Apontamentos de Estruturas de Dados

Pesquisas e Ordenações

Ricardo Santos| rjs@estg.ipp.pt

Escola Superior de Tecnologia e Gestão
Instituto Politécnico do Porto

Última versão: Novembro de 2022

Índice

10. Pesquisas

10.1. Introdução

Um problema importante e recorrente na computação é o de localizar informação. Mais sucintamente, esse problema é conhecido como pesquisa. Este é um bom tópico para uma exploração preliminar das várias questões envolvidas no desenvolvimento de algoritmos.

Claramente, a informação a ser pesquisada deve primeiro ser representada de alguma forma. É aqui que entram as estruturas de dados. É claro que num computador tudo é representado em última análise como sequências de dígitos binários (*bits*), mas esse é um nível muito baixo para a maioria dos propósitos. Precisamos desenvolver e estudar estruturas de dados úteis que estejam mais próximas da maneira como os humanos pensam, ou pelo menos mais estruturadas do que meras sequências de *bits*. Isso ocorre porque são os humanos que precisam desenvolver e manter os sistemas de *software* – os computadores apenas os executam.

Depois de escolhermos uma representação adequada, a informação representada deve ser processada de alguma forma. Isso é o que leva à necessidade de algoritmos. Nesse caso, o processo de interesse é o da pesquisa. Para simplificar as coisas, vamos supor que queremos pesquisar uma coleção de números inteiros (embora também possamos lidar com *strings* de caracteres ou qualquer outro tipo de dados). Para começar, consideremos:

1. A representação mais óbvia e simples.
2. Dois algoritmos potenciais para processamento com essa representação.

Como já observamos, os *arrays* são uma das formas mais simples possíveis de representar coleções de números (ou *strings*, ou qualquer outra coisa), então usaremos isso para armazenar as informações a serem pesquisadas. Nos capítulos de seguida, veremos estruturas de dados mais complexas que podem tornar o armazenamento e a pesquisa mais eficientes.

Vamos supor, por exemplo, que o conjunto de inteiros que desejamos pesquisar é *{1, 4, 17, 3, 90, 79, 4, 6, 81}*. Podemos escrevê-los num *array* `A` como,

```
A = [1,4,17,3,90,79,4,6,81]
```

Se perguntarmos onde está o 17 nesse *array*, a resposta é 2, o índice desse elemento. Se perguntarmos onde está o 91, a resposta é *não está em lugar nenhum*. É útil poder representar **lugar nenhum** por um número que não é usado como índice possível. Como começamos o nosso índice contando a partir de 0, qualquer número negativo serviria. Seguiremos a convenção de usar o número -1 para representar **lugar nenhum**. Outras convenções (talvez melhores) são possíveis, mas não vamos considerar isso aqui.

10.2. Especificação de um problema de pesquisa

Podemos agora formular uma especificação do nosso problema de pesquisa através das seguintes definições:

Dado um array A e um inteiro x, encontrar um inteiro i tal que

1. *i é qualquer j para o qual A[j] é x.*
2. *caso contrário, se não existe j tal que A[j] é x, então i é −1,*

A primeira cláusula diz que se x ocorrer no *array* A então i é a posição onde ocorre, e a segunda diz que se não ocorrer então i deve ser -1. Se houver mais de uma posição onde *x* ocorre, então esta especificação permite retornar qualquer uma delas – por exemplo, este seria o caso se A fosse *[17, 13, 17]* e *x* fosse *17*. Assim, a especificação é ambígua. Assim, diferentes algoritmos com diferentes comportamentos podem satisfazer a mesma especificação – por exemplo, um algoritmo pode retornar a menor posição em que x ocorre e outro pode retornar a maior. Não há nada de errado com especificações ambíguas. Na verdade, na prática, elas ocorrem com bastante frequência.

10.3. Algoritmo simples: Pesquisa linear

Podemos expressar convenientemente o algoritmo mais simples possível numa forma de pseudocódigo que se parece com português, mas assemelha-se a um programa sem alguma precisão ou detalhe que um computador geralmente requer:

// Pressupõe que recebemos um *array* A de tamanho *n* e uma chave *x*.
Para i = 0,1,...,n-1,
se A[i] for igual a x,
então temos um i adequado e podemos terminar retornando i.
Se chegarmos a este ponto,
então x não está em A e, portanto, devemos terminar retornando -1.

Alguns aspetos, como as reticências ". . . ", são potencialmente ambíguos, mas nós, como seres humanos, sabemos exatamente o que significa, então não precisamos nos preocupar com eles. Numa linguagem de programação como C ou Java, alguém iria escrever algo mais preciso como:

```
1. for ( i = 0 ; i < n ; i++ ) {
2.         if ( a[i] == x ) {return i;}
3. }
4. return -1;
```

No caso de Java, o código estaria dentro de um método de uma classe sendo necessários mais detalhes como o parâmetro `a` para o método e uma declaração da variável auxiliar `i`. No caso da linguagem C o código estaria dentro de uma função e são necessários os mesmos detalhes. Em ambos, seria necessário haver código adicional para gerar o resultado num formato adequado.

Neste caso, é fácil analisar que o algoritmo satisfaz a especificação (assumindo que n é o tamanho correto do *array*) – basta observar que, por começarmos a contar do zero, a última posição do *array* é seu tamanho menos um. Se esquecermos isso e deixarmos que execute de 0 a n, obteremos um algoritmo incorreto. O efeito prático desse erro é que a execução desse algoritmo dá origem a um erro quando o item a ser localizado no *array* na verdade não está

lá já que tenta aceder a um local inexistente. Dependendo do idioma específico, sistema operativo e sistema computacional que está a ser usado, o efeito real desse erro será diferente.

De seguida é proposta uma implementação de um método de classe para a pesquisa linear que considera elementos do tipo genérico.

```
1.    /**
2.     * Searches the specified array of objects using a
3.     * linear search algorithm.
4.     *
5.     * @param data    the array to be sorted
6.     * @param min     the integer representation of the min value
7.     * @param max     the integer representation of the max value
8.     * @param target  the element being searched for
9.     * @return        true if the desired element is found
10.    */
11.   public static <T extends Comparable<? super T>>  boolean
12.               linearSearch (T[] data, int min, int max, T target)
13.   {
14.      int index = min;
15.      boolean found = false;
16.
17.      while (!found && index <= max)
18.      {
19.         if (data[index].compareTo(target) == 0)
20.            found = true;
21.         index++;
22.      }
23.      return found;
24.   }
```

10.4. Um algoritmo mais eficiente: Pesquisa Binária

É sempre necessário considerar se é possível melhorar o desempenho de um algoritmo específico, como o que acabamos de criar. Na pior das hipóteses, pesquisar um *array* de tamanho *n* leva *n* ciclos. Em média, levará *n/2* ciclos. Para grandes coleções de dados, como por exemplo todas as páginas da Internet na prática seria inaceitável. Assim, devemos tentar organizar a coleção de tal forma que um algoritmo mais eficiente seja possível. Como veremos mais adiante, há muitas possibilidades, e quanto mais exigimos em termos de eficiência, mais complicadas tendem a se tornar as estruturas de dados que representam as coleções. Aqui vamos considerar uma das mais simples – ainda representamos as coleções por *arrays*, mas agora enumeramos os elementos em ordem crescente.

Assim, ao invés de trabalharmos com o *array* anterior *[1, 4, 17, 3, 90, 79, 4, 6, 81]*, iremos usar o *[1, 3, 4, 4, 6, 17, 79, 81, 90]*, que possui os mesmos itens, mas listados em ordem crescente. Então podemos usar um algoritmo aprimorado, que em forma de pseudocódigo seria:

// Assume que recebemos um array ordenado A de tamanho n e uma chave x.
// Usar inteiros left e right (inicialmente definidos como 0 e n-1) e mid.
Enquanto a esquerda é menor que a direita,
defina mid como a parte inteira de (esquerda+direita)/2, e
se x for maior que A[mid],
então definir left para mid + 1,
caso contrário, definir right para mid.
Se A[left] é igual a x,
então terminar retornando left,
caso contrário terminar retornando -1.

e corresponderia a um bloco de código C ou Java como:

```
1.    int a = [1,3,4,4,6,17,79,81,90];
2.    int n = 9;
3.    int x = 79;
4.    /* PROGRAM */
5.    int left = 0, right = n-1, mid;
6.    while ( left < right ) {
```

```
7.          mid = ( left + right ) / 2;
8.          if ( x > a[mid] ) {left = mid+1;}
9.          else {right = mid;}
10.     }
11.     if ( a[left] == x ) {return left;}
12.     else {return -1;}
```

Este algoritmo funciona dividindo repetidamente o *array* em dois segmentos, um da esquerda para o meio e o outro do meio *+1* para a direita, onde `mid` é a posição a meio caminho da esquerda para a direita e onde, inicialmente, esquerda e direita são as posições mais à esquerda e mais à direita do *array*. Como o *array* está ordenado é fácil ver em qual de cada um dos pares de segmentos está o item x e a pesquisa pode ser restrita a esse segmento. Além disso, como o tamanho do *sub-array* da esquerda para a direita é reduzido pela metade a cada iteração do ciclo `while`, precisamos apenas de *log2 n* passos na média ou no pior caso. Para vermos que esse resultado do tempo de execução é uma grande melhoria, na prática, em relação ao algoritmo de pesquisa linear anterior, observe que *log2 1000000* é aproximadamente *20*, de modo que para um *array* de tamanho *1000000* apenas *20* iterações são necessárias no pior caso do algoritmo de pesquisa binária, enquanto *1000000* iterações seriam necessárias no pior caso do algoritmo de pesquisa linear.

Com o algoritmo de pesquisa binária não é tão óbvio que tomamos os devidos cuidados com os limites da condição do ciclo `while`. Além disso, estritamente falando, este algoritmo não está correto porque não funciona para o *array* vazio (que tem tamanho zero), mas isso pode ser facilmente corrigido.

Vale a pena considerar se as versões de lista ligada dos dois algoritmos apresentados funcionariam ou ofereceriam alguma vantagem. É bastante claro que poderíamos realizar uma pesquisa linear por meio de uma lista ligada essencialmente da mesma maneira que com um *array*, com o ponteiro relevante retornado em vez de um índice. Converter a pesquisa binária em forma de lista ligada é problemático, porque não há uma maneira eficiente de dividir uma lista ligada em dois segmentos. Parece que a nossa abordagem baseada em *array* é a melhor que podemos fazer com as estruturas de dados que estudamos até agora. No

entanto, veremos em capítulos seguintes como estruturas de dados mais complexas (árvores) podem ser usadas para formular algoritmos de pesquisa recursiva eficientes.

Observe que ainda não analisamos o esforço necessário para ordenar o *array* para que o algoritmo de pesquisa binária possa ser usado. Até sabermos isso, não podemos ter a certeza de que usar o algoritmo de pesquisa binária é realmente mais eficiente do que usar o algoritmo de pesquisa linear no *array* original não ordenado. Isso também pode depender de mais detalhes, como quantas vezes precisamos realizar uma pesquisa no conjunto de n itens – apenas uma vez, ou até n vezes. Voltaremos a essas questões mais adiante. Primeiro, precisamos considerar com mais detalhes como comparar a eficiência do algoritmo de maneira confiável.

De seguida é proposta uma implementação de um método de classe para a pesquisa binária que considera elementos do tipo genérico.

```
1.    /**
2.     * Searches the specified array of objects using a
3.     * binary search algorithm.
4.     * @param data    the array to be sorted
5.     * @param min     the integer representation of the minimum value
6.     * @param max     the integer representation of the maximum value
7.     * @param target  the element being searched for
8.     * @return        true if the desired element is found
9.     */
10.   public static <T extends Comparable<? super T>>  boolean
11.          binarySearch (T[] data, int min, int max, T target){
12.
13.      boolean found = false;
14.      int midpoint = (min + max) / 2;  // determine the midpoint
15.
16.      if (data[midpoint].compareTo(target) == 0)
17.         found = true;
18.      else if (data[midpoint].compareTo(target) > 0) {
19.         if (min <= midpoint - 1)
20.            found = binarySearch(data, min, midpoint - 1, target);
```

```
21.       }
22.       else if (midpoint + 1 <= max)
23.          found = binarySearch(data, midpoint + 1, max, target);
24.
25.       return found;
26.    }
```

10.5. Exercícios Propostos

Exercício 1

Qual é o algoritmo de pesquisa mais eficiente, a pesquisa binária ou a pesquisa linear?

Exercício 2

Criar uma classe que represente uma entidade, por exemplo um carro, assim como algumas que ache relevantes para o representar. Armazenar instâncias dessa entidade num *array* e usar os seguintes métodos de pesquisa: pesquisa linear e pesquisa binária para encontrar um determinado elemento.

11. Algoritmos de Ordenação

11.1. Introdução

A ordenação é uma das tarefas de processamento de dados mais importantes e fundamentais.

> **Algoritmo de ordenação:** algoritmo que reorganiza registos em listas de forma a que sigam alguma relação de ordenação bem definida nos valores das chaves em cada registo.

Um algoritmo de ordenação interno funciona em listas na memória principal, enquanto um algoritmo de ordenação externo funciona em listas armazenadas em ficheiros. Alguns algoritmos de ordenação funcionam muito melhor como ordenações internas do que como ordenações externas, mas alguns funcionam bem em ambos os casos. Um algoritmo de ordenação é estável se preserva a ordem original dos registos com chaves iguais.

Foram inventados muitos algoritmos de ordenação, irão ser explicados os algoritmos de ordenação mais simples assim como alguns mais avançados.

11.2. Bubble Sort

Um dos algoritmos de ordenação mais antigos é o *Bubble Sort*. Este algoritmo simula o processo de bolhas de gás num líquido em que estas trocam de posição com o líquido até que seja atingido um equilíbrio. A ideia por trás desta ordenação é fazer passagens repetidas pela lista do início ao fim, comparando elementos adjacentes e trocando qualquer um que esteja fora da ordem. Após a primeira passagem, o maior elemento terá sido movido para o final da lista; após a segunda passagem, o segundo maior terá sido movido para a penúltima posição; e assim por diante. A ideia é que os valores grandes "subam" para o topo da lista em cada passagem.

Uma possível implementação através da abordagem que temos seguido seria a seguinte:

```
1.   /**
2.     * Sorts the specified array of objects using a bubble sort
3.     * algorithm.
4.     *
5.     * @param data  the array to be sorted
6.     */
7.    public static <T extends Comparable<? super T>> void
8.              bubbleSort (T[] data) {
9.       int position, scan;
10.      T temp;
11.      for (position =  data.length - 1; position >= 0; position--) {
12.         for (scan = 0; scan <= position - 1; scan++) {
13.            if (data[scan].compareTo(data[scan+1]) > 0) {
14.               /** Swap the values */
15.               temp = data[scan];
16.               data[scan] = data[scan + 1];
17.               data[scan + 1] = temp;
18.            }
19.         }
20.      }
21.   }
22.
```

Deve ficar claro que o algoritmo faz exatamente as mesmas comparações chave, independentemente do conteúdo do *array*, portanto, precisamos considerar apenas a complexidade básica do algoritmo. Na primeira passagem pelos dados, todos os elementos do *array*, exceto o primeiro, são comparados com seu predecessor, portanto são feitas *n-1* comparações. Na passagem seguinte é feita uma comparação a menos sendo *n-2*. Essa lógica continua até à última passagem, onde é feita apenas uma comparação.

Como se consegue compreender o *Bubble Sort* não é um algoritmo muito rápido. Existem várias versões diferentes do *Bubble Sort* para o melhorar. Por exemplo, uma variável booleana pode ser definida como `false` no início de cada passagem pela lista e colocada a

`true` sempre que é feita uma troca. Se a variável for falsa quando a passagem for concluída significa que não foi feita nenhuma troca o que implicada que o *array* está ordenado e por esse motivo o algoritmo pode parar. Independentemente dessas melhorias no pior caso possível a complexidade é a mesma sendo apenas vantajoso na complexidade de melhor caso que é de apenas *n*. A complexidade média ainda assim ainda está em $O(n^2)$ que não representa grande melhoria.

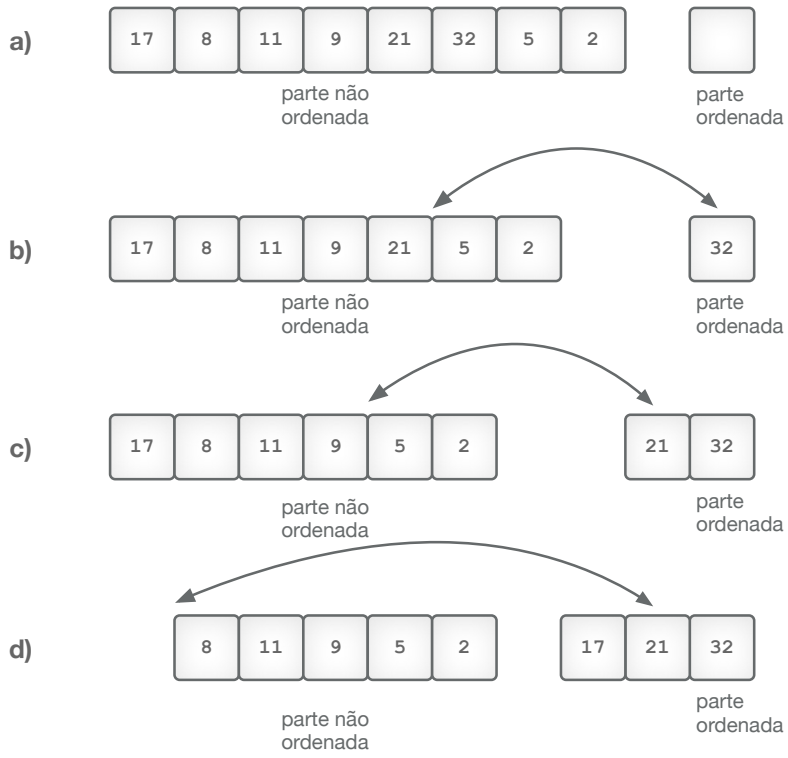
11.3. Selection Sort

A ideia por trás do *Selection Sort* é fazer passagens repetidas pela lista, em cada passagem encontra o maior (ou menor) valor na parte não classificada da lista e coloca-o no final (ou início) da parte não classificada, diminuindo assim a porção não classificada e aumentando a porção classificada. Assim, o algoritmo funciona "selecionando" repetidamente o item que vai no final da parte não ordenada da lista. De forma muito resumida o *Selection Sort* ordena uma lista de valores ao colocar repetidamente os valores nas suas posições finais. De seguida são apresentados os passos de alto nível para uma abordagem em que é usado o menos valor da lista em cada passagem:

- 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

Como já foi referido anteriormente podemos selecionar o maior valor em vez do menor e colocar no final da lista em vez do início. Esta é uma decisão que acabar por ser pessoal.

Para compreendermos melhor o algoritmo de ordenação de seguida é apresentado um exemplo passo a passo para a ordenação do *array [17, 8, 11, 9, 21, 32, 5, 2]*.



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 início da parte ordenada (1º passo)

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

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

Uma possível implementação através da abordagem que temos seguido seria a seguinte:

```
1.    /**
2.     * Sorts the specified array of integers using the selection
3.     * sort algorithm.
4.     *
5.     * @param data  the array to be sorted
6.     */
7.    public static <T extends Comparable<? super T>> void
```

```
8.              selectionSort (T[] data) {
9.       int min;
10.      T temp;
11.      for (int index = 0; index < data.length-1; index++){
12.         min = index;
13.         for (int scan = index+1; scan < data.length; scan++)
14.            if (data[scan].compareTo(data[min])<0) {
15.               min = scan;
16.            }
17.         /** Swap the values */
18.         temp = data[min];
19.         data[min] = data[index];
20.         data[index] = temp;
21.      }
22.   }
23.
```

Esse algoritmo encontra o valor mínimo na parte não classificada da lista n-1 vezes e coloca-o onde pertence. Tal como o *Bubble Sort*, faz exatamente a mesma coisa, não importa qual seja o conteúdo do *array*, então precisamos considerar apenas a sua complexidade básica.

Embora o número de comparações que o *Selection Sort* faz seja idêntico ao *Bubble Sort*, a ordenação através do *Selection Sort* geralmente é consideravelmente mais rápida. Isso acontece porque o *Bubble Sort* normalmente faz muitas trocas em cada passagem pela lista, enquanto o *Selection Sort* faz apenas uma. No entanto, nenhum desses algoritmos é considerado rápido.

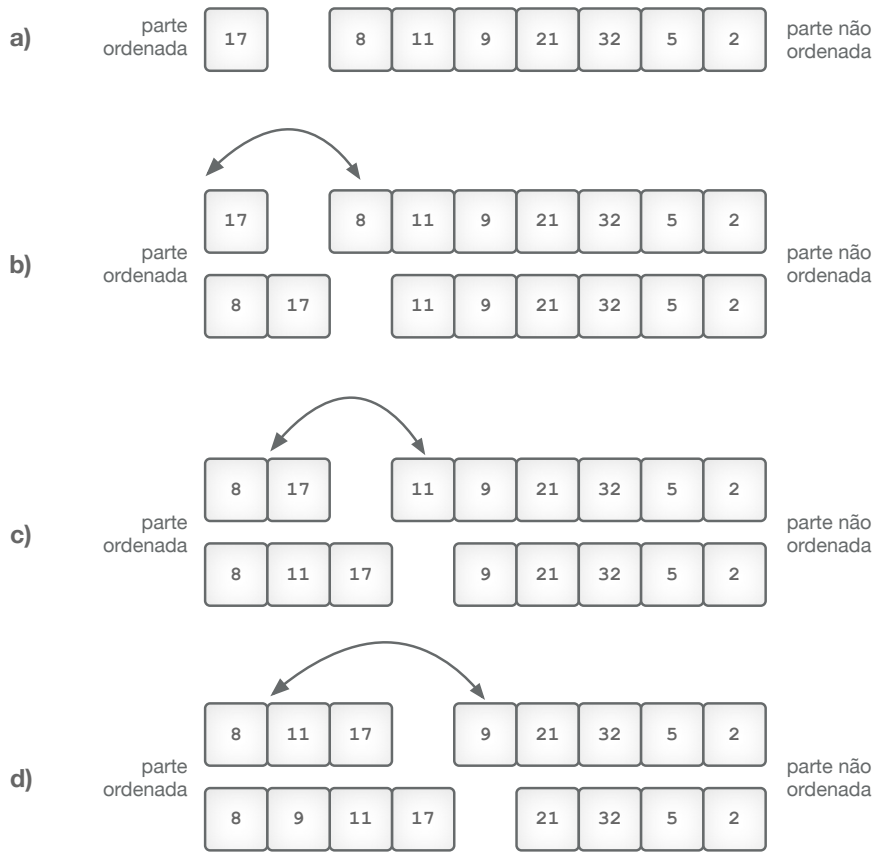
11.4. Insertion Sort

O *Insertion Sort* funciona ao usar repetidamente um elemento da parte não classificada de uma lista e inserindo-o na parte classificada da lista até que todos os elementos tenham sido inseridos. Este algoritmo é aquele que é normalmente usado no dia a dia por pessoas quando pretendem ordenar um conjunto de documentos. Uma lista com apenas um elemento está ordenada, então os elementos inseridos começam com o segundo elemento do *array*. O

elemento inserido é mantido na variável elemento e os valores na parte ordenada do *array* são movidos para cima para dar espaço para o elemento inserido no mesmo ciclo onde a pesquisa é feita para encontrar o lugar certo para fazer a inserção. Uma vez encontrado, o ciclo termina e o elemento inserido é colocado na parte ordenada do *array*. De seguida são apresentados os passos de alto nível para a implementação do algoritmo:

- 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 corretas.

Para compreendermos melhor o algoritmo de ordenação de seguida é apresentado um exemplo passo a passo para a ordenação do *array [17, 8, 11, 9, 21, 32, 5, 2]*.



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 correta na parte ordenada (1º passo)
c) O *array* depois do primeiro valor da parte não ordenada ter sido inserido na sua posição correta na parte ordenada (2º passo)
d) O *array* depois do primeiro valor da parte não ordenada ter sido inserido na sua posição correta na parte ordenada (3º passo)

Uma possível implementação através da abordagem que temos seguido seria a seguinte:

```
1.    /**
2.     * Sorts the specified array of objects using an insertion
3.     * sort algorithm.
4.     *
5.     * @param data  the array to be sorted
6.     */
7.    public static <T extends Comparable<? super T>> void
8.              insertionSort (T[] data) {
9.       for (int index = 1; index < data.length; index++) {
10.         T key = data[index];
11.         int position = index;
12.
13.         /** Shift larger values to the right */
14.         while (position > 0 && data[position-1].compareTo(key) > 0){
15.            data[position] = data[position-1];
16.            position--;
17.         }
18.
19.         data[position] = key;
20.      }
21.   }
22.
```

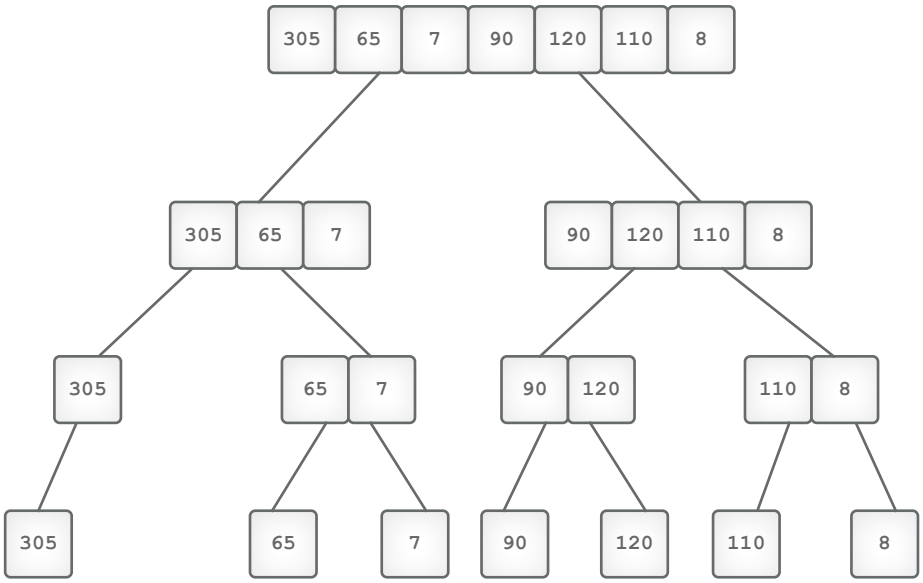
A ordenação *Insertion Sort* faz coisas diferentes dependendo do conteúdo da lista, portanto, devemos considerar o seu comportamento para os casos: pior, melhor e médio. Se a lista já estiver ordenada é feita uma comparação para cada um dos *n-1* elementos conforme eles são "inseridos" nas suas localizações atuais. O pior caso ocorre quando cada elemento inserido deve ser colocado no início da parte já ordenada da lista; isso acontece quando a lista está na ordem inversa. Nesse caso, o primeiro elemento inserido requer uma comparação, o segundo duas, o terceiro três e assim por diante, então devem ser inseridos *n-1* elementos. No caso médio, a ordenação *Insertion Sort* faz cerca de metade do número de comparações que faz no pior caso. Infelizmente, ambas as funções estão em $O(n^2)$, então a ordenação *Insertion Sort* não é uma ótima ordenação. No entanto, a ordenação *Insertion Sort* em média é um pouco melhor do que a ordenação *Bubble Sort* e *Selection Sort* e, no melhor dos casos, é o melhor dos três algoritmos de ordenação $O(n^2)$.

11.5. Merge Sort

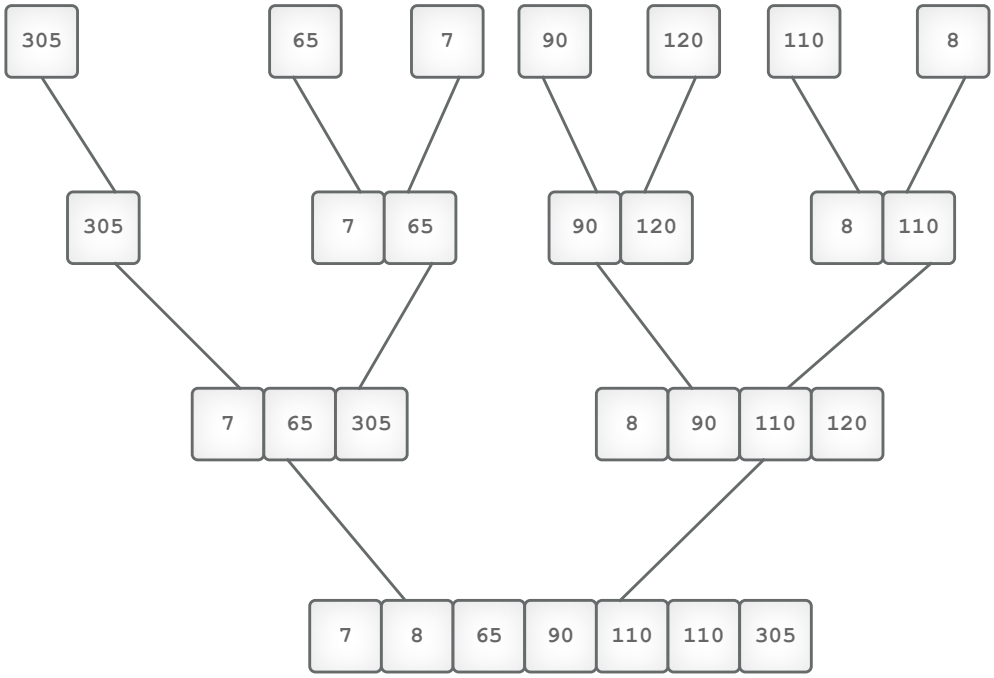
O *Merge Sort* é um exemplo clássico de algoritmo de Divisão e Conquista que resolve um grande problema dividindo-o em partes, resolvendo os problemas menores resultantes e, de seguida, combinando essas soluções numa solução para o problema original. A estratégia do *Merge Sort* é ordenar metades de uma lista (recursivamente) e depois juntar os resultados na lista classificada final. A junção é uma operação bastante rápida e dividir um problema ao meio repetidamente permite chegar rapidamente a listas que já estão ordenados (listas de comprimento um ou menos). Por esse motivo este é um algoritmo que apresenta uma boa performance. De seguida são apresentados os passos de alto nível para a implementação do algoritmo:

- 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.

Para compreendermos melhor o algoritmo de ordenação de seguida é apresentado um diagrama que apresenta os passos primeiro da divisão:



De seguida da junção das várias listas ordenadas que permite obter uma lista ordenada final.



Uma possível implementação através da abordagem que temos seguido seria a seguinte:

```
1.   /**
2.     * Sorts the specified array of objects using the merge sort
3.     * algorithm.
4.     *
```

```
5.      * @param data  the array to be sorted
6.      * @param min   the integer representation of the minimum value
7.      * @param max   the integer representation of the maximum value
8.      */
9.     public static <T extends Comparable<? super T>> void
10.             mergeSort (T[] data, int min, int max) {
11.       T[] temp;
12.       int index1, left, right;
13.
14.       /** return on list of length one */
15.       if (min==max)
16.          return;
17.
18.       /** find the length and the midpoint of the list */
19.       int size = max - min + 1;
20.       int pivot = (min + max) / 2;
21.       temp = (T[])(new Comparable[size]);
22.       mergeSort(data, min, pivot); // sort left half of list
23.       mergeSort(data, pivot + 1, max); // sort right half of list
24.
25.       /** copy sorted data into workspace */
26.       for (index1 = 0; index1 < size; index1++) {
27.          temp[index1] = data[min + index1];
28.       }
29.       /** merge the two sorted lists */
30.       left = 0;
31.       right = pivot - min + 1;
32.       for (index1 = 0; index1 < size; index1++) {
33.          if (right <= max - min) {
34.             if (left <= pivot - min)
35.                if (temp[left].compareTo(temp[right]) > 0)
36.                   data[index1 + min] = temp[right++];
37.                else
38.                   data[index1 + min] = temp[left++];
39.             else
40.                data[index1 + min] = temp[right++];
41.          else
42.             data[index1 + min] = temp[left++];
```

```
43.       }
44.    }
```

A junção requer um local para armazenar o resultado dessa junção de duas listas e a duplicação da lista original fornece espaço para a junção. Esta operação ordena recursivamente as duas metades da lista auxiliar e junta-as de volta na lista original. Embora seja possível ordenar e juntar no local, ou juntar usando uma lista com apenas metade do tamanho original, os algoritmos para fazer a ordenação por junção dessa forma são complicados e têm muita sobrecarga - é mais simples e rápido usar uma lista auxiliar do tamanho do original, embora exija muito espaço extra.

Ao analisar este algoritmo, a medida do tamanho da entrada é, obviamente, o comprimento da lista ordenada, e as operações contadas são comparações chave. A comparação de chaves ocorre na etapa da junção: os menores itens nas sublistas juntas são comparados e o menor é movido para a lista de destino. Essa etapa é repetida até que uma das sublistas se esgote, caso em que o restante da outra sublista é copiado para a lista de destino. A junção nem sempre exige o mesmo esforço: depende do conteúdo das sublistas. Na melhor das hipóteses, o maior elemento de uma sublista é sempre menor que o menor elemento da outra (o que ocorre, por exemplo, quando a lista de entrada já está ordenada).

Na melhor das hipóteses, o *Merge Sort* faz apenas cerca de *(n log n)/2* comparações chave, o que é bastante rápido. Estamos a falar obviamente de uma eficiência *O(n log n)*. Na pior das hipóteses, fazer mais comparações ocorre ao juntar duas sublistas, de modo que uma se esgote quando houver apenas um elemento na outra. Nesse caso, cada operação de junção para uma lista de destino de tamanho *n* requer *n-1* comparações. O comportamento de pior caso do *Merge Sort* é portanto, também de *O(n log n)*. Como caso médio, vamos supor que cada comparação chave das duas sublistas tem a mesma probabilidade de resultar num elemento de uma sublista sendo movido para a lista de destino como da outra. Isso é como atirar uma moeda: é tão provável que o elemento movido de uma sublista ganhe a comparação como um elemento da outra. E como ao atirar uma moeda esperamos que a longo prazo os elementos escolhidos de uma lista sejam aproximadamente os mesmos que

os elementos escolhidos da outra, de modo que as sublistas se esgotem mais ou menos ao mesmo tempo. Essa situação é aproximadamente a mesma que o comportamento do pior caso, portanto, em média o *Merge Sort* fará aproximadamente o mesmo número de comparações que no pior caso.

Assim, em todos os casos, o *Merge Sort* tem uma eficiência de *O(n log n)*, o que significa que é significativamente mais rápida do que as outras ordenações que vimos até agora. A principal desvantagem é que usa *O(n)* locais de memória extra para fazer o seu trabalho.

11.6. Quicksort

O mais estudado, amplamente utilizado e mais rápido de todos os algoritmos de ordenação por comparação de chaves é o *Quicksort*, inventado por *Tony Hoare* em 1959.

O *Quicksort* também é um algoritmo de Divisão e Conquista. Funciona através da seleção de um único elemento da lista denominado de elemento pivô e reorganiza a lista para que todos os elementos menores ou iguais ao pivô estejam à sua esquerda, e todos os elementos maiores ou iguais à sua direita. Essa operação é chamada de particionamento. Depois da lista ser particionada, o algoritmo chama-se recursivamente de modo a ordenar as sublistas à esquerda e à direita do pivô. Eventualmente, as sublistas têm comprimento de um ficando ordenadas e terminando a recursão.

Como já foi referido o principal componente do *Quicksort* é o algoritmo de particionamento. Esse algoritmo deve escolher um elemento pivô e reorganizar a lista o mais rápido possível para que o elemento pivô esteja na sua posição final, todos os valores maiores que o pivô têm de estar à direita e todos os valores menores à sua esquerda. Embora existam muitas variações desse algoritmo, a abordagem geral é escolher um elemento arbitrário como pivô, percorrer a partir da esquerda até encontrar um valor maior que o pivô e da direita até encontrar um valor menor que o pivô. Esses valores são então trocados e ciclos são retomados. O elemento pivô pertence à posição em que as pesquisa se encontram. Embora pareça muito simples, o algoritmo de particionamento do *Quicksort* é bastante elegante e

difícil de implementá-lo corretamente. Por esse motivo, geralmente é uma boa ideia copiá-lo de uma fonte onde tenha sido testado extensivamente.

Como já foi referido O *Quicksort* ordena uma lista de valores através da repartição da lista em torno de um elemento. De seguida são apresentados os passos de alto nível para a implementação do algoritmo:

- 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 partição é arbitrário, no entanto por eficiência seria bom que o elemento partição dividisse a lista ao meio. Vamos dividir a solução em dois métodos:

- `quickSort` - executa o algoritmo recursivo
- `findPartition` - reorganiza os elementos em duas partições

```
1. /**
2.     * Sorts the specified array of objects using the quick sort
3.     * algorithm.
4.     *
5.     * @param data  the array to be sorted
6.     * @param min   the integer representation of the minimum value
7.     * @param max   the integer representation of the maximum value
8.     */
9.    public static <T extends Comparable<? super T>> void
10.           quickSort (T[] data, int min, int max) {
11.      int indexofpartition;
12.
13.      if (max - min  > 0) {
14.         /** Create partitions */
15.         indexofpartition = findPartition(data, min, max);
16.
17.         /** Sort the left side */
```

```
18.         quickSort(data, min, indexofpartition - 1);
19.
20.         /** Sort the right side */
21.         quickSort(data, indexofpartition + 1, max);
22.       }
23.    }
```

```
1. /**
2.     * Used by the quick sort algorithm to find the partition.
3.     *
4.     * @param data  the array to be sorted
5.     * @param min   the integer representation of the minimum value
6.     * @param max   the integer representation of the maximum value
7.     */
8.    private static <T extends Comparable<? super T>> int
9.                findPartition (T[] data, int min, int max) {
10.      int left, right;
11.      T temp, partitionelement;
12.      int middle = (min + max)/2;
13.
14.      // use middle element as partition
15.      partitionelement = data[middle];
16.      left = min;
17.      right = max;
18.
19.      while (left<right) {
20.         /** search for an element that is > the partitionelement */
21.         while (data[left].compareTo(partitionelement) <0 {
22.            left++;
23.}
24.         /** search for an element that is < the partitionelement */
25.         while (data[right].compareTo(partitionelement) > 0)
26.            right--;
27.
```

```
28.          /** swap the elements  */
29.          if (left<right) {
30.             temp = data[left];
31.             data[left] = data[right];
32.             data[right] = temp;
33.          }
34.       }
35.
36.       /** move partition element to partition index*/
37.       temp = data[min];
38.       data[min] = data[right];
39.       data[right] = temp;
40.
41.       return right;
42.    }
```

O melhor caso do *Quicksort* ocorre quando as partições são balanceadas, isto é, cada partição com $n/2$ elementos. A equação de recorrência nesse caso é igual à do *Merge Sort*

$$T(n)=2T(n/2)+\Theta(n)$$

Já o pior caso do *Quicksort* ocorre quando as chamadas recursivas produzem partições com 0 e n−1 elementos. A partição de tamanho zero fica à direita ou à esquerda do pivô (depende do *array*).

$$T(n)=T(n\text{-}1)+\Theta(n)$$

A complexidade no caso médio do *Quicksort* é $O(n \log n)$ e é um dos pontos fortes do algoritmo. O caso médio é uma medida estatística. Isso significa que, ao executar o *Quicksort*, espera-se que o tempo de execução seja $O(n \log n)$.

É claro, o *Merge Sort* também é $O(n \log n)$. Entretanto, na prática, o *Quicksort* normalmente é mais rápido que o *Merge Sort*, pois a notação assintótica oculta as constantes multiplicativas e os termos de menor ordem de grandeza que são determinantes ao comparar algoritmos com a mesma complexidade.

O *Merge Sort* é um algoritmo de ordenação rápido cuja complexidade de caso melhor, pior e médio estão todas em *O(n log n)*, mas infelizmente usa espaço extra *O(n)* para fazer o seu trabalho. O *Quicksort* tem complexidade de caso melhor e média em *O(n log n)*, mas infelizmente a sua complexidade de pior caso está em $O(n^2)$. A melhoria da mediana de três torna o comportamento do pior caso do *Quicksort* extremamente improvável e a velocidade incomparável do *Quicksort* na prática torna-o o algoritmo de ordenação preferido ao ordenar por comparação de chaves.

11.7. Exercícios Propostos

Exercício 1

A complexidade do *Bubble Sort* e *Selection sort* é exatamente a mesma. Isso significa que não há razão para preferir um ao outro?

Exercício 2

Reescreva o algoritmo *Bubble Sort* para incorporar uma verificação para ver se o *array* está ordenado após cada passagem e interromper o processamento quando isso ocorrer.

Exercício 3

Ajuste o algoritmo de ordenação *Selection Sort* apresentado acima para ordenar usando o elemento máximo em vez do mínimo na parte não ordenada do *array*.

Exercício 4

Porquê que o *Merge Sort* necessita espaço extra?

Exercício 5

O que interrompe a recursão no *Merge Sort* e no *Quicksort*?

Exercício 6

O que é o valor pivot no *Quicksort*?

Exercício 7

Escreva uma versão não recursiva do *Merge Sort* que use uma stack para acompanhar as sublistas que ainda precisam ser ordenadas. Cronometre sua implementação em relação ao algoritmo *Merge Sort* não modificado e resuma os resultados.