Materia:

ESTRUCTURA DE DATOS

Docente:

María Jacinta Martinez Castillo

Integrantes:

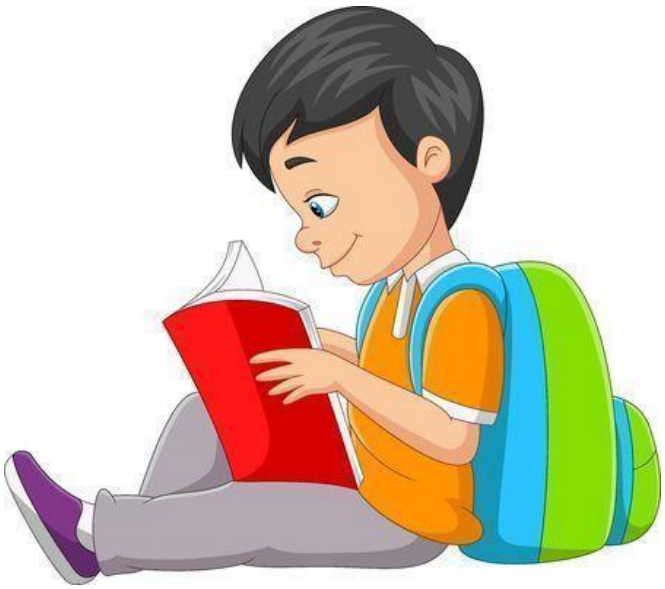
Hernández Acevedo Brayan

Ríos Méndez Cesar

Sentíes Robles Pedro

Hora:

17:00 - 18:00



REPORTE DE PRÁCTICA

	**UNIDAD: 2** **NO. DE PRÁCTICA: 2** **NOMBRE DE PRÁCTICA: RECURSIVIDAD**	**NO. DE CONTROL Y NOMBRE:** **HERNANDEZ ACEVEDO BRAYAN** **21010962** **RIOS MENDEZ CESAR** **21011029** **SENTIES ROBLES PEDRO** **21011047**
AGO-DIC 2021	**DOCENTE: M.A.T.I. MARÍA JACINTA MARTINEZ CASTILLO**	**FECHA ENTREGA: 27/03/23**

INTRODUCCIÓN

La recursividad es un concepto fundamental en programación que se refiere a la capacidad de una función para llamarse a sí misma de forma repetitiva. Esta técnica permite resolver problemas complejos dividiéndolos en subproblemas más pequeños que se resuelven de forma recursiva.

En términos generales, una función recursiva se compone de dos elementos principales: el caso base y el caso recursivo. El caso base es la condición que indica cuándo la función debe dejar de llamarse a sí misma, mientras que el caso recursivo es el proceso que se ejecuta de forma repetitiva para resolver el problema en cuestión.

La recursividad es especialmente útil en algoritmos que trabajan con estructuras de datos como árboles, listas y grafos, ya que permite recorrer estas estructuras de manera más eficiente. Sin embargo, es importante tener en cuenta que el uso excesivo de la recursividad puede generar un elevado consumo de memoria y un mayor tiempo de ejecución.

En resumen, la recursividad es una técnica poderosa y elegante que permite resolver problemas complejos mediante la descomposición en subproblemas más pequeños, pero requiere un buen entendimiento y manejo para evitar problemas de rendimiento y errores

OBJETIVO

Capacitar a los estudiantes para que comprendan los conceptos fundamentales de la recursividad y sepan cómo aplicar la recursión para resolver problemas complejos en estructuras de datos. Al finalizar el tema, los estudiantes deben ser capaces de diseñar e

implementar funciones recursivas en lenguajes de programación, como Java o Python, para resolver problemas de estructuras de datos como árboles, listas enlazadas, pilas y colas. Además, los estudiantes deben estar familiarizados con las técnicas de programación recursiva, como la división y conquista, y la retroalimentación, y entender cómo estas técnicas pueden ayudar a resolver problemas más complejos en estructuras de datos. En general, el objetivo es dotar a los estudiantes de las herramientas necesarias para comprender y aplicar la recursión en la resolución de problemas en estructuras de datos en un entorno informático.

MARCO TEÓRICO

2.1 Definición

La recursividad es un concepto que se indica cuando un método se llama a sí mismo. Cuando creamos un método recursivo debemos tener en cuenta que este tiene que terminar por lo que dentro del método debemos asegurarnos de que no se está llamando a si mismo todo el rato, Lo que quiere decir que el ciclo es finito.

2.2 Procedimientos iterativos

La estructura iterativa, normalmente conocida como iteración, puede definirse como una secuencia de paso e instrucciones que tienen un destino. Su objetivo primordial es que puedan ser ejecutadas en repetidas ocasiones.

Bucles
Los bucles, en la programación iterativa, se encargan de reproducir en repetidas ocasiones un bloque o paquete de instrucciones. Dicho bucle puede repetirse un número fijo de veces, por ejemplo, X veces, o dependiendo del número de criterios ordenados, ya sea una prueba de una o varias condiciones.

Como ya hemos afirmado con anterioridad, de forma común se conocen dos clases de estructura iterativa o bucles.

Por un lado, nos podemos encontrar con uno que ejecuta un numero dado de veces controlado primordialmente por un contador o por un índice que ayuda a aumentar la estructura iterativa. Este se puede incluir dentro de la familia FOR.
Por otra parte, tenemos una clase de bucle que solo se ejecuta si este está cumpliendo una condición, de lo contrario no accederá. Dicha condición se comprobará al inicio y/o también puede comprobarse al final de la construcción. Este tipo podemos incluirlo al grupo WHILE OR REPEAT, donde WHILE es el inicio y REPEAT es el final de la programación iterativa.

2.3 Procedimientos Recursivos

Para resolver un problema, resuelve un subproblema que sea una instancia más pequeña del mismo problema, y después usa la solución de esa instancia más pequeña para resolver el problema original.

Para que un algoritmo recursivo funcione, los subproblemas más pequeños eventualmente deben llegar al caso base. Cuando calculamos !n, !, los subproblemas se hacen más y más pequeños hasta que calculamos 0!0!0, !. Debes *asegurarte de que, eventualmente, llegues al caso base.*

Podemos extraer la idea de la recursión en dos reglas sencillas:

Cada llamada recursiva debe ser sobre una instancia más pequeña del mismo problema, es decir, un subproblema más pequeño.

Las llamadas recursivas eventualmente deben alcanzar un caso base, el cual se resuelve sin más recursividad.

Regresemos a las muñecas rusas. Aunque no aparecen en ningún algoritmo, puedes ver que cada muñeca encierra a todas las muñecas más pequeñas (de manera análoga al caso recursivo), hasta que la muñeca más pequeña no encierra a ninguna otra (como el caso base).

2.4 Cuadro comparativo iterativo vs recursivo

ITEM	ITERATIVO	RECURSIVO
CICLO	WHILE, IF	IF
Recurso tiempo	Ocupa menos recurso de tiempo.	Depende de la cantidad de entrada, por ende se demora más.
Recurso Espacio Memoria	Ocupa menos espacio en memoria, porque la función de ejecuta menos veces.	Ocupa más espacio en memoria, porque a la medida que se va ejecutando va abriendo mas métodos

Análisis de algoritmos

El análisis de algoritmos es una parte importante de la teoría de la complejidad computacional, que proporciona una estimación teórica de los recursos necesarios de un algoritmo para resolver un problema computacional específico. La mayoría de los algoritmos están diseñados para trabajar con entradas de longitud arbitraria. El análisis de algoritmos es la determinación de la cantidad de recursos de tiempo y espacio necesarios para ejecutarlo.

Por lo general, la eficiencia o el tiempo de ejecución de un algoritmo se establece como una función que relaciona la longitud de entrada con el número de pasos, conocido como complejidad de tiempo o volumen de memoria, conocido como complejidad de espacio.

Complejidad en el tiempo

La complejidad del tiempo, en los términos más simples, es la duración que tarda en ejecutarse un algoritmo. De manera similar, la complejidad del espacio es la cantidad de espacio necesaria para que se ejecute un algoritmo. Cuando se refiera a la complejidad del tiempo, lo verá comúnmente como notación Big-O, como O (n), O (log n), O (n^2), etc.

La n representa la entrada, mientras que la O representa la Función de tasa de crecimiento en el peor de los casos.

La complejidad del tiempo y el espacio son temas importantes que siempre surgirán en una entrevista técnica por varias razones. La primera razón es que toda empresa valora sus recursos, y tener algoritmos eficientes que ocupen menos tiempo y espacio siempre ayudará a la empresa a ahorrar recursos. La segunda razón es demostrar su conocimiento de escalabilidad. A medida que un algoritmo se utiliza en conjuntos de datos cada vez más grandes, ¿cómo cambia eso sus requisitos de tiempo de ejecución y espacio?

Su comprensión del rendimiento de un algoritmo puede ayudar a ahorrar una gran cantidad de recursos a una empresa. Si un algoritmo tarda una hora en calcularse, mientras que otro tarda 10 minutos, esos 50 minutos pueden ser extremadamente valiosos y se pueden utilizar para abordar otras tareas. Además, si duplicamos la cantidad de datos y entradas que el algoritmo necesitaba procesar, el tiempo para el primer algoritmo podría aumentar potencialmente a dos horas, mientras que el segundo algoritmo podría tomar solo 20 minutos. Como puede ver, ser consciente de la complejidad del tiempo de un algoritmo puede ahorrar mucho tiempo.

Complejidad en el espacio
Es la memoria que utiliza un programa para su ejecución. Lo que implica que la eficiencia en memoria de un algoritmo lo indica la cantidad de espacio requerido para ejecutarlo, es decir, el espacio memoria que ocupan todas las variables propias del algoritmo.
Eficiencia en los algoritmos
El análisis de la eficiencia es una parte clave del análisis de algoritmos. Esto incluye medir el tiempo de ejecución y la memoria exigida por un algoritmo. Esto implica contar el número de operaciones que se realizan para completar una tarea. El número de operaciones se expresa como una función cuadrática del tamaño del problema, o como una función exponencial si el problema es complejo.

El análisis de la eficiencia también permite comparar los algoritmos para ver cuál es el más eficiente. Esto se logra midiendo el número de operaciones que se realizan para completar una tarea. Los desarrolladores de software también pueden usar el análisis de eficiencia para identificar áreas de mejora en un algoritmo existente.

METODOLOGÍA

- Entendimiento del concepto de recursividad: Se debe comprender el concepto de recursividad y cómo se aplica en la programación de estructuras de datos. Es importante comprender cómo funciona la recursividad y cómo puede utilizarse para resolver problemas complejos de manera más eficiente.

- Identificación de los casos base: Es importante identificar los casos base para cada función recursiva. Estos son los casos en los que la recursión se detiene y se devuelve un valor final. Es importante identificar estos casos para evitar bucles infinitos y asegurarse de que el algoritmo termine.

- Identificación de los casos recursivos: Después de identificar los casos base, se deben identificar los casos recursivos. Estos son los casos en los que la función se llama a sí misma con una entrada diferente para acercarse al caso base. Es importante tener en cuenta que cada llamada recursiva debe acercarse al caso base.

- Diseño y prueba de la función recursiva: Una vez que se han identificado los casos base y los casos recursivos, se puede diseñar la función recursiva. Es importante probar la función para asegurarse de que devuelve los resultados correctos.

- Optimización de la función recursiva: La recursividad puede ser muy ineficiente si se usa incorrectamente, por lo que es importante optimizar la función recursiva para garantizar que sea lo más eficiente posible. Esto puede implicar reducir la cantidad de llamadas recursivas o usar técnicas como la recursión de cola.

- Documentación de la función recursiva: Es importante documentar la función recursiva para que otros programadores puedan entender su funcionamiento. La documentación debe incluir información sobre los casos base y recursivos, así como cualquier consideración de optimización o limitación en el uso de la función recursiva.

DESARROLLO

EJERCICIO 1 NÚMEROS PARES CON SU RESPECTIVO CUBO

```java
public class NumerosParesCubos2 {
    public static void main(String[] args) {
        imprimirNumerosParesCubos(2, 1);
    }

    public static void imprimirNumerosParesCubos(int num, int count) {
        if (count <= 10) {
            System.out.println(num + "^3 = " + (Math.pow(num, 3)));
            imprimirNumerosParesCubos(num+2, count+1);
        }
    }
}
```

El programa "NumerosParesCubos2" es un ejemplo de cómo se puede utilizar la recursividad para resolver problemas en programación. Este programa en particular, tiene como objetivo imprimir los primeros 10 cubos de números impares a partir de un número inicial dado.

En términos generales, la recursividad es una técnica de programación que permite que una función se llame a sí misma para resolver un problema. En este caso, la función imprimirNumerosPrimosCubo se llama a sí misma para imprimir los cubos de los números impares en una secuencia determinada. La recursividad en este programa se utiliza para repetir un proceso y obtener un resultado final.

La recursividad tiene algunas ventajas en programación, como la simplicidad y la facilidad de lectura del código, además de poder resolver problemas de manera eficiente y en algunos casos, elegante. Sin embargo, la recursividad también puede tener algunos inconvenientes, como el hecho de que puede ser menos eficiente en algunos casos que una solución iterativa, especialmente cuando se utilizan grandes cantidades de memoria para las llamadas recursivas.

En el caso de este programa, la recursividad funciona de manera eficiente ya que se trata de un proceso relativamente simple y la cantidad de llamadas recursivas es limitada. Además, el uso de la recursividad en este programa permite que el código sea más legible y fácil de entender.

Otro aspecto importante a destacar de este programa es la utilización de la función Math.pow para calcular el cubo de los números. Esta función, proporcionada por la biblioteca estándar de Java, permite realizar operaciones de potencia de manera sencilla, sin tener que implementar la función de potencia de manera manual.

En conclusión, el programa "NumerosParesCubos" es un ejemplo práctico de cómo se puede utilizar la recursividad para resolver problemas en programación. Este programa demuestra cómo la recursividad puede ser una herramienta eficiente y elegante para resolver problemas, especialmente cuando se trata de procesos relativamente simples.

A continuación, se explica el paso a paso del programa:

1. En el método main, se llama a la función imprimirNumerosPrimosCubo y se le pasan dos argumentos: el número inicial y un contador con valor inicial de 1.

2. Dentro de la función imprimirNumerosPrimosCubo, se verifica si el contador es menor o igual a 10. Si este es el caso, se imprime el cubo del número actual utilizando la función Math.pow y se llama a la función imprimirNumerosPrimosCubo nuevamente con el siguiente número impar y el contador incrementado en 1.

3. Si el contador es mayor que 10, la función finaliza.

4. Cada vez que se llama a la función imprimirNumerosPrimosCubo, se verifica si el contador aún no ha alcanzado el valor límite de 10. Si este es el caso, se imprime el cubo del número actual y se llama a la función nuevamente con el siguiente número impar y el contador incrementado en 1.

5. El proceso de recursión continúa hasta que el contador alcanza el valor límite de 10, momento en el que la recursión termina.

EJERCICIO 2 NÚMEROS PRIMOS CON SU RESPECTIVO CUBO

```
3  public class NumerosPrimosCubos {
4      public static void main(String[] args) {
5          imprimirNumerosPrimosCubo(3, 1);
6      }
7
8      public static void imprimirNumerosPrimosCubo(int num, int count) {
9          if (count <= 10) {
10             System.out.println(num + "^3 = " + (Math.pow(num, 3)));
11             imprimirNumerosPrimosCubo(num+2, count+1);
12         }
13     }
```

El programa "NumerosPrimosCubos" es un ejemplo de cómo la recursividad puede ser utilizada para resolver problemas complejos de manera más simple y elegante. En este caso, el programa utiliza la recursividad para imprimir los primeros 10 cubos de números primos a partir de un número inicial dado.

El programa se divide en dos funciones principales: imprimirNumerosPrimosCubo y esPrimo. La función imprimirNumerosPrimosCubo es la función recursiva que se encarga de imprimir los cubos de los números primos. Esta función recibe dos argumentos: el número actual y un contador. En cada llamada recursiva, la función verifica si el contador aún no ha alcanzado el valor límite de 10 y si el número actual es primo. Si ambas condiciones se cumplen, se imprime el cubo del número actual y se llama a la función imprimirNumerosPrimosCubo nuevamente con el siguiente número primo y el contador incrementado en 1. Este proceso de recursión se repite hasta que se han impreso los 10 primeros cubos de números primos.

La función esPrimo es una función auxiliar que se utiliza para verificar si un número dado es primo o no. Esta función utiliza un ciclo for para dividir el número dado entre todos los números menores que él mismo y determinar si tiene algún divisor. Si encuentra un divisor, devuelve false, lo que significa que el número no es primo. Si el ciclo termina sin encontrar ningún divisor, devuelve true, lo que significa que el número es primo.

El uso de la recursividad en este programa permite una solución elegante y clara al

problema de imprimir los primeros 10 cubos de números primos. En lugar de utilizar un ciclo for complejo para iterar a través de todos los números y verificar si son primos, la recursividad nos permite dividir el problema en tareas más pequeñas y manejables. El código se vuelve más legible y fácil de entender y se reduce el riesgo de errores en la lógica de programación.

A continuación, se explica el paso a paso del programa:

1. En el método **main**, se llama a la función **imprimirNumerosPrimosCubo** y se le pasan dos argumentos: el número inicial y un contador con valor inicial de 1.

2. Dentro de la función **imprimirNumerosPrimosCubo**, se verifica si el contador es menor o igual a 10. Si este es el caso, se verifica si el número actual es primo utilizando una función auxiliar llamada **esPrimo**. Si el número es primo, se imprime el cubo del número actual utilizando la función **Math.pow** y se llama a la función **imprimirNumerosPrimosCubo** nuevamente con el siguiente número primo y el contador incrementado en 1.

3. Si el contador es mayor que 10, la función finaliza.

4. La función **esPrimo** recibe un número como argumento y devuelve **true** si el número es primo y **false** en caso contrario. Esta función utiliza un ciclo **for** para verificar si el número es divisible por algún otro número menor a él mismo. Si encuentra un número que divide al número dado, devuelve **false**. Si el ciclo termina sin haber encontrado ningún divisor, devuelve **true**.

5. Cada vez que se llama a la función **imprimirNumerosPrimosCubo**, se verifica si el contador aún no ha alcanzado el valor límite de 10. Si este es el caso, se verifica si el número actual es primo. Si el número es primo, se imprime el cubo del número actual y se llama a la función nuevamente con el siguiente número primo y el contador incrementado en 1.

6. El proceso de recursión continúa hasta que el contador alcanza el valor límite de 10, momento en el que la recursión termina.

EJERCICIO 3 CUENTA VOCALES

```java
import javax.swing.JOptionPane;

public class ContadorVocales {

    public static void main(String[] args) {
        String str = JOptionPane.showInputDialog(null, "Introduce una cadena de texto:");
        String resultado = contarVocales(str);
        JOptionPane.showMessageDialog(null, resultado);
    }
```

```java
    public static String contarVocales(String str) {
        return contarVocalesHelper(str, 0, 0, 0, 0, 0, 0);
    }

    private static String contarVocalesHelper(String str, int index, int aCount, int eCount, int iCount, int oCount, int uCount) {
        if (index == str.length()) {
            return "a: " + aCount + "\n" +
                   "e: " + eCount + "\n" +
                   "i: " + iCount + "\n" +
                   "o: " + oCount + "\n" +
                   "u: " + uCount;
        }

        char c = Character.toLowerCase(str.charAt(index));
        if (c == 'a') {
            return contarVocalesHelper(str, index + 1, aCount + 1, eCount, iCount, oCount, uCount);
        } else if (c == 'e') {
            return contarVocalesHelper(str, index + 1, aCount, eCount + 1, iCount, oCount, uCount);
        } else if (c == 'i') {
            return contarVocalesHelper(str, index + 1, aCount, eCount, iCount + 1, oCount, uCount);
        } else if (c == 'o') {
            return contarVocalesHelper(str, index + 1, aCount, eCount, iCount, oCount + 1, uCount);
        } else if (c == 'u') {
            return contarVocalesHelper(str, index + 1, aCount, eCount, iCount, oCount, uCount + 1);
        } else {
            return contarVocalesHelper(str, index + 1, aCount, eCount, iCount, oCount, uCount);
        }
    }
}
```

Este programa es un contador de vocales que utiliza el diálogo emergente de JOptionPane para recibir una cadena de texto de entrada y luego cuenta el número de vocales que contiene. El resultado se muestra también en un cuadro de diálogo emergente.

El programa comienza pidiendo al usuario que introduzca una cadena de texto utilizando el método showInputDialog de JOptionPane. Este método devuelve la cadena introducida por el usuario. Luego, el programa llama a la función contarVocales, que se encarga de contar el número de vocales en la cadena.

La función contarVocales utiliza una función auxiliar contarVocalesHelper que se llama

recursivamente hasta que se llega al final de la cadena. La función auxiliar toma como parámetros la cadena, un índice que indica la posición actual en la cadena y contadores para cada vocal. En cada llamada recursiva, se comprueba si se ha llegado al final de la cadena. Si no es así, se comprueba si el carácter actual es una vocal. Si es así, se incrementa el contador correspondiente y se llama recursivamente a la función auxiliar con el contador actualizado y el índice incrementado en 1. Si el carácter actual no es una vocal, simplemente se llama recursivamente a la función auxiliar con el índice incrementado en 1 y los contadores sin cambios.

Cuando se llega al final de la cadena, se devuelve una cadena formateada que muestra el número de ocurrencias de cada vocal. Esta cadena se muestra al usuario utilizando el método showMessageDialog de JOptionPane.

A continuación se describe el paso a paso del programa:

1. El programa comienza pidiendo al usuario que ingrese una cadena de texto utilizando una ventana emergente de JOptionPane. El usuario ingresa la cadena y presiona el botón "Aceptar".

2. El programa llama a la función "contarVocales", pasando la cadena de texto ingresada como argumento.

3. La función "contarVocales" llama a la función "contarVocalesHelper" y le pasa la cadena de texto ingresada, el índice 0 y los contadores de vocales a, e, i, o y u inicializados en 0.

4. La función "contarVocalesHelper" comprueba si el índice actual es igual a la longitud de la cadena de texto. Si es así, devuelve una cadena de texto que muestra el recuento de cada vocal.

5. Si el índice actual no es igual a la longitud de la cadena de texto, la función

"contarVocalesHelper" toma el carácter en el índice actual y lo convierte en minúscula utilizando el método "Character.toLowerCase".

6. La función "contarVocalesHelper" compara el carácter con cada vocal (a, e, i, o, u) utilizando la estructura "if-else".

7. Si el carácter es una "a", se llama a la función "contarVocalesHelper" de nuevo con el índice incrementado en 1 y el contador de "a" incrementado en 1.

8. Si el carácter es una "e", se llama a la función "contarVocalesHelper" de nuevo con el índice incrementado en 1 y el contador de "e" incrementado en 1.

9. Si el carácter es una "i", se llama a la función "contarVocalesHelper" de nuevo con el índice incrementado en 1 y el contador de "i" incrementado en 1.

10. Si el carácter es una "o", se llama a la función "contarVocalesHelper" de nuevo con el índice incrementado en 1 y el contador de "o" incrementado en 1.

11. Si el carácter es una "u", se llama a la función "contarVocalesHelper" de nuevo con el índice incrementado en 1 y el contador de "u" incrementado en 1.

12. Si el carácter no es una vocal, se llama a la función "contarVocalesHelper" de nuevo con el índice incrementado en 1 y los contadores de vocales sin cambios.

13. La función "contarVocalesHelper" se llama a sí misma recursivamente hasta que el índice actual es igual a la longitud de la cadena de texto.

14. La función "contarVocales" devuelve la cadena de texto devuelta por la función "contarVocalesHelper".

15. La cadena de texto devuelta se muestra al usuario en una ventana emergente de

JOptionPane. La ventana muestra el recuento de cada vocal en la cadena de texto ingresada por el usuario.

CONCLUSIÓN

.En conclusión, la recursividad es una técnica poderosa y versátil en la programación de estructuras de datos. Permite la solución elegante y eficiente de problemas que requieren la manipulación repetitiva de datos. Además, la recursividad es una herramienta fundamental en la programación orientada a objetos, ya que muchos de sus métodos y funciones se construyen de manera recursiva.

Sin embargo, la recursividad también puede llevar a errores como el desbordamiento de la pila (stack overflow) o la generación de bucles infinitos. Por lo tanto, es importante diseñar algoritmos recursivos cuidadosamente y aplicar la recursividad de manera eficiente para evitar estos problemas.

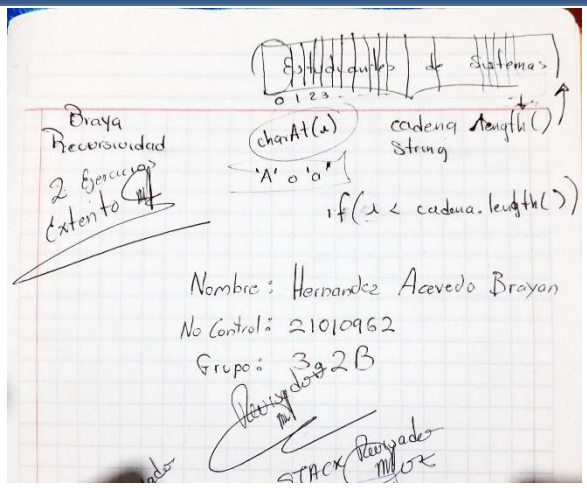
BIBLIOGRAFÍA

Morales, J. I. S. (s. f.). Recursividad. Ugr.es. Recuperado 14 de abril de 2023, de https://ccia.ugr.es/~jfv/ed1/c/cdrom/cap6/cap66.htm

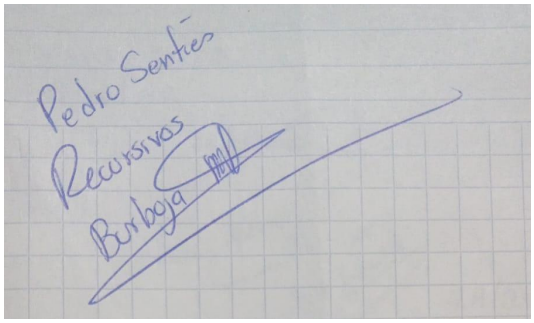
TylerMSFT. (s. f.). *Funciones recursivas*. Microsoft.com. Recuperado 14 de abril de 2023, de https://learn.microsoft.com/es-es/cpp/c-language/recursive-functions

Firmas:

Brayan:



Pedro: Tenia otra, pero no recuerdo en que libreta esta ☹



Cesar: Creo tiene dos pero no esta en su casa y no las mando al grupo ☹