Problema 1. Anàlisi d'algorismes (1 punt)

ASES és una acadèmia que s'anuncia als aularis de la UPC tot deixant uns petits llibrets amb alguns resums de teoria i problemes resolts. Aquest quadrimestre, la publicitat per als cursos de recuperació d'ADA incloïa el text següent, copiat textualment:

> **Ejemplo de análisis del coste de un algoritmo recursivo con una recurrencia poco común:** Un ejemplo muy común que suele aparecer es la recurrencia del algoritmo que calcula el n-ésimo número de Fibonacci. La recurrencia es la siguiente:
>
> $$T(n) = \Theta(1) + T(n-1) + T(n-2).$$
>
> La manera de resolver este problema es acotando superior e inferiormente el comportamiento de esta recurrencia. Sabiendo que $T(n-1)$ tardará más que $T(n-2)$ ya que hasta llegar al caso sencillo dará más pasos, podemos decir que:
>
> $T(n) \leq 2T(n-1) + \Theta(1)$ — por teorema maestro — $\mathrm{O}(2^n)$
>
> $T(n) \geq 2T(n-2) + \Theta(1)$ — por teorema maestro — $\Omega(2^{n/2})$.
>
> Viendo las cotas superiores e inferiores podemos decir que Fibonacci se comporta: $\Theta(2^n)$.

Digueu de forma raonada què en penseu.

Problema 2. Algorismes d'ordenació (2 punts)

Volem ordenar un vector amb n elements. Per fer-ho, podríem usar *mergesort* o *quicksort*, per exemple. Recordeu que, a molt alt nivell, *mergesort* consisteix a

- ordenar amb *mergesort* la primera meitat,
- ordenar amb *mergesort* la segona meitat,
- fusionar les dues meitats.

Per altra banda, *quicksort* consisteix a

- particionar el vector,
- ordenar amb *quicksort* la part esquerra,
- ordenar amb *quicksort* la part dreta.

Recordeu també que el cost de *mergesort* és sempre $\Theta(n \log n)$ perquè el cost de fusionar les dues meitats és $\Theta(n)$. Per altra banda, el cost de *quicksort* en el cas pitjor és $\Theta(n^2)$, i es dóna quan la partició (feta en temps $\Theta(n)$) deixa un element a un costat i els demés a l'altra. A més, el cost en el cas millor de *quicksort* és $\Theta(n \log n)$, i es dóna quan la partició deixa la banda esquerra de la mateixa mida que la banda dreta.

Considerem dos nous algorismes d'ordenació, anomenats A i B. L'algorisme A consisteix a:

- ordenar amb B la primera meitat,
- ordenar amb B la segona meitat,
- fusionar les dues meitats.

L'algorisme B consisteix a:

- particionar el vector,
- ordenar amb A la part esquerra,
- ordenar amb A la part dreta.

Calculeu els costos en el cas pitjor, millor i mitjà dels algorismes A i B.

Problema 3. Estructures de dades (3 punts)

Sens dubte ja sabeu que el passat diumenge un equip de la UPC es va classificar per participar a les finals mundials del concurs de programació de l'ACM. Un dels membres d'aquest equip és en Tomàs Lloret, un fiber que apareix amb el cabell pintat de groc a l'aventura gràfica "Sobreviu a la FIB". El problema següent s'inspira en un dels problemes que els semifinalistes del 3r Concurs de Programació de la UPC van haver de resoldre.

Donada una taula $T[0 \,..\, n-1]$ amb n enters i un índex i (amb $0 \leq i < n$), la i-èsima suma parcial de T és $\sum_{0 \leq j \leq i} T[j]$. Considereu un TAD que permeti calcular sumes parcials de taules. En particular, les operacions considerades són:

- `create(`n`)`: Crea una taula $T[0 \,..\, n-1]$ amb tots els elements inicialitzats a 0.
- `set(`i`,`x`)`: Assigna l'enter x a la posició i de la taula (amb $0 \leq i < n$).
- `get(`i`)`: Retorna el valor de la posició i de la taula (amb $0 \leq i < n$).
- `sum(`i`)`: Retorna la i-èsima suma parcial de la taula (amb $0 \leq i < n$).

Descriviu (amb detall però sense arribar a donar codi ni pseudo-codi) una estructura de dades i els algorismes necessaris per implementar aquest TAD fent que les operacions `set` i `sum` tinguin cost $\mathrm{O}(\log n)$, que `get` tingui cost constant i que `create` tingui cost $\mathrm{O}(n)$.

Dibuixeu l'estat de la vostra estructura de dades quan es té una taula amb els valors $[12, 41, 31, 12, -10, 13, -21, 12]$.

Pista: Per simplicitat, podeu suposar que n és una potència de 2. Penseu en arbres binaris complets. No us calen punters.

Problema 4. Arbres binaris de cerca (2 punts)

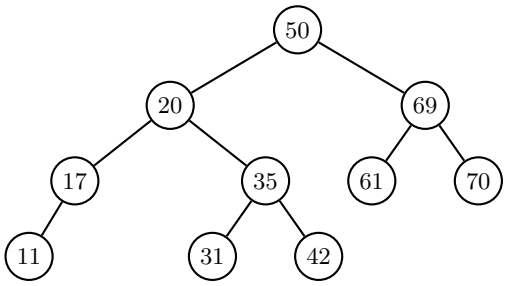
Considereu un arbre binari de cerca amb les definicions següents:

```
struct node {
    elem x;             // Informació
    node* fe;           // Punter al fill esquerre
    node* fd;           // Punter al fill dret
};

typedef node* abc;      // Un arbre binari de cerca ve donat pel punter a la seva arrel.
```

Implementeu i analitzeu una funció `node* següent (abc a, elem x)` que, donat un arbre binari de cerca `a` i un element `x`, retorni un punter al node que conté el mínim element d'entre els que són més gran que `x` en `a`. Si `x` és més gran o igual que l'element màxim de l'arbre, retorneu un punter nul.

Per exemple, per a l'arbre de la figura, el següent de 31 hauria de ser un punter a 35, el següent de 32 també hauria de ser un punter a 35, el següent de 69 hauria de ser un punter a 70, i el següent de 70 hauria de ser un punter nul.



Fixeu-vos que `x` pot no ésser a l'arbre o que l'arbre pot ésser buit.

Problema 5. Grafs i algorisme d'Strassen (2 punts)

Sigui $G = (V, E)$ un graf no dirigit amb $V = \{1, \ldots, n\}$. A partir de G, definim M com la seva matriu d'adjacència numèrica donada per $M[u, v] = 1$ quan $\{u, v\} \in E$ i $M[u, v] = 0$ quan $\{u, v\} \notin E$.

La matriu $M^2 = M * M$ representa el producte matricial de M per ella mateixa, és a dir, $M^2[u, v] = \sum_{w=1}^{n} M[u, w]M[w, v]$.

Un triangle en un graf és un conjunt de tres vèrtexs u, v i w, tots diferents, tals que $\{u, v\}$, $\{u, w\}$, $\{v, w\}$ són arestes de G.

a) Siguin u i v dos vèrtexs qualssevol de V. Descriviu amb paraules el valor de $M^2[u, v]$.

b) Tot utilitzant l'algorisme de Strassen de multiplicació de matrius en temps $O(n^{2.81})$ com a caixa negra, descriviu un algorisme del mateix cost per determinar si un graf donat per la seva matriu d'adjacència conté algun triangle.

Pista: Considereu la matriu $M^3 = M * M * M$ i examineu la seva diagonal.

Solució1

Tota l'anàlisi és correcta, excepte el darrer pas: De $T(n) = \mathrm{O}(2^n)$ i $T(n) = \Omega(2^{n/2})$ no es pot deduir que $T(n) = \Theta(2^n)$ perquè aquestes dues funcions no són del mateix ordre: $\lim 2^n/2^{n/2} = \lim 2^{n/2} = \infty$.

[De fet, el cost de l'algorisme recursiu per calcular l'n-èsim nombre de Fibonacci és ϕ^n on $\phi = \frac{1}{2}(1+\sqrt{5}) \approx 1.62$ és el nombre auri.]

[També es podria argumentar que és ben curiosa la diferenciació entre la recurrència poc comú i l'exemple molt comú. Igualment, el senyor Fibonacci va morir fa molts anys, per tant no és gaire adequat dir que es comporta com una funció.]

Solució 2

Siguin $A(n)$ i $B(n)$ el cost en el cas pitjor per ordenar un vector d'n posicions amb l'algorisme A i l'algorisme B respectivament. Llavors $A(n) = 2B(n/2) + \Theta(n)$ i $B(n) = A(n-1) + \Theta(n)$. Desplegant B en A, trobem $A(n) = 2A(n/2-1) + \Theta(n)$ i aplicant el teorema mestre obtenim $A(n) = \Theta(n \log n)$. Desplegant A en B obtenim $B(n) = \Theta(n \log n)$. Per tant, el cost en el cas pitjor de A i de B és $\Theta(n \log n)$.

Siguin ara $P(n)$ i $Q(n)$ el cost en el cas millor per ordenar un vector d'n posicions amb l'algorisme A i l'algorisme B respectivament. Llavors $P(n) = 2Q(n/2) + \Theta(n)$ i $Q(n) = 2P(n/2) + \Theta(n)$. Desplegant Q en P, trobem $P(n) = 4P(n/4) + \Theta(n)$ i aplicant el teorema mestre obtenim $P(n) = \Theta(n \log n)$. Desplegant P en Q obtenim $Q(n) = \Theta(n \log n)$. Per tant, el cost en el cas millor de A i de B és $\Theta(n \log n)$.

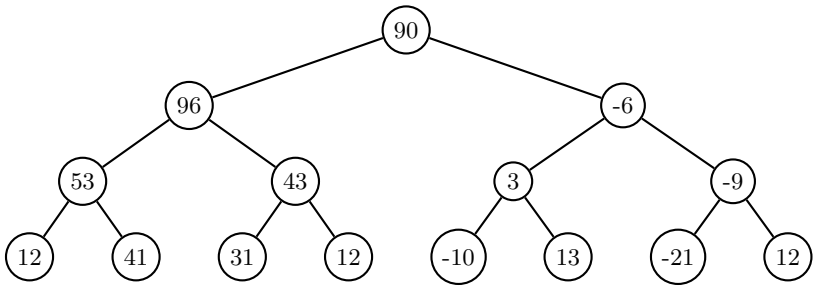
Com que el cost de A i de B en el cas pitjor i en el cas millor coincideix, el cost de A i de B en el cas mitjà també ho ha de fer i són, doncs, $\Theta(n \log n)$.

[Errors freqüents: No saber aplicar els teoremes mestres. Confondre les diferents classes de costos. No saber escriure una recurrència. Utilitzar las clases de costos como funcions i no coma a classes. No especificar costos (i recurrències) en funció de la talla de l'entrada.]

Solució 3

Organitzem les dades de la taula com a fulles d'un arbre binari complet amb n fulles. Els nodes interns de l'arbre contenen la suma dels elements de les fulles que tenen per sota.

Per a la taula $[12, 41, 31, 12, -10, 13, -21, 12]$, tindríem l'arbre següent:



L'arbre el podem enmagatzemar en una taula amb $2n - 1$ posicions per nivells (igual que un *heap*). [L'arrel es troba a la posició 1, el fill esquerre d'un element a la posició i es troba a la posició $2i$ (si no era una fulla), el fill dret d'un element a la posició i es troba a la posició $2i + 1$ (si no era una fulla), el pare d'un element a la posició i es troba a la posició $\lceil i/2 \rceil$ (si no era l'arrel). L'element i de la taula es troba a la posició $n + i - 1$.]
Les operacions s'implementarien com segueix:

- `create`: Crea la taula, tota plena de zeros. El cost serà $\Theta(n)$, tal com cal.
- `get`(i): Retorna directament l'element i (que es troba a la posició $n + i - 1$ a la taula de l'arbre). El cost serà $\Theta(1)$, tal com cal.
- `set`(i, x): Es calcula primer la diferència (positiva o negativa) entre el nou valor i l'antic. Després, aquesta diferència s'afegeix a tots els nodes en el camí de la fulla pertinent fins a l'arrel, per tal de restablir el fet que cada subarbre contingui la suma dels elements a les seves fulles. El cost serà proporcional a l'alçada de l'arbre, és a dir $\Theta(\log n)$, tal com cal.
- `sum`(i): Es recorre el camí de la i-èsima fulla fins a l'arrel, mantenint un comptador inicialitzat al valor de la fulla. Cada cop que es puja per un fill dret, s'afegeix al contador el valor del germà esquerre, per comptabilitzar la suma del subarbre esquerre. El cost serà proporcional a l'alçada de l'arbre, és a dir $\Theta(\log n)$, tal com cal.

[Si n no fós una potència de dos, afegiríem fulles fictícies amb zeros, fins a obtenir la següent potència de dos.]

[Errors freqüents: No seguir la pista. Creure que totes les operacions per arbres binaris tenen cost $\mathrm{O}(n \log n)$. Confondre arbres binaris generals amb arbres binaris de cerca.]

Solució 4

```
node* seguent (abc a, elem x) {
    if (not a) return a;
    if (x >= a->x) return seguent(a->dre, x);
    node *p = seguent(a->esq, x);
    return p ? p : a;
}
```

El cost és proporcional a l'alçada de l'arbre.

Solució 5

a) El contingut de $M^2[u][v]$ és el nombre de veins comuns de u i v.

[Cada w contribueix 1 a la suma $\sum_{w=1}^{n} M[u,w]M[w,v]$ exactament quan $\{u,w\} \in E$ i $\{v,w\} \in E$.]

b) Fixem-nos que $M^3[u,u] = \sum_{i=1}^{n} M^2[u,i]M[i,u]$. Demostrem que G conté un triangle si i només si existeix un u tal que $M^3[u,u] > 0$.

- $\Rightarrow$ Si u, v i w formen un triangle a G, aleshores u i w tenen algún veí comú (v, per exemple), i són veins l'un de l'altre. Per tant, $M^2[u,w] > 0$ i $M[w,u] > 0$. Això implica que $M^3[u,u] > 0$ perquè $M^2[u,w]M[w,u] > 0$ i cap $M^2[u,i]M[i,u]$ és negatiu.
- $\Leftarrow$ Si $M^3[u,u] > 0$, aleshores existeix un w tal que $M^2[u,w]M[w,u] > 0$ i per tant $M^2[u,w] > 0$ i $M[w,u] > 0$. Això vol dir que u i w tenen algún veí comú, i que w i u són veins entre ells i per tant diferents. Per tant u, w i qualsevol dels seus veins comuns formen un triangle a G.

Així, l'algorisme consisteix en calcular M^3 utilitzant Strassen dos cops i mirar si a la seva diagonal hi ha algun valor no nul. El cost és doncs $\Theta(n^{2.81}) + \Theta(n^{2.81}) + \Theta(n) = \Theta(n^{2.81})$.