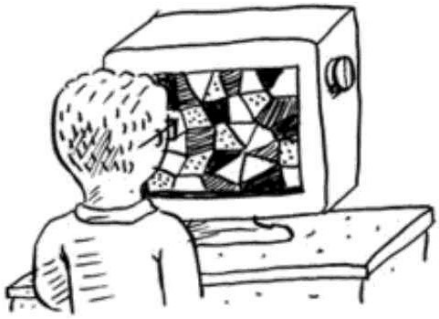
Vamos supor que você esteja procurando o nome de uma pessoa em uma agenda telefônica (que frase antiquada!). O nome começa com *K*. Você pode começar na primeira página da agenda e ir folheando até chegar aos *K*s. Porém você provavelmente vai começar pela metade, pois sabe que os *K*s estarão mais perto dali.

Ou suponha que esteja procurando uma palavra que começa com *O* em um dicionário. Novamente, você começa a busca pelo meio.

Agora, imagine que você entre no Facebook. Quando faz isso, o Facebook precisa verificar que você tem uma conta no site. Logo, ele procura seu nome de usuário em um banco de dados. Digamos que seu usuário seja karlmageddon. O Facebook poderia começar pelos *A*s e procurar seu nome – mas faz mais sentido que ele comece a busca pelo meio.



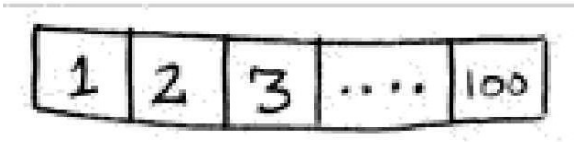
Isto é um problema de busca. E todos estes casos usam um algoritmo para resolvê-lo: *pesquisa binária*.

A pesquisa binária é um algoritmo. Sua entrada é uma lista ordenada de elementos (explicarei mais tarde por que motivo a lista precisa ser ordenada). Se o elemento que você está buscando está na lista, a pesquisa binária retorna a sua localização. Caso contrário, a pesquisa binária retorna `None`.

Por exemplo:

Procurando empresas em uma agenda com a pesquisa binária.

Eis um exemplo de como a pesquisa binária funciona. Estou pensando em um número entre 1 e 100.



Você deve procurar adivinhar o meu número com o menor número de tentativas possível. A cada tentativa, digo se você chutou muito para cima, muito para baixo ou corretamente.

Digamos que começou tentando assim: 1, 2, 3, 4… Veja como ficaria.

Uma tentativa ruim de acertar o número.

Isso se chama *pesquisa simples* (talvez *pesquisa estúpida* seja um termo melhor). A cada tentativa, você está eliminando apenas um número. Se o meu número fosse o 99, você precisaria de 99 chances para acertá-lo!

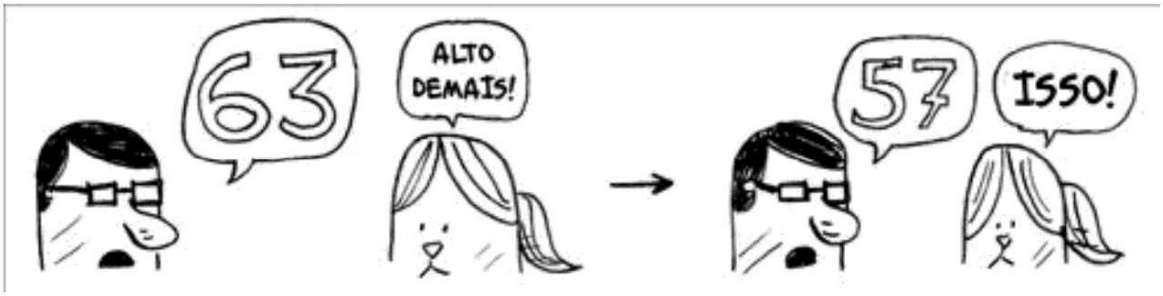
Uma maneira melhor de buscar

Aqui está uma técnica melhor. Comece com 50.

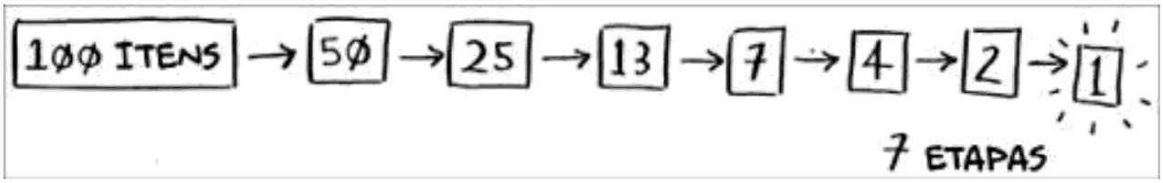
Muito baixo, mas você eliminou *metade* dos números! Agora, você

sabe que os números de 1 a 50 são muito baixos. Próximo chute: 75.

Muito alto, mas novamente você pode cortar metade dos números restantes! *Com a pesquisa binária, você chuta um número intermediário e elimina a metade dos números restantes a cada vez.* O próximo número é o 63 (entre 50 e 75).



Isso é a pesquisa binária. Você acaba de aprender um algoritmo! Aqui está a quantidade de números que você pode eliminar a cada tentativa.



Elimine metade dos números a cada tentativa com a pesquisa binária.

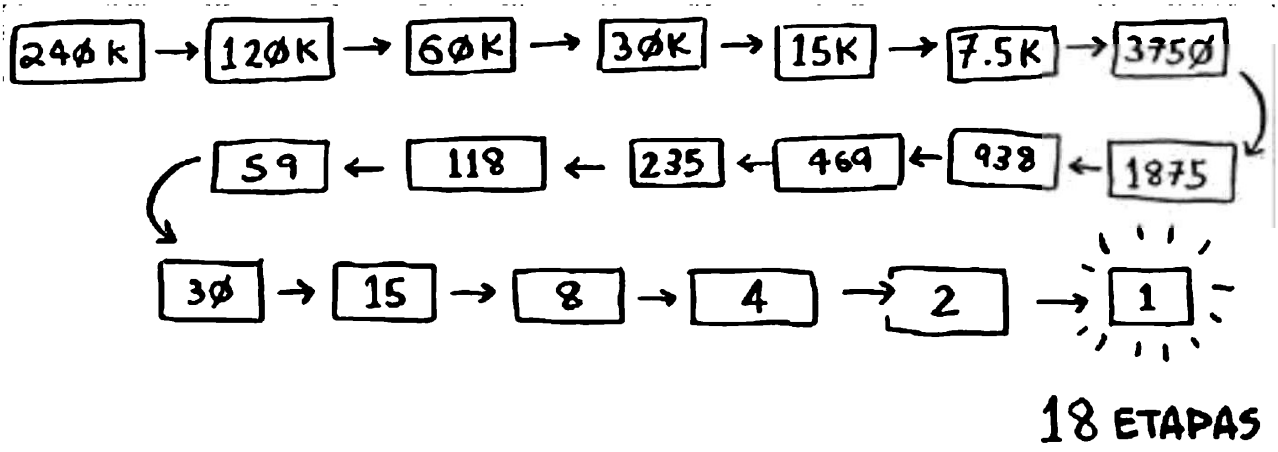
Seja qual for o número que eu estiver pensando, você pode adivinhá-lo em um máximo de sete tentativas – porque a pesquisa binária elimina muitas possibilidades!

Suponha que você esteja procurando uma palavra em um dicionário.

O dicionário tem 240.000 palavras. *Na pior das hipóteses*, de quantas etapas você acha que a pesquisa precisaria?

PESQUISA SIMPLES: _____ ETAPAS
PESQUISA BINÁRIA: _____ ETAPAS

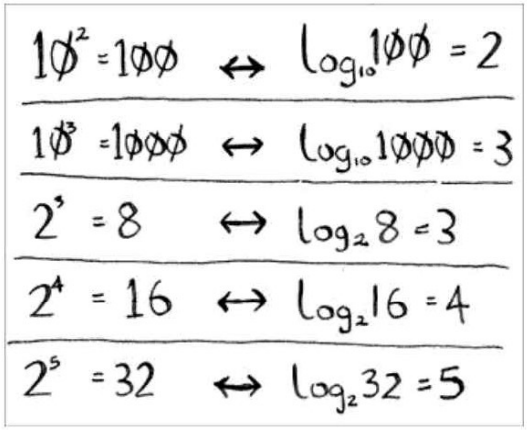
A pesquisa simples poderia levar 240.000 etapas se a palavra que você estivesse procurando fosse a última do dicionário. A cada etapa da pesquisa binária, você elimina o número de palavras pela metade até que só reste uma palavra.



Logo, a pesquisa binária levaria apenas 18 etapas – uma grande diferença! De maneira geral, para uma lista de *n* números, a pesquisa binária precisa de $\log_2 n$ para retornar o valor correto, enquanto a pesquisa simples precisa de *n* etapas.

Logaritmos

Você pode não se lembrar de logaritmos, mas provavelmente lembra-se de como calcular exponenciais. A expressão $\log_{10} 100$ basicamente diz: "Quantos 10s conseguimos multiplicar para chegar a 100?". A resposta é 2: 10×10. Então, $\log_{10} 100 = 2$. Logaritmos são o oposto de exponenciais.



Logaritmos são o oposto de exponenciais.

Neste livro, quando falamos sobre a notação Big O (explicada daqui a pouco), levamos em conta que log sempre significa $\log_2$. Quando você procura um elemento usando a pesquisa simples, no pior dos casos, terá de analisar elemento por elemento, passando por todos. Se for uma lista de oito elementos, precisaria checar no máximo oito números. Na pesquisa binária, precisa verificar log *n* elementos para o pior dos casos. Para uma lista de oito elementos, log 8 == 3, porque $2^3 == 8$. Então, para uma lista de oito números, precisaria passar por, no máximo, três tentativas. Para uma lista de 1.024 elementos, log 1.024 == 10, porque $2^4 == 1.024$. Logo, para uma lista de 1.024 números, precisaria verificar no máximo dez deles.

Nota

Falarei muito sobre logaritmos neste livro. Portanto você deve entender o conceito. Se não entender, a Khan Academy (*khanacademy.org*) tem um vídeo legal que esclarece muita coisa.

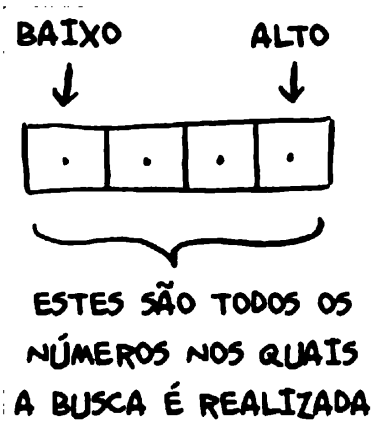
Nota

A pesquisa binária só funciona quando a sua lista está ordenada. Por exemplo, os nomes em uma agenda telefônica estão em ordem alfabética, então você pode utilizar a pesquisa binária para procurar um nome. O que aconteceria se a lista não estivesse ordenada?

Vamos ver como escrever a pesquisa binária em Python. O exemplo de código que utilizamos aqui usa arrays. Se não sabe como eles funcionam, não se preocupe; abordaremos isso no próximo capítulo. Você só precisa saber que pode armazenar uma sequência de elementos em uma linha de buckets consecutivos que se chama array. Os buckets são numerados a partir do 0: o primeiro bucket está na posição #0; o segundo, em #1; o terceiro, em #2, e assim por diante.

A função `pesquisa_binaria` pega um array ordenado e um item. Se o item está no array, a função retorna a sua posição. Dessa maneira, você é capaz de saber por qual ponto do array deve continuar procurando. No começo, o código do array segue assim:

```
baixo = 0
alto = len(lista) - 1
```



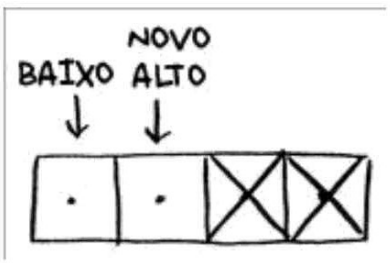
A cada tentativa, você testa para o elemento central.

```
meio = (baixo + alto) / 2 ❶
chute = lista[meio]
```

❶ `meio` será arredondado para baixo automaticamente pelo Python se `(baixo + alto)` não for um número par.

Se o chute for muito baixo, você atualizará a variável `baixo` proporcionalmente:

```
if chute < item:
  baixo = meio + 1
```



E se o chute for muito alto, você atualizará a variável `alto`. Aqui está o código completo:

```
def pesquisa_binaria(lista, item):
```

```
    baixo = 0 ❶
    alto = len(lista) - 1 ❶

    while baixo <= alto: ❷
        meio = (baixo + alto) / 2 ❸
        chute = lista[meio]
        if chute == item: ❹
            return meio
        if chute > item: ❺
            alto = meio - 1
        else: ❻
            baixo = meio + 1
    return None ❼

minha_lista = [1, 3, 5, 7, 9] ❽

print pesquisa_binaria(minha_lista, 3) # => 1 ❾
print pesquisa_binaria(minha_lista, -1) # => None ❿
```

❶ `baixo` e `alto` acompanham a parte da lista que você está procurando.

❷ Enquanto ainda não conseguiu chegar a um único elemento...

❸ ... verifica o elemento central.

❹ Acha o item.

❺ O chute foi muito alto.

❻ O chute foi muito baixo.

❼ O item não existe.

❽ Vamos testá-lo!

❾ Lembre-se, as listas começam no 0. O próximo endereço tem índice 1.

❿ "None" significa nulo em Python. Ele indica que o item não foi encontrado.

EXERCÍCIOS

1.1 Suponha que você tenha uma lista com 128 nomes e esteja fazendo uma pesquisa binária. Qual seria o número máximo de etapas que você levaria para encontrar o nome desejado?

1.2 Suponha que você duplique o tamanho da lista. Qual seria o número máximo de etapas agora?

Digite o texto aqui

Tempo de execução

Sempre que falo sobre um algoritmo, falo sobre o seu tempo de execução. Geralmente, você escolhe o algoritmo mais eficiente – caso esteja tentando otimizar tempo e espaço.

Voltando à pesquisa simples, quanto tempo se otimiza utilizando-a? Bem, a primeira abordagem seria verificar número por número. Se fosse uma lista de 100 números, precisaríamos de 100 tentativas. Se fosse uma lista de 4 bilhões de números, precisaríamos de 4 bilhões de tentativas. Logo, o número máximo de tentativas é igual ao tamanho da lista. Isso é chamado de *tempo linear*.

A pesquisa binária é diferente. Se a lista tem 100 itens, precisa-se de, no máximo, sete tentativas. Se tem 4 bilhões, precisa-se de, no máximo, 32 tentativas. Poderoso, não? A pesquisa binária é executada com *tempo logarítmico*. A tabela a seguir resume as nossas descobertas até agora.

Tempo de execução para algoritmos de pesquisa.

Notação Big O

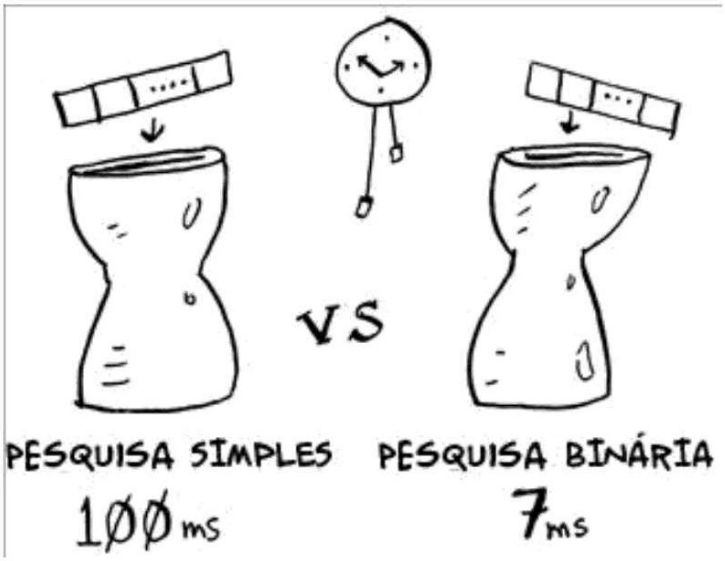
A notação *Big O* é uma notação especial que diz o quão rápido é um algoritmo. Mas quem liga para isso? Bem, acontece que você frequentemente utilizará o algoritmo que outra pessoa fez – e quando faz isso, é bom entender o quão rápido ou lento o algoritmo é. Nesta seção, explicarei como a notação Big O funciona e fornecerei uma lista com os tempos de execução mais comuns para os algoritmos.

Tempo de execução dos algoritmos cresce a taxas diferentes

Bob está escrevendo um algoritmo para a NASA. O algoritmo dele entrará em ação quando o foguete estiver prestes a pousar na lua, e ele o ajudará a calcular o local de pouso.

Este é um exemplo de como o tempo de execução de dois algoritmos pode crescer a taxas diferentes. Bob está tentando decidir entre a pesquisa simples e a pesquisa binária. O algoritmo precisa ser tão rápido quanto correto. Por um lado, a pesquisa binária é mais rápida, o que é bom, pois Bob tem apenas *10 segundos* para descobrir onde pousar, ou o foguete sairá de seu curso. Por outro lado, é mais fácil escrever a pesquisa simples, o que gera um risco menor de erros. Bob não quer *mesmo* erros no seu código! Para ser ainda mais cuidadoso, Bob decide cronometrar ambos os algoritmos com uma lista de 100 elementos.

Vamos presumir que leva-se 1 milissegundo para verificar um elemento. Com a pesquisa simples, Bob precisa verificar 100 elementos, então a busca leva 10 ms para rodar. Em contrapartida, ele precisa verificar apenas sete elementos na pesquisa binária ($\log_2$ 100 é aproximadamente 7), logo, a pesquisa binária leva 7 ms para ser executada. Porém, realisticamente falando, a lista provavelmente terá em torno de 1 bilhão de elementos. Se a lista tiver esse número, quanto tempo a pesquisa simples levará para ser executada? E a pesquisa binária? Tenha certeza de que sabe a resposta para essa pergunta antes de continuar lendo.



Tempo de execução para pesquisa simples vs. pesquisa binária para uma lista de 100 elementos.

Bob executa a pesquisa binária com 1 bilhão de elementos e leva 30 ms ($\log_2$ 1.000.000.000 é aproximadamente 30). "30 ms!" – ele pensa. "A pesquisa binária é quase 15 vezes mais rápida do que a pesquisa simples, porque a pesquisa simples levou 100 ms para uma lista de 100 elementos e a pesquisa binária levou só 7 ms. Logo, a pesquisa simples levará 30 × 15 = 450 ms, certo? Bem abaixo do meu limite de 10 segundos." Bob decide utilizar a pesquisa simples. Ele fez a escolha certa?

Não. Bob está errado. Muito errado. O tempo de execução para a pesquisa simples para 1 bilhão de itens é 1 bilhão ms, ou seja, 11 dias! O problema é que o tempo de execução da pesquisa simples e da pesquisa binária *cresce com taxas diferentes.*

	PESQUISA SIMPLES	PESQUISA BINÁRIA
100 ELEMENTOS	100ms	7ms
10000 ELEMENTOS	10 segundos	14 ms
1,000,000,000 ELEMENTOS	11 dias	32ms

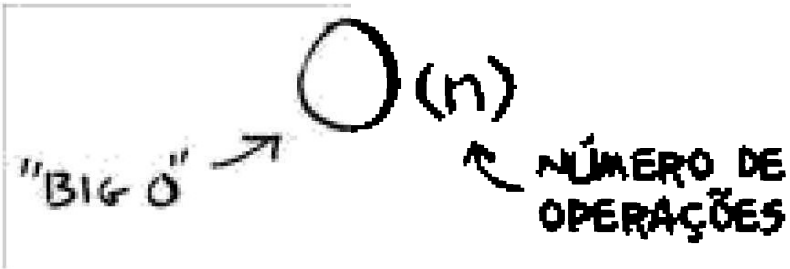
Tempos de execução crescem com velocidades diferentes!

Sendo assim, conforme o número de itens cresce, a pesquisa binária aumenta só um pouco o seu tempo de execução. Já a pesquisa

simples leva *muito* tempo a mais. Logo, conforme a lista de números cresce, a pesquisa binária se torna *muito* mais rápida do que a pesquisa simples. Bob pensou que a pesquisa binária fosse 15 vezes mais rápida que a pesquisa simples, mas isso está incorreto. Se a lista tem 1 bilhão de itens, o tempo de execução é aproximadamente 33 milhões de vezes mais rápido. Por isso, não basta saber quanto tempo um algoritmo leva para ser executado – você precisa saber se o tempo de execução aumenta conforme a lista aumenta. É aí que a notação Big O entra.

A notação Big O informa o quão rápido é um algoritmo. Por exemplo, imagine que você tem uma lista de tamanho *n*. O tempo de execução na notação Big O é O(*n*). Onde estão os segundos? Eles não existem – a notação Big O não fornece o tempo em segundos. *A notação Big O permite que você compare o número de operações*. Ela informa o quão rapidamente um algoritmo cresce.

Temos outro exemplo disso. A pesquisa binária precisa de log *n* operações para verificar uma lista de tamanho *n*. Qual é o tempo de execução na notação Big O? É O(log *n*). De maneira geral, a notação Big O é escrita da seguinte forma:



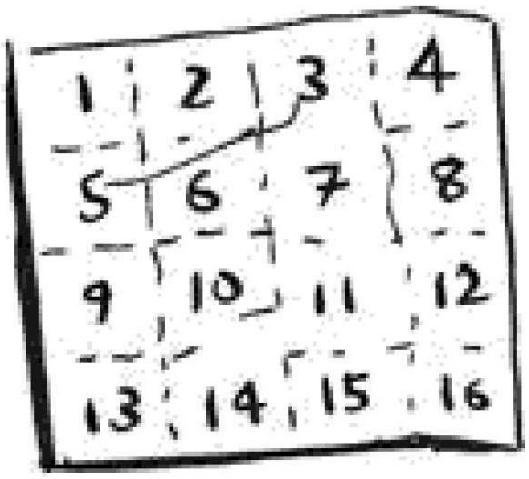
O formato da notação Big O.

Isso fornece o número de operações que um algoritmo realiza. É chamado de notação Big O porque coloca-se um "grande O" na frente do número de operações (parece piada, mas é verdade!).

Agora, vamos ver alguns exemplos. Veja se consegue descobrir o tempo de execução para esses algoritmos.

Vendo diferentes tempos de execução Big O

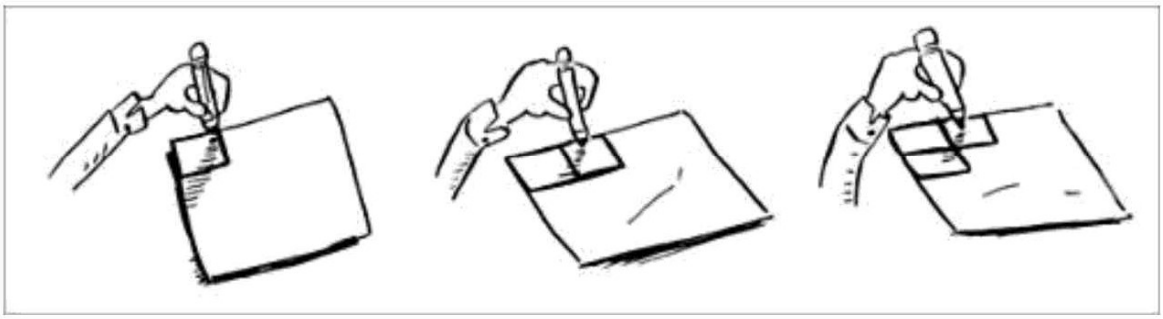
Aqui segue um exemplo prático que você pode reproduzir em casa com um pedaço de papel e um lápis. Suponha que você tenha que desenhar uma grade com 16 divisões.



Qual é um bom algoritmo para desenhar essa grade?

Algoritmo 1

Uma forma de desenhar essa grade de 16 divisões é desenhar uma divisão de cada vez. Lembre-se, a notação Big O conta o número de operações. Nesse exemplo, desenhar uma divisão é uma operação. Você precisa desenhar 16 divisões. Quantas operações você terá de fazer se desenhar uma divisão por vez?



Desenhando a grade executando uma divisão por vez.

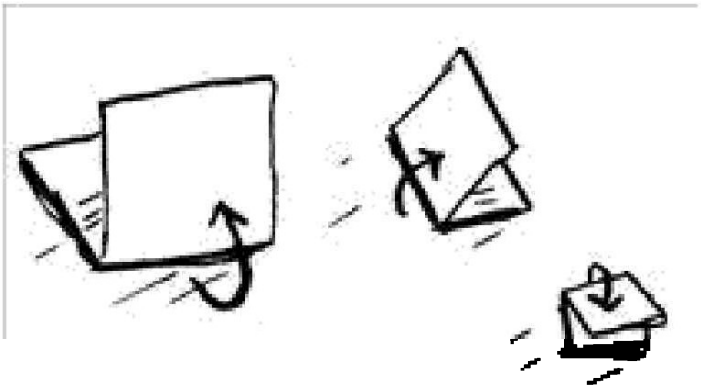
É necessário passar por 16 etapas para desenhar 16 divisões. Qual é o tempo de execução desse algoritmo?

Algoritmo 2

Tente agora este algoritmo. Dobre o papel.

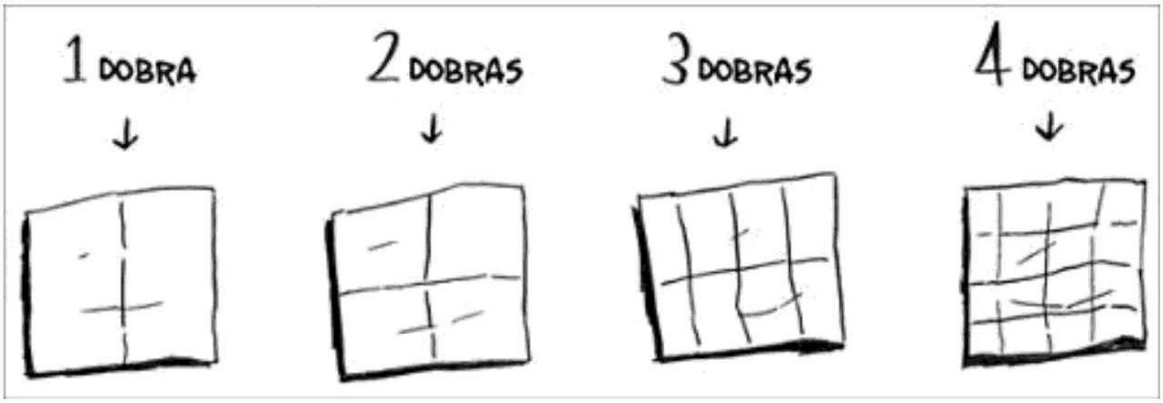
Neste exemplo, dobrar o papel uma vez é uma operação. Você fez duas divisões com essa operação!

Dobre o papel de novo, de novo e de novo.



Desdobre depois de quatro dobras e você terá uma bela grade! A cada dobra, o número de divisões duplica. Você fez 16 divisões com

quatro operações!



Desenhando uma grade com quatro dobras.

Você pode "desenhar" duas vezes mais divisões a cada dobra, logo, você pode desenhar 16 divisões em quatro etapas. Qual é o tempo de execução para esse algoritmo? Encontre o tempo de execução dos dois algoritmos antes de seguir adiante.

Respostas: O Algoritmo 1 tem tempo de execução O(n) e o algoritmo 2 tem tempo de execução O(log n).

A notação Big O estabelece o tempo de execução para a pior hipótese

Suponha que você utiliza uma pesquisa simples para procurar o nome de uma pessoa em uma agenda telefônica. Você sabe que a pesquisa simples tem tempo de execução O(n), o que significa que na pior das hipóteses terá verificado cada nome da agenda telefônica. Nesse caso, você está procurando uma pessoa chamada Adit. Essa pessoa é a primeira de sua lista. Logo, não teve de passar por todos os nomes – você a encontrou na primeira tentativa. Esse algoritmo levou o tempo de execução O(n)? Ou levou O(1) porque encontrou o que queria na primeira tentativa?

A pesquisa simples ainda assim tem tempo de execução O(n). Nesse caso, você encontrou o que queria instantaneamente. Essa é a melhor das hipóteses. A notação Big O leva em conta a *pior das hipóteses*. Então pode-se dizer que, para o *pior caso*, você analisou cada item da lista. Esse é o tempo O(n). É uma garantia – você sabe, com certeza,

que a pesquisa simples nunca terá tempo de execução mais lento do que O(n).

Nota

Além do tempo de execução para o pior dos casos, é importante analisar o tempo de execução para o "caso médio". O pior caso e o caso médio serão discutidos no Capítulo 4.

Alguns exemplos comuns de tempo de execução Big O

Aqui temos cinco tempos de execução Big O que você encontrará bastante, ordenados do mais rápido para o mais lento.

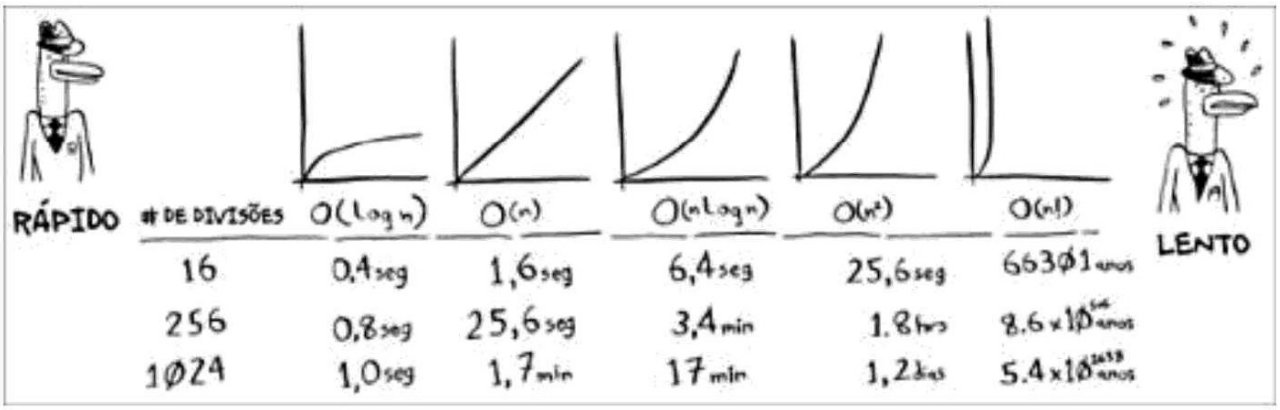
- O(log n), também conhecido como *tempo logarítmico*. Exemplo: pesquisa binária.
- O(n), conhecido como *tempo linear*. Exemplo: pesquisa simples.
- O(n * log n). Exemplo: um algoritmo rápido de ordenação, como a ordenação quicksort (explicada no Capítulo 4).
- O(n^2). Exemplo: um algoritmo lento de ordenação, como a ordenação por seleção (explicada no Capítulo 2).
- O(n!). Exemplo: um algoritmo bastante lento, como o do caixeiro-viajante (explicado a seguir!).

Suponha que você esteja desenhando novamente a grade de 16 divisões e você possa escolher cinco algoritmos diferentes para fazer isso. Se escolher o primeiro algoritmo, levará um tempo de execução de O(log n) para desenhar a grade. Você pode fazer dez operações por segundo. Com o tempo de execução O(log n), você levará quatro operações para desenhar uma grade com 16 divisões (log 16 é 4). Logo, levará 0,4 segundos para desenhar a grade. E se tiver que desenhar 1.024 divisões? Levará 1.024 = 10 operações, ou um segundo para desenhar uma grade de 1.024 divisões. Estes números são para o primeiro algoritmo.

O segundo algoritmo é mais lento: ele tem tempo de execução O(n). Levará 16 operações para desenhar 16 divisões e levará 1.024

operações para desenhar 1.024 divisões. Quanto tempo isso leva em segundos?

Aqui está quanto tempo levaria para desenhar a grade com os algoritmos restantes, do mais rápido ao mais lento:



# DE DIVISÕES	O(log n)	O(n)	O(n log n)	O(n²)	O(n!)
16	0,4 seg	1,6 seg	6,4 seg	25,6 seg	663Ø1 anos
256	0,8 seg	25,6 seg	3,4 min	1.8 hrs	8.6 x 1Ø^504 anos
1Ø24	1,0 seg	1,7 min	17 min	1,2 dias	5.4 x 1Ø^2558 anos

Existem outros tempos de execução, mas esses são os cinco mais comuns.

Isso é uma simplificação. Na realidade, você não pode converter um tempo de execução na notação Big O para um número de operações, mas a aproximação é boa o suficiente por enquanto. Voltaremos a falar da notação Big O no Capítulo 4, depois de ter aprendido um pouco mais sobre algoritmos. Por enquanto, os principais pontos são os seguintes:

- A rapidez de um algoritmo não é medida em segundos, mas pelo crescimento do número de operações.
- Em vez disso, discutimos sobre o quão rapidamente o tempo de execução de um algoritmo aumenta conforme o número de elementos aumenta.
- O tempo de execução em algoritmos é expresso na notação Big O.
- O(log *n*) é mais rápido do que O(*n*), e O(log *n*) fica ainda mais rápido conforme a lista aumenta.

EXERCÍCIOS

Forneça o tempo de execução para cada um dos casos a seguir em termos da notação Big O.

1.3 Você tem um nome e deseja encontrar o número de telefone para esse nome em uma agenda telefônica.

1.4 Você tem um número de telefone e deseja encontrar o dono dele em uma agenda telefônica. (Dica: Deve procurar pela agenda inteira!)

1.5 Você quer ler o número de cada pessoa da agenda telefônica.

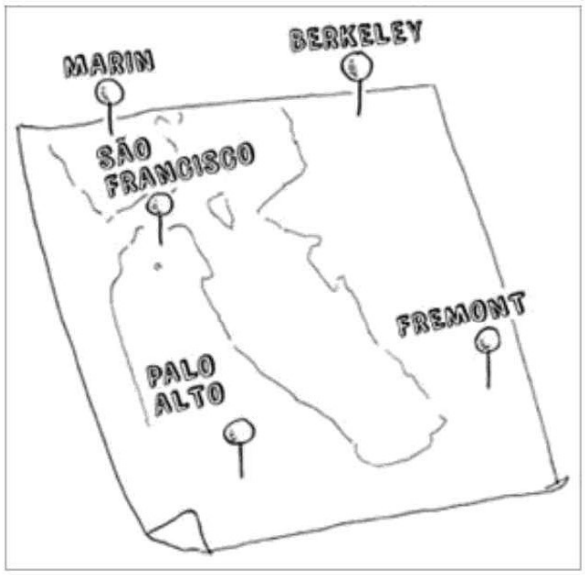
1.6 Você quer ler os números apenas dos nomes que começam com *A*. (Isso é complicado! Esse algoritmo envolve conceitos que são abordados mais profundamente no Capítulo 4. Leia a resposta – você ficará surpreso!)

O caixeiro-viajante

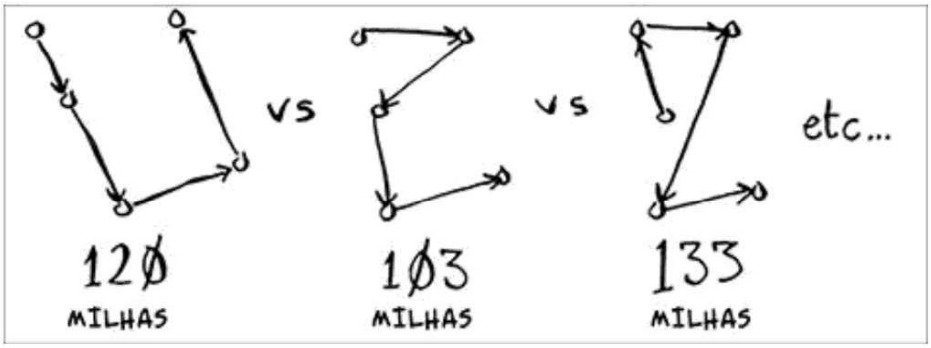
Você pode ter lido a última seção e pensado: "De maneira alguma vou executar um algoritmo que tem tempo de execução O(*n*!)." Bem, deixe-me tentar provar o contrário! Aqui está um exemplo de um algoritmo com um tempo de execução muito ruim. Ele é um problema famoso da ciência da computação, pois seu crescimento é apavorante e algumas pessoas muito inteligentes acreditam que ele possa ser melhorado. Esse algoritmo é chamado de "o problema do *caixeiro-viajante*".

Você tem um caixeiro-viajante.

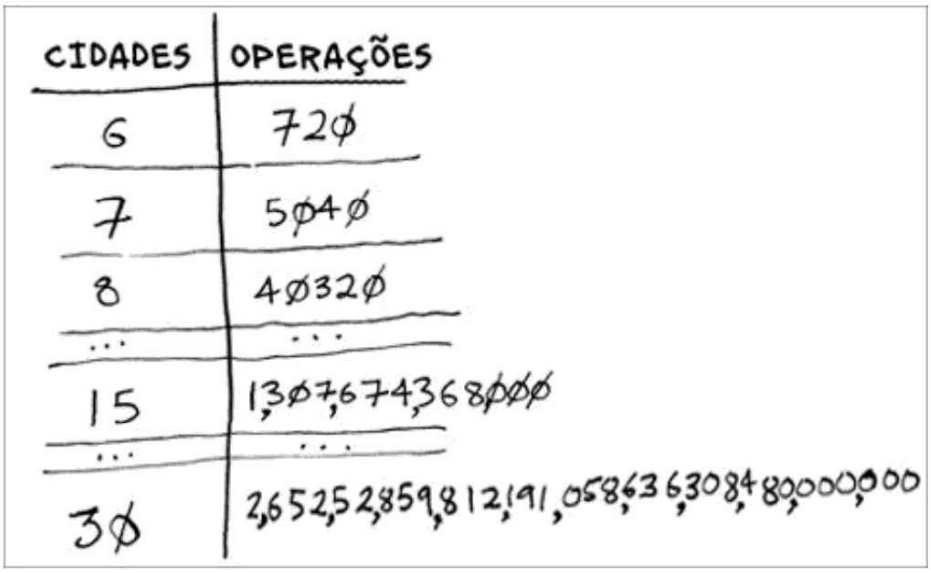
O caixeiro precisa ir a cinco cidades.



O caixeiro, o qual chamarei de Opus, quer passar por todas as cidades percorrendo uma distância mínima. Podemos enxergar o problema da seguinte forma: analisar cada ordem possível de cidades para visitar.



Ele soma a distância total e escolhe o caminho de menor distância. Existem 120 permutações para cinco cidades, logo, precisa-se de 120 operações para resolver o problema de cinco cidades. Para seis cidades, precisa-se de 720 operações (ou 720 permutações). Para sete cidades são necessárias 5.050 operações!



O número de operações aumenta drasticamente.

De maneira geral, para n itens, é necessário $n!$ (fatorial de n) operações para chegar a um resultado. Então, este é o tempo de execução $O(n!)$ ou o *tempo fatorial*. Esse algoritmo consome muitas operações, exceto para casos envolvendo números pequenos. No entanto, uma vez que lidamos com mais de 100 cidades, é impossível calcular a resposta em função do tempo – o sol entrará em colapso antes.

Esse é um algoritmo terrível! Opus deveria usar outro, não? Mas ele não pode. Esse é um problema sem solução. Não existe um algoritmo mais rápido para esse problema, e as pessoas mais inteligentes acreditam ser *impossível* melhorá-lo. O melhor que se pode fazer é chegar a uma solução aproximada. Veja o Capítulo 10 para saber mais sobre isso.

Uma observação final: se você é um leitor avançado, leia sobre árvores binárias de busca. No capítulo seguinte, há uma breve descrição do assunto.

Recapitulando

- A pesquisa binária é muito mais rápida do que a pesquisa simples.
- $O(\log n)$ é mais rápido do que $O(n)$, e $O(\log n)$ fica ainda mais rápido conforme os elementos da lista aumentam.

- A rapidez de um algoritmo não é medida em segundos.
- O tempo de execução de um algoritmo é medido por meio de seu *crescimento*.
- O tempo de execução dos algoritmos é expresso na notação Big O.