Estructuras de Datos Avanzadas

Tarea 1: Ordenamiento

Prof.: Fernando Esponda

Alumno: Guillermo Arredondo Renero

C.U.: 000197256

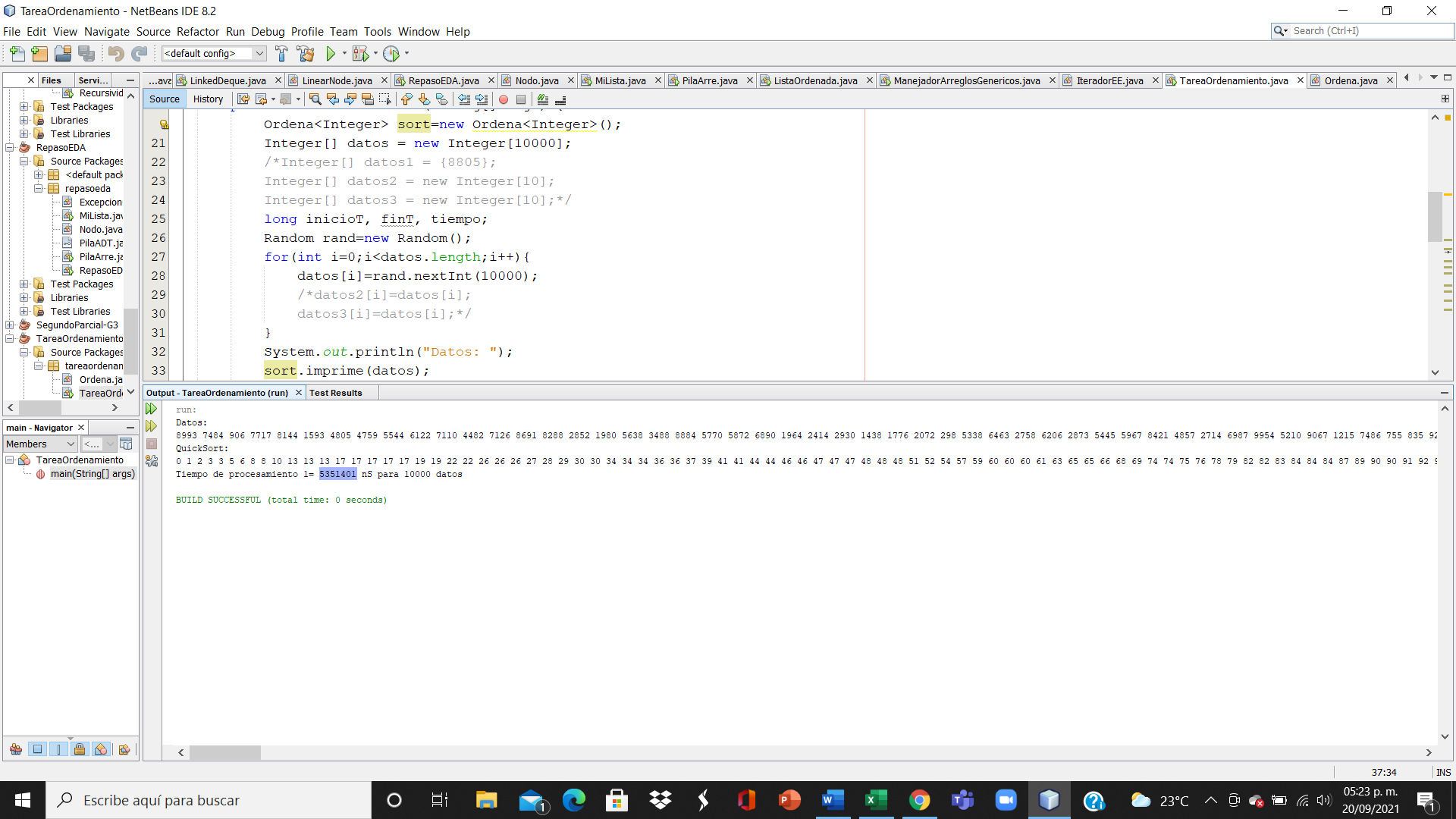
Experimento sobre Algoritmos de Ordenamiento y su eficiencia

A realizar:

1. Programe el algoritmo de Merge Sort recursivo
2. Programe el algoritmo de Merge Sort iterativo
3. Programe el algoritmo de Quick Sort
4. Diseñe un experimento para comparar el desempeño de las tres implementaciones. Su diseño debe contestar, al menos, las siguientes preguntas:
   1. ¿Son asintóticamente iguales?
   2. ¿Hay diferencia en tiempo (fracciones de segundo) al cambiar el número de elementos a ordenar?
   3. ¿Importa el orden inicial de los elementos?
   4. Además de tiempo, ¿cree usted que haya otra dimensión de eficiencia en la que uno sea mejor que otro?

Con el propósito de evaluar los comportamientos de los algoritmos de ordenación y su eficiencia se desarrolló una aplicación que llene un arreglo de **n** datos de forma aleatoria haciendo uso de la clase *Random* y acotando superiormente por el valor de 10,000 (véase Imagen 1). Al mismo tiempo, se contabilizó el número de operaciones que cada algoritmo contiene para generar una tabla de comparación. Por último, se utilizó la función *nanoTime()* para medir el tiempo de procesamiento de cada algoritmo. Con la esperanza de obtener el tiempo más aproximado, la aplicación ejecutaba un algoritmo por vez, y los datos obtenidos se escribían en una tabla. Del mismo modo, para aumentar la precisión de los valores estimados, se realizaron 3 veces los experimentos con el mismo arreglo y mismo algoritmo, consecuentemente misma longitud **n** y se obtuvo la media de tiempo.

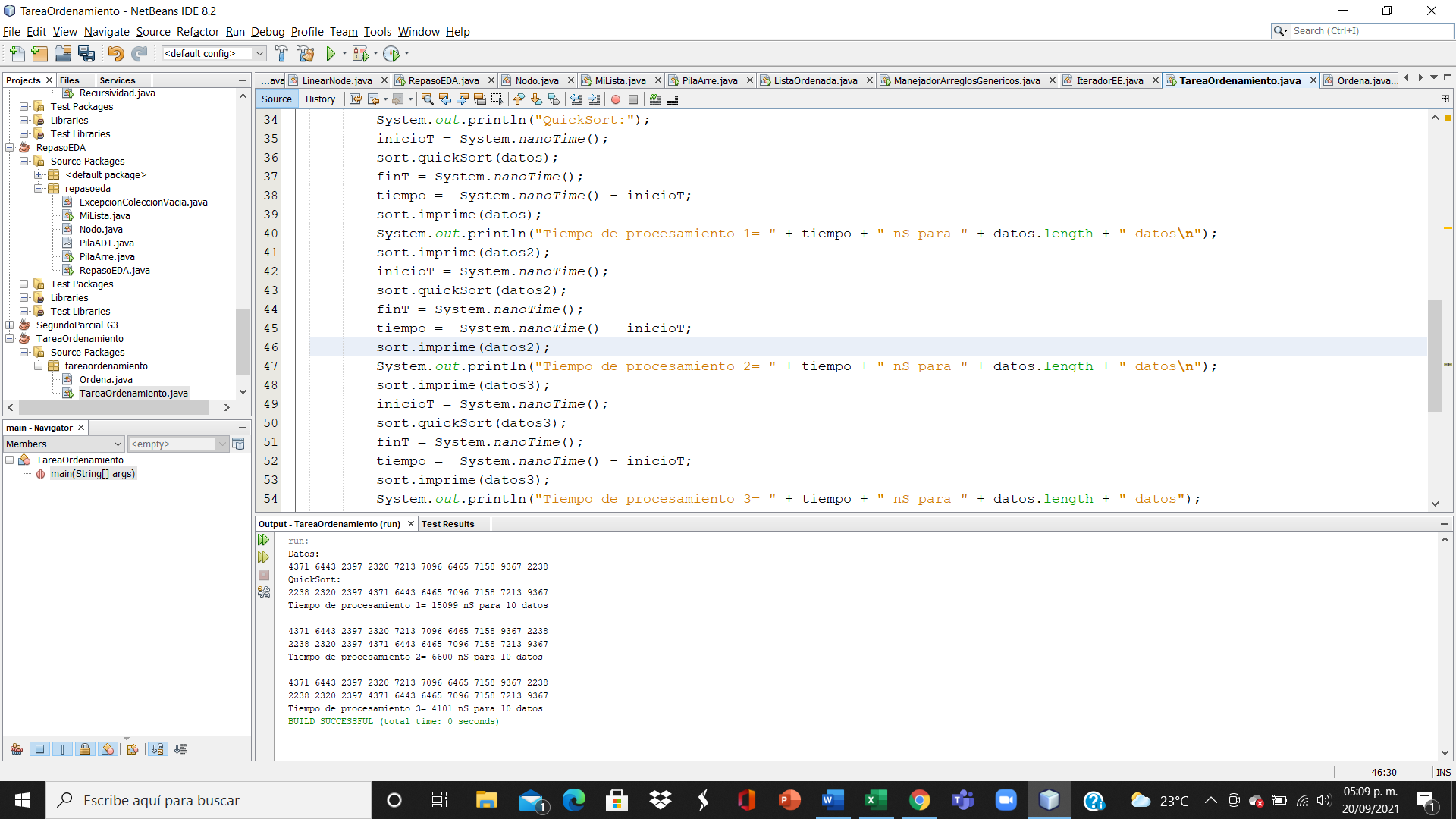
**Imagen 1.**



Algo muy curioso respecto a llevar a cabo tres instanciaciones del mismo arreglo y tomar el tiempo de cada una de las llamadas al método es que los tiempos tienden a decrecer conforme la misma instrucción es llamada. Investigando, se descubrió, dentro de la información API de la función *nanoTime()* que al generar llamadas sucesivas de la función el cálculo de tiempo tiende a la entropía por efecto del desbordamiento (overflow) numérico por superar los 263 (véase Tabla 1 e Imagen 2). Por ello, se pasó a realizar medias aritméticas considerando distintos datos en el arreglo, pero conservando la **n** longitud de entrada y tomando el tiempo una única vez.

|  |  |  |  |
| --- | --- | --- | --- |
| **Tabla 1** | | | |
| **n (longitud)** | **Tiempo1 (nS)** | **Tiempo2(nS)** | **Tiempo3 (nS)** |
| 5 | 15700 | 38800 | 2800 |
| 6 | 34500 | 6300 | 7400 |
| 10 | 28600 | 8200 | 6700 |
| 100 | 142200 | 88100 | 86100 |
| 1000 | 1778800 | 564400 | 279000 |
| 10000 | 7535400 | 14621700 | 17373600 |
| 50000 | 65226600 | 14839600 | 13226300 |

**Imagen** **2**





El experimento se diseñó para hacer uso de *Random* para generar un arreglo aleatoriamente desordenado de enteros, el cual se usa en un rango de 0-9999. Asimismo, se hace uso del algoritmo de QuickSort recursivo (siendo más eficiente en uso de memoria que el iterativo), el cual a su vez se apoya de una llamada a una función llamada partición que se encarga de acomodar los elementos menores a la izquierda del pivote y los elementos mayores o iguales a la derecha. Es importante notar que la elección del pivote se toma por default al inicio del arreglo, dado a que aún haciendo casos en el que el pivote se elija en el último lugar o en medio igual se encuentran casos específicos en los que el algoritmo modifique su tiempo de procesamiento. El caso “protegido” comentado en clase no se toma en cuenta puesto que la elección aleatoria del pivote podría crear una descompensación en el tiempo del algoritmo. Así, queda demostrada la generalidad de tomar el pivote en el primer elemento del subarreglo correspondiente.

Sin embargo, es importante notar que en el diseño del experimento es necesario tomar en cuanta los diversos casos posibles, entre ellos los siguientes:

* Caso en el que el arreglo es nulo
* Caso en el que el arreglo se encuentra desordenado
* Caso en el que el arreglo ya se encuentra ordenado ascendentemente
* Caso en el que el arreglo ya se encuentra ordenado descendentemente
* Caso en el que el arreglo cuenta con partes ordenas y otras desordenadas

En el primer caso, las conclusiones son evidentes: el número de operaciones ni siquiera depende de la longitud (dado que es cero) ni del número de operaciones, pues la única operación efectuada sería la de preguntar si el arreglo es distinto de nulo.

En el caso de un arreglo desordenado y uno semi-desordenado, la utilización del método *Random* es de mucha utilidad, puesto que generaliza las distintas ordenaciones y presenta un aproximado de tiempo más objetivo. No obstante, es importante notar que por medio del uso de la función random.nextInt(10,000) es posible la repetición de elementos dentro del arreglo, sobre todo en los casos de **n** >= 10,000. Esta combinación de casos se presenta en la siguiente tabla:

**Tabla 2. Desordenado por *Random***

|  |  |  |
| --- | --- | --- |
| **Algoritmo** | **n (longitud)** | **Tiempo Promedio (nS)** |
| QuickSort | 1 | 3366.00 |
|  | 5 | 14166.33 |
|  | 6 | 13232.67 |
|  | 10 | 20999.67 |
|  | 100 | 202833.67 |
|  | 10000 | 4899266.67 |
|  | 1000000 | 455029833.33 |
|  | 10000000 | 24742063300.00 |
|  | 100000000 | no completado |
| MergeSort Rec | 1 | 3133.33 |
|  | 5 | 15066.67 |
|  | 6 | 15966.67 |
|  | 10 | 18266.67 |
|  | 100 | 215333.33 |
|  | 10000 | 4755733.33 |
|  | 1000000 | 303296700.00 |
|  | 10000000 | 3996842033.33 |
|  | 100000000 | no completado |
| MergeSort Iter | 1 | 1900.00 |
|  | 5 | 17600.00 |
|  | 6 | 18333.33 |
|  | 10 | 20966.67 |
|  | 100 | 143266.67 |
|  | 10000 | 4300300.00 |
|  | 1000000 | 328288566.67 |
|  | 10000000 | 4331857900.00 |
|  | 100000000 | no completado |

Como puede verse en la Tabla 2 conforme aumenta el número de elementos que se desean ordenar el tiempo en nano segundos aumenta, lo cual es lógico, pues es necesario un mayor número de operaciones, así sean particiones, uniones o creación de arreglos auxiliares.

Con base en la tabla pasada y tomando los tiempos de QuickSort como tiempo base (otorgándoles el valor de 100) encontramos las siguientes relaciones a través de la fórmula tiempoAComparar/TiempoBase \* 100:

**Tabla 3. Comparación entre Tiempos de Algoritmos**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Tiempo por algoritmo** | | |  |
| **n (longitud)** | QS | MS REC | MS ITER |  |
| 1 | 3366.00 | 3133.33 | 1900.00 |  |
| 5 | 14166.33 | 15066.67 | 17600.00 |  |
| 6 | 13232.67 | 15966.67 | 18333.33 |  |
| 10 | 20999.67 | 18266.67 | 20966.67 |  |
| 100 | 202833.67 | 215333.33 | 143266.67 |  |
| 10000 | 4899266.67 | 4755733.33 | 4300300.00 |  |
| 1000000 | 455029833.33 | 303296700.00 | 328288566.67 |  |
| 10000000 | 24742063300.00 | 3996842033.33 | 4331857900.00 |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  | **Tiempo por algoritmo** | | |  |
| **n (longitud)** | QS | MS REC | MS ITER |  |
| 1 | 100.00 | 93.09 | 56.45 |  |
| 5 | 100.00 | 106.36 | 124.24 |  |
| 6 | 100.00 | 120.66 | 138.55 |  |
| 10 | 100.00 | 86.99 | 99.84 |  |
| 100 | 100.00 | 106.16 | 70.63 |  |
| 10000 | 100.00 | 97.07 | 87.77 |  |
| 1000000 | 100.00 | 66.65 | 72.15 |  |
| 10000000 | 100.00 | 16.15 | 17.51 |  |

Ahora, tomando como unidades base los tiempos del MergeSort Recursivo observamos lo siguiente:

**Tabla 4.**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Tiempo por algoritmo** | | |
| **n (longitud)** | QS | MS REC | MS ITER |
| 1 | 107.43 | 100.00 | 60.64 |
| 5 | 94.02 | 100.00 | 116.81 |
| 6 | 82.88 | 100.00 | 114.82 |
| 10 | 114.96 | 100.00 | 114.78 |
| 100 | 94.20 | 100.00 | 66.53 |
| 10000 | 103.02 | 100.00 | 90.42 |
| 1000000 | 150.03 | 100.00 | 108.24 |
| 10000000 | 619.04 | 100.00 | 108.38 |

Cuando un valor es menor que el valor del tiempo base, se entiende que le toma menos tiempo en procesar la misma cantidad de elementos. En estos casos, es fácil ver que el algoritmo de QuickSort hace honor a su nombre, y gana en tiempo a los algoritmos de MergeSort en varios casos. Aunque hay que considerar el hecho de que los algoritmos están probados por separado, y a causa del uso del método *Random* los elementos pueden estar distribuidos de distinta forma causando mayor variabilidad.

Por lo tanto, observemos el caso en que los algoritmos corren el mismo arreglo, para ello se utiliza el siguiente algoritmo:

int length = 10000000;

Integer[] datos = new Integer[length];

Integer[] datos2 = new Integer[length];

Integer[] datos3 = new Integer[length];

Random rand = new Random();

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

datos[i] = rand.nextInt(10000);

datos2[i] = datos[i];

datos3[i] = datos[i];

}

Así, se obtienen los siguientes datos:

**Tabla 5.**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Tiempo por algoritmo** | | |
| **n (longitud)** | QS | MS REC | MS ITER |
| 5 | 45866.67 | 7433.33 | 17266.67 |
| 6 | 15933.33 | 9766.67 | 21033.33 |
| 10 | 17766.67 | 13500.00 | 23166.67 |
| 100 | 163733.33 | 78033.33 | 85600.00 |
| 10000 | 5107366.67 | 11455300.00 | 3869600.00 |
| 1000000 | 446930166.67 | 244286166.67 | 319692233.33 |
| 10000000 | 25676032500.00 | 3535052333.33 | 4172942000.00 |
|  |  |  |  |
|  |  |  |  |
|  | **Tiempo por algoritmo relaciones** | | |
| **n (longitud)** | QS | MS REC | MS ITER |
| 5 | 617.04 | 100.00 | 232.29 |
| 6 | 163.14 | 100.00 | 215.36 |
| 10 | 131.60 | 100.00 | 171.60 |
| 100 | 209.82 | 100.00 | 109.70 |
| 10000 | 44.59 | 100.00 | 33.78 |
| 1000000 | 182.95 | 100.00 | 130.87 |
| 10000000 | 726.33 | 100.00 | 118.04 |

Con base en los datos suministrados por la tabla, es evidente que el algoritmo MergeSort supera en tiempo a cualquiera de los demás algoritmos una vez que marca un puntaje menor con respecto a los otros dos en todos excepto un caso. Curiosamente es en el punto en que la **n** es el mismo valor que el rango del método *Random*. Esto pareciera demostrar que el MergeSort funciona mejor de forma general ante un arreglo aleatorio; además, que, en ese caso específico, QuickSort ordenó el arreglo en menos de la mitad de tiempo. Pero es necesario evaluar los demás casos.

En el caso de arreglos ordenados ascendentemente el método *Random* queda inutilizable, por lo tanto, utilizaremos un algoritmo sencillo a través de un ciclo:

(…)

Random rand=new Random();

int x = rand.nextInt(length);

int y = rand.nextInt(x);

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

datos[i] = x; //nótese que en este caso el orden no cambiara por lo que se puede usar una sola variable de arreglo

x += y;

}

De este algoritmo se obtienen los siguientes datos:

**Tabla 6.**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Tiempo por algoritmo** | | |
| **n (longitud)** | QS | MS REC | MS ITER |
| 5 | 12766.67 | 6733.33 | 11666.67 |
| 6 | 15333.33 | 8533.33 | 15166.67 |
| 10 | 22833.33 | 17033.33 | 18266.67 |
| 100 | 198366.67 | 86433.33 | 91333.33 |
| 10000 | 23803833.33 | 5492533.33 | 7253200.00 |
| 1000000 | 15175276233.33 | 140346200.00 | 182485766.67 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  | **Tiempo por algoritmo** | | |
| **n (longitud)** | QS | MS REC | MS ITER |
| 5 | 189.60 | 100.00 | 173.27 |
| 6 | 179.69 | 100.00 | 177.73 |
| 10 | 134.05 | 100.00 | 107.24 |
| 100 | 229.50 | 100.00 | 105.67 |
| 10000 | 433.39 | 100.00 | 132.06 |
| 1000000 | 10812.74 | 100.00 | 130.03 |

Una vez más observamos que Merge Sort Recursivo supera en tiempo a los otros dos algoritmos. Además, se puede observar la gran diferencia que se genera con respecto a un arreglo ordenado para el QuickSort. La proporción es prácticamente cuadrática con respecto al del algoritmo MergeSort Recursivo.

En el caso de arreglos ordenados descendientemente se usará el mismo algoritmo, pero restando y se consiguen los datos siguientes:

**Tabla 7.**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Tiempo por algoritmo** | | |
| **n (longitud)** | QS | MS REC | MS ITER |
| 5 | 16633.33 | 8566.67 | 16200.00 |
| 6 | 13600.00 | 7733.33 | 22533.33 |
| 10 | 16500.00 | 13100.00 | 29266.67 |
| 100 | 528233.33 | 88666.67 | 93633.33 |
| 10000 | 201740300.00 | 11205566.67 | 3818700.00 |
| 1000000 | Stack overflow | 86303633.33 | 142332800.00 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  | **Tiempo por algoritmo** | | |
| **n (longitud)** | QS | MS REC | MS ITER |
| 5 | 194.16 | 100.00 | 189.11 |
| 6 | 175.86 | 100.00 | 291.38 |
| 10 | 125.95 | 100.00 | 223.41 |
| 100 | 595.75 | 100.00 | 105.60 |
| 10000 | 1800.36 | 100.00 | 34.08 |
| 1000000 | Stack overflow | 100.00 | 164.92 |

Analizando los datos de los tres casos podemos observar que QuickSort mantiene una similitud respecto a MergeSort en arreglos desordenados, incluso logrando ser más rápido por proporciones significativas.

Problema: Desafortunadamente, la medición de tiempo a través del uso de la función nanoTime() como se explicó en un inicio sufre la dificultad que al ser llamada sucesivamente pierde precisión por el desbordamiento numérico. Para comprobar este sesgo se cambió de lugar la llamada de los métodos, descubriendo que la posición afectaba a los periodos de tiempo marcados por el sistema o eso parece.

Así, y con la esperanza de no desacreditar el trabajo pasado se diseñó una última prueba unitaria para tres arreglo de **n** datos:

1. [60, 86, 70, 47, 68, 15, 2, -5, -47, 100, -200]

|  |  |  |  |
| --- | --- | --- | --- |
| n | QuickSort | MergeSort Iter | Merge Sort Rec |
| 11 | 16200 | 26200 | 19800 |

1. [60, 86, 70, 47, 68, 15, 2, -5, -47, 100, -200, 5, 489, 34, -75, 34, -65, 49, 138, 301, -853, 154, 0]

|  |  |  |  |
| --- | --- | --- | --- |
| n | QuickSort | MergeSort Iter | Merge Sort Rec |
| 23 | 21300 | 31200 | 35600 |

1. [60, 86, 70, 47, 68, 15, 2, -5, -47, 100, -200, 5, 489, 34, -75, 34, -65, 49, 138, 301, -853, 154, 0, 78,165,156,1548,95,0,549,7,6,64,81,-61,-91,500,461, -2,-23,-1,94,76,-48,-49,-47,35,-36,14,18]

|  |  |  |  |
| --- | --- | --- | --- |
| n | QuickSort | MergeSort Iter | Merge Sort Rec |
| 50 | 47100 | 55500 | 47500 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Gracias a esta última prueba, puede verse un comportamiento que, a menor **n**, el algoritmo de QuickSort se desenvuelve con mayor velocidad frente a los otros dos algoritmos como se había visto anteriormente, sobre todo ante arreglos desordenados; sin embargo, conforme crece la cantidad de datos o el arreglo se encuentra en mayor grado ordenado, el algoritmo de QuickSort pierde su efectividad en tiempo, y el método de MergeSort Recursivo lo sobrepasa.

Esto se debe a que el algoritmo MergeSort se comporta de la misma forma todo el tiempo independientemente de las posiciones de los datos, pues siempre divide el arreglo a la mitad, mientras que el QuickSort lo va dividiendo de acuerdo a la información contenida en ellos. Por esto se puede afirmar que, ante los peores casos, el algoritmo de QuickSort (como se vio en los arreglos ordenados e inversos) tiende a una complejidad de O(n2) en lugar de su característico O(n.log(n)). En cambio, ambos métodos de Merge Sort mantienen el comportamiento O(n.log(n)) en cualquier circunstancia.

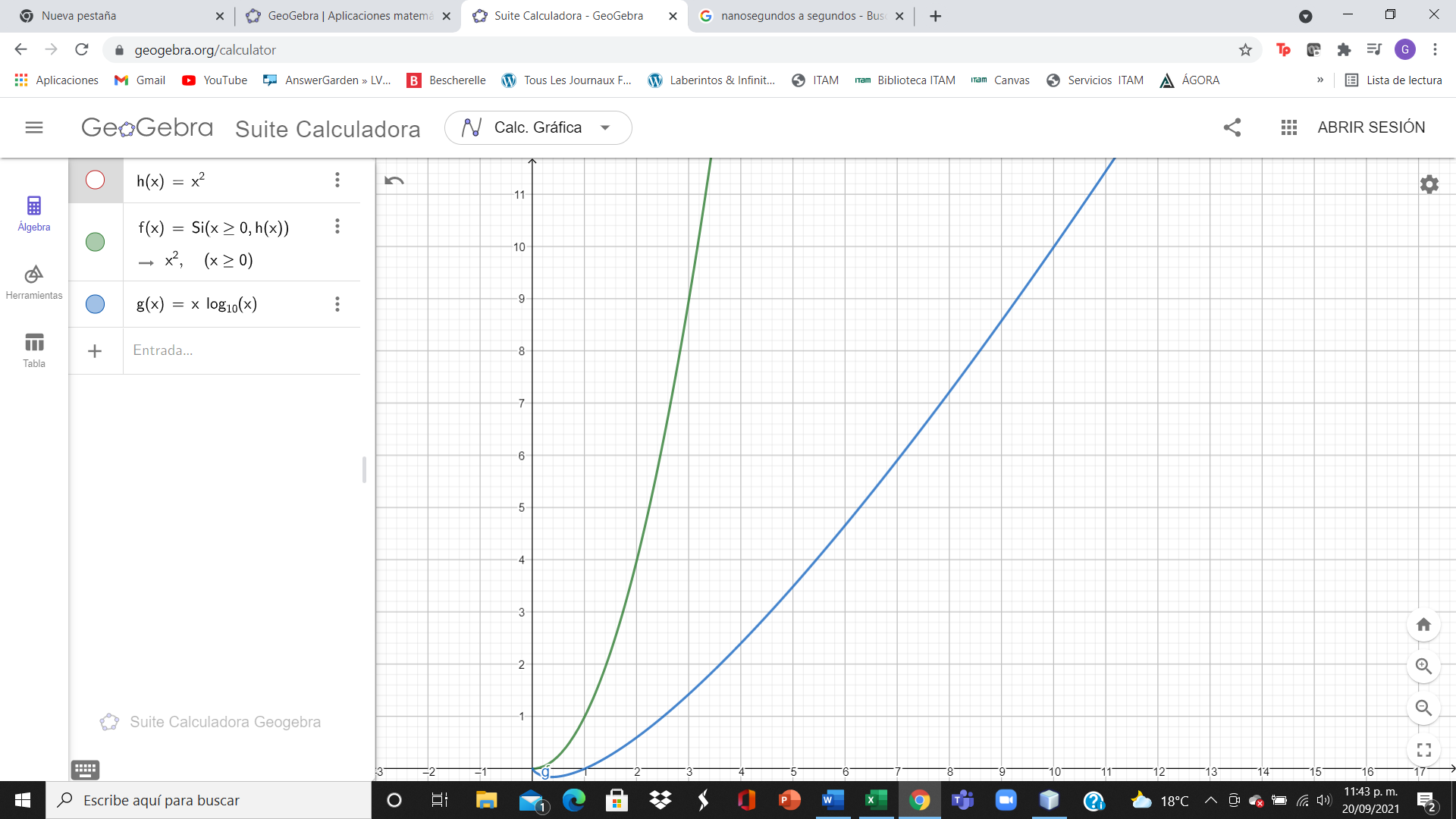
De hecho, podemos observar que en el algoritmo utilizado el número de operaciones aproximadamente está dado de la siguiente forma en el caso promedio:

|  |  |
| --- | --- |
| **Algoritmo** | **#operaciones** |
| QuickSort | (n+6) log(n) |
| MergeSort Rec | (2n+4) log(n)+2 |
| MergeSort Iter | n log(n-1) |

1. ¿Son asintóticamente iguales?

En estas gráficas puede verse claramente, que, en cualquiera de los casos, ante elementos de grandes magnitudes y en el caso de arreglos ordenados, el comportamiento de QuickSort es mucho mayor que el de MergeSort. Mas, como se observa en la primera gráfica, ante bajo número de datos, el QuickSort prueba su eficiencia en tiempo.

Este comportamiento puede verse con mayor claridad en las ecuaciones matemáticas aproximadas, donde la gráfica verde es el comportamiento de QuickSort y la gráfica azul el comportamiento de Merge Sort.



1. Además de tiempo, ¿cree usted que haya otra dimensión de eficiencia en la que uno sea mejor que otro?

Algo muy importante a considerar en el grado de eficiencia es el uso y aprovechamiento de la memoria utilizada, además de la facilidad de comunicación (o acceso entre las variables), en este sentido no es lo mismo hacer comparaciones de algoritmos de ordenamiento para variables de tipo entero que para variables de tipo double o String. Cuanta mayor sea la dificultad de comparación de las variables, igual será más complicado para el algoritmo desenvolverse de forma eficiente. Además, en el uso de objetos comparables, la comparación entre elementos resulta de llamadas a métodos diferentes que hacen mayor uso de memoria.

Este último punto se aplica de igual forma para la diferencia entre algoritmos recursivos e iterativos. Por un lado, es importante reconocer que hay algoritmos naturalmente recursivos, y cuya eficiencia se desenvuelve mejor des esta forma, pero en otros casos resulta en mayores complicaciones que pueden causar saturación de la memoria interna o exceso del tiempo utilizado para el procesamiento, sobre todo ante entradas de **n** elementos muy elevadas, como es el caso de QuickSort. Asimismo, es relevante considerar la utilización de variables temporales como hace MergeSort, el cual no solo guarda las instrucciones necesarias a llevar a cabo posteriormente (por recursividad), sino también acumula los arreglos temporales utilizados en la función *mezcla()*. En este sentido, el método de QuickSort resultaría más eficiente, puesto que no utiliza más que una variable de tipo T, en lugar de la creación de un espacio de arreglo completo.

Por último, considero que el factor más importante en la comparación de eficiencia es tomar en cuenta el uso que se le dará al algoritmo, el contexto en el que se trabaja y las posibilidades de entrada. Por ejemplo, si lo que se busca es ordenar elementos de arreglos que guardan los días de la semana (7) lo mejor sería usar QuickSort, pero si se piensa trabajar con la nómina de una compañía probablemente lo mejor sea usar MergeSort.

Nota: Investigando, descubrí que hacer uso del algoritmo de QuickSort en el que se elige el pivote de anera aleatoria, no solo evita los ataques, sino que mejora el rendimiento del método, pues es más complicado conseguir que se genere una asíntota del tipo O(n2).

Referencias:

Esponda, F. (2021) *Apuntes de Clase. Estructura de Datos Avanzadas.* ITAM. México.

Algoritmos:

package tareaordenamiento;

/\*\*

\*

\* @author Guillermo Arredondo Renero

\*/

public class Ordena <T extends Comparable<T>>{

//quickSort

public void quickSort(T[] datos){

quickSort(0, datos.length, datos);

}

private void quickSort(int inf, int sup, T[] datos){

if(sup-inf<=1)

return;

int pivote = particion(inf, sup-1, datos); // num operaciones: 6n=n

quickSort(inf,pivote,datos); // se llama n/2

quickSort(pivote+1,sup, datos); // se llama n/2

}

private int particion(int inf, int sup, T[] datos){

T temp;

while(inf < sup){ //num operaciones dentro de while: 6

int aux = inf+1;

if(datos[inf].compareTo(datos[aux])>0){

temp = datos[aux];

datos[aux] = datos[inf];

datos[inf] = temp;

inf++;

}

else{

temp = datos[sup];

datos[sup] = datos[aux];

datos[aux] = temp;

sup--;

}

}

return inf;

}

//mergeSort Iterativo

public void mergeSortIt(T[] datos){

int min = 0, max = datos.length-1;

int size = max-min;

for(int step = 1; step<=size; step = 2\*step)

for(int inicio = 0; inicio < max; inicio += 2\*step){

int pivote = inicio + step-1;

int fin = Integer.min(inicio + 2\*step - 1, max); // 1 operacion

mezcla(datos, inicio, pivote, fin); // 2n+4 aprox

}

}

//mergeSort Recursivo

public void mergeSort(T[] datos){

mergeSort(datos, 0, datos.length-1);

}

private void mergeSort(T[] datos, int min, int max){

if(min == max)

return;

int pivote = (min+max)/2;

mergeSort(datos, min, pivote);

mergeSort(datos, pivote+1, max);

mezcla(datos, min, pivote, max);

}

// mezcla

private void mezcla(T[] datos, int inicio, int mitad, int fin){

T[] temp = (T[]) new Comparable[fin-inicio+1];

int der = mitad+1, izq = inicio;

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

if(der<=fin)

if(izq<=mitad){

if(datos[izq].compareTo(datos[der])<0){

temp[i]=datos[izq++];

}

else{

temp[i]=datos[der++];

}

}

else

temp[i]=datos[der++];

else

temp[i]=datos[izq++];

}

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

datos[inicio+i]=temp[i];

}

}

//imprime

public void imprime(T[] datos) {

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

System.out.print(datos[i] + " ");

System.out.println();

}

}