

Estrutura de Dados

Prof. Orlando Saraiva Júnior
orlando.nascimento@fatec.sp.gov.br

Busca Binária



Prática

Analisar os códigos

busca1.py

busca2.py

Qual será o mais rápido ?



Suponha que você esteja procurando por uma pessoa na lista telefônica (que frase antiquada!).

O nome dela começa com K. Você poderia começar do início e continuar virando as páginas até chegar aos Ks.

Mas é mais provável que você comece em uma página do meio, porque você sabe que os Ks estarão perto do meio da lista telefônica.



Ou suponha que você esteja procurando uma palavra em um dicionário e ela comece com O. Novamente, você começará perto do meio.

Agora, suponha que você se conecte ao Facebook. Ao fazer isso, o Facebook precisa verificar se você possui uma conta no site. Portanto, ele precisa procurar seu nome de usuário em seu banco de dados. Suponha que seu nome de usuário seja karlmageddon. O Facebook poderia começar com o A e procurar seu nome — mas faz mais sentido começar em algum lugar no meio.

Este é um problema de busca. E todos esses casos usam o mesmo algoritmo para resolver o problema: **busca binária**.

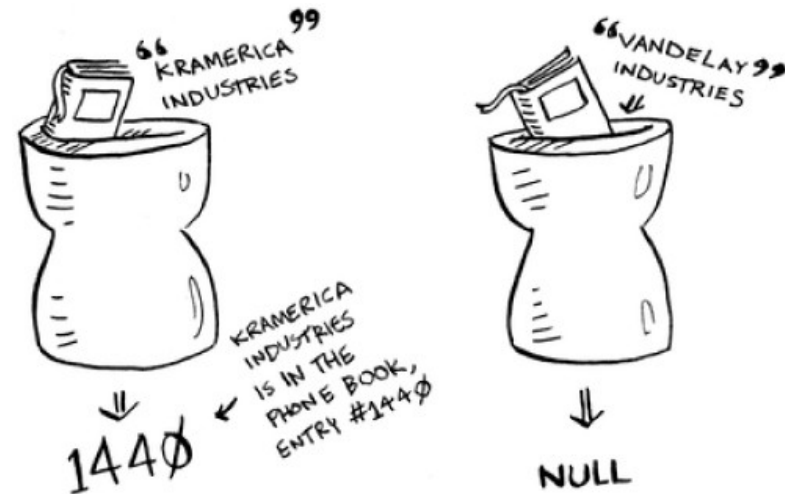
A busca binária é um algoritmo; sua entrada é uma lista ordenada de elementos (explicarei mais tarde por que ela precisa ser ordenada). Se um elemento que você está procurando estiver nessa lista, a busca binária retornará a posição onde ele está localizado. Caso contrário, a busca binária retornará nulo.

Procurando empresas em uma lista telefônica com busca binária

Aqui está um exemplo de como a busca binária funciona. Estou pensando em um número entre 1 e 100.

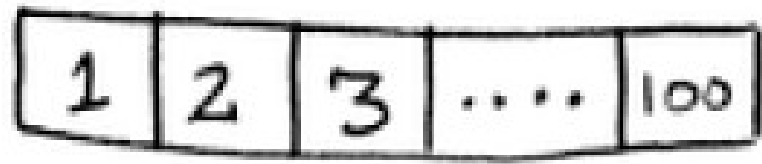
A busca binária é um algoritmo; sua entrada é uma lista ordenada de elementos (explicarei mais tarde por que ela precisa ser ordenada).

Se um elemento que você está procurando estiver nessa lista, a busca binária retornará a posição onde ele está localizado. Caso contrário, a busca binária retornará nulo.



Procurando empresas em uma lista telefônica com busca binária

Aqui está um exemplo de como a busca binária funciona. Estou pensando em um número entre 1 e 100.



Você precisa tentar adivinhar o meu número com o menor número de tentativas possível. A cada tentativa, eu lhe direi se o seu palpite está muito baixo, muito alto ou correto.

Suponha que você comece a chutar assim: 1, 2, 3, 4... Veja como seria.

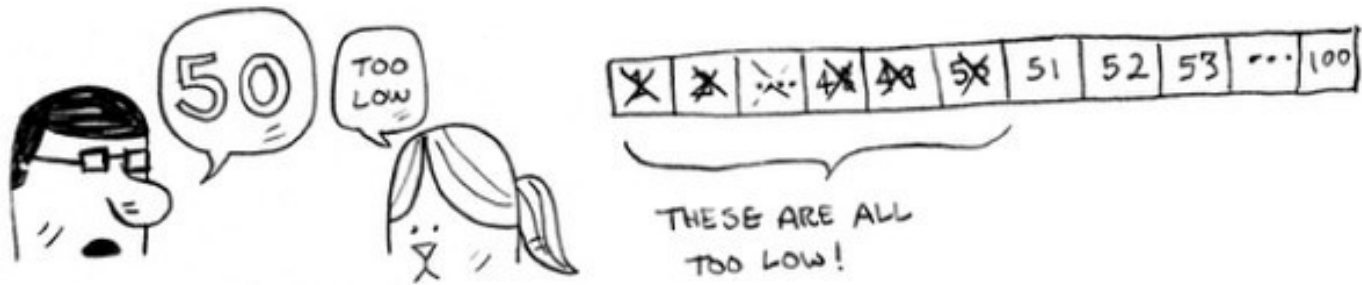
Esta é uma busca simples (talvez busca estúpida seja um termo melhor). A cada palpite, você elimina apenas um número. Se o meu número fosse 99, você poderia tentar 99 vezes para chegar lá!

Busca binária



Uma maneira melhor de pesquisar

Aqui está uma técnica melhor. Comece com 50.



Muito baixo, mas você acabou de eliminar metade dos números! Agora você sabe que de 1 a 50 são todos muito baixos. Próximo palpite: 75.

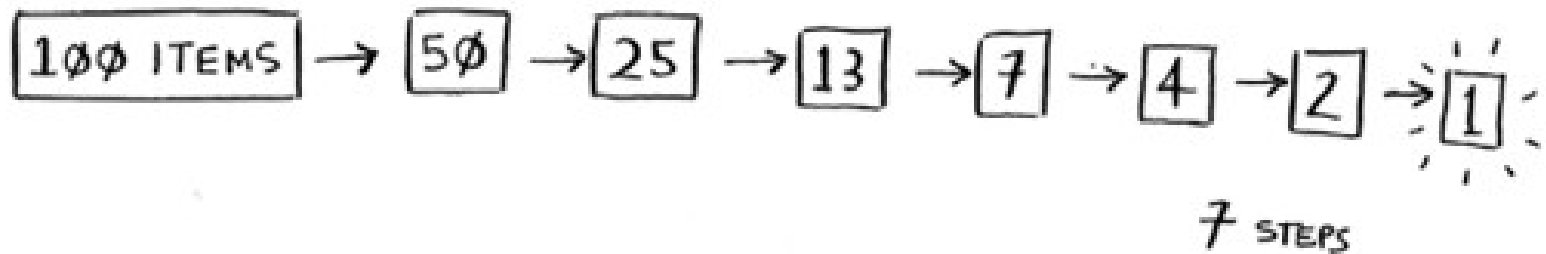


Muito alto, mas, novamente, você reduz pela metade os números restantes! Com a busca binária, você adivinha o número do meio e elimina metade dos números restantes todas as vezes.

O próximo é 63 (meio caminho entre 50 e 75).



Esta é a busca binária. Você acabou de aprender seu primeiro algoritmo! Veja quantos números você pode eliminar a cada vez.



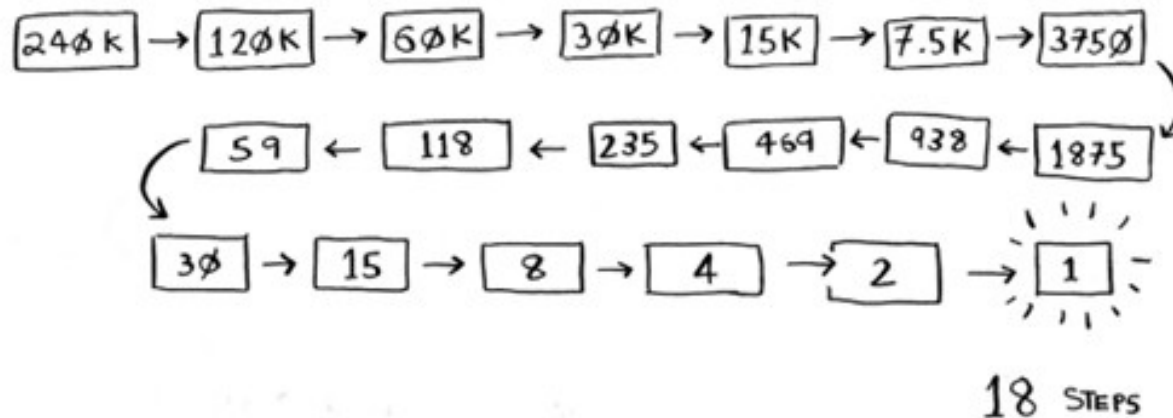
Elimine metade dos números todas as vezes com a busca binária.

Seja qual for o número que estou pensando, você pode tentar no máximo sete vezes — porque você elimina muitos números a cada tentativa!

Suponha que você esteja procurando uma palavra no dicionário. O dicionário tem 240.000 palavras. Na pior das hipóteses, quantos passos você acha que cada busca levará?

Uma busca simples pode levar 240.000 passos se a palavra que você procura for a última do livro.

A cada passo da busca binária, você reduz o número de palavras pela metade até restar apenas uma.



Portanto, a busca binária levará 18 passos — uma grande diferença! Em geral, para qualquer lista de n , a busca binária levará $\log_2 n$ passos para ser executada no pior caso, enquanto a busca simples levará n passos.

Você pode não se lembrar do que são logaritmos, mas provavelmente sabe o que são exponenciais. $\log_{10}100$ é como perguntar: "Quantos 10s multiplicamos para obter 100?". A resposta é 2: 10×10 . Portanto, $\log_{10}100 = 2$.

Logs são a inversão das exponenciais.

$$10^2 = 100 \quad \Leftrightarrow \quad \log_{10} 100 = 2$$

$$10^3 = 1000 \quad \Leftrightarrow \quad \log_{10} 1000 = 3$$

$$2^3 = 8 \quad \Leftrightarrow \quad \log_2 8 = 3$$

$$2^4 = 16 \quad \Leftrightarrow \quad \log_2 16 = 4$$

$$2^5 = 32 \quad \Leftrightarrow \quad \log_2 32 = 5$$

Logaritmos são a inversão das exponenciais.

Quando falo sobre tempo de execução na notação Big O, logaritmo sempre significa \log_2 . Ao procurar um elemento usando a busca simples, no pior caso, você pode ter que olhar para cada elemento.

Portanto, para uma lista de 8 números, você teria que verificar no máximo 8 números. Para uma busca binária, você teria que verificar $\log n$ elementos no pior caso.

Para uma lista de 8 elementos, $\log 8 == 3$, porque $2^3 == 8$. Portanto, para uma lista de 8 números, você teria que verificar no máximo 4 números.

Para uma lista de 1.024 elementos, $\log 1.024 = 10$, porque $2^{10} == 1.024$.

Portanto, para uma lista de 1.024 números, você teria que verificar no máximo 10 números.



Suponha que você tenha uma lista ordenada de 128 nomes e esteja pesquisando nela usando a busca binária.

Qual é o número máximo de passos necessários?





Suponha que você tenha uma lista ordenada de 128 nomes e esteja pesquisando nela usando a busca binária.

Qual é o número máximo de passos necessários?

7 passos



A busca binária só funciona quando a sua lista está ordenada. Por exemplo, os nomes em uma lista telefônica são ordenados em ordem alfabética, então você pode usar a busca binária para procurar um nome.

```
def binary_search(list, item):
```

```
    low = 0
```

```
    high = len(list)-1
```

low and high keep track of which part of the list you'll search in.

```
    while low <= high:
```

While you haven't narrowed it down to one element ...

```
        mid = (low + high) / 2
```

```
        guess = list[mid]
```

... check the middle element.

```
        if guess == item:
```

Found the item.

```
            return mid
```

```
        if guess > item:
```

The guess was too high.

```
            high = mid - 1
```

```
        else:
```

The guess was too low.

```
            low = mid + 1
```

The item doesn't exist.

```
    return None
```

Let's test it!

```
my_list = [1, 3, 5, 7, 9]
```

```
print binary_search(my_list, 3) # => 1
```

Remember, lists start at 0.
The second slot has index 1.

```
print binary_search(my_list, -1) # => None
```

"None" means nil in Python. It indicates that the item wasn't found.

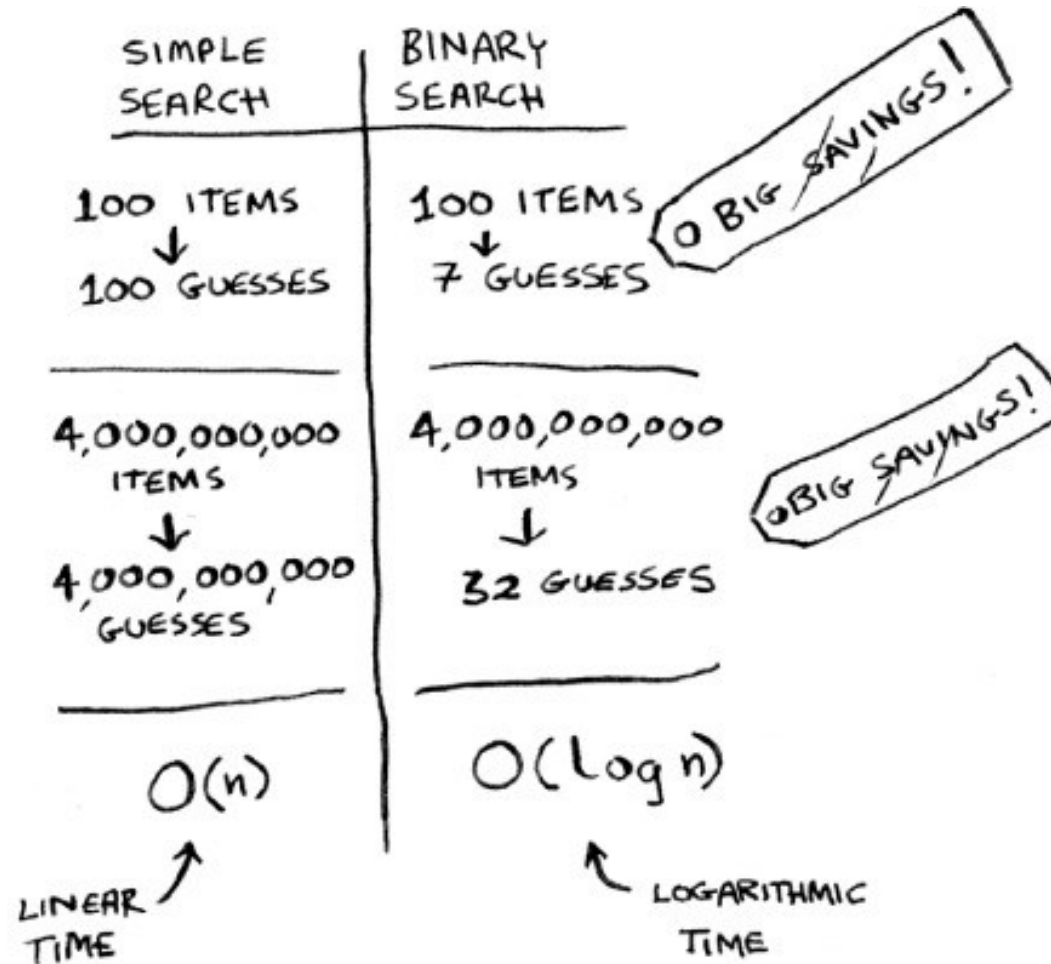
Sempre que falamos sobre um algoritmo, falamos sobre seu tempo de execução. Você quer escolher o algoritmo mais eficiente, esteja você tentando otimizar para tempo ou espaço. Sobre a busca binária, quanto tempo você economiza usando-a ?

Bem, a primeira abordagem foi verificar cada número, um por um. Se for uma lista de 100 números, são necessárias até 100 tentativas.

Se for uma lista de 4 bilhões de números, são necessárias até 4 bilhões de tentativas. Portanto, o número máximo de tentativas é igual ao tamanho da lista. Isso é chamado de **tempo linear**.

A busca binária é diferente. Se a lista tiver 100 itens, são necessárias no máximo 7 tentativas. Se a lista tiver 4 bilhões de itens, são necessárias no máximo 32 tentativas. A busca binária é executada em **tempo logaritmico**.

Tempo de Execução



Quais algoritmos são mais rápidos que outros? Você certamente pode executar experimentos com cada programa em um computador específico e com um conjunto específico de dados para ver qual é o mais rápido.

Essa capacidade é útil, mas quando o computador ou os dados mudam, você pode obter resultados diferentes.

Os computadores geralmente ficam mais rápidos à medida que tecnologias melhores são inventadas, e isso faz com que todos os algoritmos sejam executados mais rapidamente. As mudanças nos dados, no entanto, são mais difíceis de prever.

Você já viu que uma busca binária leva muito menos etapas do que uma busca linear porque o número de itens a serem pesquisados aumenta. A notação Big O é uma notação especial que indica a velocidade de um algoritmo.

Bob está escrevendo um algoritmo de busca para a NASA. Seu algoritmo será ativado quando um foguete estiver prestes a pousar na Lua e ajudará a calcular onde pousar.

Este é um exemplo de como o tempo de execução de dois algoritmos pode crescer em taxas diferentes. Bob está tentando decidir entre busca simples e busca binária. O algoritmo precisa ser rápido e correto.



Por um lado, a busca binária é mais rápida. E Bob tem apenas 10 segundos para descobrir onde pousar - caso contrário, o foguete sairá do curso.

Por outro lado, a busca simples é mais fácil de escrever e há menos chance de bugs serem introduzidos. E Bob realmente não quer que bugs no código pousem um foguete! Para ser extremamente cuidadoso,

Bob decide cronometrar ambos os algoritmos com uma lista de 100 elementos. Vamos supor que leva 1 milissegundo para verificar um elemento. Com a busca simples, Bob precisa verificar 100 elementos, então a busca leva 100 ms para ser executada.

Por outro lado, ele só precisa verificar 7 elementos com a busca binária ($\log^2 100$ é aproximadamente 7),



Notação Big O



SIMPLE
SEARCH

100ms

VS



BINARY
SEARCH

7ms



Prática

Mas, realisticamente, a lista terá algo em torno de um bilhão de elementos.

Se tiver, quanto tempo levará a busca simples?

Quanto tempo levará a busca binária?



Bob executa uma busca binária com 1 bilhão de elementos e leva 30 ms ($\log^2 1.000.000.000$ é aproximadamente 30).

O tempo de execução para uma busca simples com 1 bilhão de itens será de 1 bilhão de ms, o que equivale a 11 dias!

O problema é que os tempos de execução para a busca binária e a busca simples não crescem na mesma proporção.



Notação Big O



	SIMPLE SEARCH	BINARY SEARCH
100 ELEMENTS	100ms	7ms
10,000 ELEMENTS	10 seconds	14ms
1,000,000,000 ELEMENTS	11 days	32ms

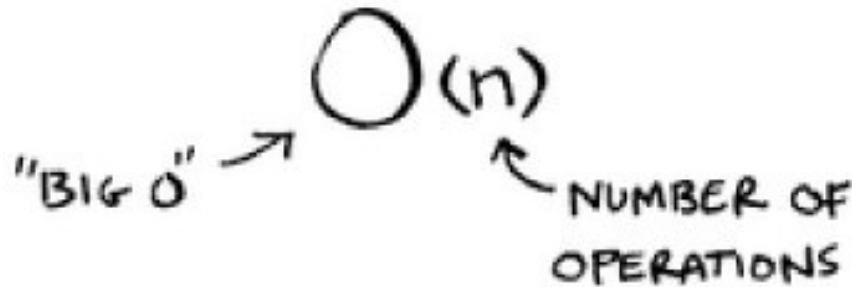
A notação Big O indica a velocidade de um algoritmo. Por exemplo, suponha que você tenha uma lista de tamanho n . A busca simples precisa verificar cada elemento, portanto, levará n operações.

O tempo de execução na notação Big O é **$O(n)$** . Onde estão os segundos?

Não há nenhum — a notação Big O não indica a velocidade em segundos. A notação Big O permite comparar o número de operações. Ela indica a velocidade de crescimento do algoritmo.

A busca binária precisa de operações de $\log n$ para verificar uma lista de tamanho n . Qual é o tempo de execução na notação Big O? É **$O(\log n)$** .

Em geral, a notação Big O é escrita da seguinte forma.



A handwritten diagram illustrating the components of the Big O notation $O(n)$. The letter 'O' is written in a large, hand-drawn font. To its left, the text '"BIG O"' is written, with an arrow pointing from it to the 'O'. To the right of the 'O' is the variable '(n)'. Below the '(n)', the text 'NUMBER OF OPERATIONS' is written, with an arrow pointing from it to the '(n)'.

Aqui estão cinco tempos de execução do Big O que você encontrará com frequência, classificados do mais rápido para o mais lento:

$O(\log n)$ também conhecido como tempo logarítmico. Exemplo: Busca binária.

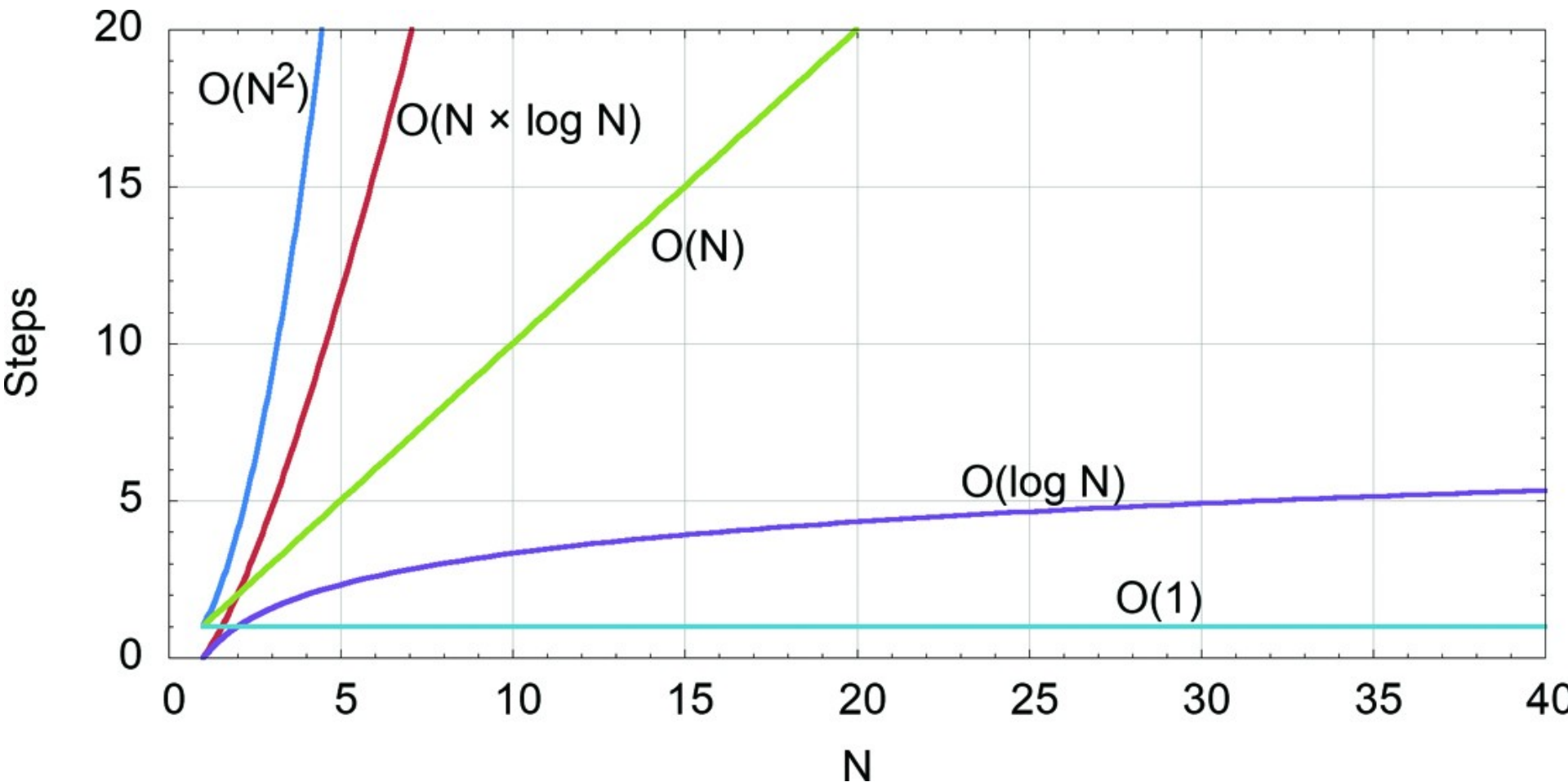
$O(n)$ também conhecido como tempo linear. Exemplo: Busca simples.

$O(n * \log n)$ Exemplo: Um algoritmo de ordenação rápido, como o quicksort

$O(n^2)$ Exemplo: Um algoritmo de ordenação lento, como o selection sort.

$O(n!)$ Exemplo: Um algoritmo realmente lento, como o do caixeiro-viajante

Notação Big O



Suponha que você esteja desenhando uma grade de 16 caixas novamente e possa escolher entre 5 algoritmos diferentes para isso.

Se usar o primeiro algoritmo, levará tempo **$O(\log n)$** para desenhar a grade. Você pode realizar 10 operações por segundo. Com tempo $O(\log n)$, levará 4 operações para desenhar uma grade de 16 caixas ($\log 16$ é 4). Portanto, levará 0,4 segundos para desenhar a grade. E se você tiver que desenhar 1.024 caixas? Levará $\log 1.024 = 10$ operações, ou 1 segundo, para desenhar uma grade de 1.024 caixas. Esses números estão usando o primeiro algoritmo.

O segundo algoritmo é mais lento: leva tempo **$O(n)$** . Serão necessárias 16 operações para desenhar 16 caixas e 1.024 operações para desenhar 1.024 caixas.

Veja quanto tempo levaria para desenhar uma grade para o resto dos algoritmos, do mais rápido ao mais lento:

Suponha que você esteja desenhando uma grade de 16 caixas novamente e possa escolher entre 5 algoritmos diferentes para isso.

Se usar o primeiro algoritmo, levará tempo **$O(\log n)$** para desenhar a grade. Você pode realizar 10 operações por segundo. Com tempo $O(\log n)$, levará 4 operações para desenhar uma grade de 16 caixas ($\log 16$ é 4). Portanto, levará 0,4 segundos para desenhar a grade. E se você tiver que desenhar 1.024 caixas? Levará $\log 1.024 = 10$ operações, ou 1 segundo, para desenhar uma grade de 1.024 caixas. Esses números estão usando o primeiro algoritmo.

O segundo algoritmo é mais lento: leva tempo **$O(n)$** . Serão necessárias 16 operações para desenhar 16 caixas e 1.024 operações para desenhar 1.024 caixas.

Veja quanto tempo levaria para desenhar uma grade para o resto dos algoritmos, do mais rápido ao mais lento:

Notação Big O



FAST



SLOW

# OF BOXES	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n!)$
16	0.4 sec	1.6 sec	6.4 sec	25.6 sec	6630 years
256	0.8 sec	25.6 sec	3.4 min	1.8 hrs	2.7×10^{498} years
1024	1.0 sec	1.7 min	17 min	1.2 days	1.7×10^{2631} years

Você pode ter lido a última seção e pensado: "Nunca vou encontrar um algoritmo que leve tempo $O(n!)$ ".

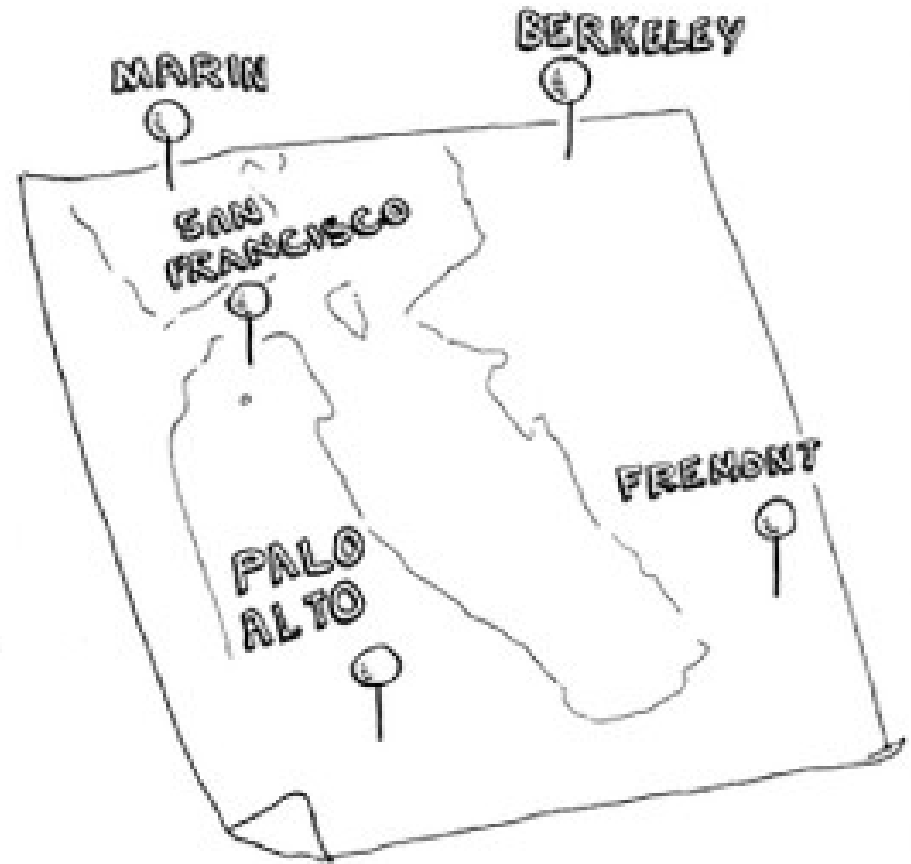
Bem, deixe-me tentar provar que você está errado! Aqui está um exemplo de um algoritmo com um tempo de execução realmente ruim.

Este é um problema famoso na ciência da computação, porque seu crescimento é assustador e algumas pessoas muito inteligentes acham que ele não pode ser melhorado.

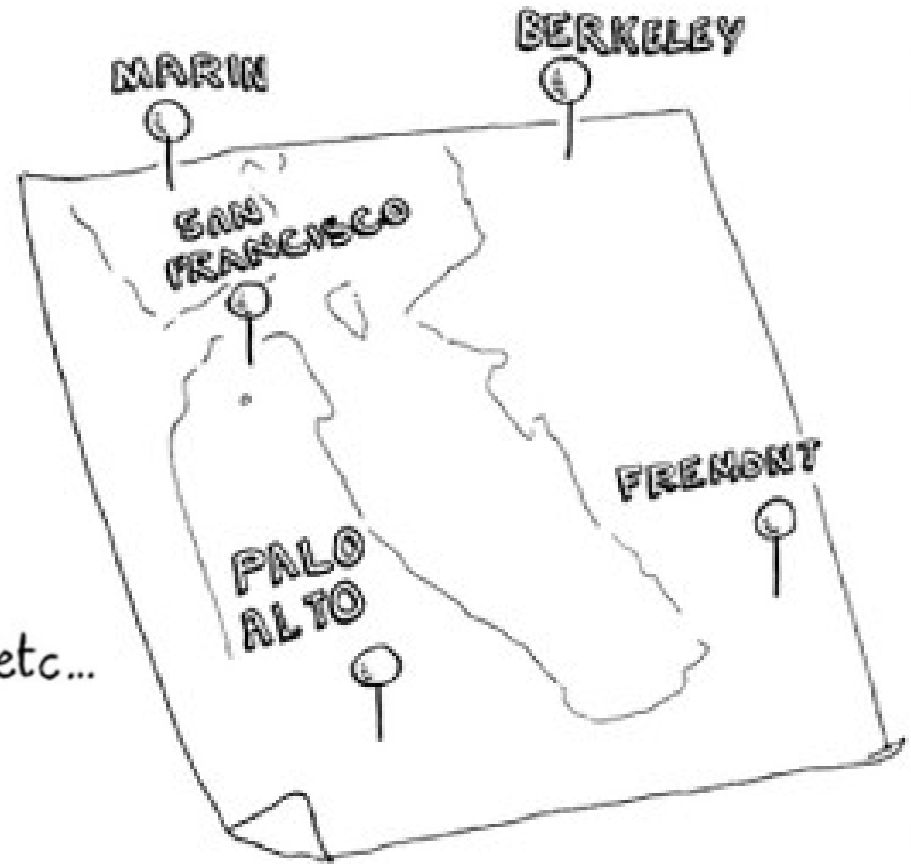
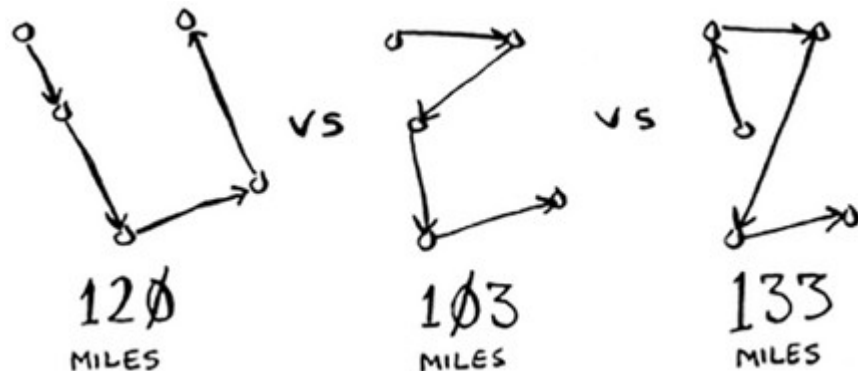
É chamado de problema do caixeiro viajante.

Você tem um vendedor. O vendedor precisa ir a cinco cidades.

Este vendedor, a quem chamarei de Opus, quer atingir todas as cinco cidades enquanto percorre a distância mínima.



Aqui está uma maneira de fazer isso: observe todas as ordens possíveis em que ele poderia viajar para as cidades.



Ele soma a distância total e, em seguida, escolhe o caminho com a menor distância. Há 120 permutações com 5 cidades, então serão necessárias 120 operações para resolver o problema para 5 cidades.

Para 6 cidades, serão necessárias 720 operações (há 720 permutações).

Para 7 cidades, serão necessárias 5.040 operações!

CITIES	OPERATIONS
6	720
7	5040
8	40320
...	...
15	1,307,674,368,000
...	...
30	2,652,528,598,121,91,058,636,308,480,000,000

Em geral, para n itens, serão necessárias $n!$ (n fatoriais) operações para calcular o resultado. Portanto, este é um tempo $O(n!)$, ou tempo fatorial.

São necessárias muitas operações para tudo, exceto para os menores números. Quando se lida com mais de 100 cidades, é impossível calcular a resposta a tempo .

Este é um algoritmo terrível! Opcionalmente deveria usar um diferente, certo? Mas ele não pode.

Este é um dos problemas não resolvidos na ciência da computação. Não existe um algoritmo rápido conhecido para ele, e pessoas inteligentes acham impossível ter um algoritmo inteligente para este problema. O melhor que podemos fazer é encontrar uma solução aproximada.

Síntese

A busca binária é muito mais rápida do que a busca simples.

$O(\log n)$ é mais rápido que $O(n)$, mas fica muito mais rápido à medida que a lista de itens pesquisados aumenta.

A velocidade do algoritmo não é medida em segundos.

Os tempos do algoritmo são medidos em termos de crescimento de um algoritmo.

Os tempos do algoritmo são escritos na notação Big O.

Dúvidas

Prof. Orlando Saraiva Júnior
orlando.nascimento@fatec.sp.gov.br

Prática



Acessar a pasta ***experimento*** .

Os códigos irão gerar e consumir N dados, com duas estruturas distintas: lista de tuplas e dicionários.

Qual apresentará melhor resultado nas buscas ?
Justifique sua resposta.





Lista de tuplas (list)

Precisa olhar elemento por elemento até achar o alvo. No pior caso (quando o nome está no final ou não existe), percorre os 1 milhão de elementos.

Complexidade: **$O(n)$** .

Dicionário (dict)

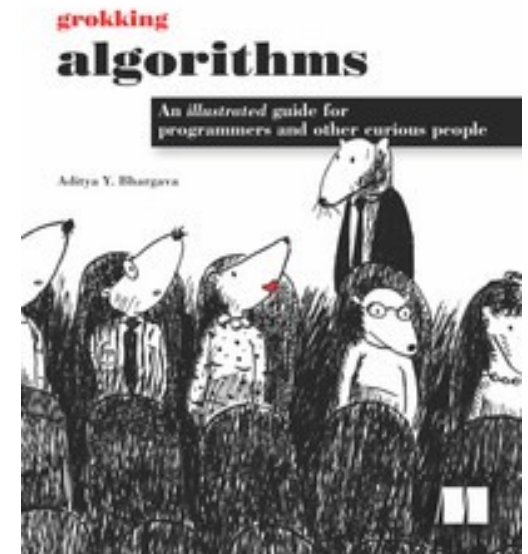
Usa uma tabela hash: o Python calcula um índice baseado na chave (nome). Recuperação é feita em tempo constante médio, independente do tamanho.

Complexidade média: **$O(1)$** .



Bibliografia

BHARGAVA, Aditya. Grokking algorithms: an illustrated guide for programmers and other curious people. Shelter Island, NY: Manning Publications, 2016.



Bibliografia

CANNING, John; BRODER, Alan; LAFORE, Robert. Data structures & algorithms in Python. Boston: Addison-Wesley Professional, 2019.

