

Algoritmos & Estructuras de Datos
Guía Práctica
Universidad de San Andrés

Magalí Marijuán

I. Javier Mermet

Matías Sandacz

5 de diciembre de 2022

Índice general

I	Introducción	2
1	Lenguaje C	3
1.1	Ejercicios para entrar en calor	3
1.2	Punteros y Arreglos	3
1.2.1	Más arreglos!	4
1.3	Ordenamiento	5
2	Recursividad	7
3	Lectura de Archivos	10
4	Magia con Punteros	11
II	Estructuras de Datos	12
5	Listas	13
6	Pilas	14
7	Colas	15
8	Heaps	16
9	Árboles	17
10	Grafos	18
11	Hashing	19

Parte I

Introducción

Capítulo 1

Lenguaje C

1.1 Ejercicios para entrar en calor

1. Escribir la función que dado $n \in \mathbb{N}$ devuelve si es primo. Recuerden que un número es primo si los únicos divisores que tiene son 1 y el mismo.
2. Escribir la función que dado $n \in \mathbb{N}$ devuelve la suma de todos los números impares menores que n .

1.2 Punteros y Arreglos

3. ¿Cuál es el valor de a y b luego de ejecutar el programa?

```
void myFunc(int* a, int b)
{
    (*a)++;
    b++;
}

void main()
{
    int a = 10;
    int b = 10;
    myFunc(&a, b);
}
```

4. ¿Que valor se imprime por consola luego de cada llamado a *printf*?

```
void main()
{
    int x = 10;
    int* px = &x;

    printf("%d \n", px);
    printf("%d \n", (*px));
}
```

```

    (*px)++;

    printf("%d \n", px);
    printf("%d \n", (*px));
}

```

5. Programar las siguientes funciones en C:

(a) `void crearArreglo(int v)`

Crea un arreglo estático de enteros de tamaño 8, inicializando todos sus elementos con *v*, y lo imprime en pantalla.

(b) `int* crearArregloDin(int n, int v)`

Dado un natural *n*, crea un arreglo de enteros de ese tamaño, inicializando todos sus elementos con *v*, y devuelve un puntero al mismo.

(c) Invocar la siguiente función con cualquiera de los arreglos inicializados anteriormente y convencerse de que sus elementos están ubicados de manera **contigua** en memoria. *Recordar que cada elemento de tipo int ocupa 4 bytes.*

```

void mostrarMemoria(int* arr, int size)
{
    for(int i=0; i<size; i++)
    {
        printf("Elemento: %d, Direccion: %d\n", i, &arr[i]);
    }
}

```

6.Cuál es la diferencia entre **malloc** y **calloc**? Cuando utilizamos la función **free**?

7. Dado el siguiente struct que representa una Persona

```

typedef struct Persona{
    int edad;
    char* nombre;
} Persona;

```

`Persona* inicializarPersonas(int n)`

Completar la función `inicializarPersonas`, que dado un entero *n*, crea un arreglo de *n* personas, y devuelve un puntero al mismo.

(a) Utilizando **malloc**.

(b) Utilizando **calloc**.

1.2.1 Más arreglos!

8. Programar las siguientes funciones en C:

(a) `int maximo(int* arr, int size)`

Dado un arreglo de enteros *arr* de tamaño *size*, devuelve el máximo elemento.

- (b) `void sumador(int* arr, int size, int c)`

Dado un arreglo de enteros *arr* de tamaño *size*, y un entero *c*, suma *c* a todos los elementos de *arr*.

- (c) `char* copiar(char* arr)`

Crea una copia de *arr*, y devuelve un puntero a la copia.

- (d) `int* reverso(int* arr, int size)`

Dado un arreglo de enteros *arr* de tamaño *size*, devuelve su reverso.

Ejemplo: Dado [1, -2, 85, 65] se debe devolver [65, 85, -2, 1].

i. Se puede modificar *arr*.

ii. Sin modificar *arr*.

- (e) `bool estaOrdenado(int* arr, int size)`

Dado un arreglo *arr* de enteros de tamaño *size*, retorna true si es monótonamente creciente o monótonamente decreciente.

- (f) `bool esPalindromo(char* s)`

Dado un string *s*, retorna true si es un palíndromo. *Recuerden que un palíndromo es una palabra que se lee igual en un sentido que en otro (por ejemplo; Ana, Anna, Otto).*

1.3 Ordenamiento

9. `void ordenar(int* arr, int size)`

Dado un arreglo de enteros *arr* de tamaño *size*, escribir una función que lo ordene de menor a mayor.

- (a) Utilizando *selection sort*.
- (b) Utilizando *insertion sort*.
- (c) Dar la complejidad temporal en el peor caso de ambos algoritmos.
- (d) Cual es la complejidad temporal en el peor caso de ambos algoritmos, suponiendo que el arreglo de entrada ya está ordenado?
- (e) Suponiendo que queremos obtener únicamente los $k < size$ menores elementos del arreglo, ¿Cuál de los dos algoritmos utilizaría? ¿Por qué?

10. Decimos que un algoritmo de ordenamiento es **estable** si preserva el orden relativo de los elementos con la misma clave.

Supongamos que queremos ordenar el arreglo de tuplas $A = [(3,6), (1,5), (3,2)]$, utilizando como clave el primer componente de cada tupla. En este caso, dos resultados diferentes son posibles, uno de los cuales mantiene un orden relativo de elementos con claves iguales, y una en la que no:

$[(1,5), (3,6), (3,2)]$ Preserva el orden relativo entre (3,6) y (3,2)

$[(1,5), (3,2), (3,6)]$ No preserva el orden relativo entre (3,6) y (3,2)

- (a) Decidir si los algoritmos de ordenamiento presentados en el ejercicio anterior son estables. Justificar.
11. Consideremos el siguiente algoritmo de ordenamiento, llamado **bubble sort** (suponemos que n es el tamaño del arreglo):

```
int i = 0;
while( i < n-1 )
{
    int j = 0;
    while( j < n-1 )
    {
        if( a[j] > a[j+1] )
            swap(a, j, j+1);
        j++;
    }
    i++;
}
```

- (a) Describir con palabras qué hace este algoritmo.
- (b) ¿Cuántas veces se ejecuta el swap del ciclo interior como máximo (i.e. en el peor caso)?
12. `bool twoSum(int* arr, int size, int target)`

Dado un arreglo de enteros sin repetidos arr de tamaño $size$, se debe devolver true si existen dos elementos $a1$ y $a2$ tal que $a1 + a2 = target$. En caso contrario, devolver false.

- (a) Dar una solución con complejidad temporal $O(n^2)$.
- (b) Dar una solución con complejidad temporal $O(n \log n)$.
Sugerencia: Ordenar el arreglo.

Capítulo 2

Recursividad

1. `int fibo(int n)`

Los números de Fibonacci son una sucesión de números naturales definida por las ecuaciones:

$$F_n = F_{n-1} + F_{n-2}$$

con $F_1 = F_2 = 1$. Escribir una función recursiva para computar el n -ésimo número de Fibonacci.

2. `int suma(int n)`

Escribir una función recursiva para calcular la suma de los primeros n números.

3. `int factorial(int n)`

Escribir una función recursiva para calcular el factorial de n .

4. `int potencia(int a, int n)`

Escribir una función recursiva para calcular a^n .

5. `void imprimir(int* arr, int n)`

Escribir una función recursiva para imprimir todos los elementos de un arreglo con n elementos.

6. `void imprimirReverso(int* arr, int n)`

Escribir una función recursiva para imprimir todos los elementos de un arreglo con n elementos en orden reverso.

7. `int max(int* arr, int n)`

Escribir una función recursiva para calcular el máximo elemento de un arreglo de n elementos.

8. `int ocurrencias(int* arr, int n, int elem)`

Escribir una función recursiva para contar la cantidad de veces que aparece el elemento $elem$ en un arreglo arr de n elementos.

9. `void reverso(char* str)`

Escribir una función recursiva para calcular el reverso de un String.

10. Como vieron en la teórica, **merge sort** es el algoritmo recursivo más conocido para ordenar un arreglo. Utiliza una técnica algorítmica conocida como **Divide and Conquer**, y consta en dividir el arreglo en dos partes iguales, ordenar cada una recursivamente, y finalmente combinar los resultados.

(a) Implementar merge sort utilizando pseudocódigo. Dar su complejidad temporal.

(b) ¿Que desventaja tiene merge sort al ser un algoritmo recursivo?

Sugerencia: Pensar en la complejidad espacial y el stack del proceso.

11. La **búsqueda binaria** es un algoritmo eficiente para encontrar un elemento **en un arreglo ordenado**. El código es el siguiente: (Suponiendo que n es el tamaño del arreglo).

```
bool binarySearch(int* a, int n, int target)
{
    int low = 0, high = n - 1;

    // Iteramos hasta agotar los elementos
    while (low <= high)
    {
        // Calculamos el punto medio del arreglo
        int mid = (low + high)/2;

        // Encontramos el target
        if (target == nums[mid]) {
            return true;
        }

        // Si el target es menor que el elemento del medio,
        // descartamos todos los elementos a la derecha.
        else if (target < nums[mid]) {
            high = mid - 1;
        }

        // Si el target es mayor que el elemento del medio,
        // descartamos todos los elementos a la izquierda.
        else {
            low = mid + 1;
        }
    }

    // Target no está en el arreglo.
    return false;
}
```

- (a) Describir con palabras qué hace este algoritmo.

- (b) Mostrar con un ejemplo que el algoritmo **no** es correcto si el arreglo de entrada no está ordenado.
- (c) Implementar la búsqueda binaria utilizando recursión. Dar su complejidad temporal.

12. `int* buscarRango(int* arr, int size, int target)`

Dado un arreglo de enteros *arr* de tamaño *size*, ordenado de menor a mayor, encontrar los índices de comienzo y de final de *target*. Devolverlos en un arreglo. Además, se puede suponer que *target* es un elemento de *arr*.

Por ejemplo, si *arr* = [5,7,7,8,8,8,10], *target* = 8, se debe devolver [3,5].

- (a) El algoritmo dado debe tener complejidad $O(n)$.
- (b) (Difícil) El algoritmo dado debe tener complejidad $O(\log n)$.

Sugerencia: Utilizar búsqueda binaria

Capítulo 3

Lectura de Archivos

Capítulo 4

Magia con Punteros

```
1. typedef struct Persona {
    char* nombre;
    char* apellido;
    char* domicilio;
    int edad;
} Persona;

char* masGrande(Persona** personas, int n)
{
    // COMPLETAR
}
```

Completar la función *masGrande*, que dado un arreglo de n personas, devuelve el nombre de la persona de mayor edad.

```
2. void imprimirMatriz(int** matrix, int n, int m)
```

Dada una matriz de n filas y m columnas, imprimir todos sus elementos en pantalla. Se deben imprimir todos los elementos de la primer fila, luego todos los de la segunda fila, y así sucesivamente. Por ejemplo, dada la matriz de 2×3 $[[1,2,3], [4,5,6]]$, se debe imprimir 1, 2, 3, 4, 5, 6.

```
3. void traspuesta(int** matrix, int n)
```

Dada una matriz cuadrada de $n \times n$, modificarla para obtener su traspuesta. Por ejemplo, dada la matriz cuadrada $[[1,2], [3,4]]$, se debe modificar para obtener $[[1, 3], [2, 4]]$. Recuerden que la matriz traspuesta es aquella que surge como resultado de realizar un cambio de columnas por filas y filas por columnas en la matriz original.

Parte II

Estructuras de Datos

Capítulo 5

Listas

Capítulo 6

Pilas

Capítulo 7

Colas

Capítulo 8

Heaps

Capítulo 9

Árboles

Capítulo 10

Grafos

Capítulo 11

Hashing