Sorting

algunos algoritmos de ordenamiento por comparación (arreglos)

- **Bubble Sort:** en cada pasada el elemento mayor sube (burbujea) al último lugar

- **Select Sort:** en cada pasada se busca el elemento menor (o mayor) y se lo coloca en el primer (o último) lugar

- **Insert Sort:** cada vez se inserta un elemento entre los ya ordenados

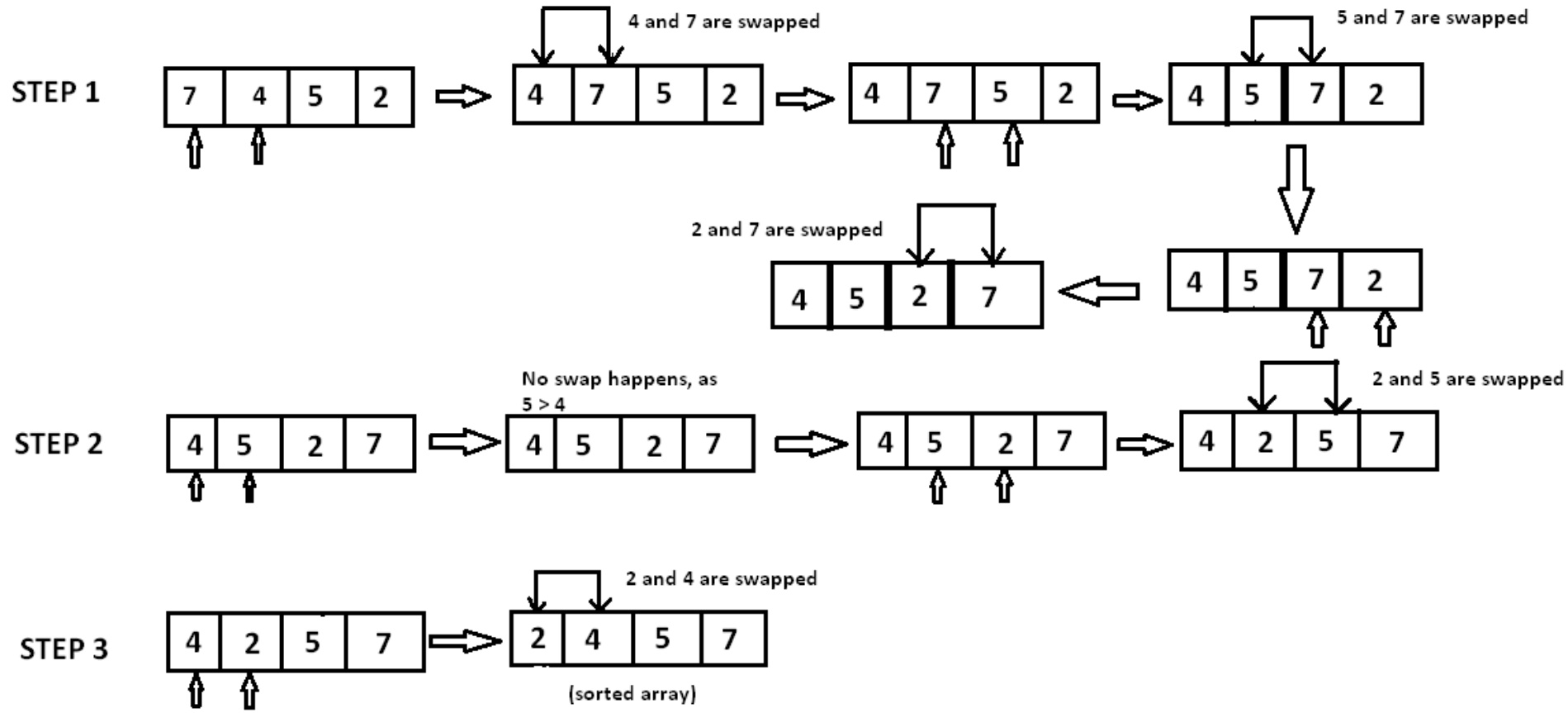
- **Merge Sort:** dividir en mitades, ordenar cada mitad, mezclar las mitades ordenadamente.

BUBBLE SORT

```
//pre: cantidad de posiciones de vec
//pos: ordena vec de menor a mayor
void bubbleSort(int * vec, int n) {
    for (int i = 0; i < n-1; i++)
        for (int j = 0;  j < n-i-1;  j++){
            if (vec[j] > vec[j + 1])
              swap(vec[j], vec[j+1]);
}
```

Estrategia: *Repetidamente compara pares de elementos adyacentes y los intercambia si están desordenados.*

Diagrama BubbleSort

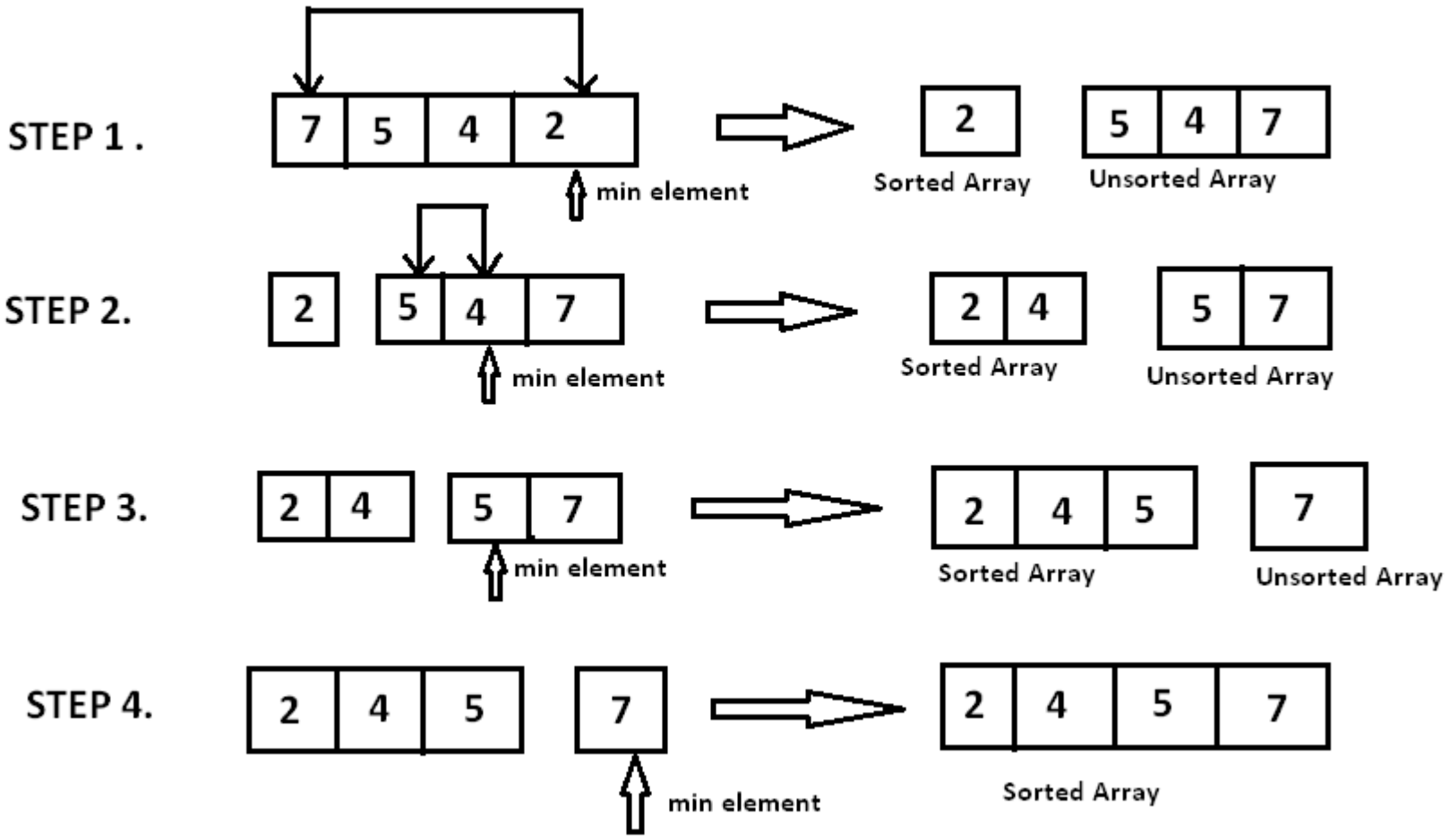


SELECTION SORT

```
void selectionSort( int *vec, int n ){
        for ( int i = 0 ; i < n-1 ; i++ ){
                int posMin = i;
                for ( int j = i+1 ; j < n ; j++ ){
                        if (vec[j]<vec[posMin])
                                posMin=j;
                }
                if (posMin != i)
                        swap(vec[j], vec[posMin]);
        }
}
```

Estrategia: *En repetidas pasadas se busca el mínimo elemento y se lo coloca en la posición correcta para conseguir el orden.*

Diagrama SelectionSort

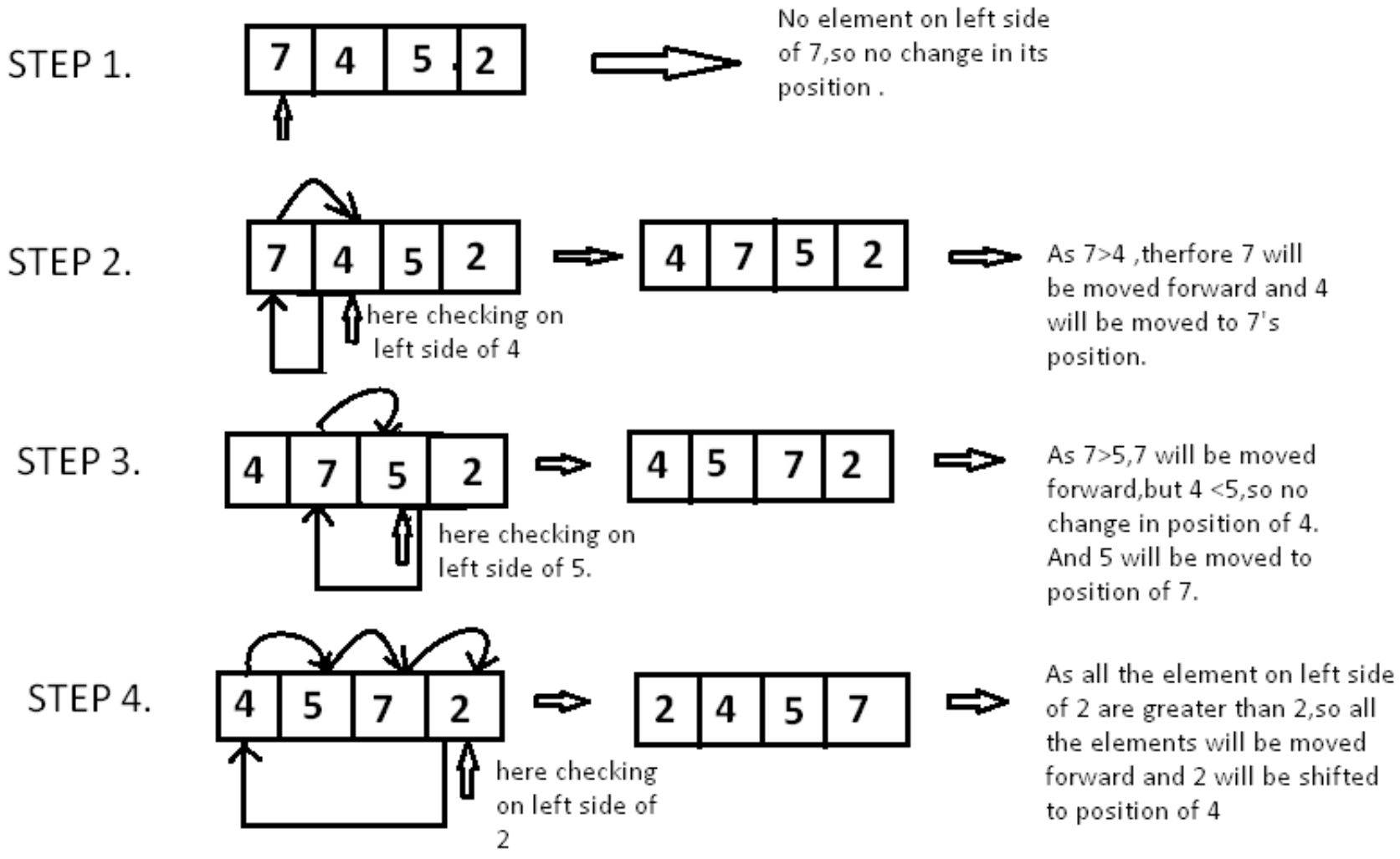


INSERTION SORT

```
void   insertionSort( int *vec, int n ){
      for ( int i = 0 ; i < n; i++ ){
          int valor = vec[i];
          int j=i;
          for( ; j > 0  && valor < vec[j-1];  j-- ){
                    vec[j] = vec[j-1];
          }
          vec[j] = valor;
      }
}
```

Estrategia: *En cada iteración se toma un elemento y se busca su posición correcta según el orden deseado, en la parte izquierda, intercambiando valores y ordenando por partes (en cada iteración la parte izquierda, cada vez mayor, va quedando ordenada).*

Diagrama InsertionSort



MERGE SORT

```
void mergeSort( int *vec, int izq, int der ){
        if ( izq < der ){
                int medio = (izq + der) / 2;
                mergeSort(vec, izq, medio);
                mergeSort(vec, medio+1, der);
                merge(vec, izq, medio, der);
        }
}
```

MERGE SORT

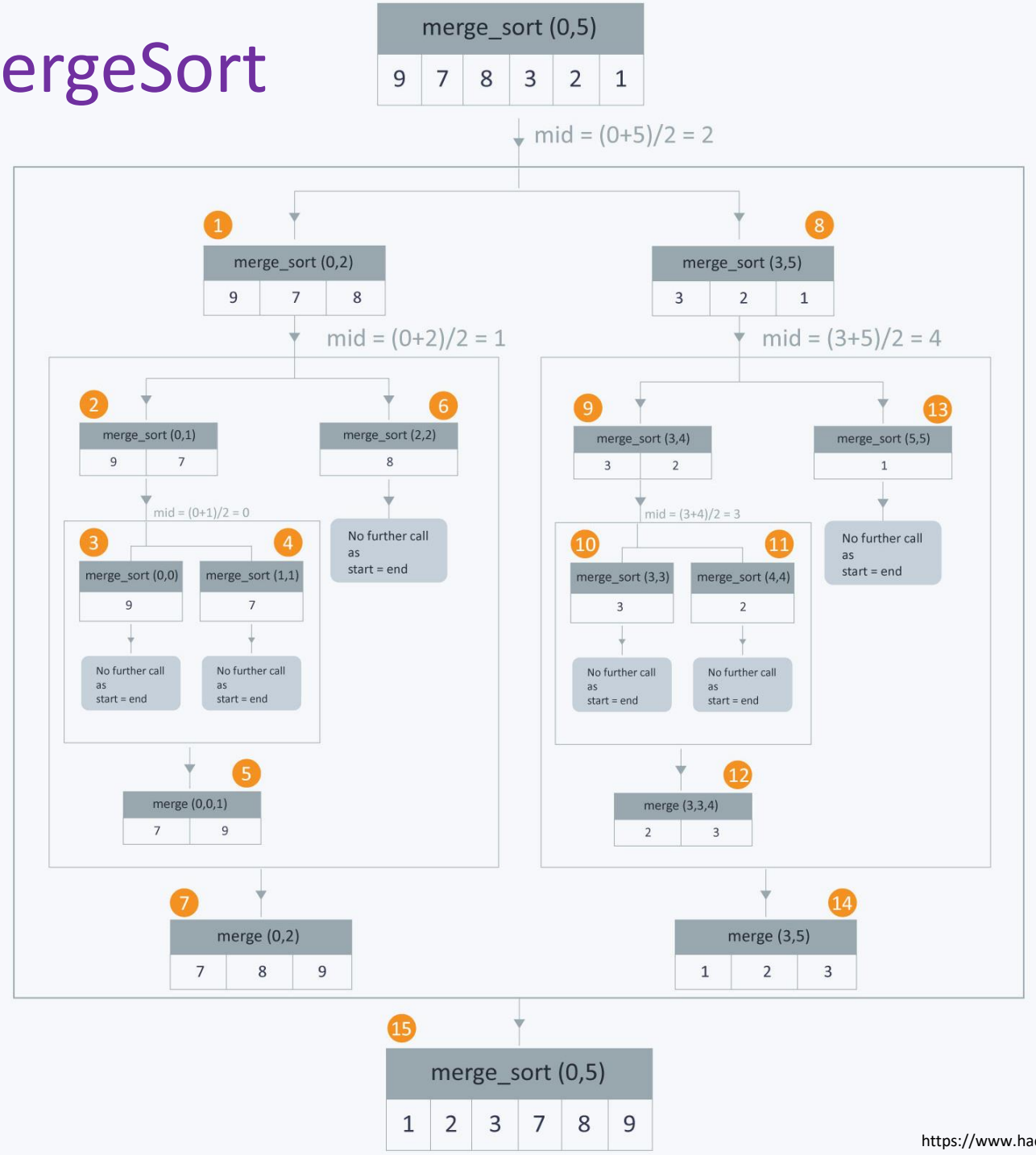
```
void merge ( int *vec, int ini, int med, int fin ){
        int i, j, k;
        int n1 = med-ini+1; //largo de vector izquierdo
        int n2 = fin-med;     //largo de vector derecho
         //crea vectores auxiliares
        int vecIzq[n1];
        int vecDer[n2];
         //copia datos a los vectores auxiliares
        for (i=0; i<n1 ; i++)
                vecIzq[i] = vec[ini + i];
        for (j=0; j<n2 ; j++)
                 vecDer[j] = vec[med + 1 + j];
//sigue en la próxima
```

usar memoria dinámica

MERGE SORT

```
//viene de merge
        //ordena los datos del arreglo original
        i=0; j=0; k=ini;
        while (i < n1 && j< n2){
                if (vecIzq[i] <= vecDer[j]){
                        vec[k]=vecIzq[i];       i++;
                }
                else{
                        vec[k]=vecDer[j];       j++;
                }
                k++;
        }
        while (i < n1){ vec[k]=vecIzq[i];  i++; k++;}
        while (j< n2){ vec[k]=vecDer[j];  j++; k++;}
}
```

Diagrama MergeSort



https://www.hackerearth.com

Análisis informal

Tiempo de Ejecución

- Mergesort se invoca recursivamente $2\log_2 n$ veces (tantas veces como se puede dividir el arreglo a la mitad)
- Mergesort invoca a Merge una vez
- Merge realiza O(n) comparaciones (en cada pasada)
- En total: $O(n.\log_2 n)$ comparaciones

Espacio en Memoria

- Mergesort requiere espacio adicional para los n elementos

Bibliografía
(para algoritmos de ordenamiento)

- **Estructuras de Datos y Análisis de Algoritmos**
  Mark Allen Weiss
  (Capítulo 7: Sorting**)**

- **Estructuras de Datos y Algoritmos.**
  A. Aho, J. E. Hopcroft & J. D. Ullman
  (Capítulo 8: Clasificación**)**