Tarea 3

Tony Santiago Montes Buitrago - 202014562
Juan Carlos Marin Morales - 202013973

Departamento de Ingeniería de Sistemas y Computación, Universidad de los Andes

9 de septiembre de 2021

Para los siguientes programas realice los siguientes pasos

I Determinar cual es el peor de los casos respecto al tiempo de ejecución

II Hacer una tabla que enumere las operaciones que se puede suponer que se ejecutan en tiempo constante y determinar cuántas veces se ejecuta cada una.

III Derivar una fórmula para el tiempo de ejecución del algoritmo

IV En caso de que la fórmula sea una ecuación de recurrencia, resolver la ecuación

V Determinar el orden de complejidad del algoritmo

Nota: Los algoritmos de solución de cada problema no son necesariamente las soluciones más eficientes.

Función 1

```
fun isPrime (n: nat) ret b: bool
var i : nat
{true}
b,i := true, 2
{i ≤ n ∧ b = (∀k|2 ≤ k < i : n mod k > 0)}
do i<n ∧ b ⟶ b,i := (n mod i > 0),i+1 od
ret b
{b = (∀k|2 ≤ k < n : n mod k > 0)}
```

I El peor de los casos para esta función es que el número n sí sea primo, dado que para este caso b nunca va a ser `false` y por ende i tendrá que recorrer todos los números desde 2 hasta n.

II

NOTA: A lo largo de todos los ejercicios se considerará la asignación paralela como 2 asignaciones.

c_i	Operación	No Ciclo	En Ciclo	Total Veces
c_1	Asignación (:=)	2	2	$2 + 2(n-2)$
c_2	Comparación ($<, >$)	0	1+1	$2(n-2)$
c_3	Lógica ($\wedge$)	0	1	$(n-2)$
c_4	Aritmética (`mod` ,+)	0	1+1	$2(n-2)$

III $f(n) = 2c_1 + 2(n-2)c_1 + 2(n-2)c_2 + (n-2)c_3 + 2(n-2)c_4$
$f(n) = (n-2) \cdot (2c_1 + 2c_2 + c_3 + 2c_4) + 2c_1$
$f(n) = n(2c_1 + 2c_2 + c_3 + 2c_4) - 4c_1 - 4c_2 - 2c_3 - 4c_4 + 2c_1$
$f(n) = n(2c_1 + 2c_2 + c_3 + 2c_4) - 2c_1 - 4c_2 - 2c_3 - 4c_4$

IV No aplica

V $f(n)$ es $O(n(2c_1 + 2c_2 + c_3 + 2c_4))$
$f(n)$ es $O(n)$

Función 2

```
fun division (a: nat, b: nat) ret <q,r> pair of nat
{b > 0}
q, r:= 0, a
{b > 0 ∧ a = q * b + r}
do r ≥ b → q, r := q + 1, r − b
od
ret <q, r>
{a = q * b + r ∧ r < b}
```

I El peor caso para este algoritmo está dado para un a = n, y un b = 1, y siendo así entre más grande sea n mayor sera el tiempo de ejecucion, siendo que el algoritmo depende de la diferencia entre a y b

II

c_i	**Operación**	**No Ciclo**	**En Ciclo**
c_1	asignación	2 veces	$2\frac{a}{b}$
c_2	comparación $\geq$	0	$1(\frac{a}{b}+1)$ veces dentro del do
c_3	+/-	0	$2\frac{a}{b}$ veces dentro del do, a/b veces

III T(a,b) $= 2c_1 + 2\frac{a}{b}c_1 + (\frac{a}{b}+1)c_2 + 2\frac{a}{b}c_3$
T(a,b) $= 2c_1 + 2\frac{a}{b}c_1 + \frac{a}{b}c_2 + c_2 + 2\frac{a}{b}c_3$
T(a,b) $= 2c_1 + c_2 + \frac{a}{b}(2c_1 + c_2 + 2c_3)$

IV No aplica

V Siendo el peor caso un a = n y b = 1, entonces el ciclo del algoritmo realizaría exactamente $\frac{a}{b} = \frac{n}{1} = n$ iteraciones. por lo que
$T(n) = 2 * c_1 + c_2 + n(2c_1 + c_2 + 2c_3)$
Entonces por teorema de sumas de funciones de complejidad:
$T(n)$ es $O(max(2c_1, c_2, n(2c_1 + c_2 + 2c_3)))$
Entonces
$T(n)$ es $O(n)$

Función 3

```
fun search(a: array [0,n) of real, e: real, f: nat) ret i: nat
{0 ≤ f < n}
if  a[f] = e → i := f
 [] a[f] ≠ e ∧ f = 0 → i := −1
 [] a[f] ≠ e ∧ f > 0 → i := search(a, e, f − 1)
fi
ret i
{(i = −1 ∧ (∀k|0 ≤ k ≤ f : a[k] ≠ e)) ∨ a[i] = e}
```

I El peor de los casos respecto al tiempo de ejecución es en el que, la búsqueda se inicia con el ultimo índice del arreglo a, es decir f = n-1, y el elemento que se está buscando no se encuentra.

II

c1	asignacion	1 en el peor caso
c2	comparacion de numeros, y	7 en el peor caso
c3	Acceder a un elemento del arreglo	3 en el peor caso
T(m)	Llamado recursivo	1 en el peor caso

III T(n) $= c_1 + 7c_2 + 3c_3 + T(n-1)$

IV

En el caso mas simple, (n = 1) entonces las operaciones realizadas serian las de la segunda guarda, por lo que:
f(1) $= c_1 + 7c_2 + 3c_3$

T(n) $= c_1 + 7c_2 + 3c_3 + T(n-1)$
Despejando la ecuación
T(n) - T(n-1) $= c_1 + 7c_2 + 3c_3$
Queda una ecuación de recurrencia no homogénea
Para resolver la parte homogénea:
Polinomio Característico de grado 1:

$\lambda - 1 = 0$
Cuya raíz es $\lambda = 1$
Por lo que la solución homogénea es:
$h(n) = k_1 1^n$

Para resolver la parte no homogénea proponemos una solución particular

$p(n) = d_1$
Como 1^n ya es una solución en h(n) entonces multiplicamos por n
$p(n) = n \cdot d_1$
Reemplazamos la solución particular por los llamados a T(n) en la ecuación original

$n \cdot d_1 - (n-1)d_1 = c_1 + 7c_2 + 3c_3$
$n \cdot d_1 - n \cdot d_1 + d_1 = c_1 + 7c_2 + 3c_3$
$d_1 = c_1 + 7c_2 + 3c_3$
$p(n) = n(c_1 + 7c_2 + 3c_3)$
Teniendo ya la solución homogénea y particular, armamos la solución general sumando estas dos, $f(n) = h(n) + p(n)$

$f(n) = k_1 1^n + n(c_1 + 7c_2 + 3c_3)$
$f(n) = k_1 + n(c_1 + 7c_2 + 3c_3)$
Teniendo que $f(1) = c_1 + 7c_2 + 3c_3$
$c_1 + 7c_2 + 3c_3 = k_1 + 1(c_1 + 7c_2 + 3c_3)$
$0 = k_1$
Finalmente

$f(n) = n(c_1 + 7c_2 + 3c_3)$

V $f(n)$ es $O(n)$

Función 4

```
fun search (a : array [0,n) of real, e: real, s: nat, f: nat) ret i: nat
var m : nat
{0 ≤ s < n ∧ s ≤ f < n}
m:= s + f
if m mod 2 = 0 ⟶ m:= m / 2
 [] m mod 2 = 1 ⟶ m:= (m–1) / 2
fi
{0 ≤ s ≤ m ≤ f < n ∧ (s = f ∨ m < f)}
if a[m] = e ⟶ i:= m
 [] a[m] ≠ e ∧ s=f ⟶ i:= –1
 [] a[m] ≠ e ∧ s<f ⟶ i:= search(a,e,m+1,f);
                      if i = –1 ⟶ i:= search(a,e,s,m);
                       [] i ≠ –1 ⟶ skip;
                      fi
fi
ret i
{(i = −1 ∧ (∀k|s ≤ k < f : a[k] ≠ e)) ∨ a[i] = e}
```

I El peor de los casos es que se busque en todo el arreglo, es decir $s = 0$ y $f = n - 1$ y que el elemento no esté en el arreglo, ya que en este caso tendría que revisar en todos los elementos de la lista.

II

$\mathbf{c_i}$	**Operación**	**Repeticiones**
c_1	Asignación ($:=$)	$1+1+1+1$ en el peor caso
c_2	Comparación ($\neq, =, <$)	$1+1+1+2+2+1+1$ en el peor caso
c_3	Lógica ($\wedge$)	2 en el peor caso
c_4	Aritmética ($+$, `mod` , $-$)	$1+1+1+1$ en el peor caso
c_5	División ($/$)	1
c_6	Acceso lista ($a[m]$)	$1+1+1$ en el peor caso
$T(m)$	Llamado recursivo	2

III $T(n) = 4c_1 + 9c_2 + 2c_3 + 4c_4 + c_5 + 3c_6 + 2 \cdot T(n/\mathbf{b})$

NOTA: Como en el primer llamado recursivo se procesa la segunda mitad de los datos y en el segundo llamado recursivo se procesa la primera mitad de los datos, se puede afirmar que: $\mathbf{b = 2}$

$T(n) = 4c_1 + 9c_2 + 2c_3 + 4c_4 + c_5 + 3c_6 + 2 \cdot T(n/2)$

IV El caso más simple (en el peor caso) sería que $n = 1$; para este caso se cumpliría la segunda guarda, ya que:

$(a[m] \neq e \wedge s = f) \equiv (a[0] \neq e \wedge 0 = 1 - 1) \equiv (\texttt{true} \wedge \texttt{true}) \equiv \texttt{true}$

y entonces no habría llamados recursivos y se ejecutarían únicamente las siguientes operaciones:

$T(1) = 3c_1 + 5c_2 + c_3 + 3c_4 + c_5 + 3c_6$

Se tiene la ecuación de recurrencia no lineal:

$T(n) = 4c_1 + 9c_2 + 2c_3 + 4c_4 + c_5 + 3c_6 + 2 \cdot T(n/2)$
$T(n) = 2 \cdot T(n/2) + (4c_1 + 9c_2 + 2c_3 + 4c_4 + c_5 + 3c_6)$

Y se obtiene de la fórmula general ($T(n) = a \cdot T(n/b) + f(n)$), que:

$f(n) = 4c_1 + 9c_2 + 2c_3 + 4c_4 + c_5 + 3c_6$,
$a = 2$ y
$b = 2$

En este caso se obtuvo una ecuación de recurrencia no lineal, por lo cual utilizaremos la técnica de cambio de dominio para transformarla en una ecuación lineal.

Para esto se definen las funciones g y γ de la forma:

$n = g(m) = b^m$
$\gamma(m) = T(g(m)) = T(n)$

Y se operan:

$$\gamma(m) = T(n) = T(g(m)) = a \cdot T(b^m/b) + f(b^m)$$
$$\gamma(m) = a \cdot T(b^{m-1}) + f(b^m)$$
$$\gamma(m) = a \cdot T(g(m-1)) + f(b^m)$$
$$\gamma(m) = a \cdot \gamma(m-1) + f(b^m)$$

$$\gamma(m) = 2 \cdot \gamma(m-1) + (4c_1 + 9c_2 + 2c_3 + 4c_4 + c_5 + 3c_6)$$
$$\gamma(m) - 2 \cdot \gamma(m-1) = 4c_1 + 9c_2 + 2c_3 + 4c_4 + c_5 + 3c_6$$

Y se obtiene una ecuación lineal, cuyo polinomio característico es:

$$\lambda - 2$$

Y su única raíz es: $\lambda = 2$

Por lo que la solución homogénea es:

$$h(m) = d_1 \cdot 2^m$$

Y para obtener la solución particular se utiliza lo que está al otro lado de la ecuación:

$$p(m) = d_2$$

Y se reemplaza la solución particular en la ecuación para hallar el valor de d_2:

$$d_2 - 2 \cdot d_2 = 4c_1 + 9c_2 + 2c_3 + 4c_4 + c_5 + 3c_6$$
$$d_2 = -(4c_1 + 9c_2 + 2c_3 + 4c_4 + c_5 + 3c_6)$$

Y finalmente para obtener la solución general, se suman la homogénea y la particular:

$$\gamma(m) = h(m) + p(m)$$
$$\gamma(m) = d_1 \cdot 2^m + d_2$$

Y se reemplaza $\gamma(m)$ por $T(n)$, aplicando $m = log_2(n)$:

$$T(n) = d_1 \cdot 2^{log_2(n)} + d_2 = d_1 \cdot n + d_2$$
$$T(n) = d_1 \cdot n - (4c_1 + 9c_2 + 2c_3 + 4c_4 + c_5 + 3c_6)$$

Para hallar el valor de la constante d_1 se reemplaza con el caso base:

$$T(1) = d_1 \cdot 1 - (4c_1 + 9c_2 + 2c_3 + 4c_4 + c_5 + 3c_6)$$
$$3c_1 + 5c_2 + c_3 + 3c_4 + c_5 + 3c_6 = d_1 - (4c_1 + 9c_2 + 2c_3 + 4c_4 + c_5 + 3c_6)$$
$$d_1 = 7c_1 + 14c_2 + 3c_3 + 7c_4 + 2c_5 + 6c_6$$

Obteniendo finalmente:

$$T(n) = d_1 \cdot n + d_2 = n \cdot (7c_1 + 14c_2 + 3c_3 + 7c_4 + 2c_5 + 6c_6) - (4c_1 + 9c_2 + 2c_3 + 4c_4 + c_5 + 3c_6)$$

V $T(n)$ es $O(n(7c_1 + 14c_2 + 3c_3 + 7c_4 + 2c_5 + 6c_6))$
$T(n)$ es $O(n)$

Función 5

```
fun sumMainDiagonal ( a: matrix[0,n)[0,n) of real) ret s: real
var i: nat
var j: nat
{true}
i:=0
{s = (∑|0 ≤ k < i : a[k,k]}
do i < n →
      j:= 0
      {(i ≥ j ∧ s = (∑k|0 ≤ k < i : a[k,k])) ∨ (i < j ∧ s = (∑k|0 ≤ k ≤ i : a[k,k]))}
      do i < n →
            if i = j → s := s + a[i,j]
3           [] i ≠ j → skip
            fi
            j:= j+1
      od
      i:= i+1
od
ret s
{s = (∑k|0 ≤ k < n : a[k,k])}
```

I En este caso el tiempo de ejecución siempre es el mismo para un mismo tamaño de la matriz, ya que el algoritmo siempre recorre por completo la matriz. Así el tiempo de ejecución se hace peor conforme la matriz crece. Siendo así que el peor caso siempre es tener que recorrer toda la matriz.

II

k_i	Operación	No Ciclo	1er Ciclo	2do ciclo
k_1	asignación	1	2	1 vez fija 1 vez por iteración del primer ciclo
k_2	comparación de números	0	1	3
k_3	Suma	0	1	1 vez fija 1 vez por iteración del primer ciclo
k_4	Acceder a un elemento de la matriz	0	0	1 vez por iteración del primer ciclo

III

$f(n) = k_1 + n(3k_1 + k_2 + 2k_3 + k_4 + n(k_1 + 3k_2 + k_3))$
$f(n) = k_1 + n(3k_1 + k_2 + 2k_3 + k_4 + nk_1 + 3nk_2 + nk_3)$
$f(n) = k_1 + 3nk_1 + nk_2 + 2nk_3 + nk_4 + n^2k_1 + 3n^2k_2 + n^2k_3$
$f(n) = n^2(k_1 + 3k_2 + k_3) + n(3k_1 + k_2 + 2k_3 + k_4) + k_1$

IV No Aplica

V $f(n)$ es $O(max(n^2(k_1+3k_2+k_3), n(3k_1+k_2+2k_3+k_4), k_1))$
$f(n)$ es $O(n^2(k_1+3k_2+k_3))$
$f(n)$ es $O(n^2)$

Función 6

```
fun sumMainDiagonal(a: matrix[0,n)[0,n) of real) ret s: real
var i: nat
{true}
i:= 0
{s = (∑k|0 ≤ k < i : a[k,k])}
do i < n → s, i := s + a[i,i], i + 1 od
ret s
{s = (∑k|0 ≤ k < n : a[k,k])}
```

I Al igual que en la función anterior, el tiempo de ejecución es el mismo para un mismo tamaño de la matriz, en este caso el peor caso siempre es tener que recorrer todos los índices entre 0 y n-1. Igualmente el tiempo de ejecución se hace mas grande conforme la matriz crece.

II

k_i	**Operacion**	**No Ciclo**	**En Ciclo**	**Total Veces**
k_1	asignacion	1	2	2n+1
k_2	comparacion de numeros	0	1	n+1
k_3	Suma	0	1	2n
k_4	Acceder a un elemento de la matriz	0	1	n

III

$f(n) = k_1 + k_2 + n(2k_1 + k_2 + 2k_3 + k_4)$

IV No Aplica

V $f(n) = O(max(k_1, k_2, n(2k_1 + k_2 + 2k_3 + k_4)))$
$f(n) = O(n(2k_1 + k_2 + 2k_3 + k_4))$
$f(n) = O(n)$

Función 7

```
fun search (a:matrix [0,m)[0,n) of real, e: real) ret <i,j>: pair of nat
var i,j: nat
i,j := 0,0;
{(i < m ∧ j < n ∧ (∀k,l|0 ≤ k < i ∧ 0 ≤ l < n : a[k,l] ≠ e)) ∧ (∀l|0 ≤ l < j : a[i,l] ≠ e))}
do i≠ m ∧ a[i,j] ≠ e ⟶
        if j<n−1 ⟶ j:= j+1
```

```
        [] j=n-1 --> i , j := i+1 , 0
      fi
od
ret <i , j>
```

$\{(i = m \wedge (\forall k, l | 0 \leq k < m \wedge 0 \leq l < n : a[k, l] \neq e)) \vee a[i, j] = e\}$

I El peor de los casos es que el elemento e no se encuentre en ninguna posición de la matriz, ya que en este caso tendrá que revisar la matriz posición por posición desde el comienzo hasta el final.

II

c_i	**Operación**	**No Ciclo**	**En Ciclo**	**Total**
c_1	Asignación $(:=)$	2	1 + 1 (diferentes)	$2 + m * n + m$
c_2	Comparación $(\neq, =, <)$	0	3 + 1 (diferentes)	$3(m * n) + m$
c_3	Lógica $(\wedge)$	0	1	$(m * n)$
c_4	Aritmética $(+, -)$	0	2 + 1 (diferentes)	$2(m * n) + m$
c_5	Acceso matriz $(a[i, j])$	0	1	$(m * n)$

NOTA: En este caso específico, los operadores de la segunda guarda del if interno (y su respectivo programa) solo se ejecutan m veces, mientras que las de la guarda del do y la primera guarda del if sí se ejecutan $m * n$ veces.

III $f(n) = 2c_1 + (m*n)c_1 + mc_1 + 3(m*n)c_2 + mc_2 + (m*n)c_3 + 2(m*n)c_4 + mc_4 + (m*n)c_5$
$f(n) = (m * n) \cdot (c_1 + 3c_2 + c_3 + 2c_4 + c_5) + m \cdot (c_1 + c_2 + c_4) + c_1$

IV No aplica

V $f(n)$ es $O(max((m * n) \cdot (c_1 + 3c_2 + c_3 + 2c_4 + c_5), m \cdot (c_1 + c_2 + c_4), c_1))$
$f(n)$ es $O((m * n) \cdot (c_1 + 3c_2 + c_3 + 2c_4 + c_5))$
$f(n)$ es $O(m * n)$