- Merge Sort

Complejidad: n*log(n)
Complejidad espacial: O(n)

Hace uso de dos funciones: *mergeSort* y *mezclar*

```
Función mezclar(lista, int ini,int med, int fin): //recibe lista con índices de 2 sublistas
    lista result_lista; // lista a retornar (del mismo size)
    int izq=ini
    int der=med+1
    int ind=0
    Mientras izq<=med && der<=fin:  // mientras que los indices no superen límite
        Si lista[izq]<lista[der]: // la 1era sublista tiene el menor elem
            result_lista[ind] = lista[izq]
            izq++
            ind++
        Sino:
            result_lista[ind] = lista[der]
            der++
            ind++

    // Faltaría que se añadan los que faltan a lista_result
    Si izq<med: // No es necesario
        Desde i=izq hasta i=med:
            result_lista[ind] = lista[i]
            i++
    Si der<fin:  // Tampoco es necesario
        Desde i=der hasta i=fin:
            result_lista[ind] = lista[i]
            i++
    Retorna result_lista
```

```
Funcion MergeSort(&lista, int izq, int der): //recibe la lista por referencia, índices extremos
    Si izq==der: // si solo tiene 1 elem
        return lista
    Sino:  // si hay mas de 1 elem
        int med = (izq+der)/2  // elem central
        MergeSort(lista,izq,med)
        MergeSort(lista,med,der)
        lista = ordenar(lista)
```

-

- Heap Sort

Complejidad: n*log(n)
Complejidad espacial: O(1)

Se basa en el uso de heap (montículo). Crea un hep por cada elemento (heapify)
El recorrido es desde el último al primero. (inverso - lo ordena de forma decreciente pero de derecha a izquierda, por eso está ordenado)

- Si en heapify el padre es menor que ambos hijos, heapSort puede hacer recorrido del primero al último
- Internamente transforma de array a BST

Hace uso de dos funciones: *heapify* y *heapSort*

```
Función heapify(&array, size, ind_raiz):
        raiz = ind_raiz  // obtiene el índice de la raíz
        int left = 2*raiz + 1
        int right = 2*raiz+2

        // Compara el hijo izquierdo con la raíz del subárbol
        Si left<size y array[raiz]<array[left]: // se valida que el índice no sobrepase n
                raiz = left;

        // Compara el hijo derecho con la raíz del subárbol
        Si right<size y array[raiz]<array[right]:
                raiz = right;

        // Si la raiz "inicial" no es igual a la final
        Si ind_raiz!=raiz:
                Intercambia array[raiz] y array[ind_raiz]
                heapify(array,size,raiz) // llamado recursivo (se detiene cuando la raiz
                ya es más grande que sus hijos)


Función heapSort(& array, n):
        Desde i=n/2-1 hasta i=0, con paso -1: // se construye heap máximo
                // (se considera el caso en el que cada uno tiene máximo un hijo)
                heapify(array,n,i)

        // El elemento máximo ya se encuentra al inicio, lo demás no se sabe

        Desde i=n-1 hasta i=1, con paso -1:
                Intercambia array[0] con array[i]
                heapify(array,i,0)
```

- Quick Sort

Complejidad: n*log(n)
Complejidad espacial: O(n) - peor caso , O(log(n)) - caso promedio

Se define un pivote, **divide y vencerás**. A la izquierda se tiene los elementos menores al pivote y, a la derecha, a los mayores

Tiene dos funciones: *pivote* y *quickSort*

```
Función pivote(&array, izq, der): // la implementación es muy variable
        int piv = array[der] // eleccion del pivote
        int i = izq;

        Desde j=inicio hasta j=der-1, con paso +1:
                Si array[j]<piv:
                        Intercambia array[i] con array[j]
                        i++
        Intercambia array[i] con array[j] // pone al pivote en su posición correcta
        Retorna i // indice del pivote


Función quickSort(&array, inicio, fin):
        Si inicio<fin: // se realiza llamado recursivo
                int piv = pivote(array, inicio,fin);
                quickSort(array,inicio,piv)
                quickSort(array,piv+1,fin)
```

-

- Bubble Sort

Complejidad: n2**
Complejidad espacial: O(1)

Ordenamiento burbuja. Es más sencillo por su *complejidad estructural*
Hace uso de una función *bubbleSort*

Función bubbleSort(&array, inicio, fin):
    Para i=inicio, Hasta i=fin-1, con paso +1:
        Para j=i+1, Hasta j=fin, con paso +1:
            Si array[i]>array[j]:
                Intercambia array[i] y array[j]

Algoritmo con complejidad lineal en el mejor de los casos:

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

* Se debe implementar una variable booleana que indique si se encuentra desordenado

-

- Insertion Sort

Complejidad: O(n2)**
Complejidad espacial: O(1)

Recorre el arreglo. Inicia asumiendo que el primer elemento ya está ordenado. Evalúa a partir del segundo: si no está ordenado, los elementos se desplazan a la derecha y el elemento que debe ir se pone en su lugar. En cada iteración, asume que los elementos que se encuentran a la izquierda ya están ordenados, por eso inicia desde el segundo (índice 1)

- Es estable (conserva orden relativo de los elementos)
- Para pocos datos, la complejidad promedio es menor que n*log(n)

Usa la función *insertionSort*

```
Función insertionSort(&array, size):
	Desde i=1, hasta i=size-1, con paso +1:
		int value = array[i]
		int temp = i-1   // se recorre lista al revés
		Mientras temp>=0 y array[temp]>value: // se debe ordenar
			//elementos se desplazan a la derecha
			array[temp+1] = array[temp]
			temp-=1
		array[i] = value // el elemento se inserta en la posición que debe ir
```

-

- Counting Sort

Complejidad: O(n+k) ***k:rango de valores posibles (max-min+1)***
Complejidad espacial: O(n+k)

Ordena los elementos de acuerdo al número de apariciones de cada uno. Es eficiente cuando se trabaja con un rango pequeño de elementos (menor que n).

- En su forma base, solo sirve para ordenar enteros no negativos
- Se puede modificar para ordenar cualquier tipo de dato (implementando una función de mapeo que retorne un key único entre 0 y n-1)
- Es necesario saber el rango de valores (por eso se halla el máximo y el mínimo)

Usa la función *countingSort*

Función countingSort(array, size):
  int max = max(array)
  int min = min(array)
  int rango = max - min + 1 // se obtiene el rango entre los elementos
  frec = array [rango] // se crea un array para almacenar las frecuencias

  Para i = 0, Hasta i = size, con paso - 1: // para contabilizar
    frec[array[i]-min] ++; // se incrementa en 1 la frecuencia en array frec
    // *frec* representa la frecuencia de elementos desde (**min** hasta **max**)

  Para j = 0, Hasta j = size, con paso - 1: // para la frecuencia acumulativa
    frec[array[j]] = frec[array[j]] + frec[array[j-1]]

  lista result[size] // lista resultante
  Para ind=size-1,hasta ind=0, con paso -1: //inversa para preservar estabilidad
    frec[array[ind]-min]-= 1 // disminuye la frecuencia
    result[frec[array[ind]-min]] = array[ind] // disminuye la frecuencia

  Para i = 0, i=size-1, con paso 1: // ordena el arreglo
    array[i] = result[i]

- Radix Sort

Complejidad: O(n*k) : k = número de dígitos para representar al mayor elemento
Complejidad espacial: O(n+k)

Ordena los elementos de una lista por dígitos (desde el menos hasta el más significativo).

- Hace uso de *countingSort*
- También puede usarse para ordenar strings (no es lo recomendable)

```
Función getdigit(numero, exponente):  // considerar iniciará con exponente=0
        int d=1
        Desde i=1, hasta i=d, con paso 1:
                d*=10
        return (numero/d)%10


Función radixSort(&lista, size):
        int max = Máximo elemento
        int digitos = Cantidad de dígitos de max
        Desde d=0, hasta d=digitos, con paso 1:
                lista conteo[10]   // para contar las 10 posibilidades (*dígito)

                Desde elem=lista[0], hasta elem=lista[size-1]:  // frecuencia
                        conteo[get_digit(elem,d)];  // contabiliza dígito en pos d

                Desde ind=1,hasta ind<10, con paso 1:
                        conteo[i] += conteo[i-1]     // frecuencia acumulada

                lista result[size]  // para "ordenar" por dígito
                Desde i=size-1, hasta i=0, con paso -1:
                        digit = get_digit(lista[i])   // dígito que se evaluó
                        conteo[digit] -= 1     // cantidad disminuye
                        result[conteo[digit]] = lista[i]   // orden respecto al dígito
```

-

- Intro Sort

Complejidad: n*log(n)
Complejidad espacial: log(n)

Combinación de *quickSort*, *heapSort*, *insertionSort*
- Empieza usando quickSort
- Si el nivel de recursión es profundo, cambia a heapSort(evita peor caso en quickSort)
- Cuando son muy pocos elementos se cambia a insertionSort (es estable y, en este caso, su complejidad promedio es menor que n*log(n))

```
Función distribuir(&array, izq, der, limite):
        Si der-izq>=16:
                Si limite==0:  // cuando se llega al límite, va por heapSort
                        heapSort(array,der-izq+1)
                        return;
                Sino:  // implementa quickSort - indirectamente
                        int pivo = %pivote% // se define al pivote
                        distrribuir(array, izq, pivo-1, limite-1);
                        distribuir(arr, pivo+ 1, der, limite-1);

        Sino:  // cuando tiene menos de 16 elementos, se llama a insertion
                insertionSort(array,der-izq+1);
```

- Selection Sort

  Complejidad: n2**
  Complejidad espacial: O(1)

  Envía los elementos menores a la izquierda. Es inestable (no conserva posición de elementos al ordenarlos)

```
Función selectionSort(& array, int size):
        Desde i=0, i<size, con paso 1:
                menor = array[i]   // se obtiene el menor
                Desde j=i+1, j<size, con paso 1:
                        Si array[menor]>array[j]:
                                Intercambia array[menor] y array[j]
                Intercambia array[i] con array[menor] //para tener el menor en "i"
```

-

- Bucket Sort

Complejidad: O(n+k) : k=número de buckets
Complejidad espacial: O(n+k) <> O(n)

Distribuye la lista en un grupo de cubetas o “buckets”
- Requiere conocer el rango de valores
- Lo óptimo sería *k=sqrt(n)*
- En general, se puede probar con valores *k* variados
- Puede usar otro algoritmo para ordenar sus buckets
- También puede implementarse una *lista enlazada* (se inserta el elemento en la posición adecuada para que termine ordenado)

```
Función bucketSort(&lista, size):
        int max, min; // obtener máximo y mínimo
        int k ;  // definir n° de buckets
        vector buckets[k];  // “matriz” para representar los buckets

        Desde i=0, hasta i=size-1, con paso 1: // se recorre la lista
                index = floor((arr[i] - min_value) / range * k)
                Agregar lista[i] a buckets[k]

        // este paso solo es necesario si no se implementa lista enlazada
        Desde i=0, hasta i=k-1, con paso 1:  // ordenar cada bucket
                Ordenar bucket[i] (con otro algoritmo - Insertion Sort)

        int index = 0 // indice para cubetas
        Desde i=0, hasta i=k-1, con paso 1:  // concatenación de buckets
                Desde j=0, hasta j=buckets[ i ].size(), con paso 1:
                        lista[index++] = buckets[ i ][ j ]  // se ordena la lista
```