

Examen Final EDA

Duració: 3h

10/01/2022

Problema 1

(2.5 pts.)

Per simplificar l'anàlisi, en tot aquest problema podeu assumir que una crida al mètode *insert* de la classe *unordered_set* té sempre cost $\Theta(1)$.

- (a) (1 punt) Recordem que la classe *string* en C++ té un mètode *substr(int i, int l)* tal que donat un *string* *s*, la crida *s.substr(i, l)* retorna l'*string* que comença a la posició *i* i té llargada *l*. Assumirem que el cost d'una crida a aquest mètode és $\Theta(l)$. Per exemple, si *s* és "paraula", aleshores *s.substr(1,4)* retorna *arau*. Considereu el codi següent:

```
void mystery(const string& s, unordered_set<string>& res){
    res.insert(s);
    if (s.size() > 1) { // call to size has cost Theta(1)
        mystery(s.substr(1, s.size()-1), res);
        mystery(s.substr(0, s.size()-1), res);
    }
}

int main(){
    string s; cin >> s;
    unordered_set<string> res;
    mystery(s, res);
    for (string x : res) cout << x << endl;
}
```

Si l'*string* que es llegeix al *main* és "pep", quants *strings* s'escriuran per la sortida estàndard? Quins són aquests *strings*? No cal raonar la resposta.

Si dins el *main* es llegeix un *string* de mida *n*, quin cost té la crida a *mystery*?

(b) (1 punt) Considerem ara una nova funció *mystery*:

```
void mystery(const string& s, unordered_set <string>& res) {  
    for (int i = 0; i < s.size (); ++i)  
        for (int j = i; j < s.size (); ++j)  
            res . insert (s . substr (i ,j-i+1));  
}
```

Si dins el *main* es llegeix un *string* de mida n , quin cost té ara la crida a *mystery*?

(c) (0.5 pts.) Ompliu el següent codi perquè calculi el mateix que l'apartat anterior:

```
void mystery(const string& s, unordered_set <string>& res){  
    for (int k = 1; k ≤ s.size (); ++k)  
        for (  )  
            res . insert (s . substr (i ,k ));  
}
```

Problema 2**(2.5 pts.)**

Després de molts anys, el professorat d'EDA decideix renovar el funcionament del Joc. A partir d'ara, les partides seran entre 3 jugadors. El resultat de cada partida és una tripleta (j_1, j_2, j_3) que indica que el jugador j_1 ha guanyat la partida, j_2 ha estat el segon, i j_3 ha estat el pitjor jugador. Enlloc de les clàssiques rondes, on a cada ronda s'eliminava un jugador, ara es realitzaran un munt de partides entre tripletes de jugadors i en guardarem els resultats. Finalment, per determinar la nota del joc volem establir un rànquing de jugadors, és a dir, una llista ordenada de jugadors on els millors jugadors haurien de sortir a les primeres posicions. Per tal que cap estudiant se senti agreujat, volem garantir que per tot parell de jugadors j_1 i j_2 , si j_1 ha quedat en millor posició que j_2 en alguna partida, aleshores j_1 ha d'aparèixer abans que j_2 al rànquing.

a) (0.75 pts.) Ompliu les caselles del codi següent per trobar un rànquing correcte.

```

struct Match {
    int first ; int second; int third ;
    Match(int f, int s, int t): first (f), second(s), third (t){}
};
int n;
vector<Match> matches;
bool good_ranking (const vector<int>& pos_in_ranking) {
    
}

bool find_ranking (vector<int>& ranking, vector<int>& pos_in_ranking,
                  vector<bool>& used, int idx){
    if (idx == n) return good_ranking(pos_in_ranking );
    else {
        for (int i = 0; i < n; ++i) {
            if (not used[i]) {
                ranking[  ] =  ;
                pos_in_ranking[  ] =  ;
                used[i] = true;
                if (find_ranking (ranking, pos_in_ranking ,used,idx+1)) return true;
                used[i] = false;
            } } }
        return false; }

int main () {
    cin >> n; // Students are numbers from 0 to n-1
    int f, s, t; // Read results of matches
    while (cin >> f >> s >> t) matches.push_back(Match(f,s,t));
    vector<int> ranking(n), pos_in_ranking (n);

```

```
vector<bool> used(n,false);
bool b = find_ranking (ranking, pos_in_ranking ,used ,0);
cout << "Ranking found: " << b << endl;
if (b) print_vector (ranking);
}
```

- b) (0.5 pts). Assumim que tenim m partides i que ens donen una mala implementació de *good_ranking* que sempre triga temps $\Theta(m)$. En funció d' n , quantes vegades es crida a la funció *good_ranking* en el cas pitjor? A partir d'aquest nombre dóna una fita inferior en funció d' n i m del cost en cas pitjor del codi anterior. Valorarem la precisió d'aquest fita.

- c) (1.25 pts.) És possible solucionar el problema anterior en temps polinòmic en n i m en cas pitjor? Si és possible, explica molt breument com ho faries i justifica el cost. Si no és possible, explica per què.

Problema 3**(2.5 pts.)**

- (a) (0.75 pts.) Després d'una llarga vida dedicada a obscurs negocis, el cap d'una perillosa organització mafiosa decideix reunir, a mode de comiat, els seus n col·laboradors per agrair-los la feina feta. No obstant, treballar en assumptes tan delicats ha fet que cadascun d'ells tingui una llista de col·laboradors amb qui no vol coincidir. Sabem, a més, que si A apareix a la llista de persones que B vol evitar, B apareix també a la llista de

persones que A vol evitar. El cap disposa de 5 dies, i vol citar cada treballador exactament un dia de manera que es respectin els desitjos de no-coincidència. Seríeu capaços de determinar, en temps polinòmic en n , si es poden organitzar aquestes 5 trobades?

Nota: en aquesta pregunta i les següents, **cal** justificar les respostes escrivint reduccions i utilitzant que, per certs problemes que hem vist a classe, es coneixen (o no) algorismes polinòmics que els resolen. No cal demostrar que les reduccions són correctes, però heu de deixar clar des de quin problema a quin altre es fa la reducció.

- (b) (1 pt.) Després de pensar-s'ho una estona, el cap decideix que només vol dedicar 2 dies per reunir a tothom. Seríeu capaços de solucionar aquest problema en temps polinòmic en n ?

- (c) (0.75 pts.) Finalment, el cap canvia de parer i decideix que les restriccions dels col·laboradors no tenen massa sentit i per tant, les ignorarà. Continua disposant només de 2 dies, i no vol que s'agrupin tots els col·laboradors més importants un dia, i els de menys importància l'altre. Per tal d'aconseguir-ho, sap el patrimoni de tots els seus col·laboradors i vol aconseguir que el patrimoni total dels col·laboradors reunits el primer dia sigui igual al patrimoni dels reunits el segon dia. Seríeu capaços de solucionar aquest problema en temps polinòmic en n ?

Problema 4**(2.5 pts.)**

Donat un vector v d' n enters diferents ordenats de forma creixent i un enter x , volem determinar si x apareix a v . Si és el cas, també volem saber la seva posició. Com bé sabem, aquest problema el podem solucionar en temps $\Theta(\log n)$ en cas pitjor. No obstant, ens asseguren que de fet x sempre hi apareix i que gairebé sempre ho fa en les primeres posicions del vector. Amb aquesta informació a les mans, ens interessa trobar un algorisme tal que, si l'aparició d' x dins v és a la posició i , aleshores l'algorisme triga temps $\Theta(\log i)$ en cas pitjor.

- (a) (1 pt.) Completa el següent codi per resoldre, en temps $\Theta(\log i)$, el problema que acabem de presentar.

```
// Si x apareix dins v[l...r] retorna i tal que v[i] = x
// Si x no apareix dins v[l...r] retorna -1
// Cost: Theta(log(r-l+1))
int bin_search (int x, const vector<int>& v, int l, int r);

int search (int x, const vector<int>& v) {
    int n = v.size ();
    if (n == 0) return -1;
    int b = 1;
    while ( [ ] and [ ] ) b *= 2;
    return bin_search (x, v, b/2, [ ] );
}
```

- (b) (1.5 pts.) Demuestra que, efectivament, si l'aparició d' x dins v és a la posició i , aleshores la funció *search* triga temps $\Theta(\log i)$ en cas pitjor.



Examen Final EDA

Duració: 3h

02/06/2022

Problema 1

(2 pts.)

Responen a les preguntes següents:

- (a) (1 punt) Si sabem que v és un vector de nombres naturals estrictament més petits que un cert valor K , què retorna la funció següent?

```
vector<int> mystery (const vector<int>& v) {  
    vector<int> aux(K,0);  
    vector<int> res;  
    for (int x : v) ++aux[x];  
    for (int i = 0; i < K; ++i)  
        for (int j = 0; j < aux[i]; ++j) res.push_back(i);  
    return res;  
}
```

Si la mida de v és n , quin és el cost de la funció anterior en funció d' n ? Assumiu que el cost d'un *push_back* és $\Theta(1)$ i que K és una constant que no depèn d' n .

- (b) (0.5 pts.) Existeix algun arbre binari de cerca, amb com a mínim 3 nodes, tal que els nodes continguin nombres enters tots diferents i sigui també un *min-heap*? Si existeix, dibuixeu-lo. Si no, expliqueu per què no pot existir.

- (c) (0.5 pts.) Sabem que el cost d'un cert algorisme ve donat per la recurrència $T(n) = aT(n/3) + \Theta(n^2)$. Analitzeu el cost de l'algorisme en funció del paràmetre natural $a > 0$.

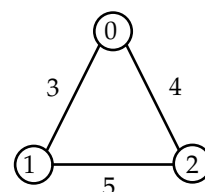
Problema 2

(3 pts.)

En aquest problema considerem grafs no dirigits amb costos enters positius a les arestes. Els vèrtexs s'identifiquen amb nombres consecutius 0, 1, 2, etc. Representem els grafs amb llistes d'adjacència usant vectors de vectors de parells d'**ints**, on el primer component és el cost de l'aresta i el segon, el vèrtex adjacent:

```
typedef pair<int,int> CostVertex;
typedef vector<vector<CostVertex>> Graph;
```

- (a) (0.75 pts.) Considereu el graf mostrat a continuació:



Completeu el codi següent perquè la variable g prengui per valor el graf donat.

$Graph\ g = \{ \quad \quad \quad \};$

(b) (0.75 pts.) Considereu el fragment de codi següent:

```
vector<CostVertex> mystery(const Graph& g) {
    int n = g.size ();
    vector<CostVertex> t(n, {-1, -1});
    vector<int> a(n, INT_MAX);
    a[0] = 0;
    vector<bool> marked(n, false);
    priority_queue <CostVertex, vector<CostVertex>, greater<CostVertex> > pq;
    pq.push({0, 0});
    while (not pq.empty()) {
        int u = pq.top ().second;
        pq.pop ();
        if (not marked[u]) {
            marked[u] = true;
            for (CostVertex cv : g[u]) {
                int c = cv.first ;
                int v = cv.second;
                if (a[v] > a[u] + c) {
                    a[v] = a[u] + c;
                    t[v] = {c, u};
                    pq.push({a[v], v});
                }
            }
        }
    }
    return t;
}
```

Al final de l'execució de la funció *mystery*, quin significat té el valor de $a[u]$ per a un vèrtex u donat? No cal justificació.

(c) (1.5 pts.) Considerem el codi de l'apartat (b). Suposem que g és connex.

Si t és el vector retornat per *mystery*(g), sigui T el conjunt d'arestes $\{u, v, c\}$ tals que $0 \leq u < n$, $t[u] = \{c, v\}$ i $v \neq -1$. Es pot veure (no cal que ho demostreu) que T indueix un subgraf de g que és connex i sense cicles, o sigui un arbre.

Justifiqueu que T és un arbre **d'expansió**.



És necessàriament un arbre d'expansió **mínim**? Si la resposta és afirmativa, justifiqueu-ho. Si no, doneu-ne un contraexemple raonat.

Problema 3

(2.5 pts.)

El conegut joc del *Paraulògic* presenta el següent problema: formar el major nombre de paraules utilitzant només una llista de lletres donades, totes elles diferents, d'entre les quals destaca una lletra obligatòria. Per formar una paraula, cada lletra es pot utilitzar tantes vegades com es vulgui, però sempre cal utilitzar com a mínim una vegada la lletra obligatòria.

D'entre totes les paraules que es poden formar, destaquen els *tutis*, paraules que contenen cada lletra almenys un cop. És obvi que la lletra obligatòria no juga cap paper en la generació de *tutis*.

- (a) (1.5 pts.) Completa el següent codi per tal de generar tots els *tutis* de com a molt mida N que es poden formar a partir d'una sèrie de lletres **minúscules** donades. No cal que implementis cap poda. Recordeu que la classe *string* permet afegir i treure elements del final de l'objecte com si es tractés d'un *vector<char>*.

```
// determina si la paraula p apareix al diccionari
bool existeix (const string& p);

vector<char> lletres;

// vegades conté el nombre de vegades que cada lletra apareix dins sol_parcial
// pendents indica quants elements de lletres no apareixen a sol_parcial
void tutis (int n, string& sol_parcial , vector<int>& vegades, int pendents) {
    if ( sol_parcial .size () == n){
        if (  ) cout << sol_parcial << endl;
    }
    else {
        for (char c : lletres ) {
```

```
} } }
```

```
void tutis (int n) {
    string sol_parcial ;
    vector<int> vegades('z'-'a'+1, 0);
    tutis(n, sol_parcial , vegades, lletres .size ());
}

int main ( ) {
    int N; cin >> N;
    char c; while (cin >> c) lletres .push_back(c);
    for (int n = lletres .size (); n ≤ N; ++n) {
        cout << "Tutis de mida " << n << endl;
        cout << string(20,'=') << endl;
        tutis (n);
    }
}
```

- (b) (1 punt.) Si les lletres a utilitzar són $\{c,s,a,t,e,p\}$ i en un cert moment de l'execució volem generar *tutis* de mida 8 i *sol_parcial* és *casass*, podem aturar la cerca perquè no podrem estendre *casass* a un *tutis* de mida 8. Indica com modificaries el codi anterior per incorporar aquesta idea.

Problema 4**(2.5 pts.)**

Per a cada afirmació donada a continuació, marqueu amb una X la casella corresponent segons si és certa o falsa. Podeu deixar respostes en blanc.

Cada resposta correcta sumarà 0.5 punts. Cada resposta equivocada restarà 0.5 punts. Les respostes en blanc no compten. En cas de tenir més respostes equivocades que correctes la nota de l'exercici serà 0.

	(1)	(2)	(3)	(4)	(5)
CERT					
FALS					

- (1) Siguin A i B dos problemes tals que $A \in \mathbf{P}$ i que existeix una reducció polinòmica d' A cap a B . Aleshores necessàriament $B \in \mathbf{P}$.
- (2) Donats dos problemes A, B **NP**-complets qualssevol, existeix una reducció polinòmica d' A cap a B i viceversa.
- (3) Existeix algun problema que pertany a **P** però no pertany a **EXP**.
- (4) Si trobéssim un problema A que fos **NP**-difícil i pertanyés a **P**, podríem afirmar que $\mathbf{P} = \mathbf{NP}$.
- (5) Tots els problemes **NP**-difícils pertanyen a **EXP**.

Examen Final EDA

Duració: 3h

16/01/2023

Problema 1

(2 pts.)

Responen les preguntes següents:

(a) (0.75 pts.) Considereu el procediment següent:

```
void f (int x) {  
    if (x ≠ 0) {  
        f(x/2);  
        cout << x%2;  
    }  
}
```

Sigui x un nombre natural i sigui n el nombre de bits de x . Quin és el cost de f en funció d' n ?

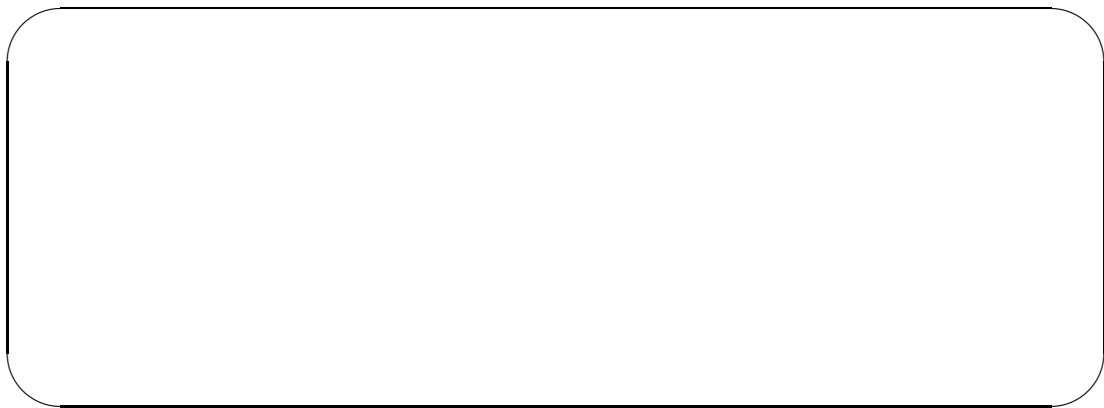
(b) (1.25 pts.) Considereu la funció següent, on assumirem que la mida de v és sempre una potència de 2:

```
double mystery (const vector<double>& v) {  
    if (v.size () == 1) return v [0];  
    else {  
        vector<double> aux;  
        for (int i = 0; i < v.size (); i+=2)  
            aux.push_back((v[i] + v[i+1])/2); // assumim cost Theta(1)  
        return mystery(aux);  
    }  
}
```

Què calcula aquesta funció? Justifica formalment la teva resposta.

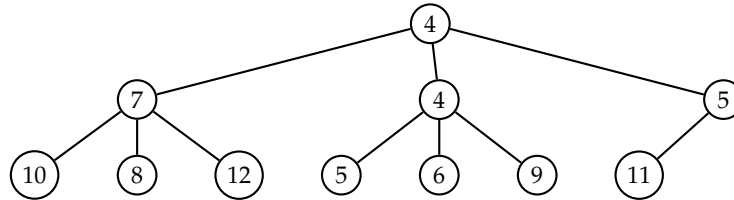


Si $n = v.size()$, quin és el cost d'aquest programa en funció d' n ?



Problema 2**(3 pts.)**

Diem que un arbre ternari d'alçada h és *complet* si els seus h primers nivells estan plens i l'últim nivell té totes les fulles el màxim a l'esquerra possible. Un *min-heap ternari* és un arbre ternari complet tal que el valor de tot node és menor o igual que el valor dels seus fills. Un exemple de min-heap ternari d'alçada 2 és el següent:



- (a) (0.75 pts.) Demostra que per a tot $h \geq 0$ tenim $1 + 3 + 3^2 + \dots + 3^h = \frac{3^{h+1}-1}{2}$.

- (b) (0.75 pts.) Quin és el mínim nombre de nodes que un min-heap ternari d'alçada h pot tenir? Utilitzeu aquesta quantitat per demostrar que l'alçada d'un min-heap ternari amb n nodes és $O(\log n)$.

- (c) (0.5 pts.) De la mateixa manera que ho fem amb els min-heaps binaris, guardarem un min-heap ternari amb n nodes en un vector de mida $n + 1$, on la primera posició no la

utilitzarem. Per exemple, el min-heap de la figura anterior el guardarem en el vector

0	1	2	3	4	5	6	7	8	9	10	11
X	4	7	4	5	10	8	12	5	6	9	11

Donat un node que es guarda a la posició i del vector, en quines posicions es guarden els seus fills? I el seu pare? No cal que justifiqueu la resposta.

- (d) (1 pt.) Us donem a continuació una implementació parcial d'un min-heap ternari per a guardar enters. Completeu-la per tal que, donat un min-heap ternari amb n elements, la funció `remove_min` tingui cost $\Theta(\log n)$ en cas pitjor.

```

class THeap {
    vector<int> v;
    void sink (int i);
    public:
    THeap () {v.push_back(0);}
    int size ( ) const;
    int min ( ) const;
    void add (int x);
    int remove_min ( );
};

int THeap::remove_min ( ){
    int x = v[1];
    v[1] = v.back ();
    v.pop_back ();
    sink (1);
    return x; }

void THeap::sink (int i) {
    if (  < v.size()) {
        int pos_min =  ;



        if (v[pos_min] < v[i]) {



        } } }

```

Problema 3**(2 pts.)**

Com ja sabem, donat un conjunt finit de variables $\{x_1, x_2, \dots, x_n\}$, diem que un **literal** és una variable (x_i) o bé la negació d'una variable ($\neg x_i$). Una **clàusula** és una disjunció de literals, per exemple, $x_3 \vee \neg x_1 \vee \neg x_2$. Una fórmula en **CNF** és una conjunció de clàusules.

El conegut problema **CNF-SAT** consisteix en, donada una fórmula F en CNF, determinar si F té almenys un model. És a dir, decidir si existeix una funció α que assigna cert o fals a cada variable i satisfà F .

Per resoldre aquest problema assumirem que les fórmules venen donades en el format DIMACS, on la primera línia indica el nombre de variables i clàusules, i les variables són nombres $\{1, 2, \dots, n\}$.

Fórmula:

$$\begin{aligned} &(x_1 \vee \neg x_2) \wedge \\ &(x_2 \vee \neg x_3 \vee x_1) \wedge \\ &(x_3) \wedge \\ &(x_2 \vee x_3) \end{aligned}$$

Format DIMACS:

```
p cnf 3 4
1 -2 0
2 -3 1 0
3 0
2 3 0
```

- (a) (1.5 pts.) Omple el següent codi per tal de determinar si una fórmula en CNF és satisfactible. Dins la funció *SAT* no pots utilitzar cap *if*:

```
int main ( ){
    vector<vector<int>> F;
    int n, m; // n variables, m clauses
    string aux;
    cin >> aux >> aux >> n >> m;
    for (int i = 0; i < m; ++i) {
        F.push_back({});
        int lit ;
        while (cin >> lit and lit != 0) F.back().push_back( lit );
    }
    vector<bool> alpha(1); // alpha[0] not used because var 0 does not exist
    cout << SAT(n, F, alpha) << endl;
}

bool evaluate_lit (int lit , const vector<bool>& alpha) {
    if ( lit > 0) return alpha[ lit ];
    else return not alpha[- lit ];
}
```

```
bool evaluate (const vector<vector<int>>& F, const vector<bool>& alpha) {
    for (int i = 0; i < F.size (); ++i) {
```

```
    }
    return true; }
```

```
bool SAT (int n, const vector<vector<int>>& F, vector<bool>& alpha) {
```

```
    if (alpha.size () == n+1)
        return evaluate (F,alpha );
```

```
    else {
```

```
        bool b1 = SAT(n,F,alpha);
```

```
        bool b2 = SAT(n,F,alpha);
```

```
        return
```

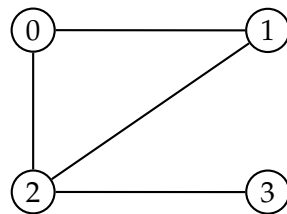
```
    } }
```

- (b) (0.5 pts.) En funció d' n , quantes vegades es crida la funció *evaluate* en el cas pitjor? I en el cas millor?

Problema 4**(3 pts.)**

Per a tot enter $k \geq 1$, donat un graf $G = (V, E)$ no dirigit, el problema **k-COL** consisteix en determinar si existeix una funció $c : V \rightarrow \{1, 2, \dots, k\}$ de manera que per a tota aresta $\{u, v\} \in E$ es compleixi $c(u) \neq c(v)$.

- (a) (0.7 pts.). Ens asseguruen que un procediment *reduccio* és una reducció polinòmica de **k-COL** cap a **CNF-SAT**. Donat el graf de l'esquerra i $k = 3$, aquest procediment escriu la fórmula en CNF de la dreta, on cada línia és una clàusula, \vee indica una disjunció, les negacions de variables es representen amb "-" i, intuïtivament, una variable $x(i, j)$ serà certa si i només si el vèrtex i té el color j . Hem afegit línies en blanc per millorar la llegibilitat.



$\neg x(0, 1) \vee \neg x(1, 1)$
 $\neg x(0, 1) \vee \neg x(2, 1)$
 $\neg x(1, 1) \vee \neg x(2, 1)$
 $\neg x(2, 1) \vee \neg x(3, 1)$

$\neg x(0, 2) \vee \neg x(1, 2)$
 $\neg x(0, 2) \vee \neg x(2, 2)$
 $\neg x(1, 2) \vee \neg x(2, 2)$
 $\neg x(2, 2) \vee \neg x(3, 2)$

$\neg x(0, 3) \vee \neg x(1, 3)$
 $\neg x(0, 3) \vee \neg x(2, 3)$
 $\neg x(1, 3) \vee \neg x(2, 3)$
 $\neg x(2, 3) \vee \neg x(3, 3)$

De forma més precisa, si G és un graf amb n vèrtexs $\{0, 1, 2, \dots, n-1\}$ representat com a llistes d'adjacència, i $1 \leq k \leq n$, el procediment *reduccio* és el següent:

```
string x (int u, int k) { // retorna l'string "x(u,k)"
    return "x(" + to_string(u) + "," + to_string(k) + ")";
}

void reduccio (const vector<vector<int>>& G, int k) {
    int n = G.size();
    for (int c = 1; c <= k; ++c)
        for (int u = 0; u < n; ++u)
            for (int v : G[u])
                if (v > u) cout << "-" << x(u,c) << " v -" << x(v,c) << endl;
}
```

Expliqueu per què el procediment anterior no compleix totes les propietats d'una reducció polinòmica. *Pista:* si necessiteu un contraexemple, n'hi ha prou amb considerar un cert graf amb 3 vèrtexs i $k = 2$.

- (b) (0.7 pts.) Expliqueu com modificaríeu el procediment anterior per a què sigui una reducció polinòmica correcta. No és necessari escriure codi en C++.

- (c) (1.6 pts.) Considereu les següents afirmacions sobre **k-COL**:

- (1) Si G és una instància positiva de **2-COL**, aleshores també és una instància positiva de **3-COL**.
- (2) Si G és una instància positiva de **4-COL**, aleshores també és una instància positiva de **3-COL**.
- (3) Si trobéssim un algorisme polinòmic per **3-COL**, també existiria un algorisme polinòmic per **4-COL**.
- (4) Si trobéssim un algorisme polinòmic per **4-COL**, també existiria un algorisme polinòmic per **3-COL**.

Ompliu la següent taula amb una C o una F depenent de si l'afirmació corresponent és Certa o Falsa. Cada resposta correcta suma 0.4 punts. Cada resposta incorrecta resta 0.4. Les respostes en blanc no compten.

1	2	3	4
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>

Examen Final EDA

Duració: 3h

12/06/2023

Problema 1

(2 pts.)

Responen les preguntes següents utilitzant, quan calgui, els teoremes mestre adients:

(a) (1 pt.) Considereu la funció següent:

```
int f(const vector<int>& v, int e, int d) {
    if (d ≤ e) return 1;
    return f(v, A, B) + f(v, C, D);
}
```

Ompliu les caixes A, B, C, D per tal que, donat un vector v de mida n , una crida $f(v, 0, v.size() - 1)$ tingui cost $\Theta(\log n)$. Feu el mateix per a cost $\Theta(n)$.

(b) (1 pt.) Considereu el codi següent:

```
bool cerca2 (int x, const vector<int>& v, int e, int d) {
    for (int i = e; i ≤ d; ++i)
        if (v[i] == x) return true;
    return false; }
```

```
bool cerca3 (int x, const vector<int>& v, int e, int d) {
    if (e > d) return false;
    int m = (e+d)/2;
    if (v[m] == x) return true;
    else if (v[m] < x) return cerca3(x, v, m+1, d);
    else return cerca3(x, v, e, m-1); }
```

```
bool cerca (int x, const vector<int>& v, int e, int d) {
    if (d - e < 2) {
        for (int i = e; i ≤ d; ++i)
            if (v[i] == x) return true;
        return false;
    }
    int n = d - e + 1, p1 = e + n/3, p2 = d - n/3;
    if (cerca2(x, v, e, p1 - 1)) return true;
    if (cerca3(x, v, p1, p2)) return true;
    return cerca(x, v, p2 + 1, d); }
```

Si v és un vector de mida n , quin és el cost en cas pitjor, en funció d' n , d'una crida a $\text{cerca}(x, v, 0, v.size() - 1)$?

Problema 2**(2 pts.)**

Considerem una implementació d'arbres binaris de cerca on el nodes tenen l'estructura següent:

```
struct Node {  
    int key;  
    Node* left ; // Punter al fill esquerre  
    Node* right ; // Punter al fill dret  
};
```

Us demanem que, a partir d'un arbre binari de cerca amb n nodes, construïu un *max-heap* que contingui totes les claus de l'arbre en temps $\Theta(n)$. Si hi ha més d'un *max-heap* possible, escolliu el que vulgueu. Recordeu que un *heap* es pot implementar com un vector. Heu d'implementar una funció

```
vector<int> to_heap (Node* n);
```

on n és un punter a l'arrel de l'arbre binari de cerca. Podeu utilitzar funcions auxiliars. Us demanem codi en C++. Descripcions a alt nivell rebran 0 punts.

Problema 3**(3 pts.)**

En aquest problema representarem un graf dirigit amb nodes $\{0, 1, \dots, n-1\}$ com una matriu d'adjacència $n \times n$, on la fila i , columna j , indica si existeix un arc $i \rightarrow j$.

```
typedef vector<vector<bool>> Graf;
```

Un graf *torneig* és un graf dirigit on per a cada parell de vèrtexs diferents hi ha exactament un arc, i sense arcs des de cap vèrtex cap a ell mateix.

- (a) (1 pt.) Escriviu una funció que determini si un graf donat d' n vèrtexs és un graf torneig en temps $\Theta(n^2)$ en el cas pitjor.

```
bool es_torneig (const Graf& G) {
```

```
}
```

- (b) (1 pt.) Demostreu per inducció en el nombre de vèrtexs que tot graf torneig té un camí Hamiltonià, és a dir, un camí que visita tots els vèrtexs exactament una vegada (pista: mostreu que un nou vèrtex es pot inserir en un camí d' $n-1$ vèrtexs per obtenir un camí d' n vèrtexs).

- (c) (1 pt.) Valent-vos de la demostració anterior, doneu un algorisme de cost com a molt $\mathcal{O}(n^2)$ que retorni un camí Hamiltonià d'un graf torneig. Us pot ser útil representar el camí com una llista de nodes (enters). No és necessari que doneu codi, una descripció a alt nivell serà suficient. Justifiqueu el cost, en cas pitjor, del vostre algorisme.

Problema 4**(3 pts.)**

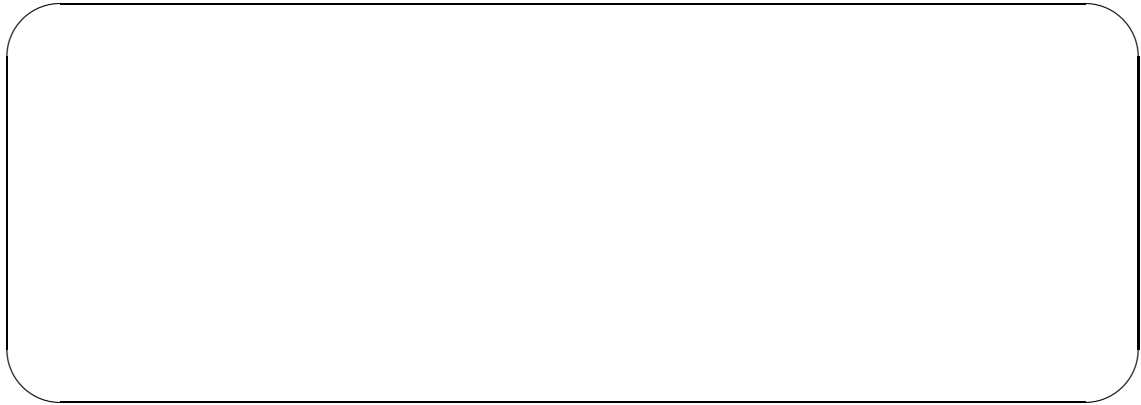
Per a cadascuna de les preguntes següents, responeu raonadament si són certes, falses o no ho sabem. En cas de ser certes, indiqueu dos possibles problemes A i B que compleixin la propietat mencionada. En cas de no saber-se, expliqueu què implicaria que fos certa i què implicaria que fos falsa.

a) (1 pt.) **Existeixen** dos problemes diferents A i B tals que:

- $A \in P$
- $B \in NP$ -difícil
- B es pot reduir polinòmicament cap a A

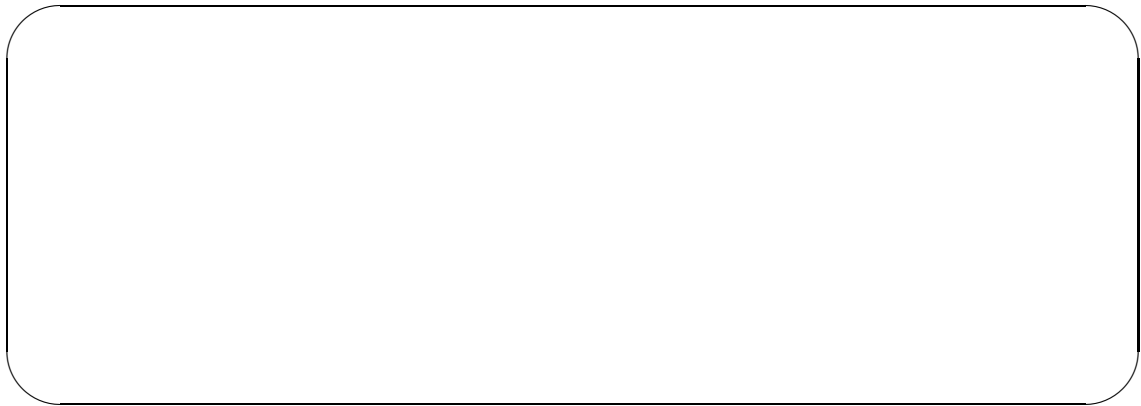
b) (1 pt.) **Existeixen** dos problemes diferents A i B tals que:

- $A \in P$
- $B \in NP$ -complet
- A es pot reduir polinòmicament cap a B



c) (1 pt.) **Existeixen** dos problemes diferents A i B tals que:

- $A \in NP$ -complet
- $B \in NP$ -difícil
- A es pot reduir polinòmicament cap a B i B cap a A .



Solució de l'Examen Final EDA**08/01/2021****Proposta de solució al problema 1**

- (a) És fàcil veure que la funció visita cada element del vector una vegada. Per a cada element, la instrucció `++M[x]` busca l'element x al diccionari i l'incrementa. A continuació hi ha una altra cerca de x al diccionari. Per tant, el cost asimptòtic total és n vegades el cost d'una cerca.

Si utilitzem un AVL, aleshores cada cerca té cost en cas pitjor $O(\log n)$, pel que el cost total serà $O(n \log n)$.

Si utilitzem una taula de dispersió, cada cerca té cost en cas mitjà $\Theta(1)$, pel que el cost total serà $\Theta(n)$.

- (b) Ens hem de fixar en la funció recursiva *majority_pairs* que pren dos arguments. Fins al primer bucle, totes les operacions són constants. El bucle té cost $\Theta(n)$, i crea un vector de mida com a molt $n/2$. La crida recursiva, doncs es fa sobre un vector de mida la meitat. A continuació, es fa una crida a la funció *times* sobre un vector de mida n . El cost d'aquesta funció, ja que el vector es passa per referència, es pot descriure com $C(n) = C(n-1) + \Theta(1)$, que té solució $C(n) \in \Theta(n)$. Tot plegat, veiem que el cost de la funció *majority_pairs* es pot descriure amb la recurrència $T(n) = T(n/2) + \Theta(n)$. Aplicant el teorema mestre, podem afirmar que el cost en cas pitjor és $\Theta(n)$.

Proposta de solució al problema 2

- (a) El codi resultant és:

```
void write_choices (vector<int>& partial_sol, int partial_sum, int idx) {
    if (partial_sum > money) return;
    if (idx == p.size ()) {
        if (partial_sum == money) {
            cout << "{";
            for (uint i = 0; i < partial_sol.size (); ++i)
                cout << (i == 0 ? "" : ",") << partial_sol[i];
            cout << "}" << endl;
        }
    }
    else {
        partial_sol.push_back(idx);
        write_choices (partial_sol, partial_sum + p[idx], idx+1);
        partial_sol.pop_back();
        write_choices (partial_sol, partial_sum, idx+1);
    }
}
```

- (b) En el codi anterior, podem la solució quan ja ens hem gastat més dels diners que tenim.

Adicionalment, podarem una solució parcial quan, fins i tot considerant que escollim tots els immobles que ens queden per processar, no podem arribar a la quantitat de diners que ens hem de gastar. És a dir, podem la cerca quan hem descartat massa immobles.

Per implementar-ho de manera eficient, passarem un paràmetre més al procediment *write_choices*, que contindrà la suma dels preus dels immobles que ens queden per

processar. Dins el main, sumarem inicialment tots els preus i aquest serà el valor del paràmetre en la crida a *write_choices*. En les dues crides recursives, el paràmetre es veurà decrementat en $p[idx]$. Finalment, si anomenem *remaining_sum* a aquest paràmetre, després de la primera poda ja existent afegirem:

if (*partial_sum* + *remaining_sum* < *money*) **return**;

Proposta de solució al problema 3

- (a) Sí, és una instància positiva. Per exemple, podem agrupar els nombres en 3 conjunts $\{3, 8, 20\}, \{6\}, \{22\}$ de manera que no posem dos nombres que tinguin 1's a posicions comuns al mateix conjunt. Recordem que:

$$3 = 00011_2, 6 = 00110_2, 8 = 01000_2, 20 = 10100_2, 22 = 10110_2$$

- (b) Cal veure diverses coses:

- *Testimonis*: els candidats a testimonis són totes les possibles maneres que tenim de distribuir els elements de N en p conjunts. Observem que la mida d'un candidat és polinòmica respecte la mida de la instància. De fet, és lineal.
 - *Verificador*: rebrà un parell (N, p) i p conjunts S_1, S_2, \dots, S_p on s'han distribuït els nombres de N . Per a cada conjunt S_i , considerarà totes les parelles de nombres de S_i (com a molt un nombre quadràtic de parelles a cada S_i), i comprovarà que la representació en binari de la parella no tingui 1's en posicions comunes (això es pot fer en temps lineal en el nombre de bits del nombre més gran). Tot plegat es pot fer en temps polinòmic.
 - *Testimonis només per instàncies positives*: una instància és positiva si i només si hi ha una manera de distribuir els nombres en subconjunts de manera correcta i aquesta distribució és precisament un testimoni. Per tant, les instàncies positives tenen testimonis, mentre que les instàncies negatives no en tindran.
- (c) La mida de (N, p) és polinòmica respecte la mida de (G, k) . D'una banda $p = k$. Pel que fa a N , aquest conté un nombre per cada vèrtex de G , i la mida d'aquests nombres coincideix amb el nombre d'arestes de G .

Anem a veure que $(G, k) \in \text{COLORABILITAT} \Leftrightarrow (N, p) \in \text{DISTINCT-ONES}$:

$(G, k) \in \text{COLORABILITAT} \Rightarrow (N, p) \in \text{DISTINCT-ONES}$:

Si (G, k) és una instància positiva és perquè existeix una funció c que coloreja els vèrtexs amb k colors de manera que si dos vèrtexs tenen una aresta en comú aleshores tenen color diferent.

Podem distribuir els nombres x_i en p (que és igual a k) conjunts de la manera següent: x_i va al conjunt S_j si i només si $c(v_i) = j$ (és a dir, si v_i té color j). Només ens cal veure que no posem al mateix conjunt dos nombres que tenen 1s en posicions comunes. Fem-ho per reducció a l'absurd: assumim que existeixen dos nombres x_r i x_s que van a un conjunt S_j i tenen un 1 a la posició i . Si tenen un 1 comú en el bit i -èssim és perquè els v_r i v_s pertanyen a l'aresta e_i . Si van al conjunt S_j és perquè $c(v_r) = c(v_s) = j$. I això no pot ser perquè els vèrtexs units per una aresta tenen colors diferents.

$(N, p) \in \text{DISTINCT-ONES} \Rightarrow (G, k) :$

Si (N, p) és una instància positiva és perquè podem agrupar els nombres de N en p conjunts S_1, S_2, \dots, S_p de manera que si dos nombres tenen 1s en posicions comunes aleshores van a conjunts diferents. Recordem, a més, que $p = k$.

Aleshores considerem la coloració següent pels vèrtexs de G : $c(v_i) = j$ si i només si x_i pertany al conjunt S_j . Com que $p = k$, aquesta coloració utilitza com a molt k colors. A més, si prenem dos vèrtexs v_r i v_s que tenen una aresta e_i en comú, sabem per definició que x_r i x_s tindran el bit i -èssim a 1, i per tant no podran anar al mateix conjunt. Així doncs, c els assignarà un color diferent.

Proposta de solució al problema 4

- (a) És fàcil veure que el nombre de nodes $N(k)$ d'un arbre binomial B_k es pot descriure amb la recurrència:

$$N(k) = \begin{cases} 1, & \text{si } k = 0 \\ 2 \cdot N(k-1), & \text{si } k > 0 \end{cases}$$

Podem demostrar per inducció sobre k que la solució a la recurrència és $N(k) = 2^k$. El cas base és trivial, ja que $2^0 = 1$. Assumim ara la hipòtesi d'inducció $N(k-1) = 2^{k-1}$. Per tant $N(k) = 2 \cdot N(k-1) = 2 \cdot 2^{k-1} = 2^k$, com volíem demostrar.

- (b) Com que el nombre de nodes d'un arbre binomial d'ordre k és 2^k , i només hi pot haver com a molt un arbre binomial d'ordre k per a cada k , podem veure que en un heap binomial amb n nodes hi haurà un (i només un) arbre binomial d'ordre k si i només si el k -èssim bit menys significatiu d' n en binari és 1.
- (c) Si l'arrel d' A és menor que l'arrel de B , aleshores afegirem B com el fill de més a l'esquerra de l'arrel d' A . Es cas contrari, afegirem A com el fill de més a l'esquerra de l'arrel de B .
- (d) El codi completat és el següent:

```
void BinomialHeap::merge(BinomialHeap& h){
// Make sure both have the same size (to make 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;
        carry = NULL;
    }
    else if (roots[k] == NULL) {
        if (carry == NULL) newRoots[k] = h.roots[k];
        else {
            newRoots[k] = NULL;
            carry = mergeTreesEqualOrder(carry, h.roots[k]);
        }
    }
    else if (h.roots[k] == NULL) {
        if (carry == NULL) newRoots[k] = roots[k];
        else {

```

```
        newRoots[k] = NULL;
        carry = mergeTreesEqualOrder(carry, roots[k]);
    }
    else {
        newRoots[k] = carry;
        carry = mergeTreesEqualOrder(roots[k], h. roots[k]);
    }
}

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

Podem apreciar que totes les operacions que s'hi fan són constants, pel que només hem de comptar quantes voltes dóna el bucle. El bucle fa tantes voltes com la mida de *roots*. La clau és adonar-se que aquest vector té tantes posicions com bits necessitem per representar n , i això són $\Theta(\log n)$ posicions. Així doncs, el cost és $\Theta(\log n)$.

Solució de l'Examen Final EDA**07/06/2021****Proposta de solució al problema 1**

- (a) Notem que $n \log(n^2) = 2n \log n$, i per tant, aquest funció té el mateix grau de creixement que $n \log n$. De fet, aquestes dues són les que tenen menor grau de creixement, seguides per $n(\log n)^2$ i posteriorment per n^2 .

Tot i que no se'ns demanava la justificació, l'adjuntem:

$$\lim_{x \rightarrow \infty} \frac{n \log n}{n(\log n)^2} = \lim_{x \rightarrow \infty} \frac{1}{\log n} = 0$$

Per tant, $n \log n$ és menor asimptòticament parlant.

De forma similar:

$$\lim_{x \rightarrow \infty} \frac{n(\log n)^2}{n^2} = \lim_{x \rightarrow \infty} \frac{(\log n)^2}{n} = 0$$

- (b) Sabem que, o bé tots els problemes NP-complets pertanyen a P o cap d'ells hi pertany. No obstant, no se sap quina de les dues situacions és la correcta. Per tant, no sabem si l'afirmació que ens fan és certa o falsa.

Si fos certa, necessàriament tots els problemes NP-complets pertanyerien a P . Si fos falsa, cap d'ells ho faria.

- (c) Ens fixem que quan no estem al cas base el primer bucle s'executa a vegades, on a cada volta es fa una crida recursiva de mida la meitat. En acabar el primer bucle, passem a fer un treball (els dos bucles ennierats) que té cost $\sum_{i=1}^{n-1} \Theta(i)$, que equival a $\Theta(n^2/2)$ i per tant $\Theta(n^2)$. Així doncs, el cost d'aquesta funció recursiva es pot descriure per:

$$T(n) = a \cdot T(n/2) + \Theta(n^2)$$

d'on identifiquem $b = 2$, $k = 2$ i $\alpha = \log_2 a$. Aplicant el teorema mestre de recurrències divisores podem afirmar que

- Si $a < 4$, aleshores $\alpha < k$ i per tant, $T(n) \in \Theta(n^2)$
- Si $a = 4$, aleshores $\alpha = k$ i per tant, $T(n) \in \Theta(n^2 \log n)$
- Si $a > 4$, aleshores $\alpha > k$, i per tant, $T(n) \in \Theta(n^{\log_2 a})$

Proposta de solució al problema 2

```
(a) bool is_stable (const signed_graph& G, int u, vector<int>& team) {
    for (auto& e : G[u]) {
        int e_t = expected_team(team[u], e.sign);
        if (team[e.target] != no_team and team[e.target] != e_t) return false;
        if (team[e.target] == no_team) {
            team[e.target] = e_t;
            if (not is_stable (G, e.target, team)) return false;
        }
    }
    return true;
}
```

- (b) Ho podem fer perquè si existeix una manera correcta de dividir els vèrtexs en equips A i B , aleshores segur que existeix una manera correcta on el vèrtex 0 va a l'equip A . Només cal adonar-se que, donada una solució on 0 va a l'equip B , si canviem l'equip de tots els vèrtexs, obtindrem una solució correcta on 0 va a l'equip A .

Proposta de solució al problema 3

- (a) Dissenyarem una funció recursiva `canals(int l, int r)` que escriurà un conjunt vàlid de canals de transmissió per als treballadors $\{l, l+1, \dots, r-1, r\}$. Si $l = r$ només hi ha un treballador i per tant no cal cap canal.

En cas contrari, prenem m el punt mig entre l i r , i considerem $n = r - l + 1$. Això ens parteix els treballadors en dues meitats d'aproximadament $n/2$ elements. Primer escriurem, amb una crida recursiva, tots els canals necessaris per comunicar els treballadors $\{l, l+1, \dots, m-1\}$ i a continuació, amb una altra crida recursiva, els canals necessaris per comunicar els treballadors $\{m+1, m+2, \dots, r\}$. Fixem-nos que les úniques comunicacions que ens falten són d'entre treballadors de meitats diferents (i també d'entre treballadors de l'esquerra cap a m). Per tal de comunicar les dues meitats, crearem els canals següents:

- $l \rightarrow m, l+1 \rightarrow m, \dots, m-1 \rightarrow m$
- $m \rightarrow m+1, m \rightarrow m+2, \dots, m \rightarrow r-1, m \rightarrow r$

Intuïtivament, m és l'intermediari per les comunicacions entre treballadors de l'esquerra i treballadors de la dreta. Notem, a més, que hem creat exactament $n - 1$ canals addicionals.

Adjuntem codi per més concreció, tot i que no es demanava.

```
void canals (int l, int r) {
    if (l >= r) return;
    else {
        int m = (l+r)/2;
        canals(l, m-1);
        canals(m+1, r);
        for (int k = l; k < m; ++k)
            cout << k << " --> " << m << endl;
        for (int k = m+1; k <= r; ++k)
            cout << m << " --> " << k << endl;
    }
}
```



```

    }
}

int main() {
    int n; cin >> n;
    canals(1,n);
}

```

- (b) Fixem-nos que el nombre de canals que crea la nostra funció es pot descriure per la recurrència:

$$C(n) = 2C(n/2) + \Theta(n)$$

que, pel teorema mestre de recurrències divisores té solució $C(n) \in \Theta(n \log n)$.

Proposta de solució al problema 4

- (a) Construïrem una instància de 2SAT que serà la conjunció de les següents clàusules:

- Per a cada $i \in V$, una clàusula p_i .
- Per a cada $i \in N$, una clàusula $\neg p_i$.
- Per a cada $(i, j) \in I$, una clàusula $\neg p_i \vee \neg p_j$.
- Per a cada $(i, j) \in R$, una clàusula $\neg p_i \vee p_j$.

- (b) Com que hem construït una reducció polinòmica del problema dels equipaments cap a 2SAT, i sabem que 2SAT és un problema que es pot resoldre en temps polinòmic, aleshores podem concloure que el problema dels equipaments també es pot resoldre en temps polinòmic.

Tot i que no es demanava, remarquem que l'algorisme polinòmic pot consistir en calcular primer la reducció de l'apartat anterior i aplicar un algorisme polinòmic per 2SAT sobre la fórmula resultant.

Solució de l'Examen Final EDA**10/01/2022****Proposta de solució al problema 1**

- (a) El codi escriurà les 5 paraules següents:

pep, pe, ep, e, p

Pel que fa al cost, sabem que el cost de *mystery* ve donat per la recurrència $T(n) = 2T(n-1) + \Theta(n)$. Això és així perquè es fan dues crides recursives de mida $n-1$ i les altres operacions tenen cost constant, excepte les crides a *substr*, que tenen cost $\Theta(n-1) = \Theta(n)$. Aquesta recurrència té solució $T(n) \in 2^n$.

- (b) Sigui $n = s.size()$. Anem a comptar quantes vegades s'executa l'*insert* de dins el bucle. Per cada i , el bucle intern dona exactament $n-i$ voltes. Com que i varia entre 0 i $n-1$, això ens dóna $n + (n-1) + \dots + 1 = \Theta(n^2)$ crides a *substr*. Però amb això no en fem prou perquè cada crida a *substr* té cost $j-i+1$.

Anem a comptar el cost de totes aquestes crides. Recordem que per unes i, j concretes el cost de la crida a *substr* és $j-i+1$. Per una i concreta, j es mou entre i i $n-1$ i per tant, el cost de les crides a *substr* amb aquesta i és $1 + 2 + \dots + (n-i) = \Theta((n-i)^2)$. Si sumem sobre totes les i 's el cost total és $\sum_{i=0}^n \Theta((n-i)^2) = \Theta(n^3)$.

- (c) Una possible solució és:

```
void mystery(const string& s, unordered_set<string>& res){
    for (int k = 1; k ≤ s.size (); ++k)
        for (int i = 0; i + k ≤ s.size (); ++i)
            res . insert (s . substr (i ,k ));
}
```

Proposta de solució al problema 2

- (a) Una possible solució és:

```
bool find_ranking (vector<int>& ranking, vector<int>& pos_in_ranking,
                  vector<bool>& used, int idx){
    if (idx == n) return good_ranking(pos_in_ranking);
    else {
        for (int i = 0; i < n; ++i) {
            if (not used[i]) {
                ranking[idx] = i;
                pos_in_ranking[i] = idx;
                used[i] = true;
                if (find_ranking(ranking, pos_in_ranking, used, idx+1)) return true;
                used[i] = false;
            } } }
        return false;
    }

bool good_ranking (const vector<int>& pos_in_ranking) {
    for (Match& g : matches) {
        if (pos_in_ranking[g.first] > pos_in_ranking[g.second]) return false;
    }
}
```

```

    if ( pos_in_ranking [g.second] > pos_in_ranking [g.third ] ) return false;
  }
  return true;
}

```

- (b) Essencialment el codi genera totes les permutacions dels n jugadors i comprova si n'hi ha alguna de correcta. En cas pitjor no n'hi haurà cap de correcta i per tant haurà de generar i comprovar totes les permutacions. Així doncs es faran $n!$ crides a *good_ranking*.

Com que cada crida a *good_ranking* té cost en cas pitjor $\Theta(m)$, això ens dona una fita inferior del codi de $\Theta(m \cdot n!)$.

- (c) Es pot solucionar el problema en temps polinòmic en n i m . Per a fer-ho construïm un graf dirigit on els nodes són els jugadors. Per cada partida amb resultat (j_1, j_2, j_3) afegim dos arcs $j_1 \rightarrow j_2$ i $j_2 \rightarrow j_3$. Per tant afegirem com a molt $\Theta(m)$ arcs. Un cop hem construït el graf buscarem una ordenació topològica en temps $O(n + m)$. Si l'ordenació topològica acaba sense haver afegit tots els jugadors voldrà dir que no hi ha un rànquing possible.

Nota: si no anem amb compte podríem afegir arcs repetits, però això no és un problema per a la correcció o el cost de l'algorisme de cerca topològica explicat a classe.

Proposta de solució al problema 3

- (a) Anomenem X al problema d'aquest apartat. No és raonable pensar que podem solucionar X en temps polinòmic. Vegem per què. Considerem 5-COL el problema del 5-colorejat de grafs, que sabem que és NP-complet. Existeix una reducció de 5-COL cap a X : donat un graf G amb vèrtexs V i arestes E , considerem el conjunt de col·laboradors $\{c_u | u \in V\}$, i per cada aresta $\{u, v\} \in E$ imposem que el col·laborador c_u vol evitar el col·laborador c_v . Com que hem reduït polinòmicament 5-COL a X , si $X \in P$ també tindrem 5-COL $\in P$, i això és un problema obert a dia d'avui.
- (b) Anomenem Y al problema d'aquest apartat. Podem afirmar que $Y \in P$. Vegem per què. Considerem 2-COL el problema del 2-colorejat de grafs, que sabem que pertany a P . Existeix una reducció de Y cap a 2-COL: donat un conjunt de col·laboradors \mathcal{C} i una llista d'incompatibilitats I_c per cada $c \in \mathcal{C}$, construïm una instància de 2-COL que consisteix en el graf G amb vèrtexs $\{v_c | c \in \mathcal{C}\}$ i arestes $\{\{v_c, v_d\} | d \in \mathcal{C}, c \in I_d\}$. Com que hem trobat una reducció de Y cap a 2-COL i aquest últim pertany a P , aleshores $Y \in P$.
- (c) Anomenem Z al problema d'aquest apartat. No és raonable pensar que podem solucionar Z en temps polinòmic. Vegem per què. Considerem PARTICIÓ, el problema de determinar si podem partir un conjunt d'enters en dues parts que sumin igual. Sabem que aquest problema és NP-complet. Existeix una reducció de PARTICIÓ cap a Z : donat un conjunt d'enters S , considerem el multiconjunt de col·laboradors $\{c_s | s \in S\}$, tal que el patrimoni de c_s és precisament s . Com que hem reduït polinòmicament PARTICIÓ a Z , si $Z \in P$ també tindrem PARTICIÓ $\in P$, i això és un problema obert a dia d'avui.

Proposta de solució al problema 4

- (a) Una possible solució és:

```

int search(int x, const vector<int>& v) {
    int n = v.size ();
    if (n == 0) return -1;
    int b = 1;

```

```
while (b < n and v[b] < x) b *= 2;  
return bin_search(x, v, b/2, min(n-1, b));  
}
```

- (b) El primer que cal fer és observar el comportament del bucle. Després de k voltes, el valor de b és 2^k . Fixem-nos que s'atura tan bon punt troba un valor b tal que $v[b] \geq x$. Per tant, s'atura amb un nombre de voltes k tal que $v[2^k] \geq x$ però tal que $v[2^{k-1}] < x$ i això ens indica que $k = \lceil \log i \rceil$. Per tant, el cost del bucle és $\Theta(\log i)$. Remarquem també que, si el bucle s'atura perquè $b \geq n$, el mateix raonament és vàlid.

Finalment, vegem el cost de la crida a *bin_search*. En el cas pitjor $b \geq n - 1$ i, per tant, el cost de la crida és $\Theta(\log(b - b/2 + 1))$. Com que després de k voltes, b val 2^k , si el bucle ha fet k voltes el cost de la crida a *bin_search* serà $\Theta(\log(2^k - 2^{k-1} + 1)) = \Theta(\log(2^{k-1} + 1))$. Sabem que el nombre de voltes és $k = \lceil \log i \rceil$, i per tant el cost de la crida a *bin_search* és $\Theta(\log i)$.

Resumint, tot plegat té un cost en cas pitjor de $\Theta(\log i)$.

Solució de l'Examen Final EDA

02/06/2022

Proposta de solució al problema 1

- (a) La funció retorna el vector v ordenat de forma creixent.

És fàcil veure que després del primer bucle, $aux[i]$ conté el nombre de vegades que i apareix dins v . Per tant, el segon bucle recorre tots els nombres entre 0 i $K - 1$ i els afegeix a res tantes vegades com apareixen dins v .

Pel que fa al cost de la funció, la inicialització d' aux té cost $\Theta(1)$ perquè K no depèn d' n . El primer bucle té cost $\Theta(n)$ perquè recorre tots els elements de v i incrementa un comptador per cadascun d'ells. Finalment, per analitzar els bucles enllargats ens fixem primer que el bucle extern dona un nombre constant de voltes. Pel que fa al bucle intern, es fan tantes crides a $push_back$ com elements té el vector v . Com que cada crida té cost $\Theta(1)$, el cost dels bucles és $\Theta(n)$. Finalment, observem que retornar res també té cost $\Theta(n)$ i, per tant, el cost global de la funció és $\Theta(n)$.

- (b) No pot existir un arbre com aquest. Per entendre per què, sigui a el nombre que hi ha a l'arrel de l'arbre.

Si l'arbre té fill esquerre i fill dret, siguin fe i fd els nombres que hi apareixen, respectivament. En ser un arbre de cerca, es compleix $fe < a < fd$. Però aleshores, l'element a no compleix la propietat de *min-heap*: que els seus dos fills siguin majors o iguals que ell.

Si l'arbre no té fill esquerre o fill dret, tindrem un node a alçada 2 sense haver omplert totalment el nivell 1 de l'arbre, i això no pot passar en un *heap*.

- (c) Utilitzant la notació de classe, tenim que $b = 3$, $k = 2$ i $\alpha = \log_3 a$. Aleshores, si $a < 9$ el cost és $\Theta(n^2)$; si $a = 9$ el cost és $\Theta(n^2 \log n)$; i si $a > 9$ el cost és $\Theta(n^{\log_3 a})$.

Proposta de solució al problema 2

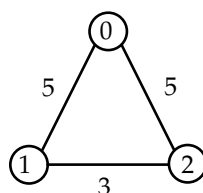
- (a) Una possible solució:

$Graph\ g = \{ \{ \{3, 1\}, \{4, 2\} \}, \{ \{3, 0\}, \{5, 2\} \}, \{ \{4, 0\}, \{5, 1\} \} \};$

- (b) Al final de l'execució de la funció *mystery*, el valor de $a[u]$ és la distància mínima del vèrtex 0 al vèrtex u .

- (c) La funció *mystery* implementa l'algorisme de Dijkstra per trobar els camins mínims des del vèrtex 0 a tots els vèrtexs del graf g . Donat que el graf és connex, des de 0 s'arriba a cada vèrtex. Com que T consisteix en les arestes dels camins mínims des de 0, és un arbre d'expansió.

Però no és necessàriament un arbre d'expansió mínim. Per exemple, la funció *mystery* aplicada a aquest graf:



retorna les arestes $\{ \{0, 1\}, \{0, 2\} \}$. L'arbre corresponent té cost 10. Però l'arbre corresponent a les arestes $\{ \{0, 1\}, \{1, 2\} \}$ també és un arbre d'expansió i té cost 8.

Proposta de solució al problema 3

(a) Una possible solució:

```
void tutis (int n, string& sol_parcial , vector<int>& vegades, int pendants) {
    if ( sol_parcial .size () == n){
        if ( pendants == 0 and existeix ( sol_parcial )) cout << sol_parcial << endl;
    }
    else {
        for (char c : lletres ) {
            sol_parcial .push_back(c);
            ++vegades[c-'a' ];
            if (vegades[c-'a' ] == 1) --pendents;
            tutis (n, sol_parcial , vegades, pendants);
            sol_parcial .pop_back ();
            --vegades[c-'a' ];
            if (vegades[c-'a' ] == 0) ++pendents;
        }
    }
}
```

(b) Una solució consisteix en afegir la següent instrucció a l'inici del procediment *tutis*:

```
if ( ( n - sol_parcial .size ()) < pendants) return;
```

És a dir, si el nombre de “caselles” per omplir a *sol_parcial* és menor que el nombre de lletres que encara hem d'utilitzar, no cal continuar perquè no les podrem utilitzar totes.

Proposta de solució al problema 4

	(1)	(2)	(3)	(4)	(5)
CERT		X		X	
FALS	X		X		X

Solució de l'Examen Final EDA

16/01/2023

Proposta de solució al problema 1

- (a) Observem que f és un procediment recursiu, pel que escriurem la recurrència que descriu el seu cost i la solucionarem. És fàcil observar que totes les operacions que fa el procediment són $\Theta(1)$ (comparacions entre enters, divisions per 2 i residus entre 2), excepte la crida recursiva. Sabem que, si x té n bits, $x/2$ tindrà $n - 1$ bits, pel que la recurrència que descriu el cost és $C(n) = C(n - 1) + \Theta(1)$. Si apliquem el teorema mestre per recurrències substractores del tipus $C(n) = a \cdot C(n - c) + \Theta(n^k)$, podem identificar $a = 1$, $c = 1$ i $k = 0$, que sabem que té solució $C(n) = \Theta(n^{k+1}) = \Theta(n)$.
- (b) La funció retorna la mitjana aritmètica dels nombres del vector v . Per a veure-ho, si la mida de v és 2^k , ho podem demostrar per inducció sobre k .

Per $k = 0$, el vector té un sol element i per tant, la mitjana coincideix amb l'únic element del vector, tal com fa el codi.

Sigui ara $k > 0$ i assumim la hipòtesis d'inducció: per a tot vector de mida 2^{k-1} , la funció retorna la mitjana dels seus nombres. Aleshores, donat un vector $v = (x_1, x_2, \dots, x_{2^k})$ la funció construeix aux , un vector de mida 2^{k-1} amb els elements $((x_1 + x_2)/2, (x_3 + x_4)/2, \dots, (x_{2^{k-1}+1} + x_{2^k})/2)$. Per tant, la hipòtesis d'inducció ens garanteix que la crida recursiva retornarà la mitjana d'aquest conjunt:

$$\frac{\frac{x_1+x_2}{2} + \frac{x_3+x_4}{2} + \dots + \frac{x_{2^{k-1}+1}+x_{2^k}}{2}}{2^{k-1}}$$

que és igual a

$$\frac{x_1 + x_2 + x_3 + x_4 + \dots + x_{2^{k-1}+1} + x_{2^k}}{2^k}$$

és a dir, la mitjana aritmètica dels elements de v .

Pel que fa al seu cost, veiem que és una funció recursiva. El vector es passa per referència, i això té cost $\Theta(1)$. Crear el vector buit aux també té cost $\Theta(1)$. El bucle fa $n/2$ voltes, i a cada volta es fa un treball $\Theta(1)$. Per tant, el cost del bucle és $\Theta(n)$. Finalment, és fàcil veure que el vector aux té mida $n/2$. Així doncs, la recurrència que descriu el cost de la funció és $C(n) = C(n/2) + \Theta(n)$. Si apliquem el teorema mestre per recurrències divisores del tipus $C(n) = a \cdot C(n/b) + \Theta(n^k)$, podem identificar $a = 1$, $b = 2$, $k = 1$ i calcular $\alpha = \log_2(1) = 0$. Com que $k > \alpha$, sabem que la solució és $C(n) = \Theta(n^k) = \Theta(n)$.

Proposta de solució al problema 2

- (a) Ho demostrarem per inducció sobre h . El cas base ($h = 0$) és fàcil perquè és obvi que $1 = \frac{3-1}{2}$.

Sigui $h > 0$ i assumim la hipòtesis d'inducció: $1 + 3 + \dots + 3^{h-1} = \frac{3^h-1}{2}$. Aleshores

$$1 + 3 + \dots + 3^{h-1} + 3^h = \frac{3^h-1}{2} + 3^h = \frac{3^h-1+2 \cdot 3^h}{2} = \frac{3^{h+1}-1}{2}$$

- (b) Per construir un min-heap ternari d'alçada h amb el menor nombre de nodes, haurem d'omplir els h primers nivells i situar un únic node en el nivell $h + 1$.

És fàcil veure que en el primer nivell tenim 1 node, en el segon nivell 3 nodes, en el tercer 3^2 nodes, i en general, en el nivell i tenim 3^{i-1} nodes.

Feta aquesta observació, el nombre mínim de nodes d'un min-heap ternari és $1 + 3 + \dots + 3^{h-1} + 1$, que gràcies a l'apartat anterior sabem que equival a $\frac{3^h-1}{2} + 1 = \frac{3^h+1}{2}$.

Per tant, si n és el nombre de nodes d'un min-heap ternari qualsevol d'alçada h , sabem que

$$n \geq \frac{3^h + 1}{2}$$

que equival a afirmar que

$$h \leq \log_3(2n - 1)$$

.

Per tant, podem concloure que $h \in O(\log n)$.

- (c) Donat un node en la posició i , els seus tres fills (en cas que existeixin tots tres) estaran en les posicions $(3i - 1, 3i, 3i + 1)$. El seu pare estarà a la posició $\lfloor \frac{i+1}{3} \rfloor$. També és correcte l'expressió $\lceil \frac{i-1}{3} \rceil$.

- (d) Una possible solució és:

```
void Heap::sink (int i) {
    if (3*i - 1 < v.size ()) {
        int pos_min = 3*i - 1;
        if (3*i < v.size () and v[3*i] < v[pos_min]) pos_min = 3*i;
        if (3*i + 1 < v.size () and v[3*i + 1] < v[pos_min]) pos_min = 3*i + 1;
        if (v[pos_min] < v[i]) {
            swap(v[i], v[pos_min]);
            sink(pos_min);
        }
    }
}
```

Proposta de solució al problema 3

- (a) Una possible solució és:

```
bool evaluate (const vector<vector<int>>& F, const vector<bool>& alpha) {
    for (int i = 0; i < F.size (); ++i) {
        bool some_true = false;
        for (int j = 0; not some_true and j < F[i].size (); ++j)
            some_true = evaluate_lit (F[i][j], alpha);
        if (not some_true) return false;
    }
    return true;
}
```



```

bool SAT(int n, const vector<vector<int>>& F, vector<bool>& alpha) {
    if (alpha.size() == n+1)
        return evaluate(F, alpha);
    else {
        alpha.push_back(false);
        bool b1 = SAT(n, F, alpha);
        alpha.back() = true;
        bool b2 = SAT(n, F, alpha);
        alpha.pop_back();
        return b1 or b2;
    }
}

```

- (b) La clau és observar que, essencialment, aquest programa prova totes les possibles α i, per cada una d'elles crida a la funció *evaluate*. Com que hi ha 2^n possibles α , aquest és el nombre de crides. En aquest programa, el cas millor i pitjor coincideixen.

Si ho volguéssim justificar més formalment, podríem calcular $C(k)$, el nombre de crides a *evaluate* que fa la funció *SAT* quan α té mida $(n+1) - k$.

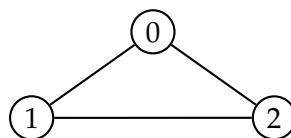
És fàcil veure que $C(0) = 1$, perquè en aquest cas α tindrà mida $n+1$ i estarem en el cas base de *SAT*, que només fa una crida a *evaluate*.

Per a $k > 0$ tenim que $C(k) = 2C(k-1)$, perquè es fan dues crides recursives on s'ha afegit un element a α .

És fàcil veure que aquesta recurrència té solució $C(k) = 2^k$. Quan fem la crida a *SAT* des del main, α té mida 1, i així doncs el nombre de crides a *evaluate* que es faran serà $C(n) = 2^n$.

Proposta de solució al problema 4

- (a) Una de les propietats de les reduccions és que transformen instàncies negatives en instàncies positives. Considerem, per exemple, el graf



Clarament aquesta és una instància negativa de **2-COL**. No obstant, la funció *reduccio* ens construeix una fórmula on totes les variables apareixen només de manera negada. Per tant, és obvi que fent totes les variables falses podem satisfer la fórmula. Així doncs, *reduccio*($G, 2$) és una instància positiva de **2-COL** i aquesta reducció no compleix la propietat que hem esmentat.

- (b) La clau està en adonar-se que, essencialment, la fórmula que construeix *reduccio* assegura que dos vèrtexs units per una aresta no poden tenir el mateix color. No obstant, en cap cas assegura que cada vèrtex té almenys un color. Això ho podem assegurar afegint, per a cada vèrtex i (amb $0 \leq i < n$) una clàusula:

$$x(i, 1) \vee x(i, 2) \vee x(i, 3) \vee \dots \vee x(i, k)$$

Tot i que no era necessari, mostrem el codi C++ corresponent. Caldria afegir el següent bucle al final de *reduccio*:

```
for (int u = 0; u < n; ++u) {  
    for (int c = 1; c ≤ k; ++c) cout << (c==1 ? "" : " v ") << x(u,c);  
    cout << endl;  
}
```

- (c)
- Si G és una instància positiva de **2-COL**, aleshores també és una instància positiva de **3-COL**: Cert
 - Si G és una instància positiva de **4-COL**, aleshores també és una instància positiva de **3-COL**: Fals
 - Si trobéssim un algorisme polinòmic per **3-COL**, també existiria un algorisme polinòmic per **4-COL**: Cert
 - Si trobéssim un algorisme polinòmic per **4-COL**, també existiria un algorisme polinòmic per **3-COL**: Cert

Solució de l'Examen Final EDA

12/06/2023

Proposta de solució al problema 1

(a) Pel cas $\Theta(\log n)$ considerarem la crida recursiva

return $f(v, e, e) + f(v, e, (e+d)/2);$

En aquest cas, observem que la crida a f de l'esquerra es calcularà amb temps $\Theta(1)$, mentre que la de la dreta és una crida recursiva on la distància entre e i d s'ha dividit per 2. Per tant, la recurrència que descriu el cost d'aquesta funció és $C(n) = C(n/2) + \Theta(1)$. Si apliquem el teorema mestre per recurrències divisores del tipus $C(n) = a \cdot C(n/b) + \Theta(n^k)$, podem identificar $a = 1$, $b = 2$, $k = 0$ i calcular $\alpha = \log_2(1) = 0$. Com que $k = \alpha$, sabem que la solució és $C(n) = \Theta(n^k \log n) = \Theta(\log n)$.

Pel cas $\Theta(n)$ considerarem la crida recursiva

return $f(v, e, (e+d)/2) + f(v, (e+d)/2, d);$

En aquest cas, en ambdues crides recursives la distància entre e i d s'ha dividit per 2. Per tant, la recurrència que descriu el cost d'aquesta funció és $C(n) = 2C(n/2) + \Theta(1)$. Si apliquem el teorema mestre per recurrències divisores del tipus $C(n) = a \cdot C(n/b) + \Theta(n^k)$, podem identificar $a = 2$, $b = 2$, $k = 0$ i calcular $\alpha = \log_2(2) = 1$. Com que $\alpha > k$, sabem que la solució és $C(n) = \Theta(n^\alpha) = \Theta(n)$.

(b) Per analitzar el cas de la crida a *cerca* en cas pitjor, considerarem el cas en que s'efectuen les tres crides a funció. Notem que en totes tres, la distància entre els dos últims paràmetres és una tercera part de la distància original. Per analitzar el cost de la crida a *cerca2* veiem que és una funció no recursiva que, en cas pitjor, travessa els $n/3$ elements que hi ha entre e i d , fent un treball constant per a cadascun d'ells. Per tant el cost d'aquesta crida és $\Theta(n/3) = \Theta(n)$.

Per analitzar la crida a *cerca3*, hem de considerar que és una funció recursiva. Fixem-nos que totes les operacions que s'hi fan són de cost constant, però efectuem, en cas pitjor, una crida recursiva on la distància entre els paràmetres s'ha dividit per 2. Per tant, la recurrència que descriu el cost d'aquesta funció és $C(n) = C(n/2) + \Theta(1)$. Si apliquem el teorema mestre per recurrències divisores del tipus $C(n) = a \cdot C(n/b) + \Theta(n^k)$, podem identificar $a = 1$, $b = 2$, $k = 0$ i calcular $\alpha = \log_2(1) = 0$. Com que $k = \alpha$, sabem que la solució és $C(n) = \Theta(n^k \log n) = \Theta(\log n)$. Com que, en la crida a *cerca3* la distància entre els darrers paràmetres és $n/3$, el cost és $\Theta(\log(n/3)) = \Theta(\log n)$.

Finalment, hem de considerar la crida recursiva a *cerca*, on altra vegada, la distància entre els darrers paràmetres s'ha dividit per 3. Per tant, la recurrència que descriu el cost total de la crida a *cerca* és: $C(n) = C(n/3) + \Theta(\log n) + \Theta(n) = C(n/3) + \Theta(n)$. Si apliquem el teorema mestre per recurrències divisores del tipus $C(n) = a \cdot C(n/b) + \Theta(n^k)$, podem identificar $a = 1$, $b = 3$, $k = 1$ i calcular $\alpha = \log_3(1) = 0$. Com que $k > \alpha$, sabem que la solució és $C(n) = \Theta(n^k) = \Theta(n)$.

Proposta de solució al problema 2

Una possible solució és:

```

void to_Heap (Node* n, vector<int>& h) {
    if (not n) return;
    to_Heap(n->right,h);
    h.push_back(n->key);
    to_Heap(n->left,h);
}

vector<int> to_Heap (Node* n) {
    vector<int> h(1);
    to_Heap(n,h);
    return h;
}

```

Proposta de solució al problema 3

- a) **bool** es_torneig (**const** Graf& G) {
 int n = G.size ();
 for (**int** u = 0; u < n; ++u) {
 if (G[u][u]) **return** false;
 for (**int** v = u+1; v < n; ++v) {
 if (G[u][v] == G[v][u]) **return** false;
 }
 } **return** true;
}
- b) Base: És clar que un graf torneig amb un vèrtex o dos vèrtexs té un camí Hamiltonià. Inducció: Suposem que tot graf torneig amb n vèrtexs té un camí Hamiltonià. Considerem un graf torneig G amb $n + 1$ vèrtexs. Sigui u un vèrtex qualsevol de G . Llavors $G - u$ és un graf torneig de n vèrtexs i, per hipòtesi d'inducció, té un camí Hamiltonià v_1, v_2, \dots, v_n . Si (u, v_1) és un arc de G , llavors u, v_1, v_2, \dots, v_n és un camí Hamiltonià de G . Si no, (v_1, u) ha de ser un arc de G (perquè és torneig). Si (u, v_2) també és un arc de G , llavors v_1, u, v_2, \dots, v_n és un camí Hamiltonià de G . Si no, (v_2, u) ha de ser un arc de G (perquè és torneig). Continuant així fins a considerar v_n , arribem a que si (u, v_n) no és un arc de G llavors (v_n, u) ho ha de ser (perquè és torneig), i llavors v_1, \dots, v_n, u és un camí Hamiltonià de G .

- c) Una possible solució és la següent:

```

list <int> cami (const Graf& G) {
    int n = G.size ();
    list <int> L;
    if (n == 0) return L;
    if (n == 1) return {0};
    if (G[0][1]) L = {0, 1}; else L = {1, 0};
    for (int u = 2; u < n; ++u) insereix (L, u, G);
    return L;
}

void insereix (list <int>& L, int u, const Graf& G) {
    auto it1 = L.begin ();

```

```

    if (G[u][*it1]) { L.push_front(u); return; }
    auto it2 = it1; ++it2;
    while (it2 != L.end()) {
        if (G[*it1][u] and G[u][*it2]) { L.insert(it2, u); return; }
        ++it1; ++it2;
    }
    L.push_back(u);
}

```

Proposta de solució al problema 4

- (a) **No sabem** si aquesta afirmació és certa o falsa. Si fos certa, podríem demostrar que $P = NP$, que és un problema obert. En efecte, ja sabem que $P \subseteq NP$. Per veure l'altra inclusió, prenem un problema $C \in NP$ qualsevol i vegem que pertany a P . Com que B és NP -difícil i $C \in NP$, podem reduir C cap a B , i com que B es pot reduir cap a A , composant les dues reduccions sabem reduir C cap a A . L'algorisme que primer redueix C cap a A i a continuació resol A (en temps polinòmic) és un algorisme polinòmic pel problema C . Per tant $C \in P$.

Si fos falsa, aleshores per a qualsevol parell de problemes $A \in P$ i $B \in NP$ -difícil, no podríem reduir B cap a A . Si prenem B que sigui NP -complet (en particular NP -difícil), aleshores tindríem un problema $B \in NP$ que no es pot reduir a $A \in P$. Com tots els problemes de P es poden reduir entre ells, això implicaria que B no pertany a P i, per tant, que les classes P i NP no són iguals, i hauríem resolt un problema obert.

- (b) Aquesta afirmació és **certa**. Considerem A : determinar si un vector està ordenat i B : trobar un cicle hamiltonià en un graf. Com que $A \in P$, també pertany a NP . I com que B és NP -complet (en particular NP -difícil), segur que existeix una reducció d' A cap a B (per la definició de problema NP -difícil).
- (c) Aquesta afirmació és **certa**. Sigui A el problema del 3-colorejat de grafs i B el problema de trobar un cicle Hamiltonià en un graf. Són tots dos problemes NP -complets (i per tant, també NP -difícils). Com que sabem que tots els problemes NP -complets es poden reduir entre ells, l'afirmació és certa.