

Cognoms

Nom

DNI

Examen Final EDA

Duració: 3h

08/01/2021

-
- L'enunciat té 6 fulls, 12 cares, i 4 problemes.
 - Poseu el vostre nom complet i número de DNI a cada problema.
 - Contesteu tots els problemes en el propi full de l'enunciat a l'espai reservat.
 - Llevat que es digui el contrari, sempre que parlem de cost ens referim a cost asimptòtic en temps.
 - Llevat que es digui el contrari, **cal justificar les respostes.**
-

Problema 1

(2.5 pts.)

Donat un vector v d' n naturals volem determinar si existeix un element *dominant*, és a dir, si existeix un element que apareix més de $n/2$ vegades. Per exemple:

- Si $v = \{5, 2, 5, 2, 8, 2, 2\}$, aleshores 2 és l'element dominant perquè apareix $4 > 7/2$ vegades.
- Si $v = \{3, 2, 3, 3, 2, 3\}$, aleshores 3 és l'element dominant perquè apareix $4 > 6/2$ vegades.
- Si $v = \{6, 1, 6, 1, 6, 9\}$, no hi ha cap element dominant perquè cap d'ells apareix més de $6/2$ vegades.

Volem obtenir una funció en C++ que rebí el vector v i retorni l'element dominant de v , o el nombre -1 en cas que no existeixi cap element dominant.

- (a) (1.25 pts.) Un antic estudiant d'EDA llegeix el problema i se'n adona que, si s'utilitzen diccionaris, es pot aconseguir una solució molt neta i prou eficient:

```
int majority_map (const vector<int>& v) {  
    map<int,int> M;  
    int n = v.size ();  
    for (auto& x : v) {  
        ++M[x];  
        if (M[x] > n/2) return x;  
    }  
    return -1;  
}
```

Si assumim que el *map* s'implementa com un AVL, analitzeu el seu cost en cas pitjor en funció de n .

Si ara assumim que no hi ha cap element dominant, que enlloc d'un *map* utilitzem un *unordered_map*, i que aquest s'implementa com una taula de dispersió, quin és el cost del codi en cas mitjà en funció de n ?

- (b) (1.25 pts.) Un estudiant brillant ens proporciona una solució basada en dividir i vèncer.

```
int times (const vector<int>& v, int l, int r, int x) {
    if (l > r) return 0;
    return (v[l] == x) + times(v, l+1, r, x); }

int majority_pairs (const vector<int>& v, int tie_breaker ) {
    if (v.size () == 0) return tie_breaker ;
    else {
        int n = v.size ();
        if (n % 2 == 1) tie_breaker = v.back ();
        vector<int> aux;
        for (int i = 0; i < n - 1; i+=2)
            if (v[i] == v[i+1]) aux.push_back(v[i]);
        int cand = majority_pairs (aux, tie_breaker );
        if (cand == -1) return -1;
        int n_times = times(v, 0, n-1, cand);
        if (n_times > n/2 or (2*n_times == n and cand == tie_breaker )) return cand;
        else return -1;
    } }
```

Cognoms

Nom

DNI

```
int majority_pairs (const vector<int>& v) {  
    return majority_pairs (v,-1);  
}
```

Analitzeu el cost en cas pitjor, en funció de n , d'una crida $majority_pairs(v)$.

Aquesta cara estaria en blanc intencionadament si no fos per aquesta nota.

Cognoms**Nom****DNI****Problema 2****(2 pts.)**

Tal i com passa a les pitjors pel·lícules de sobretaula, la mort d'una tieta llunyana ens deixa una enorme herència d' M milions d'euros. Com a bons nous rics, ens disposem a gastar tot el dineral que ens ha tocat el més ràpid possible. Per a fer-ho obrim el web d'un portal immobiliari i fem una llista de tots els habitatges que ens interessin, amb el seu corresponent preu en milions d'euros.

Finalment, abans d'abandonar per sempre el món de la informàtica decidim escriure un programa que ens indiqui totes les maneres de comprar un subconjunt dels habitatges que ens interessin per **exactament** M milions d'euros.

Per exemple, si $M = 10$ i guardem tots els preus dels habitatges en un vector $p = [4, 2, 6, 2, 3]$, aleshores el programa ha d'escriure per pantalla $\{0, 2\}$ i $\{1, 2, 3\}$. És a dir, les solucions contenen els índexos en el vector p dels habitatges que hem de comprar.

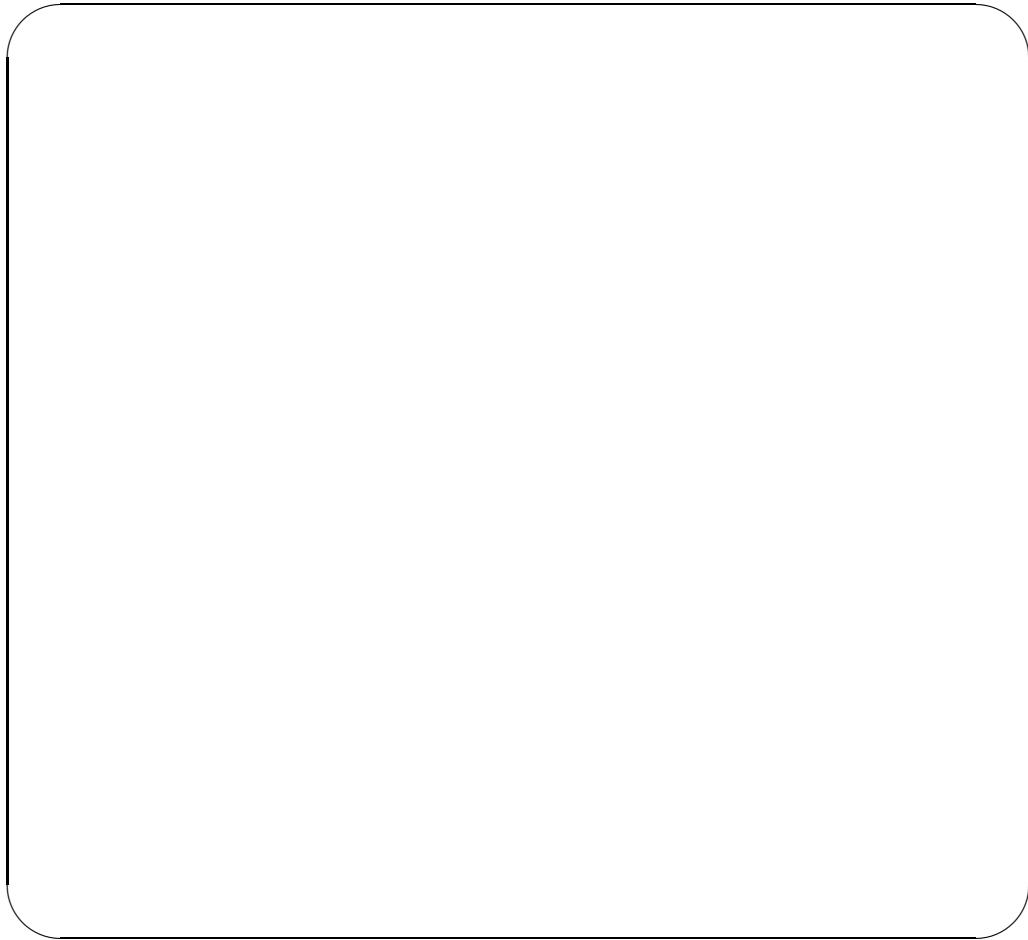
(a) (1 pt.) Completa el següent codi perquè resolgui aquest problema:

```
vector<int> p; // prices of the properties
int money;    // total money we have to spend

void write_choices (vector<int>& partial_sol, int partial_sum, int idx) {
    if (  >  ) return;
    if (  ) {
        if (partial_sum == money) {
            cout << "{";
            for (int i = 0; i < partial_sol.size(); ++i)
                cout << (i == 0 ? "" : ",") << partial_sol[i];
            cout << "}" << endl;
        }
    }
    else {
        
        write_choices ( partial_sol ,  ,  );
        
        write_choices ( partial_sol ,  ,  );
    }
}

int main() {
    int n;
    cin >> money >> n;
    p = vector<int>(n);
    for (auto &x : p) cin >> x;
    vector<int> partial_sol ;
    write_choices ( partial_sol , 0, 0); }
```

- (b) (1 pt.) Després d'executar-lo sobre llistes d'habitatges de mida mitjana, ens n'adonem que el programa és massa lent. Expliqueu com implementaríeu algun mecanisme de poda addicional per millorar el comportament del programa. No cal que escriviu codi, però sí que heu de donar prou detalls perquè a partir de la vostra descripció es pugui implementar la poda fàcilment.



Cognoms

Nom

DNI

--	--	--

Problema 3

(2.5 pts.)

Considereu els dos problemes decisionals següents:

COLORABILITAT: donats

- un graf G amb vèrtexs V i arestes E ,
- i un natural k ,

volem saber si existeix una funció $c : V \rightarrow \{1, 2, \dots, k\}$ de manera que per a tota aresta $\{u, v\} \in E$ tenim que $c(u) \neq c(v)$.

DISTINCT-ONES: donats

- un conjunt N de naturals (potser amb elements repetits),
- i un natural p ,

volem saber si podem distribuir els naturals de N en p conjunts (és a dir, cada nombre ha d'anar a parar a exactament un conjunt), de manera que si dos nombres van a parar al mateix conjunt, no poden tenir cap 1 en la mateixa posició de la seva representació en binari. És a dir, $8 = 1000_2$ i $5 = 0101_2$ poden anar al mateix conjunt, però $3 = 011_2$ i $6 = 110_2$ no poden anar junts perquè el segon bit dels dos és 1.

(a) (0.5 pts.) Considereu la instància del problema DISTINCT-ONES:

- $N = \{3, 6, 8, 20, 22\}$
- $p = 3$

És una instància positiva? Justifiqueu la vostra resposta.

(b) (1 pt.) Demostreu que DISTINCT-ONES \in NP.

(c) (1 pt.) Demostreu que la següent reducció de COLORABILITAT a DISTINCT-ONES és una reducció polinòmica correcta:

Donada una instància (G, k) de COLORABILITAT, on G té vèrtexs $V = \{v_1, v_2, \dots, v_n\}$ i arestes $E = \{e_0, e_1, \dots, e_{m-1}\}$, construïm la següent instància (N, p) de DISTINCT-ONES:

- $N = \{x_1, x_2, \dots, x_n\}$, on tots els x_i tenen m bits i el k -èssim bit de x_i és 1 si i només si $v_i \in e_k$. Cada x_i , doncs, està associat a un vèrtex v_i de G .
- $p = k$.

Cognoms

Nom

DNI

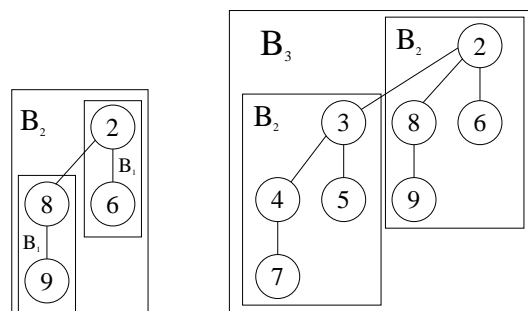
Problema 4

(3 pts.)

Definim B_k , un *arbre binomial d'ordre k* de la manera següent:

- B_0 és un arbre format per un únic node.
- Per a tot $k > 0$, l'arbre B_k resulta de prendre un arbre B_{k-1} amb arrel r i afegir-hi, com a fill de més a l'esquerra de r , un altre arbre B_{k-1} .

En la següent imatge podeu veure com es formen els arbres B_2 i B_3 , respectivament.

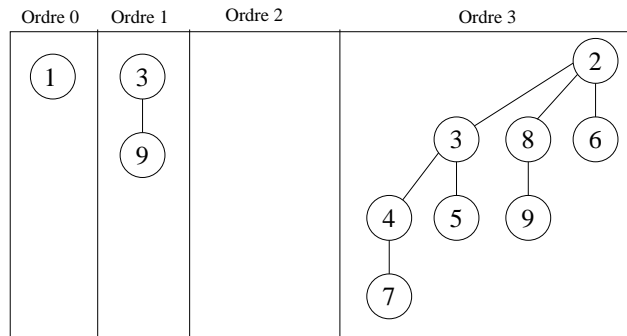


(a) (0.5 pts.) Quants nodes té un arbre B_k ? Demostra-ho formalment.

Un *heap binomial* és un conjunt d'arbres binomials que satisfan les següents propietats:

- Cada arbre binomial satisfà la propietat de min-heap: la clau d'un node és major o igual que la clau del seu pare.
- Per a cada $k \geq 0$, hi ha com a molt un arbre binomial d'ordre k .

A la següent figura podeu veure un heap binomial amb 11 nodes:



- (b) (0.75 pts.) Volem construir un heap binomial amb n nodes. Quants arbres binomials de cada ordre tindrà? Exemple: si $n = 11$ tindrem un arbre binomial d'ordre 0, un d'ordre 1 i un d'ordre 3 (fixeu-vos en la figura superior).

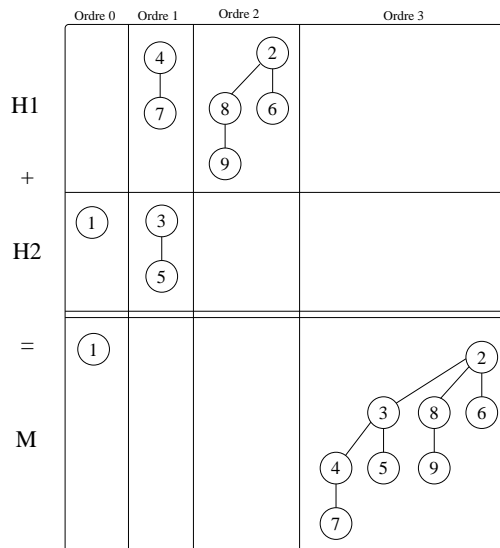
- (c) (0.75 pts.) Donats dos arbres binomials A i B d'ordre k que satisfan la propietat de min-heap, com els podem combinar en temps constant per tal de formar un arbre binomial d'ordre $k + 1$ que satisfaci la propietat de min-heap i que contingui tots els nodes d' A i B ?

Cognoms

Nom

DNI

- (d) (1 pt.) Una de les particularitats dels heaps binomials és que podem fusionar dos heaps binomials que tinguin $\Theta(n)$ nodes en temps $\Theta(\log n)$. L'algorisme s'assembla molt al de la suma en binari: processarem els arbres de menor a major ordre. Per a cada k , fusionarem els arbres d'ordre k , considerant que podem tenir un "carry" d'ordre k de la suma anterior. Teniu un exemple a la següent figura:



Ompliu el següent codi per a fusionar dos heaps binomials que tenen claus enteres:

```

class BinomialHeap {
    class Node {
    public:
        int key;
        vector<Node*> children;
        Node(int k, vector<Node*> c): key(k), children(c){}
    };
    typedef Node* Tree;
    vector<Tree> roots; // roots[k] is the binomial tree of order k, NULL if none
    BinomialHeap(vector<Tree>& r): roots(r) {}

    // Given t1, t2 binomial trees of order k that satisfy the min-heap property,
    // returns a binomial tree of order k+1 satisfying the min-heap property that
    // contains all elements of t1 and t2. You can use this function in your code.
    Tree mergeTreesEqualOrder(Tree t1, Tree t2);
    void merge(BinomialHeap& h);
public:
    BinomialHeap(){}
    void push(int k);
    void pop();
    int top();
};

```

```

void BinomialHeap::merge ( BinomialHeap& h ){
    // Make sure vector roots has same size in both heaps (makes code simpler)
    while (h.roots.size () < roots.size ()) h.roots.push_back(NULL);
    while (h.roots.size () > roots.size ()) roots.push_back(NULL);
    vector<Tree> newRoots(roots.size ());
    Tree carry = NULL;
    for (int k = 0; k < roots.size (); ++k) {
        if ( roots[k] == NULL and h.roots[k] == NULL) {
            newRoots[k] =  ;
            carry =  ; }
        else if ( roots[k] == NULL) {
            if ( carry == NULL) newRoots[k] =  ;
            else {
                newRoots[k] =  ;
                carry = mergeTreesEqualOrder(  ,  ); }
        }
        else if ( h.roots[k] == NULL) {
            if ( carry == NULL) newRoots[k] =  ;
            else {
                newRoots[k] =  ;
                carry = mergeTreesEqualOrder(  ,  ); }
        }
        else {
            newRoots[k] =  ;
            carry = mergeTreesEqualOrder(  ,  ); }
    }

    if ( carry != NULL) newRoots.push_back(  );
    roots = newRoots;
}

```

Raoneu per què, si els dos heaps tenen $\Theta(n)$ elements, aquesta funció triga temps $\Theta(\log n)$.