Solució de l'Examen Parcial EDA 02/11/2023

Proposta de solució al problema 1

(a)

2	3	5	10	1	6	7	13	Mergesort
2	3	5	10	7	13	1	6	Inserció
6	2	5	3	7	1	13	10	Quicksort
1	2	3	5	6	13	10	7	Selecció

Nota: Al vostre examen, l'ordre de les files de taula podria ser diferent.

(b) Calculem el límit:

$$\lim_{n\to\infty} \frac{(\log_2 \log_2 n)^2}{\log_2 n}$$

Per a fer-ho, apliquem un canvi de variable $n = 2^m$, i per tant $\log_2 n = m$. El límit anterior és el mateix que:

$$\lim_{m\to\infty} \frac{(\log_2 m)^2}{m} = 0$$

Per tant, podem afirmar que $(\log_2 \log_2 n)^2$ és menor asimptóticament que $\log_2 n$. És a dir, $(\log_2 \log_2 n)^2 \in O(\log n)$ però $(\log_2 \log_2 n)^2 \notin \Omega(\log n)$.

(c) L'algorisme començarà sobre l'arrel, i a cada node, es mourà cap a un dels seus fills mentre puguem descartar que el node actual no sigui l'ancestre comú més petit. Donat un node amb clau z, considerarem els següents casos:

- Si $z = x$, aleshores y és un descendent del node actual, i per tant, el node actual és l'ancestre comú més petit.
- Si $z = y$, s'aplica l'argument anterior intercanviant els papers d'x i y.
- Si $x < z < y$, aleshores x apareix al subarbre esquerre del node actual i y al dret. Podem afirmar en aquest cas que el node que visitem és l'ancestre comú més petit.
- Si $z < x$, aleshores necessàriament també $z < y$. Sabem, doncs, que tant x com y apareixeran al subarbre dret del node que estem visitant. Com que l'ancestre comú més petit formarà part d'aquest subarbre dret, ens movem cap a la seva arrel.
- En cas contrari, tenim $y < z$, i per tant també $x < z$. Sabem ara que tant x com y apareixeran al subarbre esquerre del node que estem visitant. Com que l'ancestre comú més petit formarà part d'aquest subarbre esquerre, ens movem cap a la seva arrel.

Proposta de solució al problema 2

(a) Sabem que, en cas pitjor, l'ordenació per inserció té cost $\Theta(n^2)$ i el mergesort, $\Theta(n \log n)$.

Ens podem adonar que totes les operacions que es fan al codi tenen cost $\Theta(1)$: comparació entre enters, accessos a vector, operacions aritmètiques entre enters i crides a *push_back*. Per tant, en farem prou amb calcular el nombre de voltes que fan els dos bucles.

L'observació més important és que el valor d'i s'inicialitza (a zero) només un cop, abans d'entrar al bucle més extern. Addicionalment, és només el bucle intern qui modifica i. Com que aquest bucle incrementa i en una unitat a cada volta, i l'última execució d'aquest bucle s'aturarà quan $i \geq n$, podem afirmar que, en total, el bucle intern farà n voltes.

Com que cada execució del bucle intern incrementa i en almenys una unitat, i el bucle extern també acaba quan $i \geq n$, podem afirmar que el bucle extern farà com a molt n voltes.

Per tant, el cost del programa és $\Theta(n)$ més el cost de l'ordenació. Així, doncs, pel cas de l'ordenació per inserció tindrem cost en cas pitjor $\Theta(n) + \Theta(n^2) = \Theta(n^2)$, i pel mergesort cost $\Theta(n) + \Theta(n \log n) = \Theta(n \log n)$.

(b) Altra vegada, totes les operacions que es fan al codi tenen cost $\Theta(1)$ i en farem prou, doncs, amb calcular el nombre de voltes que fan els dos bucles.

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 i estarem en el cas pitjor. Donada una i concreta, el bucle intern s'executarà $n - i - 1$ vegades. Com que i va des de 0 fins a $n - 1$ el cost total és $(n-1) + (n-2) + \cdots + 1 = \Theta(n^2)$.

El cas millor es donarà quan tots els elements són iguals. En aquest cas, quan $i = 0$ el bucle intern s'executa $n - 1$ vegades, marcant tots els elements amb un -1. Totes les altres voltes del bucle extern (és a dir, $i > 0$) tindran cost $\Theta(1)$, perquè es complirà que $v[i] = -1$. Així doncs, el cost en aquest cas és $\Theta(n)$.

(c) Sigui $n = r - l$. Observem que les crides recursives de *merge_count* fan decréixer aquest valor. En concret, *merge_count* fa dues crides recursives amb $n/2$ i una crida a *combine*. Addicionalment, s'assignen dos vectors $r1, r2$, que és fàcil veure que com a molt tenen mida $n/2$ (perquè contenen les parelles $< z, t >$ per a un vector de mida $n/2$). Per tant, les assignacions tenen cost $\Theta(n)$.

Pel que fa al cost de *combine*, sabem que rep dos vectors de mida com a molt $n/2$. La inicialització del vector *present* té cost $\Theta(n)$. A continuació venen dos bucles ennierats. El bucle intern sempre dona $n/2$ voltes, i el bucle extern també $n/2$ voltes. Així doncs, el cost és $\Theta(n^2)$.

Finalment, l'últim bucle abans de retornar també fa $n/2$ voltes i, per tant, té cost $\Theta(n)$. En resum, la crida a *combine* té cost $\Theta(n^2)$.

Així doncs, la recurrència que descriu el cost del *merge_count* és $C(n) = 2 \cdot C(n/2) + \Theta(n^2)$. Si apliquem el teorema mestre per a recurrències divisores del tipus $C(n) = a \cdot C(n/b) + \Theta(n^k)$, podem identificar que $a = b = 2$, $k = 2$ i, per tant $\alpha = \log_2 2 = 1$. Com que $k > \alpha$, tenim que la recurrència té solució $C(n) \in \Theta(n^k) = \Theta(n^2)$.

(d) Una possible solució és:

```
vector<pair<int,int>> combine (const vector<pair<int,int>>& v1,
                               const vector<pair<int,int>>& v2) {
  vector<pair<int,int>> res;
  int i = 0, j = 0;
  while (i < v1.size () and j < v2.size ()) {
    if (v1[i]. first < v2[j]. first ) {
      res .push_back(v1[i ]);
      ++i;
    }
    else if (v1[i]. first > v2[j]. first ) {
      res .push_back(v2[j ]);
      ++j;
    }
    else {
      res .push_back({v1[i ]. first ,v1[i ].second + v2[j ].second });
      ++i;
      ++j;
    }
  }
  while (i < v1.size ()) {res .push_back(v1[i ]); ++i;}
  while (j < v2.size ()) {res .push_back(v2[j ]); ++j;}
  return res ; }
```

Tot i que no era necessari, a continuació argumentem per què el codi anterior fa que *merge_count* tingui cost $\Theta(n \log n)$ en cas pitjor.

El cost d'aquesta nova versió de *combine* és $\Theta(n)$, amb $n = v1.size() + v2.size()$. De fet, cada iteració de cada bucle aquest algorisme processa en temps constant un element de $v1$ o des de $v2$ (o de tots dos), que no es processaran mai més.

Per tant, la recurrència del cost de *merge_count* és ara $C(n) = 2C(n/2) + \Theta(n)$, és a dir, hem substituït el temps quadràtic de la implementació de *combine* de (2c) per la implementació de cost lineal anterior. La solució d'aquesta recurrència, aplicant el teorema mestre, o adonant-nos que és idèntica al de mergesort, és $\Theta(n \log n)$.

Solució de l'Examen Parcial EDA 21/04/2023

Proposta de solució al problema 1

(a) El cas pitjor es dona quan s'efectuen totes les iteracions al bucle més extern sense trobar una solució. Centrem-nos, doncs, en aquest cas.

Sabem que el bucle extern farà n voltes. Per cadascuna d'aquestes voltes, el bucle intern en farà n. Per tant, el cos de bucle més intern s'executarà exactament n^2 vegades.

Aquest cos executarà dues instruccions *swap*, una comparació entre enters i dues crides a *suma*. Si no fos per aquestes dues crides, el cos tindria cost $\Theta(1)$. Una crida a *suma* involucra passar un vector per referència, inicialitzar la variable s a zero, retornar un enter (tot plegat cost $\Theta(1)$), així com un bucle que itera n vegades acumulant el valor dels elements de v a la variable s. Així doncs, el cost d'una crida a *suma* és $\Theta(n)$. Resumint, el cost del cos del bucle més intern és $\Theta(1) + 2\Theta(n) = \Theta(n)$.

Per tant, com que aquest s'executa n^2 vegades, tenim un cost total de $n^2 \cdot \Theta(n) = \Theta(n^3)$.

(b) La part del codi a completar és:

```
int x = v1[i] + dif/2;
```

Passem a analitzar el cost en cas pitjor d'una crida a *sol*. Com abans, el cas pitjor tindrà lloc quan executem totes les iteracions del bucle més extern.

La primera línia de *sol* efectua dues crides a *suma* i una resta, pel que té cost $\Theta(n)$. La segona línia té cost $\Theta(1)$, doncs només es fan operacions aritmètiques bàsiques i una comparació amb zero. En el cas pitjor, el bucle farà n voltes, i en cadascuna d'elles s'efectuarà una crida a *cerca* i altres operacions $\Theta(1)$. Així doncs, ja podem concloure que el cost serà n vegades el cost d'una crida a *cerca*.

La funció *cerca* és una funció recursiva que, en cas pitjor, efectua tot d'operacions aritmètiques de cost $\Theta(1)$ i dues crides recursives. Com que m és el punt mig del vector, ens n'adonem que les crides recursives són de mida la meitat. Així doncs, la recurrència que descriu el cost d'aquesta funció és $C(n) = 2 \cdot C(n/2) + \Theta(1)$. Si apliquem el teorema mestre per a recurrències divisores del tipus $C(n) = a \cdot C(n/b) + \Theta(n^k)$, podem identificar que $a = b = 2$, $k = 0$ i, per tant $\alpha = \log_2 2 = 1$. Com que $\alpha > k$, tenim que la recurrència té solució $C(n) \in \Theta(n^\alpha) = \Theta(n)$.

Per tant, el cost total serà $n \cdot \Theta(n) = \Theta(n^2)$.

(c) Per tal de millorar el cost, farem que la crida a *cerca* sigui més eficient. En concret, a la funció *sol*, just abans de l'inici del bucle, ordenarem el vector v_2 amb un algorisme d'ordenació que ens garanteixi cas pitjor $n \log n$, com pot ser el *mergesort*. Una vegada ordenat, enlloc de la funció *cerca* podem utilitzar una cerca dicotòmica, que tindrà cost $\Theta(\log n)$.

Si ho comparem amb l'anàlisi de cost anterior, veiem que haurem d'afegir el cost de l'ordenació, i reemplaçar el cost de *cerca* pel cost de la cerca dicotòmica. Per tant, el cost total serà $\Theta(n \log n) + n \cdot \Theta(\log n) = \Theta(n \log n)$.

Proposta de solució al problema 2

(a) $n = 10, k = 2 \Rightarrow 10 = 2^3 + 2^1$

$n = 10, k = 3 \Rightarrow 10 = 2^2 + 2^2 + 2^1$

$n = 10, k = 4 \Rightarrow 10 = 2^2 + 2^1 + 2^1 + 2^1$

$n = 10, k = 5 \Rightarrow 10 = 2^1 + 2^1 + 2^1 + 2^1 + 2^1$

(b) Com que sabem que la representació en binari d'n és la representació com a potències de 2 amb el menor nombre de sumands, si k és menor que el nombre d'uns de la representació en binari no hi ha solució possible.

D'altra banda, és fàcil adonar-se que la representació d'n en potències de dos amb el major nombre de sumands és la suma d'n sumands 2^0. Així doncs, si $k > n$ tampoc hi haurà solució.

(c) Una possible solució és:

```
vector<int> pos_uns (int n) {
  vector<int> v;
  int pos = 0;
  while (n ≠ 0) {
    if (n%2 == 1) v.push_back(pos);
    ++pos;
    n = n/2;
  }
  return v;
}
```

Veiem que totes les operacions que fa la funció tenen cost $\Theta(1)$ (assumint que el *push_back* té cost $\Theta(1)$). Així doncs, el cost vindrà donat pel nombre de voltes que faci el bucle. Si n_0 és el valor inicial de la variable n, en acabar la primera iteració, n val com a molt $n_0/2$ (recordem que la divisió entera en C++ trunca cap a baix). En acabar la segona, val com a molt $n_0/2^2$, en acabar la tercera com a molt $n_0/2^3$ i, en general després de la iteració k-èssima valdrà com a molt $n_0/2^k$. Per tant, podem garantir que quan $n_0 < 2^k$ el bucle s'aturarà perquè n valdrà zero. Això passa quan $\log n_0 < k$, i per tant, podem garantir que el bucle donarà com a molt $\Theta(\log n_0)$ voltes. Com que n_0 era el valor inicial d'n, podem concloure que el cost és $\Theta(\log n)$.

(d) Una possible solució és:

```
void escriu_suma_potencies (int n, int k) {
  vector<int> uns = pos_uns(n);
  if (uns.size () > k or k > n)
    cout << "No hi ha solucio" << endl;
  else {
    priority_queue <int> Q;
    for (auto x : uns) Q.push(x);
    while (Q.size () < k) {
      int m = Q.top ();
      Q.pop ();
      Q.push(m-1);
      Q.push(m-1);
    }
    bool primer = true;
    while (not Q.empty()){
      if (not primer) cout << " + ";
      else primer = false;
      cout << "2^" << Q.top ();
      Q.pop ();
    }
    cout << endl;
  }
}
```

Solució de l'Examen Parcial EDA 03/11/2022

Proposta de solució al problema 1

(a) El programa anterior escriu el valor N^N per pantalla.

Només cal demostrar que $f(x,n)$ retorna x^n per $n \geq 1$. Es pot demostrar fàcilment per inducció. Per $n = 1$, és obvi que retorna $x = x^1$ gràcies a la primera línia de la funció. Per $n > 1$, si assumim per hipòtesi d'inducció que $f(x, n-1) = x^{n-1}$, aleshores la variable tmp conté x^{n-1}. El bucle suma x vegades el valor de tmp, és a dir x^{n-1}, i el guarda dins res. Per tant res conté $x \cdot x^{n-1} = x^n$.

Pel que fa al cost, ens n'adonem que la part recursiva de la funció fa tot d'operacions constants més (1) una crida recursiva de mida $n-1$, i (2) un bucle de cost $\Theta(x)$. Seria incorrecte dir que el cost del bucle és $\Theta(n)$ perquè en les successives crides a la funció no es compleix que $x = n$, sinó que x es manté constant i n va decreixent.

Per a solucionar-ho, calcularem d'una banda el cost de la funció sense considerar el bucle i d'una altra banda el cost de totes les execucions del bucle. Si no considerem el bucle, el cost de la funció ve donat per $C(n) = C(n-1) + \Theta(1)$, que té solució $C(n) \in \Theta(n)$. Pel que fa al bucle, tenim en compte que per una crida inicial $f(N,N)$ amb $N > 1$ es faran $N-1$ execucions del bucle, cadascuna d'elles amb cost $\Theta(N)$. Tot plegat, el bucle ens dona un cost $\Theta((N-1)N) = \Theta(N^2)$. Per tant, el cost del main és $\Theta(N) + \Theta(N^2) = \Theta(N^2)$.

(b) Calculem el límit:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\ln(\ln(n^2))}{\ln(\ln n)} = \lim_{n\to\infty} \frac{\ln(2\ln n)}{\ln(\ln n)} = \lim_{n\to\infty} \frac{\ln(2) + \ln(\ln n)}{\ln(\ln n)} =$$

$$\lim_{n\to\infty} \frac{\ln(2)}{\ln(\ln n)} + \lim_{n\to\infty} \frac{\ln(\ln n)}{\ln(\ln n)} = 0 + 1 = 1$$

Per tant, podem afirmar que $f(n) \in \Theta(g(n))$.

Pel que fa a $F(n)$ i $G(n)$, podríem calcular també $\lim_{n\to\infty} \frac{F(n)}{G(n)}$. Com que quan calculem el límit quan $n \to \infty$ no ens importen els valors que F o G puguin prendre per $n \leq 10$, el límit també serà 1 i per tant $F(n) \in \Theta(G(n))$.

Proposta de solució al problema 2

(a) Recordem que $n = h.size() = s.size()$. La funció proporcionada només fa operacions amb cost $\Theta(1)$ (operacions aritmètiques bàsiques, accessos a vectors, càlcul de mínim i màxim) però té dos bucles ennierats. El bucle intern té cost $\Theta(n)$, i aquest cos és independent de la iteració del bucle extern on estiguem. El bucle extern s'executa n vegades, i per tant el cost total és $\Theta(n \cdot n) = \Theta(n^2)$.

(b) El codi omplert és:

```
int radium_v2 (const vector<int>& h, const vector<int>& s) {
  int r = 0, j = 0;
  for (int i = 0; i < s.size (); ++i){
    while (j < h.size () and h[j] < s[i]) ++j;
    int rad;
    if (j == h.size ()) rad = s[i] - h[h.size ()-1];
    else if (j == 0)    rad = h[0] - s[i];
    else                rad = min(s[i] - h[j - 1], h[j] - s[i]);
    r = max(r,rad);
  }
  return r;
}
```

Pel que fa al cost, ens fixem que la funció fa tot d'operacions de cost $\Theta(1)$, però té dos bucles ennierats. El que hem de tenir en compte és que el valor de j, que controla el bucle intern, no s'inicialitza en cada volta del bucle extern, sinó que j val 0 a l'inici de la funció i es va incrementant al llarg de tota l'execució, fins a valer com a molt n. Per tant, durant tota l'execució de la funció es fan com a molt n voltes al bucle **while** i, per tant, té cost $\Theta(n)$. Remarquem que aquest cost no és per a cada volta del bucle extern, sinó que ja considera totes les voltes. Finalment només ens queda analitzar el bucle extern, que s'executa n vegades i, per tant, té cost $\Theta(n)$ si obviem el cost del **while**, que ja l'hem comptat a part. Així doncs, el cost total és $\Theta(n + n) = \Theta(n)$.

(c) Una possible solució és:

```
int find (const vector<int>& h, int l, int r, int p) {
  if (l > r) return h.size ();
  else {
    int m = (l+r)/2;
    if (h[m] < p) return find (h,m+1,r,p);
    else if (m > l and h[m-1] ≥ p) return find(h,l,m-1,p);
    else return m;
  }
}
```

Per analitzar el seu cost ens n'adonem que totes les operacions tenen cost $\Theta(1)$, excepte la crida recursiva que sempre té mida la meitat de la mida original. Per tant, el cost de la funció ve donat per la recurrència $C(n) = C(n/2) + \Theta(1)$, que té solució $C(n) = \Theta(\log n)$.

(d) La línia que acabem d'introduir sempre té cost en cas pitjor $\Theta(\log n)$. Per tant, com que el bucle **for** dona n voltes, i a cada volta fa un treball $\Theta(\log n)$ més altres operacions de temps $\Theta(1)$, tenim que el cost total és $\Theta(n \log n)$.

Solució de l'Examen Parcial EDA 08/11/2022

Proposta de solució al problema 1

(a) El bucle que inicialitza el vector v triga temps $\Theta(n)$. El mateix passa amb la crida a *random_shuffle*.

Pel que fa als bucle ennierats, fixem-nos primer que dins el vector v hi posem tots els nombres entre 1 i n, i a continuació els reordenem de manera aleatòria. Si no els haguéssim ordenat, per cada valor de i, el bucle més intern faria $v[i] = i + 1$ voltes (i per tant incrementaria s en una unitat $i + 1$ vegades). Com que i es mou des de 0 fins a $n - 1$, el nombre de voltes totals del bucle intern seria $1 + 2 + 3 + \ldots + n = n(n + 1)/2$. Així doncs, podem afirmar que el codi sense l'ordenació calcula $n(n + 1)/2$ i té cost $\Theta(n(n + 1)/2) = \Theta(n^2)$.

L'única cosa que canvia degut a l'ordenació de v és que deixa ser cert que per $i = 0$ el bucle intern faci 1 volta, que per $i = 1$ en faci 2, que per $i = 2$ en faci 3, etc. En el nostre codi, per cada valor de i el bucle intern farà $v[i]$ voltes. Però com que v és una permutació de $\{1, 2, \ldots, n\}$, hi haurà una iteració on es farà 1 volta, una altra on se'n faran 2, una altra on se'n faran 3, etc. Per tant, el codi calcula el mateix amb el mateix cost asimptòtic.

(b) El seu cost és $\Theta(n \log \log n)$. Anem a veure per què.

El bucle extern dona n voltes, i el cost del bucle intern és independent del valor de j. Per tant, el cost total serà n vegades el cost del bucle intern.

Pel que fa al bucle intern, aquest s'atura quan $k \geq n$. En entrar a la primera iteració k val 2, a la segona val 4, a la tercera val 16, a la quarta val 256, etc. Podem afirmar que a la iteració i-èsima k val 2^{2^i}. Es donaran voltes, per tant, mentre $2^{2^i} < n$, el que equival a que $2^i < \log n$, i que $i < \log \log n$. Per tant, el bucle intern fa $\Theta(\log \log n)$ iteracions i el cost total del codi és $\Theta(n \log \log n)$.

(c) Per analitzar el cost de l'algorisme d'inserció, podem comptar el nombre d'intercanvis que s'han de fer. Sabem que el nombre d'intercanvis que es faran quan s'hagi de col·locar un nombre a la posició que li pertoca es correspon al nombre d'elements a la seva esquerra que són estrictament majors que ell en la configuració inicial.

Els dos primers nombres no tenen cap element major a la seva esquerra. Pel 2 i pel $2n - 1$ en tenim 1 per a cadascun. Pel 3 i pel $2n - 2$ en tenim 2 per a cadascun. Pel 4 i pel $2n - 3$ en tenim 3 per a cadascun, i així successivament, fins a la parella n, $n + 1$ pels que en tenim $n - 1$ per cadascun. Així doncs, el nombre d'intercanvis serà $1 + 1 + 2 + 2 + 3 + 3 + \ldots + (n - 1) + (n - 1) = 2 \cdot n(n - 1)/2 = n(n - 1)$, que és $\Theta(n^2)$. Per tant, aquest és el cost de l'algorisme d'ordenació per inserció per aquesta entrada.

Proposta de solució al problema 2

(a) En el cas pitjor, realitzarem totes les iteracions (n) al bucle que hi ha dins *inefficient* i acabarem retornant n. A cada bucle es crida a la funció *find*. Com que es passa el vector per referència, el pas de paràmetres és $\Theta(1)$. Totes les altres operacions dins *inefficient* tenen cost $\Theta(1)$, pel que només ens hem de centrar en les crides a *find*.

Podem veure que *find* és una funció recursiva, on el que decreix és el valor de *pos*, que inicialment és $n-1$. Com que totes les operacions són constants, el seu cost ve descrit per la recurrència $T(n) = T(n-1) + \Theta(1)$, que té solució $T(n) \in \Theta(n)$.

Per tant, com que cada crida a *find* té cost $\Theta(n)$ i es fan n crides, el cost total és $\Theta(n^2)$.

(b) Una possible solució és:

```
int efficient (const vector<int>& v, int l, int r) {
  if (l > r) return v.size();
  int m = (l+r)/2;
  if (v[m] > m) {
    if (m == l or v[m-1] == m-1) return m;
    else return efficient (v,l,m-1);
  }
  else return efficient (v,m+1,r); // we know v[m] == m
}
```

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.

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 $T(n) \in \Theta(n \log n)$.