


Problemes 2

- 2.1.  Donat un conjunt $\{x_1, x_2, \dots, x_n\}$ de punts de la recta real, doneu un algorisme, el més eficient que pogueu, per a determinar el conjunt més petit d'interval·ls tancats amb longitud unitat, que cobreixen tots els punts (cada punt ha d'aparèixer almenys a un interval).

Una solució: Para ver como resolver el problema voy a considerar que el conjunto de puntos X están ordenados en orden creciente de valor ($x_1 \leq x_2 \leq \dots \leq x_n$). Si no lo estuviesen, se pueden ordenar en tiempo $O(n \log n)$ utilizando merge sort.

Una solución óptima Y al problema se puede representar por una secuencia creceinte $y_1 < y_2 < \dots < y_k$ de valores, indicando los puntos de inicio de los k intervalos de longitud 1 que forman Y . La secuencia es estrictamente creciente ya que si no lo fuese tendríamos intervalos repetidos y la solución no sería óptima.


Si Y es óptima, cada intervalo $[y_j, y_j + 1]$ tiene que contener al menos un punto de X . Si no, podríamos eliminar el intervalo y podríamos cubrir todos los puntos con un intervalo menos.

Además, si desplazamos el inicio del intervalo al primer punto de X que cubre, seguimos teniendo una solución óptima. Ya que cada intervalo cubre al menos todos los puntos que ya cubría antes y sigue siendo una solución. Así tenemos que siempre hay una solución óptima en la que los puntos de inicio de los intervalos son valores en el conjunto de puntos dado.

Utilizando esta idea de desplazar a la derecha podemos encontrar siempre una solución óptima $x_{i_1} < x_{i_2} < \dots < x_{i_k}$ en la que ningún punto del conjunto inicial está cubierto por más de un intervalo. Basta seguir los intervalos en orden y desplazar el inicio del intervalo $i + 1$ -ésimo hasta el primer punto cubierto por el intervalo $i + 1$ que no esté cubierto por el intervalo i .

Si analizamos este último tipo de solución óptima, $x_{i_1} < x_{i_2} < \dots < x_{i_k}$ en la que los conjuntos de puntos cubiertos por cada intervalo son disjuntos dos a dos, tenemos que, $x_{i_1} = x_1$, si no x_1 no estaría cubierto. Además, $x_{i_{j+1}}$ tiene que ser el primer punto en X con valor mayor que $x_{i_j} + 1$, ya que si no este valor no estaría cubierto en la solución.

Esta última solución óptima se puede obtener en tiempo $O(n)$ asumiendo que los puntos estén ordenados, y en tiempo $O(n \log n)$ si tenemos que ordenarlos.

- 2.2.  Un grup de n amics ha de comprar un regal que val C euros, on C és un enter no negatiu. Tenim una llista amb els pressupostos B_i de cadascun dels amics, és a dir, una llista \mathbf{B} de n enters positius $\mathbf{B} = (B_1, \dots, B_n)$.

Per fer la compra hem de determinar (si és possible) una *aportació*, una llista de quantitats $X = (x_1, \dots, x_n)$, essent x_i la quantitat que aporta l'amic i . L'aportació ha de cobrir el cost del regal, és a dir, $\sum_{i=1}^n x_i = C$. A més, l'aportació particular de cap amic no pot superar mai el seu pressupost, és a dir, per $1 \leq i \leq n$, $x_i \leq B_i$.

El cost d'una aportació X és $c(X) = \max\{x_i \mid 1 \leq i \leq n\}$. Diem que una aportació \mathbf{x}^* és *equitativa* si el seu cost és mínim amb relació al conjunt de totes les possibles aportacions.

Per exemple, suposem que $C = 100$, $n = 3$ i $\mathbf{B} = (3, 45, 100)$. Llavors és possible comprar el regal i una aportació equitativa és $\mathbf{x}^* = (3, 45, 52)$. Si els pressupostos foren $\mathbf{B} = (3, 100, 100)$, una aportació equitativa seria $\mathbf{x}^* = (3, 48, 49)$, però en canvi $\mathbf{x}^* = (3, 45, 52)$ no ho seria.

- Sigui B_{\min} el pressupost més baix. Demostra que si el regal es pot comprar i $nB_{\min} < C$ hi ha una aportació equitativa en la qual tots els amics amb pressupost B_{\min} aporten B_{\min} .
- Proporciona un algorisme golafre que determini si es pot o no comprar el regal i, en cas afirmatiu, retorni una aportació equitativa.

Una solució:

- Donat que el regal es pot comprar existeix al menys una solució equitativa. D'altra banda, com que $nB_{\min} < C$ existeix al menys un pressupost $B_j > B_{\min}$. Demostrarem aquest apartat per reducció a l'absurd. Es a dir, suposem que per a tota aportació equitativa \mathbf{x}^* existeix un amic i amb pressupost B_{\min} però $x_i^* < B_{\min}$; sense pèrdua de generalitat podem suposar que aquest amic és l'amic $i = 1$, $B_1 = B_{\min}$. Sigui \mathbf{x}^* una aportació equitativa i $\Delta(\mathbf{x}^*) = B_1 - x_1^* > 0$. Sigui B_j el pressupost de l'amic que més diners aporta (x_j^* és màxim, $c(\mathbf{x}^*) = x_j^*$) i $x_j^* > x_1^*$ (altrament el regal no podria ser comprat). Llavors podem obtenir una nova aportació \mathbf{x}' tal que $x_1' = x_1^* + 1$, $\Delta(x_1') = \Delta(x_1^*) - 1$, i $x_j' = x_j^* - 1$. Per tant, $c(\mathbf{x}') \leq c(\mathbf{x}^*)$, i tenim una contradicció si $c(\mathbf{x}') < c(\mathbf{x}^*)$. Així que $c(\mathbf{x}') = c(\mathbf{x}^*)$ i \mathbf{x}' és també equitativa, doncs té el mateix cost que \mathbf{x}^* . Per a que això passi, hem de tenir al menys un altre amic j' que fa aportació màxima $x_{j'}^* = x_j^*$. I d'altra banda o bé $\Delta(x_1') > 0$ o bé $\Delta(x_{i'}) > 0$ per un cert i amb $B_i = B_{\min}$, doncs la nostra hipòtesi (per fer a la reducció a l'absurd) és que per a tota aportació equitativa hi ha al menys un amic amb pressupost mínim que no aporta tot el seu pressupost. Així podríem obtenir una nova aportació \mathbf{x}'' que també és equitativa però j' aporta una unitat menys al regal i l'amic 1 (o i) aporta una unitat més al regal, i iterar el mateix raonament fins a concloure que existeix una aportació equitativa per a la qual tots els amics de pressupost mínim aporten tots els seus diners, en contradicció amb la nostra hipòtesi de partida.

Una demostració alternativa. D'una banda es pot comprar el regal i per a qualsevol solució ("aportació") \mathbf{x} es verifica

$$\sum_{i=1}^n x_i = C.$$

Sigui $k < n$ el número d'amics amb pressupost mínim B_{\min} ($k \neq n$ ja que $n \cdot B_{\min} < C$). Sense pèrdua de generalitat podem suposar que els amics amb pressupost mínim són els amics $1, 2, \dots, k$. Llavors, considerem el problema amb els $n' = n - k$ amics amb pressupost $> B_{\min}$ i cost del regal $C' = C - k \cdot B_{\min}$, i sigui \mathbf{x}' una aportació equitativa per aquest nou problema. Tal aportació segur que existeix, ja que els n' amics poden comprar un regal de cost C' .

Tornant al problema original, sigui $\bar{\mathbf{x}}$ definida de la següent manera:

$$(B_{\min}, \dots, B_{\min}, x'_{k+1}, \dots, x'_n)$$

En aquesta aportació $\bar{\mathbf{x}}$ tots els amics amb pressupost mínim B_{\min} aporten tots els seus diners. És vàlida:

$$\sum_{i=1}^n \bar{x}_i = k \cdot B_{\min} + \sum_{i=k+1}^n x'_i = k \cdot B_{\min} + C' = C.$$

I finalment, és equitativa. Com que $\sum_{i=k+1}^n x'_i = C'$, es dedueix que

$$c(\mathbf{x}') = \max_{k < i \leq n} \{x'_i\} \geq \left\lfloor \frac{C'}{n'} \right\rfloor,$$

Observem que $c(\bar{\mathbf{x}}) = c(\mathbf{x}') \geq B_{\min}$, ja que $c(\mathbf{x}') \geq \lfloor C'/n' \rfloor \geq B_{\min}$:

$$C > n \cdot B_{\min} \Rightarrow C - k \cdot B_{\min} > n \cdot B_{\min} - k \cdot B_{\min} \Rightarrow \frac{C - k \cdot B_{\min}}{n - k} > B_{\min} \Rightarrow \left\lfloor \frac{C - k \cdot B_{\min}}{n - k} \right\rfloor \geq B_{\min}.$$

Suposem que $c(\bar{\mathbf{x}})$ **no** es mínima. És a dir, existeix una \mathbf{x}^* pel problema amb cost C i n amics, equitativa amb $c(\mathbf{x}^*) < c(\bar{\mathbf{x}})$. La suma de les aportacions dels amics $k+1$ a n en \mathbf{x}^* ha de ser $C^* \geq C - k \cdot B_{\min}$. Llavors considerant només $x^*[k+1..n]$ com a solució del problema amb cost C^* per a $n-k$ amics $c(\mathbf{x}^*) \geq c(\mathbf{x}')$ doncs els $n' = n - k$ ara han de pagar un regal més car i \mathbf{x}' és equitativa. Però $c(\mathbf{x}') = c(\bar{\mathbf{x}})$ i arribem a una contradicció.

(b) Aquest és l'algorisme golafre que proposem, amb cost $\Theta(n \log n)$:

```

if (B[1]+B[2]+...+B[n] < C) {
    cout << "el regal no es pot comprar" << endl;
    return false;
} else {
    ordenar els amics de menor a major pressupost
    // B[1] <= B[2] <= ... <= B[n]
    i = 1;
    while ((n+1-i) * B[i] < C) {
        x[i] = B[i];
        C = C - B[i];
        ++i;
    }
    // el remanent C es distribueix equitativament entre els (n-i+1)
    // amics que encara no han aportat, els seus pressupostos són
    // tots >= B[i] i B[i] * (n-i+1) >= C; als últims r = C mod (n+1-i)
    // amics els fem aportar una unitat més cadascú---al menys
    // hi ha r amics amb pressupost >= q+1
    q = C / (n+1-i); r = C % (n+1-i)
    for (j = i; j <= n; ++j) {
        x[j] = q;
        if (i + r > n) ++x[j];
    }
    return x;
}

```

L'apartat previ demostra que si $nB_{\min} < C$ llavors existeix una aportació equitativa en la qual tots els amics amb pressupost mínim aporten tots els seus diners. Aquest criteri s'aplica iterativament: l'amic amb pressupost mínim aporta tots els seus diners i recursivament s'ha de fer una distribució equitativa dels diners pendents entre els $n-1$ amics restants. Es pot fer

iterativament fins que només queden n' amics per aportar, tots amb pressupost $\geq B'_{\min} =$ “el pressupost més petit dels n' amics”, i el import pendent de pagar és $C' \leq n' B'_{\min}$. En aquest cas és evident que la aportació equitativa és aquella en la que tots els n' amics paguen $q = \lfloor C'/n' \rfloor$ o $q + 1$ (alguns d’ells, no tots, paguen $q + 1$).

I aquesta és exactament l’aportació calculada pel nostre algorisme. Una solució alternativa:

```
if (B[1]+B[2]+...+B[n] < C) {
    cout << "el regal no es pot comprar" << endl;
    return false;
} else {
    ordenar els amics de menor a major pressupost
    // B[1] <= B[2] <= ... <= B[n]
    for (int i = 1; i <= n; ++i) {
        x[i] = min(B[i], C/(n-i+1));
        C = C - x[i];
    }
    return x;
}
```

- 2.3. **(Revisió)** Un professor rep n sol·licituds de revisions d'examen. Abans de començar, el professor mira la llista dels n estudiants que han sol·licitat revisió i pot calcular, per a cada estudiant i , el temps t_i que utilitzarà per atendre l' i -èsim estudiant. Per a estudiant i , el temps d'espera e_i és el temps que el professor triga a revisar els exàmens dels estudiants que fan la revisió abans que i .

Dissenyeu un algorisme per a computar l'ordre en que s'han de revisar els exàmens dels n estudiants de manera que es minimitzi el temps total d'espera: $T = \sum_{i=1}^n e_i$.

- 2.4. Tenim un graf no dirigit $G = (V, E)$. Donat un subconjunt $V' \subseteq V$ el *subgraph induït* per V' és el graf $G[V'] = (V', E')$ on $E' = E \cap (V' \times V')$, és a dir, conté totes les arestes que tenen els dos extrems a V' . El grau d'un vèrtex a un graf és el nombre d'arestes incidents al vèrtex. Doneu un algorisme eficient per al següent problema: donat G i un enter positiu k , trobar el subconjunt (si hi ha algun) més gran V' de V , tal que cada vèrtex a V' té grau $\geq k$ a $G[V']$.

- 2.5. El problema de la *partició interval* (*Interval Partitioning Problem*) és similar al problema de la selecció d'activitats vist a classe però, en lloc de tenir un únic recurs, tenim molts recursos (és a dir, diverses còpies del mateix recurs). Doneu un algorisme que permeti programar totes les activitats fent servir el menor número possible de recursos.

2.6. Sigui $G = (V, E)$ un graf no dirigit. Un subconjunt $C \subseteq V$ s'anomena *recobriments de vèrtexs* de G si


$$\forall \{u, v\} \in E : \{u, v\} \cap C \neq \emptyset.$$

Donat $G = (V, E)$, un recobriments de vèrtexs C és diu que és *minimal* si per qualsevol $C' \subseteq V$ a G , tal que $C' \subset C$ hem de tenir que C' no és un recobriments de vèrtexs.

Un conjunt $C \subseteq V$ és un *recobriments de vèrtexs mínim* si C és un recobriments de vèrtexs a G amb mínima cardinalitat. (Quan G és un arbre, hi ha un algorisme polinòmic per a trobar un recobriments de vèrtexs mínim a G , però per a G generals el problema és NP-hard).

En aquest problema demostrareu que, a diferència del problema de trobar un recobriments de vèrtexs mínim, el problema de trobar un recobriments de vèrtexs minimal pot ser resolt en temps polinòmic.

- (a) Demostreu que un recobriments de vèrtexs minimal no necessàriament ha de ser un recobriments de vèrtexs mínim.
- (b) Demostreu que tot recobriments de vèrtexs mínim també és minimal.
- (c) Doneu un algorisme polinòmic per trobar un recobriments de vèrtexs minimal a G .

- 2.7.  Sigui X un conjunt de n intervals a la recta real. Una coloració pròpia de X assigna un color a cada interval, de manera que dos intervals que se superposen tenen assignats colors diferents. Descriviu i analitza un algorisme golafre eficient per obtenir el mínim nombre de colors necessaris per acolorir (amb una coloració pròpia) un conjunt d'intervals X . Podeu assumir que l'entrada està formada per dos vectors $L[1..n]$ i $R[1..n]$, representant els extrems esquerres (L) i drets (R) dels intervals a X .

Una solució: Ordenem els vectors L i R de manera que estiguin en ordre creixent d'extrem esquerre:

$$L[1] \leq L[2] \leq \dots \leq L[n]$$

Suposem que hem determinat que s'han necessitat k^* colors per la coloració dels intervals 1 a $i-1$. Més encara, hi ha $k \leq k^*$ colors aparentment en ús (per cobrir el punt $L[i] - \epsilon$). Una estructura de dades conté la informació dels k intervals (específicament els seus extrems drets) que té un dels k colors assignats i suposarem que podem accedir i eliminar eficientment d'aquesta estructura (un min-heap) l'interval que té R mínima. Sigui R_{\min} aquest valor. Llavors si $R_{\min} \leq L[i]$ llavors el color assignat al interval corresponent es pot assignar al interval $X[i] = (L[i], R[i])$. S'elimina de l'estructura l'interval amb R_{\min} , que tenia assignat el color j , $1 \leq j \leq k$, i s'afegeix $X[i]$, assignat-li el color j i la seva prioritat seria $R[i]$. Si $R_{\min} > L[i]$ tots els k colors estan en ús i s'haurà d'assignar un nou color, el $k+1$, a l'interval $X[i]$ i s'afegirà a l'estructura amb prioritat $R[i]$. Si $k^* = k$ llavors actualitzem $k^* := k+1$. En acabar, k^* és el mínim nombre de colors necessari que s'ens demana.

El cost de l'algorisme és $O(n \log n)$ per a ordenar els n intervals per L creixent i $O(n \log n)$ per a processar els n intervals: a cada una de les n iteracions s'ha d'actualitzar el min-heap amb una inserció (la de l'interval $X[i]$ en curs) i ocasionalment amb una eliminació del mínim quan ens consta que un color queda alliberat.

A cada iteració podríem esborrar tots els intervals del min-heap amb R més petita o igual que $L[i]$ de manera que el min-heap no contingues intervals "acabats" i que ja no necessiten tenir color assignat; però és innecessari i molt més senzill esborrar a cada iteració només un interval (o cap si no és possible).

- 2.8. ✎ Tenim un alfabet Σ on per a cada símbol $a \in \Sigma$, p_a es la probabilitat que aparegui el caràcter a . Demostreu que, per a qualsevol símbol $a \in \Sigma$, la seva profunditat en un arbre prefix que produeix un codi de Huffman òptim és $O(\lg \frac{1}{p_a})$. (Ajuts: en un arbre prefix que s'utilitzi per a dissenyar el codi Huffman, la probabilitat d'un nus és la suma de les probabilitats dels fills. La probabilitat de l'arrel és, doncs, 1.)

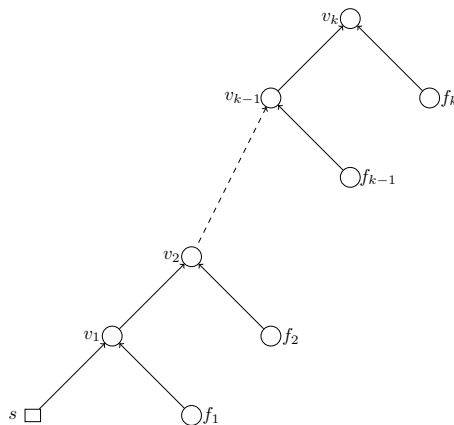
Solució: Sigui T un arbre prefix corresponent a un codi Huffman òptim, i sigui s la fulla que conte a .

Sigui $P = s, w_1, w_2, \dots, v_k$ la seqüència de nusos que van de s fins a l'arrel v_k , i siguin f_i els fills de v_i que no són a P .

Sabem que $p(v_i) = p(v_{i-1}) + p(f_j)$. Però $p(f_j) \geq p(f_{j-1})$ i $p(f_j) \geq p(v_{j-2})$, per tant $p(f_j) \geq (p(f_{j-1}) + p(v_{j-2}))/2$, tenim $p(f_j) \geq p(v_{j-1})/2$.

Posant tot junt, $p(v_j) = p(v_{j-1}) + p(f_j) \geq 1.5p(v_{j-1})$.

Utilitzant inducció podem demostrar que $p(v_k) \geq 1.5^{k-1}p(v_1) \geq 1.5^{k-1}p(a)$ i a més, $p(v_k) = 1$. Això implica $1.5^{k-1}p_a \leq 1$ i per tant la profunditat de a es $k+1 = O(\lg 1/p_a)$.



- 2.9. Si utilitzem l'algorisme de Huffman per a comprimir un text format per n , símbols que apareixen amb freqüències $f_1, f_2, f_3, \dots, f_n$ quina és la màxima longitud de compressió d'un símbol que podem obtenir? Doneu un exemple de les freqüències on es compleix aquesta condició.

- 2.10. La companyia Googol vol detectar a la web en quin idioma està escrita cada pàgina a la web. Per això, ha dissenyat un programa que quan descarrega una pagina, pot demanar si la pàgina està escrita en idioma L_i ($1 \leq i \leq 8$) i el programa respon SI o NO. Googol sap que de totes les pàgines a internet, el 40% estan escrites en L_1 , el 17% en L_2 , el 15% en L_3 , el 11% en L_4 , el 9% L_5 , el 5% L_6 , el 2% L_7 , i el 1% L_8 .
- (a) Googol, ha dissenyat un programa al que, quan descarrega una pagina, pot demanar si la pàgina esta escrita en idioma L_i ($1 \leq i \leq 8$) i el programa respon SI o NO. En quin ordre s'han a de fer les preguntes per que el nombre mitjà de preguntes necessaries per a identificar l'idioma sigui mínim?
 - (b) Un dels informàtics a Googol pensa que potser seria millor fer servir un programa que, donats dos conjunts disjunts d'idiomes, diu si la pàgina està escrita amb un idioma del primer conjunt o amb un idioma del segon. Dissenya un algorisme, que fent servir aquest tipus de qüestions, trobi l'idioma en el que està escrit la pàgina. Quin serà el nombre esperat de qüestions utilitzant el teu algorisme? (Ajut: Penseu en el codis de Huffman)

2.11. ♣(Planificació). Ens donen un conjunt de treballs $S = \{a_1, a_2, \dots, a_n\}$, a on per a completar el treball a_i es necessiten p_i unitats de temps de processador. Únicament tenim un ordinador amb un sol processador, per tant a cada instant únicament podem processar una treball. Sigui c_i el temps on el processador finalitza de processar a_i , que dependrà dels temps dels treballs processats prèviament. Volem minimitzar el temps "mitja" necessari per a processar tots els treballs (el temps amortitzat per treball), es a dir volem minimitzar $\frac{\sum_{i=1}^n c_i}{n}$. Per exemple, si tenim dos treballs a_1 i a_2 amb $p_1 = 3, p_2 = 5$, i processem a_2 primer, aleshores el temps mitja per a completar els dos treballs és $(5 + 8)/2 = 6.5$, però si processem primer el treball a_1 i després a_2 el temps mitja per processar els dos treballs serà $(3 + 8)/2 = 2.2$

- Considerem que la computació de cada treball no es port partir, es a dir quan comença la computació de a_i les properes p_i unitats de temps s'ha de processar a_i . Doneu un algorisme que planifiqui la computació dels treballs a S de manera que minimitze el temps mitja per a completar tots els treballs. Doneu la complexitat del vostre algorisme i demostreu la seva correctesa.
- Considereu ara el cas de que no tots els treballs a S estan disponibles des de el començament, es a dir cada a_i porta associat un temps r_i fins al que l'ordinador no pot començar a processar a_i . A més, podem suspendre a mitges el processament d'un treball per a finalitzar més tard. Per exemple si tenim a_i amb $p_i = 6$ i $r_i = 1$, pot començar a temps 1, el processador aturar la seva computació a temps 3 i tornar a computar a temps 10, aturar a temps 11 i finalitzar a partir del temps 15. Doneu un algorisme que planifiqui la computació dels treballs a S de manera que es minimitze el temps mitja per a completar tots els treballs.

Solució.

Notemos que en la función a optimizar, $\frac{\sum_i c_i}{n}$, el denominador no depende de la planificación. Por lo tanto la planificación con coste mínimo es la del coste medio mínimo y viceversa. Los algoritmos que propondré resuelven el problema de buscar una planificación con coste mínimo.

- El algoritmo ordena los trabajos en orden creciente de p_i , y los planifica en ese orden. El coste es el de la ordenación, $O(n \log n)$.

Para ver que es correcto utilizo un argumento de intercambio. Supongamos que la planificación con coste mínimo no sigue el orden creciente de tiempo de procesado. Para simplificar asumo que el orden a_1, \dots, a_n es el que proporciona coste óptimo y que en él se produce una inversión, es decir $p_i > p_{i+1}$, para algún i .

Tenemos que $c_i = p_1 + \dots + p_i$, por lo tanto

$$\sum_i c_i = np_1 + (n-1)p_2 + \dots + (n-i)p_i + \dots + 1p_n.$$

Si intercambiamos a_i con a_{i+1} solo cambia la contribución al coste de estos dos elementos que pasa de ser $(n-i)p_i + (n-i-1)p_{i+1}$ a ser $(n-i)p_{i+1} + (n-i-1)p_i$. El incremento en coste debido al intercambio es

$$(n-i)p_{i+1} + (n-i-1)p_i - [(n-i)p_i + (n-i-1)p_{i+1}] = p_{i+1} - p_i < 0.$$

Por tanto, la ordenación no es óptima y tenemos una contradicción.

- En este segundo apartado tendremos que seguir el criterio del apartado anterior, pero teniendo en cuenta que se incorporarán a lo largo del tiempo nuevos trabajos. La regla voraz del algoritmo es: procesar en cada instante de tiempo el proceso disponible al que le quede menos tiempo por finalizar. Utilizando el mismo argumento de intercambio que en el apartado (a) la regla voraz es correcta.

Tenemos que ir con cuidado en la implementación ya que el número total de instantes de tiempo es $\sum_i t_i$ y este valor puede ser exponencial en el tamaño de la entrada. Sin embargo, los tiempos


en los que se para la ejecución de un proceso coinciden con los de disponibilidad de un nuevo proceso. Necesitamos controlar solo los instantes de tiempo en los que finaliza la ejecución de un proceso o en los que un proceso está disponible, un número polinómico.

El algoritmo ordena en orden creciente de r_i los procesos y mantiene una cola de prioridad con los procesos disponibles y no finalizados, utilizando como clave lo que le falta al proceso para finalizar su ejecución.

- Ordenar por r_i ;
- Insertar en la cola todos los procesos con $r_k = r_1$ (clave p_k), $i =$ primer proceso no introducido en la cola, $t = r_1$.
- mientras cola no vacía
 - $(j, p) = pop()$, si $t + p \leq r_i$ procesamos lo que queda de a_j , $t = t + p$, y repetimos hasta que la cola quede vacía o $t + p > r_i$.
 - Si $t + p > r_i$, insertamos $(j, t + p - r_i)$, $t = r_i$.
 - Insertamos en la cola todos los procesos con $r_k = r_i$ (clave p_k), $i =$ primer proceso no introducido en la cola.

La implementación es correcta ya que el conjunto de trabajos disponibles y no finalizados solo se modifican cuando hay un nuevo trabajo disponible o cuando iniciamos el procesamiento de uno de ellos. En el primer caso ese caso actualizamos la cola y el posible trabajo que se estaba ejecutando se interrumpe, y se vuelve a insertar en la cola con el tiempo restante. En el segundo, sacamos al proceso con menor tiempo para finalizar y iniciamos o reiniciamos su ejecución.

El coste de la ordenación es $O(n \log n)$ y el coste de cada inserción en la cola es $O(\log n)$. Para contabilizar el número total de inserciones, notemos que cada proceso se inserta en la cola cuando está disponible, lo que nos da n inserciones. Un proceso puede volver a reinsertarse en la cola varias veces, sin embargo, por cada tiempo de disponibilidad se reinserta un proceso como mucho, esto nos da $\leq n$ inserciones debido a paradas en la ejecución. Sumando todo, el coste del algoritmo es $O(n \log n)$.

- 2.12.  (Ordenació) Sigui A una taula que conté n claus, entre les quals com a màxim hi ha k claus diferents (no necessàriament enters), on $k \leq \lg n$. Volem ordenar la taula mantenint la posició inicial dels elements replicats. Doneu un algorisme que resolgui el problema en temps $o(n \lg n)$.

Una solució:

Usamos selección con coste lineal para localizar la mediana. El vector original se particiona respecto a la mediana x en tres partes, en un vector auxiliar para hacerlo de manera estable. Los $n_{<}$ elementos menores que x , todas las $n_{=}$ repeticiones de x (respetando su orden original) y los $n_{>}$ elementos mayores que x . Recursivamente se ordena el primer y el tercer bloque. El coste es

$$S(n, k) = \Theta(n) + S(n_{<}, k/2) + S(n_{>}, k/2)$$

cuya solución es $\mathcal{O}(n \lg k)$. Si pensamos en el árbol de recursión cada nivel contribuye $\Theta(n)$ al coste (hay que buscar la mediana y particionar cada uno de los subvectores asociados a los nodos en ese nivel y el tamaño conjunto de todos los subvectores es $\leq n$). Al bajar un nivel desde un nodo (=subvector) con k' elementos distintos tanto el subárbol izquierdo como el derecho contendrán $\leq k'/2$ elementos distintos, por lo tanto la altura del árbol de recursión es $\leq \lceil \log_2 k \rceil$ y el coste del algoritmo es $\mathcal{O}(n \lg k) = \mathcal{O}(n \lg \lg n) = o(n \lg n)$.

Una altra solució:

El algoritmo que propongo extrae en un vector auxiliar B las claves no repetidas que aparecen en la entrada junto con información adicional para poder reconstruir A ordenada.

Cada elemento de B tendrá asociada una lista en la que iremos insertando por el final los elementos de A que tienen como clave la clave almacenada en esa posición de B .

El algoritmo es una adaptación de la ordenación por inserción. Para cada elemento de A , miramos si su clave está en B o no, utilizando búsqueda dicotómica. Si está en $B[i]$ insertamos el elemento al final de la cola asociada a $B[i]$. Si no está en B , insertamos la clave en la posición de B que le corresponde y el elemento en la cola correspondiente.

Como B está ordenado por clave y las listas de cada elemento de B mantienen el orden relativo en A , si copiamos en A los elementos almacenados en las listas de B , el resultado final es el que nos piden.

El coste de construir B :

- Por cada elemento de A , una búsqueda dicotómica $\mathcal{O}(\log k)$ y una inserción en lista $\mathcal{O}(1)$.
- El coste total debido a la inserción de elementos en B es el de la ordenación por inserción $\mathcal{O}(k^2)$.
- Reconstruir A ordenada, $\mathcal{O}(n)$.

Así el coste total es $\mathcal{O}(n \lg k + k^2 + n)$. Como $k \leq \lg n$, $n \lg k \leq n \lg(\lg n) = o(n \lg n)$ y $k^2 \leq \lg^2 n = o(n \lg n)$. Por tanto, el algoritmo tiene coste $o(n \lg n)$.

Una altra solució:

Esta solución es muy similar a la anterior, pero usando un diccionario D (por ejemplo un árbol red-black, un AVL, ...) para los k elementos distintos. Cada uno tiene asociada una cola con sus apariciones en A , respetando el orden. Recorremos A para construir el diccionario D tiene coste $\mathcal{O}(n \lg k)$, pues cada uno de los elementos de A tiene que ser buscado en D e insertado como nuevo elemento si fuera una primera aparición del elemento, o bien insertarse en la cola que corresponda cuando el elemento está repetido. Luego solo hay que hacer un recorrido en inorden del diccionario, traspasando los contenidos de las colas al vector A . El coste de esta fase es $\mathcal{O}(k + \sum_i n_i) = \mathcal{O}(k + n) = \mathcal{O}(n)$,

donde n_i es el número de veces que aparece el i -ésimo elemento distinto, $1 \leq i \leq k$. El coste total es $\mathcal{O}(n \lg k + n) = \mathcal{O}(n \lg \lg n) = o(n \lg n)$. Es la misma solución que la anterior pero en vez de tener un vector ordenado de elementos distintos tenemos una estructura de datos más sofisticada. En el vector ordenado podemos hacer búsquedas dicotómicas con coste $\mathcal{O}(\lg k)$ pero la inserción ordenada tiene coste $\mathcal{O}(k)$. Al usar un árbol de búsqueda estaríamos reemplazando el término k^2 en el coste por un $k \lg k = \mathcal{O}(\lg n \lg \lg n)$ que ya está contabilizado dentro del coste $\mathcal{O}(n \lg k)$ de la primera fase. No supone ningún cambio en el coste asintótico global del algoritmo.


- 2.13. Donada una taula A amb n registres, on cada registre conté un enter de valor entre 0 i 2^n , i els continguts de la taula estan desordenats, dissenyeu un algorisme lineal per a obtenir una llista ordenada dels elements a A que tenen valor més gran que els $\log n$ elements més petits a A , i al mateix temps, tenen valor més petit que els $n - 3 \log n$ elements més grans a A .

- 2.14. Tenim un taula T amb n claus (no necessàriament numèriques) que pertanyen a un conjunt totalment ordenat. Doneu un algorisme $O(n + k \log k)$ per a ordenar els k elements a T que són els més petits d'entre els més grans que la mediana de les claus a T .

2.15. Com ordenar eficientment elements de longitud variable:

- (a) Donada una taula d'enters, on els enters emmagatzemats poden tenir diferent nombre de dígit. Però sabem que el nombre total de dígit sobre tots els enters de la matriu és n . Mostreu com ordenar la matriu en $O(n)$ passos.
- (b) Se us proporciona una sèrie de cadenes de caràcters, on les diferents cadenes poden tenir diferent nombre de caràcters. Com en al cas previ, el nombre total de caràcters sobre totes les cadenes és n . Mostreu com ordenar les cadenes en ordre alfabètic fent servir $O(n)$ passos. (Tingueu en compte que l'ordre desitjat és l'ordre alfabètic estàndard, per exemple, $a < ab < b$.)

- 2.16. Hi ha un concurs de TV amb n participants on cada participant escull un enter entre 0 i 1000000. El premi és per als dos concursants que escullen els enters més propers. Dissenyeu un algorisme que, en temps lineal, li digui al presentador quins són els dos concursants guanyadors o l'indiqui que hi ha més d'un parell de concursants candidats a rebre el premi.

- 2.17.  Donat un vector A amb n elements, és possible posar en ordre creixent els \sqrt{n} elements més petits i fer-ho en $O(n)$ passos?

Una solució: Seleccionar l'element \sqrt{n} -èsim i particionar al voltant, d'aquest element (cost $O(n)$). Ordenar la part esquerra en $O(\sqrt{n}^2)$.

Alternativament, construir un min-heap en $O(n)$ i extreure el mínim element \sqrt{n} cops, el nombre de passos és $O(n + \sqrt{n} \lg n) = O(n)$.

- 2.18. ✎ Et donen un immens graf $G = (V, E)$ amb pesos w a les arestes i has de calcular el MST. Quan finalitzes el càlcul te n'adones que has fet un error copiant el pes d'una aresta $e \in E$. Li has donat un pes $w'(e)$ i havia de ser $w(e)$. Dona un algorisme que trobi el MST correcte en temps lineal.

Una solución

Sea T el MST calculado a partir de G . Voy a analizar los 4 casos posibles, y ver que tenemos que hacer en cada caso.

- $e \in T$ y $w'(e) \leq w(e)$.

En este caso T continua siendo un MST del grafo correcto ya que su coste ha bajado el máximo posible con relación al cambio.

- $e \in T$ y $w'(e) > w(e)$.

En este caso tenemos que examinar las aristas en el corte obtenido al eliminar e de T , si hay una arista e' en el corte con $w(e') < w'(e)$, por la regla azul, tenemos que reemplazar e por e' para obtener el MST.

Recorrer las aristas de un corte implica acceder a las listas de vecinos de los vértices en un lado del corte, el coste total es $O(m)$.

- $e \notin T$ y $w'(e) \leq w(e)$.

En este caso tenemos que examinar el ciclo formado al añadir e to T , si e ahora es la arista de peso mínimo en el ciclo, de acuerdo con la regla roja, tenemos que reemplazar la arista de peso máximo en el ciclo por e .

Como en T solo hay un camino entre los dos extremos de E , tenemos que extraer esta parte del ciclo y examinarla. Lo podemos hacer en $O(n)$

- $e \notin T$ y $w'(e) > w(e)$.

En este caso T continúa siendo un MST del grafo corregido ya que e fue descartada y ahora tiene un peso mayor.

El coste total del algoritmo propuesto es $O(n + m)$.

- 2.19. ✎ (Connexions limitades) Donat un graf no dirigit ponderat $G = (V, E, w)$, i un enter k , definim G_k com el graf resultant d'esborrar tota aresta de G amb pes igual o superior a k ; és a dir, $G_k = (V, E')$ on $E' = E \setminus \{e \in E \mid w(e) \geq k\}$.

Considereu un graf connex no dirigit ponderat $G = (V, E, w)$ on cada aresta té un pes enter únic (i, per tant, totes les arestes tenen pesos diferents). Proposeu un algorisme de cost temporal $\mathcal{O}(|E| \log |E|)$ per a determinar el valor més gran de k pel qual G_k no és connex.

Una solució:

El problema se puede resolver aplicando el algoritmo de Kruskal. Kruskal inserta las aristas en orden de peso. Al no haber aristas con peso repetido, el peso k de la última arista que se agrega al MST es el valor buscado. Si todas las aristas de peso $\geq k$ se eliminan de G , entonces $G = G_k$ **no** es conexo ya que Kruskal no ha acabado antes de tratar la arista con peso k . Para cualquier peso $k' < k$ sucedería lo mismo.

El coste del algoritmo de Kruskal es $\mathcal{O}(|E| \log |E|)$, tal como se nos pide en el enunciado.

Una altra solució:

Otra posible alternativa sería la siguiente. Para un valor concreto de k , considerar el grafo G_k , y utilizar un BFS o un DFS para determinar si es o no conexo. Utilizar este algoritmo combinado con una búsqueda dicotómica. Este algoritmo resuelve el problema planteado pero tiene coste $\mathcal{O}(|E| \log_2 W)$ donde $W = \max_{e \in E} w(e)$, por lo que no lo resuelve con el coste pedido salvo en el caso en el que $W = \mathcal{O}(|E|)$. Pero si en vez de hacer la dicotomía para el buscar el valor k entre 0 y W , lo que hacemos es ordenar todas las aristas por peso (coste: $\Theta(|E| \log |E|)$) obteniendo así una lista de pesos w_1, \dots, w_m y hacemos la dicotomía para buscar el mayor peso w_i tal que G_{w_i} es inconexo entonces el coste del algoritmo es $\Theta(|E| \log |E|)$ pues el coste de cada DFS/BFS es $\Theta(|E|)$ y el número de iteraciones en la búsqueda dicotómica es $\Theta(\log |E|)$, lo que nos da un coste $\Theta(|E| \log |E|)$ globalmente.

2.20. Un *bottleneck spanning tree* T d'un graf no dirigit i ponderat $G = (V, E, w)$, on $w : E \rightarrow \mathbb{R}^+$, és un arbre d'expansió de G on el pes més gran és mínim sobre tots els arbres d'expansió de G . Diem que el valor d'un bottleneck spanning tree és el pes de la aresta de pes màxim a T .

(a) Demostreu la correctesa o trobeu un contraexemple pels enunciats següents:

- *Un bottleneck spanning tree és també un arbre d'expansió mínim.*
- *Un arbre d'expansió mínim és també un bottleneck spanning tree.*

(b) Doneu un algorisme amb cost $O(|V| + |E|)$ que donat un graf G i un enter b , determini si el valor d'un bottleneck spanning tree és $\leq b$.

- 2.21. Demostreu que un graf G té un únic MST si, per a tot tall C de G , existeix una única aresta $e \in C$ amb valor mínim. Demostreu que el el recíproc no és cert, i.e. pot ser el cas de que per un o més talls C tinguem més d'una aresta mínim pes, però que el MST sigui únic

- 2.22. Tenim un graf no dirigit i connex $G = (V, E)$ i una coloració de les arestes amb dos colors, roig i blau ($c : E \rightarrow \{R, B\}$). Doneu un algorisme per a obtenir un arbre d'expansió amb el mínim nombre d'arestes blaves.

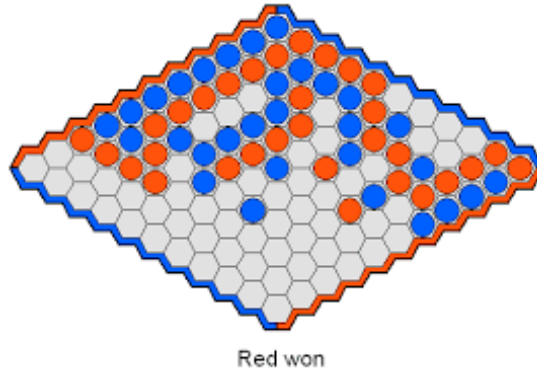
2.23. Tenim un graf connex no dirigit $G = (V, E)$ on cada node $v \in V$ té associat un valor $\ell(v) \geq 0$; considerem el següent joc unipersonal:

- (a) Els nodes inicialment no estan marcats i la puntuació del jugador és 0.
- (b) El jugador selecciona un node $u \in V$ no marcat. Sigui $M(u)$ el conjunt de veïns de u a G que ja han sigut marcats. Aleshores, s'afegeix a la puntuació del jugador el valor $\sum_{v \in M(u)} \ell(v)$ i marquem u .
- (c) El joc es repeteix fins que tots els nodes siguin marcats o el jugador decideixi finalitzar la partida, deixant possiblement alguns nodes sense marcar.

Per exemple, suposeu que el graf té tres nodes A, B, C on A està connectat a B i B amb C , amb $\ell(A) = 3$, $\ell(B) = 2$, $\ell(C) = 3$. En aquest cas, una estratègia òptima seria marcar primer A , després C i finalment B . Aquest ordre dona al jugador una puntuació total de 6.

- (a) És possible obtenir una puntuació millor deixant algun dels nodes sense marcar? Justifiqueu la vostra resposta.
- (b) Dissenyeu un algorisme voraç per tal d'obtenir la millor puntuació possible. Justifiqueu la seva correctesa i doneu-ne el cost.
- (c) Suposeu ara que $\ell(v)$ pugui ser negatiu. Continua el vostre algorisme proporcionant la puntuació màxima possible?
- (d) Considereu la següent modificació del joc per aquest cas en què els nodes poguessin tenir valors negatius: eliminem primerament de G tots els nodes $v \in V$ amb $\ell(v) < 0$, per tal de seguidament executar el vostre algorisme sobre el graf resultant d'aquesta eliminació. Doneu un exemple on aquesta variació no proporcioni la màxima puntuació possible.

- 2.24. El joc d'HEX té com un dels seus inventors, al matemàtic John Nash. En aquest joc, dos jugadors, un amb color negre i l'altre amb color blanc, fan tornos on a cada torn el jugador que li toca col·loca una pedra del seu color a una posició encara buida, a una xarxa $n \times n$ de cel·les hexagonals. Un cop col·locada una pedra, no es pot moure. L'objectiu de cada jugador és connectar els costats del mateix color a la graella, amb un camí continu fet amb les seves pedres. Dues cel·les es consideren connectades si comparteixen una vora les dues tenen la pedra amb el mateix color.



Descriviu un algorisme eficient per al seguiment d'una partida. L'algorisme, després de cada jugada, ha d'indicar si el jugador que acaba de jugar ha guanyat el joc de HEX.

- 2.25. (SOS Rural). Considerad un camino rural en el Pirineo con casas muy dispersas a lo largo de él. Por motivos de seguridad se quieren colocar estaciones SOS en algunos puntos de la carretera. Los expertos indican que, teniendo en cuenta las condiciones climáticas de la zona, se debería garantizar que cada casa se encuentre como máximo a una distancia de 15km, siguiendo la carretera, de una de las estaciones SOS.

Proporcionad un algoritmo eficiente que consiga este objetivo utilizando el mínimo número de estaciones SOS posibles. Justificad la corrección y el coste de vuestro algoritmo e indicad la complejidad en tiempo de la solución propuesta.

- 2.26. ✎ El centre de documentació de la UE gestiona el procés de traducció de documents pels membres del parlament europeu. En total han de treballar amb un conjunt de n idiomes. El centre ha de gestionar la traducció de documents escrits en un idioma a tota la resta d'idiomes.

Per fer les traduccions poden contractar traductors. Cada traductor està especialitzat en dos idiomes diferents; és a dir, cada traductor pot traduir un text en un dels dos idiomes que domina a l'altre, i viceversa. Cada traductor té un cost de contractació no negatiu (alguns poden treballar gratis).

Malauradament, el pressupost per a traduccions és massa petit per contractar un traductor per a cada parell d'idiomes. Per tal d'optimitzar la despesa, n'hi hauria prou en establir cadenes de traductors; per exemple: un traductor anglès \leftrightarrow català i un català \leftrightarrow francès, permetria traduir un text de l'anglès al francès, i del francès a l'anglès. Així, l'objectiu és contractar un conjunt de traductors que permetessin la traducció entre tots els parells dels n idiomes de la UE, amb cost total de contractació mínim.

El matemàtic del centre els hi ha suggerit que ho poden modelitzar com un problema en un graf amb pesos $G = (V, E, w)$. G té un node $v \in V$ per a cada idioma i una aresta $(u, v) \in E$ per a cada traductor (entre els idiomes u i v de la seva especialització); el pes de cada aresta seria el cost de contractació del traductor en qüestió. En aquest model, un subconjunt de traductors $S \subseteq E$ permet portar a terme la feina si al subgraf $G_s = (V, S)$ hi ha un camí entre tot parell de vèrtexs $u, v \in V$; en aquest cas direm que S és una *selecció vàlida*. Aleshores, d'entre totes les seleccions vàlides han de triar una amb cost mínim.

- (a) Demostreu que quan S és una selecció vàlida de cost mínim, $G_s = (V, S)$ no té cicles.
- (b) Proporcioneu un algorisme eficient per a resoldre el problema. Justifiqueu la seva correctesa i el seu cost.

Solució:

- (a) Supongamos que $G_s = (V, S)$ es una selección válida de coste mínimo que tiene ciclos. Si eliminamos una arista (u, v) de un ciclo en $G_s = (V, S)$ seguimos teniendo caminos entre todos los vértices, ya que podemos ir de u a v a través de lo que queda del ciclo.
Como la selección tiene coste mínimo, y eliminando una arista de un ciclo también es solución. Tenemos que todas las aristas de un ciclo tienen coste 0. Así, mientras tengamos ciclos vamos eliminando una arista de peso 0 del ciclo. Hasta que tengamos una selección válida con coste mínimo sin ciclos.
- (b) Por el aparatdo a) nos basta con buscar un árbol con peso mínimo que cubra todos los idiomas. Es decir tenemos que obtener un MST del grafo. Utilizando el algoritmo de Prim, podemos encontrarlo en tiempo $O(n \log m)$

- 2.27. Tenim un tauler de dimensions $n \times n$, amb n fitxes col·locades a certes posicions $(x_1, y_1), \dots, (x_n, y_n)$ i una fila i . Volem determinar el mínim nombre de moviments necessaris per a posar les n fitxes a la fila i (una a cada casella). Els moviments permesos són: cap a la dreta, esquerra, amunt i avall. Durant aquests moviments es poden apilar tantes fitxes a la mateixa posició com calgui. Però en finalitzar ha de quedar una fitxa per casella.

Pista: El nombre de moviments verticals (amunt/avall) necessaris es pot calcular fàcilment.

- 2.28. Sigui $T = (V, E)$ un arbre no dirigit amb n vèrtexs. Per a $u, v \in V$, sigui $d(u, v)$ el nombre d'arestes del camí de u a v en T . Definim el *centre* de T com el vèrtex

$$c = \operatorname{argmin}_{v \in V} \{ \max_{u \in V} d(u, v) \}.$$

Describiu un algorisme de cost $O(n)$ per a obtenir un centre de T .

- 2.29. CinemaVis ha de programar l'aparició d'un seguit d'anuncis a una pantalla gegant a la Plaça del Mig la diada de Sant Jordi. La tirada d'anuncis es pot iniciar a temps 0 (l'inici programat) però mai abans. A més, CinemaVis disposa d'un conjunt de n anuncis per fer la selecció. L'anunci i té una durada de 1 minut i té associat dos valors reals no negatius t_i i b_i . L'anunciat pagarà b_i euros a CinemaVis si l'anunci i s'emet a l'interval $[0, t_i]$ i 0 euros si s'emet després. Cap dels n anuncis es pot mostrar més d'una vegada. CinemaVis vol projectar la selecció d'anuncis que li proporcionï màxim benefici. Dissenyeu un algorisme, el més eficient que podeu, per a resoldre aquest problema.

2.30. (*Agenda*) A la vostra agenda teniu una llista L de totes les tasques que heu de completar en el dia de avui. Per a cada tasca $i \in L$ s'especifica la durada $d_i \in \mathbb{N}$ que indica el temps necessari per a completar-la i un factor de penalització $p_i \in \mathbb{Z}^+$ que n'agreuja el retard. Heu de determinar en quin ordre realitzar totes les tasques per obtenir el resultat que menys penalització total acumuli.

Tingueu en compte que:

- en un instant de temps només podeu realitzar una única tasca,
- una vegada comenceu a fer una tasca, heu de continuar-la fins a finalitzar-la, i
- s'han de completar totes les tasques.

El criteri d'optimització és la penalització total que s'acumula. La penalització real associada a una tasca $i \in L$ és el temps de finalització t_i de la seva realització, multiplicat per la seva penalització p_i . El temps de finalització t_i es correspon al temps transcorregut des de l'inici de la jornada laboral (és a dir, des de l'instant de temps 0) fins al moment en que s'ha finalitzat la tasca.¹

Considereu l'algorisme voraç que programa les tasques en ordre decreixent de factor de penalització p_i . Determineu si aquest algoritme resol el problema. En cas que no ho faci, proporcioneu un algorisme (tant eficient com pogueu) per resoldre'l.

¹Observeu que, segons les restriccions del problema, la realització d'una tasca $i \in L$ amb temps de finalització t_i haurà començat a l'instant $t_i - d_i$.

- 2.31. Als circuits VLSI, s'utilitza un encaminament Manhattan sobre la placa aïllant on va muntat el circuit. Les connexions horitzontals van per la cara de sota i les connexions verticals per la cara de sobre. Quan es necessiten connectar les connexions horitzontals amb les verticals, es perfora la placa amb el que s'anomena una *via*. Les connexions del circuit amb l'exterior es realitzen amb *pins* (veure la figura, on les connexions de sobre estan dibuixades en sòlid i les que van per sota amb línia discontinua). Tant les connexions horitzontals com verticals segueixen unes pistes dibuixades sobre la placa en forma d'una graella, i els pins estat alineats a un extrem de la placa. Sigui h el nombre de pistes horitzontals utilitzades. Si $L = \{(p_1, q_1), (p_2, q_2), \dots, (p_n, q_n)\}$ són una seqüència de parells de pins a connectar (dos a dos), volem dissenyar un algorisme que connecti els parells de pins utilitzant el mínim nombre de pistes horitzontals h . Per exemple, considereu la següent figura:


Tenim com a entrada $L = \{(1, 3), (2, 7), (4, 8), (5, 6)\}$, el nombre de h és 3, i no es pot fer amb $h = 2$.

- 2.32. Doneu un algorisme que resolgui el següent problema i doneu la seva complexitat en funció de n . Justifiqueu-ne la correctesa:

Volem anar de Barcelona a Paris seguint l'autopista a través de Lyon, i tenim n benzineres, $B_1 \dots, B_n$, al llarg d'aquesta ruta. Amb el diposit ple, el vostre cotxe pot funcionar K km. La benzinera B_1 és a Barcelona, i cada B_i , $2 \leq i \leq n$ és a $d_i < K$ km. de la benzinera B_{i-1} . La benzinera B_n és a Paris. Quina és l'estratègia per aturar-se el mínim nombre de cops al llarg del viatge?

- 2.33. Ja sabeu que fer la fusió ordenada de dues seqüències ordenades d' m i n elements, respectivament, comporta fer $m + n$ moviments de dades (penseu, per exemple, en un *merge* durant l'ordenació amb *mergesort* d'un vector). Però si hem de fer la fusió d' N seqüències, dos a dos, l'ordre en què es facin les fusions és rellevant. Imagineu que tenim tres seqüències A , B i C amb 30, 50 i 10 elements, respectivament. Si fusionem A amb B i després el resultat el fusionem amb C , farem $30 + 50 = 80$ moviments per a la primera fusió i $80 + 10 = 90$ per a la segona, amb un total de 170 moviments. En canvi, si fusionem primer A i C i el resultat el fusionem amb B farem un total de 130 moviments.

Dissenyau un algorisme golafre (*greedy*) per fer les fusions i obtenir la seqüència final ordenada amb mínim nombre total de moviments. Justifiqueu la seva correctesa i calculeu-ne el cost temporal del vostre algorisme en funció del nombre de seqüències N .

- 2.34.  (Mercat) A un mercat d'abastaments hi ha un producte amb infinites existències en el qual estem interessats. Ens passen una llista $P = \{p_1, \dots, p_n\}$ amb la informació sobre els preus (en euros) pels propers n dies, on $p_i > 0$ és el preu que tindrà el producte l' i -èssim dia. Per garantir un abastament equitatiu, hi ha una regla que s'ha de complir cada dia: l' i -èssim dia ningú no pot comprar més de i unitats del producte.

Per exemple, suposeu que durant els propers tres dies el preu del producte serà 7, 10 i 4 euros, respectivament. Aleshores, com a màxim podríem comprar 1 unitat el primer dia, 2 unitats el segon i 3 unitats el tercer. Amb això hauríem comprat un total de 6 unitats i hauríem gastat $7 + (2 \cdot 10) + (3 \cdot 4) = 39$ euros.

Només disposem de k euros per gastar en la compra d'aquest producte. Tenint aquesta k i la llista de preus P per als propers n dies, doneu un algorisme eficient per planificar-ne la compra durant aquests dies de manera que comprem el màxim nombre d'unitats del producte.

Una solució:

Se ordenan los precios de menor a mayor. EL volumen de compra del día i -ésimo es el máximo posible entre i y el presupuesto remanente.

procedure COMPRAS(P, k)

 Crear un *min-heap* H con P

 ▷ los elementos son $1, \dots, n$ con prioridades

 ▷ p_1, \dots, p_n

$R := k; n_{prod} := 0; c := +\infty$

while $H \neq \emptyset \wedge c > 0$ **do**

 ▷ si algún día no se puede comprar ($c = 0$) tampoco

 ▷ lo podríamos hacer en las siguientes iteraciones

 Extraer el día i de precio mínimo p_i de H

$c := \min(i, \lfloor R/p_i \rfloor)$

$n_{prod} := n_{prod} + c; R := R - c \cdot p_i;$

 ;

return n_{prod}

Sea i el día de precio mínimo en $P = \{\langle 1, p_1 \rangle, \dots, \langle n, p_n \rangle\}$. Escribimos el conjunto P de esta forma para enfatizar que hay n días y para cada día tenemos un precio y un límite del número de productos que se pueden comprar. Entonces el algoritmo *greedy* compra $c = \min(i, \lfloor k/p_i \rfloor)$ y a continuación aplica el mismo criterio para el subproblema con $P' = \{\langle 1, p_1 \rangle, \dots, \langle i-1, p_{i-1} \rangle, \langle i+1, p_{i+1} \rangle, \dots, \langle n, p_n \rangle\}$ y presupuesto $k' = k - c \cdot p_i$.

Supongamos otra solución distinta que compra **más** productos que la solución *greedy*. Esa solución comprará $c' \leq c$ productos en el día i de mínimo precio (porque no se pueden comprar más productos, por definición el *greedy* compra el máximo posible y disponiendo del presupuesto completo). Esto significa que en nuestra solución alternativa mejor que compra más que la *greedy* tenemos que comprar al menos $\Delta c > c - c'$ productos en otros días, productos que la solución voraz no compra. Para esos Δc productos se dispone como mucho de un extra $\Delta k = (c - c')p_i$ que es lo que nos hemos "ahorrado" comprando menos productos el día i . Pero comprándolos al mejor precio posible p^* necesitamos $p^* \cdot \Delta c$ euros y $p^* \cdot \Delta c > \Delta k = p_i(c - c')$ porque $\Delta c > c - c'$ y $p^* \geq p_i$ por definición. Llegamos a una contradicción y concluimos que no puede haber ninguna otra solución que compre **más** productos que la *greedy*, luego el voraz maximiza el número de productos comprados.

Su coste es $\mathcal{O}(n \log n)$; $\mathcal{O}(n)$ para crear el *heap* y $\mathcal{O}(\log n)$ en cada una de las $\leq n$ iteraciones.

Notas:

No funcionen els següents criteris alternatius d'ordenació del llistat de preus:

- Ordre creixent per ràtio p_i/i : un possible contraexemple seria la instància del problema amb $P = \{10, 12, 15\}$ i $k = 49$.
- Ordre creixent per ràtio i/p_i : un possible contraexemple seria la instància del problema amb $P = \{1, 3, 5\}$ i $k = 12$.

2.35. 🛠️ (Cap d'estació) Som els encarregats de gestionar les arribades i sortides de trens d'una nova estació que es preveu força concorreguda. Durant la nit anterior a la inauguració van arribant faxos de diferents estacions de la xarxa amb la informació referent a l'arribada dels seus trens i del temps que han de quedar-se estacionats a la nostra estació abans de continuar el viatge. Cada fax conté l'hora h d'arribada d'un tren i el nombre de minuts e que s'ha de quedar estacionat a la nostra estació. En començar el dia, recopilem tots els faxos en una llista $L = \{(h_1, e_1), \dots, (h_n, e_n)\}$ i ens disposem a organitzar l'ús de l'estació.

Doneu un algorisme eficient per a calcular quin és el mínim nombre d'andanes que necessitem habilitar a l'estació per tal que tot tren que arribi pugui estacionar, sense haver-se d'esperar que un altre tren marxi per ocupar el seu lloc.

Una solució:

Aquest problema és, en realitat, el clàssic conegut com *Interval Coloring* o *Interval Partitioning*.² En el nostre cas tenim un conjunt de n trens, i cada tren i té un instant d'arribada h_i i un instant de partida $h_i + e_i$. L'objectiu és utilitzar el nombre mínim de recursos (en el nostre cas, andanes) per programar totes les estades dels trens a les andanes de l'estació. S'ha de resoldre de manera que cap tren hagi de retardar el seu temps d'arribada perquè no hi ha cap andana lliure a l'estació (o, equivalentment, que no ens trobéssim que dos o més trens vulguin estacionar a la mateixa via al mateix temps). La Figura 1 il·lustra un exemple amb diferents planificacions.

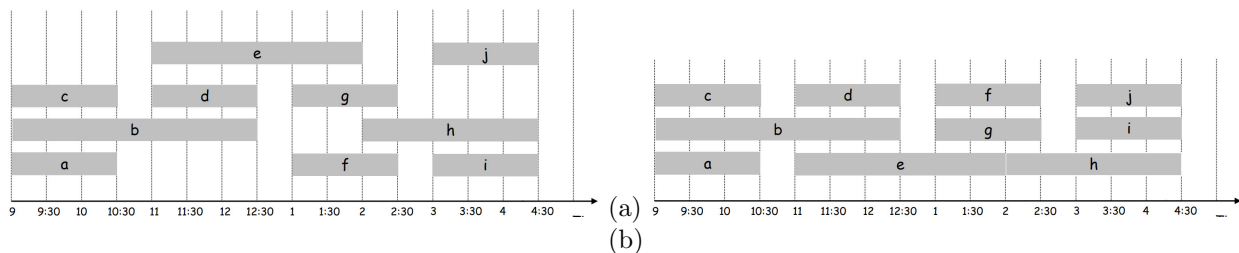


Figura 1: Diferents possibles solucions per a l'estacionament de 10 trens (etiquetats d'a a j) representats per l'interval de temps entre la seva arribada i la seva partida. La solució (a) necessita 4 andanes, mentre que la solució (b) en necessita només 3. Cada fila representa els trens que poden fer servir la mateixa andana en el seu pas per l'estació (no coincideixen).

Definim la profunditat d'un conjunt d'interval·ls com el nombre màxim d'interval·ls que coincideixen en qualsevol instant de temps. Aleshores observem que el nombre d'andanes necessàries serà almenys la profunditat del conjunt d'entrada. Per tant, qualsevol planificació dels trens que utilitzi un nombre d'andanes igual a la profunditat és, de fet, una planificació òptima perquè no podem fer-ho millor.

Podem trobar sempre una planificació òptima? La resposta és sí, i per això dissenyem un senzill algorisme *greedy* que programarà els estacionaments dels trens utilitzant un nombre d'andanes igual a la profunditat. Considerem els interval·ls d'estacionament dels trens en ordre creixent de l'hora d'inici i assignem a cada tren qualsevol andana compatible (és a dir, que estigui lliure en el moment d'arribada del tren). Si totes les andanes es troben ocupades quan intentem assignar un nou tren, aleshores habilitarem una nova andana. Mantindrem un control del nombre d'aules obertes, que serà el que retornarem com a resposta.

Pseudocodi de l'algorisme:

- 1: **function** CAP D'ESTACIÓ ($L = \{(h_1, e_1), \dots, (h_n, e_n)\}$)
- 2: Ordenar L per temps d'arribada dels trens (h_i) en ordre creixent³

²Atenció, no l'*Interval Scheduling*!!

³No importa com es resolguin els empats.


```

3:   $d \leftarrow 0$ 
4:  for  $j = 1$  to  $n$  do
5:      if tren  $j$  és compatible amb alguna andana  $k \in [1, d]$  then
6:          assignar tren  $j$  a l'andana  $k$ 
7:      else
8:          habilitar l'andana  $d + 1$ 
9:          assignar tren  $j$  a aquesta nova andana
10:      $d \leftarrow d + 1$ 
11:  return  $d$ 

```

L'assignació de trens a andanes (línies 6, 8 i 9) no és realment necessària per aquest problema, donat que l'enunciat només ens demana que calculem el nombre mínim d'andanes (d , línia 14). Ens hauríem de preocupar de com guardar aquestes assignacions si l'exercici ens demanés, a més, que detalléssim l'ocupació de les d andanes en el temps.

L'ordenació (línia 2) necessita temps $\mathcal{O}(n \log n)$. El temps d'execució total de l'algorisme dependrà de com implementem l'acció de trobar alguna andana compatible (línia 5). Si senzillament recorrem les d andanes ocupades en aquell moment per veure si alguna està lliure, el cost de l'algorisme pujarà a $\mathcal{O}(n \log n + n^2) = \mathcal{O}(n^2)$. En canvi, podem aconseguir un temps total de $\mathcal{O}(n \log n)$ si, per a cada andana k mantenim el temps en què marxa l'últim tren estacionat en ella (o, el que és el mateix, el temps en què queda lliure l'andana) i mantenim les andanes utilitzades fins aquell moment en una cua de prioritats (min-heap). Si el tren j és compatible amb alguna andana $k \in [1, d]$, ho serà segur amb la primera que queda lliure (i que és el mínim de la cua). Amb aquesta implementació, l'algorisme ens quedaria:

```

function CAP D'ESTACIÓ ( $L = \{(h_1, e_1), \dots, (h_n, e_n)\}$ )
  Ordenar  $L$  per temps d'arribada dels trens ( $h_i$ ) en ordre creixent1
  P.insert( $h_1 + e_1$ )                                ▷ S'assigna el primer tren a la primera andana i s'encua
  for  $j = 2$  to  $n$  do
     $m \leftarrow$  P.pop()                                ▷ Temps en què queda lliure la primera andana
    if ( $h.j \geq m$ ) then                                ▷ Si el tren  $j$  arriba després que quedi lliure...
      P.push( $h.j + e.j$ )                                ▷ S'assigna el tren  $t$  a l'andana i s'actualitza
    else                                                ▷ Si el tren  $t$  arriba abans que quedi lliure...
      P.push( $m$ )                                          ▷ Tornem l'andana a la cua
      P.push( $h.j + e.j$ )                                ▷ S'assigna una nova andana al tren  $j$  i s'encua
  return P.size()                                       ▷ Total d'andanes utilitzades

```

Hem de demostrar que, efectivament, es genera una planificació correcta i òptima que minimitza el nombre d'andanes. La correctesa la podem argumentar si observem que l'algorisme greedy mai programa dos trens incompatibles (dos trens que coincideixen algun temps a l'estació) a la mateixa andana, simplement per la seva definició. A més, tots els trens queden planificats.

Per demostrar l'optimitat, sigui d el nombre d'andanes que l'algorisme greedy assigna. Aleshores es va habilitat l'andana d -èsima perquè havíem d'estacionar una tren, per exemple j , que era incompatible amb tots els $d - 1$ altres trens. Donat que són incompatibles, es dedueix que aquests $d - 1$ trens marxen de l'estació després de h_j (temps d'arribada del tren j). Donat que hem ordenat per els temps d'arribada dels trens, aquests $d - 1$ trens també havien arribat a l'estació abans (o al mateix temps) de h_j . Per tant, tenim d estacionaments superposats en aquest moment.⁴ Això implica que la profunditat és almenys d i la nostra planificació és òptima.

⁴O, tècnicament, a temps $h_j + \epsilon$ per a una petita constant ϵ .

Observacions:

- Ordenar L' per temps de sortida del trens $(h_i + e_i)$ en ordre creixent no funciona. Contraexemple: $L = \{(1, 2), (2, 3), (6, 1), (4, 4)\}$.
- Mirar només la compatibilitat del tren j en tractament amb el tren anterior no és correcte.

Una solució alternativa:

Una altra idea per resoldre el problema és considerar les arribades i les sortides ordenades per separat. Un cop ordenades, es pot calcular el nombre de trens a l'estació en qualsevol moment fent un seguiment dels trens que han arribat, però que encara no han sortit. El cost asimptòtic d'aquesta solució és igual que l'anterior, $\mathcal{O}(n \log n)$.

```
function CAP D'ESTACIÓ ( $L = \{(h_1, e_1), \dots, (h_n, e_n)\}$ )  
     $A \leftarrow \{h_1, \dots, h_n\}$  ▷ Arribades  
     $S \leftarrow \{h_1 + e_1, \dots, h_n + e_n\}$  ▷ Sortides  
    Ordenar  $A$  en ordre creixent  
    Ordenar  $S$  en ordre creixent  
     $i, j \leftarrow 1$   
     $d, \text{andanes} \leftarrow 0$   
    while  $i \leq n \wedge j \leq n$  do  
        if  $A[i] \leq S[j]$  then ▷ El tren arriba abans de la darrera sortida  
             $\text{andanes} \leftarrow \text{andanes} + 1$   
             $i \leftarrow i + 1$   
        else ▷ El tren arriba després que hagi sortit l'últim tren  
             $\text{andanes} \leftarrow \text{andanes} - 1$   
             $j \leftarrow j + 1$   
         $d \leftarrow \max(d, \text{andanes})$   
    return  $d$  ▷ Total d'andanes utilitzades
```