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. Existeix 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, el problema de trobar un camí de un vèrtex s a un vèrtex t en un graf G. Aquest problema de camins es resol en temps lineal fent servir un BFS desde s.

La reducció: Anem a construir un graf associat a l'entrada del nostre problema. Per això, considerarem l'espai de configuracions on els dos 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, un d'ells mou cap a un altre posició. Així tenim

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

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.

Finalment, prenem com a posició inicila $s = (a, b)$ i com a posició final $t = (c, d)$.

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 $\ell = 1$ després 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. **(Amb pocs missatges?)**

> Siguin A i B dos conjunts, ambdós amb n elements, tal que els elements de A es troben a l'ordinador P i els de B al Q. P i Q es poden comunicar entre ells enviant-se missatges i poden executar localment qualsevol tipus d'operacions. Per simplificar-ho, podeu suposar que tots els elements són diferents.
> Volem un algorisme per trobar l'n-èsim element més petit d'$A \cup B$. Per fer-ne l'anàlisi temporal volem comptabilitzar només el nombre de missatges, assumint que un missatge pot contenir o bé un número entre 1 i n, o bé un element del conjunt.
> Es pot trobar l'n-èsim element més petit d'$A \cup B$ amb $o(n)$ missatges en cas pitjor? Justifiqueu-ne la vostra resposta.

Una solució curta Donats $2n$ elements en A i B, el n-èssim element més petit de $A \cup B$ és la mediana. Per obtenir un cost $o(n)$, necessitem un algoritme que redueixi la mida del problema a una fracció del mateix en cada pas. Primer, ordenem localment cada conjunt A i B (no afecta el cost de pas de missatges). Siguin m_a i m_b les medianes de A i B, wlog, si $m_a < m_b$, la mediana cercada està entre m_a i m_b. Podem descartar $n/2$ valors de cada conjunt i procedir recursivament en les meitats restants. L'algoritme acaba quan queden dos elements (el menor d'ells és la mediana). L'algoritme realitza $O(\log n)$ passos (missatges entre ordinadors), on $O(\log n) \subset o(n)$.

Una solució amb més detalls (amb un fi docent)

Per a poder obtenir cost $o(n)$ necessitem tenir un algoritme que a cada pas redueixi la mida del problema a una fracció de la mida original; és a dir, $O(\log n) \subset o(n)$. Primer, podem ordenar localment cada conjunt A i B, ja que no afecta el cost $O(\log n)$ de pas de missatges. Amb A i B ordenats, sigui $m_a = A[n/2]$ i $m_b = B[n/2]$ (la mediana de cada conjunt). Com que no hi ha elements repetits, un dels valors és més gran que l'altre. Sense pèrdua de generalitat, suposem que $m_a < m_b$. Hi ha com a mínim n elements menors que m_b i com a mínim n elements més grans que m_a, així que la mediana que busquem compleix $m_a \leq m \leq m_b$. Per això, els primers $n/2$ elements de A són menors o iguals que m_a (i que m) i els últims $n/2$ elements de B són més grans o iguals que m_b (i més grans que m). En eliminar el mateix nombre de valors més grans que menors, la mediana del conjunt de valors que resten continua sent m. A més, eliminem el mateix nombre $n/2$ de valors de cada conjunt.

```
procedure MEDIANA(a_1, a_2, b_1, b_2)
                                                    ▷ Cas base
    if a_2 − a_1 = 1 return min(A[a_1], B[b_2])
                                                    ▷ Cas general recursiu
    r ← (a_2 − a_1)/2
    (m_a, m_b) ← (A[a_1 + r], B[b_1 + r])
    if m_a < m_b then
        (a_1, b_2) ← (a_1 + r, b_2 − r)
    else
        (b_1, a_2) ← (b_1 + r, a_2 − r)
```

return MEDIANA(a_1, a_2, b_1, b_2)

L'algoritme mantindrà en tot moment els dos trossos de cada conjunt dels quals hem d'obtenir la mediana i finalitzarà quan tinguem dos trossos amb un únic element. En aquest cas base, el menor dels dos elements és la mediana. La crida **Mediana**(a1,a2,b1,b2) obtindrà la mediana de $\{A[i] \mid a1 \leq i < a2\} \cup \{B[j] \mid b1 \leq j < b2\}$. Per resoldre el problema serà suficient executar **Mediana**(0,n,0,n). Al llarg de l'algoritme mantindrem la propietat $a2 - a1 = b2 - b1$, així garantim que el tros d'A i el tros de B a tractar tenen la mateixa mida. L'algoritme realitza $O(\log n)$ passos (missatges entre ordinadors), on $O(\log n) \subset o(n)$.

1.5. **(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.