Solució de l'Examen Parcial EDA 06/11/2020

Proposta de solució al problema 1

(a)

```
bool tri_search (const vector<int>& v, int l, int r, int x) {
  if (l > r) return false;
  else {
    int n_elems = (r−l+1);
    int f = l + n_elems/3;
    int s = r − n_elems/3;
    if (v[f] == x or v[s] == x) return true;
    if (x < v[f]) return tri_search (v,l,f−1,x);
    if (x < v[s]) return tri_search (v,f+1,s−1,x);
    else return tri_search (v,s+1,r,x);
  }
}
```

En el cas pitjor es fan totes les crides recursives fins que $l > r$. La recurrència que expressa el cost del programa en aquest cas és:

$T(n) = T(n/3) + \Theta(1)$

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

(b) Siguin $f = n(\log n)^{1/2}$ i $g = n(\log n)^{1/3}$.

Per veure que són $\Omega(n)$ i no $\Theta(n)$ només cal veure que els límits següents són infinit:

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

$$\lim_{x\to\infty} \frac{n(\log n)^{1/3}}{n} = \lim_{x\to\infty} (\log n)^{1/3} = \infty$$

Per veure que són $O(n \log n)$ i no $\Theta(n \log n)$ només cal veure que els límits següents són zero:

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

$$\lim_{x\to\infty} \frac{n(\log n)^{1/3}}{n \log n} = \lim_{x\to\infty} \frac{(\log n)^{1/3}}{\log n} = \lim_{x\to\infty} \frac{1}{(\log n)^{2/3}} = 0$$

Finalment per veure que $f \notin \Theta(g)$, vegem que el límit següent no és una constant major estricta que zero:

$$\lim_{x\to\infty} \frac{n(\log n)^{1/3}}{n(\log n)^{1/2}} = \lim_{x\to\infty} \frac{(\log n)^{1/3}}{(\log n)^{1/2}} = \lim_{x\to\infty} \frac{1}{(\log n)^{1/6}} = 0$$

Proposta de solució al problema 2

(a) La idea d'aquest algorisme és que, cada vegada que trobem un natural es compten les aparicions posteriors d'aquest en el vector i es marquen amb un -1 per a no considerar-les més en el futur. Per tant, quan visitem un element marcat amb un -1 ens estalviem el bucle més intern.

Si construïm un vector on tots els nombres són diferents, aleshores aquesta optimització no serveix per a res. A més, si tots els nombres són diferents no tenim cap element dominant (a no ser que $n = 1$) i els dos bucles s'executen el màxim nombre de vegades. El cos del bucle més intern és clarament constant, pel que només hem de comptar quantes vegades s'executa. Donat una i concreta, el bucle intern s'executa $n - i$ vegades. Com que i va des de 0 fins a $n - 1$, el cost total és $n + (n - 1) + (n - 2) + \cdots + 1 = \Theta(n^2)$.

El cost en cas millor es dóna, per exemple, quan tenim un vector amb un únic element repetit n vegades. En aquest cas, quan $i = 0$ visitarem tots els elements del vector marcant-los amb un -1. Per a totes les altres i, el bucle més intern no s'executarà. Per tant, el cost en cas millor és $\Theta(n)$.

Si ens asseguren que tenim com a molt 100 naturals diferents, aleshores el bucle intern s'executarà com a molt 100 vegades. És a dir, hi haurà com a molt 100 is per les quals el bucle intern s'executarà. Aquestes is contribuiran en el cas pitjor un cost de $\Theta(100n) = \Theta(n)$. Per la resta de les is (en tenim com a màxim n) el bucle intern no s'executarà i per tant, contribuiran amb un cost de $\Theta(n)$. Així doncs, el cost en cas pitjor ha canviat i ha passat a ser $\Theta(n)$.

(b) Per aquest exercici primer recordem que la ordenació per inserció té cost $\Theta(n^2)$ en cas pitjor i $\Theta(n)$ en cas millor. Pel que fa al *quicksort*, tenim un cost de $\Theta(n^2)$ en cas pitjor i $\Theta(n \log n)$ en cas millor. Per analitzar el cost de *dominant_sort*, oblidem-nos de moment de la crida a *own_sort*. La resta del bucle veiem que com a màxim visita cada element del vector una vegada, fent-hi un treball constant. Cal remarcar que a vegades no visita tots els elements, ja que s'atura quan detecta l'element dominant. El cas pitjor el tenim quan visita tots els elements i no troba cap dominant (això triga $\Theta(n)$). El cas millor es dóna quan el primer element que visita és el dominant, però podem observar que per a detectar que és dominant ha de visitar almenys $n/2$ elements, pel que el cost també és $\Theta(n)$. Per tant, el codi sempre triga $\Theta(n)$ (obviant la crida a *own_sort*).

Si *own_sort* és una ordenació per inserció, el cost en cas millor és $\Theta(n) + \Theta(n) = \Theta(n)$, i en cas pitjor és $\Theta(n) + \Theta(n^2) = \Theta(n^2)$.

Si *own_sort* és un *quicksort*, el cost en cas millor és $\Theta(n) + \Theta(n \log n) = \Theta(n \log n)$, i en cas pitjor és $\Theta(n) + \Theta(n^2) = \Theta(n^2)$.

(c) El codi complet és:

```
int dominant_divide (const vector<int>& v, int l, int r) {
  if (l == r) return v[l];
  int n_elems = (r−l+1);
  int m = (l+r)/2;
  int maj_left  = dominant_divide(v, l, m);
  if ( maj_left ≠ −1 and times(v, l, r, maj_left ) > n_elems/2) return maj_left ;
  int maj_right = dominant_divide(v, m+1, r);
```

if ($maj\_right \neq -1$ **and** $times(v, l, r, maj\_right) > n\_elems/2$) **return** maj_right;
return -1; }

Per analitzar el seu cost ens adonem que en el cas pitjor es fan dues crides recursives de tamany la meitat i dues crides a *times*. La resta del codi té cost constant. Observem ara que la funció *times* rep v per referència i per tant el seu cost es pot descriure per $T(n) = T(n-1) + \Theta(1)$, que té solució $T(n) \in \Theta(n)$. Així doncs, la recurrència que descriu els cost en cas pitjor d'aquest programa és

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

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