

INSTITUTO TECNOLÓGICO DE MEXICO

CAMPUS ORIZABA

NOMBRE DE LA ASIGNATURA:

ESTRUCTURA DE DATOS

TEMA:

REPORTE DE PRACTICA TEMA 2 RECURSIVIDAD

NOMBRE DE LOS INTEGRANTES:

HERNÁNDEZ DÍAZ ARIEL - 21010195

PALACIOS MENDEZ XIMENA MONSERRAT - 21010209

CARRERA:

INGENIERIA INFORMATICA

GRUPO:

3a3A

FECHA DE ENTREGA:

27 DE MARZO DE 2023

INTRODUCCIÓN

Una función recursiva es una función que se llama a sí misma. Esto es, dentro del cuerpo de la función se incluyen llamadas a la propia función. Esta estrategia es una alternativa al uso de bucles. Una solución recursiva es, normalmente, menos eficiente que una solución basada en bucles. Esto se debe a las operaciones auxiliares que llevan consigo las llamadas a las funciones.

Cuando un programa llama a una función que llama a otra, la cual llama a otra y así sucesivamente, las variables y valores de los parámetros de cada llamada a cada función se guardan en la pila o stack, junto con la dirección de la siguiente línea de código a ejecutar una vez finalizada la ejecución de la función invocada.

Esta pila va creciendo a medida que se llama a más funciones y decrece cuando cada función termina. Si una función se llama a sí misma recursivamente un número muy grande de veces existe el riesgo de que se agote la memoria de la pila, causando la terminación brusca del programa.

A pesar de todos estos inconvenientes, en muchas circunstancias el uso de la recursividad permite a los programadores especificar soluciones naturales y sencillas que sin emplear esta técnica serían mucho más complejas de resolver. Esto convierte a la recursión en una potente herramienta de programación. Sin embargo, por sus inconvenientes, debe emplearse con cautela.

Las asignaciones de memoria pueden ser dinámicas o estáticas y hay diferencias entre estas dos y se pueden aplicar las dos en un programa cualquiera.

COMPETENCIA(S) ESPECÍFICA (S):

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

MARCO TEÓRICO

1.-Recursividad:

La recursividad es una técnica de programación importante. Se utiliza para realizar una llamada a una función desde la misma función.

Como ejemplo útil se puede presentar el cálculo de números factoriales. El factorial de 0 es, por definición, 1. Las factoriales de números mayores se calculan mediante la multiplicación de $1 * 2 * \dots$, incrementando el número de 1 en 1 hasta llegar al número para el que se está calculando la factorial.

El siguiente párrafo muestra una función, expresada con palabras, que calcula una factorial.

"Si el número es menor que cero, se rechaza. Si no es un entero, se redondea al siguiente entero. Si el número es cero, su factorial es uno. Si el número es mayor que cero, se multiplica por él factorial del número menor inmediato."

Para calcular la factorial de cualquier número mayor que cero hay que calcular como mínimo la factorial de otro número. La función que se utiliza es la función en la que se encuentra en estos momentos, esta función debe llamarse a sí misma para el número menor inmediato, para poder ejecutarse en el número actual. Esto es un ejemplo de recursividad.

Las funciones recursivas se clasifican en:

- Recursividad directa: se llama a sí mismo (método de recursividad simple) o en varias partes del programa (recursividad múltiple).
- Recursividad indirecta o mutua: dos métodos se llaman entre sí.

La recursividad y la iteración (ejecución en bucle) están muy relacionadas, cualquier acción que pueda realizarse con la recursividad puede realizarse con

iteración y viceversa. Normalmente, un cálculo determinado se prestará a una técnica u otra, sólo necesita elegir el enfoque más natural o con el que se sienta más cómodo.

Claramente, esta técnica puede constituir un modo de meterse en problemas. Es fácil crear una función recursiva que no llegue a devolver nunca un resultado definitivo y no pueda llegar a un punto de finalización. Este tipo de recursividad hace que el sistema ejecute lo que se conoce como bucle "infinito".

Para entender mejor lo que en realidad es el concepto de recursión veamos un poco lo referente a la secuencia de Fibonacci.

Principalmente habría que aclarar que es un ejemplo menos familiar que el de la factorial, que consiste en la secuencia de enteros.

0,1,1,2,3,5,8,13,21,34,

Cada elemento en esta secuencia es la suma de los precedentes (por ejemplo $0 + 1 = 0$, $1 + 1 = 2$, $1 + 2 = 3$, $2 + 3 = 5$, ...) sean $\text{fib}(0) = 0$, $\text{fib}(1) = 1$ y así sucesivamente, entonces puede definirse la secuencia de Fibonacci mediante la definición recursiva (define un objeto en términos de un caso más simple de sí mismo):

$\text{fib}(n) = n$ if $n == 0$ or $n == 1$

$\text{fib}(n) = \text{fib}(n - 2) + \text{fib}(n - 1)$

if $n >= 2$

Por ejemplo, para calcular $\text{fib}(6)$, puede aplicarse la definición de manera recursiva para obtener:

$\text{Fib}(6) = \text{fib}(4) + \text{fib}(5) = \text{fib}(2) + \text{fib}(3) + \text{fib}(5) = \text{fib}(0) + \text{fib}(1) + \text{fib}(3) + \text{fib}(5) = 0 + 1 + \text{fib}(3) + \text{fib}(5) + \text{fib}(1) + \text{fib}(2) + \text{fib}(5) = 0 + 1 + \text{fib}(0) + \text{fib}(1) + \text{fib}(5) = 0 + 1 + \text{fib}(5) = 0 + 1 + 3 + \text{fib}(3) + \text{fib}(4) = 0 + 1 + 3 + 1 + \text{fib}(0) + \text{fib}(1) + \text{fib}(4) = 0 + 1 + \text{fib}(1) + \text{fib}(2) + \text{fib}(4) = 0 + 1 + 1 + \text{fib}(2) + \text{fib}(3) = 0 + 1 + 5 + \text{fib}(0) + \text{fib}(1) + \text{fib}(3) = 0 + 1 + 5 + 1 + \text{fib}(1) + \text{fib}(2) = 0 + 1 + 6 + \text{fib}(0) + \text{fib}(1) = 0 + 1 + 6 = 8$

Obsérvese que la definición recursiva de los números de Fibonacci difiere de las definiciones recursivas de la función factorial y de la multiplicación . La definición recursiva de fib se refiere dos veces a sí misma . Por ejemplo, fib (6) = fib (4) + fib (5), de tal manera que al calcular fib (6), fib tiene que aplicarse de manera recursiva dos veces. Sin embargo, calcular fib (5) también implica calcular fib (4), así que al aplicar la definición hay mucha redundancia de cálculo. En ejemplo anterior, fib(3) se calcula tres veces por separado. Sería mucho más eficiente "recordar" el valor de fib(3) la primera vez que se calcula y volver a usarlo cada vez que se necesite. Es mucho más eficiente un método iterativo como el que sigue para calcular fib (n).

```
If (n < = 1)
    return (n);
lolib = 0 ;
hifib = 1 ;
for (i = 2; i < = n; i++)
{
    x = lolib ;
    lolib = hifib ;
    hifib = x + lolib ;
} /* fin del for*/
return (hifib) ;
```

Compárese el número de adiciones (sin incluir los incrementos de la variable índice, i) que se ejecutan para calcular fib (6) mediante este algoritmo al usar la definición recursiva. ¡En el caso de la función factorial, tienen que ejecutarse el mismo número de multiplicaciones para calcular $n!$ Mediante ambos métodos: recursivo e iterativo. Lo mismo ocurre con el número de sumas en los dos métodos al calcular la multiplicación. Sin embargo, en el caso de los números de Fibonacci, el método recursivo es mucho más costoso que el iterativo.

2.- Propiedades de las definiciones o algoritmos recursivos:

Un requisito importante para que sea correcto un algoritmo recursivo es que no genere una secuencia infinita de llamadas así mismo. Claro que cualquier algoritmo que genere tal secuencia no termina nunca. Una función recursiva f debe definirse en términos que no impliquen a f al menos en un argumento o grupo de argumentos. Debe existir una "salida" de la secuencia de llamadas recursivas.

Si en esta salida no puede calcularse ninguna función recursiva. Cualquier caso de definición recursiva o invocación de un algoritmo recursivo tiene que reducirse a la larga a alguna manipulación de uno o casos más simples no recursivos.

3.- Cadenas recursivas:

Una función recursiva no necesita llamarse a sí misma de manera directa. En su lugar, puede hacerlo de manera indirecta como en el siguiente ejemplo:

a (formal parameters) b (formal parameters)

```
{ {
```

```
..
```

b (arguments); a (arguments);

```
..
```

```
}
```

/*fin de a*/

```
 } /*fin de b*/
```

En este ejemplo la función a llama a b, la cual puede a su vez llamar a, que puede llamar de nuevo a b. Así, ambas funciones a y b, son recursivas, dado que se llaman así mismo de manera indirecta. Sin embargo, el que lo sean no es obvio a partir del examen del cuerpo de una de las rutinas en forma individual. La rutina a, parece llamar a otra rutina b y es imposible determinar que se puede llamar así misma de manera indirecta al examinar sólo a.

Pueden incluirse más de dos rutinas en una cadena recursiva. Así, una rutina a puede llamar a b, que llama a c, ..., que llama a z, que llama a a. Cada rutina de la cadena puede potencialmente llamarse a sí misma y, por lo tanto, es recursiva. Por supuesto, el programador debe asegurarse de que un programa de este tipo no genere una secuencia infinita de llamadas recursivas.

4.- Definición recursiva de expresiones algebraicas:

Como ejemplo de cadena recursiva consideremos el siguiente grupo de definiciones:

una expresión es un término seguido por un signo más seguido por un término, o un término solo un término es un factor seguido por un asterisco seguido por un factor, o un factor solo.

Un factor es una letra o una expresión encerrada entre paréntesis. Antes de ver algunos ejemplos, obsérvese que ninguno de los tres elementos anteriores está definido en forma directa en sus propios términos. Sin embargo, cada uno de ellos se define de manera indirecta. Una expresión se define por medio de un término, un término por medio de un factor y un factor por medio de una expresión.

De manera similar, se define un factor por medio de una expresión, que se define por medio de un término que a su vez se define por medio de un factor. Así, el conjunto completo de definiciones forma una cadena recursiva.

La forma más simple de un factor es una letra. Así A, B, C, Q, Z y M son factores. También son términos, dado que un término puede ser un factor solo. También son expresiones dado que una expresión puede ser un término solo. Como A es una expresión, (A) es un factor y, por lo tanto, un término y una expresión. A + B es un ejemplo de una expresión que no es ni un término ni un factor, sin embargo (A + B) es las tres cosas. A * B es un término y, en consecuencia, una expresión, pero no es un factor. A * B + C es una expresión, pero no es un factor.

Cada uno de los ejemplos anteriores es una expresión valida. Esto puede mostrarse al aplicar la definición de una expresión de cada uno. Considérese, sin embargo, la cadena A + * B. No es ni una expresión, ni un término, ni un factor. Sería instructivo para el lector intentar aplicar la definición de expresión, término y factor para ver que ninguna de ellas describe a la cadena A + * B. De manera similar, (A + B*) C y A + B + C son expresiones nulas de acuerdo con las definiciones precedentes.

A continuación, se codificará un programa que lea e imprima una cadena de caracteres y luego imprima "valida" si la expresión lo es y "no valida" de no serlo. Se usan tres funciones para

reconocer expresiones, términos y factores, respectivamente. Primero, sin embargo se presenta una función auxiliar getsymb que opera con tres parámetros: str, length y p pos. Str contiene la entrada de la cadena de cadena de caracteres; length representa el número de caracteres en str. Ppos apunta a un puntero pos cuyo valor es la posición str de la que obtuvimos un carácter la última vez. Si pos < length, getsymb regresa el carácter cadena str [pos] e incrementa pos en 1. Si pos > = length, getsymb regresa un espacio en blanco. getsymb (str, length, ppos)

```
char str[];  
int length, *ppos;  
{  
char C;
```

```
if (*ppos < length)
c = str [*ppos];
else
c = ' ';
(*ppos)++;
return ( c );
} /* fin de getsymb*/
```

La función que reconoce una expresión se llama expr. Regresa TRUE (o1) (VERDADERO) si una expresión valida comienza en la posición pos de str y FALSE (o0) FALSO en caso contrario. También vuelve a colocar pos en la posición que sigue en la expresión de mayor longitud que puede encontrar. Suponemos también una función readstr que lee una cadena de caracteres, poniendo la cadena en str y su largo en length.

5.- Programación Recursiva:

Es mucho más difícil desarrollar una solución recursiva en C para resolver un problema específico cuando no se tiene un algoritmo. No es solo el programa sino las definiciones originales y los algoritmos los que deben desarrollarse. En general, cuando encaramos la tarea de escribir un programa para resolver un problema no hay razón para buscar una solución recursiva. La mayoría de los problemas pueden resolverse de una manera directa usando métodos no recursivos. Sin embargo, otros pueden resolverse de una manera más lógica y elegante mediante la recursión.

Volviendo a examinar la función factorial. El factor es, probablemente, un ejemplo fundamental de un problema que no debe resolverse de manera recursiva, dado que su solución iterativa es directa y simple. Sin embargo, examinaremos los elementos que permiten dar una solución recursiva. Antes que nada, puede reconocerse un gran número de casos distintos que se deben resolver. Es decir,

¡quiere escribirse un programa para calcular $0!$, $1!$, $2!$ Y así sucesivamente. Puede identificarse un caso "trivial" para el cual la solución no recursiva pueda obtenerse en forma directa. ¡Es el caso de $0!$, que se define como 1. El siguiente paso es encontrar un método para resolver un caso "complejo" en términos de uno más "simple", lo cual permite la reducción de un problema complejo a uno más simple. La transformación del caso complejo al simple resultaría al final en el caso trivial. Esto significaría que el caso complejo se define, en lo fundamental, en términos del más simple.

Examinaremos que significa lo anterior cuando se aplica la función factorial. $4!$ Es un caso más complejo que $3!$ La transformación que se aplica al número a para obtener 3 es sencillamente restar 1. Si restamos 1 de 4 de manera sucesiva llegamos a 0, que es el caso trivial. Así, si se puede definir $4!$ en términos de $3!$ y, en general, $n!$ en términos de $(n - 1)!$, se podrá calcular $4!$ mediante la definición de $n!$ en términos de $(n - 1)!$ al trabajar, primero hasta llegar a $0!$ y luego al regresar a $4!$. En el caso de la función factorial se tiene una definición de ese tipo, dado que:

$$n! = n * (n - 1)!$$

$$\text{Así, } 4! = 4 * 3! = 4 * 3 * 2! = 4 * 3 * 2 * 1! = 4 * 3 * 2 * 1 * 0! = 4 * 3 * 2 * [*] = 24$$

Estos son los ingredientes esenciales de una rutina recursiva: poder definir un caso "complejo" en términos de uno más "simple" y tener un caso "trivial" (no recursivo) que pueda resolverse de manera directa. Al hacerlo, puede desarrollarse una solución si se supone que se ha resuelto el caso más simple. La versión C de la función factorial supone que está definido $(n - 1)!$ y usa esa cantidad al calcular $n!$

Otra forma de aplicar estas ideas a otros ejemplos antes explicados. En la definición de $a * b$, es trivial el caso de $b = 1$, pues $a * b$ es igual a a . En general, $a + b$ puede definirse en términos de $a + (b - 1)$ mediante la definición $a + b = a + (b - 1) + 1$.

1) + a. De nuevo, el caso complejo se transforma en un caso más simple al restar 1, lo que lleva, al final, al caso trivial de $b = 1$. Aquí la recursión se basa únicamente en el segundo parámetro, b .

Con respecto al ejemplo de la función de Fibonacci, se definieron dos casos triviales: $\text{fib}(0) = 0$ y $\text{fib}(1) = 1$. Un caso complejo $\text{fib}(n)$ se reduce entonces a dos más simples: $\text{fib}(n - 1)$ y $\text{fib}(n - 2)$. Esto se debe a la definición de $\text{fib}(n)$ como $\text{fib}(n - 1) + \text{fib}(n - 2)$, donde se requiere de dos casos triviales definidos de manera directa. $\text{fib}(1)$ no puede definirse como $\text{fib}(0) + \text{fib}(-1)$ porque la función de Fibonacci no está definida para números negativos.

6.- Asignación estática y dinámica de memoria:

Hasta este momento solamente hemos realizado asignaciones estáticas del programa, y más concretamente estas asignaciones estáticas no eran otras que las declaraciones de variables en nuestro programa. Cuando declaramos una variable se reserva la memoria suficiente para contener la información que debe almacenar. Esta memoria permanece asignada a la variable hasta que termine la ejecución del programa (función `main`). Realmente las variables locales de las funciones se crean cuando éstas son llamadas, pero nosotros no tenemos control sobre esa memoria, el compilador genera el código para esta operación automáticamente.

En este sentido las variables locales están asociadas a asignaciones de memoria dinámicas, puesto que se crean y destruyen durante la ejecución del programa.

Así entendemos por asignaciones de memoria dinámica, aquellas que son creadas por nuestro programa mientras se están ejecutando y que, por tanto, cuya gestión debe ser realizada por el programador.

7.- Divide y vencerás

Se emplea en el diseño de algoritmos para la resolución de problemas a partir de la división de subproblemas del mismo tipo, pero con menor tamaño. Si los

subproblemas se mantienen grandes, se aplicarán de nuevo para obtener subproblemas pequeños para ser solucionados con la implementación de recursión.

Básicamente la técnica divide y vencerás (DV) consiste en:

- Descomponer el caso a resolver en un cierto número de sub-casos más pequeños del mismo problema.
- Resolver sucesiva e independientemente todos estos sub-casos.
- Combinar las soluciones obtenidas para obtener la solución del caso original.

Características

- Subproblemas del mismo tipo que el original.
- Los subproblemas se resuelven independientemente.
- No existe solapamiento entre subproblemas.

Condiciones para que DV sea óptimo

- Selección de cuando utilizar el algoritmo ad hoc, calcular el umbral de recursividad.
- Poder descomponer problema en subproblemas y recombinar de forma eficiente a partir de las soluciones parciales.
- Los sub-casos deben tener aproximadamente el mismo tamaño.

8.- Recursividad simple

En la recursividad simple, dentro del módulo recursivo, existe una única invocación a sí mismo, como sucede en el caso de la función factorial. En general, los módulos recursivos simples son fácilmente transformados a su versión iterativa.

Llamada a una función recursiva

En cada llamada se genera un nuevo ejemplar de la función con sus objetos locales:

- La función se ejecuta normalmente hasta la llamada a sí misma. Se crean en la pila de recursión nuevos parámetros y variables locales con los mismos nombres.
- El nuevo proceso de función comienza a ejecutarse.
- Se crean varias copias hasta llegar a los casos bases, donde se resuelve directamente el valor, y se va saliendo, liberando memoria hasta llegar a la primera llamada (última en cerrarse).

9.- Recursividad múltiple

En la recursividad múltiple, el módulo se invoca a sí mismo más de una vez dentro de una misma activación. Este tipo de módulos son más difíciles de llevar a su forma iterativa. Algunos ejemplos de funciones recursivas múltiples son la función que implementa la sucesión de Fibonacci y la de las Torres de Hanoi.

Como una variante de la recursión múltiple se puede tener recursión anidada, que ocurre cuando dentro de una invocación recursiva ocurre como parámetro otra invocación recursiva. Un ejemplo de esto puede verse en la de Ackerman.

10.- Recursividad anidada

Esta se da cuando alguna llamada recursiva recibe como parámetro a otra llamada recursiva:

1. función R(m){
2. si B(m) devolver H(m)
3. sino R(P(R(K(m))))

Donde:

- R: Función recursividad
- m: Parámetro
- B: Condición booleana del parámetro

- H: Transformación del parámetro
- P: Transformación de la función recursiva
- K: Transformación del parámetro

11.- Arreglos

Un arreglo es una estructura de datos utilizada para almacenar datos del mismo tipo. Los arreglos almacenan sus elementos en ubicaciones de memoria contiguas. En Java, los arreglos son objetos. Todos los métodos se pueden ser invocados en un arreglo.

12.- Matrices

Un array en Java puede tener más de una dimensión. El caso más general son los arrays bidimensionales también llamados matrices o tablas. La dimensión de un array la determina el número de índices necesarios para acceder a sus elementos.

Los vectores que hemos visto en otra entrada anterior son arrays unidimensionales porque solo utilizan un índice para acceder a cada elemento. Una matriz necesita dos índices para acceder a sus elementos. Gráficamente podemos representar una matriz como una tabla de n filas y m columnas cuyos elementos son todos del mismo tipo.

13.- ¿Qué es iteración?

En el mundo de la programación, una iteración es cuando un conjunto de instrucciones, características o actividades se repiten una o varias veces hasta que una condición se cumple o deja de cumplirse. Así pues, cuando el conjunto de instrucciones se repite, se entiende el proceso de código como una iteración. Aun así, cuando se sigue repitiendo de forma indefinida, se registra un loop o un bucle.

En cada iteración del conjunto de instrucciones se deben repetir los pasos de una forma continua, sin tener ningún punto de quiebre en el proceso. A menos que se cumpla que la condición sea verdadera o falsa, según sea la manera en la que está definida la iteración.

Implementando la iteración

Pero ¿cómo funciona la iteración o cómo se implementa? La iteración se puede implementar a partir de los loops o bucles de instrucciones en forma de código. Así pues, hay tres tipos de loops que se utilizan en la construcción de los programas de código. Estos son:

El bucle for: este loop se utiliza para ejecutar un grupo específico de instrucciones hasta que se cumpla una condición en específica. La sintaxis general de cómo se ve un bucle for está definida de la siguiente manera: `for ([expresion-inicial]; [condicion]; [expresion-final])` y, por último, la sentencia o cuerpo de la instrucción.

El bucle while: mientras tanto, el bucle while es el que controla una serie de instrucciones que se repiten, siempre y cuando la condición sea verdadera. La sintaxis general de cómo se ve un bucle while se define de la siguiente manera: `while (condicion)` acompañado del contenido o el cuerpo de la instrucción del bucle.

El bucle do-while: es una estructura de control de la mayoría de los lenguajes de programación estructurados cuyo propósito es ejecutar un bloque de código y repetir la ejecución mientras se cumpla cierta condición expresada en la cláusula while.

14. Análisis de Algoritmos

Dado que el objeto fundamental de este tema son los algoritmos vamos a empezar por establecer una definición de lo que es un algoritmo: Secuencia finita de instrucciones donde cada una de las cuales tiene un significado preciso y puede ejecutarse con una cantidad finita de recursos computacionales en un tiempo finito.

El análisis de algoritmos es una herramienta para hacer la evaluación del diseño de un algoritmo, permite establecer la calidad de un programa y compararlo con otros que puedan resolver el mismo problema, sin necesidad de desarrollarlos. El análisis de algoritmos estudia, desde un punto de vista teórico, los recursos computacionales que requiere la ejecución de un programa, es decir su eficiencia (tiempo de CPU, uso de memoria, ancho de banda, ...). Además de la eficiencia en el desarrollo de software existen otros factores igualmente relevantes: funcionalidad, corrección, robustez, usabilidad, modularidad, mantenibilidad, fiabilidad, simplicidad y aún el propio costo de programación.

El análisis de algoritmos se basa en:

- El análisis de las características estructurales del algoritmo que respalda el programa.
- La cantidad de memoria que utiliza para resolver un problema.
- La evaluación del diseño de las estructuras de datos del programa, midiendo la eficiencia de los algoritmos para resolver el problema planteado.
- Determinar la eficiencia de un algoritmo nos permite establecer lo que es factible en la implementación de una solución de lo que es imposible.

- Tipos de Análisis:

- Peor caso (usualmente): $T(n)$ = Tiempo máximo necesario para un problema de tamaño n
- Caso medio (a veces): $T(n)$ = Tiempo esperado para un problema cualquiera de tamaño n

(Requiere establecer una distribución estadística)

- Mejor caso (engañoso): $T(n)$ = Tiempo menor para un problema cualquiera de tamaño n

Los criterios para evaluar programas son diversos: eficiencia, portabilidad, eficacia, robustez, etc. El análisis de complejidad está relacionado con la eficiencia del programa. La eficiencia mide el uso de los recursos del computador por un

algoritmo. Por su parte, el análisis de complejidad mide el tiempo de cálculo para ejecutar las operaciones (complejidad en tiempo) y el espacio de memoria para contener y manipular el programa más los datos (complejidad en espacio). Así, el objetivo del análisis de complejidad es cuantificar las medidas físicas: tiempo de ejecución y espacio de memoria y comparar distintos algoritmos que resuelven un mismo problema.

El tiempo de ejecución de un programa depende de factores como:

- - los datos de entrada del programa
- - la calidad del código objeto generado por el compilador
- - la naturaleza y rapidez de las instrucciones de máquina utilizadas
- - la complejidad en tiempo del algoritmo base del programa

El tiempo de ejecución debe definirse como una función que depende de la entrada; en particular, de su tamaño. El tiempo requerido por un algoritmo expresado como una función del tamaño de la entrada del problema se denomina complejidad en tiempo del algoritmo y se denota $T(n)$. El comportamiento límite de la complejidad a medida que crece el tamaño del problema se denomina complejidad en tiempo asintótica. De manera análoga se pueden establecer definiciones para la complejidad en espacio y la complejidad en espacio asintótica.

El tiempo de ejecución depende de diversos factores. Se tomará como más relevante el relacionado con la entrada de datos del programa, asociando a un problema un entero llamado tamaño del problema, el cual es una medida de la cantidad de datos de entrada.

15.- Complejidad en el tiempo

La complejidad en el tiempo es el número de operaciones que realiza un algoritmo para completar su tarea (considerando que cada operación dura el mismo tiempo). El algoritmo que realiza la tarea en el menor número de operaciones se considera el más eficiente en términos de complejidad temporal. Sin embargo, la complejidad espacial y temporal se ve afectada por factores como el sistema operativo y el hardware, pero no los incluiremos en discusión.

Ahora para entender la complejidad temporal, tomaremos un ejemplo en el que compararemos dos algoritmos diferentes que se utilizan para resolver un problema concreto.

El problema es la búsqueda. Tenemos que buscar un elemento en un arreglo (en este problema, vamos a suponer que el arreglo está ordenado de forma ascendente). Para resolver el problema tenemos dos algoritmos:

1. Búsqueda lineal.
2. Búsqueda binaria.

Tomemos como ejemplo un arreglo con diez elementos, y tenemos que encontrar el número diez en el arreglo.

```
const arreglo = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
```

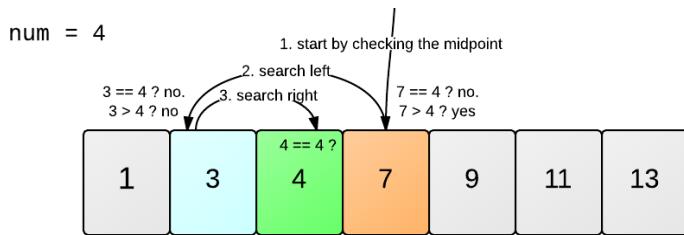
```
const digito_buscado = 10;
```

El algoritmo de búsqueda lineal comparará cada elemento del arreglo con digito_buscado. Cuando encuentre el digito_buscado en el arreglo regresara true.

Ahora vamos a contar el número de operaciones que realiza. En este caso, la respuesta es 10 (ya que compara cada elemento del arreglo). Entonces la búsqueda lineal utiliza diez operaciones para encontrar el elemento dado (este es el número máximo de operaciones para este arreglo; este caso también es conocido como el peor caso de un algoritmo).

En general la búsqueda lineal tardará un número n de operaciones en su peor caso (donde n es el tamaño del arreglo).

Analizaremos el algoritmo de búsqueda binaria para este caso.



Si intentamos aplicar esta lógica en nuestro problema entonces, primero compararemos `digito_buscado` con el elemento central del arreglo, es decir 5. Ahora como 5 es menor que 10, entonces empezaremos a buscar el `digito_buscado` en los elementos del arreglo mayores que 5, de la misma manera hasta que obtengamos el elemento deseado 10.

Ahora, intenta contar el número de operaciones que la búsqueda binaria ha necesitado para encontrar el elemento deseado. Se necesitaron aproximadamente cuatro operaciones. Este fue el peor caso de la búsqueda binaria. Esto demuestra que existe una relación logarítmica entre el número de operaciones realizadas y el tamaño total del arreglo.

$$\text{número de operaciones} = \log(10) = 4 \text{ (aprox.)}$$

para la base 2

Podemos generalizar este resultado para la búsqueda binaria como:

Para un arreglo de tamaño n , el número de operaciones realizadas por la búsqueda binaria es: $\log(n)$

La búsqueda binaria puede entenderse fácilmente con este ejemplo: En muchos casos, la complejidad de tiempo de un algoritmo es igual para todas las instancias de tamaño n del problema. En otros casos, la complejidad de un algoritmo de tamaño n es distinta dependiendo de las instancias de tamaño n del problema que resuelve. Esto nos lleva a estudiar la complejidad del peor caso, mejor caso, y caso promedio.

Por otro lado, para un tamaño dado (n), la complejidad del algoritmo en el peor caso resulta de tomar el máximo tiempo (complejidad máxima) en que se ejecuta el algoritmo, entre todas las instancias del problema (que resuelve el algoritmo) de tamaño n ; la complejidad en el caso promedio es la esperanza matemática del tiempo de ejecución del algoritmo para entradas de tamaño n , y la complejidad mejor caso es el menor tiempo en que se ejecuta el algoritmo para entradas de tamaño n .

Por defecto se toma la complejidad del peor caso como medida de complejidad $T(n)$ del algoritmo.

Aún cuando son los programas que los llegan realmente a consumir recursos computacionales (memoria, tiempo, etc.) sobre una máquina concreta, se realiza el análisis sobre el algoritmo de base, considerando que se ejecuta en una máquina hipotética.

- La complejidad de un programa depende de:
- La máquina y el compilador utilizados
- El tamaño de los datos de entrada que depende del tipo de datos y del algoritmo
- El valor de los datos de entrada.
- Complejidad temporal: Se define como el tiempo que tarda un algoritmo en una entrada de tamaño n .
- La complejidad en el caso peor proporciona una medida pesimista, pero fiable.

Dado que se realiza un estudio teórico de la complejidad, ignorando aspectos como las características de la máquina y el compilador, se tiene en cuenta que las diferencias en eficiencia se hacen más significativos para tamaños grandes de los datos de entrada se analiza la complejidad en términos de su comportamiento asintótico, dejando de lado la forma exacta de la función de complejidad.

16. Complejidad en espacio

La misma idea que se utiliza para medir la complejidad en tiempo de un algoritmo se utiliza para medir su complejidad en espacio. Decir que un programa es $O(n)$ en espacio significa que sus requerimientos de memoria aumentan proporcionalmente con el tamaño del problema. Esto es, si el problema se duplica, se necesita el doble de memoria.

Del mismo modo, para un programa de complejidad $O(n^2)$ en espacio, la cantidad de memoria que se necesita para almacenar los datos crece con el cuadrado del tamaño del problema: si el problema se duplica, se requiere cuatro veces más memoria. En general, el cálculo de la complejidad en espacio de un algoritmo es un proceso sencillo que se realiza mediante el estudio de las estructuras de datos y su relación con el tamaño del problema.

Los requerimientos estáticos de memoria se refieren al tamaño de los objetos que resuelven el problema. El espacio requerido por los objetos de datos de los tipos primitivos y de los tipos estructurados de los lenguajes de programación, son fácilmente calculables, dependiendo de la implementación del mismo.

Se les conoce como objetos de datos estáticos porque la cantidad de espacio que requieren es deducible en su declaración, por lo que ya en compilación (si el lenguaje es compilado), el compilador conoce cuánta memoria requerirán, que será invariante a lo largo del alcance del objeto.

El análisis de los requerimientos dinámicos de memoria es relativo a los lenguajes que proveen mecanismos de asignación dinámica de la misma. En este caso, la complejidad en espacio viene dada por la cantidad de objetos existentes en un punto del programa, según las reglas de alcance.

Así, la cantidad de espacio requerido no será la sumatoria de todas las declaraciones de datos, sino la máxima cantidad de memoria en uso en un momento dado de la ejecución del programa.

El problema de eficiencia de un programa se puede plantear como un compromiso entre el tiempo y el espacio utilizados. Por eso, la etapa de diseño es tan

importante dentro del proceso de construcción de software, ya que va a determinar en muchos aspectos la calidad del producto obtenido.

17. Eficiencia de Algoritmos

Un algoritmo será más eficiente cuanto menos recursos computacionales consuma: tiempo y espacio de memoria requerido para su ejecución.

La eficiencia de un algoritmo se cuantifica en términos de su complejidad temporal (tiempo de cómputo del programa) y su complejidad espacial (memoria que utiliza el programa durante su ejecución).

¿Para que emplear tiempo en diseñar algoritmos eficientes si los ordenadores tienen cada vez más recursos computacionales?

En una computadora capaz de procesar un dato cada 0.0001 seg, ¿Cuánto tiempo se tarda en ejecutar un algoritmo de coste exponencial 2^n ?

N	Tiempo
10	Aprox. 0.1 s
20	Aprox. 2 min
30	> 1 día
40	> 3 años

Y ¿con una computadora capaz de procesar un dato 100 veces más rápido?

Tabla 1.2 Tiempo de Ejecución algoritmo de orden 2^n

N	Tiempo
50	35 años

La selección de un algoritmo es un proceso en el que se deben tener en cuenta muchos factores, entre los cuales podemos destacar los siguientes:

La complejidad en tiempo del algoritmo. Es una primera medida de la calidad de una rutina y establece su comportamiento cuando el número de datos a procesar es grande.

La complejidad en espacio del algoritmo. Sólo cuando esta complejidad resulta razonable es posible utilizar este algoritmo con seguridad. Si las necesidades de memoria del algoritmo crecen considerablemente con el tamaño del problema el rango de utilidad del algoritmo es baja y se debe descartar.

La dificultad de implementar el algoritmo. En algunos casos el algoritmo óptimo puede resultar tan difícil de implementar que no se justifique desarrollarlo para la aplicación que se le va a dar.

El tamaño del problema que se va a resolver. Un algoritmo para un proceso tamaño pequeño no justifica la realización de un análisis, ya que da lo mismo una implementación que otra.

17. Cuadro comparativo entre procedimiento iterativos vs recursivos

CICLOS O ITERACIONES		RECURSIVIDAD
DEFINICIONES	<p>“Los ciclos son también llamados iteraciones, se usan en programación para ejecutar el mismo conjunto de instrucciones hasta que se cumpla cierta condición.” (Universidad Autónoma del Estado de Hidalgo, 2011)</p> <p>“Las estructuras repetitivas (bucles) son aquellas que reiteran una o un grupo de instrucciones “n” veces y dependen de una variable de control del ciclo. Es decir, ejecutan una o varias instrucciones un número de veces definido.” (González, 2014)</p>	<p>“La recursividad es una técnica de programación que se utiliza para realizar una llamada a una función desde ella misma, de allí su nombre.” (Delphin, 2009)</p> <p>“Técnica de programación muy potente que puede ser usada en lugar de la iteración.” (LCC, 2013)</p> <p>“La recursividad es aquella propiedad que posee un método por el cual puede llamarse a sí mismo” (Chávez, 2016)</p> <p>De las definiciones anteriores podemos decir que la recursividad es aquella propiedad que tienen los métodos de llamarse a sí mismos para poder resolver un problema que es repetitivo.</p>
VENTAJAS	<p>Las ventajas del uso de ciclos son las siguientes:</p> <ul style="list-style-type: none"> • Trabajan más rápido en memoria ya que ocupa menos tiempo de ejecución. • No utiliza mucha memoria. • Puede trabajar con un número de repeticiones mucho más grande que la recursividad. • Ejecuta una línea de código a la vez. <p>Ejemplos:</p> <p>//Método para sumar n números de uno en uno</p> <pre>public int sumarN(int n){ int suma=0; for(int i=0;i<=n;i++){ suma+=i; } return suma; }</pre> <p>//Método que imprime una cuenta regresiva</p> <pre>public void cuentaRegresiva1(int n){ for(int i=n;i>=0;i--){ System.out.print(i+"-"); } }</pre>	<p>Las ventajas de la recursividad son las siguientes:</p> <ul style="list-style-type: none"> • Son programas cortos. • Solucionan problemas recurrentes. • Solucionan problemas que no necesariamente son lineales. • Da soluciones más naturales, lógicas y elegantes. • Resuelve problemas que los ciclos no pueden resolver. <p>Ejemplos:</p> <p>//Método que regresa el factorial de un número</p> <pre>public int factorial(int n){ if(n==1){ return 1; } return n*factorial(n-1); }</pre> <p>//Serie de Fibonacci</p> <pre>public int Fibonacci(int n){ if(n==0) return 0; else{ if(n==1) return 1; else return Fibonacci(n-2)+Fibonacci(n-1); } }</pre>
DESVENTAJAS	<p>Las desventajas de utilizar ciclos son:</p> <ul style="list-style-type: none"> • Son difíciles de leer si existen ciclos anidados • No pueden o es difícil resolver ciertos problemas que no tienen una estructura lineal 	<p>Las desventajas de utilizar la recursividad es:</p> <ul style="list-style-type: none"> • Consumir muchos recursos de memoria al trabajar con los datos. • No puede trabajar con una gran cantidad de datos ya que llena el espacio en memoria y por ende finaliza su ejecución sin éxito.

- En algunas ocasiones encontrar la solución llega a ser difícil.

Ejemplo:

```
//Método que muestra la serie de Fibonacci
//Este método es más complicado de deducir
//que el método con recursividad
public static void Fibonacci2(int n){

    if(n==0){
        System.out.print("0-");
    }
    else{
        if(n==1){
            System.out.println("1-");
        }
        else{
            int aux2=1;
            int aux3=1;
            for(int i=2;i<=n;i++){
                System.out.print(aux3+"-");
                aux3=aux2+aux3;

                aux2=aux3-aux2;
            }
        }
    }
}
```

- Utiliza mucho tiempo de ejecución.

Ejemplos:

```
//Método para sumar números de uno en uno
//Manda excepción si el número llega a ser muy
//grande como 10,000
public int sumarRec(int n){

    if(n<=0){
        return 0;
    }
    else{
        return n+sumarRec(n-1);
    }
}

//Método que muestra un cuenta regresiva
//Al igual que el método anterior este manda una
//excepción si la cuenta es demasiado grande
public void cuentaRegresiva2(int n){

    System.out.println(n);
    if(n>0){
        cuentaRegresiva2(n-1);
    }
}
```

RECOMENDACIONES

De manera general las recomendaciones serían las siguientes:

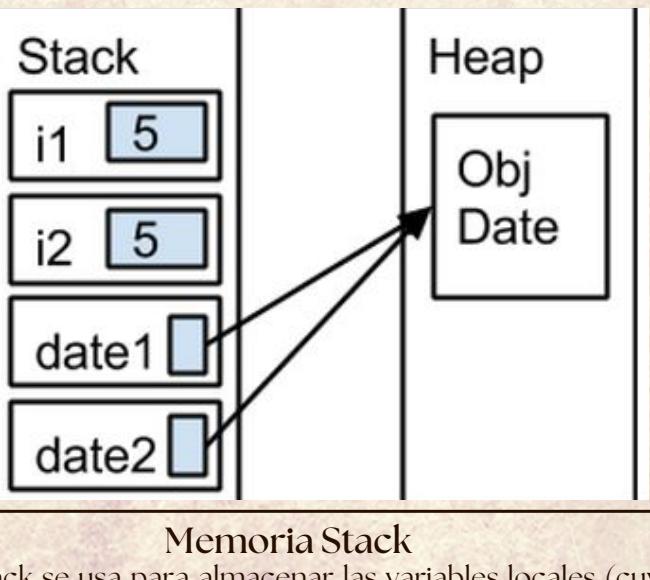
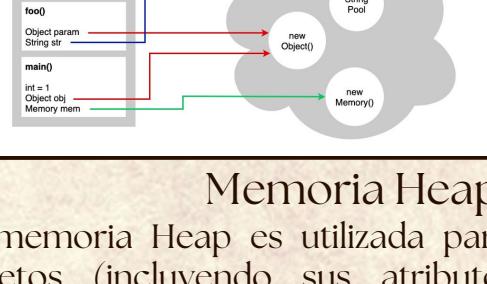
- Si se desea utilizar la recursividad para resolver un problema repetitivo hay que tener en cuenta que cada llama ocupa 4 bytes de memoria, y cada parámetro otros 4 bytes, por lo cual el problema a resolver no debe tener un número excesivo de repeticiones, esto variará de computadora en computadora dependiendo el hardware con el que la misma cuente.
- El uso de la recursividad se recomienda en estructuras no lineales como los árboles, además de problemas que con el uso de ciclos serían difíciles de resolver, tal es el caso de la serie de Fibonacci o en problemas sobre combinaciones.
- El uso de ciclos es recomendable para repeticiones más prolongadas ya que el trabajo se hace más rápido y utiliza menos memoria.
- Es importante recalcar que para cada método recursivo existe un método equivalente que funciona con ciclos, sin embargo en el caso contrario no siempre es posible.

Investigación de Stack, heap, Etc.

TIPOS DE MEMORIA

1. TIPOS DE MEMORIA

Un programa usa la memoria RAM para almacenar la información (del programa) mientras se está ejecutando. Al interior de Java existen dos clasificaciones para almacenar los valores del programa, estas son conocidas como memoria Stack y memoria Heap.



Memoria Stack

La memoria Stack se usa para almacenar las variables locales (cuyo ámbito de acción está limitada solo a la función donde se declaró) y también las llamadas de funciones en Java. Las variables almacenadas en esta memoria por lo general tienen un periodo de vida corto, viven hasta que terminen la función o método en el que se están ejecutando.

Aquí se guardan las variables y sus valores de tipos de datos primitivos (booleanos, números, strings, entre otros).

Memoria Heap

la memoria Heap es utilizada para almacenar los objetos (incluyendo sus atributos), los objetos almacenados en este espacio de memoria normalmente tienen un tiempo de duración más prolongado que los almacenados en Stack.

La memoria Heap es un poco más lenta, pero nos permite almacenar grandes cantidades de información y dentro de esta memoria es posible guardar los valores de los objetos y las instancias de nuestra clase.

Esta zona de memoria es estática y no se modifica durante el desarrollo del programa.



PUNTOS IMPORTANTES

¿Cuánto duran los elementos en el Heap?
 • Los objetos no referenciados del Heap serán eliminados cuando el GarbageCollector decida pasar.

¿Cuánto duran los elementos en el Stack?
 • En cuanto finalice el método, las referencias a objetos, las variables y todo lo relacionado con el método desaparece del stack.



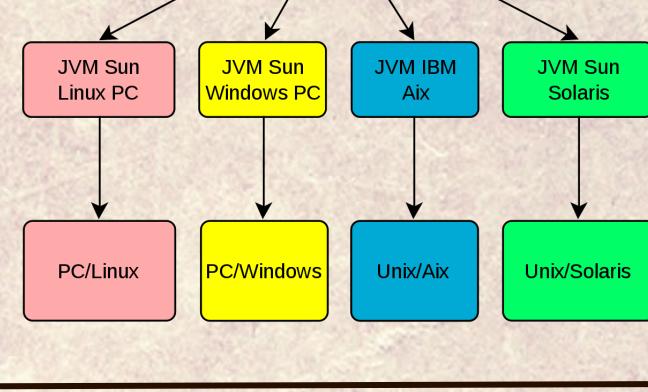
Stack

Heap

2. ¿QUE SE ALMACENA EN CADA MEMORIA USADA POR JAVA VIRTUAL MACHINE DE JAVA.?

Al mismo tiempo que los archivos .class se están cargando y empezando a inicializar, pasan también por los distintos tipos de memoria de la máquina virtual de Java:

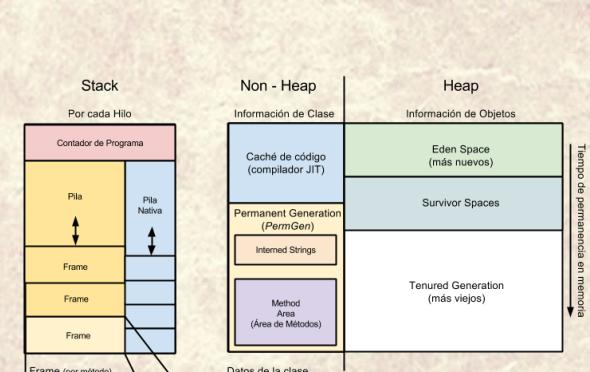
- Área de métodos: Aquí se almacena toda la información sobre los distintos niveles de las clases del programa desarrollado en Java.
- Montón: Almacena toda la información de los objetos que contiene el código del programa desarrollado en Java.
- Pila: Para cada uno de los subprocesos, la máquina virtual de Java crea una pila en tiempo de ejecución que se almacena en esta parte de la memoria de la JVM.
- Registros: Almacena las instrucciones de ejecución de cada uno de los subprocesos que genera la máquina virtual de Java.



3. EXPLIQUE Y DE UN EJEMPLO DEL USO DE LAS MEMORIAS AL MOMENTO DE EJECUTAR UN PROGRAMA (DEBERÁ DE INCLUIR LA CLASE QUE SIRVE DE BASE PARA SU EXPLICACIÓN).

En Java podemos hablar de tres zonas de memoria

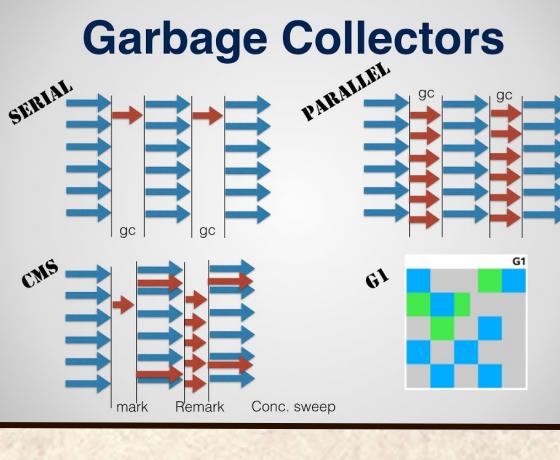
- La ZONA DE DATOS, donde se almacenan las instrucciones del programa, las clases con sus métodos y constantes.
- La memoria HEAP, donde se almacenan las variables que se crean en los programas con la sentencia "new".
- La memoria STACK, donde se almacenan los métodos y las variables locales que se crean dentro de los mismos.
- Garbage Collector, es un proceso automático de baja prioridad que se ejecuta dentro de la JVM. Se encarga de limpiar aquella memoria del HEAP que ya no se utiliza y, por tanto, podría ser utilizada por otros programas.



GARBAGE COLLECTOR

EL PROCESO GC TAMBIÉN SE PUEDE EJECUTAR MANUALMENTE DESDE UN PROGRAMA JAVA SI CREEMOS QUE EN UN PUNTO DETERMINADO HEMOS PODIDO DEJAR MUCHOS OBJETOS SIN REFERENCIAR. A CONTINUACIÓN, SE MUESTRA MEDIANTE UN PEQUEÑO PROGRAMA JAVA CÓMO SE PUEDE CALCULAR LA MEMORIA LIBRE EN UN MOMENTO DETERMINADO EN LA JVM Y CÓMO EJECUTAR EL GARBAGE COLLECTOR MANUALMENTE DESDE UN PROGRAMA:

```
PACKAGE ES.JPASCU.TEST;  
IMPORT JAVA.UTIL.ARRAYLIST;  
IMPORT JAVA.UTIL.LIST;  
PUBLIC CLASS TESTJVMMEMORY {  
PRIVATE STATIC FINAL LONG MEGABYTE = 1024L * 1024L;  
PUBLIC STATIC LONG BYTESTOMEGBYTES(LONG BYTES) {  
RETURN BYTES / MEGABYTE;  
}  
PUBLIC STATIC VOID MAIN(STRING[] ARGS) {  
// VOY LLENANDO MEMORIA HEAP CON OBJETOS QUE LUEGO NO TENDRÁN  
REFERENCIA.  
  
// ESTA MEMORIA SERÁ LIBERADO POR LA GC DE JAVA  
FOR (INT I = 0; I <= 1000000; I++) {  
ITEM ITEM = NEW ITEM("I", "UN MURCIANO EN EL POLO", 2);  
ITEM = NULL;  
}  
  
// OBTENEMOS LA RUNTIME DE JAVA  
RUNTIME RUNTIME = RUNTIME.GETRUNTIME();  
// EJECUTAMOS EL PROCESO DE GARBAGE COLLECTOR  
RUNTIME.GC();  
  
// CALCULAMOS LA MEMORIA MÁXIMA Y LA MEMORIA DISPONIBLE  
LONG MEMORY = RUNTIME.TOTALMEMORY() - RUNTIME.FREEMEMORY();  
LONG MEMORIAX = RUNTIME.MAXMEMORY();  
SYSTEM.OUT.PRINTLN("MEMORIA MÁXIMA EN BYTES: " + MEMORIAX);  
SYSTEM.OUT.PRINTLN("MEMORIA MÁXIMA EN MB: "  
+ BYTESTOMEGBYTES(MEMORIAX));  
SYSTEM.OUT.PRINTLN("MEMORIA UTILIZADA EN BYTES: " + MEMORY);  
SYSTEM.OUT.PRINTLN("MEMORIA UTILIZADA EN MB: "  
+ BYTESTOMEGBYTES(MEMORY));  
}
```



EJEMPLO RECURSIVO

```
public static int numPot(int n1, int n2) {  
if(n2<=0){  
    return 1;  
} else {  
    return n1 * numPot(n1, n2-1);  
}  
}
```

Aplicando con lo del ejemplo anterior, tenemos este método en recursividad, en cuál es que mediante un número ingresado va a pedir a qué potencia se eleva.

Para esto se va a pedir dos valores, de que tipo entero, llamadas n1 y n2. Posteriormente mediante el caso base va a preguntar que si el n2 es menor e igual a cero, si el valor que le hayamos asignado a n2 es 0 nos retornará el 1, ya que como sabemos un número elevado a la potencia cero da como resultado 1. En caso contrario si n2 tiene un valor mayor a cero, entonces hará lo siguiente:

Retornará el valor del n1 por nombre del método y seguido de eso en el parámetro volverá a pedir el n1 por valor de n2 -1. Al hacer eso se ejecutará la operación, cómo operación aritmética en palabras simples. Y ya al momento de realizarla lanzará un mensaje de que la potencia de tus números es y el resultado que nos dio.

ESTRUCTURA DE DATOS

CLAVE 3A3A

HORARIO DE 15:00-16:00

INTEGRANTES DEL EQUIPO

MEDINA BERMÚDEZ ANGEL EDUARDO - 21010201

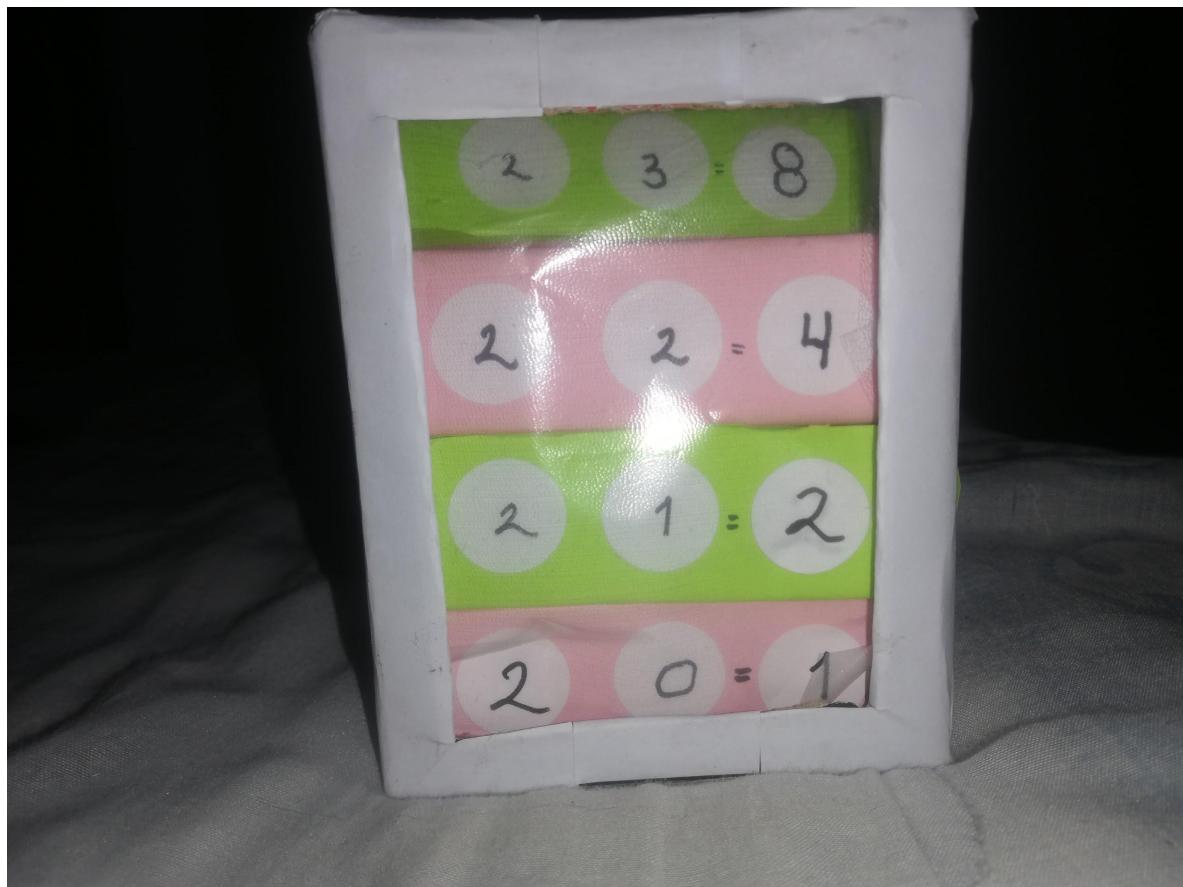
HERNÁNDEZ DÍAZ ARIEL - 21010195

NICANOR FLORES CARLOS EDUARDO - 21010202

PALACIOS MENDEZ XIMENA MONSERRAT 21010209

Nota: Fotos del trabajo didáctico del algoritmo recursivo presentado, de acuerdo a la presentación que realizo en el salón de clase.

Método recursivo de potencia de un número.



RECURSOS, MATERIALES Y EQUIPO

- Computadora
- Java
- Lectura de los materiales de apoyo del tema 2 (AULA PADLET)
- Notas de clase (problemas resueltos en clase y materiales de trabajo de la profesora)

MÉTODOS RECURSIVOS

Problema 1.- El problema 1 nos pide que de forma recursiva mostremos una lista de números desde el 1 hasta n número. El código es el siguiente:

```
public static String funcionRecursiva(int j , int n) {  
    if(j<=n) {  
        //System.out.println("Incremento"+j);  
        return "\n"+j+funcionRecursiva(j+1,n);  
    }  
    else {  
        return "";  
    }  
}
```

En el código anterior mostrado, se explica que por medio del caso base es como el método se sigue llamando así mismo, de tal forma que cuando j sea igual a n el programa automáticamente termina. Para esto j vale 1 y n podemos ponerle cualquier valor un ejemplo puede ser 10, entonces cuando retorne regresara a donde fue llamado en este caso al mismo método y así incrementa j hasta llegar a 10, y ya que llegue a 10 automáticamente el programa se desapila y queda vacía la pila. Pero por otro lado que si J fuera mayor a N solo va a concatenar, pero no mostrara algún resultado como tal. Cabe mencionar que los datos se fueron almacenando en el espacio de memoria que se abrió (pila) y se fue apilando, y cuando el caso base termina automáticamente los datos se van desapilando uno por uno hasta que ya no haya nada, y la pila este vacía. **A continuación, se muestra el resultado de dicho método recursivo:**



Problema 2.- En este problema nos pide declarar un arreglo con valores y posteriormente visualizar el mismo en pantalla. Aquí el código del método Imprime Array:

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

En este código nos pide un arreglo definido con valores y posteriormente visualizarlo, entonces aquí por medio de el caso base nos dice que j tiene que ser menor a la longitud del arreglo, si lo es va a retornar a donde fue llamado, en este caso al mismo método, no sin antes incrementar el valor de j, así para que cuando j sea mayor se termine el método. Al final se sabe que se desapila y queda vacía la pila. Pero por otro lado que si J fuera mayor a la longitud del arreglo solo va a concatenar, pero no mostrara algún resultado como tal. **Aquí la ejecución del método:**



Problema 3.- El tercer problema nos pide crear un método con la función de Fibonacci, se trata de una sucesión que comienza en 0 y 1, y a partir de ello el siguiente número es la suma de los dos anteriores, en este caso hasta llegar al numero que queremos que llegue esta sucesión. Presento el código del método:

```

public static String fibonacci(int posicion, String res, int aux,
    int a, int b) {
    if (posicion > 0) {
        return fibonacci(posicion-1, res+"" +a,
            aux=a, a=b, b=aux+b);
    }
    return res;
}

```

Para esto vamos a necesitar 5 variables dentro del parámetro en este caso la posición será nuestra pieza para que el caso base funcione correctamente, En ese caso imprimimos la sucesión hasta la posición 20 (contando el 0). Si el caso base es verdad, entonces retornara el método a donde fue llamado, en este caso así mismo. Así hasta llegar al tope de la posición ingresada, en automático de que el programa termine se desapila y se queda vacía la memoria (Pila):



Problema 4.- Este otro método nos pide un numero decimal y posteriormente pasarlo al sistema octal, para esto de nuevo se usa el caso base, en la cual nos dice que si el dato es menor o igual a cero nos retornara el mismo dato, ya que es imposible sacar su octal de un numero negativo o cero, en otro caso entonces hará una operación para sacar su residuo y su cociente en caso de que ya no se pueda dividir el dato. Aquí el método:

```

public static int decimalAOctal(int decimal) {

    if (decimal<=0)
        return decimal;
    else
        return decimal%8+decimalAOctal(decimal/8)*10;
}

```

Obviamente en este caso no incrementa, solo hace la operación y regresa a donde fue llamado, y por último el método termina y se desapila. **Esto resultó:**



Problema 5.- Método de clase de tipo entero que pida 2 números y determine si son amigos o no (por medio del resultado de la operación).

Aquí el código del método:

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

En este método nos pide dos valores uno que controle el incremento y el otro que vamos a utilizar para mostrar el numero amigo de dicho numero que lleguemos a ingresar, usamos de nuevo el caso base que ya viene siendo una regla en esta función de recursividad, solo usamos un caso base el cual va a manejar el tope hasta llegar a ser mayor del dato este termina y se desapila los valores ingresados. Aquí se va a incrementar y regresa a donde fue llamado. Nos retorna el resultado de la suma final. En mi caso ingresé el 220 y para el valor de k le puse el número 1. Cabe mencionar que los datos se fueron almacenando en el espacio de memoria que se abrió (pila) y se fue apilando, y cuando el caso base termina automáticamente los datos se van desapilando uno por uno hasta que ya no haya nada, y la pila este vacía.

Presento el resultado:



El resultado esta bien y nos dice que si son números amigos.

Problema 6.- Método de clase de tipo entero que pida un numero decimal y lo convierta en un numero binario. Aquí el código del método numBinario:

```
public static int numBin(int num) {  
    if(num<2) return num;  
    else return num%2+numBin(num/2)*10;  
}
```

Este método es similar al método de decimal a octal, solo que en este caso es convertir decimal a binario, se toma en cuenta el dato para usarlo en el caso base, para esto si el caso es verdadero y el dato que se introdujo fue un valor negativo, 0,1 tal caso pues ya no se hace nada ya que al cero o 1 dará el mismo resultado. Por otro lado, si dato es mayor a 2 hará la operación para sacar su residuo y cociente en el caso de que ya no se pueda dividir mas el numero introducido. Para esto junto de la operación se manda a llamar al método y como retorna entonces regresa a donde fue llamado, de tal forma regresa ahí mismo. Para finalizar a lo ultimo nos muestra el resultado siempre y cuando ya se haya terminado el caso base, si es así el programa se desapila y termina.

Aquí la ejecución del método:



En este caso yo ingrese el 3 como dato para pasar a binario, y entonces esto me resulto, haciendo la operación de forma manual, nos damos cuenta de que el resultado es correcto.

Problema 7.- Método que reciba como parámetro un valor entero y muestre el resultado de ese numero elevado a la potencia. (Sin usar la función Math.pow)

Aquí el método realizado:

```

public static int numPot(int n1, int n2) {
    if(n2<=0) {
        return 1;
    } else {
        return n1 * numPot(n1, n2-1);
    }
}

```

En este método nos pide como parámetro un valor (el numero) y otro (valor a elevar ese número). El segundo dato lo utilizaremos como pivote para realizar el caso base, tomaremos como valores el 2 para el dato1 y el 0 para el dato2, si llegase a ser igual a cero automáticamente ingresa esos valores y los apila, y así como entran se desapilan finalizando el método.

Por otro lado, como otro ejemplo para el dato1 le asignaremos el número 2 y para el dato2 le asignaremos el 3, entonces en el caso base nos pregunta si el dato 2 es igual a cero, no lo es ya que nosotros ingresamos el 3, así que se va a la parte falsa, y aquí toma de referencia las variables que anteriormente nos pidió, así que para dato1 toma el 2 y para el dato2 toma el 3, haciendo la operación de forma manual, nos resulta el 8, ese 8 se almacena primero junto con los datos (dato1 ósea el 2 y dato2 el numero 3), posteriormente, hace esto que donde resta el dato2 menos 1 ya se convierte en un nuevo valor, en este caso ya es un 2. Entonces vuelve a regresar a donde fue llamado y vuelve hacer lo mismo, como el dato2 sigue siendo mayor hace la misma acción, hasta que cuando restamos el dato2 (que ya es un 1) menos 1, se convierte en cero. Ya con nuevos valores, vuelve a preguntar dentro del caso base si el dato2 es menor o igual a cero, y si no es menor, pero si es igual, entonces nos regresa un 1 el cual es el resultado de los nuevos valores, y para finalizar ya con esto todos valores que se apilaron se desapilan, hasta que ya no queda nada en pocas palabras pila vacía.

Aquí el ejecutable del método:

Caso verdadero



Caso falso



Problema 8.- Método de clase de tipo String que reciba como parámetro un arreglo, lo ordene y por último lo muestre ordenado.

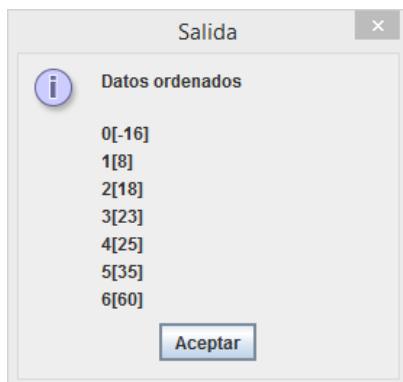
Aquí los códigos del Método Ordenamiento burbuja:

```
public static String verArray(int j, int datos[]) {  
    if (j<datos.length) {  
        return "\n" + j + "["+datos[j]+"]"+verArray((j+1),datos);  
    }  
    else {  
        return "";  
    }  
}  
public static String ordenaBurbuja(int datos[], int i) {  
    if(i<datos.length-1) {  
        return burbujaIntercambio(datos,i,i+1)+ordenaBurbuja(datos, i+1);  
    }else return "";  
}  
  
public static String burbujaIntercambio(int datos[], int i, int j) {  
    int aux=0;  
    if (j<datos.length) {  
        if(datos[i]>datos[j]) {  
            aux=datos[i];  
            datos[i]= datos[j];  
            datos[j]= aux;  
        }  
        return burbujaIntercambio(datos,i,j+1);  
    }  
    return "";  
}
```

En el código de burbuja intercambio es donde vamos a intercambiar de lugar los valores que definimos en el arreglo, en el caso base pregunta que, si j es menor a la longitud del arreglo y si ya que j vale 0, entra a la parte donde se intercambian

los valores, comenzando desde el inicio del arreglo, se compara cada par de elementos adyacentes. Si ambos no están ordenados (el segundo es menor que el primero), se intercambian sus posiciones. En cada iteración, un elemento menos necesita ser evaluados (el último), ya que no hay más elementos a su derecha que necesiten ser comparados, puesto que ya están ordenados. Y esto se va controlando con la *j* que es la que va controlando las vueltas, regresando a donde fue llamado. Ya que el caso base se cumple el programa termina, pero no sin antes mencionar que este método necesita de otro para ordenar la burbuja y en este caso llama al método de *burbujaIntercambio* y en este caso incrementa solo la variable (*i*). Por otro lado, en el método de *verArray* nos muestra el array ya ordenado. Algo que hay que comentar es que para el método de *verArray* el método base nos dice que si *j* sigue siendo menor a la longitud del arreglo va a ir concatenando los valores para posteriormente verlos en pantalla. Y en la de *ordenaBurbuja* es similar a diferencia de que retorna al mismo método y llama al método anterior e incrementa solo la variable (*i*) en este y el método anterior. Cabe mencionar que los datos se fueron almacenando en el espacio de memoria que se abrió (pila) y se fue apilando, y cuando el caso base termina automáticamente los datos se van desapilando uno por uno hasta que ya no haya nada, y la pila este vacía.

Para finalizar ya con el método *verArray* nos muestra el arreglo implementado con el ordenamiento burbuja, y así fue el resultado:



Estos fueron los valores del arreglo, y como sabemos es correcto. Además de que con el ultimo método hecho ya el método termina y desapila los valores, como he comentado anteriormente.

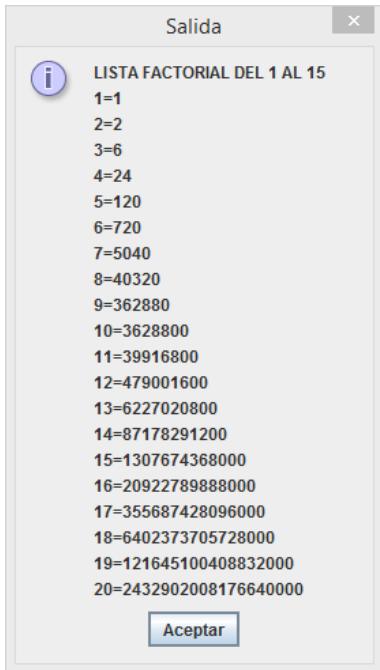
Problema 9.- Método de clase que reciba como parámetro 2 valores enteros y posteriormente muestre las factoriales de n hasta 20.

Aquí el código del método Factorial:

```
public static String factorial(int n, long fact) {  
    if (n<=20) {  
        return n+"="+fact+"\n"+factorial(n+1, fact*(n+1));  
    }  
    return "";  
}
```

En este método como parámetro nos pide 2 valores, el primero que en este caso es la n nos pedirá con que valor vamos a empezar la lista y este mismo nos ayudara en el caso base el cual va desde n hasta 20, mientras n sea menor a 20 si es verdadero hará la operación la cual va a multiplicar el elemento por los anteriores. Por ejemplo, si ingresamos el 5 multiplicara el $5 \times 4 \times 3 \times 2 \times 1$ y después su resultado el 120. En este método su objetivo es mostrar una lista de factoriales de n hasta 20, el caso base dice que si n es menor o igual a 20, pero como a la variable n le pusimos como valor 1 entonces si es verdad, va retornar el valor de n (concatenando el valor de fact en este caso con el valor de 1) y como se explico hace un momento va a ir haciendo la operación multiplicando por el anterior hasta que llegue n a valer 20 y es así como el método termina no sin antes desapilar los valores ingresados dentro de la pila. Pero por otro lado que si N fuera mayor a 20 solo va a concatenar, pero no mostrara ningún resultado como tal. Cabe mencionar que los datos se fueron almacenando en el espacio de memoria que se abrió (pila) y se fue apilando, y cuando el caso base termina automáticamente los datos se van desapilando uno por uno hasta que ya no haya nada, y la pila este vacía.

Si hizo lo que se quería visualizar. Aquí el ejecutable del método:



Problema 10.- En este método pide que como parámetro un valor entero y muestre ese valor, pero en octal, pero aquí va a mostrar una lista de n al 20 y su correspondiente en octal de forma inversa.

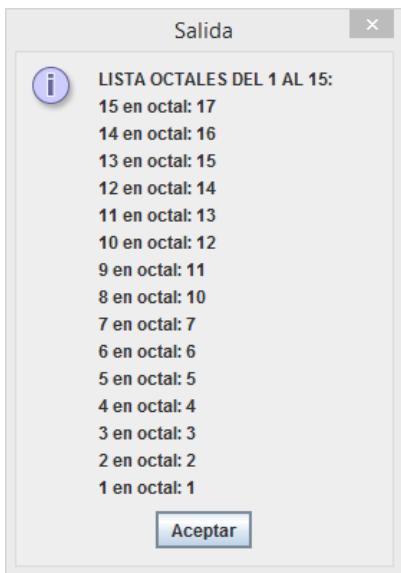
Aquí el código del método toOctal

```
public static String toOctal(int numero) {  
    if (numero > 0)  
        return numero + " en octal: " + Integer.toOctalString(numero) + "\n" + toOctal(numero - 1);  
    else return "";  
}
```

En este método se ocupó la función que por default eclipse (java) nos da para pasar un numero a octal que es la de Integer.toOctalString así de esta manera nos será más fácil que cuando queramos mostrar la lista no tengamos que hacer tanta operación. Por otro lado, en el caso base nos dice que si el numero es mayor a 0 entonces hará la acción de pasar dichos números a lo antes mencionado, después seguido de hacer el llamado al mismo método para así dar las vueltas (de forma inversa) hasta que llegue el numero a valer 0 es entonces que el método concatena y muestra el resultado en pantalla, pero por otro lado que si el numero no fuera mayor a 0 y fuera menor solo va a concatenar, pero no mostrara algún resultado como tal. Cabe mencionar que los datos se fueron almacenando en el

espacio de memoria que se abrió (pila) y se fue apilando, y cuando el caso base termina automáticamente los datos se van desapilando uno por uno hasta que ya no haya nada, y la pila este vacía.

Aquí el resultado del método:



Problema 11.- En este presente método su objetivo es ingresar y mostrar datos en una matriz de 4×4 , y se usara la recursividad indirecta. Tendremos como parámetro dos variables de tipo entero.

Aquí los códigos fuentes del método leerMatriz, leerMatriz2 y verMatriz:

```

public static String leerMatriz(int arr[][],int i,int j) {
    String cad="";
    if (i<arr.length)
        return cad+"\n"+leerMatriz2(arr ,i,0)+leerMatriz(arr,i+1,j);
        else return "";
}
public static String leerMatriz2(int arr[][],int i,int j) {
    if(j<arr[0].length) {
        arr[i][j]= ToolsPanel.leerInt("DATO");
        return arr[i][j]+ " " + leerMatriz2(arr,i,j+1);
    }
    else {
        return " ";
    }
}
public static String verMatriz(int arr[][],int i,int j) {
    if(j<arr[0].length) {
        return "\n"+arr[i][j]+ " "+verMatriz(arr,i+1,0);
    }
    else {
        return "";
    }
}

```

Bueno en este método lo que se quiere es ingresar datos en una matriz de 4 x 4 y mostrarlos. Primero creamos el método de leerMatriz2 el cual vamos a usar para ingresar los datos en la matriz, en el parámetro vamos a tener como variables i y j las cuales vamos a usar (una como nuestra columna y la otra como nuestra fila) el caso base nos dirá que si j es menor al tamaño del arreglo (en este caso j vale 0) entonces sí es verdad así que vamos a ingresar los datos asignándoselos al arreglo en i y j, después retornaremos el arreglo con los valores que se ingresaron anteriormente, seguido de la llamada al mismo método la cual se va hacer un incremento así hasta que termine ese método y termine el método. Para esto en el método de leerMatriz la variable i será nuestra otra ayuda, ya que con este ingresaremos los datos en las filas haciendo un llamado al método de leerMatriz2. Básicamente que leerMatriz y leerMatriz2 se ingresan los datos de manera individual, pero se toman de la mano para llamarse indirectamente y así esos datos regresan a LeerMatriz2. Por otra parte, el método de verMatriz nos va a visualizar los datos ingresados para las filas y columnas. Por si no se menciono para verMatriz y leerMatriz2 el caso base necesitara J para que el método funcione e igual se incremente al igual que la i. Mientras tanto en leerMatriz la i hace la misma acción. Cabe mencionar que los datos se fueron almacenando en el espacio de memoria que se abrió (pila) y se fue apilando, y cuando el caso base

termina automáticamente los datos se van desapilando uno por uno hasta que ya no haya nada, y la pila este vacía.

Puedo mencionar que en este método la recursividad indirecta fue de gran ayuda porque de un método hicimos otro y otro, para llamarlos de uno hacia otro.

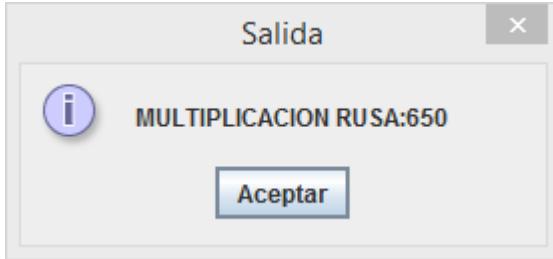
Sin más aquí el método ya resuelto:



Problema 12.- En este método de tipo entero nos pide como parámetro dos variables enteras las cuales utilizaremos para pedir 2 valores. **Aquí el código del método de multiplicación rusa:**

```
public static int multiplicacionRusa(int n1,int n2) {  
    if(n1==1) {  
        return (n2);  
    }  
    else {  
        if(n1%2!=0) return (n2+multiplicacionRusa(n1/2, n2*2));  
        else return (multiplicacionRusa(n1/2, n2*2));  
    }  
}
```

Se ponen los múltiplos como cabeceras de dos columnas. Simultáneamente a uno de los números lo multiplican por 2 y al otro lo dividen por 2. Esto se repite hasta que el número que está siendo dividido llega a 1. Luego se toman los valores de la columna multiplicada y se suman sólo aquellos que en su contra parte tengan un número impar. Y así queda el método ya resulto. **Aquí el resultado:**



En este caso tomamos el valor de 50 para n_1 y 13 para n_2 y nos dio el resultado correcto de 650. Ya con el caso base concluido, automáticamente los datos que se apilaron al principio, se desapilan, como dice el ultimo que entra es el primero que sale y el primero que entro es el último que sale.

Problema 13.- El método de tipo entero pide como parámetro 2 valores de tipo entero y mostrar su resultado de la función Ackermann de dichos números.

Aquí el código de función Ackermann:

```
public static long ackermann(int m, long n) {  
    if (m==0) {  
        return n+1;  
    }else if(m>0 && n==0) {  
        return ackermann(m-1,1);  
    }else {  
        return ackermann(m-1, ackermann(m,n-1));  
    }  
}
```

El código anterior consta de la función llamada "Ackermann" que recibe dos parámetros del tipo "int". La función sigue las reglas de la función de Ackerman por lo que si al evaluar, el parámetro "m", este vale cero, la función regresará el valor de "n" más uno, si no es el caso, la comprobación se hará para el valor de "n = 0", si es el caso, la función se mandará a llamar a sí misma y se pasará como argumentos a "m" con valor de "m - 1" y "n" con el valor de "n = 1" y regresará el resultado de llamarse a sí misma. Finalmente, si "m" y "n" son valores mayores a cero " $m > 0$ ", " $n > 0$ ", el valor que la función regresará será el resultante de llamarse a sí misma con los parámetros "m-1" para "m" y el resultado de llamarse a sí misma con los parámetros "m = m - 1" y "n = n - 1" para "n". con lo anterior dicho como ejemplo a m se le asigno como valor el 0 y a n el valor de 4 se fue al primer caso base y **resultado lo siguiente:**



En caso de que m sea 1 y n sea 4 dará el siguiente resultado:



Aquí lo que paso es que para el primer caso base comparo si m era igual a 0 pero no lo era, así que fue al segundo caso base y tampoco ya que m vale 1 y por consiguiente n era mayor y no igual a 0, y en el ultimo caso tampoco, pero hizo esto que el valor de n le resto 1 y de esa manera fue dando en total de 5 vuelta y como, pues restándole el valor que había en n menos 1. Ya en la quinta vuelta con los valores nuevos (m con el valor 1 y n con el valor 0) comparo con el primer caso base y no se pudo, pero con el segundo caso base si cumplió ya que m era mayor a 0 y n era igual a 0 entonces retorna a donde fue llamado en este caso al mismo método no sin antes restarle al valor de m 1 dándonos 0 como nuevo valor para m, volvió al primer caso base y ahora pregunta que si m es igual a cero y si, entonces retorna la suma de n + 1, que como sabemos n vale 1, ya con n que vale 2, vuelve al mismo caso base y vuelve a preguntar si m es igual a cero y si, otra vez hace la misma acción hasta que cuando n vale 5 se suma con el 1 y retorna 6 como se mostró en el código ejecutable. En cualquiera de los casos, siempre donde se quedarán es en el primer caso base y este es el que hará los incrementos del valor de N Y el segundo de restarle 1 a M Y el tercero en restarle 1 a M y a N. Cabe mencionar que los datos se fueron almacenando en el espacio de memoria que se abrió (pila) y se fue apilando, y cuando el caso base termina

automáticamente los datos se van desapilando uno por uno hasta que ya no haya nada, y la pila este vacía.

Problema 14.- El método de tipo boolean nos pide como parámetro una variable de tipo String y que con esta determinemos si es palíndromo o no.

Aquí el código del método Palíndromo:

```
public static boolean esPalindrome(String texto){  
    texto= texto.replaceAll(" ", " ");  
    if(texto.length() <= 1)  
    {  
        return true;  
    }else  
    {  
        if(texto.charAt(0) == texto.charAt(texto.length()-1))  
        {  
            return esPalindrome(texto.substring(1, texto.length()-1));  
        }else  
        {  
            return false;  
        }  
    }  
}
```

En este método se busca determinar si un a frase o palabra es palíndromo o no, para esto como ejemplo tendremos la palabra “reconocer” para eso por medio del primer caso base dice que si el tamaño de longitud de la variable texto que se ingresó en el parámetro es menor a 1 devuelve un true, pero vemos que el tamaño del texto supera al 1 entonces se va a la parte falsa, y es aquí donde nos dice como caso base que si cada letra de la variable texto es igual a cada letra del mismo pero de forma inversa, viendo que la palabra es “reconocer” es claro que es la misma de forma inversa, así que cuando retorna y aparte hace el llamado al mismo método extrae la primer y ultima letra de la palabra, después vuelve hacer lo mismo con la palabra que hay que ahora es “econoce” va al primer caso base y como vemos sigue siendo superior a 1 en cuanto a tamaño, así que se va al segundo caso base y hace de nuevo el llamado al mismo método y otra vez quita el ultimo y primer letra de la letra y nos sale una nueva palabra. Se hace los mismo así lo sucesivamente hasta que nadamas queda la letra “n” y llegando al primer caso base nos vuelve a decir que si de lo que quedo de la palabra

“reconocer” en este caso el tamaño de lo que queda que es “n” es menor o igual a 1, entonces vemos que si es verdad ya que solo queda una sola letra a lo cual automáticamente nos retornara un true y con eso nos dice que la palabra si es palíndromo. Anexo el resultado en pantalla:



En caso de que la palabra no sea palíndromo como por ejemplo la palabra “perro” hará la misma acción anterior solo que esta vez en el segundo caso base al ver que la palabra así normal y de forma inversa no es la misma automáticamente retornara un false.



Cabe mencionar que los datos se fueron almacenando en el espacio de memoria que se abrió (pila) y se fue apilando, y cuando el caso base termina automáticamente los datos se van desapilando uno por uno hasta que ya no haya nada, y la pila este vacía.

Problema 15.- En este método de tipo entero, se necesitan tres variables de tipo entero, la cual una de ellas nos maneja el tope hasta donde lo queremos, otra nos maneja la tabla del número que queremos multiplicar y por último otra que lleve el numero de vueltas. **Aquí el código fuente del método Tabla de Multiplicar:**

```

public static String multiplicationTable(int i, int num, int tope) {
    int res=0;
    String table = "";
    if (i <=tope) {
        res = num *i;
        table+= num + "*" + i + "=" + res + "\n" + multiplicationTable(i+1, num,tope);
    }
    return table;
}

```

El caso base manejará el método, y este dice que si “i” es menor o igual al tope (para la variable i le asignaremos el valor de 1 y para tope el valor de 10) así que como aun i es menor al tope entra , y es aquí donde el valor que le asignamos a la variable num (el valor de 5 como ejemplo) se multiplicara por i y el resultado se almacenara en la variable “res”. Ya después va a concatenar la variable “num”, “i”, y “res”, seguido de esto hace el llamado al mismo método y aquí es donde los datos se van almacenando en la pila y como pues por cada vuelta que se va dando por medio de la variable “i”, así hasta llegar al tope dado, ya que se terminó esa parte retornara lo antes concatenado y así los datos antes apilados, se desapilan, haciendo que termine el método.

Aquí el resultado:



En estos métodos para evitar llamar cada uno por medio del main, se creo otra clase la cual tendrá un menú con todos los métodos recursivos y es la siguiente:

TESTMENU

```
package Menus;
```

```

import javax.swing.JOptionPane;
import Metodos.FuncionesRecursivas;
import ToolsPanel.ToolsPanel;

public class TestPruebaRecursiva {
    public static void menu3(){
        String sel="";
        do {
            sel=TestPruebaRecursiva.listaDesplegable();
            switch (sel) {
                case "Funcion Recursiva":
                    ToolsPanel.imprime("LISTA DEL ARREGLO:
\n"+FuncionesRecursivas.funcionRecursiva(ToolsPanel.leerInt("Ingresa un dato"),
ToolsPanel.leerInt("Ingresa un dato")));
                    break;
                case "Imprime Array":
                    int a[]={1,2,3,7,8,9,12};
                    ToolsPanel.imprime("LISTA DEL ARREGLO:
\n"+FuncionesRecursivas.imprimeArray(a, 0));
                    break;
                case "Fibonacci":
                    ToolsPanel.imprime("SUCESION:\n"+FuncionesRecursivas.fibonacci(20, sel, 0,0 ,
1));
                    break;
                case "Tabla de multiplicar":
                    ToolsPanel.imprime("TABLA DE
MULTIPLICAR\n"+FuncionesRecursivas.multiplicationTable(ToolsPanel.leerInt("Ing
resa un numero a multiplicar"),ToolsPanel.leerInt("Ingresa otro numero para
multiplicar el primero"),10));
                    break;
            }
        }
    }
}

```

```

case "decimalAOctal":
    ToolsPanel.imprime("NUMERO
OCTAL:\n"+FuncionesRecursivas.decimalAOctal(ToolsPanel.leerInt("Ingresa un
dato para pasar a octal")));
    break;
case "Suma Amigos":
    ToolsPanel.imprime("NUMEROS AMIGOS:
\n"+FuncionesRecursivas.SumaDivi(ToolsPanel.leerInt("Ingresa tu numero
amigo"), 1));
    break;
case "Numero binario":
    ToolsPanel.imprime("NUMERO
BINARIO:\n"+FuncionesRecursivas.numBin(ToolsPanel.leerInt("Ingresa un dato
para pasar a binario")));
    break;
case "Potencia de un numero":
    ToolsPanel.imprime("POTENCIA
"+FuncionesRecursivas.numPot(ToolsPanel.leerInt("Ingresa un numero"),
ToolsPanel.leerInt("Ingresa tu numero a elevar")));
    break;
case "Ordena Burbuja":
    int dat[] = {35,8,-16,25,60,18,23};
    FuncionesRecursivas.ordenaBurbuja(dat, 0);
    ToolsPanel.imprime("Datos ordenados
\n"+FuncionesRecursivas.verArray(0, dat));
    break;
case "Factorial del 1 al 15":
    ToolsPanel.imprime("LISTA FACTORIAL DEL 1 AL
15\n"+FuncionesRecursivas.factorial(ToolsPanel.leerInt("Ingresa un numero para
iniciar"), 1));
    break;

```

```

case "Octales del 1 al 20":
    ToolsPanel.imprime("LISTA OCTALES DEL 1 AL
20:\n"+FuncionesRecursivas.toOctal(ToolsPanel.leerInt("Ingresa un numero como
tope")));
    break;
case "Matrices":
    int vec[][] = new int[4][4];
    ToolsPanel.imprime("MATRICES DE 4 X 4 \n"
+FuncionesRecursivas.leerMatriz(vec,0,0));
    break;
case "Multiplicacion Rusa":
    ToolsPanel.imprime("MULTIPLICACION
RUSA:"+FuncionesRecursivas.multiplicacionRusa(ToolsPanel.leerInt("Ingresa un
dato"),ToolsPanel.leerInt("Ingresa otro dato")));
    break;
case "Funcion Akermann":
    ToolsPanel.imprime("FUNCION
AKERMANN\n"+FuncionesRecursivas.akermaNN(ToolsPanel.leerInt("Ingresa un
dato"),ToolsPanel.leerInt("Ingresa otro dato")));
    break;
case "Palindromo":
    ToolsPanel.imprime("RESULTADO PALINDROMO\n"
+FuncionesRecursivas.esPalindrome(ToolsPanel.leerString("Ingresa una frase")));
    break;
}
} while(!sel.equalsIgnoreCase("Salir"));
}

public static String listaDesplegable() {
    String valores[] = {"Funcion Recursiva", "Imprime
Array", "Fibonacci", "Tabla de multiplicar", "decimalAOctal", "Suma Amigos", "Numero

```

```

binario", "Potencia de un numero", "Ordena Burbuja", "Factorial del 1 al 15", "Octales
del 1 al 20", "Matrices", "Multiplicacion Rusa", "Funcion
Akermann", "Palindromo", "Salir"};
String res = (String) JOptionPane.showInputDialog(null, "M E N U",
"Selecciona
opcion:", JOptionPane.QUESTION_MESSAGE, null, valores, valores[0]);
return (res);
}

public static void main(String[] args) {
// TODO Auto-generated method stub
TestPruebaRecursiva.menu3();
}
}

```

De hecho en cada ventana emergente de ejecución de cada método se probó y si funciono.

MÉTODOS ITERATIVOS

1- Uso del while en funciones iterativas

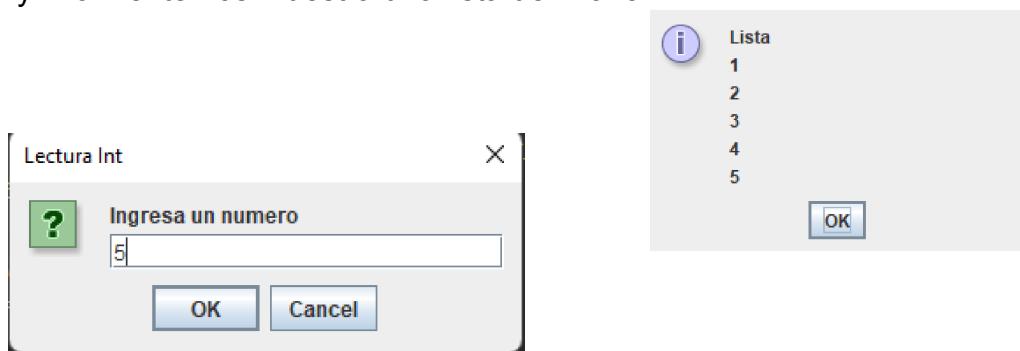
A la palabra reservada while le sigue una condición encerrada entre paréntesis. El bloque de sentencias que le siguen se ejecuta siempre que la condición sea verdadera

En este caso tenemos la impresión de una lista de números usando el condicional while donde declaramos **n** y **j** como enteros ,dándole valor a **j** de 1 , donde **n** es el tope o el límite de números que queramos que imprima la lista , inicializamos la variable de tipo carácter **cad**, y una vez declaradas las variables entra al while donde mientras **j** sea mayor o igual a **n** que es el número de caracteres que queramos que tenga la lista ,**cad** guardará los números que vayan incrementando

en **j** ,para el final saliendo del while imprima la lista más todos los números que almaceno **cad**

```
public static void usowhile(int n) {  
    int j=1;  
    String cad="";  
    while(j<=n) {  
        cad+="\n"+j++;  
    }  
    ToolsPanel.imprime("Lista " +cad);  
}
```

Como resultado tenemos que al imprimir pide la cantidad de números a imprimir **n** y finalmente nos muestra una lista del 1 al 5



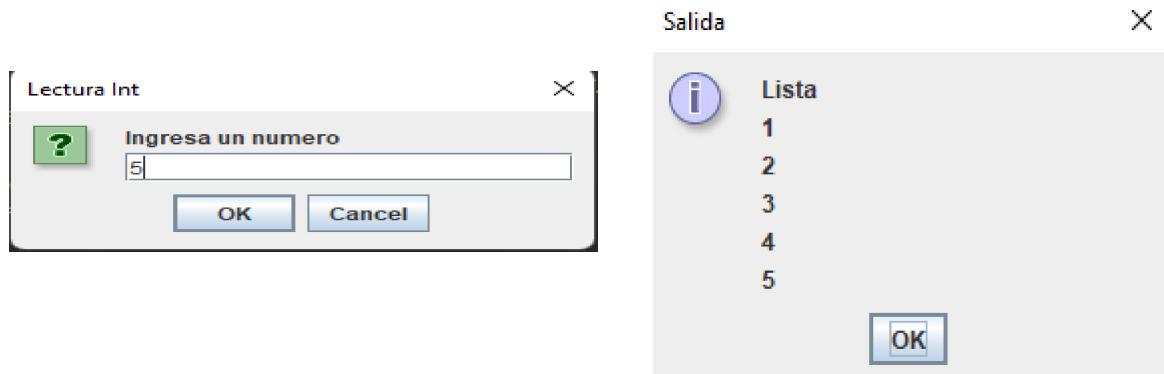
2- Uso del ciclo for en funciones iterativas

El bucle for se emplea cuando conocemos el número de veces que se ejecutará una sentencia o un bloque de sentencias, tal como se indica en la figura. La forma general que adopta la sentencia for es

Para este caso la función que realiza el for es la misma que el while el cual imprime una lista de números según la cantidad que quiera el usuario, donde **n** sigue siendo la cantidad de números que quiere que tenga la lista , y una vez inicia el ciclo for donde inicializa **j** con 1 y mientras **j** se mayor o igual a lo que hay en **n** , **j** incrementará ,hasta **j++**

```
public static void usoFor(int n) {  
    for (int j=1; j<=n; j++) {  
        j++;  
    }  
}
```

Como resultado tenemos que al imprimir pide la cantidad de números a imprimir `n` y finalmente nos muestra una lista del 1 al 5



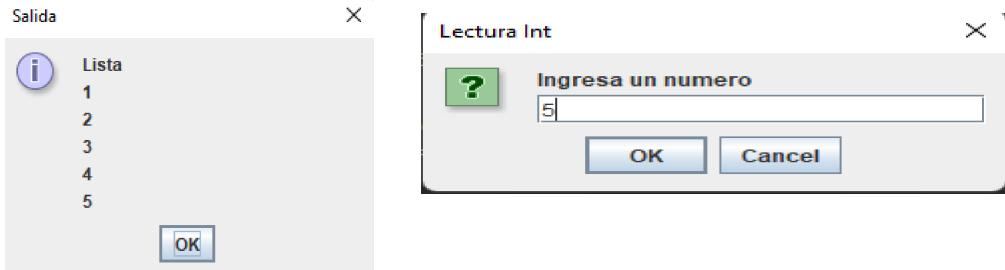
3.- Uso del Do While en funciones iterativas

Los bucles do-while y while en Java son los únicos que quedan en la lista de artículos sobre estructuras de Java que incluyen una condición booleana

Para este caso la función es la misma que while y for, donde `n` sigue siendo la cantidad de números que queremos que tenga la lista, para luego inicializar `j` como entero dándole el valor de 1, para luego entrar al do while, donde irá incrementando `j` siempre y cuando `j` sea mayor o igual a `n`.

Como resultado tenemos que al imprimir pide la cantidad de números a imprimir `n`

```
public static void usoDoWhile(int n) {  
    int j=1;  
    do {  
        j++;  
    }while(j<=n);  
}
```



4.- Imprimir un arreglo como función iterativa

Un arreglo es una estructura de datos utilizada para almacenar datos del mismo tipo. Los arreglos almacenan sus elementos en ubicaciones de memoria contiguas. En Java, los arreglos son objetos. Todos los métodos pueden ser invocados en un arreglo.

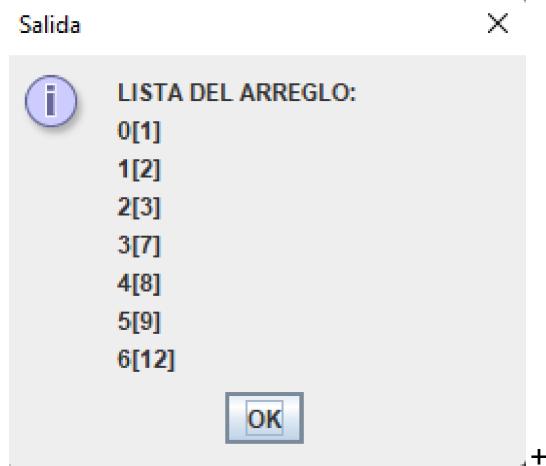
En este ejemplo tenemos la impresión de una lista pero haciendo uso de un arreglo donde `a[]` es el arreglo que va a contener los números de la lista, se declara una variable de tipo carácter `lista`, y entra a un ciclo for donde se inicializa `j` con 0, y mientras `j` sea mayor a lo que haya dentro del vector, `j` incrementará dentro de `lista` que es igual a `j` en la posición que se encuentre `j` dentro de el vector `a[]`, y al final imprime el contenido que hay dentro de `lista`

```
public static void imprimeArray(int a[]) {
    String lista="";
    for(int j=0; j<a.length;j++) {
        lista+=j+"["+a[j]+"]\n";
    }
    ToolsPanel.imprime("LISTA DEL ARREGLO:\n"+lista);
}
```

Para obtener el resultado del array se necesita agregar los números dentro del arreglo directamente en el menú

```
case "Imprime Array":  
    int a[] = {1,2,3,7,8,9,12};  
    iterativas.imprimeArray(a);  
    break;
```

Y ya por ultimo tenemos como resultado la lista de números ingresados dentro del arreglo



5.- Tabla de multiplicar como función iterativa

En este ejemplo tenemos el método de la tabla de multiplicar donde se inicializa `numero` como entero , `tabla` como carácter,al igual que `res` (donde se almacenará el resultado),`tope`(que es el número hasta el que llegara la tabla),para luego entrar a un ciclo for donde se inicializa `i` con 1 ,y mientras `i` sea mayor o igual a `tope` ,`i` incrementara dentro de `res` dentro de la que el número se multiplica por el incremento `i` , lo cual se almacenará dentro de `tabla` que nos guardara cada elemento de la tabla de multiplicar para que una vez fuera del ciclo imprima la `tabla` .

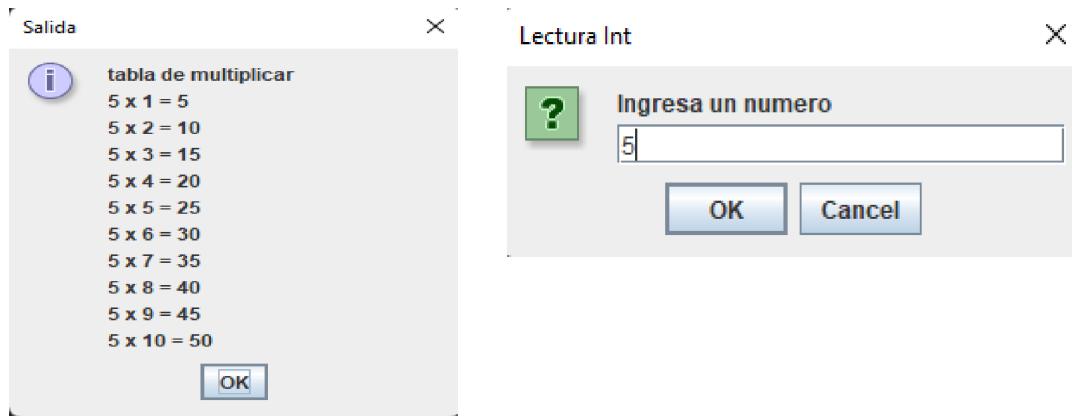
```

public static void multiplyTable(int numero) {
    String tabla = "";
    int res=0, tope=10;
    for (int i = 1; i <=tope; i++) {
        res = numero *i;
        tabla+= numero + " x " + i + " = " + res + "\n";
    }
    ToolsPanel.imprime("tabla de multiplicar\n"+tabla);
}

```

Y como resultado nos pedirá primero que tabla de multiplicar deseamos imprimir

Mostrándonos así la tabla del número que hayamos indicado



6-Suma de divisiones como función iterativa

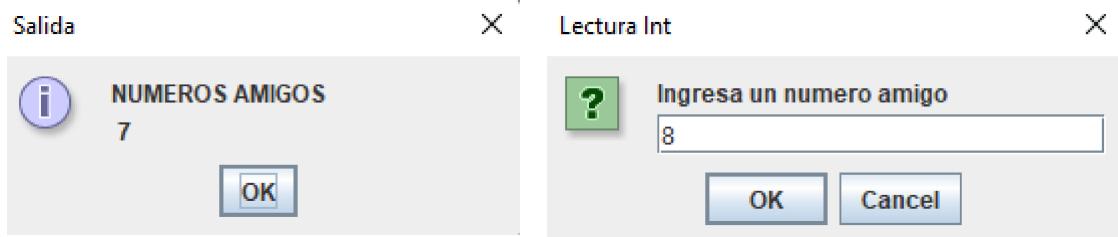
En este ejemplo el método será de tipo entero y haremos la suma de divisiones ,inicializando a `dato` como entero , `i` el cual se inicializa con 1 y `suma` con 0 ,todos de tipo entero,entrando a un ciclo do while ,donde si el residuo que tenga dato es igual a que `i` sea exactamente 0 ,entonces la suma sera igual a que `i` vaya incrementando , mientras que `i` sea mayor a `dato` sal salir del do while retorne lo que hay dentro de `suma` .

```

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

```

Como resultado en consola tenemos que pide un numero amigo, e imprime el total de números amigos que tiene como la suma de los divisores del número 8



7-De decimal a octal como función iterativa

En este ejemplo tendremos que el método será de tipo carácter en la conversión de un número decimal a octal, donde inicializamos `decimal` como entero, `octal` y `caracteres Octales` el cual dentro lleva una secuencia de números del 0 al 7 y estas dos variables de tipo carácter ,para luego entrar a un while donde mientras `decimal` sea mayor a 0 , se inicializa una variable de tipo entero llamada `residuo` el cual almacenará si el residuo de `decimal` es igual a 8, y `octal` se igual a devolver los `caracteres tales` dentro del `residuo` más el mismo `octal` y `decimal` divide el valor en 8 ,para que saliendo del ciclo while retorne lo que contiene `octal` .

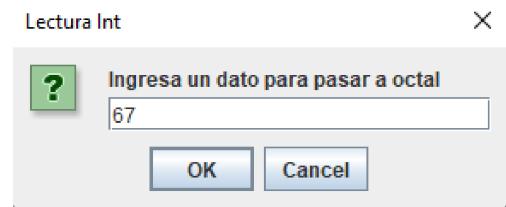
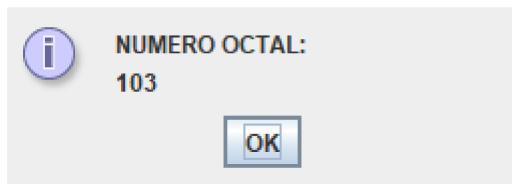
```

public static String decimalAOctal(int decimal) {
    String octal = "";
    String caracteresOctales = "01234567";
    while (decimal>0) {
        int residuo = decimal % 8;
        octal = (caracteresOctales.charAt(residuo) + octal);
        decimal /= 8;
    }
    return octal +"\n";
}

```

Teniendo como resultado en consola que , y nos devuelve el dato en su valor

C Salida



8-Matrices como funciones iterativas

Para este ejemplo será algo diferente pues tenemos dos métodos el primero que nos dejara leer los números que se ingresen a la matriz ([LeerMatriz](#)) y el otro que nos permitirá ver los elementos que encuentran dentro de la matriz ([verMatriz](#))

[Leer Matriz](#) ,en este primer método se creará la matriz `arr[][]` de tipo entero, para luego inicializar `i` y `j` como enteros ,para luego entrar al ciclo for donde `i` será igual con 0 , si `i` es igual a lo que hay dentro del contenido de la matriz `arr[][]` ,`i` va incrementar para luego leer lo que hay dentro de `arr[][]` en las posiciones `[i][j]` .

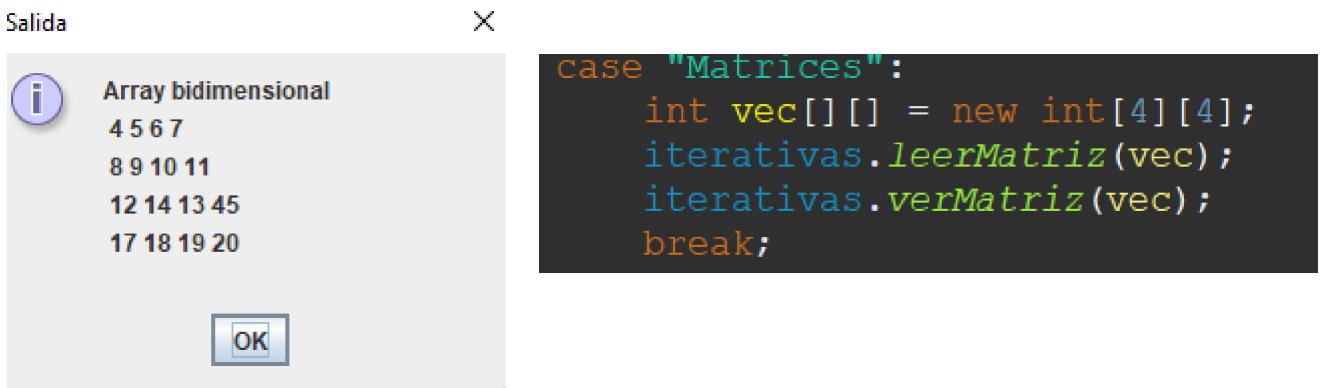
[ver Matriz](#) , en este segundo método se utilizará nuevamente la matriz `arr[][]` ,por lo que se tiene que crear , para luego inicializar `i` y `j` como enteros y la variable `cad` como carácter para luego entrar al ciclo for donde donde `i` será igual con 0 , si `i` es igual a lo que hay dentro del contenido de la matriz `arr[][]` ,`i` va incrementar , y en otro ciclo for donde `j` será igual con 0 , si `j` es igual a lo que hay dentro del contenido de la matriz `arr[][]` ,`j` va incrementar dentro de `cad` el cual va a mostrar lo que hay dentro de `arr[][]` en la posición `[i][j]` , y fuera del segundo for a `cad` solo se le asignará un salto de línea ,para que una vez fuera de ambos ciclos imprima lo que hay dentro de `cad` que es `arr[][]` en la posición `[i][j]` .

```

public static void leerMatriz(int arr[][]) {
    int i,j;
    for (i=0; i<arr.length; i++) {
        for(j=0; j<arr[0].length; j++) {
            arr[i][j]= ToolsPanel.leerInt("DATO");//JOptionPane.showInputDialog("DATO");
        }
    }
}
public static void verMatriz(int arr[][]) {
    int i,j;
    string cad="";
    for (i=0; i<arr.length; i++) {
        for(j=0; j<arr[0].length; j++) {
            cad+=arr[i][j]+" ";
        }
        cad+="\n ";
    }
    ToolsPanel.imprime("Array bidimensional\n "+cad);
}

```

Antes de mostrar en consola se debe asignar en el menú el tamaño de la matriz en este caso será de 4x4 y una vez listo en consola pedirá los valores que irán dentro de la matriz dando como resultado



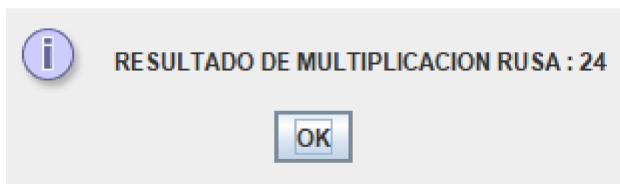
9- Multiplicación Rusa como Función Iterativa

Para este ejemplo tenemos la multiplicación rusa. El método de multiplicación rusa consiste en multiplicar sucesivamente por 2 el multiplicando y dividir por 2 el multiplicador hasta que el multiplicador tome el valor 1. Luego, se suman todos los multiplicandos correspondientes a los multiplicadores impares.

Continuando con el ejemplo inicializamos `a` , `b` y `c` se le asigna 0, siendo estos de tipo entero para luego entrar a un while donde mientras `a` sea distinto de 0 entonces si el residuo de `a` es 2 o distinto de 0 `c` será igual a `c + b` , si no `a` será igual `a` dividido entre 2 y `b` será igual a `b` multiplicado por 2 y al final saliendo del while imprime el resultado de la multiplicación que se encuentra en `c`

```
public static void multiplicacionRusa(int a, int b) {  
    int c=0;  
    while(a!=0){  
        if(a % 2 != 0){  
            c = c + b;  
        }  
        a = a / 2;  
        b = b * 2;  
    }  
    ToolsPanel.imprime("RESULTADO DE MULTIPLICACION RUSA : " +c);  
}
```

Como resultado en consola tenemos que pide dos valores y al realizar las Salida 



10-Número Factorial en funciones iterativas

La función factorial se representa con un signo de exclamación “!” detrás de un número. Esta exclamación quiere decir que hay que multiplicar todos los números enteros positivos que hay entre ese número y el 1;Por ejemplo:

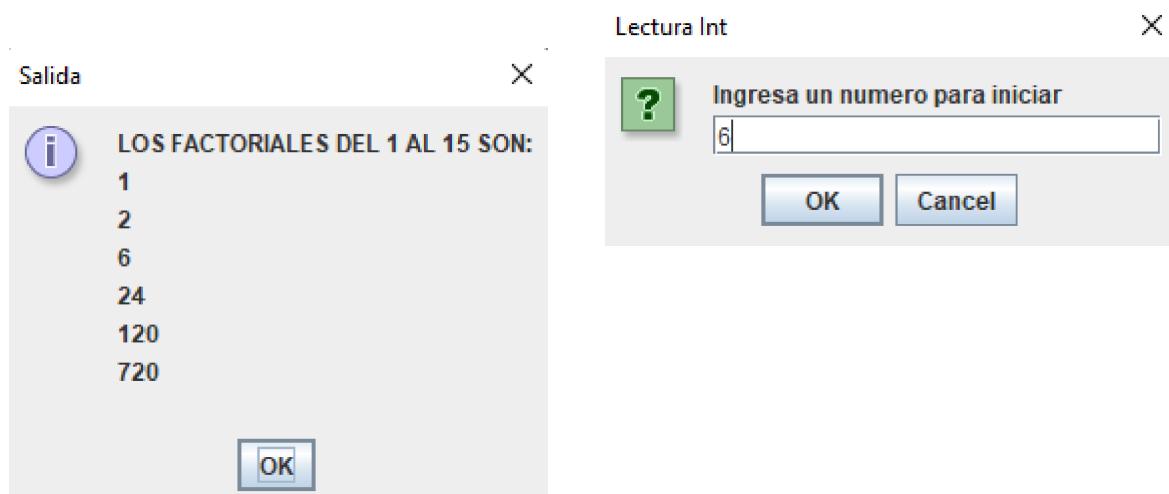
$$6! = 1 \times 2 \times 3 \times 4 \times 5 \times 6 = 720$$

A este número, 6! le llamamos generalmente “6 factorial”, aunque también es correcto decir “factorial de 6”.

Para este ejemplo se inicializará `dato` , `f` con 1 y `c` con 1 ,estas variables de tipo entero ,y `cad` de tipo carácter ,para luego entrar en un while donde mientras `c` sea menor o igual a `dato` , entonces `f` que se multiplica por el valor de `dato` tomará el valor que tenga en `c` ,y `cad` guardará dentro de ella la suma de `f` y `c` irá incrementando ,para que una vez fuera del while imprima los valores del 1 al 15 que hay dentro de `cad`

```
public static void factorial(int dato) {  
    int f = 1, c = 1;  
    String cad="";  
    while(c <= dato) {  
        f *= c;  
        cad+=f+"\n";  
        c++;  
    }  
    ToolsPanel.imprime("LOS FACTORIALES DEL 1 AL 15 SON:\n"+cad+"\n");  
}
```

Mostrando en consola los valores en factorial del 1 al 15 que se encuentran en el dato proporcionado por el usuario , en este caso tomaremos el 6!



11-Números de fibonacci en funciones iterativas

La secuencia de Fibonacci es una sucesión infinita de números naturales, descrita por primera vez por el matemático italiano Fibonacci en el siglo XIII. Esta serie numérica empieza con 0 y 1, siguiendo con la suma de los dos números anteriores hasta el infinito: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233.

Para este ejemplo tenemos la variable `posición` de tipo `long` , al igual que `siguiente` el cual tendrá valor de 1, `actual` con 0,y `temporal` con 0, y de tipo carácter `cad` ,para luego entrar al for donde se declara otra variable de tipo `long` `x` con valor de 1 y mientras `x` sea menor o igual a la `posición` ,`x` irá incrementando para luego `cad` el cual tomará el valor de `actual` ,y `temporal` sea igual a `actual` ,`actual` será igual a `siguiente` y `siguiente` será igual a `siguiente` más `temporal` . Y una vez fuera del ciclo for se imprime la lista de números que se encuentra guardada en `cad` .

```
public static void fibonacci(long posicion) {
    long siguiente = 1, actual = 0, temporal = 0;
    String cad="";
    for (long x = 1; x <= posicion; x++) {
        cad+=actual + ", ";
        temporal = actual;
        actual = siguiente;
        siguiente = siguiente + temporal;
    }
    ToolsPanel.imprime("LISTA DE FIBONACCI:\n"+cad);
}
```

Pidiendo en consola un número tope en este caso 15 y mostrando como resultado de la lista de fibonacci hasta el número tope iniciando de la posición 0



12-Número Potencia como función iterativa

¿Qué es una potencia? Las potencias son una manera abreviada de escribir una multiplicación formada por varios números iguales. Son muy útiles para simplificar multiplicaciones donde se repite el mismo número.

Partes de una potencia: Las potencias están formadas por la base y por el exponente. La base es el número que se está multiplicando varias veces y el exponente es el número de veces que se multiplica la base.

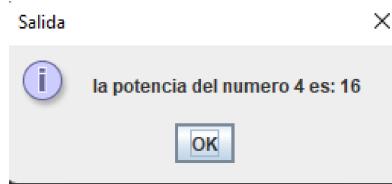
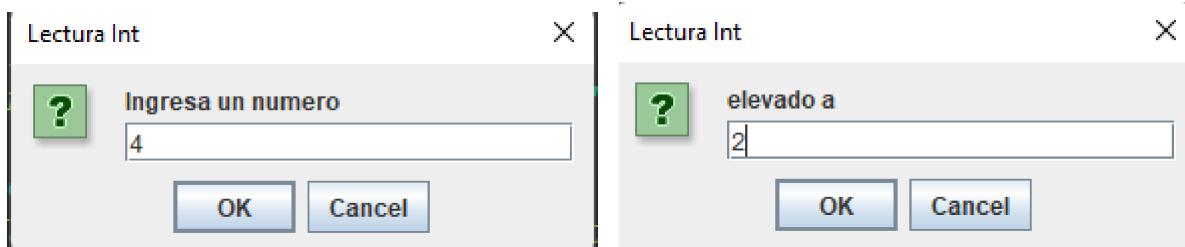
Base: Es el número que se está multiplicando varias veces.

Exponente: Número de veces que se multiplica la base.

Para este ejemplo tenemos que inicializamos dos variables enteras `n1` y `n2` como enteros al igual que `resultado` el cual será igual a lo que haya en `n1`, para luego iniciar el ciclo while donde mientras `n2` sea mayor a 1 el `resultado` es igual a `resultado` multiplicado por `n1` y incrementara `n1--`, y una vez saliendo del while imprime `n1` y el `resultado`

```
public static void numPot(int n1, int n2) {  
    int resultado=n1;  
    while(n2>1){  
        resultado= resultado*n1;  
        n2--;  
    }  
    ToolsPanel.imprime("la potencia del numero " +n1+" es: "+resultado);  
}
```

En consola pedira dos valores primero `n1` el cual sera el numero a elevar y `n2` sera la potencia , Esto nos dará como resultado a 4 elevado a la potencia 2



13-Número AkermaNN en funciones iterativas

En teoría de la computación, una función de Ackermann es una función matemática recursiva encontrada en 1926 por Wilhelm Ackermann. Tiene un

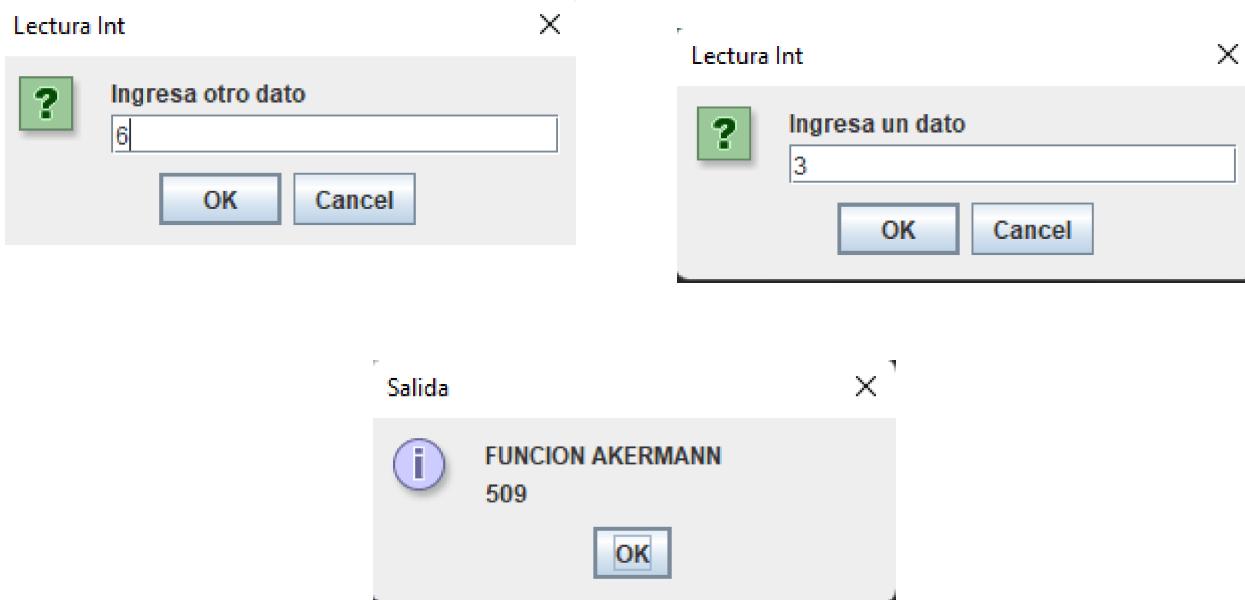
crecimiento extremadamente rápido, lo que es de interés para la ciencia computacional teórica y la teoría de la computabilidad. Hoy en día hay una serie de funciones que son llamadas funciones de Ackermann. Todas ellas tienen una forma similar a la ley original de la función de Ackermann y también tienen un comportamiento de crecimiento similar. Esta función toma dos números naturales como argumentos y devuelve un único número natural. Como norma general se define como sigue:

```
public static long akermann(int m, long n) {
    while (m==0) {
        return n+1;
    }while(m>0 && n==0) {
        return akermann(m-1,1);
    }while(m>0 && n>0) {
        return akermann(m-1,akermann(m,n-1));
    }
    return n;
}
```

Para este ejemplo tendremos que el método será de tipo long en el que inicializamos como enteró `m` y como long `n`, para pasara directamente a un ciclo while donde mientras `m` sea exactamente igual a 0, este retornara `n` mas 1, cerrando este primer while pasamos al segundo el cual mientras `m` sea mayor a 0 y `n` sea exactamente igual a 0, retorna el método el cual tendrá dentro que `m-1`, saliendo del segundo while pasamos al tercer while el cual mientras `m` sea mayor a 0 y `n` sea mayor a 0, retornara el mismo método el cual tendrá dentro `m-1` y el mismo método que lleva dentro `m,n-1`, y por último ya saliendo de este tercer while retornara solamente lo que hay en `n`

$$A(m, n) = \begin{cases} n + 1, & \text{si } m = 0; \\ A(m - 1, 1), & \text{si } m > 0 \text{ y } n = 0; \\ A(m - 1, A(m, n - 1)), & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$

Pidiendo en consola dos datos e imprimiendo al final el resultado de esta función

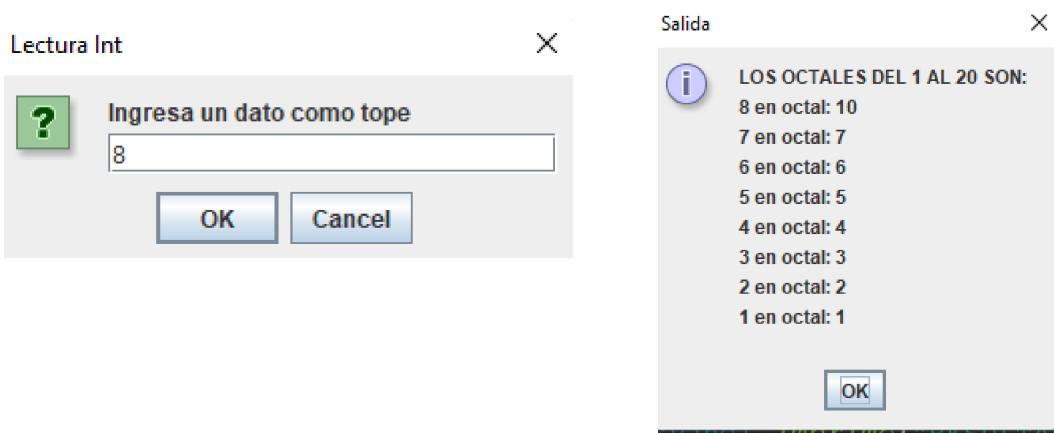


14-Lista de Octales en funciones iterativas

Retomando el tema de los octales en esta ocasión realizaremos una lista de números octales empezando con declarar `dato` y `c` variables de tipo entero y `cad` de tipo carácter entrando al for donde mientras `c` es igual a 1 y `c` sea mayor o igual a `dato`, el `dato` se restaura y se reemplaza por 1, el cual se almacena en `cad` que suma y reemplaza por `dato` utilizando la función `Integer.toOctalString` y agregando el `dato`, y una vez fuera del for imprimir lo que contiene `cad` que es la lista del 1 al 20 del número al cual se volvió a octales

```
public static void listaOctal(int dato) {
    int c;
    String cad="";
    for(c=1; c<=dato; dato-=1 ){
        cad+=dato+" en octal: " + Integer.toOctalString(dato) + "\n";
    }
    ToolsPanel.imprime("LOS OCTALES DEL 1 AL 20 SON:\n"+cad+"\n");
}
```

Pidiendo en consola un número en este caso un 8 y dando como resultado la lista de octales del 1 al 20



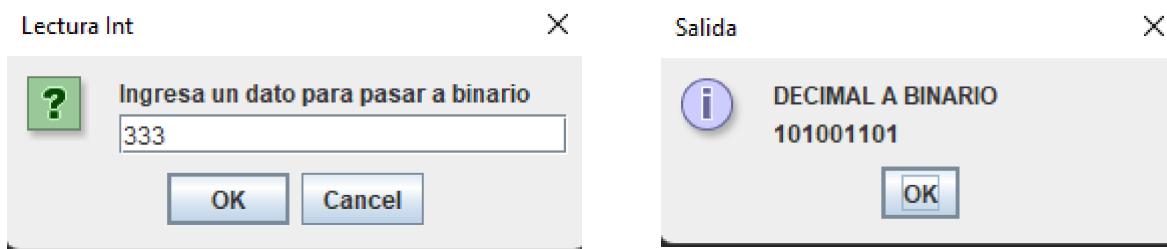
15- De decimal a binario en funciones iterativas

El sistema binario, popularmente conocido porque es el sistema que utilizan los ordenadores y el resto de dispositivos electrónicos, es un sistema de base 2. Eso significa que es un sistema que solo utiliza dos cifras para representar todos sus números y en el caso del código binario estas dos cifras son el 0 y el 1.

Para este ejemplo siendo muy corto tenemos solamente una variable de tipo entero `bin` y otra de tipo carácter `binario`, la cual es igual a la función que utilizaremos que es `Integer.toBinaryString()` la cual tendrá dentro `bin` que es nuestro número a convertir ,para luego imprimir lo que hay almacenado en `binario` que es nuestro número ya convertido

```
public static void decimalBin(int bin) {
    String binario = Integer.toBinaryString(bin);
    ToolsPanel.imprime("DECIMAL A BINARIO \n" +binario);
}
```

Pidiendo en consola solo un dato en este caso 333 e imprime el numero convertido en su valor binario



16-Ordena Burbuja

El algoritmo de la burbuja es un algoritmo de ordenamiento de arreglos popular, este dividirá el arreglo en 2 secciones o particiones, una ordenada y la otra desordenada, durante la ejecución la partición ordenada irá creciendo y la desordenada irá disminuyendo.

En este ejemplo se declara un arreglo de tipo entero llamado `datos[]`, y las variables de tipo entero `i`, `j` y `aux`, para luego entrar a el primer ciclo for en el cual mientras `i` sea 0, e `i` se mayor a lo que hay dentro de `datos[] -1`, incrementa en `i`, dentro de este for se concatena otro ciclo for el cual mientras `j` sea 0, e `j` se mayor a lo que hay dentro del `datos[]`, incrementa en `j` para que dentro de ese segundo for se concatena un if el cual si `datos[]` en la posición `[i]`es menor a `datos[]`en la posición `[j]`, `aux` sera igual a `datos[]` en la posición `[i]`, y `datos` en la posición `[i]`sea igual a `datos[]`en la posición `[j]`, `datos[]` en la posicion `[j]`,será igual a `aux`

```

public static void ordenaBurbuja(int datos[]) {
    int i,j,aux;
    for(i=0; i<datos.length-1;i++) {
        for(j=i+1; j< datos.length ; j++) {
            if (datos[i]>datos[j]) {
                aux=datos[i];
                datos[i]= datos[j];
                datos[j]= aux;
            }
        }
    }
}

```

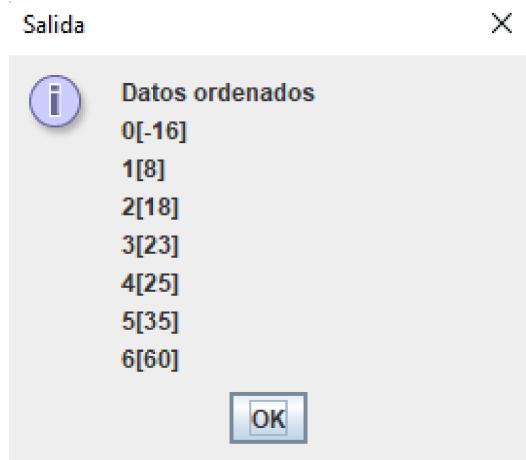
Dentro del menú se agregarán los datos del arreglo de forma desordenada

```

case "Ordena Burbuja":
    int dat[] = {35,8,-16,25,60,18,23};
    iterativas.ordenaBurbuja(dat);
    ToolsPanel.imprime("Datos ordenados \n"+iterativas.verArray(dat));
    break;

```

Dando como resultado en consola los datos en forma ordenada



17- Palíndromo en funciones iterativas

Un palíndromo es aquellas palabras que se leen igual de izquierda a derecha que de derecha a izquierda. Por ejemplo, la palabra alola si la invertimos sigue siendo

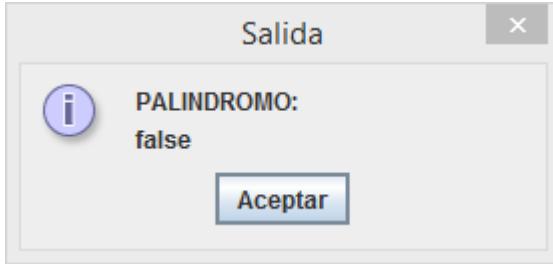
la misma palabra. La idea es que si la palabra original es igual a la palabra invertida, será un palíndromo. El resultado sería el mismo.

```
public static boolean espalindromo(String cadena){  
    cadena= cadena.replaceAll(" ", " ");  
    boolean valor=true;  
    int i,ind;  
    String cadena2="";  
    //quitamos los espacios  
    for (int x=0; x < cadena.length(); x++) {  
        if (cadena.charAt(x) != ' ')  
            cadena2 += cadena.charAt(x);  
    }  
    //volvemos a asignar variables  
    cadena=cadena2;  
    ind=cadena.length();  
    //comparamos cadenas  
    for (i= 0 ;i < (cadena.length()); i++){  
        if (cadena.substring(i, i+1).equals(cadena.substring(ind -1, ind)) == false ) {  
            valor=false;  
            break;  
        }  
        ind--;  
    }  
    return valor;  
}
```

En este método lo que queremos es mostrar el palíndromo de una frase o palabra, de que manera. Pues comparando la primer y última letra de la misma para comparar si es la misma, de tal forma de que no llegase a ser palíndromo nos arroja un falso. Aquí el ejecutable del método:



En caso de que la palabra no sea palíndromo como por ejemplo la palabra “perro” hará la misma acción anterior solo que esta vez en el segundo caso base al ver que la palabra así normal y de forma inversa no es la misma automáticamente retornara un false.



Aquí tenemos lo que es el menú iterativo, de esta forma es más fácil llamarlos desde la clase donde están. Así no tenemos necesidad de llamar a cada uno desde el main, es más compacto y fácil de manejar:

TEST MENU ITERATIVO

```
package Menus;

import javax.swing.JOptionPane;

import Metodos.FuncionesIterativas;

import ToolsPanel.ToolsPanel;

public class TestPruebalterativa {

    public static String listaDesplegable() {

        String valores[] = {"Funcion Iterativa", "Imprime
Array", "Fibonacci", "Tabla de multiplicar", "decimalAOctal", "Suma Amigos", "Numero
binario", "Potencia de un numero", "Ordena Burbuja", "Factorial del 1 al 15", "Octales
del 1 al 20", "Matrices", "Multiplicacion Rusa", "Funcion
Akermann", "Palindromo", "Salir"};

        String res = (String) JOptionPane.showInputDialog(null, "M E N U",
"Selecciona opcion:", JOptionPane.QUESTION_MESSAGE, null, valores, valores[0]);

        return (res);
    }
}
```

```
}

public static void menu3(){

    String sel="";
    do {
        sel=TestPruebalterativa.listaDesplegable();
        switch (sel) {
            case "Funcion Iterativa":
                FuncionesIterativas.usowhile(ToolsPanel.leerInt("Ingresa un numero"));

                break;
            case "Imprime Array":
                int a[]={1,2,3,7,8,9,12};

                FuncionesIterativas.imprimeArray(a);

                break;
            case "Fibonacci":
                FuncionesIterativas.fibonacci(ToolsPanel.leerInt("Ingresa un dato como tope"));

                break;
            case "Tabla de multiplicar":
                FuncionesIterativas.multiplyTable(ToolsPanel.leerInt("Ingresa un numero"));

                break;
            case "decimalAOctal":
```

```
        ToolsPanel.imprime("NUMERO
OCTAL:\n"+FuncionesIterativas.decimalAOctal(ToolsPanel.leerInt("Ingresa un dato
para pasar a octal")));
        break;
        case "Suma Amigos":
        ToolsPanel.imprime("NUMEROS AMIGOS\n
"+FuncionesIterativas.SumaDiviciones(ToolsPanel.leerInt("Ingresa un numero
amigo")));
        break;
        case "Numero binario":
        FuncionesIterativas.decimalBin(ToolsPanel.leerInt("Ingresa un dato para pasar a
binario"));
        break;
        case "Potencia de un numero":
        FuncionesIterativas.numPot(ToolsPanel.leerInt("Ingresa
un numero"), ToolsPanel.leerInt("elevado a"));
        break;
        case "Ordena Burbuja":
        int dat[]={35,8,-16,25,60,18,23};
        FuncionesIterativas.ordenaBurbuja(dat);
        ToolsPanel.imprime("Datos ordenados
\n"+FuncionesIterativas.verArray(dat));
        break;
        case "Factorial del 1 al 15":
```

```

        FuncionesIterativas.factorial(ToolsPanel.leerInt("Ingresa
un numero para iniciar"));

        break;

        case "Octales del 1 al 20":


FuncionesIterativas.listaOctal(ToolsPanel.leerInt("Ingresa un dato como tope"));

        break;

        case "Matrices":


int vec[][] = new int[4][4];

        FuncionesIterativas.leerMatriz(vec);

        FuncionesIterativas.verMatriz(vec);

        break;

        case "Multiplicacion Rusa":


FuncionesIterativas.multiplicacionRusa(ToolsPanel.leerInt("Ingresa un
dato"),ToolsPanel.leerInt("Ingresa otro dato"));

        break;

        case "Funcion Akermann":


ToolsPanel.imprime("FUNCION
AKERMANN\n"+FuncionesIterativas.akermann(ToolsPanel.leerInt("Ingresa un
dato"),ToolsPanel.leerInt("Ingresa otro dato")));

        break;

        case "Palindromo":


ToolsPanel.imprime("RESULTADO PALINDROMO\n"
+FuncionesIterativas.espalindromo(ToolsPanel.leerString("Ingresa una frase")));

```

```
        break;  
    }  
  
    }while(!sel.equalsIgnoreCase("Salir"));  
  
}  
  
public static void main(String[] args) {  
  
    // TODO Auto-generated method stub  
  
    menu3();}  
}
```

CONCLUSIÓN

Se puede decir que la recursividad es una técnica de programación bastante útil y muy interesante de estudiar. A través de los temas abordados y los ejemplos el futuro programador está en la capacidad de incluir esta técnica cuando se le presente un problema.

La asignación de memoria sea estática o dinámica, en realidad se tendrá que aplicar en cualquier programa al momento de su codificación, teniendo en cuenta que cada programador tiene su estilo de programar.

Conocer qué es la recursividad y cómo utilizarla es bastante útil para el programador en su quehacer para solucionar problemas por medio de algoritmos, eficientes, eficaces, de fácil mantenimiento y entendimiento.

En la cotidianidad existe un caso que permite interpretar a la recursividad y hasta dónde se puede llegar con ella: es el de las muñecas Matrushka.

BIBLIOGRAFÍA

- Cerinza, N. G. (n.d.). FAEDIS. Edu.co. Retrieved April 10, 2023, from http://virtual.umng.edu.co/distancia/ecosistema/odin/odin_desktop.php?path=Li4vb3Zhcy9pbmdlbmllcmllhX2luZm9ybWF0aWNhL2VzdHJ1Y3R1cmFfZGVfZGF0b3MvdW5pZGFkXzlv
- Macaray, J.-F., & Nicolas, C. (1996). Programacion java. Gestion 2000.
- Recursividad. (2003, September 11). Monografias.com. <https://www.monografias.com/trabajos14/recursividad/recursividad>
- Wikipedia contributors. (n.d.-a). Función de Ackermann. Wikipedia, The Free Encyclopedia. https://es.wikipedia.org/w/index.php?title=Funci%C3%B3n_de_Ackermann&oldid=142444318
- Wikipedia contributors. (n.d.-b). Ordenamiento de burbuja. Wikipedia, The Free Encyclopedia. https://es.wikipedia.org/w/index.php?title=Ordenamiento_de_burbuja&oldid=150091421
- Acosta, E. V. (2023, enero 30). Métodos arreglo de Java- Cómo imprimir arreglos en Java. freecodecamp.org. <https://www.freecodecamp.org/espanol/news/como-imprimir-arreglos-en-java/>
- de Roer, D. D. (2022, octubre 24). Palíndromo en Java. Disco Duro de Roer -. <https://www.discoduroderoer.es/palindromo-en-java/>
- Del Amo Blanco, I. (2014, diciembre 22). Factoriales. ¿Para qué los utilizamos? Smartick. <https://www.smartick.es/blog/matematicas/numeros-enteros/factoriales/>
- Multiplicación rusa — Programación. (s/f). Usm.cl. Recuperado el 10 de abril de 2023, de <http://progra.usm.cl/apunte/ejercicios/1/multiplicacion-rusa.html>

- ¿Qué es el sistema binario? (s/f). Codelearn.es. Recuperado el 10 de abril de 2023, de <https://codelearn.es/blog/que-es-el-sistema-binario/>
- ¿Qué es la secuencia de Fibonacci? (s/f). Wearedrew.co. Recuperado el 10 de abril de 2023, de <https://blog.wearedrew.co/concepts/que-es-la-secuencia-de-finobacci>
- Ruesgas, S. S. (2014, diciembre 21). Potencias: qué son y para qué sirven con vídeo tutorial. Smartick. <https://www.smartick.es/blog/matematicas/algebra/potencias-que-son-y-par-a-que-sirven/>
- Sentencias iterativas. (s/f). Ehu.es. Recuperado el 10 de abril de 2023, de <http://www.sc.ehu.es/sbweb/fisica3/basico/iterativo/iterativo.html>
- Wikipedia contributors. (s/f). Función de Ackermann. Wikipedia, The Free Encyclopedia. https://es.wikipedia.org/w/index.php?title=Funci%C3%B3n_de_Ackermann&oldid=142444318
- (S/f-a). Javautodidacta.es. Recuperado el 10 de abril de 2023, de <https://javautodidacta.es/bucles-do-while-y-while-en-java/>
- (S/f-b). Javautodidacta.es. Recuperado el 10 de abril de 2023, de <https://javautodidacta.es/bucles-do-while-y-while-en-java/#:~:text=%C2%BF%Qu%C3%A9%20son%20los%20bucles%20do%2Dwhile%20y%20while%20en%20Java%3F,-Reflejos%20de%20reflejos&text=Los%20bucles%20do%2Dwhile%20y%20while%20en%20Java%20te%20permiten,una%20condici%C3%B3n%20booleana%20de%20control.>