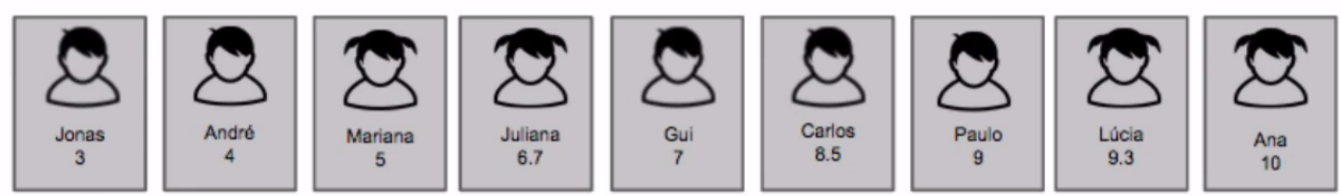
01

Análise assimptótica das buscas

Transcrição

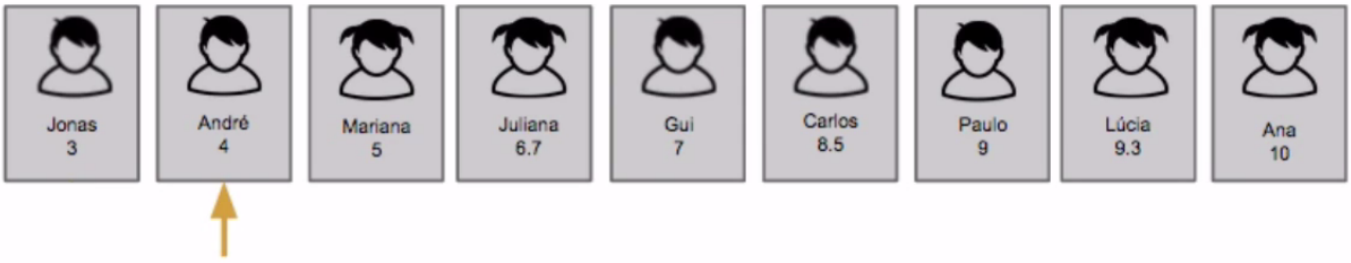
Vamos analisar o quão mais rápida é a busca tradicional - que passa por cada um dos elementos procurando um em específico.

Usaremos exemplos, o primeiro será com a Mariana. Será que ela faz parte do *array*?

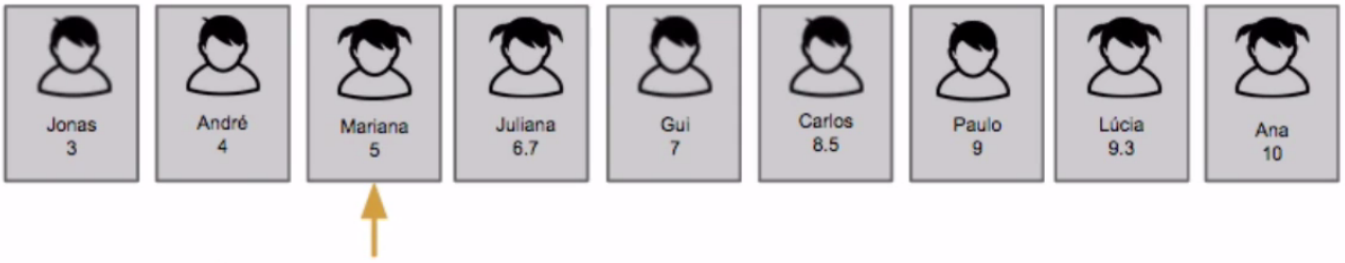


Caso ela faça parte da lista, como funcionava o nosso algoritmo? Ele passava por cada um dos elementos e então, verificávamos se o item procurado estava dentro. Vamos analisar o primeiro elemento:

O Jonas é a Mariana? Não. Seguimos para a posição 1.



O André é a Mariana? Não. Vamos para a posição 2.



A Mariana está nesta posição? Sim! Ela não estava na 0, também não estava na 1, mas estava na posição 2. Assim nós passamos por cada um dos elementos, até encontrarmos o que buscávamos.

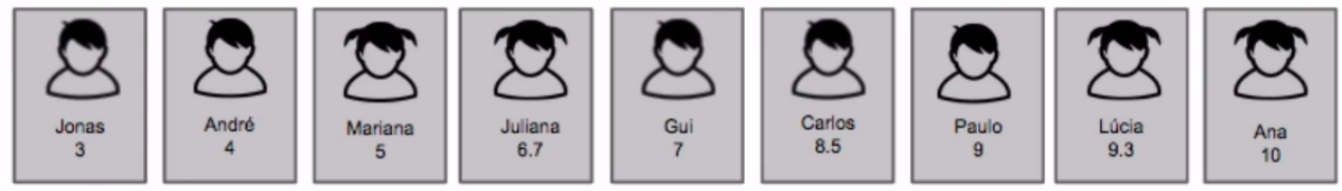
Em uma busca tradicional, nós iremos varrer todos os elementos do *array*. Se tivermos sorte, o elemento que buscamos estará na primeira posição. Se dermos azar, estaremos buscando o último elemento e teremos que passar pelo Jonas, o André, a Mariana, a Juliana, o Gui, o Carlos, o Paulo, a Lúcia e a Ana. Em uma situação ainda pior, procuraríamos alguém que não está no *array*, porém, foi preciso passar por todos os elementos para termos certeza de que ele não fazia parte.

Levando em consideração que precisamos passar por todos os elementos, qual será o número de operações que teremos que fazer? Se temos **cinco** elementos, faremos **cinco** operações. Se temos **100** elementos, faremos **100** operações. Se temos **1000** elementos, faremos **1000** operações. Basicamente, temos que fazer uma operação para cada elemento. Isto significa que se temos *n* elementos, teremos que fazer *n* operações. O algoritmo que busca todos os elementos cresce de acordo com uma linha ascendente que indica: "Se tenho um elemento, farei uma operação. Se tenho dois, farei duas operações. Se tenho cinco,farei cinco..." Ele é um algoritmo linear e segue uma linha.

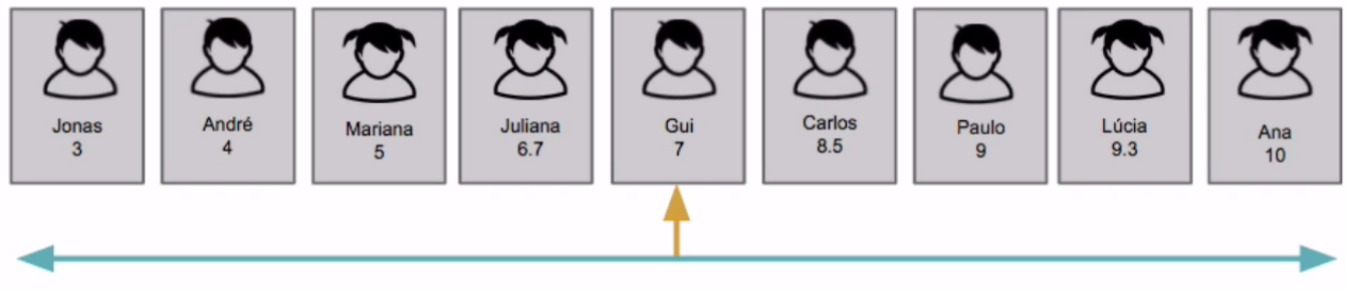
Mesmo que trabalhássemos com *2n* ou *3n*, não faria diferença. O gráfico seria um pouco diferente, mas não de maneira significativa. Nós veremos isto em seguida, quando o compararmos com outros tipos de buscas que nós conhecemos. Nós chamamos a busca tradicional de **busca linear**, porque ela irá passar por todos os elementos, tentando encontrar o elemento procurado. Ela é linear e passará pelos *n* elementos de um *array* de tamanho *n*.

O desempenho de busca binária

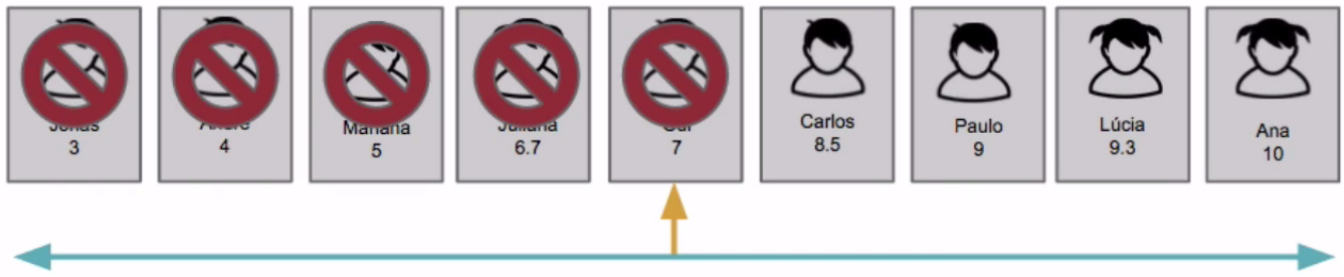
Analisaremos a **busca binária**. Observe a lista com nove elementos:



Em vez de fazermos um busca tradicional, com um `for` que varre da esquerda para a direita, nós optamos por arriscar um elemento do meio.



Depois, o que faríamos? Verificaríamos: o elemento está para esquerda ou para direita? Por exemplo, se vamos procurar o 9.3, ele está posicionado à direita. Então, iremos descartar todos do meio para a esquerda, e analisaremos apenas o pedaço que sobrou à direita.



Agora iremos observar o pedaço e dividiremos no meio novamente.

Comparamos o elemento, e fazemos a pergunta: ele é o que procuramos? Não. A nota está para esquerda ou para direita? Está na direita. O fazemos? Descartamos os que estão na esquerda.

Dividiremos os elementos que sobraram no meio...

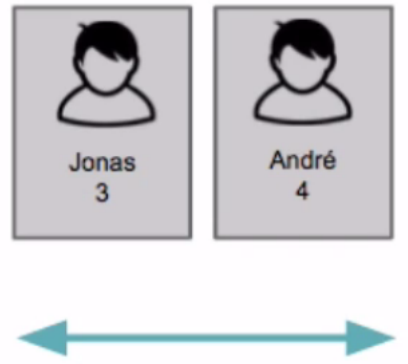
E então, encontramos.

Como funciona a busca binária? Nós dividimos o *array* no meio, verificamos as duas partes e ficamos com apenas uma delas. Seguimos repetindo o proceso de dividir e selecionar uma das partes. A cada nova operação, o número de elementos que descartamos não será **um** como na busca antiga - que descartávamos um elemento por vez. Em cada operação, nós descartaremos metade do *array*. Quando fazemos uma operação, eliminamos metade, depois mais a metade do que sobrou. Ficamos com 1/2, depois com 1/4, em seguida com 1/8, seguimos descartando sempre. O número de elementos diminuir mais rapidamente. Porém, o quão mais rápido?

Vamos ver o que acontece quando buscamos em um *array* com a busca binária. Se temos um único elemento, não precisaremos dividi-lo.

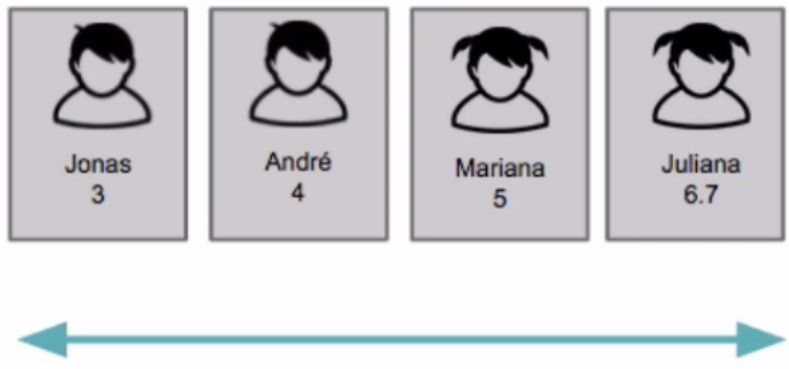
Fazemos a comparação e acabou a busca. Praticamente não faremos nada.

No entanto, se tivermos dois elementos, precisaremos fazer uma operação de divisão.



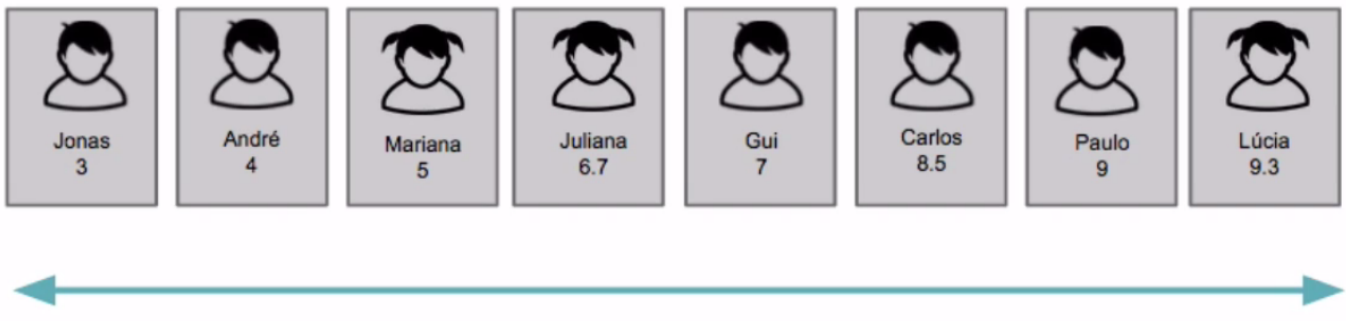
Iremos analisar o item do meio, que será o da posição 0, verei se ele é maior ou menor, se continuaremos para direita ou para a esquerda. Neste caso, faremos **uma** operação de divisão. Não temos como evitá-la, quando trabalhamos com dois elementos.

E se tivermos três ou quatro elementos, o que faremos? Primeiro, uma operação irá dividir o *array* no meio.



Então, analisaremos em qual metade está o elemento. Ficaremos com uma delas e depois, dividiremos novamente no meio. Ou seja, precisaremos de **duas** operações para ficar com um elemento. Na primeira operação, ficaremos com dois elementos, com a segunda, restará apenas um.

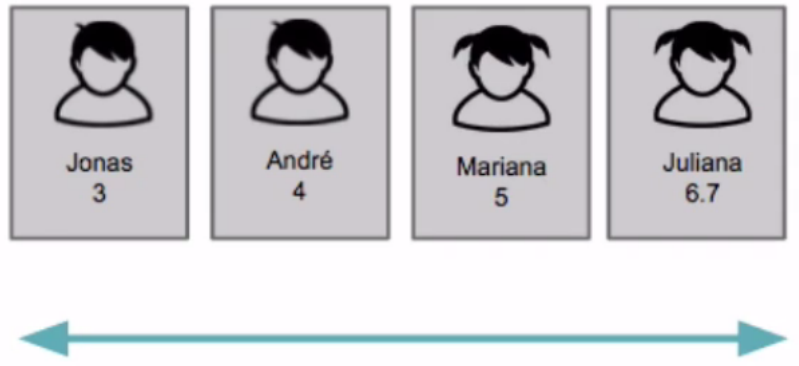
E quando temos oito elementos?



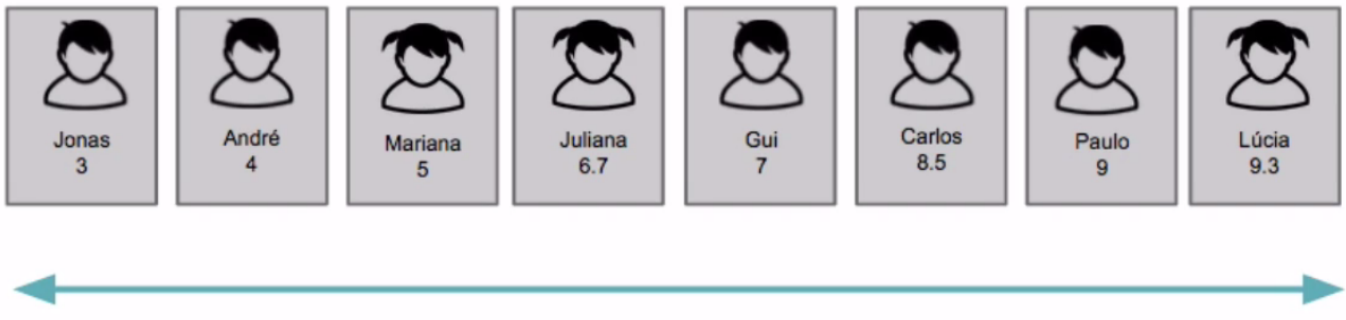
Após fazermos uma comparação e sobrarão quatro elementos. Com mais uma operação, sobrarão dois. Com outra comparação, sobrará um elemento. Quando temos oito elementos, no máximo em **três** operações descobriremos o item procurado.

E se temos mais elementos, como nove, dez ou até 16 elementos? Como fazemos? Com uma operação quebraremos o número na metade e ficarão no máximo oito. Com duas operações, sobrarão quatro. Com três operações, sobrarão dois. E no máximo, com quatro operações, sobrará um. Está surgindo um padrão no processo. A cada novo passo, nós dividimos o número de elemento por 2. Isto significa que a cada operação de divisão, analisávamos 2^1 (elementos), ou seja, dois elementos.

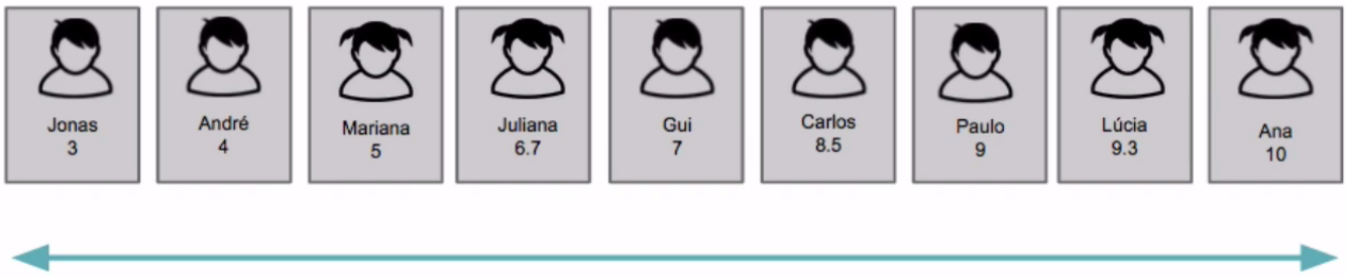
Com duas operações, nós somos capazes de analisar um número de elementos igual a 2^2, ou seja, quatro elementos.



Com três operações, somos capazes de analisar um número de elementos igual a 2^3, ou seja, oito elementos.



Com quatro operações, somos capazes de analisar um número de elementos igual a 2 elevado 4, ou seja, dezesseis elementos.



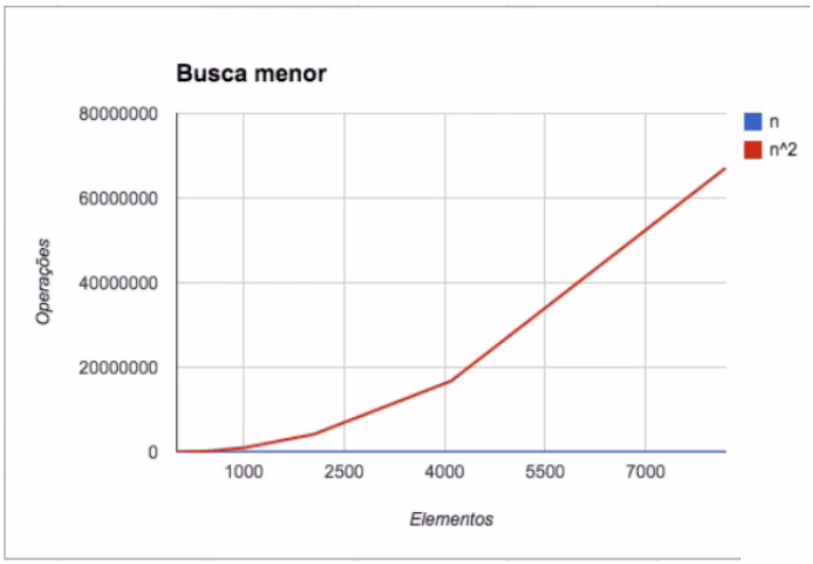
E com cinco operações, analisaremos um número de elementos igual a 2 elevado 5, o que significa 32 elementos. Se forem seis operações, analisaremos um número de elementos igual a 2 elevado a 6, 64. Com dez operações, serão 2 elevado a 10, 1024 elementos. Com dez operações conseguimos comparar 1024 elementos. É um número bem elevado de itens em relação ao número de operações. Então, se temos o valor 1024, como chegaremos ao número da potência dez? Como conseguimos que 2 elevado ao número de operações fosse igual a 1024? Ele é o ***log*** na ***base 2***, que significa: qual número que eu exponenciar resultará no valor total. O número 2 elevado a qual potência será igual a 1024? O número 2 elevado a qual potência será igual a 32? Ou o número 2 elevado a qual potência será igual a 8? Se temos 2048 elementos, 2 elevado a qual potência resultará neste valor? A resposta será onze. Então, precisaremos de **onze** operações. O número de operações irá crescer de acordo com o log do número de elementos. Quando faço uma busca binária, ele é *O* de *log n* na *base 2* - em geral, usamos *base 2*, na computação.

Analisando a busca binária

Agora que sabemos que uma busca tradicional é **linear**, se temos *n* elementos, ela executará *n* operações ou algo na grandeza de *n*, como *O(n)*, *3n*, *3n* + 15, *5n* - 17. Ela sempre estará em função de *n* solto (e não exponenciado ao quadrado ou ao cubo). A busca **binária** é igual a *log n* na *base 2*, o que significa que ela crescerá de acordo com o *log n*.

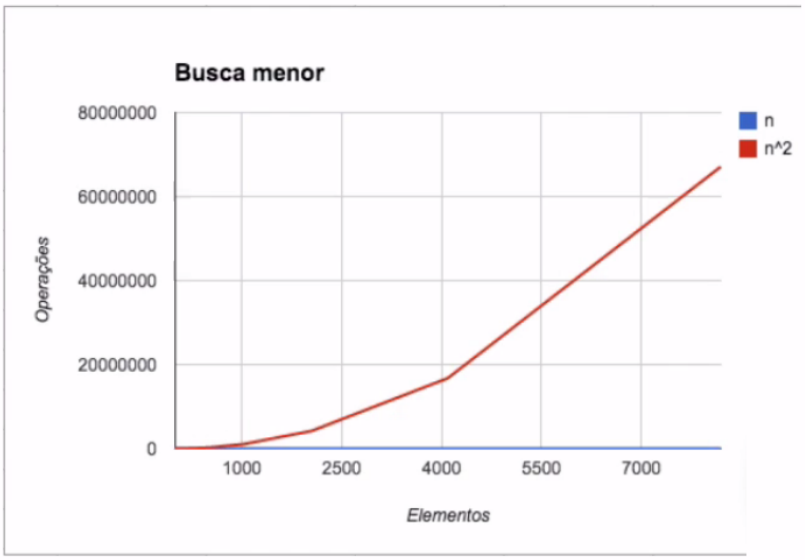
Por exemplo, o número 2 terá que ser elevado a qual potência para resultar em 1024? A resposta é dez. Então precisaremos executar dez operações, com 1024 elementos. Se a busca cresce de maneira logarítmica, como poderemos compará-la com a linear? Criaremos um gráfico para compararmos o algoritmo **linear (n*) e o** quadrático *(n^2*).

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



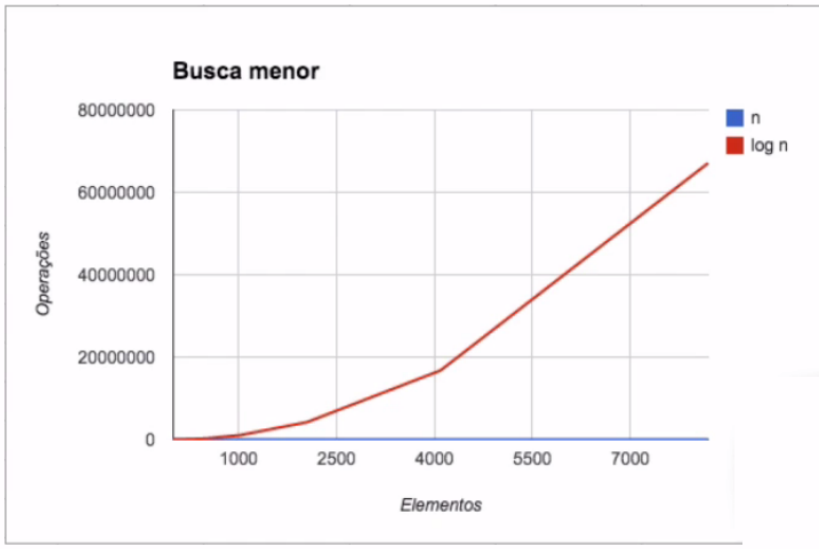
Observe como a linha vermelha do **quadrático** cresce no gráfico, enquanto a linha azul do **linear** permanece próxima ao eixo inferior, porque executa o número de operações exatamente igual a *n*.

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



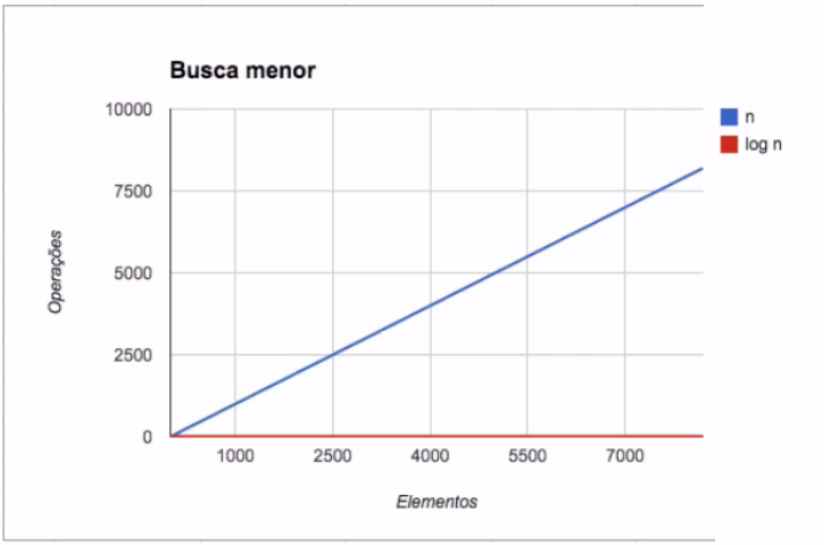
A busca binária é logarítmica e irá buscar *log* de *n* na *base 2* - escreveremos apenas *logn*.

Elementos	n	log n
1	1	0
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



Quando temos 1 elemento, o resultado será igual a 0, ou seja, ele fará **zero** divisões. Faremos uma comparação simples, porém, iremos ignorar o +1, -1, ou +15. O importante é a maneira como o gráfico crescerá.

Elementos	n	log n
1	1	0
2	2	1
4	4	2
8	8	3
16	16	4
32	32	5
64	64	6
128	128	7
256	256	8
512	512	9
1024	1024	10
2048	2048	11
4096	4096	12
8192	8192	13

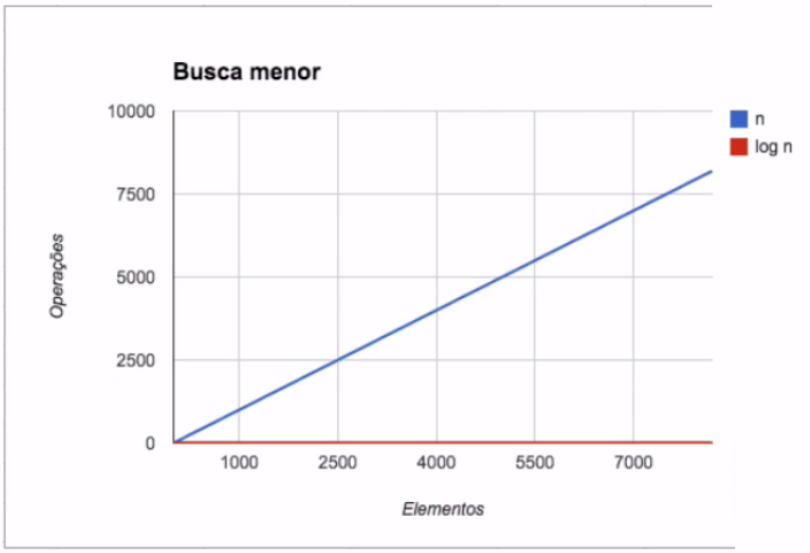


Quando tivermos um elemento, precisaremos dividir zero vezes. Se forem dois elementos, faremos **uma** divisão. Se forem quatro elementos, faremos **duas** divisões. Com oito elementos, faremos **três** divisões. O número de operações estará relacionado com a potência em que o número 2 será exponenciado. Por exemplo, 2 elevado à quarta potência será igual a 16.

Elementos	n	log n
1	1	0
2	2	1
4	4	2
8	8	3
16	16	4

Com 32 elementos, faremos **cinco** comparações. Com 64, faremos **seis** comparações. Com 128, faremos **sete** comparações. O número de divisões que faremos será os valores que estão na coluna *logn*.

Elementos	n	log n
1	1	0
2	2	1
4	4	2
8	8	3
16	16	4
32	32	5
64	64	6
128	128	7
256	256	8
512	512	9
1024	1024	10
2048	2048	11
4096	4096	12
8192	8192	13



Iremos comparar com os resultados da outra busca (coluna *n*): Se temos um *array* com 8192, em uma busca linear tradicional, faremos 8192 comparações. Numa busca logarítmica (binária), nós faremos quantas comparações? No máximo, treze operações.

Qual é a sacada? Nós dividimos o *array* exatamente no meio e analisamos. Depois repetimos o processo enquanto for preciso, em vez de fazermos 8192 operações, faremos apenas 13. É por isso que, por exemplo, quando verificamos uma lista de presença, nós direcionamos nosso olhar para uma posição aproximada, onde esperamos que o elemento esteja localizado. Se olhássemos um item por vez, teríamos que fazer 8192 comparações. Em uma sala de aula com esta quantidade de alunos, seria preciso fazer um número muito elevado de comparações.

As pessoas em geral, naturalmente, fazem uma busca mais inteligente, quando trabalham com um *array* ordenado. Nós começamos a procura a partir de uma posição aproximadamente correta. Fazemos este processo instintivamente, em vez de varrermos o *array* inteiro. Por isso, a busca binária é extremamente mais rápida do que a sequencial normal, que passará por todos os elementos. Mas para isto, precisamos que o *array* já esteja ordenado.