04

A comparação do desempenho de um algoritmo linear com um quadrático

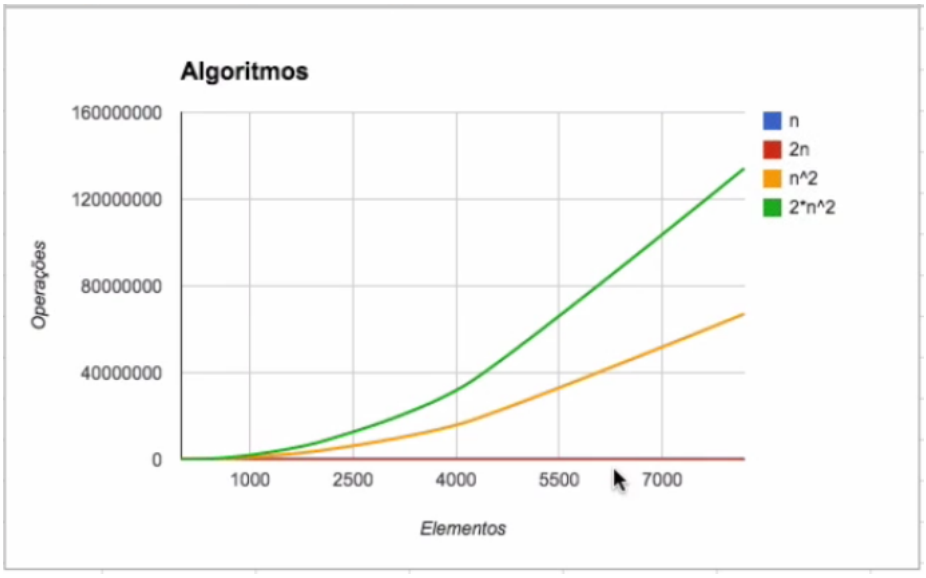
A comparação do desempenho de um algoritmo linear com um quadrático

Vamos comparar os algoritmos que nós analisamos... Iremos criar uma nova aba e inserir os elementos do nosso `buscaMenor`, incluindo `n` e `2n`, `n^2` (n^2) e `2*n^2` ($2n^2$).

A	B	C	D	E
Elementos	n	2n	n^2	2*n^2

Após preenchermos todas as células da tabela, teremos o número de operações e já poderemos gerar o gráfico. Selecionaremos todos os dados e, em seguida, iremos clicar em **Insert** e **Chart**. Vou selecionar a coluna e a linha que serão o cabeçalho e o rodapé do nosso gráfico, que será inserido com linhas.

Agora conseguiremos visualizar graficamente o número de operações que estão acontecendo.

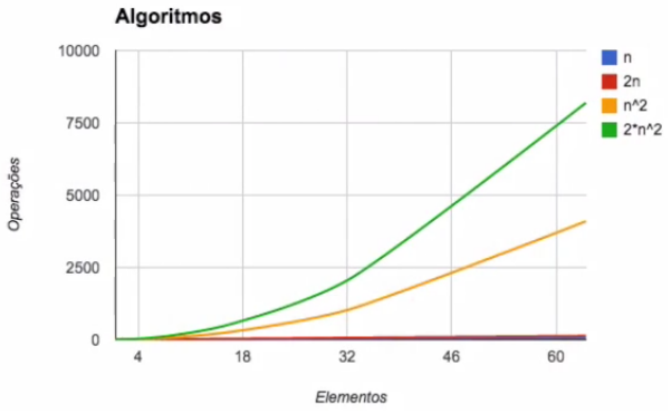


Faremos alterações nos títulos da imagem identificando o eixo inferior com o título **Elementos** e eixo vertical com **Operações**. Nosso gráfico receberá o nome **Algoritmos**.

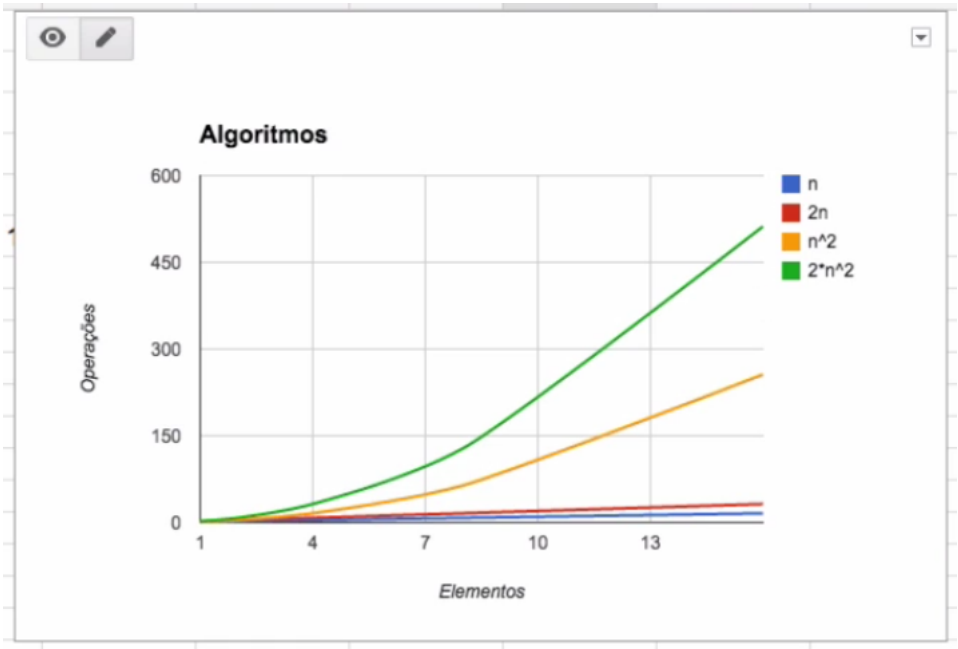
No entanto, observe que tem algo estranho no gráfico... Nós não conseguimos visualizar a linha azul (*n*), nem a linha vermelha (*2n*), porque as duas são muito pequenas e permanecem próximas do eixo inferior. Como os valores das linhas verde e amarela são muito superiores, mal conseguimos ver as outras linhas. O algoritmo quadrático cresce tão rápido e vai tão alto, que quando fazemos a comparação não conseguimos localizar os lineares.

Por isso, iremos excluir alguns elementos da tabela para conseguirmos visualizar a duas linhas. Ficaremos com 64 elementos... Quase é possível visualizar todas as linhas no gráfico.

A	B	C	D	E
Elementos	n	2n	n^2	2*n^2
1	1	2	1	2
2	2	4	4	8
4	4	8	16	32
8	8	16	64	128
16	16	32	256	512
32	32	64	1024	2048
64	64	128	4096	8192



Precisaremos excluir mais elementos da tabela para enxergar as duas linhas que faltam. Ficaremos com 16 elementos.



Agora conseguimos visualizar um pouco melhor... As linhas quadráticas amarela e verde logo se distanciam das linhas do algoritmo linear (azul e vermelha) e crescem absurdamente. No entanto, analisando a velocidade dos dois algoritmos, o quadrático é muito lento em relação ao linear. Quando temos apenas 16 elementos, se usarmos um algoritmo linear faremos entre 16 e 32 contas. Porém, se estivermos usando um algoritmo que é quadrático faremos enter 256 e 512 operações. A diferença entre os dois algoritmos é absurda.

Observe também que a diferença no resultado entre *n* e *2n* não parece ser tão grande. O mesmo acontece com os valores do quadrático... Não há tanta diferença entre as duas linhas. Provavelmente se incluíssemos uma terceira opção, um algoritmo cúbico por exemplo, o número de operações seria ainda maior e a diferença entre as duas linhas não seria tão expressiva. Naturalmente, se dobrarmos os valores, significa que ele será duas vezes pior. No entanto, a diferença entre as linhas de um mesmo algoritmo não é a mesma quando modificamos de linear para quadrático. Isto significa que quando analisamos de forma geral os algoritmos, nós não nos preocupamos tanto com o fator de multiplicação, como temos em *2n* ou $2n^2$ ou

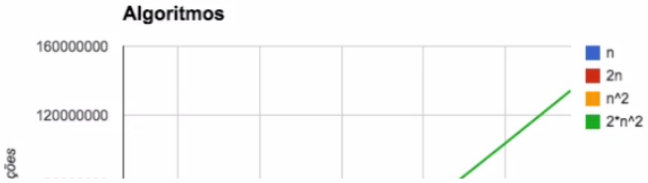
mesmo, *2n-5*. O resultado será irrelevante em uma comparação geral. O interesse não está no fator de multiplicação, mas sim, na potência.

Observe que a **potência 1** linear cresce como uma linha, enquanto a **potência 2** quadrática deslancha no nosso gráfico e vai embora. As **potência 3 e 4** quadrática irão ainda mais adiante. É na **potência** que estamos interessados. Por isso, se o nosso foco for no desempenho dos algoritmos, iremos preferir os lineares (e só depois os quadráticos e outros adiante). Inclusive, podem existir algoritmos ainda mais rápidos (ou lentos) do que os comparados na nossa análise. Podem existir novos algoritmos para solucionar outros problemas!

Comparando diversos desempenhos

Ao compararmos dois algoritmos em relação ao desempenho de velocidade, iremos ignorar o **fator de multiplicação 2** e analisaremos apenas o *n* e o quadrático.

	A	B	C	D	E
1	Elementos	n	2n	n^2	2*n^2
2	1	1	2	1	2
3	2	2	4	4	8
4	4	4	8	16	32
5	8	8	16	64	128
6	16	16	32	256	512
7	32	32	64	1024	2048



Nós observamos que quando analisamos um número elevado de elementos, as linhas do algoritmo quadrático deslanchavam. Entre *n* e *2n* não existe diferença. Qual é a conclusão que podemos tirar do gráfico quando temos a intenção de encontrar o menor produto de uma lista? Com certeza, ordenar a lista não é uma boa ideia. Para colocar os elementos em ordem apenas com o objetivo de encontrar o menor de todos, é preferível usar o algoritmo `buscaMenor`, porque este é linear.

Porém, se estamos interessados em saber quais são os três elementos mais baratos, os cinco mais caros ou o produto localizado na metade da lista, aparentemente é mais simples ordenar a lista. Inclusive, porque o `buscaMenor` não resolveria o problema. Essa é a sacada: o `buscaMenor` só pode ser utilizado para encontrarmos o **menor** e o **maior** elemento. No entanto, quando ordenamos a lista, podemos resolver diversos problemas. Então, primeiro iremos verificar: "para resolver o problema, usar o `buscaMenor` será útil? Ou ordenar é a melhor opção? Qual algoritmo resolverá mais rápido?" Caso os dois algoritmos resolvam o nosso problema, a melhor escolha será utilizar o `buscaMenor`, por este ser linear e crescer mais devagar em comparação com o quadrático. Se temos poucos elementos, por exemplo 4... Fazer 8 ou 32 operações não faz diferença para o computador, como veremos em seguida.

Mas a sacada principal, quando comparamos dois algoritmos, analisamos se ele é linear ou quadrático - ou outras classificações. Ele será linear quando tiver um `for` que passe por todos os elementos. No entanto, quando tivermos um `for` com outro laço dentro, que também passe por todos elementos, teremos a quantidade de elementos multiplicada pela quantidade de elementos. Neste caso, o algoritmo será quadrático, ou seja, sendo dividido ou multiplicado por 2.

Vamos compará-los agora no computador, e ver quanto tempo demoraria para resolver os nossos problemas.

Comparando o desempenho do algoritmo em um computador

Sabemos que quando temos um algoritmo linear, ele irá crescer como uma linha no gráfico. Se temos um algoritmo quadrático ele terá um crescimento de forma quadrática. Mas e o computador? Quantas operações ele executará?

No mundo real, nós não dizemos "daqui a cinco operações estarei lá... Me encontre em cinquenta operações." Nós usamos segundos, minutos, etc...

A	B	C	D	E
Elementos	n	2n	n^2	2*n^2
1	1	2	1	2
2	2	4	4	8
4	4	8	16	32
8	8	16	64	128
16	16	32	256	512
32	32	64	1024	2048
64	64	128	4096	8192
128	128	256	16384	32768
256	256	512	65536	131072
512	512	1024	262144	524288

Então, vamos imaginar um computador que execute 8 mil operações por segundo. Iremos criar uma nova coluna na tabela que irá se chamar *Comp8000 n*, referente ao algoritmo linear de *n*. Ele irá executar a quantidade de operações de *n* dividido por 8000. Então, o programa irá demorar 0,000125 segundos para executar as contas. Será uma fração muito pequena de segundo e o processo será muito rápido.

O computador só irá demorar 1 segundo, quando tivermos cerca de 8192 operações...

Agora se tivermos um algoritmo quadrático, como o *Comp8000 n^2* irá reagir? Iremos dividir *n^2* por 8000. Quando executarmos apenas uma operação, o tempo será o mesmo utilizado por outro algoritmo (0,000125 segundos). À medida que o número de operações aumenta, a demora também será um pouco maior. Inicialmente, como o tempo é inferior a 1 segundo, a diferença é irrelevante. Porém, logo o computador irá demorar 32 segundos para executar o algoritmo e, para fazer o maior número de operações da nossa tabela, precisará de 8388,608 segundos para executá-lo.

Estimando o valor por hora, a demora será de 2h33. Enquanto o *Comp8000 n* leva 1 segundo para executar um algoritmo linear, o *Comp8000 n^2* irá demorar mais de 2 horas para executar um algoritmo quadrático. É por isso que consideramos que multiplicar *n* por 2, não é o grande critério quando fazemos uma análise geral do algoritmo. Com 8 mil elementos, a

diferença entre *n* e *2n* irá saltar de 1 para 2 segundos. No entanto, estamos mais interessados na diferença entre 1 segundo e 2 horas...

Como o computador executa 8 mil operações por segundo, poderíamos argumentar que para o usuário final a diferença de frações de segundo é irrelevante... Logo com 32 ou 64 elementos, a diferença entre os dois algoritmos não será grande, já que ambos terão uma demora de menos de 1 segundo. Podemos implementar qualquer um, o que for da preferência de cada um. Porém, entre a demora de 32 segundos e 0,06 segundo, teremos que analisar qual será o mais útil.

O algoritmo linear irá crescer de forma linear, enquanto o algoritmo quadrático terá um crescimento gigantesco. Logo, serão necessárias 2 horas para executar o algoritmo, enquanto seria possível demorar 1 segundo, caso o algoritmo fosse linear.

Questionando quando usar um algoritmo e outro

Observe que a análise do nosso algoritmo não está focada se os valores são pequenos ou grandes. Estamos interessados no número de operações que o computador faz a longo prazo, quando as linhas deslancham no gráfico, para analisarmos se o crescimento será linear, quadrático ou outras possibilidades. Quando fazemos uma análise que leva tais aspectos em consideração a chamamos de **assintótica**.

Ao fazermos uma **análise assintótica** do algoritmo, identificamos se ele é linear ou quadrático. É possível que para valores pequenos, seja irrelevante a diferença entre um e outro. Porém, quando temos um aumento nos valores, as diferenças se tornarão mais expressivas. Por exemplo, no **Comp 800 n^2**, o tempo para execução de operações passa das 2 horas.

Mas, poderíamos pensar que a demora poderia ser menor com um computador mais rápido. Mesmo que o computador utilizado seja duas vezes mais potente, ainda precisaríamos de 1 hora para encontrarmos a solução do problema. Se o computador fosse quatro vezes mais rápido (e mais caro também), teríamos ainda uma demora de meia hora, enquanto seria possível executar em 1 segundo a mesma quantidade de operações porém, com outro algoritmo. Por maior que seja a rapidez do computador, o desempenho de um algoritmo linear para outro continuará sendo imensa. O que precisamos descobrir é se o algoritmo linear irá resolver o nosso problema.

Observe que esta é a gravidade dos algoritmos quadráticos ou de outros ainda piores e utilizarmso um algoritmo qualquer, se pensarmos em qual seria a melhor solução. Esta é grande vantagem de trabalharmos com algoritmos mais espertos e mais

rápidos. Ele nos permite trabalhar com o mesmo computador, mantendo os gastos baixos. Você consegue usar o seu computador de forma mais rápida e eficiente. Se fosse preciso usar o método quadrático o tempo inteiro, seria preciso investir uma grande quantia de dinheiro em computadores absurdamente potentes. Imagine se para resolver um algoritmo quadrático, tivessemos que comprar um computador quadráticamente mais potente...

Como temos uma limitação na potência dos computadores, é muito importante que os algoritmos sejam mais rápidos e consumam menos de processamento do nosso computador. Por isso, que sempre iremos favorecer algoritmos que seguirão uma tendência linear do que a quadrática.

Nós vimos como se comporta um algoritmo `buscaMenor` (linear) e `Ordenacao` (quadrático). Então, paramos para pensar: "se vamos apenas buscar o menor, não faz sentido ordenarmos nossos elementos." No entanto, se o objetivo for identificar diversos pedaços de uma lista, por exemplo, os três menores ou os dois maiores, talvez seja mais simples ordenar todos os itens e depois fazer inúmeras pesquisas com base nos elementos já ordenados. Depende da análise de qual método vale mais a pena. Porém, se o nosso objetivo for apenas encontrar o menor, certamente o algoritmo linear será a melhor opção.

Agora, também trabalhamos com outro algoritmo: o `insertionSort` . Falamos dele justo quando percebemos que algo não funcionava bem no `selectionSort` , que parecia ficar lento quando adicionávamos muitos elementos.

O `insertionSort` é linear ou quadrático? Qual resposta iremos obter, se fizermos uma análise similar a que fizemos até agora?

Analisando o insertion sort

Vamos analisar o algoritmo `insertionSort` . Como ele funciona mesmo? Nós criamos um laço que varre do começo até o fim de uma lista, da esquerda para direita. Ele passa por todos os elementos *n* vezes - porque temos *n* elementos. Iremos voltar ao exemplo das cartas de baralho, que tinham cinco itens. O que era feito com cada um delas?

Quando precisávamos comparar o primeiro elemento, a carta 7, com todos os outros itens posicionados antes, percebíamos que não havia nada. Então, já começamos a análise a partir do elemento na segunda posição, a carta 4. Então, comparávamos a carta com todos as outras atrás e as reposicionávamos.

Depois, seguíamos para a carta 10 e a comparávamos com todas as posicionadas antes dela. Não precisávamos mudar a ordem.

Seguíamos a análise para a carta 2 e a comparávamos com todas atrás dela.

Trocávamos para a carta de posição até identificar qual era a mais adequada.

A nossa análise chegava até a carta 8 e novamente a comparávamos com as que estavam posicionadas atrás.

Até que todas estivessem na posição correta.

Observe que o nosso laço passou por todos itens *n* vezes e dentro dele, criamos um `for` que varreu todos os elementos da esquerda. Ou seja, um laço precisava passar por todos os elementos até a direita, enquanto o outro, no pior caso, passava por todos os itens posicionados à esquerda.

Ainda que tivéssemos uma variação nas operações, não faria muita diferença para o algoritmo - mesmo que fosse uma divisão, multiplicação, ou a soma de uma constante. A diferença só existe quando multiplicamos ou dividimos por *n*. É que estamos fazendo.

Vamos para o código e analisaremos o algoritmo `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 > 0 && produtos[analise].getPreco() < produtos[analise -1].getPreco())
            troca(produtos, analise, analise -1);
            analise--;
        }
```

O algoritmo passa por todos os elementos, exceto o primeiro. Então, poderíamos deduzir que seria preciso trabalhar com *n-1*. Nós já comentamos anteriormente que quando fazemos uma análise de longo prazo, se vamos subtrair ou não é irrelevante. A **análise assintótica** irá ignorar esta característica. Isto significa que vamos passar pelos *n* elementos.

Como é especificado no `while` , para cada um dos *n* elementos, o laço irá varrer para a esquerda.

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

Isto significa que ele irá repetir o processo enquanto analisar do elemento 1 até 5. Logo, teremos *n* multiplicado por *n*. Isto significa que, assim como o `selectionSort` , o algoritmo `insertionSort` também é quadrático. Ele também terá um desempenho terrível, quando consideramos o tempo de execução das operações. Mas será que não existem outros algoritmos de ordenação e de busca, que podem ser até mais rápidos do que os quadráticos? Ou será que o mundo está fadado a fazer as coisas com o mesmo método? Sempre precisaremos de computadores absurdamente potentes para fazer a ordenação de um número elevado de elementos? Devem existir outros algoritmos... Veremos mais adiante quais são eles.

O importante é sabermos como analisar estes algoritmos, para poder compará-los. Até o momento, aprendemos que: temos três algoritmos, um que busca o menor elemento e dois que fazem ordenação. Aprendemos a analisá-los e a identificar o crescimento de cada um deles, para afirmar "esse cresce bem e este outro não será útil se tivermos muitos elementos. Vou ter que procurar um novo algoritmo."

Além da maneira linear e quadrática, existem outras maneiras que um algoritmo podem crescer. Será que existem outras maneiras?