UNIVERSIDAD ADOLFO IBAÑEZ
FACULTAD DE INGENIERÍA Y CIENCIAS

Guía 1

TICS311: Estructura de Datos y Algoritmos

Semestre 01-2023

1 Lenguaje C

Conceptos básicos que hay que tener claro:

- Tipos de variables, casting.
- Control de flujo.
- Funciones.
- Arreglos, strings y structs.
- Punteros: direcciones de memoria, operador &, operador *, puntero nulo, punteros a punteros.
- Paso por valor vs paso por referencia.
- Aritmética de punteros.
- Arreglos, punteros y funciones.
- Struct, punteros y funciones.
- Memoria estática (stack) vs memoria dinámica (heap). Funciones malloc y free.

2 Análisis de algoritmos

1. Escriba las siguientes complejidades en notación Θ:
   (a) $n \log n + 0.001n^2 + 3000n$
   (b) $10 \log n + 50 \log (n^2)$
   (c) $2^n n^3 + 3^n$
   (d) $2^{2n} + 2^n$
   (e) $500n^2(\log n)^3 + 100n^3(\log n)^2$
   (f) $n + \sqrt{n} + 10$
   (g) $100 \log n + 2n$
   (h) $n^3 + n^2 \log n$
   (i) $40n^{10} + 400n^{\log n} + 100$
   (j) $n! + n^{2n}$
   (k) $(\log n)^{10} + \sqrt{n}$
   (l) $2^n + n^{20}$
2. Ordene las siguientes complejidades según orden de crecimiento:
   (a) $n^{0.5}$ $(\log n)^2$ $\log n$ $\log \log n$ $n \log n$

(b) 2^n $n!$ 1.5^n $\log\log n$ n^{10}

(c) $n^2 \log n$ n^3 $2^n n$ 2^{2n} 3^n

(d) n^{10} $n \log n$ $n^{\log n}$ $n\sqrt{\log n}$ 2^n

3. Indique las alternativas correctas:

   (a) La función $n \log n$ es:

   (i) $\Omega(n^2)$ (ii) $O(n^2)$ (iii) $\Omega(n)$ (iv) $\Omega(\log n)$ (v) $\Theta(1)$

   (b) La función $n^{1.5}$ es:

   (i) $\Theta(\sqrt{n^3})$ (ii) $\Omega(n^2)$ (iii) $\Omega(n)$ (iv) $\Omega(2^n)$ (v) $O(1)$

   (c) La función 3^n es:

   (i) $O(4^n)$ (ii) $\Omega(3^{2n})$ (iii) $\Omega(n^3)$ (iv) $\Omega(1)$ (v) $O(3^{\sqrt{n}})$

   (d) La función 2^{n+1} es:

   (i) $\Theta(2^n)$ (ii) $\Theta(2^{2n})$ (iii) $O(2^{2n})$ (iv) $\Omega(2^n n)$ (v) $O(n^2)$

4. Considere el siguiente algoritmo para invertir un arreglo:

```
function invertir(arreglo A, largo n):
1       B ← A.copy()
2       for 0 <= i < n:
3            A[i] = B[n-1-i]
```

   Acá la función `A.copy()` entrega una nueva copia del arreglo A. ¿Cuál es la complejidad tiempo y la complejidad espacio de este algoritmo? Explique su análisis.

5. Considere el siguiente algoritmo para chequear si todos los elementos de un arreglo son distintos o no:

```
function distintos(arreglo A, largo n):
1       for 0 <= i < n:
2            for i < j < n:
3                 if A[i] = A[j]:
4                      return false
5       return true
```

   ¿Cuál es la complejidad tiempo de este algoritmo? Explique su análisis.

6. Suponga que tiene una función `ordenar` con complejidad $\Theta(n \log n)$ que recibe un arreglo A de largo n y lo ordena de menor a mayor. Utilice la función `ordenar` para escribir un algoritmo que resuelva el problema de la pregunta anterior en tiempo $\Theta(n \log n)$. Explique su análisis.

7. Considere el siguiente algoritmo para combinar dos arreglos ordenados de tamaño n en un arreglo ordenado de tamaño $2n$:

```
function combinar(arreglo A1, arreglo A2, largo n):
```

```
1       B ← create_array(2n)
2       pos1 ← 0
3       pos2 ← 0
4       i ← 0
5       while pos1 < n AND pos2 < n:
6            if A1[pos1] < A2[pos2]:
7                 B[i] ← A1[pos1]
8                 pos1 ← pos1 + 1
9            else:
10                B[i] ← A2[pos2]
11                pos2 ← pos2 + 1
12           i ← i+1

13       while pos1 < n:
14            B[i] ← A1[pos1]
15            pos1 ← pos1 + 1
16            i ← i+1

17       while pos2 < n:
18            B[i] ← A2[pos2]
19            pos2 ← pos2 + 1
20            i ← i+1

21       return B
```

Acá la función `create_array(k)` crea un arreglo vacío de tamaño `k`. ¿Cuál es la complejidad tiempo y la complejidad espacio de este algoritmo? Explique su análisis.

3 Recursión

1. Considere el siguiente algoritmo recursivo para calcular la potencia de a^n. Por simplicidad, asumiremos que n es siempre una potencia de 2.

```
function pow(a, n):
1       if n == 1:
2            return a
3       x ← pow(a, n/2)
4       return x*x
```

   (a) Ejecute la función con argumentos `pow(3, 8)`, indicando todas las llamadas recursivas a la función `pow` y qué retorna cada llamada.

   (b) Escriba una ecuación recursiva para la complejidad tiempo del algoritmo (en función de n), y encuentre el crecimiento de la complejidad (notación Θ).

2. Considere el siguiente algoritmo recursivo para calcular el mínimo sobre un arreglo A de largo n :

```
function minimo(arreglo A, largo n):
1       if n == 1:
2            return A[0]
```

```
3       x ← minimo(A, n-1)
4       if A[n-1] < x:
5            return A[n-1]
6       else:
7            return x
```

(a) Ejecute la función con argumentos `minimo([4,2,5], 3)`, indicando todas las llamadas recursivas a la función `minimo` y qué retorna cada llamada.

(b) Escriba una ecuación recursiva para la complejidad tiempo del algoritmo, y encuentre el crecimiento de la complejidad (notación Θ).

3. Considere el siguiente algoritmo recursivo para chequear si un arreglo es un palindrome o no (es decir, si al invertirlo obtenemos el mismo arreglo):

```
function palindrome(arreglo A, inicio, fin):
1       if inicio >= fin:
2            return true
3       if A[inicio] != A[fin]:
4            return false
5       else:
6            return palindrome(A, inicio+1, fin-1)
```

(a) Ejecute la función con argumentos `palindrome([5,4,4,5], 0, 3)`, indicando todas las llamadas recursivas a la función `palindrome` y qué retorna cada llamada. Haga lo mismo con `palindrome([5,3,4,5], 0, 3)`.

(b) Escriba una ecuación recursiva para la complejidad tiempo del algoritmo (en función del largo del arreglo $n = fin - inicio + 1$), y encuentre el crecimiento de la complejidad (notación Θ). Por simplicidad, puede asumir para su análisis que n es par.

4. Escriba una función recursiva `suma_pares(n)` que recibe un número $n \geq 0$ y calcula la suma de todos los números pares entre 0 y n (acá tomamos 0 como número par).

5. Escriba una función recursiva `division(a,b)` que calcule la división entera a/b entre los enteros positivos a y b.