Solució de l'Examen Parcial EDA 08/11/2021

Proposta de solució al problema 1

(a) La funció *mystery* determina si n és un nombre primer.

Pel que fa al seu cost, fixem-nos que les tres primeres línies del codi tenen cost constant perquè només fan operacions aritmètiques i comparacions entre enters. La part interessant del codi és el bucle. En el cas pitjor es fan totes les voltes al bucle. Per facilitar el raonament podem assumir que el bucle comença a $i = 1$ (afegir dues voltes al bucle no canvia el cost asimptòtic). Com que parem quan $i^2 > n$, és a dir $i > \sqrt{n}$, es farien $\sqrt{n}$ voltes si i s'incrementés en una unitat a cada volta. Com que s'incrementa en dos, se'n fan la meitat: $\sqrt{n}/2$ voltes. Si considerem que cada volta només fa un treball constant (assignacions, operacions aritmètiques i comparacions entre enters), el cost total en cas pitjor és $\Theta(\sqrt{n})$.

(b) El seu cost és $\Theta(n\sqrt{n}) = \Theta(n^{3/2})$.

Les dues primeres línies tenen cost constant. Anem a estudiar el bucle. El primer fet a observar és que el bucle més intern (el que varia k) té cost $\Theta(n)$. El bucle més extern fa n voltes. Per algunes d'elles s'executarà el bucle intern i per les altres es farà un treball constant. Anem a comptar quantes vegades s'executa el bucle intern. Aquest només s'executa quan $i = j^2$. La variable j inicialment val zero, i cada vegada que s'executa el bucle intern s'incrementa en una unitat. Ens podem fixar que el bucle intern s'executarà per primera vegada quan $i = j^2 = 0$, i llavors j passarà a valer 1. La segona vegada serà quan $i = j^2 = 1$ i llavors j passarà a valer 2. La tercera vegada serà quan $i = j^2 = 4$, i així successivament. Per tant, el bucle intern s'executarà tantes vegades com quadrats hi hagi entre 0 i $n-1$, és a dir, $\lceil\sqrt{n}\rceil$. La resta de vegades, és a dir, $(n - \lceil\sqrt{n}\rceil)$ vegades, el bucle intern no s'executa. Així doncs el cost total és $(n - \lceil\sqrt{n}\rceil) \cdot \Theta(1) + \lceil\sqrt{n}\rceil \cdot \Theta(n) = \Theta(n \cdot \sqrt{n})$.

(c) Ens fixem que *pairs* és una funció recursiva on a cada crida recursiva decreix el nombre d'elements en $v[l \ldots r]$. Inicialment aquest nombre és n. Si ens centrem en el cas recursiu, podem veure que hi ha dues crides recursives on el nombre d'elements és la meitat, una sèrie d'instruccions de cost constant i dos bucles ennierats. El bucle extern tracta la part esquerra del vector, entre l i m, on hi ha essencialment $n/2$ elements. Per cadascun d'aquests elements, el bucle intern recorre la part dreta del vector, entre $m+1$ i r, on també tenim aproximadament $n/2$ elements. Tot plegat, el condicional de dins el bucle s'executa bàsicament $n^2/4$ vegades, que equival a un treball $\Theta(n^2)$. Per tant, la recurrència que descriu el cost d'aquesta funció és

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

que té solució $T(n) \in \Theta(n^2)$.

(d) Com que K no depèn d'n, la primera línia del codi té cost constant, així com la segona. El primer bucle té cost $\Theta(n)$. Només ens falta analitzar el segon bucle, que repeteix K vegades un treball constant. Com que K també és constant, el bucle triga $\Theta(1)$. Tot plegat el cost de la funció és $\Theta(n)$.

La correcció del codi es basa en la següent observació. El primer bucle calcula, per cada nombre k entre 0 i $K-1$, quantes vegades apareix k a v. Aquest valor

es guarda a $times[k]$. Si un nombre k apareix p vegades, aleshores hi haurà exactament $p \cdot (p-1)/2$ parelles (i, j) amb $0 \leq i < j < n$ tals que $v[i] = v[j] = k$. Com que suma sobre totes les k possibles, el segon bucle ens calcula la quantitat desitjada.

Proposta de solució al problema 2

(a)
```
int first_occurrence (int x, const vector<int>& v, int l, int r) {
  if (l > r) return -1;
  else {
    int m = (l+r)/2;
    if      (v[m] < x) return first_occurrence (x,v,m+1,r);
    else if (v[m] > x) return first_occurrence (x,v,l,m-1);
    else if (m == l or v[m-1] ≠ x) return m;
    else    return first_occurrence (x,v,l,m-1);
  }
}
```

(b) El codi a omplir és $res = n - q - p$;

Per analitzar el seu cost, veiem que el codi comença amb una crida a *first_occurrence* de cost, en cas pitjor, $O(\log n)$. A continuació la propera part interessant és el primer bucle, de cost $\Theta(n)$. El segon bucle, tot i que només visita la meitat dels elements de v, té el mateix cost $\Theta(n)$, perquè sabem que un *swap* triga temps $\Theta(1)$. Finalment, tenim una altra crida a *first_occurrence*, altra vegada de cost, en cas pitjor, $O(\log n)$. Tot plegat el cost del codi és $\Theta(n)$.

Per tal d'entendre per què el codi és correcte, hem d'entendre que essencialment aquest intenta calcular la primera i la darrera aparició d'x a v, i calcular quants elements hi ha entre aquestes dues posicions. El punt clau és adonar-se que si invertim el vector i neguem tots els seus elements, obtenim un vector ordenat creixentment tal que l'última aparició d'x en el v original coincideix amb la primera aparició de $-x$ en el nou v. A més, si la primera aparició de $-x$ en el nou vector v és a la posició q, aleshores la darrera aparició d'x en el vector v original és $n - 1 - q$. Per tant, el nombre d'elements entre p (primera aparició) i $n - 1 - q$ (última aparició) és $(n - 1 - q) - p + 1 = n - q - p$.

(c) Podem aconseguir fàcilment un algorisme de cost, en cas pitjor, $O(\log n)$ si calculem millor la darrera aparició d'x a v. Per tal de fer-ho, podem modificar el codi de $first\_ocurrence$ lleugerament:

```
int last_occurrence (int x, const vector<int>& v, int l, int r) {
  if (l > r) return -1;
  else {
    int m = (l+r)/2;
    if      (v[m] < x) return last_occurrence (x,v,m+1,r);
    else if (v[m] > x) return last_occurrence (x,v,l,m-1);
    else if (m == r or v[m+1] ≠ x) return m;
    else    return last_occurrence (x,v,m+1,r);
```

```
    }
}
```

Una crida a *last_occurrence* té cost en cas pitjor $O(\log n)$. Aleshores, només ens caldrà trobar la primera i la darrera aparició d'x, diguem-ne p i q, i retornar $q - p + 1$ o bé 0 si no hi ha cap aparició.