GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ALGORITMOS DE ORDENAMIENTOS

1. Introducción

En múltiples problemas podemos encontrar que la solución radica en ordenar una serie de elementos dados o que sencillamente nuestro algoritmo solución fuera más eficiente si los datos iniciales estuvieran ordenados de acuerdo cierto criterio. Es por eso que se hace necesario conocer los diferentes algoritmos de ordenación sus principales caracterisiticas de las cuales se derivan su ventajas y desventajas las cuales conducen a las situaciones donde son aplicables o no.

2. Conocimientos previos

2.1. Arreglos

Un arreglo es una serie de elementos del mismo tipo ubicados en zonas de memoria continuas que pueden ser referenciados por un índice y un único identificador, esto quiere decir que podemos almacenar 10 valores enteros en un arreglo sin tener que declarar 10 variables diferentes.

Los arreglos unidimensionales son estructuras de datos caracterizadas por:

- Una colección de datos del mismo tipo.
- Son referenciados mediante un mismo nombre.
- Almacenados en posiciones de memoria físicamente contiguas, de ahí que la posición más baja corresponde al primer elemento y la mas alta al del ultimo elemento.
- El formato general de la declaración de una variable de este tipo en C++ puede ser estática:

```
Tipo_de_datos Nombre_de_la_variable [capacidad]
```

o dinámica:

```
Tipo_de_datos * Nombre_de_la_variable = new Tipo_de_datos [capacidad]
```

En caso de Java existe una sola forma de declaración y tiene la siguiente sintaxis

```
Tipo_de_datos [] Nombre_de_la_variable = new Tipo_de_datos [capacidad]
```

Todo arreglo se compone de un determinado número de elementos. Cada elemento es referenciado por la posición que ocupa dentro del vector. Dichas posiciones son llamadas índice y siempre son correlativos. Existen tres formas de indexar los elementos de un arreglo:

- Indexación base-cero (0): En este modo el primer elemento del vector será la componente cero (‘0’) del mismo, es decir, tendrá el índice ‘0’. En consecuencia, si el vector tiene ‘n’componentes la última tendrá como índice el valor ‘n-1’. La mayoría de los lenguajes de programación asumen esta forma de indexar los elementos de un arreglo.

- Indexación base-uno (1): En esta forma de indexación, el primer elemento de la elemento tiene el índice '1'y el último tiene el índice 'n'(para un arreglo de 'n'componentes). En varios problemas es conveniente utilizar este en ves de usar la indexación que propone los lenguajes de programación. El detalle con este proceder es que a la hora de definir el tamaño del arreglo no pude ser de n sino de n+<una cantidad mayor que cero>.
- Indexación base-n (n): Este es un modo versátil de indexación en la que el índice del primer elemento puede ser elegido libremente, en algunos lenguajes de programación se permite que los índices puedan ser negativos e incluso de cualquier tipo escalar (también cadenas de caracteres).

2.2. Vectores

Es un modelo basado en un modelo de arreglo dinámico que permite el trabajo y gestión con los elementos almacenados en un arreglo dinámico. Pero a diferencia de los arreglos regulares, el almacenamiento en vectores se maneja automáticamente, lo que permite que se amplíe y se contrate según sea necesario. Para utilizar un vector en C++ se debe incluir en la cabecera del archivo el siguiente fragmento:

```
#include <vector>
```

Para declarar un vector solo basta con poner algo como lo que sigue:

```
vector<T> nombre_del_vector;
```

Donde *T* debe ser uno de los tipos de datos definidos por el lenguaje de progrmación o por el propio programador. El vector es muy útil cuando se va acceder a los elementos conocidos su posición dentro de la estructura y se va añadir o eliminar elementos de última posición. No presenta igual desempeño cuando las inserciones y eliminaciones se producen en otras posiciones. Una forma muy eficiente de utilizar el vector es una vez creado inicializarlo con la cantidad máxima de elementos que puede alamcenar siempre y cuando se conozca este valor.

```
#include <iostream>
#include <vector>
using namespace std;

int main (){
   unsigned int i;
   // variantes para constuir un vector:
   // un vector de enteros vacios
   vector<int> first;

   /*un vector con cuatro elementos, cada elementos con el valor 100*/
   vector<int> second (4,100);

   /*creando un vector iterando sobre otro*/
```

```
    vector<int> third (second.begin(),second.end());

    /*creando un vector copia del tercer vector*/
    vector<int> fourth (third);

    /*creando un vector a partir de un vector*/
    int myints[] = {16,2,77,29};
    vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

    /*adicionando un elemento al vector*/
    first.push_back(2);

    /*limpiar un vector*/
    second.clear();

    /*saber si un vector no tiene elemento*/
    if(fourth.empty()) cout<<"Vector empty"<<endl;
    else cout<<"Vector not empty"<<endl;

    cout << "The contents of fifth are:";
    for (i=0; i < fifth.size(); i++)
       cout << " " << fifth[i];
    cout << endl;
    return 0;
}
```

3. Desarrollo

Los algoritmos de ordenamiento se clasifican o se agrupan de acuerdo a dos criterios el primero es estabilidad del algoritmo. Se dice que un algoritmo de ordenamiento es estable cuando en la colección existente valores similares y una vez ordenados estos valores mantienen entre ellos el orden inicial , no cumplirse esto se dice que el algoritmo de ordenamiento no estable.

El otro criterio por el cual se clasifica los algoritmos de ordenamientos es el referido al uso de memoria adicional a la reservada para almacenar los elementos. Para ordenar una secuencia de elementos es necesario tener dichos elementos contenidos dentro una estructura computacional de almacenamiento como son los arreglos, listas y vectores. Cuando un método utiliza memoria adicional a la utilizada con el uso de alguna estructura computacional de almacenamiento se dice que el algoritmo es no *in-situ* en caso contrario de solo utilizar la memoria relacionada con las estructura computacional de almacenamiento se dice que el algorimo es in-situ. En otras paralabras si el algoritmo utiliza alguna estructura de datos adicional a la que contiene los elementos de la secuencia es un algoritmo no *in-situ* en caso contrario es *in-situ*

3.1. Algoritmos cuadráticos

3.1.1. Burble Sort

La ordenación de burbuja (Bubble Sort en inglés) es un sencillo algoritmo de ordenamiento. Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada.

3.1.2. Selection Sort

El ordenamiento por selección (Selection Sort en inglés) es un algoritmo de ordenamiento que su funcionamiento es el siguiente:

- Buscar el mínimo elemento de la lista.
- Intercambiarlo con el primero.
- Buscar el siguiente mínimo en el resto de la lista.
- Intercambiarlo con el segundo.

Y en general

- Buscar el mínimo elemento entre una posición i y el final de la lista.
- Intercambiar el mínimo con el elemento de la posición i.

3.1.3. Insertion Sort

El ordenamiento por inserción (insertion sort en inglés) es una manera muy natural de ordenar para un ser humano, y puede usarse fácilmente para ordenar un mazo de cartas numeradas en forma arbitraria. Inicialmente se tiene un solo elemento, que obviamente es un conjunto ordenado. Después, cuando hay k elementos ordenados de menor a mayor, se toma el elemento k+1 y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y este es el más pequeño). En este punto se inserta el elemento k+1 debiendo desplazarse los demás elementos.

3.2. Algoritmos logaritmico

3.2.1. Heap Sort

El ordenamiento por montículos (heapsort en inglés) es un algoritmo de ordenamiento no recursivo, no estable.

Este algoritmo consiste en almacenar todos los elementos del vector a ordenar en un montículo (heap), y luego extraer el nodo que queda como nodo raíz del montículo (cima) en sucesivas iteraciones obteniendo el conjunto ordenado. Basa su funcionamiento en una propiedad de los montículos, por la cual, la cima contiene siempre el menor elemento (o el mayor, según se haya

definido el montículo) de todos los almacenados en él. El algoritmo, después de cada extracción, recoloca en el nodo raíz o cima, la última hoja por la derecha del último nivel. Lo cual destruye la propiedad heap del árbol. Pero, a continuación realiza un proceso de "descenso "del número insertado de forma que se elige a cada movimiento el mayor de sus dos hijos, con el que se intercambia. Este intercambio, realizado sucesivamente "hunde"el nodo en el árbol restaurando la propiedad montículo del árbol y dejando paso a la siguiente extracción del nodo raíz.

El algoritmo, en su implementación habitual, tiene dos fases. Primero una fase de construcción de un montículo a partir del conjunto de elementos de entrada, y después, una fase de extracción sucesiva de la cima del montículo. La implementación del almacén de datos en el heap, pese a ser conceptualmente un árbol, puede realizarse en un vector de forma fácil. Cada nodo tiene dos hijos y por tanto, un nodo situado en la posición i del vector, tendrá a sus hijos en las posiciones 2*i, y 2*i+1 suponiendo que el primer elemento del vector tiene un índice = 1. Es decir, la cima ocupa la posición inicial del vector y sus dos hijos la posición segunda y tercera, y así, sucesivamente. Por tanto, en la fase de ordenación, el intercambio ocurre entre el primer elemento del vector (la raíz o cima del árbol, que es el mayor elemento del mismo) y el último elemento del vector que es la hoja más a la derecha en el último nivel. El árbol pierde una hoja y por tanto reduce su tamaño en un elemento. El vector definitivo y ordenado, empieza a construirse por el final y termina por el principio.

3.2.2. Merge Sort

El algoritmo de ordenamiento por mezcla (merge sort en inglés) es un algoritmo de ordenamiento externo estable basado en la técnica divide y vencerás. Conceptualmente, el ordenamiento por mezcla funciona de la siguiente manera:

1. Si la longitud de la lista es 0 ó 1, entonces ya está ordenada. En otro caso.
2. Dividir la lista desordenada en dos sublistas de aproximadamente la mitad del tamaño.
3. Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla.
4. Mezclar las dos sublistas en una sola lista ordenada.

El ordenamiento por mezcla incorpora dos ideas principales para mejorar su tiempo de ejecución:

1. Una lista pequeña necesitará menos pasos para ordenarse que una lista grande.
2. Se necesitan menos pasos para construir una lista ordenada a partir de dos listas también ordenadas, que a partir de dos listas desordenadas. Por ejemplo, sólo será necesario entrelazar cada lista una vez que están ordenadas.

3.2.3. Quick Sort

El ordenamiento rápido (quicksort en inglés) es un algoritmo creado por el científico británico en computación C. A. R. Hoare, basado en la técnica de divide y vencerás, que permite, en promedio, ordenar n elementos en un tiempo proporcional a $N \log N$.

El algoritmo trabaja de la siguiente forma:

- Elegir un elemento de la lista de elementos a ordenar, al que llamaremos pivote.
- Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
- La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
- Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.

3.3. Otros

3.3.1. CountingSort

El ordenamiento por cuentas (counting sort en inglés) es un algoritmo de ordenamiento en el que se cuenta el número de elementos de cada clase para luego ordenarlos. Sólo puede ser utilizado por tanto para ordenar elementos que sean contables (como los números enteros en un determinado intervalo, pero no los números reales, por ejemplo).

El primer paso consiste en averiguar cuál es el intervalo dentro del que están los datos a ordenar (valores mínimo y máximo). Después se crea un vector de números enteros con tantos elementos como valores haya en el intervalo [mínimo, máximo], y a cada elemento se le da el valor 0 (0 apariciones). Tras esto se recorren todos los elementos a ordenar y se cuenta el número de apariciones de cada elemento (usando el vector que hemos creado). Por último, basta con recorrer este vector para tener todos los elementos ordenados.

3.3.2. Radix Sort

En informática, el ordenamiento Radix (radix sort en inglés) es un algoritmo de ordenamiento que ordena enteros procesando sus dígitos de forma individual. Como los enteros pueden representar cadenas de caracteres (por ejemplo, nombres o fechas) y, especialmente, números en punto flotante especialmente formateados, radix sort no está limitado sólo a los enteros.

La mayor parte de los ordenadores digitales representan internamente todos sus datos como representaciones electrónicas de números binarios, por lo que procesar los dígitos de las representaciones de enteros por representaciones de grupos de dígitos binarios es lo más conveniente. Existen dos clasificaciones de radix sort: el de dígito menos significativo (LSD) y el de dígito más significativo (MSD). Radix sort LSD procesa las representaciones de enteros empezando por el dígito menos significativo y moviéndose hacia el dígito más significativo. Radix sort MSD trabaja en sentido contrario.

Las representaciones de enteros que son procesadas por los algoritmos de ordenamiento se les llama a menudo “claves”, que pueden existir por sí mismas o asociadas a otros datos. Radix sort LSD usa típicamente el siguiente orden: claves cortas aparecen antes que las claves largas, y claves de la misma longitud son ordenadas de forma léxica. Esto coincide con el orden normal de las representaciones de enteros, como la secuencia “1, 2, 3, 4, 5, 6, 7, 8, 9, 10”. Radix sorts MSD usa orden léxico, que es ideal para la ordenación de cadenas de caracteres, como las palabras o representaciones de enteros de longitud fija. Una secuencia como “b, c, d, e, f, g, h, i, j, ba”será ordenada léxicamente como “b, ba, c, d, e, f, g, h, i, j”. Si se usa orden léxico para ordenar representaciones de enteros de longitud variable, entonces la ordenación de las representaciones de los números del 1 al 10 será “1, 10, 2, 3, 4, 5, 6, 7, 8, 9”, como si las claves más cortas estuvieran justificadas a la izquierda y rellenadas a la derecha con espacios en blanco, para hacerlas tan largas como la clave más larga, para el propósito de este ordenamiento.

3.4. Bibliotecas con funciones de ordenamiento

Hoy en día la mayoría de los lenguajes de programación poseen un grupo de bibliotecas o funciones que permiten ordenar una colección de elementos sean de un tipo dato primitivo del lenguaje o bien creados por el desarrollador. Este elemento es muy importante por dos motivos:

- Reduce el esfuerzo. No debemos emplear tiempo en la implementación de un algoritmo que ordene, sino que podemos utilizar alguna función de las definidas en lenguaje de programación escogido.
- Reduce la posibilidad de errores que puede surgir a la hora de implementar por nuestro propios esfuerzos un algoritmo de ordenación. Las funciones o métodos que nos brinda los lenguajes de programación hay seguridad de su funcionamiento correcto y de manera eficiente

A continuación vamos a ver algunas de esas funciones de los lenguajes de programación C++ y Java

3.4.1. Bibliotecas de C++

Para ordenar el lenguaje de programación C++ cuenta con la biblioteca *algorithm* la cual posee las siguientes funcionalidades

- *sort()*. Esta función ordena los elementos de una colección de manera ascedente. Como detalle de la función es que no garantiza el orden inicial entre elementos de igual valor. Su complejidad es O(N log (N)) tanto en el caso promedio como en el peor de los casos. Una segunda variante de esta funcionalidad se le pasa un función para comparar los elementos de la colección. Esta variante es utilizada cuando se desea ordenar tipos de datos creados por el programador.

```
#include <algorithm>
void sort( iterator start, iterator end );
void sort( iterator start, iterator end, StrictWeakOrdering cmp );
```

- *stable_sort*. Esta función es similar a la anterior con la diferencia que si mantiene el orden inicial de los elementos cuando estos tienen similar valor. Otra diferencia es el tiempo de ejecucción el cual puede alcanzar N $(logN)^2$ en el peor de los casos.

```
#include <algorithm>
void stable_sort( iterator start, iterator end );
void stable_sort( iterator start, iterator end, StrictWeakOrdering cmp );
```

- Para realizar el ordenamiento Heap Sort se cuenta con las funcionalidades *sort_heap*, *is_heap*,*make_heap*, *pop_heap*, *push_heap*
- *partial_sort* Es una función que permite ordenar los primeros N elementos de una colección. N elementos es definido por la cantidad de elementos en rango comprendido [start,end).

```
#include <algorithm>
void partial_sort( iterator start, iterator middle, iterator end );
void partial_sort( iterator start, iterator middle, iterator end,
    StrictWeakOrdering cmp );
```

3.4.2. Bibliotecas de Java

En el caso del lenguaje de programación Java cuenta con la clase *Collections* perteneciente al paquete java en el subpaquete util. Esta clase con un número métodos estáticos entre los cuales podemos encontrar métodos para ordenar colecciones. El algoritmo sort ordena los elementos de un objeto List , el cual debe implementar a la interfaz Comparable . El orden se determina en base al orden natural del tipo de los elementos, según su implementación mediante el método compareTo de ese objeto. El método compareTo está declarado en la interfaz Comparable y algunas veces se le conoce como el método de comparación natural. La llamada a sort puede especificar como segundo argumento un objeto Comparator , para determinar un ordenamiento alterno de los elementos.

```
void sort(List list)
void sort(List list, Comparator c)
```

El primer método lo utilizaremos cuando los elementos de la lista implementan la interfaz Comparable vista anteriormente y el segundo lo utilizaremos cuando querramos utilizar nuestro propio comparador o cuando no nos guste el funcionamiento del comparador por defecto de los elementos de nuestra lista.

Ambas versiones garantizan un coste de O(nlog(n)) y puede acercarse a un rendimiento lineal cuando las los elementos se encuentran cerca de su orden natural. El algoritmo utilizado es una pequeña variación del algoritmo de mergesort y la operación que realiza es destructiva, es decir, no podremos recuperar el orden original si no hemos guardado la lista previamente.

Ordenamiento ascendente Se utiliza el algoritmo sort para ordenar los elementos de un objeto List en forma ascendente (línea 20). Recuerde que List es un tipo genérico y acepta un argumento de tipo, el cual especifi ca el tipo de elemento de lista; en la línea 15 se declara a lista como un objeto List de objetos String . Observe que en las líneas 18 y 23 se utiliza una llamada implícita al método toString de lista para imprimir el contenido de la lista en el formato que se muestra en las líneas segunda y cuarta de los resultados.

```
import java.util.List;
import java.util.Arrays;
import java.util.Collections;
public class Ordenamiento1{
   private static final String palos[] ={ "Corazones", "Diamantes", "Bastos",
       "Espadas" };
   // muestra los elementos del arreglo
   public void imprimirElementos(){
      List< String > lista = Arrays.asList( palos ); // crea objeto List
      // imprime lista
      System.out.printf( "Elementos del arreglo desordenados:\n%s\n", lista );
      Collections.sort( lista ); // ordena ArrayList
      // imprime lista
      System.out.printf( "Elementos del arreglo ordenados:\n%s\n", lista );
   } // fin del metodo imprimirElementos

   public static void main( String args[] ){
      Ordenamiento1 orden1 = new Ordenamiento1();
      orden1.imprimirElementos();
   } // fin de main
} // fin de la clase Ordenamiento1
```

Ordenamiento descendente se ordena la misma lista de cadenas utilizadas en el ejemplo, en orden descendente. El ejemplo introduce la interfaz Comparator , la cual se utiliza para ordenar los elementos de un objeto Collection en un orden distinto. En la línea 21 se hace una llamada al método sort de Collections para ordenar el objeto List en orden descendente. El método static reverseOrder de Collections devuelve un objeto Comparator que ordena los elementos de la colección en orden inverso.

```
import java.util.List;
import java.util.Arrays;
import java.util.Collections;
public class Ordenamiento1{
   private static final String palos[] ={ "Corazones", "Diamantes", "Bastos",
       "Espadas" };
   // muestra los elementos del arreglo
   public void imprimirElementos(){
      List< String > lista = Arrays.asList( palos ); // crea objeto List
      // imprime lista
      System.out.printf( "Elementos del arreglo desordenados:\n%s\n", lista );
```

```
      Collections.sort( lista, Collections.reverseOrder() );
      // imprime lista
          System.out.printf( "Elementos del arreglo ordenados:\n%s\n", lista )
              ;
   } // fin del metodo imprimirElementos

   public static void main( String args[] ){
      Ordenamiento1 orden1 = new Ordenamiento1();
      orden1.imprimirElementos();
   } // fin de main
} // fin de la clase Ordenamiento1
```

Mientras para ordenar arreglos Java proporciona la clase *Array* que de similar manera que *Collections* posee un grupo de métodos para trabajar pero con arreglos.

4. Implementación

4.1. C++

```
int n;
int niz[MAX_N],tmp[MAX_N];

inline void bubbleSort(){
   bool swapped;
   int it = 0;
   do{
      swapped = false;
      for(int i=0;i<n-it-1;i++){
         if (niz[i] > niz[i+1]){
            swap(niz[i], niz[i+1]);
            swapped = true;
         }
      }
      it++;
   } while (swapped);
}

inline void selectionSort(){
   for (int i=0;i<n-1;i++){
      int minPos = i;
      for(int j=i+1;j<n;j++){
         if(niz[j] < niz[minPos]){
            minPos = j;
         }
      }
      swap(niz[i], niz[minPos]);
   }
}
```

```
inline void insertionSort(){
   for(int i=1;i<n;i++){
      int j = i - 1;
      int tmp = niz[i];
      while (j >= 0 && niz[j] > tmp){
         niz[j+1] = niz[j];
         j--;
      }
      niz[j+1] = tmp;
   }
}

int count[MAX_K];

void countingSort(){
   for (int i=0;i<n;i++)
      count[niz[i]]++;
      int ii = 0;
   for (int i=0;i<MAX_K;i++){
      while (count[i] > 0){
         niz[ii++] = i;
         count[i]--;
      }
   }
}


inline void merge(int left, int mid, int right){
   int h,i,j,k;
   h = left;
   i = left;
   j = mid+1;
   while (h <= mid && j <= right){
      if (niz[h] <= niz[j]){
         tmp[i] = niz[h];
         h++;
      }
      else{
         tmp[i] = niz[j];
         j++;
      }
      i++;
   }
   if (h > mid){
      for(k=j;k<=right;k++){
         tmp[i] = niz[k];
         i++;
      }
   }
```

```
   else{
      for(k=h;k<=mid;k++){
         tmp[i] = niz[k];
         i++;
      }
   }
   for(k=left;k<=right;k++) niz[k] = tmp[k];
}

void mergeSort(int left, int right){
   if (left == right) return;
   int MID = (left+right)/2;
   mergeSort(left, MID);
   mergeSort(MID+1, right);
   merge(left, MID, right);
}

//La llamada se realiza de la siguiente manera
//mergeSort(0,n-1);

void qsort(int left, int right){
   if (left < right){
      int i = left;
      int j = right;
      int pivot = niz[(i+j)/2];
      while (i<=j){
         while (niz[i]<pivot) i++;
         while (niz[j]>pivot) j--;
         if (i<=j){
            swap(niz[i], niz[j]);
            i++; j--;
         }
      }
      qsort(left,j);
      qsort(i,right);
   }
}
//La llamada se realiza de la siguiente manera:
//qsort(0,n-1);

int heap_size;

inline void Heapify(int pos){
   if (pos > heap_size) return;
   int ret = pos;
   int left = pos*2;
   int right = pos*2+1;
   if (left <= heap_size && niz[left] > niz[ret]) ret = left;
   if (right <= heap_size && niz[right] > niz[ret]) ret = right;
   if (ret != pos){
```

```
        swap(niz[pos], niz[ret]);
        Heapify(ret);
    }
}

inline void Pop(){
    int pos = 1;
    swap(niz[pos], niz[heap_size--]);
    while (pos <= heap_size){
        int ret = pos;
        int left = pos*2;
        int right = pos*2+1;
        if (left <= heap_size && niz[left] > niz[ret])
            ret = left;
        if (right <= heap_size && niz[right] > niz[ret])
            ret = right;
        if (ret != pos){
            swap(niz[pos], niz[ret]);
            pos = ret;
        }
        else  break;
    }
}

int main(){
    n = 5;
    niz[1] = 4;
    niz[2] = 2;
    niz[3] = 5;
    niz[4] = 1;
    niz[5] = 3;

    heap_size = n;

    for (int i=n/2;i>=1;i--)
        Heapify(i);
    while (heap_size > 1)
        Pop();
    for (int i=1;i<=n;i++)
        printf("%d ",niz[i]);
        printf("\n");
    return 0;
}
```

4.2. Java

```
public static void qSort(int[] a, int low, int high) {
    if (low >= high) return;
```

```
    int separator = a[low + rnd.nextInt(high - low + 1)];
    int i = low;
    int j = high;
    do {
        while (a[i] < separator) ++i;
        while (a[j] > separator) --j;
        if (i > j) break;
        int t = a[i];
        a[i] = a[j];
        a[j] = t;
        ++i;
        --j;
    } while (i <= j);
    qSort(a, low, j);
    qSort(a, i, high);
}

public static void mergeSort(int[] a, int low, int high) {
    if (high - low < 2)return;
    int mid = (low + high) >>> 1;
    mergeSort(a, low, mid);
    mergeSort(a, mid, high);
    int[] b = Arrays.copyOfRange(a, low, mid);
    for (int i = low, j = mid, k = 0; k < b.length; i++) {
        if (j == high || b[k] <= a[j]) {
            a[i] = b[k++];
        }
        else{
            a[i] = a[j++];
        }
    }
}

static void swap(int[] a, int i, int j) {
    int t = a[j];
    a[j] = a[i];
    a[i] = t;
}

public static void heapSort(int[] a) {
    int n = a.length;
    for (int i = n / 2 - 1; i >= 0; i--) pushDown(a, i, n);
    while (n > 1) {
        swap(a, 0, n - 1);
        pushDown(a, 0, --n);
    }
}

static void pushDown(int[] h, int pos, int size) {
    while (true) {
```

```
            int child = 2 * pos + 1;
            if (child >= size) break;
            if (child + 1 < size && h[child + 1] > h[child])
               child++;
            if (h[pos] >= h[child]) break;
            swap(h, pos, child);
            pos = child;
         }
      }

      public static void bubbleSort(int[] a) {
         for (int i = 0; i + 1 < a.length; i++) {
            for (int j = 0; j + 1 < a.length; j++) {
               if (a[j] > a[j + 1]) {
                  swap(a, j, j + 1);
               }
            }
         }
      }

      public static void selectionSort(int[] a) {
         int n = a.length;
         int[] p = new int[n];
         for (int i = 0; i < n; i++) p[i] = i;
            for (int i = 0; i < n - 1; i++) {
               for (int j = i + 1; j < n; j++) {
                  if (a[p[i]] > a[p[j]]) {
                     swap(p, i, j);
                  }
               }
            }
         int[] b = a.clone();
         for (int i = 0; i < n; i++) a[i] = b[p[i]];
      }

      public static void insertionSort(int[] a) {
         for (int i = 1; i < a.length; i++) {
            for (int j = i; j > 0; j--) {
               if (a[j - 1] > a[j]) {
                  swap(a, j - 1, j);
               }
            }
         }
      }

      public static void countingSort(int[] a) {
         int max = 0;
         for (int x : a) {
            max = Math.max(max, x);
         }
```

```
    int[] cnt = new int[max + 1];
    for (int x : a) {
        ++cnt[x];
    }
    for (int i = 1; i < cnt.length; i++) {
        cnt[i] += cnt[i - 1];
    }
    int n = a.length;
    int[] b = new int[n];
    for (int i = 0; i < n; i++) {
        b[--cnt[a[i]]] = a[i];
    }
    System.arraycopy(b, 0, a, 0, n);
}

public static void radixSort(int[] a) {
    final int d = 8;
    final int w = 32;
    int[] t = new int[a.length];
    for (int p = 0; p < w / d; p++) {
        // counting-sort
            int[] cnt = new int[1 << d];
        for (int i = 0; i < a.length; i++)
            ++cnt[((a[i] ^ Integer.MIN_VALUE) >>> (d * p)) & ((1 << d) - 1)];
        for (int i = 1; i < cnt.length; i++)
            cnt[i] += cnt[i - 1];
        for (int i = a.length - 1; i >= 0; i--)
            t[--cnt[((a[i] ^ Integer.MIN_VALUE) >>> (d * p)) & ((1 << d) - 1)]] =
                a[i];
        System.arraycopy(t, 0, a, 0, a.length);
    }
}
```

5. Complejidad

El ordenamiento de burbuja tiene una complejidad $O(n^2)$ igual que ordenamiento por selección. Cuando una lista ya está ordenada, a diferencia del ordenamiento por inserción que pasará por la lista una vez y encontrará que no hay necesidad de intercambiar las posiciones de los elementos, el método de ordenación por burbuja está forzado a pasar por dichas comparaciones, lo que hace que su complejidad sea cuadrática en el mejor de los casos.

Tanto el merge sort, quick sort y heap sort su complejidad es de O ($n \log n$) pero existen casos especiales como que la coleción ya este ordenada que para ese caso la complejidad del quick sort es O(n^2) . Aunque heap sort tiene los mismos límites de tiempo que merge sort, requiere sólo O(1) espacio auxiliar en lugar del O(n) de merge sort, y es a menudo más rápido en implementaciones prácticas. Quick sort, sin embargo, es considerado por mucho como el más rápido algoritmo de ordenamiento de propósito general. En el lado bueno, merge sort es un ordenamiento estable,

paraleliza mejor, y es más eficiente manejando medios secuenciales de acceso lento. Merge sort es a menudo la mejor opción para ordenar una lista enlazada: en esta situación es relativamente fácil implementar merge sort de manera que sólo requiera O(1) espacio extra, y el mal rendimiento de las listas enlazadas ante el acceso aleatorio hace que otros algoritmos (como Quick sort) den un bajo rendimiento, y para otros (como heap sort) sea algo imposible.

El counting Sort se trata de un algoritmo estable cuya complejidad computacional es O($n + k$), siendo n el número de elementos a ordenar y k el tamaño del vector auxiliar (máximo - mínimo). La eficiencia del algoritmo es independiente de lo casi ordenado que estuviera anteriormente. Es decir no existe un mejor y peor caso, todos los casos se tratan iguales.El algoritmo counting, no se ordena in situ, sino que requiere de una memoria adicional.El algoritmo posee una serie de limitaciones que obliga a que sólo pueda ser utilizado en determinadas circunstancias (solo para números enteros, no vale para ordenar cadenas y es desaconsejable para ordenar números decimales. Incluso con números enteros es cuando el rango entre el mayor y el menor es muy grande.).

6. Aplicaciones

Además de existir problemas que su solución radica propiamente en el ordenamiento de los datos con que se trabaja, existe otros problemas o algoritmos que para lograr una solución o aplicarlos es necesario como precondición que la colección de datos este ordenada. Además la idea del algoritmos como heap, merge o counting sirven de base o idea para otros algoritmos.

7. Ejercicios propuestos

A continuación una lista de ejercicios que se resuelven utilizando aplicando los conocimientos abordados en esta guía:

- MOG - A - A ordenar!
- MOG - A - Igualando Lista
- MOG - D - Elementos consecutivos
- MOG - A - Cake
- MOG - A - Formación
- MOG - B - Phone List
- DMOJ - Lista de teléfonos
- DMOJ - Permutación de Palabras
- DMOJ - Aislador
- DMOJ - Olivander