Ejercicio 34. ¿Qué es un algoritmo?

Son instrucciones previamente definidas que se utilizan para realizar una tarea determinada.

Ejercicio 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.

El método recursivo:

En java:

```java
public static int suma(int n) {
    if (n == 0) {
        return 0;
    } else {
        return n + suma(n - 1);
    }
}
```

En C:

```c
#include <stdio.h>

    int suma(int n) {
        if (n == 0) {
            return 0;
        } else {
            return n + suma(n - 1);
        }
    }
```

El método iterativo:

En java:

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

En C:

```c
#include <stdio.h>

    int suma(int n) {
        int acumulado = 0;
        for (int i = 1; i <= n; i++) {
            acumulado += i;
        }
        return acumulado;
    }
```

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

$$\lim_{x\to\infty} \frac{x}{x+1} = 1$$

$$\lim_{x\to\infty} \frac{x+1}{x^2} = 0$$

Ejercicio 37. La fórmula para calcular el espacio recorrido por un móvil que se deja caer al vacío (suponiendo $v0 = 0$) es $e = 12gt2$, 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?

La complejidad es constante.

Ejercicio 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);
}
```

Del método primero la complejidad es constante de O(0). El otro método nEsimo, la complejidad es lineal de O(n).

Ejercicio 39. Calcule la complejidad temporal de los algoritmos del ejercicio 35.

La complejidad de ambos métodos es constante de O(0).

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

Apartado 2:

```java
public static int factorial(int n){
    int acumulado=1;
    for (int i=1;i<=n; i++){
        acumulado*=i;

    }
    return acumulado;
}
```

La complejidad es lineal de O(n).

Ejercicio 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.

```
public static int potencia(int n, int base){
    int acumulado=1;
    for (int i = 0; i<n; i++ ){
        acumulado*=base;
    }

    return acumulado;
}
```

La complejidad de este método iterativo es lineal de O(n).

```
public static int potencia(int n, int base) {
    if (n == 0) {
        return 1;
    } else {
        return base * potencia(n - 1, base);
    }
}
```

La complejidad de este método recursivo es lineal de O(n).

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

```
public class ProductoCartesiano {

    public static List<List<Integer>> productoCartesiano(List<Integer>
conjuntoA, List<Integer> conjuntoB) {
        List<List<Integer>> resultado = new ArrayList<>();

        for (Integer elementoA : conjuntoA) {
            for (Integer elementoB : conjuntoB) {
                List<Integer> par = new ArrayList<>();
                par.add(elementoA);
                par.add(elementoB);
                resultado.add(par);
            }
        }
    }
}
```

Este método que calcula el producto cartesiano es de complejidad cuadrática de O (elemento^2).

Ejercicio 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;
}
```

t()= matriz.length + matriz[i].lenght= O(matriz.length) es lineal.

Ejercicio 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)

```java
public static boolean buscar(int e, int[] array) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == e) {
            return true;
        }
    }
    return false;
}
```

La complejidad temporal de este método es de O(n), donde n en el mejor de los casos puede tomar el valor de 1, en el peor tomará el valor de la última posición del array y en el promedio será de O(n/2).

Ejercicio 45. Escriba un algoritmo recursivo para buscar un número en un array ordenado de enteros. Su cabecerá será la misma que la del ejercicio 43. Calcule su complejidad en el caso peor.

```java
static boolean busquedaBinaria(int[] arr, int objetivo) {
    int izquierda = 0;
    int derecha = arr.length - 1;
    int iteraciones = 0;

    while (izquierda <= derecha) {
        iteraciones++;
        int elemento = derecha - izquierda + 1;
        int medio = (izquierda + derecha) / 2;

        if (arr[medio] < objetivo) {
            izquierda = medio + 1;
        } else if (arr[medio] > objetivo) {
            derecha = medio - 1;
        } else {
            System.out.println("Número de iteraciones: " +
iteraciones);
            return true;
        }
    }

    System.out.println("Número de iteraciones: " + iteraciones);
    return false;
}
```

La complejidad de este método es O($\log_2$(n)), en el mejor la n toma el valor de 2 para que sea O(1) y en el peor es cuando n toma el valor de la última posición del array.

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

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

}
```

El método fibonachi tiene una complejidad constante O(n).

Ejercicio 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.

Cuadro 1: Tiempos de ejecución del método del ejercicio 46

Llamada	Tiempo de ejecución
sumaNPrimeros(100)	0,003 ms
sumaNPrimeros(1000)	0,013 ms
sumaNPrimeros(10000)	0,131 ms
sumaNPrimeros(100000)	1,20 ms
sumaNPrimeros(1000000)	3,27 ms
sumaNPrimeros(10000000)	6,19 ms
sumaNPrimeros(100000000)	36 ms
sumaNPrimeros(1000000000)	325 ms

Este método es de complejidad lineal de O(n). Esto quiere decir que cuanto más grande sea n más va a tardar en ejecutarse.

Ejercicio 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.

Cuadro 2: Tiempos de ejecución del método del ejercicio 47

Llamada	Tiempo de ejecución
sumaNPrimeros(100)	0,085 ms
sumaNPrimeros(1000)	2,16 ms
sumaNPrimeros(10000)	18,1 ms
sumaNPrimeros(100000)	1531 ms

Este método es de complejidad cuadrática $O(n^2)$. Esto quiere decir que cuanto más grande sea n más va a tardar en ejecutarse.

Ejercicio 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 $T(n) \in O(f(n))$, si y sólo si existen dos números naturales ky $n0$ tales que, para todo m, también natural, que cumpla $m > n0$, entonces $T(m) \leq k \cdot f(m)$.

Lo que quiere decir es que hay un T(n) comprendido en f(n), mientras k y n0 son naturales y m tiene que ser mayor que n0. Si se cumple esto aseguramos que T(m) va a ser menor o igual a k por f(m).

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

1. Encontrar ky $n0$ que muestren que la siguiente función, T: N →N, es de orden $O(log2(n))$.

$T(n) = 3 \cdot log2(n) + 2$▷

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

$T(n) \in O(log2(n))$, porque el límite cuando tiende a infinito de $3 \cdot log2(n) + 2/log2(n) = 3$ y si escogemos que $c = 5$ y un $n_0 = 2$, nos damos cuenta de que $3 < 5$.

$T(n) \in O(n)$, porque el límite cuando tiende a infinito de $3 \cdot log2(n) + 2/n = 1/n$ y si escogemos que $c = 1$ y un $n_0 = 1$, nos damos cuenta de que $1/n < c$.

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

$T(n) \in O(log3(n))$, porque el límite cuando tiende a infinito de $3 \cdot log2(n) + 2/log3(n) = 1/1.58$ y si escogemos que $c = 5$ y un $n_0 = 3$, nos damos cuenta de que $1/1.58 < 5$.

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

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

1. $f0(x) = 1$,
2. $f1(x) = x$,
3. $f2(x) = x2$,
4. $f3(x) = log2(x)$, y
5. $f4(x) = 2x$.

Ejercicio 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.

El método tiene una complejidad logarítmica O(log n).

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

Ejercicio 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. $log(n)$, = 30

2. n, = 110
3. $nlog(n)$, = 210
4. $n2$, = 10010
5. $n8$ = 10000000010 y
6. 10n = 10^{100} + 10 .

Realice los cálculos anteriores, pero ahora bajo los siguientes supuestos:
1. $n= 100▷000$. Resultados: 5, 100000, 10000000, 10000000000000, 10^{100000}.
2. $n= 100▷000$ y el tiempo de instrucción (o bloque de instrucciones) 1 ms. Resultados: 6, 100001, 10000001, 10000000000001, 10^{100000} + 1.

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

Por la gran cantidad de posibilidades en los movimientos que puede realizar el jugador.