02

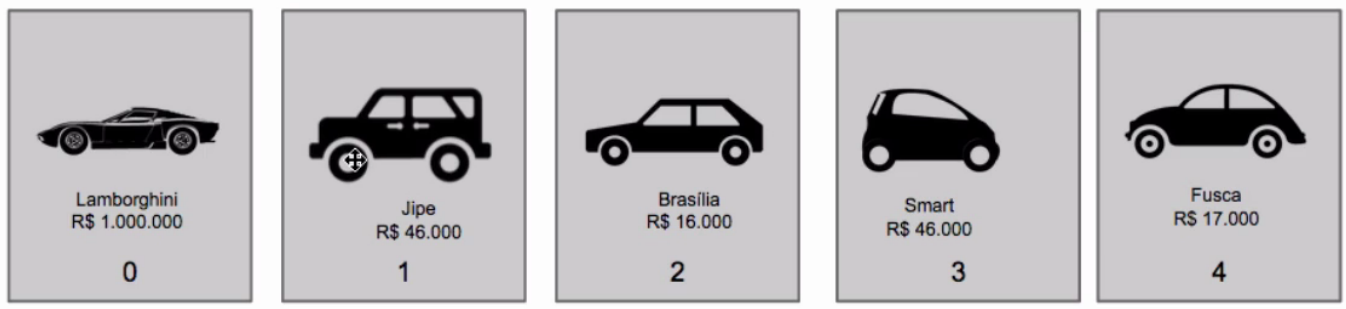
Ordenando ao selecionar o mais barato

Ordenando ao colecionar o mais barato

Ordenou os seus produtos? Eu irei ordenar os meus agora...

Tentei encontrar uma maneira de ordenar os produtos:

- Lamborghini custa R$ 1.000.000
- Jipe custa R$ 46.000
- Brasília custa R$ 16.000
- Smart custa R$ 46.000
- Fusca custa R$ 17.000



Lista de produtos

De todos os carros, a **Brasília** é o mais barato. Logo, este será o primeiro carro na nossa lista.

Em seguida, continuamos buscando o carro com menor preço. O Fusca custa R$ 17.000 é o segundo mais barato. Mudaremos ele de posição.

Tanto o Smart quando o Jipe custam R$ 46.000. Vamos movê-los na lista também.

A Lamborguini custa R$ 1.000.000, sendo o carro mais caro dos cinco elementos. Então, será o elemento no fim da nossa lista.

Os elementos estão ordenados... Como eu criei a ordem?

Vamos descrever passo a passo o que eu fiz durante o processo: no começo a Lamboghini estava no início. Mas será que ela realmente é o carro com o menor preço? Vamos analisar se temos outros carros mais baratos que a Lamborghini...

- Jipe custa R$ 46.000
- Brasilia custa R$ 16.000
- Smart custa R$ 46.000
- Fusca custa R$ 17.000

Em comparação com a Lamborghini, o Jipe é mais barato, mas a Brasília é ainda mais. O Smart também é mais barato, porém a Brasília tem o menor preço. O Fusca é barato, mas a Brasília continua sendo mais barata. Na verdade, vários carros são mais baratos do que a Lamborghini. E a **Brasília** é o elemento mais barato. Isto sigifica que depois da Lamborghini, todos

os carros são mais baratos. Não faz sentido que ela seja o primeiro item da lista. Então, a Brasília passa a estar no início, porque é o produto mais barato.

Agora nós sabemos que quem ocupa a primeira casa é o carro mais barato. Não precisamos mais nos preocupar com ele.

Seguimos para o próximo carro. Vamos ver quem merece estar na próxima posição? Da segunda casa adiante, qual carro é o mais barato? É o **Fusca**. Trocamos o Fusca de lugar e agora sabemos qual é o segundo carro mais barato.

Quero encontrar o terceiro carro mais barato. Aquele que merece ocupar a terceira casa e será o elemento com o menor preço entre os restantes... Qual é o mais barato? Iremos selecionar o **Smart**.

Para finalizarmos, qual é o carro mais barato da quarta casa para o fim? O Jipe. Então, vamos trocá-lo com a Lamborguini. Com a nossa nova lista, simplificamos o processo de busca.

A ideia de ordenarmos os elementos a partir do mais barato é para simplificarmos a busca do menor preço de acordo com a posição.

Vamos refazer nossos passos?

A partir da primeira casa, fizemos a pergunta: qual é o carro mais barato? Identificamos que era a Brasília. Vamos trocá-la de lugar. Encontrar um produto com referência na posição era algo que já havíamos feito anteriormente...

Seguimos com a pergunta: a partir da segunda posição, qual é o mais barato? O Fusca. Nós já escrevemos o algoritmo `maisBarato` a partir de uma posição. O método "encontre o menor a partir de determinada posição" nós já temos. O carro mais barato desda a terceira casa é o Smart, então, trocamos o produto de posição.

O carro mais barato a partir da quarta posição é o Jipe. Movemos o elemento de lugar e resolvemos a questão de como encontrar o carro mais barato, com a ordenação. Porém, não encontramos o `maisBarato` apenas uma vez... Respondemos a mesma pergunta cinco vez. Quando encontramos o carro mais barato, trocamos o elemento de lugar. Em seguida, encontramos o segundo mais barato e o trocamos de lugar. Fazemos o mesmo com o terceiro, quarto e quinto elementos. Quando chegamos no último, já sabíamos que ele era o carro mais caro de todos.

Então, o que fizemos? Varremos nosso *array* procurando o mais barato a partir de uma determinada posição. Nós perguntamos: "A partir desse carro, qual é o mais barato? É esse." Trocamos o elemento de lugar na lista. E assim, ficamos felizes com o resultado.

Vamos passar isto para o código? É o nosso próximo passo.

Implementando em Java a ordenação pela seleção do menor valor

Chegou a hora de tentarmos implementar nosso processo: a sequência de passos que ordena o nosso *array*.

```
private static int buscaMenor(Produto[] produtos, int inicio, int termino) {
    int maisBarato = inicio;
    for(int atual = inicio; atual <= termino; atual++){
        if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()) {
            maisBarato = atual;

        }
    }
    return maisBarato
}
```

Vamos começar a criar o processo: após entrarmos no pacote `br.com.alura.algoritmos`, criaremos a nova classe `TestaOrdenacao`. Ela também terá um método `main`.

Dentro do metódo `main`, vou copiar um pedaço do código, em que temos nosso *array* de produtos:

```
package br.com.alura.algoritmos;

public class TestaOrdenacao {

    public static void main(String[] args) {
        Produto produtos[] = {
                new Produto("Lamborghini", 1000000),
                new Produto("Jipe", 46000),
                new Produto("Brasília", 16000),
                new Produto("Smart", 46000),
                new Produto("Fusca", 17000)
        };
    }
}
```

Temos os elementos (Lamborghini, Jipe, Brasília, Smart e Fusca) e gostaríamos de ordená-los. Você se lembra como nós fizemos isto? Nós analisamos cada produto da nossa lista com o `for` que já usamos anteriormente.

```
for(int atual = 0; atual < produtos.length; atual++)
```

Então, queremos verificar até a última casa com produtos. Para ordenar os elementos, nós observamos o produto da primeira casa e fazemos a pergunta "a partir daqui, qual dos produtos é o menor de todos?" Encontramos a resposta e, então, movemos o elemento mais barato para o início da lista.

Você lembra que já escrevemos a função que busca o menor elemento? Era `buscaMenor`:

```
private static int buscaMenor(Produto[] produtos, int inicio, int termino) {
    int maisBarato = inicio;
    for(int atual = inicio; atual <= termino; atual++){
        if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()) {
            maisBarato = atual;
        }
    }
    return maisBarato;
}
```

Como funciona `buscaMenor`? Nós colocamos: os produtos (`produtos`), o início (`inicio`) e o fim (`termino`).

```
private static int buscaMenor(Produto[] produtos, int inicio, int termino)
```

Então, se dissermos para o programa "busque o menor produto (`produtos`) , a partir da posição atual (`atual`), até o fim do nosso *array* (`produtos.length`)", ele irá buscar o menor de todos a partir da primeira posição .

```
int menor = buscaMenor(produtos, atual, produtos.length)
```

Agora que encontramos o menor produto, quero trocá-lo de posição para a casa que seja do menor preço. Isto é: o `produtoAtual` , que é o elemento na posição `produtos[Atual]` .

```
Produto produtoAtual = produtos[atual];
```

O `produtoMenor` é o elemento na posição `produtos[menor]` .

```
Produto produtoMenor = produtos[menor];
```

Vou fazer a inversão: o `produtos[atual]` é o `produtoMenor` e o `produtos[menor]` é o `produtoAtual` .

```
for(int atual =0; atual < produtos.length; atual++) {
    int menor = buscaMenor(produtos, atual, produtos.length);
    Produto produtoAtual = produtos[atual];
    Produto produtoMenor = produtos[menor];
    produtos[atual] = produtoMenor;
    produtos[menor] = produtoAtual;
}
```

O que nós fizemos? Passamos por cada casa com produto a partir da **primeira casa**, e verificamos qual produto tem menor preço. Após encontrarmos o menor, trocamos os elementos de lugar. Seguimos para o produto da **segunda casa** e buscamos qual elemento é o menor a partir dele. Descobrimos qual é e o movemos de lugar. Repetimos o mesmo processo com os

elementos da terceira, quarta e quinta casa. Assim, fomos encontrando os produtos com menor preço e ordenando como "primeiro menor", "segundo menor", "terceiro menor"... Quando identificamos todos os produtos, nosso *array* estava ordenado corretamente.

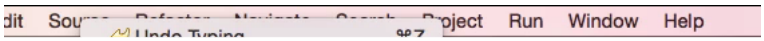
Vamos imprimir o resultado?

Para isso, criaremos um `for` que passe por todos os produtos e imprima as informações de todos:

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

Iremos imprimir a informação dos produtos.

Para rodar o nosso programa, vamos clicar no botão direito e depois, em *Run As* e *Java Application*.



Porém, o programa irá mostrar na nossa tela uma mensagem, avisando que algo não deu certo no processo.

```
Exception in thread "main"java.lang.ArrayIndexOutOfBoundsException: 5
        at br.com.alura.algoritmos.TestaOrdenacao.buscaMenor(TestaOrdenacao.java: 3)
        at br.com.alura.algoritmos.TestaOrdenacao.main(TestaOrdenacao.java: 15)
```

O que fizemos errado? Em algum lugar, o programa tentou acessar a posição `5`. A mensagem nos mostra que foi no `buscaMenor`.

Do lado do erro, temos um link que nos leva direto para o `buscaMenor`:

```
private static int buscaMenor(Produto[] produtos, int inicio, int termino) {
    int maisBarato = inicio;
    for(int atual = inicio; atual <= termino; atual++){
        if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()) {
            maisBarato = atual;
        }
    }
    return maisBarato;
}
```

Observe que a variável `atual` percorre até `termino` que é igual a 5, pois `atual <= termino`.

```
for(int atual = inicio; atual <= termino; atual++){
```

Isto significa que o programa verifica até a posição `5`.

`<=` (menor ou igual) faz muita diferença no nosso algoritmo. Como a nossa função busca exatamente até determinado elemento, quando a chamamos não queremos que verifique `produtos.length`. De fato, o que desejamos que ela faça é que ela vá até `produtos.length - 1`. Você se lembra que na nossa classe `TestaMenorPreco`, nossa última posição era `4` (que era o tamanho do *array* menos `1`).

```
Produto produtos[] = {
        new Produto ("Lamborghini", 1000000),
        new Produto("Jipe", 46000),
        new Produto("Brasília", 16000),
        new Produto("Smart", 46000),
        new Produto("Fusca", 17000)
};

int maisBarato = buscaMenos(produtos, 0, 4)
System.out.println(maisBarato);
System.out.println("O carro" + produtos[maisBarato].getNome()
```

```
                + " é o mais barato, e custa"
                + produtos[maisBarato].getPreco());
```

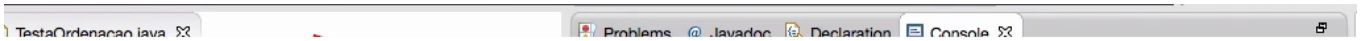
Voltamos para a classe `TestaOrdenacao` ... Na nossa função, a verificação deve ser feita de `atual` (a primeira posição) até `produtos.length - 1` (a última posição). Não é o tamanho do nosso *array*... Um erro que ocorre porque nossa função está utilizando `<=` (menor ou igual).

```
for(int atual =0; atual < produtos.length; atual++) {
    int menor = buscaMenor(produtos, atual, produtos.length - 1);
    Produto produtoAtual = produtos[atual];
    Produto produtoMenor = produtos[menor];
    produtos[atual] = produtoMenor;
    produtos[menor] = produtoAtual;
}
```

Dependendo da maneira como for implementada, algumas vezes veremos a função com `<` (menor), outras com `=` (igual). Nós escolhemos implementar a função com `<=` , por isso iremos utilizar a última posição do *array* (`produtos.length - 1`). Vários livros fazem da mesma forma. Caso outro livro mostre uma implementação diferente, que use `<` , não precisaremos do `-1` e utilizaremos apenas `produtos.length` . Tudo depende da teoria, do livro que você estiver utilizando. No caso, estamos seguindo um exemplo com "<=", logo, vamos usar `produtos.length - 1` .

Agora iremos testar o nosso código e o programa irá nos mostrar nosso *array* ordenado:

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



Vamos recapitular o que nós fizemos?

Nós temos um *array* de produtos, verificamos em cada casa. Em seguida, observamos todos os elementos à direita, encontro o mais barato e troco de lugar. Depois, passamos a ignorar o produto, porque ele já está ordenado. Continuamos com a pergunta: "A partir do atual, qual é o mais barato?" Descobrimos o elemento e o passamos para as primeiras posições. Repetimos o processo, apenas com os produtos ainda não ordenados.

Nós sempre verificamos a partir do `atual` e vamos até onde termina o *array*, que finaliza no último elemento (`produtos.length - 1`). Fazemos isto, porque no `for` nós buscamos `<=` ao último elemento.

```
for(int atual = inicio; atual <= termino; atual++) {
```

No fim, o nosso código irá imprimir os nossos elementos em ordem.

Algoritmos e o 'menos 1'

Nós escrevemos o código da nossa ordenação, após selecionarmos o maior elemento de todos, o segundo maior, o terceiro maior... Fomos identificando um de cada vez, até ordenarmos o *array*. Agora podemos responder rapidamente diversas perguntas diferentes:

- Quem é o primeiro ou o segundo elemento mais barato?
- Quem é o terceiro mais caro?
- Qual é o do meio? Qual é o do meio menos 1?
- Qual é o mais caro? Qual é o mais barato?

Todas estas perguntas podemos responder de forma rápida...

Observe o nosso código:

```
for(int atual = 0; atual < produtos.length; atual++) {
    int menor = buscaMenor(produtos, atual, produtos.length - 1);
    Produto produtoAtual = produtos[atual];
    Produto produtoMenor = produtos[menor];
    produtos[atual] = produtoMenor;
    produtos[menor] = produtoAtual;
}

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

Será que nós conseguimos melhorar o código ainda mais? Nós estamos passando por todas as casas dos produtos (`0, 1, 2, 3 e 4`). Porém, quando os 4 primeiros produtos estão ordenados, a última casa fica com o item que sobra. Por isso, é muito comum que o nosso `for` termine em `produtos.length - 1` ... Porque quando sobra apenas o último elemento para ser ordenado, ele já ocupa a casa correta. Ele também foi ordenado.

O último elemento que sobra no processo de ordenação é o mais caro ou o maior de todos. Então, é comum que o algoritmo seja implementado com `-1` .

Assim, cada elemento ocupará a sua posição correta.

Extraindo a ordenação por seleção de elementos

Indicamos o produto mais barato, o segundo mais barato, o terceiro mais barato, do nosso código... E com isso, temos um *array* organizado e podemos extrair uma função de ordenação.

```
for(int atual = 0; atual < produtos.length - 1; atual++) {
    int menor = buscaMenor(produtos, atual, produtos.length - 1);
    Produto produtoAtual = produtos[atual];
    Produto produtoMenor = produtos[menor];
    produtos[atual] = produtoMenor;
    produtos[menor] = produtoAtual;
}
```

Lembre-se: uma função de ordenação irá receber o *array* que será ordenado. Vamos recortar o código

```
for(int atual = 0; atual < produtos.length - 1; atual++) {
        int menor = buscaMenor(produtos, atual, produtos.length - 1);
        Produto produtoAtual = produtos[atual];
        Produto produtoMenor = produtos[menor];
        produtos[atual] = produtoMenor;
        produtos[menor] = produtoAtual;
    }
```

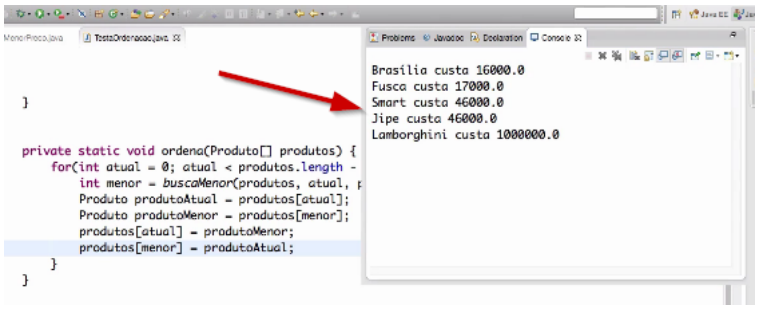
E fazer com que ele ordene os nossos produtos:

```
ordena(produtos);
```

Criamos o método `ordena` com a parte recortada acima:

```
private static void ordena(Produto[] produtos) {
    for(int atual = 0; atual < produtos.length - 1; atual++) {
        int menor = buscaMenor(produtos, atual, produtos.length - 1);
        Produto produtoAtual = produtos[atual];
        Produto produtoMenor = produtos[menor];
        produtos[atual] = produtoMenor;
        produtos[menor] = produtoAtual;
    }
}
```

Como nosso *array* troca as posições dos elementos que estão dentro, vamos ordená-lo. Ao testamos novamente o código, veremos que funciona:



Lembrando que em diversas linguagens, seja porque não sabemos o limite ou quantos elementos temos dentro do *array*, é comum que uma função de ordenação receba o `tamanho`. É comum recebermos os valores `produtos.length` e `produtos.length -1` como argumento. Isto significa que o tamanho do *array* (`produtos.length`) costuma ser usado em funções de ordenação.

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

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

Como em várias linguagem isto acontece, iremos substituir `produtos.length` por `tamanho` no nosso código:

```
private static void ordena(Produto[] produtos, int tamanho) {
    for(int atual = 0; atual < tamanho - 1; atual++) {
        int menor = buscaMenor(produtos, atual, tamanho - 1);
        Produto produtoAtual = produtos[atual];
        Produto produtoMenor = produto[menor];
        produtos[atual] = produtoMenor;
        produtos[menor] = produtoAtual;
    }
}
```

Usamos o termo `tamanho`, porém isto é uma ilusão. Pode ser que existam 50 posições, mas que na verdade tenhamos apenas 10 elementos. Logo, ao invés de `tamanho`, iremos chamar `quantidadeDeElementos`, porque é quantidade de elemento que temos no *array*. Então, iremos de 0 até a `quantidade de elementos` menos `1`. O nosso código ficará assim:

```
private static void ordena(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 = produto[menor];
        produtos[atual] = produtoMenor;
        produtos[menor] = produtoAtual;
    }
}
```

Em casos em que não sabemos com exatidão a quantidade de elementos que temos dentro do *array*, usamos o `length` como parâmetro para a nossa função. Em Java, nós temos como saber e podemos usar `produtos.length`.

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

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

Em outras linguagens, na definição ou o formato de uma ordenação é comum recebermos este argumento. Por isso, estamos criando o código detalhadamente para que ele funcione corretamente.

Em seguida, iremos testar o código e ver o que está acontecendo.

Visualizando a troca de posições dos elementos durante a seleção

Vamos testar passo a passo o que está acontecendo com o nosso código... Para isso, colocaremos alguns `System.out` s a mais para ordenarmos o nosso *array*. Quando passarmos por cada uma das casas, colocamos esses `System.out` s para que o programa imprima "Estou na casinha 0, 1, 2, 3..."

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

Porém, não iremos passar na casinha 4. Por ser a última, sabemos que ela é o maior elemento de todos.

```
private static void ordena(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);
        Produto produtoAtual = produtos[atual];
        Produto produtoMenor = produto[menor];
        produtos[atual] = produtoMenor;
        produtos[menor] = produtoAtual;
    }
}
```

Além de passarmos pelas casas, o que faremos depois? Vamos trocar os elementos de lugar:

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

Quais elementos serão trocados? Nós iremos trocar o `getNome()` com o `produtoMenor.getNome()` .

```
System.out.println("Trocando "+ produtoAtual.getNome() + " " + produtoMenor.getNome());
```

Estamos trocando uma posição por outra, o que significa "trocar esse carro com aquele".

```
private static void ordena(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 " + menor)
        Produto produtoAtual = produtos[atual];
        Produto produtoMenor = produto[menor];

        System.out.println("Trocando "+ produtoAtual.getNome() + " " + produtoMenor.getNome());

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

E veremos as trocas acontecerem! A quantidade de alterações deverá ser a mesma que o número de casinhas menos 1. Ou seja, se temos cinco casas, apenas verificaremos as casas 0, 1, 2, 3. A casa 4 não será analisada.

Ao rodarmos o programa, poderemos conferir as trocas que aconteceram:

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

Estamos na casinha 0 e trocamos a Lamborghini pelo 2 (a Brasília). A alteração faz sentido... A Brasília era o carro mais barato e a Lamborghini estava no início da lista, por isso trocamos os dois carros de lugar.

Quem ficou na casinha 1? Foi o Jipe. Agora na casinha 1, irei trocar o Jipe com o Fusca, porque buscamos o menor de todos, a partir da posição atual. Colocamos o Fusca no início e movemos o Jipe para o fim.

Passamos para a casinha 2, onde está a Lamborghini. Vamos trocá-la de posição com o Smart, que é o carro mais próximo, com o menor preço.

Na casinha 3, eu tenho a Lamborghini. Iremos trocá-la de lugar com o Jipe. Após movê-los, sobra apenas a casinha 4. Esta não é preciso revisar, porque já sabemos qual elemento é o maior de todos. O Lamborghini é o produto mais caro.

Em seguida, imprimo os resultados. No total, fizemos as trocas das quatro casinhas: 0, 1, 2 e 3.

Em seguida veremos o processo visualmente, agora que o código foi implementado.

Simulando no papel o algoritmo de seleção de menores elementos para ordenação

Vamos revisar o nosso algoritmo de ordenação?

Nós fizemos um laço, que passava por cada casa com a variável `atual`, selecionando qual elemento era o menor a partir de determinada posição. Quem é o menor a partir da casinha 0? É a Brasília. Então, iremos trocar o produto atual da casinha 0 para o produto da casinha menor.

Passamos para o elemento 1. Qual é o menor a partir dele? É o Fusca. Vamos selecioná-lo e fazer a troca de posições. Com o laço, fomos resolvendo nosso problema com os outros produtos.

```
for(int atual = 0; atual < quantidadeDeElementos - 1; atual++) {

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

    produtos[atual] = produtoMenor;
```

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

Isto quer dizer que a nossa função de ordenação recebe não só o *array* com os elementos, mas também a quantidade de itens que temos dentro dele. No nosso caso, a quantidade é 5, o que significa que iremos trabalhar da casinha 0 até a casinha 3. O que sobrar na casinha 4 será o produto mais caro de todos.

Vamos trabalhar com outras variáveis também: `atual` e `menor`. Precisamos saber em que casinha estamos e qual é o menor a partir da posição atual. Eu criei duas variáveis auxiliares: `produtoAtual` e `produto menor`. Elas irão referenciar os produtos para que as trocas de posições sejam possíveis.

Em seguida, iremos simular os algoritmos: A variável `atual` começou com 0. A variável `menor`, a partir da posição 0, será 2.

Nosso `produtoAtual` estará apontando para a Lamborghini. Já o `produtoMenor` irá apontar para a Brasília. Quando invertemos as posições dos dois elementos da lista, a Brasília passa a ocupar o lugar da Lamborghini.

A casinha `1` será a nossa próxima variável `atual`. A partir desta posição, iremos verificar qual elemento é o `menor`. Identificamos que a partir da posição `1`, o `menor` é `4`. Vamos fazer o mesmo que fizemos anteriormente: trocaremos os elementos de lugar e substituiremos o Fusca pelo Jipe.

Passamos para a casinha 2. Qual é o menor elemento a partir da segunda posição? É o Smart. Vamos referenciar o `produtoAtual` e o `produtoMenor` e finalizamos invertendo os produtos de lugar. Vamos bem até aqui!

Agora estamos na posição 3. Fazemos novamente a pergunta: qual é o menor elemento a partir da minha posição `atual`? O `menor` é o `4`. Voltamos a referenciar o `produtoAtual` e o `produtoMenor` para depois, trocarmos a Lamborghini e o Jipe de posição.

Como a Lamborghini já ficou no fim da nossa lista, não precisaremos verificar a última posição. Se passarmos pela casinha 4, iremos sair do nosso laço. Então, já encontramos a solução do nosso problema e os produtos estão ordenados.

Selection Sort

O algoritmo que nós usamos para resolver o problema de selecionar o menor a partir de uma parte do nosso *array* e trocá-lo de posição é chamado de **ordenação por seleção** (*Selection Sort*). Ele seleciona o menor a partir do instante atual e permite que o reposionemos na lista.

O *Selection Sort* é capaz de resolver o nosso problema de ordenação. Ele passa por cada elemento e pergunta "Quem deve estar nesta posição? É esse". Então, coloca cada item em uma ordem.

Além disso, ele utiliza a nossa função para encontrar o `menor`. Isto significa que ao implementarmos primeiro a função de seleção do menor, simplicamos o nosso algoritmo de ordenação .

O algoritmo de ordenação, o *Selection Sort*, se baseia no algoritmo de seleção do menor.

Questionando a velocidade de um algoritmo

Nós vimos como é o processo de ordenação dos carros. Produtos, em geral, também já conseguimos ordenar. Por exemplo, se quiséssemos ordenar os resultados das provas do Enem ou de um concurso público, poderíamos utilizar o mesmo processo.

E se quiséssemos descobrir quem ficou em primeiro lugar no campeonato de futebol de acordo com a pontuação alcançada? Poderíamos seguir o mesmo processo de ordenação: selecionamos o primeiro (aquele que obteve menos pontos) e o último (o elemento que mais ganhou pontos). Com este tipo de ordenação podemos utilizar o algoritmo usado até agora, o processo de seleção, o **Selection Sort**.

Porém, tem alguma coisa errada aqui! Toda vez que tentamos ordenar uma lista de elementos como no exemplo dos cinco carros, precisamos comparar o atual com todos os outros itens seguintes para reposicioná-los. Nós sempre passamos por todos os elementos. Isto significa que quando fizermos o `for`, dentro dele, terei que verificar cada elemento. Parece trabalhoso... Se tivermos um `for` de 0 a 100, para cada um dos elementos teremos que criar outros `for`s que passem por dentro de cada um deles. Ficamos com a sensação de que é preciso fazer muita coisa. Vamos formalizar esta "sensação" e entender melhor o que queremos dizer com "muita coisa". Também entenderemos o que significa um algoritmo **lento**.

É correto termos a impressão de que podemos realizar o processo de ordenação mais rapidamente. E nesta tentativa de fazer as coisas de uma forma mais rápida e melhor, nós iremos buscar novas alternativas para o *Selection Sort*.

Vamos buscar uma nova maneira de ordenação dos nossos elementos e, a partir disto, poderemos comparar qual é o mais rápido ou o mais lento realmente.

Esse será o nosso próximo passo: encontrar uma maneira de ordenar elementos de uma forma diferente da que utilizamos até agora. Uma maneira que as pessoas também utilizem no cotidiano, por exemplo, quando jogam baralho...