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) 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 rebrà 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 coincideix 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 $\Theta(\log n)$.