INFORMÁTICA (30717)

Práctica 5: Resolución de problemas en Java (II)

Grado en Estudios en Arquitectura

Universidad de Zaragoza

Material de prácticas del curso 2025–2026

Universidad Zaragoza

Tabla de contenidos

1 Objetivos de la práctica 3

2 Criba de Eratóstenes 3

3 Vectores en Java 3

4 Ejercicios 3
4.1 Tiempos de ejecución 4
4.2 Medición de tiempo de ejecución 4

5 Solución 1: implementación básica 5

6 Solución 2: optimización con la raíz cuadrada 6

7 Solución 3: vector de booleanos 6

8 Solución 4: solo números impares 7

9 Entrega de la solución 8

1. Objetivos de la práctica

Los objetivos de la quinta práctica de la asignatura son los siguientes:

- Desarrollar algoritmos en Java.
- Ejecutar algoritmos en el entorno *Eclipse*.

2. Criba de Eratóstenes

La **criba de Eratóstenes** es un algoritmo eficiente para **encontrar todos los números primos menores o iguales a un número dado**. El algoritmo funciona de la siguiente manera:

1. Se guardan en un vector todos los números desde 2 hasta el número dado.
2. Se comienza con el primer número primo, 2, y se descartan todos los múltiplos de ese número.
3. Se repite el proceso con el siguiente número primo en el vector hasta que se hayan procesado todos los números.

Por ejemplo, para encontrar los números primos menores o iguales a 30, el algoritmo eliminaría los múltiplos de 2, luego los múltiplos de 3, y así sucesivamente, dejando solo los números primos $2, 3, 5, 7, 11, 13, 17, 19, 23, 29$.

3. Vectores en Java

A modo de recordatorio, un **vector en Java** es una colección de elementos del mismo tipo. Se puede declarar y utilizar un vector de números enteros de la siguiente manera:

```
int[] vector = new int[10];

vector[0] = 1;
vector[1] = 2;

int valor = vector[0];

for (int i = 0; i < vector.length; i++) {
    vector[i] = i + 1;
}
```

4. Ejercicios

La criba de Eratóstenes se puede implementar de varias maneras, dependiendo de cómo se manejen los datos y se optimice el proceso. A continuación se presentan cuatro posibles soluciones. Cada solución se deberá implementar en una clase separada, de nombre `Solucion1.java`, `Solucion2.java`, `Solucion3.java` y `Solucion4.java`, respectivamente.

El objetivo es modificar cada una de las implementaciones para **contar el número de tachones** que realiza cada una de las implementaciones de la criba, y **medir su tiempo de ejecución**.

4.1. Tiempos de ejecución

Anota el tiempo de ejecución en Microsoft Excel para cada una de las implementaciones, y cuatro valores de n diferentes: 10^4, 10^5, 10^6 y 10^7. Los valores de n se representarán en el eje X de un gráfico, y el tiempo de ejecución en el eje Y.

n	Solución 1	Solución 2	Solución 3	Solución 4
10 000	21	21	19	18
100 000	52	55	52	43
1 000 000	215	209	184	220
10 000 000	1286	1106	1060	1012

Cuadro 1: Tabla de tiempos de ejecución en milisegundos de las diferentes implementaciones de la criba de Eratóstenes.

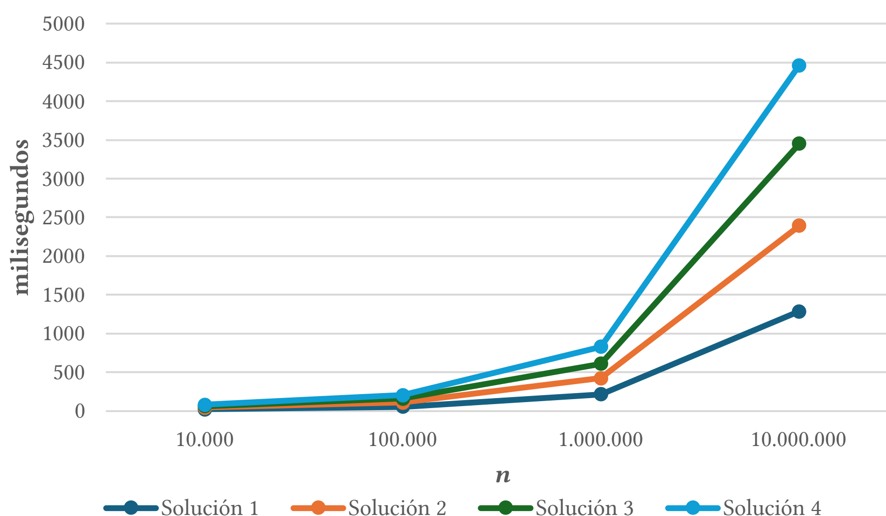

Figura 1: Comparación de tiempos de ejecución de las cuatro implementaciones.

Posible falta de memoria

Para valores de n muy grandes, es posible que el programa no tenga suficiente memoria para ejecutar la criba de Eratóstenes, especialmente en las implementaciones que utilizan un vector de enteros. En ese caso, puedes aumentar la memoria asignada a Java desde `Run > Run Configurations...`, pestaña `Arguments`, añadiendo `-Xmx4g` en `VM arguments`.

4.2. Medición de tiempo de ejecución

Para medir el tiempo de ejecución de cada implementación, puedes utilizar la clase `System` de Java para obtener el tiempo antes y después de ejecutar el algoritmo:

```
long startTime = System.currentTimeMillis();

// Introduce la implementacion de la criba de Eratostenes aqui

long endTime = System.currentTimeMillis();
long duration = endTime - startTime;
```

```
System.out.println("Tiempo de ejecucion: " + duration + " milisegundos");
```

5. Solución 1: implementación básica

En esta implementación se utiliza un **vector de números enteros para representar los números desde 0 hasta** n, donde cada posición indica si el número correspondiente es primo o no. Un número será primo si su posición en el vector no es cero, sino igual a su propio valor.

$$numero[i] = \begin{cases} i & \text{si } i \text{ es primo} \\ 0 & \text{si } i \text{ no es primo} \end{cases}$$

```
Scanner sc = new Scanner(System.in);
System.out.print("Introduce un numero entero: ");
int n = sc.nextInt();

int[] numero = new int[n + 1];
for (int i = 0; i <= n; i++) {
    numero[i] = i;
}

for (int i = 2; i <= n; i++) {
    if (numero[i] != 0) {
        int j = 2;

        while (i * j <= n) {
            numero[i * j] = 0;
            j = j + 1;
        }
    }
}

int contadorPrimos = 0;
for (int i = 2; i <= n; i++) {
    if (numero[i] != 0) {
        contadorPrimos++;
        System.out.println(i + " es primo");
    }
}

System.out.println();
System.out.println("Total: " + contadorPrimos + " numeros primos");
```

Tamaño del vector

Cada entero ocupa 4 bytes en Java, por lo que el vector utilizado para la criba ocupará $4(n+1)$ bytes. Por ejemplo, si $n = 100$, el vector ocupará $4(100+1) = 404$ bytes.

6. Solución 2: optimización con la raíz cuadrada

En la solución previa se itera hasta n, lo cual es ineficiente. En esta solución se optimiza el proceso iterando solo hasta la raíz cuadrada de n.

Optimización con la raíz cuadrada

Si un número n no es primo, uno de sus factores d cumple $1 \leq d \leq \sqrt{n}$. Si no tiene factores en ese rango, entonces es primo. Por tanto, basta con probar divisores hasta $\sqrt{n}$.

```
Scanner sc = new Scanner(System.in);
System.out.print("Introduce un numero entero: ");
int n = sc.nextInt();

int[] numero = new int[n + 1];
for (int i = 0; i <= n; i++) {
    numero[i] = i;
}

for (int i = 2; i * i <= n; i++) {
    if (numero[i] != 0) {
        int j = 2;

        while (i * j <= n) {
            numero[i * j] = 0;
            j = j + 1;
        }
    }
}

int contadorPrimos = 0;
for (int i = 2; i <= n; i++) {
    if (numero[i] != 0) {
        contadorPrimos++;
        System.out.println(i + " es primo");
    }
}

System.out.println();
System.out.println("Total: " + contadorPrimos + " numeros primos");
```

Tamaño del vector

El vector sigue ocupando $4(n+1)$ bytes, porque continúa usando enteros. La mejora aparece en el número de iteraciones necesarias para marcar múltiplos.

7. Solución 3: vector de booleanos

En esta implementación se utiliza un **vector de booleanos** para representar si cada número es primo o no. El vector se inicializa con `true` para todos los números desde 2 hasta n, y luego se marcan como `false` los múltiplos de cada número primo encontrado.

```
Scanner sc = new Scanner(System.in);
System.out.print("Introduce un numero entero: ");
int n = sc.nextInt();

boolean[] esPrimo = new boolean[n + 1];
for (int i = 2; i <= n; i++) {
    esPrimo[i] = true;
}

for (int i = 2; i * i <= n; i++) {
    if (esPrimo[i]) {
        int j = 2;

        while (i * j <= n) {
            esPrimo[i * j] = false;
            j = j + 1;
        }
    }
}

int contadorPrimos = 0;
for (int i = 2; i <= n; i++) {
    if (esPrimo[i]) {
        contadorPrimos++;
        System.out.println(i + " es primo");
    }
}

System.out.println();
System.out.println("Total: " + contadorPrimos + " numeros primos");
```

Tamaño del vector

El vector utilizado ocupará aproximadamente $1(n+1)$ bytes, ya que se utiliza un booleano por cada número.

8. Solución 4: solo números impares

En esta implementación se utiliza un **vector de booleanos** para representar si cada número **impar** es primo o no. Como el único número par primo es el 2, podemos ignorar los pares mayores que 2 y reducir a la mitad el tamaño del vector.

Los números impares mayores que 2 pueden escribirse como:

$$p = 2i + 3 \quad \text{con } i = 0, 1, 2, \ldots, \left\lfloor \frac{n-3}{2} \right\rfloor$$

Por tanto, el vector solo necesita:

$$\left\lfloor \frac{n-3}{2} \right\rfloor + 1$$

posiciones. Para cada primo p, eliminamos sus múltiplos impares menores que n: $3p, 5p, 7p, \ldots$.

```
Scanner sc = new Scanner(System.in);
System.out.print("Introduce un numero entero: ");
int n = sc.nextInt();

int max = ...;      // calcular el tamano del vector de impares
boolean[] numero = new boolean[max + 1];

for (int i = 0; i <= max; i++) {
    numero[i] = true;
}

for (int i = 0; ... <= n; i++) {
    int k = ...;
    while (... <= n) {
        numero[...] = false;
        k = k + 1;
    }
}

int contadorPrimos = 1;
System.out.print("2 es primo \n");

for (int i = 0; i <= ...; i++) {
    if (numero[i] != false) {
        contadorPrimos = contadorPrimos + 1;
        System.out.println(... + " es primo");
    }
}

System.out.println();
System.out.println("Total: " + contadorPrimos + " numeros primos");
```

Tamaño del vector

El vector utilizado ocupará aproximadamente $1\left(\lfloor(n-3)/2\rfloor+1\right)$ bytes. Para $n=100$, son 49 bytes, aproximadamente el 12 % del tamaño del vector de enteros de la primera solución.

9. Entrega de la solución

Sólo debe realizar la entrega un miembro del grupo. Sube a Moodle un fichero comprimido en formato ZIP que contenga:

- `Solucion1.java`: implementación básica utilizando un vector de enteros.
- `Solucion2.java`: optimización utilizando la raíz cuadrada de n.
- `Solucion3.java`: implementación utilizando un vector de booleanos.
- `Solucion4.java`: implementación con vector de booleanos y solo números impares.
- `tiempos.xlsx`: fichero de Excel con los tiempos de ejecución para los valores de n indicados.