Universidad Nacional Autónoma de México

Facultad de Ciencias

Análisis y Diseño de Algoritmos

Tarea 3

Martínez Avila Santiago

Reynoso Sánchez Arturo Yitzack

12 de noviembre de 2021

1

Peso: 10 puntos. Considera el problema *búsqueda en arreglo decreciente*. Entrada: un arreglo $A : [a_1, ..., a_n]$ con la propiedad de estar ordenado de forma dereciente (para todo par de índices distintos $i < j$, $A[i] \geq A[j]$), y un valor x. Salida: el menor índice z de A, tal que $A[z] \leq x$, o el tamaño de A si semejante índice no existe.

Ejemplos: Entrada: [8, 5, 5, 2, 1], 4. Salida: 3.

Entrada: [8, 5, 5, 2, 1], 0. Salida: 5.

Propón un algoritmo de complejidad $O(\log n)$ que resuelva el problema (3 puntos).

Demuestra su corrección. (3 pts)

Demuestra su complejidad. (3 pts)

Todo lo anterior, de forma clara y concisa. (1 pt)

Ejercicio extra: Peso, 3pts. Si tu algoritmo anterior fue recursivo, da la solución iterativa del problema, y viceversa. Esperamos que la solución de la versión recursiva la demuestres por inducción en el número de elementos del arreglo, y la complejidad analizando el árbol de recursión y el costo de cada subinstancia; y que la corrección del algoritmo iterativo sea mediante invariantes demostrados por inducción, y la complejidad sea mediante el análisis del número de iteraciones y el costo de cada iteración.

ALGORITMO ITERATIVO
Correctitud
Vamos a demostrar que el algoritmo iterativo propuesto es correcto. Probaremos el siguiente:

invariante: Si existe el menor índice z de A tal que $A[z] \leq x$, entonces $z \in [a, b]$ al inicio de cada iteración j.

- *Inicialización*: Antes de entrar a la primera iteración, $a = 0$ y $b = len(A) - 1 \implies z \in [a, b]$ pues z es índice de A.
- *Mantenimiento*: Supongamos que al inicio de la j-ésima iteración, tenemos que $z \in [a, b]$. Para evitar confusiones a' y b' son las variables al inicio de la $(j + 1)$-ésima iteración a a y b son las variables al inicio de la j-ésima iteración.
  - Caso 1: $A[c] \leq x$
    Como cuando $A[c - 1] > x$ el algoritmo termina, sólo consideramos $A[c - 1] \leq x$
    Por el algoritmo tenemos que $a' = a$ y $b' = c$
    Como z es el mínimo índice, $z \leq c = b'$
    Y como $z \in [a, b]$ por hipótesis, $a' = a \leq z$
    $\therefore z \in [a', b']$

Algorithm 1 Iterativo

```
1: procedure MinIt(A, x)
2:     int a ← 0
3:     int b ← len(A)
4:     if A[b − 1] > x then
5:         return len(A)
6:     end if
7:     while a < b do
8:         c ← (a + b)//2
9:         if A[c] ≤ x then
10:            if A[c − 1] > x then
11:                return c
12:            else
13:                b ← c
14:            end if
15:        else
16:            a ← c + 1
17:        end if
18:    end while
19:    return (a + b)//2
20: end procedure
```

- Caso 2: $A[c] > x$
  Por el algoritmo tenemos que $a' = c + 1$ y $b' = b$
  Como $z \in [a, b]$ por hipótesis, $z \leq b = b'$
  Como z es el mínimo índice y$A[c] > x$, $c < z \implies a' = c + 1 \leq z$
  $\therefore z \in [a', b']$

De ambos casos, $z \in [a, b]$ al inicio de la $(j + 1)$-ésima iteración.

- *Terminación*: Como ya vimos que $z \in [a, b]$ al inicio de cada iteración, en particular para la última iteración k y la prenúltima $k - 1$. Para evitar confusiones para el inicio de la iteración $k - 1$, a y b son las variables; para el inicio de la iteración k, a' y b' son las variables; y para el final del algoritmo, a'' y b'' son las variables.

  - Caso 1: $A[c] \leq x$ y $A[c] > x$
    Es evidente que c es el primer índice tal que $A[c] \leq x$. Por lo tanto $z = c$
    Y notemos que $c = (a' + b')//2 \implies a' \leq c < b'$
    Como en la iteración k no se asignan nuevos valores ni a a' ni a b', $a'' = a'$ y $b'' = b' \implies a'' \leq c \leq b''$
    $\therefore z \in [a'', b'']$
    Y como el algoritmo regresa c, que es igual a z, es correcto.
  - Caso 2: $a' \geq b'$
    Para saber qué implica este caso, necesitamos saber cómo son a' y b', esto es,

tenemos que analizar la iteración $k-1$.

* Caso 2.1: $A[c] \leq x$
  Como cuando $A[c-1] > x$ el algoritmo termina, sólo consideramos $A[c-1] \leq x$
  Por el algoritmo tenemos que $a' = a$ y $b' = c$
  Y como $a' \geq b'$, $a \geq c$
  Pero como $c = (a+b)//2$, entonces $a \leq c$
  Por lo que, $a = a' = c = b'$
  Ahora supongamos que $c > 0$, esto implica que $A[c-1] \leq x$ y $a \geq c > c-1$
  Notemos que z es el mínimo índice por lo que $z \leq c-1$ y por hipótesis $z \in [a, b]$
  Pero $z \leq c-1 < a$, que es una contradicción, por lo que $a' \not\geq b'$ si $c > 0$
  De lo anterior concluimos que $c = 0$, que si cumple que $A[c-1] \leq x$ pues A es decreciente y $A[c-1] = A[-1] = A[len(A)-1]$.
  Y notemos que si $a' = 0 = b'$ entonces $(a'+b')//2 = 0$ como $A[0] \leq x$, c es el primer indíce del arreglo tal que esto pasa.
  $\therefore z = (a'+b')//2$ y $z \in [a', b']$
  Y como el algoritmo regresa $(a'+b')//2$, que es igual a z, es correcto.
* Caso 2.2: $A[c] > x$
  Por el algoritmo tenemos que $a' = c+1$ y $b' = b$
  Recordemos que $a' \geq b' \implies c+1 \geq b$
  Y como $c < b$, $c+1 \leq b$, por lo que $a' = c+1 = b = b'$
  $\implies (a'+b')//2 = c+1 \in [a', b']$
  Que por hipótesis, $z \in [a', b']$, por lo que $z = c+1 = (a'+b')//2$
  Y como el algoritmo regresa $(a'+b')//2$, es correcto.

Como en ambos casos, el algoritmo regresa z, el agoritmo es correcto.

Complejidad

Ahora analizamos su complejidad, viendo cada caso dentro del ciclo.

- Caso 1: $A[c] \leq x$
  - Caso 1.1: $A[c-1] > x$
    El ciclo termina
  - Caso 1.2: $A[c-1] \leq x$
    Porel algoritmo tenemos que $b' = c$ y $a' = a$
    $\implies b' = (a+b)//2 \leq \frac{a+b}{2}$ y $a' = a$
    $\implies b' - a' \leq \frac{a+b}{2} - a = \frac{b-a}{2}$

- Caso 2: $A[c] > x$
  Por el algoritmo tenemos que $a\bar{c}+1$ y $b' = b$

$\implies a' = (a+b)//2 + 1 \geq \frac{a+b}{2}$ y $b' = b$
$\implies b' - a' \leq b - \frac{a+b}{2} = \frac{b-a}{2}$

Por lo tanto, en ambos casos, en la siguiente iteración se considera un subarreglo de tamaño menor o igual a la mitad del tamaño del subarreglo de la iteración actual.
Y en el análisis de la correctitud, se vio que para que el ciclo termine hay dos opciones: que $a \geq b$ o que en uno de los *if's* se regrese directamente un valor. Pero es evidente que si el algoritmo termine por la segunda opción se realizó una cantidad menor o igual de pasos que con la primera opción. Por esta razón, nos centraremos en el caso en que concluye debido a que $a \geq b$.
Nuevamente del análisis de la correctitud, cuando $a \geq b$, en realidad terminabamos con $a = b$, por lo que el algoritmo termina cuando considermaos un subarreglo de tamaño 1.
Concluimos que en el algoritmo consideramos, después de cada iteración del ciclo, un subarreglo de tamaño menor o igual al tamaño del subarreglo de la iteración anterior y estocontinua hasta que el tamaño del subarreglo es 1. Esto quiere decir que el ciclo *while* tiene complejidad *O(log n)*.
Finalmente, notemos que en cada iteración hay una cantidad constante de instrucciones con complejidad *O(1)* pues consiste de asiganaciones, if's y return's. Y fuera del ciclo es una situación análoga. Por lo tanto, la complejidad de todo el algoritmo es *O(log n)*.

Algorithm 2 Recursivo

```
    procedure MinRec(A, x)                                          ▷
 2:     int n ← len(A)                                              ▷ O(1)
        if A[n − 1] > x then                                        ▷ O(1)
 4:         return n                                                ▷ O(1)
        else if A[0] ≤ x then:                                      ▷ O(1)
 6:         return 0                                                ▷ O(1)
        else:                                                       ▷ O(1)
 8:         return minRecAuxRec(A, x, 0, n − 1)                     ▷ O(1)
        end if
10: end procedure
```

Algorithm 3 Recursivo

```
    procedure minRecAux(A, x, low, high)                            ▷
 2:     int c = ((low + high)//2) + 1                               ▷ O(1)
        if A[c] ≤ x and A[c − 1] > x then                           ▷ O(1)
 4:         return c                                                ▷ O(1)
        else if A[c] ≤ x then:                                      ▷ O(1)
 6:         return minRecAux(A, x, low, c − 1)                      ▷ T(n/2)
        else:                                                       ▷ O(1)
 8:         return minRecAux(A, x, c + 1, high)                     ▷ T(n/2)
        end if
10: end procedure
```

ALGORITMO RECURSIVO

Correctitud

Vamos a demostrar que el algoritmo `minRec` es correcto. Para ello primero demostraremos que el algoritmo recursivo `minRecAux` es correcto. El propósito del algoritmo `minRecAux` es usarlo cuando sepamos que la salida correcta es un número en $\{1, 2, ..., n-1\}$.

`minRecAux` recibe un arreglo A, el valor x, y dos parámetros *low* y *high*, que serán el índice menor y el índice mayor dentro de los cuales realizar la búsqueda y donde sepamos que el índice correcto se encuentra entre *low* y *high* sin incluir a *low*, y donde $0 < i < j < n$. La prueba se hará por inducción sobre n, el tamaño del arreglo A.

Sea $P(n)$ la proposición de que el algoritmo `minRecAux()` es correcto para un tamaño $n > 1$.

Caso base n = 2. Inicializamos el algoritmo como `minRecAux(A, x, 0, 1)`. Sea $x \in \mathbb{R}$. En este caso, el arreglo A tiene dos elementos . En este punto sabemos que la salida correcta es 1, pues si fuera 0, habríamos capturado el valor 0 en la línea 5 del algoritmo `minRec`. El valor de c es 1. Como ocurre que $A[1] \leq x$ y que $A[0] > x$ (pues 1 es la salida correcta), tenemos que el algoritmo devuelve el valor de c, 1. Por lo tanto, el algoritmo es correcto.

Hipótesis de inducción. Supongamos que el algoritmo `minRecAux()` es correcto para arreglos de tamaño menores a $n \in \mathbb{N}$, con $n > 2$. Ahora hay que demostrar que el algoritmo es correcto para arreglos de tamaño m, con $2 \leq m < n$.

.

Paso inductivo. Sea *high* - *low* = n. Entonces la búsqueda se realiza en un subarreglo de A de tamaño n. La variable c será el resultado de aplicar la función techo a $(low + high)/2$. En particular $c > 0$ y garantizamos que $A[c-1]$ existe. Entonces surgen tres casos:

1. $A[c] \leq x$ y $A[c-1] > x$. Nos indica que c es el primer índice tal que A[c] es menor o igual a x. Entonces el algoritmo nos devuelve c, que es correcto.

2. $A[c] \leq x$. Si evaluamos este caso, entonces estamos evaluando $A[c] \leq x$ y $A[c-1] > x$ (ya que el primero fue falso). Si es correcto, nos indica que c no fue el primer índice menor o igual a x, de modo que el índice correcto debe estar en {low ,low +1,...,c-1}. De modo que debemos buscar en el subarreglo $A[low : (c-1)]$. Entonces retornamos el valor de `minRecAux(A, x, low, c-1)`. Por hpótesis de inducción, esta llamada recursiva se hace sobre un subarreglo de A con longitud menor a n, ya que si $n > 2$, entonces c es estrictamente menor a *high*. Por lo tanto, la llamada recursiva devuelve el valor correcto.

3. $A[c] > x$. en este caso el índice buscado es mayor a c. Por lo tanto, llamamos a `minRecAux(A, x, c+1, high)`, buscamos en la parte del subarreglo con índices {c+1, c+2,..., high}. Por la hipótesis de inducción, este subarreglo tiene longitud menor a n, por lo que devuelve el índice correcto.

Por lo tanto, cuando sabemos que el índice correcto está en {1,2,..., n-1} el algoritmo `minRecAux()` devuelve el valor correcto.

Ahora bien, cuando x es mayor o igual al elemento mayor en el arreglo (que es $A[0]$) o cuando x es menor a todos los elementos de A, podemos devolver una dar en el algoritmo recursivo `minRecAux()` cuando de la siguiente manera:

1. el primer elemento de A es menor o igual que x, simplemente devolvemos 0 (línea 6).
2. el último elemento de A es mayor que x, devolvemos n (línea 4).

.

En otro caso, entonces la salida corresponde a alguno de los índices {1,2,..,n-1} y entonces ocupamos el algoritmo recursivo `minRecAux()` que ya demostramos que es correcto. Por lo tanto, el algoritmo `minRec()` es correcto.

Complejidad

Para demostrar su complejidad, notamos que las líneas de código del algoritmo `minRec()` antes de entrar al algoritmo recursivo `minRecAux()` tienen complejidad constante O(1). El algoritmo `minRecAux()` es llamado con las entradas `minRecAux(A, x, 0, n-1)`. En complejidad en tiempo, el peor caso se da cuando entramos al algoritmo recursivo auxiliar y lo llamamos con un arreglo con dos elementos, i.e. cuando llegamos al caso base.

`minRecAux()` divide el espacio de búsqueda en dos partes de tamaño igual (o que difieren en 1) en cada llamada recursiva, hasta llegar en el peor caso al caso base con un arreglo de 2 elementos. Dado un arreglo de n elementos,el árbol de recursión del algoritmo `minRecAux()` (y por consiguiente `minRec()`)tiene profundidad $\log_2(n)$. En cada nivel, realizamos un número constante de comparaciones O(1). Entonces el tiempo total que toma es O(1) multiplicado por $\log_2(n)$ niveles, y como $\log_2$ y log difieren en una constante, entonces tenemos que el tiempo total $T(n)$ que toma el algoritmo `minRec()` es $O(\log n)$.