Análisis de complejidad: Contador de números repetidos

Cristian Andrés Parra Radillo

Algoritmos y complejidad

Ingeniería de Sistemas

Universidad del norte

2022

Resumen:

En el siguiente trabajo presenta un algoritmo capaz de contar números repetidos creado pseudo-aleatoriamente de tamaño N. Se implementa la ejecución del algoritmo en el lenguaje de programación Java, mediante el IDE NetBeans y a su vez se desarrolla el mejor y peor algoritmo para resolver el problema(best case a worst case).

Así mismo, se hace uso del aplicativo Matlab para poder graficar el número de comparaciones y el tiempo de ejecución con respecto al número de entradas.

Introducción:

La complejidad de un algoritmo se representa como O(), nos ayuda a determinar que tan optimo puede llegar a ser en la solución de problemas. Para problemas donde la data sea muy grande, algortimos con una complejidad muy grande, tendrán un alto costo a nivel de procesamiento. Es por eso, que en este informe abordaremos dos casos: El primero es el best case, donde se busca una complejidad de menor grado y un worst case, donde veremos el alto consumo cuando la data sea muy grande. Usaremos estos dos métodos para hacer un análisis detallado de los tiempos de ejecución y numero de comparaciones que hacer cada algoritmo.

Definición del problema y métodos:

Se genera Pseudo-Aleatoriamente un conjunto N de números enteros positivos que se guardaran en un archivo de texto(.txt). Se deberán contar todos los números repetidos e imprimirlos por consola. Así mismo, se debe contar el número de comparaciones y tomar el tiempo que le toma al algoritmo contar los números repetidos.

En los métodos, usaremos las operaciones IO en java, para la creación del archivo de texto. Haremos uso la función **Math.random()** de java para generar los números de manera aleatorio y el método **System.nanoTime()** el tiempo de ejecución del programa. Por último, con los conceptos de complejidad, hallaremos la función que representa el número de comparaciones con respecto a N (Número de datos) y con todos estos datos, crearemos un archivo de texto con los parámetros (N, Comparaciones, tiempo) que usaremos para graficar en Matlab.

Resultados:

Worst Case

Algoritmo:

```
Scanner in = new Scanner(f);                        // 1 iteraciones
int x;                                             // 1 iteraciones
//Incializamos un vector para agregar los datos
int Numeros[] = new int[size + 1];                  // 1 iteraciones

//Ponemos la cadena en un vector
for (int i = 0; i < size; i++) {
    x = in.nextInt();                                  // n iteraciones
    Numeros[i] = x;                                   // n iteraciones
}

//Ordenamiento del vector
for (int i = 0; i < (Numeros.length - 1); i++) {
    for (int j = i + 1; j < Numeros.length; j++) {
        if (Numeros[i] > Numeros[j]) {                          // n^2 iteraciones
            //Intercambiamos valores
            int aux = Numeros[i];                         // n^2 iteraciones
            Numeros[i] = Numeros[j];                        // n^2 iteraciones
            Numeros[j] = aux;                          // n^2 iteraciones

        }
    }
}

//contamos los valores repetidos
int contador = 0;                                  // 1 iteraciones
int aux = Numeros[0];                                 // 1 iteraciones
for (int i = 0; i < size; i++) {
    if (aux == Numeros[i]) {                                  // n iteraciones
        contador++;                                   // n iteraciones
    } else {
        System.out.println("El numero " + Numeros[i - 1] + " se repite: " + contador + " "); //n iteraciones
        contador = 1;                                     // n iteraciones
        aux = Numeros[i];                                  // n iteraciones
    }
}

in.close(); //
```

Para fines prácticos del ejercicio, solo tendremos en cuenta el conteo de los repetidos. Es decir, no se analizará la creación de los archivos y tiempos de estos mismos.

Para la realización de este worst case, partimos con la creación de un arreglo para guardar los datos del archivo, después procedemos a ejecutar un algoritmo de ordenamiento y ya ordenamos podemos contar los repetidos e imprimirlos por consola.

Para el conteo de iteraciones que tiene el algoritmo analizamos cuantas veces este pasa por cada línea de código. Haciendo los cálculos respectivos, podemos determinar una función con respecto a n que nos determine el número de iteraciones que hace:

$$f(n) = 4n^2 + 7n + 5$$

El orden de complejidad será el grado más alto al que se encuentre nuestra función en este caso será: $O(n) = n^2$

Para la ejecución del algoritmo tomamos valores de n = $2^i \; para \; i = 1 \; hasta \; 19$

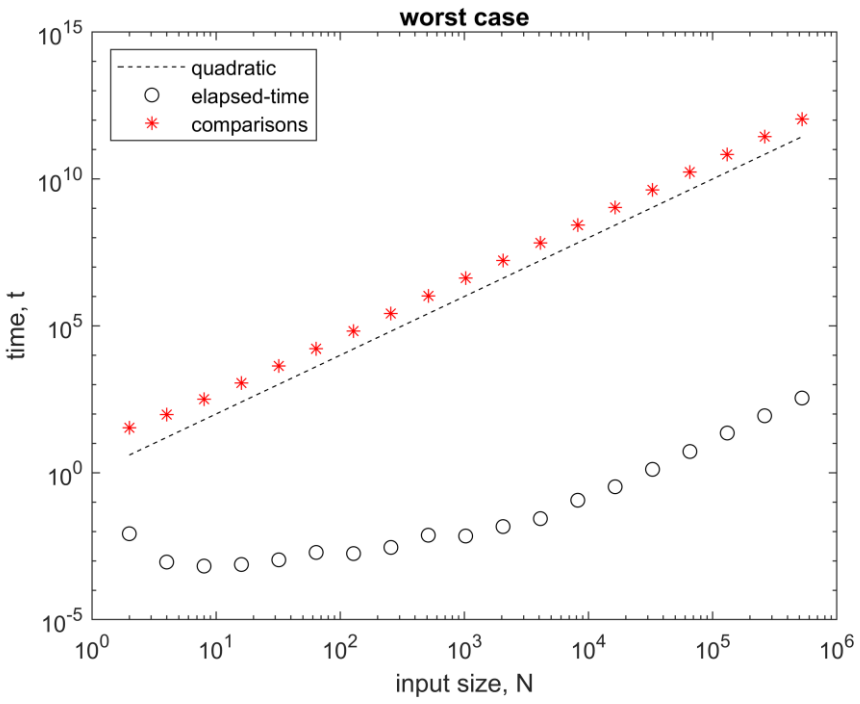
Archivo de texto generado por el algoritmo:

```
2,34,0.0083933
4,96,9.112E-4
8,316,6.647E-4
16,1140,7.52599E-4
32,4324,0.0010814
64,16836,0.001935299
128,66436,0.0017724
256,263940,0.0028689
512,1052164,0.007472301
1024,4201476,0.0070043
2048,16791556,0.0147145
4096,67137540,0.0274045
8192,268492804,0.116077201
16384,1073856516,0.3332859
32768,4295196676,1.3038738
65536,17180327940,5.2849098
131072,68720394244,22.475856501
262144,274879741956,86.930002699
524288,1099515297796,348.896584899
```

Fg 1.1

Podemos encontrar las 19 ejecuciones con n diferentes. Cada fila tiene el valor de n, las iteraciones del algoritmo y el ultimo valor es el tiempo en segundos. (Se hizo la conversión de nanosegundos a segundos previamente a guardar los datos)

Representación de los datos echa con el aplicativo Matlab:



Fg 1.2

Best Case

Algoritmo:

```
// attempts to create a Scanner object
Scanner in = new Scanner(f);                    // 1 iteracion
int x;                                          // 1 iteracion
int count = 0;                                  // 1 iteracion
int Numeros[] = new int[size + 1];              // 1 iteracion
// reads integers in file until EOF
while (in.hasNextInt()) {
    // cuando aparece un numero se toma como indice y en ese posicion
    // se le suma +1
    x = in.nextInt();                           // n iteraciones
    Numeros[x]++;                               // n iteraciones
    ++count;                                    // n iteraciones
}
// Escribimos el vector, siendo el indice el número y el valor
// el numero de veces que aparece el numero
for (int i = 1; i <= size; i++) {
    System.out.println("[" + i + "] = " + Numeros[i]);    // n iteraciones
}
in.close();
```

Para nuestro best case, también usaremos un arreglo, en este caso el índice o posición representa cada número y su valor será el número de veces que aparece cada número dentro del archivo texto. Al final escribimos el vector por consola para que el usuario vea los números repetidos.

Para las iteraciones que hace este algoritmo tenemos la siguiente función:

$$f(n) = 4n + 4$$

El orden de complejidad será el termino con mayor exponente, en este caso$O(n) = n$

Para la ejecución del algoritmo tomamos valores de n = $2^i \; para \; i \; = 1 \; hasta \; 19$

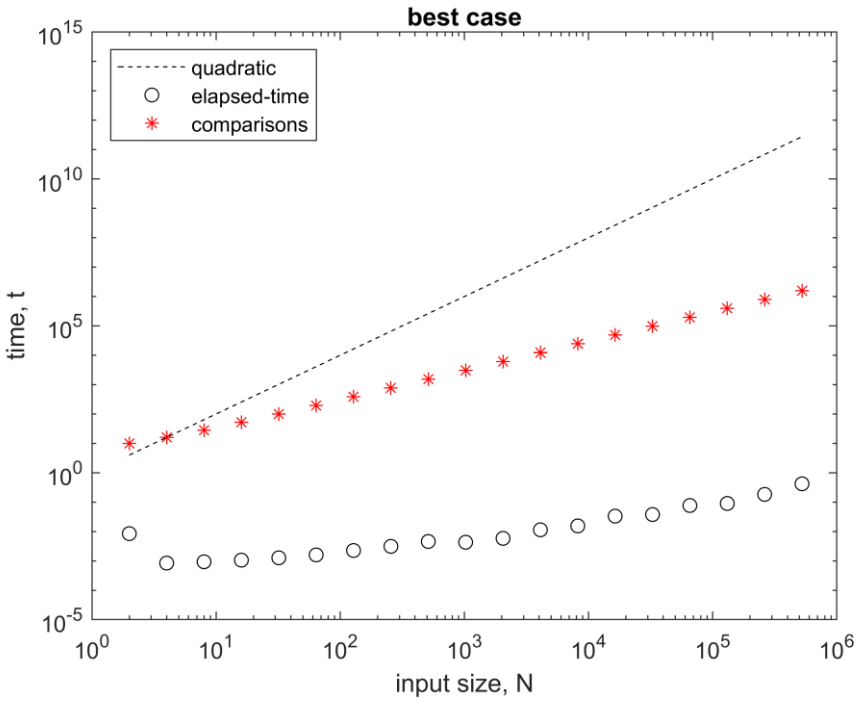
Archivo de texto generado por el algoritmo:

```
2,12,0.011169
4,20,7.013E-4
8,36,0.001018501
16,68,0.001293
32,132,0.001187801
64,260,0.001401099
128,516,0.0014496
256,1028,0.002474901
512,2052,0.003535401
1024,4100,0.003765901
2048,8196,0.005018701
4096,16388,0.007450301
8192,32772,0.0131295
16384,65540,0.026746999
32768,131076,0.032432499
65536,262148,0.061433601
131072,524292,0.0748166
262144,1048580,0.1736227
524288,2097156,0.301447799
```

Fg 2.1

Podemos encontrar las 19 ejecuciones con n diferentes. Cada fila tiene el valor de n, las iteraciones del algoritmo y el ultimo valor es el tiempo en segundos. (Se hizo la conversión de nanosegundos a segundos previamente a guardar los datos)

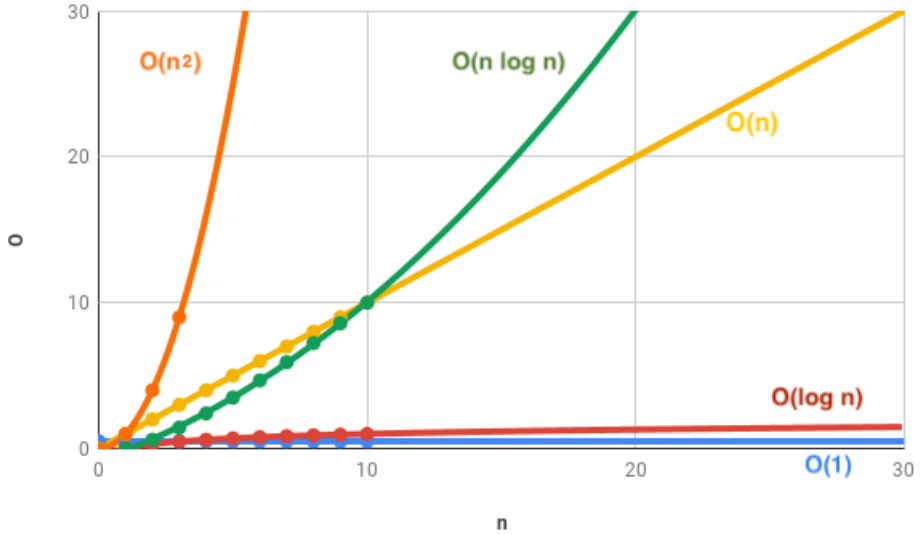
Representación de los datos echa con el aplicativo Matlab:



Fg 2.2

Discusiones:

Como podemos observar en la *Fg 1.1* y en *Fg 2.1* tomando un N muy pequeño. El algoritmo tiende a ejecutarse con tiempo bastante similar. Entendamos porque pasa esto viendo la siguiente gráfica:



Podemos ver claramente que para un n pequeño la complejidad del algoritmo es casi que igual. Sin embargo, a medida que el N va tomando valores más grandes, esa tendencia o similitud en complejidad se va perdiendo totalmente.

Así mismo, en los resultados ya expuestos vimos como el orden de complejidad de las dos formas presentada son diferentes. Para el best case tenemos un orden de complejidad $O(n) = n$ y para el worst case tenemos un orden de complejidad $O(n) = n^2$.

En la figura 1.2 y 2.2 para $n = 2^{19}$ el best case solo le tomo 0.3 segundos en cambio al worst case le tomo 348 segundos en completar la tarea y es aquí donde se puede entender la importancia de la complejidad.

Por otro lado, podríamos decir que existe un algoritmo de $O(n) = n \log n$ que sería nuestro average case para la solución de este ejercicio.

Conclusiones:

En conclusión, podemos decir que la complejidad de los algoritmos es de suma importancia en el desarrollo y solución de problemas mediante aplicaciones de software. Ya que una complejidad en este caso, cuadrática era totalmente ineficiente para $n > 2^{19}$ haciendo un gasto totalmente innecesario que podría aprovecharse de mejores maneras. Es importante que todo programador haga uso de todas las herramientas posibles para optimizar lo más posible un algoritmo y siempre poder llegar al best case.

Bibliografia:

Scalise, E., & Carmona, R. (2001). Análisis de Algoritmos y Complejidad.

AHO Alfred V. Ullman Jeffrey D. Foundations of Computer Science C Edition. Computer Science Press. An Imprint of W.H. Freeman and Company. New York. 1.996

BRASSARD G, y BRATLEY p. Fundamentos de Algoritmia. Prentice Hall Inc., (1997)

Mitchell Contreras, CTO of Munily, https://www.freecodecamp.org/espanol/news/la-complejidad-de-los-algoritmos-simples-y-las-estructuras-de-datos-en-js/