

Tecnológico Nacional de México

Campus Orizaba

ESTRUCTURA DE DATOS

Carrera:

Ingeniería en sistemas computacionales

Tema 2: RECURSIVIDAD

Alumnos:

Tezoco Cruz Pedro-20011328

Flores Domínguez Ángel Gabriel-21010951

Grupo:3g2B

Fecha 22/04/23

INTRODUCCION.

La recursividad es una técnica utilizada en programación en la que una función se llama a sí misma para resolver un problema. Esto significa que la función se divide en subproblemas más pequeños hasta que se llega a una solución. Los procedimientos recursivos son aquellos que utilizan la técnica de recursividad para resolver un problema. Estos procedimientos se dividen en subproblemas más pequeños que se resuelven llamando a la misma función de manera recursiva.

Por otro lado, los procedimientos iterativos son aquellos en los que una acción se repite una cantidad determinada de veces. Estos procedimientos utilizan estructuras de control como ciclos o bucles para repetir la acción. Son más eficientes en términos de uso de memoria y son más sencillos de entender y depurar que los procedimientos recursivos.

En cuanto a un cuadro comparativo, los procedimientos iterativos son ideales para problemas que se pueden resolver de manera iterativa, utilizan estructuras de control como ciclos o bucles, repiten una acción una cantidad determinada de veces, son más eficientes en términos de uso de memoria y son más sencillos de entender y depurar. Por otro lado, los procedimientos recursivos son ideales para problemas que se pueden dividir en subproblemas más pequeños, utilizan la técnica de recursividad, se dividen en subproblemas más pequeños que se resuelven llamando a la misma función de manera recursiva, pueden ser menos eficientes en términos de uso de memoria y pueden ser más difíciles de entender y depurar.

En este reporte sobre estos conceptos, se podría incluir una definición clara de cada uno, ejemplos de uso en la programación y una comparación entre los dos en términos de eficiencia, facilidad de uso y situaciones en las que uno sería más adecuado que el otro. También se podrían incluir consideraciones adicionales, como la complejidad temporal y espacial de cada técnica, así como ejemplos de problemas que pueden ser resueltos tanto de manera iterativa como recursiva.

COMPETENCIA ESPECIFICA.

Te permitirá resolver problemas de manera más sencilla y elegante en el futuro. Al dominarla, podrás escribir código más claro y fácil de entender, lo que te permitirá ser más eficiente en tu trabajo como programador. Además, la recursividad te ayudará a comprender mejor cómo funcionan los programas y cómo se resuelven los problemas en la programación.

MARCO TEORICO.

2.1 Definición de recursividad.

La recursividad es una técnica de programación que básicamente te permite que una función se llame a sí misma para resolver un problema. Esto es útil cuando tienes problemas complejos que pueden ser divididos en problemas más pequeños, y cada subproblema se puede resolver de la misma manera que el problema original.

Esta técnica es como tener un set de muñecas rusas, donde cada muñeca más pequeña está dentro de una más grande. Cada muñeca es un subproblema que se resuelve utilizando la misma técnica recursiva, hasta llegar a la muñeca más pequeña que es el problema original resuelto.

La recursividad es utilizada en muchos campos de la informática, como la programación, la teoría de la computación y la inteligencia artificial. Es una técnica poderosa que te permitirá resolver problemas complejos de manera más eficiente y elegante. Además, te ayudará a entender mejor cómo funcionan los programas y cómo se resuelven los problemas en la programación.

2.2 Procedimientos iterativos.

Los procedimientos iterativos son una técnica de programación que se basa en la repetición de un conjunto de instrucciones hasta que se cumpla una determinada condición. Esta técnica es ampliamente utilizada en la programación, especialmente en el desarrollo de algoritmos y estructuras de datos.

Los procedimientos iterativos están fundamentados en la idea de que algunos problemas se pueden resolver de manera repetitiva mediante la aplicación de una serie de instrucciones. A través de la repetición de estas instrucciones, se pueden realizar tareas de manera más eficiente y sencilla. Por ejemplo, un procedimiento iterativo puede utilizarse para recorrer un conjunto de datos y realizar una determinada operación en cada elemento. En lugar de escribir instrucciones para cada elemento individualmente, se pueden escribir instrucciones que se repitan para cada elemento, lo que permite automatizar el proceso y ahorrar tiempo.

Además, los procedimientos iterativos permiten realizar tareas complejas que requieren muchas iteraciones. Por ejemplo, pueden utilizarse para calcular funciones matemáticas o para simular el comportamiento de sistemas complejos.

2.2 Procedimientos recursivos.

Los procedimientos recursivos son una técnica de programación que se basa en la llamada a una función a sí misma para resolver un problema. Esta técnica es muy útil para resolver problemas que se pueden descomponer en subproblemas más pequeños del mismo tipo. La idea central es dividir el problema en subproblemas más pequeños hasta que se llegue a un problema lo suficientemente simple para ser resuelto de manera directa. Los procedimientos recursivos se utilizan en muchos campos de la programación y son muy flexibles.

2.3 Cuadro comparativo de procedimiento iterativos vs procedimientos recursivos.

	PROCEDIMIENTOS ITERATIVOS	PROCEDIMIENTOS RECURSIVIDAD
DEFINICION	Un proceso que se repite	Un proceso que se llama a sí mismo
USO DE MEMORIA	Requiere menos memoria	Requiere más memoria
MANTENIMIENTO DEL ESTADO	Utiliza variables	Utiliza pila de llamadas
VELOCIDAD	Generalmente más rápida	Generalmente más lenta
FACILIDAD DE COMPRENSION	Más fácil de entender	Puede ser más difícil de entender
EJEMPLOS	Ciclos for y while	Cálculo de factorial y Fibonacci

En resumen, los procedimientos iterativos son procesos que se repiten mediante ciclos, mientras que los procedimientos recursivos se llaman a sí mismos para resolver un problema. Los procedimientos iterativos requieren menos memoria y generalmente son más rápidos que los procedimientos recursivos, pero los procedimientos recursivos pueden ser más fáciles de entender y son útiles para resolver problemas que se pueden descomponer en subproblemas más pequeños del mismo tipo.

Temas que se pidieron anexar.

- Análisis de algoritmos

El análisis de algoritmos es una técnica fundamental en la informática y la programación, ya que permite evaluar el rendimiento y la eficiencia de los algoritmos utilizados para resolver problemas computacionales.

En la actualidad, existen diversos algoritmos para resolver un mismo problema, por lo que es importante elegir el más adecuado según las necesidades y requerimientos de cada situación. Además, el análisis de algoritmos permite identificar y corregir posibles ineficiencias en el proceso de resolución del problema, mejorando así la calidad y el tiempo de ejecución del algoritmo.

Esta técnica se utiliza en diversos campos de la informática, como la inteligencia artificial, la ciencia de datos, la optimización de procesos, la criptografía, entre otros. Por lo tanto, el análisis de algoritmos es una habilidad fundamental para cualquier profesional de la informática y es esencial para la creación de sistemas y software de alta calidad y rendimiento.

- Complejidad en el tiempo

La complejidad en el tiempo se refiere a la cantidad de tiempo que tarda un algoritmo en resolver un problema, en función del tamaño de la entrada de datos. Es una medida fundamental en el análisis de algoritmos y es importante para determinar la eficiencia y la escalabilidad de un algoritmo.

En la actualidad, la complejidad en el tiempo se utiliza en diversas áreas de la informática, como la inteligencia artificial, la ciencia de datos, la optimización de procesos, la criptografía, entre otros. Al conocer la complejidad en el tiempo de un algoritmo, se pueden seleccionar los algoritmos más adecuados según los requisitos y limitaciones de tiempo de cada situación.

Además, la complejidad en el tiempo también es esencial para la creación de sistemas y software de alta calidad y rendimiento, ya que permite identificar y corregir posibles ineficiencias en el proceso de resolución del problema, mejorando así la calidad y el tiempo de ejecución del algoritmo.

En resumen, la complejidad en el tiempo es una medida fundamental en el análisis de algoritmos, que permite evaluar la eficiencia y escalabilidad de los algoritmos utilizados para resolver problemas computacionales. Esta técnica se utiliza en diversas áreas de la informática y es esencial para la creación de sistemas y software de alta calidad y rendimiento.

- Complejidad en el espacio

La complejidad en el espacio es un concepto clave en el análisis de algoritmos y se refiere a la cantidad de memoria necesaria para la ejecución de un algoritmo en particular. Esta complejidad se mide en términos de la cantidad de memoria

adicional necesaria para la ejecución del algoritmo, en comparación con la memoria requerida por el problema en sí mismo.

La complejidad en el espacio puede ser importante en situaciones donde la memoria es limitada, como en dispositivos móviles o sistemas embebidos. Además, la complejidad en el espacio también puede influir en el rendimiento de los algoritmos, ya que una mayor complejidad puede resultar en un mayor uso de memoria y un mayor tiempo de ejecución.

Por lo tanto, comprender la complejidad en el espacio de un algoritmo es fundamental para la selección y optimización de algoritmos, así como para la planificación y diseño de sistemas informáticos que tienen limitaciones de memoria.

- Eficiencia de los algoritmos

La eficiencia de los algoritmos es un aspecto fundamental en la informática y en el diseño de software. Se refiere a la capacidad de un algoritmo para resolver un problema específico de manera rápida y con la menor cantidad de recursos posible, como tiempo de procesamiento y memoria. Una de las razones por las que la eficiencia de los algoritmos es importante es porque puede tener un gran impacto en el rendimiento general del sistema. Por ejemplo, si un algoritmo que se ejecuta en un servidor tarda demasiado en completarse, esto puede afectar el tiempo de respuesta de una aplicación y la experiencia del usuario. Además, un algoritmo ineficiente puede requerir más recursos, lo que podría afectar la escalabilidad del sistema y la capacidad de manejar grandes volúmenes de datos. Por lo tanto, es importante tener en cuenta la eficiencia de los algoritmos al diseñar y desarrollar software, para garantizar que las aplicaciones sean rápidas, escalables y eficientes en el uso de los recursos. La eficiencia también es un factor importante en la elección de algoritmos y estructuras de datos, y puede ayudar a optimizar el rendimiento de los sistemas informáticos en general.

Materiales:

- Una computadora
- Apache NetBeans

- GitLab
- Word

Desarrollo de la practica.

Esta practica se iba desarrollando diariamente en las horas de la clase, primero iniciamos con los métodos recursivos, esos como habíamos explicado, Los métodos iterativos son técnicas matemáticas que nos ayudan a encontrar soluciones numéricas aproximadas a problemas que no tienen solución exacta. ¿Cómo lo hacen? Pues, mediante iteraciones repetitivas, que nos permiten acercarnos cada vez más a la solución.

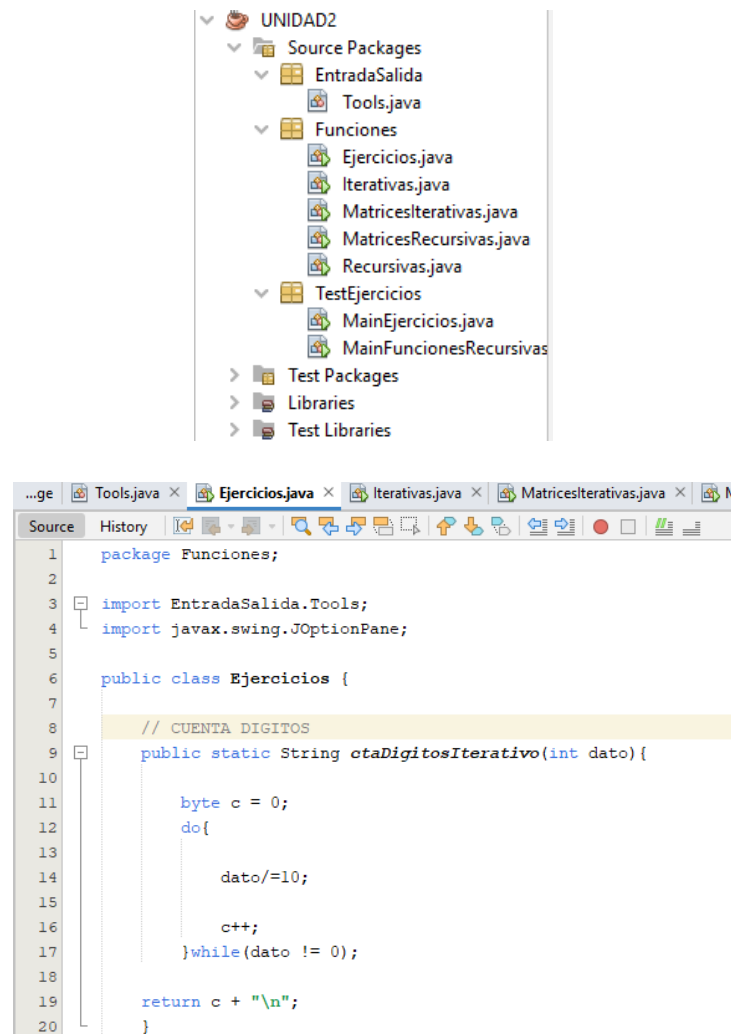
Creamos tres clases, una llamada ejercicios, otra llamada iterativos y una llamada main ejercicios, en ellas agregábamos métodos, los cuales íbamos creando a lo largo de la clase.

Después empezamos con los métodos recursivos los cuales como ya explicamos son una técnica que se usa en programación y matemáticas para resolver problemas grandes dividiéndolos en problemas más pequeños y resolviéndolos de manera repetitiva. Es como cuando tienes una tarea grande, la divides en tareas más pequeñas y las vas resolviendo una por una. Con los métodos recursivos, hacemos lo mismo pero con problemas.

Ocupamos los mismos métodos iterativos y los íbamos cambiando de modo recursivo, creabamos menus con los cuales podíamos ejecutar los métodos y ponerlos en practica, creamos 2 clases, una llamada recursivas y otra llamada main funciones recursivas, dentro de ellos pusimos los métodos recursivos necesarios.

Después empezamos con las matrices iterativas y recursivas en el cual igual agregamos métodos con sus respectivas ejecuciones.

Resultados.



Primero, se define una variable "c" que inicialmente tiene un valor de cero. Luego, se utiliza un bucle do-while para dividir el número ingresado por 10 en cada iteración, lo que nos permitirá contar la cantidad de dígitos en el número. En cada iteración, se incrementa la variable "c" en uno. El bucle continúa hasta que el número ingresado se convierte en cero, momento en el que se devuelve el valor final de "c", que representa la cantidad de dígitos en el número original. Finalmente, este valor se convierte en una cadena de texto y se devuelve a través de la función

```

23 public static double FactorialIterativo( int n){
24     byte j = 1; double fact;
25
26     if(n == 0 || n == 1) return 1;
27     else{
28         fact = 1;
29     }
30     while( j <= n){
31
32         fact*=j;
33         j++;
34     }
35
36     return fact;
37 }

```

Este código implementa una función llamada "FactorialIterativo" que calcula el factorial de un número entero positivo "n" utilizando un enfoque iterativo. En la primera línea, se declara una variable byte llamada "j" y se inicializa en 1, y una variable double llamada "fact". Luego, la función verifica si el valor de "n" es 0 o 1. Si es así, la función devuelve 1, ya que el factorial de 0 o 1 es 1.

Si "n" es mayor que 1, la función utiliza un bucle "while" para calcular el factorial. En cada iteración del bucle, se multiplica el valor actual de "fact" por el valor actual de "j", y se incrementa "j" en 1.

El bucle se repite hasta que "j" sea mayor que "n". Finalmente, la función devuelve el valor de "fact", que es el factorial de "n".

```

40 public static String binarioIterativo(int valor){
41
42     String bin = " ";
43     while(valor!= 0){
44         bin = valor%2 + bin;
45         valor /= 2;
46     }
47     return bin;
48 }

```

Este código implementa una función llamada "binarioIterativo" que convierte un número entero positivo "valor" en su representación binaria utilizando un enfoque iterativo. En la primera línea, se declara una variable de cadena llamada "bin" y se inicializa en un espacio en blanco.

Luego, la función utiliza un bucle "while" para calcular la representación binaria de "valor". En cada iteración del bucle, se divide "valor" por 2 y se toma el residuo de

la división. Este residuo se concatena con la cadena "bin" utilizando el operador de concatenación de cadenas "+". Luego, "valor" se divide por 2 y se asigna el resultado a "valor" para la próxima iteración. El bucle se repite hasta que "valor" sea igual a 0. Finalmente, la función devuelve la cadena "bin", que contiene la representación binaria de "valor".

```
//RECURSIVOS
public static String Rekursivas(int j, int n){
//para enviar mensaje
    if(j <= n){
        return j + "\n" + Rekursivas(j+1, n);
    }
    else return "";
}
```

Este código implementa una función llamada "Rekursivas" que imprime una lista de números enteros desde "j" hasta "n" utilizando un enfoque recursivo. La función toma dos parámetros enteros "j" y "n". La función utiliza una declaración "if" para verificar si "j" es menor o igual a "n". Si es así, la función devuelve una cadena que contiene el valor actual de "j" concatenado con un carácter de salto de línea "\n", seguido de una llamada recursiva a "Rekursivas" con "j+1" y "n" como argumentos.

La función se llama recursivamente hasta que "j" sea mayor que "n". En este punto, la función devuelve una cadena vacía. Por lo tanto, cuando se llama a esta función con un valor inicial de "j" y "n", la función imprimirá una lista de números enteros desde "j" hasta "n" en orden ascendente, con cada número en una nueva línea.

```
public static byte ctaDigitosRekursivo(byte j, int n){
//para enviar mensaje
if(n !=0){
    n/=10;
    return (byte) (ctaDigitosRekursivo((byte) (j+1), n));
}
return j;
}
```

Este código implementa una función llamada "ctaDigitosRekursivo" que cuenta el número de dígitos de un número entero positivo "n" utilizando un enfoque recursivo.

La función toma dos parámetros, un byte "j" inicializado en 0 y un número entero "n". La función utiliza una declaración "if" para verificar si "n" no es igual a 0. Si es así, "n" se divide por 10 para eliminar el último dígito y se llama recursivamente a la función "ctaDigitosRecursivo" con "j+1" y el nuevo valor de "n" como argumentos.

La función se llama recursivamente hasta que "n" sea igual a 0. En este punto, la función devuelve el valor actual de "j", que representa el número de dígitos en "n". El valor de retorno se convierte en un byte antes de ser devuelto. Por lo tanto, cuando se llama a esta función con un número entero positivo "n", la función devuelve el número de dígitos en "n".

```
public static int FactorialRecursivo( byte j, int num){
    double fact = 0;
    if(num == 0 || num == 1) return 1;
    else fact=1 ;
    if( j <= num){
        return j*FactorialRecursivo((byte)(j+1), num);
    }
    return (int)fact;
}
```

Este código implementa una función llamada "FactorialRecursivo" que calcula el factorial de un número entero positivo "num" utilizando un enfoque recursivo. La función toma dos parámetros, un byte "j" inicializado en 1 y un número entero "num". La función utiliza una declaración "if" para verificar si "num" es igual a 0 o 1. Si es así, la función devuelve 1, ya que el factorial de 0 o 1 es 1. De lo contrario, la función inicializa una variable "fact" en 1. Luego, la función utiliza una declaración "if" para verificar si "j" es menor o igual a "num". Si es así, la función llama recursivamente a la función "FactorialRecursivo" con "j+1" y "num" como argumentos y multiplica el resultado por "j".

La función se llama recursivamente hasta que "j" sea mayor que "num". En este punto, la función devuelve el valor actual de "fact", que representa el factorial de "num". El valor de retorno se convierte en un entero antes de ser devuelto. Por lo tanto, cuando se llama a esta función con un número entero positivo "num", la función devuelve el factorial de "num".

```

public static String binarioRecursivo(byte j, int valor){
    String bin = "";
    if(j <= valor){
        binarioRecursivo((byte) (j+1), valor%2 );
        bin = valor%2 + bin;
        valor /= 2;
        return bin + binarioRecursivo((byte) (j+1), valor);
    }
    else return null;
}

```

Este código implementa una función llamada "binarioRecursivo" que convierte un número entero "valor" en su representación binaria utilizando un enfoque recursivo. La función toma dos parámetros, un byte "j" inicializado en 1 y un número entero "valor". La función utiliza una declaración "if" para verificar si "j" es menor o igual a "valor". Si es así, la función llama recursivamente a la función "binarioRecursivo" con "j+1" y el resultado de "valor % 2" como argumentos.

Luego, la función agrega el resultado de "valor % 2" al principio de la cadena "bin" y divide "valor" por 2. La función utiliza una declaración "return" para devolver la concatenación de la cadena "bin" y la llamada recursiva a la función "binarioRecursivo" con "j+1" y el nuevo valor de "valor" como argumentos. La función se llama recursivamente hasta que "j" sea mayor que "valor". En este punto, la función devuelve la cadena "bin". Por lo tanto, cuando se llama a esta función con un número entero "valor", la función devuelve la representación binaria de "valor" como una cadena de caracteres.

```

public static String DBinaRecursivo (int n){
    String bin = "";
    if(n !=0){
        bin = n%2 + bin;
        n/=2;
        return DBinaRecursivo(n) + bin ;
    }
    return bin;
}

```

Este código implementa una función llamada "DBinaRecursivo" que convierte un número decimal "n" en su representación binaria utilizando un enfoque recursivo. La función utiliza una cadena vacía "bin" para almacenar los dígitos binarios y una declaración "if" para verificar si "n" es diferente de 0. Si es así, la función utiliza el operador de módulo "%" para obtener el dígito binario menos significativo de "n" y

lo agrega al principio de la cadena "bin". Luego, la función divide "n" por 2 y realiza una llamada recursiva a "DBinaRecursivo" con el nuevo valor de "n" como argumento.

La función se llama recursivamente hasta que "n" sea igual a 0. En este punto, la función devuelve la cadena "bin". Por lo tanto, cuando se llama a esta función con un número entero "n", la función devuelve la representación binaria de "n" como una cadena de caracteres.

```
public static void ordenaBurbuja(int a[]){
    int i, j, aux;

    for(i= 0; i<a.length-1; i++){
        for(j =i+1 ; j < a.length; j++){
            if(a[i] > a[j]){
                //intercambio
                aux = a[i];
                a[i] = a[j];
                a[j] = aux;
            }
        }
    }
}
```

El código implementa una función llamada "ordenaBurbuja" que ordena un array de enteros utilizando el algoritmo de ordenamiento burbuja. El algoritmo compara pares de elementos adyacentes y los intercambia si están en el orden incorrecto. La función utiliza dos bucles "for" anidados para comparar y mover los elementos hasta que el array esté ordenado en orden ascendente.

```
public static void imprimeArray( int a[]){
    String cad = "";
    for (int i=0; i <= a.length-1;i++){

        cad += "[" + i + "] = " + a[i]+ "\n";
    }
    Tools.Imprime("Datos ordenados: \n " + cad);
}
```

El código implementa una función llamada "imprimeArray" que imprime los elementos de un array de enteros en orden, colocando entre corchetes el índice de cada elemento y su valor. La función utiliza un bucle "for" para recorrer el array y construir una cadena de texto que contiene la información de cada elemento. Luego,

la función utiliza la función "Imprime" del objeto "Tools" para mostrar por pantalla la cadena de texto construida.

```
public static String imprimeRecursivo(int a[], int i){
    String cad = "";
    if(i < a.length){
        cad += "[" + i + "] = " + a[i] + "\n";
        return cad + imprimeRecursivo(a, i+1);
    }
    return cad ;
}
```

El código implementa una función llamada "imprimeRecursivo" que imprime los elementos de un array de enteros en orden, colocando entre corchetes el índice de cada elemento y su valor. La función utiliza una llamada recursiva para imprimir los elementos de manera secuencial, comenzando por el elemento en la posición "i" hasta el último elemento del array.

La función toma como parámetros el array "a" y un índice "i", que indica desde qué posición debe empezar a imprimir los elementos. Si el índice "i" es menor que la longitud del array, la función construye una cadena de texto que contiene la información del elemento en la posición "i" y llama a sí misma con un valor de índice "i+1". Si "i" es igual o mayor a la longitud del array, la función devuelve la cadena de texto construida.

```
public static void ordenaBurbuja_i(int a[], int i){
    if(i < a.length-1){
        ordenaBurbuja_j(a, i, i+1);
        ordenaBurbuja_i(a, i+1);
    }
}

public static void ordenaBurbuja_j(int a[], int i, int j){
    int aux = 0;
    if(j < a.length){
        if(a[i] > a[j]){
            aux = a[i];
            a[i] = a[j];
            a[j] = aux;
        }
        ordenaBurbuja_j(a, i, j+1);
    }
}
```

Este código implementa el algoritmo de ordenación burbuja de forma recursiva utilizando dos funciones. La función `ordenaBurbuja_i` es la función principal que llama a la función `ordenaBurbuja_j` para comparar y ordenar los elementos del array. En cada iteración de `ordenaBurbuja_i`, se compara el elemento `i` del array con todos los elementos restantes, comenzando por el siguiente elemento (`j = i+1`). Si `a[i]` es mayor que `a[j]`, los dos elementos se intercambian. La función `ordenaBurbuja_j` se llama recursivamente para comparar `a[i]` con el siguiente elemento (`a[j+1]`) y continuar el proceso de ordenación. La función `ordenaBurbuja_i` se llama recursivamente con el índice `i` incrementado en 1, para ordenar el siguiente par de elementos.

```
public static String Recursivo(int j, int n){
    if(j <= n){
        return j + "\n" + Recursivas(j+1, n);
    }
    else return "";
}
```

Este código implementa una función recursiva llamada "Recursivo" que toma dos argumentos enteros "j" y "n". La función devuelve una cadena de caracteres que contiene todos los enteros en el rango [j, n], cada uno en una nueva línea. El algoritmo utiliza una estructura de control de flujo "if-else" para verificar si el valor de "j" es menor o igual al valor de "n". Si es cierto, la función devuelve la cadena de caracteres que contiene el valor de "j" concatenado con una nueva línea, seguido por la llamada recursiva de la función "Recursivo" con el argumento "j+1" y "n". Si "j" es mayor que "n", la función devuelve una cadena vacía. En resumen, la función "Recursivo" genera una lista de enteros en el rango [j, n] utilizando la recursividad.

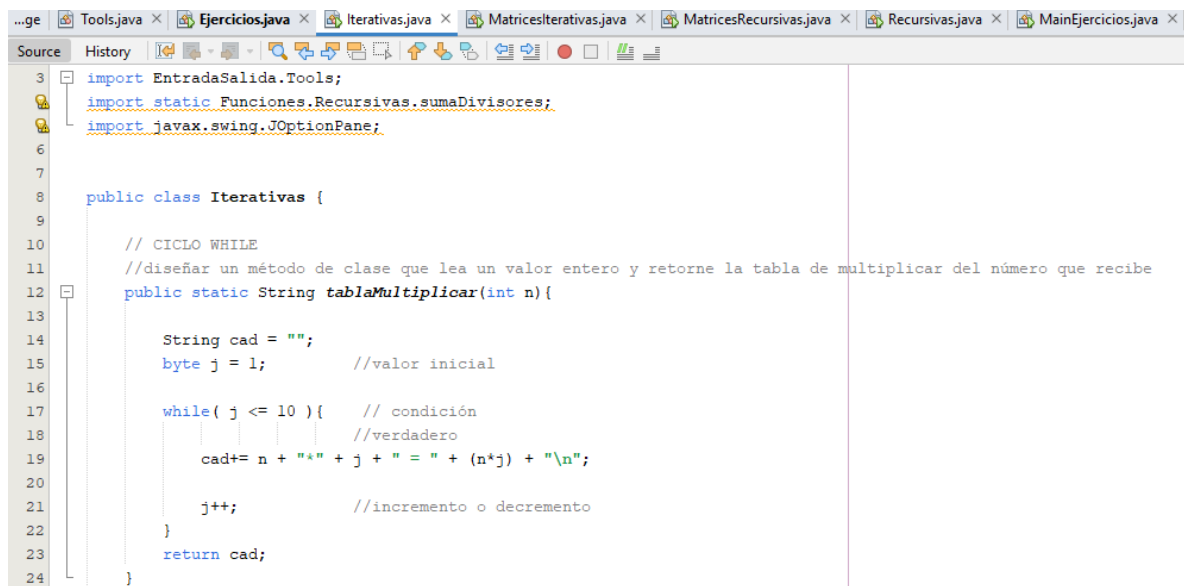
```
public static String numParesRecursivo(byte j, int i){
    String cad = "";
    if( i < 10){
        return cad += "El cubo de: " + j + " es = " + (int)Math.pow(j, 3) + "\n" + numParesRecursivo((byte)(j+2), i+1);
    }
    return "";
}
```

Este código es una función recursiva llamada "numParesRecursivo" que recibe dos parámetros: "j" y "i". La función utiliza un condicional para determinar si "i" es menor que 10. Si es verdadero, entonces la función retorna una cadena que contiene el cubo del número "j" y la llamada recursiva a la función con el parámetro "j+2" y "i+1".

Si "i" es mayor o igual a 10, entonces la función retorna una cadena vacía. En resumen, esta función genera una cadena que contiene el cubo de los números pares desde "j" hasta el décimo número par.

```
public static String numPrimosRecursivo(byte j, int i){
    String cad = "";
    if( i < 10){
        for (int x = 2; x < i; x++) {
            if( j % 2 == 0 ){
                // ...
            }
        }
        return cad += "El cubo de: " + j + " es = " + (int)Math.pow(j, 3) + "\n" + numPrimosRecursivo((byte)(j+1), i+1);
    }
    return "";
}
```

El código intenta imprimir los primeros 10 números primos a partir de un número inicial "j" utilizando una función recursiva. Sin embargo, el código actual no está implementado correctamente, ya que dentro del ciclo "for" falta una condición para detectar los números primos, y además la recursión está mal planteada. En resumen, el código no está completo y necesita ser corregido para cumplir con su objetivo.

The screenshot shows an IDE with several tabs open: Tools.java, Ejercicios.java, Iterativas.java, MatricesIterativas.java, MatricesRecursivas.java, Recursivas.java, and MainEjercicios.java. The 'Source' tab is active, displaying the code for the 'Iterativas' class. The code includes imports for 'EntradaSalida.Tools', 'Funciones.Recursivas.sumaDivisores', and 'javax.swing.JOptionPane'. The 'Iterativas' class contains a comment about a 'while' cycle and a method 'tablaMultiplicar' that takes an integer 'n' and returns a string. The method initializes a string 'cad' and a byte 'j' to 1, then enters a 'while' loop that runs as long as 'j' is less than or equal to 10. Inside the loop, it appends a line from the multiplication table to 'cad' and increments 'j'. Finally, it returns the constructed string 'cad'.

```
3 import EntradaSalida.Tools;
4 import static Funciones.Recursivas.sumaDivisores;
5 import javax.swing.JOptionPane;
6
7
8 public class Iterativas {
9
10     // CICLO WHILE
11     //diseñar un método de clase que lea un valor entero y retorne la tabla de multiplicar del número que recibe
12     public static String tablaMultiplicar(int n){
13
14         String cad = "";
15         byte j = 1; //valor inicial
16
17         while( j <= 10 ){ // condición
18             //verdadero
19             cad+= n + "*" + j + " = " + (n*j) + "\n";
20
21             j++; //incremento o decremento
22         }
23         return cad;
24     }
```

Este código define un método llamado "tablaMultiplicar" que toma un número entero "n" como argumento y devuelve una cadena de texto que representa la tabla de multiplicar de ese número del 1 al 10. El método comienza inicializando una cadena de texto vacía llamada "cad" y una variable "j" a 1. Luego, utiliza un bucle "while" para recorrer los números del 1 al 10, y en cada iteración, agrega una nueva línea a la cadena "cad" que representa una multiplicación de "n" por "j" y su resultado.

Finalmente, el método devuelve la cadena "cad" que contiene la tabla de multiplicar de "n" del 1 al 10.

```
//diseñar un método de clase que lea 15 valores enteros e imprima el mayor:
public static byte numMayor(){
    byte j = 1, dato, mayor = 0;

    while( j <= 15 ){

        dato = Tools.leeByte("Dato:");

        if( dato > mayor){
            mayor = dato;
        }
        j++;
    }

    return mayor;
}
```

Este código define un método llamado "numMayor" que no toma ningún argumento y devuelve un número entero de tipo "byte" que representa el número mayor de una lista de 15 números ingresados por el usuario. El método comienza inicializando una variable "j" y dos variables de tipo "byte" llamadas "dato" y "mayor", donde "j" es el contador del bucle, "dato" es la variable que almacena el número ingresado por el usuario y "mayor" es la variable que almacena el número mayor encontrado hasta el momento.

Luego, utiliza un bucle "while" para pedir al usuario que ingrese 15 números y en cada iteración, se lee el número ingresado por el usuario utilizando un método "leeByte" de una clase llamada "Tools". Después, se compara este número con la variable "mayor" para ver si es mayor y si es así, se actualiza el valor de "mayor" con este nuevo número. Finalmente, cuando se han ingresado los 15 números, el método devuelve la variable "mayor" que contiene el número más grande encontrado.

```
public static double Factorial( int n){
    byte j = 1; double fact;

    if(n == 0 || n == 1) return 1;
    else{
        fact = 1;
    }

    while( j <= n){

        fact*=j;
        j++;
    }

    return fact;
}
```

Este código define un método llamado "Factorial" que toma un número entero "n" como argumento y devuelve un número de punto flotante de tipo "double" que

representa el factorial de ese número. El método comienza inicializando una variable "j" a 1 y una variable "fact" a 1. Luego, utiliza una declaración "if" para comprobar si el número es igual a 0 o 1. Si es así, el método devuelve directamente el valor 1, ya que el factorial de 0 y 1 es 1.

Si el número no es igual a 0 o 1, el método utiliza un bucle "while" para calcular el factorial del número ingresado. En cada iteración, se multiplica el valor actual de "fact" por "j" y se incrementa el valor de "j". Este proceso se repite hasta que "j" es igual al valor de "n". Finalmente, el método devuelve el valor final de "fact" que contiene el factorial del número ingresado.

```
public static String tablasMultiplicarDW(int n){  
  
    String cad = "";  
    byte j = 1;  
  
    do{  
        cad+= n + "*" + j + " = " + (n*j) + "\n";  
  
        j++;  
    }while( j <= 10 );  
    return cad;  
}
```

Este código define un método llamado "tablasMultiplicarDW" que toma un número entero "n" como argumento y devuelve una cadena de texto que representa la tabla de multiplicación de "n" del 1 al 10. El método comienza inicializando una variable "cad" de tipo cadena de texto como una cadena vacía y una variable "j" de tipo "byte" a 1.

Luego, utiliza un bucle "do-while" para calcular la tabla de multiplicación. En cada iteración, se concatena a la variable "cad" una cadena que muestra la multiplicación de "n" por el valor actual de "j" y se incrementa el valor de "j". Este proceso se repite hasta que el valor de "j" es mayor que 10. Finalmente, el método devuelve la cadena de texto que contiene la tabla de multiplicación calculada.

```
public static byte numMayorDW() {  
    byte j = 1, dato, mayor = 0;  
    do{  
        dato = Tools.leeByte("Dato:");  
  
        if( dato > mayor){  
            mayor = dato;  
        }  
        j++;  
    }while( j <= 15 );  
  
    return mayor;  
}
```

Este código define un método llamado "numMayorDW" que no toma ningún argumento y devuelve un número entero de tipo "byte" que representa el número mayor de una lista de 15 números ingresados por el usuario.

El método comienza inicializando una variable "j" y dos variables de tipo "byte" llamadas "dato" y "mayor", donde "j" es el contador del bucle, "dato" es la variable que almacena el número ingresado por el usuario y "mayor" es la variable que almacena el número mayor encontrado hasta el momento. Luego, utiliza un bucle "do-while" para pedir al usuario que ingrese 15 números y en cada iteración, se lee el número ingresado por el usuario utilizando un método "leeByte" de una clase llamada "Tools". Después, se compara este número con la variable "mayor" para ver si es mayor y si es así, se actualiza el valor de "mayor" con este nuevo número.

Este proceso se repite hasta que se han ingresado los 15 números. Si se ha encontrado un número mayor, se devuelve el valor de "mayor". De lo contrario, el valor devuelto es 0, que es el valor inicial de "mayor".

```
public static double FactorialDW( int n){
    byte j = 1; double fact;

    if(n == 0 || n == 1) return 1;
    else{
        fact = 1;
    }

    do{
        fact*=j;
        j++;
    }while( j <= n);

    return fact;
}
```

Este código define un método llamado "FactorialDW" que toma un número entero "n" como argumento y devuelve el factorial de "n" como un número de punto flotante de tipo "double". El método comienza inicializando una variable "j" de tipo "byte" a 1 y una variable "fact" de tipo "double" a 1. Luego, se verifica si el valor de "n" es 0 o 1. Si es así, devuelve 1, ya que el factorial de 0 y 1 es 1. De lo contrario, se inicializa la variable "fact" a 1.

El método utiliza un bucle "do-while" para calcular el factorial de "n". En cada iteración, se multiplica el valor actual de "fact" por el valor actual de "j" y se incrementa el valor de "j". Este proceso se repite hasta que el valor de "j" es mayor que "n". Finalmente, el método devuelve el valor de "fact", que es el factorial de "n".

```

public static String tablasFor(int n){

    String cad = "";
    for(byte j = 1; j<= 10; j++ ){
        cad+= n + "*" + j + " = " + (n*j) + "\n";
    }
    return cad;
}

```

Este código define un método llamado "tablasFor" que toma un número entero "n" como argumento y devuelve una cadena de caracteres que representa la tabla de multiplicación del número "n".

El método utiliza un bucle "for" para generar la tabla de multiplicación. La variable "j" se inicializa en 1 y se repite hasta que "j" sea menor o igual que 10. En cada iteración, el método agrega a la cadena "cad" una línea que muestra la multiplicación de "n" por "j" y el resultado de esa multiplicación. Luego, la variable "j" se incrementa en 1.

Después de que se hayan completado todas las iteraciones del bucle "for", el método devuelve la cadena "cad" que contiene la tabla de multiplicación.

```

public static byte numMayorFor(){
    byte dato, mayor = 0;

    for( byte j = 1; j<= 15; j++){
        dato = Tools.leeByte("Dato:");

        if( dato > mayor){
            mayor = dato;
        }
    }
    return mayor;
}

```

Este es un método llamado "numMayorFor" que busca el número mayor de un conjunto de 15 números ingresados por el usuario. La primera línea declara dos variables de tipo byte: "dato" y "mayor", inicializando "mayor" en cero.

A continuación, se inicia un bucle "for" que se ejecutará 15 veces, con la variable "j" inicializada en 1 y aumentando en 1 en cada iteración hasta que llegue a 15.

Dentro del bucle "for", se llama al método "leeByte" de la clase "Tools" para obtener un número del usuario y asignarlo a la variable "dato". Luego, se verifica si "dato" es mayor que la variable "mayor". Si es así, se asigna "dato" a "mayor". Una vez que se completa el bucle "for", se devuelve el valor de la variable "mayor". En

resumen, este método recopila 15 números ingresados por el usuario y devuelve el número más grande de esos números.

```
public static double FactorialFor( int n){
    double fact;
    if(n == 0 || n == 1) return 1;
    else{
        fact = 1;
    }
    for(byte j = 1; j <= n; j++){
        fact*=j;
    }return fact;
}
```

El código es una función que calcula el factorial de un número entero positivo utilizando un bucle "for". En primer lugar, se comprueba si el número es igual a 0 o 1, en cuyo caso el resultado es 1. Si el número es mayor que 1, se inicializa una variable "fact" a 1 y se itera con un bucle "for" desde 1 hasta n. En cada iteración, se multiplica el valor actual de "fact" por el valor actual del contador "j". Al final, el valor final de "fact" es el factorial de n, que se devuelve como resultado.

```
public static int sumaDivisores(int dato){
    int k = 1, suma = 0;
    do{
        if (dato % k == 0) suma+=k;
        k++;
    }while(k < dato);
    return suma;
}
```

El código define una función llamada sumaDivisores que recibe un entero como parámetro y devuelve la suma de todos los divisores de dicho número.

La función utiliza un ciclo do-while que recorre todos los números desde 1 hasta uno antes del número ingresado (dato - 1). Para cada número, se verifica si es divisor del número ingresado. Si lo es, se suma a una variable acumuladora llamada suma.

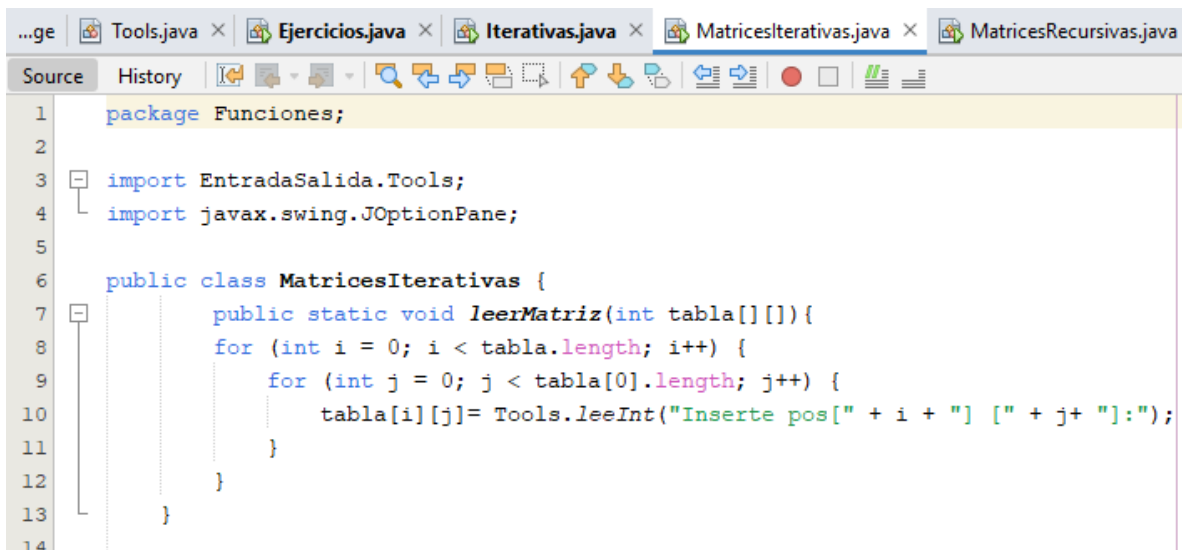
Una vez que se han revisado todos los números desde 1 hasta dato-1, se devuelve la variable suma que contiene la suma de los divisores.

```
public static String numPares() {
    String cad = "";
    byte k = 2;
    do{
        cad += k + "\n";
        k+= 2;
    }while( k <= 20);
    return cad;
}
```

Este código genera una cadena de texto que contiene los números pares del 2 al 20 separados por salto de línea. El ciclo do-while comienza con un valor inicial de k igual a 2 y continúa agregando 2 a k en cada iteración hasta que k llega a ser mayor que 20. Dentro del ciclo, se agrega k a la cadena de texto cad y se agrega un salto de línea después de cada número para que aparezcan en una nueva línea en la salida. Al final, se devuelve la cadena de texto cad.

```
public static String ctaDigitos(int dato){
    byte c = 0;
    do{
        dato/=10;
        c++;
    }while(dato != 0);
    return c + "\n";
}
```

El método ctaDigitos recibe un número entero como parámetro y cuenta la cantidad de dígitos que tiene el número. El proceso se realiza mediante un bucle do-while. En cada iteración, se divide el número por 10, lo que elimina el último dígito, y se incrementa un contador de dígitos. El bucle se repite mientras el número sea diferente de cero. Al final, el método retorna la cantidad de dígitos contados en forma de una cadena de texto.



```
1 package Funciones;
2
3 import EntradaSalida.Tools;
4 import javax.swing.JOptionPane;
5
6 public class MatricesIterativas {
7     public static void leerMatriz(int tabla[][]){
8         for (int i = 0; i < tabla.length; i++) {
9             for (int j = 0; j < tabla[0].length; j++) {
10                 tabla[i][j]= Tools.leerInt("Inserte pos[" + i + "] [" + j+ "]:");
11             }
12         }
13     }
14 }
```

El código define un método llamado "leerMatriz" que toma como parámetro una matriz de enteros "tabla". El método utiliza dos bucles "for" anidados para recorrer la matriz y solicitar al usuario que ingrese el valor para cada posición. El primer bucle "for" recorre las filas de la matriz y el segundo bucle "for" recorre las columnas de la matriz. Dentro del segundo bucle "for", se utiliza el método "leerInt" de la clase "Tools" para solicitar al usuario el valor para cada posición de la matriz,

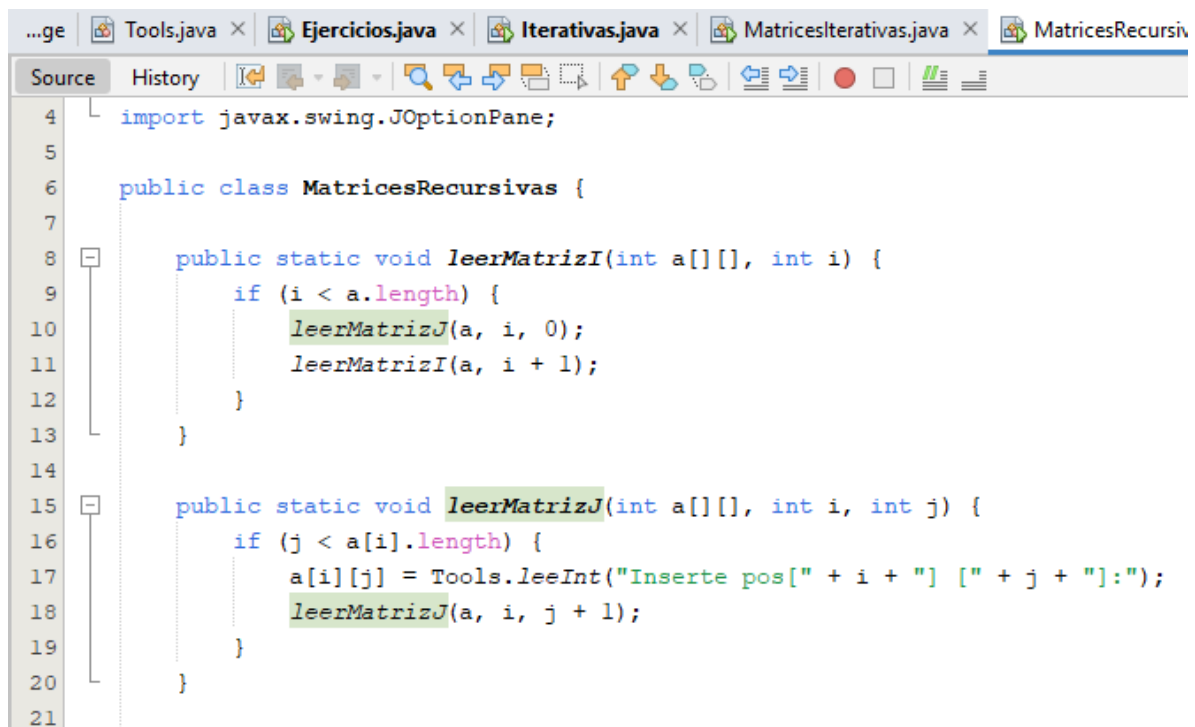
utilizando la sintaxis "Inserte pos[i][j]:". Finalmente, el método devuelve la matriz "tabla" actualizada con los valores ingresados por el usuario.

```
private static void verMatriz(int v[][]) {
    String cad = "";
    for (int i = 0; i < v.length; i++) {
        for (int j = 0; j < v[0].length; j++) {
            cad+= "[" + v[i][j] + "]" + " ";
        }
        cad += "\n";
    }

    JOptionPane.showMessageDialog(null, "Datos en la matriz;" + cad);
}
```

El código verMatriz es un método que recibe como parámetro una matriz de enteros v y muestra los datos de la matriz en un cuadro de diálogo de JOptionPane.

El método recorre la matriz con un bucle for anidado y en cada iteración, obtiene el valor en la posición i y j de la matriz y los concatena a una cadena cad junto con los caracteres [y] y un espacio en blanco entre ellos. Después de recorrer cada columna de una fila, se agrega un salto de línea a cad. Finalmente, se muestra la cadena cad en un cuadro de diálogo de JOptionPane con un mensaje que indica "Datos en la matriz:".



```
4 import javax.swing.JOptionPane;
5
6 public class MatricesRekursivas {
7
8     public static void leerMatrizI(int a[][], int i) {
9         if (i < a.length) {
10             leerMatrizJ(a, i, 0);
11             leerMatrizI(a, i + 1);
12         }
13     }
14
15     public static void leerMatrizJ(int a[][], int i, int j) {
16         if (j < a[i].length) {
17             a[i][j] = Tools.leeInt("Inserte pos[" + i + "] [" + j + "]:");
18             leerMatrizJ(a, i, j + 1);
19         }
20     }
21 }
```


Este código es una implementación de la técnica de recursión para leer una matriz de números enteros. Hay dos métodos, "leerMatrizI" y "leerMatrizJ". "leerMatrizI" se encarga de leer las filas de la matriz y llama a "leerMatrizJ" para que lea los valores de cada celda de la fila. "leerMatrizJ" se encarga de leer cada valor de la fila actual. En "leerMatrizI", si el índice actual de la fila es menor que la longitud de la matriz, se llama a "leerMatrizJ" para leer los valores de la fila actual y se llama recursivamente a "leerMatrizI" para continuar con la siguiente fila.

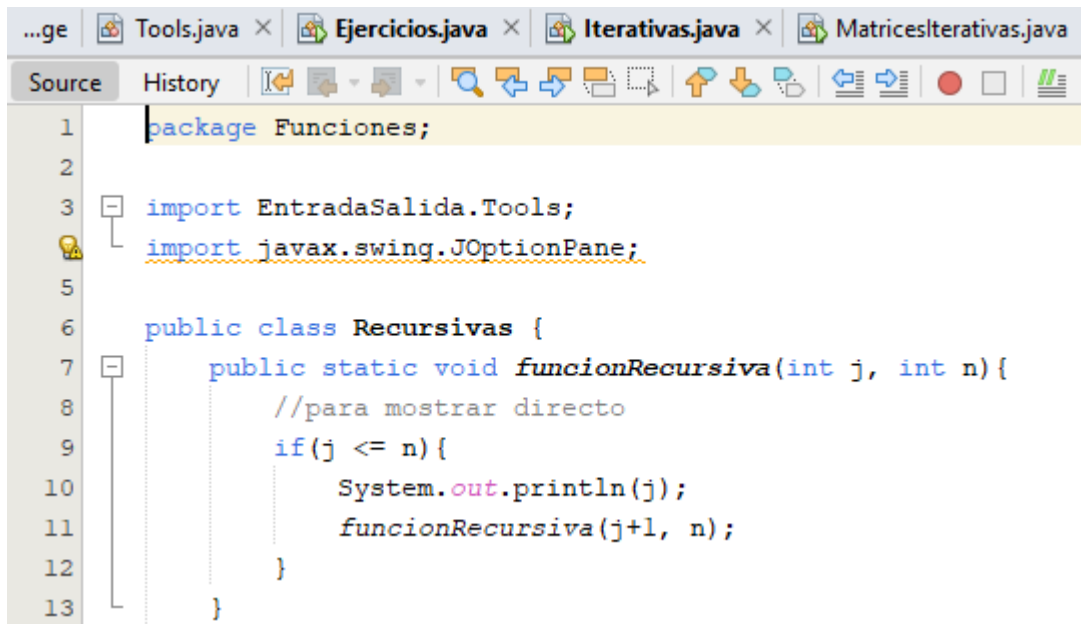
En "leerMatrizJ", si el índice actual de la columna es menor que la longitud de la fila actual de la matriz, se lee el valor de la celda actual usando la entrada estándar y se llama recursivamente a "leerMatrizJ" para continuar con la siguiente celda de la fila actual. De esta manera, los dos métodos trabajan juntos para leer todos los valores de una matriz de manera recursiva.

```
public static String verMatrizI(int a[][], int i) {
    if (i < a.length) {
        return verMatrizJ(a, i, 0) + verMatrizI(a, i + 1);
    } else {
        return "";
    }
}

public static String verMatrizJ(int a[][], int i, int j) {
    if (j < a[0].length) {
        //return "[" + i + "]" + "[" + j + "]" = "+ " + verMatrizJ(a,i,j+1);
        return "[" + a[i][j] + "]" + " " + verMatrizJ(a, i, j + 1);
    } else {
        return "\n";
    }
}
```

Este código implementa dos métodos para imprimir una matriz de enteros de manera recursiva. El método verMatrizI es el encargado de recorrer las filas de la matriz y para cada fila llama al método verMatrizJ para imprimir los elementos de esa fila. Luego de imprimir los elementos de una fila, llama a verMatrizI de manera recursiva con la siguiente fila. El método verMatrizJ es el encargado de imprimir los elementos de una fila de la matriz. Este método utiliza una variable j para recorrer los elementos de la fila y utiliza una cadena de texto cad para ir acumulando los elementos de la fila. Al finalizar de recorrer la fila, retorna la cadena cad que contiene los elementos de la fila separados por un espacio y seguidos por un salto de línea.

Cada vez que se llama a verMatrizJ, se verifica si j es menor que la longitud de la fila actual de la matriz. Si es así, se agrega el elemento correspondiente a la cadena cad y se llama de manera recursiva a verMatrizJ con el siguiente elemento. Si no, se retorna la cadena con el salto de línea para indicar que se ha terminado de imprimir la fila. En resumen, este código permite imprimir una matriz de enteros de manera recursiva, recorriendo primero las filas y luego los elementos de cada fila.



```
1 package Funciones;
2
3 import EntradaSalida.Tools;
4 import javax.swing.JOptionPane;
5
6 public class Rekursivas {
7     public static void funcionRekursiva(int j, int n){
8         //para mostrar directo
9         if(j <= n){
10             System.out.println(j);
11             funcionRekursiva(j+1, n);
12         }
13     }
```

El código implementa una función recursiva llamada "funcionRekursiva" que toma dos argumentos enteros "j" y "n". La función imprime los números enteros de "j" hasta "n" en orden ascendente en la consola de salida. Utiliza una estructura de control condicional para verificar si "j" es menor o igual que "n" y si es así, imprime el valor de "j" y luego llama a la función "funcionRekursiva" con el valor de "j+1" y "n". De esta manera, la función se llama a sí misma de manera recursiva hasta que se imprime el último valor de "n".

```
public static String Rekursivas(int j, int n){
    //para enviar mensaje
    if(j <= n){
        return j + "\n" + Rekursivas(j+1, n);
    }
    else return "";
}
```

Este código es una función recursiva que imprime una secuencia de números enteros, comenzando desde j hasta n. Si j es menor o igual a n, la función concatenará j y un salto de línea "\n" a la cadena que se va a retornar, y luego llamará a la función recursivamente con j+1 como primer parámetro y n como segundo parámetro. Este proceso se repetirá hasta que j sea mayor que n, momento en el que la función recursiva retornará una cadena vacía "". En resumen, la función retorna una cadena que contiene una secuencia de números enteros desde j hasta n, cada uno en una línea separada por un salto de línea.

```

public static int leerMayor(byte j, int mayor){
    if(j <= 5){
        int dato = Tools.leeInt("Dato:");
        if(dato > mayor) mayor = dato;
        return leerMayor((byte) (j+1), mayor);
    }
    return mayor;
}

```

Este código utiliza recursividad para leer 5 números y determinar cuál es el mayor. La función "leerMayor" toma dos argumentos: "j", que indica cuántos números se han leído hasta el momento, y "mayor", que es el número mayor actual.

La función utiliza una estructura condicional para determinar si se han leído menos de 5 números. Si es así, se lee un nuevo número con la función "Tools.leeInt" y se compara con el número mayor actual. Si el nuevo número es mayor, se actualiza la variable "mayor" con el nuevo número. Luego, la función se llama recursivamente con el argumento "j+1" para leer el siguiente número. Cuando se han leído los 5 números, la función devuelve el número mayor actual, que es el resultado final de la función.

```

public static String tablaMultiplicar(byte j, int num){
    if(j <= 10){
        return (num+ " * " + j + " = " + (num*j) + "\n" + tablaMultiplicar((byte) (j+1), num));
    }
    return "";
}

```

El código representa una función llamada tablaMultiplicar, que toma dos argumentos: j, un valor de tipo byte que representa el número actual de la tabla de multiplicar que se está generando (comenzando por 1), y num, el número que se está multiplicando. La función es recursiva y se utiliza para imprimir la tabla de multiplicar del número num desde 1 hasta 10.

La función utiliza un condicional if para controlar la recursividad. Si j es menor o igual a 10, la función concatena una cadena de texto que muestra la multiplicación actual (num * j = num*j) con un salto de línea (\n) y luego llama a sí misma con el valor de j incrementado en 1. Si j es mayor que 10, la función devuelve una cadena vacía. En resumen, la función genera una tabla de multiplicar para un número específico y la devuelve como una cadena de texto.

```

public static int Factorial( byte j, int num){
    double fact = 0;
    if(num == 0 || num == 1) return 1;
    else fact=1 ;
    if( j <= num){
        return j*Factorial((byte) (j+1), num);
    }
    return (int)fact;
}

```

Este código es una función recursiva llamada "Factorial", que calcula el factorial de un número entero positivo. El factorial de un número n se define como el producto de todos los enteros positivos desde 1 hasta n . Por ejemplo, el factorial de 5 es $5 \times 4 \times 3 \times 2 \times 1 = 120$. El código utiliza dos parámetros de entrada: "j" y "num". "j" es un contador que se utiliza en la recursión para ir multiplicando todos los números desde 1 hasta "num", y "num" es el número para el cual se quiere calcular el factorial.

La primera parte del código verifica si el número es igual a 0 o 1. En caso afirmativo, devuelve 1 ya que el factorial de 0 y 1 es 1. Si el número es mayor que 1, el código inicializa una variable "fact" en 1 y luego inicia una recursión. En cada llamada recursiva, se multiplica "j" por el resultado de la siguiente llamada recursiva, donde se incrementa "j" en 1 y se pasa el mismo valor de "num". Este proceso se repite hasta que "j" alcanza el valor de "num". Al final, se devuelve el valor de "fact" que contiene el producto de todos los enteros desde 1 hasta "num".

Sin embargo, hay un problema en este código: la variable "fact" se inicializa en 0 y nunca se actualiza dentro de la recursión, por lo que al final siempre devuelve 0. Para corregirlo, se debería cambiar la asignación "double fact = 0" a "double fact = 1".

```

public static int sumaDivisores(int k, int dato){
    int suma = 0;
    if( k< dato){
        if(dato % k == 0)
            suma+=k;
        return suma+ sumaDivisores((byte) (k+1), dato);
    }
    return suma;
}

```

Este método recursivo sumaDivisores calcula la suma de todos los divisores enteros de un número entero dato (excepto dato mismo). El método tiene dos parámetros: k que se usa para contar los divisores mientras se recorre el bucle, y dato que es el número del cual se desean calcular los divisores.

En la implementación del método se inicializa una variable llamada suma con el valor de 0. Luego, se comprueba si el valor de k es menor que dato. Si es así, se verifica si dato es divisible por k. Si lo es, se suma k a la variable suma. Finalmente, se llama a la función sumaDivisores de manera recursiva incrementando el valor de k en 1.

El proceso de recursión continúa hasta que k es igual o mayor que dato. En ese caso, la función devuelve el valor de suma. Es importante tener en cuenta que la recursión no está limitada por la cantidad de números divisores que tiene dato. Esto puede generar problemas si el número es muy grande, ya que la recursión podría sobrepasar los límites de la memoria y/o del tiempo de ejecución del programa.

```
public static String numPares(byte j, int num) {  
    String cad = "";  
    if( j <= num) {  
        return cad += j + "\n" + numPares((byte) (j+2), num );  
    }  
    return "";  
}
```

Este código es una función recursiva llamada "numPares" que recibe dos parámetros: "j" y "num". La función devuelve una cadena que contiene todos los números pares entre "j" y "num". Dentro de la función, se define una variable llamada "cad" como una cadena vacía. Luego, se comprueba si el valor de "j" es menor o igual al valor de "num". Si esto es cierto, se agrega "j" a la cadena "cad" y se agrega un salto de línea. Luego se llama de nuevo la función "numPares" con un valor de "j+2" para obtener el siguiente número par.

Este proceso se repite hasta que el valor de "j" es mayor que el valor de "num", momento en el que se devuelve la cadena vacía. En resumen, la función devuelve una cadena que contiene todos los números pares entre "j" y "num", con un salto de línea entre cada número.

```

public static byte ctaDigitos(byte c, int dato){

    if( dato != 0){
        dato/=10;
        return ctaDigitos((byte) (c+1), dato);}

    return c;
}

// METODOS RECURSIVOS INDIRECTOS

public static String Tabla(int j, int n){
    //para enviar mensaje
    if(j <= n){
        return tablaMultiplicar((byte)1, j) +"\n" + Tabla(j+1, n);
    }
    else return "";
}

```

El primer método, ctaDigitos, es una función recursiva que cuenta la cantidad de dígitos de un número entero dato. La función toma dos argumentos, el primero es un contador c inicializado en cero y el segundo es el número dato que se quiere contar. En cada llamada recursiva, se divide el número entre 10 para eliminar el último dígito, se incrementa el contador c en 1 y se llama a la función nuevamente con el número dividido entre 10 y el contador actualizado. Cuando el número llega a cero, la función devuelve el contador c que contiene la cantidad de dígitos.

El segundo método, Tabla, es una función recursiva indirecta que imprime la tabla de multiplicar del 1 al n. La función toma dos argumentos, el primer número a imprimir j y el último número a imprimir n. En cada llamada recursiva, se llama a la función tablaMultiplicar que imprime la tabla de multiplicar del número actual j y se concatena con la llamada recursiva a la función Tabla para el siguiente número j+1. Cuando el número j llega a n, la función devuelve una cadena vacía, lo que termina la recursión y devuelve la cadena completa con la tabla de multiplicar.

```

public static String Digitos(int j, int n){

    if(j <= n){
        return j + " = " + (ctaDigitos((byte) 0, j) + "\n" + Digitos(j+1, n));
    }
    else return "";
}

public static String Recur(int j){
//para enviar mensaje
    if(j >= 1){
        return j + Recur(j-1);
    }
    else return "";
}

public static String Triangulo(int j){

    if(j >= 1){
        return Recur(j) + "\n" + Triangulo(j-1);
    }
    else return "";
}
}

```

El primer código, Digitos, es una función recursiva que toma dos argumentos, j y n, y devuelve una cadena que muestra el número de dígitos de cada número desde j hasta n. La función llama a otra función recursiva ctaDigitos, que cuenta el número de dígitos en un número dado, y utiliza la recursión para llamar a sí misma con j+1 hasta que j sea mayor que n.

El segundo código, Recur, es una función recursiva que toma un argumento j y devuelve una cadena que muestra todos los números enteros desde j hasta 1, separados por un espacio. La función utiliza la recursión para llamar a sí misma con j-1 hasta que j sea menor o igual a 1.

El tercer código, Triangulo, es una función recursiva que toma un argumento j y devuelve una cadena que muestra un triángulo de números que van desde j hasta 1, separados por espacios. La función utiliza la recursión para llamar a sí misma con j-1 hasta que j sea menor o igual a 1, y llama a la función Recur para mostrar cada línea del triángulo.

Conclusiones.

Los procedimientos iterativos y recursivos son dos formas diferentes de resolver problemas mediante algoritmos. Un procedimiento iterativo utiliza un ciclo para

repetir una acción una cantidad determinada de veces, mientras que un procedimiento recursivo se llama a sí mismo repetidamente hasta que se alcanza un resultado deseado. Ambos métodos tienen sus ventajas y desventajas, y la elección de uno u otro dependerá de las necesidades específicas del problema que se esté resolviendo.

El análisis de algoritmos es una técnica para medir la eficiencia de un algoritmo. Se utiliza para determinar cuánto tiempo y memoria requiere un algoritmo para resolver un problema de tamaño dado. El análisis de algoritmos puede ayudar a identificar cuellos de botella en un algoritmo y mejorar su rendimiento.

La complejidad en el tiempo se refiere al tiempo que tarda un algoritmo en resolver un problema. La complejidad en el espacio se refiere a la cantidad de memoria que un algoritmo necesita para resolver un problema. Ambas son medidas importantes de la eficiencia de un algoritmo y pueden ser afectadas por factores como el tamaño del problema y la estructura del algoritmo.

La eficiencia de los algoritmos se refiere a la capacidad de un algoritmo para resolver un problema de manera rápida y efectiva. Un algoritmo eficiente es aquel que utiliza el menor tiempo y memoria posible para resolver un problema. La eficiencia es una consideración importante en cualquier aplicación informática, ya que puede afectar el rendimiento y la capacidad de respuesta del sistema.

En resumen, los procedimientos iterativos y recursivos son métodos comunes para resolver problemas mediante algoritmos, y el análisis de algoritmos es una técnica importante para medir su eficiencia. La complejidad en el tiempo y en el espacio son factores clave en la eficiencia de un algoritmo, y una buena comprensión de estos conceptos puede ayudar a desarrollar algoritmos más efectivos y optimizados.

Referencias

1. "Algorithms" de Robert Sedgewick y Kevin Wayne. Este es un libro de referencia para el análisis y diseño de algoritmos, que cubre una amplia variedad de técnicas y enfoques. ISBN: 9780321573513.
2. "Introduction to Algorithms" de Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein. Este libro es un clásico en el campo del análisis de algoritmos, y se enfoca en los aspectos teóricos y prácticos de los algoritmos. ISBN: 9780262033848.
3. "The Art of Computer Programming, Volumes 1-3" de Donald E. Knuth. Esta es una serie de libros clásicos que abarca muchos aspectos de la informática, incluyendo el diseño de algoritmos y la programación. ISBN: 9780321751041.
4. "Data Structures and Algorithms in Java" de Michael T. Goodrich y Roberto Tamassia. Este libro se centra en el diseño y análisis de estructuras de datos y algoritmos en el lenguaje de programación Java. ISBN: 9781118771334.
5. "Big O Cheat Sheet" de Eric Rowell. Este es un recurso en línea útil que proporciona una lista de complejidades de tiempo comunes para diferentes tipos de algoritmos. Disponible en: <https://www.bigocheatsheet.com/>.
6. "Asymptotic Notation" de Khan Academy. Este es un tutorial en línea gratuito que explica los conceptos básicos de la notación asintótica, que se utiliza para describir la complejidad de tiempo y espacio de los algoritmos. Disponible en: <https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation>.