Problemes resolts 1

1.1. **(Cim)**

> Suposem que tenim un vector A amb n nombres enters diferents, amb la propietat: existeix un únic índex p tal que els valors $A[1 \cdots p]$ estan en ordre creixent i els valors $A[p \cdots n]$ estan en ordre decreixent. Per exemple, en la següent vector tenim $n = 10$ i $p = 4$:
>
> $$A = (2, 5, 12, 17, 15, 10, 9, 4, 3, 1)$$
>
> Dissenyeu un algorisme eficient per trobar p suposant que una matriu A amb la propietat anterior ja és a memòria.

Una solució: Primer de tot formalitzarem la propietat del vector A d'entrada. Exisateix un index $p \in [1, \ldots n]$ tal que, per $1 \leq i < p$, $A[i] \leq A[i+1]$, i per $p \leq j < n$, $A[i] \geq A[i+1]$. El nostre algorisme ha de trobar aquest valor p.

Proposem l'algorisme recursiu que es descriu en l'Algorisme FindPeak. Donat el vector A, la resposta s'obté amb una crida amb $i = 1$ i $j = n$. L'algorisme fa una cerca binària, en cada pas, comparem els dos elements intermedis i veurem si estem en la part creixent o decreixent d'A. El cas base ressòl el problema d'obtenir la posició del màxim, però per a una entrada amb mida constant.

```
function FindPeak(A, i, j)
    n = j − i + 1
    if n ≤ 5 then
        return PosMax(A, i, j)
    k = (i + j)/2
    if A[k] < A[k + 1] then
        return FindPeak(A, k + 1, j)
    else
        return FindPeak(A, i, k)
```

Correctesa: Volem trobar l'índex p. Si $A[k] < A[k+1]$, sabem que $A[i] < \cdots < A[k]$ per $i < k$ i podem prescindir de forma segura els elements $A[i \cdots k]$. De la mateixa manera, si $A[k] > A[k+1]$, sabem que $A[k+1] > \cdots > A[j]$ per $j > k+1$ i podem descartar amb seguretat els elements $A[k+1 \cdots j]$. La posició de p coincideix amb la del valor màxim al vector, per tant el cas base és correcte.

Cost temporal: El cas base té cost constant. A cada pas, es redueix la mida del problema a la meitat i, a més, el cost de les operacions és constant. Així, tenim la recurrència $T(n) = T(n/2) + c$ per a alguna constant c. Sabem que, com a la cerca binària, $T(n) = O(\log n)$.

Comentari: Al fer l'anàlisi del cost de l'algorisme no hem tingut en compte el cost de llegir el vector. El cost contant el de fer la lectura del vector és $O(n)$.

1.2. **(Movent Robots)**

> Tenim un conjunt de robots que es mouen en un edifici, cadascun d'ells és equipat amb un transmissor de ràdio. El robot pot utilitzar el transmissor per comunicar-se amb una estació base. No obstant això, si els robots són massa a prop un de l'altre hi ha problemes amb la interferència entre els transmissors. Volem trobar un pla de moviment dels robots, de manera que puguin procedir al seu destí final, sense perdre mai el contacte amb l'estació base.
> Podem modelar aquest problema de la següent manera. Se'ns dóna un graf $G = (V, E)$ que representa el plànol d'un edifici, hi ha dos robots que inicialment es troben en els nodes a i b. El robot en el node a voli viatjar a la posició c, i el robot en el node b vol viatjar a la posició d. Això s'aconsegueix per mitjà d'una planificació: a cada pas de temps, el programa especifica que un dels robots es mou travessant una aresta. Al final de la planificació, els dos robots han d'estar en les seves destinacions finals.
> Una planificació és *lliure* d'interferència si no hi ha un punt de temps en el qual els dos robots ocupen nodes que es troben a distància menor de r, per a un valor determinat del paràmetre r.
> Doneu un algorisme de temps polinomial que decideixi si hi ha una planificació lliure donats, el graf, les posicions inicials i finals dels robots i el valor de r.

Una solució. Per resoldre el problema farem una reducció cap a un problema amb solució coneguda.

Per això, considerarem l'espai de configuracions on els robots es poden moure. És a dir, el conjunt de parells de posicions que estan a distància més gran o igual que r:

$$C = \{(u, v) \mid u, v \in V \text{ i } d(u, v) \geq r\}$$

Podem considerar la relació entre configuracions definida pels moviments permesos. Així tenim

$$\begin{aligned} M = \{((u, v), (u', v')) \mid & (u, v), (u', v') \in C \text{ i} \\ & ((u = u' \text{ i } (v, v') \in E) \text{ o } (v = v' \text{ i } (u, u') \in E))\}. \end{aligned}$$

A l'espai de configuracions podem considerar el graf $\mathcal{G} = (C, M)$ on dos configuracions son veïnes si i només si un del robots pot canviar de posició sense interferir amb la posició de l'altre.

Correctesa: Els robots són inicialment a la configuració (a, b) i s'han de desplaçar amb moviments vàlids fins a la configuració (c, d). D'acord amb la definició del graf de configuracions, això serà possible si i només si hi ha un camí de (a, b) a (c, d) a $\mathcal{G}$.

Algorisme: D'acord amb el raonament anterior només cal construir el graf de configuracions i comprovar si hi ha un camí entre els dos vèrtexs a $\mathcal{G}$. Podem detectar-ho amb un BFS.

Cost temporal: Per calcular el cost hem de tenir en compte la mida de l'entrada. Si $G = (V, E)$ i $n = |V|$ i $m = |E|$, tenim $|C| \leq n^2$ i $|M| \leq 2m$. Suposant que ens donen G mitjançant llistes d'adjacència la mida de l'entrada és $O(n+m)$. Construir una descripció de $\mathcal{G}$ mitjançant llistes d'adjacència té cost $O(n^2 + m)$. Fer un BFS sobre $\mathcal{G}$ té cost $O(n^2 + m)$. El cost total es $O(n^2 + m)$ però $m \leq n^2$. Llavors l'algorisme proposat té cost $O(n^2)$.

1.3. **(Distribució de notes)**

> El Professor JD ha corregit els exàmens finals del curs, de cara a tenir una distribució maca de les notes finals decideix formar k grups, cada grup amb el mateix nombre d'alumnes, i donar la mateixa nota a tots els alumnes que són al mateix grup. La condició més important és que qualsevol dels alumnes al grup i han de tenir nota d'examen superior o igual a qualsevol alumne d'un grup inferior (grups de 1 fins a $i-1$). L'ordre dintre de cadascun dels grups es irrellevant. Dissenyeu un algorisme que donada una taula A no ordenada, que a cada registre conté la identificació d'un estudiant amb la seva notes d'examen, divideix A en els k grups, amb les propietat descrita a dalt. El vostre algorisme ha de funcionar en temps $O(n \lg k)$. Al vostre anàlisis podeu suposar que n és múltiple de k i k és una potencia de 2.

Una solució: L'entrada es un vector amb les puntuacions de l'examen i la sortida ha de ser un conjunt de k grups $G_1, \ldots G_k$, cadascú format per n/k estudiants. A més l'estudiant amb nota més alta a G_i, per $1 \leq i < k$, ha de tenir nota menor o igual que la del estudiant amb nota més baixa a G_{i+1}. Com podem tenir notes repetides trencarem l'empat d'acord amb la posició que ocupa l'estudiant al vector, així podem treballar assumint que tots els valors son diferents.

Com que els grups han de tenir la mateixa mida, l'algorisme que proposo anirà calculant la mediana i dividin el vector en dos parts iguals una amb els valors mes petits o iguals que la mediana i l'altre amb la resta. AGRUPAR(N, ℓ, t) es l'algorisme recursiu que, té com a entrada una taula de alumnes-notes N i dos enters ℓ i t, i fa el següent:

- Si $\ell = 1$, torna N i t
- Troba la mediana de A i fa una partició al seu voltant.
- Considerem la sub-taula N_e esquerra i la sub-taula dreta N_d de aquesta partició.
- Cridem recursivament AGRUPAR$(N_e, \ell/2, 2t)$ i AGRUPAR$(N_d, \ell/2, 2t+1)$.

La crida inicial la farem amb N, $\ell = k$ i $t = 0$.

Correctesa: La correctesa ve de com particionem els elements. Sempre tenim dos meitats i els elements a N_e són més petits o igual que la mediana i els elements a N_d són més grans o iguals que la mediana. Aconseguirem $\ell = 1$ desprès de $\lg k$ iteracions, en aquell moment la taula considerada té n/k elements. La variable t comptabilitza l'ordre de las crides. Al primer nivell tenim només una taula i $t = 0$. Al segon tindrem dos taules, la de l'esquerra etiquetada amb 0 i la de la dreta amb 1. Al següent nivell, tindrem 0,1,2,3 (e-e,e-d,d-e,d-d). Llavors t comptabilitza l'ordre correcte de les particions per garantir la propietat requerida.

Cost temporal: El cost de l'algorisme recursiu és pot expresar amb la recurrència $T(n,k) = 2T(n/2, k/2) + \Theta(n)$ amb $T(n,1) = \Theta(1)$, per a tot n. Desplegant la recursió tenim

$$\begin{aligned} T(n,k) &= 2T(n/2,k/2) + cn = 4T(n/4,k/4) + 2c(n/2) + cn \\ &= 4T(n/4,k/4) + 2cn = k + cn \lg k. \end{aligned}$$

llavors, $T(n) = \Theta(n \lg k)$.

1.4. **(Ordenant 1)**

> 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: Si. Podem seleccionar l'element $\sqrt{n}$-èsim i particionar A al voltant, d'aquest element (amb cost $O(n)$). Desprès, ordenar la part esquerra en $O(\sqrt{n}^2) = O(n)$.

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

1.5. **(Majoritari)**

> Donat un vector $A[1..n]$, un element x s'anomena majoritari si x apareix més de $n/2$ cops a A. Donada una taula A doneu un algorisme amb cost $O(n)$ que determini si existeix un element majoritari en A i, en cas que existeixi, digui quin és l'element majoritari.

Una solución

El elemento majoritario, si lo hay, tiene que coincidir con la mediana. Podemos obtener la mediana en $O(n)$ pasos utilizando el algoritmo de selección con coste $O(n)$. Una vez obtenida la mediana, comprobamos si es o no el valor mayoritario con un recorrido del vector A. En total con coste $O(n)$.

1.6. **(Ordenant 2)**

Es poden ordenar en temps $O(n \lg \lg n)$, n enters diferents amb valors entre 1 i $n \lg n$?

Una solució: Cierto. Si tomamos base $b = n^c$ con $0 < c < 1$, p.e., $b = \sqrt{n}$, el coste de LSD radix sort es $\Theta((n+b)\log_b f(n))$ siendo $f(n) = n \lg n$. Con $b = n^c$ tenemos

$$\log_b f(n) = \frac{\lg(n \lg n)}{c \lg n} = \frac{1}{c} + O\left(\frac{\log \log n}{\log n}\right),$$

y el coste es $\Theta((n+b)\log_b f(n)) = \Theta(n) = o(n \lg \lg n)$. Puede tomarse una base $b = o(n^c)$ para toda $c > 0$ y conseguir que el coste de LSD sea $O(n \log \log n)$, pero tiene que ser $b = \omega((\log n)^k)$ para toda $k > 0$; tomar $b = n^c$ con $c \in (0,1)$ es la opción más sencilla para demostrar que la afirmación es correcta.

1.7. **(Pivots)**

> Considereu un algorisme que té com a entrada $3n$ enters diferents donats en tres vectors no ordenats de n elements i que retorna dos enters x i y tals que n dels valors donats són $\leq x$, n d'ells tenen valor entre x i y, i els altres n són $\geq y$. És cert que aquest algorisme ha de tenir complexitat $\Omega(n \lg n)$?

Una solució: Falso. Si ponemos los $3n$ enteros en un vector A (coste $\Theta(n)$) y utilizamos el algoritmo de selección de coste lineal $\Theta(n)$ para hallar los elementos $(n+1)$-ésimo y el $2n$-ésimo de A tendremos los elementos x e y pedidos y se obtienen con coste $o(n \log n)$.

1.8. **(Pes mitjà gran)**

> Ens donen un graf no dirigit i ponderat $G = (V, E, w)$ mitjançant les llistes d'adjacència dels seus vèrtexs. El pes mitjà d'un vèrtex u es defineix com la suma dels pesos de les arestes $(u, v) \in E$ dividit pel grau d'u. Es pot trobar en temps lineal un vèrtex amb pes mitjà més gran o igual que el de tres quartes parts dels vèrtexs, i més petit o igual que el de l'altre quart?

Una solució: El pes mitjà d'un vertex el podem obtenir recorrent le seva llista d'adjecència. Per tant podem obtenir els pesos mitjans de tots els vèrtexs en temps $O(n + m)$.

Desprès fem servir l'algorisme de selecció lineal per trobar l'element que ocupa la posició $3n/4$ en la llista de vèrtexs respecte els seus pesos mitjans. Això ho podem fer amb cost $O(n)$. L'algorisme té cost $O(n + m)$ i per tant és lineal en la mida de l'entrada.

1.9. **(Ordenació estable)**

> 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.

1.10. **(Ordenar k-multiconjunts)**

> Un *k-multiconjunt* és un multiconjunt amb k elements diferents, cadascun dels quals apareix exactament n/k cops. Per exemple, $\{1, 1, 2, 2, 3, 3, 4, 4, 5, 5\}$ és un 5-multiconjunt (ordenat) de mida $n = 10$. Doneu un algorisme $\Theta(n \lg k)$ per a ordenar un k-multiconjunt de mida n. Considerem que $n = 2^i$ i $k = 2^j$ per a alguns i i j, $i \geq j$.

Una solució:

Sigui S un k-multi conjunt. Considerem el següent algorisme:

- Utilitzar selecció determinista per a trobar la mediana s_m de S.
- partim S al voltant de s_m, treien els elements amb valor s_m que emmagatzemen a un vector Aux.
- Com hi ha n/k elements a S que son iguals a s_m, la partició divideix els elements restants de S en dues parts, cada part té com a molt $n/2$ elements i apareixen com a molt $k/2$ valors diferents. Siguin S_l i S_r les dues meitats,
- Aplicar recursivament l'algorisme a cada meitat (mentre aquestes tinguin grandària $\geq n/k$. Tornarem S_l ordenat, seguit de Aux i seguit de S_r ordenat.

A cada nivell de la recursió, dividim S en dos subconjunts amb grandària $\leq |S|/2$ en $O(n)$ passos. L'algorisme s'aturara quan arribe a subproblemes amb grandària n/k, es a dir $n/2^j$. Per tant l'alçada de l'arbre de recursió serà $j = \lg k$ (fins arribar a un S amb grandària n/k). A cada nivell, el cost de cada crida recursiva és lineal, i les crides es fan sobre conjunts disjunts de valors. Per tant, el cost d'un nivell és $O(n)$. Així tenim cost total $O(n \lg k)$.

Comentari: Un algorisme que ordena el multiconjunt, però no en el temps demanat.

Podem ordenar el multiconjunt donat a un vector A mantenint un vector V ordenat que contingui els valors que trobem al vector i associant a cada valor del vector V una cua de les posicions dels elements del vector d'entrada que contenen aquest valor.

L'algorisme és el següent:

- Guardem $A[0]$ a $V[0]$ i 0 a la cua associada a $V[0]$.
- per $1 \leq i < n$, cerquem $A[i]$ a V amb cerca dicotòmica. Si trobem el valor afegim i a la cua associada. En cas contrari, inserim el valor $A[i]$ en la seva posició a V i inserim i a la cua associada a la posició corresponent de V.
- Finalment, obtenim la sortida recorrent en ordre, una darrera de l'altre, les cues associades als elements d'V.

L'algorisme és correcte, ja que cada llista té posicions amb el mateix valor i es manté ordenada al llarg de tota l'execució.

Com V té sempre k o menys elements la cerca dicotòmica té cost $O(\log k)$. Per tant, el cost total de les cerques a V és $O(n \log k)$. Per un altra part, tenim el cost d'inserció, que és com a molt $O(k^2)$. L'algorisme té el cost demanat sempre que $k^2/logk = O(n)$, observem que això no passa sempre, per exemple quan $n = ck$ per alguna constant c.