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 $\Theta(n \log n)$.