## MEDIR EL TIEMPO DE EJECUCIÓN DE UN PROGRAMA

Una forma muy sencilla para medir el tiempo de ejecución que lleva hacer una tarea(un bloque de instrucciones, un método etc.) es anotar el tiempo "la hora" al inicio y final de dicha tarea.

En java, disponemos del método System.currentTimeMillis(), el cual nos devuelve un long que representa el tiempo actual en milisegundos, por lo tanto realizando una resta entre el tiempo final y el inicial, podemos obtener aproximadamente el tiempo en milisegundos que llevó la tarea que estamos midiendo.

El esquema general para medir será:

```
long tiempo_inicio, tiempo_fin;
tiempo_inicio = System.currentTimeMillis();
TareaQueQuieroMedir();
tiempo_fin = System.currentTimeMillis();
System.out.println("la tarea llevó : "+ ( tiempo_fin - tiempo_inicio ) +" millisegundos");
```

Ejemplo: medimos los milisegundos que consume un bucle

```
class Unidad3{
  public static void main(String[] args) {

    long tiempo_inicio = System.currentTimeMillis();

    long total = 0;
    for (int i = 0; i < 10_000_000; i++) {
      total += i;
    }

    long tiempo_fin= System.currentTimeMillis();
    long tiempo_medido = tiempo_fin - tiempo_inicio;
    System.out.println(tiempo_medido);
  }
}
```

## EVITAR LAS CONCATENACIONES DE STRING EN BUCLES DE MUCHAS ITERACIONES.

Recuerda el ejercicio de "que lado de la calle" de acepta el reto. Leía números de portales y si dicho número era par imprimía DERECHA o bien IZQUIERDA en caso contrario. Para simular un bucle con muchas iteraciones adaptamos dicho ejemplo de forma que los números de portal son la i de un for que dé por ejemplo un 100_000 vueltas (o más si tu ordenador es una bala)

Medimos el siguiente código

```
public class Unidad3 {

   public static void main(String[] args) {
      long tiempo_inicio = System.currentTimeMillis();

      for (int i = 0; i < 100_000; i++) {
         if (i % 2 == 0) {
            System.out.println("DERECHA");
         } else {
            System.out.println("IZQUIERDA");
         }
      }
      long tiempo_fin = System.currentTimeMillis();
      long tiempo_medido = tiempo_fin - tiempo_inicio;
      System.out.println(tiempo_medido);

   }

}
```

Recuerda las observaciones que hicimos sobre la gestión de memoria de los literales String en el boletín de Strings, y en consecuencia, observa que en los 2 ejemplos anteriores no se generan Objetos Strings nuevos en cada vuelta del bucle. Los strings que se manejan en el bucle son los literales "DERECHA" e "IZQUIERDA" y son siempre los mismos en toda la ejecución del bucle, es decir, aunque el bucle tenga 1 millón de vueltas sólo se crearon 2 Strings.

*Intento fallido de evitar llamadas a println() para mejorar la eficiencia*
Invocar a un método implica crear una entrada en la pila y "lleva tiempo" crear entradas en la pila así que si podemos evitarlas podríamos mejorar la eficiencia. Para ello escribo el siguiente código que evita invocar 100_000 veces a println()

```
public class Unidad3 {

   public static void main(String[] args) {
      long tiempo_inicio = System.currentTimeMillis();
      String resultado="";
      for (int i = 0; i < 100_000; i++) {
         String lado;
         resultado=resultado+(i % 2 == 0?"DERECHA\n":"IZQUIERDA\n");
      }
      System.out.println(resultado);
      long tiempo_fin = System.currentTimeMillis();
      long tiempo_medido = tiempo_fin - tiempo_inicio;
      System.out.println(tiempo_medido);

   }

}
```

¡Que desastre!: evito 100_000 println() pero genero 100_000 Strings "resultado" utilizando concatenación con +. Recuerda que los literales String "DERECHA" e "IZQUIERDA" se crean una única vez y por tanto estos no generan problema, el problema es que hay una concatenación de strings y como ya sabemos por ser lo Strings inmutables esto implica la creación de un nuevo String.

La siguiente solución con StringBuilder evita el desastre anterior porque no se crean objetos dentro del bucle. Mejora mucho todo a pesar de que ahora hago la llamada en bucle a append().

```
public class Unidad3 {

   public static void main(String[] args) {
      long tiempo_inicio = System.currentTimeMillis();
      StringBuilder resultado=new StringBuilder("");
      for (int i = 0; i < 100_000; i++) {
         String lado=i % 2 == 0?"DERECHA\n":"IZQUIERDA\n";
         resultado.append(lado);
      }
      System.out.println(resultado);
      long tiempo_fin = System.currentTimeMillis();
      long tiempo_medido = tiempo_fin - tiempo_inicio;
      System.out.println(tiempo_medido);

   }

}
```

De todo lo anterior concluimos que en bucles de muchas iteraciones en la medida de lo posible debemos evitar la creación de objetos, y un caso concreto importante es ***evitar la concatenación de Strings dentro de bucles de muchas iteraciones ya que***

***generan muchos objetos intermedios que serán "huérfanos" que disparan la actividad del recolector de basura*.** Por tanto, cuando tengas esta necesidad, es mejor usar StringBuider. Este es un caso concreto en que StringBuilder es más eficiente que String, pero en muchas otras situaciones normalmente la clase String se considera más eficiente y sobre todo, más segura, por ser inmutable. La seguridad de lo inmutable ,que evita errores difíciles de localizar en las aplicaciones, está muy valorada en la programación actual.

**OPTIMIZAR EL ALGORITMO QUE SE ESCRIBE EN EL INTERIOR DEL BUCLE SUELE SER LO MÁS IMPORTANTE PARA MEJORAR EL RENDIMIENTO.**

**Ejemplo. Calcular números primos**

Escribir un programa que calcule los números primos comprendidos entre 2 y 200000.

Recordamos que un número es primo si sólo es divisible por sí mismo y por la unidad. El 1 no se considera un número primo por eso para simplificar buscamos desde 2. En nuestro código intentamos encontrar un número que divide al número que estamos inspeccionando para ver si es primo o no.
Una primera versión podría ser:

```
class Unidad3{
    public static void main(String[] args) {
        for(int i=2;i<200_000;i++){
            boolean esPrimo=true;
            for(int j=2;j<i;j++){
               if(i%j==0){
                   esPrimo=false;
               }
            }
            if(esPrimo){
                System.out.print(i+" ");
            }
        }
    }
}
```

Medimos el tiempo que tarda en ejecutarse el bucle anterior. Si tienes un ordenador muy potente puedes aumentar el intervalo de búsqueda para que la respuesta no sea sólo de segundos y observar mejorar las diferencias entre soluciones.

```
class Unidad3{
    public static void main(String[] args) {
    long tiempo_inicio = System.currentTimeMillis();

        for(int i=2;i<200_000;i++){
            boolean esPrimo=true;
            for(int j=2;j<i;j++){
               if(i%j==0){
                   esPrimo=false;
               }
            }
            if(esPrimo){
                System.out.print(i+" ");
            }
        }
       long tiempo_fin= System.currentTimeMillis();
       long tiempo_medido = tiempo_fin - tiempo_inicio;
       double segundos=tiempo_medido/1000d;//pasar a segundos
       System.out.println("\n\ntiempo de ejecución: "+ segundos + " segundos");
```

```
   }
}
```

Anota el tiempo para ver a continuación mejoras

**Mejorar el rendimiento del programa teniendo en cuenta que un número primo siempre es impar así que evito comprobar los números pares.**
Para que el programa sea eficiente, evitaremos comprobar combinaciones que a priori ya sabemos que no son útiles para buscar números primos, para ello tenemos cuenta que los números primos tienen la siguiente característica: un número primo es solamente divisible por sí mismo y por la unidad, por tanto, un número primo no puede ser par excepto el 2.

Con esto reduzco la búsqueda a la mitad de valores así que más o menos puedo reducir la mitad el tiempo

El primo "2" que es una excepción se imprime a parte para que quede el primer for más limpio

```
class Unidad3{
   public static void main(String[] args) {
   long tiempo_inicio = System.currentTimeMillis();
      System.out.print("2 ");
      for(int i=3;i<200_000;i+=2){
         boolean esPrimo=true;
         for(int j=3;j<i;j++){
            if(i%j==0){
               esPrimo=false;
            }
         }
         if(esPrimo){
            System.out.print(i+" ");
         }
      }
     long tiempo_fin= System.currentTimeMillis();
     long tiempo_medido = tiempo_fin - tiempo_inicio;
     double segundos=tiempo_medido/1000d;//pasar a segundos
     System.out.println("\n\ntiempo de ejecución: "+ segundos + " segundos");


   }
}
```

**Añadir otra mejora usando break**
Una vez que descubro que un número no es primo porque tiene un divisor, puedo salir del bucle interno ya que me da igual que tenga o no más divisores.
¡Añadimos un break al if que determina que un número no es primo!

```
class Unidad3{
   public static void main(String[] args) {
   long tiempo_inicio = System.currentTimeMillis();
      System.out.print("2 ");
      for(int i=3;i<200_000;i+=2){
         boolean esPrimo=true;
         for(int j=3;j<i;j++){
            if(i%j==0){
               esPrimo=false;
               break;
            }
         }
```

```
         if(esPrimo){
            System.out.print(i+" ");
         }
       }
      long tiempo_fin= System.currentTimeMillis();
      long tiempo_medido = tiempo_fin - tiempo_inicio;
      double segundos=tiempo_medido/1000d;//pasar a segundos
      System.out.println("\n\ntiempo de ejecución: "+ segundos + " segundos");


   }
}
```

¡pedazo mejora! ¡y que sencilla!

**Y otra mejora más**

Como en los códigos anteriores sólo busco primos entre los impares, es una pérdida de tiempo dividir un número impar entre uno par, por tanto dividimos sólo con *j* impares ¡es cambiar sólo el incremento del j!

```
class Unidad3{
   public static void main(String[] args) {
   long tiempo_inicio = System.currentTimeMillis();
       System.out.print("2 ");
       for(int i=3;i<200_000;i+=2){
          boolean esPrimo=true;
          for(int j=3;j<i;j+=2){
             if(i%j==0){
                esPrimo=false;
                break;
             }
          }
          if(esPrimo){
              System.out.print(i+" ");
          }
       }
      long tiempo_fin= System.currentTimeMillis();
      long tiempo_medido = tiempo_fin - tiempo_inicio;
      double segundos=tiempo_medido/1000d;//pasar a segundos
      System.out.println("\n\ntiempo de ejecución: "+ segundos + " segundos");


   }
}
```

**Última mejora buscando sólo hasta √ n.**

Esta mejora nos recuerda lo importante que es entender el problema para poder resolverlo lo mejor posible. Esta mejora sería más natural de un matemático con algunos conocimientos de programación, que de un programador con pocos conocimientos de matemáticas y que no intenta mejorar sus conocimientos acerca de los números primos. Recuerda que un número si no es primo se llama compuesto porque se puede expresar como una multiplicación de factores. Si un número *n* es compuesto se sabe(nos lo creemos aunque no entendamos ahora porqué) que alguno de sus factores (sin contar el 1) debe ser menor o igual que √ n. Por lo tanto, el número de divisiones a realizar es mucho menor. Sólo hay que buscar hasta √ n.

Es decir en lugar de buscar divisores hasta que j<i puedo buscar simplemente hasta que j<Math.sqrt(i)

```
class Unidad3{
    public static void main(String[] args) {
    long tiempo_inicio = System.currentTimeMillis();
        System.out.print("2 ");
        for(int i=3;i<200_000;i+=2){
            boolean esPrimo=true;
            for(int j=3;j<Math.sqrt(i);j+=2){
               if(i%j==0){
                   esPrimo=false;
                   break;
               }
            }
            if(esPrimo){
                System.out.print(i+" ");
            }
        }
        long tiempo_fin= System.currentTimeMillis();
        long tiempo_medido = tiempo_fin - tiempo_inicio;
        double segundos=tiempo_medido/1000d;//pasar a segundos
        System.out.println("\n\ntiempo de ejecución: "+ segundos + " segundos");


    }
}
```

**AL TRABAJAR CON BUCLES, OJO AL HACER OPERACIONES ARITMÉTICAS, SE PUEDEN DISPARAR LOS RESULTADOS**

**Ejercicio U3_B8_E1**: Si experimentas con el siguiente código a partir del cálculo de fact(32) empiezan a pasar cosas raras. A partir de 34 siempre vale 0 ¿Cómo es posible?. Intenta dar una explicación a esto y soluciónalo usando BigInteger.

```
import java.util.Scanner;
public class Unidad3{
   public static void main(String[] args){
       Scanner teclado= new Scanner(System.in);
       System.out.println("Teclea número entero para calcular factorial:");
       int n=teclado.nextInt();
       int factorial=1;
       while(n>0){
           factorial*=n;
           n--;
       }
       System.out.println("Su factorial es: "+factorial);
   }
}
```

```
Teclea número entero para calcular factorial:
32
Su factorial es: -2147483648
Teclea número entero para calcular factorial:
33
Su factorial es: -2147483648
Teclea número entero para calcular factorial:
34
Su factorial es: 0
Teclea número entero para calcular factorial:
35
Su factorial es: 0
```

## EL PROBLEMA DE RENDIMIENTO DE LA RECURSIVIDAD

La recursividad debido a la utilización intensa de la pila de llamadas puede presentar dos tipos de problemas de eficiencia:

- Lentitud: produce programas más lentos que su equivalente con bucles
- Desbordamiento de pila

Ejemplo: con el siguiente código, ya estudiado, comprueba que para valores no muy altos de multiplicación se produce desbordamiento de pila (por ejemplo para 100000 en el ejemplo, en mi ordenador ya desbordó). multiplicar 2*100000 no es un gran cálculo, el problema es que intentamos crear 100000 entradas en la pila ...

```
class Unidad3 {
    public static void main(String[] args) {
        Multiplicacion m = new Multiplicacion();
        System.out.println(m.multiplicar(2,100000));
    }
}


class Multiplicacion{
  int multiplicar(int a, int b){
      if(b==1)
        return a;
      else
        return a + multiplicar(a,b-1);
  }
}
```

Si hubieramos hecho la multiplicación al revés

m.multiplicar(100000,2)

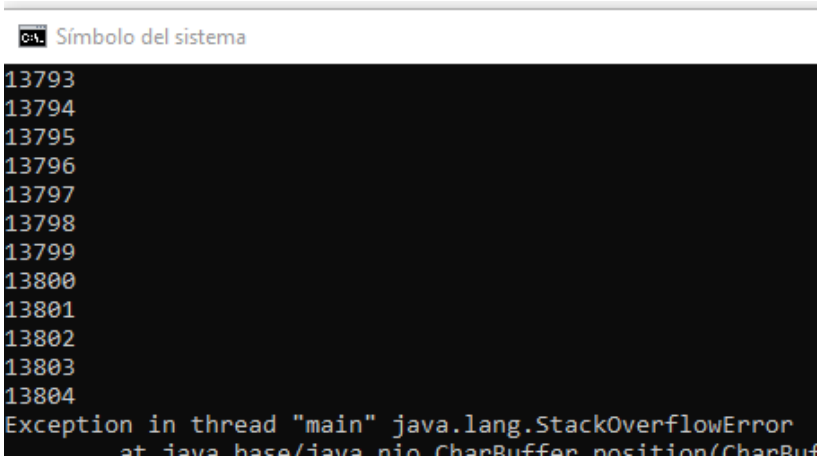
No habría desbordamiento ya que la generación de entradas en la pila se hace en función del valor de b

**Ejercicio U3_B8_E2**:Si conseguiste calcular el factorial basándote en bucles con biginteger, comprobarás que puedes calcular sin problemas por ejemplo el 100000!. Ahora desarrolla una versión recursiva para factorial pero que trabaje con BigInteger, pero observa que por ejemplo para 100000! se produce desbordamiento de pila

¿Cuántas llamadas puedo apilar?. Depende del tamaño que la JVM asigne a la pila. Esto dependerá de la versión de JDK. También dependerá del tamaño que ocupa la información asociada a cada llamada. Podemos hacernos una idea ejecutando el siguiente código que llama indefinidamente recursivamente a un método

```
public class Unidad3 {

    private static long cuentaLLamadas=0L;

    public static void main(String[] args){
        miMetodo();
    }

    private static void miMetodo(){
        System.out.println(++cuentaLLamadas);
        miMetodo();
    }

}
```

Según hardware, sistema operativo y procesos en ejecución tendremos un valor u otro




## OTROS EJEMPLO CON FACTORIALES: EL ALGORITMO OTRA VEZ LA GRAN CLAVE DE MEJORA DE EFICIENCIA

Dentro del concepto de factorial están contenidas multitud de relaciones matemáticas que se pueden aplicar para resolver elegantemente un problema como ocurre en el siguiente ejemplo de *acepta el reto*.

No hay que calcular el factorial, pero curiosamente hay que entender muy bien que es el factorial y saber relacionarlo con otros conceptos. Necesitaremos ayuda de páginas web de mates porque en principio no tendremos ni flores del increíble mundo del factorial

### Divisores del factorial (acepta el reto)

https://www.aceptaelreto.com/problem/statement.php?id=126

Es un ejemplo difícil de solucionar. De este ejemplo aprendemos que una vez más la clave es el algoritmo, no la fuerza bruta, al menos en este caso.

**fuerza bruta inútil:** El primer instinto puede ser calcular el factorial del número n y comprobar si es divisible por p

```
import java.util.Scanner;

public class Unidad3 {
    static int fac(int n){
        int f=1;
        while(n>0){
            f=f*n;
            n--;
        }
        return f;
    }
    public static void main(String[] args){
        Scanner sc= new Scanner(System.in);
        int p=sc.nextInt();
        int n=sc.nextInt();
        while(!( p<0 &&n<0)){
            if(fac(n)%p==0){
                System.out.println("YES");
            }else{
                System.out.println("NO");
            }
```

```
            p=sc.nextInt();
            n=sc.nextInt();
        }
    }

}
```

Esto funciona para los casos de prueba ejemplo, pero si lo envíamos a *acepta el reto* nos da TLE.
Podría ser por dos problemas relacionados con que recibamos n grande:

- n es un número menor que $2^{31}$. , es decir un int java. Pero ya sabemos que al calcular factoriales necesitaríamos usar BigInteger por que enseguida produce números gigantes fuera de int y long. Pero en este caso, para adelantar trabajo ya os digo que esto no es el problema
- de nuevo, si n es muy grande hay otro problema, la solución simple anterior genera un bucle con muchas iteraciones que consumen mucho tiempo. **Este va a ser el problema.** Para este problema el Juez estipula un tiempo de ejecución bajo para que use un algoritmo eficiente. Si el algoritmo no es eficiente me excedo del tiempo permitido y me responde con TLE.

El juez no quiere que se use un algoritmo ineficiente que se basa en un pesado cálculo del factorial. **No quiere que calculemos el factorial de n.**

**Algoritmo correcto:**

**Si p es primo, entonces se que si p<=n divide a n!**

**mini explicación del algoritmo correcto en fichero aparte:**
Puedes saltarte esta explicación y simplemente usar la frase anterior para resolver el problema. Es algo bastante técnico y concreto y no es nuestro objetivo pararnos tanto

La solución (que ahora da wrong answer) quedaría

```
import java.util.Scanner;
public class DivisorFactorial{
   public static void main(String[] args){
      Scanner sc = new Scanner(System.in);
      int n;
          int p;

      p = sc.nextInt();
      n = sc.nextInt();
      //no se acepta porque no contempla 0!
     while(!( p<0 &&n<0)){

         if (p<=n){
            System.out.println("YES");
         }else{
            System.out.println("NO");
         }
         p = sc.nextInt();
         n = sc.nextInt();
      }
```

```
    }
}
```

Si finalmente incorporo el caso especial de  0! esto se acepta:

```
import java.util.Scanner;

public class Unidad3 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n;
        int p;

        p = sc.nextInt();
        n = sc.nextInt();

        while (!(p < 0 && n < 0)) {

            if (n == 0 && p == 1) {// fact(0)==1//único p que genera YES aunque p>n
                System.out.println("YES");

            } else if (p <= n) {
                System.out.println("YES");
            } else {
                System.out.println("NO");
            }
            p = sc.nextInt();
            n = sc.nextInt();
        }

    }

}
```