Dividir y Conquistar

Soluciones los ejercicios de la clase de D&C
Algoritmos y Estructuras de datos II

21 de Octubre de 2016

Teorema Maestro

Para resolver recurrencias de la forma:

$$T(n) = \begin{cases} at\left(\frac{n}{b}\right) + f(n) & n > 1 \\ \theta(1) & n = 1 \end{cases}$$

Tres casos:

$$T(n) = \begin{cases} \theta(n^{\log_c a}) & \text{Si } \exists \varepsilon > 0 \text{ tal que } f(n) \in O(n^{\log_c a - \varepsilon}) \\ \theta(n^{\log_c a} \log n) & \text{Si } f(n) \in \theta(n^{\log_c a}) \\ \theta(f(n)) & \text{Si } \exists \varepsilon > 0 \text{ tal que } f(n) \in \Omega(n^{\log_c a + \varepsilon}) \text{ y} \\ & \exists \delta < 1 \text{ tal que } \forall n \geq n_0 \text{ se cumple } af\left(\frac{n}{b}\right) \leq \delta f(n) \end{cases}$$

1 Búsqueda en arreglo ordenado

Problema

Dado un arreglo de elementos ordenados, encontrar el elemento e, si está. Devolver el índice donde se encuentra, o -1 si no está.

Idea de la resolución

Usamos búsqueda binaria sobre el arreglo ordenado, siguiendo los siguientes pasos:

- Casos base: el arreglo es vacío (devolvemos -1) o tiene 1 elemento (comparamos con nuestro elemento y devolvemos el índice si es igual, o -1 si no.
- **(Dividir)** Dividir el arreglo en dos partes.
- **(Conquistar)** Comparar el elemento buscado con el que está en medio del arreglo para ver en qué mitad puede estar. Hacer recursión sobre esa mitad.

Necesitamos utilizar subíndices, porque sino perdemos la posición en el arreglo original.

Resolución

```
BUSQUEDA(arreglo A, elem e)
1  return BUSQUEDADC(A, e, 0, len(A)-1)

BUSQUEDADC(arreglo A, elem e, int bajo, int alto)
 1  if alto < bajo
 2     then
 3          ▷ El arreglo es vacio
 4          return -1
 5  if alto == bajo
 6     then
 7          ▷ El arreglo tiene 1 elemento
 8          if A[alto] == e
 9            then
10                 return alto
11          return -1
12
13  medio ← {(bajo+alto)/2}
14
15  if A[medio] < e
16     then
17          return BUSQUEDADC(A, e, medio+1, alto)
18  return BUSQUEDADC(A, e, bajo, medio)
```

Análisis de complejidad

La recursión es del tipo necesario para poder usar el Teorema Maestro. Veamos cuánto vale cada cosa.
En cada paso nos quedamos con un sólo subproblema que tiene la mitad de tamaño que el original, así que $a = 1$ y $c = 2$. Además de la recursión, sólo hacemos menos de 6 comparaciones y asignaciones, por lo tanto $f(n) \in \mathcal{O}(1)$.

Tenemos entonces T(n)=a*($\frac{n}{2}$) + $\mathcal{O}(1)$
No podemos usar el primer caso del Teorema Maestro, porque debería ser $f(n) \in \theta(n^{-\varepsilon})$, con $\varepsilon > 0$, y esto es imposible pues $1 > n^{-\varepsilon}$ para cualquier $\varepsilon > 0$.
Podemos en cambio usar el segundo caso, porque f(n) $\in \theta(1)$ (ya vimos $f(n) \in \mathcal{O}(1)$, y claramente $f(n) \in \Omega(1)$).
Reemplazando, obtenemos $T(n) \in \theta(n^{\log_2 1} \log n)$, o sea $T(n) \in \theta(\log n)$

2 Máximo

Problema

Encontrar el máximo en un arreglo A compuesto por una secuencia estrictamente creciente seguida de una estrictamente decreciente.
O sea, existe un k ($0 \leq k \leq (tam(A) - 1)$) tal que
$(\forall i, i < k)A[i] < A[i + 1]$ y $(\forall j, k \leq j \leq (tam(A) - 2))A[j] > A[j + 1]$
Suponemos que hay al menos un elemento a izquierda y uno a derecha del máximo (las secuencias creciente y decreciente tienen al menos 2 elementos). De esto se deduce que la secuencia tiene al menos 3 elementos.

Idea de la resolución

- El **caso base** es cuando la secuencia tiene 3 elementos, dado que por lo que supusimos anteriormente (y si mantenemos el invariante a lo largo de la resolución) las secuencias crecientes y decreciente deben tener dos elementos cada una, por lo que el máximo debe ser el del medio.
- Para los otros casos, **dividimos** la secuencia en dos. Llamaremos c_2 al elemento del medio, c_1 al de su izquierda y c_3 al de su derecha. Ahora podemos encontrar tres casos:
  - $c_2 > c_1$ y $c_2 > c_3$. Como las secuencias son estrictamente creciente y decreciente, este caso sólo se puede dar si c_2 es el máximo, por lo que devolvemos c_2
  - $c_1 > c_2$ y $c_2 > c_3$. En este caso ya estamos en la secuencia decreciente, por lo que el máximo tiene que estar en la primera mitad. Llamamos a la función entre el principio y c_2.
  - $c_2 > c_1$ y $c_3 > c_1$. Estamos en la secuencia creciente, el máximo tiene que estar en la segunda mitad. Llamamos a la función entre c_2 y el final.
  - El último caso ($c_1 > c_2$ y $c_3 > c_2$) no se puede dar nunca.

Resolución

```
MAXIMO(arreglo A)
 1  if len(A) == 3
 2     then
 3          return A[1]
 4
 5  medio ← {(len(A)−1)/2}
 6
 7  if A[medio] > A[medio − 1] and A[medio] > A[medio + 1]
 8     then
 9          return A[medio]
10  if A[medio − 1] > A[medio] and A[medio] > A[medio + 1]
11     then
12          return MAXIMO(A[0,medio])
13
14  return MAXIMO(A[medio, long(A) - 1])
```

Análisis de complejidad

En cada paso nos quedamos con un sólo subproblema que tiene la mitad de tamaño que el original, así que $a = 1$ y $c = 2$. Para calcular $f(n)$, tenemos que ver qué hacemos en cada paso además de la recursión. Vemos que esto se reduce a una cantidad constante de comparaciones y asignaciones, por lo tanto $f(n) \in \mathcal{O}(1)$. Estamos en la misma situación que en el caso anterior, por lo que podemos usar el segundo caso del Teorema Maestro obteniendo que $T(n) \in \Theta(logn)$

3 Subsecuencia de suma máxima

Problema

Dada una secuencia de n enteros, se desea encontrar el máximo valor que se puede obtener sumando elementos consecutivos.
Por ejemplo, para la secuencia (3, -1, 4, 8, -2, 2, -7, 5), este valor es 14, que se obtiene de la subsecuencia (3, -1, 4, 8).
Si una secuencia tiene todos números negativos, se entiende que su subsecuencia de suma máxima es la vacía, por lo tanto el valor es 0.

Idea de la resolución

Para usar Dividir y Conquistar para resolver este problema, consideramos que, si dividimos la secuencia en dos partes, pueden pasar tres cosas:

- La subsecuencia de suma máxima está en la mitad izquierda
- La subsecuencia de suma máxima está en la mitad derecha
- La subsecuencia de suma máxima tiene parte en la mitad izquierda y parte en la mitad derecha.

Si encontramos la subsecuencia de suma máxima de cada uno de estos tipos y calculamos el máximo entre estos tres, obtenemos el resultado que buscamos. Los primeros dos casos los podemos calcular llamando recursivamente a la función en ambas mitades y buscando el máximo entre estos dos. Sin embargo si la subsecuencia de suma máxima está en el medio, ninguno devolverá el valor correspondiente a esta, por lo tanto tenemos que buscarla además. Es fácil encontrar la subsecuencia de suma máxima que contiene a los elementos centrales en tiempo lineal. Luego sólo queda encontrar el máximo.

- Casos base: La secuencia no tiene elementos (devolvemos 0, la suma de la subsec. vacía), o tiene un sólo elemento, en cuyo caso devolvemos el máximo entre el único elemento y 0.
- **(Dividir)** Dividimos la secuencia en 2.
- **(Conquistar)** Llamamos recursivamente a la función sobre ambas mitades. Guardamos los resultados de las subsecuencias de suma máxima izquierda y derecha.
- **(Combinar)** Calculamos la subsecuencia de suma máxima que contiene a los dos elementos centrales, encontrando la máxima hacia la derecha y hacia la izquierda, y sumando los resultados. Luego calculamos el máximo entre esta y las calculadas anteriormente.

Resolución

```
SumaMaxima(A)
 1  if len(A) == 0
 2     then
 3          return 0
 4  if len(A) == 0
 5     then
 6          return Max(A[1], 0)
 7
 8  mitad ← (len(A)−1)/2
 9
10  max_izq ← SumaMaxima(A[0, mitad])
11
12  max_der ← SumaMaxima(A[mitad + 1, len(A) − 1])
13
14  ▷ Calculo la subsecuencia de suma maxima que contiene al elemento A[mitad] y a A[mitad + 1]
15  max_centro ← SumaMaxInicial(A[mitad + 1, len(A) − 1]) + SumaMaxInicial(reverso(A[0, mitad]))
16  return Max(max_izq, max_der, max_centro)
```

```
SumaMaxInicial(A)
 1  suma ← 0
 2  max ← 0
 3  i ← 0
 4  while i < len(A)
 5  suma ← suma + A[i]
 6  if suma ≥ max
 7     then
 8          max ← suma
 9  i + +
10  return max
```

Análisis de complejidad

En cada paso nos quedamos con dos subproblemas que tienen la mitad de tamaño que el original, así que $a = 2$ y $c = 2$. Además de la recursión, se calcula la subsecuencia del medio. Para esto llamamos dos veces a SumaMaxInicial, recorriendo en total toda la secuencia una vez (cada llamada recorre la mitad). Esto tiene complejidad lineal en la cantidad de elementos del arreglo. Más allá de esto, se hace una cantidad constante de comparaciones y asignaciones, por lo que podemos decir que $f(n) \in \theta(n)$.
No podríamos usar el primer caso, porque $n^{\log_2 2-\varepsilon} = n^{1-\varepsilon}$, y no es cierto que $f(n) \in \theta(n^{1-\varepsilon})$. En cambio sí podemos usar el segundo caso, porque como ya dijimos $f(n) \in \theta(n)$. Reemplazando, obtenemos $T(n) \in \theta(n^{\log_2 2} * \log n)$, o sea $T(n) \in \theta(n * \log n)$.

4 Matriz Creciente

Problema

Se tiene una matriz A de n ? n números naturales, de manera que A[i, j] representa al elemento en la fila i y columna j ($1 \leq$ i, j $\leq$ n). Se sabe que el acceso a un elemento cualquiera se realiza en tiempo O(1). Se sabe también que todos los elementos de la matriz son distintos y que todas las filas y columnas de la matriz están ordenadas de forma creciente (es decir, i < n $\Rightarrow$ A[i, j] < A[i + 1, j] y j < n $\Rightarrow$ A[i, j] < A[i, j + 1]).

1. Implementar, utilizando la técnica de dividir y conquistar, la función

   está(in n: nat, in A: matriz, in e : nat) $\rightarrow$ bool

   que decide si un elemento e dado aparece en alguna parte de la matriz. Se debe dar un algoritmo que tome tiempo estrictamente menor que $\mathcal{O}(n2)$. Notar que la entrada es de tamaño $\mathcal{O}(n2)$.

2. Calcular y justificar la complejidad del algoritmo propuesto. Para simplificar el cálculo, se puede suponer que n es potencia de dos.

Idea de la resolución

En primer lugar, analicemos cómo está formada la matriz. Todas sus filas y columnas son crecientes, por lo que el mayor elemento de la matriz debe ser el elemento de la esquina inferior derecha, y el menor el de la esquina superior izquierda. No podemos, sin embargo, establecer ninguna relación entre, por ejemplo, los elementos de las otras dos esquinas.

Vamos a dividir la matriz en cuatro cuadrantes, y a concentrarnos en los elementos $A[\frac{n}{2}][\frac{n}{2}]$ y $A[\frac{n}{2}+1][\frac{n}{2}+1]$.

Veamos en principio que estos cuatro cuadrantes mantienen la propiedad de ser matrices crecientes, por lo que A$[\frac{n}{2}][\frac{n}{2}]$ va a ser el mayor elemento del cuadrante superior izquierdo y A$[\frac{n}{2}+1][\frac{n}{2}+1]$ el menor del cuadrante inferior derecho.

Llamaremos e al elemento que buscamos. Pueden darse dos situaciones:

- e $\leq$ A$[\frac{n}{2}][\frac{n}{2}]$

  Veamos que, si este es el caso, entonces claramente $e <$ A$[\frac{n}{2}+1][\frac{n}{2}+1]$, porque A$[\frac{n}{2}][\frac{n}{2}] <$ A$[\frac{n}{2}+1][\frac{n}{2}+1]$, ya que está en la columna y fila siguiente (y todos los elementos son distintos). Como habíamos dicho que ese elemento era el mínimo del cuadrante inferior derecho, concluímos que e no puede estar ahí ya que son todos mayores que él. Por lo tanto, llamamos a la función en los otros tres cuadrantes, viendo si e está en alguno de estos.

- e > A$[\frac{n}{2}][\frac{n}{2}]$

  En este caso e no puede estar en el cuadrante superior izquierdo, porque, como A$[\frac{n}{2}][\frac{n}{2}]$ es el máximo en ese sector, e es mayor a todos ellos. Por lo tanto, llamamos a la función en los otros tres cuadrantes.

Lo único que falta analizar es cuál es el caso base, que claramente se da cuando la matriz tiene un sólo elemento, y lo único que hay que hacer es fijarnos si es el que buscamos.

Resolución

```
MatrizCreciente(A, e, n)
1  return MatrizCrecienteDC(A, e, 0, n, 0, n)
```

```
MatrizCrecienteDC(A, e, x_1, x_2, y_1, y_2)
1  if y_1 == y_2 and x_1 == x_2
2     then
3          return A[x_1][y_1]
4  mitad_x ← (x_1+x_2)/2
5  mitad_y ← (y_1+y_2)/2
6  if e leq A[mitad_x][mitad_y]
7     then
8          return MatrizCrecienteDC(A, e, x_1, mitad_x, y_1, mitad_y) or
   MatrizCrecienteDC(A, e, mitad_x, x_2, y_1, mitad_y )or MatrizCrecienteDC(A, e, x_1, mitad_x, mitad_y, y_2)
9  return MatrizCrecienteDC(A, e, mitad_x, x_2, mitad_y, y_2) or
   MatrizCrecienteDC(A, e, mitad_x, x_2, y_1, mitad_y )or MatrizCrecienteDC(A, e, x_1, mitad_x, mitad_y, y_2)
```

Análisis de Complejidad

Calcularemos la complejidad en base a m, el tamaño total de la matriz. Si n es la dimensión de las filas y columnas, entonces $m = n^2$. En cada paso nos quedamos con tres subproblemas que tienen $\frac{1}{4}$ tamaño que el original. Por lo tanto, $a = 3$ y $c = 4$. Además de la recursión, hacemos sólamente comparaciones, por lo tanto $f(n) \in \mathcal{O}(1)$. Veamos si podemos utilizar el primer caso del Teorema Maestro. Para esto necesitamos que $f(n) \in \Theta(n^{log_4 3-\epsilon})$, con $\epsilon > 0$. Podemos ver que $\log_4 3$ es un número entre 0 y 1. Entonces, si elegimos, por ejemplo, $\epsilon = log_4 3$, estamos cumpliendo con el requisito, ya que $\Theta(n^{\log_4 3-\epsilon})$ sería $\Theta(n^0) = \Theta(1)$ y sabemos que $f(n)$ tiene esa complejidad. Por lo tanto, utilizando el primer caso obtenemos T(n) $\in \Theta(m^{\log_4 3})$, lo cual es claramente estrictamente menor que $\Theta(m)$, y por lo tanto menor que $\Theta(n^2)$, cumpliendo con la complejidad pedida.

5 Camino Máximo en Árbol

Problema

Dado un árbol binario de números enteros, se desea calcular la máxima suma de los nodos pertenecientes a un camino entre dos nodos cualesquiera del árbol. Un camino entre dos nodos n_1 y n_2 está formado por todos los nodos que hay que atravesar en el árbol para llegar desde n_1 hasta n_2, incluyéndolos a ambos. Un camino entre un nodo y sí mismo está formado únicamente por ese nodo. Suponer que el árbol está balanceado.

Se pide dar un algoritmo MáximaSumaCamino(a : ab(int)) → int que resuelva el problema utilizando la técnica de *Dividir y Conquistar*, calculando y justificando claramente su complejidad.

Idea de la resolución 1

Análogamente al ejercicio sobre arreglos, pueden suceder 3 cosas:

1. Que el camino máximo esté en el subarbol izquierdo.

2. Que el camino máximo esté en el subarbol derecho.

3. Que el camino máximo esté parte en el subarbol izquierdo, y parte en el subarbol derecho, teniendo que pasar por la raíz.

Sabiendo ésto, nuestro algoritmo va a buscar en cada subarbol el camino máximo, y además va a construir uno alternativo que pase por ambos subárboles. Además tenemos que tener cuidado con los números negativos, y tenemos que tener en cuenta que el camino más chico posible es de un nodo contra sí mismo, o sea, el vacío no es un camino válido.

Resolución

MaxCamino(*a: ab(int)*)

```
1  if nil?(a)
2     then
3          return −∞
4  mc_i ← MaxCamino(izq(a))
5  mc_d ← MaxCamino(der(a))
6  mcdr_i ← MaxCaminoDesdeRaiz(izq(a))
7  mcdr_d ← MaxCaminoDesdeRaiz(der(a))
8  mc_r ← max(raiz(a), raiz(a) + mcdr_i, raiz(a) + mcdr_d, raiz(a) + mcdr_i + mcdr_d)
9  return max(mc_i, mc_d, mc_r)
```

MaxCaminoDesdeRaiz(*a: ab(int)*)

```
1  if nil?(a)
2     then
3          return −∞
4  mcdr_i ← MaxCaminoDesdeRaiz(izq(a))
5  mcdr_d ← MaxCaminoDesdeRaiz(der(a))
6  return max(raiz(a) + mcdr_i, raiz(a) + mcdr_d, raiz(a))
```

Idea de la resolución 2

La solución 1 anda bien, pero podríamos hacer algo mejor, para evitar recorrer un nodo del árbol más de una vez. Nos gustaría evitar tener que volver a recorrer el árbol para poder armar ese camino que toma parte en el subarbol izquierdo y en el derecho, para lograr ésto vamos a hacer que nuestra función nos devuelva una tupla, cuyo primer elemento sea el árbol óptimo que encontró, y la segunda componente sea el mejor camino que empieza en la raíz. Teniendo esa última componente, armar un camino que pase por el subarbol izquierdo, el derecho, y la raíz va a ser fácil.

Resolución

MaxCamino(*a: ab(int)*)

```
1  if nil?(a)
2      then
3           return (−∞, −∞)
4  opt_i, craiz_i ← MaxCamino(izq(a))
5  opt_d, craiz_d ← MaxCamino(der(a))
6  craiz_r ← max(raiz(a), raiz(a) + craiz_i, raiz(a) + craiz_d)
7  opt_r ← max(opt_i, opt_d, craiz_d, raiz(a) + craiz_i + craiz_d)
8  return (opt_r, craiz_r)
```