

Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona

Cognoms, Nom

[illegible]

Titulació: Grau en Enginyeria Informàtica

Assinatura: Algorísmia (A)

Duració: 2h 30m

Curs: Q2 2021–2022 (Final)

Data: 8 de juny de 2022

Instruccions generals:

- Entregueu per separat les solucions de cada exercici (Ex. 1, Ex. 2, Ex. 3, Ex. 4). **Només un full per exercici.** Poseu el vostre nom i cognoms, **en lletra de bloc**, a cada full entregat.
- L'exercici 4 (qüestions curtes) s'ha de resoldre fent servir l'espai reservat per a cada resposta.
- Heu de donar i argumentar la correctesa i l'eficiència dels algorismes que proposeu. Per fer-ho podeu donar una descripció d'alt nivell de l'algorisme suficient per tal que, amb les explicacions i aclariments oportuns, justifiqueu que l'algorisme és correcte i té el cost indicat.
- A les qüestions curtes (Ex. 4), heu de justificar totes les vostres afirmacions; en cas contrari la nota de la pregunta serà 0. Quan la pregunta sigui una qüestió de *Cert o Fals?*, heu de dir si l'enunciat és cert o fals abans de justificar-ne la resposta.
- Podeu fer crides a algorismes que s'han vist a classe, però si la solució és una variació, n'haureu de donar els detalls.
- Es valorarà especialment la claredat i concisió de la presentació.
- La puntuació total d'aquest examen és de **10 punts**.

Pàgina en blanc. No contesteu res aquí.

© Professorat GRAU-A

- Doneu un algorisme eficient per a calcular quin és el mínim nombre d'andanes que necessitem habilitar a l'estació per tal que tot tren que arribi pugui estacionar, sense haver-se d'esperar que un altre tren marxi per ocupar el seu lloc.

Aquest problema és, en realitat, el clàssic conegut com *Interval Coloring* o *Interval Partitioning*. En el nostre cas tenim un conjunt de n trens, i cada tren i té un instant d'arribada h_i i un instant de partida $h_i + e_i$. L'objectiu és utilitzar el nombre mínim de recursos (en el nostre cas, andanes) per programar totes les estades dels trens a les andanes de l'estació. S'ha de resoldre de manera que cap tren hagi de retardar el seu temps d'arribada perquè no hi ha cap andana lliure a l'estació (o, equivalentment, que no ens trobéssim que dos o més trens vulguin estacionar a la mateixa via al mateix temps).

Figure 1 consists of two Gantt charts, (a) and (b), illustrating the scheduling of tasks a through j. The timeline for both charts is marked from 9:00 to 4:30 in 30-minute intervals. In chart (a), tasks are scheduled as follows: a (9:00-10:00), b (9:30-12:00), c (9:00-10:00), d (11:00-12:00), e (11:30-2:00), f (1:30-2:30), g (1:30-2:30), h (2:30-4:00), i (3:30-4:00), and j (3:30-4:00). In chart (b), the schedule is modified: task e is extended to cover the period from 11:30 to 2:30, and task h is extended to cover the period from 2:30 to 4:30. The other tasks (a, b, c, d, f, g, i, j) remain in their original positions.

Definim la profunditat d'un conjunt d'interval·ls com el nombre màxim d'interval·ls que coincideixen en qualsevol instant de temps. Aleshores observem que el nombre d'andanes necessàries serà almenys la profunditat del conjunt d'entrada. Per tant,

qualsevol planificació dels trens que utilitzi un nombre d'andanes igual a la profunditat és, de fet, una planificació òptima perquè no podem fer-ho millor.

Podem trobar sempre una planificació òptima? La resposta és sí, i per això dissenyem un senzill algorisme *greedy* que programarà els estacionaments dels trens utilitzant un nombre d'andanes igual a la profunditat. Considerem els intervals d'estacionament dels trens en ordre creixent de l'hora d'inici i assignem a cada tren qualsevol andana compatible (és a dir, que estigui lliure en el moment d'arribada del tren). Si totes les andanes es troben ocupades quan intentem assignar un nou tren, aleshores habilitarem una nova andana. Mantindrem un control del nombre d'aules obertes, que serà el que retornarem com a resposta.

Pseudocodi de l'algorisme:

```
1: function CAP D'ESTACIÓ ( $L = \{(h_1, e_1), \dots, (h_n, e_n)\}$ )
2:   Ordenar  $L$  per temps d'arribada dels trens ( $h_i$ ) en ordre creixent1
3:    $d \leftarrow 0$ 
4:   for  $j = 1$  to  $n$  do
5:     if tren  $j$  és compatible amb alguna andana  $k \in [1, d]$  then
6:       assignar tren  $j$  a l'andana  $k$ 
7:     else
8:       habilitar l'andana  $d + 1$ 
9:       assignar tren  $j$  a aquesta nova andana
10:     $d \leftarrow d + 1$ 
11:   end if
12: end for
13:   return  $d$ 
14: end function
```

L'assignació de trens a andanes (línies 6, 8 i 9) no és realment necessària per aquest problema, donat que l'enunciat només ens demana que calculem el nombre mínim d'andanes (d , línia 14). Ens hauríem de preocupar de com guardar aquestes assignacions si l'exercici ens demanés, a més, que detalléssim l'ocupació de les d andanes en el temps.

L'ordenació (línia 2) necessita temps $\mathcal{O}(n \log n)$. El temps d'execució total de l'algorisme dependrà de com implementem l'acció de trobar alguna andana compatible (línia 5). Si senzillament recorrem les d andanes ocupades en aquell moment per veure si alguna està lliure, el cost de l'algorisme pujarà a $\mathcal{O}(n \log n + n^2) = \mathcal{O}(n^2)$. En canvi, podem aconseguir un temps total de $\mathcal{O}(n \log n)$ si, per a cada andana k mantenim el temps en què marxa l'últim tren estacionat en ella (o, el que és el mateix, el temps en què queda lliure l'andana) i mantenim les andanes utilitzades fins aquell moment en una cua de prioritats (min-heap). Si el tren j és compatible amb alguna

¹No importa com es resolguin els empats.

```

1: function CAP D'ESTACIÓ ( $L = \{(h_1, e_1), \dots, (h_n, e_n)\}$ )
2:   Ordenar  $L$  per temps d'arribada dels trens ( $h_i$ ) en ordre creixent1
3:   P.insert( $h_1 + e_1$ )    ▷ S'assigna el primer tren a la primera andana i s'encua
4:   for  $j = 2$  to  $n$  do
5:      $m \leftarrow$  P.pop()      ▷ Temps en què queda lliure la primera andana
6:     if  $h.j \geq m$  then    ▷ Si el tren  $j$  arriba després que quedi lliure...
7:       P.push( $h.j + e.j$ )  ▷ S'assigna el tren  $t$  a l'andana i s'actualitza
8:     else                 ▷ Si el tren  $t$  arriba abans que quedi lliure...
9:       P.push( $m$ )           ▷ Tornem l'andana a la cua
10:      P.push( $h.j + e.j$ )   ▷ S'assigna una nova andana al tren  $j$  i s'encua
11:    end if
12:  end for
13:  return P.size()        ▷ Total d'andanes utilitzades
14: end function

```

Per demostrar l'optimitat, sigui d el nombre d'andanes que l'algoritme greedy assigna. Aleshores es va habilitat l'andana d -èsima perquè havíem d'estacionar una tren, per exemple j , que era incompatible amb tots els $d - 1$ altres trens. Donat que són incompatibles, es dedueix que aquests $d - 1$ trens marxen de l'estació després de h_j (temps d'arribada del tren j). Donat que hem ordenat per els temps d'arribada dels trens, aquests $d - 1$ trens també havien arribat a l'estació abans (o al mateix temps) de h_j . Per tant, tenim d estacionaments superposats en aquest moment.² Això implica que la profunditat és almenys d i la nostra planificació és òptima.

- Ordenar L' per temps de sortida del trens $(h_i + e_i)$ en ordre creixent no funciona. Contraexemple: $L = \{(1, 2), (2, 3), (6, 1), (4, 4)\}$.

²O, tècnicament, a temps $h_j + \epsilon$ per a una petita constant ϵ .

2. **Resistència de closques (3 punts)** La duresa de la closca dels ous es pot determinar per la quantitat de carbonat de calci, CaCO_3 , que conté. Volem verificar la duresa de la closca dels ous de les nostres gallines però, donat que som informàtics i no químics, volem idear un mètode més algorísmic per fer-ho usant un edifici d' N plantes i p ous de mostra. Hem observat el següent:

- Si tinguéssim només un ou, aniríem al primer pis, llançaríem l'ou per la finestra i mirariem si es trenca. Si no ho fa, llavors repetiríem el procés des del segon pis, des del tercer, etc. fins que es trenqués o declarem que són indestructibles ;-)

Aquesta estratègia és òptima donat que només tenim un ou i llavors el màxim de llançaments que cal fer és N ; amb menys llançaments no podríem garantir que trobarem sempre l'altura exacta a partir de la qual es trenquen els ous.

- En cas que disposem de dos ous (ou_1 i ou_2) podem fer molts menys llançaments. Per exemple, amb $N = 100$ només cal fer 14 llançaments en el pitjor dels casos. Veiem per què:

El primer, ou_1 , es llança des del pis $x = 9$. Poden passar dues coses: **(a)** si es trenca, llavors fem cerca seqüencial amb l' ou_2 entre els pisos 1 i 8 (en total 9 llançaments com a màxim); **(b)** si no es trenca, llavors es torna a llançar l' ou_1 des del pis 22, i poden tornar a passar dues coses: **(b.1)** si es trenca, fem cerca seqüencial amb l' ou_2 entre els pisos 10 i 21 (en total de $14 = 12 + 2$ llançaments com a màxim), però **(b.2)** si no es trenca l' ou_1 al pis 22, llavors llançarem novament l' ou_1 des dels pisos 34, 45, 55, 64, 72, 79, ... fins que es trenqui i llavors completarem la cerca de manera seqüencial amb l' ou_2 en l'interval apropiat.

- Amb un número prou alt d'ous de prova, la millor estratègia és una “cerca binària”; però si p no és prou gran la “cerca binària” **no** funciona.

Donats N i la quantitat p d'ous de prova, calculeu quin és el menor nombre de llançaments $L_{N,p}$ que garanteix trobar el pis més alt des del qual l'ou no es trencarà en llançar-lo. Determineu també com s'han de dur a terme els llançaments.

- Dona (i justifica) una recurrència per al cas de dos ous, és a dir per obtenir $L_{N,2}$.
- Dona (i justifica) una recurrència per a qualsevol N i p , és a dir per obtenir $L_{N,p}$.
- Dona un algorisme per determinar l'estratègia de llançaments amb el nombre mínim de llançaments. Justifica el seu cost en temps i espai.

Una solució:

- Denotemos $D_n := L_{n,2}$ el número mínimo necesario de lanzamientos cuando tenemos n plantas, $n \leq N$, y 2 huevos. Para simplificar la recurrència añadiremos

os hacer, como
ner huevo). Si
car el punto de
 $n-x$. O sea que
solución ópti-
d, así tenemos

tos cuando te-
 endo el mismo
 iso x -ésimo del
 $\leq n$. O bien se
 ntrar el punto
 $q - 1$ huevos,
 uscar el punto
 ndose de los q
 entonces

ate del número
 $= 1$, para todo
 ≥ 0 . Poniendo

1 < q ≤ p.

Finalmente tenemos que obtener, sin incremento del coste asintótico ni de espacio ni de tiempo, el primer piso $X_{n,q}$ en el cual debemos lanzar un huevo en la estrategia óptima que resuelve el problema para n pisos y q huevos. Si $n = 1$ o $q = 1$ tendremos $X_{n,q} = 1$ (línea 13 en el código más abajo), ya que en dichos casos siempre habremos de comenzar en el primer piso, y en general anotaremos el valor x que minimiza el caso general de la recurrencia para $L_{n,q}$ (línea 18 del código). Para reconstruir la solución usamos una función recursiva `reconstruye_plan(n,q,X,a)` que recibe n , q , la matriz X y el número del piso a en el cual empieza el rango de pisos a considerar, e imprime el árbol de decisiones que guía los lanzamientos. Hay otras muchas opciones, p.e., podría construirse el árbol como estructura de datos explícitamente.

La llamada inicial será `reconstruye_plan(N,p,X,1)`. El coste de dicha función es proporcional al total de nodos en el árbol de decisiones implícito, que es $\Theta(N)$. También se podría hacer una versión “interactiva” en la que nos dijera un número de piso desde donde lanzar, informaríamos del resultado (el huevo se rompe o no), entonces nos daría un nuevo número de piso desde el cual efectuar el siguiente lanzamiento (con el mismo huevo si no se ha roto en el paso anterior, con un nuevo huevo si se ha roto en el último lanzamiento), etc. Esto correspondería a seguir nada más una rama del árbol de decisiones y el coste sería proporcional a $\Theta(L_{N,p})$.

```

1 // Pre:
2 //     vector< vector<int> > L(N+1, p+1); // la columna 0 no se usa
3 //     vector< vector<int> > X(N+1, p+1); // fila y columna 0 no usadas
4 // Post: L[N][p] = número mínimo de lanzamientos requerido
5 //       X[i][j] = ver explicaciones en el texto
6 // Coste:  $\mathcal{O}(N^2p)$ 
7 void calcular_plan_lanzamiento_huevos(int N, int p,
8     vector<vector<int>>& L, vector<vector<int>>& X) {
9     for (int q = 1; q <= p; ++q) { L[0][q]=0; L[1][q]=1; X[1][q]=1; }
10    for (int n = 0; n <= N; ++n) { L[n,1] = 1; X[n,1] = 1; }
11    for (int q = 1; q <= p; ++q) {
12        for (int n = 2; n <= N; ++n) {
13            L[n][q] = 1+L[n-1][q]; X[n][q] = 1;
14            for (int x = 2; x <= n; ++x) {
15                int Lx = max(L[x-1][q-1], L[n-x][q]);
16                if (L[n][q] > 1 + Lx) {
17                    L[n][q] = 1 + Lx;
18                    X[n][q] = x;
19                }
20            } // end for x
21        } // end for n
22    } // end for q
23 }
24
25 // Pre: Si  $n > 0$  y  $q > 0$  entonces  $1 \leq X[n][q] \leq n$ 
26 // Post: Escribe el árbol que codifica al plan para hallar
27 // el piso buscado entre los pisos  $a$  y  $a+n-1$ 
28 void reconstruye_plan(int n, int q,
29     const vector<vector<int>>& X, int a) {
30     if (n > 0 and q > 1) {
31         cout << "Lanza_huevo_" << q \
32             << "_desde_piso_" << a - 1 + X[n][q] << endl;
33         cout << "Si_se_rompe_al_lanzar_desde_" << a-1+X[n][q] << endl;
34         reconstruye_plan(X[n][q]-1, q-1, X, a);

```



```

35
36     cout << "Si no se rompe al lanzar desde " << a-1+X[n][q] << endl;
37     reconstruye_plan(n-X[n][q], q, X, a+X[n][q]);
38 } else if (q == 1) {
39     cout << "Busca secuencialmente entre " << a \
40         << " y " << a+n-1 << endl;
41 }
42 }

```

3. **Flow Bicing (2.5 punts)** La ciutat de Barcelona necessita actualitzar el seu servei de bicicletes compartides de cara a l'estiu. Les bicicletes en funcionament s'estacionen en una xarxa d' n punts d'aparcament distribuïda per la ciutat.

Per a cada punt d'aparcament de bicicletes v l'ajuntament té una estimació del nombre d'usuaris α_v que necessitaran una bicicleta a primera hora del matí. A més, la nit anterior s'ha fet una ronda de reconeixement a cada punt v per saber el nombre de bicicletes β_v que hi haurà disponibles l'endemà. Cada dia, per posar en marxa el servei, cal que tots els punt d'aparcament tinguin disponibles tantes bicicletes com s'hagi previst que demanaran els seus usuaris.

L'ajuntament també té una estimació de la quantitat màxima de bicicletes que, en cas que fos necessari, es podrien transportar entre dos punts d'aparcament diferents. Entre tot parell $e = (u, v)$ de punts d'aparcament de la xarxa, es poden transportar fins a $c(e) \geq 0$ bicicletes de u cap a v .

Dissenyeu un algorisme per a poder decidir si, amb l'estimació de necessitat de bicicletes que té l'ajuntament, podem tenir el servei preparat per funcionar a l'endemà. En cas afirmatiu, el vostre algorisme també ha d'explicitar el nombre de bicicletes que s'han de traslladar d'una estació a una altra per deixar-ho tot preparat. En cas negatiu, digueu quins són els punts d'aparcament que no reben totes les bicicletes necessàries a primera hora del matí. Justifiqueu-ne la correctesa i el cost temporal.

Una solució:

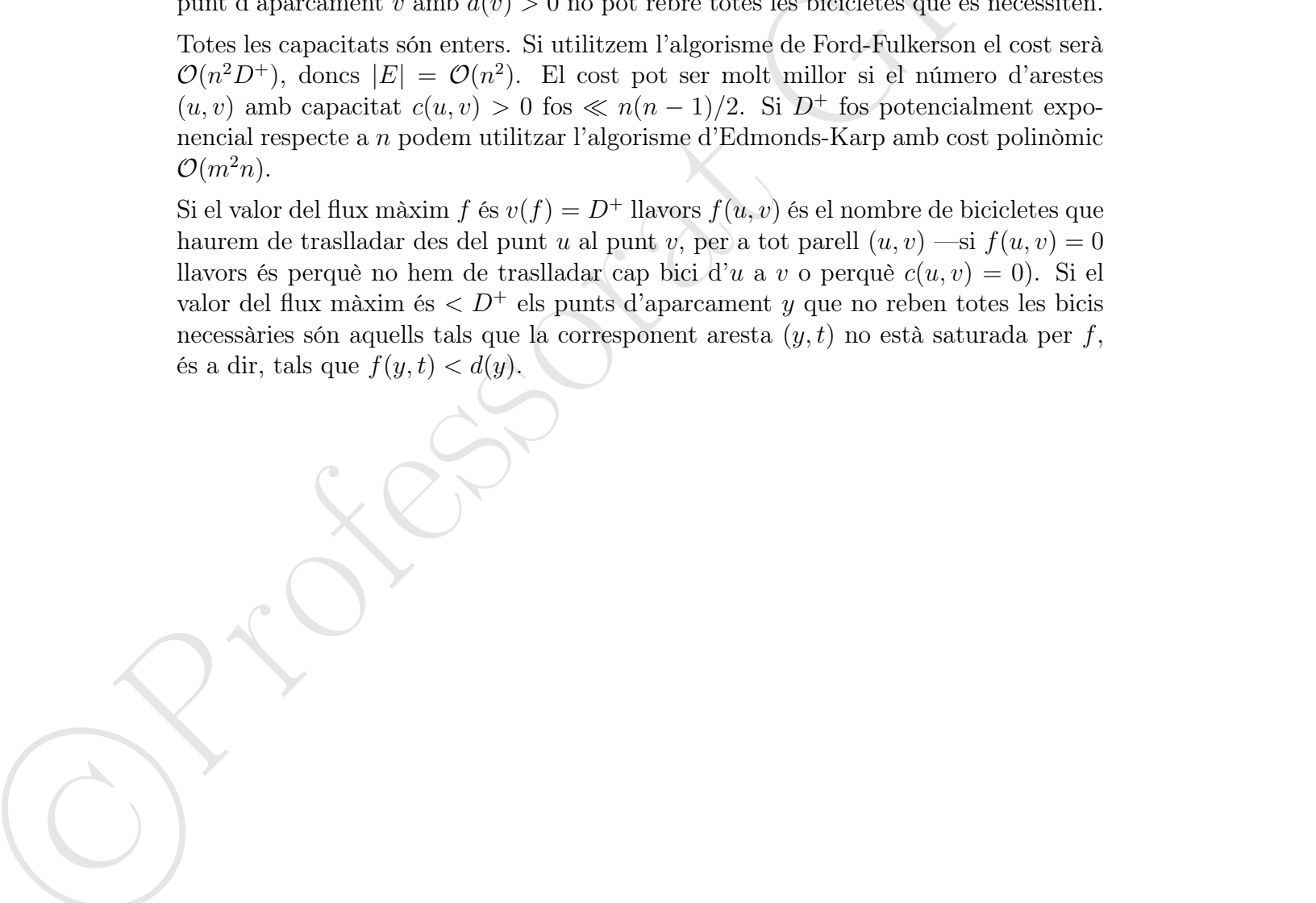
Donat un punt d'aparcament v sigui $d(v) = \alpha_v - \beta_v$. Si $d(v) > 0$ el punt d'aparcament té demanda de bicicletes, que hauràn d'arribar des d'altres punts; si $d(v) < 0$ llavors el punt d'aparcament té un excedent de bicicletes que pot redirigir als altres punts d'aparcament. Sigui $D^+ = \sum_{v:d(v)>0} d(v)$ y $D^- = \sum_{v:d(v)<0} d(v)$. Clarament una condició necessària (pero no suficient) per a poder satisfer totes les demandes és que $|D^-| \geq D^+$.

Definim una xarxa \mathcal{N} ón:

- Vèrtexos: $V = \{s, t\} \cup$ punts d'aparcament,
- Arestes:

$$E = \{(u, v) \mid c(u, v) > 0\} \cup \{(v, u) \mid c(u, v) > 0\} \\ \cup \{(s, x) \mid d(x) < 0\} \cup \{(y, t) \mid d(y) > 0\}$$

- Capacitat $c(s, x) = -d(x)$
- Capacitat $c(y, t) = d(y)$



Si el valor del flux màxim f és $v(f) = D^+$ llavors $f(u, v)$ és el nombre de bicicletes que haurem de traslladar des del punt u al punt v , per a tot parell (u, v) —si $f(u, v) = 0$ llavors és perquè no hem de traslladar cap bici d’ u a v o perquè $c(u, v) = 0$). Si el valor del flux màxim és $< D^+$ els punts d’aparcament y que no reben totes les bicis necessàries són aquells tals que la corresponent aresta (y, t) no està saturada per f , és a dir, tals que $f(y, t) < d(y)$.

4. Qüestions curtes (0.5 punts \times 5 = 2.5 punts)

- (a) Donats n enters positius on cadascun d'ells és com a molt V , RADIXSORT els pot ordenar en temps $\mathcal{O}(n)$ de la següent forma: considera cada enter en base 10 i després els ordena per dígit (començant pels de menys pes) utilitzant COUNTINGSORT. Cert o fals?

Una solució: Fals. Cada enter expressat en base 10 tindrà $\mathcal{O}(\log_{10} V)$ dígit. L'algorisme proposat de RADIXSORT usant COUNTINGSORT tindrà cost total $\mathcal{O}(n \log_{10} V)$. Excepte si $V = \Theta(1)$ aquest cost és més que lineal; per exemple, si $V = \Theta(\log n)$ llavors el cost de l'algorisme proposat és $\Theta(n \log \log n)$, si $V = \Theta(n^c)$ per alguna constant $c > 0$ llavors el cost del RADIXSORT serà $\Theta(n \log n)$, etc.

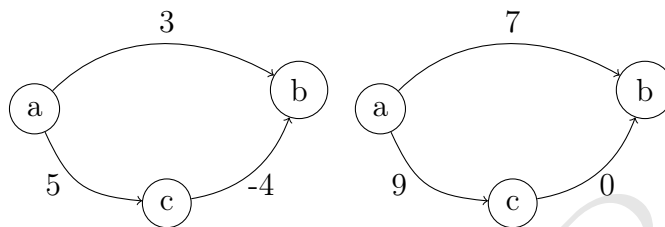
- (b) Considereu un algorisme que te com a entrada un vector *no ordenat* de $3n$ enters diferents $A[1, \dots, 3n]$ i torna dos enters x i y tals que n elements d' A són $\leq x$, n elements d' A tenen valor entre x i y , i n elements d' A són $\geq y$. És cert que aquest algorisme ha de tenir complexitat $\Omega(n \lg n)$?

Una solució: Fals. Es pot fer amb temps $\Theta(n)$ seleccionant l'element $(n+1)$ -èsim (x) i l'element $(2n+1)$ -èsim (y), utilitzant l'algorisme de selecció amb cost lineal en cas pitjor.

- (c) Sigui $G = (V, E, w)$ un graf connex, no dirigit, sense multiarestes i ponderat ($w : E \rightarrow \mathbb{Z}^+$), on no es repeteixen pesos. Les dues arestes de menys pes d' E són sempre a tots els MST que s'hi puguin construir. Cert o fals?

Una solució: Cert. Primer, donat que tots els pesos són diferents, el MST és únic. Siguin e_1 i e_2 les arestes de menys pes i la segona aresta de menys pes, respectivament. A l'algorisme de Kruskal, e_1 és sempre la primera aresta que s'afegeix en la construcció del MST. Donat que e_2 no pot crear un cicle amb e_1 , serà necessàriament la següent aresta que afegirà l'algorisme de Kruskal. Aleshores, per la correcció de l'algorisme de Kruskal, les dues arestes de menys pes, e_1 i e_2 sempre són al MST.

- Una solució:** Fals. Per exemple, pel digraf $G = (V, E)$ el camí més curt d' a a b passa per c i té pes 1; al digraf G' (a la dreta) el camí més curt entre a i b és l'arc (a, b) i té pes 7, no el camí $a \rightsquigarrow c \rightsquigarrow b$ que té pes 9.



- Una solució:** El valor f del flux màxim és igual a la capacitat d' F . Si treiem una aresta qualsevol e d' F el conjunt $F' = F \setminus \{e\}$ és necessàriament un (s, t) -tall amb capacitat mínima de la nova xarxa \mathcal{N}' . Si l'aresta escollida e és la que té capacitat $c(e)$ mínima dintre d' F llavors el valor de flux màxim disminueix en la quantitat més petita possible ($f' = f - c(e)$).