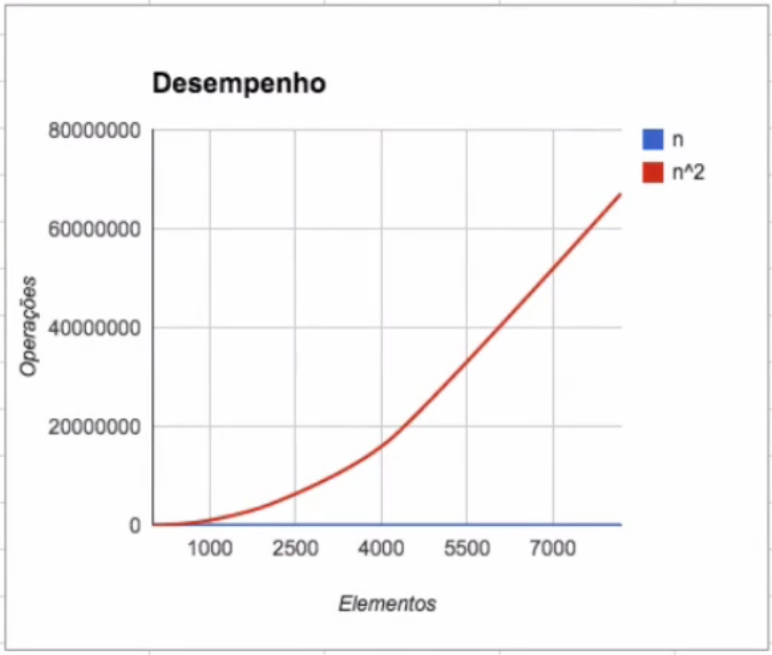
03

Algoritmos cúbicos

Algoritmos que rodam com o tempo constante

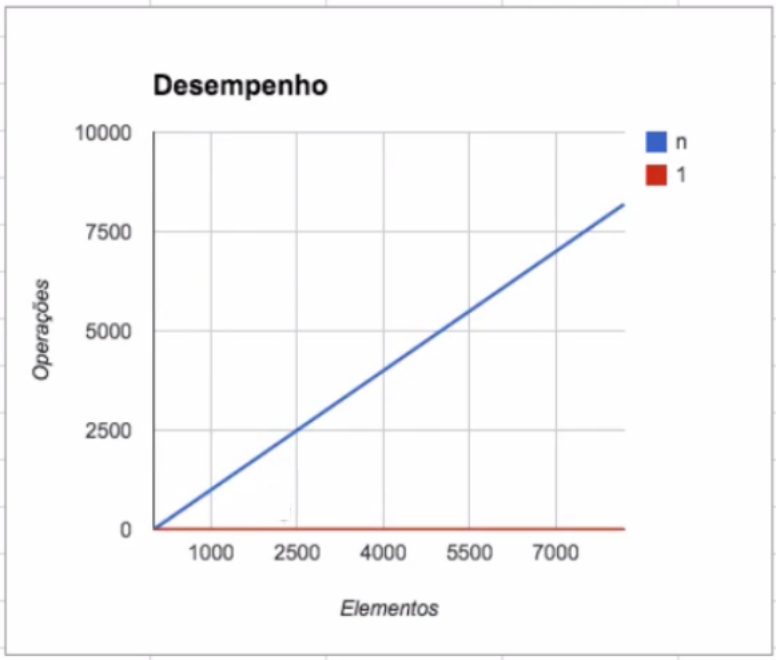
Nós observamos que um algoritmo linear comparado com um quadrático, irá performar muito melhor. Enquanto a linha do quadrático irá crescer rapidamente no gráfico, a linear irá permanecer bem próxima ao eixo inferior. Quase nem a percebemos, mesmo quando o número de elementos passa de 8 mil elementos.

Elementos	n	n^2
1	1	1
2	2	4
4	4	16
8	8	64
16	16	256
32	32	1024
64	64	4096
128	128	16384
256	256	65536
512	512	262144
1024	1024	1048576
2048	2048	4194304
4096	4096	16777216
8192	8192	67108864



Porém, será que existe algum algoritmo mais rápido do que o linear? Pode haver um que seja realmente rápido... Por exemplo, se queremos descobrir qual elemento está posicionado no meio de um *array*? Temos uma lista com cinco casas, o elemento buscado será o que estiver na posição do meio. Praticamente, executamos uma operação. Será muito rápido! Temos que descobrir apenas qual elemento está no meio, porém não temos que fazer inúmeros cálculos. O mesmo se quisermos descobrir qual é o elemento que está no começo ou no fim do *array*. Estas operações apenas pedem que "informe quem está aqui ou ali". Dizemos que tais operações têm **tempo constante**. Elas dependem de um número constante e por isso, são muito rápidas. Imagine que, se o tempo de demora for 1 no início, também será 1 quando chegar ao fim.

Elementos	n	1
1	1	1
2	2	1
4	4	1
8	8	1
16	16	1
32	32	1
64	64	1
128	128	1
256	256	1
512	512	1
1024	1024	1
2048	2048	1
4096	4096	1
8192	8192	1



Mesmo que o algoritmo linear (linha azul) cresça como uma linha no gráfico, observe como o algoritmo com o tempo constante (linha vermelha) permanecerá reto. Então, o algoritmo constante (que é *1*), será mais rápido do que um algoritmo linear (que é *n*).

Os algoritmos constantes trabalhados até agora podem identificar:

- O elemento do meio
- O primeiro elemento
- O último elemento

Parecem questões menores, porém, iremos descobrir utilidades para elas. Por exemplo, já vimos que se temos um *array* ordenado, quanto tempo vamos precisar para indicar o carro que tem o preço do meio? Se os elementos estão ordenados, basta selecionar o item no meio da lista. Isto é, se o *array* estiver ordenado, o algoritmo para selecionar o item do meio será constante. Por exemplo, se precisarmos selecionar os três elementos mais baratos ou os cinco mais caros, também iremos usar um algoritmo constante . Caso o *array* esteja ordenado, todas estas perguntas podem ser respondidas rapidamente. No entanto, precisaremos usar também bons algoritmos de ordenação.

Na comparação entre algoritmos, os constantes são mais rápidos do que os lineares.

Algoritmos cúbicos

Depois do quadrático, o que poderemos ter? Está claro que se aumentarmos o número da potência, os valores ficarão ainda maiores.

Vamos comparar o algoritmo quadrático (*n^2*) com o cúbico (*n^3*)? Os valores da terceira coluna serão o resultado de n^3.

Elementos	n^2	n^3
1	1	1
2	4	8
4	16	64
8	64	512
16	256	4096
32	1024	32768
64	4096	262144
128	16384	2097152
256	65536	16777216
512	262144	134217728

Observe que a linha vermelha deslancha no gráfico!

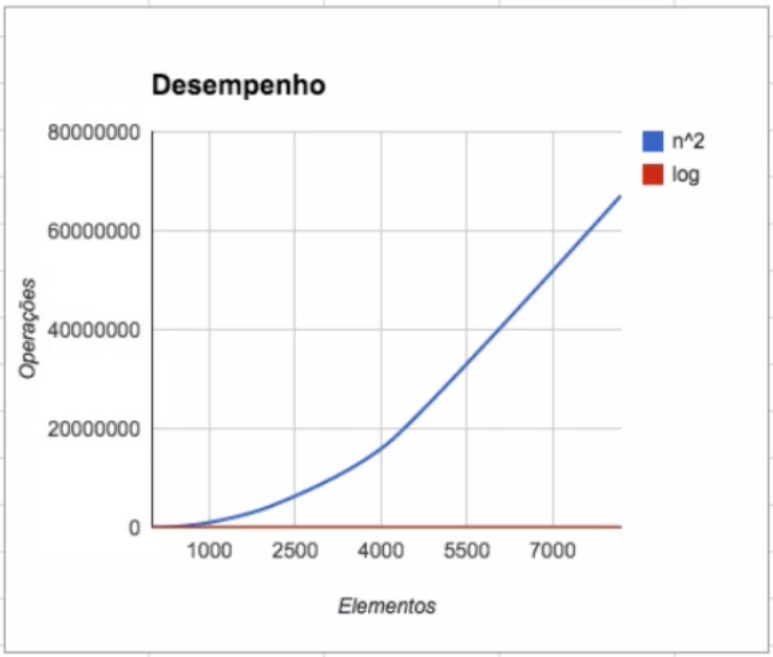
A diferença entre os algoritmos é bastante expressiva. Nitidamente, o cúbico terá um desempenho pior do que o quadrático.

Para 8192 elementos, enquanto *n^2* faz 67.108.864 operações, *n^3* fará 54.905.581.388 - um valor tão grande que nem é possível visualizá-lo inteiro na célula da tabela... Por isso, rodar um algoritmo cúbico no computador é inviável. Pelo menos, usando um semelhante ao do exemplo, que faz 8 mil operações por segundo.

Algoritmos com desempenho baseado em log

Existe o algoritmo constante e o linear. Será que não temos um algoritmo além destes dois? Podem existir vários!

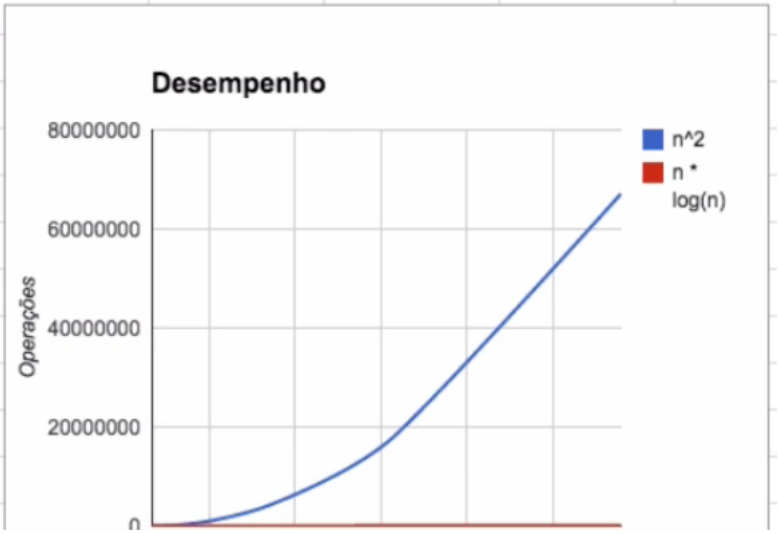
Elementos	n^2	log
1	1	0
2	4	1
4	16	2
8	64	3
16	256	4
32	1024	5
64	4096	6
128	16384	7
256	65536	8
512	262144	9
1024	1048576	10
2048	4194304	11
4096	16777216	12
8192	67108864	13



Outro algoritmo interessante será usado quando procuramos um nome em uma lista telefônica ou em uma agenda, em que os contatos já estão ordenados. Ele não será constante, porque nao sabemos exatamente onde estará posicionado o nome da pessoa. Nós apenas tentamos adivinhar a posição correspondente... Este tipo de algoritmo de busca, também é um algoritmo inteligente (ainda iremos falar bastante sobre ele), que não é constante, nem linear. É algo entre os dois. O algoritmo recebe o nome de **logaritmo**, porque depende do *log* do nosso número (*log na base 2*). Nós iremos ver como e quando usar algoritmos do gênero...

E como funciona o algoritmo *log na base 2*? Ele será o *log* do elemento relacionado na tabela, na base 2. Observe os resultados: quando tivermos dois elementos, ele fará 1 operação. Quando trabalharmos com o maior número de elementos da tabela, ele fará 13 operações. Com 8192 elementos, ele executará 13 operações. É um número muito baixo e um ótimo resultado.

Elementos	n^2	n * log(n)
1	1	0
2	4	2
4	16	8
8	64	24
16	256	64
32	1024	160
64	4096	384
128	16384	896
256	65536	2048
512	262144	4608
1024	1048576	10240
2048	4194304	22528



Se tivermos um *array* de produtos para serem mostrados, poderemos resolver muito rápido as 13 operações.

Voltando ao exemplo da agenda de telefone, provavelmente nós temos menos de 8 mil contatos salvos na agenda. Se fizermos uma busca da maneira como estamos habituados, teremos que executar 8 mil operações. Mas se fizermos uma busca logarítmica, vamos fazer 13 operações - e nós não precisamos fazer uma busca com tantos elementos na nossa agenda... A busca que fazemos em uma agenda telefônica, em geral, é mais esperta.

Mais adiante, nós iremos ensinar ao computador como fazer estas buscas mais espertas, assim como buscamos naturalmente na agenda de contatos.

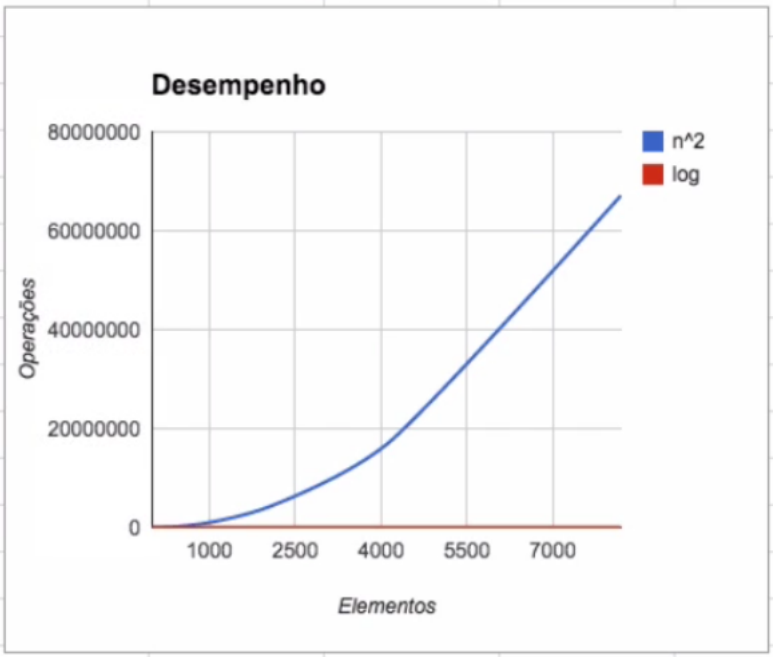
Quando observamos a implementação de um algoritmo que é logarítmico, percebemos que ele cresce menos no gráfico e por isso, tem um desempenho melhor do que uma linear.

Então, até o momento, analisamos os algoritmos constantes, logarítmicos, lineares e depois, a quadrática.

Algoritmos n*log n

Nós conhecemos a logarítmica, a constante, a linear e a quadrática. Vamos incluir esta última novamente na tabela.

Elementos	n^2	log
1	1	0
2	4	1
4	16	2
8	64	3
16	256	4
32	1024	5
64	4096	6
128	16384	7
256	65536	8
512	262144	9
1024	1048576	10
2048	4194304	11
4096	16777216	12
8192	67108864	13

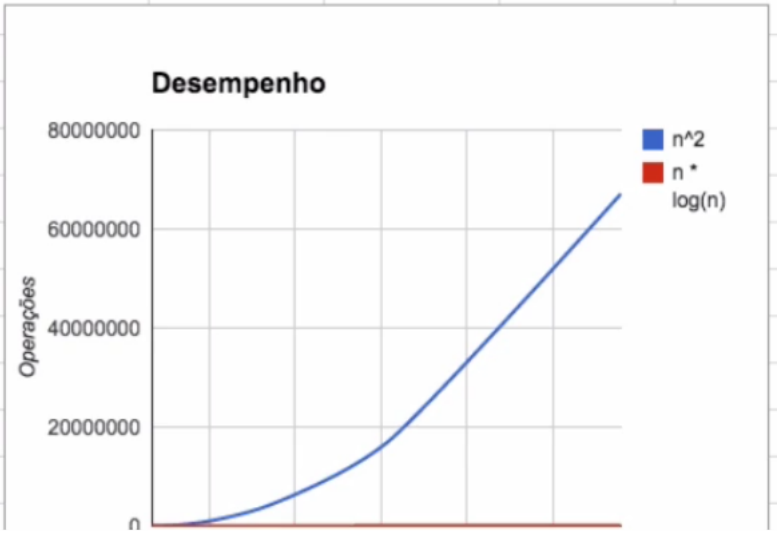


Observe que a quadrática (*n^2*) comparada com a logarítmica (*log*) tem uma diferença gigantesca no gráfico.

Em uma fazemos 13 operações, enquanto na outra fazemos mais de 67 milhões de operações.

E será que entre a linear (*n*) e a quadrática (*n^2*), não existe outra entre as duas linhas? Sim, temos. Iremos multiplicar o *n* pelo *log* de *n*(*nlog(n)*).

Elementos	n^2	n * log(n)
1	1	0
2	4	2
4	16	8
8	64	24
16	256	64
32	1024	160
64	4096	384
128	16384	896
256	65536	2048
512	262144	4608
1024	1048576	10240
2048	4194304	22528



Então, nossos resultados serão superiores aos resultados alcançados com a logarítmica. Quando trabalharmos com 8.192 elementos, teremos que executar 106.496 operações. É maior do que o resultado 13, da logarítmica. Porém, ainda é muito inferior do que o número 67.108.864, da quadrática.

É possível perceber no gráfico do *nlog(n)*, que a linha vermelha surge sutilmente, enquanto a linha azul deslancha. Ao compararmos os algoritmos *nlog(n)* com `n^2` , a primeira opção parece ser muito melhor, por isso daremos preferência para

ela.

Então, temos os algoritmos constante (*1*), o linear (n), o linear logarítmico (*n*log(n)*) e o quadrático (*n^2*). Naturalmente, depois do quadrático ainda teremos mais opções.

Algoritmos cúbicos

Depois do quadrático, o que poderemos ter? Está claro que se aumentarmos o número da potência, os valores ficarão ainda maiores.

Vamos comparar o algoritmo quadrático (*n^2*) com o cúbico (*n^3*)? Os valores da terceira coluna serão o resultado de n^3.

Elementos	n^2	n^3
1	1	1
2	4	8
4	16	64
8	64	512
16	256	4096
32	1024	32768
64	4096	262144
128	16384	2097152
256	65536	16777216
512	262144	134217728

Observe que a linha vermelha deslancha no gráfico!

A diferença entre os algoritmos é bastante expressiva. Nitidamente, o cúbico terá um desempenho pior do que o quadrático.

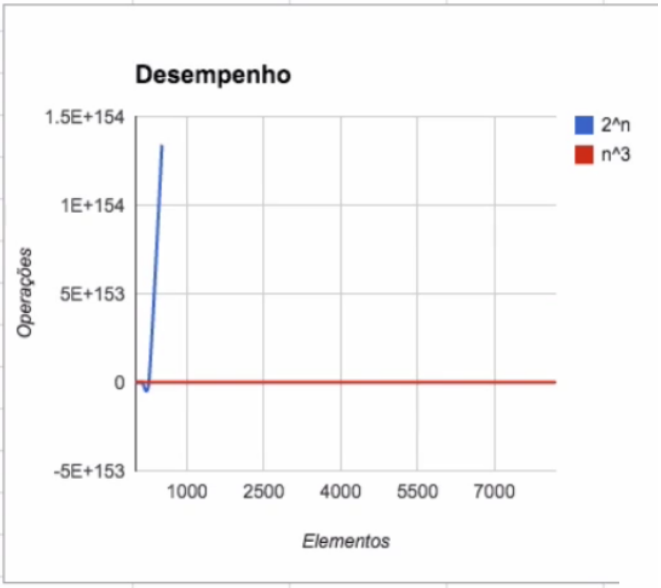
Para 8192 elementos, enquanto *n^2* faz 67.108.864 operações, *n^3* fará 54.905.581.388 - um valor tão grande que nem é possível visualizá-lo inteiro na célula da tabela... Por isso, rodar um algoritmo cúbico no computador é inviável. Pelo menos, usando um semelhante ao do exemplo, que faz 8 mil operações por segundo.

Algoritmo exponencial

Depois do cúbico, será que não teremos mais nada? Sim, ainda podemos elevar à quarta, à quinta, à sexta potência... E o desempenho irá piorando em cada um deles. Evitaremos usá-los, porque será impossível rodá-los.

Anteriormente, vimos que temos o constante (*1*), o linear (*n*), o linear logarítmico (*nlog(n)*) e o quadrático (*n^2*), depois o cúbico (*n^3*)... Mas, será que existe outra categoria em que os resultados crescerão muito rápido e que também irão explodir no gráfico? Sim. *2* elevado a *n* (*2^n*) terá um desempenho ainda pior em relação a todos os outros.

Elementos	2^n	n^3
1	2	1
2	4	8
4	16	64
8	256	512
16	65536	4096
32	4294967296	32768
64	1.84467E+19	262144
128	3.40282E+38	2097152
256	1.15792E+77	16777216
512	1.34078E+154	134217728
1024	#NUM!	1073741824
2048	#NUM!	8589934592
4096	#NUM!	68719476736
8192	#NUM!	549755813888



À medida que fazemos os cálculos podemos perceber que vão surgindo valores elevados:

- 2 elevado a 16 será igual a 65.536.
- 2 elevado a 32 será igual 4.294.967.296.
- 2 elevado a 64 será igual a 1.84467E+19

O resultado de 2 elevado a 64 será um número tão grande, que o programa decidiu escrever em notação científica e apenas indicou a presença de 19 casas, além do valor apresentado... Ele fez o mesmo com outros valores, até desistir e começar a apresentar a mensagem de erro **#NUM!**, nas últimas células da coluna.

No gráfico, o programa foi incapaz de desenhá-lo completamente e a linha do algoritmo exponencial começa a vazar pelo eixo inferior. Isto prova que *2^n* é um algoritmo inviável. Quando trabalharmos com 16 elementos, já será difícil executar a quantidade de operações, com 32 elementos, o tempo de demora será absurdo.

Precisaremos encontrar outra solução para não utilizarmos o algoritmo exponencial.

Observe que existem classes de algoritmos que não podem ser usadas e que irão nos fazer procurar novas soluções.

Análise assintótica de uma algoritmo

Nós vimos que temos diversos tipos de algoritmos e que a performance dos mesmos irá variar bastante. Por isso, estaremos sempre atentos nos casos em que tivermos dois laços alinhados, porque pode ser difícil utilizar alguns deles.

Com dois laços passando por todos os elementos de um *array*, teremos que tomar alguns cuidados - em especial com o segundo, para que ele não seja quadrático.

Por exemplo, 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--;
            }
        }
    }
```

A função `troca` é exatamente mudar um elemento de posição com outro.

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

Será um logaritmo constante, porque ele simplesmente faz esta troca.

Também analisamos outro algoritmo, que passa por todos os elementos e é linear.

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

Ainda que passe apenas na metade ou no dobro dos elementos, ele continuará sendo linear.

Descobrimos que entre os algoritmos analisados, existia um **logarítmico**. Levando em consideração o desempenho, depois do **linear**, tínhamos o **linear logarítmico** e o **quadrático** (como o `insertionSort` e o `selectionSort`).

Quando inseríamos mais um `for` passando por fora do laço e que também varria todos os elementos, ele se tornava **cúbico**. Com mais um `for`, ele era elevado à potência 4, e com outro `for` adicionado era elevado à potência 5.

Analisamos também o **exponencial** (*2^n*) no qual a linha crescia muito rápido no gráfico.

Todos estes valores (constante, linear, logarítmico, quadrático, cúbico e exponencial) são maneiras de descobrirmos a forma geral da curva que o algoritmo se aproxima. Esta é o que chamamos de **análise assintótica**.

É incomum dizermos que um algoritmo linear é 1. A forma correta é *O(1)*, isto significa que analisamos o algoritmo e ele é constante. *O(n)* é linear. $O(n^2)$ é quadrático. $O(n^3)$ é cúbico. *O(2^n)* significa que é exponencial. Utilizamos a letra **O** maiúscula para dizer como o algoritmo se comporta - qual será o seu desempenho assintoticamente.