Nombre: Alejandro Mena
Materia: Análisis de algoritmos (AA17II)

Comparación de tiempos de ejecución de algoritmos de ordenamiento

Para este proyecto se deberá generar 1.000.000 de enteros aleatorios de 10 cifras (similar a cédulas de identidad). Luego se los separará en las siguientes configuraciones:

- Configuración A: 100.000 arreglos de 10 elementos
- Configuración B: 10.000 arreglos de 100 elementos
- Configuración C: 1.000 arreglos de 1.000 elementos
- Configuración D: 100 arreglos de 10.000 elementos
- Configuración E: 10 arreglos de 100.000 elementos
- Configuración F: 1 arreglo de 1.000.000 elementos

Se usarán los algoritmos de ordenamiento:

- Bubble sort
- Shell sort
- Heap sort
- Quick sort
- Radix sort

Finalmente se deberá comparar los tiempos de ejecución para cada uno de los algoritmos en algún lenguaje de programación de preferencia, y reportar los resultados

Desarrollo

El lenguaje de programación a usar es JAVA. Las implementaciones fueran obtenidas de repositorios en Github.

Dado el requerimiento de generar números aleatorios de 10 cifras, dos de estas implementaciones fueron modificadas solamente en el tipo de dato del arreglo que recibían como argumento. Al inicio recibían tipo de dato int y se modificó para que trabajen con tipo de dato long. Las implementaciones modificadas fueron Radix Sort y Heap Sort.

El resto de implementaciones trabajaban con tipos de dato de referencia, en esos casos se construyó un arreglo paralelo, que almacenaba datos de la clase wrapper Long.

El procedimiento fue el siguiente:

1. Crear un arreglo de dos dimensiones, con su longitud determinada por variables para cambiar los valores dependiendo de cada configuración.
2. Iterar sobre los arreglos y llenarlos con números aleatorios de 10 cifras. Estos generados por una función construida.
3. Después de llenado cada arreglo, proceder a ordenarlo con un algoritmo de ordenamiento a la vez.
4. Registrar mediante funciones del lenguaje JAVA, el tiempo que tardo el algoritmo.
5. Hacer un sumatorio total de los tiempos de ejecución, del algoritmo de ordenamiento y así obtener el tiempo total que tarda el algoritmo en ordenar una configuración especifica.

A continuación, un ejemplo de segmento de código que ilustra el procedimiento realizado:

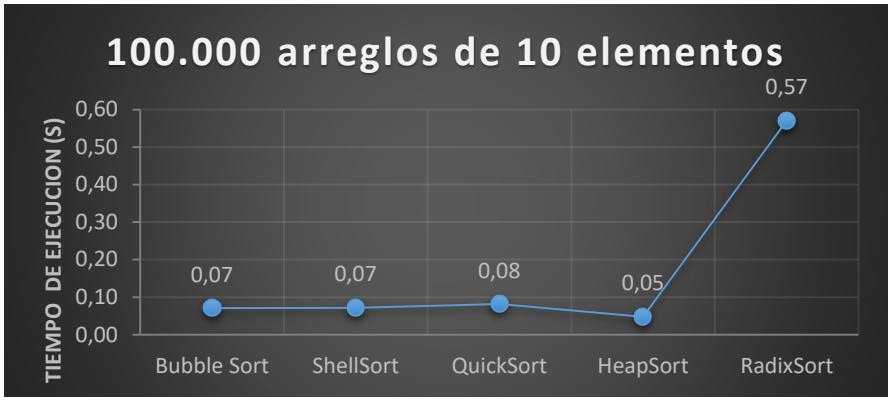
```
int k=1000,n=1000;
long[][] array=new long[k][n]; //arreglo de tamaño variable
long sum=0;
for(int i=0;i<k;i++){
        for(int j=0;j<n;j++){       //llenado del arreglo
            array[i][j]=rand_10dig();  //genera numero aleatorio de 10 cifras
        }
long time_1 = System.nanoTime();
RadixSort.radixSort(array[i]);  // ejecucion del algoritmo de ordenamiento
long time_2 = System.nanoTime();
long difference = time_2 - time_1; // registro del tiempo de ejecución
sum+=difference;  // sumatorio de tiempos de ejecución
 }
```

Análisis de tiempos de ejecución para cada configuración

Los gráficos a continuación contienen comparaciones de los algoritmos usados con cada configuración. El tiempo mostrado en los gráficos, es el tiempo total que tomo ordenar la configuración.

En el análisis, no se procedió creando los arreglos y aplicando los 5 algoritmos de ordenamiento a los mismos arreglos. Se probó cada algoritmo siempre con una configuración en específico, pero esta configuración siempre se generaba de nuevo de manera aleatoria y para cada algoritmo. Aun así, los datos si reflejan un comportamiento lógico.

Configuración A:



Para esta configuración de arreglos de 10 elementos radix sort tiene mayor tiempo de ejecución.

Se podría pensar que radix sort debería tener el menor tiempo de ejecución en comparación con quick sort y heap sort, ya que corre en tiempo lineal. Sin embargo, esto ocurre cuando el factor w en radix sort es menor que log n. Esto es:

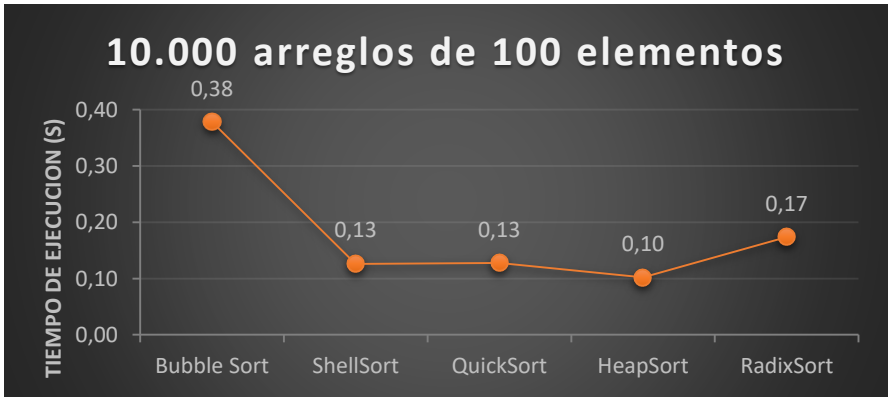
$$O(w\,n) \sim O(nlogn)$$

$$w\,n \;<\; n\,logn$$

$$w \;<\; log\,n$$

Por ahora esto no sucede, no obstante, se observará la ventaja de usar radix sort en algunas configuraciones más adelante.

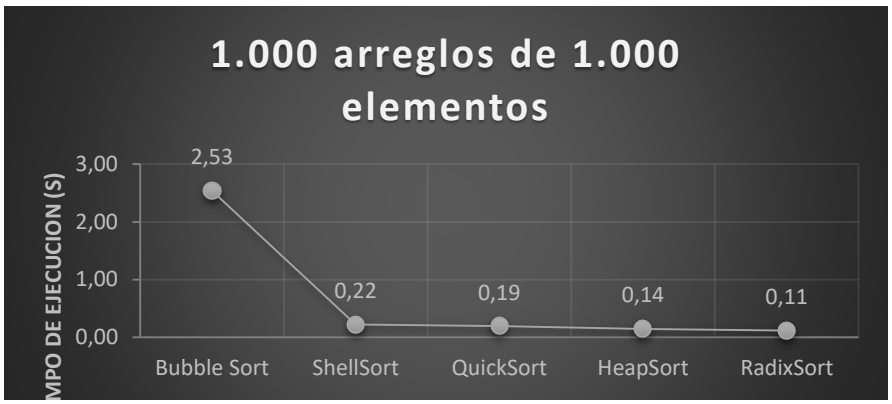
Configuración B:



Existe un cambio en la tendencia. Ahora bubble sort es el que tarda más. Esta tendencia ha de mantenerse dado su tiempo de ejecución $O(n^2)$ y debería suceder lo mismo con shell sort.

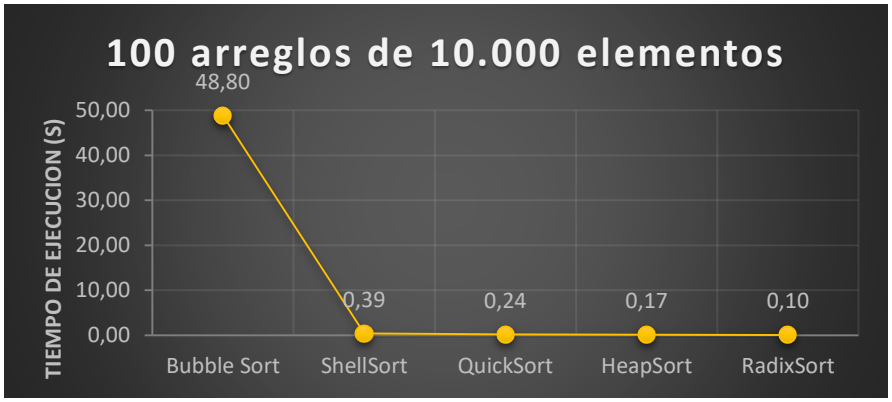
Mientras que quick sort y merge sort aumentaron muy poco, radix sort se acerca más a estos.

Configuración C:



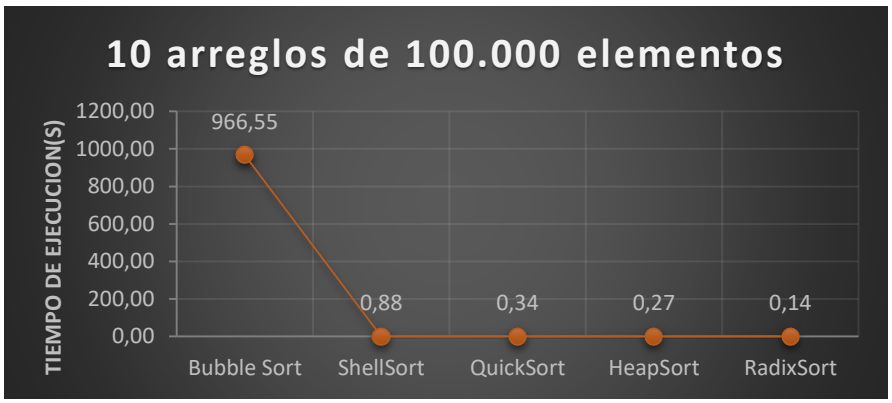
Como vemos, se mantiene la tendencia de que bubble sort tarda más. Al mismo tiempo shell sort se mantiene cerca de quick sort y heap sort, los cuales, de nuevo, aumentaron muy poco su tiempo de ejecución. Por otro lado, el tiempo de radix sort se mantiene debajo de los de quick sort y heap sort.

Configuración D:



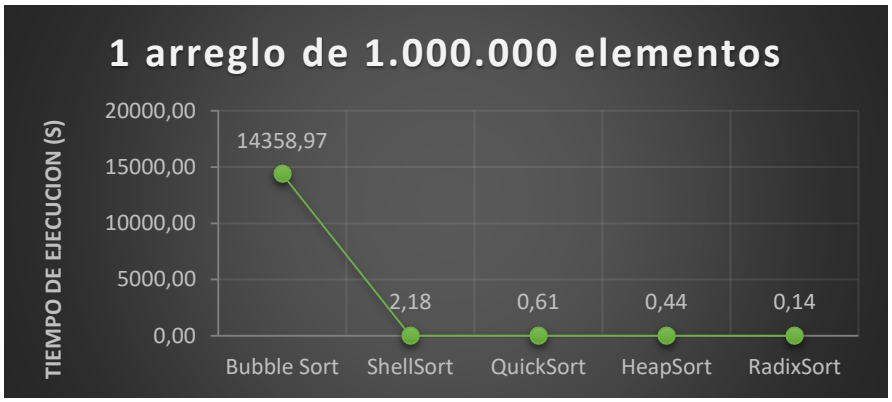
De nuevo se observa que bubble sort tiene mayor tiempo de ejecución. Por otra parte, shell sort no aumenta tanto como se esperaba, esto depende de la implementación y la secuencia de pasos escogida (en la implementación se aprecia una secuencia de pasos de N/3). Quick sort y heap sort se mantienen bajos y radix sort por debajo de estos.

Configuración E:



Bubble sort ahora tarda 966,55 segundos o 16.1 minutos. Los otros algoritmos aumentaron su tiempo de ejecución, pero no alcanzan a llegar a un segundo.

Configuración F:



En esta configuración el tiempo de ejecución de radix sort se mantiene aún por debajo de quick sort, heap sort y shellsort. Shell sort ahora toma 2,18 segundos y Bubble sort tarda 14358 segundos o aproximadamente 3,98 horas.

Conclusiones:

- Radix sort es el mejor para ordenar arreglos de números en comparación con quick sort y heap sort, siempre que el factor w (número de bits necesarios para representar el numero) sea menor que el factor log n.
- En general para elementos que se puedan comparar y con los que se desee un tiempo de ejecución aceptable en la mayoría de los casos, es mejor usar quick sort y heap sort.
- Bubble sort al igual que insertion sort, a pesar de tener tiempo $O(n^2)$ funcionan muy bien para arreglos pequeños.
- Shell sort puede tener un tiempo de ejecución menor que $O(n^2)$ si en la implementación se escoge la *secuencia de pasos* correcta.
- Dado que no se usó el mismo arreglo para las diferentes comparaciones, obtener los datos usando medias estadísticas podría haber aportado una mayor validez a los datos.

Bibliografía:

https://en.wikipedia.org/wiki/Radix_sort

https://en.wikipedia.org/wiki/Shellsort