UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Tarea 1

Análisis de algoritmos

Rivera Morales David
320176876

Gallegos Cortes José Antonio
320316566

February 17, 2025

1 Ejercicio 1

Dado un entero positivo n, determinar el valor de $\lfloor \log(n) \rfloor$.

Solución

Algoritmo

Algorithm 1 Calcular $\lfloor \log(n) \rfloor$

Require: Un entero positivo n
Ensure: El valor de $\lfloor \log(n) \rfloor$

```
1: count ← 0
2: if n ≤ 1 then
3:     return 0
4: end if
5: while n > 1 do
6:     n ← n/2
7:     count ← count + 1
8: end while
9: return count
```

Análisis de complejidad

1. $count \leftarrow 0$ tiene 1 Asignación, tiempo constante
2. if $n \leq 1$ tiene 1 salto, 1 lectura (lee a n), 1 comparación (compara n con 1)
3. return 0 tiene 1 salto
4. endif tiene 1 salto
5. while $n > 1$ do tiene 1 salto, 1 lectura (lee a n) y 1 comparación (compara n con 1)
6. $n \leftarrow n/2$ tiene 1 lectura (lee n), 1 operación aritmetica (la división) y finalmente 1 escritura (escribe a n)
7. $count \leftarrow count + 1$ tiene 1 lectura (count), 1 suma y 1 escritura.
8. end while tiene 1 salto
9. return count tiene un salto y una lectura.

En tiempo si hacemos un analisis "básico" en una sola operación vemos que el 1 **se hace siempre** (es constante), el if puede o no hacerse por lo que a todo el if (es decir a 2,3 y 4) le pondremos **3** y a su return $\frac{1}{2}$, el while en una sola iteración tiene $3 + 3 + 3 + 1 = 10$ operaciones, y finalmente el return tiene 2 operaciones, pero como este return depende de que no se haya hecho el primer return tambien le pondremos $\frac{1}{2}$, por lo que ese return tiene

realmente **1 y media operaciones.**
Si contamos todo en una sola iteracion tenemos que la linea 1 se va a hacer siempre (1 operación), al igual que el if (ya que se va a comprobar si se cumple o no, tenemos 3 operaciones) y uno de los dos return se va a cumplir tambien (1 operación) de ambos return y 1 lectura del "count" final, por lo que al final tenemos 6 operaciones que se hacen si o si, ahora veamos que pasa con el while.

Complejidad Temporal: El algoritmo realiza divisiones sucesivas entre 2 hasta llegar a 1:

- En cada iteración del ciclo **while**, la variable n se divide entre 2
- El número de iteraciones es aproximadamente $\log_2(n)$
- Cada iteración tiene un costo constante $O(1)$

Por lo tanto, la complejidad temporal total es:

$$O(\log n) + 6$$

Pero como el logaritmo es más grande podemos quitarle la suma, por lo que queda simplemente como $O(\log n)$

Complejidad Espacial: El algoritmo utiliza un número constante de variables adicionales (*count* y n). Por ello, la complejidad espacial es:

$$O(1)$$

En resumen, la complejidad temporal del algoritmo es $O(\log n)$ y la complejidad espacial es $O(1)$.

2 Ejercicio 2

Dado un arreglo A de n enteros y un entero objetivo K, ¿existen un par de índices $i \neq j$, tales que $A[i] + A[j] = K$?

Solución

Algoritmo

Algorithm 2 Encontrar par de números que suman K

Require: Un arreglo A de n enteros y un entero objetivo K
Ensure: Verdadero si existe un par de índices $i \neq j$ tales que $A[i] + A[j] = K$

```
1: hashMap ← {}
2: for i ← 0 to n − 1 do
3:    if hashMap.containsKey(K − A[i]) then
4:       return true
5:    end if
6:    hashMap.put(A[i], i)
7: end for
8: return false
```

Análisis de complejidad

Código	Operaciones	Total
$hashMap \leftarrow \{\}$	1 asignación.	1
for($i \leftarrow 0$ to $n-1$)	1 salto (for), 1 escritura (i), 1 lectura (n), 1 resta	4
if $hashMap.containsKey(K-A[i])$	1 salto, 3 lecturas (A, i, K), 1 resta y una operación hash	5
Return True	1 salto	1
end if	1 salto	1
$hashMap.put(A[i], i)$	2 lecturas (A, i) y la operación del hash	3
end for	1 salto	1
return false	1 salto	1

Las operaciones hash según la https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.htmldocume de Java son constantes. Las que se hacen si o si son la asignación del Hash, el for y el return false, es decir 1+4+1 = 6, mientras que en una sola iteración hace 17 operaciones, ahora para el for ocurre lo siguiente:

Complejidad Temporal: El algoritmo realiza las siguientes operaciones:

- Recorre el arreglo una sola vez, visitando sus n elementos
- Para cada elemento realiza operaciones de hash (búsqueda e inserción) que son $O(1)$ en promedio

Por lo tanto, la complejidad temporal total es:

$$O(17n + 6)$$

Pero nuevamente, se pueden quitar las constantes por lo que nos queda solamente $O(n)$

Complejidad Espacial: El algoritmo utiliza un `hashMap` que puede almacenar hasta n elementos. Por ello, la complejidad espacial es:

$$O(n)$$

3 Ejercicio 3

Dado un entero positivo n, determinar la cantidad de números primos menores o iguales a n.

Solución

Algoritmo

Algorithm 3 Contar números primos usando la Criba de Eratóstenes

Require: Un entero positivo n
Ensure: La cantidad de números primos menores o iguales a n

```
esPrimo ← [true] * (n + 1) {Arreglo booleano inicializado en verdadero}
esPrimo[0] ← false
esPrimo[1] ← false
contador ← 0
for i ← 2 to √n do
    if esPrimo[i] then
        for j ← i² to n step i do
            esPrimo[j] ← false
        end for
    end if
end for
for i ← 2 to n do
    if esPrimo[i] then
        contador ← contador + 1
    end if
end for
return contador
```

Análisis de complejidad

Tenemos la siguiente tabla

Código	Operaciones	Total
$esPrimo \leftarrow [true] * (n+1)$	1 asignación + 1 multiplicacion + 1 lectura (n) + 1 suma	4
$esPrimo[0] \leftarrow false$	1 escritura	1
$esPrimo[1] \leftarrow false$	1 escritura	1
$contador \leftarrow 0$	1 escritura	1
for ($i \leftarrow 2$ to $\sqrt{n}$)	1 salto, 1 escritura (i), 1 lectura (n), 1 raíz cuadrada	4
if $esPrimo[i]$	1 salto, 1 lectura ($esPrimo[i]$)	2
for ($j \leftarrow i^2$ to n, step i)	1 escritura, 1 multiplicación (i * i), 2 lectura	4
$esPrimo[j] \leftarrow false$	1 escritura	1
end if	1 salto	1
end for	1 salto	1
for ($i \leftarrow 2$ to n)	1 salto, 1 escritura (i), 1 lectura (n)	3
if $esPrimo[i]$	1 salto, 2 lectura ($esPrimo[i]$, i)	3
$contador \leftarrow contador + 1$	1 lectura ($contador$), 1 suma, 1 escritura	3
end if	1 salto	1
end for	1 salto	1
return $contador$	1 salto, 1 lectura	2

Table 1: Análisis de complejidad del algoritmo de la Criba de Eratóstenes

Complejidad Temporal: El algoritmo si o si hace las primeras 6 lineas, y las últimas, de la 11 a la 17, por lo que el algotimo tiene 13+14 = 27, en el ciclo for ocurre lo siguiente:

El algoritmo realiza las siguientes operaciones:

- Inicialización del arreglo: $O(n)$
- Marcado de múltiplos: $O(n \log \log n)$
- Conteo final de primos: $O(n)$

Por lo tanto, la complejidad temporal total es:

$$O(n \log \log n + 26)$$

Pero como se pueden quitar las constantes solo nos queda $O(n \log \log n)$

Complejidad Espacial: El algoritmo utiliza un arreglo booleano de tamaño $n + 1$. Por ello, la complejidad espacial es:

$$O(n)$$

4 Ejercicio 5

Dado un arreglo A de n enteros, ¿existe un elemento de A tal que aparece en A al menos $n/2$ veces?

Solución

Algoritmo

Algorithm 4 Encontrar elemento mayoritario (apariciones $\geq n/2$)

Require: Un arreglo A de n enteros.
Ensure: Un elemento que aparece al menos $n/2$ veces, o null si no existe.

```
candidate ← null
count ← 0
for cada x en A do
    if count = 0 then
        candidate ← x
        count ← 1
    else
        if candidate = x then
            count ← count + 1
        else
            count ← count − 1
        end if
    end if
end for
occurrence ← 0
for cada x en A do
    if x = candidate then
        occurrence ← occurrence + 1
    end if
end for
if occurrence ≥ n/2 then
    return candidate
else
    return null {No existe elemento mayoritario}
end if
```

Análisis de complejidad

Para este algoritmo tenemos las siguientes operaciones:

Código	Operaciones	Total
$candidate \leftarrow$ null	1 asignación	1
$count \leftarrow 0$	1 asignación	1
for (cada x en A)	1 salto, 2 lecturas (A)	2
if $count = 0$	1 salto, 1 lectura ($count$) y 1 comparación	3
$candidate \leftarrow x$	1 escritura ($candidate$)	1
$count \leftarrow 1$	1 escritura ($count$)	1
else if $candidate = x$	1 salto, 2 lecturas ($candidate$, x), 1 comparación	4
$count \leftarrow count + 1$	1 lectura ($count$), 1 suma, 1 escritura	3
else	1 salto	1
$count \leftarrow count - 1$	1 lectura ($count$), 1 resta, 1 escritura	3
end if	1 salto	1
end for	1 salto	1
$occurrence \leftarrow 0$	1 asignación	1
for (cada x en A)	1 salto, 2 lecturas (A)	3
if $x = candidate$	1 salto, 2 lecturas (x, $candidate$), 1 comparación	4
$occurrence \leftarrow occurrence + 1$	1 lectura ($occurrence$), 1 suma, 1 escritura	3
end if	1 salto	1
end for	1 salto	1
if $occurrence \geq \frac{n}{2}$	1 salto, 2 lecturas ($occurrence$, n), 1 división	4
return $candidate$	1 salto, 1 lectura	2
else return null	1 salto	1
end if	1 salto	1

Table 2: Análisis de complejidad del algoritmo de Boyer-Moore para encontrar el elemento mayoritario

Donde las operaciones que se hacen siempre son del 1 al 4, de la 14 a la 16, la 21, uno de los dos return y la 25, por lo que al contarlas todas tenemos 1+1+2+3+3+4+3+4+1+1 = 23. Mientras que para los ciclos for ocurre lo siguiente:

Complejidad Temporal: El algoritmo realiza dos recorridos sobre el arreglo A:

- El primer recorrido (líneas 3 a 13) para identificar un candidato mayoritario, con una complejidad de $O(n)$.
- El segundo recorrido (líneas 14 a 18) para verificar que el candidato realmente aparece al menos $n/2$ veces, también con una complejidad de $O(n)$.

Por lo tanto, la complejidad temporal total es:

$$O(n) + O(n) + 23 = O(n)$$

Complejidad Espacial: El algoritmo utiliza únicamente un número constante de variables adicionales ($candidate$, $count$ y $occurrence$). Por ello, la complejidad espacial es:

$$O(1)$$