03

Implementando a ordenação por inserção

Implementando a ordenação por inserção

Nós já temos um algoritmo de ordenação (o *Selection Sort*), que foi implementado com o método chamado **ordena**.

```
ordena(produtos, produtos.length);
```

Vou renomear o método `ordena` para `selectionSort`, pois teremos outras diversas ordenações:

```
selectionSort(produtos, produtos.length);
```

Queremos fazer uma nova ordenação que chamaremos de `novoSort`. Ela também receberá o nosso *array* e o tamanho da nossa lista.

```
selectionSort(produtos, produtos.length);
novoSort(produtos, produtos.length);
```

Vamos comentar a linha de cima:

```
\\selectionSort(produtos, produtos.length);
```

Depois iremos ordenar o que precisa ser ordenado... O método será igual e terá a mesma assinatura do *selectionSort*:

```
private static void novoSort(Produto[] produtos, int quantidadeDeElementos) {
}
```

Nós queremos implementar um novo laço que passe por todos elementos. Então, o nosso `for` será:

```
for(int atual = 0; atual < quantidadeDeElementos; atual++);
```

Assim passaremos por todos os itens do *array*.

Por exemplo, selecionei o elemento da **posição 2** e a partir dele quero analisar os elementos anteriores. Iremos compará-los para encontrar a posição que o elemento atual deve ser inserido. Isto significa que sempre iremos comparar o elemento analisado com os das posições anteriores, ou seja, a partir daquela posição:

```
int analise = atual;
```

E o que nós levaremos em conta na análise? Além do produto (`produtos[analise]`), iremos considerar seu preço (`getPreco()`). Caso o preço seja menor do que o elemento posicionado anteriormente (`produtos[analise -1]`), o produto

não está no lugar correto e precisa ser reposicionado. Enquanto (*while*) os elementos não estiverem ordenados, precisaremos seguir trocando os itens de posição.

```
while(produtos[analise].getPreco() < produtos[analise -1].getPreco());
```

Precisamos nos certificar de que os elementos serão trocados de posição...

O `produtoAnalise` será o `produtos[analise]` e o `produtoAnaliseMenos1` será `produtos[analise -1]`.

```
Produto produtoAnalise = produtos[analise];
Produto produtoAnaliseMenos1 = produtos[analise -1];
```

Como será feita a troca dos elementos? Iremos trocar `produtos[analise]` e `produtos[analise -1]` de posições.

```
produtos[analise] = produtoAnaliseMenos1;
produtos[analise -1] = produtoAnalise;
```

Começamos a analisar cada casinha e fizemos as seguintes considerações: "esse elemento é mais caro ou mais barato que o elemento à esquerda? Se ele for mais caro, não precisaremos continuar com nossa análise, porque ele já está posicionado corretamente. Porém, se o preço dele for menor, precisaremos trocá-lo de lugar".

Após realizarmos a primeira troca, precisaremos continuar com as verificações para identificar se o valor também é menor do que os anteriores. Seguiremos pelos itens da esquerda (`analise --`) até encontrar a posição devida do elemento, na ordenação.

O nosso código ficará assim:

```
private static void novoSort(Produto[] produtos, int quantidadeDeElementos) {
    for(int atual = 0; atual < quantidadeDeElementos; atual++) {
        int analise = atual;
        while(produtos[analise].getPreco() < produtos[analise - 1].getPreco()) {
            Produto produtoAnalise = produtos[analise];
            Produto produtoAnaliseMenos1 = produtos[analise -1];
            produtos[analise] = produtoAnaliseMenos1;
            produtos[analise -1] = produtoAnalise;
            analise--;
        }
    }
}
```

Ao analisar a posição atual, verificamos se o preço do elemento é menor e se a posição é a devida. Fizemos as trocas quando necessário e seguimos a análise para o próximo (anterior) item. Repetimos o processo até encontrarmos a posição que ele deveria ser inserido (o que acontece quando o preço do produto é mais caro do que o valor do elemento situado antes na lista). Quando o valor é maior, encontramos a posição justa e paramos o processo.

Vamos testar o nosso código e ver o que acontece? Quando rodamos o programa, a saída será um *Exception -1*.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
        at br.com.alura.algoritmos.TestaOrdenacao.novoSort(TestaOrdenacao.java:27)
        at br.com.alura.algoritmos.TestaOrdenacao.mais(TestaOrdenacao.java:15)
```

Ao acessarmos o `TestaOrdenacao.java:15` linha 27, descobriremos que o nosso problema está na ordenação das posições `analise` e `analise -1` .

Quando `analise` for igual a 0, `analise -1` será igual a -1. Logo, não poderemos acessar o elemento. Isso significa que existe um limite de análise. E para que o programa não tente acessar a posição -1, precisamos criar uma condição e definir que `analise` seja maior que 0 (`analise > 0`).

```
private static void novoSort(Produto[] produtos, int quantidadeDeElementos) {
    for(int atual = 0; atual < quantidadeDeElementos; atual++) {
        int analise = atual;
        while(analise > 0 && produtos[analise].getPreco() < produtos[analise - 1].getPreco()) {
            Produto produto Analise = produtos[analise];
            Produto produtoAnaliseMenos1 = produtos[analise -1];
            produtos[analise] = produtoAnaliseMenos1;
            produtos[analise -1] = produtoAnalise;
            analise--;
        }
    }
}
```

Porque se o `analise` for igual 1, o `analise -1` será igual a 0. Enquanto não ultrapassarmos o limite da esquerda, iremos continuar reposicionando os elementos.

Agora se testarmos o algoritmo, o programa irá rodar corretamente e imprimirá os resultados.

```
Brasília custa 16000
Fusca custa 1700
Jipe custa 46000
Smart custa 46000
Lamborghini custa 1000000
```

Revisando: nós começamos da esquerda para direita, passando por cada uma das casinhas. Recebemos cada carta (ou cada produto) e comparamos os elementos com os que tínhamos recebido antes. A partir disto, encontramos a posição que deveríamos inseri-los.

No exemplo do baralho, quando já tínhamos três cartas e a quarta foi recebida, analisamos as anteriores para definir a posição que deveríamos colocá-la. Nosso laço foi construído sempre comparando os elementos na posição atual com os itens anteriores. Quando o item recebido não atendia a condição de ser menor, ou chegávamos ao limite definido na esquerda, nossa análise chegava ao fim. Para cada um dos nossos elementos, respeitando o `for` do `atual` , encontramos a posição correta na ordem. Para cada carta de baralho que recebemos, encontramos a posição adequada na lista.

Em seguida iremos simular este processo com calma.

Logando as informações da ordenação por inserção

Nós já implementamos o código da nova ordenação. Vamos ver se ele funciona corretamente?

```
private static void novoSort(Produto[] produtos, int quantidadeDeElementos) {
    for(int atual = 0; atual < quantidadeDeElementos; atual++) {
        int analise = atual
        while(analise > 0 && produtos[analise].getPreco() < produtos[analise - 1].getPreco()) {
            Produto produto Analise = produtos[analise];
            Produto produtoAnaliseMenos1 = produtos[analise -1];
            produtos[analise] = produtoAnaliseMenos1;
            produtos[analise -1] = produtoAnalise;
            analise--;
        }
    }
}
```

Adicionaremos o `System.out` que já havíamos utilizado antes.

```
System.out.println("Estou na casinha " + atual);
```

Em cada uma das casinhas que serão analisadas, iremos acrescentar o `System.out` e informar nossa posição.

```
private static void novoSort(Produto[] produtos, int quantidadeDeElementos) {
    for(int atual = 0; atual < quantidadeDeElementos; atual++) {
        System.out.println("Estou na casinha " + atual);

        int analise = atual
        while(analise > 0 && produtos[analise].getPreco() < produtos[analise - 1].getPreco()) {
            Produto produto Analise = produtos[analise];
            Produto produtoAnaliseMenos1 = produtos[analise -1];
            produtos[analise] = produtoAnaliseMenos1;
            produtos[analise -1] = produtoAnalise;
            analise--;
        }
    }
}
```

O que acontece durante o processo: quando começamos a analisar os elementos, fazemos mudanças nas posições. Então, nós queremos imprimir as trocas. Cada uma delas é feita entre um item e outro. E o que estamos trocando? As posições `analise` com a `analise -1` .

```
System.out.println("Estou na casinha " + analise + " com " + (analise -1));
```

Também já podemos incluir o nome das variáveis:

```
private static void novoSort(Produto[] produtos, int quantidadeDeElementos) {
    for(int atual = 0; atual < quantidadeDeElementos; atual++) {
        System.out.println("Estou na casinha " + atual);

        int analise = atual
        while(analise > 0 && produtos[analise].getPreco() < produtos[analise - 1].getPreco()) {
            System.out.println("Estou trocando " + analise + " com " + (analise -1));
```

```
                Produto produto Analise = produtos[analise];
                Produto produtoAnaliseMenos1 = produtos[analise -1];
                System.out.println("Estou trocando " + produtoAnalise.getNome()
                                                + " com " + produtoAnaliseMenos1.getNome());
                produtos[analise] = produtoAnaliseMenos1;
                produtos[analise -1] = produtoAnalise;
                analise--;
            }
        }
    }
```

As variáveis ainda estão com nomes grandes. Nós iremos melhorá-los... Porém, antes veremos o algoritmo rodando. Mudaremos as posições até que todos os elementos estejam inseridos no lugar correto.

Vamos ver as trocas acontecendo?

```
Estou na casinha 0
Estou na casinha 1
Estou trocando 1 com 0
Estou trocando Jipe com Lamborghini
Estou na casinha 2
Estou trocando 2 com 1
Estou trocando Brasília com Lamborghini
Estou trocando 1 com 0
Estou trocando Brasília com Jipe
Estou na casinha 3
Estou trocando Smart com Lamborghini
Estou na casinha 4
Estou trocando 4 com 3
Estou trocando Fusca com Lamborghini
Estou trocando 3 com 2
Estou trocando Fusca com Smart
Estou trocando 2 com 1
Estou trocando Fusca com Jipe
Brasília custa 16000.0
Fusca custa 17000.0
Smart custa 46000.0
Jipe custa 46000.0
Lamborghini custa 1000000.0
```

O programa passou pela casinha 0 e recebeu a Lamborghini. Tem algum produto posicionado antes da casinha 0? Não. Então, ele não trocou os produtos de lugar.

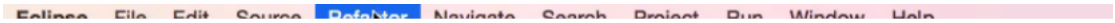
Depois, o programa recebeu o segundo carro e observou que o Jipe é mais barato do que a Lamborghini. Os carros foram trocados de posição. Por que o algoritmo não tentou seguir com as trocas? Porque quando ele chega na posição 0, precisa parar.

Vamos imprimir o resultado deste *array* e ver passo a passo o processo.

O código abaixo já imprime todos os produtos.

```
for(Produto produto : produtos) {
    System.out.println(produto.getNome() + " custa " +
            produto.getPreco());
}
```

Iremos adicionar uma nova impressão e extrair um método. Clico em `Extract Method` , que iremos chamá-lo de **imprime** (`imprime(produtos)`).



Dentro do nosso novo *Sort*, iremos imprimir os produtos no fim de cada nova rodada.

Iremos adicionar alguns `System.out` s:

```
imprime(produtos);
System.out.println();
System.out.println();
System.out.println();
System.out.println();
```

Ao rodarmos o algoritmo, o resultado será:

```
Estou na casinha 0
Lamborghini custa 1000000.0
Jipe custa 46000.0
Brasília custa 16000.0
```

```
Smart custa 46000.0
Fusca custa 17000.0
```

Primeiro, estou na casinha 0 (da Lamborghini). Não terei outra casinha para compará-la e a ordem permanecerá a mesma, com todos os produtos na mesma posição. Porém, quando eu for analisar a casinha 1, será diferente...

```
Estou na casinha 1
Estou trocando 1 com 0
Estou trocando Jipe com Lamborghini
Jipe custa 46000.0
Lamborghini custa 1000000.0
Brasília custa 16000.0
Smart custa 46000.0
Fusca custa 17000.0
```

Vou analisar o Jipe com a Lamborghini. Qual elemento é o mais barato? O Jipe. Por isso, o programa me informa que trocará o Jipe com a Lamborghini. E por que ele não continua com as trocas? Porque ele chegou no limite da esquerda, a casinha 0. Em seguida, o programa imprime o nosso *array* com os dois produtos reposicionados. Vou para o próximo produto...

```
Estou trocando 1 com 0
Estou trocando Jipe com Lamborghini
Jipe custa 46000.0
Lamborghini custa 1000000.0
Brasília custa 16000.0
Smart custa 46000.0
Fusca custa 17000.0
```

Estou na casinha 2, onde está a Brasília (que custa R$16000). Comparo o produto com a Lamborghini (que custa R$1000000) e percebo que a Brasília é menor. Troco os dois produtos de posição. Em seguida, comparo a Brasília com o Jipe e farei uma nova troca. Quando ela ocupar a posição 0, chegaremos no limite da esquerda e o programa não fará mais alterações. Continuarei a análise na casinha 3.

```
Estou na casinha 3
Estou trocando 3 com 2
Estou trocando Smart com Lamborghini
Brasília custa 16000.0
Jipe custa 46000.0
Smart custa 46000.0
Lamborghini custa 1000000.0
```

Esta é a casinha do Smart. Comparo o preço do produto com o da Lamborghini. Qual tem o menor valor? O Smart. Farei a análise de preços com o Jipe. Porém, a nossa comparação deve deve identificar o produto **menor**...

```
while(analise > 0 && produtos[analise].getPreco() < produtos[analise -1].getPreco())
```

Então, o que farei é verificar: o valor R$46000 é menor que R$ 46000? Não. Logo, o programa ficará no Jipe, sem realizar a segunda troca. Em seguida, ele imprime como ficou a nova ordem: Brasília, Jipe, Smart, Lamborghini e Fusca.

Agora vou para a última casinha.

```
Estou na casinha 4
Estou trocando 4 com 3
Estou trocando Fusca com Lamborghini
Estou trocando 3 com 2
Estou trocando Fusca com Smart
Estou trocando 2 com 1
Estou trocando Fusca com Jipe
Brasília custa 16000.0
Fusca custa 17000.0
Jipe custa 46000.0
Smart custa 46000.0
Lamborghini custa 1000000.0
```

Estou na casinha 4. O programa irá comparar o Fusca com os outros elementos até encontrar o lugar correto para inseri-lo. Serão feitas trocas de posições com a Lamborghini, com o Smart e com o Jipe. No entanto, quando a análise comparar o Fusca com a Brasília, não haverá trocas. O programa detém o processo aqui.

A ordem final dos elementos será: Brasília, Fusca, Jipe, Smart e Lamborghini.

Como ficou o nosso algoritmo?

Ele passa em todas as casinhas e a partir de cada uma delas, analisa os elementos anteriores. Quando os elementos verificados atendiam as condições, eram trocados de lugar. O processo foi repetido diversas vezes até que as trocas não fossem mais necessárias. O reposicionamento se tornava desnecessário em duas condições:

- Quando a análise chegava na posição limite à esquerda
- Quando o preço do elemento anterior era menor do que o analisado.

Temos assim a ordenação que insere cada elemento na posição adequada.

Pequenas refatorações e melhoria do código

Nós acompanhamos o nosso algoritmo rodando passo a passo. Observe que percebemos um detalhe: na primeira casinha, quando o `atual` é 0, `analise` também será igual 0.

```
int analise = atual;
```

Se `analise` é igual a 0, ele nem entrará no `while` .

```
while(analise > 0 && produtos[analise].getPreco() < produtos[analise-1].getPreco())
```

É óbvio... quando você observa o primeiro produto e procura onde inseri-lo, não existe outro item para compará-lo. É desnecessário começar da posição 0. Por isso, é comum que o algoritmo seja iniciado a partir do 1. Então, para o `novoSort` ,

modificamos o `atual` para começar a partir do 1:

```
for(int atual = 1; atual < quantidadeDeElementos; atual++) {
}
```

Da mesma forma, é impossível começar a ordenação quando recebemos a primeira carta de baralho... Também não podemos fazer nada quando analisamos o primeiro produto, ou o primeiro preço de uma lista. Por isso, nosso algoritmo está ignorando a primeira casinha. Se fizermos o teste, veremos que tudo está funcionando corretamente.

Temos como resultado a seguinte ordenação:

```
Brasília custa 16000.0
Fusca custa 17000.0
Jipe custa 46000.0
Smart custa 46000.0
Lamborghini custa 1000000.0
```

Observe que estamos trocando dois produtos de diferentes posições dentro do `while ( analise e analise -1 )`:

```
while(analise > 0 && produtos[analise].getPreco() < produtos[analise-1].getPreco()) {

    Produto produtoAnalise = produtos[analise];
    Produto produtoAnaliseMenos1 = produtos[analise -1];

    produtos[analise] = produtoAnaliseMenos1;
    produtos[analise -1] = produtoAnalise;
}
```

Abaixo, no `selectionSort` , também é feita a troca entre o `atual` por `menor` :

```
private static void selectionSort(Produto[] produtos, int quantidadeDeElementos) {
    for(int atual = 0; atual < quantidadeDeElementos -1; atual++) {

        int menor = buscaMenor(produtos, atual, quantidadeDeElementos -1);
        Produto produtoAtual = produtos[atual];
        Produto produtoMenor = produtos[menor];

        produtos[atual] = produtoMenor;
        produtos[menor] = produtoAtual;
    }
}
```

Trocar duas posições de um *array* é uma tarefa muito comum. Então, precisamos saber fazer isto de diversas maneiras.

Vamos extrair esse código que troca posições, para deixar meu algoritmo mais simples. Então substituímos isso:

```
            Produto produto Analise = produtos[analise];
            Produto produtoAnaliseMenos1 = produtos[analise -1];
            System.out.println("Estou trocando " + produtoAnalise.getNome()
```

```
                                        + " com " + produtoAnaliseMenos1.getNome());
                produtos[analise] = produtoAnaliseMenos1;
                produtos[analise -1] = produtoAnalise;
```

Por isso:

```
   troca(produtos, analise, analise -1);
```

Nosso laço ficará assim:

```
  while(analise > 0 && produtos[analise].getPreco() < produtos[analise - 1].getPreco()) {
      System.out.println("Estou trocando " + analise + " com " + (analise -1));
      troca(produtos, analise, analise -1);
      analise--;
  }
```

Usamos o comando "Ctrl + 1" com essa nova linha de código selecionada e vamos criar o método (`Create method 'troca(Produto[], int, int)'`)

```
  private static void troca(Produto[] produtos, int analise, int i) {
  }
```

O programa criou o método que recebe os produtos `analise` e `i` . Como os nomes não ficaram bons, vamos alterá-los para `primeiro` e o `segundo` :

```
  private static void troca(Produto[] produtos, int primeiro, int segundo) {
  }
```

Vamos colar o código que extraímos anteriormente e incluir as variáveis `primeiro` e `segundo` :

```
  private static void troca(Produto[] produtos, int primeiro, int segundo) {
      Produto primeiroProduto = produtos[primeiro];
      Produto segundoProduto = produtos[segundo];
      System.out.println("Estou trocando " + primeiroProduto.getNome()
                                              + " com " + segundoProduto.getNome());
      produtos[primeiro] = segundoProduto;
      produtos[segundo] = primeiroProduto;
  }
```

Então, estamos trocando o `primeiroProduto` com o `segundoProduto` .

No nosso `novoSort` , estamos trocando o `analise` com `analise -1` , logo:

```
  troca(produtos, analise, analise -1);
```

No caso do `selectionSort` , a troca é feita entre o `atual` e o `menor` , dentro do *array* de `produtos` .

```
private static void selectionSort(Produto[] produtos, int quantidadeDeElementos) {
    for(int atual = 0; atual < quantidadeDeElementos -1; atual++) {
        System.out.println("Estou na casinha " + atual);

        int menor = buscaMenor(produtos, atual, quantidadeDeElementos -1);
        System.out.println("Trocando " + atual + " com o " + atual);

        troca(produtos, atual, menor);
    }

}
```

Observe que na nossa função já iremos imprimir os nomes das variáveis que estamos trocando.

```
private static void troca(Produto[] produtos, int primeiro, int segundo) {
    Produto primeiroProduto = produtos[primeiro];
    Produto segundoProduto = produtos[segundo];
    System.out.println("Estou trocando " + primeiroProduto.getNome()
                                            + " com " + segundoProduto.getNome());
    produtos[primeiro] = produtoAnaliseMenos1;
    produtos[segundo] = produtoAnalise;
}
```

Temos a seguinte linha imprimindo a posição no `while`:

```
System.out.println("Estou trocando " + analise + " com " + (analise -1));
```

Poderíamos inclui-la dentro do algoritmo, já com as variáveis `primeiro` e `segundo`.

```
private static void troca(Produto[] produtos, int primeiro, int segundo) {
    System.out.println("Estou trocando " + primeiro + " com " + segundo);
    Produto primeiroProduto = produtos[primeiro];
    Produto segundoProduto = produtos[segundo];
    System.out.println("Estou trocando " + primeiroProduto.getNome()
                                            + " com " + segundoProduto.getNome());
    produtos[primeiro] = produtoAnaliseMenos1;
    produtos[segundo] = produtoAnalise;
}
```

Para não deixarmos nenhuma repetição, iremos excluir também o trecho do `selectionSort`:

```
System.out.println("Trocando " + atual + " com o " + menor);
```

O `selectionSort` ficará assim:

```
private static void selectionSort(Produto[] produtos, int quantidadeDeElementos) {
    for(int atual = 0; atual < quantidadeDeElementos -1; atual++) {
        System.out.println("Estou na casinha " + atual);
```

```
            int menor = buscaMenor(produtos, atual, quantidadeDeElementos -1);
            troca(produtos, atual, menor);
        }

    }
```

Então, iremos definir quais elementos serão trocados entre si, para depois serem reposicionados. O nosso código ficará assim:

```
    private static void troca(Produto[] produtos, int primeiro, int segundo) {
        System.out.println("Estou trocando " + primeiro + " com " + segundo));

        Produto primeiroProduto = produtos[primeiro];
        Produto segundoProduto = produtos[segundo];
        System.out.println("Estou trocando " + primeiroProduto.getNome()
                                        + " com " + segundoProduto.getNome());
        produtos[primeiro] = segundoProduto;
        produtos[segundo] = primeiroProduto;
    }
```

No algoritmo `novoSort`, que detecta a posição em que o elemento será inserido, fazíamos uma troca com o item anterior. Assim como no algoritmo `selectionSort`, que seleciona qual item merece estar em cada casinha, também fazíamos uma troca e reposicionávamos os elementos. Nós realizávamos trocas nos dois algoritmos. Logo iremos extrai-los desta função.

Novamente, iremos rodar o algoritmo e conferir se funcionou corretamente.

```
Brasília custa 16000.0
Fusca custa 17000.0
Jipe custa 46000.0
Smart custa 46000.0
Lamborghini custa 1000000.0
```

Temos a ordenação...

Agora vamos tirar o `novoSort` e colocar o `selectionSort`.

```
selectionSort(produtos, produtos.length);
//novoSort(produtos, produtos.length);
```

Rodaremos o algoritmo para verificar se também funciona corretamente. O resultado será:

```
Estou na casinha 2
Estou trocando 2 com 3
Estou trocando Lamborghini com Smart
Estou na casinha 3
Estou trocando 3 com 4
Estou trocando Lamborghini com Jipe
Brasília custa 16000.0
Fusca custa 17000.0
```

```
Smart custa 46000.0
Jipe custa 46000.0
Lamborghini custa 1000000.0
```

A ordem está correta. Voltaremos para o algoritmo algoritmo que estamos trabalhando, o `selectionSort` :

```
//selectionSort(produtos, produtos.length);
novoSort(produtos, produtos.length);
```

Um detalhe foi melhorado no nosso algoritmo: nós passamos a utilizar o `atual` a partir da posição `1` .

Porém, percebemos que são muito comuns **trocas** nos *array*s e que é interessante deixa-las organizadas em um único lugar. Extraímos tudo para uma função, que irá trocar a posição dos elementos dentro do *array*.

Simulando no papel com o nosso código

Nós já implementamos o algoritmo, agora vamos simulá-lo na memória.

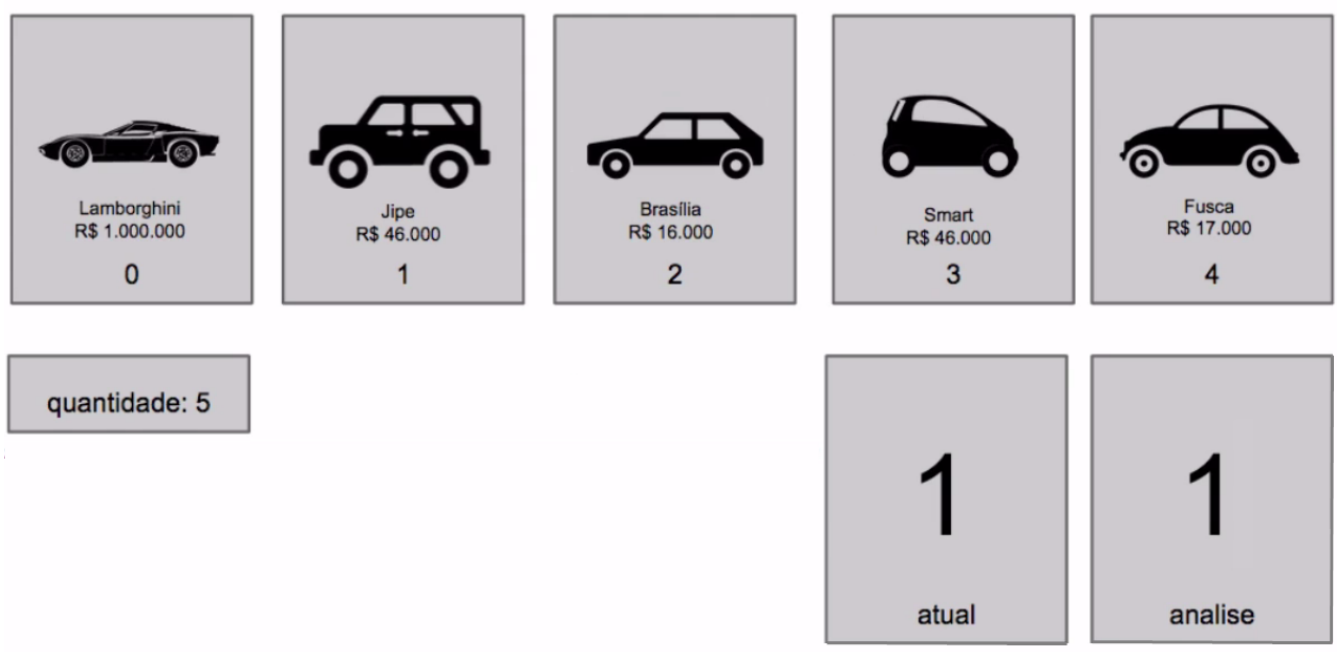
```
for(int atual = 1; atual < quantidadeDeElementos; atual++ ) {
    int analise = atual;
    while(analise > 0 &&
            produtos[analise].getPreco() < produtos[analise-1].getPreco())
        troca(produtos, analise, analise - 1);
        analise--;
    }
}
```

O que nós fizemos primeiro? Trabalhamos com o **array** e a *quantidade de elementos*.

Precisamos usar a variável `atual` e `analise` na memória.

Iremos começar com `atual` igual a 1. Por que? Porque como só recebemos uma carta de baralho, não existe dúvidas sobre a posição em que o elemento 0 deve ser inserido. Por isso, começamos com o segundo elemento, o **Jipe**.

```
int analise = atual;
```



`atual` é igual a 1. `Analise` é igual a `atual` e também será igual a 1. Ela é maior que 0? Sim. O preço do `produto[analise]` (o Jipe) é menor do que o `produtos[analise-1]` (a Lamborghini)? Sim, o Jipe é mais barato. Então, trocaremos os dois produtos de posição.

Continuaremos com `analise--` ... Diminuiremos 1 da `analise` , que será igual a 0 e não atende a primeira condição (`while(analise > 0`). Não faremos novas alterações.

Passamos para o próximo elemento na posição 2, a **Brasília**.

A variável `atual` será igual a 2. Ela é menor que a `quantidadeDeElementos` ? Sim. Como `analise` é igual a `atual` , será maior que 0. Vamos verificar se atende a segunda condição: o preço da Brasília é menor do que a Lamborghini? Sim. Trocaremos os elementos de lugar. Depois, iremos diminuir 1 da `analise` .

`analise` é igual 1. Verificaremos se a **Brasília** está posicionada adequadamente. O preço do produto (que custa R$16000) é menor do que o Jipe (que custa R$46000)? Sim. Trocaremos os elementos de lugar e a Brasília estará posicionada na **primeira casa**...

Ela já ocupa a posição correta? Se avançarmos para o elemento anterior, `analise` será igual a 0. Como não atende a primeira condição, a ordem ficará como está.

Seguimos para o próximo produto, o **Smart**. `atual` será igual a 3.

Aonde o produto merece ser inserido? Até o momento, só podemos compará-lo com os quatro itens recebidos anteriormente. Começamos com `analise` igual a 3 e a variável é maior que 0. `produtos[analise].getPreco()` é menor do que `produtos[analise-1].getPreco()` ... Como atende as duas condições do `while` , iremos trocá-los de lugar.

Iremos continuar com a análise do **Smart**. Agora vamos compará-lo com o **Jipe**. `analise` será igual a 2 e maior que 0. No entanto, o preço do produto não é menor do que o preço do anterior. Os produtos irão permanecer na mesma posição.

Vamos para o próximo elemento, `atual` será igual a 4, que é igual a 5 (e menor que a `quantidadeDeElementos`). `analise` será igual a 4 e o produto analisado será o **Fusca**.

`analise` é maior que 0? Sim. Seguimos para a próxima condição: `produtos[analise].getPreco()` é menor do que `produtos[analise-1].getPreco()` ? Sim, porque o preço do Fusca é menor do que da Lamborghini. Vamos reposicioná-los na lista.

Diminuiremos 1 da `analise` , que será igual a 3.

A variável é maior que 0 e o preço do Fusca é menor do o Smart. Vamos modificar a ordem.

Seguimos para `analise` é igual a 2.

O produto atende as duas condições. Como o preço do Fusca é menor do que Jipe, vamos trocá-los de lugar.

Agora `analise` é igual a 1.

Também atendeu a primeira condição, porém o preço do Fusca é maior do que a **Brasília** e por isso, não atende a segunda condição. Isto significa que o Fusca merece ficar na posição 1.

Seguimos para `atual` igual a 5.

Ele respeita o `for` que específica que `atual` precisa ser menor que `quantidadeDeElementos` ? Não. Então, paramos por aqui. Nosso algoritmo já está ordenado.

Agora nosso *array* está ordenado, podemos obter várias informações sobre os elementos, além do beneficio de tê-lo organizado. A ordenação nos possibilita resolver diversos problemas do mundo real:

- Encontrar quais são os maiores e os menores
- Os melhores e os piores

Com uma ordem (uma classificação), conseguimos dizer qual elemento é o primeiro, o segundo e o terceiro... Conseguimos identificar qual é o 10º ou o 15º maior item, devido à posição dos elementos que estão ordenados no *array*. Todos esses sistemas podem ser implementados.

Conseguimos isto, porque nós introduzimos um segundo algoritmo que passa por cada elemento e a partir da posição dos itens analisados observamos os anteriores e identificamos aonde ele merece ser inserido. Assim, posicionamos os elementos de acordo com a ordem.

Insertion Sort

Agora que já temos o nosso algoritmo, vamos ver o que acontece quando simulamos na nossa memória (nós também armazenamos variáveis na mente) ou no computador? Iremos usar no exemplo as variáveis `atual` e `analise` e cartas de baralho.

Assim que eu recebo a primeira carta, ainda não posso fazer nada com ela. A minha análise começa apenas quando recebo a segunda. Com os dois elementos na minha mão, comparo a minha carta analisada com a anterior. Percebo que ela é maior e por isso, irá continuar na mesma posição. Recebo uma nova carta, que será a próxima a ser analisada (minha variável `analise`) e será a terceira na fileira. Chega mais uma... Já tenho quatro cartas na minha mão. Comparo-a com o elemento anterior (minha `analise` anterior) e decido movê-la para a posição 3. Sigo comparando-a com as cartas anteriores e então identifico que ela é menor do que as outras. Ela passa a ocupar a segunda posição na minha fileira. Recebo mais uma carta e a analiso, sempre em comparação com as anteriores. Decido que a minha variável irá ocupar a posição 2. Agora tenho as cinco cartas ordenadas, todas do nipe de Copa: 3, 5, 6, 9 e 10.

Observe que quando recebemos cartas de baralho, vamos aos poucos encontrando a posição adequada para cada uma. O que fazemos durante o processo é inserir os itens na posição correta. Este algoritmo receberá o nome de **Insertion Sort**. É ele que nos faz inserir os elementos na posição adequada.