Tarea 6:

Implementación y Análisis de algoritmos de ordenación MergeSort y RadixSort

Profesor: Benjamín Bustos.

Profesor Auxiliar: Boris Romero

Alumno: Jorge Gutiérrez

Fecha: 9 de Diciembre de 2015

Introducción

Uno de los mayores avances de la humanidad ha sido la creación de maquinas capaces de hacer tareas productivas de forma rápida y eficiente. Así la humanidad ha sido capaz de aumentar sus esperanzas de vida y la calidad de esta utilizando estas maquinas, tanto para la fabricación de alimentos, ropa , autos, documentos, maquinas, en fin, la automatización de la producción de bienes. Por otro lado estas maquinas además de estar presente en la producción de bienes están participando activamente en el proceso de organización de la vida en la sociedad. Es fácil notar que la gente usa aparatos electrónicos comúnmente y casi como si fuera algo normal .

Ahora la procedencia de todos estos artefactos tiene un ancestro en común y en si son una forma mas avanzada de antepasado capaz de procesar información millones de veces mas rápido que la anterior, así es como podemos relacionar el ábaco con la computadora y dado esta relación, y pensando que ambos tienen un origen común, entonces esto significa que ambas pueden resolver el mismo problema, tal que el computador puede resolverlo mas rápido que el ábaco. Si nos preguntamos cual es este problema nos daremos cuenta rápidamente que es. “Contar y ordenar”.

Así es como el humano para poder ser capaz de resolver este problema mediante computadoras a creado distintas formas de abordarlo, llamadas algoritmos.

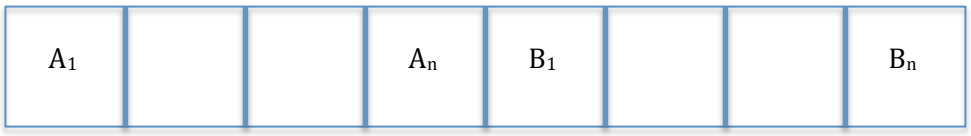
En este informe se probara una forma de ordenar elementos, en especifico números usando un método llamado MergeSort y RadixSort, ambos que se aplicaran sobre el ordenamiento de números, los cuales se pondrá a prueba y se discutirán sus resultados.

Diseño de la Solución

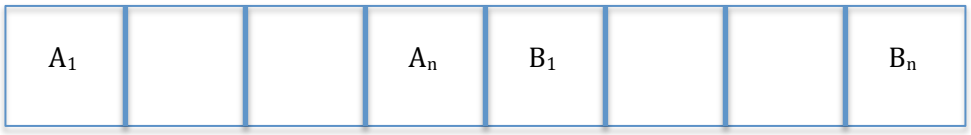
MergeSort:

La filosofía de la ordenación mediante MergeSort nos dice que si dos arreglos están ordenados, entonces se puede obtener un arreglo ordenado a partir de estos siguiendo las siguientes premisa

1) Si B_1 es mayor que A_n entonces unimos estos dos arreglos de la siguiente forma



2) Si B_1 es mayor que A_n entonces unimos estos dos arreglos de la siguiente forma



Así bajo estas premisas se puede obtener un arreglo ordenado de n arreglos, mas aun cuando para iniciar la ordenación se comienzan con n arreglos de tamaño 1 y se aplican estas simples reglas recursivamente sobre cada uno de los arreglos generados por su unión.
Notemos que siempre a partir del caso base podemos encontrar elementos que cumplan con las premisas mencionadas anteriormente.
Notemos que este algoritmo siempre divide cada sub problema en dos de la mitad del tamaño, lo que implica(según teorema maestro) que

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1) \Rightarrow T(n) \in \theta(n) = nlog(n)$$

RadixSort:

La premisa de este tipo de ordenamiento se basa en ordenar filas de una matriz, para ello se elige un numero como máximo del conjunto de los elementos, en valor absoluto, que se pueden ordenar, el cual en este caso es el numero $2^{36} \sim 10^{20}$.

Luego de tener nuestros limites claros, empezamos con la ordenación, se comienza comparando los últimos dígitos de cada numero y se ingresa cada numero con el mismo digito en un arreglo, generando así 10 arreglos como lo muestra la figura 4. Esto se obtiene dividiendo cada numero por una potencia de 10 desde $10^0 \rightarrow 10^{20}$ y luego obteniendo su modulo (%10)

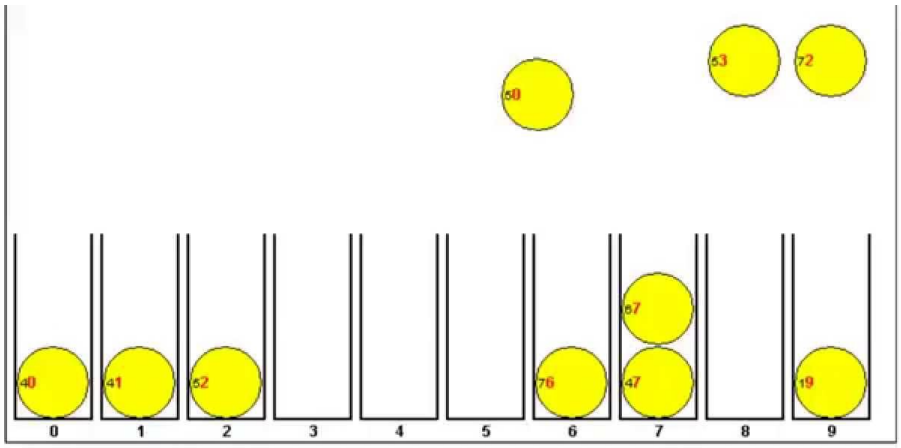


Figura 4: Primer paso de ordenación RadixSort

Luego se obtiene un arreglo obteniendo los antiguos elementos en el siguiente orden: desde el arreglo que corresponde a los con resto 0 hasta el arreglo que corresponde al 9

Así con este arreglo se continua comparando los dígitos, pero esta vez con el siguiente digito (aumentando en 1 el exponente de la potencia de 10) y se repite el mismo procedimiento hasta llegar al ultimo digito (llegando a 10^{20}).

Notemos que cada vez que vamos aumentando el exponente, los números van ingresando a esta nueva estructura de 10 arreglos ordenados ya por los procesos anteriormente nombrados.

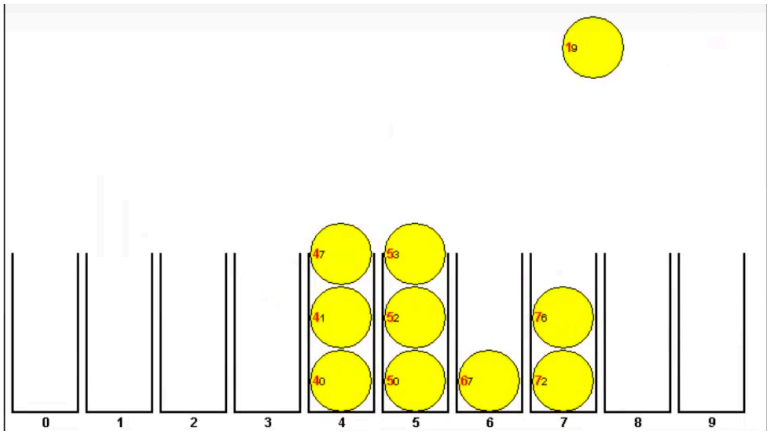


Figura 5: RadixSort en un paso mas adelante

Así, al final del proceso, obtendremos un arreglo ordenado.

Notemos también que la complejidad de este algoritmo recae en la cantidad de dígitos que revisamos , en nuestro caso 20 veces. Esto implica

$$T(n) = 20n \Rightarrow T(n) \in \theta(n) = wn$$

Implementación

MergeSort:

Para la implementación de este algoritmo, lo importante es la forma de compara los elementos, la cual se muestra a continuación.

```
int i = izq, m = mid;
        for (int j = 0; j < lenght; j++)  //Inicio de comparaciones de array
        {                 //Desde el array donde están estos subarray(temp)
            if (i == mid)              //si llegamos al inicio del arreglo
                temp[j] = lista[m++];  // se agrega al array lista
            else if (m == der)         //si llegamos al final del arreglo
                temp[j] = lista[i++];   // se agrega al array lista
            else if (lista[m]<lista[i]) // caso 1 (menor que  el inicial)
                temp[j] = lista[m++];    // se agrega al array lista
            else                       // caso 2 (mayor que  el inicial)
                temp[j] = lista[i++];  // se agrega al array lista
        }
        for (int j = 0; j < lenght; j++)
            lista [izq + j] = temp[j];  // Se agrega el ultimo elemento
```

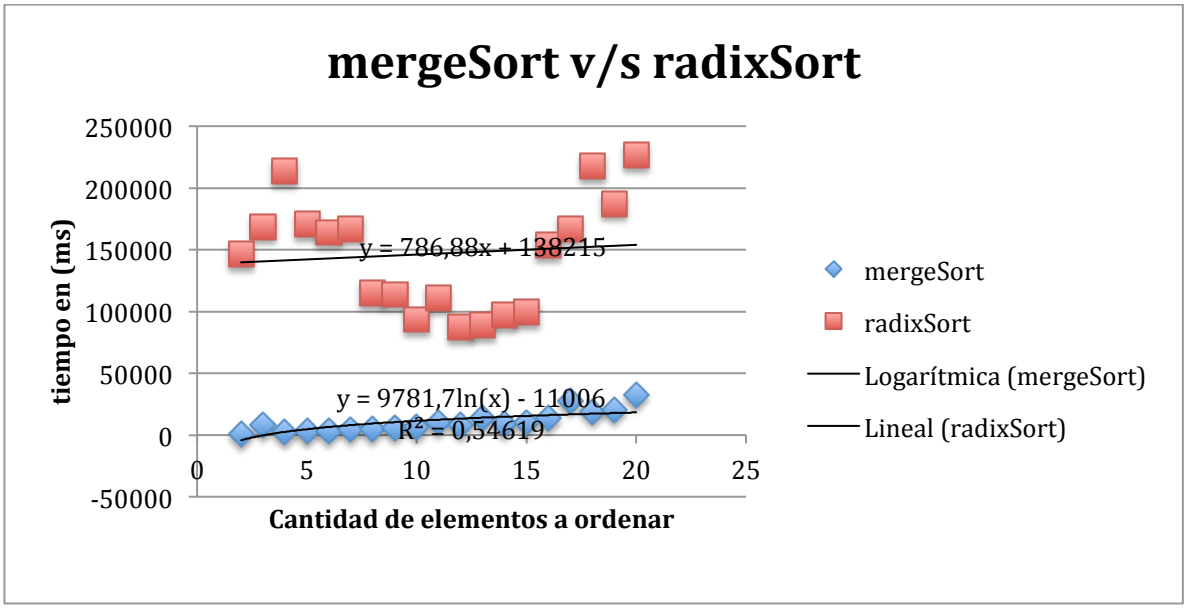
RadixSort:

Para resolver este algoritmo usando Java, se programo mediante iteración la comparación y mediante una estructura llamada RadixList (Anexo1) se fueron almacenando los elementos según su ultimo digito y se obtenían nuevamente.

```
      long compare = 0; //Comparador (10^compare)
      RadixList list = new RadixList(); // Nuestra Estructura
      for(int i=0;i<21;i++){
      compare=(long) Math.pow(10, i); //calculo de comparador (10^compare)
            for (long j:a){  // Para cada elemento
                  list.add((int)((j/compare)%10), j); //Se agrega en nuestra
estructura según su ultimo digito
                  }
            a=list.get(); // Se obitene nuevamente la lista ordenada por el
ultimo digito
            list= new RadixList(); //Refresca la estructura
            }
```

Resultados y Conclusiones

Para esta tarea se creo una función que comparaba la eficiencia de cada uno de estos algoritmos, los resultados se pueden ver en el siguiente grafico.



A pesar de que no se observa claramente, se nota que mergeSort se ajusta muy bien a la escala logarítmica, lo cual corresponde con la estimación presentada anteriormente. Por otro lado, y muy poco claro se observa que radixSort tiene un crecimiento mas pronunciado y eso podemos decir que es debido a el factor w presentado anteriormente o por la implementación de un divisor que clasifica la entrada según si es mayor o menor que 0.

1) Estructura RadixSort:

```
public class RadixList {
     private LongList[] rList=new LongList[10];
     public int size=0;

     public RadixList(){
          for(int i=0;i<10;i++){
               this.rList[i]= new LongList();
               }
     }

     public void add(int i,long element){
          this.rList[i].add(element);
          size+=1;
     }

     public long[] get(){
          LongList list = new LongList();
          int cont=0;
          for(int i=0;i<10;i++){
               for(long j:rList[i].get()){
                    list.add(j);
               }
          }
          return list.get();
     }
     }
```