

El coste del bucle interno es $O(k)$ y se ejecuta $n - k$ veces. Por lo tanto el coste total del algoritmo es $O(nk)$.

3. Si partimos de un grafo $G = (V, E)$, podemos calcular las distancias $d(u, v)$ entre cualquier par de vértices. Considerando como conjunto de ciudades V y la distancia d , para un valor de k , $r(C)$ es igual a 1 si y solo sí tenemos una selección de vértices para la que todos los demás vértices están a distancia ≤ 1 . Lo que es equivalente a decir que la selección es un conjunto dominador de tamaño k en G . DOMINATING-SET es un problema NP-completo, por lo tanto si pudiésemos resolver nuestro problema en tiempo polinómico P=NP, por lo que el problema es NP-difícil.
4. El algoritmo voraz propuesto tiene coste polinómico. Sea C^* la selección óptima. Al finalizar el algoritmo voraz, C es una selección de k ciudades, por tanto $r(C^*) \leq r(C)$. Para tener una 2-aproximación tenemos que demostrar que $r(C) \leq 2r(C^*)$.

Supongamos que no es cierto, es decir, $r(C^*) < r(C)/2$.

Observemos en primer lugar que, al finalizar el algoritmo, cualquier par de centros en C están a distancia $> r(C)/2$.

Por cada ciudad con centro comercial $c_i \in C$, consideramos el subconjunto de las ciudades C_i cuya a distancia a c_i es $\leq r(C)/2$. Como al finalizar el algoritmo, cualquier par de centros en C están a distancia $> r(C)/2$, los C_i son disjuntos. Y la unión de los C_i 's es el conjunto de todos los vértices.

Además, como por hipótesis $r(C^*) < r(C)/2$, en cada C_i tenemos que tener un elemento de C^* . Como $|C| = |C^*|$, cada C_i contiene exactamente un único elemento de C^* al que llamamos c_i^* . Además, c_i^* es el centro en C^* que está más cerca de c_i , luego $d(c_i, c_i^*) \leq r(C^*)$.

Para una ciudad cualquiera s , sea c_i^* su centro más cercano en C^* y C_i el subconjunto de ciudades en el que viene incluida por el algoritmo voraz. Tenemos entonces

$$d(s, C) \leq d(s, c_i) \leq d(s, c_i^*) + d(c_i^*, c_i) \leq 2r(C^*),$$

donde la segunda desigualdad se deduce de la desigualdad triangular y la tercera se deduce de que toda distancia entre una ciudad y su centro en C^* ha de ser necesariamente $\leq r(C^*)$ y $d(c_i, c_i^*) \leq r(C^*)$. Con lo que llegamos a una contradicción, porque existe alguna ciudad s que nos da el valor $r(C)$, esto es, $r(C) = d(s, C)$ y concluimos entonces que $d(s, C) = r(C) \leq 2r(C^*)$, contradiciendo nuestra hipótesis de que $r(C) > 2r(C^*)$ (o $r(C)/2 > r(C^*)$).

7.3. Programación Dinámica

Solución 3.54 1. Solo son legales las siguientes configuraciones de columnas:

```

+++
| |
+++
| |
+++
| |
+++
| |
+++
| |
+++
| |
+++
| |
+++
sin fichas 1 ficha (x4)
0          1-4      5   6   7   <-- ids. de configuracion

```

Es decir, existen solo 8 configuraciones posibles. Si en vez de 4 casillas tuviéramos columnas de tamaño m , el número de configuraciones sería aproximadamente $2^{m/2} = (\sqrt{2})^m$, es decir, crece muy rápidamente en función de m ... pero sigue siendo constante respecto a n .

- Denotaremos $V(n)$ el mayor valor posible para una configuración legal cualquiera con n columnas, y $V(n, x)$ el mayor valor posible para una configuración de n columnas cuya última columna sea la configuración x . Evidentemente

$$V(n) = \max_{0 \leq i \leq 7} V(n, i),$$

y lo podremos calcular con coste $\Theta(1)$ una vez dispongamos de los valores $V(n, x)$ para toda x .

En el algoritmo que plantearemos usaremos v_x para denotar el valor de una columna con la configuración x (p.e., $v_0 = 0$, $v_5 = v_1 + v_4, \dots$), y definimos $\mathcal{L}_{x,y} = \text{true}$ si y sólo si x e y son configuraciones compatibles: así, por ejemplo, $\mathcal{L}_{0,y} = \text{true}$ para toda y , $\mathcal{L}_{x,x} = \text{false}$ para cualquier $x \neq 0$, etc.

Rellenar tablas con los valores v_x y $\mathcal{L}_{x,y}$ tiene coste $\Theta(1)$ pues solo hay 8 configuraciones. La recurrencia con la que obtendremos nuestro algoritmo de PD es

$$V(n, x) = \max_{y: 0 \leq y \leq 7 \wedge \mathcal{L}_{y,x}} \{V(n-1, y) + v_x\}, \quad n > 1$$

y $V(1, x) = v_x$. El coste del algoritmo es $\Theta(n)$ puesto que podemos calcular las $V(n, x)$'s en tiempo constante a partir de las $V(n-1, y)$'s; es decir, en el algoritmo iterativo cada iteración tendrá coste $\Theta(1)$ y hay n iteraciones. Como para calcular $V(n, x)$ sólo se necesita conocer los valores $V(n-1, y)$, basta un par de vectores de tamaño 8, así que el espacio de memoria adicional que requiere nuestro algoritmo es $\Theta(1)$ (para la tabla 8×8 de compatibilidades y los dos vectores $V[x] \equiv V(n, x)$ y $Vp[y] \equiv V(n-1, y)$, ambos de tamaño 8). Si se quisiera reconstruir la configuración que alcanza el valor máximo tendríamos que guardar para cada k y x cuál es la configuración y que ha maximizado $V(k, x)$, esto es, cuál es la y tal que $V(k, x) =$

$V(k - 1, y) + v_x$; esto requiere más coste en espacio ($\Theta(n)$ en vez de $\Theta(1)$), pero el coste temporal sigue siendo $\Theta(n)$, pues el coste del bucle principal no cambia, y el bucle final que reconstruye la solución tiene coste $\Theta(n)$, yendo desde el final (la columna n) hacia el principio.

```

void valor_tablero_optimo(const vector<int>& valor, int n) {
    vector<int> val_conf(8);
    vector<vector<bool>> L(8, vector<bool>(8));
    // rellenar val_conf y L
    val_conf[0] = 0; val_conf[1] = valor[0]; // ...
    L[0][0] = L[0][1] = ... = true;
    L[1][1] = L[2][2] = ... = false;
    ...
    vector<int> V(8);
    vector<int> Vp = valor_conf;
    for (int i = 2; i <= n; ++i) { // coste Theta(n)
        for (int x = 0; x < 8; ++x) { // coste Theta(1)
            V[x] = -infinity;
            for (int y = 0; y < 8; ++y) // coste Theta(1)
                if (L[x][y])
                    V[x] = max(V[x], Vp[y] + valor_conf[x]);
        }
        Vp = V;
    }
    // calcular aqui valmax = max(V[x]: 0 <= x < 8)
    return valmax;
}

```

Solución 3.55 Para que la solución resulte lo más sencilla posible conviene considerar que existen $3n$ tipos de ladrillos, cada uno de los tipos originales dan lugar a tres nuevos tipos $< l_i, w_i, h_i >$ (tipo i), $< w_i, h_i, l_i >$ (tipo $n + i$) y $< h_i, l_i, w_i >$ (tipo $2n + i$), $1 \leq i \leq n$. Lo que caracteriza a cada nuevo tipo es qué dimensión es la que da la altura del ladrillo. Definimos también $a_i = \text{área del tipo } i$ y $L_{i,j} = \text{true}$ si y sólo si un ladrillo de tipo i puede ser colocado encima de un ladrillo de tipo j . Observad que si $a_i \geq a_j$ entonces $L_{i,j} = \text{false}$, pero que hay que comparar las dimensiones de la base para saber si un ladrillo i puede ser colocado sobre un ladrillo j si $a_i < a_j$.

```

struct brick {
    double l, w, h;
};

vector<Brick> b(3*n+1); // no usamos b[0]
bool compatible(int i, int j) {
    return (b[i].w < b[j].w and b[i].l < b[j].l) or

```

```

        (b[i].w < b[j].l and b[i].l < b[j].w);
}

```

Para plantear la PD podemos optar por una recurrencia basada colocar un primer ladrillo y a continuación, de manera óptima, todos los que hay por debajo, o basada en colocar el último ladrillo y a continuación de manera óptima todos los que haya por encima.

Supongamos que ordenamos los $3n$ tipos de ladrillos por área creciente, así que si $i < j$ entonces $a_i \leq a_j$ y a priori un ladrillo de tipo i puede estar encima de un ladrillo de tipo j (pero no al revés).

Sea H_i la altura máxima de un monolito cuya cúspide es un ladrillo de tipo i . Observad que al estar ordenados H_{3n} = altura del tipo $3n$, ya que ningún otro tipo de ladrillo puede ponerse por debajo de un ladrillo de tipo $3n$. Entonces

$$H_i = \max\{H_j + h_i : j > i \wedge L_{i,j}\}$$

Y la respuesta buscada es, naturalmente,

$$H = \max_{1 \leq i \leq 3n} \{H_i\}.$$

El cálculo de las H_i 's puede hacerse con coste $\Theta(n-i)$ cada una, lo que da finalmente un coste global $\Theta(n^2)$. Los valores $L_{i,j}$ pueden precomputarse con coste en espacio y tiempo cuadráticos, o bien, calcularse sobre la marcha, con coste $\Theta(1)$ en cada ocasión.

Solución 3.56 Per trobar una solució recursiva observem que a una solució òptima o bé es fa un reboot el primer dia o no, i el que segueix es una solució òptima condicionada a quan es va fer l'últim reboot. Això ens porta a considerar subproblems en els que comptabilitzem el nombre de dies des de que es va fer l'últim reboot.

Sigui $C_{i,j}$ = el màxim nombre de clients que es pot donar servei entre i i n (inclosos), assumint que l'últim reboot es va fer fa j dies, és a dir el dia $i - j$. Volem calcular $C_{1,1}$.

Notem que $C_{n,j} = \min\{x_n, s_j\}$, $\forall j$, l'últim dia no servim res fent un reboot i és millor no rebotar. Així tenim el cas base.

Per calcular $C_{i,j}$ tenim que o bé s'ha fet un reboot el dia i (i s'ha donat servei a 0 usuaris), o el dia i ha funcionat normalment amb productivitat $\min\{x_i, s_j\}$. Per tant, comptabilitzant la distancia a l'últim reboot tenim

$$C_{i,j} = \max\{C_{i+1,1}, C_{i+1,j+1} + \min\{x_i, s_j\}\}.$$

Podem computar C , taula amb grandària n^2 , fent un recorregut per files començant per la fila n (que és el case base). Per omplir la fila i es necessita tenir ja calculada la fila $i+1$. Cada entrada de la taula es pot computar en $O(1)$. Per tant, la complexitat de l'algorisme és $O(n^2)$ i l'espai addicional es també $O(n^2)$. Es pot reduir la complexitat en espai si en comptes de mantenir $C_{i,j}$ per a tota i i tota j , només guardem una fila i i la fila immediatament a sota, la $i+1$.

Si volem trobar la seqüència de reboots, com sempre afegirem la matriu de “punters” i la farem servir per construir la solució. És a dir, podem guardar una matriu R de booleans, de manera que $R_{i,j}$ que sigui **true** si $C_{i,j}$ es maximitza rebotant el dia i i **false** en cas contrari.

```
j = 1;
for (int i = 1; i <= n; ++i) {
    if (R[i][j]) {
        cout << "reboota dia " << i << endl;
        j = 1;
    } else {
        ++j;
    }
}
```

Solución 3.57 La clave para resolver este problema es dar con la generalización apropiada que nos permita obtener una recurrencia.

Sea $C_{i,j}^p$ el coste óptimo de dividir la subsecuencia $\{s_i, \dots, s_j\}$ en p rangos. Entonces la solución buscada es $C_{1,n}^r$. Como cada rango debe necesariamente contener al menos un elemento, uno de los casos base es $C_{i,j}^p = +\infty$ si $j < i + p - 1$. Por otro lado, $C_{i,j}^1 = s_i + \dots + s_j$: si solo hacemos un rango ($p = 1$) el valor de la solución es la suma de los elementos el rango. Para la recurrencia

$$C_{i,j}^p = \min_{i < k \leq j} \min_{1 \leq q < p} \max\{C_{i,k-1}^q, C_{k,j}^{p-q}\}$$

Almacenar los elementos $C_{i,j}^p$ requiere espacio $\Theta(n^2 \cdot r)$. Para llenar cada elemento se necesita coste $\Theta(n \cdot r)$. Todo ello nos lleva a un coste elevado para el algoritmo completo: tendrá coste $\Theta(n^3 \cdot r^2)$. Para llenar la matriz de las C 's debemos empezar llenando la submatriz $C_{i,j}^1$, $i \leq j$, (con coste $\Theta(n^2)$) y a partir de ahí cada nueva submatriz $C_{i,j}^p$ a partir de las submatrices $C_{i,j}^t$, $1 \leq t < p$, con coste $\Theta(n^3r)$ cada una.

Una solución más eficiente se obtiene considerando el coste $C_{i,p}$ de particionar la subsecuencia (s_i, \dots, s_n) en p rangos de manera óptima. Sea $S_{i,j} = s_i + \dots + s_j$. Entonces $C_{i,1} = S_{i,n}$, $C_{i,p} = +\infty$ si $p > n - i + 1$ y

$$C_{i,p} = \min_{i < k \leq n} \max\{C_{k,p-1}, S_{i,k-1}\} = \min_{i < k \leq n} \max\{C_{k,p-1}, S_{i,k-1}\}$$

Naturalmente el coste que nos interesa es $C_{1,r}$. El coste de calcular cada entrada $C_{i,p}$ es $O(n - i)$ y el coste total del algoritmo será $O(n^2r)$, una muy notable mejora respecto al primer algoritmo.

Solución 3.58 El problema és el mateix que el de calcular un MIS en un arbre que teniu a les transparències.

Utilitzarem PD sobre arbres. El següent algorisme lineal utilitza un camp auxiliar v a cada node u que emmagatzemarà el grau màxim de simpatia del subarbre arrelat a u .

L'entrada sera l'arbre T del PAS a la UPC, cada nus amb el seu grau de simpatia, els valors $s[u]$, amb el grau de simpatia del nus u i els valors v (per exemple, inicialitzats a -1).

```

procedure FESTA( $T$ )
     $u := \text{ROOT}(T)$ 
    if  $u$  és una fulla then
         $v[u] := s[u]$ 
    else
         $vnets := s[u]$ 
        for  $t \in \text{GRANDCHILDREN}(T)$  do
             $x := \text{ROOT}(t)$ 
            if  $v[x] = -1$  then  $v[x] := \text{FESTA}(t)$ 
            end if
             $vnets := vnets + v[x]$ 
        end for
         $vfills := 0$ 
        for  $t \in \text{CHILDREN}(T)$  do
             $x := \text{ROOT}(t)$ 
            if  $v[x] = -1$  then  $v[x] := \text{FESTA}(t)$ 
            end if
             $vfills := vfills + v[x]$ 
        end for
         $v[u] := \max(vnets, vfills)$ 
    end if
    return  $v[u]$ 
end procedure
```

Si para cada nodo x se almacena la suma $\sigma[x]$ de los valores de sus hijos entonces el algoritmo puede reescribirse así:

```

procedure FESTA( $T$ )
     $u := \text{ROOT}(T)$ 
    if  $u$  és una fulla then
         $v[u] := s[u]; \sigma[u] := 0$ 
    else
         $vnets := s[u]$ 
         $vfills := 0$ 
        for  $t \in \text{CHILDREN}(T)$  do
             $x := \text{ROOT}(t)$ 
            if  $v[x] = -1$  then  $\langle v[x], \sigma[x] \rangle := \text{FESTA}(t)$ 
            end if
             $vfills := vfills + v[x]$ 
             $vnets := vnets + \sigma[x]$ 
        end for
         $v[u] := \max(vnets, vfills); \sigma[u] := vfills$ 
    end if
    return  $\langle v[u], \sigma[u] \rangle$ 
end procedure
```

La festa amb major grau de simpatia es la que correspon al valor v de l'arrel; però si volem que参与 la gerenta cal implementar la festa corresponent a ella i maximitzant el valors dels seus “nets”—això no implica que tots els “nets” de la gerenta siguin convidats, però el que és segur és qu eno s'ha de convidar a cap fill (doncs la gerenta participa a la festa). Per reconstruir la solució, com sempre afegirem els punters necessaris per registrar la decisió presa a cada node (si $v[x]$ és $vfills$ o $vnets$. L'algoritsme fa un recorregut sobre l'arbre i el seu cost é $\Theta(n)$. A més necessitarem espai addicional per les $v[x]$'s (i potser les $\sigma[]$'s).

Solución 3.61 Asumiremos que todos los libros caben en un estante si se ponen solos: $t_i \leq L$ para toda i , $1 \leq i \leq n$. En caso contrario el problema no tiene solución.

Definamos $p(i, j)$ el “coste” de poner (exclusivamente) los libros i a j en un mismo estante:

$$p(i, j) = \begin{cases} \max\{h_i, \dots, h_j\}, & \text{si } t_i + \dots + t_j \leq L, \\ +\infty, & \text{en caso contrario,} \end{cases}$$

es decir, la altura máxima de los libros colocados si caben en el estante, y $+\infty$ si no se pueden poner.

Usando una matriz $A_{ij} = t_i + \dots + t_j$ que podemos llenar fácilmente con coste $\Theta(n^2)$ se puede llenar, también con coste $\Theta(n^2)$ una matriz $P_{i,j} = p(i, j)$; tendremos $P_{i,j} = \max\{P_{i,j-1}, h_j\}$ si $A_{i,j} \leq L$ y $P_{i,j} = +\infty$ si $A_{i,j} > L$. Una vez llevado a cabo este preprocesso (con tiempo y espacio cuadrático) el problema lo resolveremos determinando cuál es el coste de colocar óptimamente los libros (usando tantas estanterías como convenga), es decir, decidiendo con qué libros iniciamos cada estante. Dicho de otro modo, querremos saber en la secuencia de libros $1, 2, \dots, n$ preestablecida con cuáles se inicia cada estante. Por ejemplo, con el 1 (libros del 1 al 6), con el 7 (libros del 7 al 15) y con el 16 (libros del 16 al n).

Si C_i denota el coste óptimo de colocar los primeros i libros y L_i es el libro que inicia la última estantería en la configuración óptima para i libros entonces

$$C_{i+1} = \min\{C_{L_i-1} + P_{L_i,i+1}, C_i + P_{i+1,i+1}\}, \quad i > 1, \quad (7.1)$$

siendo $C_1 = h_1$ y $L_1 = 1$. Por otro lado $L_{i+1} = L_i$ si la minimización de C_{i+1} se consigue con el primer término de (7.1), y $L_{i+1} = i+1$ en caso contrario. Esta recurrencia requiere espacio lineal para las C 's (y las L 's) y coste también lineal para llenarlas; pero el preprocesso ya ha necesitado coste cuadrático en tiempo y espacio y ese sería el coste global. La respuesta buscada es C_n y los libros que abren cada estante se pueden obtener muy fácilmente a partir de los L_i 's.

Alternativamente, sin recurrir a la tabla auxiliar L_n de tamaño $\Theta(n)$, si definimos C_i como el coste de colocar óptimamente los libros i a n entonces podemos escribir la siguiente recurrencia

$$C_i = \min_{i < j \leq n+1} \{C_j + P_{i,j-1}\}, \quad 1 \leq i \leq n$$

ya que C_i será el coste de colocar los libros desde el libro i hasta el $j - 1$ en un mismo estante (y el primero del estante es el libro i) y los restantes libros, desde el j en adelante se colocan óptimamente en otros estantes (el j arranca un nuevo estante). De entre todas las posibilidades para j obviamente elegimos la que minimiza el coste. En la recurrencia anterior usaremos como convenio $C_{n+1} = 0$ (no hay libros a colocar); así $C_n = C_{n+1} + P_{n,n} = P_{n,n} = h_n$. Esta nueva recurrencia necesita espacio de memoria lineal para las C_i 's pero tiempo $\Theta(n^2)$ pues para calcular C_i se han de hacer $\Theta(n - i)$ iteraciones. Y si prescindimos de “recordar” $Q := P_{i,j-1}$ para obtener $Q' := P_{i,j}$ y tuvieramos que calcular cada $P_{i,j-1}$ desde cero entonces tendríamos coste $\Theta(j - i)$ para calcular $C_j + P_{i,j-1}$ y coste $\sum_{i < j \leq n+1} \Theta(j - i) = \Theta((n - i)^2)$ para calcular C_i . Para calcular todas las C_i 's el coste sería entonces $\sum_{1 \leq i \leq n} \Theta((n - i)^2) = \Theta(n^3)$. Esto hay que evitarlo, por ejemplo así:

```
// calculo de C[i], coste: Theta(n-i)
int mn = +INFINITY;
int j = i+1; int P = h[i]; int W = t[i];
while (j <= n+1) {
    // P = P(i,j-1), W = t[i] + ... + t[j-1]
    if (mn > C[j] + P) mn = C[j] + P;
    if (W + t[j] > L)
        P = +INFINITY;
    else
        P = max(P, h[j]);
    W += t[j];
    ++j;
}
C[i] = mn;
```

El valor finalmente buscado es C_1 . De cualquier modo el coste global del algoritmo basado en esta última recurrencia es cuadrático, igual que la solución que podemos construir partiendo de la recurrencia (7.1).

Solución 3.62 Este problema se asemeja al del cálculo de la distancia de edición. Definimos C_{ij} como la calidad/puntuación máxima obtenible alineando $X[1..i]$ con $Y[1..j]$. Al alinear estos dos segmentos tendremos dos cadenas de longitud ℓ , siendo ℓ al menos tan grande como la mayor entre i y j y como mucho $i + j$, esto es, $\max(i, j) \leq \ell \leq i + j$. Consideremos la última columna de la mejor alineación posible entre $X[1..i]$ e $Y[1..j]$. Tenemos las siguientes posibilidades:

X_i	X_i	-
Y_j	-	Y_j

Entonces para un caso general tendremos

$$C_{ij} = \max\{C_{i-1,j-1} + \delta(X_i, Y_j), C_{i-1,j} + \delta(X_i, -), C_{i,j-1} + \delta(-, Y_j)\}$$

Para los casos de base tenemos $C_{00} = -\infty$, $C_{0,1} = \delta(-, Y_1)$, $C_{1,0} = \delta(X_1, -)$ y $C_{0,i} = C_{i,0} = -\infty$ si $i > 1$, puesto que no podemos alinear una cadena de longitud mayor que 1 con una cadena vacía.

Para llenar la matriz se necesita tiempo constante por entrada, y en total hay $m \times n$ entradas, así que el coste en tiempo y espacio del algoritmo es $\Theta(m \cdot n)$. Si para cada (i, j) guardamos cuál de las tres opciones se ha de tomar para maximizar el coste entonces podremos reconstruir la alineación óptima con coste $O(m + n)$ (y coste $\Theta(m \cdot n)$ en espacio). La matriz C se ha de llenar de izquierda a derecha y de arriba a abajo. Para calcular la componente (i, j) necesitamos la fila anterior (las componentes $(i - 1, j - 1)$ y $(i - 1, j)$) y las columnas anteriores en la misma fila (de hecho la componente $(i, j - 1)$). Si no necesitáramos reconstruir la alineación óptima y nos bastase saber la puntuación, se podría modificar el algoritmo de PD para que trabaje en espacio $\Theta(m)$ en vez de $\Theta(m \cdot n)$, pero el coste en tiempo seguirá siendo igualmente $\Theta(m \cdot n)$.

Solución 3.63 Dado un árbol t llamemos $a(t)$ al mínimo tiempo necesario para despertar a todos los sensores que hay en t , incluida la raíz; se necesita una unidad de tiempo para despertar a un sensor, y cada nodo del árbol tiene que despertar a sus sucesores, uno en cada instante. Mientras un nodo v ya despierto se encarga de despertar a uno de sus sucesores, cualquier otro nodo w despierto puede estar haciendo lo propio con alguno de sus sucesores.

Si t es una hoja, evidentemente $a(t) = 1$. Consideremos ahora un árbol t con hijos t_1, t_2, \dots, t_k . Evidentemente parece lógico comenzar despertando a los hijos de t por valor $a(t_i)$ decreciente. Para empezar podemos suponer sin pérdida de generalidad que, con coste $O(k \log k)$, indexamos los hijos de manera que $a(t_1) \geq a(t_2) \geq \dots \geq a(t_k)$.

Consideremos un subárbol t_j . Si existe $i < j$ tal que $a(t_i) \geq a(t_j) + j - i$ (equivalentemente, tal que $b(t_i) := a(t_i) + i \geq a(t_j) + j =: b(t_j)$) entonces se podrá despertar a todos los nodos de t_j antes o al mismo tiempo que a los de t_i .

Por otro lado si no existe tal valor de i , entonces se necesitarán $\Delta_j = a(t_j) + j - \max_{1 \leq i < j} \{a(t_i) + i\}$ unidades de tiempo adicionales para despertar a los nodos de t_j . Si convenimos que $\Delta_1 = a(t_1)$, y $\Delta_j = \max \{0, a(t_j) + j - \max_{1 \leq i < j} \{a(t_i) + i\}\}$ entonces

$$a(t) = 1 + \sum_{1 \leq j \leq k} \Delta_j$$

Una vez ordenados los hijos de t por su valor $a(t_j)$ podemos calcular en tiempo $O(k)$ los valores $b(t_j) = a(t_j) + j$, $B(t_j) = \max_{1 \leq i < j} \{b(t_i)\} = \max \{B(t_{j-1}), b(t_{j-1})\}$ y $\Delta_j = b(t_j) - B(t_j)$, así como la suma de las Δ 's, en definitiva, $a(t)$.

Una alternativa es darse cuenta de que

$$a(t) = \max_{1 \leq i \leq k} \{a(t_i) + i\},$$

bajo la condición de que $a(t_1) \geq \dots \geq a(t_k)$. Podemos demostrar que ordenar los hojos de t de manera que $\{a(t_i)\}$ forme una secuencia decreciente y activar siguiendo ese orden es óptimo mediante un argumento de intercambio. Supongamos que en el orden óptimo

el nodo v despierta a sus hijos en el orden t_1, t_2, \dots, t_k pero en este orden, los hijos no están ordenados en orden decreciente de valores de a . En ese caso tendremos un valor i , $1 \leq i < k$ tal que $a(t_i) < a(t_{i+1})$. En el orden óptimo la contribución de estos dos elementos es $a(t_i) + i$ y $a(t_{i+1}) + i + 1$. Si los intercambiamos de posición la contribución sería $a(t_i) + i + 1$ y $a(t_{i+1}) + i$.

Como $a(t_i) < a(t_{i+1})$, tenemos $a(t_i) + 1 \leq a(t_{i+1})$, y también $a(t_i) + i + 1 \leq a(t_{i+1}) + i$.

Así tenemos que $\max\{a(t_i) + i + 1, a(t_{i+1}) + i\} < \max\{a(t_i) + i, a(t_{i+1}) + i + 1\}$ y por lo tanto intercambiando los dos elementos no podemos empeorar el valor total del máximo.

El algoritmo de PD para explorar el árbol y llenar los valores de los nodos es muy similar al problema 58 de la lista. Se hacen las llamadas recursivas en los k hijos del nodo raíz y al vuelta se ordenan (desde un punto de vista puramente lógico) los hijos por $a(t_i)$ decreciente y se calcula $a(t)$ tal como se ha descrito más arriba—calculando las Δ_i 's o por la vía más sencilla de calcular el máximo de la secuencia $\{a(t_i) + i\}$. Por lo tanto en cada nodo visitado pagamos coste proporcional a $O(k \log k) = O(k \log n)$ por ordenar más $O(k)$ para completar el cálculo de $a(t)$, de donde podemos concluir que el coste del algoritmo será $O(n + m \log n) = O(n \log n)$, puesto que el número de aristas en el árbol es $m = n - 1$.

Solución 3.64 Consideremos la cuerda (hay exactamente una) que tiene un extremo en j . Si el otro extremo, $a(j)$, está entre i y $j - 1$ (inclusive) entonces dicha cuerda puede formar parte del subconjunto de cuerdas viables con extremos (a, b) y $i \leq a \leq b \leq j$, esto es, contribuir a $T(i, j)$. Entonces

$$T(i, j) = \max\{T(i, j - 1), 1 + T(i, a(j) - 1) + T(a(j) + 1, j - 1),$$

es decir, será el máximo entre agregar la cuerda $(a(j), j)$ al conjunto y optimizar el resto de las cuerdas elegidas de entre todas las que tienen extremos entre i y $a(j) - 1$ y las que tienen extremos entre $a(j) + 1$ y $j - 1$, porque no pueden cortar a la cuerda $(a(j), j)$, y no agregar la cuerda $(a(j), j)$ al conjunto y entonces optimizar la elección con cuerdas cuyos extremos están en el rango $[i..j - 1]$.

Si el extremo $a(j)$ está fuera del rango $[i..j - 1]$ entonces dicha cuerda **no** puede formar parte de ningún subconjunto viable y por lo tanto $T(i, j) = T(i, j - 1)$.

Para calcular $T(0, 2n - 1)$ según se nos pide en el apartado (b) aplicamos técnicas estándar de PD. Una matriz $2n \times 2n$ guarda los valores $T_{ij} := T(i, j)$. De hecho solo necesitamos guardar la triangular superior pues $T_{ij} = 0$ si $i \geq j$. Haremos un preprocessamiento que recorre el conjunto de las cuerdas (u, v) y fijamos $a[u] := v$ y $a[v] := u$. Todo ello tiene coste $\Theta(n)$.

Entonces para llenar cada $T_{i,j}$ con $i < j$ podemos hacerlo con coste $\Theta(1)$, una vez determinamos si $k = a[j]$ está dentro o fuera del rango $[i..j - 1]$. Solo necesitaremos acceder a $T_{i,j-1}$ o a $T_{i,k-1}$ y $T_{k+1,j-1}$. Fijaos que esto exige que en el momento en que llenamos $T_{i,j}$ estén ya llenadas todas las entradas de la fila i con una columna menor que j y todas las filas mayores que i (al menos hasta la columna j). Es por ello que la

matriz T debe rellenarse de abajo ($i = 2n - 2$) a arriba ($i = 0$) y de izquierda ($j = i + 1$) a derecha ($j = 2n - 1$), una vez ponemos a 0 todas las componentes T_{ij} con $i \geq j$. El coste del algoritmo es $\Theta(n^2)$ tanto en tiempo como en espacio, una vez realizado el preproceso que nos da el vector a (con coste $\Theta(n)$ en tiempo y en espacio).

Solución 3.65 Este problema se asemeja al problema 3 de la partición de secuencias. Definimos $C_{i,p}$ como el coste de una partición óptima de la subcadena $X[i..n]$ en p trozos no vacíos. Naturalmente querremos obtener $C_{1,k}$. La recurrencia es

$$C_{i,p} = \min_{i < k \leq n} \{ \max\{C_{k,p-1}, v_{i,k-1}\} \}$$

si $i \leq j$ y $p > 1$, y donde

$$v_{i,j} = \prod_{\sigma \in \Sigma} (\text{freq}(\sigma, x[i..j]) + 1)$$

Esto es, la partición óptima de $x[i..n]$ en p trozos saldrá de escoger un primer trozo $x[i..k - 1]$, partir la restante subcadena $x[k..n]$ óptimamente en $p - 1$ trozos; el coste de dicha partición es el del valor máximo de una de las p partes. Y por supuesto, como queremos minimizar dicho coste, tendremos que elegir el valor de k de la mejor manera posible.

Por otro lado $C_{i,1} = v_{i,n}$ y si $p > n - i + 1$ tomaremos $C_{i,p} = +\infty$ puesto que no se puede trocear la subcadena $X[i..n]$ en p trozos no vacíos, al haber solo $n - i + 1 < p$ símbolos.

Por otro lado la matriz de pesos $v_{i,j}$

$$v_{i,j} = \prod_{\sigma \in \Sigma} (\text{freq}(\sigma, x[i..j]) + 1)$$

puede calcularse en tiempo $O(n^2)$ si vamos guardándonos información actualizada sobre la frecuencia de cada símbolo en el segmento $x[i..j]$. Si definimos $F_{i,j}(\sigma) = \text{freq}(\sigma, x[i..j])$ entonces $F_{i,j}(\sigma) = F_{i,j-1}(\sigma)$ si $x[j] \neq \sigma$ y $F_{i,j}(\sigma) = F_{i,j-1}(\sigma) + 1$ si $x[j] = \sigma$. Suponiendo que $|\Sigma| = O(1)$ (lo cual es cierto cuando $\Sigma = \{A, C, G, T\}$) solo necesitamos coste constante para calcular cada $F_{i,j}(\sigma)$ y cada $v_{i,j}$ a partir de valores anteriores, y en definitiva el precómputo de la tabla v requiere solo tiempo $O(n^2)$ (y obviamente espacio $O(n^2)$).

El cálculo de $C_{i,p}$ requiere tiempo $O(n - i)$ si disponemos de las $v_{i,j}$'s y el coste de obtener toda la tabla de C 's será $O(n^2k)$ —incluyendo el coste del preproceso que es $O(n^2)$. El coste en espacio es $O(|\Sigma| \cdot n^2 + n^2 + nk) = O(n^2)$, correspondiente a guardar las tablas $F_{i,j}(\sigma)$, $v_{i,j}$ y $C_{i,p}$.

Solución 3.66 ■ Si tenim $n = 6$, $d_1 = d_2 = 5$, $d_3 = -3$, $d_4 = d_5 = d_6 = 0$, l'algorisme marca esdeveniment 2 com a illegal i troba els esdeveniments $\{3, 6\}$. La solució òptima, en canvi, és $\{4, 5, 6\}$.

■ Definim $S(k, j)$ com el màxim nombre d'esdeveniments observables del conjunt $\{1, \dots, k\}$ si al final la posició del telescopi és j (j entre $-n - 1$ i $n + 1$).

També definim $C(k, j)$ com el conjunt d'esdeveniments que correspon a $S(k, j)$.

Volem calcular $S(n, d_n)$ i $C(n, d_n)$.

Observem que, quan $k = 1$, només podem moure el telescopi com a molt un grau. Llavors tenim que $S(1, j) = 0$, per $j \notin \{-1, 0, 1\}$ (en aquests casos també tenim $C(1, j) = \emptyset$). $S(1, j) = 1$ per $j \in \{-1, 0, 1\}$, si $d_1 = j$ (en aquest cas $C(1, j) = \{1\}$), i 0 altrament (en tal cas $C(1, j) = \emptyset$).

També tenim $S(i, n+1) = S(i, -n-1) = 0$ per tot i (per tant $C(i, n+1) = C(i, -n-1) = \emptyset$). Ja que el telescopi no es pot moure més de n graus.

En general, podem moure el telescopi un grau o no moure'l. Això ens dona,

$$S(k, j) = \max\{S(k-1, j-1), S(k-1, j), S(k-1, j+1)\},$$

si $d_k \neq j$ (en tal cas guardem la informació per quin dels tres valors s'assoleix el màxim; és a dir, si el màxim és $S(r, m)$, doncs $C(k, j) = S(r, m)$) , i

$$S(k, j) = \max\{S(k-1, j-1), S(k-1, j), S(k-1, j+1)\} + 1,$$

si $d_k = j$ (en aquest cas, si el màxim és $S(r, m)$, doncs $C(k, j) = C(r, m) \cup \{k\}$.)

La recurrència es pot implementar en temps $O(1)$ per element fent servir un recomptat per files. El cost total de l'algorisme és $O(n^2)$ i l'espai addicional també és $O(n^2)$.

Solución 3.68 No, no es una buena solución.

Los pesos en el grafo son no negativos, si el grafo tiene un ciclo con peso positivo al multiplicar por -1 los valores tendrá un ciclo con peso negativo y las distancias no están definidas.

De hecho calcular el camino con peso máximo en un grafo con pesos no negativos es NP-difícil. (Longest path problem).

Solución 3.69 En el curso hemos visto un algoritmo para resolver este problema en grafos dirigidos acíclicos en tiempo $O(n+m)$. Este algoritmo calcula un orden topológico y después para cada vértice, en este orden, calcula la distancia mínima a d a partir de las distanciasmínimas a d de sus sucesores.

En este caso el grafo es *layered*, por ello podemos evitar el cálculo del orden topológico, ya que un BSF desde s nos proporciona en cada nivel los layers del grafo. Empezaríamos desde d (último nivel) y calcularíamos distancias mínimas a d de todos los vértices en el nivel actual, a partir de las distancias a d en el nivel siguiente teniendo en cuenta la lista de vecinos y relajando las aristas correspondientes.

Solución 3.70 Para resolver el problema tenemos que ver si hay un ciclo en el grafo tal que el producto del peso asignado a sus aristas es mayor que 1.

Es decir buscamos camino $u_0, u_1, \dots, u_k, u_{k+1} = u_0$ tal que

$$\prod_{i=0}^k c_{i,i+1} > 1,$$

donde c_{ij} es el factor de conversión de la moneda i a la moneda j .

Podemos transformar el producto en sumas usando logaritmos,

$$\log 1 < \log\left\{\prod_{i=0}^k c_{i,i+1}\right\} = \sum_{i=0}^k \log c_{i,i+1},$$

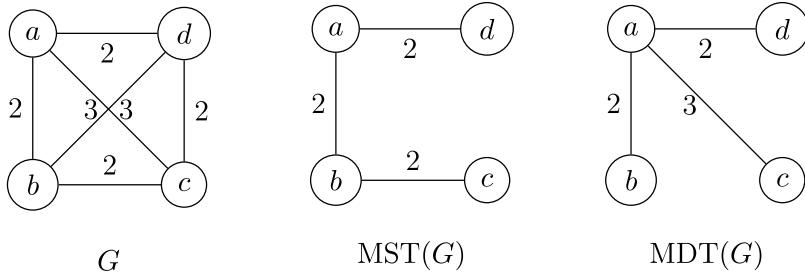
Multiplicando por -1 , y teniendo en cuenta que $\log 1 = 0$,

$$0 > \sum_{i=0}^k -\log c_{i,i+1}.$$

Si construimos un grafo completo con peso en la arista (i,j) , $w_{i,j} = -\log c_{ij}$ tenemos un grafo dirigido, con pesos positivos y negativos, en el que tenemos que ver si contiene o no un ciclo con peso negativo. Lo podemos hacer utilizando el algoritmo de Bellman-Ford que nos permite comprobar esta condición en tiempo $O(nm^2)$, con una ejecución para cada vértice como vértice inicial.

Sin embargo al ser un grafo completo es suficiente con ejecutar Bellman-Ford desde un único vértice origen, ya que todos los ciclos en el grafo son accesibles desde él. Así el coste es $= (nm)$.

Solución 3.71 1. Falso. Considerad el siguiente grafo,



En él un MST tiene coste 6, mientras que un árbol de distancias mínimas utiliza una de las aristas de peso 3 con coste total 7.

2. Falso.

Si T es un MST una de las aristas de peso mínimo que sale de s pertenece a él.

En un árbol de distancias mínimas T' , todas las aristas de peso mínimo que salen de s pertenecen a T' . Por lo tanto T y T' comparten al menos una arista.

Solución 3.72 Podemos multiplicar por -1 todos los pesos de las aristas. El DAG no tiene ciclos por tanto, con esta modificación, no generamos ciclos con peso negativo. Podemos utilizar Bellman-Ford para calcular las distancias con coste $O(nm)$.

También podemos utilizar el algoritmo para cálculo de distancias mínimas en un DAG. El algoritmo utiliza el orden topológico y tanto la recurrencia como los casos base funcionan independientemente de que los pesos sean positivos o negativos. Lo que nos da un algoritmo con coste $O(n + m)$.

Solución 3.75 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 cèpiga 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

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

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

Solución 3.76 Farem servir la notació $G(i, j)$ per al graf que conté les arestes comuns als grafs G_i, \dots, G_j ($i \leq j$).

Per la part (a) podem fer servir Dijkstra, al graf $G(0, m)$, per trobar la distància de s i t i un camí amb aquesta distància. Però tenint en compte que les arestes del graf no tenen pes, podem trobar un camí de distància mínima fent un BFS desde s .

Podem obtenir el graf que conté només les arestes comunes als grafs $G_0, G_1, G_1, \dots, G_m$ fent, per a cada vèrtex i ,

- inicialitzar un vector $N[n]$ a 1's excepte per la posició i que es posa a 0.
- per a cada graf j , obtenir en un vector $A[n]$ on estan marcat a 1 els veïns de i a G_j , fent un recorregut de la seva llista d'adjacència. Després fem $N = N \wedge A$.
- afegim a la llista d'adjacència de i els vèrtexs amb $N[j] == 1$.

Observem que el el BFS té cost $O(n + |E_0|)$ però construir $G(0, m)$ té cost

$$O(n + |E_0| + \dots + |E_m|)$$

i aquest és el cost de l'algorisme.

Per a la part (b). Si P_0, P_1, \dots, P_m és una solució óptima, o bé $P_0 = P_1 = \dots = P_m$ (i estem al cas (a)) o podem trobar el valor més petit de j tal que $P_{j+1} = P_{j+2} = \dots = P_m$ ($j \geq 0$).

En aquest segon cas, P_m és un camí mínim a $G(j+1, m)$ (i a cascú dels grafs G_{j+1}, \dots, G_m) i P_0, P_1, \dots, P_j és una solució óptima del problema definit pels grafs G_0, \dots, G_j . A més el cost de la solució és,

$$CT(P_0, \dots, P_{j-1}) + 1 + (m-j)|P_m|.$$

Sigui $C(i)$ el cost mínim de la solució per G_0, \dots, G_i , fent servir l'estructura de suboptimalitat tenim la següent recurrència:

$$C(i) = \min\{(i+1)\ell(0, i), \min_{0 \leq j < i} \{C(j) + (i-j)\ell(j+1, i) + 1\}\}.$$

on $\ell(i, j)$ és la distància mínima de s a t en $G(i, j)$ i aquesta distància també és mínima als grafs G_i, \dots, G_j , $+\infty$ en cas contrari o si no hi ha camí de s a t a $G(i, j)$.

La recurrència ens dona un algorisme amb temps $O(m)$ per element si podem accedir en temps constant als valors $\ell(i, j)$.

Podem fer un preprocès per obtenir el grafs $G(i, j)$ i els valors $\ell(i, j)$ per tots els $O(m^2)$ parells (i, j) amb $0 \leq i \leq j \leq m$.

Per $0 \leq i \leq m$, els grafs $G(i, j)$ per $j > i$ es poden calcular incrementalment, $G(i, j)$ es el graf que conté les arestes comuns a $G(i, j-1)$ i G_j . Si definim $E = \max_{0 \leq i \leq m} |E|_i$, el cost per graf és $O(n+E)$. Aquest cost també es correcte per als elements de ℓ . Així tenim un cost total de preprocès $O(m^2(n+E))$.

El cost total de l'algorisme és $O(m^2(n+E))$ que és polinòmic en la mida de l'entrada. És pot implementar amb una taula amb un recorregut per files.

Per obtenir la solució hauríem de mantenir els punters adients per tal de reconstruir els enters on el camí canvia. Una vegada tenim aquest punters, per cada interval hauríem (amb un BFS) de trobar el camí de s a t . El cost addicional és $O(m(n+E))$ que es menor que el de la implementació de la recurrència.

Solución 3.77 Este problema es en esencia idéntico al problema de la partición lineal (problema 3). otros de la colección. Llamamos $T_{i,r}$ al coste (número de días que demora hacer las copias) óptimo para la subsecuencia de los libros i a n usando r monjes. Sea $W_{i,j} = p_i + \dots + p_j$ la suma de las páginas de los libros i a j , $i \leq j$ (podemos asumir $W_{i,j} = 0$ si $i > j$). Entonces la base de la recursión es $T_{i,0} = +\infty$ para cualquier i , $1 \leq i \leq n$ (no se pueden copiar libros si no hay monjes) y $T_{n,r} = p_n$ si $r \geq 1$ (el tiempo en realidad es $c \cdot p_n$, para alguna constante de proporcionalidad, pero es irrelevante; podemos tomar $c = 1$, una página=una unidad de tiempo). Si $i < n$ y $r \geq 1$ entonces

$$T_{i,r} = \min_{i < k \leq n} (\max\{W_{i,k-1}, T_{k,r-1}\}),$$

esto es, tenemos que optimizar entre todas las posibles elecciones de k siendo el coste entonces el mayor tiempo entre lo que tarda el monje que copia los libros i a $k - 1$ y el tiempo que tardan los otros $r - 1$ monjes encargados de copiar, con una distribución óptima, los libros k a n . Usando las técnicas estándar de PD este problema se resuelve con coste $O(n^2m)$ en tiempo y $O(nm + n^2)$. Para conseguirlo uno debe tener una matriz $\Theta(nm)$ para las T 's y una matriz W para las $W_{i,j}$. Observad que la matriz se debe llenar de abajo a arriba: la fila i depende de las filas por debajo, las filas k con $k > i$.

Solución 3.78 Un problema muy semejante al problema #56. La clave es definir $R(i, j)$ el número máximo de robots que se pueden destruir a partir del dia i asumiendo que i es el j -ésimo dia desde la última activación del PEM. Entonces $R(n, j) = \min\{x_n, f(j)\}$ para toda j , $1 \leq j \leq n$, y en general

$$R(i, j) = \max(\min(x_i, f(j)) + R(i + 1, 1), R(i + 1, j + 1)), \quad 1 \leq i < n, 1 \leq j \leq n.$$

Podemos fijar $R(i, j) = -\infty$ si $i < j$ pues no pueden haber transcurrido más de i días desde la última PEM. El problema se puede resolver rellenando la tabla de las $R(i, j)$'s de tamaño $\Theta(n^2)$ en tiempo $\Theta(n^2)$ puesto que para calcular cada entrada solo se requiere tiempo constante. Observad que para calcular los valores de la fila i se necesitan valores de la fila $i + 1$ por lo que hay que comenzar de abajo hacia arriba. Por otro lado, salvo que se quiera reconstruir los instantes en los que activar el PEM, podemos usar solo espacio lineal guardando dos vectores con las filas i e $i + 1$ de la matriz R .

Solución 3.79 Naturalmente, si el número de tipos de muñeca que hay es n nunca podremos anidar más de n muñecas, ni en la mejor de las circunstancias; pero denotemos $M = M(\{(h_i, w_i)\}_{1 \leq i \leq n})$ el máximo número de matrioshkas que podemos encajar, y M_i el número máximo de matrioshkas encajables siendo una matrioshka de tipo i la más externa. Entonces $M = \max_{1 \leq i \leq n} M_i$ y podremos calcularlo en tiempo $O(n)$ una vez hallados los valores M_i . Pero para poder establecer una recurrencia para los M_i 's necesitamos establecer un orden entre las matrioshkas. Si ordenamos las muñecas por altura creciente y en caso de empate por anchura creciente, e indexamos los tipos de muñeca de acuerdo con este orden entonces sabemos que ninguna muñeca de tipo j con $j \geq i$ podrá ser encajada dentro de una muñeca de tipo i . Si denotamos $L_{i,j} = \text{true}$ si una muñeca de tipo i puede albergar una muñeca de tipo j entonces $L_{i,j} = \text{false}$ si $i \leq j$; y puede ser cierto o falso cuando $i > j$.

Ahora ya podemos establecer la recurrencia. En primer lugar tenemos $M_1 = 1$ pues ninguna muñeca puede estar dentro de una muñeca nº1. Y

$$M_i = \begin{cases} 1 + \max_{1 \leq j < i \wedge L_{i,j}} \{M_j\}, & \text{si existe } j \text{ tal que } L_{i,j}, \\ 1, & \text{en otro caso.} \end{cases}$$

La primera parte de la recurrencia sale de descomponer la solución óptima con una muñeca de tipo i como más externa en: (1) la muñeca i fuera y (2) inmediatamente dentro de la muñeca i colocar una solución óptima con una muñeca de tipo j como muñeca más externa.

El valor M_i puede calcularse con coste $\Theta(i)$ y el vector de M 's rellenarse con coste cuadrático. No hace falta precalcular los $L_{i,j}$'s y almacenarlos. Puede calcularse *on-the-fly* si una muñeca j cabe dentro de una muñeca i .

Solución 3.80 Sea $\mathcal{P}_{i,j} = \text{true}$ si la subcadena $a_i \dots a_j$ es un palíndromo y falso en caso contrario. Entonces si llamamos T_i al número mínimo de trozos que hay que dar para dividir $a_i \dots a_n$ en subcadenas palíndromas tendremos que $T_n = 1$ y

$$T_i = \min_{i < k \leq n \wedge \mathcal{P}_{i,k-1}} \{1 + T_k\}, \quad 1 \leq i < n.$$

En la recurrencia previa, siempre habrá al menos un valor de k tal que $\mathcal{P}_{i,k-1} = \text{true}$; concretamente $k = i + 1$, puesto que $\mathcal{P}_{i,i} = \text{true}$ para cualquier i , $1 \leq i \leq n$. El cálculo de cada T_i requiere un coste $O(n - i)$ —usando la tabla precalculada $\mathcal{P}_{i,j}$ — y el coste del valor óptimo buscado T_1 requerirá tiempo $O(n^2)$ más el tiempo de preprocesso. El coste en espacio viene dominado por el coste en espacio de almacenar los \mathcal{P} 's que es $\Theta(n^2)$, mientras que los valores T 's los tendremos en un vector de tamaño $\Theta(n)$. Si prescindimos del preprocesso que nos da los $\mathcal{P}_{i,j}$'s entonces podemos calcular con coste $O(n^2)$ cada T_i y con coste $O(n^3)$ la solución óptima buscada.

Para concluir vamos a ver como podemos calcular $\mathcal{P}_{i,j}$ en tiempo $O(n^2)$. Básicamente podemos hacerlo mediante “programación dinámica”! En efecto, $\mathcal{P}_{i,i} = \text{true}$, $\mathcal{P}_{i,i+1} = (a_i = a_{i+1})$, $\mathcal{P}_{i,j} = (a_i = a_j) \wedge \mathcal{P}_{i+1,j-1}$. De la matriz solo necesitamos la triangular superior y es fácil llenarla, usando las propiedades que acabamos de ver, pero cuidado, hay que llenar la matriz por diagonales NW-SE, la diagonal k -ésima se caracteriza por cumplir $j - i = k$.

```
void precalcular_segmentos_palindromicos(vector<bool>& P, const string& a) {
    int n = a.size();
    // P = matriz de n x n booleanos
    // coste de la construcción: Theta(n^2)
    P = vector<vector<bool>>(n, vector<bool>(n));

    // Coste de los dos primeros bucles: $\\Theta(n)$
    for (int i = 0; i < n; ++i) P[i][i] = true;
    for (int i = 0; i < n-1; ++i) P[i][i+1] = (a[i] == a[i+1]);

    // Coste del bucle principal:
    // sum_{2 <= k < n} Theta(n-k) = Theta(n^2)

    for (int k = 2; k < n; ++k) {
        for (int i = 0, int j = k; j < n; ++i, ++j)
            // P[i+1][j-1] está en
            // la diagonal j-1-(i+1) = j-i-2 = k-2 ya calculada
            P[i][j] = (a[i] == a[j]) and P[i+1][j-1];
    }
}
```