

# Algorítmica

## Colección de Problemas

Q1-2020-2021



---


Maria José Blesa  
Josep Díaz  
Conrado Martínez  
Maria José Serna




---

# Índice general

<b>1. Repaso de Conceptos Básicos</b>	<b>1</b>
<b>2. Algoritmos Voraces</b>	<b>9</b>
<b>3. Soluciones</b>	<b>19</b>
3.1. Repaso de Conceptos Básicos . . . . .	19
3.2. Algoritmos Voraces . . . . .	21
<b>4. Soluciones del Profesor</b>	<b>27</b>
4.1. Repaso de Conceptos Básicos . . . . .	27
4.2. Algoritmos Voraces . . . . .	49

Los problemas marcados con un  aparecen resueltos al final de la colección e incluso es posible que la solución se presente en clase, pero intentad resolverlos vosotros mismos antes de mirar las soluciones.

El documento en PDF tiene diversos elementos de navegación: se puede clicar en el índice general para acceder directamente al correspondiente capítulo, se puede clicar sobre el símbolo  para acceder directamente a la correspondiente solución, etc.


Si no se dice lo contrario y se necesita suponerlo así, considerad que todos los logaritmos son en base 2, aunque procurará explicitarse siempre que sea necesario escribiendo  $\log_2$  o bien lg.

A fecha de hoy (30 de octubre de 2020), los capítulos 3, 4 y 5 están sujetos a actualizaciones, solo los dos primeros capítulos contienen las listas de problemas oficiales publicados para este cuatrimestre Q1-2020-2021. Las sucesivas ediciones de este documento irán incorporando las versiones definitivas de las listas restantes.





---

## Repaso de Conceptos Básicos

-  1. Suposem que tenim un vector  $A$  amb  $n$  nombres enters diferents, amb la propietat: existeix un únic índex  $p$  tal que els valors  $A[1 \cdots p]$  estàn en ordre creixent i els valors  $A[p \cdots n]$  estàn en ordre decreixent. Per exemple, en el següent vector tenim  $n = 10$  i  $p = 4$ :

$$A = (2, 5, 12, 17, 15, 10, 9, 4, 3, 1)$$

Dissenyau un algorisme eficient per trobar  $p$  donada una matriu  $A$  amb la propietat anterior.

-  2. Un vector  $A[n]$  conté tots els enters entre 0 i  $n$  excepte un.
- a) Dissenyau un algorisme que, utilitzant un vector auxiliar  $B[n + 1]$ , detecti l'enter que no és a  $A$ , i ho faci en  $O(n)$  passos.
  - b) Suposem ara que  $n = 2^k - 1$  per a  $k \in \mathbb{N}$  i que els enters a  $A$  venen donats per la seva representació binària. En aquest cas, l'accés a cada enter no és constant, i llegir qualsevol enter té un cost  $\lg n$ . L'única operació que podem fer en temps constant es "recuperar" el  $j$ -èssim bit de l'enter a  $A[i]$ . Dissenyau un algorisme que, utilitzant la representació binària per a cada enter, trobi l'enter que no és a  $A$  en  $O(n)$  passos.
-  3. El coeficient de Gini és una mesura de la desigualtat ideada per l'estadístic italià Corrado Gini. Normalment s'utilitza per mesurar la desigualtat en els ingressos, dins d'un país, però pot utilitzar-se per mesurar qualsevol forma de distribució desigual. El coeficient de Gini és un nombre entre 0 i 1, on 0 es correspon amb

la perfecta igualtat (tots tenen els mateixos ingressos) i on el valor 1 es correspon amb la perfecta desigualtat (una persona té tots els ingressos i els altres cap).

Formalment, si  $r = (r_1, \dots, r_n)$ , amb  $n > 1$ , és un vector de valors no negatius, el *coeficient de Gini* es defineix com:

$$G(r) = \frac{\sum_{i=1}^n \sum_{j=1}^n |r_i - r_j|}{2(n-1) \sum_{i=1}^n r_i}.$$

Proporcioneu un algorisme eficient per calcular el coeficient de Gini donat el vector  $r$ .

- 👁 4. Per a cadascú dels algorismes, digueu quin és el temps en cas pitjor, quan l'entrada és un enter positiu  $n > 0$ .

```

a)  for  $i = 1$  to  $n$  do
       $j = i$ 
      while  $j < n$  do
         $j = 2 * j$ 
      end while
    end for

b)  for  $i = 1$  to  $n$  do
       $j = n$ 
      while  $i * i < j$  do
         $j = j - 1$ 
      end while
    end for

c)  for  $i = 1$  to  $n$  do
       $j = 2$ 
      while  $j < i$  do
         $j = j * j$ 
      end while
    end for

d)   $i = 2$ 
      while  $(i * i < n)$  i  $(n \bmod i \neq 0)$  do
         $i = i + 1$ 
      end while

```

- 👁 5. El problema 2SAT té com a entrada un conjunt de clàusules, on cada clàusula és la disjunció (OR) de dos literals (un literal és una variable booleana o la negació d'una variable booleana). Volem trobar una manera d'assignar un valor cert o fals a cadascuna de les variables perquè totes les clàusules es satisfaguin—és a dir, hi hagi al menys almenys un literal cert a cada clàusula. Per exemple, aquí teniu una instància de 2SAT:

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_2) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_4).$$

Aquesta instància és satisfactible: fent  $x_1, x_2, x_3, x_4$  cert, fals, fals i cert, respectivament.

El propòsit d'aquest problema és conduir-vos a una manera de resoldre 2SAT de manera eficient reduint-ho al problema de trobar les components connexes fortes d'un graf dirigit. Donada una instància  $F$  de 2SAT amb  $n$  variables i  $m$  clàusules, construïu un graf dirigit  $G_F = (V, E)$  de la següent manera.

- $G_F$  té  $2n$  nodes, un per a cada variable i un per a la seva negació.
- $G_F$  té  $2m$  arcs: per a cada clàusula  $(\alpha \vee \beta)$  de  $F$  (on  $\alpha, \beta$  són literals),  $G_F$  té un arc des de la negació d' $\alpha$  a  $\beta$ , i un de la negació de  $\beta$  a  $\alpha$ .

Tingueu en compte que la clàusula  $(\alpha \vee \beta)$  és equivalent a qualsevol de les implicacions  $\neg\alpha \Rightarrow \beta$  o  $\neg\beta \Rightarrow \alpha$ . En aquest sentit,  $G_F$  representa totes les implicacions directes en  $F$ .

- a) Realitzeu aquesta construcció per a la instància de 2SAT indicada amunt.
- b) Demostreu que si  $G_F$  té una component connexa forta que conté  $x$  i  $\neg x$  per a alguna variable  $x$ , llavors no és satisfactible.
- c) Ara demostreu la inversa: és a dir, que si no hi ha cap component connexa forta que contingui tant un literal com la seva negació, llavors la instància ha de ser satisfactible.
- d) A la vista del resultat previ, hi ha un algorisme de temps lineal per resoldre 2SAT?

- 👁 6. En una festa, un convidat es diu que és una celebritat si tothom el coneix, però ell no coneix a ningú (tret d'ell mateix). Les relacions de coneixença donen lloc a un graf dirigit: cada convidat és un vèrtex, i hi ha un arc entre  $u$  i  $v$  si  $u$  coneix a  $v$ . Doneu un algorisme que, donat un graf dirigit representat amb una matriu d'adjacència, indica si hi ha o no cap celebritat. En el cas que hi sigui, cal dir qui és. El vostre algorisme ha de funcionar en temps  $O(n)$ , on  $n$  és el nombre de vèrtexs.

- 👁 7. Llisteu les següents funcions en ordre *creixent*, és a dir, si l'ordre és  $f_1; f_2; \dots$ , aleshores  $f_2 = \Omega(f_1); f_3 = \Omega(f_2)$ ; etc.

$$(\log n)^{100}, n \log n, 3^n, \frac{n^2}{\log n}, n2^n, 0,99^n, n^3, \sqrt{n}.$$

- 👁 8. Digueu si cadascuna de les afirmacions següents són certes o falses (i per què).

- a) Asimptòticament  $(1 + o(1))^{\omega(1)} = 1$
- b) Si  $f(n) = (n + 2)n/2$  aleshores  $f(n) \in \Theta(n^2)$ .
- c) Si  $f(n) = (n + 2)n/2$  aleshores  $f(n) \in \Theta(n^3)$ .
- d)  $n^{1,1} \in O(n(\lg n)^2)$
- e)  $n^{0,01} \in \omega((\lg n)^2)$

9. Digueu si la següent demostració de

$$\sum_{k=1}^n k = O(n)$$

és certa o no (i justifiqueu la vostra resposta).

**Demostració:** Per a  $k = 1$ , tenim  $\sum_{k=1}^1 k = 1 = O(1)$ . Per hipòtesi inductiva, assumim  $\sum_{k=1}^n k = O(n)$ , per a una certa  $n > 1$ . Llavors, per a  $n + 1$  tenim

$$\sum_{k=1}^{n+1} k = n + 1 + \sum_{k=1}^n k = n + 1 + O(n) = O(n).$$

10. Demostreu que  $\sum_{j=1}^{\lg n} 4^j = O(n^2)$ .
11. Donat el següent algorisme per conduir un robot, que té com a entrada un enter no negatiu  $n$ ,

Funció CAMINAR- $(n)$

si  $n \leq 1$  retornar i aturar-se  
 caminar nord  $n$  metres  
 caminar est  $n$  metres  
 caminar sud  $n$  metres  
 caminar oest  $n - 1$  metres  
 caminar nord 1 metre  
 CAMINAR  $(n - 2)$

Sigui  $C(n)$  el nombre de metres que el robot camina quan executem l'algorisme amb paràmetre  $n$ . Doneu una estimació asimptòtica del valor de  $C(n)$ .

12. Tenim un conjunt de robots que es mouen en un edifici, cadascun d'ells és equipat amb un transmissor de ràdio. El robot pot utilitzar el transmissor per comunicar-se amb una estació base. No obstant això, si els robots són massa a prop un de l'altre hi ha problemes amb la interferència entre els transmissors. Volem trobar un pla de moviment dels robots, de manera que puguin procedir al seu destí final, sense perdre mai el contacte amb l'estació base.

Podem modelar aquest problema de la següent manera. Se'ns dóna un graf  $G = (V, E)$  que representa el plànol d'un edifici, hi ha dos robots que inicialment es troben en els nodes  $a$  i  $b$ . El robot en el node  $a$  vol viatjar a la posició  $c$ , i el robot en el node  $b$  vol viatjar a la posició  $d$ . Això s'aconsegueix per mitjà d'una planificació: a cada pas de temps, el programa especifica que un dels robots es mou travessant una aresta. Al final de la planificació, els dos robots han d'estar en les seves destinacions finals.



Una planificació és *lliure* d'interferència si no hi ha un punt de temps en el qual els dos robots ocupen nodes que es troben a distància menor de  $r$ , per a un valor determinat del paràmetre  $r$ .

Doneu un algorisme de temps polinomial que decideixi si hi ha una planificació lliure donats, el graf, les posicions inicials i finals dels robots i el valor de  $r$ .


- 👁 13. El quadrat d'un graf  $G = (V, E)$  és un altre graf  $G' = (V, E')$  on  $E' = \{(u, v) \mid \exists w \in V, (u, w) \in E \wedge (w, v) \in E\}$ . Dissenyeu i analitzeu un algorisme que, donat un graf representat amb una matriu d'adjacència, calculi el seu quadrat. Feu el mateix amb un graf representat amb llistes d'adjacència.
- 👁 14. Es diu que un vèrtex d'un graf connex és un *punt d'articulació* del mateix si, en suprimir aquest vèrtex i totes les arestes que hi incideixen, el graf resultant deixa de ser connex. Per exemple, un graf en forma d'anell no té cap punt d'articulació mentre que tot node que no sigui fulla d'un arbre és punt d'articulació. Dissenyeu un algorisme que, donat un graf connex no dirigit  $G = (V, E)$ , indiqui quins vèrtexs del graf són punts d'articulació. Calculeu el seu cost. Indiqueu quina implementació del graf és la que proporciona un algorisme més eficient.
- 👁 15. Un graf dirigit és *fortament connex* quan, per cada parell de vèrtexs  $u, v$ , hi ha un camí de  $u$  a  $v$ . Doneu un algorisme per determinar si un graf dirigit és fortament connex.
- 👁 16. Un graf dirigit  $G = (V, E)$  és *semiconnex* si, per qualsevol parell de vèrtexs  $u, v \in V$ , tenim un camí dirigit de  $u$  a  $v$  o de  $v$  a  $u$ . Doneu un algorisme eficient per determinar si un graf dirigit  $G$  és semiconnex. Demostreu la correctesa del vostre algorisme i analitzeu-ne el cost.
- 👁 17. Definim els *k-mers* com les subcadena de DNA, amb grandària  $k$ . Per tant, per a un valor donat  $k$  podem assumir que tenim una base de dades amb tots els  $4^k$  *k-mers*. Una manera utilitzada en l'experimentació clínica per a identificar noves seqüències de DNA, consisteix a agafar mostres aleatòries de una cadena i determinar quins *k-mers* conté, on els *k-mers* es poden solapar. A partir d'aquest procés, podem reconstruir tota la seqüència de DNA.

Volem resoldre un problema més senzill. Donada una cadena  $w \in \{A, C, T, G\}^*$ , i un enter  $k$ , sigui  $S(w)$  el multi-conjunt de tots els *k-mers* que apareixen a  $w$ . Notem que  $|S(w)| = |w| - k + 1$ . Donat un multi-conjunt  $C$  de *k-mers*, trobar, si n'hi ha, la cadena de DNA  $w$  tal que  $S(w) = C$ .

Trobeu un algorisme per resoldre aquest problema i analitzeu la seva complexitat.

(Ajut: A partir de qualsevol entrada al problema de la seqüenciació, hem de construir en temps polinòmic un graf dirigit  $G$  que sigui entrada al problema del camí Eulerià, i tal que existeixi una solució al problema de la seqüenciació sii  $G$  té un camí Eulerià. Podeu considerar com a vèrtexs els  $(k - 1)$ -mers que apareixen en el

multiconjunt. Heu de decidir com definir les arestes  $(u, v)$  i demostrar la correctesa de la vostra construcció.)

-  18. El Professor JD ha corregit els exàmens finals del curs, de cara a tenir una distribució maca de les notes finals decideix formar  $k$  grups, cada grup amb el mateix nombre d'alumnes, i donar la mateixa nota a tots els alumnes que són al mateix grup. La condició més important és que qualsevol dels alumnes al grup  $i$  han de tenir nota d'examen superior a qualsevol alumne d'un grup inferior (grups de 1 fins a  $i-1$ ). L'ordre dintre de cadascun dels grups es irrelevant. Dissenyeu un algorisme que donada una taula  $A$  no ordenada, que a cada registre conté la identificació d'un estudiant amb la seva notes d'examen, divideix  $A$  en els  $k$  grups, amb les propietat descrita a dalt. El vostre algorisme ha de funcionar en temps  $O(n \lg k)$ . Al vostre anàlisi podeu suposar que  $n$  és múltiple de  $k$  i  $k$  és una potencia de 2.

-  19. Resoleu les següents recurrències


a)  $T(n) = 16T(n/2) + \binom{n}{3} \lg^4 n$

b)  $T(n) = 5T(n/2) + \sqrt{n}$

c)  $T(n) = 2T(n/4) + 6,046\sqrt{n}$

d)  $T(n) = 2T(n/2) + \frac{n}{\lg n}$

e)  $T(n) = T(n-10) + n$

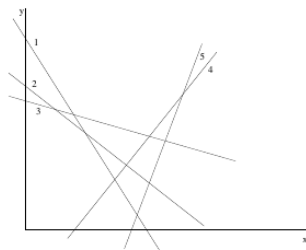
-  20. Considereu el següent algorisme de dividir-i-vèncer per al problema de *trobar un clique* en un graf no dirigit  $G = (V, A)$  (recordeu un clique és un subgraf  $C$  de  $G$  on tots els vèrtexs estan connectats entre ells).

CLIQUE( $G$ )

- 1 Enumereu els vèrtexs  $V$  com  $1, 2, \dots, n$ , on  $n = |V|$
- 2 Si  $n = 1$  tornar  $V$
- 3 Dividir  $V$  en  $V_1 = \{1, 2, \dots, \lfloor n/2 \rfloor\}$  i  $V_2 = \{\lfloor n/2 \rfloor + 1, \dots, n\}$
- 4 Sigui  $G_1 = G[V_1]$  el subgraf induït per  $V_1$  i sigui  $G_2$  el subgraf induït per  $V_2$  (les arestes de  $G_1$  són les arestes que connecten vèrtexs en  $V_1$  i similar amb  $G_2$ )
- 5 Recursivament trobeu cliques  $C_1 = \text{CLIQUE}(G_1)$  i  $C_2 = \text{CLIQUE}(G_2)$
- 6 Combineu aquests dos cliques de la manera següent:
  - 6.1 Inicialitzar  $C_1^+$  com a  $C_1$  i  $C_2^+$  com a  $C_2$
  - 6.2 Per a cada vèrtex  $v \in C_2$ , si  $v$  està connectat a tots els vèrtexs a  $C_1^+$ , aleshores afegir  $v$  a  $C_1^+$
  - 6.3 Per a cada vèrtex  $u \in C_1$ , si  $u$  està connectat a tots els vèrtexs a  $C_2^+$ , aleshores afegir  $u$  a  $C_2^+$
  - 6.4 Retorneu el més gran d'entre  $C_1^+$  i  $C_2^+$

Contesteu les següents preguntes:

- (a) Demostreu que l'algorisme CLIQUE sempre retorna un subgraf de  $G$  que és un clique.
  - (b) Doneu una expressió asimptòtica del nombre de passos de l'algorisme CLIQUE.
  - (c) Doneu un exemple d'un graf  $G$  on l'algorisme CLIQUE retorna un clique que no és de grandària màxima.
  - (d) Creieu que és fàcil modificar CLIQUE de manera que sempre done el clique màxim, sense incrementar el temps pitjor de l'algorisme? Expliqueu la vostra resposta.
21. El problema de l'eliminació de superfícies ocultes és un problema important en informàtica gràfica. Si des de la teva perspectiva, en Pepet està davant d'en Ramonet, podràs veure en Pepet però no en Ramonet. Considereu el següent problema, restringit al pla. Us donen  $n$  rectes no verticals al pla,  $L_1, \dots, L_n$ , on la recta  $L_i$  ve especificada per l'equació  $y = a_i x + b_i$ . Assumim, que no hi han tres rectes que es creuen exactament al mateix punt. Direm que  $L_i$  és *maximal* en  $x_0$  de la coordenada  $x$ , si per qualsevol  $1 \leq j \leq n$  amb  $j \neq i$  tenim que  $a_i x_0 + b_i > a_j x_0 + b_j$ . Direm que  $L_i$  és *visible* si té algun punt maximal.



Donat com a entrada un conjunt de  $n$  rectes  $\mathcal{L} = \{L_1, \dots, L_n\}$ , doneu un algorisme que, en temps  $O(n \lg n)$ , torne las rectes no visibles. A la figura de sobre teniu un exemple amb  $\mathcal{L} = \{1, 2, 3, 4, 5\}$ . Totes les rectes excepte la 2 són visibles (considerem rectes infinites).

22. Supposeu que sou consultors per a un banc que està molt amoïnada amb el tema de la detecció de frauds. El banc ha confiscat  $n$  targetes de crèdit que se sospita han estat utilitzades en negocis fraudulents. Cada targeta conté una banda magnètica amb dades encriptades, entre elles el número del compte bancari on es carrega la targeta. Cada targeta es carrega a un únic compte bancari, però un mateix compte pot tenir moltes targetes. Direm que dues targetes són *equivalents* si corresponen al mateix compte.

És molt difícil de llegir directament el número de compte d'una targeta intel·ligent, però el banc té una tecnologia que donades dues targetes permet determinar si són equivalents.





La qüestió que el banc vol resoldre és la següent: donades les  $n$  targetes, volen conèixer si hi ha un conjunt on més de  $n/2$  targetes són totes equivalents entre si. Suposem que les úniques operacions possibles que pot fer amb les targetes és connectar-les de dues en dues, al sistema que comprova si són equivalents.

Doneu un algorisme que resolgui el problema utilitzant només  $O(n \lg n)$  comprovacions d'equivalència entre targetes. Sabríeu com fer-ho en temps lineal?

- 👁 23. Donada una taula  $A$  amb  $n$  registres, on cada registre conté un enter de valor entre 0 i  $2^n$ , i els continguts de la taula estan desordenats, dissenyeu un algorisme lineal per a obtenir una llista ordenada dels elements a  $A$  que tenen valor més gran que els  $\log n$  elements més petits a  $A$ , i al mateix temps, tenen valor més petit que els  $n - 3 \log n$  elements més grans a  $A$ .
- 👁 24. Tenim una taula  $T$  amb  $n$  claus (no necessàriament numèriques) que pertanyen a un conjunt totalment ordenat. Doneu un algorisme  $O(n + k \log k)$  per a ordenar els  $k$  elements a  $T$  que són els més petits d'entre els més grans que la mediana de les claus a  $T$ .
- 👁 25. Com ordenar eficientment elements de longitud variable:
- a) Donada una taula d'enters, on els enters emmagatzemats poden tenir diferent nombre de dígit. Però sabem que el nombre total de dígit sobre tots els enters de la matriu és  $n$ . Mostreu com ordenar la matriu en  $O(n)$  passos.
  - b) Se us proporciona una sèrie de cadenes de caràcters, on les diferents cadenes poden tenir diferent nombre de caràcters. Com en el cas previ, el nombre total de caràcters sobre totes les cadenes és  $n$ . Mostreu com ordenar les cadenes en ordre alfabètic fent servir  $O(n)$  passos. (Tingueu en compte que l'ordre desitjat és l'ordre alfabètic estàndard, per exemple,  $a < ab < b$ .)
- ✎ 26. Donat un vector  $A$  amb  $n$  elements, és possible posar en ordre creixent els  $\sqrt{n}$  elements més petits i fer-ho en  $O(n)$  passos?

---

## Algoritmos Voraces

-  27. **(Revisió)** Un professor rep  $n$  sol·licituds de revisions d'examen. Abans de començar, el professor mira la llista dels  $n$  estudiants que han sol·licitat revisió i pot calcular, per a cada estudiant  $i$ , el temps  $t_i$  que utilitzarà per atendre l' $i$ -èsim estudiant. Per a l'estudiant  $i$ , el temps d'espera  $e_i$  és el temps que el professor triga a revisar els exàmens dels estudiants que fan la revisió abans que  $i$ .
- Dissenyeu un algorisme per a computar l'ordre en que s'han de revisar els exàmens dels  $n$  estudiants de manera que es minimitzi el temps total d'espera:  $T = \sum_{i=1}^n e_i$ .
-  28. Tenim un graf no dirigit  $G = (V, E)$ . Donat un subconjunt  $V' \subseteq V$  el subgraph induït per  $V'$  és el graf  $G[V'] = (V', E')$  on  $E' = E \cap (V' \times V')$ . El grau d'un vèrtex a un graf és el nombre d'arestes incidents al vèrtex. Doneu un algorisme eficient per al següent problema: donat  $G$  i un enter positiu  $k$ , trobar el subconjunt (si hi ha algun) més gran  $V'$  de  $V$ , tal que cada vèrtex a  $V'$  té grau  $\geq k$  a  $G[V']$ .
-  29. El problema de la *partició interval* (*Interval Partitioning Problem*) és similar al problema de la selecció d'activitats vist a classe però, en lloc de tenir un únic recurs, tenim molts recursos (és a dir, diverses còpies del mateix recurs). Doneu un algorisme que permeti programar totes les activitats fent servir el menor número possible de recursos.
-  30. **Streaming.** Tenim  $n$  paquets de vídeo que s'han d'enviar seqüencialment per una única línia de comunicació (això s'anomena *streaming*). El paquet  $i$ -èsim té una grandària de  $b_i$  bits i triga  $t_i$  segons a travessar, on  $t_i$  i  $b_i$  són enters positius (no es poden enviar dos paquets al mateix temps). Hem de decidir una planificació de l'ordre en què hem d'enviar els paquets de manera que, un cop tenim un ordre, no

pot haver-hi retard entre la fi d'un paquet i el començament del següent. Assumirem que comencem a l'instant 0 i finalitzem al  $\sum_{i=1}^n t_i$ . A més, el proveïdor de la connexió vol utilitzar una amplada de banda no massa gran, per tant s'afegeix la restricció següent: Per a cada enter  $t > 0$ , el nombre total de bits que enviem de temps 0 a  $t$  ha de ser  $\leq rt$ , per a una  $r > 0$  fixada (noteu que aquesta restricció no diu res per a períodes de temps que no comencen a l'instant 0). Direm que una planificació és *vàlida* si satisfà la restricció prèvia.

El problema a resoldre és el següent: donat un conjunt de  $n$  paquets, cadascun especificat per la seva grandària  $b_i$  i la durada de la seva transmissió  $t_i$ , i donat el valor del paràmetre  $r$ , hem de determinar si existeix una planificació vàlida dels  $n$  paquets. Per exemple, si  $n = 3$  amb  $(b_1, t_1) = (2000, 1)$ ,  $(b_2, t_2) = (6000, 2)$  i  $(b_3, t_3) = (2000, 1)$  amb  $r = 5000$ , aleshores la planificació 1, 2, 3 és vàlida, donat que compleix la restricció.

Per a resoldre aquest problema, heu de resoldre els apartats següents:

- a) Demostreu o doneu un contraexemple a l'enunciat de: és veritat que existeix una planificació vàlida si, i únicament si, per a cada paquet  $i$  tenim  $b_i \leq rt_i$ .
- b) Doneu un algorisme que per a una entrada de  $n$  paquets (cadascun especificat per  $(b_i, t_i)$ ) i el paràmetre  $r$ , determini si existeix una planificació vàlida. El vostre algorisme hauria de tenir complexitat polinòmica en  $n$ .

31. Sigui  $G = (V, E)$  un graf no dirigit. Un subconjunt  $C \subseteq V$  s'anomena *recobriment de vèrtexs* de  $G$  si

$$\forall \{u, v\} \in E : \{u, v\} \cap C \neq \emptyset.$$

Donat  $G = (V, E)$ , un recobriment de vèrtexs  $C$  és diu que és *minimal* si per qualsevol  $C' \subseteq V$  a  $G$ , tal que  $C' \subset C$  hem de tenir que  $C'$  no és un recobriment de vèrtexs.

Un conjunt  $C \subseteq V$  és un *recobriment de vèrtexs mínim* si  $C$  és un recobriment de vèrtexs a  $G$  amb mínima cardinalitat. (Quan  $G$  és un arbre, hi ha un algorisme polinòmic per a trobar un recobriment de vèrtexs mínim a  $G$ , però per a  $G$  generals el problema és NP-hard).

En aquest problema demostrareu que, a diferència del problema de trobar un recobriment de vèrtexs mínim, el problema de trobar un recobriment de vèrtexs minimal pot ser resolt en temps polinòmic.

- a) Demostreu que un recobriment de vèrtexs minimal no necessàriament ha de ser un recobriment de vèrtexs mínim.
- b) Demostreu que tot recobriment de vèrtexs mínim també és minimal.
- c) Doneu un algorisme polinòmic per trobar un recobriment de vèrtexs minimal a  $G$ .

32. COOPC és una companyia de lloguer de cotxes que vol diversificar el seu negoci per tal de competir en els desplaçaments curts. La companyia té oficines distribuïdes a Barcelona i un dipòsit central de cotxes al Prat. COOPC vol un mètode que li permeti processar les peticions de trajectes curts rebuts pel dia següent. Per això, vol determinar el nombre de cotxes que ha de deixar a cada oficina a l'inici del dia dedicats a trajectes curts. Aquesta assignació ha de permetre tenir suficients cotxes disponibles al llarg del dia (en cadascuna de les oficines) per tal de poder cobrir tots els trajectes curts del dia. A més, COOPC vol minimitzar el nombre de cotxes destinats durant el dia a desplaçaments curts.

La informació de la qual disposa COOPC per a cada sol·licitud de trajecte curt per al proper dia és la següent: una tupla  $(l, t, l', t')$  on el parell  $(l, t)$  indica l'oficina i l'hora de recollida del vehicle i el parell  $(l', t')$  indica l'oficina i l'hora de devolució.

A efecte d'aquesta aproximació COOPC assumeix que tots els cotxes són idèntics i que disposa d'una flota prou gran per a poder cobrir qualsevol nivell de demanda de trajectes curts. A més, assumeix també que els cotxes seran recollits i retornats puntualment a les hores i locals estipulats.

Dissenyeu un algorisme voraç que permeti obtenir el nombre de cotxes que s'han d'assignar a cada oficina, de manera que, al llarg del dia, sempre hi hagi almenys un cotxe disponible en una oficina a l'hora en què algun client l'ha de recollir d'allà. Tenint en compte que mantenir immobilitzat un cotxe té un cost alt, l'assignació obtinguda ha de garantir el requisit de disponibilitat i ha d'assignar el menor nombre possible de cotxes a cada oficina.

33. (**Vídeo blocs**) Tenim  $n$  blocs de vídeo que s'han d'enviar per una única línia de comunicació a mida que es filmen. El bloc  $i$ -èsim té una grandària de  $b_i$  bits i està disponible per iniciar la seva transmissió a temps  $s_i$ , on  $s_i$  i  $b_i$  són enters positius. Podeu assumir que transmetre un bit requereix una unitat de temps. A més, la transmissió del paquet  $i$  no pot començar abans de l'instant de temps  $s_i$ , però sí després.

Encara que no es poden enviar dos paquets al mateix temps, la transmissió d'un bloc de vídeo es pot suspendre a qualsevol instant per completar-la més endavant. Per exemple, si tenim dos blocs de vídeo amb  $b_1 = 5$ ,  $s_1 = 2$ ,  $b_2 = 3$  i  $s_2 = 0$  podríem enviar a temps 0 els 3 bits del segon paquet i a temps 3 el 5 bits del bloc 1 o bé, enviar a temps 0, 2 bits del bloc 2, després els 5 bits del bloc 1 i finalitzar amb el bit restant del bloc 2. També seria possible enviar a temps 0, 2 bits del bloc 2, després 2 bits del bloc 1, finalitzar el bit restant del bloc 2, i finalitzar els bits del bloc 1. Observeu que no podem començar la transmissió de cap bit del bloc 1 fins al instant 2.

Una vegada fixada la planificació de l'ordre en què enviarem els bits dels diferents blocs, ens interessa el temps en què finalitza la transmissió de cada bloc. Anomenem  $f_i$  al temps en què finalitza la transmissió del bloc  $i$ . Observeu que, per a l'exemple anterior, la primera planificació dona  $f_1 = 8$  i  $f_2 = 3$ , la segona planificació dona

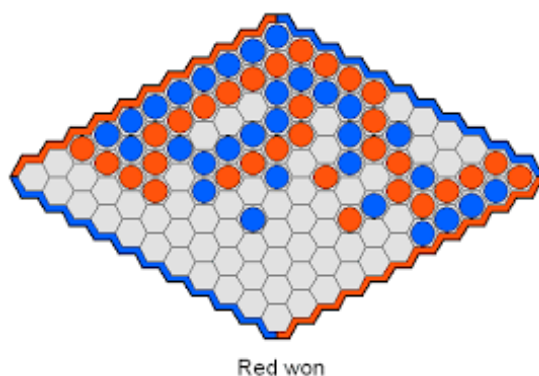
$f_1 = 7$  i  $f_2 = 8$ , mentre que la tercera dona  $f_1 = 8$  i  $f_2 = 5$ .

Volem obtenir una planificació de la transmissió dels  $n$  blocs de vídeo que minimitzi la suma dels temps de finalització de la transmissió dels blocs, és a dir  $\sum_{i=1}^n f_i$ . Doneu un algorisme que, per a una entrada de  $n$  blocs, cadascun especificat per  $(b_i, s_i)$ , determini una planificació (començant a temps 0) resolgui el problema proposat.

34. Un *bottleneck spanning tree*  $T$  d'un graf no dirigit i ponderat  $G = (V, E, w)$ , on  $w : E \rightarrow \mathbb{R}^+$ , és un arbre d'expansió de  $G$  on el pes més gran és mínim sobre tots els arbres d'expansió de  $G$ . Diem que el valor d'un bottleneck spanning tree és el pes de la aresta de pes màxim a  $T$ .
- a) Demostreu la correctesa o trobeu un contraexemple pels enunciats següents:
    - *Un bottleneck spanning tree és també un arbre d'expansió mínim.*
    - *Un arbre d'expansió mínim és també un bottleneck spanning tree.*
  - b) Doneu un algorisme amb cost  $O(|V| + |E|)$  que donat un graf  $G$  i un enter  $b$ , determini si el valor d'un bottleneck spanning tree és  $\leq b$ .
35. Demostreu que un graf  $G$  té un únic MST si, per a tot tall  $C$  de  $G$ , existeix una única aresta  $e \in C$  amb valor mínim. Demostreu que el reciproc no és cert, i.e. pot ser el cas de que per un o més talls  $C$  tinguem més d'una aresta mínim pes, però que el MST sigui únic.
36. Tenim un graf connex no dirigit  $G = (V, E)$  on cada node  $v \in V$  té associat un valor  $\ell(v) \geq 0$ ; considerem el següent joc unipersonal:
- a) Els nodes inicialment no estan marcats i la puntuació del jugador és 0.
  - b) El jugador selecciona un node  $u \in V$  no marcat. Sigui  $M(u)$  el conjunt de veïns de  $u$  a  $G$  que ja han sigut marcats. Aleshores, s'afegeix a la puntuació del jugador el valor  $\sum_{v \in M(u)} \ell(v)$  i marquem  $u$ .
  - c) El joc es repeteix fins que tots els nodes siguin marcats o el jugador decideixi finalitzar la partida, deixant possiblement alguns nodes sense marcar.
- Per exemple, suposeu que el graf té tres nodes  $A, B, C$  on  $A$  està connectat a  $B$  i  $B$  amb  $C$ , amb  $\ell(A) = 3$ ,  $\ell(B) = 2$ ,  $\ell(C) = 3$ . En aquest cas, una estratègia òptima seria marcar primer  $A$ , després  $C$  i finalment  $B$ . Aquest ordre dona al jugador una puntuació total de 6.
- a) És possible obtenir una puntuació millor deixant algun dels nodes sense marcar? Justifiqueu la vostra resposta.
  - b) Dissenyeu un algorisme voraç per tal d'obtenir la millor puntuació possible. Justifiqueu la seva correctesa i doneu-ne el cost.
  - c) Suposeu ara que  $\ell(v)$  pugui ser negatiu. Continua el vostre algorisme proporcionant la puntuació màxima possible?



- d) Considereu la següent modificació del joc per aquest cas en què els nodes poguessin tenir valors negatius: eliminem primerament de  $G$  tots els nodes  $v \in V$  amb  $\ell(v) < 0$ , per tal de seguidament executar el vostre algorisme sobre el graf resultant d'aquesta eliminació. Doneu un exemple on aquesta variació no proporcioni la màxima puntuació possible.
37. El joc d'HEX té com un dels seus inventors, al matemàtic John Nash. En aquest joc, dos jugadors, un amb color negre i l'altre amb color blanc, fan torns on a cada torn el jugador que li toca col·loca una pedra del seu color a una posició encara buida, a una xarxa  $n \times n$  de cel·les hexagonals. Un cop col·locada una pedra, no es pot moure. L'objectiu de cada jugador és connectar els costats del mateix color a la graella, amb un camí continu fet amb les seves pedres. Dues cel·les es consideren connectades si comparteixen una vora les dues tenen la pedra amb el mateix color. Descriviu un esquema eficient que determini, després de cada jugada, si el jugador que acaba de jugar ha guanyat el joc d'HEX.



38. (SOS Rural). Considerad un camino rural en el Pirineo con casas muy dispersas a lo largo de él. Por motivos de seguridad se quieren colocar estaciones SOS en algunos puntos de la carretera. Los expertos indican que, teniendo en cuenta las condiciones climáticas de la zona, se debería garantizar que cada casa se encuentre como máximo a una distancia de 15km, siguiendo la carretera, de una de las estaciones SOS.
- Proporcionad un algoritmo eficiente que consiga este objetivo utilizando el mínimo número de estaciones SOS posibles. Justificad la corrección y el coste de vuestro algoritmo e indicad la complejidad en tiempo de la solución propuesta.
39. El centre de documentació de la UE gestiona el procés de traducció de documents pels membres del parlament europeu. En total han de treballar amb un conjunt de  $n$  idiomes. El centre ha de gestionar la traducció de documents escrits en un idioma a tota la resta d'idiomes.

Per fer les traduccions poden contractar traductors. Cada traductor està especialitzat en dos idiomes diferents; és a dir, cada traductor pot traduir un text en un

dels dos idiomes que domina a l'altre, i viceversa. Cada traductor té un cost de contractació no negatiu (alguns poden treballar gratis).

Malauradament, el pressupost per a traduccions és massa petit per contractar un traductor per a cada parell d'idiomes. Per tal d'optimitzar la despesa, n'hi hauria prou en establir cadenes de traductors; per exemple: un traductor anglès  $\leftrightarrow$  català i un català  $\leftrightarrow$  francès, permetria traduir un text de l'anglès al francès, i del francès a l'anglès. Així, l'objectiu és contractar un conjunt de traductors que permetessin la traducció entre tots els parells dels  $n$  idiomes de la UE, amb cost total de contractació mínim.

El matemàtic del centre els hi ha suggerit que ho poden modelitzar com un problema en un graf amb pesos  $G = (V, E, w)$ .  $G$  té un node  $v \in V$  per a cada idioma i una aresta  $(u, v) \in E$  per a cada traductor (entre els idiomes  $u$  i  $v$  de la seva especialització); el pes de cada aresta seria el cost de contractació del traductor en qüestió. En aquest model, un subconjunt de traductors  $S \subseteq E$  permet portar a terme la feina si al subgraf  $G_s = (V, S)$  hi ha un camí entre tot parell de vèrtexs  $u, v \in V$ ; en aquest cas direm que  $S$  és una *selecció vàlida*. Aleshores, d'entre totes les seleccions vàlides han de triar una amb cost mínim.

- a) Demostreu que quan  $S$  és una selecció vàlida de cost mínim,  $G_s = (V, S)$  no té cicles.
- b) Proporcioneu un algorisme eficient per a resoldre el problema. Justifiqueu la seva correctesa i el seu cost.

- 👁 40. Tenim un tauler de dimensions  $n \times n$ , amb  $n$  fitxes col·locades a certes posicions  $(x_1, y_1), \dots, (x_n, y_n)$  i una fila  $i$ . Volem determinar el mínim nombre de moviments necessaris per a posar les  $n$  fitxes a la fila  $i$  (una a cada casella). Els moviments permesos són: cap a la dreta, esquerra, amunt i avall. Durant aquests moviments es poden apilar tantes fitxes a la mateixa posició com calgui. Però en finalitzar ha de quedar una fitxa per casella.

Pista: El nombre de moviments verticals (amunt/avall) necessaris es pot calcular fàcilment.

- 👁 41. La cadena de sota és el títol d'una cançó codificat amb codis de Huffman.

0011000101111101100111011101100000100111010010101

Donades les freqüències de les lletres que es donen a la taula de sota, obtingueu els codis de Huffman i feu-los servir per decodificar el títol.

lletra	a	h	v	w	'	'	e	t	l	o
freqüència	1	1	1	1	2	2	2	3	3	3

Tingueu en compte que, quan hi ha múltiples eleccions, cal un criteri de desempat. Per això podeu suposar que quan es fusionan dos arbres es posa a la esquerra el

que té, la fulla de més a la esquerra, més a l'esquerra a l'ordre inicial donat a la taula. A més s'assigna a la branca esquerra el símbol 0 i a la dreta el símbol 1.

42. Tenim un alfabet  $\Sigma$  on per a cada símbol  $a \in \Sigma$ ,  $p_a$  es la probabilitat que aparegui el caràcter  $a$ . Demostreu que, per a qualsevol símbol  $a \in \Sigma$ , la seva profunditat en un arbre prefix que produeix un codi de Huffman òptim és  $O(\lg \frac{1}{p_a})$ . (Ajuts: en un arbre prefix que s'utilitzi per a dissenyar el codi Huffman, la probabilitat d'un nus és la suma de les probabilitats dels fills. La probabilitat de l'arrel és, doncs, 1.)
43. Una tribu de matemàtics ha decidit utilitzar un alfabet molt concís de 3 símbols (més el blanc  $\flat$ ) per a comunicar-se entre ells. Utilitzant cadenes dels símbols  $\{+, -, \star\}$  els matemàtics codifiquen totes les paraules que necessiten per a comunicar-se. Donada la seqüència  $+ - - + \star \flat \star + \star - \star \flat \star$ , trobeu la freqüència de cada un dels símbols  $\{+, -, \star, \flat\}$ . Dibuixeu l'arbre de Huffman. Quina és la compressió màxima que podeu obtenir utilitzant Huffman? Quin és el guany de compressió respecte a utilitzar una compressió de longitud fixada? (i.e. utilitzant  $\{0, 1\}^2$ ). Tingueu en compte que els matemàtics es comuniquen entre si utilitzant Internet, per tant, abans d'enviar el missatge, l'ordinador converteix els símbols del seu alfabet en cadenes binàries via ASCII.
44. Sigui  $T = (V, E)$  un arbre no dirigit amb  $n$  vèrtexs. Per a  $u, v \in V$ , sigui  $d(u, v)$  el nombre d'arestes del camí de  $u$  a  $v$  en  $T$ . Definim el *centre* de  $T$  com el vèrtex

$$c = \operatorname{argmin}_{v \in V} \{ \max_{u \in V} d(u, v) \}.$$

Describiu un algorisme de cost  $O(n)$  per a obtenir un centre de  $T$ .

45. CinemaVis ha de programar l'aparició d'un seguit d'anuncis a una pantalla gegant a la Plaça del Mig la diada de Sant Jordi. La tirada d'anuncis es pot iniciar a temps 0 (l'inici programat) però mai abans. A més, CinemaVis disposa d'un conjunt de  $n$  anuncis per fer la selecció. L'anunci  $i$  té una durada de 1 minut i té associat dos valors reals no negatius  $t_i$  i  $b_i$ . L'anunciat pagarà  $b_i$  euros a CinemaVis si l'anunci  $i$  s'emet a l'interval  $[0, t_i]$  i 0 euros si s'emet després. Cap dels  $n$  anuncis es pot mostrar més d'una vegada. CinemaVis vol projectar la selecció d'anuncis que li proporcionin màxim benefici. Dissenyeu un algorisme, el més eficient que podeu, per a resoldre aquest problema.
46. (Agenda) A la vostra agenda teniu una llista  $L$  de totes les tasques que heu de completar en el dia de avui. Per a cada tasca  $i \in L$  s'especifica la durada  $d_i \in \mathbb{N}$  que indica el temps necessari per a completar-la i un factor de penalització  $p_i \in \mathbb{Z}^+$  que n'agreuja el retard. Heu de determinar en quin ordre realitzar totes les tasques per obtenir el resultat que menys penalització total acumuli.

Tingueu en compte que:

- en un instant de temps només podeu realitzar una única tasca,

- una vegada comenceu a fer una tasca, heu de continuar-la fins a finalitzar-la, i
- s'han de completar totes les tasques.

El criteri d'optimització és la penalització total que s'acumula. La penalització real associada a una tasca  $i \in L$  és el temps de finalització  $t_i$  de la seva realització, multiplicat per la seva penalització  $p_i$ . El temps de finalització  $t_i$  es correspon al temps transcorregut des de l'inici de la jornada laboral (és a dir, des de l'instant de temps 0) fins al moment en que s'ha finalitzat la tasca.<sup>1</sup>

Considereu l'algorisme voraç que programa les tasques en ordre decreixent de factor de penalització  $p_i$ . Determineu si aquest algoritme resol el problema. En cas que no ho faci, proporcioneu un algorisme (tant eficient com pugueu) per resoldre'l.

- ✎ 47. Doneu un algorisme que resolgui el següent problema i doneu la seva complexitat en funció de  $n$ . Justifiqueu-ne la correctesa:

Volem anar de Barcelona a Paris seguint l'autopista a través de Lyon, i tenim  $n$  benzineres,  $B_1 \dots, B_n$ , al llarg d'aquesta ruta. Amb el diposit ple, el vostre cotxe pot funcionar  $K$  km. La benzinera  $B_1$  és a Barcelona, i cada  $B_i$ ,  $2 \leq i \leq n$  és a  $d_i < K$  km. de la benzinera  $B_{i-1}$ . La benzinera  $B_n$  és a Paris. Quina és l'estratègia per aturar-se el mínim nombre de cops al llarg del viatge?

- 👁 48. El problema del Tall Maxim (MAX-CUT). Donat un graf no dirigit  $G = (V, E)$ , el problema del maximum cut (MAX-CUT) es trobar la partició  $S \cup \bar{S}$  de  $V$  tal que maximitze el nombre d'arestes entre  $S$  i  $\bar{S}$  ( $C(S, \bar{S})$ ), on  $S \subseteq V$  i  $\bar{S} = V - S$ . La maximització entre totes les possibles particiones de  $V$ . El problema del MAX-CUT és NP-hard en general. Donat  $G = (V, E)$  considereu el següent algorisme golafre

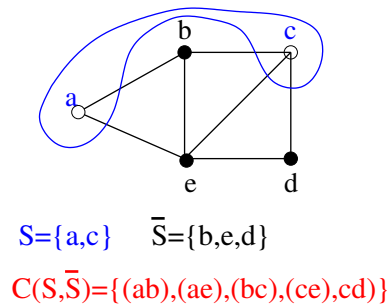


Figura 2.1: Exemple de MAX-CUT, amb cost òptim 5

(greedy) pel problema del MAX-CUT:

<sup>1</sup>Observeu que, segons les restriccions del problema, la realització d'una tasca  $i \in L$  amb temps de finalització  $t_i$  haurà començat a l'instant  $t_i - d_i$ .

- a) Ordeneu els vèrtexs en ordre decreixent respecte al seu grau (nombre veïns). Considereu el resultat de l'ordenació s'escriu com a  $(v_1, v_2, \dots, v_n)$ .
  - b) Inicialment tenim  $S = \emptyset = \bar{S}$ . Al primer pas  $v_1 \in S$ .
  - c) Al pas  $i$ -èsim col·loquem  $v_i$  a  $S$  o  $\bar{S}$  de manera que maximitze  $|C(S, \bar{S})|$ .
- a) Demostreu la correctessa i doneu la complexitat de l'algorisme golafre?
- b) Demostreu que aquest algorisme golafre és una 2-aproximació al MAX-CUT
49. Donat com a entrada un graf no dirigit  $G = (V, E)$ , definim el problema de GST (graf sense triangles) com el problema de seleccionar el màxim nombre d'arestes  $E$ , de manera que el graf  $G' = (V, E')$  no contingui cap triangle (i.e. no hi han 3 vèrtexs  $x, y, z$  tal que  $(x, y), (x, z)$  i  $(y, z)$  siguin arestes a  $G'$ ). Aquest problema és NP-hard. Dissenyeu un algorisme que done una 2-aproximació al problema. Quina és la complexitat del vostre algorisme?
50. Consideramos el siguiente escenario: tenemos un conjunto de  $n$  ciudades con distancias mínimas entre ellas (verifican la desigualdad triangular). Queremos seleccionar un subconjunto  $C$  de  $k$  ciudades en las que queremos ubicar un centro comercial. Asumiendo que las personas que viven en una ciudad comprarán en el centro comercial más próximo se quiere buscar una ubicación  $C$  de manera que todas las ciudades tengan un centro comercial a distancia menor que  $r$  en la que intentamos minimizar  $r$  sin perder cobertura. Para ello diremos que  $C$  es un  $r$ -recubrimiento si todas las ciudades están a distancia como mucho  $r$  de una ciudad en  $C$ . Sea  $r(C)$  el mínimo  $r$  para el que  $C$  es un  $r$ -recubrimiento. Nuestro objetivo es encontrar  $C$  con  $k$  vértices para el que  $r(C)$  es mínimo.
- a) Demuestra que si  $k \geq n$ , la solución formada por todas las ciudades es óptima. A partir de ahora asumiremos que  $k \leq n$ .
  - b) Diseña un algoritmo que dado  $C$  calcule  $r(C)$ . Analiza su coste.
  - c) El problema es NP-difícil?
  - d) Suponiendo que  $S$  es el conjunto de ciudades, considera el siguiente algoritmo:
 

```

      Seleccionar cualquier ciudad  $s \in S$  y define  $C = \{s\}$ 
      while  $|C| \neq k$  do
        seleccionar una ciudad  $s \in S$  que maximiza la distancia de  $s$  a  $C$ ;
         $C = C \cup \{s\}$ ;
      end while;
      return C
      
```
- Demuestra que es un algoritmo de aproximación con tasa de aproximación 2.
51. Ens donen un conjunt de treballs  $S = \{a_1, a_2, \dots, a_n\}$ , a on per a completar el treball  $a_i$  es necessiten  $p_i$  unitats de temps de processador. Únicament tenim un

ordinador amb un sol processador, per tant a cada instant únicament podem processar una treball. Sigui  $c_i$  el temps on el processador finalitza de processar  $a_i$ , que dependrà dels temps dels treballs processats prèviament. Volem minimitzar el temps "mitja" necessari per a processar tots els treballs (el temps amortitzat per treball), es a dir volem minimitzar  $\frac{\sum_{i=1}^n c_i}{n}$ . Per exemple, si tenim dos treballs  $a_1$  i  $a_2$  amb  $p_1 = 3, p_2 = 5$ , i processem  $a_2$  primer, aleshores el temps mitja per a completar els dos treballs és  $(5 + 8)/2 = 6,5$ , però si processem primer el treball  $a_1$  i després  $a_2$  el temps mitja per processar els dos treballs serà  $(3 + 8)/2 = 5,5$ .

- a) Considerem que la computació de cada treball no es pot partir, es a dir quan comença la computació de  $a_i$  les properes  $p_i$  unitats de temps s'ha de processar  $a_i$ . Doneu un algorisme que planifiqui la computació dels treballs a  $S$  de manera que minimitzi el temps mitja per a completar tots els treballs. Doneu la complexitat del vostre algorisme i demostreu la seva correctesa.
- b) Considereu ara el cas de que no tots els treballs a  $S$  estan disponibles des de el començament, es a dir cada  $a_i$  porta associat un temps  $r_i$  fins al que l'ordinador no pot començar a processar  $a_i$ . A més, podem suspendre a mitges el processament d'un treball per a finalitzar més tard. Per exemple si tenim  $a_i$  amb  $p_i = 6$  i  $r_i = 1$ , pot començar a temps 1, el processador aturar la seva computació a temps 3 i tornar a computar a temps 10, aturar a temps 11 i finalitzar a partir del temps 15. Doneu un algorisme que planifiqui la computació dels treballs a  $S$  de manera que es minimitzi el temps mitja per a completar tots els treballs.

---

## Soluciones

### 3.1. Repaso de Conceptos Básicos

**Solución 1.1** L'algorisme recursiu es descriu en l'Algorisme FINDPEAK. Donada la matriu  $A$ , la resposta s'obté amb una crida amb  $i = 1$  i  $j = n$ . L'algorisme és una cerca binària, en cada pas, comparem els dos elements intermedis i veurem si estem en la part creixent o decreixent. El cas base ressol el problema d'obtenir la posició del màxim, però per a una entrada amb mida constant.

```
function FINDPEAK( $A, i, j$ )  
     $n = j - i + 1$   
    if  $n \leq 5$  then  
        return POSMAX( $A, i, j$ )  
    end if  
     $k = (i + j)/2$   
    if  $A[k] < A[k + 1]$  then  
        return FINDPEAK( $A, k + 1, j$ )  
    else  
        return FINDPEAK( $A, i, k$ )  
    end if  
end function
```

**Correctesa:** Volem trobar l'índex  $p$ . Si  $A[k] < A[k + 1]$ , sabem que  $A[i] < \dots < A[k]$  per  $i < k$  i podem prescindir de forma segura els elements  $A[i \dots k]$ . De la mateixa manera, si  $A[k] > A[k + 1]$ , sabem que  $A[k + 1] > \dots > A[j]$  per  $j > k + 1$  i podem descartar amb seguretat els elements  $A[k + 1 \dots j]$ . La posició de  $p$  coincideix amb la del valor màxim al vector, per tant el cas base és correcte.

**Cost temporal:** El cas base té cost constant. A cada pas, es redueix la mida del problema a la meitat i, a més, el cost de les operacions és constant. Així, tenim la recurrència  $T(n) = T(n/2) + c$  per a alguna constant  $c$ . Sabem que, com a la cerca binària,  $T(n) = O(\log n)$ .

**Solució 1.12** Per resoldre el problema considerarem l'espai de configuracions on els robots es poden moure. És a dir, el conjunt de parells de posicions que estan a distància més gran o igual que  $r$ :

$$C = \{(u, v) \mid u, v \in V \text{ i } d(u, v) \geq r\}$$

Podem considerar la relació entre configuracions definida pels moviments permesos. Així tenim

$$M = \{((u, v), (u', v')) \mid (u, v), (u', v') \in C \text{ i } ((u = u' \text{ i } (v, v') \in E) \text{ o } (v = v' \text{ i } (u, u') \in E))\}.$$

A l'espai de configuracions podem considerar el graf  $\mathcal{G} = (C, M)$  on dos configuracions son veïnes si i només si un del robots pot canviar de posició sense interferir amb la posició de l'altre.

Els robots són inicialment a la configuració  $(a, b)$  i s'han de desplaçar amb moviments vàlids fins a la configuració  $(c, d)$ . Això serà possible únicament si hi ha un camí de  $(a, b)$  a  $(c, d)$ . D'acord amb el raonament anterior tenim que comprovar hi ha un camí entre dos vèrtexs a  $\mathcal{G}$ . Podem detectar-ho amb un BFS en tems  $O(|C| + |M|)$ .

Per calcular el cost hem de tenir en compte la mida de l'entrada. Si  $G = (V, E)$  i  $n = |V|$  i  $m = |E|$ , tenim  $|C| \leq n^2$  i  $|M| \leq 2m$ . Suposant que ens donen  $G$  mitjançant llistes d'adjacència la mida de l'entrada és  $n+m$ . Construir una descripció de  $\mathcal{G}$  mitjançant llistes d'adjacència té cost  $O(n^2 + m)$ . Fer un BSF sobre  $\mathcal{G}$  té cost  $O(n^2 + m)$ . El cost total es  $O(n^2 + m)$  però  $m \leq n^2$ . Llavors l'algorisme proposat té cost  $O(n^2)$ .

**Solució 1.18** Sigui AGRUPAR l'algorisme recursiu que, té com a entrada una taula de alumnes-notes  $N$  i dos enters  $\ell$  i  $t$ , i fa el següent:

- Mentre  $\ell \neq 1$  troba la mediana de  $A$  i fa una partició al seu voltant en temps  $O(|N|)$ .
- Considerem la sub-taula  $N_e$  esquerra i la sub-taula dreta  $N_d$ .
- Cridem recursivament AGRUPAR( $N_e, \ell/2, 2t$ ) i AGRUPAR( $N_d, \ell/2, 2t + 1$ ).
- Quan  $\ell = 1$ , la taula constitueix la partició  $t$ .

La crida inicial la farem amb  $N$ ,  $\ell = k$  i  $t = 0$ . La correctesa ve de com particionem els elements. Sempre tenim dos meitats i els elements a  $N_e$  són més petits que la mediana i els elements a  $N_d$  són més grans o iguals que la mediana. Aconseguirem  $\ell = 1$  després de  $\lg k$  iteracions, en aquell moment la taula té  $n/k$  elements. La variable  $t$  comptabilitza l'ordre de las crides. Al primer nivell tenim només una taula i  $t = 0$ . Al segon tindrem



dos taules, la de l'esquerra etiquetada amb 0 i la de la dreta amb 1. Al següent nivell, tindrem 0,1,2,3 (e-e,e-d,d-e,d-d). Llavors  $t$  comptabilitza l'ordre correcte de les particions per garantir la propietat requerida.

El cost de l'algorisme és  $T(n, k) = 2T(n/2, k/2) + \Theta(n)$  amb  $T(n, 1) = \Theta(1)$ , per a tot  $n$ . Desplegant la recursió tenim

$$\begin{aligned} T(n, k) &= 2T(n/2, k/2) + cn = 4T(n/4, k/4) + 2c(n/2) + cn \\ &= 4T(n/4, k/4) + 2cn = k + cn \lg k. \end{aligned}$$

llavors,  $T(n) = \Theta(n \lg k)$ .

**Solució 1.26** Seleccionar l'element  $\sqrt{n}$ -èsim i particionar al voltant, d'aquest element (cost  $O(n)$ ). Ordenar la part esquerra en  $O(\sqrt{n}^2)$ .

Alternativament, construir un min-heap en  $O(n)$  i extreure el mínim element  $\sqrt{n}$  cops, el nombre de passos és  $O(n + \sqrt{n} \lg n)$ .

## 3.2. Algoritmos Voraces

**Solució 2.27** L'algorisme ordena els estudiants en ordre creixent de  $t_i$ .

Per veure que aquesta ordenació és òptima farem servir un argument d'intercanvi. Suposem que els estudiants estan ordenats de forma òptima y que aquesta ordenació no és la nostre. Llavors hem de tenir un estudiant  $i$  pel què  $t_i > t_{i+1}$ . Si calculem el temps de espera de  $i$  i de  $i+1$  tenim

$$e_i = \sum_{j < i} t_j \text{ i } e_{i+1} = \sum_{j < i+1} t_j = e_i + t_j.$$

Si intercanvien la posició de l'estudiant  $i$  amb la del  $i+1$  els temps d'espera dels estudiants amb  $j < i$  o  $j > i+1$  no canviam (d'acord amb la definició). Després del intercanvi el temps d'espera del estudiant  $i+1$  és  $e_i$  i el del estudiant  $i$  és  $e_i + t_{i+1}$ . Si sumen aquest temps tenim

$$e_i + e_i + t_{i+1} < e_i + e_i + t_i = e_i + e_{i+1}.$$

Per tant el temps d'espera total ha millorat i en conseqüència arribem a que l'ordenació no és òptima.

**Solució 2.33** Nuestro algoritmo voraz utilizará la siguiente regla: en cada instante de tiempo programamos el envío de un bit del paquete disponible al que le quedan menos bits por finalizar la transmisión.

Observemos que si una solución óptima no sigue esta regla buscamos el primer instante de tiempo  $t$ , en esta planificación, en que se envía un bit de un paquete  $i$  al que le faltan  $m_i$  bits por transmitir mientras que hay un paquete  $j$  disponibles a tiempo  $t$  al que le faltan  $m_j$  bits por transmitir a tiempo  $t$  con  $m_j < m_i$ . Consideramos ahora los instantes de tiempo,  $\geq t$ , en que se envían los bits de  $i$  y de  $j$ . Podemos iniciar la

transmisión de  $j$  a tiempo  $t$ , retrasando todos sus bits, al tiempo del bit de  $j$  previo. Compensar, retrasando la transmisión de los bits de  $i$  hasta llegar a utilizar el espacio dejado por el último bit de  $j$ . La nueva planificación es válida y proporciona nuevos tiempos de finalización  $f'_i, f'_j$ . Si  $f_j < f_i$ , al replanificar los paquetes tenemos  $f'_j < f_j$  y  $f'_i = f'_j$ . En este caso la planificación no sería óptima. Si  $f_j > f_i$ ,  $f'_j = f_i$  y  $f'_i = f'_j$ , la suma no cambia, por lo que la nueva planificación sigue siendo óptima. Realizando los intercambios correspondientes podemos conseguir una planificación óptima con el criterio de nuestro algoritmo voraz.

Para implementar el algoritmo voraz tenemos que ir con cuidado para que el número de pasos sea polinómico en el número de bloques y no en función del tiempo de finalización de la planificación que es función de los valores de los números  $s_i$  i  $t_i$ .

Para determinar eficientemente los paquetes activos ordenamos los paquetes en orden creciente de  $s_i$ , en caso de empate por orden creciente de  $b_i$ . Mantendremos una cola de prioridad, con clave el número de bits que faltan por enviar, de los paquetes disponibles y cuya transmisión no ha terminado. Por otra parte mantendremos dos índices de vídeos  $v_a$  y  $v_s$  (actual y siguiente), tres contadores de tiempo,  $t$  tiempo actual,  $t_a$  tiempo previsto finalización  $v_a$ ,  $t_s$  tiempo en que  $v_s$  estará disponible.

Inicialmente  $t = t_a = t_s = s_1$ ,  $v_s = 1$ ,  $v_a = 0$ . Iremos avanzando el tiempo de acuerdo con diferentes eventos:

- Si  $t_a = t_s$ , introducimos en la cola los paquetes  $i$  con  $s_i = t_s$  con clave  $b_i$ .  $v_s$  será el primer paquete no introducido en la cola y  $t_s = s_{v_s}$ .

Extraemos el min de la cola en  $v_a$  incrementamos  $t_a$  con el número de bits que faltan de transmitir de  $v_a$ .

- Si  $t_a < t_s$ , planificamos los bits que faltan  $v_a$  entre  $t$  y  $t_a$ ,  $t = t_a$ .

Si la cola no está vacía extraemos el min de la cola en  $v_a$ , actualizamos  $t_a$  con  $t$  más el número de bits que faltan de transmitir de  $v_a$ .

Si la cola está vacía, actualizamos  $t$  y  $t_a$  con el valor de  $t_s$ .

- Si  $t_a > t_s$ , transmitimos bits de  $v_a$  en los tiempos  $t, \dots, t_s - 1$ , introducimos  $v_a$  en la cola con clave el número de bits que falten por transmitir.

Actualizamos los contadores de tiempo,  $t$  y  $t_a$  para que coincidan con  $t_s$ .

La implementación es correcta ya que la prioridad en la cola solo se puede alterar cuando tenemos un nuevo vídeo disponible. Observemos que un cada paquete de vídeo entra al menos una vez en la cola. Cuando reintroducimos de nuevo un paquete, lo hacemos coincidiendo con la introducción de un paquete nuevo, pero en este caso solo reintroducimos un paquete. Esto nos da un total de  $O(n)$  inserciones en la cola de prioridad.

El coste total del algoritmo es  $O(n \log n)$ : la ordenación inicial  $O(n \log n)$  y las  $n$  inserciones en la cola de prioridad con coste  $O(n \log n)$ .

**Solució 2.39** 1. Supongamos que  $G_s = (V, S)$  es una selección válida de coste mínimo que tiene ciclos. Si eliminamos una arista  $(u, v)$  de un ciclo en  $G_s = (V, S)$  seguimos teniendo caminos entre todos los vértices, ya que podemos ir de  $u$  a  $v$  a través de lo que queda del ciclo.

Como la selección tiene coste mínimo, y eliminando una arista de un ciclo también es solución. Tenemos que todas las aristas de un ciclo tienen coste 0. Así, mientras tengamos ciclos vamos eliminando una arista de peso 0 del ciclo. Hasta que tengamos una selección válida con coste mínimo sin ciclos.

2. Por el apartado a) nos basta con buscar un árbol con peso mínimo que cubra todos los idiomas. Es decir tenemos que obtener un MST del grafo. Utilizando el algoritmo de Prim, podemos encontrarlo en tiempo  $O(n \log m)$

**Solució 2.47** L'estratègia que farem servir és esperar el màxim per a carregar benzina, però sense quedar-nos amb el dipòsit buit. Aquest criteri es pot implementar com el següent algorisme voraç:

```

function GREEDY( $d_1, \dots, d_n, K$ )
   $d = 0$ ;  $R = \emptyset$ ;  $j = 1$ 
  for  $i = 1 \dots n - 1$  do
    if  $d + d_{i+1} > K$  then
       $j := j + 1$ ;
       $R = R \cup \{B_j\}$ ;
       $d = 0$ ;
    end if
     $d := d + d_{i+1}$ ;
  end for
  return ( $j$ );
end function

```

Per demostrar la correctesa de l'algorisme compararem la solució obtinguda pel nostre algorisme amb una possible solució òptima amb menys aturades. Sigui  $R = \{b_1, \dots, b_j\}$  les benzineres seleccionades per GREEDY. Suposem que la solució òptima requereix  $m < j$  aturades, sigui  $S = \{c_1, \dots, c_m\}$  el conjunt de les benzineres a una solució òptima.

Primer demostrarem que, per  $j = 1, \dots, m$ ,  $d(B_1, b_j) \geq d(B_1, c_j)$ . L'enunciat és cert per  $j = 1$ , si no ens quedaríem sense benzina abans d'arribar a  $c_1$ . Suposem que l'enunciat és cert per  $i < j$ . Llavors tenim

$$d(B_1, c_j) - d(B_1, c_{j-1}) \leq K,$$

ja que  $S$  és una solució, i

$$d(B_1, c_j) - d(B_1, b_{j-1}) \leq d(B_1, c_j) - d(B_1, c_{j-1}),$$

per hipòtesi d'inducció. Combinant les dues desigualtats tenim

$$d(B_1, c_j) - d(B_1, b_{j-1}) \leq K.$$

Llavors,  $c_j$  ha de ser abans de  $b_j$ .

Com a conseqüència del resultat tenim que  $d(b_m, B_n) \geq d(c_m, B_n) \leq K$ , llavors l'algorisme voraç no s'aturaria a cap benzinera després de  $b_m$  i tenim  $j = m$ .

Per tant GREEDY és òptim i la seva complexitat és  $O(n)$ .

**Solució 2.51** Notemos que en la función a optimizar,  $\frac{\sum_i c_i}{n}$ , el denominador no depende de la planificación. Por lo tanto la planificación con coste mínimo es la del coste medio mínimo y viceversa. Los algoritmos que propondré resuelven el problema de buscar una planificación con coste mínimo.

1. El algoritmo ordena los trabajos en orden creciente de  $p_i$ , y los planifica en ese orden. El coste es el de la ordenación,  $O(n \log n)$ .

Para ver que es correcto utilizo un argumento de intercambio. Supongamos que la planificación con coste mínimo no sigue el orden creciente de tiempo de procesamiento. Para simplificar asumo que el orden  $a_1, \dots, a_n$  es el que proporciona coste óptimo y que en él se produce una inversión, es decir  $p_i > p_{i+1}$ , para algún  $i$ .

Tenemos que  $c_i = p_1 + \dots + p_i$ , por lo tanto

$$\sum_i c_i = np_1 + (n-1)p_2 + \dots + (n-i)p_i + \dots + 1p_n.$$

Si intercambiamos  $a_i$  con  $a_{i+1}$  solo cambia la contribución al coste de estos dos elementos que pasa de ser  $(n-i)p_i + (n-i-1)p_{i+1}$  a ser  $(n-i)p_{i+1} + (n-i-1)p_i$ . El incremento en coste debido al intercambio es

$$(n-i)p_{i+1} + (n-i-1)p_i - [(n-i)p_i + (n-i-1)p_{i+1}] = p_{i+1} - p_i < 0.$$

Por tanto, la ordenación no es óptima y tenemos una contradicción.

2. En este segundo apartado tendremos que seguir el criterio del apartado anterior, pero teniendo en cuenta que se incorporarán a lo largo del tiempo nuevos trabajos. La regla voraz del algoritmo es: procesar en cada instante de tiempo el proceso disponible al que le quede menos tiempo por finalizar. Utilizando el mismo argumento de intercambio que en el apartado (a) la regla voraz es correcta.

Tenemos que ir con cuidado en la implementación ya que el número total de instantes de tiempo es  $\sum_i t_i$  y este valor puede ser exponencial en el tamaño de la entrada. Sin embargo, los tiempos en los que se para la ejecución de un proceso coinciden con los de disponibilidad de un nuevo proceso. Necesitamos controlar solo los instantes de tiempo en los que finaliza la ejecución de un proceso o en los que un proceso está disponible, un número polinómico.

El algoritmo ordena en orden creciente de  $r_i$  los procesos y mantiene una cola de prioridad con los procesos disponibles y no finalizados, utilizando como clave lo que le falta al proceso para finalizar su ejecución.

- Ordenar por  $r_i$ ;
- Insertar en la cola todos los procesos con  $r_k = r_1$  (clave  $p_k$ ),  $i =$  primer proceso no introducido en la cola,  $t = r_1$ .
- mientras cola no vacía
  - $(j, p) = \text{pop}()$ , si  $t + p \leq r_i$  procesamos lo que queda de  $a_j$ ,  $t = t + p$ , y repetimos hasta que la cola quede vacía o  $t + p > r_i$ .
  - Si  $t + p > r_i$ , insertamos  $(j, t + p - r_i)$ ,  $t = r_i$ .
  - Insertamos en la cola todos los procesos con  $r_k = r_i$  (clave  $p_k$ ),  $i =$  primer proceso no introducido en la cola.

La implementación es correcta ya que el conjunto de trabajos disponibles y no finalizados solo se modifican cuando hay un nuevo trabajo disponible o cuando iniciamos el procesamiento de uno de ellos. En el primer caso ese caso actualizamos la cola y el posible trabajo que se estaba ejecutando se interrumpe, y se vuelve a insertar en la cola con el tiempo restante. En el segundo, sacamos al proceso con menor tiempo para finalizar y iniciamos o reiniciamos su ejecución.

El coste de la ordenación es  $O(n \log n)$  y el coste de cada inserción en la cola es  $O(\log n)$ . Para contabilizar el número total de inserciones, notemos que cada proceso se inserta en la cola cuando está disponible, lo que nos da  $n$  inserciones. Un proceso puede volver a reinsertarse en la cola varias veces, sin embargo, por cada tiempo de disponibilidad se reinserta un proceso como mucho, esto nos da  $\leq n$  inserciones debido a paradas en la ejecución. Sumando todo, el coste del algoritmo es  $O(n \log n)$ .



# 4

---

## Soluciones del Profesor

### 4.1. Repaso de Conceptos Básicos

**Solución 1.2** 1. Solucion en pseudocódigo:

```
for  $i := 0$  to  $n$  do
     $B[i] := \text{false}$ 
end for
for  $i := 0$  to  $n - 1$  do
     $B[A[i]] := \text{true}$ 
end for
// Hallar  $j$  tal que  $B[j] = \text{false}$ 
 $j := 0$ 
while  $\neg B[j]$  do
     $j := j + 1$ 
end while
return  $j$ 
```

2. El algoritmo, a alto nivel, consite en:

- a) Fijar  $\ell = 0$ .
- b) Particionar el subvector en curso en dos partes: elementos cuyo bit  $\ell$ -ésimo vale 0 y elementos cuyo bit  $\ell$ -ésimo vale 1. Un grupo contendrá  $2^{k-1-\ell}$  elementos y el otro  $2^{k-1-\ell} - 1$ . Para la partición suponemos que cada  $A[i]$  es en realidad un apuntador al vector de bits, y para hacer el intercambio de cualesquiera dos elementos  $A[i]$  y  $A[j]$  durante la partición se intercambian los correspondientes

apuntadores, no se hacen transferencias de bits, y el intercambio tiene coste  $\Theta(1)$  (y la partición del subvector tiene entonces coste lineal respecto a su tamaño).

- c) Descartar el grupo con mayor número de elementos. Fijar el  $\ell$ -ésimo bit del resultado como el bit  $\ell$ -ésimo del grupo **no** descartado.
- d) Fijar  $\ell := \ell + 1$  y repetir el segundo paso hasta que el grupo no descartado esté vacío.

Si  $n = 2^k - 1$  este algoritmo hará  $k = \Theta(\log n)$  “iteraciones” cada una de las cuales tiene coste  $O(n)$  (para particionar). Ahora bien en cada iteración el tamaño del grupo a particionar es menor, de manera que el total de bits inspeccionados (y el coste será proporcional a dicho número) es

$$n + \frac{n-1}{2} + \frac{n-3}{4} + \cdots + \frac{n - (2^{k-1} - 1)}{2^{k-1}} \leq n \cdot \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{k-1}}\right) \leq n \cdot \sum_{j=0}^{\infty} \frac{1}{2^j} = 2n,$$

por lo que el coste del algoritmo es  $\Theta(n)$  (la cota  $O(n \log n)$  es correcta pero demasiado “grosera”). Alternativamente, podemos establecer que el coste de nuestro algoritmo es  $T(n) = \Theta(n) + T(n/2)$ , cuya solución podemos obtener aplicando el *master theorem* y resulta ser  $T(n) \in \Theta(n)$ .

**Solución 1.3** Dado el vector  $r$  definimos la matriz  $A = (a_{ij})_{n \times n}$ , donde  $a_{ij} = |r_i - r_j|$ ,  $1 \leq i, j \leq n$ . El numerador del coeficiente de Gini es la suma de las componentes de  $A$ , y puesto que hay  $n^2$  componentes parece que será inevitable tener un coste  $\Theta(n^2)$  para calcular el coeficiente de Gini. Pero veremos que puede calcularse en tiempo  $O(n \log n)$ . Para comenzar el denominador

$$2(n-1) \sum_{1 \leq i \leq n} r_i$$

podemos calcularlo fácilmente en tiempo  $\Theta(n)$  con un sencillo recorrido del vector  $r$ . La matriz  $A$  es obviamente simétrica por lo que la suma  $S$  de sus elementos podemos simplificarla:

$$S = \sum_{i=1}^n \sum_{j=1}^n |r_i - r_j| = 2 \sum_{i=1}^n \sum_{j=i+1}^n |r_i - r_j|$$

Por otro lado, ni el numerador ni el denominador dependen del orden específico del vector  $r$ . Entonces podemos reordenar el vector  $r$  en orden decreciente, de manera que  $|r_i - r_j| \geq 0$  si  $i \leq j$ . Por lo tanto

$$S = 2 \sum_{1 \leq i < n} \sum_{i < j \leq n} r_i - r_j.$$



Consideremos un elemento  $r_k$  cualquiera. Dicho elemento aparece en  $S$  restando para  $i = 1$  hasta  $i = k - 1$ :  $r_1 - r_k, r_2 - r_k, \dots, r_{k-1} - r_k$ . De manera parecida  $r_k$  aparece en  $S$  sumando para  $j = k + 1$  hasta  $j = n$ :  $r_k - r_{k+1}, \dots, r_k - r_n$ . Y esas son todas sus apariciones. Por tanto la contribución de  $r_k$  a  $S$  es  $(n - k)r_k - (k - 1)r_k = (n - 2k + 1)r_k$  y así tenemos que

$$S = 2 \sum_{k=1}^n (n - 2k + 1)r_k.$$

¡Pero esta expresión para  $S$  la podemos calcular en tiempo lineal  $\Theta(n)$ ! Así pues tenemos un algoritmo eficiente para calcular el coeficiente de Gini. El coste  $\Theta(n \log n)$  de nuestra solución proviene de un primer paso en el que tendremos que ordenar el vector  $r$  en orden decreciente. El resto del algoritmo tiene coste lineal.

**Solución 1.4** En todos los casos determinaremos el coste de cada bucle evaluando el coste de la vuelta  $i$ -ésima<sup>1</sup> (coste de evaluar la condición y del cuerpo del bucle), y sumaremos para todas las  $i$  posibles: si se aplicase la regla del producto asumiendo que todas las iteraciones tendrán el coste máximo la respuesta obtenida podría no ser ajustada (aunque ciertamente será una cota superior correcta).

1. El coste del bucle interno es  $\Theta(\log(n/i))$ . El coste del bucle externo es

$$\sum_{1 \leq i \leq n} \Theta(\log(n/i)) = \Theta(\log(n^n/n!)) = \Theta(n),$$

aplicando la fórmula de Stirling<sup>2</sup>:

$$n! \sim n^n e^{-n} \sqrt{2\pi n}.$$

2. El bucle interno tiene coste  $\Theta(n - i^2)$  si  $i \leq \sqrt{n}$ ; en otro caso su coste es  $\Theta(1)$  ya que no se ejecuta ninguna iteración. El coste del bucle externo es

$$\sum_{1 \leq i \leq \sqrt{n}} \Theta(n - i^2) = \Theta(n\sqrt{n}).$$

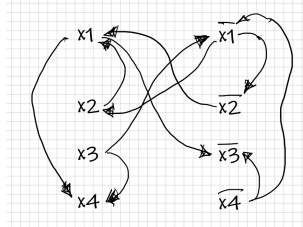
3. El bucle interno hace  $\lceil \log_2 \log_2 i \rceil$  iteraciones: después de  $k$  iteraciones la variable  $j = 2^{2^k}$ . Así que  $k$  es el menor entero tal que  $2^{2^k} \geq i$  o, lo que es lo mismo,  $k = \lceil \log_2 \log_2 i \rceil$ . En este caso la cota superior de la regla del producto nos servirá para calcular el coste del bucle externo: su coste es  $\mathcal{O}(n \log \log n)$ . Por otro lado el coste es mayor que hacer  $n/2$  iteraciones cada una de las cuales tiene coste  $\geq \lg \lg(n/2)$ ; pero  $n/2 \lg \lg(n/2) \in \Omega(n \log \log n)$ , luego el coste del algoritmo es  $\Theta(n \log \log n)$ .

<sup>1</sup>Aquí la  $i$  es genérica. La variable que controla el bucle no tiene por qué llamarse  $i$ .

<sup>2</sup>La notación  $a_n \sim b_n$  significa que  $\lim_{n \rightarrow \infty} a_n/b_n = 1$ . Utilizando notaciones asintóticas  $a_n = b_n(1 + o(1))$ .

4. Si  $n$  es compuesto tendrá algún factor primo  $\leq \sqrt{n}$  y el número de iteraciones del bucle será igual a dicho número primo. Pero si  $n$  es primo la condición  $n \bmod i \neq 0$  se cumple siempre y se harán  $\Theta(\sqrt{n})$ . Con la simplificación de que las operaciones aritméticas tienen coste constante (calcular  $i^2$  y comparar con  $n$ , incrementar la  $i$ , hacer  $n \bmod i$ , ...) el coste del algoritmo, cuando  $n$  es primo, es  $\Theta(\sqrt{n})$ . Puesto que los números implicados en este algoritmo tienen  $\leq \lg n$  bits, el complejidad en términos de operaciones entre bits es  $\mathcal{O}(\sqrt{n} \log^2 n)$ . Lo que es importante es observar que ya que el tamaño  $N$  de la entrada es  $\mathcal{O}(\log n)$ , el coste del algoritmo es  $\Theta\left((\sqrt{2})^N\right) = \Theta(1,4142\dots^N)$ , esto es, exponencial respecto al tamaño de la entrada.

**Solución 1.5** 1. Este es el grafo  $G_F$  que corresponde a la fórmula dada como ejemplo en el enunciado:



2. Si una cierta componente fuertemente conexa contiene  $x$  y  $\neg x$  entonces hay camino de  $x$  a  $\neg x$  y viceversa; esto es, sendas cadenas de razonamiento lógico  $x \implies \dots \implies \neg x$  y  $\neg x \implies \dots \implies x$ . Pero la primera implicación ( $x \implies \neg x$ ) exige que  $x$  sea falsa (porque de otro modo verdadero implicaría falso), mientras que la segunda ( $\neg x \implies x$ ) requiere que  $x$  sea verdadera (nuevamente, tendríamos que verdadero implica falso si  $x$  fuera falsa). Si ambas implicaciones se deducen de la fórmula (porque originan caminos del vértice  $x$  al vértice  $\neg x$  y a la inversa en el grafo  $G_F$ ) tenemos una contradicción,  $x$  no puede ser verdadera y falsa a la vez, y se demuestra que la fórmula **no** es satisfactible.
3. En primer lugar estableceremos que si  $x$  e  $y$  son literales cualquiera entonces si  $x \implies y$  es un camino en el grafo  $G_F$  entonces  $\neg y \implies \neg x$  también es un camino en  $G_F$ . Se demostrará fácilmente por inducción sobre el número de arcos (longitud de la cadena de implicaciones) de  $x \implies y$ . Si la longitud del camino es 1 significa que  $(x, y)$  es un arco en  $G_F$  y por lo tanto que  $\neg x \vee y$  es una cláusula en  $F$ . Pero entonces el arco  $(\neg y, \neg x)$  también se incluye en  $G_F$  y el camino  $\neg y \implies \neg x$  también existe. Si la afirmación es correcta para  $n$  demostraremos que es correcta para  $n + 1$ . Supongamos que  $x = x_0 \implies \dots \implies x_n \implies y$ , donde la última implicación se debe a la existencia del arco  $(x_n, y)$ . Entonces  $(\neg y, \neg x_n)$  es también un arco y por hipótesis de inducción  $\neg x_n \implies \neg x$ , luego  $\neg y \implies \neg x$ .

Supongamos ahora que para un cierto literal  $x$  se cumple que  $x \implies \neg x$  (y como  $x$  y  $\neg x$  se asumen en SCCs distintas sabemos que  $\neg x \not\Rightarrow x$ ). Si asignamos

$\neg x = \mathbf{true}$  entonces  $x = \mathbf{false}$  y la implicación  $x \implies \neg x$  no es contradictoria. Ahora consideremos un literal cualquiera  $y$  tal que  $y \implies x$ . Entonces por la propiedad que hemos demostrado antes  $\neg x \implies \neg y$  y bastará que asignemos  $\neg y = \mathbf{true}$  para que no exista contradicción:

$$y(= \mathbf{F}) \implies x(= \mathbf{F}) \implies \neg x(= \mathbf{T}) \implies \neg y(= \mathbf{T}).$$

El otro caso que debmos contemplar es el de un literal  $x$  tal que no existe camino entre  $x$  y  $\neg x$  ni entre  $\neg x$  y  $x$ . Podremos asignar valores **true** y **false** a  $x$  y  $\neg x$  indistintamente. Supongamos que  $y \implies x$  pero que  $y \not\implies \neg x$ . Por la propiedad de antes sabemos que  $\neg x \implies \neg y$  y que  $x \not\implies \neg y$ . Asignando  $\neg y = \mathbf{true}$  tendremos simplemente que asignar  $\neg x = \mathbf{true}$  y por lo tanto  $x$  e  $y$  se les asigna **false**. No hay ninguna contradicción:  $\neg x(= \mathbf{T}) \implies \neg y(= \mathbf{T})$  y, de otro lado,  $y(= \mathbf{F}) \implies x(= \mathbf{F})$ . El camino contradictorio  $\neg x(= \mathbf{T}) \implies y(= \mathbf{F})$  no existe porque  $x$  y  $\neg x$  no están conectados (si existiera entonces tendríamos  $\neg x \implies y \implies x$ !), tampoco  $\neg y(= \mathbf{T}) \implies x(= \mathbf{F})$  existe, por idéntica razón (si existiera concluiríamos  $\neg x \implies \neg y \implies x$ !).

Si un literal  $y \implies x$  y también  $y \implies \neg x$ , se deduce que  $x \implies \neg y$  y  $\neg x \implies \neg y$ . Asignando  $\neg y = \mathbf{true}$ , las implicaciones  $x \implies \neg y$  y  $\neg x \implies \neg y$  no son contradictorias, demos los valores que demos a  $x$  y  $\neg x$ ; como  $y = \mathbf{false}$  las implicaciones  $y \implies x$  e  $y \implies \neg x$  tampoco son contradictorias.

4. El algoritmo para saber si una fórmula booleana  $F$  que es 2-CNF es satisfactible o no comienza construyendo el grafo  $G_F$  que se describe en el enunciado; requiere tiempo  $\Theta(n + m)$ , esto es, lineal en el tamaño de la fórmula. El siguiente paso es calcular las componentes fuertemente conexas (SCC) de  $G_F$  (ver el ejercicio #15); se hace mediante un par de recorridos con coste  $\Theta(n + m)$ . Una vez calculadas las SCC recorremos el conjunto de variables y determinamos si hay alguna  $x_i$  tal que  $\neg x_i$  está en la misma SCC, con coste  $\mathcal{O}(n)$ . Si existe tal variable, declaramos  $F$  no satisfactible. En caso contrario, hacemos un orden topológico inverso del grafo acíclico dirigido de las SCC; en cada “supernodo” visitado a todos sus literales que no tienen valor asignado se les asigna el valor **true** (y a sus respectivas negaciones el valor **false**); tal como se ha argumentado en el apartado anterior este procedimiento asignará valores de verdadero/falso a todas las variables sin incurrir en contradicción y nos proporciona una asignación que satisface la fórmula  $F$ . Este último paso (obtener una asignación para la fórmula, cuando ésta es factible) también se lleva a cabo en tiempo lineal  $\Theta(n + m)$ , es lo que nos cuesta el orden topológico inverso.

**Solución 1.6** Partiremos de las siguientes observaciones:

1. en un grafo  $G = \langle V, E \rangle$  cualquiera puede haber una o ninguna celebridad, pero no puede haber más de una;
2. sean  $u$  y  $v$  un par de vértices cualesquiera de  $V$  distintos: si  $(u, v) \in E$  entonces  $u$  **no** es una celebridad porque conoce a alguien (a  $v$ );

3. si, por el contrario,  $(u, v) \notin E$  entonces  $v$  **no** es una celebridad porque hay alguien  $(u)$  que no le conoce.

Comprobar la existencia de un arco  $(u, v)$  puede hacerse en tiempo constante —ya que el grafo se nos da en una matriz de adyacencia— y por lo tanto podremos encontrar la celebridad (o su ausencia) en tiempo lineal.

Nuestra solución comienza con un conjunto que contiene todos los elementos de  $V$ , y a continuación entra en un bucle, en cada iteración toma dos vértices cualesquiera  $i$  y  $j$  y comprueba si  $(i, j) \in E$  o no, descartando en cada iteración uno de los dos vértices. El número de iteraciones será  $n - 1$ , cuando en nuestro conjunto solo quede un vértice, éste será una celebridad o bien el grafo no contendrá ninguna. Si el vértice que es una potencial celebridad es  $i$ , un bucle posterior verifica, en tiempo lineal, que  $(i, k) \notin E$  y  $(k, i) \in E$ , para toda  $k$ ,  $1 \leq k \leq n$ ,  $k \neq i$ ; si esto no se cumple entonces  $i$  no es una celebridad y  $G$  no contiene ninguna.

Podemos prescindir de representar explícitamente el conjunto de los elementos y mantener tan sólo dos vértices  $i$  y  $j$ , con  $i < j$ . El conjunto (implícito) de potenciales celebridades es  $\{i, \dots, j\}$ .

```

i := 1; j := n // n = |V|
while i < j do
    if  $(i, j) \in E$  then
        i := i + 1
    else
        j := j - 1
    end if
end while
// Chequear que i (= j) es de hecho una celebridad
is_celeb := true
k := 1
while  $k < i \wedge is\_celeb$  do
    is_celeb :=  $(k, i) \in E \wedge (i, k) \notin E$ 
    k := k + 1
end while
k := i + 1
while  $k \leq n \wedge is\_celeb$  do
    is_celeb :=  $(k, i) \in E \wedge (i, k) \notin E$ 
    k := k + 1
end while
return  $\langle i, is\_celeb \rangle$ 

```

### Solución 1.7

$$0,99^n; (\log n)^{100}; \sqrt{n}; n \log n; \frac{n^2}{\log n}; n^3; n2^n; 3^n$$

Puede usarse el criterio del límite entre cada par de funciones para ver que en efecto

$$\lim_{n \rightarrow \infty} \frac{f_i(n)}{f_{i+1}(n)} < +\infty.$$

De hecho el límite del cociente entre  $f_i$  y  $f_{i+1}$  es en todos los casos 0, por lo que  $f_i$  es de un orden de magnitud **estrictamente inferior** al de  $f_{i+1}$ .

**Solución 1.8** Para el cálculo de los límites puede ser útil la regla de L'Hôpital (aplicada, si conviene, repetidas veces):

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)},$$

si  $f$  y  $g$  son funciones derivables.

1. Falsa. Por ejemplo  $1/n = o(1)$  y  $n = \omega(1)$  pero

$$\left(1 + \frac{1}{n}\right)^n \rightarrow e \neq 1$$

2. Cierta. Puesto que

$$\lim_{n \rightarrow \infty} \frac{(n+2)n/2}{n^2} = 1/2.$$

3. Falsa.  $(n+2)n/2 \in (n^3)$  pero  $n^3 \notin \mathcal{O}(f(n))$ . En efecto

$$\lim_{n \rightarrow \infty} \frac{n^3}{(n+2)n/2} = +\infty,$$

y por tanto  $f(n) \in o(n^3)$  (o equivalentemente,  $n^3 \in \omega(f(n))$ ).

4. Falso.  $\lim_{n \rightarrow \infty} \frac{n^{1,1}}{n(\lg n)^2} = +\infty$ .

5. Cierto.  $\lim_{n \rightarrow \infty} \frac{n^{0,01}}{(\lg n)^2} = +\infty$ .

**Solución 1.9** La demostración **no** es correcta porque la constante “oculta” en la notación asintótica que se usan en la hipótesis de inducción no sirve para dar el paso de inducción. Cuando decimos que por hipótesis inductiva

$$\sum_{k=1}^n k = O(n),$$

realmente estamos diciendo que existe una constante  $c > 0$  tal que

$$\sum_{k=1}^n k \leq c \cdot n.$$

Podemos suponer que es cierto para todo  $n > 0$ , en vez de para toda  $n \geq n_0$ . Si es cierto para algún  $n_0$  y una cierta  $c'$  (lo que dice la definición de  $\mathcal{O}(f)$ ) entonces podremos tomar una constante  $c = c' \cdot \max f(i) \mid 1 \leq i \leq n_0$  y  $c \cdot f(n) \geq g(n)$  para toda  $n > 0$ . Ahora, aplicando la hipótesis de inducción:

$$\sum_{k=1}^{n+1} k = n + 1 + \sum_{k=1}^n k \leq n + 1 + c \cdot n = (c + 1) \cdot n + 1 \not\leq c \cdot n,$$

no hay ninguna constante  $c$  tal que  $(c + 1) < c$ !

**Solución 1.10** Si  $r \neq 1$  la suma de una serie geométrica cumple

$$\sum_{j=a}^b r^j = \frac{r^{b+1} - r^a}{r - 1},$$

una fórmula que se puede demostrar por inducción sobre  $n = b - a$ . También podemos “recuperar” la fórmula escribiendo

$$\begin{aligned} S_{a,b} &= \sum_{j=a}^b r^j = r^a + r^{a+1} + \dots + r^b \\ &= r^a(1 + r + \dots + r^{b-a}) \\ S_{a+1,b+1} &= S_{a,b} - r^a + r^{b+1} = r^{a+1}(1 + \dots + r^{b-a}) = rS_{a,b} \\ (1 - r)S_{a,b} &= -r^{b+1} + r^a \\ S_{a,b} &= \frac{r^{b+1} - r^a}{r - 1}. \end{aligned}$$

Aplicando la fórmula para  $r = 4$ ,  $a = 1$  y  $b = \lg n$  obtenemos

$$\sum_{j=1}^{\lg n} 4^j = \frac{4^{\lg n+1} - 4}{3} = \Theta(4^{\lg n}) = \Theta(n^2).$$

N.B. La base del logaritmo es relevante. Si el límite superior del sumatorio fuera  $\log_4 n$  entonces la suma sería  $\Theta(n)$ . En general si el límite superior de la suma es  $\log_b n$ , podemos escribir

$$\log_b n = \frac{\log_4 n}{\log_4 b},$$

y

$$4^{\log_b n} = (4^{\log_4 n})^{1/\log_4 b} = n^{1/\log_4 b} = n^{\log_b 4}$$

luego la suma será  $\Theta(n^{\log_b 4})$ .

**Solución 1.11** La distancia recorrida viene descrita por la recurrencia:

$$C(n) = n + n + n + n - 1 + 1 + C(n - 2) = 4n + C(n - 2).$$

Aplicando el teorema de recurrencias sustractivas tenemos  $C(n) = \Theta(n^2)$ .

Si pensamos que el robot está situado en la esquina inferior izquierda de un tablero  $(n + 1) \times (n + 1)$ , el algoritmo hace que el robot vaya visitando las casillas del tablero. La secuencia de pasos  $n$  norte,  $n$  este, etc. cubre todas las casillas de la periferia del tablero y lo sitúa en la esquina inferior de un tablero  $(n - 1) \times (n - 1)$ . Por lo tanto recursivamente el robot visitará la totalidad de casillas del tablero en un recorrido “espiral” descendente. Para visitar cada casilla tiene que recorrer un metro, excepto para visitar la primera de todas. Así pues  $C(n) = (n + 1)^2 - 1 = n^2 + 2n$  exactamente. Podemos demostrar que esto es cierto por inducción general para todo  $n$  par con  $n \geq 2$ . Para la base de la inducción:  $C(2) = 2 + 2 + 2 + 1 + 1 = 8 = 2^2 + 2 \cdot 2$ , se cumple. Y para el paso inductivo como  $n - 2$  es par por ser  $n$  par tenemos

$$\begin{aligned} C(n) &= 4n + C(n - 2) \stackrel{\text{h.i.}}{=} 4n + (n - 2)^2 + 2(n - 2) \\ &= 6n - 4 + n^2 - 4n + 4 = n^2 + 2n \end{aligned}$$

Para  $n$  impar la fórmula es  $C(n) = n^2 + 2n - 3$ . En el caso base con  $n = 3$  se hacen 12 pasos;  $C(3) = 12 = 9 + 6 - 2$ . Para el paso inductivo, que  $n$  sea impar implica que  $n - 2$  también lo es y podemos aplicar la hipótesis de inducción:

$$\begin{aligned} C(n) &= 4n + C(n - 2) \stackrel{\text{h.i.}}{=} 4n + (n - 2)^2 + 2(n - 2) - 3 \\ &= 6n - 4 + n^2 - 4n + 4 - 3 = n^2 + 2n - 3. \end{aligned}$$

**Solución 1.13** 1. La matriz de adyacencia  $B = (b_{ij})$  del grafo  $G'$  cumple

$$b_{ij} = \bigvee_{k=1}^n a_{ik} \wedge a_{kj},$$

donde  $A = (a_{ij})$  es la matriz de adyacencia de  $G$ . Podemos obtener  $B = A^2$  usando el algoritmo de multiplicación de matrices convencional pero reemplazando los productos por **and** ( $\wedge$ ) y las sumas por **or** ( $\vee$ ). Entonces habremos computado el cuadrado de  $G$  en tiempo  $\Theta(|V|^3)$ . En otras palabras: para cada par de vértices  $(i, j)$  (hay  $\Theta(n^2)$  pares a considerar) hay que comprobar con  $O(n)$  operaciones (de coste  $\Theta(1)$ , puesto que tenemos una matriz de adyacencias) si existe un  $k$  tal  $(i, k) \in E$  y  $(k, j) \in E$ .

Pero percatarnos que el cálculo del grafo  $G'$  se reduce a una “multiplicación” de matrices nos ayuda a encontrar una solución más eficiente aún. Podemos usar el convenio **false**  $\equiv 0$  y **true**  $\equiv 1$  y aplicar el algoritmo de Strassen de multiplicación de matrices con coste  $\Theta(|V|^{\log_2 7}) = \Theta(|V|^{2.81\dots})$ ; cambiando en un paso final las entradas  $= 0$  por **false** y las entradas  $\neq 0$  por **true**.

2. En el caso de las listas de adyacencia, para cada par de vértices  $(i, k) \in E$  recorreremos la lista de adyacencia del vértice  $k$ : cada arista  $(k, j)$  en la lista de adyacentes de  $k$  da lugar a una arista  $(i, j) \in E'$ .

```

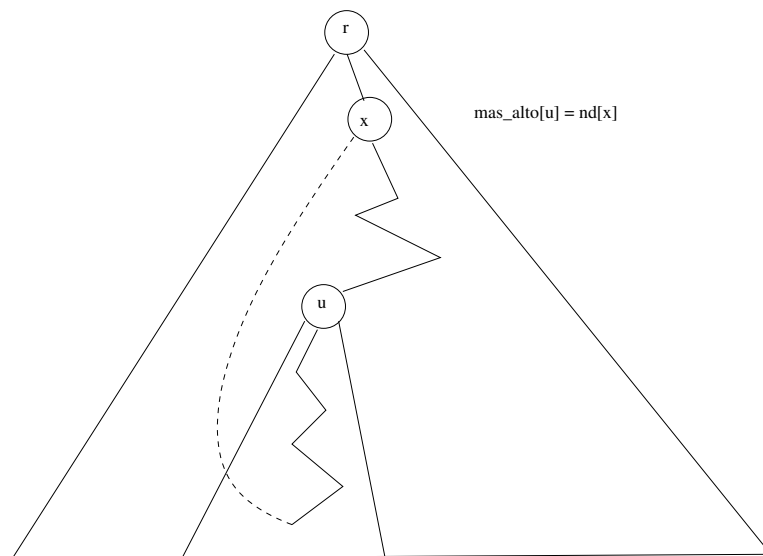
procedure SQUARE( $G$ )
  // Retorna el cuadrado de  $G$ 
   $V' := V$ ;  $E' := \emptyset$ 
  for  $(i, k) \in E$  do
    for  $j \in G.ADJACENT(k)$  do
      Añadir  $(i, j)$  a  $E'$ 
    end for
  end for
  return  $G' = \langle V', E' \rangle$ 
end procedure

```

El coste de este algoritmo es  $\Theta(nm)$ , siendo  $m = |E|$  y  $n = |V|$ . En grafos densos es  $\Theta(n^3)$ , el mismo coste que con matrices de adyacencia usando el algoritmo “simple”.

**Solución 1.14** Sin pérdida de generalidad, asumiremos que el grafo es conexo. Si no lo es, el algoritmo que describimos a continuación se aplica por separado a cada componente conexa del grafo. Un grafo que no tienen ningún punto de articulación (p.e. el anillo que se menciona en el enunciado) se dice que es *biconexo*.

Supongamos que efectuamos un DFS empezando en un vértice  $r$  y que para cada vértice  $u$  determinamos el número DFS más bajo<sup>3</sup> que es alcanzable siguiendo un camino desde  $u$  hasta uno de sus descendientes en el  $T_{DFS}$  y a continuación “subimos” con una arista de retroceso. Denominaremos  $mas\_alto[u]$  a dicho número.



<sup>3</sup>Recordemos que el número DFS de un vértice es simplemente el número de orden en el cual se visita el vértice.



Para ver si un vértice es punto de articulación o no debermos considerar varios casos:

1.  $u$  es una hoja del  $T_{\text{DFS}}$  (no tiene descendientes): entonces no es un punto de articulación ya que nunca se desconectará el grafo
2.  $u = r$  es la raíz del  $T_{\text{DFS}}$ : es punto de articulación si y sólo si tiene más de un descendiente en el  $T_{\text{DFS}}$ , su eliminación desconectaría los subárboles, pero si sólo hay uno su eliminación no desconecta a los demás vértices.
3.  $u$  no es hoja y no es la raíz: entonces es punto de articulación si y sólo si alguno de los vértices adyacentes descendientes  $w$  de  $u$  cumple  $\text{mas\_alto}[w] \geq \text{ndfs}[u]$ , es decir, si algún descendiente no puede “escapar” del subárbol enraizado en  $u$  sin pasar por  $u$ .

Por otro lado para cada vértice  $u$ ,  $\text{mas\_alto}[u]$  es el mínimo entre: 1)  $\text{ndfs}[u]$ , 2)  $\text{mas\_alto}[v]$  para todo  $v$  descendiente directo de  $u$  en el  $T_{\text{DFS}}$  y 3)  $\text{ndfs}[v]$  para todo  $v$  adyacente a  $u$  mediante una arista de retroceso.

**procedure** PUNTOSARTICULACION( $G$ )

// Pre:  $G$  es conexo

**for**  $v \in V(G)$  **do**

$\text{visitado}[v] := \text{false}$

$\text{ndfs}[v] := 0$

$\text{punto\_art}[v] := \text{false}$

$\text{num\_desc}[v] := 0$

**end for**

$\text{num\_dfs} := 0$

$\text{es\_biconexo} := \text{true}$

// Basta lanzar un DFS porque  $G$  es conexo

$v :=$  un vértice cualquiera de  $G$

PUNTOSARTICULACION-REC( $G, v, v, \text{es\_biconexo}, \text{ndfs}, \dots$ )

**return**  $\langle \text{es\_biconexo}, \text{punto\_art} \rangle$

**end procedure**

**procedure** PUNTOSARTICULACION-REC( $G, v, \text{padre}, \dots$ )

$\text{num\_dfs} := \text{num\_dfs} + 1; \text{ndfs}[v] := \text{num\_dfs}$

$\text{visitado}[v] := \text{true}$

$\text{mas\_alto}[v] := \text{ndfs}[v]$

**for**  $w \in G.\text{ADJACENT}(v)$  **do**

**if**  $\neg \text{visitado}[w]$  **then**

$\text{num\_desc}[v] := \text{num\_desc}[v] + 1$

PUNTOSARTICULACION-REC( $G, w, v, \dots$ )

$\text{mas\_alto}[v] := \min(\text{mas\_alto}[v], \text{mas\_alto}[w])$

$\text{punto\_art}[v] := \text{punto\_art}[v] \vee (\text{mas\_alto}[w] \geq \text{ndfs}[v])$

**else if**  $\text{padre} \neq w$  **then**

$\text{mas\_alto}[v] := \min(\text{mas\_alto}[v], \text{ndfs}[w])$

```

    end if
  end for
  if  $v = \text{padre}$  then //  $v$  es la raíz del  $T_{\text{DFS}}$ 
    punto_art[v] := num_desc[v] > 1
  end if
  es_biconexo := es_biconexo  $\wedge$   $\neg$ punto_art[v]
end procedure

```

El algoritmo es una aplicación del esquema de recorrido en profundidad y el trabajo que se realiza para visitar cada vértice y cada arista requiere tiempo constante. Por ejemplo, se hace el *update* de *num\_desc* y *punto\_art* para cada  $w$  adyacente a  $v$  que no haya sido no visitado y el *update* de *mas\_alto* para todo  $w$  adyacente a  $v$ . En consecuencia el coste del algoritmo es lineal respecto al tamaño del grafo, esto es,  $\mathcal{O}(|V| + |E|)$ , si representamos el grafo mediante listas de adyacencia. Si el grafo se representa con matrices de adyacencia el coste pasa a ser  $\Theta(|V|^2)$  pues el bucle sobre los vértices  $w$  itera  $n = |V|$  veces, independientemente de  $v$  (en listas de adyacencia el bucle hace solo grado( $v$ ) iteraciones).

**Solución 1.15** El algoritmo utiliza tres recorridos DFS para resolver el problema. En el primer recorrido se retorna una lista  $Q$  de los vértices. Esto es, el primer elemento de  $Q$  es el último vértice que se cierra en el DFS, el segundo es el penúltimo en cerrar, etc.

```

procedure NUMERAINVERSA( $G, Q$ )
   $Q := \text{EMPTYSTACK}()$ 
  for  $v \in V$  do
     $\text{visited}[v] := \text{false}$ 
  end for
  for  $v \in V$  do
    if  $\neg \text{visited}[v]$  then
      NUMERAINVERSA-REC( $G, v, Q, \text{visited}$ )
    end if
  end for
end procedure
procedure NUMERAINVERSA-REC( $G, v, Q, \text{visited}$ )
   $\text{visited}[v] := \text{true}$ 
  for  $w \in G.\text{SUCCESSOR}(v)$  do
    if  $\neg \text{visited}[w]$  then
      NUMERAINVERSA-REC( $G, w, Q, \text{visited}$ )
    end if
  end for
   $Q.\text{PUSH}(v)$ 
end procedure

```

En el segundo DFS se recorre transpone el digrafo cambiando la orientación de todos los arcos:  $G^T = \langle V, E^T \rangle$ , con  $E^T = \{(u, v) \mid (v, u) \in E\}$ . Claramente las SCC de  $G^T$  son idénticas a las SCC de  $G$ . Por último se recorre el digrafo  $G^T$  pero siguiendo el

orden de la lista  $Q$ . Si lanzamos un DFS en  $G^T$  desde el primer vértice de  $Q$ , llamémosle  $u$ , que es el último que se cierra en un DFS del digrafo original, entonces solo se pueden visitar vértices que pertenezcan a su misma componente fuertemente conexa; en caso contrario tendríamos una contradicción pues no podría ser el último que se cerró. En efecto, supongamos que hay algún arco en  $G^T$  que une un vértice de la SCC de  $u$  con otro vértice  $w$  en una SCC diferente. Eso significa que en el digrafo  $G$  hay un arco que une  $w$  con  $u$ . Entonces  $w$  tiene que cerrarse más tarde que  $u$ , pero hemos asumido que  $u$  tiene el mayor número de cierre. Lanzando otro DFS desde el siguiente vértice no visitado en orden número de cierre descendiente obtenemos una nueva componente fuertemente conexa (SCC=strongly connected component), etc.

```

procedure OBTEN-SCC( $G$ )
  NUMERAINV( $G, Q$ )
  // todos los vértices de  $G$  están en la lista  $Q$  por orden creciente de numeración
  // inversa, esto es, por orden de cierre
  TRANSPONER( $G$ )
  //  $G := G^T$ 
  for  $v \in V$  do
     $visited[v] := \text{false}$ 
  end for
   $ncc := 0$ 
  for  $v \in Q$  do
    if  $\neg visited[v]$  then
       $ncc := ncc + 1$ 
      VISITA-SCC( $G, v, ncc, SCC, visited$ )
    end if
  end for
  //  $ncc$  es el número de SCCs en  $G$ 
  //  $SCC[v]$  es el número de la SCC del vértice  $v, \forall v$ 
end procedure
procedure VISITA-SCC( $G, v, ncc, SCC, visited$ )
   $visited[v] := \text{true}$ 
   $SCC[v] := ncc$ 
  for  $w \in G.SUCCESOR(v)$  do
    if  $\neg visited[w]$  then
      VISITA-SCC( $G, w, ncc, SCC, visited$ )
    end if
  end for
end procedure

```

El coste del algoritmo es obviamente el de los tres recorridos DFS donde para cada vértice y arco visitados se hace un trabajo de coste  $\Theta(1)$ . Usando listas de adyacencia para representar el digrafo el coste del algoritmo es  $\Theta(|V| + |E|)$ . Con matrices de adyacencia el coste sería  $\Theta(|V|^2)$ .

Todo vértice y todo arco del digrafo debe ser necesariamente visitado para poder

determinarse a qué SCC pertenece cada vértice o incluso para saber cuántas SCC tiene el digrafo, por lo que el coste lineal  $\Theta(|V| + |E|)$  es el mejor posible.

El algoritmo aquí descrito es fundamentalmente el conocidísimo **algoritmo de Kosaraju-Sharir**. También podéis consultar el **algoritmo de Tarjan para SCC**.

**Solución 1.16** Consideremos  $G^{\text{SCC}}$ , el grafo de SCCs (componentes fuertemente conexas) de  $G$ .  $G^{\text{SCC}}$  es necesariamente acíclico. Ahora supongamos dos vértices  $u$  y  $v$  cualesquiera de  $G$ . Si  $u$  y  $v$  están en la misma SCC entonces hay camino de  $u$  a  $v$  y de  $v$  a  $u$ . Pero si  $u$  y  $v$  están en SCCs distintas entonces habrá un camino de  $u$  a  $v$  si y solo si la SCC de  $u$  es “antecesora” de la SCC de  $v$  en  $G^{\text{SCC}}$ ; análogamente habrá camino de  $v$  a  $u$  si la SCC de  $v$  es antecesora de la SCC de  $u$ . Por lo tanto, para que siempre exista como mínimo uno de los dos caminos entre dos vértices  $u$  y  $v$  es necesario y suficiente que  $G^{\text{SCC}}$  sea un digrafo lineal: existe una ordenación de los vértices de  $G^{\text{SCC}}$  tal que todos los arcos son de la forma  $(v_i, v_{i+1})$  para alguna  $i$ ,  $1 \leq i < N = \# \text{ SCCs de } G$ .

En definitiva, necesitamos demostrar que el  $G^{\text{SCC}}$  admite una linearización: si  $v$  y  $w$  son accesibles desde  $u$  entonces  $v$  es accesible desde  $w$  o  $w$  es accesible desde  $v$ . Esto significa que si ordenamos topológicamente el grafo  $G^{\text{SCC}}$  existe exactamente una raíz y el orden topológico es único. Esto es, en el bucle que obtiene uno de los vértices  $v$  con 0 predecesores, lo imprime, actualiza el número de predecesores de cada uno de los sucesores de  $v$  y añade los que llegan a 0 a la cola de visitas, solo debe existir un único vértice candidato a visitar en toda iteración:

```

procedure ESSEMICONEXO( $G$ )
   $G^{\text{SCC}} := \text{GRAFOCONDENSACION}(G)$ 
  for  $v \in V(G^{\text{SCC}})$  do
     $\text{prev}[v] := 0$ 
  end for
  for  $v \in V(G^{\text{SCC}})$  do
    for  $w \in G^{\text{SCC}}.\text{SUCCESSORS}(\langle \rangle v)$  do
       $\text{pred}[w] := \text{pred}[w] + 1$ 
    end for
  end for
  //  $\text{pred}[v]$  = núm. de predecesores de  $v$  en  $G^{\text{SCC}}$ 
   $Q := \emptyset$ 
  for  $v \in V(G^{\text{SCC}})$  do
    if  $\text{pred}[v] = 0$  then
       $Q.\text{PUSH}(\langle \rangle v)$ 
    end if
  end for
  while  $|Q| = 1$  do
     $u := Q.\text{POP}(\langle \rangle)$ 
    for  $w \in G^{\text{SCC}}.\text{SUCCESSORS}(\langle \rangle v)$  do
       $\text{pred}[w] := \text{pred}[w] - 1$ 
      if  $\text{pred}[w] = 0$  then
         $Q.\text{PUSH}(\langle \rangle w)$ 
      end if
    end for
  end while

```

```

    end if
  end for
end while
// Retorna cierto ssi el grafo solo admite un orden
// topológico único que visita todos los vértices
return  $|Q| = 0$ 
end procedure

```

**Solución 1.17** Dada la secuencia  $w$  designamos  $w_{i:k} = w_i \cdot w_{i+1} \cdots w_{i+k-1}$  el  $k$ -mero que arranca en la posición  $i$ .

Dado un multiconjunto  $C$  de  $k$ -meros construiremos un (multi)grafo  $G_C$  donde para cada  $k$ -mero  $x = (x_1, \dots, x_k)$ , tendremos dos vértices  $x_\ell = (x_1, \dots, x_{k-1})$  y  $x_h = (x_2, \dots, x_k)$ , unidos por el arco  $(x_\ell, x_h)$ . Como puede ocurrir que  $x = y$ , que  $x_\ell = y_\ell$ , que  $x_\ell = x_h$ , etc para diferentes  $k$ -meros del multiconjunto  $C$  dado, el grafo  $G_C$  puede contener arcos paralelos y bucles.

Si  $C = S(w)$  para alguna secuencia  $w$  entonces para un  $k$ -mero  $w_{i:k} = w_i \cdots w_{i+k-1}$  en el grafo tendríamos el vértice  $(w_i \cdots w_{i+k-2})$  unido por un arco a  $w_{i+1} \cdots w_{i+k-1}$  que a su vez estaría unido a  $w_{i+2} \cdots w_{i+k}$ , etc. Al construir el grafo cada  $k$ -mero  $w_{i:k}$  de  $w$  nos da origen a dos  $(k-1)$ -meros; el segundo  $w_{i+1:k-1}$  coincide con el primero de los dos  $(k-1)$ -meros asociados a  $w_{i+1:k}$ . Por eso  $C$  es el multiconjunto de  $k$ -meros asociados a una secuencia  $w$  si y solo si  $G_C$  contiene al menos un camino euleriano.

En el caso de que el camino no exista entonces  $C = S(w)$  para toda  $w$ . i existe uno y solo un camino euleriano dicho camnio da la única reconstrucción válida de  $w$ . Si hay más de un camino o un ciclo euleriano entonces existen cadenas  $w_1, w_2, \dots$  cada uno de las cuales origina el mismo multiconjunto de  $k$ -meros.

Como el grafo es dirigido el criterio para saber si es o no euleriano es levemente distinto. El digrafo  $G$  contendrá al menos un camino euleriano si y solo si todos los vértices  $v$  tiene  $\text{bal}(v) = 0$  excepto un vértice  $s$  con  $\text{val}(s) = 1$  y otro  $t$  con  $\text{bal}(v) = -1$ , siendo  $\text{bal}(v)$  la diferencia entre el grado de salida y el grado de entrada de  $v$ .

El grafo  $G_C$  puede construirse en tiempo lineal respecto al tamaño de  $C$ , que a su vez es  $\leq |w| - k + 1$  si es que  $C$  proviene de una cierta secuencia  $w$ . El número de vértices de  $G$  es  $\leq 2C$  (no se repiten los vértices) y cada elemento nos da un arco en  $G$  luego  $|E| = 2|C|$ . Determinar si  $G_C$  es o no euleriano es también fácil de determinar en tiempo lineal. Por último obtener un camino euleriano, si existe, puede hacerse en tiempo lineal respecto al tamaño del grafo (y por tanto lineal respecto al tamaño de la entrada) adaptando el **algoritmo de Hierholzer**. Otro famoso algoritmo para encontrar caminos eulerianos, el de Fleury, es mucho más ineficiente (tiene coste  $\mathcal{O}(|E|^2)$ ).

**Solución 1.19** 1. El teorema maestro (MT) para recurrencias de divide y vencerás es aplicable para costes no recursivos de la forma  $Cn^k(\log n)^\ell$  con  $k \geq 0$  y  $\ell > -1$ . En esta recurrencia ese coste no recursivo es

$$\binom{n}{3} \lg^4 n = \frac{1}{6} n^3 \lg^4 n + o(n^3)$$

por lo que aplicando el teorema con  $a = 16$ ,  $b = 2$  y  $k = 3$ , y dado que  $16 > 2^3$  se deduce que

$$T(n) = \Theta(n^{\log_2 16}) = \Theta(n^4).$$

2. Se aplica MT con  $a = 5$ ,  $b = 2$  y  $k = 1/2$ . Puesto que  $a > b^k$  la solución es  $T(n) = \Theta(n^{\lg 5}) = \Theta(n^{2,321\dots})$ .
3. Se aplica MT con  $a = 2$ ,  $b = 4$  y  $k = 1/2$ . Como  $a = b^k$  la solución es  $T(n) = \Theta(\sqrt{n} \log n)$ .
4. El MT no puede aplicarse ya que tenemos  $\ell = -1$ . Existe una versión más general del MT para este caso, pero también podemos obtener la respuesta desplegando la recurrencia:

$$\begin{aligned} T(n) &= \frac{n}{\lg n} + 2T(n/2) = \frac{n}{\lg n} + \frac{n}{\lg(n/2)} + 4T(n/4) \\ &= \frac{n}{\lg n} + \frac{n}{\lg(n/2)} + \frac{n}{\lg(n/4)} + 8T(n/8) \\ &= \dots = n \cdot \left( \frac{1}{\lg n} + \frac{1}{\lg n - 1} + \dots + \frac{1}{\lg n - k} \right) + 2^{k+1}T(n/2^{k+1}). \end{aligned}$$

Para simplificar vamos a suponer que  $n$  es una potencia de 2, digamos  $n = 2^{k+1}$  (y por tanto  $k = \lg(n) - 1$ ).

$$T(n) = n \cdot \sum_{j=1}^{\lg n} \frac{1}{j} + n \cdot T(1) = nH_{\lg n} + \mathcal{O}(n),$$

donde  $H_m$  denota el  $m$ -ésimo número armónico. Sabemos que  $H_m = \ln m + \mathcal{O}(1)$  y por lo tanto

$$T(n) = n \ln \lg n + \mathcal{O}(n) = \Theta(n \log \log n).$$

5. Se aplica el teorema para recurrencias sustractivas:  $T(n) = aT(n - c) + \Theta(n^k)$ . Como en este caso  $a = 1$ , la solución es  $T(n) = \Theta(n^{k+1}) = \Theta(n^2)$ .

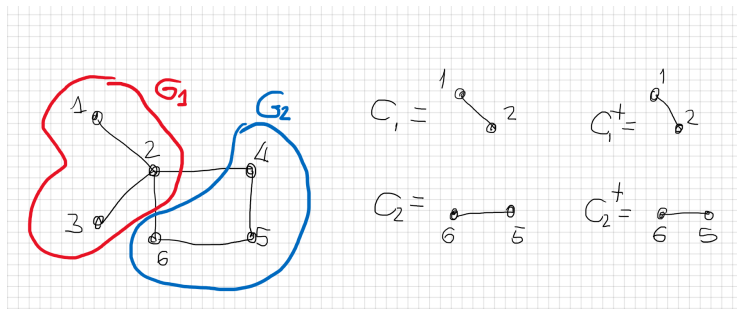
**Solución 1.20** 1. Se demuestra por inducción. Si el grafo consta de un solo vértice entonces tal vértice es una 1-clique. En general, supongamos que los subgrafos  $C_1$  y  $C_2$  retornados por las llamadas recursivas—que reciben como entrada un subgrafo con menor número de vértices que  $G$ —son, por hipótesis de inducción, cliques en  $G_1$  y  $G_2$ , respectivamente. El bucle 6.2 tiene como invariante que  $C_1^+$  es una clique de  $G$ : cada vértice que se añade a  $C_1^+$  es adyacente a todos los vértices en  $C_1^+$  por lo cual el invariante se mantiene en todo momento. Pasa otro tanto con el bucle 6.3, el invariante es que  $C_2^+$  es una clique de  $G$ . Por tanto el algoritmo devuelve una clique de  $G$ .

2. El coste del bucle 6.2 es  $\mathcal{O}(n^2)$ : para cada uno de los  $\mathcal{O}(n/2)$  vértices de  $C_2$  se comprueba, recorriendo su lista de adyacencia, si está conectado a cada uno de los  $\mathcal{O}(n)$  vértices de  $C_1^+$ . Y el coste de 6.3 es  $\mathcal{O}(n^2)$  por idéntica razón. Así que el coste del algoritmo verifica la siguiente recurrencia:

$$C(n) = \mathcal{O}(n^2) + 2C(n/2),$$

cuya solución es  $C(n) = \mathcal{O}(n^2)$ .

3.



4. El problema de decidir si un grafo dado contiene una clique de tamaño  $k$  es NP-completo, y por tanto la versión de optimización (encontrar la clique de tamaño máximo) es NP-difícil. No solo es fácil, es probablemente imposible modificar el algoritmo para que encuentre una clique de  $G$  de tamaño máximo en tiempo  $\mathcal{O}(n^2)$  —o de hecho en tiempo polinómico ( $n^{\mathcal{O}(1)}$ )

**Solución 1.21** En un primer paso ordenaremos las  $n$  rectas por pendiente  $a_i$  creciente (y por punto de intersección decreciente en caso de empate) con coste  $\mathcal{O}(n \log n)$ . Un recorrido de la secuencia resultante nos permitirá detectar con coste  $\mathcal{O}(n)$  rectas ocultas por ser paralelas a otras: si  $L_i \equiv y = a_i + b_i$  y  $L_{i+1} \equiv y = a_{i+1}x + b_{i+1}$  y resulta que  $a_i = a_{i+1}$  y  $b_i > b_{i+1}$  entonces  $L_i$  oculta a  $L_{i+1}$  y podemos eliminar de la secuencia de entrada a esta última (y anotar que  $L_{i+1}$  no es visible en absoluto).

Podemos asumir entonces que la entrada que se nos da consiste en un conjunto de  $N$  rectas tales que

$$a_1 < a_2 < \dots < a_N,$$

donde  $1 \leq N \leq n$  pues algunas rectas pueden haber sido eliminadas en el paso inicial del algoritmo.

Vamos a diseñar un algoritmo HIDDENLINES que dado un conjunto  $C$  de  $n$  líneas que no contiene ningún par de paralelas en orden creciente de pendiente retorna el *perfil visible* de  $C$ , una lista de segmentos

$$S = [[i_1, x_1], [i_2, x_2], \dots, [i_{m-1}, x_{m-1}], [i_m, +\infty]]$$

que indica que las rectas visibles en  $C$  son, por orden creciente de pendiente, de izquierda a derecha, la recta  $L_{i_1}$ , visible entre  $-\infty$  y  $x_1$ , la recta  $L_{i_2}$ , visible entre  $x_1$  y  $x_2$ , etc.

Los casos base corresponden a conjuntos de  $n \leq 3$  rectas. Si  $n = 0$  se devuelve una lista de segmentos vacía. Si  $n = 1$  la lista retornada es  $S = [[i, +\infty]]$  donde  $L - I$  es la única recta del conjunto. Si  $n = 2$  se calcula el punto de intersección  $(x_0, y_0)$  de las dos rectas dadas  $i$  y  $j$  y el resultado es  $S = [[i, x_0], [j, +\infty]]$ . Si  $n = 3$  hay dos casos a considerar: la segunda recta  $L_2$ , cuya pendiente es intermedia entre las otras dos,  $L_1$  y  $L_3$ , intersecta a  $L_1$  en  $p_{12} = (x_{12}, y_{12})$  y a  $L_3$  en  $p_{23} = (x_{23}, y_{23})$ : (1) queda por “encima” del punto de intersección  $p_{13} = (x_{13}, y_{13})$  de  $L_1$  con  $L_3$  (2) o queda por “debajo” del punto de intersección de  $L_1$  con  $L_3$ . En el primer caso el resultado consta de tres segmentos  $S = [[1, x_{12}], [2, x_{23}], [3, +\infty]]$ . En el segundo la recta  $L_2$  no es visible y el resultado es  $S = [[1, x_{13}], [3, +\infty]]$ .

En el caso recursivo nuestra solución actúa de modo conceptualmente similar a MERGESORT, usando una función MERGEPROFILES que dados dos perfiles de visibilidad calcula su “fusión”.

```

procedure HIDDENLINES( $C, i, j$ )
  // Devuelve el perfil visible del subconjunto
  // de líneas  $C[i..j]$ 
  if  $|C| \leq 3$  then
    Calcular el perfil visible  $S$  según se describe
  else
     $m := (i + j) \text{ div } 2$ 
     $S_1 := \text{HIDDENLINES}(C, i, m)$ 
     $S_2 := \text{HIDDENLINES}(C, m + 1, j)$ 
     $S := \text{MERGEPROFILES}(S_1, S_2)$ 
  end if
  return  $S$ 
end procedure

```

La función MERGEPROFILES recorre los perfiles  $S_1$  y  $S_2$ . En un momento dado tendrá una coordenada  $x$  en curso (inicialmente  $x = -\infty$ ) y considerará dos segmentos  $u = [L_u, x_u] \in S_1$  y  $v = [L_v, x_v] \in S_2$ . A partir de las pendiente y puntos de corte de las correspondientes rectas  $L_u$  y  $L_v$  podemos determinar si los segmentos intersectan, esto es, si la coordenada  $x'$  donde  $L_u$  y  $L_v$  intersectan es tal que  $x \leq x' \leq \min(x_u, x_v)$  o no. Si no es así, es fácil determinar si  $u$  queda “por encima” de  $v$  o viceversa; el segmento que queda por encima es visible y se copia en el resultado; se actualiza  $x$  con la  $x$  del segmento visible. El otro segmento se descarta. Se avanza en los dos perfiles  $S_1$  y  $S_2$ . Si  $u$  y  $v$  intersectan en  $x'$  (esto es,  $x \leq x' \leq \min(x_u, x_v)$ ) se determina cuál de los dos queda por encima en el intervalo  $[x, x']$ ; si, por ejemplo, fuera  $u$  entonces el segmento  $[L_u, x']$  se añade a  $S$  y fijamos  $x := x'$ . Nótese que, siguiendo con la suposición de que  $u$  era visible y  $v$  no hasta el punto de intersección, justo a continuación ocurre lo contrario, por lo que en la siguiente iteración el segmento  $[L_v, x_v]$  será añadido a  $S$ ,  $x := x_v$  y los segmentos  $[L_u, x_u]$  y  $[L_v, x_v]$  eliminados de  $S_1$  y  $S_2$ .

Como puede observarse en cada iteración se añade un segmento al perfil de visi-



bilidad, hasta un total de como mucho  $n$  segmentos (porque cada recta puede ser oculta o visible en **un** segmento). Por otro lado en caso peor ambos perfiles tienen  $\Theta(n)$  segmentos. El coste de todas las operaciones asociadas a cada iteración es  $\Theta(1)$ , por lo que el coste de MERGEPROFILES es  $\Theta(n)$ . Y el coste de HIDDENLINES en caso peor es  $\Theta(n \log n)$  puesto que satisface la recurrencia  $H(n) = 2H(n/2) + \Theta(n)$ .

**Solución 1.22** ■ Utilizaremos el esquema de divide y vencerás para determinar si existe un elemento mayoritario. Dividimos el conjunto de tarjetas en dos mitades y recursivamente obtenemos el elemento mayoritario, si lo hay, en cada de las dos mitades. Si el conjunto globalmente contiene un elemento mayoritario entonces dicho elemento tendrá que ser mayoritario en al menos una de las dos mitades. Es decir, si  $C$  contiene una tarjeta  $T$  equivalente a más de  $n/2$  tarjetas, entonces  $T$  será equivalente al menos a uno de los dos candidatos  $T_1$  y  $T_2$  obtenidos recursivamente uno. En una etapa recursiva, para determinar el mayoritario de un subconjunto  $C$ , se obtienen recursivamente dos candidatos  $T_1$  y  $T_2$  en los subconjuntos  $C_1$  y  $C_2$ , así como cuántos equivalentes tiene cada una de ellas en su respectivo subconjunto. Después se compara  $T_1$  con las tarjetas en  $C_2$  y obteniéndose el número de tarjetas equivalentes a  $T_1$  en  $C$ . Otro tanto se hace con  $T_2$  y  $C_1$ . Se devuelve como resultado para  $C$  al que mayor número de equivalentes tiene. EL coste no recursivo de nuestro algoritmo es  $\Theta(n)$  (cada tarjeta  $T_1$  y  $T_2$  se compara con  $\approx n/2$  en el otro subconjunto), así que el coste del algoritmo es

$$F(n) = \Theta(n) + 2F(n/2),$$

cuya solución es  $F(n) \in \Theta(n \log n)$ .

- Para entender el siguiente algoritmo, vamos primero a suponer que solo hay dos clases de tarjetas: las blancas y las negras. Habrá un empate o bien un color mayoritario pero no sabemos cuál. Lo que vamos a hacer es recorrer el conjunto de las  $n$  tarjetas, de manera que en todo momento  $c$  sea el color mayoritario hasta el momento y  $b$  el balance entre el color mayoritario y el minoritario (cuántas más del color mayoritario hemos visto frente al otro color). Cuando examinamos una nueva tarjeta si  $b = 0$  entonces haremos  $b := 1$  y el color  $c$  será el de la tarjeta que acabamos de examinar. Pero si  $b > 0$  entonces si el color de la tarjeta es  $c$  se incrementa  $b$  y si su color es el contrario se decrementa  $b$ . Claramente al finalizar el algoritmo habrá empate entre blancas y negras si  $b = 0$ , y si  $b > 0$  entonces  $c$  es el color mayoritario. Y el coste es claramente  $\Theta(n)$ .

Pues bien: el algoritmo también sirve con cualquier número de “colores” o clases de equivalencia, cada nuevo elemento examinado incrementa la cuenta del mayoritario o “aniquila” una aparición del mayoritario—con la salvedad de que el elemento que es “mayoritario” al finalizar el recorrido solo es un candidato a serlo!

Esto es: si  $C$  contiene un mayoritario el elemento declarado como candidato a mayoritario es, en efecto, el mayoritario de  $C$ . Pero si  $C$  no contiene un mayoritario el algoritmo nos devolverá un elemento cualquiera (que por supuesto aparece  $\leq$

$n/2$  veces). Por ello será necesario un segundo bucle, de coste  $\Theta(n)$ , para contar cuántas tarjetas son equivalentes a la tarjeta  $c$  y entonces reportar si  $t$  es la tarjeta mayoritaria, o bien que no existe tal tarjeta.

Este algoritmo es el famoso algoritmo MJRTY de Boyer y Moore (1980) (véase el [artículo sobre MJRTY en la Wikipedia](#))

**Solución 1.23** En una primera fase utilizaremos el algoritmo de selección de coste lineal en caso peor para hallar el  $(\log n)$ -ésimo elemento más pequeño de  $A$  y en una segunda invocación para hallar el  $(n - 3 \log n)$ -ésimo más grande. El algoritmo además habrá particionado el vector  $A[1..n]$  en tres bloques:

1.  $A[1.. \log n]$ , que contiene los  $\log n$  elementos menores del vector original
2.  $A[\log n + 1.. 3 \log n]$ , que contiene los elementos que nos interesan y que hay que ordenar
3.  $A[3 \log n + 1.. n]$ , que contiene los  $n - 3 \log n$  elementos más grandes del vector original

En la siguiente fase hay que ordenar los  $2 \log n$  números del segundo bloque mediante un algoritmo de ordenación como por ejemplo mergesort con coste lineal. Pero hay que tener en cuenta dos cosas: 1) solo hay que ordenar  $\Theta(\log n)$  elementos, lo cual necesitará  $\Theta(\log n \log \log n)$  comparaciones; 2) tanto en la primera fase de selección y partición, como en la segunda fase, de ordenación, cada comparación e intercambio de elementos es entre números de  $n$  bits. Así que el coste de la solución que proponemos es  $\Theta(n^2 + n \log n \log \log n) = \Theta(n^2)$ . Para la fase de ordenación podríamos utilizar *radix sort* para rebajar el coste a  $\Theta(n \log n)$  ( $n$  “pasadas” de coste  $\Theta(\log n)$  cada una, ya que hay  $\Theta(\log n)$  elementos solamente). Aún así el cambio no sería muy significativo porque el coste sigue siendo  $\Theta(n^2 + n \log n) = \Theta(n^2)$ .

Ahora bien este coste **es lineal** respecto al tamaño de la entrada, tal como requiere el enunciado: el tamaño de la entrada son  $n$  números de  $n$  bits cada uno, en total la entrada tiene  $n^2$  bits.

**Solución 1.24** Aplicamos el algoritmo de selección de coste en caso peor lineal para hallar la mediana y particionar el vector respecto a la mediana, todo ello en tiempo  $\mathcal{O}(n)$ . A continuación se aplica de nuevo el algoritmo de selección para encontrar el  $k$ -ésimo más pequeño en el subconjunto de los elementos más grandes que la mediana del paso anterior, al tiempo que se particiona el subvector. Al finalizar tendremos el vector organizado en tres bloques: en  $A[1..n/2]$  los elementos menores que la mediana; en  $A[n/2 + 1..n/2 + k]$  los elementos que queremos ordenar y en  $A[n/2 + k + 1..n]$  los mayores  $n/2 - k$  elementos de  $A$ . Solo resta ordenar el subvector de  $k$  elementos con un algoritmo como mergesort o heapsort, con coste  $\Theta(k \log k)$ .

**Solución 1.25** 1. Para simplificar supondremos que todos los elementos son estrictamente positivos. Si no fuera el caso, podemos “separar” con coste  $\Theta(n)$  los enteros

dados en tres grupos: positivos, negativos y nulos, pues basta mirar el signo, no hay que mirar todos los dígitos de cada número, y después se aplica el mismo tratamiento en el grupo de los negativos que en el de los positivos.

No podremos utilizar radix sort directamente porque el número de fases sería igual a la longitud del más largo de los números y es longitud puede ser  $\Theta(n)$  en caso peor, lo que nos daría coste  $\Theta(n^2)$ . La clave para poder resolver el problema es percatarse de que si  $a$  tiene más dígitos que  $b$  entonces  $a > b$ . Entonces lo que haremos será en primer lugar ordenar los elementos por número de dígitos usando *counting sort*. Denotemos a los elementos de la tabla  $x_1, \dots, x_n$  y sea  $d_i$  el número de dígitos de  $x_i$ . Averiguar cuántos dígitos tiene  $x_i$  tiene coste  $\Theta(d_i)$  (dividiendo repetidamente respecto a la base  $b$ ). Por tanto averiguar cuántos dígitos tienen todos los números tendrá coste  $\sum_i \Theta(d_i) = \Theta(\sum_i d_i) = \Theta(n)$ . Y aplicar el *counting sort* tendrá coste  $\Theta(n)$ . Una vez hecho esto aplicaremos radix sort con cada bloque de números de  $j$  dígitos. Sea  $n_j$  el número de elementos que tiene  $j$  dígitos. El coste de ordenar ese bloque con radix sort será  $\Theta(j \cdot n_j)$ . Y el coste de ordenar todos los bloques

$$\sum_{j=1}^n \Theta(j \cdot n_j) = \Theta\left(\sum_{j=1}^n j \cdot n_j\right) = \Theta(n).$$

2. En esta ocasión no sirve ordenar las cadenas por orden de longitud; dadas dos strings  $\alpha$  y  $\beta$ , que  $|\alpha| > |\beta|$  no implica que  $\alpha > \beta$  en orden alfabético. Para simplificar esta vez supondremos que todas las strings dadas son distintas—al final comentaremos cómo se puede abordar el problema de los repetidos. Una posible solución a este problema es construir un *trie*: el nodo raíz tiene  $|\Sigma| + 1$  subárboles, donde  $\Sigma = \{\sigma_1, \dots, \sigma_M\}$  es el alfabeto con el cual se forman las strings (e.g.,  $\Sigma = \{A, \dots, Z\}$  si todas las strings están formadas exclusivamente por letras mayúsculas). El primer subárbol es un trie para las strings que comienzan por  $\sigma_1$ , quitándoles dicho primer símbolo, el segundo subárbol es un trie para las strings que comienza con  $\sigma_2$ , etc. El  $(M+1)$ -ésimo subárbol contiene la strings vacía si ésta forma parte del conjunto de strings que hay que ordenar. Dicho de otro modo podemos pensar que cada string está acabada con un símbolo especial **end-of-string**, p.e., '\$', de tal forma que ninguna string va a ser prefijo de otra: 'barco\$' no es prefijo de 'barcos\$' (pero 'barco' sí es prefijo de 'barcos'). Esta propiedad (que ninguna string es prefijo de otra) es esencial para poder representar el conjunto de strings sin ambigüedad. La figura ilustra el *trie* correspondiente a el conjunto de strings  $X = \{\text{arco, abaco, barco, bota, cita, cima, cosa, bala, bravo, casa, botas, balada}\}$



## 4.2. Algoritmos Voraces

**Solució 2.28** Suposant que no ho tinguessim ja d'entrada, un recorregut del graf ens permet determinar el grau  $\deg[v]$  per a cada vèrtex del graf i marcar tots els vèrtexos com a potencials membres de  $V'$  ( $\text{keep}[v] := \text{true}$ ). Llavors posem tots els vèrtexos amb  $\deg[v] < k$  en una llista o cua  $Q$ . A continuació entrem en el següent bucle

```

while |Q| ≠ 0 do
  v := Q.POP()
  keep[v] := false
  for w ∈ G.ADJACENT(v) do
    if keep[w] then
      deg[w] := deg[w] - 1
      if deg[w] < k then
        Q.PUSH(w)
      end if
    end if
  end for
end while
// El resultat és el subgraf induït pels vèrtexos
// tals que keep[v] = true

```

El cost d'aquest algorisme és clarament  $\mathcal{O}(|V| + |E|)$  si el graf es representa amb llistes d'adjacència. D'altra banda tots els vèrtexos del subgraf induït retornat tenen grau  $\geq k$  per construcció i com que només es descarten aquells que tindrien grau  $< k$  en el subgraf induït la nostra solució és el subgraf més gran amb la propietat.

**Solució 2.29** Suposem per un moment, per facilitar la comprensió de la nostra solució, que el temps és discret, l'activitat  $i$ -èsima s'inicia en el temps  $s_i$  i acaba en el temps  $f_i$ , tots dos nombres naturals. Sigui  $T = \max\{f_i\}$ , i, sense pèrdua de generalitat, que alguna o algunes activitats s'inicia al instant 0. Així que tot passa entre 0 i  $T$ . Si tinguessim la funció  $R(t) = \text{número d'activitats actives a l'instant } t$ , llavors el nombre de recursos mínim necessari seria, òbviament

$$R^* = \max_{t \in [0, T]} R(t)$$

El problema és que una solució de força bruta seria massa ineficient. Si ordenem les activitats per ordre d'inici i considerem una certa activitat  $i$  podem fàcilment mantenir  $R^*$ : si alguna activitat  $j$  ocupant un recurs ha acabat ja ( $f_j \leq s_i$ ) el recurs ha quedat alliberat i l'activitat  $i$  el pot fer servir; si cap activitat que ocupa recursos ha finalitzat encara llavors assignem un nou recurs a l'activitat  $i$  (incrementem  $R^*$ ).

Per determinar eficientment si necessitem assignar o no un nou recurs a l'activitat  $i$  en curs, mantindrem un conjunt  $P$  amb activitats que ocupen recursos i mantenim aquest conjunt ordenat per temps de finalització, per exemple, podem fer un min-heap on  $f_j$  és la prioritat de l'activitat  $j$ . Quan processem una activitat  $i$  podríem eliminar del min-

heap totes les activitats amb  $f_j \leq s_i$ . Tanmateix, no és necessari, és suficient alliberar un recurs (si es pot) per a l'activitat “nouvinguda”.

```

procedure MINRECURSOS( $A$ )
  Ordenar les activitats  $A$  per  $s_i$  creixent i reindexar
  //  $s_1 \leq s_2 \leq s_3 \leq \dots \leq s_n$ 
   $P := \emptyset$ ;  $R^* := 1$ 
  //  $P$  és un min-heap d'activitats, ón la prioritat
  // de cada activitat és el seu temps d'acabament
  for  $i := 1$  to  $n$  do
     $\langle j, f_j \rangle := P.\text{TOP}()$ 
    //  $j$  és la actividad tal que  $s_j < s_i$  i  $f_j$  es mínima
    //  $j = -1$  si  $P$  és buida
    if  $j = -1 \vee f_j > s_i$  then
       $R^* := R^* + 1$ 
    else
       $P.\text{POP}()$ 
    end if
     $P.\text{PUSH}(\langle i, f_i \rangle)$ 
  end for
end procedure

```

Ordenar les activitats té cost  $\Theta(n \log n)$  (fent-ho amb un algorisme eficient d'ordenació). El bucle **for** fa  $n$  iteracions, cadascuna de les quals té cost  $\mathcal{O}(\log n)$  (per extreure un element del heap i afegir l'activitat en curs), així doncs, el cost global del nostre algorisme és  $\Theta(n \log n)$ .

**Solución 2.30** 1. Por ejemplo, para este conjunto de  $n = 3$  paquetes  $P_1 = (2000, 1)$ ,  $P_2 = (6000, 2)$  y  $P_3 = (7000, 1)$  emitidos en el orden indicado y con  $r = 5000$  observamos que la planificación es válida.

Instante	Bits transmitidos	Tasa (b/s)
1	2000	2000
2	5000	2500
3	8000	2666.66...
4	15000	3750

Sin embargo, el paquete  $P_3$  **no** cumple que  $b_3 = 7000 \leq rt_3 = 5000 \times 1 = 5000$ .

2. Para obtener una planificación válida ordenaremos los paquetes por tasa de transmisión  $b_i/t_i$  creciente y reindexamos de manera que asumimos  $b_1/t_1 \leq b_2/t_2 \leq \dots \leq b_n/t_n$ . A continuación con un bucle de coste lineal comprobamos que para todo  $i$ ,  $1 \leq i \leq n$ , se cumple que

$$\sum_{k=1}^i b_k \leq r \sum_{k=1}^i t_k.$$

Si se cumple entonces ordenar los paquetes en orden creciente de tasa de transmisión constituye una planificación válida. Pero lo más importante: si no se cumple con la planificación escogida entonces **no existe ninguna planificación válida**. El coste de nuestro algoritmo es  $\Theta(n \log n)$  para ordenar los paquetes, la comprobación posterior tiene coste  $\mathcal{O}(n)$ .

Para completar la demostración de corrección de nuestro algoritmo, tenemos que demostrar que si existe al menos una planificación válida, entonces la planificación “por orden creciente de tasa de transmisión” tiene que ser válida.

Dado un instante de tiempo  $t$ , llamemos  $B_\pi(t)$  al número de bits transmitidos en el periodo  $[0..t]$  según la planificación  $\pi$ . Supongamos que en el instante  $t$  se está transmitiendo el  $i$ -ésimo paquete de la planificación  $\pi$ . Si denotemos  $T_\pi(i)$  el tiempo de transmisión de los primeros  $i$  paquetes de la planificación  $\pi$ , entonces en el instante  $t$  se está transmitiendo el  $i$ -ésimo paquete de la planificación  $\pi(i)$  si

$$T_\pi(i-1) = \sum_{1 \leq k < i} t_{\pi(k)} < t$$

y

$$T_\pi(i) = \sum_{1 \leq k \leq i} t_{\pi(k)} \geq t.$$

Con estas definiciones podemos escribir que  $B_\pi(t)$  es

$$B_\pi(t) = \sum_{k=1}^{i-1} b_{\pi(k)} + \frac{b_{\pi(i)}}{t_{\pi(i)}} (t - T_{\pi(i-1)}).$$

Y la condición para que una planificación sea válida es que

$$B_\pi(t) \leq rt,$$

para todo  $t$ .

Supongamos que tenemos una planificación válida  $\Pi = (1, 2, \dots, i-1, j, \dots, i, \dots)$  para algún  $j > i$ ; es decir, los primeros paquetes transmitidos en  $\pi$  son los primeros  $i-1$  en orden de tasa  $b_k/t_k$  creciente, pero el  $i$ -ésimo que se transmite es el paquete  $j$ , es decir, uno que no es el  $i$ -ésimo en orden de tasa, y por tanto  $b_j/t_j \geq b_i/t_i$ . Consideremos ahora la planificación  $\Pi' = (1, 2, \dots, i-1, i, \dots, j, \dots)$ , esto es, una en la que se intercambian los paquetes  $i$  y  $j$ . La planificación  $\Pi'$  se parece un poco más a la planificación voraz  $\Pi_{\text{GREEDY}} = (1, 2, \dots, n)$  y vamos a demostrar que si  $\Pi$  es válida entonces  $\Pi'$  también lo es. Aplicando repetidamente este resultado podemos hacer todos los intercambios necesarios para concluir que la planificación voraz (orden creciente de tasa) es válida.

En primer lugar, para todo instante de tiempo  $t$  antes de que comience la transmisión del  $i$ -ésimo paquete  $\Pi$  y  $\Pi'$  hacen lo mismo, así que  $B_\Pi(t) = B_{\Pi'}(t) \leq rt$

si  $t \leq T_{\Pi}(i-1) = T_{\Pi'}(i-1)$ . Si el paquete  $i$  se transmite en  $k$ -ésimo lugar en  $\Pi$  ( $\Pi(k) = i$  y  $\Pi(i) = j$ ), entonces en  $\Pi'$  el  $k$ -ésimo paquete transmitido será el  $j$ . De manera que para todo  $t > T_{\Pi}(k) = T_{\Pi'}(k)$  vuelve a cumplirse  $B_{\Pi}(t) = B_{\Pi'}(t) \leq rt$ . Vemos ahora que sucede con  $t$  entre  $T_{\Pi}(i)$  y  $T_{\Pi}(k)$ .

$$\begin{aligned} B_{\Pi'}(t) &= B_{\Pi'}(T_{\Pi'}(i-1)) + \frac{b_i}{t_i}(t - T_{\Pi'}(i-1)) \\ &= B_{\Pi}(T_{\Pi}(i-1)) + \frac{b_i}{t_i}(t - T_{\Pi}(i-1)) \\ &\leq B_{\Pi}(T_{\Pi}(i-1)) + \frac{b_j}{t_j}(t - T_{\Pi}(i-1)) = B_{\Pi}(t) \leq rt \end{aligned}$$

donde en la desigualdad entre la segunda y la tercera línea utilizamos el hecho de que  $b_i/t_i \leq b_j/t_j$ .

**Solución 2.31** 1. Por ejemplo en un grafo con tres vértices  $A$ ,  $B$  y  $C$  con aristas  $(A, B)$  y  $(C, D)$  el conjunto  $\{A, C\}$  constituye un recubrimiento de vértices mínima, pero obviamente no mínimo.

2. Si  $C$  es un recubrimiento mínimo significa que al quitar cualquiera de sus vértices deja de ser un recubrimiento (sino, no sería un recubrimiento **mínimo**). Por lo tanto  $C$  también es minimal.

3. El siguiente algoritmo produce un recubrimiento minimal.

```

CE := E; C := ∅
while CE ≠ ∅ do
  e = (u, v) := CE.POP()
  C := C ∪ {u} ∪ {v}
  for w ∈ G.ADJACENT((u)) do
    CE := CE \ {(u, w)}
  end for
  for w ∈ G.ADJACENT((v)) do
    CE := CE \ {(v, w)}
  end for
end while
// C es un recubrimiento de vértices minimal

```

Si la eliminación de aristas ( $CE := CE \setminus \{(x, y)\}$ ) puede hacerse en tiempo constante entonces el coste del algoritmo es  $\mathcal{O}(|V| + |E|)$ . Para ello debe enlazarse los nodos que representan una misma arista en las listas de adyacencia. Por construcción el conjunto obtenido es minimal: si algún vértice en  $C$  se quita entonces una o más aristas adyacentes al vértice que quitamos no estaría cubierta.

Por otro lado el orden en el cual se extraen las aristas del conjunto  $CE$  dará lugar a soluciones donde  $C$  será más o menos grande. Una buena heurística es ordenar, en orden decreciente, las aristas por el mayor de los grados de los vértices en que



inciden o la suma de los grados. Aunque no garantiza que el recubrimiento obtenido sea minimal sí que nos proporcionará un recubrimiento con el mínimo de vértices o un número próximo al mínimo en muchas instancias, pues tendremos preferencia por incluir en  $C$  vértices que cibren muchas aristas.

De todas formas cualquiera que sea forma en que se escogen las aristas puede demostrarse que el algoritmo es una 2-aproximación: si  $R^*$  es un recubrimiento de vértices mínimo entonces

$$|R^*| \leq |R| \leq 2 \cdot |R^*|.$$

**Solución 2.32** En un primer paso clasificamos las tuplas de entrada por oficina de salida; llamemos  $R_\ell$  al conjunto de peticiones de coches que se hacen en la oficina  $\ell$  y supondremos que se ordena por tiempo de salida creciente. Este primer paso requiere coste  $\Theta(n \log n)$ . A continuación cada petición  $\sigma = (\ell, t, \ell', t')$  la insertamos en  $R_{\ell'}$  utilizando  $t'$  como criterio: esto es,  $\sigma$  se insertara entre peticiones  $\sigma_1 = (\ell', t_1, \dots)$  y  $\sigma_2 = (\ell', t_2, \dots)$  tales que  $t_1 \leq t \leq t_2$ . Cada  $R_\ell$  podemos implementarlo como un min-heap donde la prioridad de las peticiones  $(\ell, t, \ell', t')$  es el tiempo de salida  $t$  mientras que las devoluciones  $(\dots, \ell, t)$  que se insertan en la segunda fase tienen como prioridad el tiempo de la devolución.

Esta segunda fase nos da, para cada oficina, un conjunto ordenado de las salidas y de las devoluciones, por lo que un algoritmo similar al del problema #29 (pero aquí no tenemos intervalos!) nos proporcionará el número máximo de coches que se necesitará en cada oficina.

En la tercera y última fase, procesamos uno a uno los  $R'_\ell$ s, de la siguiente forma:

```

for  $\ell := 1 \rightarrow num\_oficinas$  do
  coches[ $\ell$ ] := 0; max_coches[ $\ell$ ] := 0
  while  $R_\ell \neq \emptyset$  do
     $\sigma := R_\ell.POP()$ 
    if  $\sigma = (\ell, t, \dots)$  then
      // Salida de un coche
      coches[ $\ell$ ] := coches[ $\ell$ ] + 1
      max_coches[ $\ell$ ] := máx(coches[ $\ell$ ], max_coches[ $\ell$ ])
    else
      // Devolución de un coche,  $\sigma = (\dots, \ell, t)$ 
      coches[ $\ell$ ] := coches[ $\ell$ ] - 1
    end if
  end while
end for

```

Este tercer bucle también tiene coste  $\mathcal{O}(\sum_\ell |R_\ell| \log |R_\ell|) = \mathcal{O}(n \log n)$ . Para cada oficina max\_coches[ $\ell$ ] es el número máximo necesarios para atender todas las solicitudes de la oficina en cuestión.

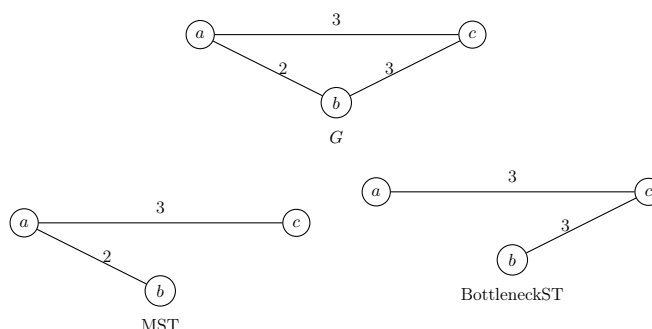
**Solución 2.34** La diferencia fundamental entre un bottleneck spanning tree y un minimum spanning tree es que la medida que tomamos sobre un árbol de expansión (ST) es,

en el primer caso, el peso máximo de las aristas en el árbol y, en el segundo la suma de los pesos de todas las aristas.

Las propiedades de un MST ya las habeis estudiado recordar las reglas rojas y azul.

Para los bottleneckST, por su definición, todos tienen la arista de peso máximo con el mismo valor.

1. ■ No todo bottleneckSTes un MST. Un contraejemplo es el siguiente



El BottleneckST tiene peso total 6 y no es un MST.

- Sí, un MST es un bottleneck spanning tree. La demostración la haremos por reducción al absurdo. Supongamos que un MST  $T$  no es un BottleneckST. Consideremos un BottleneckST  $T'$  y sea  $w$  el peso de la arista con peso mayor en  $T'$ . Esto quiere decir que:
  - (a)  $T$  tiene una arista  $e$  con peso mayor que  $w$ , si no sería BottleneckST.
  - (b) Todas las aristas en  $T'$  tienen peso menor o igual que  $w$ .

Si eliminamos  $e$  de  $T$  dividimos los vértices de  $G$  en dos partes. Pero al ser  $T'$  un ST algunas de las aristas de  $T'$  tienen que conectar estas dos partes.  $e$  no es la arista de peso mínimo en este corte por lo que deducimos que  $T$  no puede ser un MST, aplicado la regla azul.

2. Dados  $G$  y  $b$ , el algoritmo considera solo aquellas aristas con peso  $\leq b$  y con un BFS comprueba si el grafo es conexo o no. En el primer caso la respuesta es sí y en el segundo no.

Para que exista un BottleneckST en las condiciones que se pide, tiene que existir un ST en el que todas las aristas tengan peso  $\leq b$ , el algoritmo comprueba esta propiedad.

Podemos modificar la implementación del BFS para que trate solo las aristas con peso  $\leq b$  sin tener que modificar el grafo con coste  $O(n + m)$ .

**Solución 2.35** Supongamos que  $G$  tiene dos MST  $T$  i  $T'$ .

Consideremos un vértice  $u$ , el corte  $(\{u\}, V - \{u\})$ , tiene una única arista  $e = (u, v_1)$  con peso mínimo. Por la regla azul esta arista tiene que pertenecer a los dos árboles  $T$  i  $T'$ .

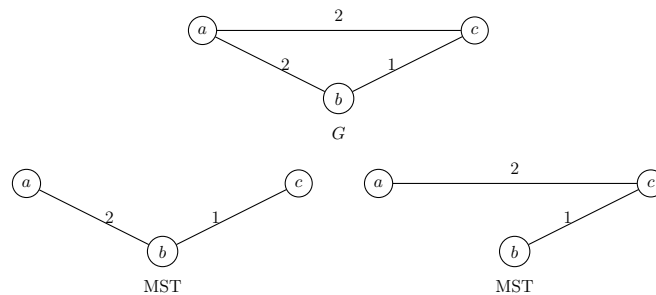
Si definimos  $S = \{u, v_1\}$ , tenemos que  $T$  y  $T'$  coinciden en  $S$ .

Consideremos el corte  $(S, V - S)$ , de nuevo solo hay una arista  $(v, v')$ ,  $v \in S$  y  $v' \notin S$ , con peso mínimo. Por la regla azul esta arista pertenece a  $T$  y a  $T'$ .

Si añadimos  $v'$  a  $S$ ,  $T$  y  $T'$  coinciden en este conjunto ampliado.

Repitiendo el proceso llegaremos a reintroducir todos los vértices de  $V$  en  $S$  y concluiremos que los dos MSTs son iguales.

Para demostrar que el contrarecíproco es falso nos basta con un contraejemplo



**Solución 2.36** 1. No. Al marcar un nodo, como los pesos son no negativos, solo podemos incrementar o mantener la puntuación. Si dejamos nodos sin marcar perdemos oportunidades de mejorar la puntuación.

2. Marcaremos los nodos en orden decreciente de valor.

El coste del algoritmo es  $O(n \log n)$  si solo marcamos los vértices (tendremos que ordenarlos) y  $O(n \log n + m)$  si queremos calcular la puntuación obtenida, ya que para cada nodo tenemos que acceder a su lista de vecinos para consultar las puntuaciones conseguidas.

Para demostrar la corrección de esta regla voraz, supongamos que tenemos un orden de marcado de los nodos, en el que aparecen todos, de mayor a menor valor.

Consideremos un nodo cualquiera  $u \in V$ . Denotaremos  $M(u)$  al conjunto de los vecinos de  $u$  que ya están marcados antes de marcar  $u$  y  $F(u) = N(u) - M(u)$  al conjunto de los vecinos de  $u$  que se marcan después. Por lo tanto  $u$  “contribuye” a la puntuación final que obtengamos con un total de :

$$\ell(u)|F(u)| + \sum_{v \in M(u)} \ell(v)$$

puntos.

Esta cantidad se maximiza cuando los vecinos con valor mayor que  $u$  se marcan antes que  $u$  y los que tienen valor menor después. Los vecinos con el mismo valor se pueden marcar antes o después de  $u$  ya que la contribución es la misma.

Al marcar en orden decreciente de valor se garantiza que la contribución de cada nodo sea la máxima posible y por lo tanto la puntuación obtenida es máxima.

3. No. Si consideramos un grafo como el del enunciado pero con valores  $\ell(A) = 5$ ,  $\ell(B) = -2$ , y  $\ell(C) = -3$ , el algoritmo marca los nodos en el orden  $A, B, C$  y con puntuación total 3. Pero podemos obtener puntuación 5 si no marcamos  $C$ .
4. Nos sirve el grafo del apartado anterior. Si prescindimos de  $B$  y  $C$  por tener valor negativo la puntuación obtenida es 0, mientras que marcando  $A$  y después  $B$  podemos conseguir puntuación 5.

**Solución 2.37** Si miramos el tablero del juego cada jugador es propietario de una parte del grafo correspondiente al tablero. Tenemos los bordes del tablero que son una zona especial, podemos añadir al tablero dos nodos adicionales por cada jugador, uno por cada borde, conectándolo a las casillas en el borde correspondiente

Un jugador gana si consigue que en su subgrafo estén los dos nodos especiales conectados, es decir en la misma componente conexa.

Como el juego es dinámico el mejor método para implementar la obtención de las componentes conexas es Kruskal que nos permite incorporar las aristas creadas en cada jugada una a una y recalculas las componentes conexas de forma incremental en cada jugada. El coste de Kruskal es  $O(m \log n)$ . En el tablero el número de aristas es proporcional al número de celdas. El número de celdas ocupadas es el número de jugadas, por lo que el coste total de mantener la partida es  $O(p \log p)$  donde  $p$  es el número de jugadas.

Nota: Hay diferentes formas de implementar Kruskal de forma eficiente, consultar la web o vuestras referencias. Utilizando una estructura de datos de union find se puede implementar eficientemente en tiempo  $O(m \log^* n)$ . Si tenéis tiempo os sugiero que leáis algo sobre las diferentes implementaciones de union-find.

**Solución 2.38** Como las casas están a lo largo del camino y las distancias se toman también sobre el camino podemos ordenar las casas de acuerdo con la posición en el camino de menor a mayor. Así tenemos  $c_1 \leq \dots \leq c_n$  la secuencia ordenada de posiciones (en el camino de las casas).

La primera casa tiene que estar cubierta por alguna estación SOS, la colocaremos lo más alejada posible de  $c_1$  en  $c_1 + 15$ . Una vez tengamos colocada una estación SOS eliminaremos las casas que están a distancia  $\leq 15$  de ella.

El algoritmo voraz aplicará esta regla hasta que la lista de ciudades a tratar quede vacía.

El algoritmo es correcto ya que coloca las estaciones SOS lo más alejadas posible de la primera casa que no está cubierta por las estaciones precedentes. Si consideramos una solución óptima, la primera estación SOS que no cumpla esta condición estaría a menos de 15Km de la primera ciudad y no las anteriores. Por ello avanzar la estación SOS en cualquier caso puede ayudarnos a reducir el número de estaciones, pero nunca lo aumentará.

El coste del algoritmo es  $O(n)$  si las ciudades nos las dan ordenadas o  $O(n \log n)$  si tenemos que ordenarlas como paso previo.

**Solución 2.40** Para cada  $j$ ,  $1 \leq j \leq n$ , la ficha situada en  $(x_j, y_j)$  tendrá que hacer exactamente  $|y_j - i|$  movimientos verticales, ni uno más ni uno menos, para situarla en la fila  $i$ . El total de movimientos verticales a efectuar es

$$V = \sum_{j=1}^n |y_j - i|,$$

que podemos calcular muy fácilmente con coste  $\Theta(n)$ .

Supongamos que  $s_k$  es el número de fichas tales que su  $x_j = k$ , es decir, el número de fichas que se apilarán en la posición  $(i, k)$  si trasladamos las  $n$  fichas con movimientos verticales hasta la fila  $i$ . Ahora nuestro algoritmo debe decidir “cómo” distribuir las  $s_k$  fichas hacia las casillas vacías en la fila  $i$  usando (y contando) el mínimo número posible de movimientos verticales. Observemos que

$$\sum_{k=1}^n s_k = n,$$

pues cada ficha está en alguna de las  $n$  columnas. Y por lo tanto si  $s_k > 1$  para alguna  $k$  entonces necesariamente tendrá que haber  $s_k - 1$  casillas vacías para acomodar esas fichas (todas menos una, que puede y debe quedarse en la columna  $k$ ).

Sea  $S_k = \sum_{j=1}^k s_j$  el número de fichas que están apiladas en las columnas 1 a  $k$  de la fila  $i$ . Si  $S_k = k$  entonces todas esas fichas pueden ponerse cada una en una columna distinta y no hay movimientos de fichas que pasen de la columna  $k$  a la  $k+1$  y sucesivas (porque no hay exceso) ni de la columna  $k+1$  a la  $k$  o anteriores porque en las columnas 1 a  $k$  ya no hay sitio. Si  $S_k > k$  entonces tenemos un excedente de  $S_k - k$  fichas que tendrán que ubicarse en las columnas  $k+1$  y siguientes. Si por otro lado  $S_k < k$  entonces tenemos columnas libres entre la 1 y la  $k$  y por ejemplo la primera columna libre en ese rango de columnas debe ser ocupada por la primera ficha “excedentaria”, es decir, por una ficha en la menor columna  $k'$  tal que  $s_{k'} > 1$  (notemos que  $k'$  puede ser menor o mayor que  $k$ ); en cualquier caso para rellenar todas las columnas disponibles necesitaremos que  $k - S_k$  fichas situadas en las columnas  $k+1$  a  $n$  se trasladen en horizontal para ocupar columnas entre la 1 y la  $k$ .

Podemos pensarlo incrementalmente, de manera que a la columna  $k$  le hacemos contabilizamos los movimientos horizontales de fichas que “transitan” por dicha columna: si  $S_k > k$  por ella pasarán  $S_k - k$  fichas en su viaje hacia un acomodo en las columnas  $k+1$  a  $n$  y hay que agregar  $S_k - k$  al total de movimientos; si  $S_k < k$  entonces hay que agregar los  $k - S_k$  movimientos horizontales del tránsito de fichas de  $k+1..n$  a  $1..k$ . En definitiva, este es el número de movimientos horizontales:

$$H = \sum_{k=1}^n |S_k - k|.$$

Y obviamente es mínimo pues siempre consideramos que por cada columna  $k$  transite el mínimo imprescindible de fichas del trozo  $[1..k]$  al trozo  $[k+1..n]$  o viceversa (y solo en uno de los dos sentidos o en ninguno!)

La función, con coste  $\Theta(n)$  nos queda así

```

int moves(const vector<int>& x, const vector<int>& y,
          int i) {
    int n = x.size()-1; // no uso la componente 0 de
                        // de los vectores x[0..n], y[0..n]
                        // ni en los vectores s y S locales

    int V = 0;
    for (int i = j; j <= n; ++j)
        V += abs(y[j]-i);

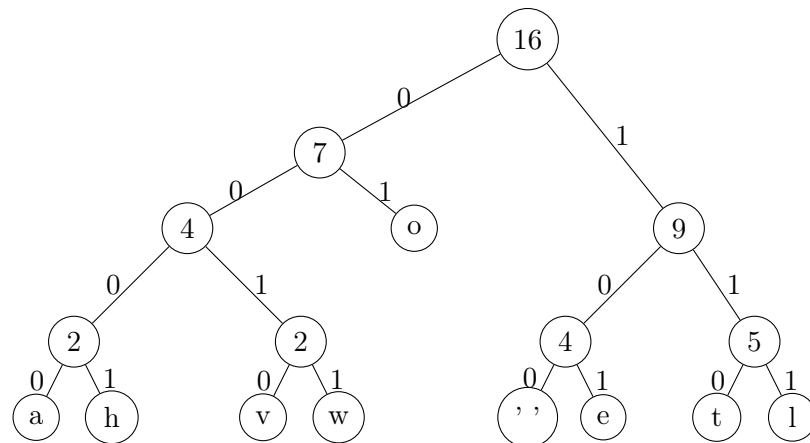
    vector<int> s(n+1, 0);
    for (int j = 1; j <= n; ++j)
        ++s[x[j]];
    // s[k] = fichas en la columna k

    vector<int> S(n+1, 0);
    for (int j = 1; j <= n; ++j)
        S[j] = s[j] + S[j-1];
    // S[k] = fichas en las columnas 1 a k = s[1]+...+s[k]

    int H = 0;
    for (int k = 1; k <= n; ++k)
        H += abs(S[k]-k);
    return H+V;
}

```

**Solución 2.41** L'arbre de Huffman és:

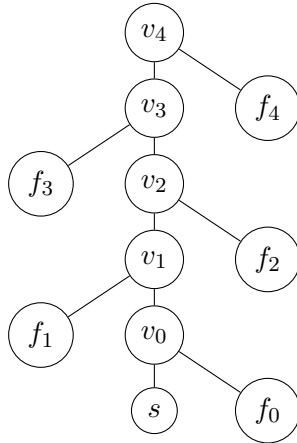


El títol de la cançó és “whole lotta love”.

**Solución 2.42** Fijamos un símbolo cualquiera  $a$ . Llamemos  $T$  al árbol con el código de Huffman del alfabeto. Localizamos la hoja  $s$  que contiene el símbolo  $a$  y consideramos el

camino  $P = s, v_0, v_1, \dots, v_k$  formado por la secuencia de nodos en el camino que va de  $s$  hasta la raíz del árbol. Por tanto  $k + 1$  es la profundidad en la que aparece  $a$ . Vamos a calcular una cota superior a  $k$ . Para ello analizaremos el caso peor.

Llamemos  $f_i$ ,  $1 \leq i \leq k$  al otro hijo de  $v_i$ . Estos nodos pueden aparecer a la derecha o a la izquierda, dependiendo del algoritmo usado. Por ejemplo un posible árbol, para  $k = 4$ , sería:



Teniendo en cuenta la construcción del árbol sabemos que:

- $p(v_0) \geq p(s) = p_a$
- Para  $i \geq 1$ ,  $p(v_i) = p(f_i) + p(v_{i-1})$
- Para  $i \geq 2$ ,  $p(f_i) \geq p(v_{i-2})$  y  $p(f_i) \geq p(f_{i-1})$ , si no fuese así habríamos seleccionado antes  $f_i$  y no sería hermano de  $v_{i-1}$ , el padre de  $v_{i-2}$  y  $f_{i-1}$ .

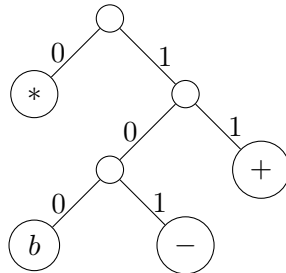
De la última propiedad tenemos  $2p(f_i) \geq p(v_{i-2}) + p(f_{i-1}) = p(v_{i-1})$ .

Por otra parte  $p(v_i) = p(f_i) + p(v_{i-1}) \geq \frac{p(v_{i-1})}{2} + p(v_{i-1}) \geq \frac{3}{2}p(v_{i-1})$ .

Deducimos que  $p(v_k) \geq \left(\frac{3}{2}\right)^k p(v_0) \geq (1,5)^k p_a$ .

Como  $v_k$  es la raíz del árbol  $p(v_k) = 1$ , tenemos la ecuación  $(1,5)^k p_a \leq 1$ . Así  $k \log 1,5 \leq \log \frac{1}{p_a}$ , y deducimos  $k = O(\log \frac{1}{p_a})$ .

**Solución 2.43**  $f(+) = 3/13$ ,  $f(-) = 3/13$ ,  $f(\star) = 5/13$ ,  $f(b) = 2/13$ . L'arbre prefixat es



que dona una compressió 11100101100110011101010000. Notem que el factor de compressió  $\alpha = 2$ . Quan utilitzem una compressió de longitud fixa, ex.  $\phi(+) = 00, \phi(-) = 01, \phi(\star) = 10, \phi(b) = 11$ , el resultat és 00011101001010001110011111 que té la mateixa longitud. Però a mida que el missatge sigui més llarg, la grandària de la compressió variable serà més petita que la grandària de la compressió fixa.

**Solució 2.44** La idea es ir eliminando en sucesivas iteraciones las hojas de árbol hasta que nos quede un único vértice o una sola arista; en el primer caso el vértice que queda es el centro del árbol  $T$  y en el segundo caso los extremos de las aristas son centros del árbol. Al ir eliminando hojas debemos eliminar en primer lugar las que eran hojas en el árbol original  $T$ , después las hojas en el árbol  $T'$  resultante de eliminar las hojas de  $T$ , y así sucesivamente.

El centro o centros del árbol  $T$  se encuentran justo en medio del camino entre las dos hojas más distantes en  $T$  y por tanto el procedimiento descrito tiene como resultado final un árbol  $T^*$  que contiene exclusivamente el centro o centros de  $T$ .

Para implementar eficientemente este algoritmo se inicializa un conjunto  $C$  con todos los vértices de  $T$  y una cola  $Q$  con todos los vértices de  $T$  de grado 1. En cada iteración se extrae un vértice  $v$  (es una hoja) de la cola  $Q$  y se elimina de  $T$ ; además el grado del vértice adyacente a  $v$  disminuye en 1 y si pasa a ser igual a 1, dicho vértice se coloca en una cola de hojas  $Q'$  para la siguiente ronda. El algoritmo se detiene cuando  $C$  contiene solo 1 o 2 vértices unidos por una arista. Gracias al uso de la cola el coste del algoritmo es  $O(n)$ , pues cada vértice excepto el centro o centros entrará y saldrá exactamente una vez en la cola  $Q$  (a través de  $Q'$ ) y también se habrá añadido y posteriormente eliminado de  $C$  exactamente una vez. Dado que todas las operaciones efectuadas para cada vértice tienen coste  $O(1)$  y que el número de aristas (una arista  $\implies$  una actualización del grado de un vértice) es  $n - 1$  el coste del algoritmo es  $O(n)$ .

```

C := V(T)
Calcular el grado  $grad[v]$  de todo vértice  $v \in T$ 
// Coste:  $\Theta(n)$ 
Q :=  $\emptyset$ 
for  $v \in C$  do
    if  $grad[v] = 1$  then
        Q.PUSH(v)
    end if
end for
// Q contiene las hojas de C
// Coste:  $\Theta(n)$ 

while  $|C| > 2$  do
    Q' :=  $\emptyset$ 
    // C = Cini  $\wedge$  Q = hojas de C
    while  $|Q| \neq 0$  do
        u := Q.FRONT()
        for  $v \in T.ADJACENT(u)$  do

```



```

    grad[v] := grad[v] - 1
    if grad[v] = 1 then
        Q'.PUSH(v)
    end if
end for
C := C \ {u}
end while
// C = Cini \ hojas de Cini
// Q' = hojas de C
Q ↔ Q' // Coste: Θ(1)
end while

```

Alternativamente podemos hacer lo siguiente

1. Lanzamos un recorrido en anchura desde un nodo  $u$  cualquiera de  $T$ . Sea  $v$  un nodo a máxima distancia de  $u$ . Dicho nodo es necesariamente una hoja en  $T$  (un nodo de grado 1)
2. Lanzamos un BFS desde  $v$ . Sea  $w$  un nodo (= hoja) a máxima distancia de  $v$ .
3. El elemento o elementos en el punto medio del camino de  $v$  a  $w$  son los centros del árbol. Si dicho camino es de longitud par  $2\ell$  el nodo a distancia  $\ell$  de  $v$  y  $w$  es el centro. Si el camino es de longitud impar  $2\ell + 1$  hay dos nodos  $x$  e  $y$  unidos por una arista a distancias  $\ell$  y  $\ell + 1$  de  $v$  y de  $w$  ( $d(x, v) = \ell$ ,  $d(x, w) = \ell + 1$ ,  $d(y, v) = \ell + 1$ ,  $d(y, w) = \ell$ ), ambo son centros de  $T$ .

Todos los recorridos necesarios para este segundo algoritmo toman tiempo  $\Theta(n + m) = \Theta(n)$  puesto que  $m = n - 1$ .

**Solución 2.45** Teniendo en cuenta que cada anuncio dura 1 minuto y que hay  $n$  anuncios podemos programar un anuncio para iniciarse en el minuto 0, otro para iniciarse en el minuto 1, y así hasta programar el anuncio a emitir entre los instantes  $n - 1$  y  $n$ . Sin pérdida de generalidad, supongamos que indexamos los anuncios de mayor a menor beneficio, esto es,  $b_1 \geq b_2 \geq \dots \geq b_n$ . Para cada uno de ellos tenemos que asignarle un slot  $[e_i - 1, e_i]$ , el minuto en que se emitirá. Entonces nuestro objetivo es hallar una permutación  $e = (e_1, \dots, e_n)$  de los números del 1 al  $n$  que maximiza el valor

$$\text{val}(e) = \sum_{i=1}^n b_i \cdot \llbracket e_i \leq t_i \rrbracket,$$

donde  $\llbracket P \rrbracket = 1$  si  $P$  es cierta y  $\llbracket P \rrbracket = 0$  en otro caso.

Nuestra estrategia voraz ordena los anuncios de mayor a menor beneficio y asigna a cada anuncio, sucesivamente del primero al último, el slot más tardío no asignado todavía en el periodo  $[0, t_i]$ , y si no hay ninguno entonces el slot más tardío no asignado en  $[t_i, n]$ . Esta estrategia puede implementarse trivialmente con coste  $O(n^2)$  (y con un esfuerzo considerable y estructuras de datos apropiadas con coste asintóticamente menor).

Asignar el anuncio  $i$  a cualquier slot anterior al asignado por el algoritmo voraz no aporta valor a la asignación y sí puede ocasionar que un anuncio  $k$  que podría reportarnos valor deje de hacerlo al encontrar un slot válido ya ocupado. Por otro lado no se puede asignar al anuncio  $i$  a ningún slot posterior al asignado por el algoritmo voraz o bien si se hace entonces se pierde el valor  $b_i$  que el algoritmo voraz sí obtiene. El argumento previo muestra que si tenemos asignar los slots para las anuncios en un cierto orden  $\sigma$ , para el anuncio  $\sigma(i)$  siempre interesará asignarle el slot libre más tardío dentro del rango  $[0, t_{\sigma(i)}]$  si es posible, el más tardío que esté libre en el rango  $[t_{\sigma(i)}, n]$  sino.

Resumiendo: una vez fijamos un orden entre los anuncios para ir asignándoles slots de tiempo la mejor estrategia posible es la que adopta el algoritmo voraz.

Para concluir nuestra demostración mostraremos que el orden para hacer las asignaciones por valor ( $b_i$ ) decreciente es óptimo. Supongamos lo contrario. Que existe alguna  $i$  tal que es mejor que el  $i$ -ésimo anuncio al que se le asigna slot tiene valor menor que el del  $(i + 1)$ -ésimo anuncio al que asignamos slots, esto es,  $b_i < b_{i+1}$ . Los slots asignados a ambos anuncios estaban libres justo con la asignación de los primeros  $i - 1$  anuncios. En los casos en que ambos anuncios consiguen ser emitidos en plazo o ninguno de los dos consigue ser emitido en plazo, podríamos intercambiar el orden en que se hacen las asignaciones y nada cambiaría. Otro tanto sucede si en la asignación óptima el anuncio  $i$  no puede ser emitido en plazo pero el  $i + 1$  sí: intercambiando el orden de asignación esto seguirá cumpliéndose, el  $i + 1$  se podrá emitir en plazo. Finalmente consideremos la situación en la que el anuncio  $i$ -ésimo ha podido emitirse en plazo pero el  $i + 1$  no. Si  $t_i \leq t_{i+1}$  el slot que se le asigna al anuncio  $i$  se le podría asignar al  $i + 1$ , y el  $i$  no se podría emitir en plazo. Pero obtenemos mejor valor emitiendo el  $i + 1$  que el  $i$ , así que dicha situación no es posible, sería una contradicción. Si  $t_{i+1} < t_i$ , se deduce que intercambiando el orden de asignación de los anuncios  $i$  e  $i + 1$  o bien el anuncio  $i$  se le sigue asignando el mismo slot que antes y al  $i + 1$  un slot más allá del  $t_i$ , o bien a ambos se les asignan slots antes de  $t_i$  y por tanto el anuncio  $i$  sigue consiguiendo su valor  $b_i$ .

Recapitulemos: si existe  $i$  con  $b_i < b_{i+1}$  en el orden para asignar slots, el intercambio de  $i$  e  $i + 1$  no altera el valor. O lo empeora, lo cual sería una contradicción. El argumento puede aplicarse repetidas veces para “eliminar” todas las  $i$  tales que  $b_i < b_{i+1}$  y concluir que el orden para asignar empleado por el voraz es óptimo.

**Solución 2.46** El criterio de ordenar las tareas en orden decreciente de factor de penalización no proporciona una solución óptima. Por ejemplo considerad este conjunto

den = 3 tareas: 

	1	2	3
d	4	4	3
p	16	12	10

 ya en orden de factor de penalización decreciente. La penalización si las tareas se ejecutan en ese orden es

$$16 \cdot 4 + 12 \cdot 8 + 10 \cdot 11 = 64 + 96 + 110 = 270$$

pero si las ejecutáramos en el orden (1, 3, 2) la penalización sería

$$16 \cdot 4 + 10 \cdot 7 + 12 \cdot 11 = 64 + 70 + 132 = 266,$$

lo que demuestra que ordenar decrecientemente por penalización no nos da una solución óptima. Lo que conviene es ejecutar las tareas por orden decreciente del cociente  $p_i/d_i$ , esto es, ejecutar en primer lugar las tareas con menor penalización por unidad de tiempo.

Consideremos una planificación óptima que no cumple con el criterio. Existirá por lo tanto un valor  $i$  tal que  $p_i/d_i < p_{i+1}/d_{i+1}$ , donde  $p_k$  es la penalización de la tarea que se ejecuta en  $k$ -ésimo lugar en la planificación óptima y  $d_k$  su duración. Denotemos  $T_i = d_1 + \dots + d_{i-1}$  el tiempo de ejecución de las tareas previas a la tarea  $i$ -ésima y  $R_i$  la suma de penalizaciones de todas las tareas, excluidas las tareas  $i$  e  $(i+1)$ -ésima. Por lo tanto la penalización de la solución óptima es

$$P^* = R_i + (T_i + d_i)p_i + (T_i + d_i + d_{i+1})p_{i+1}.$$

Si consideramos ahora una nueva planificación idéntica a la óptima excepto que se intercambian las tareas  $i$  e  $i+1$  entonces la penalización pasa a ser

$$P' = R_i + (T_i + d_{i+1})p_{i+1} + (T_i + d_{i+1} + d_i)p_i,$$

puesto que el tiempo de finalización de cualquier tarea excepto la  $i$  y la  $i+1$  sigue siendo el mismo que en la planificación óptima. Ahora bien

$$P' = P^* - d_i p_{i+1} + d_{i+1} p_i,$$

pero por hipótesis  $p_i d_{i+1} < d_i p_{i+1}$  de donde se concluye que  $P' < P^*$  y tenemos una contradicción, puesto que  $P^*$  ha de ser mínima.

**Solución 2.48** 1. El grau d'un vertex es com am molt  $n$ . Podem ordenar en ordre decreixent de grau en temps  $O(n)$  fent servir counting sort. A cada pas, l'algorisme accedeix només a la llista del seus veïns per comptar quants són a cada costat de la partició i decidir a on s'afegeix. El cost total es  $O(n + m)$ .

Per una altre part l'algorisme sempre produeix una partició del conjunt de vèrtexos, per tant és trivialment correcte.

2. Per a cada aresta  $e_i = (u_i, v_i)$  definim un *vèrtex responsable* (de  $e_i$ ) com el primer vertex d'entre  $u$  o  $v$  que tracta l'algorisme. Sigui  $r_i$  el nombre d'arestes de les que  $v_i \in V$  és responsable. Com cada aresta té exactament un vèrtex responsable, aleshores  $\sum_{i=1}^n r_i = m$ .

Quan l'algorisme tracta el vèrtex  $v_i$ , estarem afegint al menys  $r_i/2$  arestes al tall  $C$ . Els extrems  $\neq v_i$  de les  $r_i$  arestes no han estat processats encara per l'algorisme i per tant no s'ha dedicat si cauen del mateix costat que  $v_i$  o al costat contrari. I sempre es posarà al menys la meitat al costat contrari, doncs altrament no estariem escollint en cada pas el vèrtex que té grau màxim d'entre els que encara no s'ha visitat i col·locant-ho al costat que maximitza el nombre d'arestes que creua el tall.

Per tant, el nombre d'arestes al  $C$  final  $\geq \sum_{i=1}^n r_i/2 \geq m/2$ . Com tenim un problema de maximització, el  $S^*$  optim és  $\leq m$ . Per tant l'algorisme proposat es una 2-aproximació.

**Solución 2.49** Podemos utilizar el mismo algoritmo que para el problema anterior, quedándonos con las aristas del corte. El algoritmo se ejecuta en tiempo polinómico. Proporciona un grafo bipartido, que por supuesto no tiene triángulos. Ya que el número de aristas elegidas por el algoritmo voraz cumple que es  $\geq m/2$  y que el número óptimo (máximo) de aristas en el subgrafo  $G'$  libre de triángulos es  $\leq m$ , el algoritmo nos da también una 2-aproximación para este nuevo problema.

**Solución 2.50** 1. En este caso, la solución que cubre todas las ciudades es un 0-recubrimiento, cualquier distancia es igual o mayor a 0. Por lo tanto no lo podremos mejorar y ha de ser un óptimo.

2. El algoritmo es:

```

 $r := 0;$ 
for  $s \in S$  do
  if  $s \notin C$  then
     $r' := \infty;$ 
    for  $c \in C$  do;
       $r' := \min(r', \text{dist}(s, c));$ 
    end for
     $r := \max(r, r');$ 
  end if
end for
return  $r$ 

```

Por cada ciudad  $s \in S$ , en el caso de que no pertenezca al conjunto  $C$ , calculamos su distancia con  $C$ , definida como  $d(s, C) = \min_{c \in C} d(s, c)$ . Y nos quedamos siempre con la mayor de las distancias obtenidas, esto es,

$$r(C) = \max_{s \in V \setminus C} d(s, C)$$

Obsérvese que una solución óptima será un subconjunto  $C^*$  de  $k$  ciudades tal que  $r(C^*)$  es mínima entre las  $r$ 's de todos los posibles subconjuntos  $C$  de  $k$  ciudades.

El coste del bucle interno es  $O(k)$  y se ejecuta  $n - k$  veces. Por lo tanto el coste total del algoritmo es  $O(nk)$ .

3. Si partimos de un grafo  $G = (V, E)$ , podemos calcular las distancias  $d(u, v)$  entre cualquier par de vértices. Considerando como conjunto de ciudades  $V$  y la distancia  $d$ , para un valor de  $k$ ,  $r(C)$  es igual a 1 si y solo si tenemos una selección de vértices para la que todos los demás vértices están a distancia  $\leq 1$ . Lo que es equivalente a decir que la selección es un conjunto dominador de tamaño  $k$  en  $G$ . DOMINATING-SET es un problema NP-completo, por lo tanto si pudiésemos resolver nuestro problema en tiempo polinómico  $P=NP$ , por lo que el problema es NP-difícil.
4. El algoritmo voraz propuesto tiene coste polinómico. Sea  $C^*$  la selección óptima. Al finalizar el algoritmo voraz,  $C$  es una selección de  $k$  ciudades, por tanto  $r(C^*) \leq r(C)$ . Para tener una 2-aproximación tenemos que demostrar que  $r(C) \leq 2r(C^*)$ .

Supongamos que no es cierto, es decir,  $r(C^*) < r(C)/2$ .

Observemos en primer lugar que, al finalizar el algoritmo, cualquier par de centros en  $C$  están a distancia  $> r(C)/2$ .

Por cada ciudad con centro comercial  $c_i \in C$ , consideramos el subconjunto de las ciudades  $C_i$  cuya distancia a  $c_i$  es  $\leq r(C)/2$ . Como al finalizar el algoritmo, cualquier par de centros en  $C$  están a distancia  $> r(C)/2$ , los  $C_i$  son disjuntos. Y la unión de los  $C_i$ 's es el conjunto de todos los vértices.

Además, como por hipótesis  $r(C^*) < r(C)/2$ , en cada  $C_i$  tenemos que tener un elemento de  $C^*$ . Como  $|C| = |C^*|$ , cada  $C_i$  contiene exactamente un único elemento de  $C^*$  al que llamamos  $c_i^*$ . Además,  $c_i^*$  es el centro en  $C^*$  que está más cerca de  $c_i$ , luego  $d(c_i, c_i^*) \leq r(C^*)$ .

Para una ciudad cualquiera  $s$ , sea  $c_i^*$  su centro más cercano en  $C^*$  y  $C_i$  el subconjunto de ciudades en el que viene incluida por el algoritmo voraz. Tenemos entonces

$$d(s, C) \leq d(s, c_i) \leq d(s, c_i^*) + d(c_i^*, c_i) \leq 2r(C^*),$$

donde la segunda desigualdad se deduce de la desigualdad triangular y la tercera se deduce de que toda distancia entre una ciudad y su centro en  $C^*$  ha de ser necesariamente  $\leq r(C^*)$  y  $d(c_i, c_i^*) \leq r(C^*)$ . Con lo que llegamos a una contradicción, porque existe alguna ciudad  $s$  que nos da el valor  $r(C)$ , esto es,  $r(C) = d(s, C)$  y concluimos entonces que  $d(s, C) = r(C) \leq 2r(C^*)$ , contradiciendo nuestra hipótesis de que  $r(C) > 2r(C^*)$  (o  $r(C)/2 > r(C^*)$ ).