Ordenación

1. Ordenación Burbuja
2. Ordenación por Selección
3. Ordenación por Inserción
4. Ordenación por Mezcla
5. Ordenación Rápida de Hoare (Quick Sort)
6. Ordenación Mediante Montículo (HeapSort)

Ordenación Burbuja

Consiste en ciclar repetidamente a través de la lista, comparando elementos adyacentes de dos en dos. Si un elemento es mayor que el que está en la siguiente posición se intercambian.

Tiempo de Ejecución: El ciclo interno se ejecuta n veces para una lista de n elementos. El ciclo externo también se ejecuta n veces. Es decir, la complejidad es n * n = O(n2). El comportamiento del caso promedio depende del orden de entrada de los datos, pero es sólo un poco mejor que el del peor caso, y sigue siendo O(n2).

Ventajas:

- Fácil implementación.
- No requiere memoria adicional.

Desventajas:

- Muy lento.
- Realiza numerosas comparaciones.
- Realiza numerosos intercambios.

Ejemplo:

		0	1	2	3	4	5	6	7
i = 1	j								
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i = 2	0	3	1	5	8	2	4	7	**9**
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 3	0	1	3	5	2	4	7	**8**	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 4	0	1	3	2	4	5	**7**		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 5	0	1	2	3	4	**5**			
	1	1	2	3	4				
	2	1	2	3	4				
i = 6	0	1	2	3	**4**				
	1	1	2	3					
i = 7	0	1	2	**3**					
		1	**2**						

Implementación en C :

```
void bubbleSort( int *a, int n)
{

        int i, j, temp;
        int sorted;

        for( i = 1; i < n; ++i)  // repeat bubble pass n-1 times
        {                                                //  i from 1 to n-1

                sorted = true;         // for efficiency reasons

                for( j = 0; j < n-i; ++j)   // j from 0 to n-i-1
                        if( a[j] > a[j+1]) {   // n-i comparisons
                                temp = a[j];
                                a[j] = a[j+1];
                                a[j+1] = temp;
                                sorted = false;
                        }
                if (sorted) break;// no changes in a pass => already sorted
        }                       // so no need for further bubble passes

}
```

Ordenación por Selección

Consiste en lo siguiente:

- Buscas el elemento más pequeño de la lista.
- Lo intercambias con el elemento ubicado en la primera posición de la lista.
- Buscas el segundo elemento más pequeño de la lista.
- Lo intercambias con el elemento que ocupa la segunda posición en la lista.
- Repites este proceso hasta que hayas ordenado toda la lista.

Tiempo de Ejecución: El ciclo externo se ejecuta n veces para una lista de n elementos. Cada búsqueda requiere comparar todos los elementos no clasificados. Luego la complejidad es O(n2). Este algoritmo presenta un comportamiento constante independiente del orden de los datos. Luego la complejidad promedio es también O(n2).

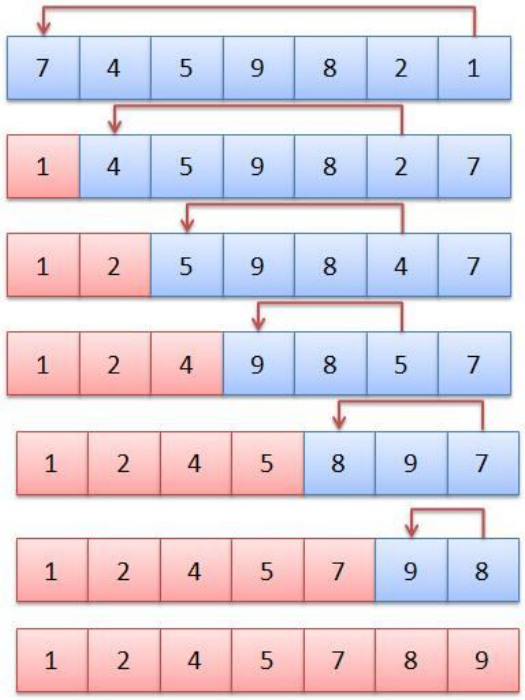
Ventajas:

- Fácil implementación.
- No requiere memoria adicional.
- Realiza pocos intercambios.
- Rendimiento constante: poca diferencia entre el peor y el mejor caso.

Desventajas:

- Lento.
- Realiza numerosas comparaciones.

Ejemplo:



Implementación en C :

```
void selectSort(int *arr, int n)
{
//pos_min is short for position of min
      int pos_min,temp;

      for (int i=0; i < n-1; i++)
      {
          pos_min = i; //set pos_min to the current index of array

            for (int j=i+1; j < n; j++)
            {

            if (arr[j] < arr[pos_min])
                  pos_min=j;
      //pos_min will keep track of the index that min is in, this is needed when a
swap happens
            }

      //if pos_min no longer equals i than a smaller value must have been found, so
a swap must occur
            if (pos_min != i)
            {
                 temp = arr[i];
                 arr[i] = arr[pos_min];
                 arr[pos_min] = temp;
            }
      }
```

Ordenación por Inserción

Este algoritmo también es bastante sencillo. ¿Has jugado cartas?. ¿Cómo las vas ordenando cuando las recibes? Yo lo hago de esta manera: tomo la primera y la coloco en mi mano. Luego tomo la segunda y la comparo con la que tengo: si es mayor, la pongo a la derecha, y si es menor a la izquierda (también me fijo en el color, pero omitiré esa parte para concentrarme en la idea principal). Después tomo la tercera y la comparo con las que tengo en la mano, desplazándola hasta que quede en su posición final. Continúo haciendo esto, insertando cada carta en la posición que le corresponde, hasta que las tengo todas en orden. ¿Lo haces así tu también? Bueno, pues si es así entonces comprenderás fácilmente este algoritmo, porque es el mismo concepto.

Para simular esto en un programa necesitamos tener en cuenta algo: no podemos desplazar los elementos así como así o se perderá un elemento. Lo que hacemos es guardar una copia del elemento actual (que sería como la carta que tomamos) y desplazar todos los elementos mayores hacia la derecha. Luego copiamos el elemento guardado en la posición del último elemento que se desplazó.

Tiempo de Ejecución: Para una lista de n elementos el ciclo externo se ejecuta n-1 veces. El ciclo interno se ejecuta como máximo una vez en la primera iteración, 2 veces en la segunda, 3 veces en la tercera, etc. Esto produce una complejidad O(n2).

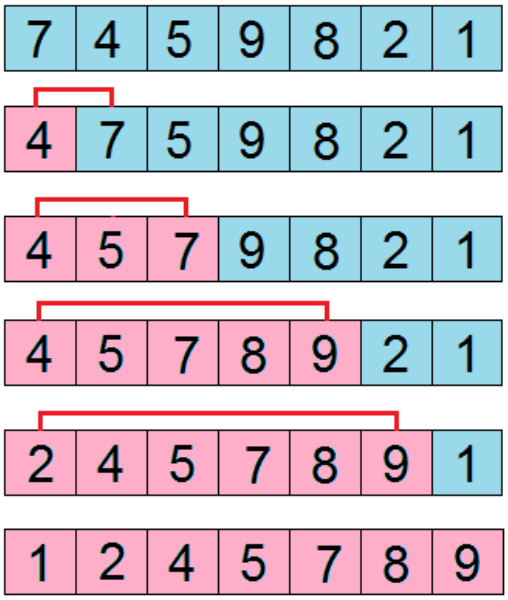
Ventajas:

- Fácil implementación.
- Requerimientos mínimos de memoria.

Desventajas:

- Lento.
- Realiza numerosas comparaciones.

Ejemplo :



Implementación en C :

```
void insertion_sort (int *arr, int length){
          int j, temp;

     for (int i = i; i < length; i++){
          j = i;

          while (j > 0 && arr[j] < arr[j-1]){
                 temp = arr[j];
                 arr[j] = arr[j-1];
                 arr[j-1] = temp;
                 j--;
                 }
          }
}
```

Ordenación por Mezcla

Fue desarrollado en 1945 por John Von Neumann. Este método utiliza la técnica de Divide y Vencerás para realizar la ordenación del vector. Su estrategia consiste en dividir el vector en dos subvectores, ordenarlos mediante llamadas recursivas, y finalmente combinar los dos subvectores ya ordenados. Conceptualmente, el ordenamiento por mezcla funciona de la siguiente manera:

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

Tiempo de ejecución :

este método ordena n elementos en tiempo Θ(nlogn) en cualquiera de los casos (peor, mejor o medio). Sin embargo tiene una complejidad espacial, en cuanto a memoria, mayor que los demás (del orden de n).

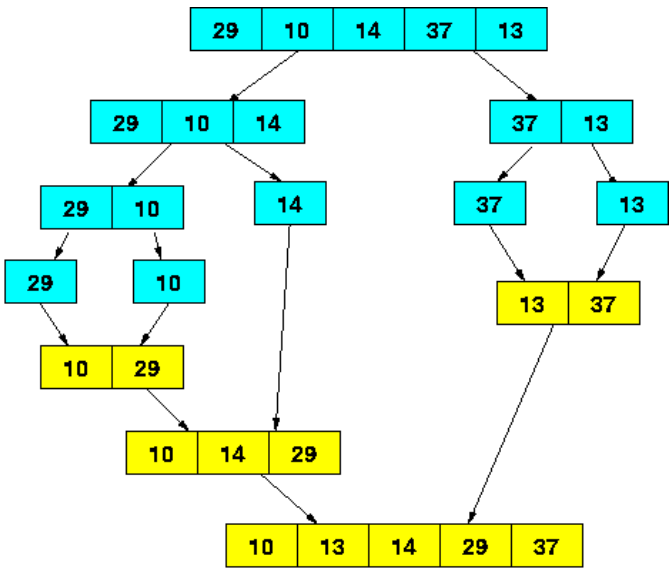
Ventajas:

- Método estable de ordenamiento mientras la operación de mezclas (merge) este bien implementada.
- Este algoritmo es efectivo para conjuntos de datos que se puedan acceder como arreglos, vectores y listas ligadas.

Desventajas :

- Su principal desventaja radica en que está definido recursivamente y su implementación no recursiva emplea una pila, por lo que requiere un espacio adicional de memoria para almacenarla.

Ejemplo:



Implementación en C :

```
void partition(int *arr,int low,int high){

    int mid;

    if(low<high){
         mid=(low+high)/2;
         partition(arr,low,mid);
         partition(arr,mid+1,high);
         mergeSort(arr,low,mid,high);
    }
}

void mergeSort(int *arr,int low,int mid,int high){

    int i,m,k,l,temp[MAX];

    l=low;
    i=low;
    m=mid+1;

    while((l<=mid)&&(m<=high)){

         if(arr[l]<=arr[m]){
             temp[i]=arr[l];
             l++;
         }
         else{
             temp[i]=arr[m];
             m++;
         }
         i++;
    }

    if(l>mid){
         for(k=m;k<=high;k++){
             temp[i]=arr[k];
             i++;
         }
    }
    else{
         for(k=l;k<=mid;k++){
             temp[i]=arr[k];
             i++;
         }
    }

    for(k=low;k<=high;k++){
         arr[k]=temp[k];
    }
}
```

Ordenación Rápida de Hoare (Quick Sort)

Esta es probablemente la técnica más rápida conocida. Fue desarrollada por C.A.R. Hoare en 1960. El algoritmo original es recursivo, pero se utilizan versiones iterativas para mejorar su rendimiento (los algoritmos recursivos son en general más lentos que los iterativos, y consumen más recursos). El algoritmo fundamental es el siguiente:

- Eliges un elemento de la lista. Puede ser cualquiera. Lo llamaremos elemento de división.
- Buscas la posición que le corresponde en la lista ordenada (explicado más abajo).
- Acomodas los elementos de la lista a cada lado del elemento de división, de manera que a un lado queden todos los menores que él y al otro los mayores (explicado más abajo también). En este momento el elemento de división separa la lista en dos sublistas (de ahí su nombre).
- Realizas esto de forma recursiva para cada sublista mientras éstas tengan un largo mayor que 1. Una vez terminado este proceso todos los elementos estarán ordenados.
- Una idea preliminar para ubicar el elemento de división en su posición final sería contar la cantidad de elementos menores y colocarlo un lugar más arriba. Pero luego habría que mover todos estos elementos a la izquierda del elemento, para que se cumpla la condición y pueda aplicarse la recursividad. Reflexionando un poco más se obtiene un procedimiento mucho más efectivo. Se utilizan dos índices: i, al que llamaremos contador por la izquierda, y j, al que llamaremos contador por la derecha. El algoritmo es éste:
  - Recorres la lista simultáneamente con i y j: por la izquierda con i (desde el primer elemento), y por la derecha con j (desde el último elemento).
  - Cuando lista[i] sea mayor que el elemento de división y lista[j] sea menor los intercambias.
  - Repites esto hasta que se crucen los índices.
  - El punto en que se cruzan los índices es la posición adecuada para colocar el elemento de división, porque sabemos que a un lado los elementos son todos menores y al otro son todos mayores (o habrían sido intercambiados).
  - Al finalizar este procedimiento el elemento de división queda en una posición en que todos los elementos a su izquierda son menores que él, y los que están a su derecha son mayores.

Tiempo de Ejecución:

Caso promedio. La complejidad para dividir una lista de n es O(n). Cada sublista genera en promedio dos sublistas más de largo n/2. Por lo tanto la complejidad se define en forma recurrente como:

$f(1) = 1$

$f(n) = n + 2\ f(n/2)$

La forma cerrada de esta expresión es:

$f(n) = n \log_2 n$

Es decir, la complejidad es $O(n \log_2 n)$.

El peor caso ocurre cuando la lista ya está ordenada, porque cada llamada genera sólo una sublista (todos los elementos son menores que el elemento de división). En este caso el rendimiento se degrada a $O(n2)$. Con las optimizaciones mencionadas arriba puede evitarse este comportamiento.

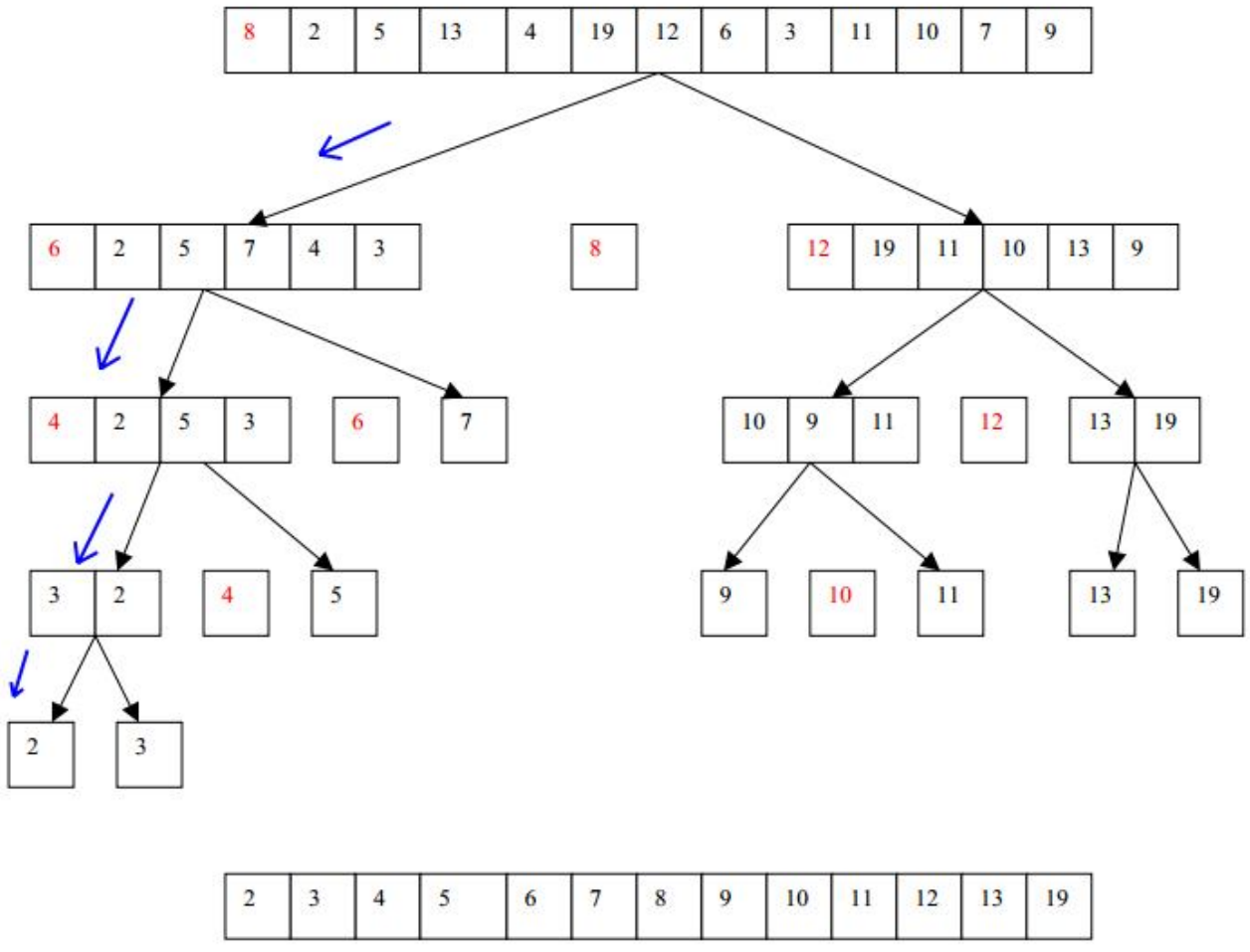
Ventajas:

- Muy rápido
- No requiere memoria adicional.

Desventajas:

- Implementación un poco más complicada.
- Recursividad (utiliza muchos recursos).
- Mucha diferencia entre el peor y el mejor caso.

Ejemplo :



Implementación en C :

```
void quicksort(int *x,int first,int last){
    int pivot,j,temp,i;
     if(first<last){
         pivot=first;
         i=first;
         j=last;

         while(i<j){
             while(x[i]<=x[pivot]&&i<last)
                 i++;
             while(x[j]>x[pivot])
                 j--;
             if(i<j){
                 temp=x[i];
                  x[i]=x[j];
                  x[j]=temp;
             }
         }
         temp=x[pivot];
         x[pivot]=x[j];
         x[j]=temp;
         quicksort(x,first,j-1);
         quicksort(x,j+1,last);
    }
}
```

Ordenación Mediante el Montículo (HeapSort)

Es un algoritmo de ordenación basado en comparaciones de elementos que utiliza un heap para ordenarlos. También se puede decir que es un algoritmo de ordenación no recursivo, no estable , con complejidad computacional.

Este algoritmo consiste en almacenar todos los elementos del vector a ordenar en un montículo y luego extraer el nodo que queda como raíz en sucesivas iteraciones obteniendo el conjunto ordenado. basa su funcionamiento en una propiedad de los montículos, por la cual, la cima siempre (depende de como se defina) contendrá el mayor o menor elemento del montículo. El proceso es :

- Se construye el montículo inicial a partir del arreglo original.
- Se intercambia la raíz con el último elemento del montículo.
- El último elemento queda ordenado.
- El último elemento se saca del montículo, no del arreglo.
- Se restaura el montículo haciendo que el primer elemento baje a la posición que le corresponde, si sus hijos son menores.
- La raíz vuelve a ser el mayor del montículo.
- Se repite el paso 2 hasta que quede un solo elemento en el montículo.

Tiempo de Ejecución:

- El orden de ejecución para el peor caso es $O(N \cdot \log(N))$.

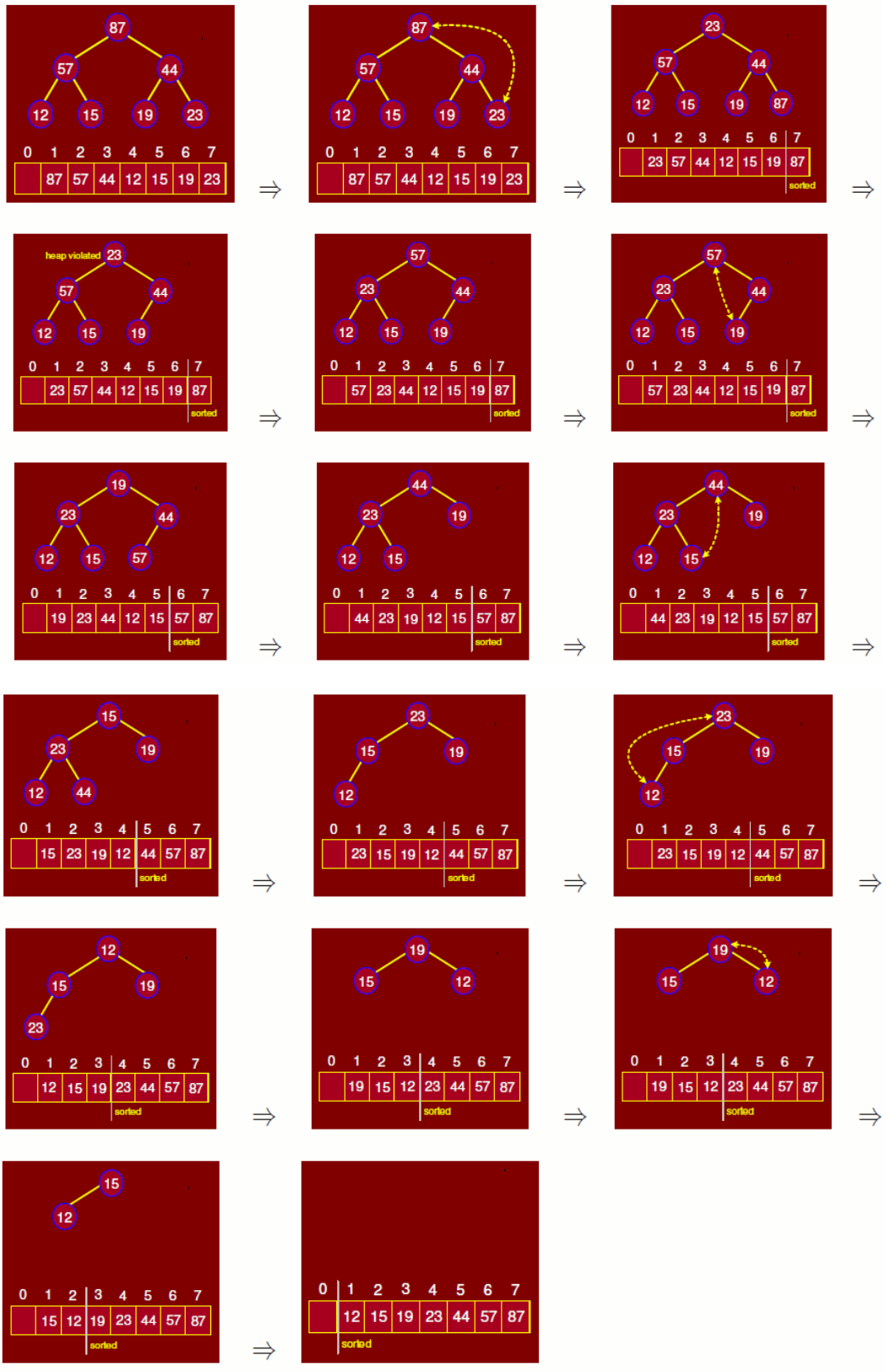
Ventajas:

- La principal ventaja es que este método funciona mas efectivamente con datos desordenados.
- Su desempeño es en promedio tan bueno como el Quicksort y se comporta mejor que este último en los peores casos.
- No utiliza memoria adicional.

Desventajas:

- No es estable, ya que se comporta de manera ineficaz con datos del mismo valor.
- Método más complejo.

Ejemplo :



Implementación en C :

```
#include <stdio.h>
#include <stdlib.h>

// A heap has current size and array of elements
struct MaxHeap
{
    int size;
    int* array;
};

// A utility function to swap to integers
void swap(int* a, int* b) { int t = *a; *a = *b;  *b = t; }

// The main function to heapify a Max Heap. The function
// assumes that everything under given root (element at
// index idx) is already heapified
void maxHeapify(struct MaxHeap* maxHeap, int idx)
{
    int largest = idx;  // Initialize largest as root
    int left = (idx << 1) + 1;  // left = 2*idx + 1
    int right = (idx + 1) << 1; // right = 2*idx + 2

    // See if left child of root exists and is greater than
    // root
    if (left < maxHeap->size &&
        maxHeap->array[left] > maxHeap->array[largest])
        largest = left;

    // See if right child of root exists and is greater than
    // the largest so far
    if (right < maxHeap->size &&
        maxHeap->array[right] > maxHeap->array[largest])
        largest = right;

    // Change root, if needed
    if (largest != idx)
    {
        swap(&maxHeap->array[largest], &maxHeap->array[idx]);
        maxHeapify(maxHeap, largest);
    }
}

// A utility function to create a max heap of given capacity
struct MaxHeap* createAndBuildHeap(int *array, int size)
{
    int i;
    struct MaxHeap* maxHeap =
              (struct MaxHeap*) malloc(sizeof(struct MaxHeap));
    maxHeap->size = size;   // initialize size of heap
    maxHeap->array = array; // Assign address of first element of array

    // Start from bottommost and rightmost internal mode and heapify all
    // internal modes in bottom up way
    for (i = (maxHeap->size - 2) / 2; i >= 0; --i)
        maxHeapify(maxHeap, i);
    return maxHeap;
}

// The main function to sort an array of given size
void heapSort(int* array, int size)
{
    // Build a heap from the input data.
```

```
    struct MaxHeap* maxHeap = createAndBuildHeap(array, size);

    // Repeat following steps while heap size is greater than 1.
    // The last element in max heap will be the minimum element
    while (maxHeap->size > 1)
    {
        // The largest item in Heap is stored at the root. Replace
        // it with the last item of the heap followed by reducing the
        // size of heap by 1.
        swap(&maxHeap->array[0], &maxHeap->array[maxHeap->size - 1]);
        --maxHeap->size;  // Reduce heap size

        // Finally, heapify the root of tree.
        maxHeapify(maxHeap, 0);
    }
}

// A utility function to print a given array of given size
void printArray(int* arr, int size)
{
    int i;
    for (i = 0; i < size; ++i)
        printf("%d ", arr[i]);
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int size = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, size);

    heapSort(arr, size);

    printf("\nSorted array is \n");
    printArray(arr, size);
    return 0;
}
```