Un algoritmo en programación es un conjunto de instrucciones definidas, ordenadas y acotadas que se utilizan para resolver un problema o llevar a cabo una tarea específica. Básicamente, un algoritmo proporciona una serie de pasos lógicos y precisos que se deben seguir para alcanzar un resultado deseado. En el campo de la informática y la programación, los algoritmos son fundamentales, ya que permiten controlar procesos y realizar cálculos de manera eficiente. Son la base sobre la cual se desarrollan programas y software.

35. Escriba dos algoritmos en Java y esos mismos dos algoritmos en C para resolver el mismo problema. Por ejemplo, puede escribir un algoritmo recursivo y otro iterativo (con un bucle) para resolver el problema de la suma de los n primeros números naturales.

- Programa escrito en java.

```
public class SumaNaturales {

        public static int sumaIterativa(int n) {

        int suma = 0;

        for (int i = 1; i <= n; i++) {

        suma += i;

        }

        return suma;

        }

}
```

- Algoritmo escrito en C.

```
#include <stdio.h>


int sumaIterativa(int n) {

        int suma = 0;

        for (int i = 1; i <= n; i++) {

        suma += i;

        }

        return suma;

}

int main() {

        int n = 10;

        printf("La suma de los primeros %d números naturales es: %d\n", n,
sumaIterativa(n));

        return 0;
```

}

En términos de límite de cocientes defunciones, O(n) significa que una función f(n) es del orden de g(n) si y solo si el límite de f(n) / g(n) cuando n tiende a infinito es un número finito y no nulo.

La complejidad temporal de la fórmula e = ½ $gt2$ en función de t es constante, es decir, O(1) que no importa cuánto aumente t, el tiempo de cálculo no cambia.

34. ¿Qué es un algoritmo?

36. Defina O(n) en términos de un límite de cociente de funciones.

37. La fórmula para calcular el espacio recorrido por un móvil que se deja caer al vacío (suponiendo v0 = 0) es e = 12 gt2, donde g es la aceleración de la gravedad en la superficie de la tierra, y t el tiempo que está cayendo el móvil. ¿Cuál es la complejidad temporal de este cálculo en función de t ?.

38. Indique la complejidad temporal asintótica de los siguientes métodos:

```
public static String primero(ArrayList<String> lista)

return lista.get(0);}

public static String nEsimo(ArrayList<String> lista, int n){

return lista.get(n);

}
```

La función primero, su complejidad temporal es O(1) porque siempre devuelve el primer elemento de la lista, del tamaño de la lista.

La función nEnisimo, su complejidad temporal es O (1) porque accede directamente a la posición n de la lista, independientemente del tamaño de la lista.

39. Calcule la complejidad temporal de los algoritmos del ejercicio35.

El algoritmo escrito en java y el escrito en c del ejercicio 35 tienen una complejidad temporal de O(n) porque realizan una cantidad de operaciones proporcional al valor de n.

40. Resuelva cualquiera de los apartados del ejercicio 11 y calcule su complejidad temporal.

La complejidad temporal del algoritmo iterativo sumaNumeroNaturales es O(n), esto es porque la cantidad de operaciones realizadas por el algoritmo aumenta linealmente con respecto al valor de n, ya que se debe sumar cada número natural desde 0 hasta n. La complejidad temporal del algoritmo iterativo para calcular los primeros n números naturales es O(n).

41. Calcule la complejidad temporal y espacial de cualquiera de los algoritmos (recursivos) del ejercicio 2 (salvo los referentes a la serie de Fibonacci). Compare dichas complejidades con el algoritmo iterativo para resolver el mismo problema.

El problema de calcular la suma de los primeros n números naturales mediante la fórmula de la suma aritmética es un caso clásico que nos permite ilustrar las diferencias entre los enfoques recursivos e iterativos en términos de complejidad temporal y espacial.

Algoritmo Recursivo:

El enfoque recursivo para resolver este problema implica definir una función que se llame a sí misma de forma recursiva hasta alcanzar un caso base. En este caso, el caso base es cuando n=1, donde la función devuelve 1. La complejidad temporal de este algoritmo es O(n), en cada llamada recursiva se reduce n en una unidad hasta alcanzar el caso base. En cuanto a la complejidad espacial, también es O(n), ya que cada llamada recursiva crea un nuevo marco de pila.

Algoritmo Iterativo:

Por otro lado, de forma iterativa implica utilizar un bucle para calcular la suma de los primeros n números naturales mediante la fórmula de la suma aritmética. La complejidad temporal de este algoritmo también es O(1), ya que realiza un número constante de operaciones independientemente del valor de n. En términos de complejidad espacial, es O(1), ya que no se requiere espacio adicional en la pila de llamadas.

42. Sea un conjunto A con cardinalidad n, y sea l un algoritmo que ejecuta una instrucción para cada elemento del producto cartesiano de A × A. Calcule la complejidad temporal de l en función de n.

Para este problema hay que ejecutar una instrucción para cada par de elementos en el producto cartesiano AXA, donde A es un conjunto con cardinalidad n. Si A tiene n elementos, entonces el producto cartesiano AXA tendría n x n que es igual a n^2 pares de elementos.

La complejidad temporal de del algoritmo en función de las n es O(n^2), ya que el número de operaciones ejecutadas es proporcional al cuadrado del tamaño del conjunto de A. Esto porque se realiza una operación para cada elemento en el producto cartesiano A X A.

43. Calcule la complejidad temporal del siguiente método:

```
public static double sumaEltosMatriz(double matriz[][]){
double suma = 0;
for(int i = 0; i < matriz.length; i++){
        for(int j = 0; j < matriz[i].length; j++){
        suma+=matriz[i][j];
        }
}return suma;
}
```

El método sumaEltosMatriz recorre todos los elementos de una matriz una vez, por lo que su complejidad temporal es O(m×n), donde m es el número de filas y n es el número de columnas de la matriz.

44. Escriba un algoritmo que busque un número en un array de enteros. Calcule su complejidad temporal en el caso peor, en el caso mejor y en el caso promedio. Su cabecera será la siguiente:

public static boolean buscar(int e, int[] array).

La complejidad temporal en el peor de los casos de la búsqueda lineal en un array de enteros es $O(n)$, donde n es el tamaño del array.

45. Escriba un algoritmo recursivo para buscar un número en un array ordenado de enteros. Su

cabecera será la misma que la del ejercicio 43. Calcule su complejidad en el caso peor.

La complejidad temporal en el peor de los casos de un algoritmo recursivo de búsqueda binaria en un array ordenado es O(logn), donde n es el tamaño del array.

46. Calcule las complejidades temporal y espacial del algoritmo recursivo para calcular el elemento n-ésimo de la sucesión de Fibonacci.

```
private static int fibonacci(int n) {
if (n <= 1) {
        return n; }
else {
        return fibonacci(n - 1) + fibonacci(n - 2);
}
 }
```

La complejidad temporal de este algoritmo se puede aproximar a O(2^n), ya que en cada llamada recursiva se realizan dos llamadas más pequeñas, excepto en los casos base. La complejidad espacial es O(n) debido a la profundidad máxima de la pila de llamadas recursivas, que es igual a n.

47.Se tiene el siguiente método:

```
public static int sumaNPrimeros(int n) {
int suma = 0;
for (int i = 1; i <= n; i++) {
suma += i;
}
```

```
return suma;

}
```

Utilizando el profiler de Netbeans se han medido los tiempos de ejecución de diferentes llamadas al método (véase el cuadro 1). Explique los resultados.

El método calcula la suma de los primeros números hasta llegar a n. La complejidad de este método es lineal, ya que utiliza un for que realiza la suma n veces, por lo tanto la complejidad es O(n).

Los tiempos de ejecución indican que en cada nueva llamada al método se multiplica por 10 la cantidad de números a sumar y el tiempo que ocupa el algoritmo al resolverse es aproximadamente 10 veces más cada vez.

Osea que el aumento del tiempo es proporcional al aumento de n, entonces los tiempos observados son consistentes con la complejidad lineal del algoritmo.

48. Se tiene el siguiente método:

```
public static int sumaNMPrimeros(int n) {

int suma = 0;

for (int i = 1; i <= n; i++) {

for(int j = 1; j <= i; j++)

suma += j;

}

return suma;

}
```

Utilizando el profiler de Netbeans se han medido los tiempos de ejecución de diferentes llamadas al método (véase el cuadro 2). Explique los resultados.

El método calcula la suma haciendo un bucle anidado. La complejidad del método es cuadrática, es decir O(n^2).

Observando los tiempos de ejecución se puede notar que los tiempos aumentan significativamente, a medida que n crece. Esto es, que a mayor n, los tiempos aumentan exponencialmente debido a la complejidad cuadrática del algoritmo.

49.Explique la definición que se muestra a continuación

Sean dos funciones

T:N⟶N y f:N⟶N. Se dice que T(n) es de orden f(n), y se escribe T(n)∈O(f(n)), si y sólo si existen dos números naturales k y n0 tales que, para todo m, también natural, que cumpla m > n 0, entonces T(m)≤k· f(m).

La definición proporciona un marco para comparar dos funciones, donde T(n) es de orden f(n), denotado como T(n) ∈ O(f(n)), si y solo si existe una constante k y un valor n0 tales que para todo m mayor que n0, T(m) es menor o igual a k multiplicado por f(m).

50. Asumiendo la definición del ejercicio 48, se pide:

1. Encontrar k y n0 que muestran que la siguiente función, T:N⟶N, es de orden O(log2(n)).

T(n) = 3·log2(n) + 2.

2. ¿Si T(n)∈O(log2(n)), entonces T(n)∈O(n)? Justifique la respuesta.

3. ¿Si T(n)∈O(log3(n)), entonces T(n)∈O(log2(n))? Justifique la respuesta.

OBSERVACIÓN: en este ejercicio no es necesario que utilice la calculadora.

Para demostrar que T (n) = 3·log2(n) + 2 es de orden O(log2 (n)), observamos que T(n) es menor o igual a 5 * log2 (n) ≥ 2. Por lo cual podemos elegir k = 5 y n0 = 2 para demostrar que T(n) es de orden O(log2 (n)).

2. No, si T (N) ∈ O (log2(n)), no implica que T (n) ∈ O(n) porque O(n) porque O(log2(n)) indica un crecimiento logarítmico, mientras que O(n) indica un crecimiento lineal.

3. Si T(n) ∈ O (log2(n)), entonces T(n) ∈ O(log2(n))) porque los logaritmos de diferentes bases están relacionados por un factor constante.

51. Estudie de forma comparativa entre ellas el crecimiento de las siguientes funciones reales de variable real:

f0(x) = 1,

f2.f1(x) =x.

f2(x) =x^2.

f3(x) =log2(x), y

f4(x) = 2^x

1. f0(x) = 1 constante, independiente valor x función da el mismo resultado.

2. f1(x) = x lineal, a medida que x aumenta función aumenta misma proporción.

3. f2(x) = x2 cuadrática, a medida que x aumenta función aumenta más rápido.

4. f3(x) = log2(x) logarítmica, a medida que x aumenta función aumenta más lento.

5. f4(x) = 2x exponencial, a medida que x aumenta función aumenta exponencialmente.

52. Calcule la complejidad temporal asintótica del método f:

```
public static int f(int n) {
if (n == 0)
return 1;
else if (n < 0)
return -1;
else{
int m = 1/f(n/2) + f(n/2);
return sumaNPrimeros(m);
}
}
```

Para los cálculos, suponga que no hay en ningún momento desbordamiento de pila, y no tenga en cuenta los efectos sobre la ejecución que pueda suponer el desbordamiento de un entero.

Cuando n es igual a cero es caso base y operación tipo constante O(1). Cuando n es menor a cero es una operación de tipo constante 0(1).

El método f es una función recursiva que toma un entero n como entrada y devuelve un entero como salida. Analizando su estructura, vemos que tiene tres casos:

Si n es cero, devuelve 1.

Si n es negativo, devuelve -1.

Para otros valores de n, realiza dos llamadas recursivas y una operación adicional.

Dado que cada llamada recursiva reduce el problema a la mitad, la complejidad temporal es O(log n). En resumen, el método f tiene una complejidad temporal logarítmica.

53. La complejidad en el caso peor de la inserción en un arraylist es diferente si el array list está ordenado de si no lo está. ¿Es cierta esta afirmación? Justifique la respuesta.

Sí, es cierta. La inserción en un ArrayList ordenado tiene una complejidad temporal de O(n) porque, además de agregar el elemento al final de la lista, debe desplazar los elementos existentes para hacer espacio. Por otro lado, en un ArrayList no ordenado, la inserción tiene una complejidad temporal de O(1), ya que simplemente agrega el elemento al final de la lista sin necesidad de desplazamientos.

54. Suponga que una instrucción tarda en ejecutarse 10 ns, y que el tamaño de la entrada es n = 100, se pide calcular el tiempo requerido para los siguientes números de ejecuciones:

1.log(n), 24

2.n,

3.nlog(n),

4.n^2,

5.n^8 y

6.10^n

Realice los cálculos anteriores, pero ahora bajo los siguientes supuestos:

n= 100.000.

n= 100.000 y el tiempo de instrucción (o bloque de instrucciones) 1ms.

Se calcula el tiempo requerido para diferentes números de ejecuciones bajo diferentes supuestos.

55. Explique por qué el problema del ajedrez todavía no está resuelto.

El ajedrez es un juego extremadamente complejo, con una cantidad astronómica de posibles posiciones y movimientos. Aunque las computadoras modernas son muy potentes, aún tienen limitaciones en términos de capacidad de procesamiento y memoria. Además, el ajedrez implica estrategias y tácticas que van más allá del mero cálculo de movimientos, lo que hace que resolver el juego sea un desafío continuo.