# Análisis de Algoritmos
# Tarea 3

Canek Peláez Valdés

7 de Enero de 2006

1. Indique cotas superiores para las siguientes relaciones de recurrencia usando técnicas generales, pero no use el Teorema Maestro. Muestre todo su trabajo.

   *a)* $T(n) = T(\frac{n}{9}) + T(\frac{8n}{9}) + n$

   R: Usando el método de substitución, adivinamos que una cota superior para $T(n)$ es $O(n \ln n)$. Entonces suponemos que $T(n) \leq cn \ln n$ para alguna $c$. Substituyendo nuestra adivinanza en la recurrencia tenemos:

$$\begin{aligned}
T(n) &\leq c\left(\tfrac{n}{9}\right)\ln\left(\tfrac{n}{9}\right) + c\left(\tfrac{8n}{9}\right)\ln\left(\tfrac{8n}{9}\right) + n \\
&= c\left(\tfrac{n}{9}\right)(\ln n - \ln(9)) + c\left(\tfrac{8n}{9}\right)(\ln(8n) - \ln(9)) + n \\
&= \tfrac{c}{9}(n\ln n - n\ln(9) + 8n\ln(8n) - 8n\ln(9)) + n \\
&= \tfrac{c}{9}(n\ln n - n\ln(9) + 8n\ln(8) + 8n\ln n - 8n\ln(9)) + n \\
&= \tfrac{c}{9}(9n\ln n + n(8\ln(8) - 9\ln(9))) + n \\
&= \tfrac{c}{9}(9n\ln n - kn) + n \quad \text{donde } k = -(8\ln(8) - 9\ln(9)) = 4.529\ldots \\
&= cn\ln n - \tfrac{1}{9}ckn + n \\
&< cn\ln n \quad \text{si } c > \tfrac{9}{9\ln(9) - 8\ln(8)} \\
&= O(n\ln n).
\end{aligned}$$

   Esto se cumple para cualquier $c > \frac{9}{9\ln(9)-8\ln(8)} \approx 1.987$, en particular tomamos $c = 2$. Ahora sólo nos falta ver nuestro caso base. Asumimos sin perder generalidad que $T(1) = 1$, ya que aunque es posible que no sea exactamente 1, sí es un tiempo constante. De igual forma, suponemos que $T(0) = 0$.

   Entonces tenemos que

$$T(2) = T\left(\frac{2}{9}\right) + T\left(\frac{16}{9}\right) + 2 = T(0) + T(1) + 2 = 0 + 1 + 2 = 3.$$

   Y como $2\ln 2 = 2$, nuestra $c = 2$ de arriba cumple que $c2\ln 2 = c2 = 4 \geq 3$, y entonces nuestra $c$ funciona para el caso base porque se cumple que

$$T(2) \leq c2\ln 2.$$

   Por lo tanto, $T(n) = O(n \ln n)$.

   *b)* $T(n) = 2T(n-2) + n$

   R: Usando de nuevo el método de la substitución; adivinamos que una cota para $T(n)$ es $O(n^2)$. Para demostrarlo, sustituimos nuestra adivinanza en la recursión, pero de la forma $c(n-b)^2$ con $b$ fija (porque $c(n-b)^2 = O(n^2)$):

$$\begin{aligned} T(n) &\leq 2c((n-b)^2-2)+n \\ &= 2c(n^2-2bn+b^2-2)+n \\ &= 2cn^2-4bcn+2cb^2-4c+n \\ &< 2cn^2 \text{ para } c \geq \tfrac{1}{4b} \text{ y } b < \sqrt{2} \\ &= O(n^2). \end{aligned}$$

Esto se cumple si $c \geq \frac{1}{4b}$ y $b < \sqrt{2}$; en particular tomamos $b = 1 < \sqrt{2}$ y $c = 1 > \frac{1}{4}$. Ahora nos falta ver nuestro caso base. Asumimos sin perder generalidad que $T(1) = 1$ y que $T(0) = 0$. Entonces tenemos que $T(2) = 2T(2-2)+2 = 2T(0)+2 = 2$, y cumple que

$$T(2) = 2 \leq 2c(2-b)^2+2 = 2(2-1)^2+2 = 4.$$

Y por lo tanto $T(n) = O(n^2)$.

*c)* $T(n) = T(\frac{2n}{3}) + 1$

R: De nuevo, por el método de substitución, adivinamos que una cota para $T(n)$ es $\ln n$, así que asumimos que $T(n) \leq c \ln n$ para alguna $c > 0$, y lo comprobamos sustituyendo en la recurrencia nuestra adivinanza:

$$\begin{aligned} T(n) &\leq c \ln\left(\tfrac{2n}{3}\right) + 1 \\ &= c(\ln(2) + \ln n - \ln(3)) + 1 \\ &= c \ln n + c \ln(2) - c \ln(3) + 1 \\ &< c \ln n \text{ para } c \geq \tfrac{1}{\ln 3 - \ln 2} \\ &= O(\ln n). \end{aligned}$$

Esto se cumple para cualquier $c \geq \frac{1}{\ln 3 - \ln 2} \approx 1.709$; en particular tomamos $c = 2$. Ahora sólo nos falta ver nuestro caso base. Sin perder generalidad asumimos $T(1) = 1$ y $T(0) = 0$, y vemos que $T(2) = T(\frac{4}{3}) + 1 = T(1) + 1 = 2$, y se cumple que

$$T(2) = 2 \leq c \ln(2) = 2.$$

Y por lo tanto $T(n) = O(\ln n)$.

*d)* $T(n) = T(\sqrt{n}) + \ln n$

R: Usando de nuevo el método de la substitución; adivinamos que una cota para $T(n)$ es $O(\ln n)$. Asumimos que $T(n) \leq c \ln n$ para alguna $c$, y lo comprobamos reemplazando nuestra adivinanza en la recurrencia; pero lo haremos de la forma $(c-b)\ln n$ para alguna $b$ (porque $(c-b)\ln n = O(\ln n)$):

$$\begin{aligned} T(n) &\leq c \ln(\sqrt{n}) + \ln n \\ &< c \ln n - b \ln n + \ln n \\ &= c \ln n \text{ si } b = 1 \\ &= O(\ln n). \end{aligned}$$

Esto se cumple para cualquier $c$ positiva si $b = 1$; en particular tomamos $c = 4$ y $b = 1$. Ahora nos fal nuestro caso base. Sin perder generalidad asumimos que $T(1) = 1$ y $T(0) = 0$. Para $n = 2$ tenemos que $T(2) = T(\sqrt{2}) + \ln(2) = T(1) + 1 = 2$, y esto cumple que

$$T(2) = 2 \leq (c-b)\ln(2) = (4-1) = 3.$$

Y por tanto $T(n) = O(\ln n)$.

2. Encuentre una fórmula simple para $P(n) = \sum_{i=1}^{n}(2i-1)$.

   R: Vemos los primeros nueve resultados para darnos una idea de cómo debe ser la fórmula:

| $n$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| $P(n)$ | 1 | 4 | 9 | 16 | 25 | 36 | 41 | 64 | 81 |

   Inmediatamente parece que una fórmula posible es $P(n) = n^2$. Así que lo demostraremos por inducción sobre $n$:

   - Caso base: para $n = 1$, tenemos que

$$P(1) = \sum_{i=1}^{1}(2i-1) = 2 - 1 = 1 = 1^2.$$

     Por lo tanto, se comple para $n = 1$.
   - Hipótesis de inducción: suponemos que para $n \leq k$ se cumple que

$$P(k) = \sum_{i=1}^{k}(2i-1) = k^2.$$

   - Por demostrar que para $n = k + 1$ tenemos que

$$P(k+1) = \sum_{i=1}^{k+1}(2i-1) = (k+1)^2.$$

     Podemos descomponer $P(k+1)$ como sigue:

$$P(k+1) = \sum_{i=1}^{k+1}(2i-1) = \sum_{i=1}^{k}(2i-1) + (2(k+1)-1) = P(k) + (2(k+1)-1).$$

     Entonces, ya que por hipótesis de inducción tenemos que $P(k) = k^2$, se sigue que

$$P(k+1) = P(k) + (2(k+1)-1) = k^2 + (2k+2-1) = k^2 + 2k + 1 = (k+1)^2.$$

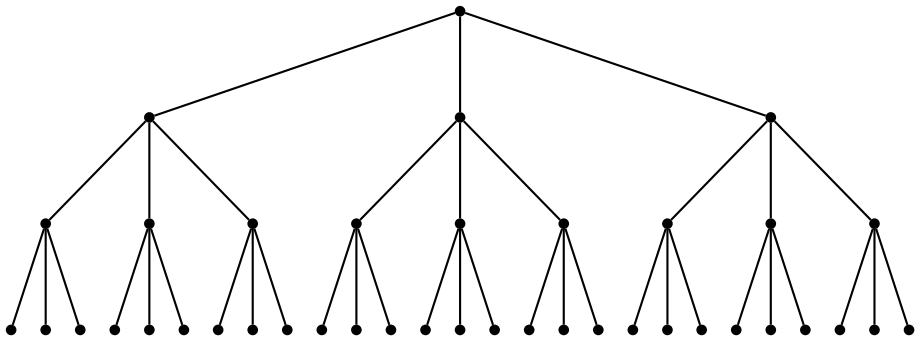
   Por lo tanto, la fórmula sencilla para $P(n)$ es

$$P(n) = n^2.$$

3. Pruebe que un árbol con raíz con al menos $n$ vértices terminales en el cual cada vértice no terminal tiene a lo más tres hijos tiene una altura de al menos $\log_3(n)$.

   R: Vamos a definir un *árbol ternario completo* de la siguiente forma: un árbol ternario completo es un árbol donde cada vértice no terminal tiene exactamente tres hijos, y donde todas las hojas están al mismo nivel.

   El árbol ternario completo de altura tres es:



Vamos a demostrar que un árbol ternario completo de altura $h$ tiene exactamente $3^h$ vértices terminales; o lo que es equivalente, que si un árbol ternario completo tiene $k$ vértices terminales, entonces su altura es exactamente $\log_3(k)$.

Por inducción sobre la altura del árbol:

- Caso base: el árbol con altura $n = 0$, el que tiene un único vértice, es un árbol ternario completo: todos sus vértices no terminales tienen tres hijos (se cumple por vacuidad), y todas sus hojas (una) están al mismo nivel.

  En el caso en que la altura es cero, tenemos que el número de vértices terminales es uno, y $3^0 = 1$, así que se cumple para el caso base.
- Hipótesis de inducción: para un árbol ternario completo de altura $n = k$, su número de vértices terminales es exactamente $3^k$.
- Por demostrar que si la altura de un árbol $T$ ternario completo es $n = k + 1$, entonces su número de vértices terminales es exactamente $3^{k+1}$.

  Como $T$ es un árbol ternario completo, todos sus vértices terminales están al mismo nivel. Si le quitamos a $T$ todos sus vértices terminales, entonces terminamos con un árbol $T'$ de altura $k$, que tiene $3^k$ vértices terminales por nuestra hipótesis de inducción.

  Pero en $T$ los vértices terminales de $T'$ *no son* terminales, y por tanto tiene cada uno de ellos tres hijos (porque $T$ es ternario completo), y esos hijos son los vértices terminales de $T$: por lo tanto, el número de vértices terminales en $T$ es

$$3 \times 3^k = 3^{k+1}.$$

Entonces todo árbol ternario completo de altura $h$ tiene $3^h$ vértices terminales. Es evidente que también funciona al revés: todo árbol ternario completo con $k$ vértices terminales tiene una altura de $\log_3(k)$.

Evidentemente a un árbol ternario completo no podemos quitarle o añadirle vértices sin que deje de serlo; la única forma es quitar todos los vértices terminales, o añadirle tres vértices terminales a cada vértice terminal.

Con estas bases, ya podemos demostrar lo que nos piden. Por inducción sobre el número $n$ de vértices terminales:

- Para $n = 1$, cuando tenemos un árbol de altura cero, cumple que cada vértice no terminal tiene a lo más tres hijos (por vacuidad). Entonces su altura es al menos $\log_3(1) = 0$, y por lo tanto cumple.
- Hipótesis de inducción: si tenemos un árbol con $n = k$ vértices terminales, en el cual cada vértice no terminal tiene a lo más tres hijos, se cumple que su altura es de al menos $\log_3(k)$.

- Por demostrar que si tenemos un árbol con $n = k + 1$ vértices terminales, en el cual cada vértice no terminal tiene a lo más tres hijos, se cumple que su altura es de al menos $\log_3(k+1)$.
  Si le quitamos al árbol un vértice terminal, entonces tendrá $k$ vértices terminales, y por hipótesis de inducción su altura será de al menos $\log_3(k)$. Lo que queremos ver es: cuándo al volver a poner el vértice terminal que le quitamos la altura se queda se igual, porque es el único caso en el que podría fallar que la altura no fuera mayor o igual a $\log_3(k+1)$. Todos los casos donde al regresar el vértice aumente la altura los descartamos, porque esos ya cumplen que su altura sea mayor o igual a $\log_3(k+1)$, ya que siempre se cumple que

  $$1 + \log_3(k) > \log_3(k+1)$$

  si $k \geq 1$.
  Supongamos entonces que al regresar el vértice que habíamos quitado, la altura permanece igual. Pero esto sólo es posible si el árbol que no tenía el vértice *no* era ternario completo; y entonces el número de vértices terminales que tenía no era de la forma $3^i$ para algún entero $i$. Pero entonces (y esto es lo importante), tenemos que

  $$\lceil \log_3(k) \rceil = \lceil \log_3(k+1) \rceil.$$

  Y entonces se cumple que la altura del árbol es de al menos $\log_3(k+1)$.

4. Sean $f(n)$ y $g(n)$ funciones asintóticamente positivas. Pruebe o contradiga lo siguiente:

   *a)* $f(n) = O(g(n))$ implica $g(n) = O(f(n))$
   R: $f(n) = O(g(n))$ quiere decir que existen $c$ una constante y $n \in \mathbb{N}$ tales que $0 \leq f(n) \leq cg(n)$ si $n \geq k$. Obviamente, $c \neq 0$.
   Supongamos que a partir de esto podemos afirmar que $g(n) = O(f(n))$. Eso significaría que existen $c'$ constante y $k' \in \mathbb{N}$ tales que $0 \leq g(n) \leq c'f(n)$ si $n \geq k'$.
   De

   $$0 \leq f(n) \leq cg(n)$$

   tenemos que (si hacemos $d = \frac{1}{c}$)

   $$0 \leq df(n) \leq g(n),$$

   y usando

   $$0 \leq g(n) \leq c'f(n)$$

   llegamos a que

   $$0 \leq df(n) \leq g(n) \leq c'f(n).$$

   Pero esto sólo ocurre si $f(n) = \Theta(g(n))$, y obviamente es falso si $f(n) = n$ y $g(n) = n^2$.

*b)* $f(n) + g(n) = \Theta(\text{mín}(f(n), g(n)))$
R: Si $f(n) + g(n) = \Theta(\text{mín}(f(n), g(n)))$, entonces existen constantes $c_1$ y $c_2$, y $k > 0$ tales que

$$c_1 \,\text{mín}(f(n), g(n)) \leq f(n) + g(n) \leq c_2 \,\text{mín}(f(n), g(n)) \quad \text{si } n \geq k.$$

Ahora, o bien $\text{mín}(f(n), g(n)) = f(n)$, o bien $\text{mín}(f(n), g(n)) = g(n)$. En el primer caso tendríamos que

$$c_1 f(n) \leq f(n) + g(n) \leq c_2 f(n) \quad \text{si } n \geq k.$$

Si damos $f(n) = n$ y $g(n) = n^2$, es obvio que no se cumple. Es análogo cuando $\text{mín}(f(n), g(n)) = g(n)$; por lo tanto, no es cierto.

*c)* $f(n) + o(f(n)) = \Theta(f(n))$
R: Si $f(n) + o(f(n)) = \Theta(f(n))$, entonces existen constantes $c_1$ y $c_2$, y $k > 0$ tales que

$$c_1 f(n) \leq f(n) + o(f(n)) \leq c_2 f(n).$$

La definición de $o$ dice:

$$o(f(n)) = \{g(n) | \forall\ c > 0,\ \exists\ k > 0 \text{ tal que } 0 \leq g(n) < cf(n)\ \forall\ n \geq k\}.$$

Sea $g(n) = o(f(n))$, cualquier función en el conjunto $o(f(n))$. Si tomamos *cualquier* $c > 0$, tenemos entonces que se cumple

$$0 \leq g(n) < cf(n)$$

a partir de alguna $k$. Si sumamos $f(n)$ en cada parte de la desigualdad tenemos

$$f(n) \leq f(n) + g(n) = f(n) + o(f(n)) < f(n) + cf(n) = (c + 1)(f(n)).$$

Así que con cualquier $c > 0$, tenemos

$$f(n) \leq f(n) + o(f(n)) < (c + 1)(f(n)),$$

y por tanto podemos tomar $c_1 = 1$ y $c_2 = c + 1$. De donde tenemos que sí es cierto que $f(n) + o(f(n)) = o(f(n))$.

5. Dado un arreglo $A$ de $n$ números, queremos contestar la siguiente pregunta: ¿hay un elemento $x$ que aparezca al menos $\frac{n}{3}$ veces en $A$? Encuentre un algoritmo de tiempo lineal que responda esta pregunta. Pruebe la correctez y tiempo de ejecución de su algoritmo.

   R: Platicamos el algoritmo antes de presentarlo formalmente. Supongamos que el arreglo $A$ está ordenado; entonces *todos* los elementos que aparezcan repetidos están juntos. Si existe un elemento que aparezca $\frac{n}{3}$ veces en $A$, entonces cubre un rango que es mayor o igual que $\frac{n}{3}$. Por lo tanto, o bien dicho elemento ocurre en $A_{[\frac{n}{3}]}$, o bien en $A_{[\frac{2n}{3}]}$, o bien en $A_{[n]}$ (tenemos que cubrir este caso si $n \neq 3k$ para alguna $k$).

   Entonces, sin necesidad de ordenar $A$, tomamos el $\frac{n}{3}$-ésimo elemento, el $\frac{2n}{3}$-ésimo elemento, y el elemento máximo, y comprobamos cada uno para ver si ocurre $\frac{n}{3}$ veces. Buscar el $k$-ésimo elemento de un arreglo como si estuviera ordenado nos toma tiempo lineal, y lo hacemos exactamente dos veces. Buscar el máximo elemento nos toma tiempo $n$. Y Después es revisar si alguno de los tres aparece $\frac{n}{3}$ veces, lo cual nos toma $3n$. Por lo tanto el algoritmo tiene tiempo $O(n)$.

   El algoritmo sería

```
1: procedure THREE-TIMES(A,n)
2:     c_1 ←KTH-ELEMENT(A,⌊n/3⌋)
3:     c_2 ←KTH-ELEMENT(A,⌊2n/3⌋)
4:     c_3 ←MAX-ELEMENT(A)
5:     count_[1] ← 0, count_[2] ← 0, count_[3] ← 0
6:     for i ← 1 to n do
7:         if A_[i] = c_1 then
8:             count_[1] ← count_[1] + 1
9:         if A_[i] = c_2 then
10:            count_[2] ← count_[2] + 1
11:        if A_[i] = c_3 then
12:            count_[3] ← count_[3] + 1
13:    if count_[1] ≥ ⌊n/3⌋ then
14:        return c_1
15:    if count_[2] ≥ ⌊n/3⌋ then
16:        return c_2
17:    if count_[3] ≥ ⌊n/3⌋ then
18:        return c_3
19:    return nil
```

Donde KTH-ELEMENT($A$,$k$) nos regresa (en tiempo lineal) el $k$-ésimo elemento de $A$ si estuvieran ordenados de mayor a menor. El algoritmo funciona porque no es posible que si un elemento aparece $\frac{n}{3}$ o más veces entonces no ocurra en el $\frac{n}{3}$-ésimo elemento, o en el $\frac{n}{3}$-ésimo elemento, o en el mayor elemento. El algoritmo es lineal porque ejecuta sólo algoritmos lineales sin anidarlos en ningún momento.

6. **(Closest sum)** Suponga que le es dada una lista de $n$ enteros $\{x_1, \ldots, x_n\}$, y otro entero $y$. El problema es encontrar dos elementos $x_i$ y $x_j$ tales que $x_i + x_j \leq y$ y tales que su suma sea lo más grande posible. De un algoritmo con tiempo $O(n \ln n)$ que resuelva este problema.

   R: Platicamos primero el problema antes de presentarlo formalmente. Ordenamos en un arreglo $A$ nuestra lista de $n$ enteros; esto nos toma $\Theta(n \ln n)$. Nuestros elementos $A_{[1]}, A_{[2]}, \ldots, A_{[n]}$ entonces están en orden de menor a mayor. Ya ordenados los elementos, tomamos $A_{[1]}$ y lo sumamos con $A_{[\frac{n}{2}]}$; si el resultado es menor que $y$, entonces sumamos $A_{[1]}$ con $A_{[\frac{3n}{4}]}$; si es mayor, con $A_{[\frac{n}{4}]}$. Lo que estamos haciendo es una búsqueda binaria del elemento de $A$ que, sumado con $A_{[1]}$, esté más cerca de $y$.

   Esto nos toma $\Theta(\ln n)$ (porque como lo haremos será buscar en subarreglos de $A$ hasta llegar a un subarreglo de tamaño 1), y tenemos que hacerlo en todos los elementos de $A$, así que en total nos llevará $\Theta(n \ln n)$, que no nos empeora el $\Theta(n \ln n)$ que tardamos en ordenar el arreglo.

```
1: procedure CLOSEST-SUM(L,y)
2:     n ← LENGTH(L)
3:     A ← TOARRAY(L)
4:     QUICKSORT(A,n)
5:     diff ← ∞
6:     x_i ← −1
7:     x_j ← −1
8:     for i ← 1 to n do
9:         t_i ← A_[i]
10:        j ← CLOSEST(A, 0, n, t_i, y)
11:        t_j ← A_[j]
12:        if y − t_i + t_j ≥ 0 and y − t_i + t_j < diff then
13:            x_i ← t_i
14:            x_j ← t_j
15:            diff ← y − x_i + x_j
16:    return y, x_i y x_j
```

```
1: procedure CLOSEST(A,ini,end,x_i,y)
2:     if ini = end then
3:         return ini
4:     middle ← ini + (end−ini)/2
5:     x_j ← A_[middle]
6:     if x_i + x_j < y then
7:         return CLOSEST(A, middle + 1, end, x_i, y)
8:     else
9:         return CLOSEST(A, ini, middle, x_i, y)
```

El algoritmo puede optimizarse en algunas partes, pero esta versión cumple que es $\Theta(n \ln n)$.

7. *a*) Se le da un conjunto $S$ de $n$ números reales. Quiere preprocesar esta entrada para que pueda contestar consultas del tipo: ¿cuántos elementos en $S$ caen en el rango $[a, \ldots, b]$? (donde $a$ y $b$ son números reales), tan rápido como sea posible. De un preprocesado y un algoritmo para contestar las consultas. Justifique la correctez y el tiempo de ejecución de sus algoritmos.

   R: El preprocesado es simple; hay que ordenar los elementos de $S$ en un arreglo $A$. Esto nos toma $\Theta(n \ln n)$.

   El algoritmo de consulta es bastante sencillo; hacemos una búsqueda binaria en $A$ del elemento $A_{[i]}$ más cercano a $a$ que le sea mayor o igual, y una búsqueda binaria del elemento $A_{[j]}$ más cercano a $b$ que le sea menor o igual. El que los elementos de $A$ sean reales no importa; la búsqueda binaria sirve ahí. La respuesta a la consulta es $j - i + 1$.

   Cada una de las dos búsquedas binarias nos toma $\Theta(\ln n)$, así que el algoritmo de consulta es también $\Theta(\ln n)$.

   El algoritmo es trivialmente correcto; el arreglo $A$ está en orden, así que todos los elementos a partir de $A_{[i]}$ son mayores que $a$ (con $A_{[i]}$ con la posibilidad de que sea igual), y todos los anteriores son menores. De igual forma todos los elementos antes de $A_{[j]}$ (incluyéndolo) son menores que $b$ (con $A_{[j]}$ con la posibilidad de que sea igual), y todos los que le siguen son mayores. Así que *todos* los elementos de $A$ (y por tanto de $S$) ente $i$ y $j$ (inclusive) están en el rango $[a, \ldots, b]$, y el total de elementos en el rango es justamente $j - i + 1$.

   *b*) Ahora suponga que le es dado un conjunto $S$ que contiene $n$ enteros en el rango $[1, \ldots, k]$. De nuevo quiere preprocesar $S$ para responder la misma pregunta de antes ($a$ y $b$ pueden ser todavía números reales). ¿Podemos hacerlo mejor en este caso? De nuevo de algoritmos para preprocesar y para contestar las consultas, justificando su correctez y tiempo de ejecución.

   R: Podemos mejorarlo bastante si los elementos son enteros. Ahora además de ordenar $A$, creamos un arreglo $R$ de tamaño $k$, donde la entrada $R_{[i]}$ será el número de elementos que hay en el rango $[1, A_{[i]}]$. Llenar el arreglo $R$ nos toma tiempo $k$, utilizando el siguiente algoritmo (con $A$ ya ordenado)

```
1: procedure GET-R(A,n,k)
2:     if A_[1] = 1 then
3:         R_[1] ← 1
4:     else
5:         R_[1] ← 0
6:     last ← A_[1]
7:     for i ← 2 to n do
8:         for j ← last + 1 to A_[i] − 1 do
9:             R_[j] ← R_[j−1]
10:        R_[A_[i]] ← R_[A_[i]−1] + 1
11:    return R
```

Cada $R_{[i]}$ es asignado exactamente una vez, con $1 \leq i \leq k$; por lo tanto su tiempo de ejecución es $O(k)$.

El algoritmo de consulta entonces es sencillamente calcular $R_{[\lfloor b \rfloor]} - R_{[\lceil a \rceil]}$ (el número de elementos en $A$ hasta el entero más grande menor o igual que $b$, menos el número de elementos hasta el entero más pequeño mayor o igual que $a$).

Por lo tanto, con un preprocesado de tiempo $O(n \ln n + k)$, obtenemos un tiempo de consulta de $O(1)$. Evidentemente, esto es imposible de hacer si los elementos de $A$ son reales.

8. Un país tiene sólo tres distintos tipos de monedas: 1 centavo, 5 centavos y 6 centavos. Queremos saber cuál es el mínimo número de monedas que podemos usar si queremos pagar por algo que cuesta $n$ centavos. De un algoritmo eficiente que tome un entero $n$ como entrada y nos regrese el conjunto mínimo de monedas que tenga el valor de $n$ exactamente. Analice los requerimientos de tiempo y espacio para su algoritmo. Pruebe la correctez de su algoritmo.

   R: Platicamos el algoritmo antes de presentarlo formalmente. Vamos utilizar un arreglo $C_{[1,\ldots,n]}$, en el cual la entrada $C_{[i]}$ nos dirá el mínimo de monedas que necesitamos para tener $i$ centavos.

   La idea es la siguiente; evidentemente $C_{[1]} = 1$. Ahora supongamos que queremos saber el valor para $C_{[i]}$ teniendo $C_{[j]}$ para $j < i$. Para $C_{[i]}$ necesitamos más centavos que para cualquiera de las anteriores entradas, pero queremos agregar el menor número posible de monedas; así que intentamos todas las posibilidades. En general, si para $C_{[i]}$ agregamos una moneda de $k$ centavos, entonces ahora necesitamos saber el mínimo número de monedas que necesitamos para juntar $i - k$ centavos; pero esa solución la tenemos en $C_{[i-k]}$.

   Entonces la fórmula para $C_{[i]}$ está dada por

$$C_{[i]} = 1 + \text{mín}\left\{C_{[i-6]}, C_{[i-5]}, C_{[i-1]}\right\}.$$

   Ahora, nos piden el conjunto de monedas, no sólo cuántas son; entonces llevaremos otro arreglo (que llamaremos $D$), en donde pondremos en la entrada $D_{[i]}$ qué moneda agregamos para juntar $i$ centavos. El algoritmo quedaría de la siguiente forma:

```
1: procedure CALCULATE-CHANGE(n)
2:     M[1] ← 1, M[2] ← 5, M[3] ← 6
3:     for i ← 1 to n do
4:         C[i] ← ∞
5:         for j ← 1 to 3 do
6:             if i ≥ M[j] and 1 + C[i−M[j]] < C[i] then
7:                 C[i] ← 1 + C[i−M[j]]
8:                 D[i] ← M[j]
9:     return C y D
```

   Ya con $C$ y $D$, podemos imprimir el conjunto de monedas de la siguiente forma: si $D_{[n]}$ es $k$, entonces utilizamos una moneda de $k$ centavos, y sacamos el resto a partir de $D_{[n-k]}$.

```
1: procedure PRINT-CHANGE(D,n)
2:     while n > 0 do
3:         print(D[n])
4:         n ← n − D[n]
```

   El algoritmo es obviamente $O(n)$ (porque el **for** anidado ejecuta un número constante de ciclos).

9. Nos dan un conjunto de ciudades $c_1, c_2, \ldots, c_n$, y una tabla $D_{[1,\ldots,n;1,\ldots,n]}$ tal que $D_{[i,j]}$ es la longitud de la carretera de $c_i$ a $c_j$ (este valor puede ser $\infty$ si no existe una carretera entre las dos ciudades). De un algoritmo eficiente que calcule la ruta más corta posible de la ciudad $c_1$ a $c_n$ que no pase más de $k$ ciudades. Justifique la correctez de su algoritmo y su tiempo de ejecución.

   R: Platicamos el algoritmo antes de presentarlo formalmente. Usaremos programación dinámica por la siguiente razón; si sabemos la ruta más corta que no pase por más de $k-1$ ciudades entre $c_1$ y cualquier otra ciudad, entonces la ruta más corta que no pase por más de $k$ ciudades entre $c_1$ y $c_n$ está dada por el mínimo de la distancia entre $c_1$ y alguna $c_i$, más la distancia entre $c_i$ y $c_j$.

   En un arreglo $C$ guardaremos en $C_{[i,j]}$ la distancia más corta entre $c_1$ y $c_j$ que no pase por más de $i$ otras ciudades. Evidentemente, $C_{[0,i]} = D_{[1,i]}$ para $1 \leq i \leq n$, porque para ir de $c_1$ a $c_i$ pasando por otras cero ciudades, pues es sólo ir de $c_1$ a $c_i$. Para calcular $C_{[i,j]}$ haremos

$$C_{[i,j]} = \text{mín}\{C_{[i-1,k]} + D_{[k,j]} | k = 1, \ldots, n\}.$$

   La longitud de la ruta más corta entre $c_1$ y $c_n$ que no pase por más de $k$ otras ciudades quedaría entonces en la entrada $C_{[k,n]}$.

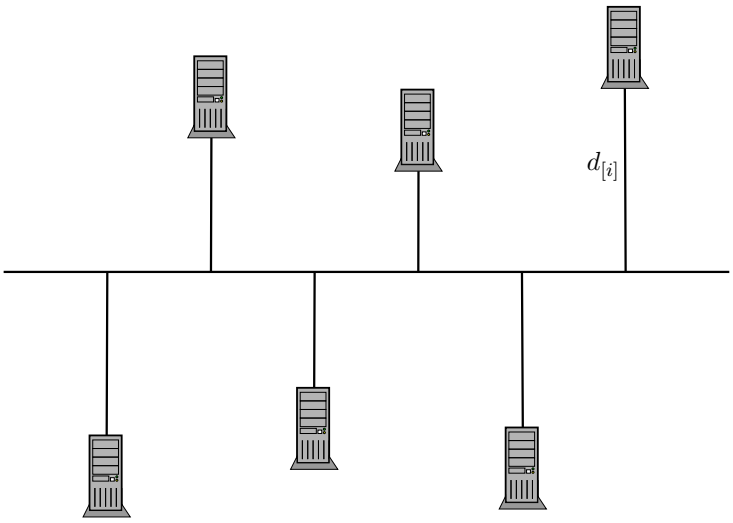
   Ahora, el problema pide que demos la ruta, no sólo su longitud, así que necesitamos guardar también los predecesores (de qué ciudad venimos). Esto lo haremos en un arreglo $P$, donde $P_{[i,j]}$ será la ciudad que tomamos para llegar a $c_j$ pasando por $i$ otras ciudades.

   El algoritmo quedaría como sigue:

```
1:  procedure CITY-ROUTE(D,k)
2:      for i ← 1 to n do
3:          C[0,i] ← D[1,i]
4:      for i ← 1 to k do
5:          for j ← 1 to n do
6:              max ← 0
7:              pred ← −1
8:              for l ← 1 to n do
9:                  if C[i−1,l] + D[l,j] > max then
10:                     max ← C[i−1,l] + D[l,j]
11:                     pred ← l
12:             C[i,j] ← max
13:             P[i,j] ← pred
14:     return C y P
```

   El algoritmo es trivialmente $O(kn^2)$.

10. El profesor Preparata es consultor para una compañía de tecnología que necesita poner una red para conectar un conjunto de $n$ máquinas. Un cable principal se tenderá de este a oeste a lo largo de una paralela, y cada máquina estará conectada al cable principal a través del camino más corto (ya sea al norte o sur de un meridiano), como se muestra en la figura. Dadas las coordenadas $x$ y $y$, ¿cómo debe el profesor escoger la ordinal óptima de el cable principal (la que minimice el total de cable usado para conectar las máquinas al cable principal, esto es la suma $\sum_{i=1}^{n} d_{[i]}$)? Pruebe que la localización óptima puede determinarse en tiempo lineal.




R: Dado que el cable principal se tiende de este a oeste a lo largo de una paralela, esto nos restringe bastante el problema. La posición óptima del cable tiene que ser de tal forma que deje $\frac{n}{2}$ máquinas al norte del cable, y $\frac{n}{2}$ máquinas al sur del cable, con una posible máquina directamente sobre el cable, si $n$ es impar.

Si el cable no parte en dos conjuntos del mismo tamaño a las máquinas, entonces siempre podemos reducir la cantidad de cable usado subiéndolo (si es que el número de máquinas al sur es menor al número de máquinas al norte), o bien bajándolo (si ocurre lo contrario). Esto sólo deja de ocurrir cuando el cable parte en dos conjuntos del mismo tamaño a las $n$ máquinas, con a lo más una quedando justo sobre el cable si $n$ es impar.

Entonces nuestro algoritmo sería: si $n$ es par, buscar el $\frac{n}{2}$-ésimo elemento si los ordenáramos por la coordenada $y$ de sur a norte, y tender el cable justo al norte de él, asegurándonos de que ningún otro elemento esté entre el cable y el $\frac{n}{2}$-ésimo elemento. Si $n$ es impar, es buscar el $(\frac{n}{2}+1)$-ésimo elemento, y tender el cable sobre él mismo.

Buscar el $k$-ésimo elemento nos toma tiempo lineal, y en el caso cuando $n$ es par, para asegurarnos de que no haya ninguna otra máquina entre el cable y el $\frac{n}{2}$-ésimo elemento, lo más sencillo es buscar también el $(\frac{n}{2}+1)$-ésimo elemento, y tender el cable justo entre ellos. En total, el tiempo que nos toma el algoritmo es lineal.

11. Suponga que tiene un arreglo bidimensional $A_{[1,\ldots,n;1,\ldots,k]}$ donde $k = \Theta(\ln n)$. Cada renglón representa un entero en el rango $[0, n]$, escrito en binario. Todos los enteros en $[0, n]$ están representados en $A$ excepto uno. Asumiendo que la única operación que puede hacer en $A$ es el leer un bit en tiempo constante, diseñe un algoritmo que en tiempo $\Theta(n)$ encuentre el elemento perdido (en su solución tenga en mente que copiar un renglón cuesta $\Theta(\ln n)$ en tiempo).

R: Platicamos el algoritmo antes de presentarlo formalmente. Antes que nada, necesitamos que $n$ sea potencia de 2. Si no lo es, se complica muchísimo la búsqueda; y además podemos agregar renglones a $A$ para que en total haya $n = 2^k$ elementos relativamente rápido ($O(n \ln n)$). Así que asumiremos que $n = 2^k$ para la $k$ dada al problema.

Vamos a recorrer la columna $k$ de $A$, o sea que recorreremos $A_{[1,k]}, \ldots, A_{[n,k]}$. Llevaremos dos contadores $p$ y $q$, y dos listas $B_1$ y $B_2$, y si $A_{[i,k]}$ es cero, incrementaremos $p$ en uno y agregaremos $A_{[i,k]}$ a $B_1$; si es uno, incrementaremos $q$ en uno y agregaremos $A_{[i,k]}$ a $B_2$. Al final tendremos o bien que $q < p$, o que $p = q$; no hay de otra, porque $n = 2^k$, o sea que es par, y entonces en el rango $[0, \ldots, n]$ hay un par más que impares. Entonces si falta un par, habrá igual número de pares que de impares; y si falta un impar, habrá menos impares que pares.

Entonces ya sabiendo que nos falta un par o un impar, vamos a repetir el proceso, pero en la $k-1$-ésima columa, y recorriendo sólo los elementos de la lista correspondiente ($B_1$ o $B_2$).

El tamaño de las lista $B_1$ y $B_2$ se reduce a la mitad en cada ciclo; cuando llegue a tamaño uno, tendremos en la lista correspondiente el índice de un número de $A$ que tiene todos los bits iguales a nuestro número perdido, excepto el primero (por esto es que necesitamos que $n = 2^k$; si son menos el "hermano" que nos da el número perdido podría no estar). Entonces le cambiamos el primer bit a este elemento, y tendremos nuestro número perdido.

El algoritmo sería

```
1:  procedure GET-MISSING(A,n,k)
2:      for i ← 1 to n do
3:          L ← L ∪ {i}
4:      for j ← k down to 1 do
5:          p ← 0, q ← 0
6:          B1 ← ∅, B2 ← ∅
7:          for all i in L do
8:              if A[i,j] = 0 then
9:                  p ← p + 1
10:                 B1 ← B1 ∪ {A[i,j]}
11:             else
12:                 q ← q + 1
13:                 B2 ← B2 ∪ {A[i,j]}
14:         if p ≤ q then
15:             L ← B1
16:         else
17:             L ← B2
18:     i ←GETFIRST(L)
19:     r ← A[i]                    ▷ en r guardamos todo el renglón i
20:     if r[0] = 0 then
21:         r[0] ← 1
22:     else
23:         r[0] ← 0
24:     return r
```

La complejidad del algoritmo es $O(n)$. El primer **for** toma tiempo $n$, y el siguiente **for** se ejecuta de la siguiente manera: la primera vez, dentro de él se ejecuta un **for** anidado que se ejecuta $n$ veces; la segunda vez, el **for** anidado se ejecuta $\frac{n}{2}$ veces, la tercera vez $\frac{n}{4}$ veces. En general, para la $k$-ésima ejecución, se ejecuta $\frac{n}{2^k}$ veces. Pero $n = 2^k$, entonces $\frac{n}{2^k} = \frac{2^k}{2^k} = 1$.

En total, el número de operaciones que hacemos es

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots = 2n.$$

12. Un ladrón está en un cuarto que contiene $n$ elementos. Cada elemento tiene un precio $p$ y un peso $w$ asociado a él. El ladrón puede cargar a lo más $b$ kilos (y puede o bien tomar o dejar un elemento, no es posible el tomar una fracción de él). El ladrón quiere escoger un subconjunto de estos elementos para que el peso de todos sea menor o igual a $b$ y que la ganancia esté maximizada.

    *a)* De un algoritmo eficiente que, dado un arreglo $W_{[1,\ldots,n]}$ con los pesos de los elementos, un arreglo $P_{[1,\ldots,n]}$ con los precios y la cota $b$, de como salida la ganancia máxima posible. Justifique la correctez de su algoritmo y su tiempo de ejecución.

    R: Explicamos el algoritmo antes de presentarlo formalmente. Utilizaremos programación dinámica, porque el problema tiene subestructura óptima (si sabemos la solución óptima

cuando el peso máximo es $i$, para $i < b$, entonces podemos calcular más fácilmente la solución cuando el peso es $b$).

Vamos a utilizar una estrategia parecida a la del ejercicio 9 (el de las ciudades); tendremos una matriz $C_{[0,\ldots,n,0,\ldots,b]}$ de tal forma que en la entrada $C_{[i,j]}$ estará la suma máxima de los precios de un subconjunto de los elementos $1,\ldots,i$ cuyos pesos no sean mayores a $j$. De esta forma, la ganancia máxima estará guardada en la entrada $C_{[n,b]}$.

El primer renglón de nuestro arreglo lo inicializamos en cero, ya que nos representa el subconjunto vacío de nuestros elementos:

$$C_{[0,i]} = 0 \quad \text{con } 0 \leq i \leq b.$$

De igual forma, nuestra primera columa tendrá en ceros sus elementos, ya que si nuestro límite de carga es cero kilos, no podremos llevar nada (a menos que haya un elemento que pese cero kilos; pero asumimos que éste no es el caso).

La fórmula para calcular las siguientes entradas es la siguiente:

$$C_{[i,j]} = \text{máx}(C_{[i-1,j]}, P_{[i]} + C_{[i-1,j-W_{[i]}]}).$$

Al calcular $C_{[i,j]}$ sólo tenemos dos opciones; añadir el $i$-ésimo elemento a nuestro subconjunto, o dejarlo. En el primer caso, significa que no podemos mejorar lo que teníamos con los elementos $1,\ldots,i-1$, y la solución a ese problema la tenemos en la entrada $C_{[i-1,j]}$ de nuestro arreglo.

En el segundo caso, que sólo puede ocurrir si $W_{[i]} \leq j$ (porque no podemos añadir el $i$-ésimo elemento si su peso es mayor a nuestro límite, que es $j$), tenemos que añadimos $P_{[i]}$ a nuestra ganancia máxima; pero gastamos $W_{[i]}$ kilos de nuestro límite. Entonces la mejor solución que podemos obtener con los otros $1,\ldots,i-1$ elementos y capacidad $j - W_{[i]}$ (porque estamos añadiendo $W_{[i]}$), la tenemos calculada en $C_{[i-1,j-W_{[i]}]}$. Si lo añadimos entonces nuestra ganancia es $P_{[i]} + C_{[i-1,j-W_{[i]}]}$.

El algoritmo sería:

```
1:  procedure THIEF-PROFIT(P,W,b,n)
2:      for i ← 0 to b do
3:          C[0,i] ← 0
4:      for i ← 1 to n do
5:          C[i,0] ← 0
6:          for j ← 1 to b do
7:              if W[i] ≤ j then
8:                  C[i,j] ← máx(C[i-1,j], P[i] + C[i-1,j-W[i]])
9:              else
10:                 C[i,j] ← C[i-1,j]
11:     return C[n,b]
```

Es obvio que el tiempo de ejecución del algoritmo es $O(nb)$.

*b*) Describa brevemente qué cambios le podría hacer a su algoritmo si además quisiera poder calcular cuáles de los elementos el ladrón debería tomar.

R: Necesitaríamos otro arreglo del mismo tamaño a $C$ (llamémosle $E$), donde pondríamos en la entrada $E_{[i,j]}$ un uno si decidimos añadir el $i$-ésimo elemento en $C_{[i,j]}$, y cero de otra forma.

Al finalizar el algoritmo, si $E_{[i,j]}$ es uno, entonces buscamos el anterior elemento añadido a partir de $E_{[i-1,j-W_{[i]}]}$. Podríamos imprimir los elementos con el siguiente algoritmo:

```
1: procedure PRINT-ITEMS(E,W,b,n)
2:     j ← b
3:     for i ← n down to 1 do
4:         if E_[i,j] = 0 then
5:             print(i)
6:             j ← j − W_[i]
```

13. Sea $G = (V, E)$ una gráfica conexa y dirigida con función de peso $w : E \to \mathbb{R}$, y suponga que $|E| \geq |V|$ y que todos los pesos de las aristas son distintos. Un segundo mejor árbol generador de peso mínimo $T_1$ se define como sigue: sea $T$ el conjunto de todos los árboles generadores de $G$, y sea $T_0$ un árbol generador de peso mínimo de $G$. Entonces un segundo mejor árbol generador de peso mínimo es un árbol generador $T_1$ tal que su peso es mayor que el peso de $T_0$, y cualquier otro árbol generador de $G$ tiene peso mayor o igual que el de $T_1$.

*a*) Muestre que el árbol generador de peso mínimo es único, pero que el segúndo mejor árbol generador de peso mínimo no lo es.

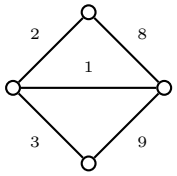
R: Sea $T$ el árbol generador de peso mínimo de $G$, y supongamos que existe $T'$ un árbol generador de $G$ tal que $w(T') = w(T)$, y que $T' \neq T$.

Sea $e' \in E(T')$ la arista de peso más pequeño tal que $e' \notin E(T)$; todas las aristas de peso menor a $e'$ en $T'$ son compartidas con $T$. Agregamos $e'$ a $T$ y se nos genera un ciclo porque es un árbol. En ese ciclo obviamente hay una arista $e \in E(T)$ que no está en $E(T')$ (si todas estuvieran en $T'$ éste tendría un ciclo).

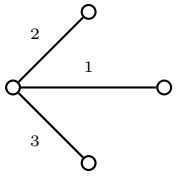
Supongamos que $w(e) < w(e')$. En ese caso, agregamos $e$ a $T'$, se nos genera un ciclo, y en ese ciclo todas las aristas que no están en $E(T)$ (que tiene que haber al menos una) son de peso mayor que $e$ (porque $e'$ era la de peso menor de $E(T')$ tal que no estuviera en $E(T)$, y $w(e) < w(e')$); entonces podríamos quitar cualquiera de esas aristas y dejar $e$, y generaríamos un árbol de peso menor a $w(T')$, pero esto contradiría que $T'$ es mínimo. Por lo tanto $w(e) > w(e')$. Pero entonces en $T$ quitamos $e$, ponemos $e'$, y generamos un árbol de peso menor a $T$; pero esto contradice que $T$ sea mínimo.

Por lo tanto no existe $T'$ distinto a $T$ con igual peso.

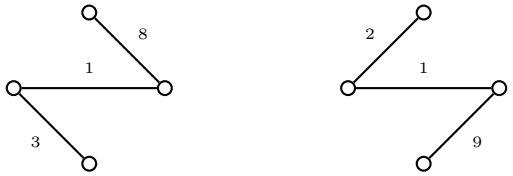
Para mostrar que el segundo mejor árbol generador de peso mínimo no es necesariamente único, presentamos la siguiente gráfica:




El siguiente es su árbol generador de peso mínimo, que tiene peso 6:



Y los siguientes dos árboles de peso 12 son dos segundos mejores árboles generadores de peso mínimo:




*b*) Sea $T_0$ un árbol generador de peso mínimo de $G$. Pruebe que existen aristas $(u, v) \in T_0$ y $(x, y) \in T$ tales que $T_0 - \{(u, v)\} + \{(x, y)\}$ es un segundo mejor árbol generador de peso mínimo de $G$.

R: Sea $T_0$ el árbol generador de peso mínimo de $G$, y supongamos que un segundo mejor árbol generador de peso mínimo $T_1$ difiere de $T_0$ en dos aristas; sean dichas aristas $u$ y $v$ en $T_1$. Si añadimos $u$ y $v$ a $T_0$ se nos generan dos ciclos, y en cada uno de sus ciclos $u$ y $v$ son las aristas más pesadas (si no lo fueran, las reemplazaríamos en $T_0$ con la arista más pesada y obtendríamos un árbol de peso menor a $T_0$, lo que no es posible porque $T_0$ es de peso mínimo).

Sea $T'$ el árbol que obtenemos de añadir $u$ a $T_0$, y de quitarle la segunda arista más pesada en el ciclo que se forma. El peso de $T'$ es mayor, porque tiene una arista más pesada ($u$) de la que le quitamos; pero además tiene peso menor a $T_1$, porque la arista $v$ de $T_1$ es más pesada que cualquier otra arista del ciclo que se formaría al agregar $v$ a $T'$ (que en ese ciclo es idéntico a $T_0$). Por lo tanto construimos un árbol $T'$ que cumple que

$$w(T_0) < w(T') < w(T_1).$$

Pero esto contradice que $T_1$ sea un segundo mejor árbol generador de peso mínimo; por lo tanto, no es posible que $T_1$ difiera en dos aristas de $T_0$. Es evidente que podemos volver a hacer lo mismo si suponemos que difiere en más de dos aristas (porque todas son mayores que cero); así que $T_0$ y $T_1$ difieren en una única arista.

Por lo tanto, existen aristas $(u, v) \in T_0$ y $(x, y) \in T$ tales que $T_0 - \{(u, v)\} + \{(x, y)\}$ es un segundo mejor árbol generador de peso mínimo.

*c*) Sea $T$ un árbol generador de $G$, y por cada dos vértices $u, v \in V$, sea $max_{[u,v]}$ una arista con peso máximo en el único camino entre $u$ y $v$ en $T$. Describa un algoritmo de tiempo $O(|V|^2)$ que, dada $T$, calcule $max_{[u,v]}$ para toda $u, v \in V$.

R: Platicaremos el algoritmo antes de presentarlo. La idea es la siguiente; vamos a tomar un vértices $v$ de $T$, y tratándolo como si fuera una raíz, correremos algo parecido a BFS en $T$, sólo que en el recorrido iremos guardando la arista de máximo peso entre $v$ y todos los demás vértices de $T$. Esto lo vamos a repetir para todo vértice en $T$.

El algoritmo sería:

```
1: procedure GET-MAX(T)
2:     for all u in V[T] do
3:         for all v in V[T] do
4:             max_[u,v] ← nil
5:     for all v in V[T] do
6:         SET-MAX(v,v,max)
7:     return max
```

```
1: procedure SET-MAX(u,v,max)
2:     for all x in Adj[v] do
3:         if w(v,x) > w(max_[u,x]) then
4:             max_[u,v] ← (v,x)
5:         SET-MAX(u,x,max)
```

El algoritmo es $O(|V|^2)$, pero tenemos que tener cuidado con nuestras estructuras de datos, porque si no se dispara la complejidad. Pero si podemos obtener en tiempo constante cada una de las adyacencias de un vértice, y una adyacencia dados dos vértices (que sí se puede, porque $T$ es un árbol), entonces el algoritmo corre trivialmente en $O(|V|^2)$. Cada llamada inicial a SET-MAX es un BFS con el primer parámetro como raíz del árbol. Esto toma tiempo $O(|V|)$ (una arista es recorrida una única vez, y hay $|V| - 1$ aristas), y se manda a llamar una única vez con $v$ como raíz para cada $v \in V$.

*d*) De un algoritmo eficiente que calcule un segundo mejor árbol generador de peso mínimo en $G$.

R: Platicamos el algoritmo antes de mostrarlo. Vamos a asumir que nos pasan el árbol generador de peso mínimo de $G$ como parámetro. Después, vamos a utilizar nuestro algoritmo del ejercicio de arriba. La idea es que, para cada arista $(u, v)$ en $E - E[T]$, vamos a añadirla a $T$; después buscaremos la arista de peso máximo entre $u$ y $v$ en $T$ (que está guardada en la tabla $max$ que construimos en el ejercicio anterior), y vamos a quitarla. Al árbol resultante le calcularemos el peso, y repetiremos el proceso con las demás aristas en $E - E[T]$. Después, tomaremos la que nos diera el árbol con peso menor, y ése será nuestro segundo mejor árbol generador de peso mínimo.

El algoritmo sería:

```
1: procedure SECOND-BEST-MST(G,T)
2:     max ←GET-MAX(T)
3:     min ← w(T)
4:     sb_w ← ∞
5:     sb_x ← nil
6:     sb_y ← nil
7:     for all (x,y) in E − E[T] do
8:         (u,v) ← max_[x,y]
9:         if min − w(x,y) + w(u,v) < sb_w then
10:            sb_w ← min − w(x,y) + w(u,v)
11:            sb_x ← x
12:            sb_y ← y
13:    return T − (sb_x, sb_y) + max_[sb_x,sb_y]
```

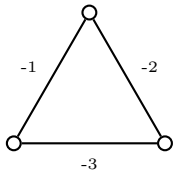
El algoritmo funciona por lo dicho en los anteriores tres incisos del ejercicio. Su complejidad es $O(|E|)$, ya que dentro del **for** se ejecutan sólo operaciones constantes (si las estructuras de datos son hechas con cuidado).

14. Muestre que si los pesos de todas las aristas de una gráfica son positivos, entonces cualquier subconjunto de aristas que conectan todos los vértices y tienen un peso total mínimo debe ser un árbol. De un ejemplo para mostrar que la misma conclusión no es verdadera si hay pesos negativos.

R: La primera parte es trivial; supongamos que el subconjunto de aristas no forma un árbol. Entonces tiene un ciclo, y como todas las aristas de este ciclo tienen peso positivo, si quitamos alguna su peso será menor. Pero se suponía era de peso mínimo el subconjunto, lo que es una contradicción. Por tanto el subconjunto forma un árbol.

La segunda parte es igual de trivial: cualquier gráfica completa con más de 2 vértices que tenga

*todas* las aristas con peso negativo, el subconjunto de aristas con peso mínimo total es *todas* las aristas. Un ejemplo sencillo es



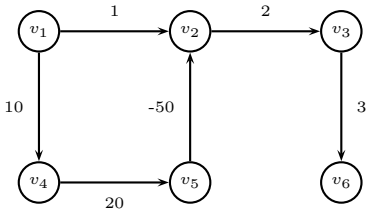

15. Sea $T$ un árbol generador de peso mínimo de una gráfica $G(V, E)$, y sea $V'$ un subconjunto de $V$. Muestre que si la subgráfica $T'$ generada de $T$ inducida por $V'$ es un árbol, entonces es un árbol generador de peso mínimo para $G'$ inducida en $G$ por $V'$.

    R: Si $T'$ es un árbol, supongamos que no es generador de peso mínimo de $G'(V', E')$. Entonces existe $T'_0$ árbol generador de peso mínimo para $G'(V', E')$, y $T' \neq T'_0$. Más aún, $w(T'_0) < w(T')$.

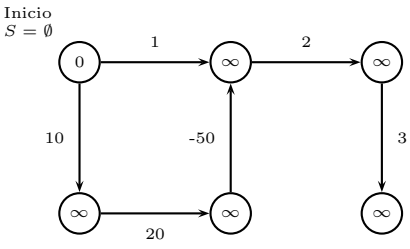
    Pero entonces tomemos a $T_0 = T - E(T') + E(T'_0)$. Obviamente $T_0$ es generador (porque $T'_0$ genera a $G'$ y sólo cambiamos aristas de $T$ que estuvieran en $G'$), y como $w(T'_0) < w(T')$, entonces $w(T_0) < w(T)$. Pero esto contradice que $T$ era de peso mínimo; por lo tanto, $T'$ es un árbol generador de peso mínimo para $G'$.

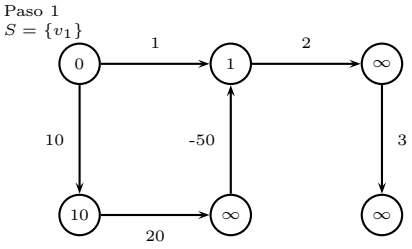
16. De un contraejemplo sencillo de una gráfica dirigida con pesos negativos en las aristas para el cual el algoritmo de Dijkstra produzca respuestas erróneas. ¿Por qué la prueba del Teorema 25.10 del Cormen falla cuando son permitidas aristas de peso negativo?

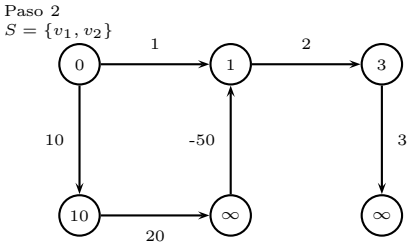
    R: Sea $G$ la siguiente gráfica:

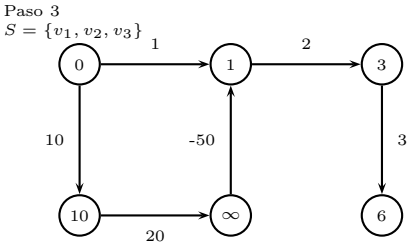



    Hacemos notar que $G$ no tiene ciclos, mucho menos ciclos negativos. Vamos a correr Dijkstra con $v_1$ como vértice origen, y en cada paso vamos a dibujar $d_{[v_i]}$ dentro de cada vértice $v_i$.

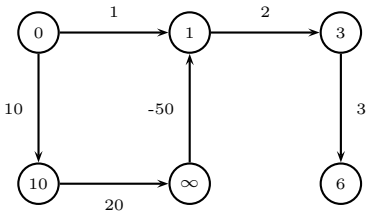


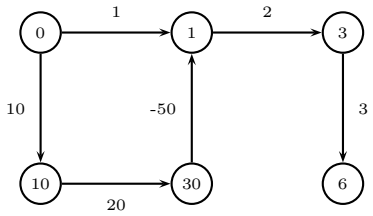



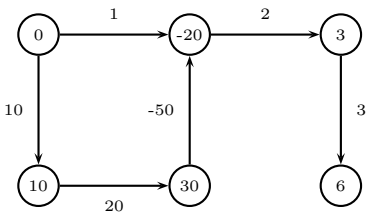
Paso 4
$S = \{s, v_1, v_2, v_3, v_6\}$




Paso 5
$S = \{s, v_1, v_2, v_3, v_6, v_4\}$




Paso 6
$S = \{s, v_1, v_2, v_3, v_6, v_4, v_5\}$




El algoritmo en este caso calcula erróneamente las distancias de $v_3$ y $v_6$ con respecto a $v_1$. Intuitivamente el algoritmo falla porque cuando un vértice se agrega a $S$, sólo se actualizan los vértices adyacentes a él; no se propagan los cambios a los adyacentes de los adyacentes.

El algoritmo *necesita* entonces asumir que cada vez que se inserta un nuevo vértice a $S$, éste viene de un camino donde todas las aristas tienen peso mayor o igual a cero. De otra forma, ocurre lo que pasó arriba: el vértice $v_4$ se descartó para ser agregado a $S$ porque al tener peso 10 la arista $(v_1, v_4)$, el algoritmo asume que el mejor camino no va por ahí porque hay otras aristas con menor peso. Si hay aristas negativas, un camino puede "mejorar"; pero el algoritmo de Dijkstra no puede predecir esto (aumentaría mucho la complejidad del mismo), y por esa razón regresa respuestas erróneas.

Formalmente, la demostración 25.10 del Cormen falla porque asume que $\delta(s, y) \leq \delta(s, u)$ en la ecuación 25.2 de la demostración, y esto no es necesariamente cierto si hay aristas con peso negativo (como acabamos de ver).

17. Para un árbol $T$ con raíz con $n$ vértices, donde cada vértice $v$ tiene un peso $w(v)$, de un algoritmo lineal de programación dinámica que encuentre el subconjunto independiente $S$ de $T$ de peso máximo.

R: Platicamos el algoritmo antes de presentarlo formalmente. La idea es tener un subconjunto $M$ de los vértices de $T$ con el máximo peso posible, y de tal forma que si $u, v \in M$, entonces $(u, v) \notin E$. Vamos a usar programación dinámica en el sentido de que iremos guardando las soluciones a subproblemas que ya hayamos calculado, y utilizaremos la mejor al momento de resolver la solución general.

El hecho de que sea un árbol con raíz $r$ nos da la ventaja de tener un obvio punto de partida: o bien $r \in M$, o bien $r \notin M$. Entonces vamos a hacer un algoritmo que nos regrese el subconjunto independiente de mayor peso de un árbol con su raíz, y otro algoritmo que nos regrese el subconjunto independiente de mayor peso de un árbol sin su raíz, y sencillamente regresaremos el mayor. Ahora, si estamos calculando el subconjunto independiente de un árbol con su raíz, es *evidente* que será la unión de los subconjuntos independientes de los subárboles *sin* sus raíces. Sin embargo, cuando calculemos el subconjunto independiente de un árbol sin su raíz, a lo mejor nos conviene uno de los subconjuntos independientes de los subárboles *sin* la raíz también. Así que por cada hijo habrá que sacar ambos y ver cual nos conviene.

El algoritmo es el siguiente:

```
1: procedure INDEPENDENT-SET(T)
2:     r ←root(T)
3:     M_1 ←INDEPENDENT-SET-WITH(r)
4:     M_2 ←INDEPENDENT-SET-WITHOUT(r)
5:     if w(M_1) ≥ w(M_2) then
6:         return M_1
7:     else
8:         return M_2
```

```
1: procedure INDEPENDENT-SET-WITH(r)
2:     M ← ∅
3:     for all c hijo de r do
4:         M ← M∪ INDEPENDENT-SET-WITHOUT(c)
5:     M ← M ∪ {r}
6:     return M
```

```
1: procedure INDEPENDENT-SET-WITHOUT(r)
2:     M ← ∅
3:     for all c hijo de r do
4:         M_1 ← INDEPENDENT-SET-WITH(c)
5:         M_2 ← INDEPENDENT-SET-WITHOUT(c)
6:         if w(M_1) ≥ w(M_2) then
7:             M ← M ∪ M_1
8:         else
9:             M ← M ∪ M_2
10:    return M
```

El algoritmo funciona porque calcula *todas* las posibilidades, y lo hace rápidamente al no tomar en cuenta las posibilidades prohibidas (jamás se intenta añadir un nodo a $M$ si su padre va a estar). Nuestra estructura de datos para $M$ debe ser inteligente, al recordar su peso y actualizarlo cada vez que un elemento se le agrega, para que consultar el peso del subconjunto nos tome tiempo constante.

El tiempo que toma el algoritmo es trivialmente $O(n)$, y es muy fácil de comprobar porque INDEPENDENT-SET-WITH y INDEPENDENT-SET-WITHOUT son llamadas dos veces *a lo más* por cada vértice en $T$.

18. **(Two finger dialing)** Suponga que quiere marcar un número de $n$ dígitos $\{r_1, \ldots, r_n\}$ en un teléfono estándar con un arreglo de teclas de $4 \times 3$. Suponga que sus dedos están inicialmente en las teclas "*" y "#". De un algoritmo lineal de programación dinámica que minimice la distancia euclidiana total que sus dedos recorren.

R: Platicamos primero el algoritmo antes de presentarlo. La idea es llevar un registro de la posición de los dedos, y en cada paso mover el dedo que recorra la menor distancia. Como siempre en programación dinámica, al realizar el paso $i$ vamos a tener resueltos los pasos $j < i$.

Entonces tendremos un arreglo $A$ que en la entrada $A_{[i]}$ tendrá en qué posición está el dedo $d_1$ después de que se marcó (no necesariamente con $d_1$) el dígito $r_i$, y un arreglo $B$ que en la entrada $B_{[2]}$ nos dirá la posición del dedo $d_2$ después de que se marcó (no necesariamente con $d_2$) el dígito $r_i$. Al inicio, evidentemente $A_{[0]} = *$ y $B_{[0]} = \#$, y para calcular $A_{[i]}$ y $B_{[i]}$ hacemos

$$A_{[i]} = \begin{cases} r_i & \text{si } d(r_i, A_{[i-1]}) < d(r_i, B_{[i-1]}) \\ A_{[i-1]} & \text{si } d(r_i, A_{[i-1]}) > d(r_i, B_{[i-1]}) \end{cases}$$

$$B_{[i]} = \begin{cases} r_i & \text{si } d(r_i, B_{[i-1]}) < d(r_i, A_{[i-1]}) \\ B_{[i-1]} & \text{si } d(r_i, B_{[i-1]}) > d(r_i, A_{[i-1]}) \end{cases}$$

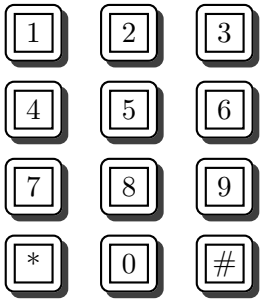
donde $d(r_i, A_{[i-1]})$ es la distancia del dígito $r_i$ en el teclado a la posición del dedo $d_1$ en el paso anterior, y análogo para $B_{[i-1]}$.

Es evidente que nos falta un caso, cuando $d(r_i, B_{[i-1]}) = d(r_i, A_{[i-1]})$. Este caso nos genera ruido porque dependiendo *de los siguientes* dígitos es como debemos decidir si se mueve $d_1$ o $d_2$. Para arreglar esto, llevaremos arreglos "gemelos" de $A$ y $B$ (los llamaremos $X$ y $Y$); mientras $d(r_i, B_{[i-1]}) \neq d(r_i, A_{[i-1]})$, entonces $A_{[i]} = X_{[i]}$ y $B_{[i]} = Y_{[i]}$. Cuando se de que $d(r_i, B_{[i-1]}) = d(r_i, A_{[i-1]})$, en $A$ y $B$ llevaremos la posición de los dedos si se hubiera movido $d_1$, y en $X$ y $Y$ llevaremos la posición de los dedos si se hubiera movido $d_2$.

Lo que estamos haciendo es guardar en $A$ y $B$ un posible recorrido y en $X$ y $Y$ el otro posible recorrido. Los arreglos se actualizarán independientemente hasta que una de estas cosas ocurra: que la distancia que recorren difieran, o que vuelva a darse una igualdad. En el primer caso, agarraremos el recorrido que menor distancia tenga, y haremos que $A$ y $B$ lo representen (no estamos copiando el arreglo; lo podemos ver como "swap" de apuntadores, o bien como un cambio de nombre). En el segundo caso, hacemos que $A$ y $B$ representen el recorrido que *no* tiene una igualdad (para evitarnos complicaciones). En el muy improbable pero posible caso de que *ambos* recorridos tengan una igualdad de nuevo, sencillamente nos olvidamos de la primera divergencia (porque nos conduce a una igualdad idéntica), y ahora $X$ y $Y$ representarán la *nueva* divergencia.

Como nuestra $i$ se irá recorriendo en un **for**, es posible que éste termine con $A$ y $B$ y $X$ y $Y$ intercambiados; así que al final del algoritmo vemos cual es el que recorre menor distancia y a ése hacemos que apunten $A$ y $B$. La idea es que en $A$ y $B$ quede el recorrido de menor distancia

Todo esto funciona perfectamente, excepto para un caso. Nuestro arreglo de teclas es:




En todos los casos siempre conviene mover el dedo más cercano a la tecla; en todos excepto en *uno*. De hecho son dos, pero equivalentes.

Si tenemos la secuencia 1 7..., el algoritmo como está especificado hará que el dedo $d_1$ se mueva de * a 1 (porque es el más cercano), y luego hará que el mismo dedo se mueva de 1 a 7. La distancia total es 5; 3 de moverse de * a 1, y 2 de moverse de 1 a 7.

Sin embargo, si hubiéramos movido el dedo $d_2$ de # a 1, y luego el dedo $d_1$ de * a 1, en total la distancia sería de 4.605... Esto es porque la distancia de # a 7 es 3.605..., y la distancia de * a 1 es 1. El caso equivalente obviamente es la secuencia 3 9....

Es la única instancia del problema en el que falla nuestro algoritmo; todos los demás casos funcionan bien porque realmente no hay tanta distancia entre las teclas. Este caso es justo cuando *más* separados están los dígitos.

Pero no hay necesidad de rehacer todo el algoritmo. Vamos a hacer trampa, y modificar nuestra función que calcula la distancia $d_{[i,j]}$, de tal forma que sea

$$d_{[i,j]} = \begin{cases} \text{la distancia euclidiana entre } i \text{ y } j & \text{si la distancia es menor que 3,} \\ 3 & \text{en otro caso.} \end{cases}$$

Esto hace que los dos casos extremos (de # a 1, y de * a 3) se comporten igual que los casos donde se mueven los dedos una distancia de 3. Esto obliga al algoritmo a llevar dos copias separadas de movimientos, y escoger la menor cuando alguna otra tecla nos determine realmente cuál es la menor.

El algoritmo sería (tomamos a $*$ y a $\#$ como 10 y 11, pero los representamos simbólicamente):

```
1: procedure TWO-FINGER-DIALING(r,n)
2:     X[0] ← A[0] ← *
3:     Y[0] ← B[0] ← #
4:     d1 ← d2 ← 0
5:     equal ← false
6:     for i ← 1 to n do
7:         if equal ≠ true then
8:             X[i] ← A[i] ← A[i−1]
9:             Y[i] ← B[i] ← B[i−1]
10:            if d(r[i], A[i−1]) < d(r[i], B[i−1]) then
11:                X[i] ← A[i] ← r[i]
12:                d1 ← d2 ← d(r[i], A[i−1])
13:            else if d(r[i], B[i−1]) < d(r[i], A[i−1]) then
14:                Y[i] ← B[i] ← r[i]
15:                d1 ← d2 ← d(r[i], B[i−1])
16:            else                          ▷ igualdad, A y B y X y Y comienzan a diferir
17:                equal ← true
18:                A[i] ← r[i]
19:                Y[i] ← r[i]
20:                d1 ← d(r[i], A[i−1])
21:                d2 ← d(r[i], Y[i−1])
22:        else
23:            eq1 ← eq2 ← false
24:            A[i] ← A[i−1]
25:            B[i] ← B[i−1]
26:            X[i] ← X[i−1]
27:            Y[i] ← Y[i−1]
28:            if d(r[i], A[i−1]) < d(r[i], B[i−1]) then
29:                A[i] ← r[i]
30:                d1 ← d(r[i], A[i−1])
31:            else if d(r[i], B[i−1]) < d(r[i], A[i−1]) then
32:                B[i] ← r[i]
33:                d1 ← d(r[i], B[i−1])
34:            else
35:                eq1 ← true
36:            if d(r[i], X[i−1]) < d(r[i], Y[i−1]) then
37:                X[i] ← r[i]
38:                d2 ← d(r[i], X[i−1])
39:            else if d(r[i], Y[i−1]) < d(r[i], X[i−1]) then
40:                Y[i] ← r[i]
41:                d2 ← d(r[i], Y[i−1])
42:            else
43:                eq2 ← true
44:            if eq1 = true and eq2 = true then   ▷ Segunda igualdad; descartamos la primera
45:                A[i] ← r[i]
46:                Y[i] ← r[i]
47:                d1 ← d(r[i], A[i−1])
```

```
48:                 d_2 ← d(r_[i], Y_[i−1])
49:             else if (d_1 ≠ d_2) or (eq_1 = true or eq_2 = true) then
50:                 equal ← false
51:                 if (d_2 < d_1) or eq_2 = true then
52:                     Swap(A,X)
53:                     Swap(B,Y)
54:                     Swap(d_1,d_2)
55:     if d_1 < d_2 then                    ▷ Comprobamos que la ruta más corta esté en A y B
56:         Swap(A,X)
57:         Swap(B,Y)
58:         Swap(d_1,d_2)
59:     return A y B
```

Reconstruir la secuencia en que se movieron los dedos es muy fácil; recorremos $A$ y $B$ y cada vez que $A_{[i-1]} \neq A_{[i]}$, entonces el dedo $d_1$ presionó el dígito en $A_{[i]}$; y de forma análoga para $B$.

El algoritmo es obviamente lineal; aunque hacemos muchas comprobaciones dentro del cuerpo del **for**, en total siempre es constante, y nunca nos regresamos.

19. **(Optimal refueling)** Suponga que quiere viajar de la ciudad $A$ a la ciudad $B$ siguiendo una ruta fija. Puede viajar $m$ kilómetros con el tanque lleno, y tiene un mapa que muestra dónde y a qué distancia están las estaciones de gasolina. De un algoritmo voraz lineal que minimice el número de paradas que tiene que hacer. ¿Qué pasaría si tuviera un mapa, y múltiples opciones de caminos?

R: Platicamos el algoritmo antes de presentarlo formalmente. Asumimos que las ciudades del camino son $c_1, \ldots, c_n$, y que en arreglos $d$ y $g$ tenemos las distancias y la información de las gasolinerías; i.e., $d_{[i]}$ nos dice la distancia entre $c_{i-1}$ y $c_i$, y $g_{[i]}$ es cero si no hay gasolinería en $c_i$, y uno si sí hay. La idea es que vamos a recorrer el camino (comenzamos con el tanque lleno), y llevaremos registro de la última ciudad con gasolinería y cuánto hemos recorrido desde que la dejamos. Cuando nuestra gasolina llegue a menos de cero, vamos a añadir la última ciudad con gasolinería a una lista $L$, y vamos a hacer como si hubiésemos llenado el tanque ahí; vamos a poner de nuevo el tanque en lleno, menos lo que hayamos recorrido desde la ciudad con la gasolinería. En caso de que esta cantidad sea negativa, regresaremos $\emptyset$, porque significa que no podemos recorrer el camino con un tanque que nos de sólo $m$ kilómetros (necesitaríamos más). Si llegamos a $c_n$, entonces regresamos $L$ como la lista donde necesitamos rellenar el tanque.

El algoritmo es el siguiente:

```
1: procedure Optimal-Refueling(d,g,m,n)
2:     L ← ∅
3:     last ← −1
4:     dist ← 0
5:     km ← m
6:     if g_[i] = 1 then
7:         last ← 1
8:     for i ← 2 to n do
9:         km ← km − d_[i]
10:        dist ← dist + d_[i]
11:        if km < 0 then
12:            L ← L ∪ {last}
13:            km ← m − dist
14:            if km ≤ 0 then                ▷ No nos alcanza el tanque, no importa qué hagamos
15:                return ∅
16:        if g_[i] = 1 then
17:            last ← i
18:            dist ← 0
19:    return L
```

El algoritmo funciona porque estamos llenando el tanque únicamente cuando es necesario (cuando se nos acabaría la gasolina si no lo hiciéramos). Es obviamente lineal; tenemos un **for** que corre de 2 a $n$, y realizamos un número constante de operaciones dentro de él.

Si tuviéramos un mapa y múltiples opciones de caminos, entonces la complejidad se nos dispararía a al menos $O(n^2)$, porque necesitaríamos ver qué camino (que contenga gasolinerías) nos conviene tomar. Una solución de programación dinámica probablemente sería lo mejor para ese problema. Es parecido al ejercicio 9 (el de las ciudades), porque podemos interpretar el pasar por no más de $k$ ciudades, como pasar por no más de $k$ gasolinerías.

20. Nos dan una gráfica dirigida $G = (V, E)$ en la cual cada arista $(u, v) \in E$ tiene un valor asociado $r(u, v)$, que es un número real en el rango $0 \leq r(u, v) \leq 1$ que representa la confiabilidad de un canal de comunicación entre el vértice $u$ y el vértice $v$. Interpretamos $r(u, v)$ como la probabilidad de que el canal de comunicación de $u$ a $v$ no fallará, y asumimos que estas probabilidades son independientes. De un algoritmo eficiente para encontrar el camino más confiable entre dos vértices dados.

R: Queremos el camino cuya probabilidad de que ninguna de sus aristas falle sea la más alta. La probabilidad de que en un camino $u \to x_0 \to \ldots \to x_k \to v$ no falle ninguna de sus aristas está dada por:

$$r(u, x_0) \prod_{i=1}^{k} r(x_{i-1}, x_i) r(x_k, v).$$

Entonces necesitamos el camino que maximice ese producto. Esto es equivalente a si encontramos el máximo del logaritmo del producto, dado que el logaritmo es estrictamente creciente. Pero esto es

$$\begin{aligned}
&\log(r(u, x_0) \prod_{i=1}^{k} r(x_{i-1}, x_i) r(x_k, v)) = \\
&\log(r(u, x_0)) + \log\left(\prod_{i=1}^{k} r(x_{i-1}, x_i)\right) + \log(r(x_k, v)). = \\
&\log(r(u, x_0)) + \sum_{i=1}^{k} \log(r(x_{i-1}, x_i)) + \log(r(x_k, v)).
\end{aligned}$$

Pero maximizar esta suma es lo mismo que minimizar esta otra:

$$-\log(r(u, x_0)) + \sum_{i=1}^{k} -\log(r(x_{i-1}, x_i)) - \log(r(x_k, v)).$$

Y entonces si definimos para una arista $(u, v)$ que su peso sea $-\log(r(u, v))$ (que es mayor que cero porque $0 \leq r(u, v) \leq 1$), el algoritmo de Dijkstra nos encontrará la suma minimizada.

Y por lo tanto Dijkstra es un algoritmo eficiente que nos resuelve el problema si definimos el peso de una arista $(u, v)$ como $-\log(r(u, v))$.

21. Sea $G = (V, E)$ una gráfica dirigida con pesos, con función de peso $w : E \to 0, 1, \ldots, W$ para algún entero no negativo $W$.

*a)* Modifique el algoritmo de Dijkstra para que calcule los caminos más cortos a partir de un vértice dado en tiempo $O(|W||V| + |E|)$. Justifique la correctez y tiempo de ejecución de su algoritmo.

R: Hay que notar que si el camino más corto entre el vértice origen y cualquier otro vértice no es $\infty$, entonces a lo más es $(|V| - 1)W$. Para que tengamos $d_{[v]} < \infty$ tenemos que haber relajado a la arista $(u, v)$ con $d_{[u]} < \infty$. Es claro que si relajamos $(u, v)$ entonces $d_{[v]}$ es a lo más el número de aristas en el camino de $s$ a $v$, multiplicado por el peso de la arista más pesada en el camino. Ya que un camino acíclico tiene a lo más $|V| - 1$ aristas y el peso máximo de una arista es $W$, tenemos que $d_{[v]} \leq (|V| - 1)W$. Como todos los pesos son enteros, entonces $d_{[v]}$ también será entero (a menos que sea $\infty$).

También notamos que en el algoritmo de Dijkstra los valores que EXTRACT-MIN regresa son monotónicamente crecientes, porque después de hacer $|V|$ operaciones de insertar, ya no hacemos otra inserción. Ya que los pesos de las aristas son no negativos, cuando relajamos $(u, v)$ tenemos que $d_{[u]} \leq d_{[v]}$. Ya que $u$ es el mínimo vértice que extraímos, sabemos que cualquier otro vértice que extraigamos después tiene un valor que a lo menos es igual a $d_{[u]}$.

Cuando las llaves son enteros en el rango $0, \ldots, k$ y los valores de las llaves extraídas crecen monotónicamente con el tiempo, podemos implementar una cola de prioridad para que cualquier secuencia de $m$ inserciones, extracciones y decrementos en las llaves tomen $O(m + k)$. Para esto, usamos un arreglo $A_{[0,\ldots,k]}$ donde $A_{[j]}$ es una lista ligada de cada elemento cuya llave es $j$. Implementamos cada lista como una lista doblemente ligada circular con un sentinela, así que podemos insertar o borrar de cada lista en tiempo $O(1)$. Las operaciones de la cola de prioridad serían como sigue:

- INSERT: Para insertar un elemento con llave $j$, sólo lo insertamos en la lista ligada en $A_{[j]}$. Toma tiempo $O(1)$.
- EXTRACT-MIN: Guardamos un índice $min$ de el valor de la llave más pequeña extraída. Inicialmente $min \leftarrow 0$. Para encontrar la llave más pequeña, buscamos en $A_{[min]}$ y, si esta lista es no vacía, usamos cualquier elemento en ella, quitándolo de la lista y regresándolo. Si $A_{[min]}$ es vacía, nos aprovechamos de que EXTRACT-MIN es monotónicamente creciente e incrementamos $min$ hasta que encontremos una lista $A_{[min]}$ que no sea vacía (usando cualquier elemento en $A_{[min]}$ como arriba), o hasta que se nos acabe el arreglo (en cuyo caso la cola de prioridad está vacía).
  Ya que hay a lo más $m$ operaciones de inserción, hay a lo más $m$ elementos en la cola de prioridad. Incrementamos $min$ a lo más $k$ veces, y quitamos y regresamos un elemento a lo más $m$ veces. Por tanto, el tiempo total de todas las operaciones EXTRACT-MIN es $O(m + k)$.
- DECREASE-KEY: Para decrementar la llave de un elemento $j$ a $i$, comprobamos que $i \leq j$, y elevamos una excepción si no. De otra forma, quitamos el elemento de la lista $A_{[j]}$ en $O(1)$ y la insertamos en $A_{[i]}$ en $O(1)$, lo que nos toma un tiempo de $O(1)$.

Para aplicar este tipo de cola de prioridad al algoritmo de Dijkstra, necesitamos que $k = (|V| - 1)W$, y necesitamos una lista aparte para las llaves con valor $\infty$. El número de operaciones $m$ es $O(|V| + |E|)$ (ya que hay $|V|$ operaciones de inserción y $|V|$ operaciones de extración, y a lo más $|E|$ de decremento de llave), y por lo tanto el tiempo total es $O(|V| + |E| + |VW|) = O(|VW| + |E|)$.

*b*) Modifique su algoritmo previo para que corra en tiempo $O((V+E)\ln W)$. Justifique la correctez y tiempo de ejecución de su algoritmo.

R: Observemos que en cualquier momento, hay a lo más $W+2$ llaves distintas en la cola de prioridad, porque una llave es $\infty$ o no, y cuando una llave $d_{[v]}$ se vuelve finita, debe ser porque se relajó una arista $(u,v)$. Para ese momento, $u$ estaba siendo insertada en $S$, y $d_{[u]} \leq d_{[y]}$ para todos los vértices $y \in V-S$. Después de relajar la arista $(u,v)$, tenemos $d_{[v]} \leq d_{[u]} + W$. Ya que cualquier otro vértice $y \in V-S$ con $d_{[y]} < \infty$ también tuvo su estimado cambiado por un relajamiento de alguna arista $x$ con $d_{[x]} \leq d_{[u]}$, debemos tener que $d_{[y]} \leq d_{[x]} + W \leq d_{[u]} + W$.

Por lo tanto, al momento de que estamos relajando las aristas de un vértice $u$, debemos tener para todos los vértices $v \in V-S$, que $d_{[u]} \leq d_{[v]} \leq d_{[u]} + W$ o bien $d_{[u]} = \infty$.

Ya que los estimados del camino más corto son valores enteros (excepto por $\infty$), en cualquier momento tenemos a lo más $W+2$ valores distintos: $d_{[u]}, d_{[u]}+1, d_{[u]}+2, \ldots, d_{[u]}+W$ y $\infty$.

Por lo tanto, podemos mantener la cola de prioridad como un min-heap binario en el cual cada nodo apunta a una lista doblemente ligada con todos los vértices para una llave dada. Habrá a lo más $W+2$ valores en el heap, y por lo tanto Extrac-Min correrá en tiempo $O(\ln W)$. Podemos mejorarlo a $O(1)$ de la siguiente manera; primero, mantendremos un apuntador $inf$ al vértice que contenga todas las llaves $\infty$. Segundo, guardaremos un arreglo $loc_{[0,\ldots,W]}$, donde $loc_{[i]}$ apuntará a la única entrada del heap cuya llave sea congruente con $i$ (mod $(W+1)$). Conforme las llaves se muevan en el heap, podemos actualizar este arreglo en tiempo $O(1)$ por movimiento.

De forma alternativa, en lugar de usar un min-heap binario, podemos usar un árbol red-black, y así las operaciones de insertar, borrar, mínimo y búsqueda (a partir de las cuales construimos las operaciones de la cola de prioridad) cada una correría en tiempo $O(\ln W)$, lo que nos daría un tiempo total de ejecución para el algoritmo de Dijkstra de $O((V+E)\ln W)$.

22. Sea $G=(V,E)$ una gráfica no dirigida. Para $u,v \in V$ de un algoritmo eficiente que calcule el máximo número posible de caminos de $u$ a $v$ mutuamente disjuntos en las aristas. Asumiendo que queremos calcular esto para caminos disjuntos en los vértices, modifique su algoritmo. Pruebe la correctez y tiempo de ejecución de su algoritmo.

R: La idea del algoritmo es bastante sencilla; tendremos un contador $c$ que inicialmente pondremos en cero. Después correremos BFS sobre $G$ con $u$ como vértice origen. Si $v$ no es alcanzado por BFS (no se colorea de negro), terminamos el algoritmo. Si sí, quitamos de $E$ las aristas $(v,\pi_{[v]}),(\pi_{[v]},\pi^2_{[v]}),\ldots,(\pi^{k-1}_{[v]},u)$, donde $k$ es la longitud (número de aristas) entre $u$ y $v$ que nos encontró el correr BFS. Dado que queremos caminos mutuamente disjuntos, las aristas de este primer camino ya no pueden aparecer en ningún otro; entonces sencillamente las quitamos, y aumentamos $c$ en uno. Después, sencillamente repetimos. El algoritmo sería:

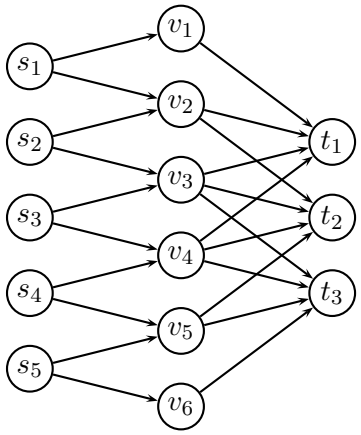
```
 1: procedure DISJOINT-PATHS(G,u,v)
 2:     c ← 0
 3:     repeat
 4:         BFS(G,u)
 5:         if color_[v] = BLACK then
 6:             t ← v
 7:             while π_[t] ≠ nil do
 8:                 E ← E − {(t, π_[t])}
 9:                 t ← π_[t]
10:             c ← c + 1
11:     until color_[v] ≠ BLACK
12:     return c
```

El algoritmo funciona porque va encontrando caminos entre $u$ y $v$, y los va quitando; termina cuando ya no hay caminos posibles entre $u$ y $v$. Correr cada BFS nos toma tiempo $O(|V|+|E|)$, y quitar las aristas (si hay camino) nos toma $O(|V|)$ en el peor de los casos. Esto lo repetimos por tantos caminos de aristas disjuntas como haya de $u$ a $v$ en $G$, que en el peor de los casos son $O(|V|)$. Así que (inocentemente), la complejidad del algoritmo es $O(|V|^2+|V||E|)$. Sin embargo, en cada corrida de BFS eliminamos a cierto número de aristas, lo que nos acelera las siguientes corridas de BFS, y de hecho podemos optimizar BFS para que se detenga en cuanto $v$ sea coloreado de negro; así que en la práctica la complejidad es bastante menor.

Para hacer lo mismo con caminos disjuntos en los vértices, sencillamente quitamos los vértices del camino (excepto $u$ y $v$) en lugar de quitar las aristas.

23. Extienda las propiedades y definiciones de flujos al problema de múltiples orígenes y destinos. Muestre que cualquier flujo en una red de flujo de múltiples orígenes y destinos corresponde a un flujo de valor igual en la red de un único origen y destino que se obtiene de añadir un super origen y un super destino, y viceversa.

R: Una *red de flujo* $G = (V, E)$ *con múltiples orígenes y destinos* es una gráfica dirigida en la cual cada vértice $(u, v) \in E$ tiene una capacidad no negativa $c(u, v) \geq 0$. Si $(u, v) \notin E$, asumimos que $c(u, v) = 0$. Distinguimos dos conjuntos de vértices en una red de flujo con múltiples orígenes y destinos: un conjunto de vértices orígenes $S = \{s_1, \ldots, s_m\}$ y un conjunto de vértices destinos $T = \{t_1, \ldots, t_n\}$. Por conveniencia, asumimos que la gráfica es conexa, y por tanto que $|E| \geq |V| - 1$. La siguiente figura muestra una red de flujo con múltiples orígenes y destinos.




Sea $G = (V, E)$ una red de flujo con múltiples orígenes y destinos y función de capacidad $c$. Sea $S$ el conjunto de vértices orígenes de la red y $T$ el conjunto de vértices destinos. Entonces un *flujo* en $G$ es una función real $f : V \times V \to \mathbb{R}$ que satisface las tres siguientes propiedades:

- **Restricción de capacidad**: Para todo $u, v \in V$, requerimos que $f(u, v) \leq c(u, v)$.
- **Simetría oblicua**: Para todo $u, v \in V$, requerimos que $f(u, v) = -f(v, u)$.
- **Conservación de flujo**: Para todo $u \in V - (S \cup T)$, requerimos que

$$\sum_{v \in V} f(u, v) = 0.$$

El *valor* de un flujo $f$ se define como

$$|f| = \sum_{s \in S} \sum_{v \in V} f(s, v).$$

Si $X, Y \subseteq V$, definimos $f(X, Y)$ como

$$f(X, Y) = \sum_{u \in X} \sum_{v \in Y} f(u, v).$$

De donde tenemos que $|f| = f(S, V)$.

Vamos a demostrar el siguiente lema, porque necesitamos los resultados para demostrar que un flujo en una red de flujo con múltiples orígenes y destinos es equivalente a un flujo en una red de flujo con un único origen y un único destino.

**Lema:** Sea $G = (V, E)$ una red de flujo con múltiples orígenes y destinos, y sea $f$ un flujo en $G$. Entonces las siguientes igualdades son ciertas:

*a*) Para todo $X \subseteq V$, tenemos que $f(X, X) = 0$.

*b*) Para todo $X, Y \subseteq V$, tenemos que $f(X, Y) = -f(Y, X)$.

*c*) Para todo $X, Y, Z \subseteq V$, con $X \cap Y = \emptyset$, tenemos que $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ y que $f(Z, X \cup Y) = f(Z, X) = f(Z, Y)$.

Demostración:

*b*)

$$\begin{aligned} f(X, Y) &= \sum_{x \in X} \sum_{y \in Y} f(x, y) \\ &= \sum_{x \in X} \sum_{y \in Y} -f(y, x) \quad \text{(simetría oblicua)} \\ &= \sum_{y \in Y} \sum_{x \in X} -f(y, x) \\ &= -f(Y, X). \end{aligned}$$

*a*) Utilizando *b*):

$$f(X, X) = -f(X, X) \quad \Rightarrow \quad f(X, X) = 0.$$

*c*)

$$\begin{aligned} f(X \cup Y, Z) &= \sum_{v \in X \cup Y} \sum_{z \in Z} f(v, z) \\ &= \sum_{v \in X} \left( \sum_{z \in Z} f(v, z) \right) + \sum_{v \in Y} \left( \sum_{z \in Z} f(v, z) \right) \quad (X \cap Y = \emptyset) \\ &= f(X, Z) + f(Y, Z). \end{aligned}$$

La otra parte es simétrica.

Con este lema, podemos demostrar que $|f| = f(V, T)$:

$$\begin{array}{rcll} |f| & = & f(S, V) & \text{(por definición)} \\ & = & f(V, V) - f(V - S, V) & \text{(por el lema, parte } c\text{)} \\ & = & -f(V - S, V) & \text{(por el lema, parte } a\text{)} \\ & = & f(V, V - S) & \text{(por el lema, parte } b\text{)} \\ & = & f(V, T) + f(V, V - (S \cup T)) & \text{(por el lema, parte } c\text{)} \\ & = & f(V, T). & \text{(por conservación de flujo)} \end{array}$$

Ahora sí podemos probar la equivalencia entre flujos de redes de flujo con múltiples orígenes y destinos, y flujos de redes de flujo con un único origen y un único destino.

Sea $G = (V, E)$ una red de flujo con múltiples orígenes y destinos, y sea $f$ un flujo en $G$ con valor $|f|$. Sea $G' = (V', E')$ la red de flujo con un único origen y un único destino que resulta de hacer $V' = V \cup \{s, t\}$, y de hacer $E' = E \cup \{(s, s_1), \ldots, (s, s_m), (t_1, t), \ldots, (t_n, t)\}$. Definimos las capacidades de las nuevas aristas como sigue:

$$c(s, s_i) = \infty \quad \forall\ s_i \in S, \quad c(t_i, t) = \infty \quad \forall\ t_i \in T.$$

Ahora, queremos demostrar que existe un flujo $f'$ en $G'$ tal que $|f'| = |f|$.

Por construcción; definimos

$$\begin{array}{rcll} f'(u, v) & = & f(u, v) & \forall\ u, v \in V, \\ f'(s, s_i) & = & \displaystyle\sum_{v \in V} f(s_i, v) & \forall\ s_i \in S, \\ f'(s_i, s) & = & -f'(s, s_i) & \forall\ s_i \in S, \\ f'(t_i, t) & = & \displaystyle\sum_{v \in V} f(v, t_i) & \forall\ t_i \in T, \\ f'(t, t_i) & = & -f'(t_i, t) & \forall\ t_i \in T. \end{array}$$

Vamos a demostrar que $f'$ cumple con las propiedades de restricción de capacidad, de simetría oblicua, y de conservación de flujo.

- **Restricción de capacidad**: Para todo $u, v \in V'$, requerimos que $f'(u, v) \leq c(u, v)$.
  Dado que $f'$ se comporta igual que $f$ en $V \times V$, y que las capacidades de las aristas en $E' - E$ es $\infty$, se cumple que para todo $u, v \in V$, $f'(u, v) \leq c(u, v)$.
- **Simetría oblicua**: Para todo $u, v \in V'$, requerimos que $f'(u, v) = -f'(v, u)$.
  De nuevo, dado que $f'$ se comporta igual que $f$ en $V \times V$, y que definimos $f'$ en el resto para que específicamente se cumpliera que $f'(u, v) = -f'(v, u)$, entonces se cumple la simetría oblicua.
- **Conservación de flujo**: Para todo $u \in V' - \{s, t\}$, requerimos que

  $$\sum_{v \in V'} f'(u, v) = 0.$$

  Tenemos que (utilizando las igualdades del lema de arriba y la definición de $f'$):

$$
\begin{aligned}
\sum_{u\in V'-\{s,t\}}\sum_{v\in V'} f'(u,v) &= \sum_{u\in V}\sum_{v\in V'} f'(u,v)\\
&= \sum_{u\in V}\sum_{v\in V} f'(u,v) + \sum_{u\in V}\sum_{v\in\{s,t\}} f'(u,v)\\
&= \sum_{u\in V}\sum_{v\in V} f(u,v) + \sum_{u\in V} f'(u,s) + \sum_{u\in V} f'(u,t)\\
&= f(V,V) + \sum_{u\in V} -f'(s,u) + \sum_{u\in V} f'(u,t)\\
&= \sum_{s_i\in S} -f'(s,s_i) + \sum_{t_i\in T} f'(t_i,t)\\
&= \sum_{v\in V}\sum_{s_i\in S} -f(s_i,v) + \sum_{v\in V}\sum_{t_i\in T} f(v,t_i)\\
&= \sum_{s_i\in S}\sum_{v\in V} -f(s_i,v) + \sum_{v\in V}\sum_{t_i\in T} f(v,t_i)\\
&= -f(S,V) + f(V,T)\\
&= -|f| + |f|\\
&= 0.
\end{aligned}
$$

Y por lo tanto, $f'$ es un flujo en $G'$. Ahora, demostraremos que $|f'| = |f|$.

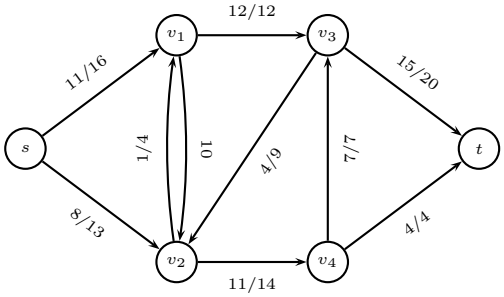
$$
\begin{aligned}
|f'| &= f'(s,V)\\
&= \sum_{v\in V} f'(s,v)\\
&= \sum_{s_i\in S} f'(s,s_i)\\
&= \sum_{s_i\in S}\sum_{v\in V} f(s_i,v)\\
&= f(S,V)\\
&= |f|.
\end{aligned}
$$

Y por lo tanto, para todo flujo $f$ en una red de flujo $G$ de múltiples orígenes y destinos, existe un flujo $f'$ equivalente en la red de flujo $G'$ que resulta de agregarle a $G$ un super origen y un superdestino.

El inverso es análogo.

24. Para la red de flujo $G = (V, E)$ y el flujo que se muestra en la figura 27.1 (b) del Cormen, encuentre un par de subconjuntos $X, Y \subseteq V$ para los cuales $f(X, Y) = -f(V - X, Y)$. Después encuentre un par de subconjuntos $X, Y \subseteq V$ para los cuales $f(X, Y) \neq -f(V - X, Y)$.

R: La gráfica es



Tomamos $X = \{v_1\}$ y a $Y = \{v_2, v_3, v_4\}$, y calculamos sus flujos; primero de $f(X, Y)$:

$$\begin{aligned} f(X,Y) &= \sum_{x \in X} \sum_{y \in Y} f(x,y) \\ &= f(v_1, v_2) + f(v_1, v_3) + f(v_1, v_4) \\ &= -1 + 12 \\ &= 11. \end{aligned}$$

Y ahora de $f(V - X, Y)$ ($V - X = \{s, v_2, v_3, v_4, t\}$):

$$\begin{aligned} f(V-X,Y) &= \sum_{x \in V-X} \sum_{y \in Y} f(x,y) \\ &= f(s,v_2) + f(s,v_3) + f(s,v_4) + f(v_2,v_2) + f(v_2,v_3) + f(f_{v2},v_4) + \\ &\quad f(v_3,v_2) + f(v_3,v_3) + f(f_{v3},v_4) + f(v_4,v_2) + f(v_4,v_3) + f(f_{v4},v_4) + \\ &\quad f(t,v_2) + f(t,v_3) + f(t,v_4) \\ &= 8 + 0 + 0 + 0 - 4 + 11 + 4 + 0 - 7 - 11 + 7 + 0 + 0 - 15 - 4 \\ &= 8 - 15 - 4 \\ &= -11. \end{aligned}$$

Y por lo tanto, $f(X, Y) = -f(V - X, Y)$.

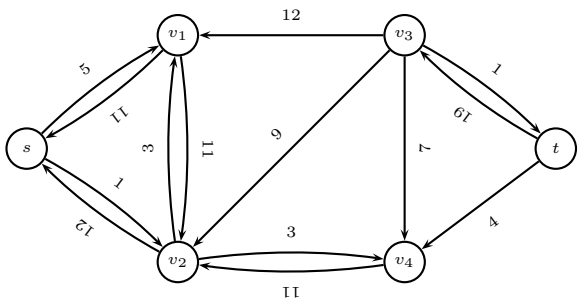
Ahora, si tomamos $X = \{s, v_1, v_2, t\}$ y $Y = \{v_3, v_4\}$ tenemos que:

$$\begin{aligned} f(X,Y) &= \sum_{x \in X} \sum_{y \in Y} f(x,y) \\ &= f(s,v_3) + f(s,v_4) + f(v_1,v_3) + f(v_1,v_4) + \\ &\quad f(v_2,v_3) + f(v_2,v_4) + f(t,v_3) + f(t,v_4) \\ &= 0 + 0 + 12 + 0 - 4 + 11 - 15 - 4 \\ &= 11. \end{aligned}$$

Ahora $V - X = \{v_3, v_4\} = Y$, y una de nuestras reglas es que $f(Y, Y) = 0$; por lo tanto, $f(X, Y) \neq f(V - X, Y)$.

25. En el ejemplo de la figura 27.6 del Cormen, ¿cuál es el mínimo corte correspondiente al máximo flujo mostrado? De los caminos aumentantes que aparecen en el ejemplo, ¿cuáles dos cancelan flujos que anteriormente se habían enviado?

R: La figura al final de correr el algoritmo quedaba:

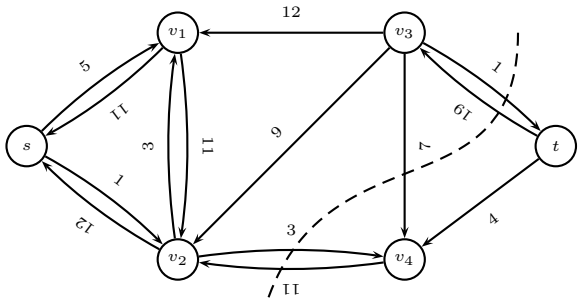



El flujo de esta red es 23, que se aprecia viendo las aristas que inciden en $s$ y las que salen de $t$. Entonces un corte mínimo $S, T$ de la red debe tener una capacidad $c(S, T) = 23 = |f|$.

Sea $S = \{s, v_1, v_2, v_4\}$ y $T = \{v_3, t\}$. La capacidad $c(S, T)$ del corte sería entonces:

$$
\begin{aligned}
c(S,T) &= \sum_{u\in S}\sum_{v\in T} c(u,v) \\
&= c(s,v_3) + c(s,t) + c(v_1,v_3) + c(v_1,t) + \\
&\quad c(v_2,v_3) + c(v_2,t) + c(v_4,v_3) + c(v_4,t) \\
&= 0 + 0 + 12 + 0 + 0 + 0 + 7 + 4 \\
&= 12 + 7 + 4 \\
&= 23.
\end{aligned}
$$

Así que $(S,T)$ es un corte mínimo de la red. Gráficamente sería

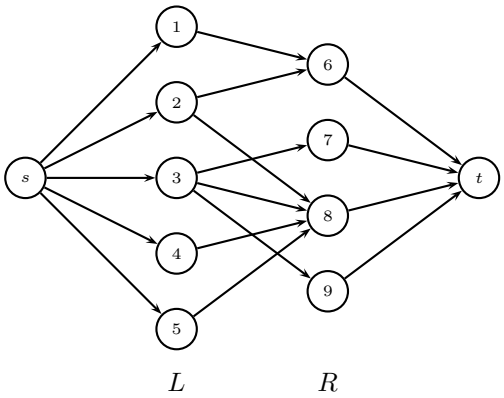



En el paso (c) el flujo de $v_2$ a $v_1$ cancela siete unidades del flujo de once que en el paso (b) se habían enviado de $v_1$ a $v_2$.

En el paso (d) el flujo de $v_2$ a $v_3$ cancela cuatro unidades del flujo de nueve que en el paso (a) se habían enviado de $v_3$ a $v_2$.

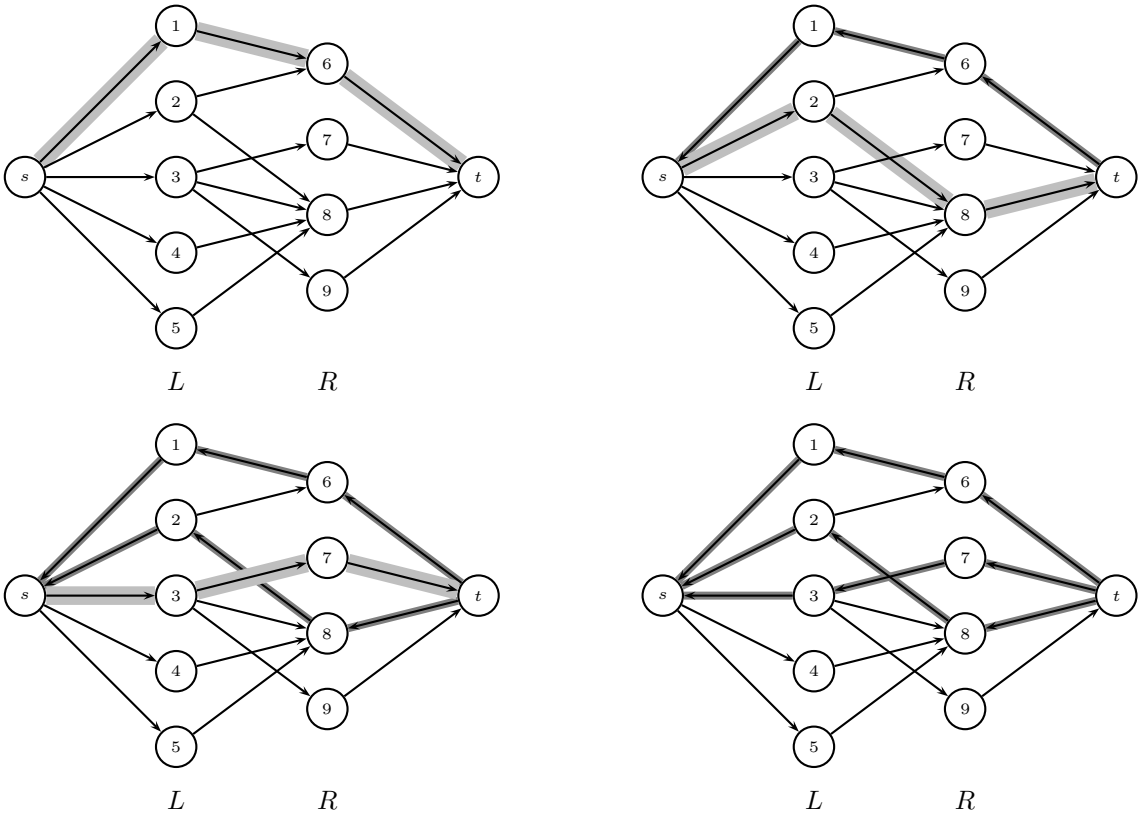
26. Corra el algoritmo de Ford-Fulkerson en la red de flujo de la figura 27.9 (b) del Cormen y muestre la red residual después de cada aumento de flujo. Numere los vértices en $L$ de arriba a abajo del 1 al 5, y en $R$ de arriba a abajo del 6 al 9. Para cada iteración, escoja el camino aumentante que sea más pequeño lexicográficamente.

R: La red es, numerando los vértices como se pide




Siguiendo las especificaciones del ejemplo, todas las aristas tienen capacidad uno e inicialmente flujo cero. Vamos a empujar un flujo entero, entonces se van a agotar las aristas a la primera, y en este caso en particular una arista se utiliza para enviar flujo sólo una vez. Por ello cuando una arista vaya de izquierda a derecha, vamos a decir que tiene flujo cero y capacidad uno; y cuando vaya de derecha a izquierda, vamos a decir que tiene flujo uno (menos uno de izquierda a derecha) y capacidad uno.

En cada paso, marcamos el camino aumentante más pequeño lexicográficamente. Vamos a dejar marcadas también las aristas que ya hayamos agotado, para que al final se vea bien las rutas que el flujo puede tomar.




27. Una gráfica bipartita $G = (V, E)$, donde $V = L \cup R$ es *d-regular* si cada vértice $v \in V$ tiene grado de exactamente $d$. Cada gráfica bipartita $d$-regular tiene $|L| = |R|$. Pruebe que cada gráfica bipartita $d$-regular tiene un emparejamiento de cardinalidad $|L|$ argumentando que un corte mínimo de la correspondiente red de flujo tiene capacidad $|L|$.

R: Por el lema 26.10 del Cormen, si en una red de flujo obtenida a partir de una gráfica bipartita tenemos un flujo $f$ con valor $|f|$, entonces existe un emparejamiento en la gráfica bipartita con cardinalidad $|f|$.

Sea $G' = (V', E')$ la red de flujo que obtenemos a partir de la gráfica bipartita $G$. Obviamente, un flujo de $G'$ no puede ser mayor a $|L|$, porque de $s$ salen $|L|$ aristas con capacidad 1, y en $t$ inciden $|L|$ aristas con capacidad 1; así que no hay forma de empujar más de $|L|$ de flujo desde $s$ o hacia $t$.

Sea $S = \{s\}$ y $T = V' - S$ un corte de $G'$. La capacidad del corte $(S, T)$ está definido como $c(S, T)$, pero $c(S, T) = c(s, T)$ (porque definimos $S = \{s\}$), y $c(s, T) = c(s, L)$ (porque de $s$ sólo salen aristas a los vértices de $L$). Por definición, tenemos que

$$c(s, L) = \sum_{v \in L} c(s, v).$$

Pero la capacidad de cada arista $(s, v)$ con $v \in L$ es 1 (así se construyen las redes de flujo a partir de gráficas bipartitas), por lo tanto

$$\sum_{v \in L} c(s, v) = |L|.$$

Por el lema 26.6 del Cormen, existe un flujo $f$ en $G'$ tal que $|f| = c(S, T) = c(s, L) = |L|$. Por el lema 26.10 del Cormen, entonces existe un emparejamiento en $G$ de cardinalidad $|L|$.

Podemos concluir más cosas; como dijimos arriba, no puede haber un flujo de más de $|L|$ en $G'$, por lo tanto $f$ es un flujo máximo en $G'$. Y por el teorema del corte mínimo-flujo máximo, $(S, T)$ es un corte mínimo de $G'$.