Tecnológico Nacional de México
Campus Orizaba

Estructura de Datos

Ingeniería en Sistemas Computacionales

Tema 2:
Recursividad

Integrantes:

Castillo Solís Luis Ángel - 21010932
Muñoz Hernández Vania Lizeth – 21011009
Romero Ovando Karyme Michelle – 21011037

Grupo:
3g2B

Fecha de entrega: 27/Marzo /2023

1. Introducción

En la programación, existen dos enfoques para resolver un problema: los procedimientos iterativos y los procedimientos recursivos. Ambos enfoques tienen sus propias ventajas y desventajas, y es importante conocerlos para elegir la mejor solución para cada problema. La recursividad se basa en la idea de que un problema se puede dividir en subproblemas más pequeños y similares al problema original, hasta que se llegue a una solución base

La recursividad puede ser una herramienta poderosa para la resolución de problemas, pero también puede ser peligrosa si se utiliza de manera inadecuada. Los procedimientos recursivos pueden consumir una gran cantidad de memoria si no se implementan correctamente, lo que puede provocar errores como el desbordamiento de pila.

Dentro de la programación, la recursividad es una técnica importante, se emplea para ejecutar una llamada a una función desde la misma función. El tema de recursividad e iteración (ejecución en bucle) están muy relacionados, cualquier acción que se realiza con la recursividad se puede realizar con la iteración y viceversa. Una explicación más simple, es que podemos utilizar la recursividad para reemplazar cualquier tipo de bucle (for, while, do while). Al momento de crear un método recursivo se debe de tener en cuenta que este tiene que terminar, para eso, es importante asegurarse que no se está llamando a si mismo todo el tiempo, es ciclo debe de ser finito. En el mundo laboral no se utiliza demasiado, ya que un error en puede ser trágico en la memoria. Aun así, la gran mayoría de las veces, utilizamos recursividad para algoritmos de búsqueda u ordenación.

2. Competencia específica

Aplica la recursividad en la solución de problemas valorando su pertinencia en el uso eficaz de los recursos.

3. Marco Teórico

La recursividad es una técnica en programación que se basa en la implementación de un algoritmo que se resuelve llamándose a sí mismo. El comportamiento de una función recursiva está determinado por sus parámetros y puede describirse como un resorte que se estira y luego vuelve a su posición original. La recursividad se puede clasificar en dos tipos: simple, donde solo hay una llamada recursiva y se puede convertir en un algoritmo iterativo, y múltiple, donde hay más de una llamada recursiva en la función y también puede haber llamadas a otras funciones.

Los procedimientos iterativos y recursivos son técnicas de programación utilizadas en Java para repetir una sección de código hasta que se cumpla una condición o para dividir un problema en subproblemas más pequeños y resolverlos.

Los procedimientos iterativos utilizan bucles para repetir una sección de código hasta que se cumpla una condición. Los bucles más comunes en Java son for, while y do-while. Estos bucles son útiles para realizar tareas repetitivas, como procesamiento de datos o recorrido de una estructura de datos.

Por otro lado, los procedimientos recursivos son funciones que se llaman a sí mismas para resolver un problema dividiéndolo en subproblemas más pequeños. La recursividad es una técnica útil para resolver problemas complejos de manera elegante y concisa. Los casos base son condiciones de salida que detienen la recursión y evitan una llamada infinita a la función.

Ambos tipos de procedimientos tienen sus propias ventajas y desventajas en términos de complejidad, velocidad de ejecución y uso de memoria, y es importante seleccionar la técnica adecuada para cada situación específica.

Tema	Procedimientos Iterativos	Procedimientos Recursivos
Definición	Utilizan estructuras de control de flujo para repetir una sección de código.	Llamadas a sí mismo para resolver un problema dividiéndolo en subproblemas más pequeños.

Uso de memoria	Suelen ser más eficientes ya que no crean múltiples llamadas a sí mismos y utilizan menos memoria.	Pueden ser menos eficientes ya que crean múltiples llamadas a sí mismos y utilizan más memoria.
Complejidad	Pueden ser más complejos de implementar para ciertos problemas.	Pueden ser más fáciles de implementar para ciertos problemas.
Código	El código suele ser más fácil de entender y leer.	El código puede ser más difícil de entender y leer debido a la llamada a sí mismo del método.
Velocidad de ejecución	El rendimiento suele ser rápido si la condición de salida se cumple rápidamente.	El rendimiento puede verse afectado si la recursión continúa durante mucho tiempo.

Sin embargo, la eficiencia de un algoritmo no solo depende del tipo de procedimiento utilizado, sino también de cómo se implementa y de la complejidad del problema que se está abordando. Por lo tanto, es importante analizar el rendimiento de un algoritmo en términos de tiempo de ejecución y uso de memoria, lo que nos lleva al análisis de algoritmos.

El análisis del algoritmo es el proceso de analizar la capacidad de resolución de problemas del algoritmo en términos del **tiempo** y el **tamaño** requeridos *(el tamaño de la memoria para el almacenamiento durante la implementación)*. Sin embargo, la principal preocupación del análisis de algoritmos es el tiempo o rendimiento requerido. En general, realizamos los siguientes tipos de análisis:

- **El peor de los casos**: el número máximo de pasos dados en cualquier instancia de tamaño N.
- **El mejor caso**: el número mínimo de pasos dados en cualquier instancia de tamaño N.

- **El caso promedio**: un número promedio de pasos dados en cualquier instancia de tamaño N.
- **El amortizado**: una secuencia de operaciones aplicadas a la entrada de tamaño promediada en el tiempo.

La complejidad en el tiempo se refiere a la cantidad de tiempo que tarda un algoritmo en completar su tarea, medida en función del tamaño de la entrada. Es decir, la complejidad en el tiempo se refiere a cuánto tiempo tardará el algoritmo en ejecutarse en función del tamaño de los datos que se están procesando. Por ejemplo, un algoritmo con complejidad en el tiempo O(n) tardará más tiempo en procesar una entrada de tamaño n=10 que una entrada de tamaño n=5.

Por otro lado, la complejidad en el espacio se refiere a la cantidad de memoria que utiliza un algoritmo para completar su tarea, medida también en función del tamaño de la entrada. La complejidad en el espacio se refiere a cuánta memoria utilizará el algoritmo en función del tamaño de los datos que se están procesando. Por ejemplo, un algoritmo con complejidad en el espacio O(n) requerirá más memoria para procesar una entrada de tamaño n=10 que una entrada de tamaño n=5.

Finalmente, mencionamos algunos puntos clave a considerar en la medición de la eficiencia de un algoritmo:

- Contar cuántas operaciones necesita para encontrar la respuesta con diferentes tamaños de la entrada.
- Tiempo de ejecución: el tiempo que tarda un algoritmo en completar su tarea en función del tamaño de la entrada.
- Espacio en memoria: la cantidad de memoria que utiliza un algoritmo en función del tamaño de la entrada.
- Precisión: la precisión con la que el algoritmo resuelve el problema para una entrada dada.
- Escalabilidad: la capacidad del algoritmo para manejar entradas cada vez más grandes y seguir siendo eficiente.

- Facilidad de implementación: lo fácil o difícil que es implementar el algoritmo en código.
- Mantenibilidad: lo fácil o difícil que es mantener y actualizar el algoritmo con el tiempo.

4. Material y Equipo

El material y equipo que se necesita para llevar a cabo la práctica son:

- ✓ Computadora
- ✓ Software y versión usados
- ✓ Materiales de apoyo para el desarrollo de la práctica

5. Desarrollo de la práctica

Para llevar a cabo una práctica sobre recursividad, se pueden seguir los siguientes pasos:

1. Comprender el concepto de recursividad y cómo funciona en la programación.
2. Identificar problemas que se puedan resolver mediante recursividad.
3. Escribir un pseudocódigo o un plan de acción detallado para la solución del problema utilizando la recursividad.
4. Escribir y depurar el código en el lenguaje de programación seleccionado.
5. Probar el programa con diferentes entradas y verificar su correcto funcionamiento.
6. Analizar el rendimiento del programa y compararlo con otras soluciones posibles.

- Precauciones o advertencias.

1. La recursividad puede consumir una gran cantidad de memoria, especialmente si se utiliza en problemas que requieren muchos niveles de llamadas recursivas. Es importante tener esto en cuenta al planificar y diseñar el algoritmo.
2. Es posible que el programa entre en un ciclo infinito si no se especifican las condiciones de salida correctamente en la función recursiva. Es importante tener cuidado y asegurarse de que la función tenga una condición de salida clara y bien definida.
3. Se recomienda entender bien el problema y planificar cuidadosamente la solución antes de comenzar a escribir el código

- Anexar sus trabajos de investigación Stack, Heap, etc.

MEMORIA ESTÁTICA

La zona estática de memoria permite que los datos etiquetados como constantes y las variables globales de un programa tengan asignada una zona en la memoria necesaria durante toda la ejecución del programa.

- El tamaño de las variables no puede cambiarse durante la ejecución del programa, es asignado en forma estática.
- El tiempo de vida de las variables de la zona estática es la duración del programa.
- Estas variables son visibles para todas las funciones que estén definidas después de ellas en el archivo en que se definan.

Aunque una variable (u objeto) sea de ámbito global, no podrán ocupar almacenamiento estático:

- Los objetos que correspondan a procedimientos o funciones recursivas.
- Las estructuras dinámicas de datos tales como listas, árboles, etc.

Asignación de memoria estática:
A partir de una posición señalada por un puntero de referencia se aloja la variable X, avanzando el puntero tantos bytes como sean necesarios para almacenarla.
La asignación de direcciones de memoria se hace en tiempo de compilación y las variables globales estarán vigentes desde que comienza la ejecución del programa hasta que termina.

MEMORIA HEAP O MONTÓN

Cuando el tamaño de un objeto a colocar en memoria puede variar en tiempo de ejecución. Ejemplos de este tipo de objetos son: las listas, los árboles, etc.

Para manejar este tipo de objetos el compilador debe disponer de un área de memoria de tamaño variable, que no se vea afectada por la activación o desactivación de procedimientos.

Esta memoria se maneja vía punteros y es la responsabilidad del mismo proceso el liberar la memoria después de su uso o puede ocurrir un escape de memoria (memory leak).

Si ocurre un memory leak este va a durar mientras el proceso siga corriendo ya que al terminar de correr el sistema operativo libera toda la memoria del heap que fue pedida.

- Se guardan los atributos de objetos creados.
- La gestión de la memoria es dinámica que se pueden modificar durante la ejecución del programa.

MEMORIA EN STACK

Se utiliza para almacenar las variables denominadas automáticas, ellas existen durante la ejecución de la función (método) que las referencia.
Los argumentos y variables locales, son asignados y desasignados en forma dinámica durante la ejecución de las funciones; pero en forma automática por código generado por el compilador, el programador no tiene responsabilidad en ese proceso.

Se puede reutilizar el espacio de memoria dedicado a la función cuando ésta termina; favoreciendo el diseño de funciones recursivas y reentrantes, asociando un espacio diferente para las variables por cada invocación de la función.

- Se guardan las variables locales de tipo primitivo y se guardan las referencias de los objetos creados.

último en entrar  primero en salir

Cada función al ser invocada crea un frame en el stack o **registro de activación**, en el cual se almacenan los valores de los argumentos y de las variables locales.

Los valores de los argumentos son escritos en memoria, antes de dar inicio al código asociado a la función (método). Es responsabilidad de la función escribir valores iniciales a las variables locales, antes de que éstas sean utilizadas; es decir, que aparezcan en expresiones para lectura.

EJEMPLOS

```
1  package equipo_estructura;
2
3  public class MemoriaEstatica {
4      // variable global
5      public static int x =7;
6
7      public static void main(String[] args) {
8          //todavia no existen las variables:
9          //local1, local2, arg1 y arg2
10         x = funcion1(4,8);
11         // ya no existen los argumentos
12         //ni variables locales de funcion1();
13     }
14
15     public static int funcion1(int arg1, int arg2
16         int local1; // no inicializada
17         int local2 = 5;
18
19         local1 = arg1 + arg2 + local2;
20
21         return (local1 + 3);
22     }
23 }
```

Zona de memoria estática

x = 7

Stack

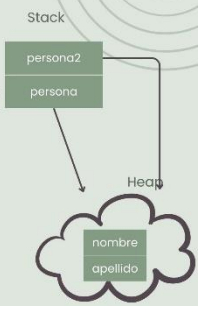
local1	?
local2	5
arg1	4
arg2	8

Al salir de la funcion1:

x = 20

Stack

```
1  package equipo_estructura;
2
3  public class MemoriaStack_Heap {
4
5      //atributos
6      private String nombre;
7      private String apellido;
8
9  //Tiempo de Ejecución
10     public static void main(String[] args){
11
12         //crea el objeto con NEW
13         MemoriaStack_Heap persona = new MemoriaStack_Heap();
14         //en memoria se asigna
15         MemoriaStack_Heap persona2 = persona;
16
17     }
18 }
```



EJEMPLO

En clase manejamos el metodo "Numeros pares" con su respectivo valor elevado al cubo.

```
public static String NumPares(byte J, int Num) {
    String Cad = "";
    if (J <= Num) {
        return Cad += "El cubo de: " + J + " = " + (int) Math.pow(J, 3)
                + "\n" + NumPares((byte)(J+2), Num);
    }
    else return "";
}
```

El tipo de memoria que se utiliza es memoria en Stack ya que las variables que se declaren dentro del metodo solo tomaran parte de la memoria hasta que termine de ejecutarse el mismo.

Se sabe que LIFO (Last In First Out) tiene 3 maneras de opreacion dentro de una pila.
La primera es apilar (Añadir un elemento a la lista).
La segunda es desapilar (Retirar un elemento de la lista).
La tercera se trata de comprobar si dicha lista se encuentra vacia.

Usando el metodo que vimos en clase podemos decir que al principio dimencionamos el tamaño de nuestra pila ya que dejamos claro que el usuario gestionara de hasta que cantidad desea guardar sus resultados, utilizando "J" como el punto de partida y "Num" como el limite del mismo.

```
public static String NumPares(byte J, int Num) {
```

Continuando tenemos que si se cumple cierta funcion se retornara a el mismo metodo hasta que deje de cumplirse la funcion o dimension que se especifica.

```
    String Cad = "";
    if (J <= Num) {
        return Cad += "El cubo de: " + J + " = " + (int) Math.pow(J, 3)
                + "\n" + NumPares((byte)(J+2), Num);
    }
```

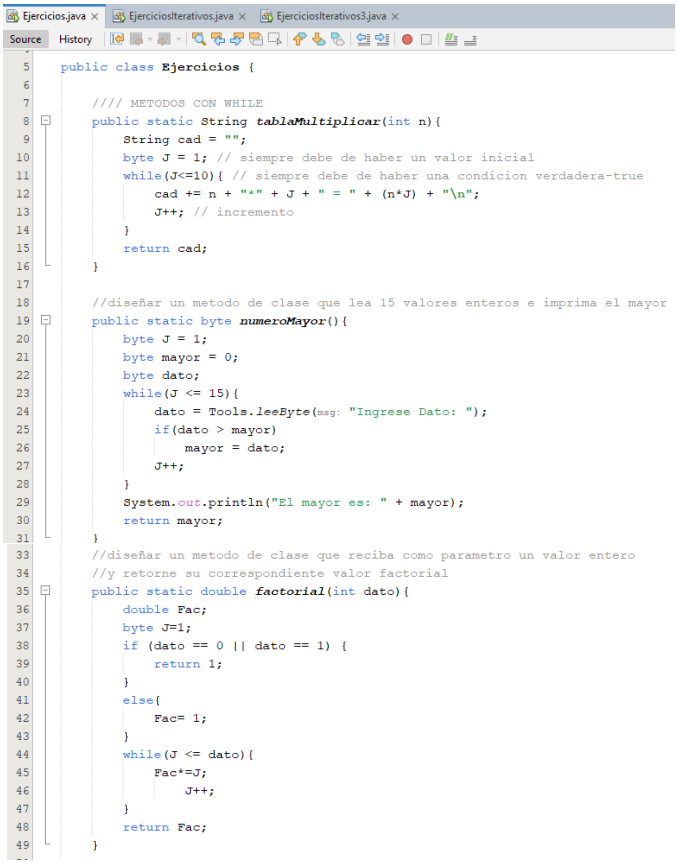
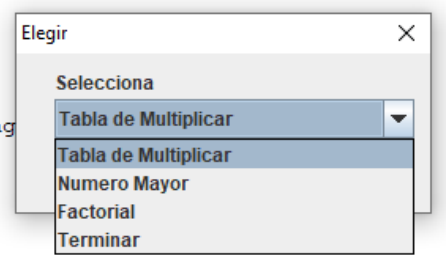
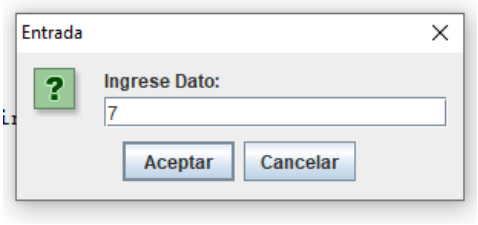
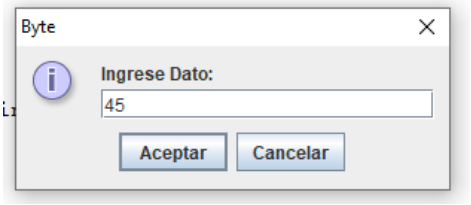
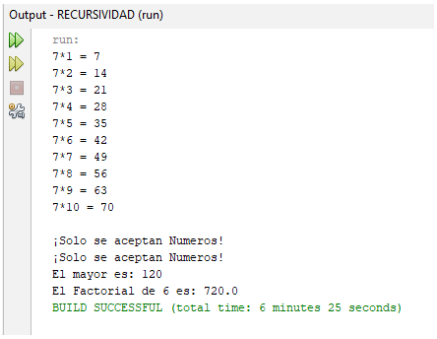
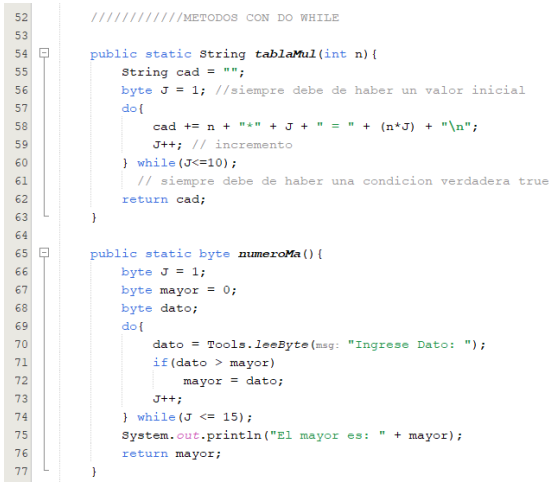
Este metodo unicamente puede apilar los datos que vayan ingresando, ya que estamos almacenando nuestros datos dentro de una cadena, si utilizamos un array podremos desapilar los datos del mismo ya que el mismo puede tener una dimension diferente

MUCHAS GRACIAS

Para esta exposición se necesitó representar los tipos de memoria con materiales reciclados. Nuestro equipo usó bloques y vasos para representar cada tipo de memoria y como es que actuaba.

6. Resultados

- Ejercicios Iterativos con los 3 bucles (for, while, do while).

Clase	Ejecución
While: 	   
Do while: 	

```
Ejercicios.java | EjerciciosIterativos.java | EjerciciosIterativos3.java
Source | History
```

```java
public class Ejercicios {

    //// METODOS CON WHILE
    public static String tablaMultiplicar(int n){
        String cad = "";
        byte J = 1; // siempre debe de haber un valor inicial
        while(J<=10){ // siempre debe de haber una condicion verdadera-true
            cad += n + "*" + J + " = " + (n*J) + "\n";
            J++; // incremento
        }
        return cad;
    }

    //diseñar un metodo de clase que lea 15 valores enteros e imprima el mayor
    public static byte numeroMayor(){
        byte J = 1;
        byte mayor = 0;
        byte dato;
        while(J <= 15){
            dato = Tools.leeByte(msg: "Ingrese Dato: ");
            if(dato > mayor)
                mayor = dato;
            J++;
        }
        System.out.println("El mayor es: " + mayor);
        return mayor;
    }
    //diseñar un metodo de clase que reciba como parametro un valor entero
    //y retorne su correspondiente valor factorial
    public static double factorial(int dato){
        double Fac;
        byte J=1;
        if (dato == 0 || dato == 1) {
            return 1;
        }
        else{
            Fac= 1;
        }
        while(J <= dato){
            Fac*=J;
                J++;
        }
        return Fac;
    }
```

```java
    //////////METODOS CON DO WHILE

    public static String tablaMul(int n){
        String cad = "";
        byte J = 1; //siempre debe de haber un valor inicial
        do{
            cad += n + "*" + J + " = " + (n*J) + "\n";
            J++; // incremento
        } while(J<=10);
          // siempre debe de haber una condicion verdadera true
        return cad;
    }

    public static byte numeroMa(){
        byte J = 1;
        byte mayor = 0;
        byte dato;
        do{
            dato = Tools.leeByte(msg: "Ingrese Dato: ");
            if(dato > mayor)
                mayor = dato;
            J++;
        } while(J <= 15);
        System.out.println("El mayor es: " + mayor);
        return mayor;
    }
```

```java
public static double facto(int dato){
    double Fac;
    byte J=1;
    if (dato == 0 || dato == 1) {
        return 1;
    }
    else{
        Fac= 1;
    }
    do{
        Fac*=J;
            J++;
    }
    while(J <= dato);

    return Fac;
}
```

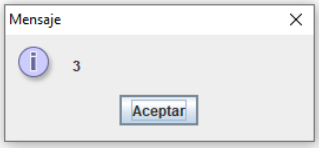
For:

```java
///// METODOS CON FOR

public static String tablaM(int n){
    String cad = "";
    for (int J = 0; J <= n; J++) {
        cad += n + "*" + J + " = " + (n*J) + "\n";
    }
    return cad;
}

public static byte numeroM(){
    byte mayor = 0;
    byte dato;
    dato = Tools.leeByte(msg: "Ingrese Dato: ");
    for (int J = 0; J < dato; J++) {
        if(dato > mayor)
            mayor = dato;
    }
    System.out.println("El mayor es: " + mayor);
    return mayor;
    }
public static double fac(int dato){
    double Fac;
    if (dato == 0 || dato == 1) {
        return 1;
    }
    else{
        Fac= 1;
    }

    for (int J = 0; J <= dato; J++) {
        Fac*=J;
            J++;
    }
    return Fac;
}
```

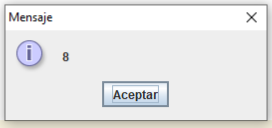
Otros métodos con bucle do while:

```java
/// METODOS CON DO WHILE
public static int sumaDivisores(int dato){
    int k=1, suma=0;
    do{
        if(dato% k==0)
            suma+=k;
        k++;
    } while(k<dato);
    return suma;
}
public static String numPares(){
    String cad="";
    byte k=2;
    do{
        cad+=k + "\n";
        k+=2;
    } while(k<=20);
    return cad;
}
public static byte ctaDigitos(int dato){
    byte c=0;
    do{
        dato/=10;
        c++;
    } while(dato!=0);
    return c;
}
```

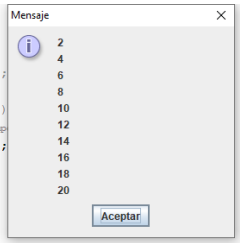
```java
public static void main(String[] args) {

    JOptionPane.showMessageDialog(parentComponent: null,
            message: Ejercicios.ctaDigitos(dato:123));
    //JOptionPane.showMessageDialog(null,
            //Ejercicios.sumaDivisores(10));
    //JOptionPane.showMessageDialog(null,
            //Ejercicios.numPares());
}
```



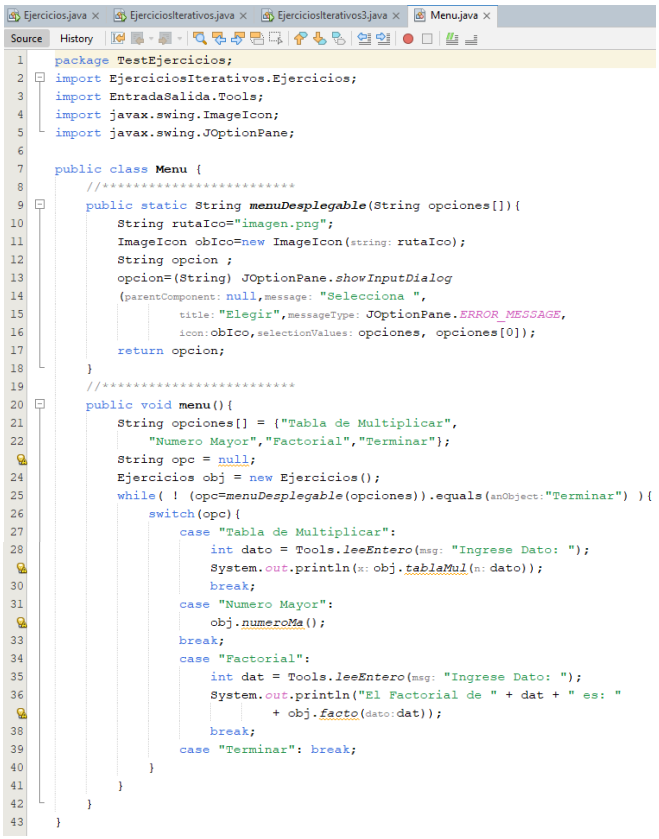

```java
public static void main(String[] args) {

    //JOptionPane.showMessageDialog(null,
            //Ejercicios.ctaDigitos(123));
    JOptionPane.showMessageDialog(parentComponent: null,
            message: Ejercicios.sumaDivisores(dato:10));
    //JOptionPane.showMessageDialog(null,
            //Ejercicios.numPares());
}
```




```java
public static void main(String[] args) {

    //JOptionPane.showMessageDialog(null,
            //Ejercicios.ctaDigitos(123));
    //JOptionPane.showMessageDialog(null,
            //Ejercicios.sumaDivisores(10)
    JOptionPane.showMessageDialog(parentComp
            message: Ejercicios.numPares());

}
```



Menú:



```
Ejercicios.java × EjerciciosIterativos.java × EjerciciosIterativos3.java × Menu.java ×
Source  History
1  package TestEjercicios;
2  import EjerciciosIterativos.Ejercicios;
3  import EntradaSalida.Tools;
4  import javax.swing.ImageIcon;
5  import javax.swing.JOptionPane;
6
7  public class Menu {
8      //************************
9      public static String menuDesplegable(String opciones[]){
10         String rutaIco="imagen.png";
11         ImageIcon obIco=new ImageIcon(string: rutaIco);
12         String opcion ;
13         opcion=(String) JOptionPane.showInputDialog
14         (parentComponent: null,message: "Selecciona ",
15                 title:"Elegir",messageType: JOptionPane.ERROR_MESSAGE,
16                 icon:obIco,selectionValues: opciones, opciones[0]);
17         return opcion;
18     }
19     //************************
20     public void menu(){
21         String opciones[] = {"Tabla de Multiplicar",
22             "Numero Mayor","Factorial","Terminar"};
23         String opc = null;
24         Ejercicios obj = new Ejercicios();
25         while( ! (opc=menuDesplegable(opciones)).equals(anObject:"Terminar") ){
26             switch(opc){
27                 case "Tabla de Multiplicar":
28                     int dato = Tools.leeEntero(msg: "Ingrese Dato: ");
29                     System.out.println(x: obj.tablaMul(n: dato));
30                     break;
31                 case "Numero Mayor":
32                     obj.numeroMa();
33                 break;
34                 case "Factorial":
35                     int dat = Tools.leeEntero(msg: "Ingrese Dato: ");
36                     System.out.println("El Factorial de " + dat + " es: "
37                             + obj.facto(dato:dat));
38                     break;
39                 case "Terminar": break;
40             }
41         }
42     }
43 }
```

Funcionamiento

Para comenzar el tema de recursividad primero se realizaron varios programas iterativos con los 3 tipos de bucles, para tener en cuenta que cada bucle cuenta con 3 puntos esenciales para poder realizarse, los cuales son: un valor inicial, una condición verdadera, y un incremento.

Una vez teniendo en cuenta la estructura de los bucles, es más fácil pasar a los métodos recursivos.

Se hizo un menú, para que hubiera un orden a la hora de probar los programas.

Ejercicios Recursivos:

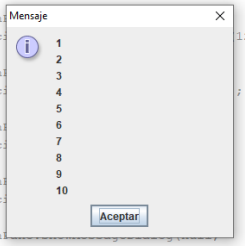
Clase

EjerciciosRe.java

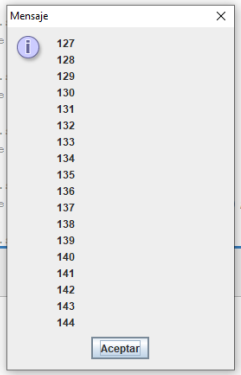
```java
public class EjerciciosRe {

    public static String funcionRecursiva(int J, int n){
        if(J>=n){
            return J  + " " +  funcionRecursiva(J-1, n);
            //return J + " = " + ctaDigitos((byte) 0, n) +
            //"\n" +funcionRecursiva(J+1, n);
            //return J + "\n" + funcionRecursiva(J+1, n);
            // System.out.println(J);
        } else
            return "";
    }

    //// METODO LEER MAYOR
    public static int leerMayor(byte J, int mayor){
        if(J<=5){
            int dato = Tools.leeEntero(msg: "Ingrese Dato: ");
            if(dato > mayor){
                mayor = dato;
                return leerMayor((byte) (J+1), mayor);
            }
        }
        return mayor;
    }

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

    public static int fac(byte J, int dato){
        double Fac;
        if (dato == 0 || dato == 1) {
            return 1;
        }
        else
            Fac = 1;
        if(J <= dato){
            return J*fac((byte)(J+1), dato);
        }
        return (int) Fac;
    }

    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;
    }

    public static String numPares1(byte J){
        String cad = " ";
        if(J<=20){
            //J+=2;
            return cad+=J + "\n" + numPares1((byte) (J+2));
        }
        return " ";
    }

    public static byte ctaDigitos(byte J, int dato){
        if(dato!=0){
            dato/=10;
            return ctaDigitos((byte) (J+1), dato);
        }
        return J;
    }

    public static String triangulo(byte J){
        if(J>=1){
            return funcionRecursiva(J, n: 1) + "\n" + triangulo((byte) (J-1));
        }
        return " ";
```

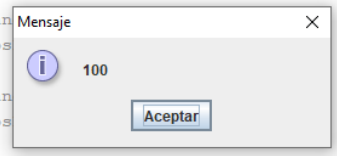
Ejecución

```
JOptionPane.showMessageDialog(parentComponent: null,
message: EjerciciosRe.funcionRecursiva(J: 1, n: 10));
```



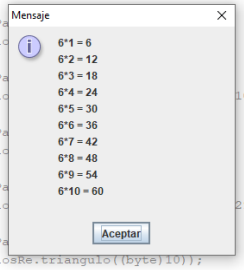

```
JOptionPane.showMessageDialog(parentComponent: null,
message: EjerciciosRe.funcionRecursiva((byte)(127), n: 144));
```

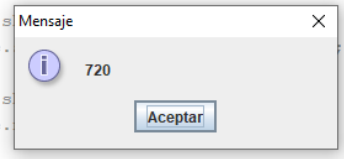





```
ane.showMessageDialog(null,
```

```
JOptionPane.showMessageDialog(parentComponent: null,
message: EjerciciosRe.tablaMul((byte)(1), n: 6));
```

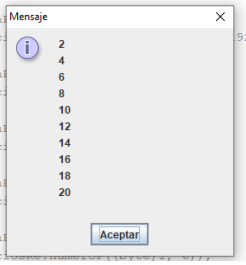




```java
public static String funcionRecur2(int J, int n){
    if(J >= n){
        return " * " +  funcionRecur2(J-1, n);
    } else
        return " ";
}

public static String piramide(byte J, int dato){
    if(J>=dato){
        return funcionRecur2(J,n: dato) + "\n" +
                piramide((byte)(J+1), dato);
    }
    return " ";
}

public static String numPares(byte J, int Num){
    String cad = " ";
    if(J <= 20){
        return cad += "El cubo de: " + J + " = " + (int)
            Math.pow(a: J, b: 3) + "\n" + numPares((byte) (J+2), Num+1);
    }
    return " ";
}

public static String numeroP(byte J, int Num){
    if(J <= 10){
        System.out.println(Num + "^3 = " + (Math.pow(a: Num, b: 3)) );
        numeroP((byte) (Num+2), J+1);
    }
    return "";
}

public static String imprimirNumerosPrimosCubo(int num, int count) {
    if (count <= 10) {
        System.out.println(num + "^3 = " + (Math.pow(a: num, b: 3)));
        return imprimirNumerosPrimosCubo(num+1, count+1);
    }
    return "";

}

public static String numPrimo(int valor){
    int num, aux = 0;
    num = valor;
    for (int i = 1; i <= num; i++) {
        if(num%i == 0){
            aux++;
        }
    }
    if(aux == 2){
        return ("Es Primo");
    } else{
        return ("No es Primo");
    }
}

public static void vocalesRe(String tex, int a, int e, int i, int o, int u){
    if(tex.length()==0){
        JOptionPane.showMessageDialog(parentComponent: null,
                "Numero de Vocales: " + (a + e + i + o + u) + "\n" +
                                        ("a-" + a) + "\n" +
                                        ("e-" + e) + "\n" +
                                        ("i-" + i) + "\n" +
                                        ("o-" + o) + "\n" +
                                        ("u-" + u) + "\n" ); return;
    }
    char primerCaracter = tex.charAt(index: 0);
    if(primerCaracter == 'a' || primerCaracter == 'A'){
        a++;
    } else if(primerCaracter == 'e' || primerCaracter == 'E'){
        e++;
    } else if(primerCaracter == 'i' || primerCaracter == 'I'){
        i++;
    } else if (primerCaracter == 'o' || primerCaracter == 'O'){
        o++;
    } else if (primerCaracter == 'u' || primerCaracter == 'U'){
        u++;
    }
    vocalesRe(tex: tex.substring(beginIndex: 1), a, e, i, o, u);
}
```

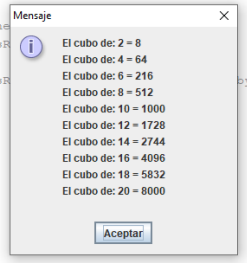
```java
JOptionPane.showMessageDialog(parentComponent: null,
message: EjerciciosRe.sumaDivisores((byte)(1), dato:10));
```


```java
JOptionPane.showMessageDialog(parentComponent: null,
message: EjerciciosRe.numPares1((byte)2));
```




```java
JOptionPane.showMessageDialog(parentComponent: null,
message: EjerciciosRe.otaDigitos((byte)(0), dato:919293));
```


```java
JOptionPane.showMessageDialog(parentComponent: null,
message: EjerciciosRe.numPares((byte)2, Num: 20));
```




```java
EjerciciosRe.imprimirNumerosPrimosCubo((byte)1, count: 3);
```

```
Debugger Console ×  RECURSIVIDAD (run) ×
run:
1^3 = 1.0
2^3 = 8.0
3^3 = 27.0
4^3 = 64.0
5^3 = 125.0
6^3 = 216.0
7^3 = 343.0
8^3 = 512.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

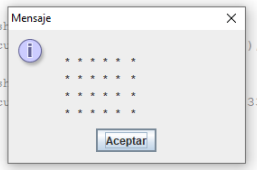
Otros ejercicios Recursivos:

EjerciciosRe.java | EjerciciosRe3.java | EjerciciosRecursivos2.java | Principal3.java

Source | History

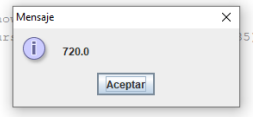
```java
package EjerciciosRecursividad;
import javax.swing.JOptionPane;

public class EjerciciosRecursivos2 {

    /// EJERCICIO 1
    public static String rectangulo(int altura, int base){

        String cad = " ";
        if(base>0){
            cad += " * ";
            return  "\n" + rectangulo2(altura-1, base )+
                    cad +  rectangulo(altura, base-1);
        }
        return cad;
    }

    public static String rectangulo2(int altura, int base){
        String cad = " ";
        if(altura>0){
            cad += " * ";
            return cad + rectangulo2(altura-1, base );
        }
        return cad;
    }
    /// EJERCICIO 2
    public static byte ctaDigitos(byte J, int dato){
        if(dato!=0){
            dato /= 10;
            return ctaDigitos((byte) (J+1), dato);
        }
        return J;
    }

    /// EJERCICIO 3
    public static double Factorial(byte J, int Dato){
        double Fac;
        if (Dato == 0 || Dato == 1) {
            return 1;
        }
        else
            Fac= 1;

        if(J <= Dato){
            return J*Factorial((byte)(J+1), Dato);
        }
        return (int)Fac;
    }
```

```java
public static void main(String[] args) {
    JOptionPane.showMessageDialog(parentComponent: null,
    message: EjerciciosRecursivos2.rectangulo(altura: 6, base:4));

    //JOptionPane.sh
    //EjerciciosRecu
```

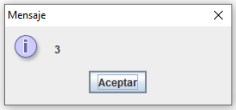



```java
    //JOptionPane.sh
    //EjerciciosRecu
```

```java
JOptionPane.showMessageDialog(parentComponent: null,
message: EjerciciosRecursivos2.Factorial((byte)1, Dato:6));

//JOptionPane.sho
//EjerciciosRecur
```




```java
JOptionPane.showMessageDialog(parentComponent: null,
message: EjerciciosRecursivos2.CtaDigitosR((byte)0, Dato:335) );
```



Funcionamiento

Una vez comprendiendo los programas iterativos es más fácil pasarlos a recursivos, pues hay que tener en cuenta 3 puntos esenciales de los bucles, porque también se deben estructurar en los métodos recursivos.

7. Conclusiones

En conclusión, tanto los procedimientos iterativos como los procedimientos recursivos son herramientas fundamentales en la programación y cada uno tiene su propio conjunto de ventajas y desventajas. Los procedimientos iterativos son útiles para problemas que requieren operaciones repetitivas en secuencia, mientras que la recursividad es una técnica para resolver problemas que se pueden descomponer en subproblemas más pequeños.

La recursividad es un tema fundamental en la programación, es una alternativa para ejecutar las estructuras de repetición (bucles). Se debe usar cuando sea realmente necesaria, es decir, cuando no exista una solución iterativa simple.

La elección del método adecuado depende del problema específico que se esté resolviendo. La recursividad es una técnica que puede ser muy útil en ciertos casos, pero también es importante tener en cuenta las posibles limitaciones y riesgos asociados con su uso.

La recursividad como la iteración se basan en la repetición, la primera logra la repetición a través de llamadas repetidas a una función y la segunda utiliza explícitamente una estructura de repetición (ciclos). Saber y utilizar la recursividad es muy útil para solucionar problemas por medio de algoritmos, eficientes, eficaces, y de fácil entendimiento.

8. Bibliografía

Bibliografía

Cormen, T. H. (2009). *Introducción a los algoritmos.* MIT press.

Universidad Militar de Nueva Granada. (s.f.). *Introducción*. Obtenido de http://virtual.umng.edu.co/distancia/ecosistema/odin/odin_desktop.php?path=Li4vb3Zhcy9pbmdlbmllcmlhX2luZm9ybWF0aWNhL2VzdHJ1Y3R1cmFfZGVfZGF0b3MvdW5pZGFkXzIv#slide_1