Problemes resolts 2

2.1. **(Cobrint amd intervals)** Donat un conjunt $\{x_1, x_2, \ldots, x_n\}$ de punts de la recta real, doneu un algorisme, el més eficient que pogueu, per a determinar el conjunt més petit d'intervals 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 ($x_1 \leq x_2 \leq \cdots \leq x_n$). 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 $y_1 < y_2 < \cdots < y_k$ 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 $x_{i_1} < x_{i_2} < \cdots < x_{i_k}$ 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 cubieto por el intervalo i.

Si analizamos este último tipo de solución óptima, $x_{i_1} < x_{i_2} < \cdots < x_{i_k}$ 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. ✎**(Regal)** 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 $\mathbf{B} = (B_1, \ldots, B_n)$.

Per fer la compra hem de determinar (si és possible) una *aportació*, una llista de quantitats $X = (x_1, \ldots, x_n)$, 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 $\mathbf{x}^*$ es *equitativa* si el seu cost és mínim amb relació al conjunt de totes les possibles aportacions.

Per exemple, supossem 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 .

(a) 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 $B_{\min}$.

(b) Proporciona un algorisme golafre que determini si es pot o no comprar el regal i, en cas afirmatiu, retorni una aportació equitativa.

Una solució:

(a) 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 podriem 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, …, 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 el els n' amics poden comprar un regal de cost C'.

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

$$(B_{\min}, \ldots, B_{\min}, x_{k+1}', \ldots, x_n')$$

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 $c(\mathbf{x}') \geq \lfloor C'/n' \rfloor \geq B_{\min}$:

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

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 $c(\mathbf{x}^*) < c(\overline{\mathbf{x}})$. 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. ✎**(Interval coloring)** 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. Descriu 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:

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

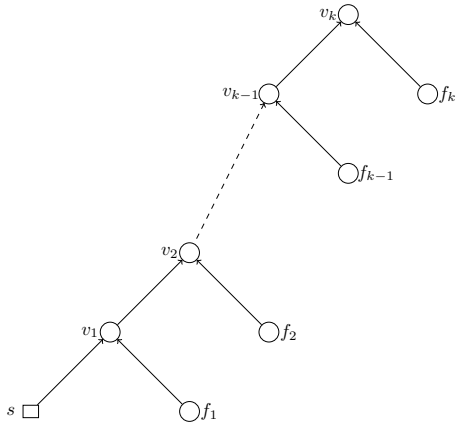
Suposem que hem determinat que s'han necessitat k^* colors per la coloració dels intervals 1 a $i-1$. Més encara, hi ha $k \leq k^*$ colors aparentment en ús (per cobrir el punt $L[i] - \epsilon$). Una estructura de dades conté la informació dels k intervals (específicament els seus extrems drets) que ténen 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 estàn 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 podriem esborrar tots els intervals del min-heap amb R més petita o igual que $L[i]$ de manera que el min-heap no contingués intervals "acabats" i que ja no necessiten tenir color assignat; però és innecesari i molt més senzill esborrar a cada iteració només un interval (o cap si no és possible).

2.4. ✎**(Afitant Huffman)** 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 prefixe corresponent a un codi Huffman òptim, i sigui s la fulla que conte a.
Sigui $P = s, w_1, w_2, \ldots, v_k$ la seqüències de nusos que van de s fins a l'arrel v_k, i siguin f_i els fills de v_i que no son 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 mes, $p(f_k) = 1$. Això implica $1.5^{k-1}p_a \leq 1$ i per tant la profunditat de a es $k + 1 = O(\lg 1/p_a)$.



2.5. ✎**(Planificació)**. Ens donen un conjunt de treballs $S = \{a_1, a_2, \ldots, a_n\}$, 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 del 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 $(3 + 8)/2 = 2.2$

(a) 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 planifique 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.

(b) 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 planifique 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.

(a) 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 $a_1, \ldots, a_n$ 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 $c_i = p_1 + \cdots + p_i$, por lo tanto

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

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.

(b) 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 vacia
  - $(j, p) = pop()$, 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 implementicón 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 insercciones. 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 $O(n \log n)$.

2.6. ✎**(Ordenació poques claus)** 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 $\mathcal{O}(n \log k)$. 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 $O(\log k)$ y una insercción en lista $O(1)$.
- El coste total debido a la inserción de elementos en B es el de la ordenación por insercción $\mathcal{O}(k^2)$.
- 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.

2.7. ✎**(Ordenant arrel quadrada més grans)** 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 $O(n + \sqrt{n} \lg n) = O(n)$.

2.8. ✎**(MST amb error)** 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 $O(n + m)$.

2.9. ✎**(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 $O(|E| \log_2 W)$ dónde $W = \max_{e \in E} w(e)$, por lo que no lo resuelve con el coste pedido salvo en el caso en el que $W = O(|E|)$. 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, $\ldots$, w_m y hacemos la dicotomia 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.

2.10. ✎**(Traducció UE)** 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 $O(n \log m)$

2.11. ✎**(Mercat)** A un mercat d'abastaments hi ha un producte amb infinites existències en el qual estem interessats. Ens passen una llista $P = \{p_1, \ldots, p_n\}$ 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 $7+(2\cdot10)+(3\cdot4) = 39$ 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,. . . , n con prioridades
    ▷ p_1,. . . , p_n
  $R := k$; $nprod := 0$; $c := +\infty$
  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
    Extrar el día i de precio mínimo p_i de H
    $c := \min(i, \lfloor R/p_i \rfloor)$
    $nprod := nprod + c$; $R := R - c * p_i$;
  ;
  return *nprod*

Sea i el día de precio mínimo en $P = \{\langle 1, p_1\rangle, \ldots, \langle n, p_n\rangle\}$. 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 $P' = \{\langle 1, p_1\rangle, \ldots, \langle i-1, p_{i-1}\rangle, \langle i+1, p_{i+1}\rangle, \ldots \langle n, p_n\rangle\}$ 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 concluímos 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 $k = 12$.

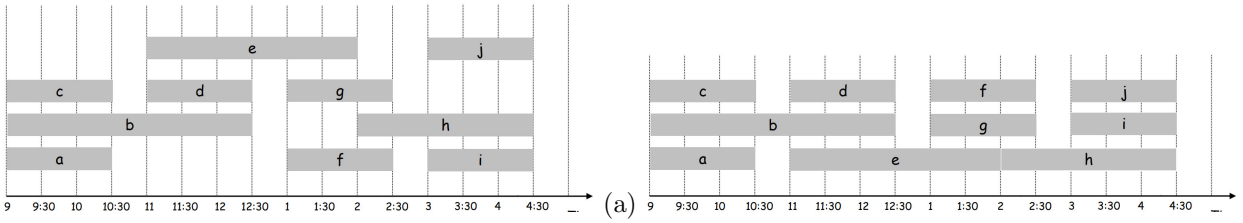


Figura 1: Diferents possibles solucions per a l'estacionament de 10 trens (etiquetats d'a a j) representats per l'interval de temps entre la seva arribada i la seva partida. La solució **(a)** necessita 4 andanes, mentre que la solució **(b)** en necessita només 3. Cada fila representa els trens que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

2.12. ✎**(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 $L = \{(h_1, e_1), \ldots, (h_n, e_n)\}$ 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*.[1] 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.

Definim la profunditat d'un conjunt d'intervals com el nombre màxim d'intervals 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 intervals 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 ($L = \{(h_1, e_1), \ldots, (h_n, e_n)\}$)

[1] Atenció, no l'*Interval Scheduling*!!

```
2:    Ordenar L per temps d'arribada dels trens (h_i) en ordre creixent[2]
3:    d ← 0
4:    for j = 1 to n do
5:        if tren j és compatible amb alguna andana k ∈ [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 ← 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), ..., (h_n, e_n)})
    Ordenar L per temps d'arribada dels trens (h_i) en ordre creixent[1]
    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 ← P.pop()              ▷ Temps en què queda lliure la primera andana
        if (h.j ≥ 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'algoritme 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'algoritme 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.[3] Això implica que la profunditat és almenys d i la nostra planificació és òptima.

[2] No importa com es resolguin els empats.

[3] O, 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 compatiblitat del tren j en tractament amb el tren anterior no és correcte.

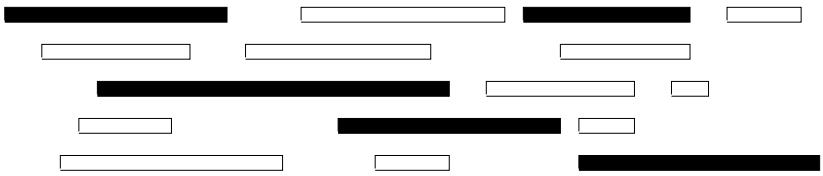
Una solució alternativa:

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 assimptòtic d'aquesta solució és igual que l'anterior, $\mathcal{O}(n \log n)$.

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

2.13. ✎ **(Camins de recobriment)** Donat un conjunt d'intervals $X = \{I_1, I_2, \ldots, I_n\}$ sobre la recta real $\mathbb{R}$, un *camí de recobriment* és un subconjunt $Y \subseteq X$ tal que tot punt $x \in \mathbb{R}$ contingut en algun interval $I_j \in X$ està contingut en algun interval d'Y. Dissenyeu un algorisme que trobi un *camí de recobriment mínim* (és a dir, amb el mínim nombre possible d'intervals) per a un conjunt d'intervals X donat. Justifiqueu la correcció del teu algorisme i calcula'n el cost en funció d'n.

Aquesta figura us mostra un exemple gràfic: els intervals emplenats en negre són un camí de recobriment mínim per al conjunt d'intervals donat (tots els rectangles).



Una solució:

El algoritmo voraz que propongo es el siguiente:

Ordenar los intervalos en orden creciente de tiempo de inicio y en caso de empate por orden decreciente de tiempo de finalización.
$S = \{I_1\}$, $\ell = 1$, $k = 1$, $j = k$
while $j \leq n$ **do**
  $y = I_k.y$, $j = k + 1$,
  while $j \leq n$ and $I_j.x \leq y$ **do**
    if $I_j.y > I_k.y$ **then**
      $k = j$,
    $j = j + 1$
  if $k \neq \ell$ **then**
    $S = S \cup \{I_k\}$, $\ell = k$, $y = I_k.y$
  else
    if $j \leq n$ **then**
      $S = S \cup \{I_j\}$, $\ell = j$, $k = j$

El algoritmo selecciona el primer intervalo y, de entre los que empiezan antes o al mismo tiempo de que este acabe, el que termina más tarde. En caso de que no haya intervalos cumpliendo esta condición y queden intervalos sin tratar, se queda con el siguiente y aplica la misma regla voraz.

Para comprobar que el algoritmo es correcto, tenemos que ver que S al final del algoritmo es una solución al problema y que contiene el número mínimo de intervalos posibles. Tenemos que considerar dos casos, el primero en que los intervalos cubren un espacio contiguo de la recta real y el segundo el caso en que esto no ocurre. Basta con demostrar la corrección en el primer caso, ya que así la tendremos para cada uno de los tramos en el segundo caso.

Veamos primero que, cuando los intervalos cubren un espacio contiguo, S al finalizar el algoritmo es una solución. Por contigüidad, siempre hay intervalos con tiempo de inicio $\leq y$ y que acaban después, salvo para el intervalo que acaba el último. Esto garantiza que entre dos intervalos añadidos a S consecutivamente se cubre todos los valores desde el inicio del primero hasta la finalización del último. Por lo que S cubre todo el espacio cubierto por la entrada.

Para demostrar que $S = \{I'_1, \ldots, I'_k\}$ es una solución óptima, consideremos una solución S^* óptima cualquiera diferente de S. Para comparar las dos soluciones, asumamos que los intervalos en $S^* = \{I^*_1, \ldots, I^*_\ell\}$ están ordenados en orden creciente de tiempo de inicio. Cómo la solución es óptima,

$\ell \leq k$, además en S^* nunca hay dos intervalos con el mismo tiempo de inicio, si no el más corto sobraría.

Sea j el primer intervalo en el que S y S^* difieren, si $j = 1$, I_1^* tiene que empezar a la vez que I_1' que es uno de los intervalos con el primer tiempo de inicio. Pero $I'1$ acaba igual o más tarde que I_1^*, de acuerdo con el criterio de ordenación. Así podemos reemplazar I_1^* con I_1' cubriendo todo el espacio que cubría I_1^*. Si $j > 1$, I_j^* tiene que empezar después de que I_{j-1}^* empiece, si no podríamos eliminar I_{j-1}^* y seguiríamos teniendo una solución con un intervalo menos. Por tanto, de acuerdo con el criterio de elección de nuestro algoritmo, I_j' acaba después o en el mismo instante que I_j^*. Así podemos reemplazar I_j^* por I_j' y seguir teniendo una solución óptima.

Repitiendo este proceso acabaremos teniendo una solución óptima que coincide con S, por lo que S es óptima.

Nota: El algoritmo simétrico que ordena por orden decreciente de tiempo de finalización y que, de entre los intervalos que acaban al mismo tiempo o después del seleccionado, añade a la solución el que empieza antes también es una solución válida.