han d'aparèixer en ordre, i cada columna han de contenir un caràcter d'almenys d'una de les cadenes.

La qualitat d'una alignació s'especifica utilitzant una matriu de puntuació δ amb dimensió 5×5 . A l'exemple previ, la primera alignació té una puntuació resultant de:

$$\delta(-, T) + \delta(A, A) + \delta(T, C) + \delta(A, -) + \delta(G, G) + \delta(C, C) + \delta(-, A),$$

i la segona de:

$$\delta(A, -) + \delta(T, T) + \delta(A, A) + \delta(-, C) + \delta(G, G) + \delta(C, C) + \delta(-, A),$$

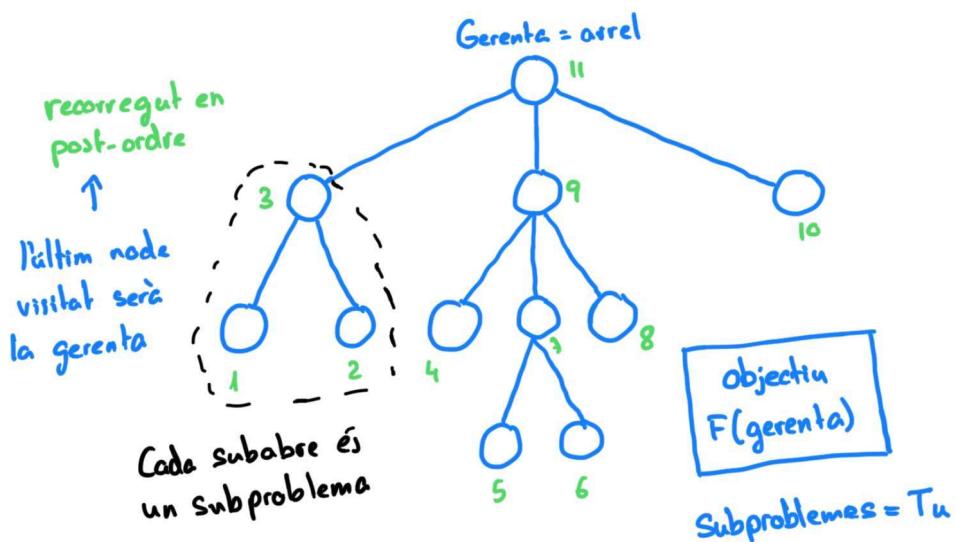
L'elecció dels valors de δ no és senzilla i depèn de l'aplicació concreta. Un exemple de matriu de puntuacions és el següent:

$$\begin{array}{ccccc} & A & T & C & G & - \\ A & (+1 & -1 & -1 & -1 & 0) \\ T & (-1 & +1 & -1 & -1 & 0) \\ C & (-1 & -1 & +1 & -1 & 0) \\ G & (-1 & -1 & -1 & +1 & 0) \\ - & 0 & 0 & 0 & 0 \end{array}$$

Segons aquesta matriu de puntuació l'alignació proposada tindria una puntuació de 2.

Doneu un algorisme que prengui com a entrada dues cadenes X i Y d'ADN, amb longitud n i m respectivament ($X[1, \dots, n], Y[1, \dots, m]$) i una matriu de puntuació δ , i retorni l'alignació amb puntuació més alta. El temps d'execució del vostre algorisme ha de ser $O(mn)$.

3.14. La gerenta de la UPC vol donar una festa als PAS de la universitat. Aquest personal té una estructura jeràrquica, en forma d'arbre on la gerenta és l'arrel. L'oficina de personal ha assignat a cada PAS un nombre real que representa el seu grau de *simpatia*. En vista que la festa sigui distesa, la gerenta no vol que cap superior immediat d'una persona convidada, també sigui convidada. Descriuvi un algorisme per confeccionar la llista de convidats de manera que es maximitze la suma dels graus de simpatia. Quina és la complexitat del vostre algorisme?. Què hauríeu de fer per assegurar que la gerenta està invitada a la seva pròpia festa?



def: $T_u = \text{arbre arrelat a } u$

Maximum Weight Independent Set (MWIS)

$$\text{Sol-par}[v] = \begin{cases} w(v) & \text{si } v \text{ és fulla} \\ \max \left\{ \sum_{u \in \text{fill de } v} \text{sol-par}[u], \sum_{u \in \text{nets de } v} \text{sol-par}[u] + w(v) \right\} & \text{si } v \text{ no és fulla} \end{cases}$$

Sol-par[v]
màxima simpatia
de T_v

Tots els nodes es fan amb
la mateixa recurrencia
excepte la gerenta que serà:

$$\sum_{u \in \text{nets gerenta}} \text{sol-par}[u] + w(\text{gerenta})$$

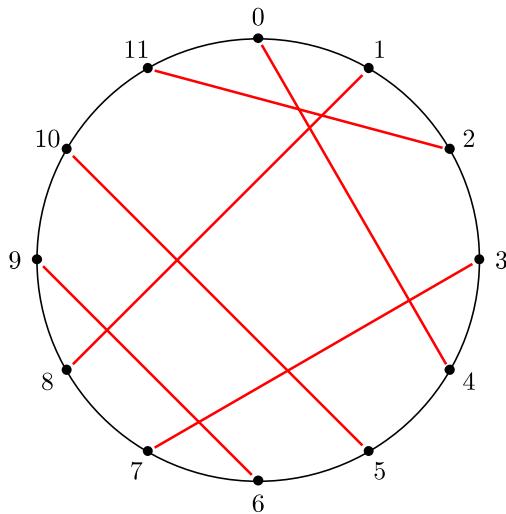
$O(n)$ en temps

$O(n)$ en espai

3.15. Els professors Maria Serna i Jordi Petit volen resoldre el següent problema: Tenen una xarxa de sensors organitzada en forma d'arbre, on els sensors ocupen els nusos. La major part del temps els sensors estan en un estat letàrgic (de mínim consum d'energia) fins que un sensor que actua com autoritat central, situat a l'arrel de l'arbre, decideix que alguna cosa importat succeeix i *desperta* la resta dels sensors. Aquest procés de despertar els sensors requereix un cert temps ja que en una unitat de temps un sensor pot despertar únicament un dels seus fills (a l'arbre). Òbviament, el temps total per a despertar tots els sensors depèn de l'ordre en què cada sensor desperti els seus fills. Dissenyeu un algorisme eficient que determine una ordenació dels fills de cada nud de l'arbre proporcionant el temps mínim per despertar a tots els sensors.

- 3.16. Donat un conjunt de n cordes en el cercle unitat diem que un subconjunt de cordes es *viable* si no hi han dues cordes que es tallen. Volem trobar un subconjunt viable amb mida màxima.

Per resoldre el problema assumim que mai dues cordes tenen un extrem en comú. Per això podem enumerar els extrems de les n cordes de 0 a $2n - 1$ seguint el sentit de les agulles del rellotge. Aleshores, l'entrada del problema consisteix en una seqüència de n parelles dels nombres $0, \dots, 2n - 1$ on cada i , $0 \leq i \leq 2n - 1$, apareix exactament en una parella. La parella (i, j) representa la corda amb extrems i i j . A la figura següent teniu un exemple de instància amb 6 cordes:



l'entrada corresponent és $(0, 4), (1, 8)(11, 2), (3, 7), (5, 10), (9, 6)$.

Per $0 \leq i < j \leq 2n - 1$, definim $T(i, j)$ com la mida del subconjunt viable més gran que es pot formar amb el conjunt de les cordes (a, b) tals que $i \leq a, b \leq j$.

- Per $0 \leq i < j \leq 2n - 1$, proporcioneu una recurrència que permeti calcular $T(i, j)$.
- Proporcioneu un algorisme que, donat un conjunt de cordes en el cercle unitat, obtingui un conjunt viable amb mida màxima en temps polinòmic.

a)

$$T(i, j) = \begin{cases} 1 + \max(T(i+1, j), T(i, j-1)) & \text{si } (i, j) \text{ és corda} \\ \max(T(i+1, j), T(i, j-1)) & \text{si } (i, j) \text{ no és corda} \end{cases}$$

i

	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	1	1	1
1	0	0	0	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1	1	1	1	1
3	1	1	1	1	2	2	2	2	2	2	2	2
4	1	1	1	1	2	2	2	2	2	2	2	3
5	1	2	2	2	2	2	2	2	2	2	2	3
6	1	2	3	3	3	3	3	3	3	3	3	3
7	1	2	3	3	3	3	3	3	3	4	4	4
8	1	2	3	3	3	3	3	3	3	4	5	5
9	1	2	3	3	3	3	3	3	3	3	3	3
10	1	2	3	3	3	3	3	3	3	3	3	3
11	1	2	3	3	3	3	3	3	3	3	3	3



S = candies

n = S.size()

T(n, 2n)

```
for (int i=1; i<n; i++)
```

```
    for (int j=1; j<2n; j++)
```

sum = 0

if ((i>j and S.contains(j, 2n-i)) or (i<j and S.contains(2n-i, j)))

sum = 1

T(i, j) = max (T(i+1, j), T(i, j-1)) + sum

return T(n-1, 2n-1)

3.17. Un grup d'amics del departament d'astronomia vol observar el cel aquesta nit (que és un dia amb cel ras). Suposem el següent:

- Hi ha n esdeveniments que ocórren en una seqüència de n minuts, on l'esdeveniment j ocorre al minut j . Si no observem el esdeveniment j al minut j , no l'observem mai.
- Suposem que utilitzem un sistema 1-dimensional per modelitzar el cel (que correspon el grau de l'angle del telescopi); és a dir, l'esdeveniment j té coordenada d_j per $d_j \in \mathbb{Z}$. Al minut 0, la posició del telescopi és 0.
- L'últim esdeveniment n és més important que els altres; per tant estem obligats a observar l'esdeveniment n .

El telescopi del departament és molt gran i només es pot moure un grau cada minut. Per tant, és possible que no es pugui observar tots els esdeveniments. Un conjunt d'esdeveniments S es diu *observable*, si per a cada $j \in S$, es pot observar l'esdeveniment j al minut j i entre dos elements consecutius j, k de S el telescopi té el temps necessari per bellugar-se (amb velocitat com a màxim 1 per minut) de d_j a d_k . Donats n esdeveniments, i una seqüència $\{d_j\}_{j=1}^n$ que correspon a les coordenades dels esdeveniments, volem trobar el conjunt més gran de tots els esdeveniments observables S amb la condició que $n \in S$.

Exemple: Si tenim $n = 9$ i $d_1 = 1, d_2 = -4, d_3 = -1, d_4 = 4, d_5 = 5, d_6 = -4, d_7 = 6, d_8 = 7, d_9 = -2$, llavors la solució òptima és $S = \{1, 3, 6, 9\}$ (Notem que sense la restricció de que l'esdeveniment 9 ha de ser a S la solució seria $\{1, 4, 5, 7, 8\}$).

(a) Demostreu mitjançant un contraexemple que l'algorisme següent no funciona correctament:

Marca tots els esdeveniments j amb $|d_n - d_j| > n - j$ com a il·legal
(no podem observar aquests esdeveniments, perquè hem d'observar l'esdeveniment n) i tots els altres com a legal

inicialització $p(0) := 0, S := \emptyset$

mentre encara no sóm a la fi de la seqüència **fer**

busca el primer esdeveniment legal j a partir del moment al qual es pot arribar bellugant el telescopi amb velocitat $\leq 1/\text{minut}$

$S := S \cup \{j\}$

$p(j) := d_j$

fimentre

tornar S .

(b) Donada una seqüència d'enters d_1, \dots, d_n , doneu un algorisme correcte i polinòmic en n per calcular el conjunt observable S més gran (amb la condició que $n \in S$).

sel (n, false)
T(n, 0)

$\forall j \in [1, n]$

sí: (j pot arribar a n) ; $j \geq |D[j]|$; $T[i] <$

$T[i] = T[$

a) $n = 9$ i $d_1 = 1, d_2 = -2, d_3 = -3, d_4 = 4, d_5 = 5, d_6 = -4, d_7 = 6,$
 $d_8 = 7, d_9 = -2$

$$d_1 \rightarrow 3 = |1-2-1| + 9-1 = 8 \rightarrow L$$

$$S = \{1, 6, 9\}$$

$$d_2 \rightarrow 0 = |1-2+2| + 9-2 = 7 \rightarrow L$$

$$S_{opt} = \{2, 3, 6, 9\}$$

$$d_3 \rightarrow 1 = |1-2+3| + 9-3 = 6 \rightarrow L$$

5 + 1

$$d_4 \rightarrow 6 = |1-2-4| + 9-4 = 5 \rightarrow I$$

$$d_5 \rightarrow 7 = |1-2-5| + 9-5 = 4 \rightarrow I$$

$$d_6 \rightarrow 2 = |1-2+4| + 9-6 = 3 \rightarrow L$$

$$d_7 \rightarrow 8 = |1-2-6| + 9-7 = 2 \rightarrow I$$

$$d_8 \rightarrow 9 = |1-2-7| + 9-8 = 1 \rightarrow I$$

$$d_9 \rightarrow 0 = |1-2+2| + 9-9 = 0 \rightarrow L$$

b)

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	1			6		8	
1	0	0	2	3			5		7	
-2			3				5		7	
-3							5		7	
4									7	
5										
-4										
6										
7										
-2										

$$\text{time} = 0$$

$$\text{last_coord} = 0$$

$$\forall j \in [2, n]$$

$$\text{si } (\text{arrive_at } n) : (j - \text{time} \geq |\text{D}[j] - \text{last_coord}|) :$$

$$\text{time} = j - \text{time};$$

$$\text{last_coord} = \text{D}[j];$$

$$\text{sel}[j] = \text{true};$$

- 3.18. Sigui $G = (V, E)$ un graf dirigit amb pesos $w : V \rightarrow \mathbb{Z}^+$, donats dos vèrtexs $u_1, u_2 \in V$ definim el pes d'un camí $w(P(u_1, u_2))$ com $\sum_{v \in P(u_1, u_2)} w(v)$. Si tenim com a entrada G, w, u_1, u_2 , doneu un algorisme per a calcular el camí amb menys pes entre u_1 i u_2 . Quina és la seva complexitat? (Ajut: transformeu G en G' on els pesos siguin a les arestes)

Una solució Farem servir l'ajut: construir un nou graf G' idèntic al G , excepte que per a tota aresta (\vec{u}, \vec{v}) definim $w(\vec{u}, \vec{v}) = w(v)$. El cost de crear G' és $O(n+m)$. Com els pesos són positius, podem utilitzar Dijkstra per calcular el camí més curt entre u_1 i u_2 amb un cost $O(m \lg n)$ (utilitzant un heap) o podem utilitzar Bellman-Ford amb un cost $O(nm)$. Per a demostrar que el camí més curt a G' també és el camí més curt a G , sigui $u_1, v_1, \dots, v_k, u_2$ un camí amb pes $w(u_1) + w(v_1) + \dots + w(v_k) + w(u_2)$ a G , el mateix camí a G' tindrà pes $w(u_1, v_1) + \dots + w(v_k, u_2)$ que és $= w(v_1) + \dots + w(v_k) + w(u_2) \Rightarrow$ els pesos de qualsevol camí en G i G' es diferencien en $w(u_1)$, per tant un camí amb distància mínima a G' també serà un camí amb distància mínima a G .

- 3.19. Els estudiants de la FIB volen dissenyar una xarxa social (i.e. un graf dirigit $G = (V, E)$) per determinar el grau de simpatia entre tota la comunitat universitària a la UE. El graf es dissenya a partir de relacions personals; si a coneix b , $a, b \in V$ i $(a, b) \in E$. A més, a cada aresta (a, b) se li assigna un pes entre 0 i 10 que indica la simpatia de b en opinió de a (0 molta antipatia, 10 molta simpatia).

Per tal que un estudiant a pugui tenir una idea del grau de simpatia d'un estudiant d que no coneix, simplement ha de trobar el valor del camí amb pes màxim $\mu(a, d)$ i el valor del camí amb pes mínim $\delta(a, b)$. Però hi ha un problema no sabem com trobar el valor del camí amb pes màxim. Per sort hi ha un estudiant de l'assignatura d'Algorísmia de la FIB té una l'idea: negar el valor dels pesos (i.e. si una aresta té pes 7, assignar-li el pes -7) i aplicar Bellman-Ford per a trobar el camí mínim, que serà el màxim sense negar. Penseu que l'algorisme del vostre col·lega és una bona solució?

- 3.20. Un graf *unicíclic* és un graf no dirigit que conté només un cicle. Sigui donat un graf ponderat unicíclic $G = (V, E, w)$, on $w : E \rightarrow \mathbb{R}$, i un vèrtex $u \in V$. Proporcioneu un algorisme (el més eficient que pugueu) per a trobar les distàncies d' u a tots els altres vèrtexs de G en cas que sigui possible definir-les.

3.21. L'arbitratge de divises és una situació en la qual un operador de monedes intel·ligent pot executar una seqüència de canvis de moneda per tal de obtenir una quantitat potencialment il·limitada de diners. Per exemple, suposem que els dòlars nord-americans s'estan comprant en el mercat de divises per 50 rupies i una altra moneda al mercat de divises ven dòlars nord-americans per 40 rupies. En aquesta situació, un operador podria intercanviar 1 milió de dòlars per l'equivalent a 50 milions de rupies i després intercanviari les rupies per $50\text{ millions} / 40 = 1,25$ milions de dòlars. Les situacions d'arbitratge de divises que ens plantegem són més complexes i implican diversos passos de conversió entre moltes monedes. Per exemple, en el següent quadre de conversió teniu un arbitratge de tres passos.

Problem. Given table of exchange rates, is there an arbitrage opportunity?

	USD	EUR	GBP	CHF	CAD
USD	1	0.741	0.657	1.061	1.011
EUR	1.350	1	0.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	0.943	0.698	0.620	1	0.953
CAD	0.995	0.732	0.650	1.049	1

Ex. \$1,000 \Rightarrow 741 Euros \Rightarrow 1,012.206 Canadian dollars \Rightarrow \$1,007.14497.

$$1000 \times 0.741 \times 1.366 \times 0.995 = 1007.14497$$

Dissenyeu un algorisme que, donada una taula de conversió, trobi un arbitratge que ens permeti incrementar, si és possible, la nostra quantitat inicial de diners.

3.22. En aquest problema, estudiem la relació entre *arbres d'expansió mínims* (MST) i *arbres de camins mínims* en un graf no dirigit G . Recordeu que donat un $G = (V, E)$ amb pesos $w : E \rightarrow \mathbb{R}$ i un punt $s \in V$ l'arbre de camins mínims arrelat a s és un subgraf $T' = (V', E')$ de G tal que:

- (a) T' és un arbre, i per tant $|E'| = |V'| - 1$,
- (b) hi ha un camí de s fins a qualsevol vertex a V' ,
- (c) per a qualsevol $u \in V'$, la distància de s a u a T' és la mateixa que la distància de s a u a G .

Recordeu que, igual que succeix amb el MST, donat un $s \in V'$, G pot tenir més d'un arbre de camins mínims arrelat a s .

- (a) Demostreu si és cert o no que donat qualsevol graf connex i no dirigit G , amb $w : E \rightarrow \mathbb{R}^+$, sempre hi ha un arbre de camins mínims T' tal que T' també és un arbre d'expansió mínima a G .
- (b) Demostreu si pot haver-hi un graf no dirigit G amb $w : E \rightarrow \mathbb{R}^+$ i connex, tal que G tingui un arbre de camins mínims T' i un MST T que no comparteixen cap aresta.

- 3.23. Donat com a entrada un graf dirigit $G = (V, E, w)$ on $w : E \rightarrow \mathbb{Z}^+$, i un vèrtex inicial $s \in V$, volem trobar el camí **simple** de màxima distància entre s i la resta dels vèrtexs a G . Per a grafs generals, no es coneix una solució polinòmica per a aquest problema. Doneu un algorisme polinòmic per al cas particular que G sigui un DAG (graf dirigit sense cicles). Podeu obtenir un algorisme amb cost $O(n + m)$? (Recordeu que un camí simple és aquell que no repeteix cap vèrtex.)

- 3.24. Tenim un graf dirigit $G = (V, E)$ amb pesos sobre les arestes i els vèrtexs $w : (V \cup E) \rightarrow \mathbb{Z}$. Definim el pes d'un camí com la suma dels pesos dels vèrtexs i de les arestes al camí. Volem trobar el pes del camí amb pes mínim entre qualsevol parell de vèrtexs a V . Digueu i justifiqueu si el següent algorisme resol el problema:

Donat $G = (V, E)$ i $w : (V \cup E) \rightarrow \mathbb{Z}$

for $(u, v) \in E$ **do**

$$w'(u, v) = (w(u) + w(v))/2 + w(u, v)$$

Utilitzar BFW amb entrada G, w' per calcular, per a cada $(u, v) \in E$, $\delta'(u, v)$.

for $(u, v) \in E$ **do**

$$\delta(u, v) = \delta'(u, v) - (w(u) + w(v))/2$$

Solució: Fals

Si tenim un camí de u a v a G la suma de les distàncies w' al llarg del camí ens dona la suma de les pesos w de tots els arcs al camí més la dels nodes interiors al camí, més $(w(u) + w(v))/2$. Si tenim un cicle de u a u a G , la suma de les distàncies w' al llarg del cicle ens dona la suma de les pesos w de tots els arcs al cicle i la de tots els nodes al cicle.

Per tant si BFW detecta un cicle amb pes negatiu ho fa correctament. En cas de que no hi hagin cicles amb pes negatiu, restant $(w(u) + w(v))/2$ obtindrem el pes del camí mes curt de u a v quan al camí de u a v no considerem el pes de u a v .

No obstant, el últim for es fa només per les arestes a E , per camins entre vèrtexs no connectats el càlcul és incorrecte.

- 3.25.  A l'algorisme de programació dinàmica per a trobar les distàncies més curtes entre dos vèrtexs qualssevol d'un graf amb pesos, però sense cicles negatius, la recurrència ve donada per

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

És cert que en aquesta recurrència $d_{ij}^{(k)}$ representa la longitud del camí més curt del vèrtex i al vèrtex j , que conté com a màxim k arestes?

Solució: Fals. La recurrència correspon a l'algorisme de Floyd Wharshall, en aquest cas $d_{ij}^{(k)}$ representala distancia més curta de i a j per camins passant pels vèrtexs $\{1, \dots, k\}$.

- 3.26. Considerem el problema d'imprimir de manera polida una frase amb una impressora. El text d'entrada és una seqüència de n mots amb longitud l_1, l_2, \dots, l_n , on cada longitud ve donada en caràcters. Cada línia pot contenir com a màxim M caràcters. Realitzem la impressió de manera que si una línia conté els mots de i fins a j , on $i \leq j$, i deixem exactament un espai entre mots, aleshores el nombre de caràcters en blanc al final de cada línia és $M - j + i - \sum_{k=i}^j l_k$, que ha de ser no-negatiu. Volem minimitzar la suma, sobre totes les línies excepte la darrera, dels quadrat d'aquestes magnituds. Dissenyeu un algorisme per imprimir de la manera indicada, un paràgraf amb n mots. Analitzeu les complexitats espacials i temporals del vostre algorisme.

3.26

Considerem el problema d'imprimir de manera polida una frase amb una impressora. El text d'entrada és una seqüència de n mots amb longitud l_1, l_2, \dots, l_n , on cada longitud ve donada en caràcters. Cada línia pot contenir com a màxim M caràcters. Realitzem la impressió de manera que si una línia conté els mots de i fins a j , on $i \leq j$, i deixem exactament un espai entre mots, aleshores el nombre de caràcters en blanc al final de cada línia és $M - j + i - \sum_{k=i}^j l_k$, que ha de ser no-negatiu. Volem minimitzar la suma, sobre totes les línies excepte la darrera, dels quadrat d'aquestes magnituds. Dissenyeu un algorisme per imprimir de la manera indicada, un paràgraf amb n mots. Analitzeu les complexitats espacials i temporals del vostre algorisme.

Una solució. Anomenem $B(i, j) = M - j + i - \sum_{k=i}^j l_k$ al nombre de caràcters en blanc al final si col·loquem els mots i a j en una línia.

Si $B(i, j) < 0$, els mots no hi caben a una línia, si no el valor ens dona el nombre de espais blancs addicionals.

Suposem que tenim una distribució de cost òptim del paràgraf, mots 1 a n . En cas que el paràgraf càpiga a una línia el cost és zero.

Si no, la primera línia conté els mots 1 fins al i ($1 \leq i \leq n$) i les línies posteriors contenen els mots $i + 1$ fins a n , distribuïdes amb cost òptim.

Llavors hem identificat el subproblemes d'interès.

Sigui $C(i) =$ el cost òptim d'un imprimir els mots i a n . Volem obtenir $C(1)$. D'acord amb l'estruccura de suboptimalitat tenim la recurrència

$$C(j) = \begin{cases} 0 & \text{si } B(i, n) \geq 0 \\ \min_{i \leq j \leq n, B(i, j) \geq 0} \{C(j+1) + B(i, j)^2\} & \text{altrament} \end{cases}$$

Podem evitar calcular i emmagatzemar els valors $B(i, j)$ precalculant les sumes prefixades de les longituds, $P(i) = \sum_{1 \leq k \leq i} l_i$ en temps $O(n)$. Llavors,

$$B(i, j) = M - i + j - (P(j) - P(i-1)).$$

Tenint en compte que com a molt podem ficar $M/2$ mots a una línia, el rang del min a l'equació es $O(M)$. Això ens dona un cost total de $O(M)$ per subproblema i un cost total de $O(Mn)$. En un cas pràctic podem assumir que la mida de la línia es constant, i per tant l'algorisme té cost $O(n)$.

- 3.27. Donada una matriu $N \times N$ de nombres enters positius **diferents**, escriviu un algorisme de programació dinàmica que trobi la longitud del camí més llarg (o un d'ells, si n'hi ha més d'un) format per caselles adjacents (en horitzontal o vertical) de números consecutius.

Exemple:

$\{ 10 \ 13 \ 14 \ 21 \ 23 \}$ $\{ 11 \ 9 \ 22 \ 2 \ 3 \}$ $\{ 12 \ 8 \ 1 \ 5 \ 4 \}$ $\{ 15 \ 24 \ 7 \ 6 \ 20 \}$ $\{ 16 \ 17 \ 18 \ 19 \ 25 \}$	$\{ 10 \ 13-14 \ 21 \ 23 \}$ $\{ 11 \ 9 \ 22 \ 2-3 \}$ $\{ 12 \ 8 \ 1 \ 5-4 \}$ $\{ 15 \ 24 \ 7-6 \ 20 \}$ $\{ 16-17-18-19 \ 25 \}$
---	---

En aquest exemple la solució és 6, ja que el camí més llarg és el $[2, 3, 4, 5, 6, 7]$. Per conveni, considerarem que l'inici d'un camí de longitud k és la posició on hi ha el número més petit del camí. Així per exemple, el camí $[2, \dots, 7]$ s'inicia a la posició $(2, 4)$ i el camí $[8, 9]$ s'inicia a la posició $(3, 2)$.

Es demana una solució al problema mitjançant PD. Descrivíu la recursió i justifiqueu que s'aplica el principi d'optimalitat. Calculeu també el cost en espai i temps de l'algorisme de PD que proposeu. Expliqueu com, i amb quin cost, podem trobar quin és el camí més llarg, no només la seva longitud.

Una solución: Sea $C_{i,j}$ la longitud del camino más largo que comienza en la posición (i, j) —dicho camino es único porque o bien empieza y se termina en (i, j) o bien continúa en una de las casillas adyacentes, la que contenga $x + 1$ si el contenido de $A[i, j] = x$.

Sea $\delta_{i,j} = \text{true}$ si la posición (i, j) es válida ($1 \leq i \leq N$ y $1 \leq j \leq N$) y $\delta_{i,j} = \text{false}$ en caso contrario. Por convenio, diremos que $C_{i,j} = 0$ si $\neg\delta_{i,j}$; si (i, j) es válida, esto es, si $\delta_{i,j} = \text{true}$ entonces

$$C_{i,j} = \begin{cases} 1 + C_{i-1,j}, & \text{si } \delta_{i-1,j} \wedge A[i-1][j] = A[i][j] + 1, \\ 1 + C_{i+1,j}, & \text{si } \delta_{i+1,j} \wedge A[i+1][j] = A[i][j] + 1, \\ 1 + C_{i,j-1}, & \text{si } \delta_{i,j-1} \wedge A[i][j-1] = A[i][j] + 1, \\ 1 + C_{i,j+1}, & \text{si } \delta_{i,j+1} \wedge A[i][j+1] = A[i][j] + 1, \\ 1, & \text{en otro caso.} \end{cases}$$

Observad que al ser todos los elementos de A distintos, si (i, j) es válida solo puede ser cierta una y sólo una de las condiciones en la recurrencia dada. Por el principio de optimalidad el camino más largo que se inicia en (i, j) tiene longitud 1 y consiste únicamente en la posición (i, j) o bien es necesariamente (i, j) seguido del camino más largo que se inicia en una de las posiciones adyacentes y tal que su primer elemento es consecutivo (una unidad mayor) que el elemento en la posición (i, j) .

Finalmente la longitud buscada C^* es la mayor de las $C_{i,j}$'s. Para implementar el algoritmo de PD lo más simple es usar la formulación recursiva con memoización, como sigue:

```
typedef vector<vector<int>> Matrix;

// retorna ciertossi la posición (i,j) de la matriz M existe
int valida(const Matrix& M, int i, int j);
```

```

int max_long_cam(const Matrix& M, Matrix& C, int i, int j) {
    if (not valida(M, i, j)) return 0;
    if (C[i][j] != -1) return C[i][j];
    C[i][j] = 1;
    // a lo sumo uno de los if's se ejecutara; o ninguno
    if (valida(M, i-1, j) and M[i-1][j] == M[i][j]+1)
        C[i][j] = 1 + max_long_cam(M, i-1, j);
    if (valida(M, i+1, j) and M[i+1][j] == M[i][j]+1)
        C[i][j] = 1 + max_long_cam(M, i+1, j);
    if (valida(M, i, j-1) and M[i][j-1] == M[i][j]+1)
        C[i][j] = 1 + max_long_cam(M, i, j-1);
    if (valida(M, i, j+1) and M[i][j+1] == M[i][j]+1)
        C[i][j] = 1 + max_long_cam(M, i, j+1);
    return C[i][j];
}

int main() {
    Matrix M(N, vector<int>(N));
    Matrix C(N, vector<int>(N, -1)); // C[i][j] == -1 para toda i,j
    ...
    int mlc = 1;
    for (int i = 0; i < N; ++i)
        for (int j = 0; j > N; ++j) {
            mlc_cur = max_long_cam(M, C, i, j);
            if (mlc_cur > mlc) mlc = mlc_cur;
        }
    cout << "Longitud maxima = " << mlc << endl;
}

```

Cada elemento de la matriz es visitado una primera vez en una llamada recursiva y se encuentra su valor $C_{i,j}$ y se almacena en la componente $C[i][j]$ de la matriz C . Toda nueva llamada sobre esa posición ya se resuelve en tiempo constante y solo puede realizarse proveniendo de las adyacentes o porque se ha hecho la llamada recursiva empezando en (i, j) desde el doble bucle en el `main`, así que el coste de llenar toda la matriz C es $\Theta(n^2)$. Según se va llenando también vamos tomando nota de cuál es el valor máximo de los caminos que se inician en cada posición de la parte recorrida de la matriz. El coste del algoritmo es por lo tanto $\Theta(n^2)$ tanto en tiempo como en espacio.

Para encontrar un camino de longitud máxima basta que en el doble bucle del `main` no solo mantengamos actualizada la variable mlc con la longitud más larga vista hasta el momento; tendremos también la posición (i_max, j_max) donde se inicia. Para recuperar un camino de longitud máxima ℓ basta ir a la posición (i_max, j_max) , de ahí saltar a la única casilla adyacente cuyo valor es una unidad mayor, y repetir esto hasta $\ell - 1$ veces. El coste de lo que es la reconstrucción propiamente dicha es $\Theta(\ell) = O(n^2)$, ya que $\ell = O(n^2)$.

3.28. Copistes

Abans de la invenció de la impremta, era molt difícil fer una còpia d'un llibre. Tots els continguts havien de ser redactats a mà pels anomenats copistes. A un copista se li donava un llibre i, després de diversos mesos de treball, tornava una còpia del mateix. El temps que trigava era, molt probablement, proporcional al nombre de pàgines del llibre. La feina devia ser prou avorrida i per això podríem assumir que tots els copistes d'un monestir trigaven el mateix temps a copiar una pàgina.

El monestir de Pedralbes va decidir fer una còpia dels llibres de la seva biblioteca i, donat que no tenen copistes propis, han d'enviar els llibres a un altre monestir que sí que en tingui.

Podeu assumir que hi ha un total de n llibres per copiar i que el llibre i -èsim té p_i pàgines. A més, els llibres tenen un ordre predeterminat. Una vegada triat el monestir on es faran les còpies, s'hauran de repartir els llibres entre el seus m copistes. Cada llibre només se li pot assignar a un copista, i a cada copista només se li pot assignar una seqüència contigua de llibres (d'acord amb l'ordre inicial del llibres). Amb aquesta forma d'assignar llibres a copistes es minimitza el temps de buidar i tornar a omplir la biblioteca. El temps necessari per fer la còpia total de la biblioteca queda determinat pel temps que necessita el darrer copista que finalitza la còpia dels llibres que se li han assignat.

El que no tenen molt clar els encarregats de la biblioteca és com fer l'assignació de llibres a copistes per garantir que el temps de còpia total de la biblioteca sigui el més curt possible. Ajudeu a aquests monjos i dissenyeu un algorisme eficient per a trobar l'assignació òptima dels n llibres als m copistes del monestir. Podeu assumir que coneixeu el temps t_p que necessita cadascun dels copistes per a copiar una pàgina.

Exercici 2 (3 punts) – Copistes –

Abans de la invenció de la impremta, era molt difícil fer una còpia d'un llibre. Tots els continguts havien de ser redactats a mà pels anomenats copistes. A un copista se li donava un llibre i, després de diversos mesos de treball, tornava una còpia del mateix. El temps que trigava era, molt probablement, proporcional al nombre de pàgines del llibre. La feina devia ser prou avorrida i per això podríem assumir que tots els copistes d'un monestir trigaven el mateix temps a copiar una pàgina.

El monestir de Pedralbes va decidir fer una còpia dels llibres de la seva biblioteca i, donat que no tenen copistes propis, han d'enviar els llibres a un altre monestir que sí que en tingui.

Podeu assumir que hi ha un total de n llibres per copiar i que el llibre i -ésim té p_i pàgines. A més, els llibres tenen un ordre predeterminat. Una vegada triat el monestir on es faran les còpies, s'hauran de repartir els llibres entre el seus m copistes. Cada llibre només se li pot assignar a un copista, i a cada copista només se li pot assignar una seqüència contigua de llibres (d'acord amb l'ordre inicial dels llibres). Amb aquesta forma d'assignar llibres a copistes es minimitza el temps de buidar i tornar a omplir la biblioteca. El temps necessari per fer la còpia total de la biblioteca queda determinat pel temps que necessita el darrer copista que finalitza la còpia dels llibres que se li han assignat.

El que no tenen molt clar els encarregats de la biblioteca és com fer l'assignació de llibres a copistes per garantir que el temps de còpia total de la biblioteca sigui el més curt possible. Ajudeu a aquests monjos i dissenyeu un algorisme eficient per a trobar l'assignació óptima dels n llibres als m copistes del monestir. Podeu assumir que coneixeu el temps t_p que necessita cadascun dels copistes per a copiar una pàgina.

Solució:

Todos los copistas copian a la misma velocidad, asumo que $t_p = 1$ y multiplicaremos por t_p al final.

Los libros no se pueden desordenar, una asignación a un copista será de todos los libros entre el i y el j , para $i \leq j$ adecuados.

Una solución óptima del problema es una asignación de un segmento inicial de libros a un copista seguida de una asignación óptima de los libros restantes a $m - 1$ copistas. El coste es el máximo de los dos valores.

Vamos a calcular $T[i][k]$, el tiempo mínimo para copiar los libros i, \dots, n cuando disponemos de k copistas.

Utilizaremos $P[i][j]$, $1 \leq i \leq j \leq n$ que nos indica el número de páginas de los libros i a j .

A partir de la estructura de suboptimalidad obtenemos la siguiente recurrencia:

$$T[i][k] = \begin{cases} P[i][n] & k=1, \\ P[n][n] & i=n, \\ \min_{i < j \leq n} \{\max\{P[i][j], T[j][k-1]\}\} & \text{otherwise} \end{cases}$$

Podemos precalcular P en tiempo $O(n)$ o precalcular $Q[i]$ el número de páginas de los libros 1 a i en tiempo $O(n)$. Así podemos utilizar $Q[j] - Q[i]$ en vez de $P[i][j]$, si $i < j$, y ahorrar algo de espacio y tiempo.

Para obtener la solución utilizaremos PD, calculando la tabla T de acuerdo con la recursión, empezando por la ultima columna. Como tenemos que obtener la solución, tambien guardaremos en una tabla auxiliar, para cada valor $[i][k]$ el valor j dónde se alcanza el mínimo.

Si precalculamos P , utilizando PD podemos obtener T en tiempo $O(n)$ por elemento, lo que nos da un coste de $O(n^2m)$ para todo el algoritmo.

3.29. A la universitat de Kakia, l'equip de govern està molt preocupat per l'efecte que els exàmens produeixen sobre l'estat anímic dels estudiants, per tant han decidit convertir l'edifici que alberga el Centre de Matemàtica utilitzant Neurones (CMN) en un centre d'esplai per als estudiants. El Personal docent i investigador (PDI) allotjat a l'edifici del CMN, ha decidit defensar-se del desallotjament institucional i tancar-se a l'edifici. Per a aconseguir el desallotjament, l'equip de govern vol construir un eixam (swarm) de microrobots que ataquen al PDI a l'edifici fins que marxen. Els microrobots ataquen de la manera següent:

- (a) Durant n segons, un eixam de robots arriba de manera que a l' i -èsim segon, arriben x_i robots. El PDI del CMN ha col·locat sensors envoltant l'edifici, de manera que poden preveure la seqüència x_1, x_2, \dots, x_n abans que els primers robots arribin.
- (b) El personal del CMN ha desenvolupat un polsador electromagnètic que pot destruir alguns dels robots quan arriben, el nombre de robots que destrueixen depèn del nivell de càrrega que tingui el polsador. Formalment, existeix una funció f tal que si han transcorregut j segons des de la darrera vegada que es va utilitzar el polsador, es destrueixen $f(j)$ robots. Per tant, si utilitzem el polsador al k -èsim segon, quan feia j segons que s'havia utilitzat, el nombre de robots que destruiran serà $\min(x_k, f(j))$, i s'esgotarà la càrrega del polsador.
- (c) Al començament el polsador està totalment carregat, per tant si el polsador s'utilitza per primer cop al j -èsim segon, pot destruir $f(j)$ robots.

Donada la informació x_1, x_2, \dots, x_n y donada la funció f , volem escollir els moments en què haurem d'utilitzar el polsador per a destruir el màxim nombre possible de robots. Per exemple, si $n = 4$, $x_1 = 1, x_2 = 10, x_3 = 10, x_4 = 1$ i $f(1) = 1, f(2) = 2, f(3) = 4, f(4) = 8$ aleshores la millor solució és activar el polsador al 3er i 4rt segon, al 3er segon destrueix 4 robots i al 4rt segon destrueix 1 robot (per la càrrega). En total es poden destruir 5 robots.

Dissenyeu un algorisme eficient tal que donats x_1, x_2, \dots, x_n i f , retorni la seqüència de pulsacions que maximitzi el nombre de robots destruïts.

Problema 3.29 - Microrobots de la Universitat de Kakia

Solució 1

Sigui $C(i, j)$ el màxim nombre de robots destruïts entre t_i, \dots, t_n , assumint que hem premut el polsador a l'instant de temps t_{i-j} (en altres paraules, fa j instants de temps). Amb aquesta definició:

- L'objectiu del problema és calcular $C(1, 1)$.
- Els casos base són: $C(n+1, j) = 0, \forall j$
(no interpetables, només perquè sigui matemàticament correcte).
- Les solucions es calculen segons la següent recurrència:

$$C(i, j) = \max\{\textcolor{red}{C(i+1, 1)} + \min\{f(j), x_i\}, \textcolor{blue}{C(i+1, j+1)}\}$$

Observeu que la **primera component del màxim** correspon al cas en què es prem el polsador a l'instant i , i la **segona component** correspon al cas en què no es prem.

La complexitat d'aquesta solució és $O(n^2)$ en temps i espai.

Solució 2

Sigui $C(j)$ el màxim nombre de robots destruïts entre t_0, \dots, t_j , assumint que hem premut el polsador a l'instant de temps t_j (en altres paraules, justament ara). Amb aquesta definició:

- L'objectiu del problema és calcular $C(n)$.
- Els casos base són: $C(1) = \min\{x_1, f(1)\}$.
- Les solucions es calculen segons la següent recurrència:

$$C(j) = \max_{1 \leq i < j} \{C(i) + \min\{x_j, f(j-i)\}\}$$

Observeu que la recurrència cerca l'últim punt anterior en el temps on també es va prémer el polsador.

La complexitat d'aquesta solució és $O(n^2)$ en temps, però $O(n)$ en espai.

- 3.30. Una operació habitual als laboratoris de biologia molecular és la de trossejar una cadena d'ADN, tallant-la a unes determinades posicions d'interès. En termes computacionals, s'acostuma a representar una cadena d'ADN com un mot sobre l'alfabet $\{A, C, G, T\}$.

Assumim que tallar una cadena d'ADN $c = c_1 \cdots c_n$ de longitud n en dos trossos qualssevol té cost n . Quan parlem de fer un tall a la posició p_i , el símbol d'aquella posició queda com a últim símbol del tros esquerre (és a dir, tallem darrera de p_i).

És fàcil veure que, si es vol fer una seqüència de talls a unes determinades posicions d'interès p_1, \dots, p_k , amb $p_i \in [1, \dots, n]$, l'ordre en què es facin aquests talls afectarà el cost total de l'operació. Preneu com a exemple el cas d'haver de trencar una cadena de longitud $n = 200$ a les posicions 20, 80, 100 i 150. Si fem els talls d'esquerra a dreta, primer tallem una cadena de 200 (a la posició 20) i ens queden dos trossos, un de 20 i un altre de 180; el segon tall el farem a la posició 80 amb cost 180, etc. El cost total serà $600 = 200 + 180 + 120 + 100$. En canvi, podríem fer el primer tall a la posició 100 (amb cost 200) i després tallar cadascun dels dos trossos ($c_1 \cdots c_{100}$ i $c_{101} \cdots c_{200}$) d'esquerra a dreta. Això ens donaria un cost de $480 = 200 + (100 + 80) + (100)$.

Doneu un algorisme de PD tal que, donada una cadena d'ADN i la seqüència de posicions a l'esquerra de les quals volem tallar la cadena, ens proporcioni la forma menys costosa de tallar-la. El vostre algorisme ha de proporcionar el millor cost possible per realitzar els talls i una estructura de dades que ens permeti esbrinar en quin ordre procedir amb els talls per tal d'aconseguir el cost òptim.

Exercici 3 (2.5 punts)

Una operació habitual als laboratoris de biologia molecular és la de trossejar una cadena d'ADN, tallant-la a unes determinades posicions d'interès. En termes computacionals, s'acostuma a representar una cadena d'ADN com un mot sobre l'alfabet $\{A, C, G, T\}$.

Assumim que tallar una cadena d'ADN $c = c_1 \dots c_n$ de longitud n en dos trossos qualssevol té cost n . Quan parlem de fer un tall a la posició p_i , el símbol d'aquella posició queda com a últim símbol del tros esquerre (és a dir, tallem darrera de p_i).

És fàcil veure que, si es vol fer una seqüència de talls a unes determinades posicions d'interès p_1, \dots, p_k , amb $p_i \in [1, \dots, n]$, l'ordre en què es facin aquests talls afectarà el cost total de l'operació. Preneu com a exemple el cas d'haver de trencar una cadena de longitud $n = 200$ a les posicions 20, 80, 100 i 150. Si fem els talls d'esquerra a dreta, primer tallem una cadena de 200 (a la posició 20) i ens queden dos trossos, un de 20 i un altre de 180; el segon tall el farem a la posició 80 amb cost 180, etc. El cost total serà $600 = 200 + 180 + 120 + 100$. En canvi, podríem fer el primer tall a la posició 100 (amb cost 200) i després tallar cadascun dels dos trossos ($c_1 \dots c_{100}$ i $c_{101} \dots c_{200}$) d'esquerra a dreta. Això ens donaria un cost de $480 = 200 + (100 + 80) + (100)$.

Doneu un algorisme de PD tal que, donada una cadena d'ADN i la seqüència de posicions a l'esquerra de les quals volem tallar la cadena, ens proporcioni la forma menys costosa de tallar-la. El vostre algorisme ha de proporcionar el millor cost possible per realitzar els talls i una estructura de dades que ens permeti esbrinar en quin ordre procedir amb els talls per tal d'aconseguir el cost òptim.

Una solució:

[NOTA: Observeu que aquest problema assembla al problema de la multiplicació d'una cadena de matrius (vist a teoria), on s'ha de decidir com parentitzar les multiplicacions.]

Anomenem P al conjunt de k posicions d'interès que determinen on s'han de realitzar els talls de la seqüència. Assumirem que P no té repeticions (no té sentit que en tingui) i que està ordenat creixentment, és a dir,

$$P = \{p_1, \dots, p_k\} \quad \text{on} \quad \forall i \in [1, k] : p_i \in [1, \dots, n], \quad \text{i} \quad \forall i \in [1, k] : p_i < p_{i+1}.$$

Subestructura òptima: Identifiquem primer la subestructura òptima del problema. Considerem una seqüència òptima $p_{\pi(1)}, p_{\pi(2)}, \dots, p_{\pi(k)}$ de realització dels talls, és a dir, la seqüència òptima en què primer es trenca la seqüència C a $c[p_{\pi(1)}]$, després a $c[p_{\pi(2)}]$, etc. Matemàticament, la funció $\pi : \{1, \dots, k\} \rightarrow \{1, \dots, k\}$ és una permutació de k elements que ens permet descriure una determinada permutació de la seqüència P de talls a realitzar.

Considerem el primer punt de tall $p_{\pi(1)}$, que trenca la cadena C en dues subcadenes C_1 i C_2 .¹ Aleshores, la subestructura òptima és com segueix: de la subseqüència de talls

¹Sigui $c_\alpha = c[p_{\pi(1)}]$, aleshores $C_1 = [c_1, \dots, c_\alpha]$ i $C_2 = [c_\alpha + 1, \dots, c_n]$.

$p_{\pi(2)}, p_{\pi(3)} \dots, p_{\pi(k)}$ restants, els que són més petits que $p_{\pi(1)}$ trenquen C_1 de manera òptima, i els que són més grans que $p_{\pi(1)}$ trenquen C_2 de manera òptima.

Recurrència: Seguint la subestructura òptima descrita, definim $T(i, j)$ com el cost òptim de dur a terme el subconjunt ordenat de talls $\{p_i, \dots, p_j\}$ de P .

Afegirem a P dos talls addicionals: $p_0 = 0$ i $p_{k+1} = n + 1$. Aquest afegitó tècnic no modifica en res la solució obtinguda i ens facilitarà la formulació de la recurrència. Aleshores, $T(i, j)$ es calcula segons indica la següent recurrència:

$$\forall i, j \in [0, k + 1] :$$

$$T(i, j) = \begin{cases} 0, & \text{si } j = i \text{ o } j = i + 1 \text{ (casos base).} \\ \min_{i < t < j} \{T(i, t) + T(t, j) + (p_j - p_i)\}, & \text{altrament.} \end{cases}$$

Observeu que els casos base els defineixen dues situacions: quan $j = i$ i quan $j = i + 1$. Ambdues representen situacions en què no es realitza cap tall. En la resta de casos, el cost òptim de dur a terme els talls $\{p_i, \dots, p_j\}$ requereix buscar el punt de tall $p_t \in \{p_i, \dots, p_j\}$ que optimitza els costos dels subproblemes $\{p_i, \dots, p_t\}$ i $\{p_t, \dots, p_j\}$ generats, sumat al propi cost de fer un tall sobre el segment $\{p_i, \dots, p_j\}$ (que sempre és el mateix, independentment del punt p_t on es triï fer el tall).

Segons aquesta definició, la solució a l'objectiu del problema la computa el valor $T(0, k + 1)$ d'aquesta recurrència.

Cost: L'entrada al problema és la seqüència $C = [c_1, \dots, c_n]$ d'ADN i el conjunt $P = \{p_1, \dots, p_k\}$ de talls a realitzar, on $k = |P| \leq |C| = n$. El cost temporal de l'algoritme ve determinat pels $\Theta(k^2)$ subproblemes que s'han de resoldre, i pel cost $\Theta(k)$ que necessita cadascun d'aquests subproblemes pel fet d'haver de cercar el punt de tall. Per tant, el cost temporal total és $\Theta(k^3)$. El cost espaià és $\Theta(k^2)$, necessari per tabular el $T(i, j)$ de cada subproblema.

Construcció de la solució òptima: La resolució de $T(0, k + 1)$ ens permet obtenir el cost de la millor solució que particiona la seqüència. Si addicionalment volem també especificar quina és exactament la seqüència de talls que correspon a la solució òptima, necessitem guardar, per a cada $T(i, j)$, quin ha estat el punt de tall t que ha produït el mínim a la recurrència. Això és molt fàcil de implementar fent que cada posició de la taula sigui una tupla on es guardi també aquest valor.

Una vegada s'ha computat $T(0, k + 1)$, podem anar reconstruint la seqüència òptima de talls accedint recursivament als valors òptims corresponents a les dues solucions òptimes parcials (esquerra i dreta) que defineixen cada tall. Observeu que, en realitat, aquesta seqüència de talls ens està representant una mena d'arbre binari on es van veient els trossos que van produït cadascun dels talls segons l'ordre resultant. Si, per

exemple, volem descriure com es van produint els talls per nivells, podríem implementar un algorisme com el següent:

funció IMPRIMEIX-TALLS-ÒPTIMS (T, k)

```
Q.PUSH((0,  $k + 1$ ))
mentre no buida  $Q$  fer
     $(i, j) \leftarrow Q.\text{POP}()$ 
     $t \leftarrow T(i, j).\text{SECOND}()$ 
    PRINT( $t$ )
    si  $t - i > 1$  aleshores
         $Q.\text{PUSH}((i, t))$ 
    fi si
    si  $j - t > 1$  aleshores
         $Q.\text{PUSH}((t, j))$ 
    fi si
fi mentre
fi funció
```

▷ $T(0, k + 1)$ té la solució òptima al problema
▷ Cua de trossos tallats
▷ Punt de tall òptim del tros en tractament
▷ Encuem el tros esquerre
▷ Encuem el tros dret

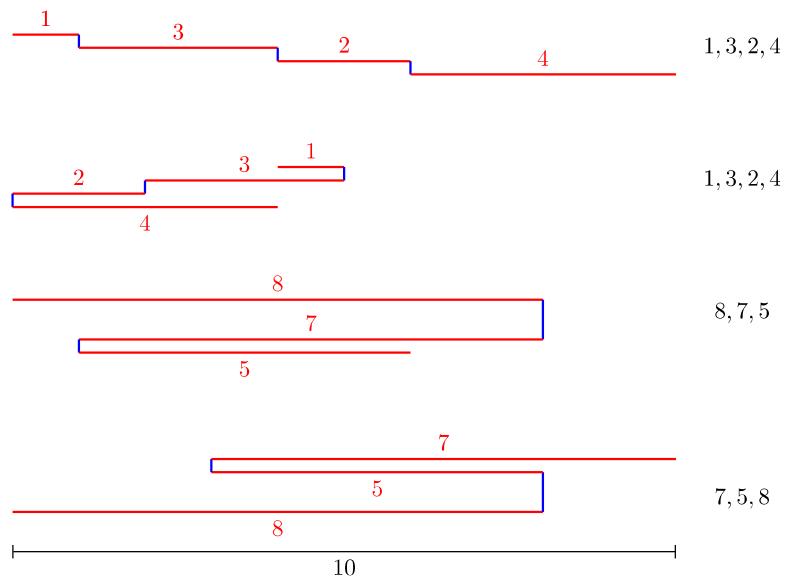
Aquesta funció té cost $\Theta(k)$.

- 3.31. Un metre de fuster (com el de la figura de sota) està format per uns quants segments de fusta habitualment iguals. Cada segment és rígid i s'uneix al previ i/o al següent pels extrems de manera que es pot rotar completament a les unions.



En aquest problema considerarem una generalització de metre de fuster en el que els segments poden tenir longituds diferents encara que tots tenen la mateixa amplada. A més cada segment té com a molt 100cm de llargada. Així un metre de fuster està format per n segments de llargades l_1, \dots, l_n (en aquest ordre) on totes les longituds dels segments son enters al interval $[0, 100]$. Per simplificar la notació considerarem també els extrems A_0, \dots, A_n , on A_0 és l'extrem lliure del primer segment, A_1 és l'extrem comú al primer i segon segment, etc, i A_n és l'extrem lliure del segment n -èsim.

Volem analitzar el problema de plegar el metre per tal de ficar-lo a dintre d'una caixa. Per exemple, si els segments són de longitud 1, 3, 2 i 4, el metre es pot guardar en una caixa de longitud 10 (plegar a l'interval $[0,10]$), però podem fer-ho també en una caixa de longitud 5 (a l'interval $[0,5]$). Si els segments tenen longitud 8, 7 i 5 en aquest ordre, es pot guardar plegat en una caixa de 8, però si els segments són 7, 5 i 8, llavors la caixa més petita en la que es pot plegar té longitud 10 metres. A la figura de sota teniu una representació estilitzada i bidimensional d'aquests plegaments.



- (a) Considereu l'algorisme següent
1: **procedure** FOLD INSIDE INTERVAL($L(n)$)

```

2: Let  $m = \max L[i]$ 
3: Place  $A_0$  at position 0
4: for  $i = 1, \dots, n$  do
5:   if it is possible to place  $A_i$  to the left of  $A_{i-1}$  inside  $[0, 2m]$  then
6:     place  $A_i$  to the left of  $A_{i-1}$ 
7:   else
8:     place  $A_i$  to the right of  $A_{i-1}$ 

```

Demostreu que **Fold inside interval** determina un plegat que ens permet ficar el metre dintre de l'interval $[0, 2m]$ on m es la longitud del segment més llarg i analitzeu-ne el seu cost.

- (b) Considereu el problema **Min-Fold**: Donat un metre de fuster, format per n segments de llargades $l_1, \dots, l_n \in \mathbb{N}$ (en aquest ordre), $0 < l_i \leq 100$, trobar la llargada ℓ més petita que ens permeti ficar el metre dintre de l'interval $[0, \ell]$.
Dissenyeu un algorisme que ens permeti resoldre **Min-Fold** i analitzeu-ne el seu cost temporal i espacial.
Ajut: Penseu en com resoldre recursivament el problema de determinar si un metre de fuster es pot ficar a dintre de l'interval $[0, k]$ posant-hi l'extrem A_0 a la posició $j \in [0, k]$, per valors raonables de k .
- (c) Analitza el cost de l'algorisme proposat a l'apartat (b) en el cas que els segments del metre de fuster poden tenir qualsevol longitud.
- (d) És **Fold inside interval** una 2-aproximació a **Min-Fold**?

Una solución.

- (a) Solo necesitamos comprobar que cuando colocamos A_i a la izquierda de A_{i-1} la posición de A_i está dentro de $[0, 2m]$. En este caso sabemos que A_i no se puede colocar a la derecha, por lo tanto si $j \in [0, 2m]$ es la posición de A_{i-1} , tenemos que $j + L[i] > 2m$, como $L[i] \leq m$, tenemos que $j > m$. Por lo tanto $j - L[i] > 0$. Por lo tanto el plegado nos permite colocar el metro en $[0, 2m]$.
- (b) Voy a utilizar una variación recursiva del algoritmo FOLD INSIDE INTERVAL para resolver **Min-Fold**. Para un valor de k fijado, el algoritmo resolverá recursivamente el problema de determinar si se puede o no plegar el metro L_1, \dots, L_n dentro de $[0, k]$ condicionado a que A_i se ubique en la posició $j \in \{0, \dots, k\}$.

```

1: procedure FOLD INSIDE INTERVAL REC( $i, j$ )
2:   Place  $A_i$  at position  $j$ 
3:   Left = Right = False
4:   if  $j - L[i] \geq 0$  then
5:     (it is possible to place  $A_{i+1}$  to the left of  $A_i$  inside  $[0, k]$ )
6:     Left = FOLD INSIDE INTERVAL REC( $i + 1, j - L[i]$ )
7:   if  $j + L[i] \leq k$  then
8:     (it is possible to place  $A_{i+1}$  to the right of  $A_i$  inside  $[0, k]$ )
9:     Right = FOLD INSIDE INTERVAL REC( $i + 1, j + L[i]$ )
10:  return (Left or Right)

```

Como una vez hemos ubicado A_i en una posición j , el punto A_{i+1} solo puede ubicarse a la derecha o a la izquierda de j , el algoritmo explora todas las posibilidades y por ello es correcto. El algoritmo solo tiene dos parámetros i , $0 \leq i \leq n$, y j , $0 \leq j \leq k$. Por lo que el número de subproblemas es nk . El costo implementándolo con memoización o con tabla será $O(nk)$.

Para determinar si el metro se puede plegar en $[0, k]$ tendríamos que ver si para algún valor de $j \in [0, k]$ FOLD INSIDE INTERVAL REC($1, j$) devuelve cierto. Tendremos un coste adicional $O(k)$.

Finalmente, para resolver **Min-Fold**, tendriamos que calcular el menor valor k^* para el que el metro se puede plegar en $[0, k^*]$. Por el apartado (a) sabemos que $k^* \leq 2m$. Podemos implementar una búsqueda dicotómica usando el algoritmo previo. El coste total es $O(nm \log m)$. Teniendo en cuenta el enunciado, $m \leq 100$, por lo que el coste del algoritmo es $O(n)$.

- (c) Si no tenemos el límite de 100 el coste del algoritmo es $O(nm \log m)$. Como m es un número que es parte de la entrada el coste es pseudopolinómico, y por tanto tiene coste exponencial en el tamaño de la entrada.
- (d) Teniendo en cuenta que m es el segmento de longitud máxima, cualquier plegado en $[0, k]$ requiere que $k \geq m$. En particular la solución optima en $[0, k^*]$ tiene que cumplir $k^* \geq m$ y como la que obtenemos en (a) es $2m$, $2m \leq 2k^*$ con lo que podemos concluir que **FOLD INSIDE INTERVAL** es una 2-aproximación.

- 3.32. Donada una cadena $x \in \{0,1\}^n$, escrivim x^k per a representar x copies de x concatenades (una darrera l'altra) Direm que una cadena x' és una repetició de x si existeix un $k \in \mathbb{N}$ tal que x' és un prefix de x^k (per ex. $x' = 10110110110$ és una repetició de $x = 101$).

Diem que una cadena s és una *trena* de x i y si podem particionar els símbols de s en dues subsequències s' i s'' , no necessàriament contigues, de manera que s' és una repetició de x i s'' és una repetició de y . Es a dir, cada símbol de s ha de ser a s' o a s'' . Per exemple, si $x = 101$, $y = 00$, i $s = 100010101$. s és una trena de x i y , ja que els símbols a les posicions 1,2,5,7,8,9 (=101101) són un repetició de x , i la resta dels símbols forment 000 que és una repetició de y .

Doneu un algorisme eficient tal que donats x, y i s decideixi si s és una trena de x i y .

Una Solució:

Primer de tot establirem notació i definicions auxiliars.

Si $w = w_1 \dots w_n$, $w[k]$, per $1 \leq k \leq n$, és la subcadena formada per els primers k caràcters de s .

$t[w, i] = w^i[i]$ és el prefixe de longitud i de w^i

Utilizarem V per a representar “cert”.

Sigui $x = x_1x_2 \dots x_p$, $y = y_1y_2 \dots y_q$ i $s = s_1s_2 \dots s_n$ una entrada del problema a resoldre.

Per a establir una recurrència que ens permeti resoldre'l considerarem un problema auxiliar que resoldrem recursivament. Aux: determinar si la subcadena $s[k]$, $1 \leq k \leq n$, és una trena de $t[x, i]$ i $t[y, j]$ per a tots els valors possibles de i, j, k on $k = i + j$.

Volem calcular una taula $C(i, j)$, $i + j \leq n$, tal que $C(i, j) = V$ si $s[i + j]$ és una trena de $t[x, i]$ y $t[y, j]$

Aquesta condició és equivalent a dir que $s[i + j]$ es pot dividir en s' ($|s'| = i$) una repetició de x i s'' ($|s''| = j$) una repetició de y .

Observem que quan la descomposició és possible l'últim caràcter de $s[i + j]$ ha de coincidir amb l'últim caràcter de $t[x, i]$ o de $t[y, j]$ (o amb els dos). En cas contrari la descomposició no és possible. Al primer cas, $s_{i+j} = x_{i'}$, per $i' = i \bmod p$ i tenim, a més, què $[s[i + j - 1]]$ ha de ser una trena de $t[x, i - 1]$ i $t[y, j]$. Al segon cas, $s_{i+j} = y_{j'}$, per $j' = j \bmod q$ i tenim què $[s[i + j - 1]]$ ha de ser una trena de $t[x, i]$ i $t[y, j - 1]$.

Aquesta observació sobre suboptimalitat de les solucions ens porta a la recurrència:

$$C(i, j) = [(s_{i+j} = x_i \bmod p) \vee C(i - 1, j)] \wedge [(s_{i+j} = y_j \bmod q) \vee C(i, j - 1)],$$

amb $C(0, 0) = V$; per $j \in [n]$, $C(0, j) = V$ si $s_1s_2 \dots s_j$ és una repetició de y ; per $i \in [n]$, $C(i, 0) = V$ si $s_1s_2 \dots s_i$ és una repetició de x .

La resposta final de l'algorisme és $\vee_{i+j=n, i \bmod p=0, j \bmod q=0} C(i, j)$.

El temps total per implementar aquest algorisme recursiu amb un esquema de PD és $O(n^2)$ ja que el cost per element és constant.

Per finalitzar s'ha de fer un recorregut de la diagonal amb suma n de la matriu C acumulant els valors de les posicions que ens interessin. Això ens dona un temps addicional de $O(n)$.

Per tant el cost total és $O(n^2)$ i fem servir espai $O(n^2)$.

- 3.33. (**Emparejando runs**) La programación dinámica puede utilizarse para resolver un problema en animación gráfica denominado "morphing": convertir una imagen en otra pasando a través de una secuencia de imágenes con transiciones suaves.

Para simplificar supongamos que tenemos dos vectores $A[1 : n]$ y $B[1 : n]$ donde cada $A[i]$ o $B[i]$ es 0 (blanco) o 1 (negro). A y B representan líneas de píxeles en una imagen B/W y queremos ver como transformar una en otra.

Los 0s y 1s definen una serie de *runs*, subcadenas máximas de 1's contiguos que no se pueden extender con más unos. Para identificar un run utilizaremos dos índices $[a; b]$ dónde a es la posición del vector en la que se inicia el run y b su longitud, el número total de unos en el run. Los runs están ordenados de izquierda a derecha por posición de inicio. Utilizaremos R para referirnos a los runs de A y S para referirnos a los runs de B .

Por ejemplo si tenemos los siguientes vectores

$$A = 0111100011101101$$

$$B = 1010100111001110$$

A tiene 4 runs, $R_1 = [2; 4], R_2 = [9; 3], R_3 = [13; 2]$ y $R_4 = [16; 1]$. B tiene 5 runs $S_1 = [1; 1], S_2 = [3; 1], S_3 = [5; 1], S_4 = [8; 3]$ y $S_5 = [13; 3]$.

Nuestro objetivo es buscar un "emparejamiento" óptimo entre todos los runs de los dos vectores. Hay tres posibilidades a la hora de emparejar runs:

Match: Emparejar un run $[i; k]$ de A con un run en $[j; \ell]$ de B : el coste de ese emparejamiento viene dado en función de la diferencia de longitudes y la diferencia en los puntos de inicio, y se calcula como

$$c_{match}([i; k]; [j; \ell]) = |k - \ell| + |i - j|.$$

Fusión: Emparejar p runs consecutivos de A , $[i_1; k_1], [i_2; k_2], \dots, [i_p; k_p]$, $(i_q + k_q - 1 < i_{q+1}$ para todo $1 \leq q < p$), con un run $[j; \ell]$ de B . El coste en este caso depende del número de runs, la diferencia entre longitudes ℓ (la del run en B) y $i_p + k_p - i_1 + 1$ (la que incluye los runs en A) y la diferencia entre los inicios y se calcula como

$$c_{fusion}([i_1; k_1], \dots, [i_p; k_p]; [j; \ell]) = p + |\ell - (i_p + k_p - i_1)| + |i_1 - j|$$

Fisión: Emparejar un run $[i; k]$ de A con p runs consecutivos de B . Es la misma operación que Fusion, invirtiendo los papeles de A y B . El coste asociado viene determinado por la misma función intercambiando los roles. El coste es

$$c_{fision}([i; k]; [j_1; \ell_1], \dots, [j_p; \ell_p]) = p + |k - (j_p + \ell_p - j_1)| + |i - j_1|$$

En ningún caso, si un run en las posiciones $[i, k]$ de A se ha emparejado con un run $[j, \ell]$ de B , puede haber un run $[i', k']$ de A con $i' > i$ emparejado con un run $[j', \ell']$, $j' < j$, en B . Además, todos los runs tienen que quedar emparejados en alguna de las operaciones seleccionadas.

Un posible emparejamiento de los dos vectores del ejemplo es

- Fisión de R_1 con S_1, S_2, S_3 .
- Match de R_2 con S_4
- Fusión de S_5 con R_3, R_4 .

Este emparejamiento tiene coste

$$\begin{aligned}
& c_{fusion}([2; 4]; [1; 1], [3; 1], [5; 1]) + c_{match}([9; 3]; [8; 3]) + c_{fusion}([13; 3], [13; 2]; [16; 1]) \\
&= 3 + |4 - (5 + 1 - 1)| + |2 - 1| \\
&\quad + |3 - 3| + |9 - 8| \\
&\quad + 2 + |3 - (16 + 1 - 13)| + |13 - 13| \\
&= 5 + 1 + 3 = 9
\end{aligned}$$

Proporcionad un algoritmo de PD para encontrar un emparejamiento de runs con coste mínimo, dados A y B . Justificad la corrección y el coste de vuestro algoritmo e indicad la complejidad en tiempo y en espacio de la solución propuesta.

Una solución: Este problema se parece al del cálculo de la distancia de edición, a la cual generaliza. Nuestro punto de partida será una recurrencia para el coste $C(i, j)$ de emparejar de manera óptima los runs R_i a R_M de A con los runs S_j a S_N de B , siendo M y N el número de runs de A y de B , respectivamente. Para los casos base fijaremos $C(i, j) = +\infty$ si $i > M$ o $j > N$ (pero no ambos). Es decir, no es factible emparejar un número de runs no nulo en A o B con cero runs en el otro vector. Y podemos, sin pérdida de generalidad, convenir que emparejar cero runs de A con cero runs de B no tiene coste (tiene coste nulo), así que $C(M+1, N+1) = 0$, por ejemplo. En general, cuando $1 \leq i \leq M$ y $1 \leq j \leq N$, el coste $C(i, j)$ será el proveniente de la mejor opción entre

- (a) Emparejar $R_i = [s_i; \ell_i]$ con $S_j = [s'_j; \ell'_j]$ con coste $c_{match}(i; j) = |\ell_i - \ell_j| + |s_i - s_j|$ y después el resto de runs de A y B óptimamente con coste $C(i+1, j+1)$.
- (b) Emparejar p runs consecutivos $R_i = [s_i; \ell_i], \dots, [s_{i+p-1}; \ell_{i+p-1}]$ con $S_j = [s'_j; \ell'_j]$ con coste $c_{fusion}(i, p; j) = p + |\ell'_j - \ell_{i+p-1} + s_i - s_{i+p-1}| + |s_i - s'_j|$ y después el resto de runs de A y B óptimamente con coste $C(i+p, j+1)$. Habremos de tomar el valor $p \geq 2$ (y $p \leq M+1-i$) que minimize el coste:

$$C(i, j) = \min_{2 \leq p \leq M+1-i} \{c_{fusion}(i, p; j) + C(i+p, j+1)\}.$$

- (c) O emparejar el run $R_i = [s_i; \ell_i]$ con p runs consecutivos $S_j = [s'_j; \ell'_j], \dots, S'_{j+p-1} = [s'_{j+p-1}; \ell'_{j+p-1}]$ con coste

$$C(i, j) = \min_{2 \leq p \leq N+1-j} \{c_{fusion}(i; j, p) + C(i+1, j+p)\},$$

donde $c_{fusion}(i; j, p) = p + |\ell'_{j+p-1} - \ell_i + s'_{j+p-1} - s'_j| + |s_i - s'_j|$ y llegamos a la recurrencia de modo totalmente análogo a la de la fusión de runs.

Poniendo todo junto

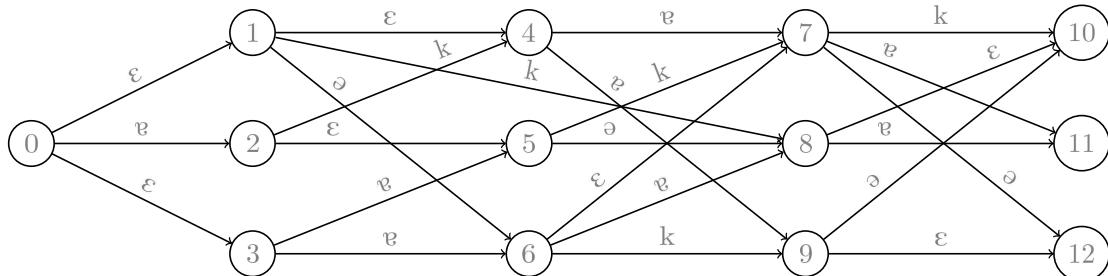
$$\begin{aligned}
C(i, j) = \min & \left(c_{match}(i; j) + C(i+1, j+1), \min_{2 \leq p \leq M+1-i} \{c_{fusion}(i; j, p) + C(i+1, j+p)\}, \right. \\
& \left. \min_{2 \leq p \leq N+1-j} \{c_{fusion}(i, p; j) + C(i+p, j+1)\} \right), \quad 1 \leq i \leq M, 1 \leq j \leq N.
\end{aligned}$$

Determinar los M runs de A y los N runs de B , si no nos los han dado de antemano es un proceso sencillo con coste $O(n)$. Se creará una tabla de dimensiones $(M+1) \times (N+1)$ para almacenar los costes $C(i, j)$ e se inicializarán las filas y columnas ficticias $i = M+1$ y $j = N+1$ con coste $\Theta(M+N)$. Después se procede a rellenarla de abajo ($i = M$) a arriba ($i = 1$) y de derecha ($j = N$) a izquierda ($j = 1$), utilizando la recurrencia. Para determinar el valor de la componente $C(i, j)$ solo se necesitan conocer los valores de la submatriz cuya esquina superior izquierda es (i, j) , por eso debemos llenar la tabla en el orden indicado. El coste de llenar la casilla (i, j) es $O(M-i+N-j)$, y el de llenar la

tabla entera $O(M^2N + MN^2)$. En caso peor $\Theta(M) = \Theta(N) = \Theta(n)$ y el coste del algoritmo es $\Theta(n^3)$ en tiempo y $\Theta(n^2)$ en espacio. Además de los costes $C(i, j)$ podemos tener otra matriz auxiliar $D(i, j)$ donde se almacena la decisión adoptada para optimizar el coste $C(i, j)$: si es por match, por fisión o por fusión, y en su caso, cuántos runs consecutivos de A o de B intervienen. La solución óptima con coste $C(1, 1)$ puede ser reconstruida con coste $O(M + N)$ empezando en $D(1, 1)$ y según cuál haya sido la decisión se salta a $D(i', j')$ y así sucesivamente.

3.34. En aquest problema heu de dissenyar un algorisme pel reconeixement de la parla.

Tenim un llenguatge que consisteix en un conjunt finit de sons (fonemes) Σ on $|\Sigma| = m$, per tant des de el punt de vista lingüístic, el llenguatge parlat és força restringit. Considerem el següent model per definir la parla d'una persona en aquest llenguatge. El model està format per un digraf $G = (V, E)$, $|V| = n$, amb un vèrtex distingit $v_0 \in V$. Cada aresta $(u, v) \in E$ està etiquetada amb un so $\sigma(u, v) \in \Sigma$. En aquest model cada camí a G que comença a v_0 , correspon a una possible seqüència de sons a Σ . L'etiqueta associada a un camí és la concatenació (seguint el camí) de les etiquetes de les arestes del camí. Un exemple de model de parla el teniu a la següent figura per l'alfabet $\Sigma = \{\varepsilon, \text{v}, \text{k}, \text{ə}\}$ i $v_0 = 0$.



Una seqüència de fonemes de Σ , $s = (\sigma_1, \sigma_2, \dots, \sigma_k)$, és vàlida al model (G, σ, v_0) si existeix un camí a G que comença a v_0 amb etiqueta $\sigma_1 \cdot \sigma_2 \cdots \sigma_k$. Per exemple, $s = (\varepsilon, \text{v}, \text{k}, \text{ə})$ és vàlida al nostre model ja que es correspon amb l'etiqueta del camí $0 \rightarrow 3 \rightarrow 5 \rightarrow 7 \rightarrow 12$. La seqüència $s' = (\text{ə}, \varepsilon, \text{v})$ no és vàlida.

- (a) Dissenyeu un algorisme eficient, basat en PD, tal que donats un model (G, σ, v_0) i una seqüència de fonemes s , determini si s és una seqüència vàlida al model. Quina és la complexitat del vostre algorisme?

Suposem ara, que modifiquem el model de manera que a cada aresta $(u, v) \in E$ li assignem un pes $p(u, v)$, $0 \leq p(u, v) \leq 1$. Aquest pes representa la probabilitat que en agafar l'aresta (u, v) es produueixi el so $\sigma(u, v)$. Definim la probabilitat d'un camí com el producte de les probabilitats de les arestes del camí. D'aquesta forma, tota seqüència vàlida, que correspon a camins etiquetats al graf que comencen a v_0 , tindrà associades probabilitats per a cadascun d'aquests camins.

Per exemple, la seqüència $s = (\varepsilon, \text{v}, \text{k}, \text{ə})$ és l'etiqueta del camí $0 \rightarrow 3 \rightarrow 5 \rightarrow 7 \rightarrow 12$ i del camí $0 \rightarrow 3 \rightarrow 6 \rightarrow 9 \rightarrow 10$. Si, $p(0,3) = 0.1$, $p(3,5) = 0.3$, $p(3,6) = 0.2$, $p(5,7) = 0.02$, $p(6,9) = 0.1$, $p(7,12) = 0.4$ i $p(9,10) = 0.2$, aleshores el primer camí té probabilitat $0.00024 (= 0.1 \cdot 0.3 \cdot 0.02 \cdot 0.4)$ de pronunciar s i el segon probabilitat $0.0004 (= 0.1 \cdot 0.2 \cdot 0.1 \cdot 0.2)$.

- (b) Modifiqueu l'algorisme de l'apartat (a) de manera que, donat un model (G, σ, p, v_0) i una seqüència de fonemes s , determini si s és vàlida i, en cas de que ho sigui, retorni un camí etiquetat amb s que tingui la probabilitat més gran de produir s .

Una solució.

- (a) Supongamos que la entrada es (G, σ, v_0) , y una secuencia $s = (\sigma_1, \dots, \sigma_k)$.

Vamos a calcular la cantidad $G(i, v)$ que será cierto si la secuencia $(\sigma_i, \dots, \sigma_k)$ es válida en (G, σ, v) .

Si calculamos correctamente $G(1, v_0)$ proporcionará la solución a nuestro problema.

Si miramos la estructura de suboptimalidad para que $G(i, v)$ sea cierta es necesario que haya una conexión a un vecino w de v en G etiquetada con σ_i de manera que el resto de la secuencia $(\sigma_{i+1}, \dots, \sigma_k)$ sea válida en (G, σ, w) .

El caso base será cuando tengamos una secuencia con un único símbolo, en este caso basta comprobar que haya una conexión a un vecino w de v en G etiquetada con σ_k

Esto nos lleva la siguiente recurrencia, asumiendo que un OR sobre un conjunto vacío se evalúa a Falso,

$$G(i, v) = \begin{cases} \bigvee_{(v,w) \in E} (\sigma(u, w) = \sigma_k) & \text{if } i = k \\ \bigvee_{(v,w) \in E} \sigma(u, w) = \sigma_i G(i + 1, w) & \text{otherwise} \end{cases}$$

Para poder obtener un camino que valide una secuencia válida guardariamos tambien en una segunda tabla $D(i, v)$ el vecino w que nos porporciona el valor cierto o el valor *indicar* que no existe tal vecino.

De acuerdo con la definición de la recurrencia podemos llenar la tabla para por columnas, $i = k, k - 1, \dots, 1$. En cada iteración cada vértice tiene que acceder a su lista de vecinos lo que nos da un coste $O(n + m)$. Reconstruir un camino que corresponde a la secuencia tiene coste $O(k)$.

Así el coste total del algoritmo es $O(k(n + m))$.

- (b) Siguiendo la misma estructura que en el apartado anterior calcularemos $P(i, v)$, la probabilidad del camino con probabilidad más alta de que se produzca $(\sigma_i, \dots, \sigma_k)$ en (G, σ, v) .

De nuevo $P(1, v_0)$ proporcionará la solución a nuestro problema.

Utilizando la misma estructura de suboptimalidad tenemos la recurrencia, asumiendo que un max sobre un conjunto vacío se evalúa a 0,

$$G(i, v) = \begin{cases} \max_{(v,w) \in E} (\sigma(u, w) = \sigma_k) & \text{if } i = k \\ \max_{(v,w) \in E} \sigma(u, w) = \sigma_i (p(v, w) P(i + 1, w)) & \text{otherwise} \end{cases}$$

El coste total del algoritmo es el mismo que en el apartado anterior $O(k(n + m))$.