Problemes resolts 2

2.1. **(Cobrint amb 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:

La entrada Un conjunto de valores X, voy a suponer que están están ordenados en orden creciente de valor ($x_1 < x_2 < \cdots < x_n$). Si no lo estuviesen, se pueden ordenar en tiempo $O(n \log n)$ utilizando merge sort.

La salida Una solución Y al problema se puede representar por una secuencia creceinte $y_1 \leq y_2 \leq \cdots \leq y_k$ de valores, indicando los puntos de inicio de los k intervalos de longitud 1 que forman Y.

Voy a analizar algunas propiedades de las soluciones y en particular de las soluciones óptimas.

- Puedo asumir que la secuencia de inicios de intervalos de una solución $(y_1, \ldots, y_k)$ es estrictamente creciente, ya que si no lo fuese tendríamos intervalos repetidos en nuestra solución que no son necesarios y podríamos construir una solución con menos intervalos.
- Si Y es una solución 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 de un intervalo en Y al primer punto de X que cubre este intervalo y que no está cubierto por un intervalo anterior, seguimos teniendo una solución con el mismo número de intervalos. Ya que cada intervalo cubre al menos todos los puntos que ya cubría antes y sigue siendo una solución.

Como consecuencia de las observaciones anteriores, tenemos que siempre hay una solución óptima Y en la que los puntos de inicio de los intervalos son valores en X, ningún punto de X está cubierto por más de un intervalo, y además $y_1 = x_1$.

El algoritmo Para obtener una solución son estas propiedades podemos empezar con $y_1 = x_1$ y continuar con la regla voraz: seleccionar el inicio del intervalo y_{j+1}-ésimo como el primer punto en X mayor o igual que $y_j + 1$.

Corrección Sea X, $x_{i_1} < x_{i_2} < \cdots < x_{i_k}$ con $x_{i_1} = x_1$, la solución obtenida por el algoritmo. Consideramos una solución óptima Y con $y_1 < y_2 < \cdots < y_\ell$.

Si $X \neq Y$, buscamos el primer valor i en el que las soluciones X e Y difieren. Cómo en X los intervalos no tienen puntos en común y Y es una solución, tenemos que $y_i < x_i < y_i + 1$. Entonces, avanzando el inicio del intervalo i de Y al valor x_i y dejando el resto igual, obtenemos otra solución óptima Y'

que tiene un intervalo más en común con X. Iterando el mismo argumento con la nueva solución Y' llegaremos a una solución óptima que coincide con X. Por lo que X es óptima.

Coste El algoritmo necesita tiempo $O(n)$, asumiendo que los puntos estén ordenados, y 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) Para simplificar la argumentación asumo que los presupuestos están ordenados, $B_1 \leq B_2 \leq \cdots \leq B_n$. Así $B_{\min} = B_1$. Notemos que cuando $nB_1 < C$, tenemos una solución equitativa designamos las aportaciones como $\lfloor C/n \rfloor$ o $\lceil C/n \rceil$ hasta cubrir el coste C.

Voy a construir una solución x con las condiciones requeridas, después demostraré que es equitativa.

Sea k el número de amigos con presupuesto B_1, $k < n$ ja que $n \cdot B_1 < C$. Designamos que $x_i = B_1$, para $i = 1, \ldots k$. Como $k < n$, $kB_1 < C$. Sea $C' = C - kB_1$.

Consideremos ahora el problema con los $n' = n - k$ amics amb pressupost $> B_1$. Si, $n'B_{k+1} > C'$, designamos las aportaciones x_j, $k < j \leq n$, como $\lfloor C'/n' \rfloor$ o $\lceil C'/n' \rceil$ hasta repartir lo que falta de pagar del regalo. En caso contrario, los amigos con presupuesto B_k contribuirán este presupuesto y todavía no se habrá cubierto el coste del regalo. Seguiremos completando la definición de x con los amigos con presupuesto mayor que B_{k+1} con el mismo procedimiento.

Al acabar tenemos una solución x que cumple, $x_i = B_i$, $1 \leq i \leq j$, $B_{j+1} > B_j$ y $(n - j)B_j < C - \sum_{k=1}^{j} B_k$.

Esta asignación es equitativa ya que, los primeros j jugadores no pueden pagar más, y la cantidad restante se reparte de forma equitativa .

(b) El algoritmo que propongo construye la solución equitativa descrita en el apartado anterior. Primero se ordenan los presupuestos en orden creciente. Después, con un recorrido, se obtienen las sumas prefijadas $S_i = \sum k = 1^i B_i$. Si $S_n < C$, el regalo no se puede comprar. Si $S_n \geq C$, el algoritmo busca la primera posición j que cumpla $B_j < B_{j+1}$ y $(n - j)B_{j+1} < C - S_j$. A partir de j se construye la solución. $x_i = B_i$, $1 \leq i \leq j$ y x_k será $(C - S_j)/(n - j)$ redondeado hacia arriba o hacia abajo hasta cubrir el total.

La componente más costosa del algoritmo es la ordenación, por lo que el coste es $O(n \log n)$.

Una altra 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.

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:

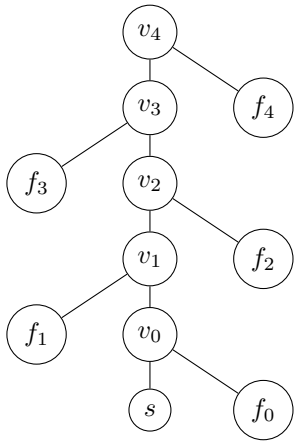
```
if (B[1]+B[2]+...+B[n] < C) {
   cout << "el regal no es pot comprar" << endl;
   return false;
else {
   ordenar els amics de menor a major pressupost
   // B[1] <= B[2] <= ... <= B[n]
   for (int i = 1; i <= n; ++i) {
       x[i] = min(B[i], C/(n-i+1));
       C = C - x[i];
   }
   return x;
}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

2.3. **(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.)

Una solució:
Fixem un símbol qualsevol a. Anomenem T a un arbre amb el codi prefix de Huffman de l'alfabet. Localitzem el full s que conté el símbol a i considerem el camí $P = s, v_0, v_1, \ldots, v_k$ format per la seqüència de nodes al camí que va de s fins a l'arrel del arbre. $k+1$ és la profunditat on apareix a. Calculem una fita superior a k. Per això analitzarem el cas pitjor.

Anomenem f_i, $1 \leq i \leq k$ a l'altre fill de v_i. Aquests nodes poden aparèixer a la dreta o a l'esquerra, depenent de l'entrad i de l'algoritme usat. Un possible arbre per a $k = 4$ és:



Tenint en compte la construcció de l'arbre sabem que:

- $p(v_0) \geq p(s) = p_a$
- Per $i \geq 1$, $p(v_i) = p(f_i) + p(v_{i-1})$
- Per $i \geq 2$, $p(f_i) \geq p(v_{i-2})$ y $p(f_i) \geq p(f_{i-1})$, si no fos així hauríem seleccionat abans f_i i no seria germà del seu pare.

De la darrera propietat tenim

$$2p(f_i) \geq p(v_{i-2}) + p(f_{i-1}) = p(v_{i-1}).$$

D'altra banda

$$p(v_i) = p(f_i) + p(v_{i-1}) \geq \frac{p(v_{i-1})}{2} + p(v_{i-1}) \geq \frac{3}{2} p(v_{i-1}).$$

Deduïm que

$$p(v_k) \geq \left(\frac{3}{2}\right)^k p(v_0) \geq (1.5)^k p_a.$$

Com que v_k és l'arrel de l'arbre $p(v_k) = 1$, tenim l'equació $(1.5)^k p_a \leq 1$. Així

$$k \log 1.5 \leq \log \frac{1}{p(a)},$$

i deduïm

$$k = O(\log \frac{1}{p(a)}).$$

2.4. **(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.

Una 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 $O(n \log n)$.

2.5. **(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 solució:
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 a T. Si e es ahora la arista de peso mínimo en el ciclo, de acuerdo con la regla roja, tenemos que reemplazar la arista de peso máximo en T 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 $O(n + m)$.

2.6. **(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.

2.7. **(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.

Una 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 $O(n \log m)$

2.8. **(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 $7 + (2 \cdot 10) + (3 \cdot 4) = 39$ 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:
Parece razonable que la regla "comprar el máximo posible cuando está más barato" nos proporcione la regla voraz para obtener una solución óptima.

Formalizando la idea, voy a demostrar que siempre hay una solución óptima en la que en el día con precio más bajo se compra el máximo de unidades posibles. Supongamos que el día j es el día con precio menor de la secuencia.

Si $k < j\,p_j$, podemos compara ℓ unidades a precio p_j para ℓ tal que $\ell p_j \leq k$ y $(\ell + 1)p_j > k$. Esta cantidad de producto es máxima posible, dado que p_j es el coste mínimo.

Si $j\,p_j < k$, en la solución óptima compramos al menos j unidades. Si en una solución óptima compramos solo α de las j unidades a precio p_j, al menos estaremos pagando $j - \alpha$ unidades a precio mayor o igual que p_j, podemos conseguir el mismo número de unidades a un precio menor igual comprándolas a precio j. Así obtendríamos una solución óptima que cumple el criterio voraz.

Para implementar la regla voraz, considero las tuplas $\langle i, p_i \rangle$, $1 \leq i \leq n$. El primer paso es ordenarlas en orden creciente de p_i. El algoritmo hace un recorrido de la secuencia ordenada comprobando para cada elemento si queda presupuesto para comprar todas las unidades del día actual, si es así las compra y actualiza el presupuesto restante. En caso contrario, compra la unidades necesarias al precio actual hasta finalizar el presupuesto y finaliza.

El coste del algoritmo es el de la ordenación $O(n \log n)$

Una altra 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 $k = 12$.

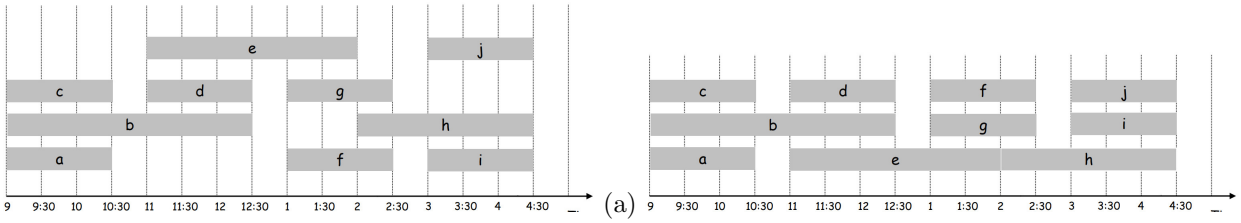


Figura 1: Diferents possibles solucions per a l'estacionament de 10 trens (etiquetats d'a a j) representats per l'interval de temps entre la seva arribada i la seva partida. La solució **(a)** necessita 4 andanes, mentre que la solució **(b)** en necessita només 3. Cada fila representa els trens que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

2.9. **(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] Atenció, no l'*Interval Scheduling*!!

```
1:  function CAP D'ESTACIÓ (L = {(h_1, e_1), ..., (h_n, e_n)})
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.

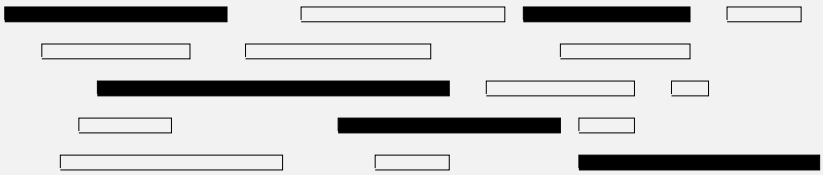
Una solució alternativa:

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*

2.10. **(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 mantiene el último intervalo en la solución para determinar cual es el siguiente. Inicialmente se selecciona el intervalo que empieza antes. Después aplica la regla voraz, seleccionar el intervalo, de entre los que empiezan antes o al mismo tiempo de que el último añadido a la solución, el que termina más tarde. En caso de que no haya intervalos cumpliendo esta condición y queden intervalos sin tratar, aplicaría de nuevo el algoritmo con los intervalos restante.

Una descripción más detallada del algoritmo 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**
      $S = S \cup \{I_j\}$, $\ell = j$, $k = j$

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.

2.11. **(Azamon es trasllada)**

> L'empresa Azamon ha decidit traslladar el contingut del seu magatzem principal a un nova localització, i per fer-ho necessitarà fer servir k camions. Per raons de seguretat cada camió ha de traslladar el mateix nombre de productes.
> Els productes están identificats per una clau que ajuda a la seva classificació posterior. Per facilitar-ne l'emmagatzematge al nou espai, s'imposa una restricció en la forma com s'ha de fer el trasllat: els productes transportats pel camió i han de tenir claus més petites (o iguals) que les claus dels productes transportats pel camió $i+1$, amb $1 \leq i < k$.
> Suposant que el nombre n de productes a transportar és un multiple de k, dissenyeu un algorisme amb cost $O(n \log k)$ tal que donat un vector no ordenat A, que conté les claus dels productes, divideixi A en k grups de manera que, si el camió i transporta el grup i, es cumpliran les condicions de seguretat i la restricció fixada pel trasllat.

Una solució:

Sigui $n = \lambda k$, de manera que cada un dels k camions utilitzats per al transport porta λ productes. Siguin $C_1, C_2, \ldots, C_n$ les claus dels n productes, amb $C_1 \leq C_2 \leq \cdots \leq C_n$. Llavors el camió #1 ha de transportar els productes 1 a λ (amb claus C_1 a C_λ, en qualsevol ordre), el camió #2 portarà els productes $\lambda + 1$ a 2λ (en qualsevol ordre), etc. En general, el camió $\#i$ transportarà els productes $(i-1) \cdot \lambda + 1$ a $i \cdot \lambda$.

El nostre algorisme aplicarà un esquema de divideix per conquerir per a resoldre el problema recursivament sobre el subconjunt de productes C_i a C_j a distribuir en q camions, on $j - i + 1 = \lambda \cdot q$. Inicialment $i = 1$, $j = n$ y $q = k$. Si $q = 2t + 1$ és senar s'aplica l'algorisme de selecció en temps lineal per a trobar el producte $(i + \lambda \cdot t)$-èssim global (és el $(\lambda \cdot t)$-èssim del subconjunt $\{C_i, \ldots, C_j\}$), amb cost $\Theta(j - i + 1)$. Després es busca el λ-èssim del subconjunt amb els productes $C_{i+\lambda \cdot t}$ a C_j, amb cost $\Theta(j - i + 1 - \lambda \cdot t))$. Si $q = 2t$ es procedeix de manera similar, obtenint el subconjunt de productes que han d'anar en el camió "central", el $(t + 1)$-èssim. Això és, amb l'algorisme de selecció en temps lineal localitzem el subconjunt d'elements que s'han de posar en el $(t+1)$-èssim camió (que és el camió $(i/\lambda + t + 1)$ en el còmput global). I després amb les corresponents dues crides recursives s'obtenen els $(q-1)/2 = t$ subconjunts de productes que han d'anar en els camions $i/\lambda + 1$ a $i/\lambda + t$ d'una banda, i els $(q-1)/2 = t$ subconjunts que van en els camions $i/\lambda + t + 2$ a $i/\lambda + 2t + 1$ per una altra banda. El raonament és completament anàleg, amb ajustos molt petits, si $q = 2t$ és parell.

L'arbre de recursió d'aquest algorisme D&C és un arbre binari complet (o gairebé: totes les "fulles", és a dir els casos en els quals $q \leq 1$, es situen en els dos últims nivells) i per tant el nombre de nivells de recursió és $\Theta(\log_2 k)$ (en la crida inicial $q := k$).Donat un cert nivell ℓ amb 2^ℓ subproblemes, la part no recursiva de cadascun d'ells és realitzar la selecció que ens dóna el "bloc central" de λ elements en un subvector de $n/2^\ell$ elements amb un cost de $\Theta(n/2^\ell)$; el cost combinat de tota la part no recursiva en el nivell ℓ és $\Theta(n)$. I el de tot l'algorisme és $\Theta(n \log k)$, ja que hi ha $\Theta(\log k)$ nivells.

2.12. **(Solitari de fitxes)**

> Una empresa de jocs està estudiant les opcions de venda del següent joc d'un jugador:
> En començar el jugador té un conjunt inicial d'n fitxes. Cada fitxa té assignat un enter no negatiu que en designa el seu nivell d'importància. El jugador només pot realitzar una operació: pot reemplaçar dues fitxes d'un mateix nivell i per una fitxa de nivell $i + 1$.
> L'objectiu del joc és anar fusionant fitxes fins que no se'n pugui obtenir cap de nova. En acabar la partida, el jugador obté tants punts com fitxes li queden al final. Com menys punts s'obtinguin, millor partida s'ha fet.
> Donat un conjunt inicial d'n fitxes, $F = \{f_1, \ldots, f_n\}$, cadascuna amb nivell $f_i \in \mathbb{Z}^+, 1 \leq i \leq n$, proporcioneu un algorisme eficient que permeti calcular quina és la millor puntuació que es pot obtenir en una partida que comenci amb la configuració F.

Una solució:

Observem que a una solució òptima mai tenim dues fitxes del mateix nivell, en cas contrari podríem reemplaçar dues fitxes per una i la solució no seria òptima. A més sempre que podem obtenir una fitxa nova aquesta és de nivell superior al que teníem abans. D'acord amb aquesta propietat l'algorisme voraç, que busca un nivell en què almenys dues fitxes i les intercanviarà per una del següent nivell resol el problema correctament.

Com l'enunciat no diu res sobre el rang dels nivells, l'algorisme que proposem, semblant a l'algorisme de Huffman, farà servir una cua de prioritat amb clau el nivell de cada fitxa. La puntuació inicial és zero. A cada pas mirarem si el mínim i el segon mínim tenen el mateix nivell. Si ho tenen introduirem a la cua de prioritat un nou element amb clau el següent nivell. En cas contrari, la fitxa de nivell mínim serà al final de la partida, per tant, incrementem la puntuació en 1, i el segon mínim passa a ser el nivell mínim per al qual tenim una fitxa.

Crear la cua de prioritat té cost $O(n \log n)$. El passos més costosos són extreure el mínim i reintroduir un valor a la cua, ambdues tenen cost $O(\log n)$. Aquestes operacions les farem com a molt $O(n)$ vegades, per tant, el cost total de l'algorisme és $O(n \log n)$.

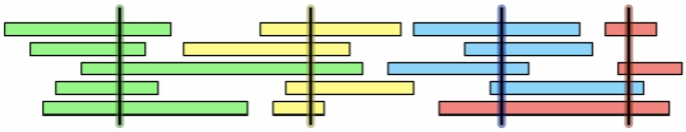


Figura 2: Exemple d'un conjunt d'intervals interceptats per 4 punts (representats com a segments verticals).

2.13. **(Interceptant intervals)**

Sigui $X = \{(a_1, b_1), \ldots, (a_n, b_n)\}$ un conjunt d'n intervals no buits. Direm que un conjunt de punts $P = \{p_1, \ldots, p_k\}$ *intercepta* un conjunt d'intervals X si tot interval a X conté com a mínim un punt de P, és a dir,

$$\forall i : 1 \leq i \leq n : (\exists j : 1 \leq j \leq k : a_i \leq p_j \leq b_i)\,.$$

(a) Descriviu i analitzeu un algorisme eficient que donats el conjunt d'intervals X i el conjunt de punts P, calculi el subconjunt més petit de P que intercepta X, o bé que determini que P **no** intercepta a X.

(b) Per a la Festa FIB d'aquest any s'han plantejat un conjunt d'activitats per tot el Campus Nord. Cada activitat té una hora i minut d'inici i finalització. Donat que es vol ver una bona difusió a xarxes d'aquell dia, els organitzadors han contractat un servei de fotografia per tal de recollir instantànies de totes les activitats. L'empresa contractada pot enviar fins a n fotògrafs per tal que fotografiïn activitats simultàniament. Donat que és un servei car, es vol minimitzar les vegades que l'equip de fotògrafs ha de desplaçar-se al campus.

Els organitzadors de la Festa FIB s'han posat en contacte amb estudiants d'Algorísmia per tal que els ajudin a decidir en quins moments requerir el servei dels fotògrafs per minimitzar-ne el cost. Descriviu com resoldríeu i implementaríeu aquest problema de la manera més eficient que se us acudeixi.

Una solució:

(a) Per proposar una solució a aquest problema ens podem inspirar en el problema de l'*Interval Scheduling* que ja coneixem (vegeu les transparències de teoria i/o el capítol 5 de *"Algorithm Design"* de Kleinberg & Tardos). L'algorisme consisteix en ordenar creixentment els intervals segons el temps de finalització i anar considerant, per cadascun d'ells, els punts que els poden anar cobrint. Primer es mirarà si el punt que cobria interval anterior també intercepta l'interval que s'està considerant (i, per tant, l'interval ja queda interceptat); si no l'intercepta aleshores s'agafarà, d'entre tots els punts que ho fan, aquell que està més a la dreta. El següent pseudocodi en fa una descripció més precisa:

Require: $X = \{(a_1, b_1), \ldots, (a_n, b_n)\}$ conjunt d'n intervals, i P conjunt de punts.
  function MINIM SUBCONJUNT INTERCEPTADOR(X, P)
    $P' \leftarrow \emptyset$
    $X' \leftarrow$ Ordenar X creixentment segons els temps de finalització dels intervals (b_i)
    while $X' \neq \emptyset$ **do**
      $x_i = (a_i, b_i) \leftarrow$ SEGÜENT(X') ▷ Següent interval a tractar
      if $(\exists p \in P'$ tal que $a_i \leq p \leq b_i)$ **then** ▷ Ja interceptat

```
        no cal fer res, ens oblidem d'x perquè ja queda interceptat
    else
        if (∄p ∈ P − P' tal que a_i ≤ p ≤ b_i) then          ▷ Cap punt l'intercepta
            return P no intercepta X
        else
            p ← argmax_{p∈P−P'} {p | a_i ≤ p ≤ b_i}          ▷ Punt que intercepta x més a la dreta
            P' ← P' ∪ {p}
    return P'
```

L'algorisme és correcte perquè si algun interval no pot ser interceptat per cap punt, aleshores retorn que P no intercepta X. Si això no passa, aleshores retorna un subconjunt $P' \subseteq P$ de punts que intercepta tots els intervals del conjunt X, i cap $x_i \in X$ queda sense interceptar.

El pas inicial d'ordenar els intervals té cost $O(n \log n)$.[4] El posterior tractament del bucle té cost $O(n)$ perquè observeu que:

- per comprovar si ja hi ha un punt a P' que intercepti x_i, n'hi ha prou amb mirar el darrer punt afegit a P', i
- tant per comprovar si hi ha més punts (dels no considerats) que interceptin x_i com per, en cas que n'hi hagi, escollir-ne el de més a la dreta, només caldrà mirar, *en total*, una vegada els k punts si aquests els tenim ordenats.

Per tant, la complexitat temporal total de l'algorisme és $O(n \log n + k \log k)$, condicionada per les ordenacions.

L'algorisme és òptim perquè el subconjunt $P' \subseteq P$ que retorna és un dels mínims (és a dir, uns dels de menor cardinalitat) d'entre tots els possibles subconjunts de P que intercepten tots els intervals d'X. Sigui $p'_1 \in P'$ el primer punt de la solució proporcionada per l'algorisme. Aquest punt p'_1 és el punt de P que intercepta el primer interval d'X' (és a dir, l'interval acaba abans que cap altre). Suposem que hi ha una solució òptima P'' que no inclou p'_1. Aleshores, el primer punt de P'' es troba abans o després de p'_1. Si es troba després, no podria interceptar el primer interval d'X' i, per tant, no seria solució. Així que ha de ser abans. En aquest cas, si el primer punt de P'' és abans que p'_1, resulta que la solució que inclou p'_1 també és una solució òptima, ja que cap altre interval acaba abans i, per tant, no es deixarien interval per interceptar. L'aplicació reiterada d'aquest argument en justifica l'optimalitat.

(b) Observem que el problema que es planteja des de la FIB és una versió del problema tractat a l'apartat a). El conjunt X d'intervals està format per les activitats que s'han plantejat per a la Festa FIB i els punts interceptadors d'aquests intervals seran els moments en què s'ha de desplegar l'equip de fotògrafs per a aconseguir instantànies de les activitats. En aquest cas, però, no tenim inicialment cap conjunt P de punts. És justament el que el nostre algorisme ha de proposar: un conjunt mínim de punts que intercepti tots els intervals de temps que defineixen les activitats.

Considereu la següent estratègia greedy: ordeneu els intervals en ordre creixent del seu temps de finalització b_i. Ara agafeu el primer interval d'aquesta llista ordenada, afegiu el seu temps de finalització al conjunt de punts interceptadors, i elimineu tots els intervals que continguin aquest punt. Repetiu aquestes darreres accions mentre quedin intervals per tractar a la llista ordenada.

El següent pseudocodi en fa una descripció més precisa:

Require: $X = \{(a_1, b_1), \ldots, (a_n, b_n)\}$ conjunt d'n intervals.

[4] N'obtindríem el mateix cost mantenint una cua de prioritat (minheap) que guardés els elements d'X segons el b_i.

function MINIM_SUBCONJUNT_INTERCEPTADOR (X)
$P' \leftarrow \emptyset$
$X' \leftarrow$ Ordenar X creixentment segons els temps de finalització dels intervals (b_i)
while $X' \neq \emptyset$ **do**
$x_i = (a_i, b_i) \leftarrow$ SEGÜENT(X') ▷ Següent interval a tractar
$P' \leftarrow P' \cup \{b_i\}$ ▷ Nou punt al final de l'interval
$X' \leftarrow X' \setminus \{x = (a, b) \mid a \leq b_i \leq b\}$ ▷ Traiem intervals interceptats pel punt
return P'

El cost temporal d'aquest algorisme és $O(n \log n)$ ja que, en considerar un nou punt, treure tots els intervals que intercepten amb ell es pot fer aprofitant el mateix recorregut sobre X (gràcies al fet que X està ordenat).

L'algorisme és correcte perquè retorna un subconjunt $P' \subseteq P$ de punts que intercepta tots els intervals del conjunt X, i cap $x_i \in X$ queda sense interceptar. Observeu que, en aquesta versió del problema, sempre existeix un subconjunt P' que cobreix X.

L'optimalitat d'aquest algorisme es dedueix de la de l'algorisme proposat a l'apartat a) ja que, el conjunt de punts solució P' que es construeix són els punts més a la dreta possibles dins dels intervals que els han definit. Nogensmenys, aquesta versió del problema es podria veure com un cas particular del presentat a l'apartat a), considerant com a conjunt inicial de punts un $P = \{b_i \mid (a_i, b_i) \in X\}$.[5]

Notes:

Els següents criteris per ordenar el conjunt d'intervals X i tractar-los no solucionen el problema de manera òptima:

- Ordenar X creixentment per a_i i agafar el primer punt que intercepta l'interval
- Ordenar X creixentment per a_i i agafar el darrer punt que intercepta l'interval
- Ordenar X decreixentment per nombre de solapaments
- Ordenar X creixentment per b_i i agafar el primer punt que intercepta l'interval
- Ordenar P decreixentment per nombre d'intervals que intercepten

És fàcil trobar contraexemples que ho demostrin.

[5] Observeu que en aplicar l'algorisme de l'apartat a) s'obtindria el mateix subconjunt $P' \subseteq P$ calculat aquí.

2.14. **(Fuster)**

> Un fuster ha d'executar n tasques per a n clients diferents. Cada tasca j, $1 \leq j \leq n$, pren temps p_j. El fuster només pot executar una única tasca alhora i, un cop ha començat amb una tasca, l'ha d'acabar abans de poder-ne començar una altra.
> Per tant, començarà a treballar a l'instant 0, executarà les tasques en algun ordre i acabarà de treballar a l'instant $P = \sum_{j=1}^{n} p_j$. Els clients estan molt interessats a que les seves tasques es facin el més aviat possible. Per això, estan disposats a pagar exponencialment més, si acaben la seva tasca abans. Concretament, si la tasca j es completa al moment C_j, aleshores el client j està disposat a pagar un preu de $w_j 2^{P-C_j}$, per alguna tarifa base acordada w_j. És a dir, per cada dia que la tasca acaba abans, el pagament es duplica. Doneu un algorisme eficient per ajudar el fuster a trobar com ordenar les tasques de manera que els pagaments totals $\sum_{j=1}^{n} w_j 2^{P-C_j}$ es maximitzin.

Una solució:
Para encontrar el orden adecuado analicemos en primer lugar el caso $n = 2$.

Si las dos tareas se ejecutan en el orden $1, 2$, $C_1 = p_1$, $C_2 = p_1 + p_2$ y $P = p_1 + p_2$. En este caso el pago total es $w_1 2^{p_2} + w_2$.

Si las dos tareas se ejecutan en el orden $2, 1$, $C_1 = p_2$, $C_2 = p_1 + p_2$ y $P = p_1 + p_2$. En este caso el pago total es $w_2 2^{p_1} + w_1$.

Para que $1, 2$ sea el orden que nos dé el máximo necesitamos que $w_1 2^{p_2} + w_2 \geq w_2 2^{p_1} + w_1$. Reorganizando, tenemos $w_1 2^{p_2} - w_1 \geq w_2 2^{p_1} - w_2$, y sacando factor común, $w_1(2^{p_2} - 1) \geq w_2(2^{p_1} - 1)$. Por tanto es necesario que $\frac{w_1}{2^{p_1}-1} \geq \frac{w_2}{2^{p_2}-1}$.

El argumento se puede generalizar a cualquier par de posiciones i e $i+1$, ya que $P - C_i = P - C_{i+1} + p_{i+1}$.

Si las dos tareas se ejecutan en el orden i, $i+1$ su contribución al pago total es $(w_i 2^{p_{i+1}} + w_{i+1}) 2^{P-C_{i+1}}$ y si se ejecutan en el orden $i+1$, i es $(w_{i+1} 2^{p_i} + w_i) 2^{P-C_{i+1}}$. En las dos ordenaciones el resto de tareas tienen el mismo tiempo de finalización. De acuerdo con el cálculo anterior, el pago total máximo se obtiene cuando $\frac{w_i}{2^{p_i}-1} \geq \frac{w_{i+1}}{2^{p_{i+1}}-1}$.

Por lo tanto el algoritmo voraz simplemente ejecuta las tareas en orden de $\varrho'_j := \frac{w_j}{2^{p_j}-1}$ decreciente.

El algoritmo es pseudopolinómico ya que aún cuando el número de comparaciones para ordenar las tareas es $O(n \log n)$ antes de ordenar tenemos que calcular los valores 2^{p_i} cuya representación necesita p_i bits, y esto es exponencial con respecto al tamaño de la entrada: (necesitamos $\Theta(p_i)$ para expresar 2^{p_i} y solamente $\Theta(\log p_i)$ bits para expresar p_i).

Observad que ordenar las tareas por w_i de mayor a menor (orden no creciente) no conduce al óptimo. Tampoco conduce al óptimo ordenar las tareas de menor a mayor tiempo de ejecución p_i. Otros criterios que no sirven: ni por $r_j = w_j/p_j$ (valor por unidad de tiempo) decreciente ni por $\varrho = w_j 2^{P-p_j}$ decreciente (equivalentemente, por $w_j/2^{p_j}$ decreciente). El que sirve, tal como hemos demostrado, es ordenar por $\varrho'_j = w_j/(2^{p_j} - 1)$ decreciente.

Esto lo ilustramos con los dos siguiente ejemplos. El primero, con $n = 3$ tareas y $P = 7$.

	1	2	3
w	12	8	3
p	4	2	1
ϱ	96	256	192
$w/2^p$	0.75	2	1.5
r	3	4	3
ϱ'	0.8	2.66	3

Criterio	Orden	Valor
w decr	123	115
p crec	321	332
ϱ decr	213	316
$w/2^p$ decr	213	316
r decr	213 o 231	316 o 283
ϱ' decr	321	**332**

El otro ejemplo es este, con $n = 3$ y $P = 13$.

	1	2	3
w	11	5	4
p	6	5	2
ϱ	1406	1280	8192
r	11/6	1	2
ϱ'	0.17...	0.16...	1.33

Criterio	Orden	Valor
w decr	123	1432
p crec	321	8523
ϱ decr	312	8549
r decr	312	8549
ϱ' decr	312	**8549**

2.15. **(MST suposat)**

Ens donen un graf ponderat $G = (V, E, w)$ juntament amb un arbre d'expansió T que és, suposadament, un MST de G. Tanmateix, hem descobert que hi ha un cert nombre d'arestes incorrectes en T, i per tant, que T no és un MST. Volem: 1) localitzar les arestes errònies de T, i 2) proporcionar un MST T' correcte de G en els casos següents:

(a) A més de G i T ens donen un subconjunt F amb k arestes a T i sabem que les arestes errònies són a F. Podeu assumir que $k = O(1)$.

(b) No tenim cap informació addicional sobre les possibles arestes errònies.

Proporcioneu algorismes tan eficients com sigui possible per a resoldre el problema plantejat en cada cas. Justifica la correctesa dels algorismes proposats i calcula el seu cost. Justifica també l'eficiència en funció de k, $|V|$ i $|E|$ en el primer cas, i de $|V|$ i $|E|$ en el segon. En qualsevol cas els vostres algorismes haurien de tenir cost $O(|E| \log |V|)$.

Una solució:

Apartat A: Conjunt d'arestes errònies $F \subset T$, $|F| = k = O(1)$

Sabem que només hi ha un nombre constant d'arestes errònies a T i que estan dins del conjunt F (de mida $k = O(1)$). Volem (1) identificar quines d'aquestes k arestes són incorrectes i (2) substituir-les per obtenir un MST correcte. Proposo el següent **algorisme**.

(a) Per a cada aresta $e \in F$, comprovem si viola la propietat del tall en MST (regla blava). És a dir, si existeix una aresta fora de T que connecta les mateixes components connexes (en treure e de T) i té pes estrictament menor.

(b) Si es detecta que e és errònia, s'elimina i es substitueix per la millor aresta possible. És a dir, la de menor pes que connecta les components que es separen en treure e.

La **correcció** de l'algorisme es basa en la propietat del tall en MSTs. És a dir, tota aresta del MST ha de ser la més lleugera en algun tall. Si trobem una aresta fora de T que redueix el pes del tall en treure e, llavors e no pot formar part d'un MST. Pel que fa al **cost**, cada comprovació (una iteració del bucle) té cost $O(|V| + |E|) = O(|E|)$ (buscar la millor aresta entre dues components). Com que $k = O(1)$, el cost total és $O(|E|)$.

Per completitud, tot i que no és necessari, es desenvolu0pa el pseudocodi de l'algorisme (veure Algorisme *CHECK_CORRECT_EDGES_IN_MST()* a les notes de sota).

Apartat B: No sabem on són les arestes incorrectes

Ara no tenim cap pista. Simplement tenim un graf G i un arbre d'expansió T que pot contenir arestes errònies. Proposo simular l'algorisme de Kruskal, que construeix un arbre d'expansió mínim (MST) triant les arestes de menor a major pes, sempre que no tanquin un cicle. No obstant això, en lloc de processar totes les arestes de G juntes, dividim (a) les arestes de T (en ordre creixent de pes) i (b) les arestes de $G - T$ (també en ordre creixent de pes). Volem processar les arestes en ordre creixent de pes, prioritzant les arestes de T per comprovar el següent.

1. Totes les arestes de T es poden incloure sense crear cicles, igual que faria Kruskal.
2. Cap aresta de $G - T$ pot millorar l'arbre, és a dir, que sempre tanqui un cicle.

A mesura que l'algorisme avança, podem trobar els següents casos.

(a) Prenem una aresta de T i no tanca cicle, aleshores està bé com en Kruskal.

(b) Prenem una aresta de T i tanca cicle, és errònia (regla vermella).

(c) Prenem una aresta de $G - T$ i sí tanca cicle, es descarta com en Kruskal.

(d) Prenem una aresta de $G - T$ i no tanca cicle, hauria d'estar al MST (aresta no inclosa).

El **cost** total de l'algorisme està dominat per l'ordenació inicial de les arestes de T i de $G - T$, que requereix $O(|E| \log |E|)$. Al bucle principal, es processen com a màxim $|E|$ arestes i cada operació d'unió o cerca a l'estructura Union-Find té un cost gairebé constant, $O(\alpha(|V|))$, on α és la inversa de la funció d'Ackermann. Per tant, el cost total de l'algorisme és $O(|E| \log |E|)$.

Per completitud, tot i que no és necessari, es desenvolupa el pseudocodi de l'algorisme (veure Algorisme *VERIFY_AND_FIX_MST()* a les notes de sota).

Nota. Per què no aplicar simplement Kruskal $O(|E| \log |V|)$ i verificar quines arestes són errònies $O(|E|)$?

Tot i que aquest enfocament sembla raonable, no és vàlid en general perquè un graf pot tenir múltiples MSTs diferents però amb el mateix cost total. Si apliquem Kruskal, obtindrem un MST òptim, però no necessàriament el mateix que T', fins i tot si T també fos un MST. Per tant, comparar directament amb el MST retornat per Kruskal podria marcar com a "errònies"arestes que en realitat sí que pertanyen a algun MST vàlid, simplement perquè no van ser seleccionades per l'execució concreta de l'algorisme.

Nota. Per què no puc enumerar tots els MSTs possibles i comparar?

En general, no és factible enumerar tots els MSTs perquè el nombre de MSTs en un graf pot ser exponencial en el nombre de vèrtexs. Per exemple, en un graf complet amb $n = |v|$ vèrtexs on totes les arestes tenen el mateix pes, el nombre de MSTs és exactament n^{n-2} (teorema de Cayley).

Algorisme *CHECK_CORRECT_EDGES_IN_MST().*

function CHECK_CORRECT_EDGES_IN_MST($G = (V, E, w), T, F$)
  for all $e = (u, v) \in F$ **do**
    ▷ Determine the 2 connected components from u and v (i.e., 2 trees)
    $(T_1, T_2) \leftarrow T - e$ ▷ Perform two $O(|V| + |E|)$ BFS (from u and then from v)
    ▷ Search for $e' = (u', v')$ such that $u' \in V(T_1)$ and $v' \in V(T_2)$
    $e' \leftarrow \arg\min\{w(e') \mid e' = (u', v') \in E - T, u' \in V(T_1), v' \in V(T_2)\}$ ▷ $O(|E|)$
    ▷ Check whether e is correct (i.e., has the smallest weight from the cut)
    if $w(e') < w(e)$ **then**
      Replace e with e' in T
    else
      Keep e in T
  return T

Algorisme *VERIFY_AND_FIX_MST().*

function VERIFY_AND_FIX_MST($G = (V, E, w), T$)
  $UF \leftarrow$ Union-Find structure over V ▷ $O(|V|)$ initialization
  $A_T \leftarrow$ stack with edges of T sorted by increasing weight ▷ $O(|E| \log |E|)$
  $A_G \leftarrow$ stack with edges in $E - T$ sorted by increasing weight ▷ $O(|E| \log |E|)$
  ▷ To store the corrected MST and erroneous edges
  $MST \leftarrow \emptyset$
  $erroneous \leftarrow \emptyset$

while $A_T \neq \emptyset$ **or** $A_G \neq \emptyset$ **do**
  if $A_T \neq \emptyset$ **and** $(A_G = \emptyset$ **or** $w(A_T.top()) \leq w(A_G.top()))$ **then**
    $e = (u, v) \leftarrow A_T.pop()$
  else
    $e = (u, v) \leftarrow A_G.pop()$
  if $e \in T$ **then**
    if $UF.find(u) = UF.find(v)$ **then** ▷ $O(\alpha(V))$
      $erroneous \leftarrow erroneous \cup \{e\}$ ▷ Incorrect (Edge should not close a cycle)
    else
      $UF.union(u, v)$ ▷ $O(\alpha(V))$
      $MST \leftarrow MST \cup \{e\}$
  else
    if $UF.find(u) \neq UF.find(v)$ **then** ▷ $O(\alpha(V))$
      $erroneous \leftarrow erroneous \cup \{e_T\}$ ▷ Incorrect (Missing edge in T)
      $UF.union(u, v)$ ▷ $O(\alpha(V))$
      $MST \leftarrow MST \cup \{e\}$
return $(MST, erroneous)$ ▷ Correct MST and list of erroneous edges

2.16. **(Mostres)**

A un laboratori de biotecnologia d'alta seguretat cada dia arriba un conjunt de mostres biològiques $M = \{M_1, \ldots, M_n\}$ de diferent origen que cal analitzar abans no es degradin. Cada mostra té una durabilitat de D_i dies, és a dir, només és analitzable qualsevol dia entre el dia de la seva arribada A_i i el dia C_i en què caduca, on $C_i = A_i + D_i$, per a tot $1 \leq i \leq n$.

El laboratori disposa d'una única màquina d'anàlisi i, per tant, només es pot analitzar una mostra al dia. A més, per motius de seguretat biològica, no es poden processar mostres en dies consecutius, ja que fer-ho podria provocar reaccions creuades al sistema.

Donat un conjunt $M = \{M_1, \ldots, M_n\}$ de mostres biològiques i les seves dates d'arribada al laboratori $A = \{A_1, \ldots, A_n\}$ i respectives durabilitats $D = \{D_1, \ldots, D_n\}$, es vol gestionar com dur a terme les anàlisis per tal de poder **maximitzar el nombre de mostres processades**, seleccionant un dia vàlid per a cadascuna de manera que:

(a) No es processi més d'una mostra el mateix dia.
(b) No es processin mostres en dies consecutius.
(c) Cada mostra s'analitzi dins del seu interval de validesa.

Proposeu un algorisme eficient que resolgui el problema. Justifiqueu-ne la correcció i analitzeu-ne el cost.

Una solució:

Formalment, volem maximitzar $|S|$, on la programació S és un subconjunt $S \subseteq \{1, \ldots, n\}$ i una assignació $f : S \to \mathbb{N}$ tal que per a tot $i \in S$, tenim

$$A_i \leq f(i) < C_i, \quad f(i) \neq f(j), \quad \text{i} \quad |f(i) - f(j)| \neq 1 \quad \text{, per a tot } i \neq j \in S.$$

Proposem un algorisme golafre. Primer, ordenem les mostres per data de caducitat creixent C_i (en cas d'empat, seleccionem primer la mostra amb A_i més primerenc). A continuació, iterem sobre la llista ordenada i, per a cada mostra M_i, intentem assignar-li el primer dia vàlid $A_i \leq d \leq C_i$ tal que d no s'hagi usat. Si existeix tal d, assignem M_i a d, marquem d com a usat.

La correcció de l'algorisme es basa en l'estratègia golafre. L'algorisme programa mostres en funció de la seva caducitat, triant sempre el dia vàlid més primerenc que no sigui consecutiu. Com que el golafre tria sempre la mostra amb caducitat més propera i el dia vàlid més aviat possible, qualsevol decisió alternativa hauria sacrificat una oportunitat amb menys marge, comprometent així el nombre total de mostres possibles. Això implica que cap altra seqüència de decisions pot portar a una solució millor, i per tant, la solució golafre és òptima.

Més formalment, suposem, per contradicció, que existeix una programació millor S^* amb $k+1$ mostres, mentre que l'algorisme golafre produeix S amb k. Sigui M_i la primera mostra (en el temps) on les programacions difereixen, amb el dia d^* assignat a M_i en S^* (on d^* no havia estat assignat en el cas de S). Com que el nostre algorisme no va triar d^*, aquest dia ja estava ocupat, era consecutiu a un dia ocupat, o estava fora de l'interval vàlid en el moment de considerar M_i. Per tant, d^* no era una opció vàlida, i S^* no pot ser una programació vàlida. Això contradiu la suposició inicial, i per tant, la solució golafre és òptima.

function PlanificaMostres(A, D, n)
  $C \leftarrow [A_i + D_i$ per a cada $i]$
  $M \leftarrow \{(A_i, C_i, i) \mid 1 \leq i \leq n\}$
  Ordena M per C_i creixent

```
DiesUsats ← ∅
S ← ∅
for all (A_i, C_i, i) ∈ M do
    for d = A_i fins a C_i − 1 do
        if d ∉ DiesUsats then
            DiesUsats ← DiesUsats ∪ {d − 1, d, d + 1}
            S ← S ∪ {(i, d)}
            break
return S
```

Respecte al cost, ordenar les n mostres per data de caducitat costa $O(n \log n)$. Per a cada mostra, podem iterar fins a D_i dies en el pitjor cas. Per tant, el cost total és $O(n \log n + n^2)$.