Practica 1

Estudio empírico de la complejidad de

18-03-2021
Metodología e la programación

Alberto García Aparicio
Laura Morales Caro
B1

INDICE

1. Introducción

Para esta primera práctica se nos pide implementar un programa en Java en el que utilicemos 4 formas diferentes para calcular la raíz cuadrada perfecta.

En la primera forma hemos utilizado la función sqrt de la librería Math para obtener la raíz cuadrada.

En la segunda forma utilizamos el método Babilonio. Este método se centra en el hecho de que cada lado de un cuadrado es la raíz cuadrada del área. Para calcular una raíz, dibuje un rectángulo cuya área sea el número al que se le busca raíz y luego aproxime la base y la altura del rectángulo hasta formar o por lo menos aproximar un cuadrado.

En la tercera forma utilizamos la forma Binary Search, se comienza por comparar el elemento del medio del arreglo con el valor buscado. Si el valor buscado es igual al elemento del medio, su posición en el arreglo es retornada. Si el valor buscado es menor o mayor que el elemento del medio, la búsqueda continua en la primera o segunda mitad, respectivamente, dejando la otra mitad fuera de consideración.

En la cuarta forma utilizamos recursividad en el método Babilonio. Cambiamos el método utilizado en la segunda forma para que sea recursivo.

2. Comentarios del código

```
public static long metodoMath(int n) {
		long tiempoInicio = System.nanoTime();
		double solucion = (float) Math.sqrt(n);
		return (System.nanoTime()-tiempoInicio);
	}
```

Llamamos a la función sqrt y le pasamos n, el número del que queremos calcular la raíz. Una vez calculada la raíz la mostraremos por pantalla y devolveremos el tiempo empleado en el cálculo de esta.

```
public static long metodoBabilonico(int n){
		long tiempoInicio = System.nanoTime();
		double x = n;
		double y = 1;
		double e = 0.000001; /*indice de error*/
		double solucion;

		while (x - y > e) {
			x = (x + y) / 2;
			y = n / x;
		}
		solucion = x;
		return (System.nanoTime()-tiempoInicio);
	}
```

En este método implementaremos la función (a + x^2)/2x. Mientras que "x" menos "y" sea mayor que el índice de error seguiremos calculando la raíz. "x" será igual a "x" mas "y" partido de dos e "y" será n partido de "x". Cuando "x" menos "y" sea menor que "e" mostraremos por pantalla la raíz cuadrada calculada y devolveremos el tiempo que hemos tardado.

```
public static long metodoBinarySearch(int n) {
                long tiempoInicio = System.nanoTime();
                double e = 0.000001; /*indice de error*/
                double principio = 0, fin = n, mitad=n/2;
                double solucion;

          while ((((mitad * mitad)-n)>e) || ((n-(mitad * mitad))>e)){

                if (mitad * mitad > n){
                fin = mitad;
            }

            if (mitad * mitad < n) {
                principio = mitad;
                    }

            mitad = (principio + fin) / 2;
            }

                solucion = mitad;

          return (System.nanoTime()-tiempoInicio);
        }
```

La raíz cuadrada sabemos que está entre el rango 0 y n por lo que pondremos el inicio a 0 y el final a n para ir buscando la raíz mediante divisiones entre dos. Iremos comparando el medio con nuestra raíz integral.

Si al multiplicar la mitad por ella misma el resultado es mayor que la raíz integral supondrá que el valor que buscamos se encuentra a la izquierda de la mitad, con lo cual el valor de final pasa a ser el que tenía la mitad y así reducir la cantidad de números a la mitad, en el caso de que la mitad multiplicada por ella misma sea menor a n el valor del principio pasara a ser el de la mitad porque eso significa al igual que antes que el resultado está a la derecha de la mitad, eso se hará sucesivamente hasta que el resultado de multiplicar la mitad por ella misma menos la raíz integral sea menor que el error.

```
public static long metodoBabilonicoRecursivo(int n){
                long tiempoInicio = System.nanoTime();
                double x = n;
                double y = 1;

        recursividad(n, x, y);

        return (System.nanoTime()-tiempoInicio);
    }
```

En este método declaramos las variables tiempoInicio, “x” e “y” y llamamos al método recursividad pasandole estas variables. Una vez hayamos terminado en el método recursividad mostraremos por pantalla el tiempo empleado.

```
private static void recursividad(int n, double x, double y) {
    double e = 0.000001; /*indice de error*/
    double solucion = 0;

            if (x - y > e) {
               x = (x + y) / 2;
               y = n / x;
            recursividad(n, x, y);
            }
            else solucion = x;

    }
```

En este método calcularemos la raíz cuadrada de forma recursiva. La variable "e" será el índice de error y miraremos que "x" menos "y" sea mayor que "e", si es así "x" le pasaremos el valor de ((x+y)/2) e "y" le pasaremos el valor de (n/x) y luego llamaremos a este método pasándole los valores de "n", "x" e "y".

3. Complejidad de los algoritmos

3.1 Complejidad de metodoMath

En la mayoría de los casos, Java intenta utilizar el algoritmo "Smart-power", que da como resultado una complejidad temporal de O(log(n)).

3.2 Complejidad de metodoBabilonico

```
public static long metodoBabilonico(int n){
        long tiempoInicio = System.nanoTime();
        double x = n;
        double y = 1;
        double e = 0.000001; /*indice de error*/
        double solucion;

        while (x - y > e) {
            x = (x + y) / 2;
            y = n / x;
        }
        solucion = x;
        return (System.nanoTime()-tiempoInicio);
    }
```

La complejidad del algoritmo babilónico es de O(log(log(n/e)). El error relativo en el paso k satisface la ecuación $e_k < 2^{f(k)}$ donde f(k):

- f(1) = 2
- f(k+1) = 2*f(k)+1 para n>1

Por inducción f(k) = $=3*2^{k-1}-1$. Siendo n la entrada de nuestro algoritmo, que se detiene cuando x-y es menor que el error. El error en el paso k, ek satisface la ecuación $E_k = e_k * n$, por lo que el algoritmo acabará cuando ek *n<m. Esto ocurre $f(k) > \log_2(n/e)$. El algoritmo acabara em O(log(log(n/e)).

3.3 Complejidad de metodoBabilonicoRecursivo

```
public static long metodoBabilonico(int n){
        long tiempoInicio = System.nanoTime();
        double x = n;
        double y = 1;

        recursividad(n,x,y);
        return (System.nanoTime()-tiempoInicio);
    }
```

La complejidad de este algoritmo es de O(1), aunque tiene una llamada al metodo "recursividad", con lo cual tiene complejidad O(log(log(n/e)).

```
public static void recursividad(int n){
                double e = 0.000001; /*indice de error*/
                double solución=0;

                if (x - y > e) {
                        x = (x + y) / 2;
                        y = n / x;
                        recursividad(n,x,y);
                }
                Else solución=x;
        }
```

Este metodo tiene exactamente las misma complejidad que el metodo iterativo metodoBabilonico siguiente los mismos pasos.

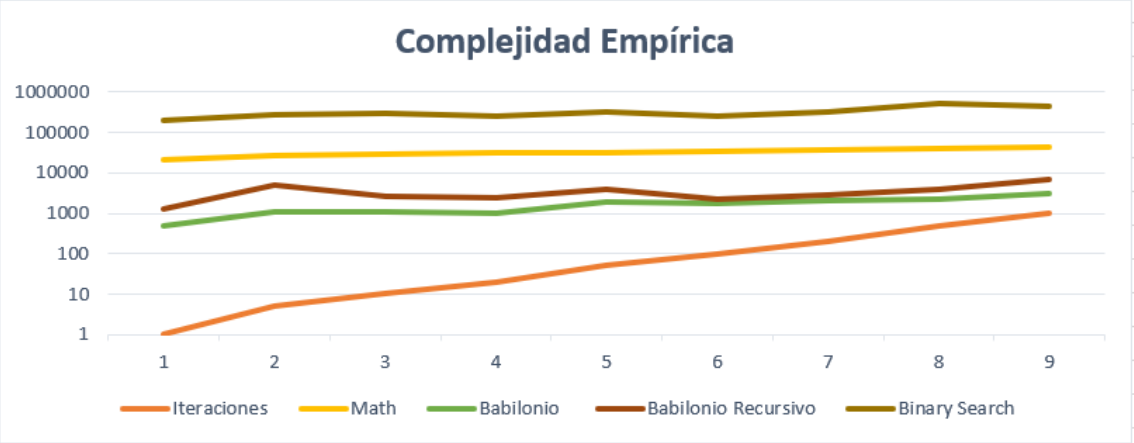
3.4 Complejidad de metodoBinarySearch

```
public static long metodoBinarySearch(int n) {
            long tiempoInicio = System.nanoTime();
            double e = 0.000001; /*indice de error*/
            double principio = 0, fin = n, mitad=n/2;
            double solucion;

        while ((((mitad * mitad)-n)>e) || ((n-(mitad * mitad))>e)){    --> O(logn)
            mitad = N   k = n° de veces que se repite el while
                        2^k<N^2 ;    k = log2 N^2  => 2 log2 N = log N

            if (mitad * mitad > n){    --> O(1)
            fin = mitad;
          }

          if (mitad * mitad < n) {     --> O(1)
            principio = mitad;
              }

          mitad = (principio + fin) / 2;     --> O(1)
          }

            solucion = mitad;   --> O(1)

        return (System.nanoTime()-tiempoInicio); -> O(1)
      }
```

La complejidad de nuestro algoritmo es de O(log n). En el mejor de los casos el algoritmo tendrá una complejidad de O(1) ya que el número que queremos encontrar se encuentra en la mitad. En el peor de los casos será que el número que queremos encontrar se encuentra en una de las extremidades.

4. Complejidad Empírica

Iteraciones	Math	Babilonio	Babilonio Recursivo	Binary Search
1	21401	499	1300	195400
5	26500	1100	5100	264400
10	28600	1100	2600	286400
20	30600	1000	2500	254400
50	31500	1900	3900	329800
100	35400	1700	2200	247400
200	38300	2100	2800	313500
500	40700	2300	3900	521700
1000	44200	3200	6900	428800



5. Bibliografía

https://es.wikipedia.org/wiki/Cálculo_de_la_raíz_cuadrada#:~:text=El%20algoritmo%20babilónico%E2%80%8B%20se,la%20raíz%20cuadrada%20del%20área.&text=Para%20calcular%20una%20raíz%2C%20dibuje,lo%20menos%20aproximar%20un%20cuadrado
https://es.wikipedia.org/wiki/Búsqueda_binaria
https://stackoverflow.com/questions/28815339/time-complexity-of-math-sqrt-java