ESTRUTURAS DE DADOS

2023/2024

Aula 08

- Pesquisa Linear e Binária
- *Insertion Sort*
- *Selection Sort*
- *Bubble Sort*
- *Quick Sort*
- *Merge Sort*

Ricardo Santos | rjs@estg.ipp.pt

ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Pesquisas

- A pesquisa é o processo de encontrar um elemento-alvo num grupo de itens, ou então determinar que não está presente
- Essa tarefa exige comparar repetidamente o alvo pelos candidatos de entre o grupo de pesquisa
- Uma ordenação eficiente não executa mais comparações do que tem obrigatoriamente de realizar
- O tamanho do grupo de pesquisa é um factor

A Interface `Comparable`

- Queremos definir os algoritmos de forma a poderem pesquisar por qualquer conjunto de objectos

- Portanto, vamos pesquisar objectos que implementem a interface `Comparable`

- Esta interface contém um método `compareTo`, que é projectado para retornar um número inteiro que especifica a relação entre dois objectos:

```
obj1.compareTo(obj2)
```

- A chamada retorna um número inferior, igual ou superior a **`0`** se **`obj1`** é inferior, igual ou superior a **`obj2`**, respectivamente

- Todos os métodos apresentados são métodos static de uma classe, por exemplo, `SortingandSearching`
- Esta classe faz uso do tipo genérico **`T`**
- Para estes métodos, podemos fazer a distinção ainda que **`T`** será uma subclasse de `Comparable (T extends Comparable)`

```
public static <T extends
      Comparable<? super T>> boolean
      linearSearch(T[] data, int min,
      int max, T target)
```

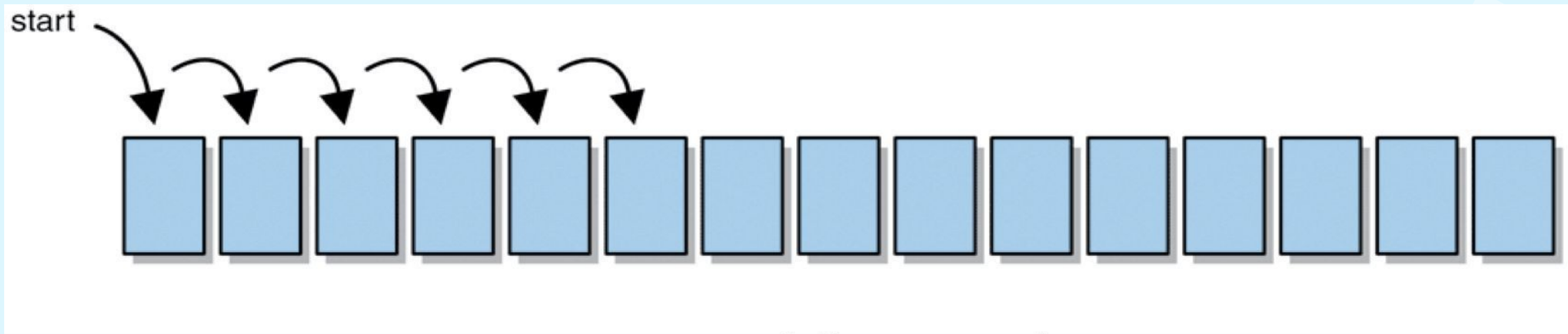
- Isto significa que o *array* do tipo `T` deve conter objectos que são comparáveis entre si
- Isso pode significar que são da mesma classe ou descendentes do mesmo ancestral
- Uma chamada para esse método seria:

```
SortingandSearching.linearSearch(
            targetarray, min, max,target)
```

Pesquisa Linear

- A pesquisa linear examina simplesmente cada item do grupo de pesquisa, um de cada vez, até que o alvo seja encontrado ou até que o grupo de pesquisa seja esgotado
- Esta abordagem não assume que os itens do grupo de pesquisa estejam em alguma ordem em particular
- Apenas temos de ser capazes de analisar cada elemento em separado (de forma linear)
- É bastante fácil de entender, mas não muito eficiente

Pesquisa Linear



```
/**
 * Searches the specified array of objects using a
 * linear search algorithm.
 *
 * @param data    the array to be sorted
 * @param min     the integer representation of the min value
 * @param max     the integer representation of the max value
 * @param target  the element being searched for
 * @return        true if the desired element is found
 */
public static <T extends Comparable<? super T>>  boolean
            linearSearch (T[] data, int min, int max, T target)
{
   int index = min;
   boolean found = false;

   while (!found && index <= max)
   {
      if (data[index].compareTo(target) == 0)
         found = true;
      index++;
   }
   return found;
}
```

Pesquisa Binária

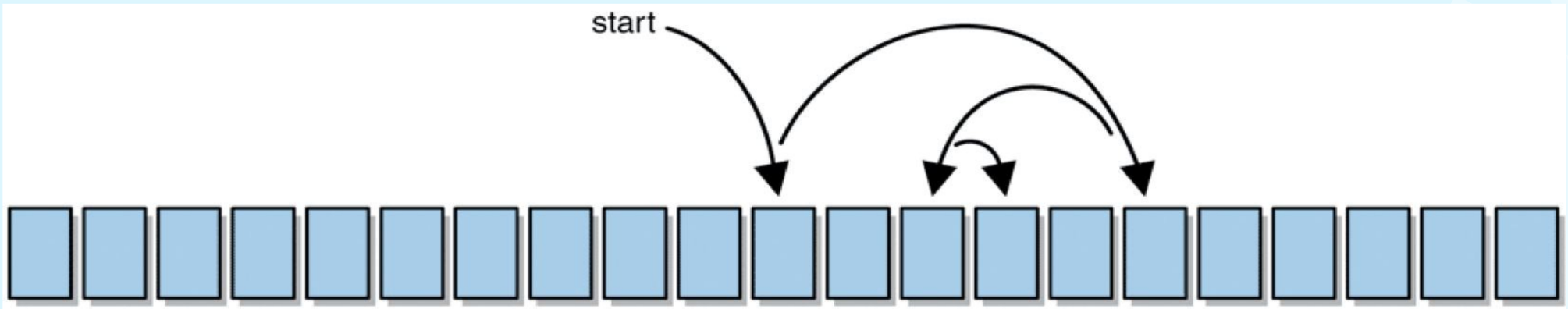
- Se o grupo de pesquisa é ordenado, então podemos ser mais eficientes do que numa pesquisa linear
- A pesquisa binária elimina grande parte do grupo de pesquisa com cada comparação
- Em vez de começar a pesquisa numa extremidade, começamos no meio

Pesquisa Binária

- Se o alvo não for encontrado sabemos que se estiver no grupo de pesquisa estará numa ou noutra metade

- Podemos, então, saltar para o meio dessa outra metade, e continuar da mesma forma

Pesquisa Binária



- Por exemplo, encontrar o número **29** na seguinte lista ordenada de números:

```
8 15 22 29 36 54 55 61 70 73 88
```

- Compare o alvo para o valor do meio **54**
- Sabemos agora que, se **29** estiver na lista, é na metade da frente da lista
- Com uma comparação, eliminamos metade dos dados
- De seguida, compare com o **22**, eliminando mais um quarto de dados, etc

- Um algoritmo de pesquisa binária é muitas vezes implementado de forma recursiva
- Cada chamada recursiva procura uma parcela menor do grupo de pesquisa
- O caso base da recursão é o acabarem os candidatos viáveis à pesquisa, o que significa que o alvo não está no grupo de pesquisa
- A qualquer momento pode haver dois valores no "meio", caso em que qualquer um pode ser usado

```
/**
 * Searches the specified array of objects using a
 * binary search algorithm.
 * @param data    the array to be sorted
 * @param min     the integer representation of the minimum value
 * @param max     the integer representation of the maximum value
 * @param target  the element being searched for
 * @return        true if the desired element is found
 */
public static <T extends Comparable<? super T>>  boolean
       binarySearch (T[] data, int min, int max, T target){

   boolean found = false;
   int midpoint = (min + max) / 2;  // determine the midpoint

   if (data[midpoint].compareTo(target) == 0)
      found = true;
   else if (data[midpoint].compareTo(target) > 0) {
      if (min <= midpoint - 1)
         found = binarySearch(data, min, midpoint - 1, target);
   }
   else if (midpoint + 1 <= max)
      found = binarySearch(data, midpoint + 1, max, target);

   return found;
}
```

Comparar Algoritmos de Pesquisa

- Em média, uma pesquisa linear examinaria **n/2** elementos antes de encontrar o alvo
- Portanto, uma pesquisa linear é **O(n)**
- O pior caso para uma pesquisa binária são **(log_2N)** comparações

- A pesquisa binária é um algoritmo logarítmico
- Tem uma complexidade de tempo **$O(\log_2 N)$**
- Mas lembrem-se que o grupo de pesquisa deve ser ordenado
- Para **N** grande, uma pesquisa binária é muito mais rápido

Ordenações

- A ordenação é o processo de organizar um grupo de itens numa ordem definida com base em critérios específicos
- Muitos algoritmos de ordenação foram concebidos
- Ordenações sequenciais requerem aproximadamente **N/2** comparações para ordenar **N** elementos
- Ordenações logarítmicas normalmente requerem **$nlog_2n$** comparações para ordenar **N** elementos

- Para isso vamos definir um problema de ordenação genérico que qualquer um dos nossos algoritmos de ordenação possa ajudar a resolver

- Tal como acontece com a pesquisa, devemos ser capazes de comparar um elemento com outro

Exemplo de Ordenação

```
public class SortPhoneList
{
   /**
    * Creates an array of Contact objects,
    * sorts them, then prints them.
    */
   public static void main (String[] args)
   {
      Contact[] friends = new Contact[7];
```

```
        friends[0] = new Contact ("Clark", "Kent", "610-555-7384");
        friends[1] = new Contact ("Bruce", "Wayne", "215-555-3827");
        friends[2] = new Contact ("Peter", "Parker", "733-555-2969");
        friends[3] = new Contact ("James", "Howlett", "663-555-3984");
        friends[4] = new Contact ("Steven", "Rogers", "464-555-3489");
        friends[5] = new Contact ("Britt", "Reid", "322-555-2284");
        friends[6] = new Contact ("Matt", "Murdock", "243-555-2837");

        SortingAndSearching.selectionSort(friends);

        for (int index = 0; index < friends.length; index++)
           System.out.println (friends[index]);
    }
}
```

```
public class Contact implements Comparable
{
   private String firstName, lastName, phone;

   /**
    * Sets up this contact with the specified information.
    *
    * @param first      a string representation of a first name
    * @param last       a string representation of a last name
    * @param telephone  a string representation of a phone number
    */
   public Contact (String first, String last, String telephone)
   {
      firstName = first;
      lastName = last;
      phone = telephone;
   }
```

```
/**
 * Returns a description of this contact as a string.
 *
 * @return  a string representation of this contact
 */
public String toString () {
   return lastName + ", " + firstName + "\t" + phone;
}

/**
 * Uses both last and first names to determine lexical ordering.
 *
 * @param other  the contact to be compared to this contact
 * @return       the integer result of the comparison
 */
   public int compareTo (Object other) {
      int result;
      if (lastName.equals(((Contact)other).lastName))
         result = firstName.compareTo(((Contact)other).firstName);
      else
         result = lastName.compareTo(((Contact)other).lastName);

      return result;
   }
}
```

Selection Sort

- O ***Selection Sort*** ordena uma lista de valores ao colocar repetidamente os valores nas suas posições finais
- Mais especificamente:
  - encontrar o menor valor da lista
  - fazer a troca com o valor na primeira posição
  - encontrar próximo menor valor da lista
  - fazer a troca com o valor na segunda posição
  - repetir até que todos os valores estejam nos seus devidos lugares

- O algoritmo pode também funcionar de forma contrária tornando-se uma decisão do programador...

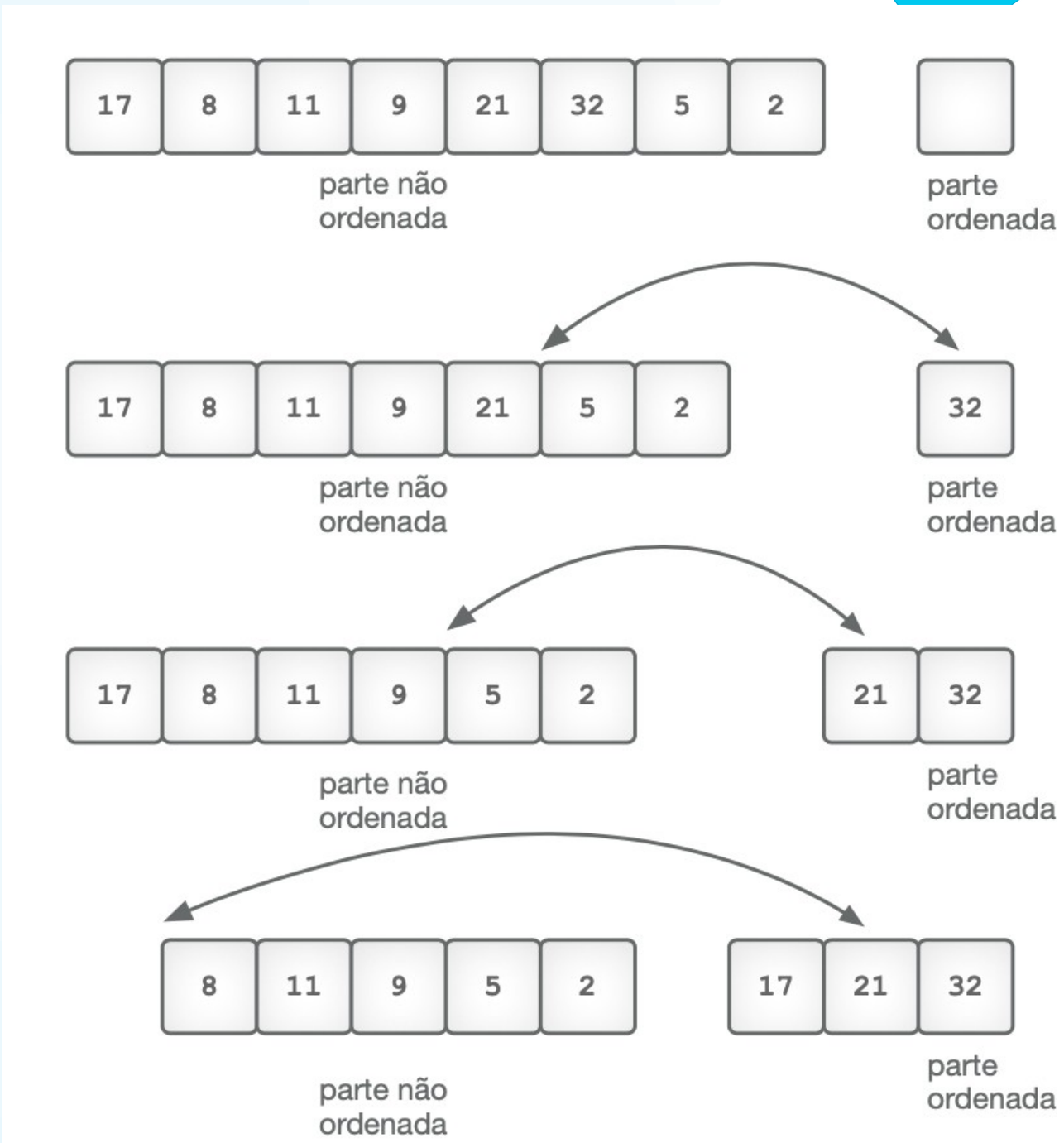
- Em vez de ser seleccionado o **menor** valor da lista pode ser seleccionado o **maior** valor da lista

(a) Configuração inicial para o ***Selection Sort***. O *array* de *input* é logicamente dividido em duas partes: ordenada e não ordenada

(b) O *array* depois do maior elemento da parte não ordenada ter sido movido para o inicio da parte ordenada (**1º passo**)

(c) O *array* depois do maior elemento da parte não ordenada ter sido movido para o inicio da parte ordenada (**2º passo**)

(d) O *array* depois do maior elemento da parte não ordenada ter sido movido para o inicio da parte ordenada (**3º passo**)



```
/**
 * Sorts the specified array of integers using the selection
 * sort algorithm.
 *
 * @param data  the array to be sorted
 */
public static <T extends Comparable<? super T>> void
          selectionSort (T[] data) {
   int min;
   T temp;
   for (int index = 0; index < data.length-1; index++){
      min = index;
      for (int scan = index+1; scan < data.length; scan++)
         if (data[scan].compareTo(data[min])<0)
            min = scan;

      /** Swap the values */
      temp = data[min];
      data[min] = data[index];
      data[index] = temp;
   }
}
```

Insertion Sort

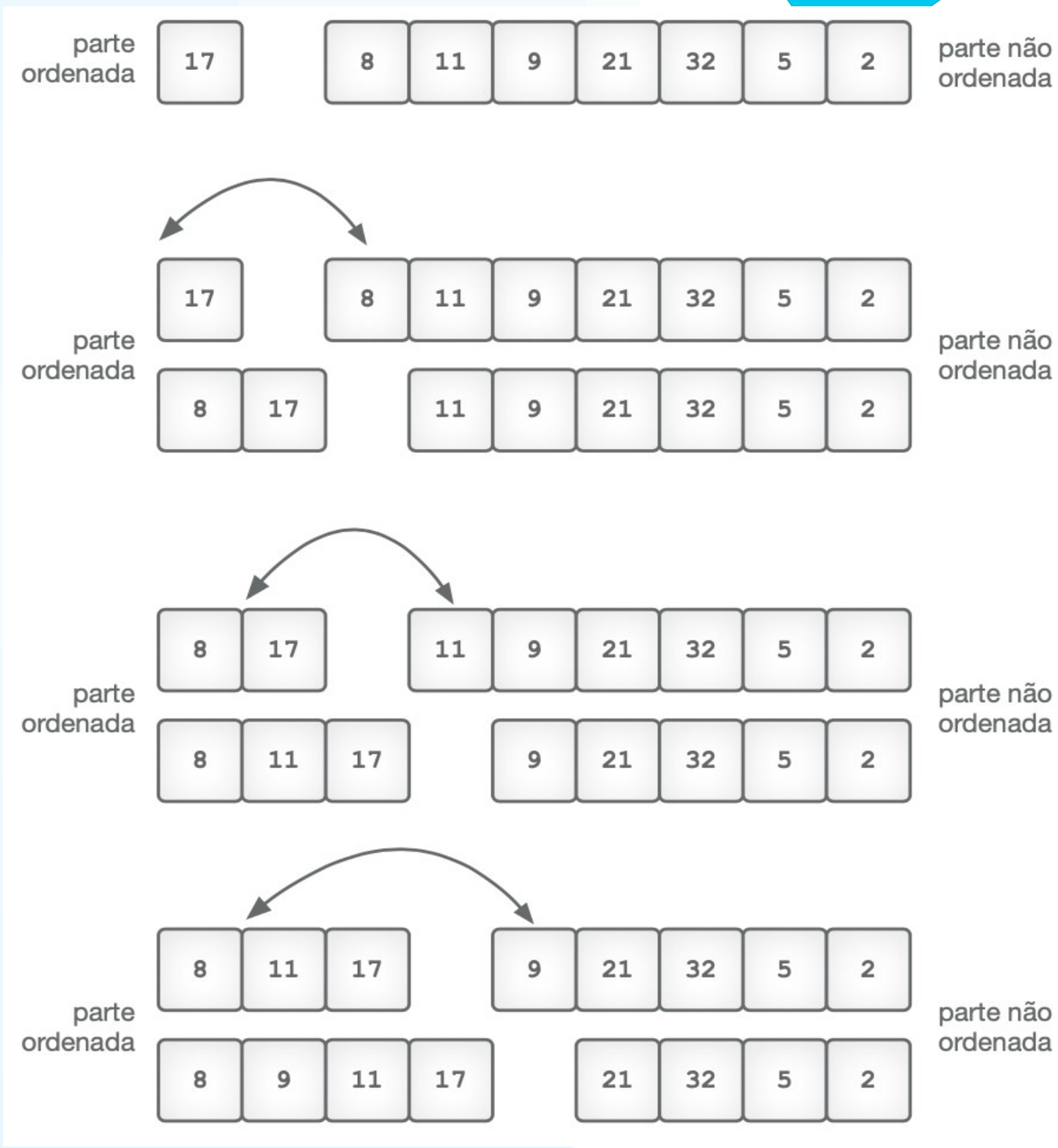
- O ***Insertion Sort*** ordena uma lista de valores ao inserir repetidamente um determinado valor num subconjunto da lista
- Mais especificamente:
  - considera o primeiro valor como uma sublista ordenada de tamanho 1
  - insere o segundo item na sublista ordenada, deslocando o primeiro item, se necessário
  - insere o terceiro item na sublista ordenada, deslocando os demais itens conforme necessário
  - repetir até que todos os valores tenham sido inseridos nas suas posições correctas

(a) A configuração inicial para o ***Insertion Sort***. O *array* de *input* é logicamente dividido em duas partes: ordenada e não ordenada.

(b) O *array* depois do primeiro valor da parte não ordenada ter sido inserido na sua posição correcta na parte ordenada (**1º passo**)

(c) O *array* depois do primeiro valor da parte não ordenada ter sido inserido na sua posição correcta na parte ordenada (**2º passo**)

(d) O *array* depois do primeiro valor da parte não ordenada ter sido inserido na sua posição correcta na parte ordenada (**3º passo**)



```
/**
 * Sorts the specified array of objects using an insertion
 * sort algorithm.
 *
 * @param data  the array to be sorted
 */
public static <T extends Comparable<? super T>> void
          insertionSort (T[] data) {
   for (int index = 1; index < data.length; index++) {
      T key = data[index];
      int position = index;

      /** Shift larger values to the right */
      while (position > 0 && data[position-1].compareTo(key) > 0){
         data[position] = data[position-1];
         position--;
      }

      data[position] = key;
   }
}
```

Bubble Sort

- O ***Bubble Sort*** ordena uma lista de valores ao comparar repetidamente os elementos vizinhos e trocando suas posições, se necessário
- Mais especificamente:
  - examina a lista, trocando elementos adjacentes se estes não estiverem ordenados; irá fazer com que o maior valor seja empurrado para o topo
  - examina a lista de novo, obtendo o segundo maior valor
  - Repetir até que todos os elementos tenham sido colocados na sua devida ordem

```
/**
 * Sorts the specified array of objects using a bubble sort
 * algorithm.
 *
 * @param data  the array to be sorted
 */
public static <T extends Comparable<? super T>> void
         bubbleSort (T[] data) {
   int position, scan;
   T temp;
   for (position =  data.length - 1; position >= 0; position--) {
      for (scan = 0; scan <= position - 1; scan++) {
         if (data[scan].compareTo(data[scan+1]) > 0) {
            /** Swap the values */
            temp = data[scan];
            data[scan] = data[scan + 1];
            data[scan + 1] = temp;
         }
      }
   }
}
```

Comparar Ordenações

- Até agora vimos três algoritmos de ordenação:
  - ***Selection Sort***
  - ***Insertion Sort***
  - ***Bubble Sort***
- Todos estes algoritmos usam ciclos aninhados e realizam aproximadamente $\mathbf{n^2}$ comparações
- São todos relativamente ineficientes
- Agora vamos estudar alguns dos algoritmos mais eficientes

Quick Sort

- O ***Quick Sort*** ordena uma lista de valores através da repartição da lista em torno de um elemento
- Mais especificamente:
  - escolhe um elemento da lista a ser o elemento de partição
  - organiza os elementos para que todos os elementos menores que o elemento da partição vão para a esquerda e os maiores para a direita
  - aplicar o algoritmo (recursivamente) a ambas as partições

- A escolha do elemento de partição é arbitrário
- Por eficiência, seria bom se o elemento de partição dividisse a lista ao meio
- No entanto o algoritmo funciona em qualquer caso
- Vamos dividir a solução em dois métodos:
  - `quickSort` - executa o algoritmo recursivo
  - `findPartition` - reorganiza os elementos em duas partições

```
/**
   * Sorts the specified array of objects using the quick sort
   * algorithm.
   *
   * @param data  the array to be sorted
   * @param min   the integer representation of the minimum value
   * @param max   the integer representation of the maximum value
   */
  public static <T extends Comparable<? super T>> void
          quickSort (T[] data, int min, int max) {
    int indexofpartition;

    if (max - min  > 0) {
       /** Create partitions */
       indexofpartition = findPartition(data, min, max);

       /** Sort the left side */
       quickSort(data, min, indexofpartition - 1);

       /** Sort the right side */
       quickSort(data, indexofpartition + 1, max);
    }
}
```

```
/**
   * Used by the quick sort algorithm to find the partition.
   *
   * @param data  the array to be sorted
   * @param min   the integer representation of the minimum value
   * @param max   the integer representation of the maximum value
   */
  private static <T extends Comparable<? super T>> int
              findPartition (T[] data, int min, int max) {
     int left, right;
     T temp, partitionelement;
     int middle = (min + max)/2;

     // use middle element as partition
     partitionelement = data[middle];
     left = min;
     right = max;

     while (left<right) {
        /** search for an element that is > the partitionelement */
        while (data[left].compareTo(partitionelement) <0 {
           left++;
        }
```

```
/** search for an element that is < the partitionelement */
while (data[right].compareTo(partitionelement) > 0)
   right--;

/** swap the elements  */
if (left<right)
{
   temp = data[left];
   data[left] = data[right];
   data[right] = temp;
   }
}

/** move partition element to partition index*/
temp = data[min];
data[min] = data[right];
data[right] = temp;

return right;
}
```

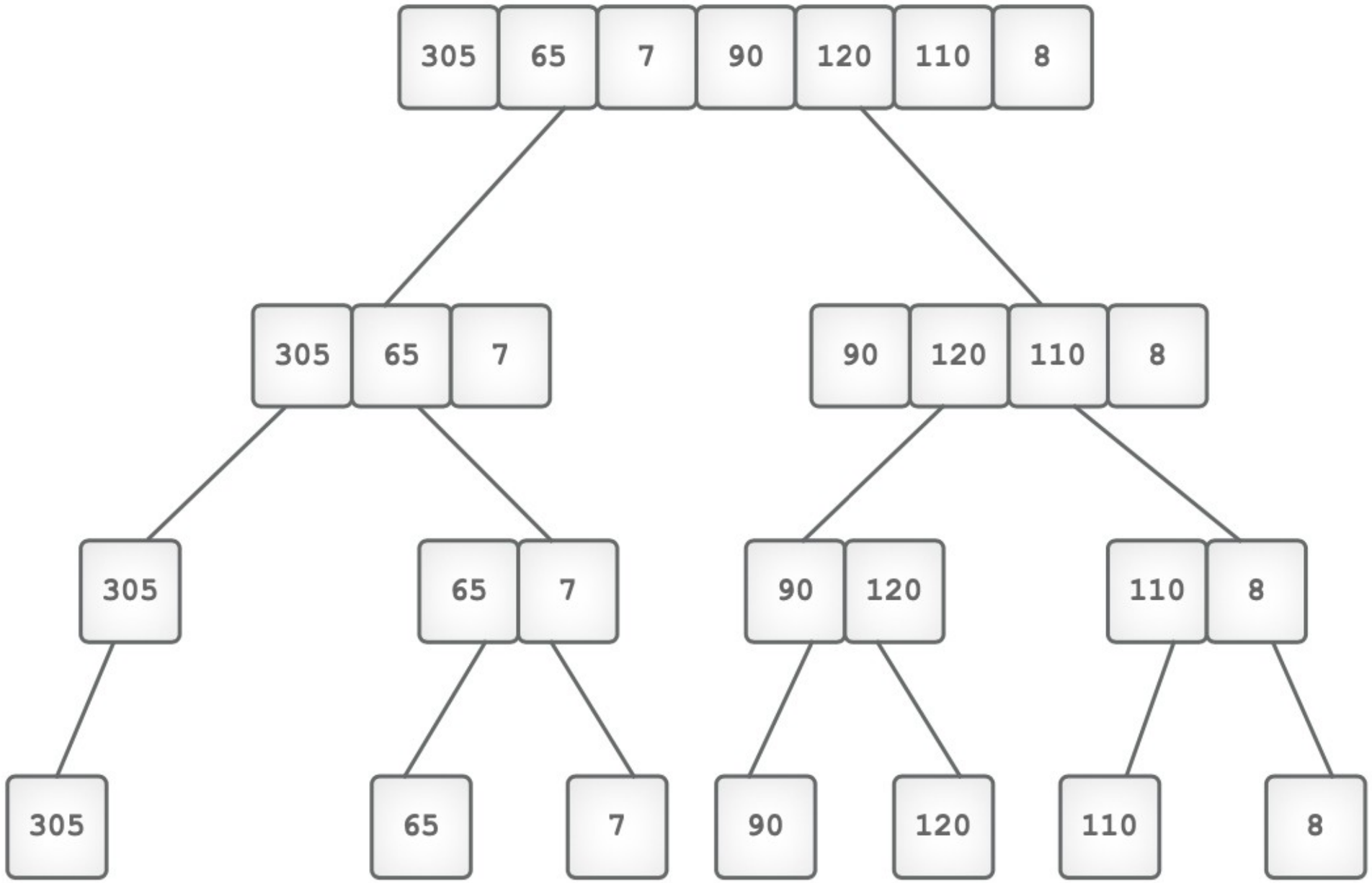
Merge Sort

- O ***Merge Sort*** ordena uma lista de valores ao dividir recursivamente a lista a meio até que cada sub-lista tenha apenas um elemento, de seguida realiza a junção das partes

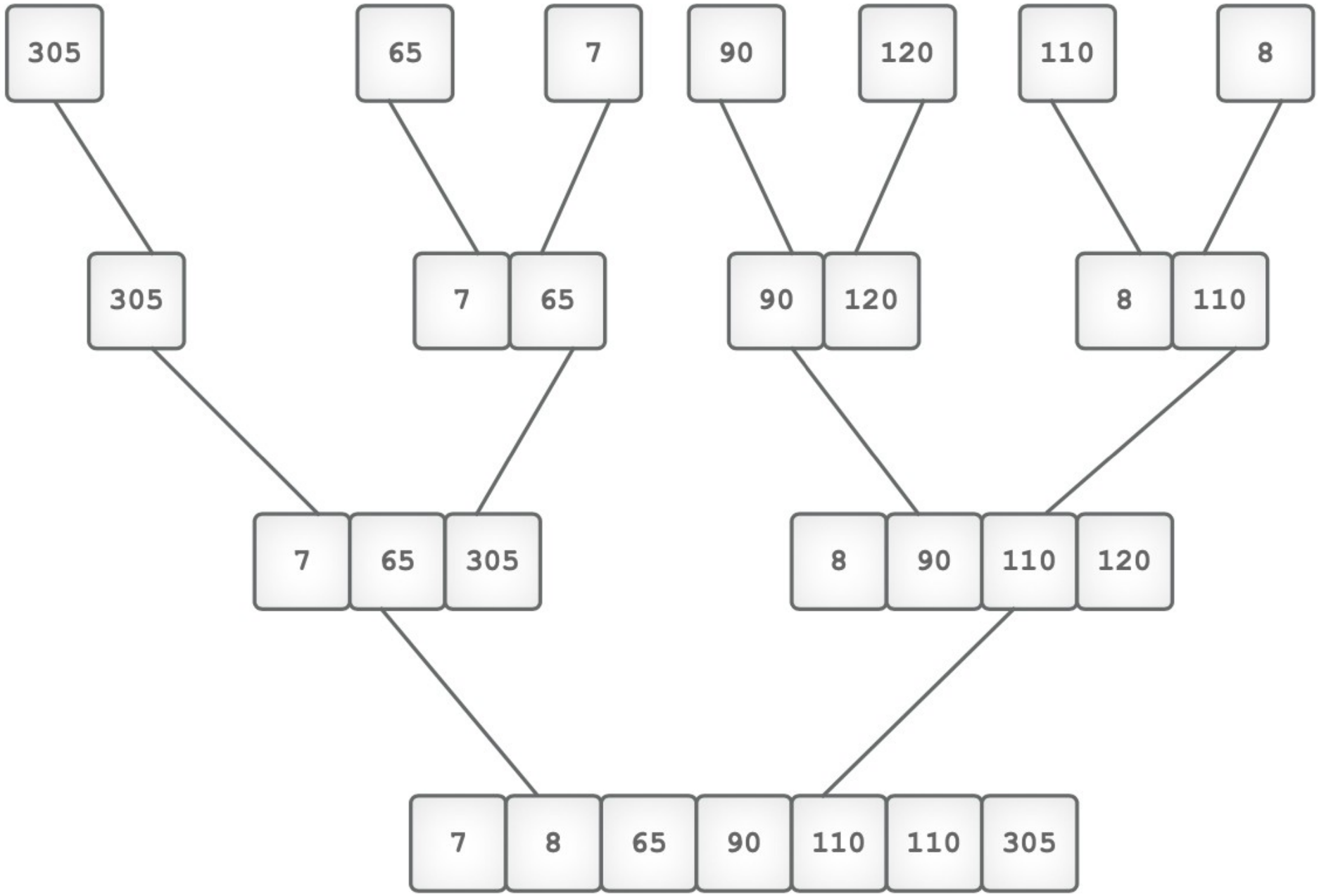
- Mais especificamente:
  - divide a lista em duas partes iguais
  - divide recursivamente cada parte ao meio continuamente até que uma parte contenha apenas um elemento
  - juntar as duas partes numa lista ordenada
  - continuar a juntar as partes à medida que a recursão se desdobra

- Mais especificamente:
  - divide a lista em duas partes iguais
  - divide recursivamente cada parte ao meio continuamente até que uma parte contenha apenas um elemento
  - juntar as duas partes numa lista ordenada
  - continuar a juntar as partes à medida que a recursão se desdobra

Decomposição do *Merge Sort*



Junção do *Merge Sort*



```
/**
  * Sorts the specified array of objects using the merge sort
  * algorithm.
  *
  * @param data  the array to be sorted
  * @param min   the integer representation of the minimum value
  * @param max   the integer representation of the maximum value
  */
public static <T extends Comparable<? super T>> void
        mergeSort (T[] data, int min, int max) {
   T[] temp;
   int index1, left, right;

   /** return on list of length one */
   if (min==max)
      return;

   /** find the length and the midpoint of the list */
   int size = max - min + 1;
   int pivot = (min + max) / 2;
   temp = (T[])(new Comparable[size]);
```

```
mergeSort(data, min, pivot); // sort left half of list
mergeSort(data, pivot + 1, max); // sort right half of list

/** copy sorted data into workspace */
for (index1 = 0; index1 < size; index1++)
   temp[index1] = data[min + index1];

/** merge the two sorted lists */
left = 0;
right = pivot - min + 1;
for (index1 = 0; index1 < size; index1++) {
   if (right <= max - min)
      if (left <= pivot - min)
         if (temp[left].compareTo(temp[right]) > 0)
            data[index1 + min] = temp[right++];

         else
            data[index1 + min] = temp[left++];
      else
         data[index1 + min] = temp[right++];
   else
      data[index1 + min] = temp[left++];
   }
}
```

Eficiência do *Quick Sort* e do *Merge Sort*

- Ambos os algoritmos `quickSort` e `mergeSort` usam uma estrutura recursiva que leva $\mathbf{log_2 n}$ passos para decompor a lista original nas suas sub-listas de um elemento

- Em cada passo, ambos os algoritmos ou comparam ou juntam todos os **n** elementos

- Então ambos os algoritmos são **O(nlogn)**