FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS UNIVERSIDAD DE CHILE

Tarea 5: Huffman Sort

Autor: Vanessa Gaete
Correo: naan.u.285@gmail.com

Profesor: Jeremy Barbay
Auxiliares: Cristóbal Muñoz
Daniela Campos
Sven Reisenegger
Bernardo Subercaseaux
Curso: CC3001

Fecha de entrega: 18 de Diciembre de 2018
Santiago, Chile

Índice de Contenidos

1. Introducción

El problema a resolver en esta tarea consiste en crear un programa en Java que implemente tres métodos de ordenación: MergenSort (bottom up),Adaptive Merge Sort y Huffman Sort, y que entregue la cantidad de comparaciones que hace cada uno para ordenar un arreglo entregado en el input.

El objetivo, aparte de entender el funcionamiento e implementación de dichos métodos, es conocer la eficiencia de cada uno mediante la cantidad de comparaciones.

Lo que debe entregar el programa como output es una línea que contenga la cantidad de runs, la cantidad de comparaciones de MergeSort, luego las de Adaptive MergeSort y por último las de Huffman Sort.

Los runs corresponden a los subarreglos que ya están ordenados dentro de un arreglo, Adaptive Merge Sort y Huffman Sort se aprovechan de ellos para disminuir la cantidad de comparaciones realizadas.

Para resolver este problema se hizo un código con cuatro métodos principales. Uno arma los runs comparando entre cada elemento del arreglo entregado. Otro implementa Merge Sort de manera iterativa y que suma las comparaciones hechas en el proceso de merge. El que implementa Adaptive Merge Sort, que a partir de un arreglo con los runs ordena iterativamente los que se crearon hace más tiempo mergeando de izq a derecha y dejando el arreglo resultante al final del arreglo de runs. Y finalmente el que implementa Huffman Sort, que utiliza una cola de prioridad basada en el tamaño de los runs, específicamente minHeap, para guardar los runs y extraer iterativamente los más pequeños para aplicarles merge.

No quedó muy claro en qué forma y orden se debían elegir los arreglos mínimos para hacer el merge (será analizado más adelante), lo cual acaba afectando levemente la cantidad de comparaciones obtenidas. Así, los resultados obtenidos varían un poco, en algunos casos, con respecto a los presentados en afeed, razón por la que se prefirió entregar los dos códigos y hacer un análisis de ambos. Cabe notar que la variación presentada solo se genera en las comparaciones correspondientes a las de Huffman Merge Sort.

2. Diseño de la solución

El algoritmo fue diseñado con ArrayLists para todos sus métodos, pues se considera más fácil de trabajar que un arreglo normal, por ejemplo, se puede insertar en la posición que sea sin necesidad de correr 'a mano' todo el resto de elementos (como en un array).

Los métodos fueron diseñados de tal forma que ninguno ocupa recursión, la mayoría se compone de bucles que recorren el arreglo iterativamente. Además ninguno de los métodos de ordenación ordena el arreglo entregado, sino el único método que haría comparaciones sería el primero. Para esto se trabaja con un arreglo que funciona solo dentro de cada método. Se utilizaron algunas bases de códigos obtenidas de auxiliares.

Para entregar la cantidad de comparaciones para hacer los runs simplemente se sumó $n-1$ a cada método (con n el tamaño del arreglo), pues si o si debe recorrerse el arreglo entero con comparaciones.

Para hacer el método de Huffman Sort se buscó una forma eficiente de encontrar los runs mínimos, si bien no se cuentan las comparaciones para ordenar el arreglo, se quiso elegir uno que no demorase tanto para optimizar el programa. Así, se implementó el algoritmo entregado en clases, que ocupa un minHeap basado en el tamaño de los runs. Para esto se creó la clase "$MinHeap$" con los métodos necesarios para poder insertar y extraer en un heap, que son las funciones utilizadas en "$HuffmanSort$". La base de esta clase se obtuvo de `https://gist.github.com/flexelem/70b120ac9bf2965f419f`, sin embargo se cambiaron algunas características para que el minHeap ordene ArrayLists según su tamaño.

Para este método de ordenamiento, la idea es crear el arreglo de los runs e insertar cada uno en el heap, luego se extraen dos mínimos para mergearlos e insertar el resultado en el heap. Como insertar y extraer (o eliminar) en un heap demora O(log n) en el peor caso, se puede notar que será eficiente implementar este método. Así, para insertar cada run en el heap se ocupa O(nlogn), con n la cantidad de runs. Para extraer los mínimos, se tiene que se extraen dos en cada iteración y se inserta uno solo (el resultado del merge), con lo que el arreglo disminuye en uno su tamaño con cada iteración, así se extraerá elementos $2(n-1)$ veces, demorando O((n-1)logn) en este proceso. Finalmente se inserta el arreglo con los runs mergeados en el heap, esto se hará $n-1$ veces con lo que tomará O((n-1)logn). Por lo tanto todo este proceso demora O(2(n-1)logn+nlogn))=O((3n-2)logn). Lo que demorará Huffman Sort será el tiempo mencionado anteriormente más el tiempo que se demora en hacer el merge. Cabe notar que n es constante para el tiempo de HuffmanSort, pues corresponde a la cantidad de runs, lo que no depende del largo del arreglo entregado al método, es por esto que no se cuentan las comparaciones hechas para encontrar los runs mínimos en el total de comparaciones de Huffman Sort.

Como ya se dijo, este método presenta diferencias con algunos de los valores entregados por afeed, esto puede deberse a la forma en que se guardan los runs después de haberlos mergeado, la cual no fue especificada en la tarea. Sin embargo se tuvo cuidado de mantener la propiedad de estabilidad de orden en el método "$merge$", es decir, cuando se tiene un empate en los elementos de dos arreglos se elige el del arreglo de la izquierda primero. Por otro lado se mezclan siempre los dos arreglos más pequeños.

Se ocupa "$merge$" como método auxiliar para hacer la mezcla que será necesaria en Merge Sort, Adaptive Merge Sort y Huffman Sort. Dicho método compara iterativamente los elementos de dos arreglos ordenados para ir mezclándolos en otro inicialmente vacío y que también queda ordenado, guardando el elemento más pequeño entre los dos comparados. Además de esto, cuenta la cantidad de comparaciones hechas para lograr lo anterior. Si se acaba uno de los arreglos, se agregan todos los elementos del otro, sin necesidad de hacer comparaciones. Se supuso que los arreglos entregados están siempre ordenados, lo que no afecta al resultado, pues en los demás métodos solo se le entregan runs, o arreglos que ya fueron mergeados.

Los arreglos que son mergeados ya están ordenados, así, se puede asegurar que los elementos que se comparan con cada iteración son los mínimos de cada arreglo y que por lo tanto el mínimo entre ambos es el mínimo

entre todos los elementos que quedan por comparar. Como en cada iteración se van eligiendo los mínimos y ubicándolos de izquierda a derecha en el nuevo arreglo está siempre ordenado y su último elemento es siempre menor o igual a los que quedan en los arreglos que están siendo mergeados. Por lo tanto el invariante es que para la iteración n, se tendrá un arreglo con n-1 elementos ordenados. El tiempo en el mejor caso es O(1), ocurre si uno de los arreglos tiene largo 1. El tiempo en el peor caso es O(n-1), cuando ninguno de los arreglos se acaba mucho antes y se deben hacer todas las comparaciones (menos una, si o si un arreglo se acaba antes).

Para obtener las comparaciones de "*mergeSort*", se implementa el método bottom-up, que no utiliza recursión. Se parte dividiendo el arreglo en subarreglos de largo uno que se mergean de a dos, luego se duplica el tamaño de los subarreglos y se vuelven a mergear, asi hasta que el tamaño es igual al tamaño del arreglo, donde se detiene, pues ya está ordenado. Para lograr lo anterior se utilizan dos "for" anidados y el método *merge*. El invariante de este algoritmo se basa también en el invariante de "*merge*", como se parte de arreglos de largo uno están ordenados, al aplicar merge, se creará otro arreglo ordenado más grande que será mergeado con otro ordenado y así sucesivamente. Por lo tanto el invariante es que después de la iteración n, todos los subarreglos de tamaño 2^n están ordenados. El tiempo que demora es O(n log n).

Para el método "*AdaptiveMS*" se pide mezclar los arreglos por orden de antiguedad, siendo los más antiguos lo que están a la izquierda, pues se crearon primero. El diseño que se implementó se basa en ir mergeando los dos primeros runs del arreglo entregado hasta que se llegue a un solo run. Para esto se van eliminando los runs que ya fueron mezclados y añadiendo el arreglo resultante del merge al final (pues este es un arreglo nuevo). Como se quiere entregar la cantidad de comparaciones y mezclar dos runs, se vuelve a ocupar "*merge*". Este proceso demora un tiempo O(n(1+log(p))), con p el número de runs. Su invariante se basa en el de "*merge*", como se entregan runs a merge, se le entregan arreglos ordenados, y así, los runs del arreglo original van siendo reemplazados por un arreglo ordenado mucho más grande hasta que solo queda uno ya completamente ordenado.

Para realizar este código se supuso que el usuario ingresa siempre un input válido, es decir, que cada instrucción estuviese compuesta únicamente de enteros separados por un espacio entre cada uno.

3. Implementación

A continuación se presentan los métodos principales y su implementación.

El cuerpo principal de merge es el siguiente:

```
        int i = 0;
        int j = 0;
        int count=0;


        while (i < A.size() && j < B.size()) {
            if (A.get(i) <= B.get(j)) {
                R.add(A.get(i));
                i += 1;
                count += 1;
            } else {
                R.add(B.get(j));
                j += 1;
                count += 1;
            }
        }
        while (i < A.size()) {
            R.add(A.get(i));
            i += 1;
        }
        while (j < B.size()) {
            R.add(B.get(j));
            j += 1;
        }

        return count;
    }
```

Se crean dos punteros, uno para cada arreglo, parten desde el indice 0. Mientras estos punteros sean menores al tamaño de su arreglo (es decir mientras realmente apunten a algo que existe en el arreglo), se agrega el menor de los elementos a los que apuntan a un nuevo arreglo a R(resultado), moviendo ese puntero al siguiente elemento y sumando uno a la cuenta, pues se hizo una comparación. Si uno de los punteros ya recorrió todo su arreglo, los elementos del otro se agregan en orden a R sin hacer comparaciones. Se retorna la cuenta.

Adaptive Merge Sort, cuerpo principal:

```
        while (A.size() > 1) {
            ArrayList<Integer> Ai = new ArrayList<>();
            count+=merge(A.get(0),A.get(1),Ai);
            A.remove(0);
            A.remove(0);
            A.add(Ai);
        }return count;
    }
```

Se hace un arreglo con los runs (A) del arreglo a ordenar. Mientras tenga más de un elemento, quedan runs por ordenar, se hace un nuevo arreglo Ai en el que se hará la ordenación para no arreglar el que se entrega a los demás métodos. Se hace merge de los dos primeros arreglos (los más antiguos), se remueven los arreglos mergeados y se añade el resultado de la mezcla. Además se van sumando la cantidad de comparaciones hechas por merge en cada iteración. Se retorna dicha cuenta.

Merge Sort:

```
        int n=A.size();
        int size;
        int ini;
```

```
        int c=0;

        //Se van haciendo subarreglos desde tamano uno. El tama o se va duplicando.
        for(size = 1; size < n; size = size*2){
            //Se van mergeando los subarreglos de a dos.
            for(ini = 0; ini < n-1; ini += 2*size){

                int fin_primero = ini + size-1;
                int fin_segundo = Math.min(n-1, fin_primero+size);

                //No se utiliza el mismo merge de antes, pues aca no se hacen nuevos arreglos, se gurdan los punteros
                c+=merge(A, ini, fin_primero, fin_segundo);
            }
        }return c;
    }
```

Primero se traspasan todos los elementos del arreglo a uno nuevo para no ordenar del entregado. El for anidado va mergeando los subarreglos de a pares, esto se hace primero para subarreglos de tamaño uno, luego para subarreglos de tamaño dos, y asi sucesivamente se va duplicando el tamaño de los subarreglos hasta que ya se ordenó todo, esto último se hace con el for que contiene al primero mencionado. El "*merge*" que se aplica en esta ocasión es un poco distinto al que ya se había mencionado, pues en este caso no se entregan arreglos, sino punteros, fuera de eso funciona de la misma manera. La cantidad de comparaciones será la suma total de las comparaciones hechas por cada "*merge*", se retorna la cuenta.

Huffman Sort:

```
    public static int HuffmanSort(ArrayList<Integer> a){
        ArrayList<ArrayList<Integer>> A = runs(a);
        MinHeap H= new MinHeap();

        //Cada run se agrega al minHeap. El heap se ordena segun los tamanos de los runs
        for(int i=0;i<A.size();i++){
            H.insert(A.get(i));
        }
        int count=0;

        while(H.tamano()>1){
            //El arreglo R es el que contendrá el resultado del merge.
            ArrayList<Integer> R = new ArrayList<>();

            //Se extraen los dos runs mas cortos del heap.
            ArrayList<Integer> firtsmin = H.extractMin();
            ArrayList<Integer> scndmin = H.extractMin();

            count+=merge(firtsmin,scndmin,R);

            //Se anade al heap.
            H.insert(R);

        }return count;
    }
```

Se crea un arreglo (A) con los runs, y un minHeap. Primero se añaden todos los runs al minHeap, que los ordenará por tamaño. Luego se extraen dos elementos del minHeap (se obtienen los dos runs de menor tamaño), se hace merge con ambos y se añade el resultado al Heap. Las comparaciones serán iguales a la suma total de comparaciones hechas por "*merge*". Se retorna la cantidad de comparaciones.

4. Resultados y Conclusiones

rge Se probaron los casos entregados en la tarea. Primero se muestran los de "*Código*2":

CASO 1	RESULTADO	RESULTADO ESPERADO
10 3 5 9	2 5 6 6	2 5 6 6

CASO 2	RESULTADO	RESULTADO ESPERADO
4 9 5 6	2 5 6 6	2 5 6 6

CASO 3	RESULTADO	RESULTADO ESPERADO
10 6 6 6	2 5 6 6	2 5 6 6

CASO 4	RESULTADO	RESULTADO ESPERADO
1 6 7 9	1 4 3 3	1 4 3 3

CASO 5	RESULTADO	RESULTADO ESPERADO
5 8 9 9	1 4 3 3	1 4 3 3

CASO 6	RESULTADO 1	RESULTADO ESPERADO
49 6 17 2 50 11 12 27 32 38 49 1 11 25 37 31	6 47 49 47	6 47 49 47

CASO 7	RESULTADO 1	RESULTADO ESPERADO
19 27 6 17 20 26 27 3 31 35 35 40 45 7 8 25	4 43 36 39	4 43 36 39

En esta tarea se pudo llevar a la práctica la implementación de distintos métodos de ordenación basados en merge, permitiendo estudiar la cantidad de comparaciones que hace cada uno para obtener un arreglo ordenado. Como conclusión se puede notar, basándose en los resultados presentados y otros vistos en afeed, que para arreglos de mayor tamaño los algoritmos más eficientes son Huffman Sort y Adaptive Merge Sort. Sin embargo, la diferencia entre los tres métodos comienza a notarse más en los arreglos de tamaño cercano a 200, obteniendo diferencias de 150 comparaciones entre Merge Sort y los otros dos. Por otra parte, se puede atribuir la diferencia entre el método de Huffman Sort implementado y los casos de afeed, a las distintas formas de hacer las comparaciones y de ir guardando los runs mergeados, sin embargo se tuvo cuidado con respetar la propiedad de estabilidad del orden y de mergear siempre los dos menores arreglos.

5. Código Fuente

El siguiente es el código completo que se implementó para resolver el problema de la tarea.

```
import java.util.*;

public class Main {

    //Crea un arreglo con los runs de un arreglo
    public static ArrayList<ArrayList<Integer>> runs(ArrayList<Integer> arr){
        ArrayList<ArrayList<Integer>> runs=new ArrayList<>();

        int fin=1;
        int i=0;
        int inicio=0;
        //Se recorre el arreglo hasta el penúltimo elemento comparando el elemento actual con el siguiente.
        while(i<arr.size()-1){
            //Si el actual es mayor se hace el run correspondiente.
            if (arr.get(i)>arr.get(i+1)){
                runs.add(new ArrayList<>(arr.subList(inicio,fin)));
                i+=1;
                inicio=fin;
                fin+=1;
            //Sino, se sigue iterando
            }else{
                i+=1;
                fin+=1;
            }
        }//Se a ade el run que falta (recordar que el arreglo no se recorrió completo)
        runs.add(new ArrayList<>(arr.subList(inicio,fin)));
        return runs;
    }


    public static int merge(ArrayList<Integer> A,ArrayList<Integer> B,ArrayList<Integer> R){
    //recibe dos arreglos que serán mergeados:A y B. Y uno que corresponde al arreglo que se utiliza dentro de los
    //métodos para no ordenar el arreglo real, este es al que se le van anadiendo los elementos.
        int i = 0;
        int j = 0;
        int count=0;


        while (i < A.size() && j < B.size()) {
            if (A.get(i) <= B.get(j)) {
                R.add(A.get(i));
                i += 1;
                count += 1;
            } else {
                R.add(B.get(j));
                j += 1;
                count += 1;
            }
        }
        while (i < A.size()) {
            R.add(A.get(i));
            i += 1;
        }
        while (j < B.size()) {
            R.add(B.get(j));
            j += 1;
        }

        return count;
```

```
60      }
61
62      public static int AdaptiveMS(ArrayList<Integer> a) {
63          ArrayList<ArrayList<Integer>> A = runs(a);
64          int count = 0;
65
66          //Mientras el arreglo de runs tenga mas de un elemento, se mergean los dos primeros
    runs y se anade el
67          //resultado al final. Se borran los que fueron mergeados.
68          while (A.size() > 1) {
69              ArrayList<Integer> Ai = new ArrayList<>();
70              count+=merge(A.get(0),A.get(1),Ai);
71              A.remove(0);
72              A.remove(0);
73              A.add(Ai);
74          }return count;
75      }
76
77
78      public static int mergeSort(ArrayList<Integer> arr){
79          ArrayList<Integer> A = new ArrayList<>();
80          //Se traspasan todos los elementos a uno nuevo para no ordenar el ingresado.
81          for (int i=0;i<arr.size();i++){
82              A.add(arr.get(i));
83          }
84
85          int n=A.size();
86          int size;
87          int ini;
88          int c=0;
89
90          //Se van haciendo subarreglos desde tamano uno. El tama o se va duplicando.
91          for(size = 1; size < n; size = size*2){
92              //Se van mergeando los subarreglos de a dos.
93              for(ini = 0; ini < n-1; ini += 2*size){
94
95                  int fin_primero = ini + size-1;
96                  int fin_segundo = Math.min(n-1, fin_primero+size);
97
98                  //No se utiliza el mismo merge de antes, pues aca no se hacen nuevos
    arreglos, se gurdan los punteros
99                  c+=merge(A, ini, fin_primero, fin_segundo);
100             }
101         }return c;
102     }
103
104     //Este metodo es similar al merge anterior solo que en vez de recibir arreglos recibe
    punteros. El arreglo A es
105     //donde se insertaran los runs mergeados.
106     public static int merge(ArrayList<Integer> A, int ini, int fin_primero, int fin_segundo)
    {
107
108         int n1 = fin_primero - ini + 1;
109         int n2 = fin_segundo - fin_primero;
110
111         ArrayList<Integer> aux1 = new ArrayList<>();
112         ArrayList<Integer> aux2 = new ArrayList<>();
113
114
115         //Se traspasan todos los elementos a unos nuevos arreglos para poder mergearlos.
116         for (int i=0; i < n1; i++ ) {
117             aux1.add(A.get(i+ini));
118         }
119         for (int i=0; i < n2; i++ ) {
120             aux2.add(A.get(i+fin_primero+1));
121         }
```

```
122
123         int i=0, j=0;
124         int k=ini;
125         int c=0;
126         while(i < n1 && j < n2){
127
128             if(aux1.get(i)<=aux2.get(j)){
129                 A.remove(k);
130                 A.add(k,aux1.get(i));
131                 i++;
132
133             }
134             else{
135                 A.remove(k);
136                 A.add(k,aux2.get(j));
137                 j++;
138             }
139             c+=1;
140             k++;
141         }
142         while(i < n1){
143             A.remove(k);
144             A.add(k,aux1.get(i));
145             i++;
146             k++;
147         }
148
149         while(j < n2){
150             A.remove(k);
151             A.add(k,aux2.get(j));
152             j++;
153             k++;
154         }
155
156         return c;
157     }
158
159
160
161
162     public static int HuffmanSort(ArrayList<Integer> a){
163         ArrayList<ArrayList<Integer>> A = runs(a);
164         MinHeap H= new MinHeap();
165
166         //Cada run se agrega al minHeap. El heap se ordena segun los tamanos de los runs
167         for(int i=0;i<A.size();i++){
168             H.insert(A.get(i));
169         }
170         int count=0;
171
172         while(H.tamano()>1){
173             //El arreglo R es el que contendrá el resultado del merge.
174             ArrayList<Integer> R = new ArrayList<>();
175
176             //Se extraen los dos runs mas cortos del heap.
177             ArrayList<Integer> firtsmin = H.extractMin();
178             ArrayList<Integer> scndmin = H.extractMin();
179
180             count+=merge(firtsmin,scndmin,R);
181
182             //Se anade al heap.
183             H.insert(R);
184
185         }return count;
186     }
187
```

```
188
189     public static void main(String[] args) {
190         // Se crea el objeto scanner que nos permite leer el input del usuario.
191         Scanner in= new Scanner(System.in);
192         while (in.hasNextLine()) {
193             String linea = in.nextLine();
194             // Se crea un arreglo de strings que guarde cada elemento entregado en el input
    separado por los espacios.
195             String[] a = linea.split(" ");
196             ArrayList<Integer> arreglo= new ArrayList<>();
197             for (int i=0; i<a.length; i++){
198                 arreglo.add(Integer.parseInt(a[i]));
199             }
200             System.out.println(runs(arreglo).size()+" "+Integer.toString(mergeSort(arreglo))
    +" "+Integer.toString(AdaptiveMS(arreglo)+arreglo.size()-1)+" "+ Integer.toString(
    HuffmanSort(arreglo)+arreglo.size()-1));
201             //System.out.println(AdaptiveMSH(runs(arreglo)).size());
202         }
203     }
204 }
205
206 class MinHeap {
207     //El heap sera un ArrayList que contendrá otros Arrays con un run en la segunda posicion
     y su tamano en la primera.
208     //por ejemplo [[[3] [1 2 3]] [[4] [1 1 3 4]]]
209     private ArrayList<ArrayList<Integer >> list;
210
211     public MinHeap() {
212
213         this.list = new ArrayList<>();
214     }
215
216     public void insert(ArrayList<Integer> item) {
217
218         list.add(item);
219         int i = list.size() - 1;
220         int parent = parent(i);
221
222         //Para insertar se comparan los tamanos de los runs.
223         while (parent != i && list.get(i).size() < list.get(parent).size()) {
224
225             swap(i, parent);
226             i = parent;
227             parent = parent(i);
228         }
229     }
230
231     //retorna el elemento minimo y lo borra.
232     public ArrayList<Integer>  extractMin() {
233
234         if (list.size() == 0) {
235
236             throw new IllegalStateException("MinHeap is EMPTY");
237
238          //Si tiene un elemento solo se debe extraer y borrarlo
239         } else if (list.size() == 1) {
240             ArrayList<Integer> min = list.remove(0);
241             return min;
242         }
243
244         // Se saca el elemento de la raiz (minimo), se remueve el ultimo elemento y se
    inserta en la raíz.
245         ArrayList<Integer>  min = list.get(0);
246         ArrayList<Integer> lastItem = list.remove(list.size() - 1);
247         list.set(0, lastItem);
248
```

```
249         // Se hace bubble down hasta que llegue a la posición correcta. Manteniendo las
    propiedades de un Heap.
250         minHeapify(0);
251 
252         // Se retorna el minimo
253         return min;
254     }
255 
256 
257     private void minHeapify(int i) {
258 
259         int left = left(i);
260         int right = right(i);
261         int smallest;
262 
263         //Se encuentra el run de tama o mas peque o entre el nodo y sus hijos
264         if (left <= list.size() - 1 && list.get(left).size() <= list.get(i).size()) {
265             smallest = left;
266         } else {
267             smallest = i;
268         }
269 
270         if (right <= list.size() - 1 && list.get(right).size() <= list.get(smallest).size())
     {
271             smallest = right;
272         }
273 
274         //Si la llave mas peque a no es la llave actual, se hace bubble-down
275         if (smallest != i) {
276 
277             swap(i, smallest);
278             minHeapify(smallest);
279         }
280 
281     }
282 
283     //intercambia los nodos i y padre de posición
284     private void swap(int i, int parent) {
285 
286         ArrayList<Integer> temp = list.get(parent);
287         list.set(parent, list.get(i));
288         list.set(i, temp);
289     }
290 
291     public int tamano() {
292 
293         return list.size();
294     }
295 
296     private int right(int i) {
297 
298         return 2 * i + 2;
299     }
300 
301     private int left(int i) {
302 
303         return 2 * i + 1;
304     }
305 
306     private int parent(int i) {
307 
308         if (i %2 == 1) {
309             return i / 2;
310         }
311 
312         return (i - 1) / 2;
```

```
313         }
314 }
```