Algorítmica

Colección de Problemas

Septiembre de 2020

Maria José Blesa
Josep Díaz
Conrado Martínez
Maria José Serna

Índice general

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.

1

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 $p = 4$:

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

   Dissenyeu 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) Dissenyeu 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 $A[i]$. Dissenyeu 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 $r = (r_1, \ldots, r_n)$, amb $n > 1$, és un vector de valors no negatius, el *coeficient de Gini* es defineix com:

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

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 $(n \mod i \neq 0)$ **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 α, β són literals), G_F té un arc des de la negació d'α a β, i un de la negació de β a α.

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 $f_1; f_2; \ldots$, aleshores $f_2 = \Omega(f_1); f_3 = \Omega(f_2)$; etc.

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

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 $\sum_{j=1}^{\log n} 4^j = O(n^2)$.

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 voli 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 $E' = \{(u, v) | \exists w \in V, (u, w) \in E \wedge (w, v) \in E\}$. 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 què 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 subcadenes 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 $(u, \vec{v})$ 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 irrellevant. 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 anàlisis 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 $1, 2, \ldots, n$, on $n = |V|$
   2 Si $n = 1$ tornar V
   3 Dividir V en $V_1 = \{1, 2, \ldots, \lfloor n/2 \rfloor\}$ i $V_2 = \{\lfloor n/2 \rfloor + 1, \ldots, n\}$
   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=CLIQUE(G_1) i C_2=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:

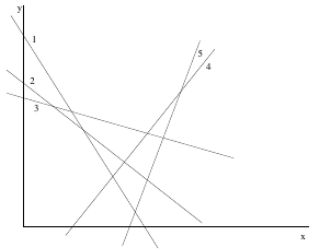
(a) Demostreu que l'algorisme CLIQUE sempre retorna un subgraf de G que és un clique.

(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, $L_1, \ldots, L_n$, 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 $\mathcal{L} = \{L_1, \ldots, L_n\}$, 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. Suposeu que sou consultors per a un banc que està molt amoïnat amb el tema de la detecció de fraus. 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 un 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 dígits. Però sabem que el nombre total de dígits 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 al 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?

2

Soluciones

2.1. 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 ressòl 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 ≤ 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 $A[i] < \cdots < A[k]$ per $i < k$ i podem prescindir de forma segura els elements $A[i \cdots k]$. De la mateixa manera, si $A[k] > A[k+1]$, sabem que $A[k+1] > \cdots > A[j]$ per $j > k+1$ i podem descartar amb seguretat els elements $A[k+1 \cdots j]$. 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, $T(n) = O(\log n)$.

Solución 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 $O(n^2)$.

Solución 1.18: Sigui AGRUPAR l'algorisme recursiu que, té com a entrada una taula de alumnes-notes N i dos enters ℓ 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, $T(n) = \Theta(n \lg k)$.

Solución 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

Soluciones del Profesor

3.1. Repaso de Conceptos Básicos

Solución 1.2:

1. Solucion en pseudocódigo:

```
for i := 0 to n do
    B[i] := false
end for
for i := 0 to n − 1 do
    B[A[i]] := true
end for
// Hallar j tal que B[j] = false
j := 0
while ¬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 ℓ-ésimo vale 0 y elementos cuyo bit ℓ-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 ℓ-ésimo bit del resultado como el bit ℓ-ésimo del grupo **no** descartado.

d) Fijar $\ell := \ell + 1$ y repetir el segundo paso hasta que el grupo no descartado esté vació.

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 $i = k-1$: $r_1 - r_k$, $r_2 - r_k$, ..., $r_{k-1} - r_k$. De manera parecida r_k aparece en S sumando para $j = k+1$ hasta $j = n$: $r_k - r_{k+1}$, ..., $r_k - r_n$. 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[1] (coste de evaluar la condición y del cuerpo del bucle), y sumaremos para todas las i psoibles: 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

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

   aplicando la fórmula de Stirling[2]:

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

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

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

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 $2^{2^k} \ge i$ 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 $\ge \lg\lg(n/2)$; pero $n/2 \lg\lg(n/2) \in \Omega(n \log\log n)$, luego el coste del algoritmo es $\Theta(n \log\log n)$.

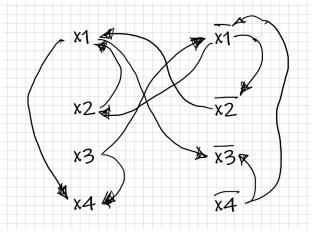
[1] Aquí la i es genérica. La variable que controla el bucle no tiene porqué llamarse `i`.

[2] La notación $a_n \sim b_n$ significa que $\lim_{n\to\infty} a_n/b_n = 1$. 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 $n \mod i \neq 0$ se cumple siempre y se harán $\Theta(\sqrt{n})$. Con la simplificación de que las operaciones aritméticas tiene coste constante (calcular i^2 y comparar con n, incrementar la i, hacer $n \mod i$, ...) 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 $\Theta\left(\left(\sqrt{2}\right)^N\right) = \Theta(1{,}4142\ldots^N)$, 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 $x \implies \cdots \implies \neg x$ y $\neg x \implies \cdots \implies x$. Pero la primer 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. PENDIENTE

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, puede hacerse 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 (obtner 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 ceelbridad 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 $\{i, \ldots, j\}$.

$i := 1$; $j := n$ // $n = |V|$
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

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

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):

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

si f y g son funciones derivables.

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

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

2. Cierta. Puesto que

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

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

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

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

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

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

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 $c = c' \cdot \text{máx}\, f(i) \,|\, 1 \leq i \leq n_0$ y $c \cdot f(n) \geq g(n)$ para toda $n > 0$. Ahora, aplicando la hipótesis de inducción:

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

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

Solución 1.10: PENDIENTE

Solución 1.11: PENDIENTE

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 algirtmo 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 $\Theta(|V|^{\log_2 7}) = \Theta(|V|^{2,81\cdots})$; 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 $(i, j) \in E'$.

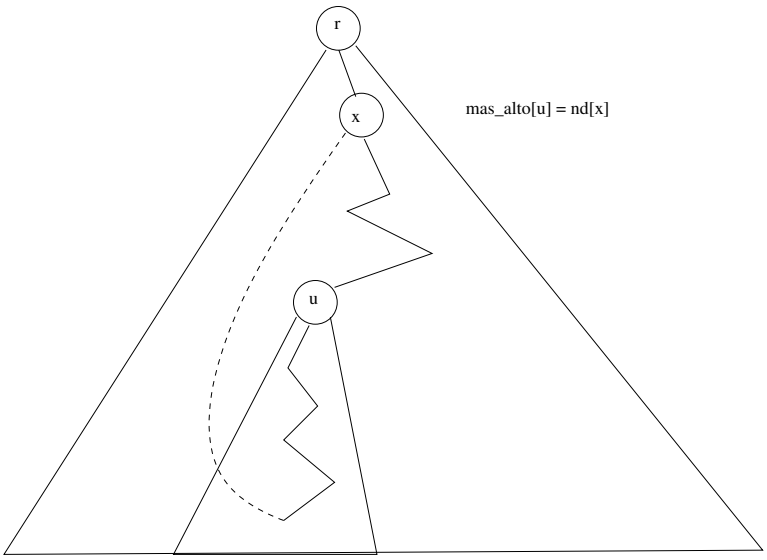
```
procedure SQUARE(G)
    // Retorna el cuadrado de G
    V' := V; E' := ∅
    for (i, k) ∈ E do
        for j ∈ G.ADJACENT(k) do
            Añadir (i, j) a E'
        end for
    end for
    return G' = ⟨V', E'⟩
end procedure
```

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[3] 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.

[3] 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_{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_{DFS}: es punto de articulación si y sólo si tiene más de un descendiente en el T_{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 $mas\_alto[w] \geq ndfs[u]$, es decir, si algún descendiente no puede "escapar" del subárbol enraizado en u sin pasar por u.

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

procedure PUNTOSARTICULACION(G)
  // Pre: G es conexo
  for $v \in V(G)$ **do**
    $visitado[v] :=$ **false**
    $ndfs[v] := 0$
    $punto\_art[v] :=$ **false**
    $num\_desc[v] := 0$
  end for
  $num\_dfs := 0$
  $es\_biconexo :=$ **true**
  // Basta lanzar un DFS porque G es conexo

```
    v := un vértice cualquiera de G
    PuntosArticulacion-Rec(G, v, v, es_biconexo, ndfs, ...)
    return ⟨es_biconexo, punto_art⟩
end procedure
procedure PuntosArticulacion-Rec(G, v, padre, ...)
    num_dfs := num_dfs + 1; ndfs[v] := num_dfs
    visitado[v] := true
    mas_alto[v] := ndfs[v]
    for w ∈ G.Adjacent(v) do
        if ¬visitado[w] then
            num_desc[v] := num_desc[v] + 1
            PuntosArticulacion-Rec(G, w, v, ...)
            mas_alto[v] := min(mas_alto[v], mas_alto[w])
            punto_art[v] := punto_art[v] ∨ (mas_alto[w] ≥ ndfs[v])
        else if padre ≠ w then
            mas_alto[v] := min(mas_alto[v], ndfs[w])
        end if
    end for
    if v = padre then // v es la raíz del T_DFS
        punto_art[v] := num_desc[v] > 1
    end if
    es_biconexo := es_biconexo ∧ ¬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 del digrafo en orden inverso de cierre en un DFS completo del digrafo.

```
procedure NumeraInversa(G, Q)
    Q := EmptyStack()
    for v ∈ V do
        visited[v] := false
    end for
    for v ∈ V do
        if ¬visited[v] then
```

```
            NumeraInversa-REC(G, v, Q, visited)
        end if
    end for
end procedure
procedure NumeraInversa-REC(G, v, Q, visited)
    visited[v] := true
    for w ∈ G.Successor(v) do
        if ¬visited[w] then
            NumeraInversa-REC(G, w, Q, visited)
        end if
    end for
    Q.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 $E^T = \{(u, v) \,|\, (v, u) \in E\}$. Por último se recorre el digrafo G^T pero siguiendo el orden de la lista Q. Si lanzamos un DFS desde el primer vértice de Q, que es el primero 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. Lanzando otro DFS desde el siguiente vértice no visitado en orden inverso de cierre 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)
    for v ∈ V do
        visited[v] := false
    end for
    ncc := 0
    for v ∈ Q do
        if ¬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, ∀v
end procedure
procedure Visita-SCC(G, v, ncc, SCC, visited)
    visited[v] := true
    SCC[v] := ncc
```

```
    for w ∈ G.Successor(v) do
        if ¬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.

Solución 1.16: PENDIENTE

Solución 1.17: PENDIENTE

Solución 1.19: PENDIENTE

Solución 1.20: PENDIENTE

Solución 1.21: PENDIENTE

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 recursivamemte 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 cúal. Lo que vamos a ahcer es recorrer el conjunto de las n tarjetas, de manera que ento do 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! Si el conjunto contiene un elemento mayoritario, ése será el que obtenemos al finalizar. Pero si el conjunto **no** contiene un mayoritario al terminar el bucle habremos obtenido un elemento que tiene un balance positivo al final del proceso pero que no aparecxe suficentes veces para ser mayoritario. 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 mayoritario o que no exiuste tal tarjeta.

  Este algoritmo es el famoso algoritmo MJRTY de Boyer y Moore (1980)en los subconjuntos C_1 y C_2 (artículo sobre MJRTY en la Wikipedia)

Solución 1.23: Eun 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 linearítmico. 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: PENDIENTE

Solución 1.25: PENDIENTE