Algorísmia QP 2021–2022

Examen Parcial **5 d'abril de 2022**

Durada: 1h 25mn

Instruccions generals:

- Entregueu per separat les solucions de cada exercici (Ex 1, Ex 2, Ex 3 i Ex 4).
- Heu de donar i argumentar la correctesa i l'eficiència dels algorismes que proposeu. Per fer-ho podeu donar una descripció d'alt nivell de l'algorisme suficient per tal que, amb les explicacions i aclariments oportuns, justifiqueu que l'algorisme és correcte i té el cost indicat.
- Podeu fer ús d'algorismes que s'han vist a classe, però igualment cal desenvolupar-ne la justificació de la correctesa i calcular-ne o justificar-ne el cost, quan així es demani. Si la solució és una variació d'un algorisme vist a classe, n'haureu de donar també els detalls d'aquesta variació i la seva afectació sobre el cost.
- Es valorarà especialment la claredat i concisió de la presentació.
- La puntuació total d'aquest examen és de **10 punts**.

Ordenació (2 punts):

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.

Vectors oscil·lants (2.5 punts):

Diem que un vector A amb dimensió $2n+2$ és oscil·lant si $A[2i-1] \leq A[2i]$, per $1 \leq i \leq n$, i $A[2i] \geq A[2i+1]$, per $0 \leq i \leq n$.

Donat un vector B d'enters, amb dimensió $|B| = 2n+2$, descriviu un algorisme de cost temporal $\mathcal{O}(n)$ que reordeni B de forma que el vector resultant sigui oscil·lant. Justifiqueu que la complexitat temporal del vostre algorisme és lineal.

Una solució:

Localizamos, con coste $\Theta(n)$, la mediana del vector B, esto es, el elemento $(n+1)$-ésimo en orden ascendente. El algoritmo de selección nos deja además reorganizado el vector B de manera que los $n+1$ elementos menores del vector original se sitúan en las primeras $n+1$ componentes y los $n+1$ elementos mayores a continuación. En un vector auxiliar C colocamos los $n+1$ elementos menores de B en la componentes impares y los $n+1$ mayores en las pares. Finalmente se copia C sobre B. Esta parte (distribuir en C, copiar de vuelta a B) se resuelve con dos sencillos recorridos de coste $\Theta(n)$. Entonces para cualquier componente de índice par en B, digamos $k = 2i$ se cumplirá necesariamente que $B[2i-1] \leq B[2i]$ y $B[2i+1] \leq B[2i]$, porque cualquier elemento en una componente de índice par es $\geq$ que cualquier elemento en una componente de índice impar. esto es, B es oscilante, y el coste total del algoritmo es $\Theta(n)$.

Una altra solució:

Una solución elemental aún más simple consiste en recorrer el vector A considerando todas sus posiciones pares $2i$ $(0 \leq i < n)$ y realizando intercambios con $A[2i-1]$ y $A[2i+1]$ para garantizar que $A[2i-1] \leq A[2i+1] \leq A[2i]$. Para verificar que el algoritmo es correcto nos bastará demostrar que antes de la iteración k se cumple como invariante que el subvector $A[0..2k-1]$ es oscilante.

Require: $n \geq 0$
Ensure: $A[2k-1] \leq A[2k] \wedge A[2k] \geq A[2k+1]$ para toda k, $0 \leq k \leq n$; $A[-1] \equiv -\infty$

```
procedure ConvertirOscilante(A[0..2n+1])
    if A[0] < A[1] then Swap(A[0], A[1])
    end if
    k ← 1
    while k ≤ n do
        reordenar (A[2k−1], A[2k], A[2k+1])
        ▷ A[2k−1] ≤ A[2k+1] ≤ A[2k]
        k ← k+1
    end while
end procedure
```

Claramente el invariante es cierto para $k = 1$ (solo se ha de tener en cuenta $A[2k-1] = A[1] < A[2k-2] = A[0]$), antes de entrar al bucle. Si el subvector $A[0..2k-1]$ es oscilante, se consigue que $A'[0..2k+1]$ sea oscilante, ya que en el nuevo vector A' se cumple $A'[2k-1] \leq A'[2k+1] \leq A'[2k]$; pero además $A'[2k-1] \leq A[2k-1]$ y $A[0..2k-2] = A'[0..2k-2]$, de manera que $A'[0..2k-1]$ sigue siendo oscilante. Y $A'[2k-1] \leq A'[2k+1] \leq A'[2k]$ por lo tanto $A'[0..2k+1]$ es oscilante.

Al terminar el bucle con $k = n+1$ se cumple el invariante y la postcondición deseada: $A[0..2(n+1)-1] = A[0..2n+1]$ es oscilante. El coste del algoritmo es $\Theta(n)$, ya que se hacen n iteraciones con coste $\Theta(1)$ cada una.

Connexions limitades (2.5 punts):

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.

Mercat (3 punts):

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:

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
    $c := \min(i, \lfloor R/p_i \rfloor)$
    $nprod := nprod + c$; $R := R - c * p_i$;
  end while;
  return *nprod*
end procedure

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 $k = 12$.