UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

GRUPO:B2

PRÁCTICA 1. Estudio empírico de la complejidad de algoritmos. Raíz Cuadrada

Trabajo realizado por (grado de participación):

Miguel de las Heras Fuentes → 50%

Isaac González del Pozo → 50%

Asignatura: Metodología de la Programación

Grupo: B2

Titulación: Grado en Ingeniería Informática

Fecha: 12/03/2021

Tabla de contenido

1. Documentación sobre el código

1.1 Método main

En primer lugar, dentro de nuestro método main (Imagen 1) , declararemos las variables necesarias para la resolución del problema, entre ellas se encontrará el vector proporcionado en el enunciado, los tiempos iniciales y finales para saber cuál es el tiempo de ejecución al ejecutar cada método. Cuando inicie dicho programa, le pedirá al usuario la unidad de tiempo que quiere utilizar para mostrar por pantalla de los métodos utilizados, comprobando que dicha opción está entre las disponibles. Posteriormente, dentro del bucle, para cada algoritmo, estableceremos un tiempo de inicio y un tiempo final para saber cuánto ha tardado el tiempo de ejecución. En el caso de que el usuario elija la opción de los nanosegundos, utilizaremos System.nanoTime y en caso contrario, haremos uso del método nanoToMilis, que explicaremos más tarde.

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

    int opc;
    int vector[] = {1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225,256, 289, 324, 361, 400 };
    long tiempoInicioMath, tiempoFinalMath;
    long tiempoInicioBabilonico,tiempoFinalBabilonico;
    long tiempoInicioBinary, tiempoFinalBinary;
    long tiempoInicioRecursivo,tiempoFinalRecursivo;
    boolean esValido;
    do {
        System.out.println("¿Qué unidad de tiempo quieres utilizar para la prueba?:\n1. Nanosegundos\n2. Milisegundos");
        esValido = comprobar(opc = teclado.nextInt());
        if (esValido == false) {
            System.out.println("Valor no válido, introduzcalo de nuevo");
        }
    }while(esValido!= true);

    for (int i = 0; i < vector.length; i++) {

        tiempoInicioMath = System.nanoTime();
        FormaMath(vector[i]);
        tiempoFinalMath = System.nanoTime();

        tiempoInicioBabilonico = System.nanoTime();
        MetodoBabilonico(vector[i]);
        tiempoFinalBabilonico = System.nanoTime();

        tiempoInicioBinary = System.nanoTime();
        BinarySearch(vector[i]);
        tiempoFinalBinary = System.nanoTime();

        tiempoInicioRecursivo = System.nanoTime();
        calcular_raiz(vector[i],1);
        tiempoFinalRecursivo = System.nanoTime();

        if (opc == 2) {
            tiempoInicioMath = nanoToMilis(tiempoInicioMath);
            tiempoFinalMath = nanoToMilis(tiempoFinalMath);

            tiempoInicioBabilonico = nanoToMilis(tiempoInicioBabilonico);
            tiempoFinalBabilonico = nanoToMilis(tiempoFinalBabilonico);

            tiempoInicioBinary = nanoToMilis(tiempoInicioBinary);
            tiempoFinalBinary = nanoToMilis(tiempoFinalBinary);

            tiempoInicioRecursivo = nanoToMilis(tiempoInicioRecursivo);
            tiempoFinalRecursivo = nanoToMilis(tiempoFinalRecursivo);
        }

        System.out.printf("i = " + vector[i] +"\t| Resultado: "+FormaMath(vector[i])+ "\t| Tiempo FormaMath: "
        + (tiempoFinalMath - tiempoInicioMath)
        +"\t| Tiempo Babilonico: "+ (tiempoFinalBabilonico - tiempoInicioBabilonico)
        + "\t| Tiempo binarySearch: " + (tiempoFinalBinary - tiempoInicioBinary)+ "\t| Tiempo recursivo: "
        + (tiempoFinalRecursivo - tiempoInicioRecursivo));
        System.out.println();

    }
```

1.2 nanoToMilis

Dentro de este método recibiremos como parámetro el tiempo de ejecución en nanosegundos que habíamos comentado anteriormente. Este método nos devolverá dicho tiempo en milisegundos (Imagen 2)

```
public static long nanoToMilis(long tiemponano) {
    long milis=(long) (tiemponano * Math.pow(10, -6));
    return milis;
}
```

1.3 FormaMath

Este será el primer método que calculará la raíz cuadrada de nuestro vector proporcionado en el enunciado. Dicho método hará uso de la clase Math proporcionada por Java, y retornará como resultado la raíz cuadrada de dicho número (Imagen 3)

```
public static int FormaMath(int vector) {

    return (int) Math.sqrt(vector);

}
```

1.4 BinarySearch

Este método será uno de los algoritmos iterativos utilizados para calcular la raíz cuadrada, al igual que el anterior, recibirá el vector proporcionado en el enunciado, y realizará el bucle necesario que realiza el algoritmo de búsqueda binaria, retornando como resultado la raíz cuadrada del número. (Imagen 4)

```
public static int BinarySearch(int vector) {
    int low = 0;
    int high = vector + 1;
    while (high - low > 1) {
        int mid = (low + high) / 2;
        if (mid * mid <= vector) {
            low = mid;
        }

        else {
            high = mid;
        }

    }
    return low;


}
```

1.5 MetodoBabilonico

Al igual que el anterior método, este será también uno de los algoritmos iterativos utilizados para calcular la raíz cuadrada, haciendo uso del algoritmo babilónico para calcular la raíz cuadrada. Este retornará como resultado la raíz cuadrada del número (Imagen 5)

```
public static int MetodoBabilonico(int vector) {
    double radicando = vector;
    double resultado = 1;
    int i = 0;
    while(i<25){
        resultado=resultado-((resultado*resultado-radicando)/(2*resultado));
        i++;
        }
    return (int) resultado;

}
```

1.6 Calcular_raiz

Este método recibirá como parámetros el valor (siendo este los distintos valores del vector) y una variable auxiliar. Dicho método será el que utilice el algoritmo de manera recursiva para calcular la raíz cuadrada de los valores del vector. (Imagen 6)

```
public static double calcular_raiz(double valor, double aux)//T(n)=T(n-1)+1
{
    if(aux*aux==valor) return aux;
    else
    {
        if(aux*aux>valor) return calcular_raiz(valor,aux/2);
        else return calcular_raiz(valor,aux+1);
    }
}
```

2. Complejidad teórica de los algoritmos

La complejidad teórica de cada algoritmo para calcular la raíz cuadrada será la siguiente:

2.1 FormaMath

Para el primer caso, el método FormaMath, la complejidad teórica sería $O(1)$ ya que se trata de un único return en dicho método, y se mantendría una constante en la complejidad, y como veremos más tarde, este método será el más eficiente de todos.

2.2 BinarySearch

En el método BinarySearch, la complejidad teórica será $O(logn)$ ya que la mayor complejidad se encuentra en el bucle while donde se irá dividiendo el intervalo que se está analizando y, por tanto, tendremos una complejidad logarítmica.

2.3 Babilónico

En el método Babilónico, la complejidad teórica será $O(n)$. Esto es debido a que tenemos un bucle while que se ejecutará n veces ya que depende de la i.

2.4 Método Recursivo

En el último método, es decir, Calcular_raiz, que sería nuestro algoritmo recursivo, la complejidad teórica correspondería a $T(n) = T(n-1) + 1$, que se resolvería de la siguiente manera:

$$T(n) = T(n-1) + 1$$

$$t_n - t_{n-1} + 1$$

$$X^n - X^{n-1} = 1 \qquad\qquad b^n p(n)^d$$

$$X^{n-1}(X-1) = 1$$

$$(X-1)^2 \qquad r = 1\ (doble)$$

$$t_n = C_1(1)^n + C_2 n(1)^n$$

$$\in \boldsymbol{O(n)}$$

3. Complejidad empírica de los algoritmos

```
¿Qué unidad de tiempo quieres utilizar para la prueba?:
1. Nanosegundos
2. Milisegundos
1
i = 1   | Resultado: 1  | Tiempo FormaMath: 75200      | Tiempo Babilonico: 2200      | Tiempo binarySearch: 1000     | Tiempo recursivo: 1400
i = 4   | Resultado: 2  | Tiempo FormaMath: 400 | Tiempo Babilonico: 800        | Tiempo binarySearch: 500      | Tiempo recursivo: 500
i = 9   | Resultado: 3  | Tiempo FormaMath: 400 | Tiempo Babilonico: 800        | Tiempo binarySearch: 500      | Tiempo recursivo: 300
i = 16  | Resultado: 4  | Tiempo FormaMath: 300 | Tiempo Babilonico: 700        | Tiempo binarySearch: 400      | Tiempo recursivo: 500
i = 25  | Resultado: 5  | Tiempo FormaMath: 300 | Tiempo Babilonico: 700        | Tiempo binarySearch: 500      | Tiempo recursivo: 500
i = 36  | Resultado: 6  | Tiempo FormaMath: 200 | Tiempo Babilonico: 700        | Tiempo binarySearch: 400      | Tiempo recursivo: 500
i = 49  | Resultado: 7  | Tiempo FormaMath: 200 | Tiempo Babilonico: 600        | Tiempo binarySearch: 400      | Tiempo recursivo: 500
i = 64  | Resultado: 8  | Tiempo FormaMath: 300 | Tiempo Babilonico: 700        | Tiempo binarySearch: 400      | Tiempo recursivo: 500
i = 81  | Resultado: 9  | Tiempo FormaMath: 300 | Tiempo Babilonico: 700        | Tiempo binarySearch: 500      | Tiempo recursivo: 500
i = 100 | Resultado: 10 | Tiempo FormaMath: 200 | Tiempo Babilonico: 600        | Tiempo binarySearch: 500      | Tiempo recursivo: 600
i = 121 | Resultado: 11 | Tiempo FormaMath: 200 | Tiempo Babilonico: 700        | Tiempo binarySearch: 500      | Tiempo recursivo: 600
i = 144 | Resultado: 12 | Tiempo FormaMath: 200 | Tiempo Babilonico: 700        | Tiempo binarySearch: 500      | Tiempo recursivo: 600
i = 169 | Resultado: 13 | Tiempo FormaMath: 200 | Tiempo Babilonico: 600        | Tiempo binarySearch: 500      | Tiempo recursivo: 700
i = 196 | Resultado: 14 | Tiempo FormaMath: 200 | Tiempo Babilonico: 700        | Tiempo binarySearch: 600      | Tiempo recursivo: 600
i = 225 | Resultado: 15 | Tiempo FormaMath: 200 | Tiempo Babilonico: 700        | Tiempo binarySearch: 500      | Tiempo recursivo: 700
i = 256 | Resultado: 16 | Tiempo FormaMath: 300 | Tiempo Babilonico: 700        | Tiempo binarySearch: 15000    | Tiempo recursivo: 1800
i = 289 | Resultado: 17 | Tiempo FormaMath: 300 | Tiempo Babilonico: 700        | Tiempo binarySearch: 500      | Tiempo recursivo: 800
i = 324 | Resultado: 18 | Tiempo FormaMath: 300 | Tiempo Babilonico: 700        | Tiempo binarySearch: 400      | Tiempo recursivo: 21400
i = 361 | Resultado: 19 | Tiempo FormaMath: 200 | Tiempo Babilonico: 600        | Tiempo binarySearch: 400      | Tiempo recursivo: 800
i = 400 | Resultado: 20 | Tiempo FormaMath: 300 | Tiempo Babilonico: 600        | Tiempo binarySearch: 400      | Tiempo recursivo: 700
```

4. Conclusiones sobre los valores obtenidos

Como podemos observar, se imprimen los tiempos de ejecución de todos los métodos utilizados para hallar la raíces cuadradas.

Hemos llegado a la conclusión de que el algoritmo más eficiente teóricamente es el algoritmo FormaMath, ya que su complejidad teórica es de $O(1)$. Después de realizar los ensayos, podemos observar que efectivamente este es el algoritmo más eficiente y se cumple por tanto la complejidad teórica calculada anteriormente