Problemes resolts 3

3.1. **(Sistema discret)**

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 te 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] ≥ b[1] then
3:          s[1] = 'A'
4:      else
5:          s[1] = 'B'
6:      i = 2
7:      while i ≤ 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 $a_1, \ldots, a_n$ i $b_1, \ldots, b_n$, proporcioni un pla d'execució que permeti executar el màxim nombre de passos de simulació.

Una solució:

(a) Per als vectors $a =< 2, 1 >$ i $b =< 2, 20 >$, suposant que l'accés fora de rang no falla, el programa retornaría la solución $< A, A >$ que és incorrecta.

(b) Para encontrar una solución óptima analizamos su estructura de suboptimalidad. Observemos que una solución óptima ejecutará pasos de simulación en el instante n, si no no seria óptima. Puede hacerlo en A on 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 ejecutándose en A en el

paso $n-1$ y en el segundo una solución óptima para $n-2$ pasos ejecutándose 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:

$$A(k) = a(k) + \max(A(k-1), B(k-2))$$
$$B(k) = b(k) + \max(B(k-1), A(k-2))$$

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. **(Partició equilibrada)**

> 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 $\overline{V}_1 = V \setminus V_1$ 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 solució:

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 $V = \{v_1, \ldots, v_n\}$ 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 $V_i = \{v_1, \ldots, v_i\}$ 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 $0 \leq x \leq m$

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

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. **(Nombre de particions)**

Donat un enter estrictament positiu $n > 0$, una *partició* d'n és una k-tupla $(\lambda_1, \ldots, \lambda_k)$ tal que $\lambda_1 + \cdots + \lambda_k = n$ i $1 \leq \lambda_1 \leq \lambda_2 \leq \cdots \lambda_k \leq n$. Denotarem per $p(n)$ el nombre de particions d'n. Per exemple, les 7 particions d'$n = 5$ són

(1,1,1,1,1) (1,1,1,2) (1,1,3) (1,4) (5)
(1,2,2) (2,3)

Escriviu un algorisme de programació dinàmica que calculi $p(n)$. Justifiqueu la seva correctesa i calculeu la seva eficiència en temps i espai.

Una solució:

Sigui $p(n, \lambda)$ el nombre de particions de n tal que la més gran de les parts és λ—anomenem *part* a cada una de les components de la tupla $(\lambda_1, \ldots, \lambda_k)$ que és una partició de n. Llavors

$$p(n) = \sum_{\lambda=1}^{n} p(n, \lambda).$$

D'altra banda podem escriure una recursió per a $p(i, \lambda)$, $1 \leq \lambda \leq i \leq n$:

(a) $p(i, i) = p(i, 1) = 1$, ja que només hi ha una partició de i en la part més gran (i única) sigui i o on totes les parts siguin 1.

(b) podem prendre per conveni $p(i, \lambda) = 0$ si $\lambda > i$, i

(c) si $\lambda < i$ aleshores

$$p(i, \lambda) = \sum_{\ell=1}^{\lambda} p(i - \lambda, \ell).$$

Per trobar el valor buscat $p(n)$ hem d'obtenir els $\Theta(n^2)$ subproblemes diferents $p(i, \lambda)$, per a $i = 1, \ldots, n$ i $\lambda = 1, \ldots, i$. El càlcul de $p(i, \lambda)$ requereix cost $\Theta(i-\lambda)$, i sumant per a totes les i i λ ens donarà un cost $\Theta(n^3)$. Per obtenir l'algorisme PD usarem una matriu P de mida $n \times n$ de manera que $P[i, j] = p(i, j)$. Només cal omplir la submatriu triangular inferior, ja que $P[i, j] = p(i, j) = 0$ si $j > i$. Els casos base de la recursió ens permeten omplir la diagonal principal amb 1's ($P[i, i] := 1$), això com la columna #1 ($P[i, 1] := 1$). Aquestes inicialitzacions es duen a terme amb cost $\Theta(n)$. Per omplir un element $P[i, j]$ es necessiten els elements $P[i - j, \ell]$ de la fila $i - j$, de manera que l'algorisme ha d'omplir la matriu d'alt a baix (i d'esquerra a dreta, encara que això no és indispensable).

Comentari Existeix una recursió molt enginyosa, basada en el teorema dels nombres *pentagonals* d'Euler que ens permet calcular $p(n)$ amb cost en espai lineal i cost en temps $\Theta(n^{3/2})$, en comptes dels costos $\Theta(n^2)$ en espai i $\Theta(n^3)$ en temps. De fet, es pot demostrar que

$$p(n) = p(n-1) + p(n-2) - p(n-5) - p(n-7) + p(n-12) + \cdots + (-1)^{m+1} p\left(n - \frac{m(3m-1)}{2}\right) + (-1)^{-m+1} p\left(n - \frac{m(3m-1)}{2}\right) + (-1)^{-m+1} p\left(n - \frac{(-m)(-3m-1)}{2}\right) + \cdots,$$

amb la suma estenent-se fins que $n < m(3m-1)/2$ o $n < (-m)(-3m-1)/2$. La base de la recurrència és $p(0) = p(1) = 1$ (també pot ser convenient assumir que $p(n) = 0$ si $n < 0$).

No obstant, la solució més ineficient i senzilla donada aquí es considera correcta i acceptable per al nivell d'aquest curs.

3.4. (**Rèpliques**)

Tenim un sistema S format per la concatenació de n subsistemes S_1, S_2, ..., 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:

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

Ara bé, per tal d'augmentar la probabilitat que el sistema funcioni correctament podem replicar els subsistemes; això, si posem $x_i > 0$ répliques 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 plantegeu la resolució d'aquest problema mitjançant un algorisme de programació dinàmica (PD) que calculi el nombre de rèpliques x_1, ..., 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 $B \geq v_1 + \ldots + v_n$ (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) Descriviu com ampliar/modificar l'algorisme per tal d'obtenir els valors x_1, ..., x_n que donen la solució óptima.

Una solució:
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 màxima 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

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

Otro caso base trivial es $P(i; C)$ con $C \leq v_i + \cdots + v_n$ ya que no podemos adquirir ni una sola unidad de los subsistemas requeridos.

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

$$P(i; C) = \max_{1 \leq x \leq y_i'} \left\{ (1 - \phi_i^x) \cdot P(i+1; C - x \cdot v_i) \right\}, \qquad (*)$$

En la solució n 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 de el valor máximo entre las opciones posibles.

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

  El algoritmo de PD comienza rellena dos matrices bidimensionales P y X con n filas ($n+1$ filas, pero la fila 0 se "descarta") y $B+1$ columnas cada una por filas, siguiendo la recurrencia. El valor deprobabiliad 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 $\Theta(nB)$. 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 ráplicas de cualquier subsistema es Y entonces el coste del algoritmo será $O(nBY)$ y es habitual que Y"lB. Puede refinarse aún más si se tiene en cuenta el coste por ráplica; si el coste por ráplica más barata es V ($v_i \geq V$) entonces el coste del algoritmo de PD es $O(nB \min(Y, B/V))$.

  La solució n óptima puede reconstruirse de la forma habitual, decidiendo el número de ráplicas de de S_n como $X[n][B]$ y recalculando el presupuesto disponible como $B - X[n][B] * v_i$. Iterando el proceso podemos calcular una solució n óptima con coste $\Theta(n)$.

3.5. **(Resistència de closques)**

La duresa de la closca dels ous es pot determinar per la quantitat de carbonat de calci, $CaCO_3$, que conté. Volem verificar la duresa de la closca dels ous de les nostres gallines però, donat que som informàtics i no químics, volem idear un mètode més algorísmic per fer-ho usant un edifici d'N plantes i p ous de mostra. Hem observat el següent:

- Si tinguéssim només un ou, anirem al primer pis, llançarem l'ou per la finestra i mirarem si es trenca. Si no ho fa, llavors repetirem el procés des del segon pis, des del tercer, etc. fins que es trenquis o declarem que són indestructibles ;-)

  Aquesta estratègia és òptima donat que només tenim un ou i llavors el màxim de llançaments que cal fer és N; amb menys llançaments no podríem garantir que trobarem sempre l'altura exacta a partir de la qual es trenquen els ous.

- En cas que disposem de dos ous (ou_1 i ou_2) podem fer molts menys llançaments. Per exemple, amb $N = 100$ només cal fer 14 llançaments en el pitjor dels casos. Veiem per què:

  El primer, ou_1, es llança des del pis $x = 9$. Poden passar dues coses: (a) si es trenca, llavors fem cerca seqüencial amb l'ou_2 entre els pisos 1 i 8 (en total 9 llançaments com a màxim); **(b)** si no es trenca, llavors es torna a llançar l'ou_1 des del pis 22, i poden tornar a passar dues coses: **(b.1)** si es trenca, fem cerca seqüencial amb l'ou_2 entre els pisos 10 i 21 (en total de $14 = 12 + 2$ llançaments com a màxim), però **(b.2)** si no es trenca l'ou_1 al pis 22, llavors llançarem novament l'ou_1 des dels pisos 34, 45, 55, 64, 72, 79, ... fins que es trenqui i llavors completarem la cerca de manera seqüencial amb l'ou_2 en l'interval apropiat.

- Amb un número prou alt d'ous de prova, la millor estratègia és una "cerca binària"; però si p no és prou gran la "cerca binària" **no** funciona.

Donats N i la quantitat p d'ous de prova, calculeu quin és el menor nombre de llançaments $L_{N,p}$ que garanteix trobar el pis més alt des del qual l'ou no es trencarà en llançar-lo. Determineu també com s'han de dur a terme els llançaments.

(a) Dona (i justifica) una recurrència per al cas de dos ous, és a dir per obtenir $L_{N,2}$.

(b) Dona (i justifica) una recurrència per a qualsevol N i p, és a dir per obtenir $L_{N,p}$.

(c) Dona un algorisme per determinar l'estratègia de llançaments amb el nombre mínim de llançaments. Justifica el seu cost en temps i espai.

Una solució:

(a) Denotemos $D_n := L_{n,2}$ el número mínimo necesario de lanzamientos cuando tenemos n plantas, $n \leq N$, y 2 huevos. Para simplificar la recurrencia añadiremos el caso $n = 0$, tomando como valor por defecto $D_0 = 0$. Cuando $n = 1$, un lanzamiento es suficiente, así $D_1 = 1$. Cuando $n > 1$, sea $x \leq n$ el piso desde el que lanzamos el huevo ou_1 por primera vez. Si se rompe necesitaremos hacer, como máximo x lanzamientos (incluido el que ya hemos hecho del primer huevo). Si no se rompe haremos el número óptimo de lanzamientos para buscar el punto de ruptura en los $n - x$ pisos restantes, como mucho esto será $1 + D_{n-x}$. O sea que nunca necesitaremos más de $\max\{x, 1 + D_{n-x}\}$ lanzamientos. La solució n óptima hace el lanzamiento desde el piso x que

minimiza esta cantidad, así tenemos la recurrencia

$$D_n = \begin{cases} n & \text{si } n \leq 1, \\ \min_{1 \leq x \leq n} \max\{x, 1 + D_{n-x}\} & \text{si } 1 < n \leq N. \end{cases}$$

(b) Denotemos como $L_{n,p}$ el número mínimo necesario de lanzamientos cuando tenemos n plantas, $0 \leq n \leq N$, y q huevos, $1 \leq q \leq p$. Siguiendo el mismo razonamiento del apartado anterior,lanzamos el huevo ou_1 en el piso x del intervalo de pisos donde estamos haciendo la búsqueda, $1 \leq x \leq n$. O bien se romperà y necesitaremos un lanzamiento més $L_{x-1,q-1}$ para encontrar el punto de ruptura en los pisos $x-1$ pisos por debajo pero solo tenemos $q-1$ huevos, o bien no se romperá y necesitaremos $L_{n-x,q}$ lanzamientos para buscar el punto de ruptura en los $n-x$ pisos por encima del x-ésimo, disponiéndose de los q huevos, el huevo ou_1 puede volver a ser usado. Si $n > 1$ y $q > 1$ entonces

$$\begin{aligned} L_{n,q} &= \min_{1 \leq x \leq n} \max\{L_{x-1,q-1} + 1, L_{n-x,q} + 1\} \\ &= 1 + \min_{1 \leq x \leq n} \max\{L_{x-1,q-1}, L_{n-x,q}\}. \end{aligned}$$

Para $n = 0$, tomaremos de nuevo que $L_{0,p} = 0$, independientemente del número de huevos. Si $n = 1$, un lanzamiento es suficiente, entonces $L_{1,q} = 1$, para todo $q \geq 1$. Si $q = 1$, solo tenemos un huevo, así $L_{n,q} = n$, para $n \geq 0$. Poniendo todo junto tenemos la recurrencia:

$$L_{n,q} = \begin{cases} n & \text{si } q = 1 \vee n \leq 1, \\ 1 + \min_{1 \leq x \leq n} \max\{L_{x-1,q-1}, L_{n-x,q}\} & \text{si } 1 < n \leq N \wedge 1 < q \leq p. \end{cases}$$

(c) En el algoritmo usaremos una matriz L y una matriz X, ambas con N filas y p columnas. De manera que que $L[n,q] = L_{n,q}$ y $X[n,q]$ contendrá el primer piso desde el que debemos lanzar un huevo en la estrategia óptima que resuelve el problema para n pisos y q huevos. Las filas 0 y 1 y la columna 1 se rellenan trivialmente, son casos base. Para rellenar $L[n,q]$ necesitamos la columna anterior $L[\cdot, q-1]$ completa y los valores de la columna q desde la fila 1 a la fila $n-1$. Las matrices se rellena por lo tanto de arriba a abajo y de izquierda a derecha siguiendo la recurrencia. Si $n = 1$ o $q = 1$ tendremos $X_{n,q} = 1$, ya que en dichos casos siempre habremos de comenzar en el primer piso, y en general anotaremos el valor x que minimiza el caso general de la recurrencia para $L_{n,q}$. Obtener el valor de cada entrada se hace en tiempo $\Theta(n) = \mathcal{O}(N)$ y por lo tanto como hay $N \cdot p$ subproblemas el coste del algoritmo en tiempo es $\mathcal{O}(N^2 p)$ y el coste en espacio es $\Theta(N \cdot p)$.

Para reconstruir la solución usamos como es habitual una función recursiva que recibe n, q, la matriz X y el número del piso a en el cual empieza el rango de pisos a considerar, e imprime el árbol de decisiones que guía los lanzamientos. La llamada inicial será con los parámetros $(N, p, X, 1)$. El coste de dicha función es proporcional al total de nodos en el árbol de decisiones implícito, que es $\Theta(N)$.

3.6. **(Pesos als vèrtexs)**

> Sigui $G = (V, E)$ un graf dirigit amb pesos $w : V \to \mathbb{Z}^+$, 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? els pesos siguin a les arestes)

Una solució:
Farem servir l'ajut: construir un nou graf G' idèntic al G, excepte que que per a tota aresta $(u, \vec{v})$ definim $w(u, \vec{v}) = w(v)$. 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 $u_1, v_1, \ldots, v_k, u_2$ un camí amb pes $w(u_1) + w(v_1) + \cdots + w(v_k) + w(u_2)$ a G, el mateix camí aG' tindrà pes $w(u_1, \vec{v_1}) + \cdots + w(v_k, \vec{u_2})$ que és $= w(v_1) + \cdots + w(v_k) + w(u_2) \Rightarrow$ els pesos de qualsevol camí en G i G' es diferencien en $w(u_1)$, per tant un camí amb distancia mínima a G' també será un camí amb distancia mínima a G.

3.7. **(Millor camí amb pes global)**

Tenim un graf dirigit $G = (V, E)$ amb pesos sobre les arestes i els vèrtexs $w : (V \cup E) \to \mathbb{Z}$. 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 $w : (V \cup E) \to \mathbb{Z}$
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**
    $\delta(u, v) = \delta'(u, v) - (w(u) + w(v))/2$

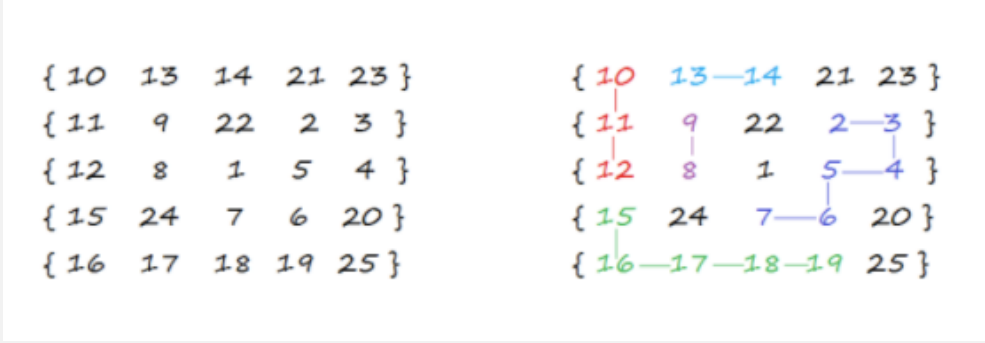
Una solució:
Fals. Si tenim un camí de u a v a G la suma de les distàncies w' al llarg del camí ens dona la suma de les pesos w de tots els arcs al camí més la dels nodes interiors al camí, més $(w(u) + w(v))/2$. Si tenim un cicle de de u a u a G, la suma de les distàncies w' al llarg del cicle ens dona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

3.8. **(Nombres consecutius)**

Donada una matriu $N \times N$ de nombres enters positius **diferents**, escriviu un algorisme de programnació 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 número s consecutius.
Exemple:



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 $[2, \ldots, 7]$ 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. Descriviu la recursiu 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 solució:
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 $\delta_{i,j} =$ **true** si la posición (i, j) es válida ($1 \leq i \leq N$ y $1 \leq j \leq N$) y $\delta_{i,j} =$ **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 $\delta_{i,j} =$ **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 implentar el algoritmo de PD lo más simple es usar la formulación recursiva con memoización. 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 rellenar toda la matriz C es $\Theta(n^2)$. Según se va rellenando también vamos tomando nota de cúal 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 màxima 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 recurperar un camino de longitud màxima 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 $\ell = O(n^2)$.

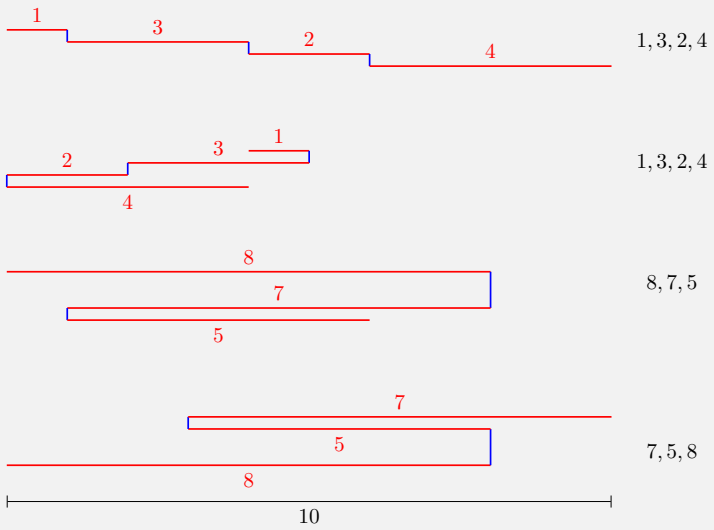
3.9. **(Metre de fuster)**

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 $l_1, \ldots, l_n$ (en aquest ordre) on totes les longituds dels segments son enters al interval $[0, 100]$. Per simplificar la notació considerarem també els extrems $A_0, \ldots A_n$, 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-ésim.
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 son 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, ..., 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 $l_1, \ldots, l_n \in \mathbb{N}$ (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.

Una solució:

(a) Solo necesitamos comprobar que cuando colocamos A_i a la izquierda de A_{i-1} la posición de A_i està 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 par resolver Min-Fold. Para un valor de k fijado, el algoritmo resolvera recursivamente el problema de determinar si se puede o no plegar el metro $L_i, \ldots, L_n$ dentro de $[0, k]$ condicionado a que A_i se ubique en la posición $j \in \{0, \ldots, k\}$.

```
1: procedure FOLD INSIDE INTERVAL REC(i, j)
2:     Place A_i at position j
3:     Left = Right = False
4:     if j - L[i] ≥ 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] ≤ 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 meomización 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 limite 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.

3.10. **(Trenat)**

> 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 $w = w_1 \cdots w_n$, $w[k]$, 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 $x = x_1 x_2 \dots x_p$, $y = y_1 y_2 \dots y_q$ i $s = s_1 s_2 \dots s_n$ 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 $C(i, j) = V$ sii $s[i + j]$ 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 $i' = i \mod p$ 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 $j' = j \mod q$ 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:

$$C(i, j) = [(s_{i+j} = x_{i \mod p}) \vee C(i - 1, j)] \wedge [(s_{i+j} = y_{j \mod q}) \vee C(i, j - 1)],$$

amb $C(0, 0) = V$; per $j \in [n]$, $C(0, j) = V$ sii $s_1 s_2 \dots s_j$ és una repetició de y; per $i \in [n]$, $C(i, 0) = V$ sii $s_1 s_2 \dots s_i$ és una repetició de x.

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

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. Així ens dona un temps addicional de $O(n)$.

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

3.11. **(Emparejando runs)**

La programación dinámica puede utilizarse para resolver un problema en animación gráfica denominado "morphing": convertir una imagen en otra pasando a traves 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 lineas de pixeles en una imagen B/W y queremos ver como transformar una en otra.
Los 0s y 1s definen una serie de *runs*, subcadenas maximales 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 estàn 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"optimo 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
$c_{match}([i;k];[j;\ell]) = |k-l| + |i-j|$.

Fusión: Emparejar p runs consecutivos de A, $[i_1;k_1]$, $[i_2;k_2]$, ..., $[i_p;k_p]$, ($i_q + k_q - 1 < i_{q+1}$ 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
$c_{fusion}([i_1;k_1],\ldots,[i_p;k_p];[j;\ell]) = p + |\ell - (i_p + k_p - i_1)| + |i_1 - j|$

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

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

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

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 solució n propuesta.

Una solució:
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 $1 \le i \le M$ y $1 \le j \le N$, 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 $c_{\text{match}}(i;j) = |\ell_i - \ell_j| + |s_i - s_j|$ y después el resto de runs de A y B óptimamente con coste $C(i+1, j+1)$.

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

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

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

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

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

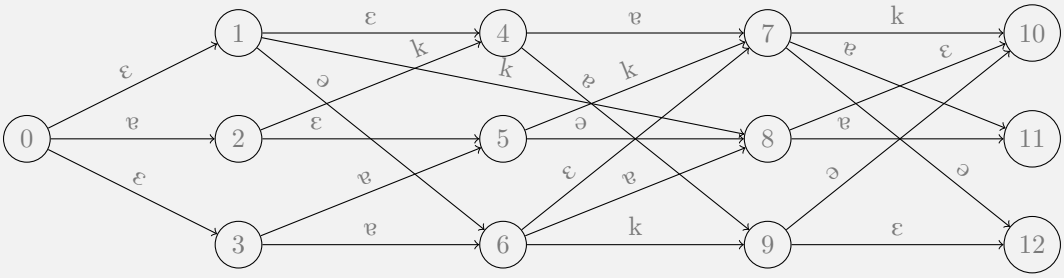
Poniendo todo junto

$$\begin{aligned} C(i,j) = \min\Big(& c_{\text{match}}(i;j) + C(i+1, j+1), \min_{2 \le p \le M+1-i} \{c_{\text{fision}}(i;j,p) + C(i+1, j+p)\}, \\ & \min_{2 \le p \le N+1-j} \{c_{\text{fusion}}(i,p;j) + C(i+p, j+1)\}\Big), \qquad 1 \le i \le M, 1 \le j \le N. \end{aligned}$$

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 rellenar la tabla en el orden indicado. El coste de rellenar la casilla (i,j) es $O(M-i+N-j)$, y el de rellenar 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, cúantos runs consecutivos de A o de B intervienen. La solució n óptima con coste $C(1,1)$ puede ser reconstruída con coste $O(M+N)$ empezando en $D(1,1)$ y según cúal haya sido la decisión se salta a $D(i',j')$ y así sucesivamente.

3.12. **(Reconeixement de la parla)**

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 seqÜencia 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 $\Sigma = \{\varepsilon, \text{ɐ}, \text{k}, \text{ə}\}$ i $v_0 = 0$.



Una seqüencia de fonemes de Σ, $s = (\sigma_1, \sigma_2, \ldots, \sigma_k)$, 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, $s = (\varepsilon, \text{ɐ}, \text{k}, \text{ə})$ és vàlida al nostre model ja que es correspon amb l'etiqueta del camí $0 \to 3 \to 5 \to 7 \to 12$. La seqüencia $s' = (\text{ə}, \varepsilon, \text{ɐ})$ no és vàlida.

> **(a)** Dissenyeu un algorisme eficient, basat en PD, tal que donats un model (G, σ, v_0) i una seqüencia de fonemes s, determini si s és una seqüencia 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 produeixi 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 seqüencia 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 seqüencia $s = (\varepsilon, \text{ɐ}, \text{k}, \text{ə})$ és l'etiqueta del camí $0 \to 3 \to 5 \to 7 \to 12$ i del camí $0 \to 3 \to 6 \to 9 \to 10$. 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 seqüencia 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 $s = (\sigma_1, \ldots, \sigma_k)$.
Vamos a calcular la cantidad $G(i, v)$ que será cierto si la secuencia $(\sigma_i, \ldots, \sigma_k)$ 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 $(\sigma_{i+1}, \ldots, \sigma_k)$ 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 guardaríamos también en una segunda tabla $D(i, v)$ el vecino w que nos proporciona el valor cierto o el valor *indicar* que no existe tal vecino.

De acuerdo con la definición de la recurrencia podemos rellenar la tabla para por columnas, $i = k, k - 1, \ldots, 1$. 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 $(\sigma_i, \ldots, \sigma_k)$ 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 $O(k(n + m))$.

3.13. **(Matrioshka)**



En Dilworth és el col·leccionista més destacat del món de matrioshkas, les nines russes nidificades, com les de la figura de sobre.
En té milers de nines buides de fusta de diferents mides. Per construir un matrioshka la nina més petita es fica dintre de la segona més petita, i aquesta nina, al seu torn, es fica dintre de la la següent i així successivament.
En Dilworth es pregunta si hi ha una altra manera de nidificar-les perquè acabi amb el màxim possible de nines nidificades. Després de tot, això faria que pogués emmagatzemar millor la seva col·lecció i ampliar-la per tal que fos encara més magnífic!
Per a cada nina tenim mesures de la seva amplada i la seva altura. Una nina amb amplada w_i i altura h_i encaixa en una altra nina d'amplada w_j i alçada h_j si i només si $w_i < w_j$ i $h_i < h_j$. Donades les mides de les nines proporcioneu un algorisme, tan eficient com pugueu, per construir la matrioshka amb el màxim nombre possible de nines nidificades.

Una solució:
Observemos que en una solución óptima tenemos una matrioshka exterior y dentro la combinación con mayor número de muñecas que cabe dentro de esta muñeca exterior.

Para poder derivar una recurrencia correcta tenemos que establecer un orden que nos permita garantizar que todas las muñecas que caben dentro de la muñeca i-ésima aparecen posteriormente (o anteriormente). Para conseguir esta propiedad ordenamos por valor decreciente de h y utilizamos orden decreciente de w como criterio secundario de desempate. Así, si la muñeca i cabe dentro de la muñeca j. tenemos $j < i$ aún cuando puede que alguna muñeca k con $k > j$ no quepa dentro de la muñeca i. El coste de esta ordenación es $O(n \log n)$.

Utilizando esta propiedad de la ordenación y la estructura de suboptimalidad podemos obtener la recurrencia que nos permita resolver el problema con PD. El objetivo es calcular el valor $N(i) =$ número máximo de muñecas que se pueden apilar dentro de la muñeca i-ésima.

Para desarrollar la recurrencia con más claridad defino $C(i) = \{j \mid i < j \leq n,\ h_j \leq h_i,\ w_j < w_i\}$, que representa el conjunto de muñecas que caben dentro de la muñeca i. DE acuerdo con la discusión anterior tenemos:

$$N(i) = \begin{cases} 0 & C(i) = \emptyset \\ \max_{j \in C(i)}\{1 + N(j)\} & \text{otherwise} \end{cases}$$

Los valores $N(i)$ se pueden calcular con un recorrido en tabla con coste $O(n^2)$ ya que $D(i)$ puede tener $O(n)$ elementos. Finalmente, podemos obtener el tamaño máximo de matrioshka calculando $\max_{1 \leq i \leq n} N(i)$ en tiempo $O(n)$.

El coste total del algoritmo de PD es $O(n^2)$.

3.14. **(Màxim expressió aritmètica)**

Tenim una expressió aritmètica que conté n nombres reals i $n-1$ operadors, cadascun $+$ o $*$. L'expressió no està parentitzada i no sabem en quin ordre hi hem d'aplicar les operacions. El que sí que sabem, però, és que volem fer les operacions en un ordre que maximitzi el valor de l'expressió resultant. És a dir, volem inserir adequadament els parèntesis a l'expressió per tal que el seu valor en avaluar-la sigui maximitzat. Per exemple:

- Per a l'expressió $6*3+2*5$, l'ordre òptim és sumar primer els nombres del mig i després realitzar les multiplicacions: $(6*(3+2))*5 = 150$.
- Per a l'expressió $0,1*0,1+0,1$, l'ordenació òptima és realitzar primer la multiplicació i després la suma: $(0,1*0,1)+0,1 = 0,11$.
- Per a l'expressió $(-3)*3+3$, l'ordenació òptima és $((-3)*3)+3) = -6$.

Doneu un algorisme, tan eficient com pugueu, per tal de calcular quin és el valor màxim obtenible en parentitzar l'expressió d'entrada i quina és exactament aquesta parentització.

Una solució:
Aquest problema és semblant al problema de la parentització de la multiplicació de matrius, vist a les classes de teoria. En aquest cas hem de decidir com parentitzar la seqüència d'entrada per tal de maximitzar l'expressió donada. Resoldrem el problema mitjançant programació dinàmica perquè, com veurem, el problema té les dues propietats requerides: subestructura òptima i solapament de subproblemes.

Per a plantejar una resolució per programació dinàmica necessitem primer definir la notació necessària. Denotarem els números de l'expressió com $a_1, a_2, \ldots, a_n$, i els operadors com $op_1, op_2, \ldots, op_{n-1}$, per tant, una expressió donada és de la forma

$$a_1\, op_1\, a_2\, \ldots\, a_{n-1}\, op_{n-1}\, a_n.$$

Sigui $M(i,j)$ el **màxim** valor que es pot obtenir de la subexpressió que comença a a_i i que acaba a a_j (és a dir, la subexpressió $(a_i\, op_i\, \ldots\, op_{j-1}\, a_j)$, i sigui $m(i,j)$ el **mínim** valor que es pot obtenir en avaluar la subexpressió que comença a a_i i acaba a a_j.

Tota expressió donada acabarà operant dues subexpressions (parentitzades) mitjançant un darrer operador, és a dir, $((expr_1)\ op_k\ (expr_2))$. Per tal que el resultat d'aquesta darrera avaluació sigui òptima (màxima), cal que l'avaluació de les dues subexpressions que s'operen també ho siguin. Però observeu que hem de mantenir tant el màxim com el mínim de tota subexpressió, perquè el valor màxim de l'avaluació global podria provenir de multiplicar dos subexpressions que avaluen a números negatius.[1]

Basant-nos en aquesta subestructura òptima observada, plantegem la recurrència que expressa la solució a un subproblema general en relació a solucions de subproblemes més petits. Per avaluar una subexpressió $a_i\, \ldots\, a_j$, podem separar-la en dos problemes al k-èsim operador i, recursivament, resoldre les subexpressions $a_i\, \ldots\, a_k$ i $a_{k+1}\, \ldots\, a_j$. En fer això hem de considerar totes de combinacions de maximització i minimització dels subproblemes.

Els casos base són $M(i,i) = m(i,i) = a_i, \forall i : 1 \leq i \leq n$.

[1] Si els nombres fossin naturals (i no reals, com diu l'enunciat), aleshores n'hi hauria prou amb matenir els valors màxims i el problema seria més senzill.

Els casos recursius són:

$$M(i,j) = \min_{i \le k < j} \Big\{ \quad \max\Big(\quad M(i,k)\, op_k\, M(k+1,j), \\ M(i,k)\, op_k\, m(k+1,j), \\ m(i,k)\, op_k\, M(k+1,j), \\ m(i,k)\, op_k\, m(k+1,j) \quad \Big) \Big\}$$

$$m(i,j) = \min_{i \le k < j} \Big\{ \quad \min\Big(\quad M(i,k)\, op_k\, M(k+1,j), \\ M(i,k)\, op_k\, m(k+1,j), \\ m(i,k)\, op_k\, M(k+1,j), \\ m(i,k)\, op_k\, m(k+1,j) \quad \Big) \Big\}$$

El valor $M(1,n)$ d'aquesta recurrència és l'objectiu del problema; és la solució que estem buscant.

Hi ha un total de $O(n^2)$ subproblemes: dos per cadascun dels parells d'índexos $1 \le i \le j \le n$. Els subproblemes $M(i,j)$ i $m(i,j)$ han de considerar $O(j-i) = O(n)$ subproblemes més petits. El temps total de l'algorisme és, doncs, $O(n^3)$.

El terme $M(1,n)$ només indicarà el valor numèric de l'avaluació de la parentització maximitzadora de tota l'expressió d'entrada. Per tal de determinar on s'han de col·locar exactament els parèntesis necessitem guardar adicionalment, per a cada $M(i,j)$ i $m(i,j)$, quina k (punt de tall) produeix el màxim i mínim, respectivament. Amb aquesta informació, una vegada haguem calculat $M(1,n)$, podrem reconstruir en temps $O(n)$ la parentització.