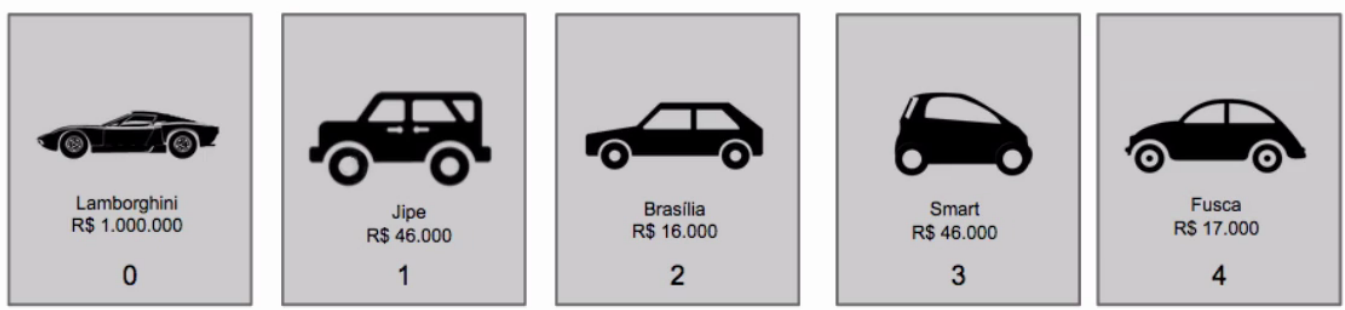
05

Algoritmos: entrada e saída

Algoritmos: entrada e saída

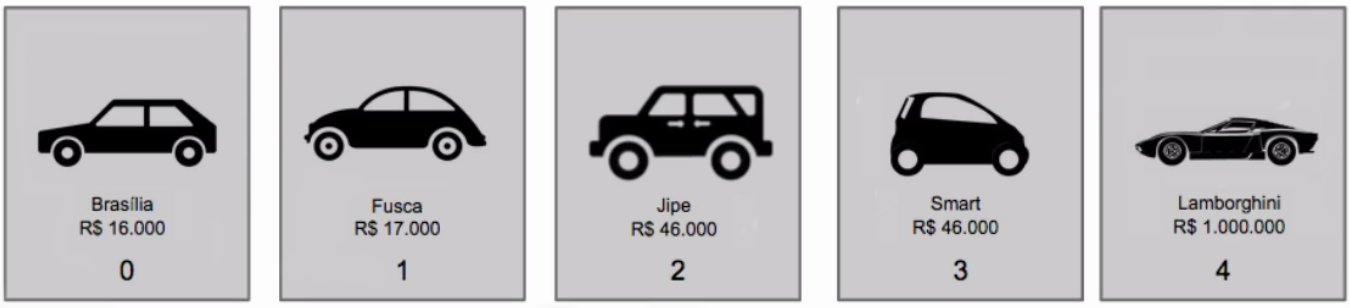
Vamos verificar de novo os três problemas que tentamos resolver. Resolvemos o problema de um *array* de vários produtos, com início e fim, e que começa do 0 e vai até o 4. Qual elemento é o menor deles? É o que está na casinha 0, 1, 2, 3 ou 4?



Considerando os diversos produtos, poderíamos perguntar: qual é o menor deles? Também poderíamos fazer outras perguntas, porém já temos o problema para ser resolvido.

Coloco um monte de elementos no meu programa e peço para que ele indique qual é o menor.

Analisamos também outro caso em que, dado um *array* com diversos elementos (alunos, políticos, times de futebol...), pedimos para que o programa coloque-os em ordem. Queremos que ele devolva os elementos de uma forma ordenada, do menor para o maior. Então, damos para o nosso programa, um *array* com vários produtos (elementos) e ele me dá uma lista no retorno, ou **saída**, com os itens na ordem correta. Este é outro problema resolvido por nós.



Por fim, nós resolvemos uma segunda variação do problema. Nós dissemos para o programa: "Se dermos esses produtos, por exemplo, várias cartas de baralho, você consegue devolver todas elas (ou todos esses elementos) de uma maneira ordenada? Porém, criando a ordem de uma maneira diferente desta vez..." E nós conseguimos nosso resultado de duas maneiras distintas. Na vida real, se pensarmos, cada pessoa ordena cartas de forma diferente : alguns usam *Selection Sort* outros usam *Insertion Sort*. Depende de cada um escolher qual processo lhe parece mais natural.

Também observamos que todos os problemas resolvidos até agora tinham uma entrada: um conjunto de dados, com várias informações. Além disso, o programa tinha que nos dar uma saída. Então, todo algoritmo terá uma entrada e uma saída. O importante é que o algoritmo resolva a questão.

Como ele irá resolver e achar o menor preço? Como ele irá ordenar o meu *array*? Essas perguntas não são tão relevantes. Para ser válido e estar correto, ele precisa dar a saída corretamente, para a entrada adequada. Assim o meu algoritmo irá funcionar bem.

O processo feito dentro do programa pode ter diferenças em desempenho, em consumo de memória e em várias coisas que iremos conversar adiante, quando analisarmos os algoritmos. Agora, o mais importante é entender que todo algoritmo tem uma entrada e uma saída. Para o algoritmo estar certo, toda entrada precisa resultar em uma saída correta.

Reduzindo um problema a outro

Nós resolvemos o problema da ordenação com duas soluções diferentes:

- selecionando quem deveria ficar em cada casinha
- encontrando a casinha onde deveríamos inserir um elemento.

As duas formas são utilizadas naturalmente pelas pessoas, sem que elas percebam que fizeram a escolha entre uma maneira e a outra, em situações diferentes. Porém, como são inventados algoritmos como estes? Será que existem apenas os dois já conhecidos e ninguém mais tenha inventado outra maneira? A verdade é que existem diversos outros. E o processo de inventar um algoritmo é um processo difícil. Além de criá-lo, precisamos provar que ele funciona e que ele é rápido... É uma tarefa trabalhosa.

Por isso, ao ser dado um problema, optamos muitas vezes por analisá-lo e tentamos identificar uma solução conhecida: "esse é um problema de ordenação.. então, vou usar aquele algoritmo que é muito bom para ordenar!"

Olhar com outros olhos o problema... Por exemplo, se quero descobrir quais foram os políticos mais votados em uma eleição, posso verificar qual é o algoritmo que irá identificar quais receberam um maior número de votos. Para isto, basta olhar para os elementos e ver a necessidade de uma ordenação. Nesse mesmo exemplo, se quero saber quais foram os quatro políticos com maior votação em uma eleição, basta ordená-los e em seguida selecionar os quatro primeiros elementos. Analisei a questão com outros outros olhos, ao invés de me limitar ao problema aparente "tenho que encontrar os quatro maiores". Pensamos de outra maneira: ao ordenar todos os candidatos, foi possível identificar os quatro com maior número de votos. Além de observar por outros ângulos, nós reduzimos o problema a outro, cuja solução já era conhecida. Sabíamos como solucionar uma ordenação, então apenas foi preciso selecionar os quatro maiores. Podemos também definir outros critérios: os cinco piores ou os três maiores. E os simplifico em uma ordenação...

Sempre que alguém nos pedir os 15 candidatos mais votados ou os três cursos com maior nota... Ordenaremos todos os cursos e pronto, estará acabado. Podemos identificar este tipo de informação. Sem grandes dificuldades, implementamos a ordenação, depois a executamos e informamos rapidamente quais são os três primeiros elementos. Fica ainda mais fácil se a quantidade de itens for pequena.

A grande sacada está em reduzir o problema. Por exemplo, se simplificarmos a questão em uma ordenação, já saberemos que é possível resolvê-lo com o *Selection Sort* - que foi reduzido a uma busca do menor.

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

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

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

Nós buscamos o menor dentro do algoritmo... Nós simplificamos a implementação do *Selection Sort* usando um algoritmo já conhecido. Dentro do *Selection Sort* usamos o algoritmo de encontrar o menor.

Então, quando reduzimos um problema para a utilização de um algoritmo que já conhecemos, nós simplificamos a maneira de resolver o nosso problema. Pode ser que ela seja muito performática, com a melhor memória... ou pode ser que não. Isto irá depender da análise do nosso algoritmo - adiante, iremos conversar sobre o quão rápido pode ser um algoritmo. Agora estamos interessados em identificar a melhor maneira de simplificar um problema, após observá-lo de diversos ângulos e perceber a melhor maneira para resolvê-lo.

A estratégia é observar um problema por diferentes ângulos e encontrar um aspecto em que a solução já é conhecida, descobrir um algoritmo que seja conhecido e que é capaz de resolvê-lo... Ou talvez, encontrar mais de um algoritmo, como foi o caso do *selection sort*: nós criamos um laço, usamos o `buscaMenor` diversas vezes e resolvemos o problema da ordenação. A sacada para solucionar os problemas do cotidiano é: observar um problema, encontrar uma nova forma de atacá-lo e depois, reduzi-lo a algoritmos já conhecidos.

Analiso um problema e penso: "para solucionar isso aqui, basta fazer um `for` e buscar o menor" ou "basta ordenar e encontrar os três maiores". A estratégia é reduzir o problema a coisas que já sabemos fazer.

Como analisar o desempenho de algoritmos?

Considerando que analisamos diversos algoritmos e já sabemos implementá-los... O que é preciso para compará-los? Como podemos dizer qual está bom e qual não está? Se alguém desejar seguir a carreira de Ciências da Computação e quiser criar novos algoritmos, como poderá afirmar que o seu algoritmo de ordenação é melhor do que outro já existente? Precisamos ter algum critério de comparação entre eles. Existem diversos critérios de comparação de algoritmos, por exemplo dois: **consumo de memória** e o **tempo para resolução do problema**.

Em relação ao primeiro critério: Quanto o algoritmo precisa processar o nosso computador? Quantas contas ele precisa fazer? Quantas operações o algoritmo precisa fazer para terminar e retornar o resultado? Dada a entrada, veremos a saída. Isto significa que poderemos comparar o quanto de memória os algoritmos consomem e com base nisto, determinar: "Esse é o melhor, porque consome menos memória."

O tipo mais famoso de comparação que podemos fazer é o de **desempenho de velocidade**, que nos informa o quão rápido pode ser um algoritmo. Porém, vamos com calma... Como iremos medir a velocidade? Em segundos, em minutos ou em horas?

Então, qual unidade iremos usar? O computador é capaz de fazer contas, por isso iremos medir a velocidade a partir da quantidade de operações que o computador precisa fazer para encontrar a resolução de um algoritmo. Se precisarmos fazer 10 ou 9 operações, não faz muita diferença na velocidade. No entanto, se a variação for entre 10 e 1 bilhão de operações, a diferença será enorme!

É desta forma que iremos avaliar o desempenho dos dois algoritmos. Analisando dessa forma: "nesse precisamos realizar poucas operações, já nesse precisamos fazer inúmeras..." Nós queremos descobrir como comparar diversos algoritmos. Temos três aqui para analisarmos:

1. **buscaMenor**
2. **selectionSort**
3. **insertionSort**

Vamos observá-los:

1. **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;
}
```

2. **selectionSort**:

```
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);
 }
}
```

3. **insertionSort**:

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

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

Qual deles é o mais rápido ou o mais devagar? Qual é a sua opinião?

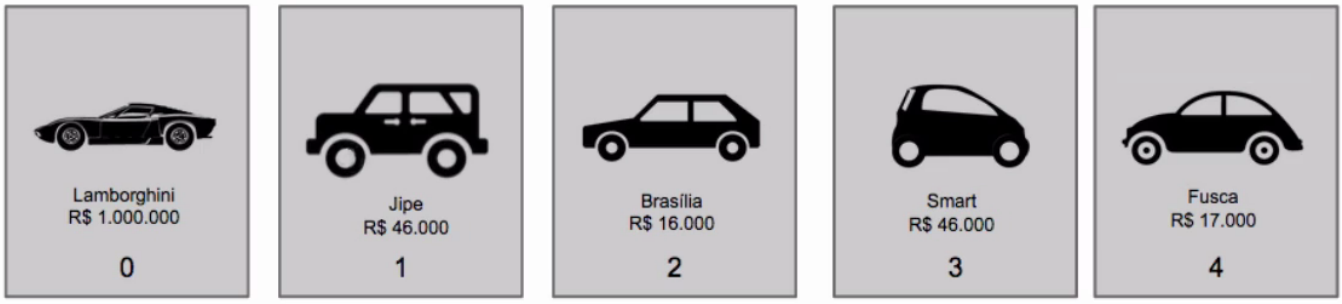
Cada um pode ser utilizado em diferente situaçõess. O `buscaMenor` serve para encontrarmos o menor (ou o maior) de todos.

Já o `insertionSort` e o `selectionSort` são algoritmos que ordenam o nosso *array*. Qual deles parece ser mais rápido e qual parece ser o mais devagar?

Já iremos verificar isso daqui a pouco...

Analisando o buscaMenor

Vamos começar analisando o algoritmo de `buscaMenor`. Ele recebeu um *array* com cinco elementos e fizemos a busca do `0` até o `4` inclusive.



O nosso algoritmo passará por cada um dos elementos. Trata-se de um `for` que passa por todos os elementos, desde o primeiro até o quinto. Nós faremos as operações do algoritmo para cada um dos elementos, ou seja, se temos cinco elementos, cinco operações. Se temos 100 elementos, por exemplo, teremos 100 operações. Será o mesmo se tenho 1.000.000 ou 5.000.000 de elementos... Basicamente, disto se trata o nosso algoritmo.

Temos um `for` . Dentro dele, faremos quantas operações? Apenas uma? Se fizermos uma operação, serão as cinco operações vezes um, o que resultará em cinco. Porém, se fizermos duas ações dentro do `for` , devido à quantidade de elementos, teremos 10 operações. Será o dobro...

Vamos verificar passo a passo:

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

O `for` passa pelo *array* inteiro. Se passar de 0 até 4, ele irá executar cinco vezes o código dentro do `for` , que é esse:

```
if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()){
    maisBarato = atual
}
```

Isto significa que dependendo da "sorte", para cada um dos elementos faremos apenas um `if` , o que resulta em cinco operações. Porém se dermos "azar", além do `if` , teremos que aplicar o `maisBarato = atual` para cada um dos cinco itens. Serão 10 operações. Então, o número total ficará entre 5 e 10... Isto significa que se tivermos 100 elementos, teremos entre 100 e 200 operações. Se tivermos 500 elementos, ficaremos entre 500 e 1000 operações. O mesmo ocorrerá se forem 1.000.000 de operações... Portanto, se tivermos *n* elementos, a quantidade de operações estará entre *n* e *2n*.

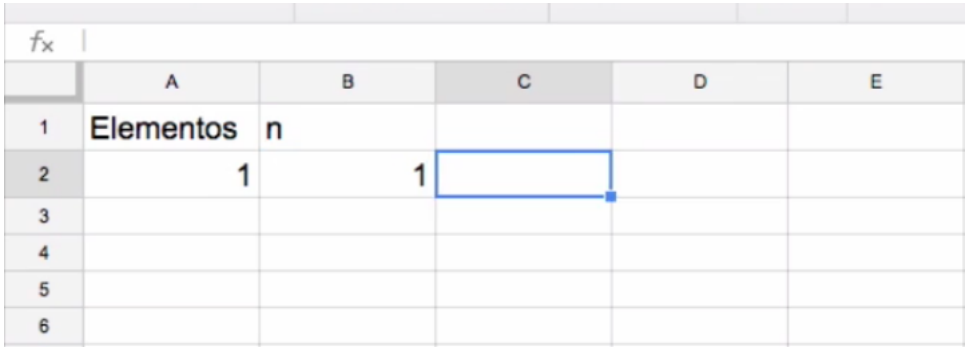
Em breve, poderemos comparar o quão bom é o nosso algoritmo de *n* ou *2n* elementos e os demais algoritmos.

Mas primeiro gostaria que você desenhasse um gráfico que mostre a quantidade de operações que serão feitas à medida que o número de elementos aumente. Por exemplo, se temos 100 elementos, quantas operações teremos? Quantas operações teremos se tivermos 1000 ou 100.000 elementos? Como será o gráfico?

Tabela de operações por tipo de algoritmo

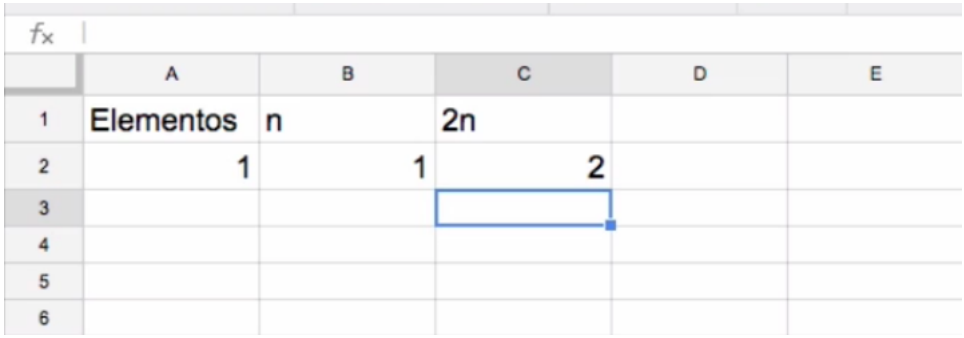
Observamos que nosso algoritmo para buscar o menor elemento rodará cerca de *n* operações, ou até 2 operações por elemento (*2n*). Vamos utilizar alguns números para verificar como isto ficaria?

Para o primeiro caso, se temos 1 elemento, qual será o valor de *n*? Exatamente 1.



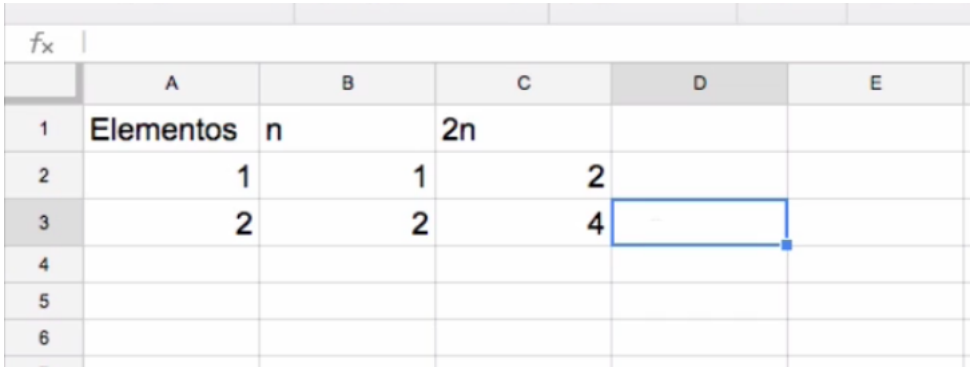
	A	B	C	D	E
1	Elementos	n			
2	1	1			
3					
4					
5					
6					

E se fosse *2n* (duas operações por elemento)? O resultado seria 2 operações.



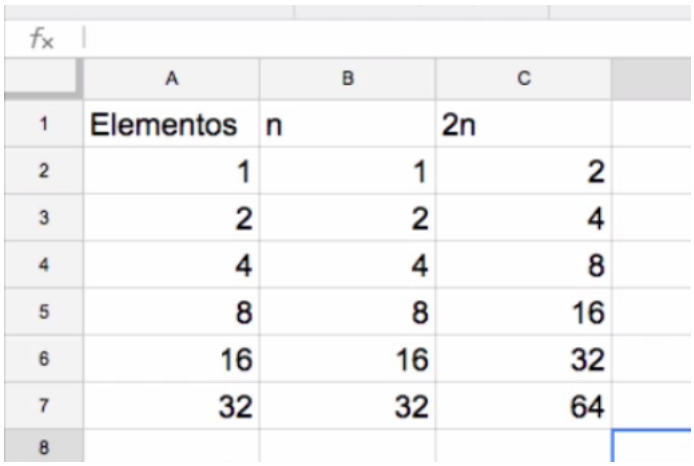
	A	B	C	D	E
1	Elementos	n	2n		
2	1	1	2		
3					
4					
5					
6					

E se tivéssemos 2 elementos, o dobro da quantidade anterior, como iríamos preencher as outras colunas? Os outros resultados também seriam o dobro...



	A	B	C	D	E
1	Elementos	n	2n		
2	1	1	2		
3	2	2	4		
4					
5					
6					

Isto significa que à medida que dobramos o número de elementos, o número de operações também irá dobrar:



	A	B	C
1	Elementos	n	2n
2	1	1	2
3	2	2	4
4	4	4	8
5	8	8	16
6	16	16	32
7	32	32	64
8			

Quando cresce o número de elementos, o número de operações irá crescer proporcionalmente. Se dobrarmos o número de elementos, o número de operações também irá dobrar.

Gráfico de um algoritmo linear

Vamos inserir os dados da tabela em um gráfico para vermos como será aproximadamente o nosso número de operações? À medida que o nosso *array* for crescendo e quisermos por exemplo, encontrar o menor preço de 8192 produtos, conseguiremos ver no gráfico, quantas operações o algoritmo terá que fazer. Vamos visualizar no gráfico o crescimento do algoritmo corretamente?

Então, selecionaremos todos os elementos e clicamos em "Insert" e "Chart" e o programa abrirá o "Chart Editor". Vou definir qual será a coluna e a linha que serão o cabeçalho e o rodapé.

Nosso gráfico terá linhas simples...

Em seguida pedimos para inserir o gráfico:

Vamos incluir novos títulos. Em baixo, iremos alterar o título do eixo horizontal (*Horizontal axis title*) por **Elementos** (referente ao número de elementos que cresce aos poucos). Na lateral, iremos alterar o eixo vertical (*Left vertical axis title*) por **Operações** (referente ao número de operações). Isto significa que, se o número de elementos cresce, o número de operações também irá crescer. Este é o algoritmo. E como ele cresce? Vamos analisar o *n*, a linha azul do gráfico...

À medida que o número de elementos cresce, o número de operações também aumentará. *n* é linear, por isso o número de operações seguirá uma linha. O mesmo acontecerá com *2n*, que terá o dobro do número de operações, mas seguirá uma linha. Naturalmente, *2n* será mais lento e terá uma demora duas vezes maior do que *n*, que é mais rápido. No entanto, ambos crescem como uma linha. Isto significa que à medida que dobramos o número de elementos, dobramos também o número de operações. Se triplicarmos o número de elementos, iremos triplicar também o número de operações. O número de operações sempre terá um aumento proporcional e linear.

Este é o nosso algoritmo que buscar o menor. Por isso, o nosso gráfico receberá o título de **Busca menor**. Ele irá buscar o menor elemento, que está situado entre as linhas *n* e *2n* do gráfico.

Sabemos que é um algoritmo válido. Agora iremos analisar outros algoritmos para poder compará-los entre si.

Analisando o *Selection Sort*

Nós vimos o algoritmo de menor elemento. Neste algoritmo, para executá-lo, percebemos que tivemos que passar por todos os elementos. Para cada um deles (para cada um dos `n` elementos), executávamos um `if` e um código que estava dentro dele. Nós executávamos entre 1 e 2 operações para cada um dos `n` elementos. Isto é, nós executávamos *n* ou *2n* operações. Podemos ter uma ideia de quantas operações este algoritmo precisa. Vamos analisar outro algoritmo: o `TestaOrdenacao`.

Nós já havíamos analisado o `selectionSort`. Ele era baseado no `buscaMenor`. Ele passava por cada uma das casinhas e buscava o menor a partir dali. Vamos analisar uma simulação dele?

Na hora de executarmos o nosso algoritmo, o que nós fazíamos?

Executávamos o `atual` indo de `0 até 4` - um `for` para cada elemento. Se tínhamos cinco elementos, tínhamos um `for` para cinco. Se tínhamos 100 elementos, tínhamos um `for` de 100. Se tínhamos 1.000.000 de elementos, tínhamos um `for` de 1.000.000. Se tínhamos `n` elementos, o nosso `for` precisava passar por cada um dos `n` elementos. Sabemos que o nosso algoritmo é um `for` que vai do `0 até 4`.

Porém, a partir da **posição 0**, mandávamos buscar o menor desde a **posição 1**. O mesmo acontecia quando buscávamos a partir da **posição 2**, e a busca pelo menor iniciava da **posição 3**. O processo era feito até chegarmos no quarto elemento.

Nós já conhecemos o `buscaMenor` e sabemos o quão lento ou rápido ele é. O `buscaMenor` está entre *n* e *2n* operações, para cada um dos elementos. Porém, se o que estamos fazendo é executar um `for` que passe por cada um dos *n* elementos, o que faremos é executar um outro algoritmo que realizará *n* ou *2n** operações. Quanto é o total? O resultado será *aproximadamente* n^2 *ou* $2n^2$*.

Vamos analisar o código? Veremos o nosso `for` de `n` elementos:

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

E, dentro dele, executaremos o algoritmo:

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

Isso significa que nós executamos *n* vezes o algoritmo que demora *n* ou *2n* operações. Isto significa que dá n^2 ou $2n^2$ operações.

Temos que executar também uma troca para cada um deles, que irá, igualmente, demorar algumas operações. Porém, vamos descartá-la para ver como fica estas operações que já estamos trabalhando.

Iremos analisar apenas como será com n^2 e $2n^2$ que é o tempo de execução do algoritmo de ordenação e compará-lo com o `TestaOrdenacao` que demora *n* ou *2n*.

Vamos analisar o gráfico destes dois algoritmos? Primeiro, gostaria que você tentasse rascunhar o nosso algoritmo que demora *n* e depois, o que irá demorar n^2 ou $2n^2$. Tente rascunhá-lo e em seguida veremos como ficará.

A tabela de operações de um algoritmo quadrático

Chegou a hora de desenharmos um gráfico para o número de operações que teremos, de acordo com o algoritmo `selectionSort`. Vamos analisá-lo?

Iremos criar uma nova planilha e a primeira coluna iremos chamar de **Elementos**. Já vimos que no `selectionSort` o número de operações que teremos será n^2. Iremos preencher a segunda coluna da tabela com *n^2*. Se tivermos um elemento, n^2 será igual a 1. Sabemos que o nosso número de operações seria entre n^2 e $2n^2$, então vamos preencher a terceira coluna com *2n^2*. Significa ele poderá ter 2 vezes o número de elementos.

O que acontecerá se tivermos 2 elementos? Neste caso iremos executar entre 4 e 8 operações.

E se tivermos o dobro do número de elementos? Se tivermos 4 elementos, teremos mais do que o dobro de operações...

Se tivermos 8 elementos, também teremos mais do que o dobro de operações. Observe que à medida que dobramos o número de elementos, o número de operações cresce explode no gráfico, afinal ele será elevado ao quadrado.

Seguimos dobrando o número de elementos e observando o que acontece com as operações.

O que percebemos é o crescimento gigantesco dos resultados...

O gráfico de um algoritmo quadrático

Chegou o momento de criarmos o gráfico do algoritmo. Clicamos em **Insert** e depois **Chart**. Em seguida, selecionaremos o rodapé e o cabeçalho. Escolhemos o gráfico de linha e o inserimos em seguida.

Vamos alterar o título do gráfico para **Selection Sort**. Abaixo, denominaremos o eixo horizontal como **Elementos** e o vertical como **Operações**. À medida que o número de elementos cresce, o número de operações também aumentará. Porém, observe o tanto que o último cresce! Quando tivermos 4000 elementos, teremos cerca de 20.000.000 operações sendo executadas. Quando o número de elementos alcançar 8192, teremos 134.217.728 operações. Observe que se tivermos, por exemplo, que colocar 8192 políticos em ordem, o computador terá que fazer 134.217.728 operações matemáticas. Parece um número muito elevado! Será que de fato são necessárias tantas operações? Para responder tal pergunta, teremos que comparar o algoritmo quadrático com o linear e assim concluir como funciona o crescimento de cada um.

Vamos comparar os algoritmos...