Cognoms **Nom** **DNI**

Examen Final EDA **Duració: 3h** **08/01/2024**

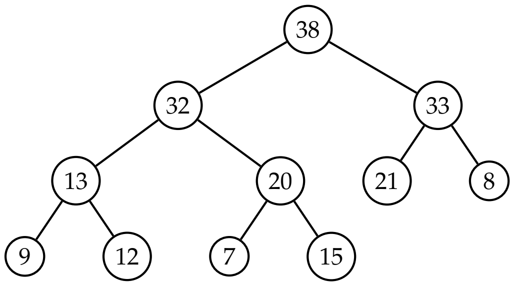
- *L'enunciat té 5 fulls, 9 cares, i 4 problemes.*
- *Poseu el vostre nom complet i número de DNI a cada full.*
- *Contesteu tots els problemes en el propi full de l'enunciat a l'espai reservat.*
- *Llevat que es digui el contrari, sempre que parlem de cost ens referim a cost asimptòtic en temps.*
- *Llevat que es digui el contrari,* **cal justificar les respostes.**

Problema 1 **(2 pts.)**

Responeu les preguntes següents:

(a) (1 pt.) Sabem que el cost d'un cert algorisme ve donat per la recurrència $T(n) = aT(n/4) + \Theta(n^2)$. Analitzeu el cost de l'algorisme en funció del paràmetre natural $a \geq 1$.

(b) (1 pt.) El max-heap de la figura següent és el resultat d'una seqüència d'operacions d'inserció i esborrat-del-màxim. La darrera operació va ser una inserció.



Expliqueu per què el 38 **no** pot ser l'últim element afegit.

Escriviu la llista dels elements que **sí** poden haver estat l'últim en ser afegit; aquesta llista no cal justificar-la.

Cognoms	Nom	DNI

Problema 2 **(3 pts.)**

Donat un vector v d'n nombres naturals, volem calcular un vector que contingui totes les parelles $<z,t>$ tal que el nombre z apareix a v exactament t vegades, amb $t > 0$. L'ordre de les parelles en el vector no ens importa. Per exemple, donat el vector $(4,1,5,1,3,4,5,1)$ un resultat correcte seria $(<3,1>, <5,2>, <1,3>, <4,2>)$.

(a) (1 pt.) Considereu el codi següent, que resol el problema plantejat:

```
vector<pair<int,int>> map_count (const vector<int>& v){
  map<int,int> m; // Podeu assumir que m és un AVL
  for (int x : v) ++m[x];
  vector<pair<int,int>> res;
  for (pair<int,int> p : m) res.push_back({p.first ,p.second});
  return res;
}
```

Quin és el cost en cas pitjor de la funció anterior en funció d'n? *Nota:* en tot aquest problema assumiu que el cost d'un *push_back* és $\Theta(1)$ i que recórrer els elements d'un *map* té cost lineal respecte el seu nombre d'elements. Us pot ser útil saber que $\log 1 + \log 2 + \cdots + \log n = \Theta(n \log n)$.

(b) (1 pt.) Assumim **(només en aquest apartat)** que tots els nombres de v són menors estrictes que 100 (que és un nombre fixat que no depèn d'n). Com aconseguiríeu resoldre el problema en temps $O(n)$ en cas pitjor? No cal que doneu codi, amb una explicació d'alt nivell n'hi ha prou. Tampoc cal que justifiqueu el cost.

(c) (1 pt.) Ompliu el codi següent per tal de resoldre el problema que tenim entre mans.

```
vector<pair<int,int>> priority (const vector<int>& v) {
  priority_queue<int> q;
  for (int x : v) q.push(x);
  vector<pair<int,int>> res;

  while (not q.empty()) {

  }
  return res; }
```

Cognoms **Nom** **DNI**

Problema 3 (2 pts.)

Disposem de dues gerres A i B amb capacitat $cap\_A > 0$ i $cap\_B > 0$ litres, respectivament. Tenim també una font per poder omplir les gerres. L'objectiu és aconseguir tenir exactament k litres en una de les gerres amb les següents operacions:

- Omplir una de les gerres fins a dalt.
- Buidar completament una de les gerres.
- Buidar el contingut d'una gerra origen cap una gerra destí fins que, o bé la gerra origen quedi buida, o bé la gerra destí quedi plena.

Ens demanen que calculem el mínim nombre d'operacions per aconseguir k litres en una de les gerres si comencem amb les gerres buides, o que indiquem que no és possible obtenir k litres. Per exemple, si $cap_A = 10$, $cap_B = 7$ i $k = 4$, ho podem fer amb 4 operacions:

1. Omplim la gerra B fins a dalt (A tindrà 0 litres, i B en tindrà 7).
2. Buidem la gerra B cap a A (A tindrà 7 litres i B estarà buida).
3. Omplim la gerra B fins a dalt (tant A com B tenen 7 litres).
4. Buidem la gerra B cap a A (A tindrà 10 litres, i B en tindrà 4).

Ompliu el codi següent per tal de resoldre aquest problema. *Pista:* Fixeu-vos que hi ha $(cap_A + 1) \times (cap_B + 1)$ possibles estats. No esperem una solució per *backtracking*.

```
int cap_A, cap_B; // variables globals
int operacions(const pair<int,int>& ini, int k);
int main(){
  int k;
  cin >> cap_A >> cap_B >> k;
  pair<int,int> ini = {0,0}; // un parell es (litres_dins_A, litres_dins_B)
  int res = operacions(ini,k);
  if (res == -1) cout << "No es poden aconseguir " << k << " litres." << endl;
  else cout << "Necessitem " << res << " operacions." << endl; }

// Omplir A
pair<int,int> mov_1(const pair<int,int>& p) {return {cap_A, p.second};}
// Omplir B
pair<int,int> mov_2(const pair<int,int>& p) {return {p.first, cap_B};}

// Buidar A
pair<int,int> mov_3(const pair<int,int>& p) {return {0, p.second};}
// Buidar B
pair<int,int> mov_4(const pair<int,int>& p) {return {p.first, 0};}
```

```
pair<int,int> mov_5 (const pair<int,int>& p) { // A cap a B

}

pair<int,int> mov_6 (const pair<int,int>& p) { // B cap a A

}

vector<pair<int,int>> un_pas (const pair<int,int>& p) {
  return {mov_1(p), mov_2(p), mov_3(p), mov_4(p), mov_5(p), mov_6(p)}; }

// retorna el mínim nombre d'operacions o -1 si no hi ha solució
int operacions (const pair<int,int>& ini, int k) {

}
```

Cognoms **Nom** **DNI**

Problema 4 (3 pts.)

Donat un graf $G = (V, E)$ no dirigit, el problema **3-COL** consisteix en determinar si existeix una funció $c : V \to \{1, 2, 3\}$ de manera que per a tota aresta $\{u, v\} \in E$ es compleixi $c(u) \neq c(v)$.

Donat un graf $G = (V, E)$ no dirigit amb $|V| = 3n$, el problema **3-COL-EQUIL** consisteix en determinar si existeix una funció $c : V \to \{1, 2, 3\}$ de manera que per a tota aresta $\{u, v\} \in E$ es compleixi $c(u) \neq c(v)$ i tal que la cardinalitat dels conjunts $C_i = \{v \in V \mid c(v) = i\}$ sigui n per a tota $1 \leq i \leq 3$.

(a) (0.5 pts.) Construïu una instància positiva de **3-COL** amb 6 vèrtexs que sigui una instància negativa de **3-COL-EQUIL**. Justifiqueu per què aquesta instància compleix el que es demana.

(b) (1.25 pts.) Demostreu que **3-COL-EQUIL** pertany a la classe NP.

(c) (1.25 pts.) Donat $G = (V, E)$ una instància de **3-COL** amb $V = \{v_1, v_2, \cdots, v_n\}$ podem generar una instància $G' = (V', E')$ de **3-COL-EQUIL** de la manera següent:

- $V' = \{v_1^1, v_1^2, v_1^3, \quad v_2^1, v_2^2, v_2^3, \quad \cdots, v_n^1, v_n^2, v_n^3\}$
- $E' = \{\{v_i^k, v_j^k\} \mid \{v_i, v_j\} \in E \text{ i } 1 \leq k \leq 3\}$

és a dir, creem 3 còpies de G.

Demostreu que la funció anterior és un reducció polinòmica de **3-COL** cap a **3-COL-EQUIL**.

Cognoms **Nom** **DNI**

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

Proposta de solució al problema 1

(a) Si apliquem el teorema mestre de recurrències divisores de l'estil $T(n) = aT(n/b) + \Theta(n^k)$, identifiquem $b = 4$ i $k = 2$. Aleshores, necessitem comparar $\alpha = \log_4(a)$ amb $k = 2$. Els dos valors coincidiran quan $\log_4(a) = 2$, és a dir, quan $a = 16$. Aleshores:

- Si $a = 16$, tenim $\alpha = k$ i per tant, $T(n) = n^k \log n = n^2 \log n$.
- Si $a < 16$, tenim $\alpha < k$ i per tant, $T(n) = n^k = n^2$.
- Si $a > 16$, tenim $\alpha > k$ i per tant, $T(n) = n^\alpha = n^{\log_4(a)}$.

(b) El 38 no pot ser l'últim element afegit perquè això implicaria que 32 era l'anterior arrel, però això és impossible perquè és menor que 33 (fill dret de l'arrel).

La llista dels elements que poden ser l'últim afegit és $15, 20, 32$.

Proposta de solució al problema 2

(a) Recordem que un map en C++ equival essencialment a un AVL i, per tant, les operacions d'afegir un parell $(clau, informacio)$, esborrar una clau o modificar la informació associada a una clau tenen cost $\log m$ si m és el nombre d'elements que hi ha al map.

La creació del map buit té cost $\Theta(1)$. A continuació, hi ha un bucle que itera sobre els n elements x de v. Per cada x, si no apareix al map, l'afegeix com a clau amb informació 1 i en cas contrari n'incrementa la seva informació. En la primera d'aquestes operacions el map té mida com a molt 0, en la segona com a molt 1, i així successivament fins a la darrera operació on el map tindrà com a molt $n-1$ elements. Com que, en cas pitjor, cada operació és logarítmica en la mida del map, el cost total és com a molt $\log(1) + \log(2) + \cdots + \log(n-1) = \Theta(n \log n)$.

A continuació iterem sobre els elements del map, que conté com a molt n elements. Per tant ho podem fer amb temps com a molt $\Theta(n)$. Cada element l'afegim al vector *res* amb temps constant. Per tant el segon bucle té cost com a molt $\Theta(n)$.

Finalment, retornar el vector *res*, que té mida com a molt n, ens costarà com molt $\Theta(n)$.

Per tant, el cost total en cas pitjor és $\Theta(n \log n) + \Theta(n) + \Theta(n) = \Theta(n \log n)$.

(b) La idea és, enlloc d'utilitzar un map, utilitzar un vector m de mida 100 on a la posició i hi guardarem el nombre d'aparicions del nombre i. El vector l'inicialitzarem a 0. El primer bucle no canvia en absolut.

El segon bucle es reemplaçarà per:

```
for (int i = 0; i < 100; ++i)
  if (m[i] > 0) res.push_back({i, m[i]});
```

(c) Una possible solució és:

```
vector<pair<int,int>> priority (const vector<int>& v) {
   priority_queue <int> q;
   for (int x : v) q.push(x);
   vector<pair<int,int>> res;
   int current = −1;
   while (not q.empty()) {
      int x = q.top ();
      q.pop ();
      if (x ≠ current) {
         current = x;
         res.push_back({current ,1});
      }
      else ++res.back (). second;
   }
   return res;
}
```

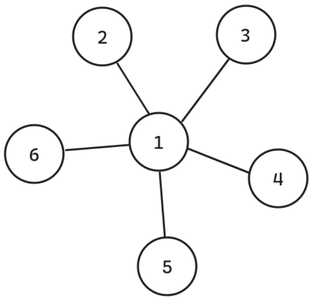
Proposta de solució al problema 3

```
pair<int,int> mov_5 (const pair<int,int>& p) { // A cap a B
   if (p. first + p.second ≤ cap_B) return {0,p. first +p.second};
   else return {p. first + p.second − cap_B,cap_B};
}

pair<int,int> mov_6 (const pair<int,int>& p) { // B cap a A
   if (p. first + p.second ≤ cap_A) return {p. first +p.second,0};
   else return {cap_A, p. first + p.second − cap_A};
}

int operacions (const pair<int,int>& ini, int k) {
   vector<vector<int>> dist(cap_A + 1, vector<int>(cap_B + 1,INF));
   queue<pair<int,int>> Q;

   Q.push(ini);
   dist [ ini . first ][ ini .second] = 0;

   while (not Q.empty()) {
      pair<int,int> u = Q.front ();
      Q.pop();
      vector<pair<int,int>> veins = un_pas(u);
      for (pair<int,int> v : veins) {
         if (dist[v. first ][v.second] == INF) {
            dist[v. first ][v.second] = dist[u. first ][u.second] + 1;
            if (v. first == k or v.second == k) return dist[v. first ][v.second];
            Q.push(v);
         }
```

```
        }
    }
    return −1;
}
```

Proposta de solució al problema 4

(a) Prenem el graf següent:



Clarament és una instància positiva de **3-COL**: es pot colorejar amb 3 colors pintant, per exemple, el node 1 amb el color 1 i els altres nodes amb el color 2.

També podem veure que és una instància negativa de **3-COL-EQUIL**: com que tenim 6 nodes, cada color s'hauria d'utilitzar dues vegades. El node central (1) s'ha de pintar amb algun color, però no podem utilitzar aquest color per cap altre node perquè estan tots connectats amb 1. Per tant, no existeix una coloració com la que busquem.

(b) Prenem com a conjunt de testimonis totes les coloracions possibles, és a dir, les funcions $c : V \to \{1,2,3\}$. Donada una instància $G = (V, E)$ amb n vèrtexs i m arestes (i per tant, mida $\Theta(n + m)$ si el graf ve representat amb llistes d'adjacència), clarament qualsevol testimoni és polinòmic respecte la mida de G, perquè necessitem com a molt $2n$ bits per representar una coloració.

Com a verificador prenem el següent algorisme: donat un graf $G = (V, E)$ i un testimoni (coloració) c:

1.- Per a tota aresta $\{u, v\} \in E$, si $c(u) = c(v)$ retornem 0.

2.- Calculem n_1, n_2, n_3 el nombre vèrtexs amb color 1,2 i 3, respectivament. Si algun d'ells no és n, retornem 0.

3.- Retornem 1.

Hem de comprovar que el verificador és polinòmic. En el pas 1, amb un graf representat amb llistes d'adjacència, podem recórrer les arestes en temps $\Theta(n + m)$ i, per cada una d'elles, fem un treball $\Theta(1)$. Per tant, el pas 1 té cost $\Theta(n + m)$. Pel pas 2, només cal recórrer els n vèrtexs i actualitzar n_1, n_2, n_3, cosa que clarament es pot fer en temps $\Theta(n)$. Per tant, el cost del verificador és polinòmic.

Sigui ara G una instància positiva. Aleshores, per definició del problema, existeix una coloració c tal que pinta els vèrtexs de cada aresta amb colors diferents i tals que $|C_i| = n$ per a $1 \leq i \leq n$. Si prenem aquesta coloració com a testimoni, veiem que el verificador no pot acabar en el pas 1 perquè totes les arestes estan colorejades correctament, i tampoc en el pas 2, perquè precisament $n_i = |C_i|$. Així doncs, existeix un testimoni pel qual el verificador retorna 1.

Sigui ara G una instància negativa. Aleshores, per definició del problema, si agafem una coloració qualsevol, o bé colorejarà els dos vèrtexs d'alguna aresta amb el mateix color o bé la cardinalitat d'algun C_i no serà n. Així doncs, si agafem un testimoni qualsevol (una coloració), el verificador retornarà 0 en el pas 1 o en el pas 2, depenent del cas. Per tant, sempre retornarà 0.

(c) Clarament, $|V'| = 3n$. A més, si la mida de G és $n + m$, on m és $|E|$, la mida de G' és $3n + 3m$ ($3n$ vèrtexs i $3m$ arestes), que clarament és polinòmica respecte $n + m$. A més, també es pot calcular en temps polinòmic, ja que només fem 3 còpies de G.

Anem a veure ara que, si G és una instància positiva, G' també ho és. Efectivament, si G és una instància positiva, sigui c una coloració correcta per G. Podem construir una coloració c' que respecta les arestes de G' i té n vèrtexs de cada color de la manera següent:

- $c'(v_i^1) = c(v_i)$ per a tot $1 \leq i \leq n$
- Per a tot $1 \leq i \leq n$:

$$c'(v_i^2) = \begin{cases} 2 & \text{si } c(v_i) = 1 \\ 3 & \text{si } c(v_i) = 2 \\ 1 & \text{si } c(v_i) = 3 \end{cases}$$

- Per a tot $1 \leq i \leq n$:

$$c'(v_i^3) = \begin{cases} 3 & \text{si } c(v_i) = 1 \\ 1 & \text{si } c(v_i) = 2 \\ 2 & \text{si } c(v_i) = 3 \end{cases}$$

És fàcil veure que si $c(v_i) \neq c(v_j)$, aleshores $c'(v_i^k) \neq c'(v_j^k)$. Així doncs, si prenem una aresta de la forma $\{v_i^k, v_j^k\}$, com que $\{v_i, v_j\} \in E$, necessàriament $c(v_i) \neq c(v_j)$ i per tant c' coloreja els dos vèrtexs de l'aresta de color diferent.

Finalment, només cal veure que c' coloreja exactament n vèrtexs amb cada color. Això és fàcil de veure si observem que per a cada vèrtex $v_i \in G$, els 3 vèrtexs v_i^1, v_i^2, v_i^3 estan colorejats a c' amb 3 colors diferents. Per tant, com que cada color apareix exactament una vegada a cada tripleta $(c(v_i^1), c(v_i^2), c(v_i^3))$, si construïm totes les tripletes d'aquest tipus podrem veure que cada color es fa servir exactament n vegades (una vegada a cada tripleta).

Per acabar de demostrar que és una reducció correcta, assumim que G' és una instància positiva de **3-COL-EQUIL** i veiem que G és una instància positiva de **3-COL**. Això és fàcil de veure si observem que G' conté 3 còpies de G, totes elles ben colorejades. Per tant, podem colorejat G amb els colors utilitzats a la primera còpia de G.

Cognoms **Nom** **DNI**

Examen Final EDA **Duració: 3h** **12/06/2023**

- *L'enunciat té 4 fulls, 8 cares, i 4 problemes.*
- *Poseu el vostre nom complet i número de DNI a cada full.*
- *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 pts.)

Responeu 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 $cerca(x, v, 0, v.size() - 1)$?

Cognoms **Nom** **DNI**

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.

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

Cognoms **Nom** **DNI**

Problema 3 **(3 pts.)**

En aquest problema representarem un graf dirigit amb nodes $\{0, 1, \cdots, n-1\}$ 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 $\Theta(n^2)$ 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.

Cognoms **Nom** **DNI**

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.

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 cos 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 *cerca*2 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 *cerca*3, 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 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 *cerca*3 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 $v_1, v_2, \ldots, v_n$. Si (u, v_1) és un arc de G, llavors $u, v_1, v_2, \ldots, v_n$ 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 $v_1, u, v_2, \ldots, v_n$ 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 $v_1, \ldots, v_n, u$ 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.

Cognoms **Nom** **DNI**

Examen Final EDA **Duració: 3h** **16/01/2023**

- *L'enunciat té 4 fulls, 8 cares, i 4 problemes.*
- *Poseu el vostre nom complet i número de DNI a cada full.*
- *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 pts.)**

Responeu 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);
  }
}
```

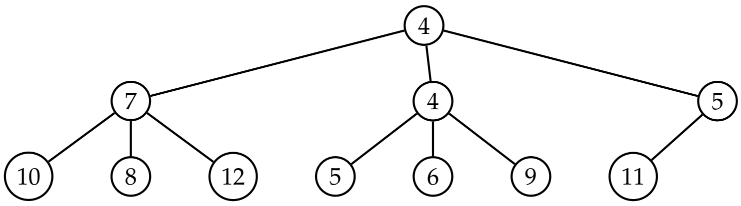
Què calcula aquesta funció? Justifica formalment la teva resposta.

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

Cognoms **Nom** **DNI**

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 $1 + 3 + 3^2 + \ldots + 3^h = \frac{3^{h+1}-1}{2}$.

(b) (0.75 pts.) Quin és el mínim nombre de nodes que un min-heap ternari d'alçada h pot tenir? Utilizeu 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]) {

    } } }
```

Cognoms **Nom** **DNI**

Problema 3 (2 pts.)

Com ja sabem, donat un conjunt finit de variables $\{x_1, x_2, \ldots, x_n\}$, 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 $\{1, 2, \ldots, n\}$.

Fórmula:

$$
\begin{array}{l}
(x_1 \vee \neg x_2) \wedge \\
(x_2 \vee \neg x_3 \vee x_1) \wedge \\
(x_3) \wedge \\
(x_2 \vee x_3)
\end{array}
$$

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
  } }
```

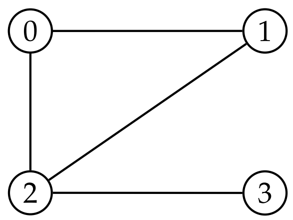
(b) (0.5 pts.) En funció d'n, quantes vegades es crida la funció *evaluate* en el cas pitjor? I en el cas millor?

Cognoms **Nom** **DNI**

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 $c : V \to \{1, 2, \ldots, k\}$ de manera que per a tota aresta $\{u, v\} \in E$ es compleixi $c(u) \neq c(v)$.

(a) (0.7 pts.). Ens asseguren 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, `v` 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.



```
-x(0,1) v -x(1,1)
-x(0,1) v -x(2,1)
-x(1,1) v -x(2,1)
-x(2,1) v -x(3,1)

-x(0,2) v -x(1,2)
-x(0,2) v -x(2,2)
-x(1,2) v -x(2,2)
-x(2,2) v -x(3,2)

-x(0,3) v -x(1,3)
-x(0,3) v -x(2,3)
-x(1,3) v -x(2,3)
-x(2,3) v -x(3,3)
```

De forma més precisa, si G és un graf amb n vèrtexs $\{0, 1, 2, \ldots, n-1\}$ 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 prodeciment 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 vértexs 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 **C**erta o **F**alsa. Cada resposta correcta suma 0.4 punts. Cada resposta incorrecta resta 0.4. Les respostes en blanc no compten.

1	2	3	4

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 veureho, 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 $v = (x_1, x_2, \ldots, x_{2^k})$ la funció construeix aux, un vector de mida 2^{k-1} amb els elements $(\ (x_1 + x_2)/2,\ \ (x_3 + x_4)/2,\ \ \ldots,\ \ (x_{2^k-1} + x_{2^k})/2\ )$. Per tant, la hipòtesis d'inducció ens garanteix que la crida recursiva retornarà la mitjana d'aquest conjunt:

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

que és igual a

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

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: $1 + 3 + \ldots + 3^{h-1} = \frac{3^h-1}{2}$. Aleshores

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

(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 $1+3+\ldots+3^{h-1}+1$, 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 $alpha$ i, per cada una d'elles crida a la funció *evaluate*. Com que hi ha 2^n possibles $alpha$, 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 $alpha$ té mida $(n+1)-k$.

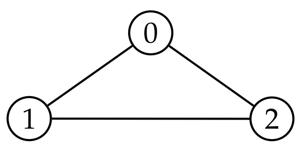
És fàcil veure que $C(0) = 1$, perquè en aquest cas $alpha$ tindrà mida $n+1$ i estarem en el cas base de SAT, que només fa una crida a *evaluate*.

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

És fàcil veure que aquesta recurrència té solució $C(k) = 2^k$. Quan fem la crida a SAT des del main, $alpha$ 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 negatives. Considerem, per exemple, el graf



Clarament aquesta és una instància negativa de $\mathbf{2-COL}$. No obstant, la funció *reduccio* ens construeix una fórmula on totes les variables apareixes 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 $\mathbf{2-COL}$ 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) v x(i,2) v x(i,3) v ... v 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

Cognoms **Nom** **DNI**

Examen Final EDA — Duració: 3h — 10/01/2022

- *L'enunciat té 4 fulls, 8 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.)

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));
}
```

Cognoms **Nom** **DNI**

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.

Cognoms **Nom** **DNI**

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?

Cognoms Nom DNI

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.) Demostra 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.

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

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 $n + (n-1) + \cdots + 1 = \Theta(n^2)$ 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 $1 + 2 + \cdots + (n-i) = \Theta((n-i)^2)$. 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 $j_1 \to j_2$ i $j_2 \to j_3$. 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 $\{v_c \mid c \in \mathcal{C}\}$ i arestes $\{\{v_c, v_d\} \mid d \in \mathcal{C}, c \in I_d\}$. 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 $\{c_s \mid s \in S\}$, 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 $\Theta(\log i)$.

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 rebi 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 interessen, 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 interessen 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 $c : V \to \{1, 2, \cdots, k\}$ 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 $V = \{v_1, v_2, \cdots, v_n\}$ i arestes $E = \{e_0, e_1, \cdots, e_{m-1}\}$, construïm la següent instància (N, p) de DISTINCT-ONES:

- $N = \{x_1, x_2, \cdots, x_n\}$, 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.
- $p = k$.

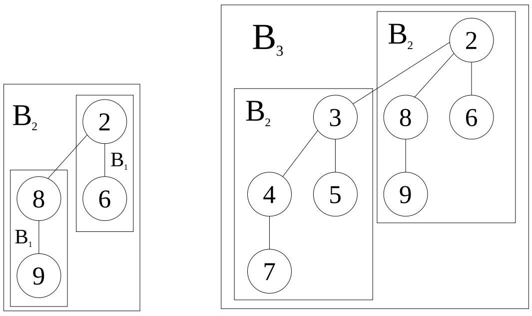
Cognoms Nom DNI

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.

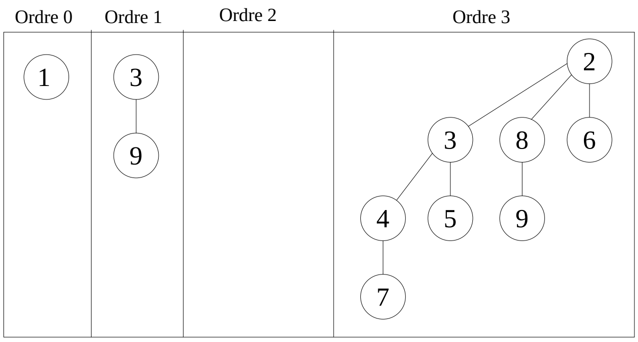


(a) (0.5 pts.) Quants nodes té un arbre B_k? Demostra-ho formalment.

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:

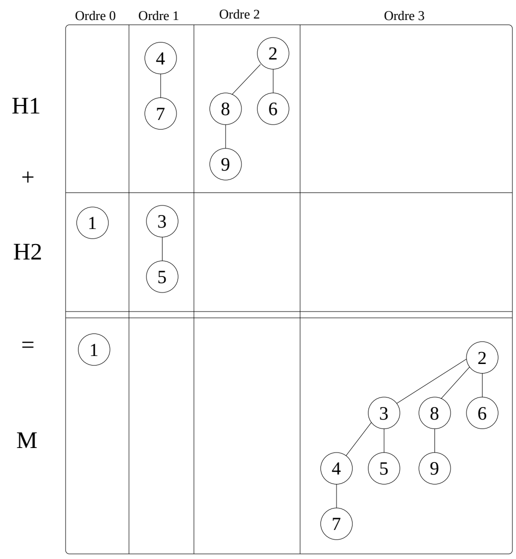


(b) (0.75 pts.) Volem construir un heap binomial amb n nodes. Quants arbres binomials de cada ordre tindrà? Exemple: si $n = 11$ tindrem un arbre binomial d'ordre 0, un d'ordre 1 i un d'ordre 3 (fixeu-vos en la figura superior).

(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 $\Theta(\log n)$.

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:

$$\textbf{if} \;\; (\textit{partial\_sum} \; + \textit{remaining\_sum} < \textit{money}) \; \textbf{return};$$

Proposta de solució al problema 3

(a) Considerem primer la representació dels nombres de N:

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

Aleshores veiem que 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. Per tant, és una instància positiva.

(b) El conjunt de candidats a testimonis el formaran totes les possibles maneres que tenim de distribuir els elements de N en p conjunts. Observem que la mida d'un candidat a testimoni és polinòmica respecte la mida de la instància. De fet, és lineal.

El verificador rebrá un parell (N, p) i p conjunts $S_1, S_2, \cdots, S_p$ on s'han distribuit 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.

A més, 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 coindideix amb el nombre d'arestes de G.

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

$(G, k) \in$ COLORABILITAT $\Rightarrow$ $(N, p) \in$ 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.

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

Si (N, p) és una instància positiva és perquè podem agrupar els nombres de N en p conjunts $S_1, S_2, \cdots, S_p$ 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 $\Theta(\log n)$.