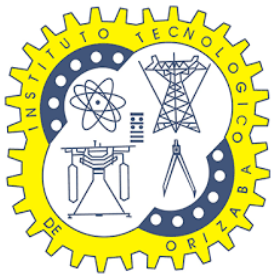


TECNOLÓGICO NACIONAL DE MÉXICO SEDE ORIZABA

PERIODO: ENERO- JUNIO 2023

CLAVE DE CLASE: 3a3A

CARRERA: ING. INFORMÁTICA

HORARIO DE CLASE: 15:00 A 16:00

MATERIA: ESTRUCTURA DE DATOS

EQUIPO: 8

TEMA:

REPORTE DE PRACTICAS SEGUNDA UNIDAD

(ITERATIVIDAD Y RECURSIVIDAD)

FACILITADORA:

MARIA JACINTA MARTINEZ CASTILLO

ASPIRANTES:

ALBERTO YERIEL GÓMEZ LÓPEZ (**21010189**)

GAEL OCTAVIO MORALES VALDEOLIVAR (**21010203**)

AXEL REYES GUEVARA (**21010213**)

INTRODUCCION

La recursividad y la iteratividad son dos conceptos fundamentales en la programación que permiten resolver problemas de manera eficiente y elegante.

La recursividad es una técnica en la que una función se llama a sí misma para resolver un problema. Es una herramienta poderosa para dividir un problema complejo en subproblemas más pequeños y manejables, lo que puede simplificar enormemente la lógica del código. Sin embargo, la recursividad también puede ser peligrosa si no se implementa correctamente, ya que puede llevar a una llamada infinita y sobrecargar la memoria.

Por otro lado, la iteratividad es una técnica en la que se utiliza un bucle para repetir una secuencia de instrucciones hasta que se cumpla una determinada condición. Es una forma eficiente de procesar grandes cantidades de datos y automatizar tareas repetitivas. Sin embargo, el uso excesivo de bucles puede ralentizar el rendimiento del programa y aumentar la complejidad del código.

En resumen, tanto la recursividad como la iteratividad son herramientas valiosas para cualquier programador y deben usarse según las necesidades específicas del problema que se está tratando de resolver.

COMPETENCIA ESPECÍFICA

Comprende y aplica de forma eficiente la recursividad basándose en algoritmos iterativos.

MARCO TEORICO

2.1 Definición.

La recursividad es una técnica de programación que consiste en una función que se llama a sí misma, con el objetivo de dividir un problema complejo en subproblemas más pequeños y manejables. Esta técnica utiliza la retroalimentación, ya que cada llamada a la función recursiva realiza una operación en una parte del problema original, y se utiliza el resultado de esa operación para resolver el problema completo.

2.2 Procedimientos Iterativos.

Los procedimientos iterativos son una técnica de programación que consiste en repetir una secuencia de instrucciones un número determinado de veces o hasta que se cumpla una determinada condición, con el fin de lograr un resultado específico.

2.3 Procedimientos Recursivos.

Debemos definir el tamaño máximo de la pila, además de un apuntador al último elemento insertado en la pila el cual denominaremos SP. La representación gráfica de una pila es la siguiente:

2.4 Cuadro Comparativo de procedimientos Iterativos vs Recursivos.

Recursividad	Iteraccion
ventajas	**ventajas**
No es necesario definir la secuencia de pasos exacta para resolver el	su estructura repetitiva
Problemas más fáciles de resolver que con estructuras iterativas.	condición de terminación.
Soluciones más simples.	Solución iterativa
Soluciones elegantes.	
desventajas	**desventaja**
Podria ser menos eficiente. Sobrecarga asociada con las llamadas a subalgoritmos	La complejidad de la solución
Una simple llamada puede puede generar un gran número de llamadas recursivas. (Fact(n) generan n llamadas recursivas)	No garantiza el éxito de su uso
El valor de la recursividad reside en el hecho de que se puede usar para resolver problemas sin fácil solución iterativa. La ineficiencia inherente de algunos algoritmos recursivos.	costos ocultos en su implementación , ya que se incorporan varias actividades a realizar por el equipo
¿La claridad compensa la sobrecarga?	

Análisis de Algoritmos.

El análisis de algoritmos es una técnica fundamental en la ciencia de la computación, que se utiliza para evaluar el rendimiento y la eficiencia de un algoritmo. Esta técnica se basa en la medición del tiempo y del espacio necesario para ejecutar un algoritmo, y en la identificación de los factores que pueden afectar su rendimiento.

El objetivo del análisis de algoritmos es proporcionar una medida cuantitativa de la eficiencia de un algoritmo, de modo que se pueda determinar si es adecuado para resolver un problema específico, y si es posible mejorarlo o encontrar un algoritmo más eficiente. El análisis de algoritmos se utiliza en una amplia variedad de campos, desde la investigación en inteligencia artificial hasta el diseño de algoritmos para la optimización de redes.

El análisis de algoritmos se basa en diferentes técnicas, como la complejidad temporal y espacial, la notación asintótica, la teoría de la información y la teoría de la complejidad computacional. La complejidad temporal se refiere al tiempo necesario para ejecutar un algoritmo, mientras que la complejidad espacial se refiere a la cantidad de espacio en memoria necesario para ejecutarlo. La notación asintótica se utiliza para describir el comportamiento del algoritmo a medida que el tamaño del problema se aproxima al infinito.

La teoría de la información se utiliza para medir la cantidad de información necesaria para describir el problema y la solución, mientras que la teoría de la complejidad computacional se centra en la clasificación de los algoritmos según su complejidad. En general, el análisis de algoritmos es una herramienta fundamental en la optimización del rendimiento y la eficiencia de los algoritmos, y su estudio es esencial para cualquier persona que desee comprender y desarrollar soluciones informáticas eficientes.

Complejidad en el Tiempo.

La complejidad en el tiempo es una medida de la eficiencia de un algoritmo, que se refiere a la cantidad de tiempo que tarda en ejecutarse en función del tamaño de la entrada. Esta medida es fundamental en el análisis de algoritmos, ya que permite evaluar y comparar el rendimiento de diferentes algoritmos para resolver un mismo problema.

La complejidad en el tiempo se representa mediante una función que indica la cantidad de operaciones o instrucciones que un algoritmo necesita para procesar una entrada de tamaño n. Por lo general, esta función se expresa en términos de la notación asintótica, que describe el comportamiento del algoritmo para entradas grandes.

La complejidad en el tiempo se clasifica en diferentes categorías, como la complejidad constante, lineal, logarítmica, cuadrática y exponencial, según el crecimiento de la función de complejidad a medida que n aumenta. La complejidad constante significa que el tiempo de ejecución no depende del tamaño de la entrada, mientras que la complejidad exponencial significa que el tiempo de ejecución aumenta de manera exponencial a medida que n aumenta.

La complejidad en el tiempo es una medida crítica para evaluar la eficiencia de los algoritmos y seleccionar el algoritmo más adecuado para resolver un problema específico. Además, también se utiliza para analizar y optimizar los algoritmos existentes y para desarrollar nuevos algoritmos más eficientes.

Complejidad en el Espacio.

La complejidad en el espacio es una medida de la cantidad de memoria necesaria para ejecutar un algoritmo. Esta medida es fundamental en el análisis de algoritmos, ya que permite evaluar y comparar el uso de la memoria de diferentes algoritmos para resolver un mismo problema.

La complejidad en el espacio se representa mediante una función que indica la cantidad de memoria que un algoritmo necesita para procesar una entrada de tamaño n. Por lo general, esta función se expresa en términos de la notación asintótica, que describe el comportamiento del algoritmo para entradas grandes.

La complejidad en el espacio se clasifica en diferentes categorías, como la complejidad constante, lineal, logarítmica, cuadrática y exponencial, según el crecimiento de la función de complejidad a medida que n aumenta. La complejidad constante significa que la cantidad de memoria utilizada por el algoritmo es constante e independiente del tamaño de la entrada, mientras que la complejidad exponencial significa que la cantidad de memoria utilizada aumenta de manera exponencial a medida que n aumenta.

La complejidad en el espacio es una medida crítica para evaluar la eficiencia de los algoritmos y seleccionar el algoritmo más adecuado para resolver un problema específico. Además, también se utiliza para analizar y optimizar los algoritmos existentes y para desarrollar nuevos algoritmos más eficientes.

Eficiencia de los Algoritmos.

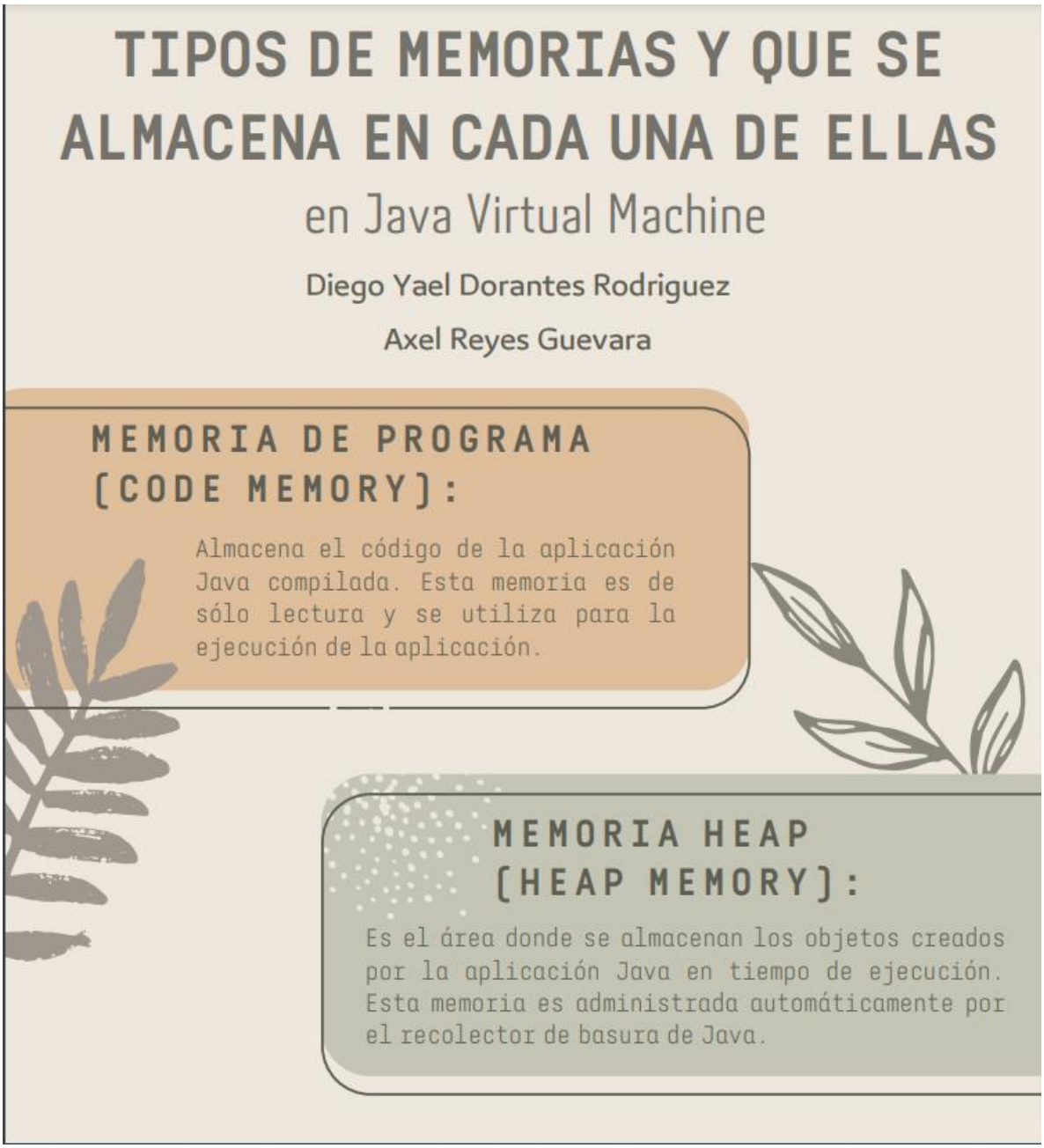
La eficiencia de un algoritmo se refiere a su capacidad para resolver un problema de manera efectiva y en un tiempo razonablemente corto. La eficiencia de un algoritmo se evalúa en términos de su complejidad en el tiempo y en el espacio, que son medidas de la cantidad de tiempo y memoria necesarios para ejecutar el algoritmo.

La eficiencia de los algoritmos es fundamental en la informática, ya que permite evaluar y comparar diferentes soluciones para un mismo problema y seleccionar la solución más adecuada. Además, también se utiliza para analizar y optimizar los algoritmos existentes y para desarrollar nuevos algoritmos más eficientes.

La eficiencia de un algoritmo se puede mejorar de diversas maneras, como mediante la optimización del código, la selección de estructuras de datos adecuadas y la aplicación de técnicas de programación avanzadas, como la programación dinámica o la división y conquista.

Es importante tener en cuenta que la eficiencia de un algoritmo no es la única consideración al seleccionar una solución para un problema. Otros factores, como la simplicidad del código, la facilidad de mantenimiento y la escalabilidad, también pueden influir en la decisión final.

Infografía y trabajo didáctico.



MEMORIA PERMGEN (PERMANENT GENERATION):

Es una sección de la memoria de la JVM que almacena metadatos sobre las clases y los métodos que se utilizan en la aplicación. A partir de Java 8, esta memoria fue eliminada y se reemplazó por el espacio de metadatos (Metaspace).

MEMORIA STACK (STACK MEMORY):

Cada hilo de la aplicación Java tiene su propia pila de ejecución. La pila almacena información sobre los métodos que se están ejecutando actualmente y las variables locales asociadas.

MEMORIA NATIVE (NATIVE MEMORY):

Es la memoria utilizada por la JVM para almacenar información no gestionada por Java. Esto puede incluir bibliotecas nativas, archivos mapeados en memoria, entre otros.

Explicación y ejemplo del uso de las memorias

Ejemplo

```
public static boolean esPalindromo(String texto, int i){
    int largo = texto.length();
    int mitad = largo/2;
    if(i<mitad && texto.charAt(i) == texto.charAt((largo-1)-i))
      return esPalindromo(texto, i+1);
    return (i==mitad);
  }
```

public static boolean esPalindromo(String texto, int i){
int largo = texto.length();
int mitad = largo/2;

Recibe una palabra junto con un valor 0 que se utiliza como un contador
Después ala variable largo se le asigna la cantidad de caracteres de la palabra,
a la variable mitad le asigna la mitad de los caracteres totales de la palabra.

if(i<mitad && texto.charAt(i) == texto.charAt((largo-1)-i))

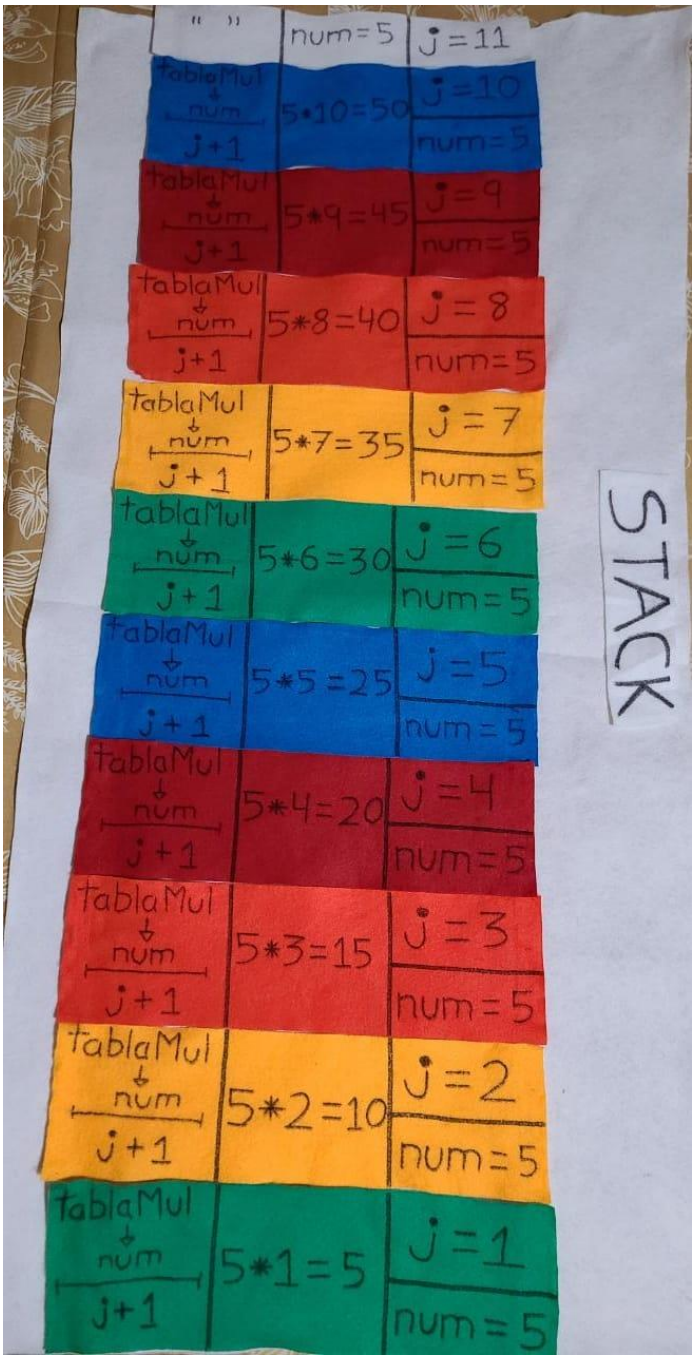
Se compara que el contador sea menor que la variable mitad y que la letra en la posición i de la palabra sea igual a la invertida de la posición i de la palabra.

return esPalindromo(texto, i+1);
return (i==mitad);

En el caso que sea verdadero retornara el llamado a si mismo incrementando la variable i de la palabra.

En caso de que sea falso retornara true o false dependiendo de si la i es igual a la variable de la mitad.

Siendo i igual a mitad significa que todas las condiciones hasta la mitad de la palabra fueron iguales dando a entender que la palabra se escribe de la misma forma como de izquierda a derecha y viceversa demostrando que la palabra es un palindromo.


" "
num=5
j=11
tablaMul
num
j+1
5*10=50
j=10
num=5
tablaMul
num
j+1
5*9=45
j=9
num=5
tablaMul
num
j+1
5*8=40
j=8
num=5
tablaMul
num
j+1
5*7=35
j=7
num=5
tablaMul
num
j+1
5*6=30
j=6
num=5
tablaMul
num
j+1
5*5=25
j=5
num=5
tablaMul
num
j+1
5*4=20
j=4
num=5
tablaMul
num
j+1
5*3=15
j=3
num=5
tablaMul
num
j+1
5*2=10
j=2
num=5
tablaMul
num
j+1
5*1=5
j=1
num=5
STACK

MATERIAL Y EQUIPO

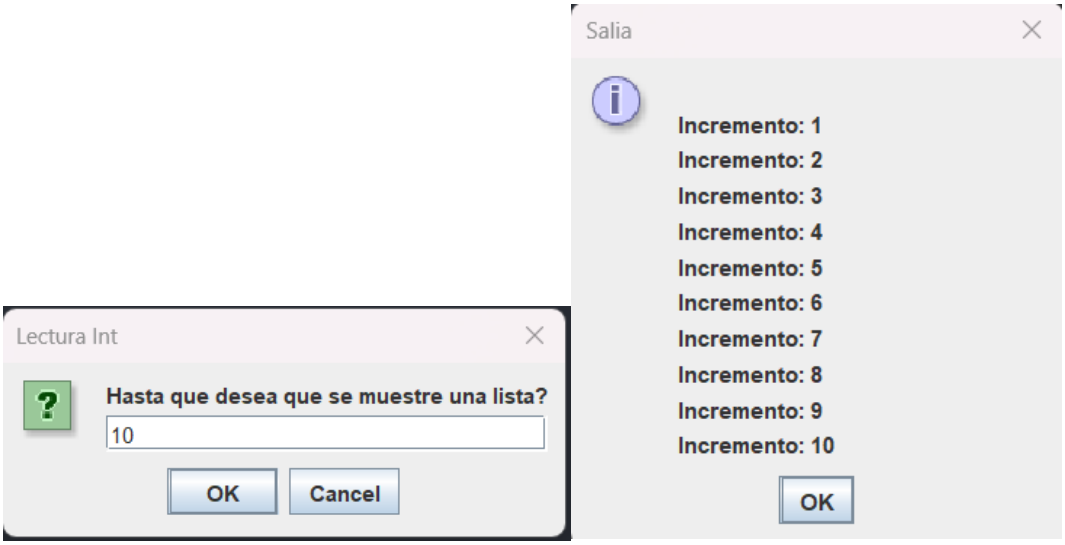
- Computadora con Windows 7 en adelante
- IDE utilizado (IntelliJ IDEA 2023.1 (Comunity Edition)
- OpenJDK 1.17 UTF-8)
- Memorias USB

DESARROLLO DE LA PRÁCTICA

Durante el desarrollo de las actividades se realizaron algoritmos en forma iterativa, dichos algoritmos fueron convertidos a su forma recursiva posteriormente, eh aquí unos breves ejemplos.

```java
public static void usoDoWhile(int n)
{
    String cad = "";
    int j = 1; //Valor Inicial
    do //Condicion True
    {
        cad+="\nIncremento: " + j;
        j++; //Incremento
    }while(j<=n);
    ToolsList.imprimePantalla(cad);
}
```

Este método iterativo recibe un valor de tipo entero, el cual usara como limite para imprimir una serie de números en orden sucesivo.

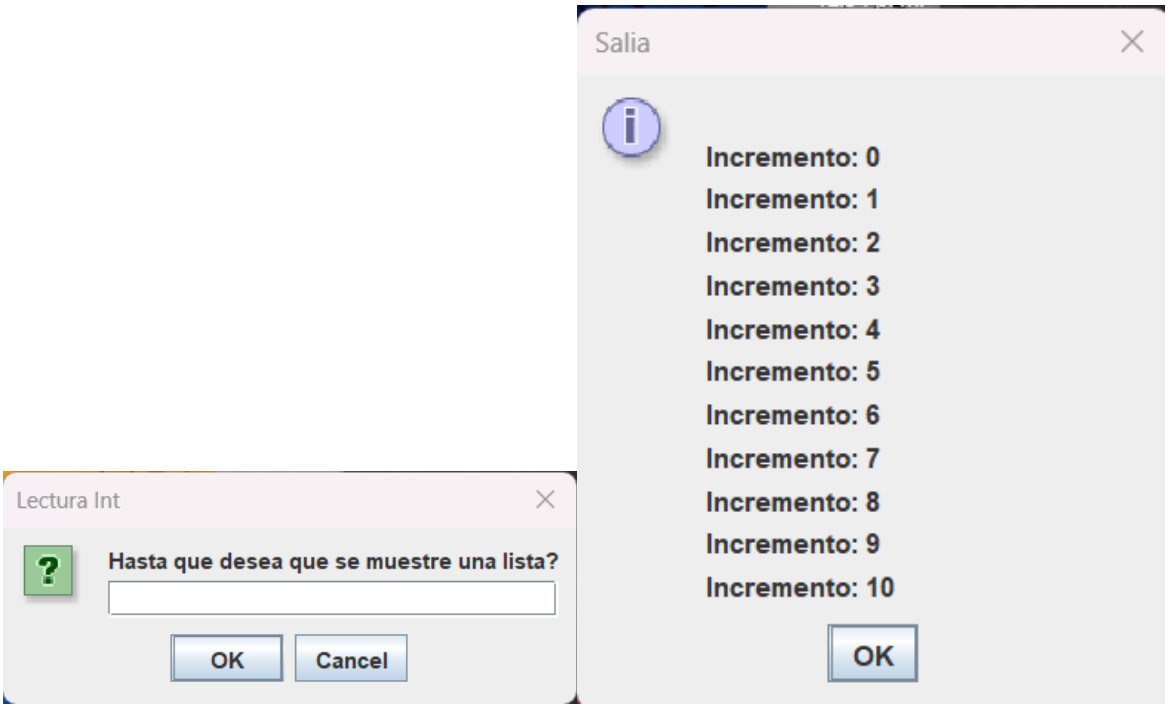


Su forma recursiva se compone de la siguiente manera.

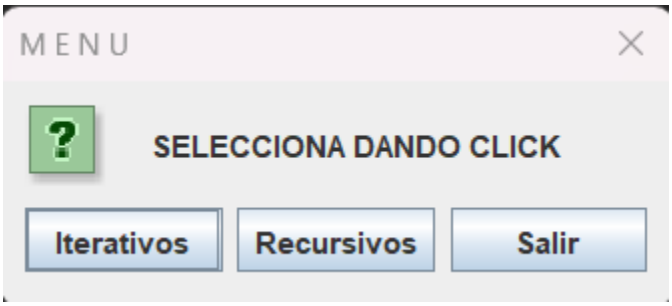
```java
public static String funcionIterativa(int j, int n) {
    if (j <= n) {
        return "\nIncremento: " + j + funcionIterativa(j + 1, n);
    } else {
        return "";
    }
}
```

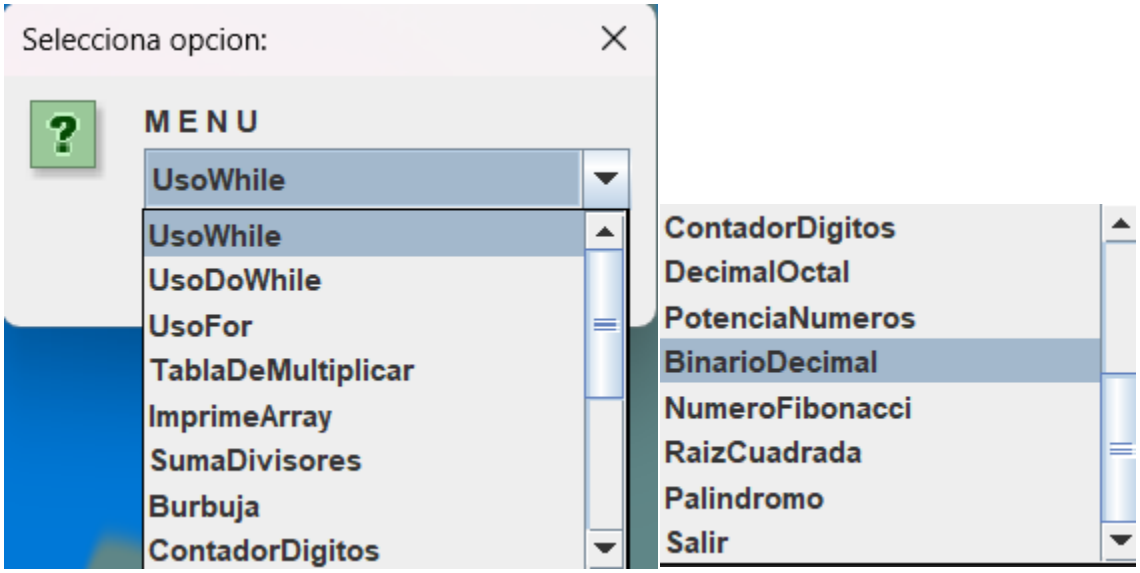
En este método se reciben 2 parametros, uno de ellos, siendo j indicara el contador utilizado durante el proceso recursivo, y el otro, siento n indicara el limite de ciclos realizados durante el método.

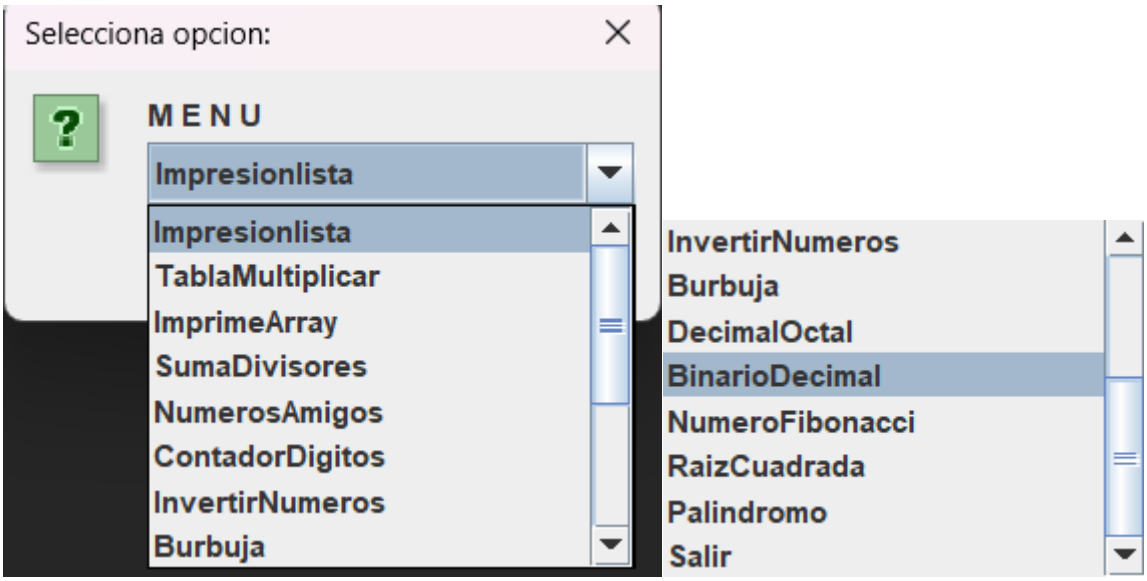
Este método funciona tal que recibe 2 parametros, j y n, a lo cual posteriormente comparara si j es menor o igual a n, utilizando una función “if”, siendo verdadera la condición retornara la palabra “Incremento” seguido de el valor asignado a j, posteriormente el método se llamara a si mismo asignándole a la variable j su valor almacenado + 1 dando a entender que este valor esta incrementando en 1.



Todos los ejercicios realizados durante el transcurso de la unidad 2:Recursividad se implementaron en un menu final, el cual contiene funciones tanto recursivas como iterativas.







CONCLUSION

En conclusión, la recursividad es una técnica de programación que permite a una función llamar a sí misma para resolver un problema de manera recursiva. Aunque la recursividad puede ser más difícil de entender y depurar que los algoritmos iterativos, es una herramienta poderosa y elegante para resolver problemas complejos.

La recursividad se utiliza comúnmente en la programación de estructuras de datos, como árboles y grafos, donde la solución de un problema puede requerir la resolución de subproblemas similares más pequeños. Además, la recursividad también se utiliza en algoritmos de ordenamiento, búsqueda y división y conquista.

Es importante tener en cuenta que la recursividad puede tener un impacto en la eficiencia del algoritmo, ya que puede requerir más memoria y tiempo de ejecución que los algoritmos iterativos. Por lo tanto, es importante evaluar cuidadosamente la eficiencia de un algoritmo recursivo y considerar alternativas iterativas cuando sea necesario.

En resumen, la recursividad es una técnica útil para resolver problemas complejos en programación y es esencial para el desarrollo de algoritmos eficientes y elegantes-

BIBLIOGRAFIAS

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms. MIT press.

Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2006). Algorithms. McGraw-Hill Higher Education.

Castillo, O. (2011). Recursividad: una técnica para resolución de problemas en programación. [Documento PDF]. Recuperado de https://www.uv.mx/personal/ocastillo/files/2011/04/Recursividad.pdf

Microsoft. (2021). Funciones recursivas. Recuperado el 22 de abril de 2023, de https://learn.microsoft.com/es-es/cpp/c-language/recursive-functions?view=msvc-170