Universidad Nacional Autónoma de México

Facultad de Ciencias

Análisis y Diseño de Algoritmos

Tarea1

Reynoso Sánchez Arturo Yitzack
Martínez Avila Santiago

8 de octubre de 2021

1

Considera el problema MIN: dado un arreglo A de n elementos comparables, devolver el valor del elemento más pequeño de A.

1. Diseña un algoritmo iterativo que resuelva el problema MIN en tiempo $O(n)$. Demuestra usando invariantes que es correcto. Demuestra que su complejidad es realmente $O(n)$.

   Nota: $A[0]$ es el primer elemento del arreglo A, y si $0 \leq i \leq j \leq n$, A[i:j] incluye a los elementos A[i], A[i+1],...,A[j-1], A[j].

Algorithm 1 MIN algorithm

```
1: procedure MIN(A)                          ▷ The minimum element of A array
2:     int x ← A[0]                          ▷ O(1)
3:     for j ← 1 to n − 1 do                 ▷ O(n)
4:         if A[j] < x then                  ▷ O(1)
5:             x ← A[j]                      ▷ O(1)
6:         end if
7:     end for
8:     return x                              ▷ O(1)
9: end procedure
```

   Vamos a demostrar que el algoritmo propuesto es correcto. Probaremos el siguiente

   invariante: al inicio de cada iteración j, la variable x es el elemento mínimo de $A[0 : j-1]$.

   - *Inicialización*: Antes de entrar a la primera iteración, j=1 y $x = A[0]$. Además, tenemos que A[0:j-1] = A[0:0]= A=[0] = x. por lo tanto, x es el elemento mínimo de A[0:0].
   - *Mantenimiento*: Al inicio de la j-ésima iteración, tenemos que x es el elemento mínimo de $A[0 : j-1]$. Ahora comparamos x con $A[j]$. Si $A[j]$ es menor que x, actualizamos x a $A[j]$ y, por lo tanto, al inicio de la $j+1$ ésima iteración, x es el mínimo elemento del subarreglo A[0:j].
   - *Terminación*: La condición para que el **for** loop termine es que $j > n$. Cada iteración aumenta el valor de j en una unidad, entonces tenemos $j = n+1$. Substituyendo $n+1$ en el invariante propuesto, tenemos que x es el elemento mínimo del arreglo $A[0 : n]$. Observemos que el arreglo $A[0 : n]$ es el arreglo A, de modo que concluimos que el algoritmo es correcto.

   Ahora analizamos su complejidad. Sea $T(n)$ el número de pasos que ejecuta este algoritmo. Entonces tenemos que:

$$\begin{aligned} T(n) &= O(1) + O(n) * (O(1) + O(1)) + O(1) \\ &= O(n) \end{aligned} \tag{1}$$

2. Diseña un algoritmo recursivo que resuelva el problema MIN en tiempo $O(n)$. Demuestra por inducción en el valor n que es correcto. Demuestra que su complejidad es realmente $O(n)$.

   Vamos a demostrar por inducción sobre el número de elementos del arreglo ($len(A)$) que el algoritmo recursivo es correcto.

Algorithm 2 MIN algorithm (recursivo)

```
1: procedure MIN(A)
2:     if len(A) = 1 then                                  ▷ O(1)
3:         return A[0]                                     ▷ O(1)
4:     end if
5:     if A[−2] > A[−1] then                               ▷ O(1)
6:         A.pop(−2)                                       ▷ O(1)
7:     else
8:         A.pop()                                         ▷ O(1)
9:     end if
10:    return MIN(A)                                       ▷
11: end procedure
```

Base Como $len(A) = 1$, el algoritmo devuelve $A[0]$, que es el único elemento del arreglo y, por lo tanto, es el elemento más pequeño.
H.I. Supongamos que el algoritmo es correcto para todo arreglo de k elementos
P.I. Sea A un arreglo de $k+1$ elementos. Consideremos A' el arreglo formado por todos los elementos de A menos el primero de estos. Como A' tiene k elementos, por H.I., $MIN(A')$ es correcto, lo que implica que devuelve el elemento más pequeño del arreglo, llamémoslo a'. Notemos que $a' \leq A'[i]$ para toda $0 \leq i \leq k-1$ y ya que todos los elementos de A' son elementos de A, $a' \leq A[i]$ para toda $1 \leq i \leq k$. Ahora consideremos el arreglo $B = [A[0], a']$ Como $len(B) \neq 1$, al aplicar $MIN(B)$, tenemos dos casos:

- Caso 1. Si $A[0] > a'$, entonces $B = [a']$ y se vuelve a ejecutar $MIN(B)$ y como $len(B) = 1$, el algoritmo devuelve a'. Y como a' ya era menor que $A[i]$ para toda $1 \leq i \leq k$ y $A[0] > a'$, concluimos que $a' \leq A[i]$ para toda $0 \leq i \leq k$.
- Caso 2. Si $A[0] \leq a'$, entonces $B = [A[0]]$ y se vuelve a ejecutar $MIN(B)$ y como $len(B) = 1$, el algoritmo devuelve $A[0]$. Y como a' ya era menor que $A[i]$ para toda $1 \leq i \leq k$ y $A[0] \leq a'$, concluimos que $A[0] \leq A[i]$ para toda $0 \leq i \leq k$.

De ambos casos, el algoritmo devuelve el elemento más pequeño del arreglo. ■

Ahora analizamos su complejidad. Llamemos $n = len(A)$. Si $n = 1$, se realizan 2 pasos. Si $n = 2$, se realizan $4 + 2$ pasos. Si $n = 3$, se realizan $4 + 4 + 2$ pasos. Si $n = 4$, se realizan $4 + 4 + 4 + 2$ pasos. Es fácil ver que para $n = k$, se realizan $4(k-1) + 2$ pasos. Por lo tanto, la complejidad del algoritmo es $O(4(n-1) + 2) = 0(4n - 4 + 2) = O(n)$

3. Diseña un algoritmo que en tiempo lineal calcule la profundidad de un árbol binario T dado. La profundidad de un árbol se define como el máximo número de aristas que separan la raíz con alguna hoja del árbol. Demuestra que el algoritmo es correcto, y que su complejidad es lineal en el número de vértices y aristas en T.

   Nota: Un nodo es un vértice. Cada nodo tiene un nodo hijo izquierdo (nodo.left) o un nodo hijo derecho (nodo.right) (que pueden ser *null*) y un nodo padre, que en el caso de la raíz su nodo padre es *null*. La altura de un nodo es el máximo número de aristas en un camino de un nodo hoja al nodo. La profundidad de un árbol coincide con la altura del nodo raíz.

Algorithm 3 ComputeHeight algorithm (recursivo)

```
1:  procedure ComputeHeight(nodo)
2:      if nodo.left = null and nodo.right = null then          ▷ O(1)
3:          return 0                                            ▷ O(1)
4:      end if
5:      hl ← ComputeDepth(nodo.left)
6:      hr ← ComputeDepth(nodo.right)
7:      if hl > hr then
8:          nodo.height ← hl + 1
9:      else
10:         nodo.height ← hr + 1
11:     end if
12:     return nodo.height
13: end procedure
```

Vamos a demostrar que el algoritmo es correcto usando inducción fuerte sobre el la altura h del árbol.

Caso base: $h = 0$. Si el nodo raíz tiene altura 0, en este caso no tiene hijos, como no tiene hijos es un nodo hoja y el algoritmo regresa cero (línea 3).

Hipótesis de inducción: Suponemos que el algoritmo es correcto para todos los árboles de altura k, $0 \leq k < h$ para alguna $h > 0$.

Sea x el nodo raíz de un árbol T_x con altura h. Sean T_l y T_r los árboles con raíz $x.left$ y $x.right$ respectivamente. Por definición: $altura(T_x) = max(altura(T_l), altura(T_r)) + 1$ (bloque del **if**, líneas 7-11). Entonces la altura de T_x es mayor a cero, y las alturas de T_r y T_l son estrictamente menores que T_l, de modo que, por la hipótesis de inducción, el algoritmo regresa de manera correcta las alturas de T_l y T_r. El algoritmo evalúa el valor máximo de las alturas de T_l y T_r, le agrega 1, y devuelve el valor. Por lo tanto, el algoritmo devuelve de manera correcta la altura del árbol T_x.

Como la profundidad de un árbol es igual a la altura del nodo raíz, el algoritmo `ComputeHeight()` con entrada el nodo raíz nos da la profundidad del árbol.

Ahora evaluamos su **complejidad**. Sea $T(n)$ el tiempo de ejecución de un árbol con n nodos. Cuando el algoritmo es llamado en un nodo hoja, regresa 0 (la tercera línea). Entonces toma un tiempo constante, digamos $T(1) = c$ con $c > 0$. Supongamos que el algoritmo es llamado en un nodo x con n nodos en el subárbol que tiene como raíz a x. Entonces el algoritmo toma un tiempo $d > 0$ constante, y luego es llamado recursivamente en los nodos hijos de x. Sea k el número de nodos del árbol con raiz en el hijo izquierdo de x, $x.left$. Entonces tenemos la siguiente ecuación de recurrencia:

$$T(n) = T(k) + T(n-k-1) + d$$

Vamos a suponer por inducción que para $1 \leq k < n$, $T(k) \in O(k)$. Sea $n \in \mathbb{N}$

Caso base. $T(1) = c$ y entonces $T(1) \in O(1)$.

Hipótesis de inducción. Para $1 \leq k < n$, $T(k) \in O(k)$

Entonces tenemos:

$$\begin{aligned} T(n) &= T(k) + T(n-k-1) + d \\ &\leq ck + c(n-k-1) + d \text{ (paso inductivo)} \\ &\leq c^*k + c^*(n-k-1) + c^* \text{ (con } c^* \geq d) \\ &= c^*(k+n-k-1+1) \\ &= c^*n \\ &= O(n) \end{aligned}$$

4. Diseña un algoritmo que, dado un arreglo A de n enteros, y un entero x, devuelva el índice más pequeño tal que $A[i] == x$, o el valor -1 si x no aparece en el arreglo. Propón el invariante natural de tu algoritmo, demuéstralo, y úsalo para concluir que el algoritmo es correcto. Demuestra por qué es de tiempo $O(n)$.

Algorithm 4

```
1: procedure F(A,x)
2:     c = −1                                   ▷ O(1)
3:     for i ← 0 to n − 1 do                    ▷ O(n)
4:         if A[i] == x then                    ▷ O(1)
5:             c = i                            ▷ O(1)
6:             break                            ▷ O(1)
7:         end if
8:     end for
9:     return c                                 ▷ O(1)
10: end procedure
```

Vamos a demostrar que el algoritmo propuesto es correcto. Probaremos el siguiente

invariante: al inicio de cada iteración i, $c = -1$.

- *Inicialización*: Antes de entrar a la primera iteración, se definió que $c = -1$.
- *Mantenimiento*: Al inicio de la i-ésima iteración, tenemos que $c = -1$ pues la única forma de que sea distinto es que en un paso anterior, digamos j con $j < i$, $A[j] == x$ en cuyo caso el algoritmo se hubiera terminado y al ser $j < i$ no sería posible llegar al paso i.
- *Terminación*: Hay dos formas de que el **for** loop termine. La condición "natural" es que $i > n-1$. Como el algoritmo no se terminó antes de llegar a $i = n$, c sigue siendo -1, entonces no hay elemento del arreglo que sea igual a x. La otra condición para que el **for** loop termine es que haya algún elemento del arreglo que sea igual a x, en cuyo caso c toma el valor del índice de dicho elemento. Como el loop recorre los elementos

del arreglo, desde el elemento 0 hasta el elemento $n-1$, es decir, en orden ascendente, y el loop se termina cuando encuentra dicho elemento, el número que regresa es el del índice más pequeño que cumple dicha propiedad. Por lo tanto, podemos concluir que el algoritmo es correcto.

Ahora analizamos su complejidad. Sea $T(n)$ el número de pasos que ejecuta este algoritmo. Entonces tenemos que:

$$\begin{aligned} T(n) &= O(1) + O(n) * (O(1) + O(1) + O(1)) + O(1) \\ &= O(n) \end{aligned} \tag{2}$$

Ejercicio opcional. Demuestra el siguiente lema (sugerimos usar inducción): Todo árbol de al menos dos vértices, tiene al menos dos hojas (una hoja es un vértice de grado 1).
Con el uso de ese lema, diseña un algoritmo que, dado un árbol de n vértices, coloree todos los vértices del árbol con colores 0 y 1, de tal forma que dos vértices adyacentes tengan colores distintos. Demuestra que el algoritmo es correcto. En este ejercicio no hace falta demostrar complejidad. Expresa el algoritmo en el mayor nivel de abstracción posible: no tepreocupes por la representación en estructura de datos del árbol.

Para demostrar el Lema, usaremos que todo árbol tiene al menos una hoja.
Demostración
Supongamos que existe un árbol que no tiene hojas. Es decir, que el grado de todos sus vértices es mayor o igual a 2. Esto implicaría que existe al menos un ciclo en dicho árbol, lo cual es una contradicción pues un árbol no tiene ciclos por definición.

Demostración del Lema

Sea T un árbol con n vértices.

- n=2 Como es un árbol, debe haber una arista entre ambos vértices. Por lo que el grado de ambos vértices es 1 y así ambos son hojas.
- H.I. Supongamos que todo árbol de k vértices, tiene al menos dos hojas.
- P.I. Supongamos que T tiene $k+1$ vértices. Como T es árbol, tiene al menos una hoja, llamémosla x. Como x es hoja, solamente existe un vértice y tal que x y y son adyacentes. Ahora consideremos la gráfica $T-x$, que, al ser x hoja, sigue siendo un árbol y tiene k vértices. Por H.I., $T-x$ tiene al menos dos hojas. Por lo que, al considerar nuevamente a T, tenemos dos casos:
  - Que y sea una hoja de $T-x$ en cuyo caso T tiene de hojas al menos a x y a la otra hoja de $T-x$
  - Que y no sea una hoja de $T-x$ en cuyo caso T tiene de hojas al menos a x y a las hojas de $T-x$

  De ambos casos, T tiene al menos dos hojas ■

En un árbol, dado un nodo (vértice) v, tiene un nodo padre (para la raíz es *null*), y nodos hijos. Sea v.C el arreglo que contiene a los nodos hijos de v.

Algorithm 5 Coloring algorithm (recursivo)

```
1: procedure ColorNodes(nodo)
2:     if nodo = root then
3:         nodo.color = 0
4:     else
5:         nodo.color = 1 - nodo.parent.color
6:     end if
7:     if nodo.C = null then
8:         return
9:     else
10:        for i ← 0 to nodo.C.length − 1 do
11:            ColorNodes(nodo.C[i])
12:        end for
13:    end if
14: end procedure
```

Vamos a demostrar que el algoritmo es correcto pero para ello usamos la siguiente definición. El nivel i-ésimo de un árbol lo definimos como el máximo número de aristas que hay entre el nodo y la raíz. Un árbol tiene entonces niveles 0, 1, ..., hasta su profundidad.
Comenzando el algoritmo en raíz, tenemos que colorea primero al nivel cero, luego al nivel 1 y así sucesivamente hasta el último nivel. Esto lo podemos observar de la siguiente forma:
En nivel 0, tenemos al nodo raíz, y le asignamos color 0. Si tiene hijos, llama al método en sus nodos hijos. Estos nodos hijos "saben" el color de su padre (el nodo raíz) y por tanto, el algoritmo asigna un color distinto (línea 5). Luego, se llama el algoritmo a los nodos "nietos". Haciendo lo mismo con los nodos "nietos" ellos conocen el color de los nodos hijos originales y les es asignado un color diferente.
Ahora bien, dado un nodo que no es raíz, conoce el color de su nodo padre y es asignado el color distinto. Luego el algoritmo evalúa si tiene hijos o no. Si no los tiene, el algoritmo ya no es llamado. Si los tiene, se llama el algoritmo a cada uno de sus hijos. En algún momento el algoritmo será llamado en las hojas y como no tiene hijos, el algoritmo ya no será llamado nuevamente.
Así nodos padres y nodos hijos tienen distinto color, es decir, nodos en niveles consecutivos tienen distinto color.
El algoritmo no regresa nada, sólo cambia la propiedad color de los nodos.