Algoritmos & Estructuras de Datos

Guía Práctica

Universidad de San Andrés

Gianmarco Cafferata Magalí Marijuán I. Javier Mermet

Matías Sandacz

24 de marzo de 2023

Índice general

Parte I

Introducción

Capítulo 1

Lenguaje C

1.1 Ejercicios para entrar en calor

Ejercicio 1.1 Escribir la función que dado n ∈ N devuelve si es primo. Recuerden que un número es primo si los únicos divisores que tiene son 1 y el mismo.

Ejercicio 1.2 Escribir la función que dado n ∈ N devuelve la suma de todos los números impares menores que n.

1.2 Punteros y Arreglos

Ejercicio 1.3 ¿Cuál es el valor de a y b luego de ejecutar el programa?

```
1  void myFunc(int* a, int b)
2  {
3  (*a)++;
4  b++;
5  }
6
7  void main()
8  {
9      int a = 10;
10     int b = 10;
11     myFunc(&a, b);
12 }
```

Ejercicio 1.4 ¿Que valor se imprime por consola luego de cada llamado a `printf`?

```
1  void main()
2  {
```

```
    int x = 10;
    int* px = &x;

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

    (*px)++;

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

Ejercicio 1.5 Implementar las siguientes funciones en C

1. `void crearArreglo(int v)`

   Crea un arreglo estático de enteros de tamaño 8, inicializando todos sus elementos con v, y lo imprime en pantalla.
2. `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.
3. 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]);
    }
}
```

Ejercicio 1.6 Cuál es la diferencia entre **malloc** y **calloc**? Cuando utilizamos la función **free**?

Ejercicio 1.7 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.

1. Utilizando **malloc**.
2. Utilizando **calloc**.

1.2.1 Más arreglos!

Ejercicio 1.8 Programar las siguientes funciones en C:

1. `int maximo(int* arr, int size)`

   Dado un arreglo de enteros *arr* de tamaño *size*, devuelve el máximo elemento.
2. `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*.
3. `char* copiar(char* arr)`

   Crea una copia de *arr*, y devuelve un puntero a la copia.
4. `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].
   1. Se puede modificar *arr*.
   2. Sin modificar *arr*.
5. `bool estaOrdenado(int* arr, int size)`

   Dado un arreglo *arr* de enteros de tamaño *size*, retorna true si es monótonamente creciente o monótonamente decreciente.
6. `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

Ejercicio 1.9

```
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.

1. Utilizando ***selection sort***.
2. Utilizando ***insertion sort***.
3. Dar la complejidad temporal en el peor caso de ambos algoritmos.
4. Cual es la complejidad temporal en el peor caso de ambos algoritmos, suponiendo que el arreglo de entrada ya está ordenado?
5. Suponiendo que queremos obtener únicamente los $k < size$ menores elementos del arreglo, ¿Cuál de los dos algoritmos utilizaría? ¿Por qué?

Ejercicio 1.10 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)

1. Decidir si los algoritmos de ordenamiento presentados en el ejercicio anterior son estables. Justificar.

Ejercicio 1.11 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++;
}
```

1. Describir con palabras qué hace este algoritmo.
2. ¿Cuántas veces se ejecuta el swap del ciclo interior como máximo (i.e. en el peor caso)?

Ejercicio 1.12

```
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.

1. Dar una solución con complejidad temporal $O(n^2)$.
2. Dar una solución con complejidad temporal $O(nlogn)$.

Capítulo 2

Recursividad

Ejercicio 2.1

```
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.

Ejercicio 2.2

```
int suma(int n)
```

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

Ejercicio 2.3

```
int factorial(int n)
```

Escribir una función recursiva para calcular el factorial de n.

Ejercicio 2.4

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

Escribir una función recursiva para calcular a^n.

Ejercicio 2.5

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

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

Ejercicio 2.6

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

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

Ejercicio 2.7

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

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

Ejercicio 2.8

```
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 arrreglo *arr* de n elementos.

Ejercicio 2.9

```
void reverso(char* str)
```

Escribir una función recursiva para convertir *str* en su reverso. Por ejemplo, dado el String "abc", se debe modificarlo para obtener "cba".

Ejercicio 2.10 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.

1. Implementar merge sort utilizando pseudocódigo. Dar su complejidad temporal.
2. ¿Que desventaja tiene merge sort al ser un algoritmo recursivo?
   Sugerencia: Pensar en la complejidad espacial y el stack del proceso.

Ejercicio 2.11 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).

```
1  bool binarySearch(int* a, int n, int target)
2  {
3      int low = 0, high = n - 1;
4
5      // Iteramos hasta agotar los elementos
6      while (low <= high)
7      {
8          // Calculamos el punto medio del arreglo
9          int mid = (low + high)/2;
10
11         // Encontramos el target
12         if (target == nums[mid]) {
13             return true;
```

```
14          }
15
16          // Si el target es menor que el elemento del medio,
17          // descartamos todos los elementos a la derecha.
18          else if (target < nums[mid]) {
19              high = mid - 1;
20          }
21
22          // Si el target es mayor que el elemento del medio,
23          // descartamos todos los elementos a la izquierda.
24          else {
25              low = mid + 1;
26          }
27      }
28
29  // Taget no está en el arreglo.
30  return false;
31
32  }
```

1. Describir con palabras qué hace este algoritmo.
2. Mostrar con un ejemplo que el algoritmo **no** es correcto si el arreglo de entrada no esta ordenado.
3. Implementar la búsqueda binaria utilizando recursión. Dar su complejidad temporal.

Ejercicio 2.12

```
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].

1. El algoritmo dado debe tener complejidad $O(n)$.
2. ⚠ El algoritmo dado debe tener complejidad $O(\log n)$.

Capítulo 3

Lectura de Archivos

Capítulo 4

Magia con Punteros

Ejercicio 4.1

```
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.

Ejercicio 4.2

```
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 $2x3$ [[1,2,3], [4,5,6]], se debe imprimir 1, 2, 3, 4, 5, 6.

Ejercicio 4.3

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

Dada una matriz cuadrada de nxn, 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

Ejercicio 5.1

1. ¿Qué cambios se le tendrían que hacer a una lista simplemente enlazada para permitir que la operación `eliminar último` tenga complejidad $O(1)$?
2. ⚠ ¿Se le ocurre cómo hacer para que la solución al punto anterior tenga una menor complejidad de memoria?

Ejercicio 5.2 (🤖) Implementar una función que permita verificar si una lista es un palíndromo (es decir, los elementos son los mismos cuando se lee de principio a fin o al revés). No se pueden usar estructuras de datos adicionales para resolver este problema.

1. ¿Qué complejidad temporal tiene la solución al problema anterior?

Ejercicio 5.3 Implementar funciones para unir dos listas $l1$ y $l2$.

1. Anexando todos los elementos de $l2$ al final de $l1$
2. Intercalando elementos de $l1$ y $l2$

Ejercicio 5.4 En diversos sistemas distribuidos es común ver arquitecturas que utilizan *anillos* para balancear carga o sincronizarse. Un ejemplo es como Cassandra maneja las particiones de datos. En este ejercicio se propone:

1. Pensar que modificaciones hay que hacer a una lista simplemente enlazada para convertirla en una lista circular.
2. Queremos que nuestro anillo tenga 256 posiciones. Pensar como limitar la cantidad de nodos acorde a esta limitación.
3. Implementar una función `insertarDato` que permita insertar datos en nuestro anillo.

```
int insertarDato(lista_circular_t* listac, int dato) {
    // garantizado que devuelve un valor entre 0 y 255 inclusive
    int particion = hash(dato);
    // completar
}
```

   En esta función, la *partición* donde se guardará el dato es un número entero entre 0 y 255 (de modo de quedar dentro del rango de 256 posiciones de nuestro anillo). Luego,

donde hay que completar, se debe buscar cuál es el nodo que maneja esa partición. Un nodo j conoce su posición inicial i_j y tiene una lista enlazada de datos. Una vez encontrado el nodo, se debe colocar en su lista de datos el dato pasado.

4. Pensar como se deben rebalancear los datos del anillo cuando se agrega un nuevo nodo en una posición aleatoria.
5. Análogamente, pensar como se deben rebalancear los datos del anillo cuando se remueve un nodo.

Capítulo 6

Pilas

Ejercicio 6.1 Implementar un programa que lea una secuencia de letras hasta llegar a un punto e imprima los caracteres ingresados en orden inverso.

Ejercicio 6.2 Escribir un programa que lee una secuencia de caracteres y verifica que todos los parentesis, corchetes y llaves están balanceados.

Ejercicio 6.3 Implementar las siguientes funciones

1. `int stack_sorted_descending(pila_t* pila);` que verifica si los elementos de la pila están ordenados en orden decreciente
2. `int stack_sorted_ascending(pila_t* pila);` que verifica si los elementos de la pila están ordenados en orden creciente

Ejercicio 6.4 Dados dos numeros n y m, y la siguiente lista de operaciones permitidas:

1. $n \to n - 1$
2. $n \to n + 2$
3. $n \to n * 2$

Escribir una función que encuentre la secuencia más corta de operaciones permitidas para llegar de n a m.

Ejercicio 6.5 Implementar una calculadora de notación polaca. A continuación, se provee un ejemplo:

```

int calcular(pila_t* pila) {
    // completar
}

int main(void) {
    pila_t* pila = pila_crear();

```

```
9       // 25 * 2 + 10 - 2 * 3 - 4
10      // Pasa a ser:
11      // - - + * 25 2 10 * 2 3 4
12
13      pila_apilar(&pila, "4");
14      pila_apilar(&pila, "3");
15      pila_apilar(&pila, "2");
16      pila_apilar(&pila, "*");
17      pila_apilar(&pila, "10");
18      pila_apilar(&pila, "2");
19      pila_apilar(&pila, "25");
20      pila_apilar(&pila, "*");
21      pila_apilar(&pila, "+");
22      pila_apilar(&pila, "-");
23      pila_apilar(&pila, "-");
24
25      int resultado = calcular(&pila);
26      // debería imprimir 50
27      printf("El resultado es %d", resultado);
28
29      return 0;
30  }
```

Capítulo 7

Colas

Ejercicio 7.1 Utilizando una cola, invertir una pila.

Ejercicio 7.2 Implementar una cola utilizando un arreglo dinámico. Pensar esta estructura como una interfaz amigable a un arreglo en memoria dinámica, de modo que los llamados a `malloc` y `realloc` sean transparentes al usuario.

Ejercicio 7.3 Implementar una cola utilizando una lista simplemente enlazada.

1. ¿Haría alguna diferencia utilizar una lista doblemente enlazada?

Ejercicio 7.4 Implementar una pila utilizando dos colas

Capítulo 8

Heaps

Ejercicio 8.1 Implementar el algoritmo `heapify` y dar su complejidad temporal.

1. Utilizando `heapify up`
2. Utilizando `heapify down`

Ejercicio 8.2 Queremos una función `top k` que recibe un arreglo de números y un valor k y devuelve los k mayores del arreglo.

1. La primer implementación ordena el arreglo y busca los k mayores en las primeras k posiciones. Indique el orden.
2. La segunda implementación usa un arreglo auxiliar de k elementos. Se recorre el arreglo de números y cada número se compara con los k en el arreglo auxiliar: si es mayor a alguno se saca el menor de ellos reemplazándolo por el nuevo. Indique el orden de esta implementación.
3. Realice una implementación cuyo orden sea $O(nlog(k))$.

Ejercicio 8.3 Mergear M arreglos ordenados en un solo arreglo ordenado en $O(nlog(m))$, donde n es la cantidad total de elementos de todos los arreglos.

Ejercicio 8.4 Implementar un algoritmo que verifique si un arreglo de números representa a un heap.

Ejercicio 8.5 Recibimos dos arreglos ordenados A y B y un número k. Nos piden encontrar los k pares de ambos arreglos que menos suman (cumpliendo que sean k). Ejemplos:

- $A = [0, 1, 3, 6]$, $B = [1, 2, 5]$
    1. Con $k = 1$ la solución es $[(0, 1)]$ ya que es el único par que suma 1.
    2. Con $k = 2$ la solución es $[(1, 1), (0, 2)]$.
    3. Con $k = 3$ no hay solución.
- $A = [0, 2, 5, 7]$, $B = [0, 1, 3, 6]$
    1. Con $k = 1$ la solución es $[(0, 1)]$.

2. Con $k = 2$ la solución es $[(2, 1), (3, 0)]$.
3. Con $k = 3$ la solución es $[(2, 6), (5, 3), (7, 1)]$.

Implemente una función que lo resuelva en $O(nlog(n))$

Ejercicio 8.6 Recibimos números constantemente (que no llegan ordenados de ninguna forma) y queremos saber en cualquier momento la mediana de todos ellos. La mediana es el número que deja al 50 % de los números a su izquierda (menores a el) y al otro 50 % a su derecha (los mayores). Ejemplos:

- Para $[1, 2, 4, 6]$ la mediana es 3.
- Para $[1, 2, 6]$ la mediana es 2.
- Para $[1, 2, 4, 4, 6]$ la mediana es 4.

Podemos querer la mediana varios momentos a medida nos llegan los números. Debemos implementar dos funciones:

1. `add` agrega un número nuevo
2. `get median` obtiene la mediana actual. Se pide que sea $O(1)$.

⚠ **Ejercicio 8.7** Una de las optimizaciones más comunes al servir modelos de Deep Learning en servidores web es lo que se denomina *microbatching*. En lugar de pasarle cada input por separado modelo, se le pasa un grupo de inputs (un *batch*) y se obtienen todas las salidas al mismo tiempo. De este modo, la *latencia* de cada request es mayor (el tiempo desde que se envía el request hasta que el cliente obtiene la respuesta) pero el *throughput* es mayor (la cantidad de requests procesados por segundo).

En este ejercicio, imaginaremos que estamos sirviendo un modelo que recibe una entrada de texto `input` y un parámetro `n`, que es un parámetro a pasarle al modelo para inferencia. El valor de `n` $\in [1, 20]$. Solo podemos armar un *batch* con elementos con el mismo valor de `n`. Queremos implementar una estructura de datos que nos permita obtener, en un momento dado, los `k` elementos más antiguos del grupo cuyo elemento más antiguo lleva mas tiempo esperando para ser procesado.

Ejemplo:

```
encolar_request(estructura, "texto1.0", 1);
encolar_request(estructura, "texto1.1", 1);
encolar_request(estructura, "texto2.1", 2);
encolar_request(estructura, "texto2.2", 2);
encolar_request(estructura, "texto2.3", 2);
encolar_request(estructura, "texto3.0", 3);
encolar_request(estructura, "texto2.4", 2);
encolar_request(estructura, "texto3.1", 3);
input_t **inputs = obtener_batch(estructura, 4);
```

Nos debería dar como resultado los elementos con `input` "texto1.0" y "texto1.1":

1. El elemento con `input` "texto1.0" es el más antiguo
2. Si bien pedimos 4 elementos, solo hay 2 en ese grupo.

Si pidieramos `obtener_batch(estructura, 4)`, obtendríamos los 4 elementos con `n=2`.
La estructura debe cumplir la siguiente interfaz:

```
// Encola un request
encolar_request(estructura_t *estructura, char *input, int n);

// Obtiene el batch de k elementos cuyo elemento más
// antiguo sea el más antiguo esperando a ser procesado.
obtener_batch(estructura_t *estructura, int k);
```

Se pide esbozar la estructura y las dos primitivas anteriores. Justificar su complejidad temporal.
Hint: pensar como podemos aprovechar que el valor de n tiene un rango fijo. Utilizar estructuras de datos explicadas previamente (colas, pilas, heaps, arreglos).

Capítulo 9

Árboles

9.0.1 Árboles Binarios

Ejercicio 9.1 Recorrer un árbol binario utilizando los siguientes recorridos

1. Preorder
2. Postorder
3. Inorder

Ejercicio 9.2 Dado un árbol binario, se define una `hoja` como un nodo sin hijos. Implementar una función que imprima por pantalla el valor de todas sus hojas de izquierda a derecha.

Ejercicio 9.3 La altura de un árbol binario se define como la cantidad de ejes entre la raíz y la hoja más lejana. Implementar la función `int altura(Node* raiz)` que dada la raíz de un árbol binario, devuelve su altura. Dar su complejidad temporal.

Ejercicio 9.4 Implemente la función `bool sumTree(Node* raíz)` que retorna verdadero si un árbol binario es un sumTree. Un sumTree es un árbol en donde el valor de todos sus nodos es igual a la suma de todos los nodos de su subárbol izquierdo más los del subárbol derecho. Dar su complejidad temporal.

9.0.2 Árboles Binarios de Búsqueda

Ejercicio 9.5 Implemente la función `bool esBST(Node* raíz)` que retorna verdadero si el árbol binario es un árbol de búsqueda. Dar su complejidad temporal.

Ejercicio 9.6 Implemente la función `bool pertenece(Node* raíz, target int)` que retorna verdadero si el target pertenece al árbol binario de búsqueda.

1. Dar su complejidad temporal.
2. Dar su complejidad temporal en el caso en que el árbol esté balanceado.

Ejercicio 9.7 Dado un árbol de búsqueda binario y un elemento, agregarlo al árbol. Se debe asegurar que el árbol resultante siga siendo un árbol de búsqueda binario válido después de agregar el elemento.

Ejercicio 9.8 Dado un árbol de búsqueda binario y un elemento perteneciente al árbol, eliminar ese elemento del árbol. Se debe asegurar que el árbol resultante siga siendo un árbol de búsqueda binario válido después de eliminar el elemento.

Ejercicio 9.9 Dado un arreglo entero nums donde los elementos están ordenados en orden ascendente, implementar una función que lo convierta en un árbol de búsqueda binario **balanceado**. Dar su complejidad temporal.

Capítulo 10

Grafos

Capítulo 11

Hashing

Capítulo 12

Respuestas

Solución del ejercicio 5.1

Este ejercicio apunta a ayudar a descubrir las listas doblemente enlazadas y las XOR lists.