

# Algoritmos & Estrutura de Dados

Entenda os conceitos básicos de algoritmos  
e estrutura de dados

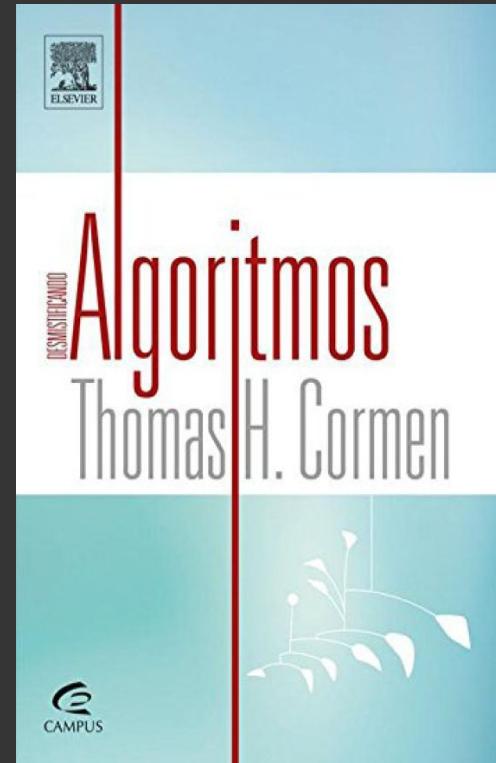
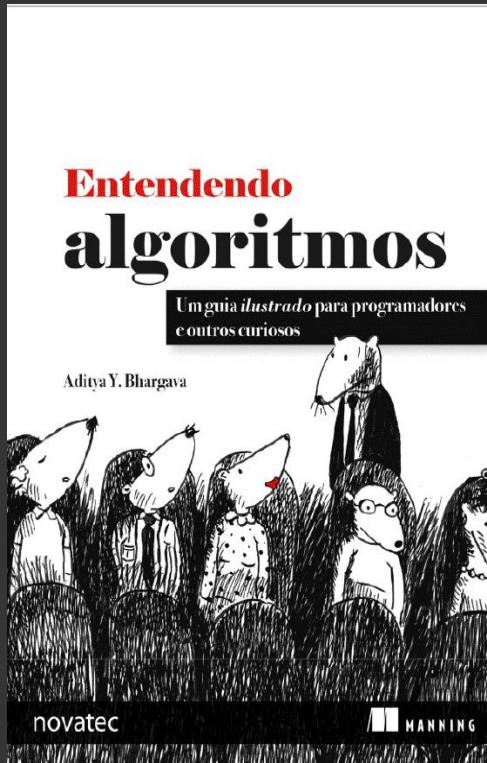
# Olá!

## Eu sou Willian Brito

- ❖ Desenvolvedor FullStack na Msystem Software
- ❖ Formado em Analise e Desenvolvimento de Sistemas
- ❖ Pós Graduado em Segurança Cibernética.
- ❖ Certificação SYCP (Solyd Certified Pentester) v2018



# Este conteúdo é baseado nessas obras



“Um **algoritmo** é um conjunto de instruções que realiza uma tarefa e **estrutura de dados** é a forma em que os dados são armazenados.”

# O que são dados ?

Os **dados** (e seus diversos tipos) são os blocos básicos da programação. Eles representam uma **unidade** ou um **elemento** de informação que pode ser Acessado através de um identificador - por exemplo, uma **variável**.

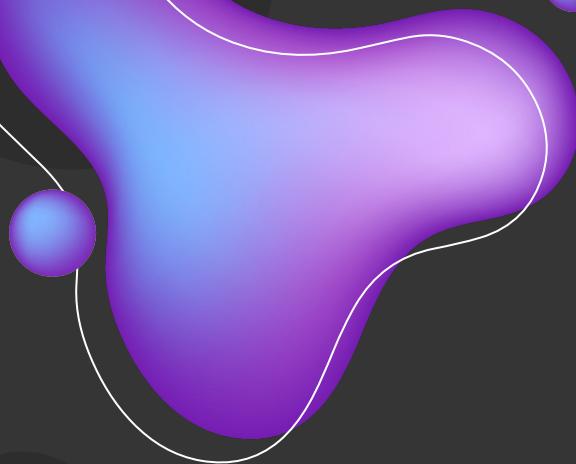
A maior parte das linguagens de programação trabalha com variações baseadas nos quatro tipos primitivos abaixo:

- **INT** ou número inteiro: valores numéricos inteiros (positivos ou negativos);
- **FLOAT** ou o chamado “ponto flutuante”: valores numéricos com casas após a vírgula (positivos ou negativos);
- **BOOLEAN** ou booleanos: representado apenas por dois valores, “verdadeiro” e “falso”. Também chamados de operadores lógicos;
- **STRING**: sequências ou cadeias de caracteres, utilizados para manipular textos e/ou outros tipos de dados não numéricos ou booleanos, como hashes de criptografia.

# Algoritmos & Estruturas de Dados

Em computação, normalmente utilizamos os **dados** de forma **conjunta**. A forma como estes dados serão agregados e organizados depende muito de como serão utilizados e processados, levando-se em consideração, por exemplo, a eficiência para buscas, o volume dos dados trabalhados, a complexidade da implementação e a forma como os dados se relacionam. Estas diversas formas de organização são as chamadas **estruturas de dados**.

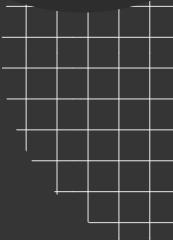
*“Podemos afirmar que um programa é composto de **algoritmos** e **estruturas de dados**, que juntos fazem com que o programa funcione como deve.”*



01

# Técnicas de Busca

Estratégias mais eficientes para busca em listas.



# Técnicas de Busca

Vamos supor que você esteja procurando o nome de uma pessoa em uma agenda telefônica. O nome começa com *K*. Você pode começar na primeira página da agenda e ir folheando até chegar aos *Ks*. Porém você provavelmente vai começar pela metade, pois sabe que os *Ks* 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 *kira*. O Facebook poderia começar pelos *As* e procurar seu nome, mas faz mais sentido que ele comece a busca pelo meio.

# Técnicas de Busca

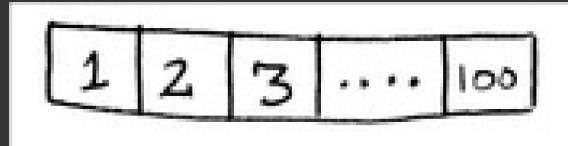
Isto é um problema de busca. E todos estes casos usam um algoritmo para resolvê-lo: **pesquisa binária**.

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 **null**.



# Técnicas de Busca

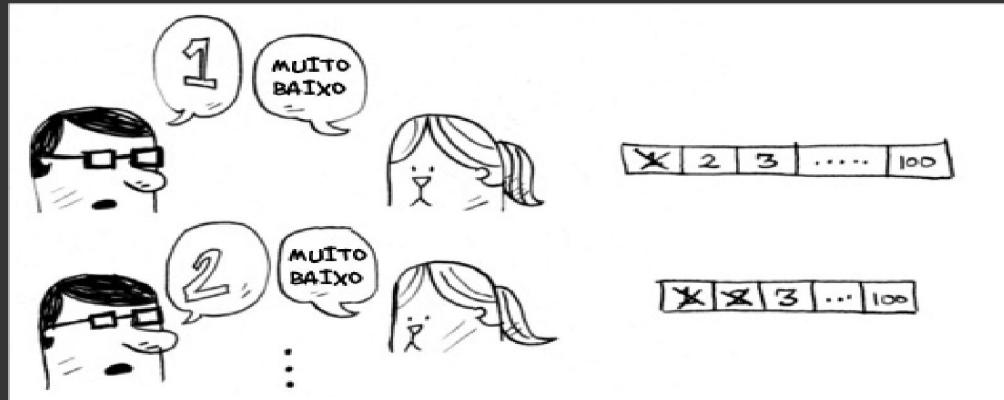
Eis um exemplo de como a pesquisa binária funciona. Estou pensando em um número entre 1 e 100.



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.

# Técnicas de Busca

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



# Técnicas de Busca

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.



# Técnicas de Busca

Muito baixo, mas você eliminou **metade** dos números! Agora, você sabe que os números de 1 a 50 são muito baixos. Próximo chute: 75.



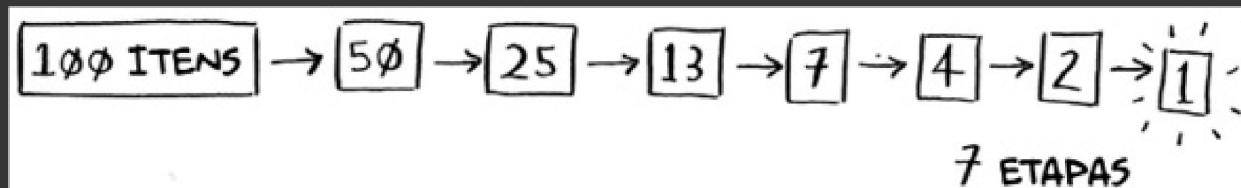
Muito alto, mas novamente você pode cortar metade dos números restantes!

*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).*

# Técnicas de Busca



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



# Técnicas de Busca

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!

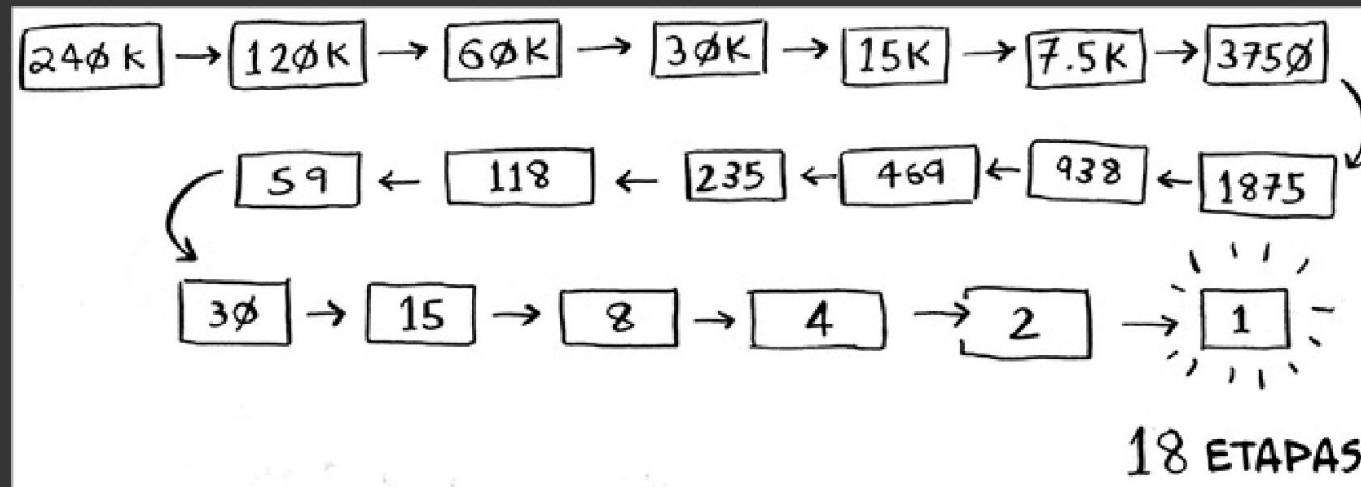
Suponha que você esteja procurando uma palavra em um dicionário. O dicionário tem 240.000 palavras. *Na pior das hipóteses*, de quantas etapas você acha que a pesquisa precisaria?

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.

# Técnicas de Busca

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.



# Técnicas de Busca

## □ 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:  $10 \times 10$ . Então,  $\log_{10} 100 = 2$ . Logaritmos são o oposto de exponenciais.

$$\begin{array}{rcl} 10^2 = 100 & \leftrightarrow & \log_{10} 100 = 2 \\ \hline 10^3 = 1000 & \leftrightarrow & \log_{10} 1000 = 3 \\ \hline 2^3 = 8 & \leftrightarrow & \log_2 8 = 3 \\ \hline 2^4 = 16 & \leftrightarrow & \log_2 16 = 4 \\ \hline 2^5 = 32 & \leftrightarrow & \log_2 32 = 5 \end{array}$$

# Técnicas de Busca

- **Logaritmos são o oposto de exponenciais.**

Neste conteúdo, quando falamos de 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  $2^{10} = 1.024$ . Logo, para uma lista de 1.024 números, precisaria verificar no máximo dez deles.

```
// # O(log n)
public int pesquisaBinaria(int valor)
{
    int limiteInferior = 0;
    int limiteSuperior = this.ultimaPosicao;

    while(true)
    {
        #region Variaveis Auxiliares

        int posicaoAtual = (int)(limiteInferior + limiteSuperior) / 2;
        int valorPosicaoAtual = this.valores[posicaoAtual];
        #endregion

        var estaNaPosicaoAtual = valorPosicaoAtual == valor;

        if(estaNaPosicaoAtual)
            return posicaoAtual;

        var naoEncontrouValor = limiteInferior > limiteSuperior;

        if(naoEncontrouValor)
            return -1;
        else
        {
            var valorAtualForMenorQueValorPesquisado = valorPosicaoAtual < valor;

            if(valorAtualForMenorQueValorPesquisado)
                limiteInferior = posicaoAtual + 1;
            else
                limiteSuperior = posicaoAtual - 1;
        }
    }
}
```

```
// # O(n)
public int pesquisaLinear(int valor)
{
    for (int i = 0; i <= this.ultimaPosicao + 1; i++)
    {
        if (this.valores[i] > valor)
            return -1;

        if (this.valores[i] == valor)
            return i;
    }

    return -1;
}
```

# 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 optimiza 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**.

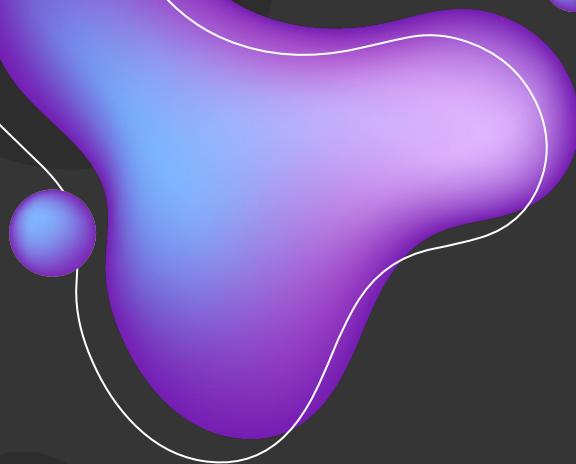
A pesquisa binária é diferente. Se a lista tem 100 itens, precisa-se de, no máximo, sete tentativas. Se tem 4 bilhões, precisa-se de, no máximo, 32 tentativas. Poderoso, não? A pesquisa binária é executada com **tempo logarítmico**. A tabela a seguir resume as nossas descobertas até agora.

# Tempo de Execução

- Comparação de **Tempo de Execução** entre uma **pesquisa simples  $O(n)$**  e uma **pesquisa binária  $O(\log n)$** .



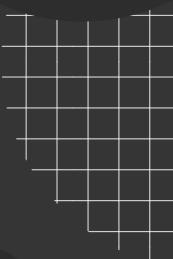
Faixa	Comparações Binária	Comparações Linear ( $N/2$ )
10	4	5
100	7	50
1.000	10	500
10.000	14	5.000
100.000	17	50.000
1.000.000	20	500.000
10.000.000	24	5.000.000
100.000.000	27	50.000.000
1.000.000.000	30	500.000.000



02

# Notação Big-O

Medindo eficiência de algoritmos independente de tecnologias



# Notação Big O

A notação **Big O** é uma notação especial que diz o quanto rápido e eficiente é um algoritmo independentemente da tecnologia e ambiente em que o algoritmo é implementado.

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 quanto 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.

# Notação Big O

## □ **Tempo de execução dos algoritmos cresce a taxas diferentes**

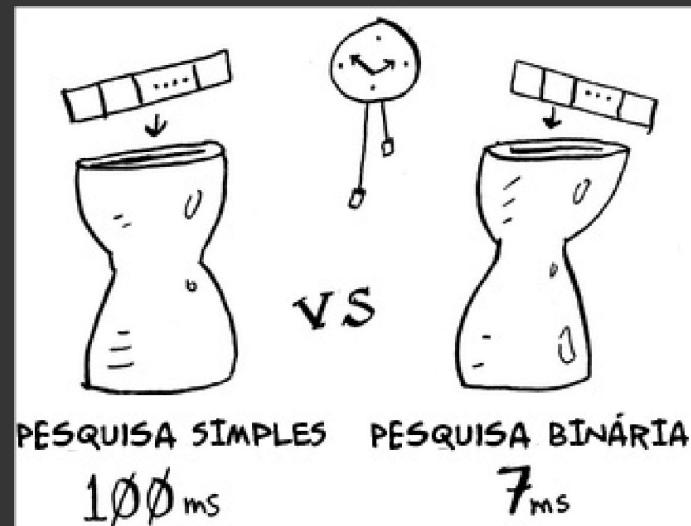
Bob está escrevendo um algoritmo para a NASA. O algoritmo dele entrará em ação quando o foguete estiver prestes a pousar na lua, e ele o ajudará a calcular o local de pouso.

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.

# Notação Big O

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?



# Notação Big O

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  $30 \times 15 = 450$  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
10000 ELEMENTOS	10 segundos
1,000,000,000 ELEMENTOS	11 dias

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

# Notação Big O

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.

# Notação Big O

## □ 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 Aline. 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 que você sabe, com certeza, que a pesquisa simples nunca terá tempo de execução mais lento do que  $O(n)$ .

# Notação Big O

## □ Alguns exemplos comuns de tempo de execução Big O

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

- **O(1) (constante):** É aquele em que não há crescimento do número de operações, pois *não depende do volume de dados de entrada (n)*. É o caso do acesso direto a um elemento de uma array, por exemplo.
- **O(log n) (logaritmo):** É aquele cujo crescimento do número de operações é menor do que o do número de itens. Um algoritmo logarítmico *reduz pela metade a entrada* toda vez que ele itera a lista. É o caso de algoritmos de busca em árvores binárias ordenadas (Binary Search Trees).

# Notação Big O

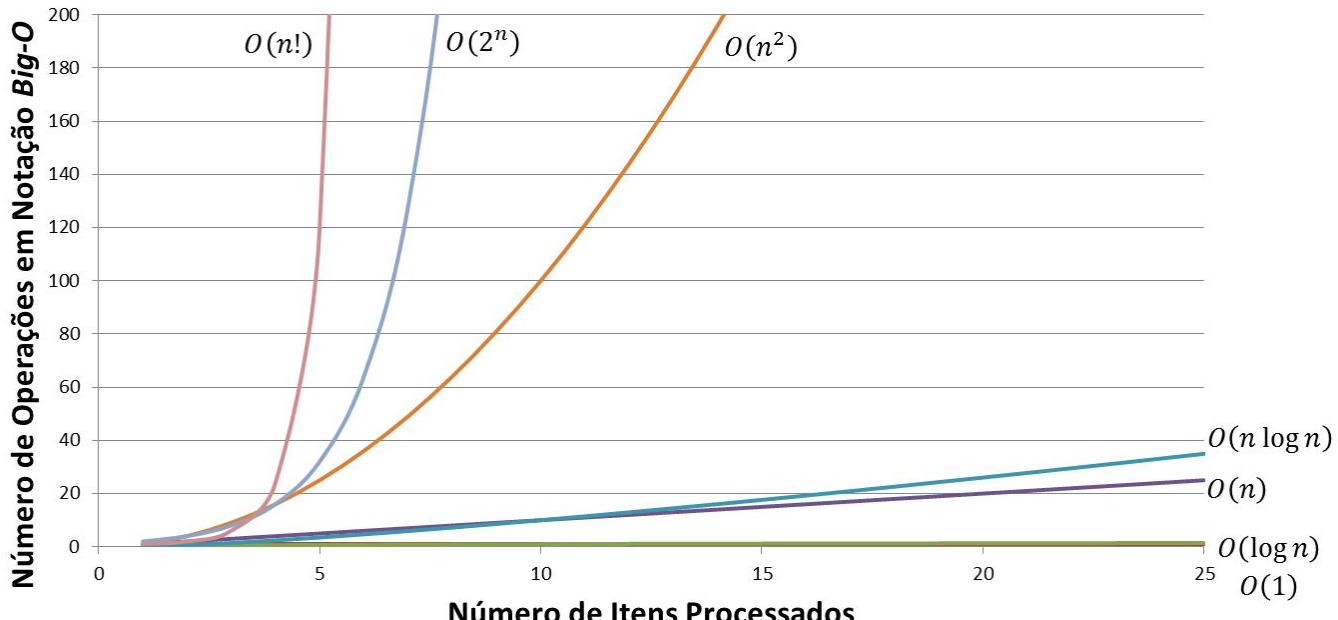
- **O(n) (linear):** É aquele cujo crescimento no número de operações é *diretamente proporcional ao crescimento do número de itens*. É o caso de algoritmos de busca em um array não ordenado, por exemplo.
- **O(n log n) (Linearitmica):** É melhor do que o quadrático, sendo geralmente até onde se consegue otimizar algoritmos que são quadráticos em sua *implementação mais direta*. É o caso do algoritmo de ordenação **QuickSort** e **MergeSort**, por exemplo (que tem essa complexidade no caso médio, mas que ainda assim é quadrático no pior caso).
- **O(n<sup>2</sup>) (quadrático):** É factível, mas tende a se tornar muito ruim quando a quantidade de dados é suficientemente grande. É o caso de algoritmos que têm dois laços (*for*) encadeados, como, por exemplo, o processamento de itens em uma matriz bidimensional.

# Notação Big O

- **O( $2^n$ ) (exponencial):** Também é bem ruim, pois o número de instruções também cresce muito rapidamente (exponencialmente), O tempo exponencial é  $2^n$ , em que 2 depende das **permutações envolvidas**. Um bom exemplo é o caso de algoritmos que fazem busca em árvores binárias não ordenadas, outro exemplo é no algoritmo de quebra de senha por força bruta, digamos que tenhamos uma senha composta apenas por números (10 números, de 0 a 9), queremos quebrar uma senha que tenha um comprimento de n. Com um algoritmo de força bruta em *todas as combinações*, teremos  $10^n$  combinações possíveis. Encontrar *todos os subconjuntos de um conjunto* também é um exemplo de tempo exponencial.
- **O(n!)** (fatorial): O número de instruções executadas cresce muito rapidamente para um pequeno crescimento do número de itens processados. Dentre os ilustrados é o pior comportamento para um algoritmo, pois rapidamente o processamento se torna inviável. É o caso da implementação inocente do Problema do **Caixeiro Viajante** ou de um algoritmo que gere *todas as possíveis permutações*.
- **OBS:** permutação são combinações de todos elementos com todos elementos

# Notação Big O

Ilustração das Complexidades Mais Comuns - Notação Big-O

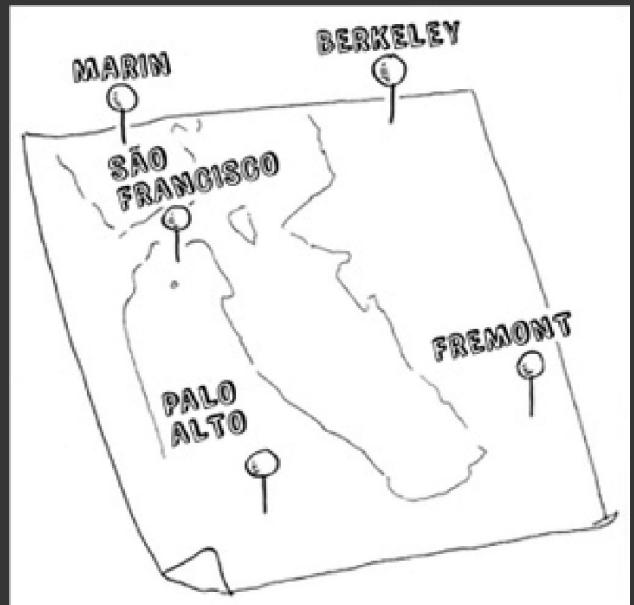


# Notação Big O

## □ O caixeiro-viajante

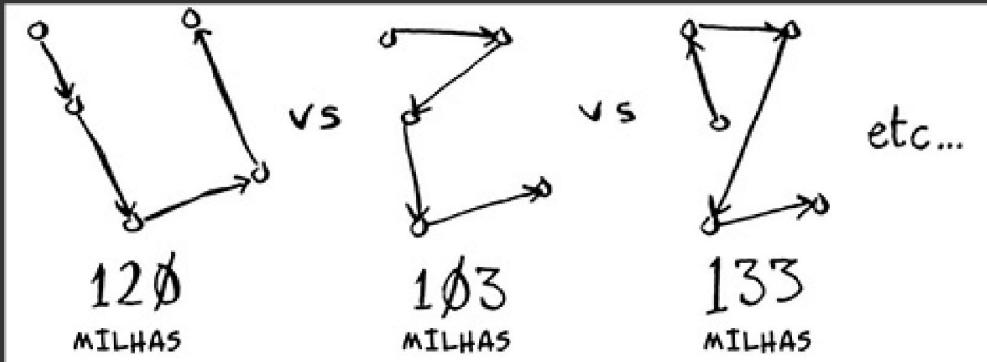
Você pode ter lido a última seção e pensado: “De maneira alguma vou executar um algoritmo que tem tempo de execução  $O(n!)$ .” Bem, deixe-me tentar provar o contrário! Aqui está um exemplo de um algoritmo com um tempo de execução muito ruim. Ele é um problema famoso da ciência da computação, pois seu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

Você tem um caixeiro-viajante. O caixeiro precisa ir a cinco cidades.



# Notação Big O

O caixeiro, o qual chamarei de Denis, está tentando descobrir a rota mais curta que o levará até as cinco cidades. Para encontrar a rota mais curta, primeiro devem-se calcular todas as rotas possíveis. Quantas rotas devem-se calcular para cinco cidades?



# Notação Big O

## □ O caixeiro-viajante, passo a passo

Vamos começar do básico. Suponha que você deseja visitar apenas duas cidades. Há duas rotas que você pode escolher.

## □ Mesma rota ou rota diferente?

Você pode pensar que essa deveria ser a mesma rota. Porque, no fim das contas, SF para Marin acaba tendo a mesma distância que Marin para SF, certo? Não necessariamente. Algumas cidades (como San Francisco) têm muitas ruas de sentido único, então você não pode voltar por onde veio. Ou seja, pode ser necessário seguir alguns quilômetros na direção errada para pegar o acesso a uma rodovia. Logo, duas rotas não são necessariamente a mesma coisa.



# Notação Big O

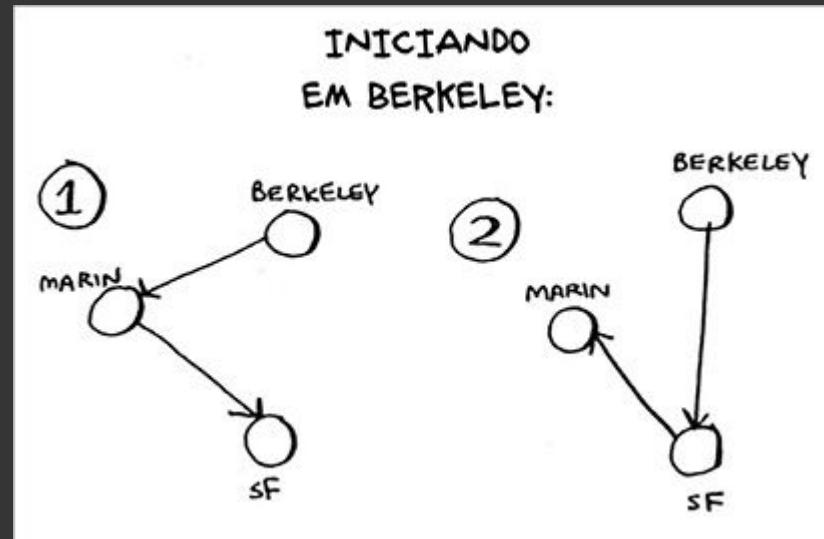
Você deve estar pensando “No problema do caixeiro-viajante, existe uma cidade específica de onde devo partir?”. Vamos dizer, por exemplo, que sou o caixeiro-viajante e que vivo em San Francisco e preciso ir para quatro cidades. San Francisco seria minha cidade de partida.

Porém, às vezes, a cidade de partida não está definida. Suponha que você seja o FedEx (serviço postal americano) tentando entregar um pacote em Bay Area (área da Baía de San Francisco). Esse pacote está vindo de Chicago para uma das cinquenta unidades da FedEx em Bay Area. Logo depois, o pacote será transportado em um caminhão que viajará para diferentes locais fazendo as entregas. Quando vindo de Chicago, para qual unidade em San Francisco o pacote deve ser enviado? Aqui o local de partida é desconhecido. Cabe a você calcular o caminho ideal e o local de partida para o caixeiro-viajante. O tempo de execução das duas versões é o mesmo, mas o exemplo ficará mais fácil se não houver uma cidade de partida, então vou usar esta versão. Duas cidades = duas rotas possíveis.

# Notação Big O

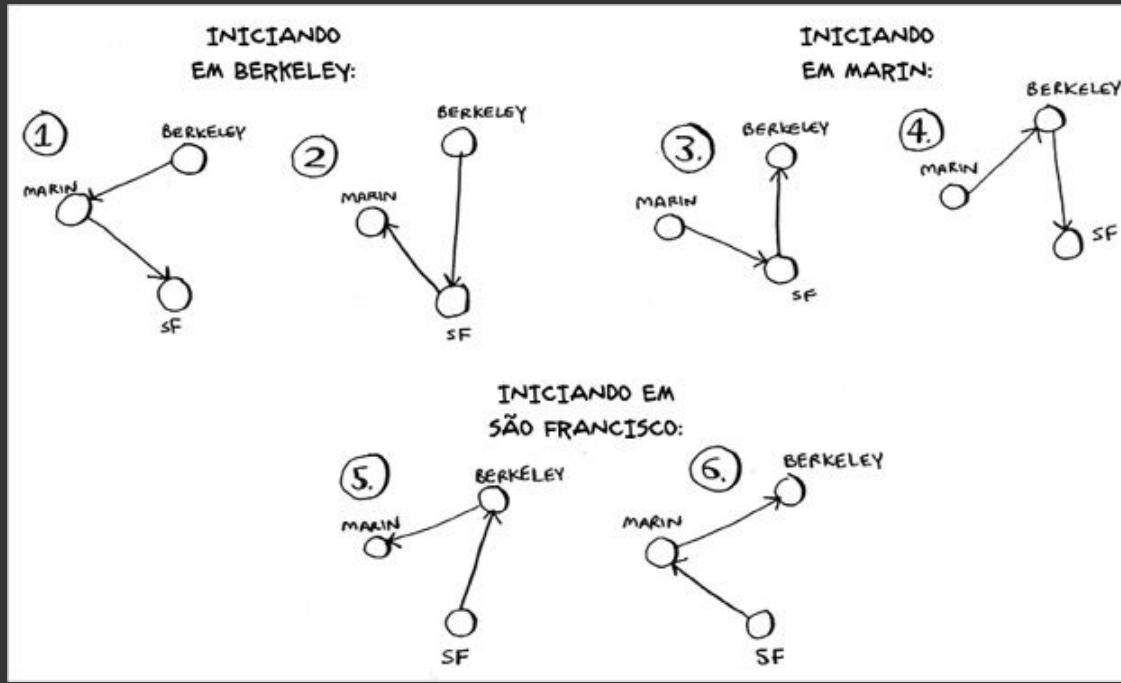
## □ Três cidades

Agora suponha que você tenha adicionado mais uma cidade. Quantas rotas existem? Se você começar em Berkeley, ainda deverá visitar mais duas cidades.



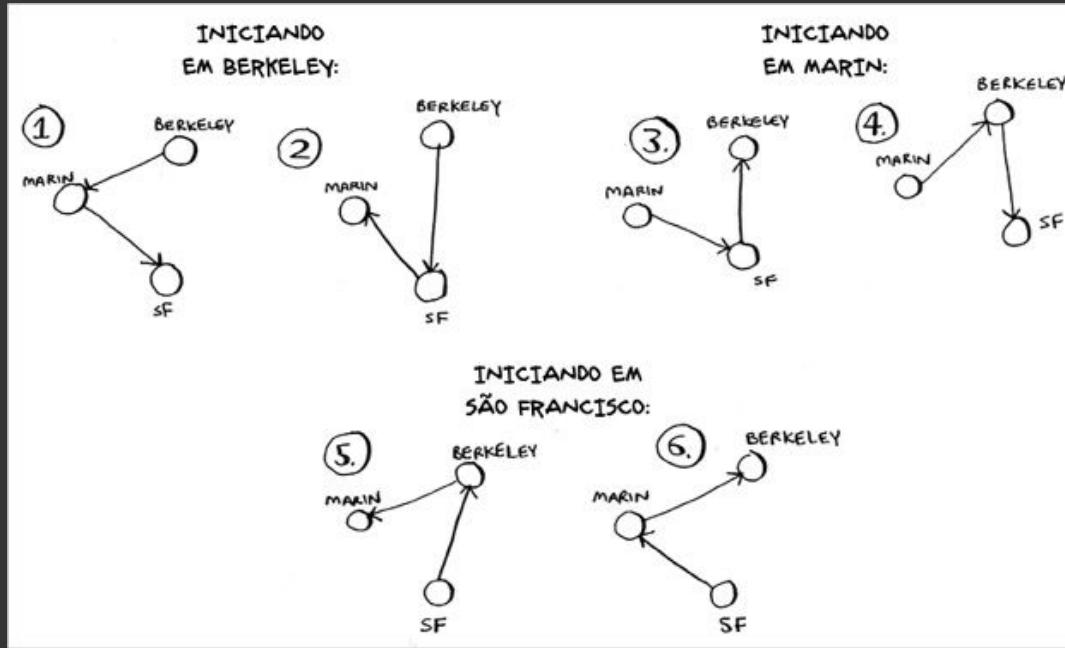
# Notação Big O

Há um total de seis rotas, duas para cada cidade em que pode começar.



# Notação Big O

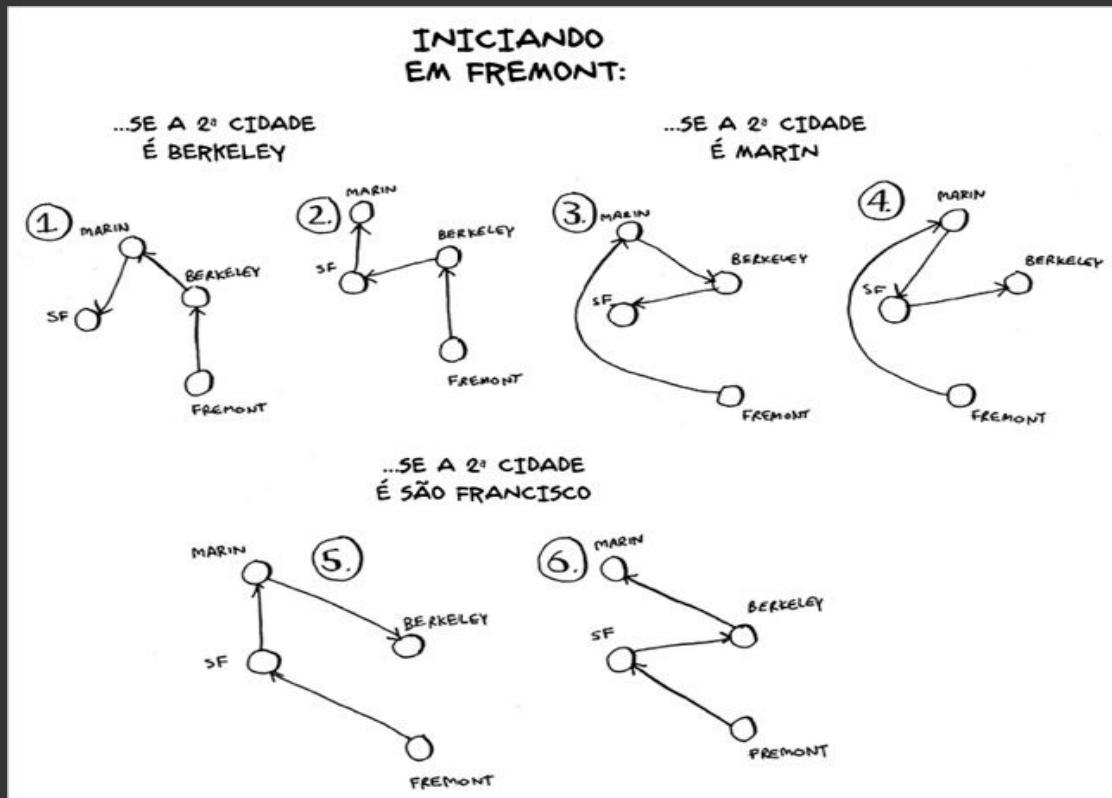
Há um total de seis rotas, duas para cada cidade em que pode começar. Então três cidades = seis rotas possíveis.



# Notação Big O

## □ Quatro cidades

Vamos adicionar outra cidade: Fremont. Suponha que você inicie lá.



# Notação Big O

Há seis rotas possíveis partindo de Fremont, e olhe só! Elas se parecem muito com as seis rotas que você calculou anteriormente, quando tinha apenas três cidades. Com exceção de que agora todas as rotas têm uma cidade adicional: Fremont!

Há um padrão aqui, e para visualizá-lo, suponha que existam quatro cidades e que você possa escolher a cidade de partida. Você escolhe Fremont. Há três cidades sobrando, e se há três cidades, existem seis rotas diferentes para trafegar entre elas. Caso você inicie em Fremont, existem seis rotas possíveis. Também pode-se iniciar em uma das outras cidades.

# Notação Big O

Quatro cidades de partida possíveis, com seis rotas possíveis para cada cidade de partida =  $4 * 6 = 24$  rotas possíveis.

Percebe o padrão? Cada vez que uma cidade é adicionada, o número de rotas que devem ser calculadas aumentam.

INICIANDO  
EM MARIN:

= 6 ROTAS POSSÍVEIS

INICIANDO EM  
SÃO FRANCISCO:

= 6 ROTAS POSSÍVEIS

INICIANDO  
EM BERKELEY:

= 6 ROTAS POSSÍVEIS

# Notação Big O

NÚMERO DE CIDADES		
1	→ 1 ROTA	
2	→ 2 CIDADES INICIAIS *	1 ROTA PARA CADA INÍCIO = 2 ROTAS AO TOTAL
3	→ 3 CIDADES INICIAIS *	2 ROTAS = 6 ROTAS AO TOTAL
4	→ 4 CIDADES INICIAIS *	6 ROTAS = 24 ROTAS AO TOTAL
5	→ 5 CIDADES INICIAIS *	24 ROTAS = 120 ROTAS AO TOTAL

# Notação Big O

Quantas rotas possíveis existem para seis cidades? Se você disse 720, está certo. Além disso, existem 5.040 rotas para sete cidades e 40.320 para oito cidades.

Isso é chamado de **função fatorial** (Falamos dela anteriormente). Então  **$5! = 120$** . Suponha que você tenha dez cidades. Quantas rotas possíveis existem?  **$10! = 3.628.800$** . Devem-se calcular cerca de 3 milhões de rotas possíveis para dez cidades. Como você pode notar, o número de rotas possíveis cresce rapidamente!

É por isso que é impossível calcular a solução “correta” para o problema do caixeiro-viajante caso o número de cidades seja muito elevado.

# Notação Big O

- Quantidade de operações para mais de 5 cidades

CIDADES	OPERAÇÕES
6	$72\phi$
7	$5\phi 4\phi$
8	$4\phi 32\phi$
...	...
15	$13\phi 7,674368\phi\phi\phi$
...	...
30	$2,6525,2859,812,191,058636,3084,80,000,000$

# Notação Big O

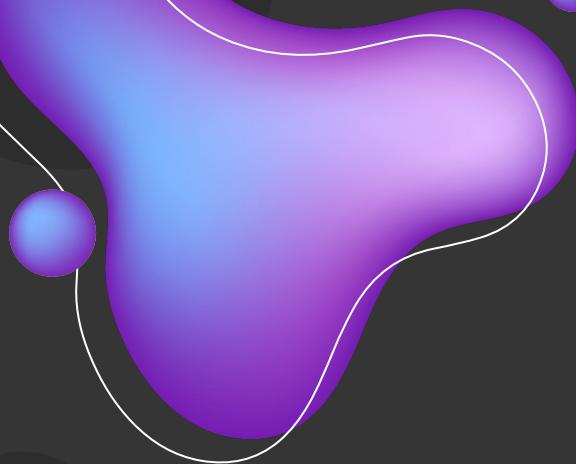
De maneira geral, para  $n$  itens, é necessário  $n!$  (fatorial de  $n$ ) operações para chegar a um resultado. Então, este é o tempo de execução  $O(n!)$  ou o *tempo factorial*. Esse algoritmo consome muitas operações, exceto para casos envolvendo números pequenos. No entanto, uma vez que lidamos 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! Denis 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.

# Notação Big O

## □ Concluindo

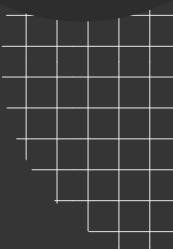
- A pesquisa binária é muito mais rápida do que a pesquisa simples.
- $O(\log n)$  é mais rápido do que  $O(n)$ , e  $O(\log n)$  fica ainda mais rápido conforme os elementos da lista aumentam.
- A rapidez de um algoritmo não é medida em segundos.
- O tempo de execução de um algoritmo é medido por meio de seu crescimento.
- O tempo de execução dos algoritmos é expresso na notação Big O.
- Sempre buscar algoritmos de aproximação(veremos nas últimas seções) para problemas mais complexos como o caixeiro-viajante.



03

# Array

Entenda o funcionamento de vetores



# Como funciona a memória

Imagine que você vai a um show e precisa guardar as suas coisas na chancelaria. Algumas gavetas estão disponíveis. Cada gaveta pode guardar um elemento, você deseja guardar duas coisas, então pede duas gavetas.



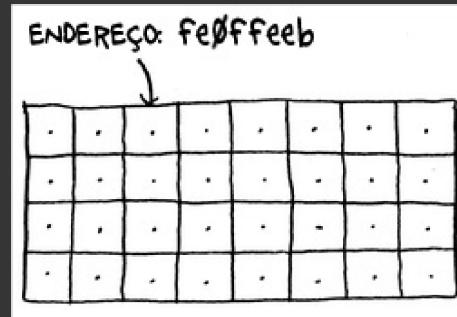
# Como funciona a memória

Você está pronto para o show! É mais ou menos assim que a memória do seu computador funciona.

O computador se parece com um grande conjunto de gavetas, e cada gaveta tem seu endereço. **feØffeb** é o **endereço** de um slot na memória.

Cada vez que quer armazenar um item na memória, você pede ao computador um pouco de espaço e ele te dá um endereço no qual você pode armazenar o seu item. Se quiser armazenar múltiplos itens, existem duas maneiras para fazer isso: **arrays** e **listas**.

Falarei sobre arrays e listas depois, bem como sobre os prós e contras de cada um. Não existe apenas uma maneira correta para armazenar itens em cada um dos casos, então é importante saber as diferenças.



# Array

Algumas vezes, você precisa armazenar uma lista de elementos na memória. Suponha que você esteja escrevendo um aplicativo para gerenciar os seus afazeres. É necessário armazenar os seus afazeres como uma lista na memória.

Você deve usar um array ou uma lista encadeada? Vamos armazenar os afazeres primeiro em um array, pois assim a compreensão fica mais fácil.

Usar um array significa que todas as suas tarefas estão armazenadas contiguamente (uma ao lado da outra) na memória.



# Array

Agora, suponha que você queira adicionar mais uma tarefa. No entanto a próxima gaveta está ocupada por coisas de outra pessoa!

É como se você estivesse indo ao cinema com os seus amigos e encontrasse um lugar para sentar, mas outro amigo se juntasse a vocês e não houvesse lugar para ele. Vocês todos precisariam se mover e encontrar um lugar onde todos coubessem. Neste caso, você precisaria solicitar ao computador uma área de memória em que coubessem todas as suas tarefas. Então você as moveria para lá.

NÃO É POSSÍVEL ADICIONAR UM AFAZER AQUI POIS ESTE ESPAÇO JÁ ESTÁ OCUPADO

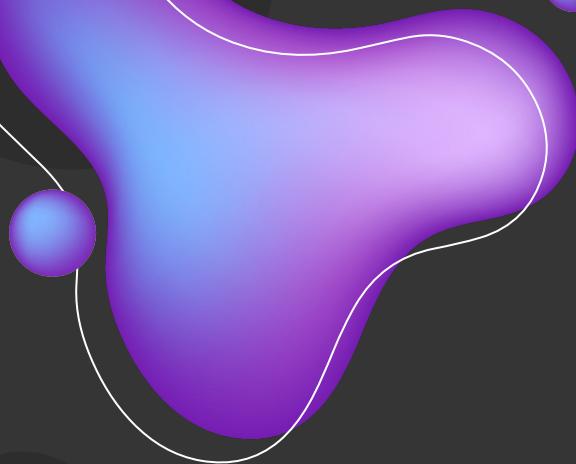
CAFÉ DA MANHÃ	JOGAR BOCHA	CHÁ	

# Array

Se outro amigo aparecesse, vocês ficariam sem lugar novamente e todos precisariam se mover uma segunda vez! Que incômodo. Da mesma forma, adicionar novos itens a um array será muito lento. Uma maneira fácil de resolver isso é “reservando lugares”: mesmo que você tenha três itens na sua lista de tarefas, você pode solicitar ao computador dez espaços, só por via das dúvidas. Então, você pode adicionar dez itens na sua lista sem precisar mover nada. Isto é uma boa maneira de contornar o problema, mas você precisa ficar atento às desvantagens:

- ❑ Você pode não precisar dos espaços extras que reservou; então a memória será desperdiçada. Você não está utilizando a memória, mas ninguém mais pode usá-la também.
- ❑ Você pode precisar adicionar mais de dez itens a sua lista de tarefas, então você terá de mover seus itens de qualquer maneira.

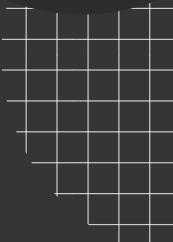
Embora seja uma boa forma de contornar o problema, não é uma solução perfeita. Listas encadeadas resolvem este problema de adição de itens.



04

# Listas Encadeadas

Corrigindo algumas deficiências dos vetores

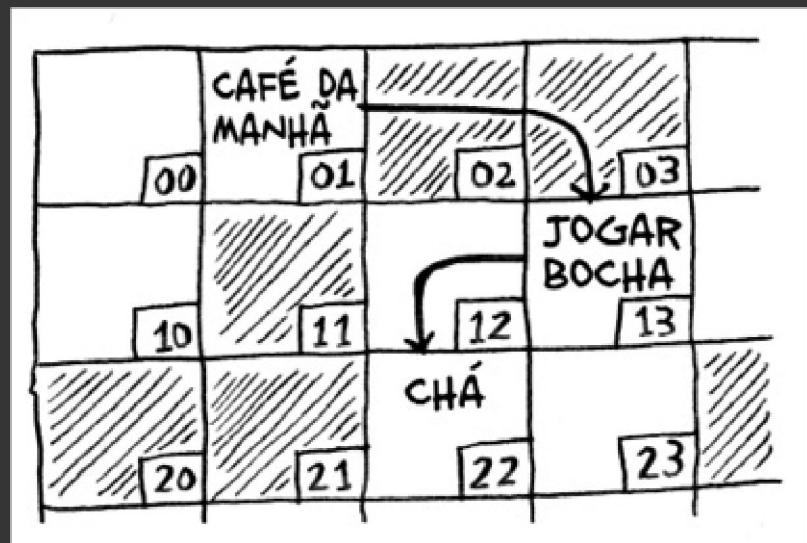


# Listas Encadeadas

Com as listas encadeadas, seus itens podem estar em qualquer lugar da memória.

Cada item armazena o endereço do próximo item da lista. Um monte de endereços aleatórios de memória estão ligados.

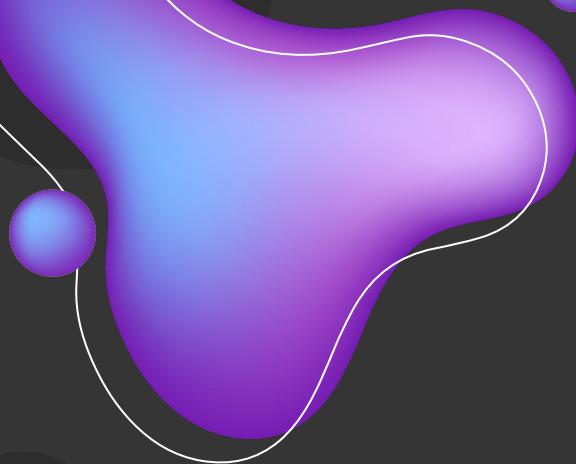
É como uma caça ao tesouro. Você vai ao primeiro endereço e ele diz “o próximo item pode ser encontrado no endereço 123”. Então vai ao endereço 123 e ele diz “O próximo item pode ser encontrado no endereço 847”, e assim por diante. Adicionar um item a uma lista encadeada é fácil: você o coloca em qualquer lugar da memória e armazena o endereço do item anterior.



# Listas Encadeadas

Com as listas encadeadas você nunca precisa mover os seus itens, também evita outro problema. Digamos que você vá a um cinema famoso com os seus amigos. Você seis estão tentando procurar um lugar para sentar, mas o cinema está cheio. Não há seis lugares juntos. Bem, algumas vezes isso acontece com arrays. Imagine que está tentando encontrar 10.000 slots para um array. Sua memória tem 10.000 slots, mas eles não estão juntos.

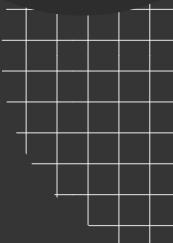
Você não consegue arrumar um lugar para o seu array! Usar uma lista encadeada seria como dizer “vamos nos dividir e assistir ao filme”. Se existir espaço na memória, você terá espaço para a sua lista encadeada. Se as listas encadeadas são muito melhores para inserções, para que servem os arrays?



# 05

# Listas Vs Arrays

Vantagens e desvantagens de cada uma das estruturas



# Listas Vs Arrays

Os websites que apresentam listas “top 10” usam uma tática trapaceira para conseguir mais visualizações. Em vez de mostrarem a lista em uma única página, eles colocam um item em cada página e fazem você clicar em “próximo” para ler o item seguinte. Por exemplo, “Os 10 melhores vilões dos quadrinhos” não estarão listados em uma única página, em vez disso, você começará pelo #10 (Lex Luthor) e seguirá clicando em “próximo” até chegar em #1(Coringa).

Esta técnica fornece aos sites dez páginas inteiras para incluir anúncios, mas fica chato ficar clicando em “próximo” nove vezes até chegar ao número 1. Seria muito melhor se a lista estivesse em uma única página e você pudesse clicar no nome de cada vilão para saber mais.

Listas encadeadas têm um problema similar. Suponha que você queira ler o último item de uma lista encadeada. Você não pode fazer isso porque não sabe o endereço dele. Em vez disso, precisa ir ao item #1 para pegar o endereço do item #2. Então, é necessário ir ao item #2 para encontrar o endereço do item #3, e assim por diante, até conseguir o endereço do último item.

# Listas Vs Arrays

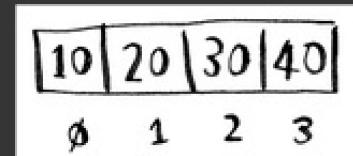
Listas encadeadas são ótimas se você quiser ler todos os itens, um de cada vez: você pode ler um item, seguir para o endereço do próximo item e fazer isso até o fim da lista. Mas se você quiser pular de um item para outro, as listas encadeadas são terríveis. Com arrays é diferente. Você sabe o endereço de cada item. Por exemplo, suponha que seu array tenha cinco itens e que você saiba que o primeiro está no endereço 00. Qual é o endereço do item #5?

A matemática lhe dá a resposta: está no endereço 04. Arrays são ótimos se você deseja ler elementos aleatórios, pois pode encontrar qualquer elemento instantaneamente em um array. Na lista encadeada, os elementos não estão próximos uns dos outros, então você não pode calcular instantaneamente a posição de um elemento na memória precisa ir ao primeiro elemento para encontrar o endereço do segundo, então ir ao segundo elemento para encontrar o endereço do terceiro e seguir fazendo isso até chegar ao elemento que deseja.

# Listas Vs Arrays

Os elementos em um array são numerados. Essa numeração começa no 0, não no 1. Neste array, por exemplo, o número 20 está na posição 1. O número 10 está na posição 0. Isso geralmente confunde novos programadores. Começar no 0 simplifica todos os tipos de array na programação, logo, os programadores não podem fugir disso. quase todas as linguagens de programação começarão os arrays numerando o primeiro elemento como 0. Logo você se acostuma!

A posição de um elemento é chamada de **índice**. Portanto, em vez de dizer “o número 20 está na posição 1”, a terminologia correta seria dizer “o número 20 está no **índice 1**”. Usarei índice para falar de posição neste conteúdo. Aqui está o tempo de execução para operações comuns de arrays e listas.

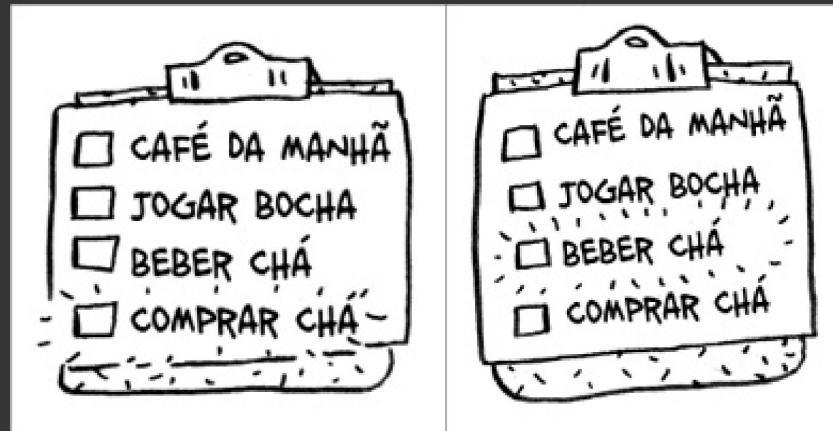


	ARRAYS	LISTAS
LEITURA	O(1)	O(n)
INSERÇÃO	O(n)	O(1)
$O(n) = $ TEMPO DE EXECUÇÃO LINEAR		
$O(1) = $ TEMPO DE EXECUÇÃO CONSTANTE		

# Listas Vs Arrays

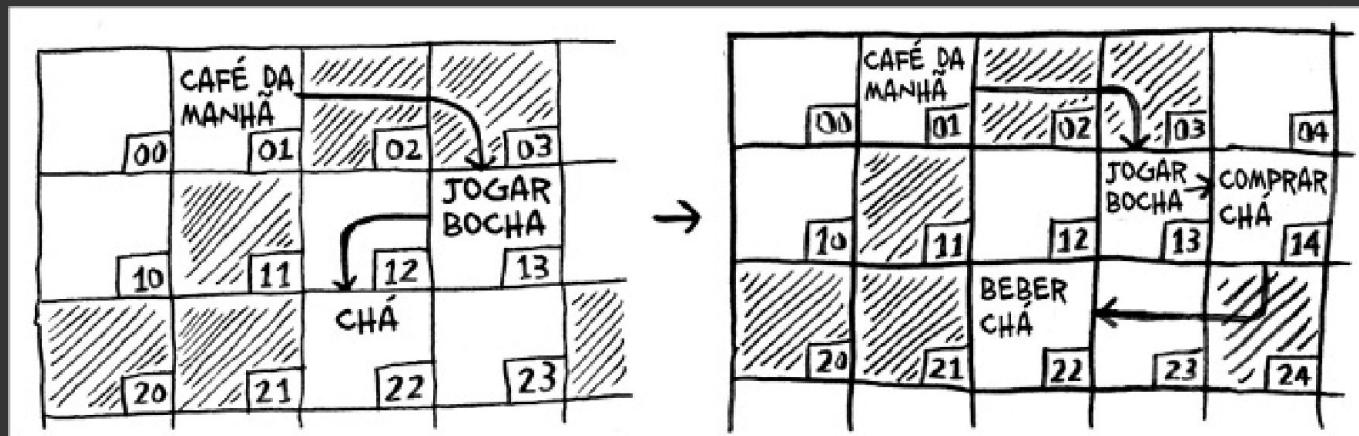
## □ Inserindo algo no meio da lista

Imagine que você queira que a sua lista de tarefas se pareça mais com um calendário. Antes, você adicionava os itens ao final da lista. Agora, quer adicionar suas tarefas na ordem em que elas devem ser realizadas.



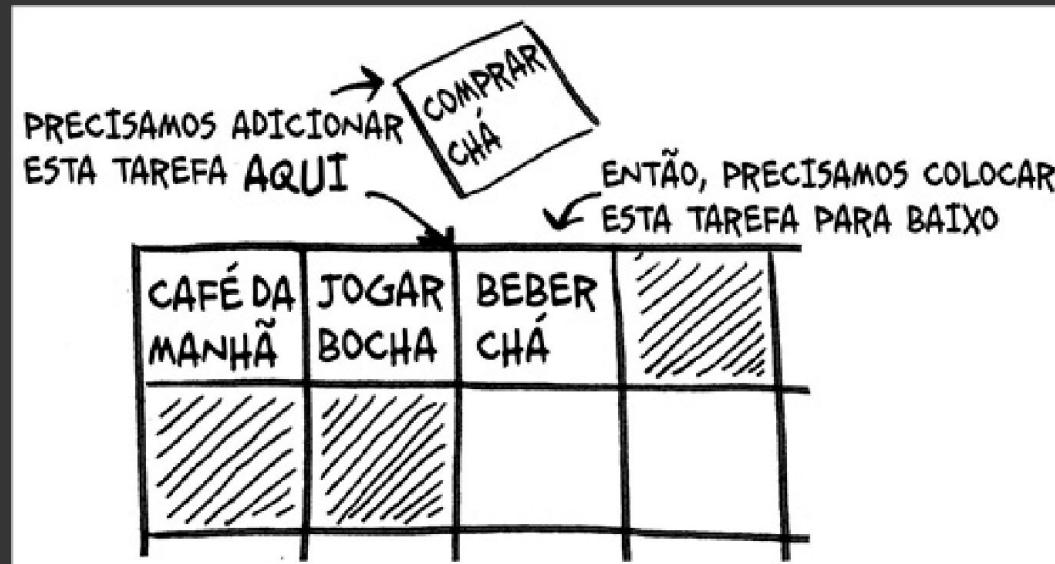
# Listas Vs Arrays

O que seria melhor se você quisesse inserir elementos no meio de uma lista: arrays ou listas encadeadas? Usando listas encadeadas, basta mudar o endereço para o qual o elemento anterior está apontando.



# Listas Vs Arrays

Já para arrays, você deve mover todos os itens que estão abaixo do endereço de inserção. Se não houver espaço, pode ser necessário mover tudo para um novo local! Por isso, listas encadeadas são melhores caso você queira inserir um elemento no meio de uma lista.



# Listas Vs Arrays

## □ Remover um elemento

E se você quiser deletar um elemento? Novamente, é mais fácil fazer isso usando listas encadeadas, pois é necessário mudar apenas o endereço para o qual o elemento anterior está apontando. Com arrays, tudo precisa ser movido quando um elemento é eliminado.

Ao contrário do que ocorre com as inserções, a eliminação de elementos sempre funcionará. A inserção poderá falhar quando não houver espaço suficiente na memória.

Aqui estão os tempos de execução para as operações mais comuns em arrays e listas encadeadas.

	ARRAYS	LISTAS
LEITURA	$O(1)$	$O(n)$
INSERÇÃO	$O(n)$	$O(1)$
ELIMINAÇÃO	$O(n)$	$O(1)$

# Listas Vs Arrays

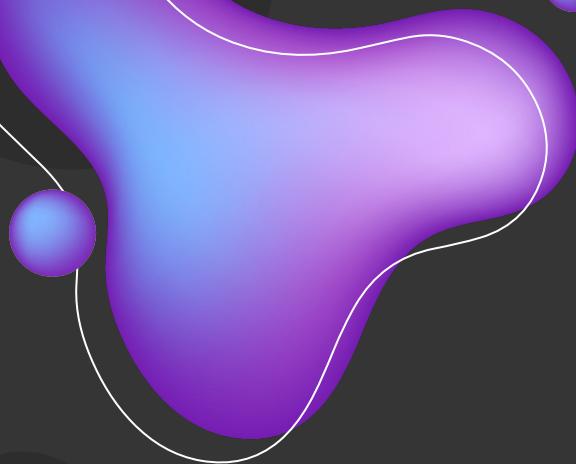
Vale a pena mencionar que inserções e eliminações terão tempo de execução  $O(1)$  somente se você puder acessar instantaneamente o elemento a ser deletado. É uma prática comum acompanhar o primeiro e último item de uma lista encadeada para que o tempo de execução para deletá-los seja  $O(1)$ .

O que é mais usado: arrays ou listas? Obviamente, isso depende do caso em que se aplicam. Entretanto, os arrays são mais comuns porque permitem acesso aleatório. Existem dois tipos de acesso: o *aleatório* e o *sequencial*. O sequencial significa ler os elementos, um por um, começando pelo primeiro. Listas encadeadas só podem lidar com acesso sequencial. Se você quiser ler o décimo elemento de uma lista encadeada, primeiro precisará ler os nove elementos anteriores para chegar ao endereço do décimo elemento. O aleatório permite que você pule direto para o décimo elemento. Muitos casos requerem o acesso aleatório, o que faz os arrays serem bastante utilizados, porém a utilização dessas estruturas muitas vezes vai dar preferência ao desenvolvedor.

# Listas Vs Arrays

## □ Concluindo

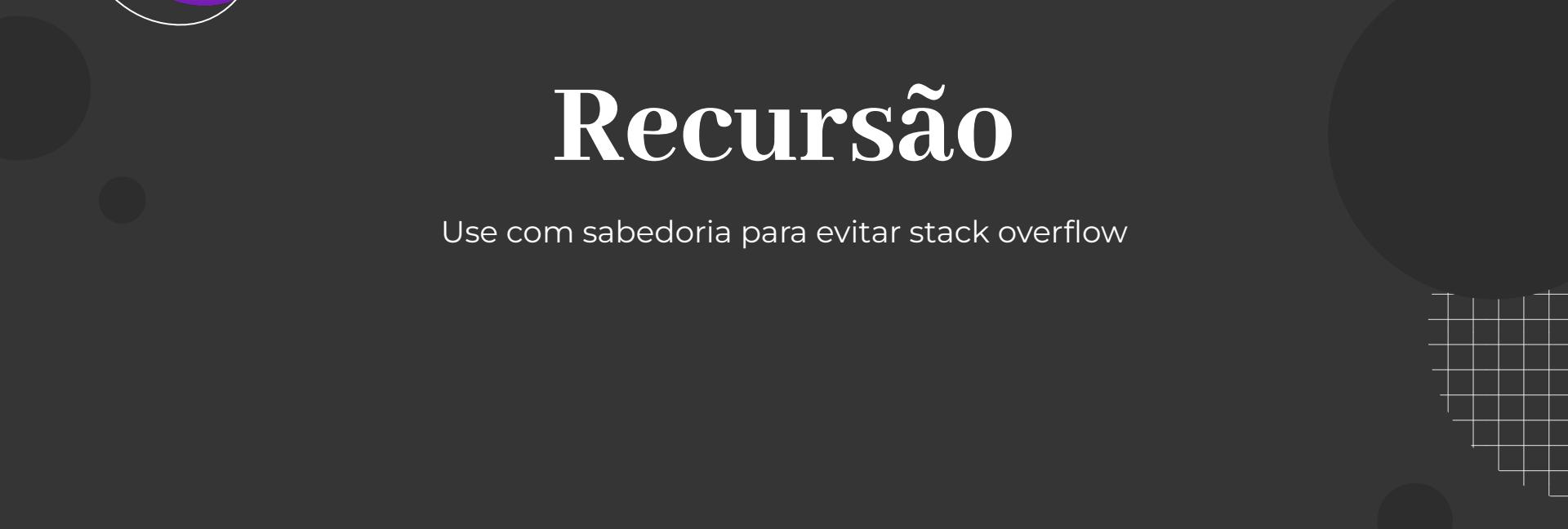
- A memória do seu computador é como um conjunto gigante de gavetas.
- Quando se quer armazenar múltiplos elementos, usa-se um array ou uma lista.
- No array, todos os elementos são armazenados um ao lado do outro.
- Na lista, os elementos estão espalhados e um elemento armazena o endereço do próximo elemento.
- Arrays permitem leituras rápidas para acesso aleatório.
- Listas encadeadas permitem rápidas inserções e eliminações.
- Todos os elementos de um array devem ser do mesmo tipo (todos ints, todos doubles, e assim por diante).



06

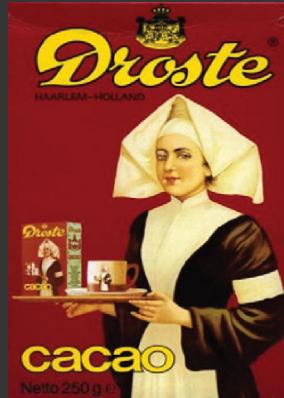
# Recursão

Use com sabedoria para evitar stack overflow



# Recursão

Provavelmente, já deve ter acontecido isto com você antes: dormindo, você sonha que está dormindo e sonhando. Em artes, esse fenômeno é conhecido como *efeito Droste* ou *miseenabyme*. Uma imagem que aparece dentro dela mesma, em um local semelhante ao da primeira imagem. Uma versão menor, que contém uma versão menor que a outra, que contém outra menor ainda, e assim por diante. O efeito Droste tem esse nome por causa da imagem que ilustrava as caixas de cacau em pó Droste, uma das principais marcas holandesas: uma enfermeira carregando uma bandeja com uma xícara de chocolate quente e uma caixa do produto.



# Recursão

Esse efeito nos remete à ideia de repetição. Mas não qualquer repetição, e sim a **repetição de um objeto dentro dele mesmo**, numa ideia de *looping* ou laço contínuo. A esse processo dá-se o nome de **recursão**, ou seja, um objeto é parcialmente definido em termos dele mesmo.

A recursão pode ser encontrada na matemática, nas ciências, na computação e no cotidiano. Experimente colocar um objeto entre dois espelhos e você terá uma ideia prática da recursão infinita (ou finita, até onde sua visão lhe permitir).

## □ Atenção

A implementação de um algoritmo recursivo em uma linguagem de programação, partindo de uma definição matemática também recursiva, é praticamente direta e imediata. Por essa razão, esse tipo de algoritmo recursivo possui código muito mais legível e compacto.

# Recursão

Uma das mais elegantes (e, em muitos casos, complexas) técnicas da matemática é a recursividade. Ela pode ser caracterizada quando se define um objeto em termos dele mesmo. O grande potencial da recursão está na possibilidade de poder definir elementos com base em versões mais simples desses mesmos elementos. Em termos computacionais, trata-se de dividir um problema maior em problemas menores, em que a resolução é feita por uma mesma função (ou método), que é recorrentemente chamada. Muitos autores chamam essa técnica computacional de “dividir para conquistar”, parafraseando o grande imperador francês Napoleão Bonaparte.

- **Para implementar uma função (ou método) recursivo, é necessário estabelecer pelo menos dois elementos:**
  - Uma **condição de parada** ou terminação. Geralmente, essa condição estabelece uma solução trivial ou um evento que encerra a auto chamada consecutiva;
  - Uma **mudança de estado a cada chamada**, ou seja, o estabelecimento de alguma diferença entre o estado inicial e o próximo estado da função (ou método). Isso pode ser feito, por exemplo, decrementando um parâmetro da função recursiva.

# Recursão

Um exemplo clássico que se utiliza para exemplificar a recursão computacional é o cálculo do fatorial de um número  $n$ .

Apenas para aquecer, você conseguiria calcular o fatorial de um número sem utilizar a técnica de recursão? Recordando: o fatorial de um número  $n$  é igual à multiplicação dos números inteiros de 1 até  $n$ , ou seja,  $1 * 2 * 3 * 4 * \dots * n$ . Dessa forma, se  $n$  for igual a 4, teríamos que o fatorial de 4 é igual a 24 ou  $1 * 2 * 3 * 4 = 24$ .

Matematicamente, teríamos que o fatorial de um inteiro positivo  $n$ , denotado  $n!$ , é definido como o produto dos inteiros de 1 até  $n$ . Se  $n = 0$ , então  $n!$ , é definido como 1 por convenção.

# Recursão

Como você pôde ver, dependendo do valor do parâmetro o valor do fatorial é calculado e retornado pela função.

Como seria a implementação dessa mesma função, mas de forma recursiva? A primeira coisa a fazer é identificar o caso base ou situação de parada. No caso do cálculo do fatorial, a condição de parada é valor = 0. Isso resulta em fatorial igual a 1. Os casos recursivos são sempre a multiplicação de um valor pelo próximo valor de  $n$ , com mudança de estado de seu valor decrementado de 1. Abaixo, temos uma forma de implementar o cálculo fatorial de forma recursiva.

```
public static int factorialComRecursoao(int valor)
{
    if (valor == 0)
        return 1;

    return valor * factorialComRecursoao(valor - 1);
}
```

```
public static string factorialSemRecursoao(int valor)
{
    var fatorial = 1;

    for (int i = 1; i <= valor; i++)
        fatorial *= i;

    return fatorial;
}
```

# Recursão

Para melhor o entendimento do funcionamento dessa função recursiva, vamos fazer um teste de rastreamento das recursões, supondo a chamada da função fatorial, com um parâmetro inicial igual a 4.

Para que a análise da execução (teste de rastreamento) fique mais clara, é conveniente tratar cada chamada da função fatorial() como uma nova instância da função.

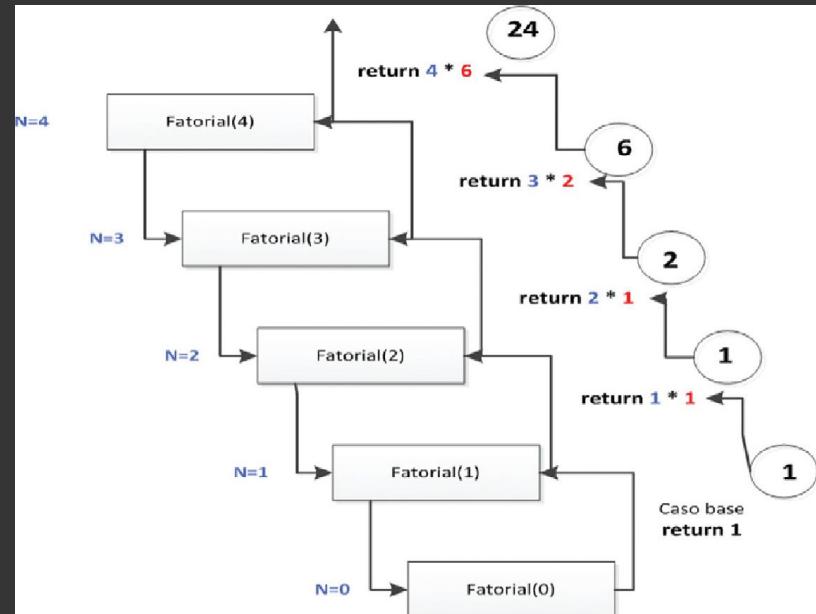
## □ Conceito

Uma instância de uma função corresponde ao processo de alocação de um novo espaço na memória (pilha) para comportar as necessidades de utilização de variáveis locais inerentes àquela função.

# Recursão

- Na tabela abaixo, a seguir, observe os valores passados pelas chamadas e retornos.

Linha	Instância	Explicação	Valor n	Passagem de parâmetro	Retorno
L1	1	n não é $\leq 1$	4		
L3	1	Chama <b>fatorial()</b>	4	3	
L1	2	n não é $\leq 1$	3		
L3	2	Chama <b>fatorial()</b>	3	2	
L1	3	n não é $\leq 1$	2		
L3	3	Chama <b>fatorial()</b>	2	1	
L1	4	n é $\leq 1$	1		
L2	4	Retorna 1	1		1
L3	3	Retorna $2 * 1$	2		2
L3	2	Retorna $3 * 2$	3		6
L3	1	Retorna $4 * 6$	4		24



# Recursão

É importante ressaltar que no caso do cálculo do fatorial de um número  $n$  utilizando recursão, observe que, para cada chamada da função fatorial, um novo espaço é alocado para empilhar essa nova função. Por isso, se o valor  $n$  for muito elevado, poderá ocorrer um **estouro da pilha (stack overflow)**, resultando em erro de execução.

Diante disso, você pode, nesse momento, estar se perguntando: qual é a vantagem de utilizar uma função recursiva em comparação com uma não recursiva?

Embora a implementação recursiva, na maioria dos casos, seja mais simples que a versão iterativa, não existe nenhuma razão determinante para preferir a versão recursiva à iterativa. Em muitos casos, as versões recursivas **consomem maior número de recursos** (principalmente memória e processamento) e são muito mais difíceis de testar quando há muitas chamadas. Entretanto, o que pode ser considerado positivo em sua utilização é a obtenção de códigos mais “**enxutos**” e mais **fáceis de compreender**, e, consequentemente, mais fáceis de implementar em linguagens de programação.

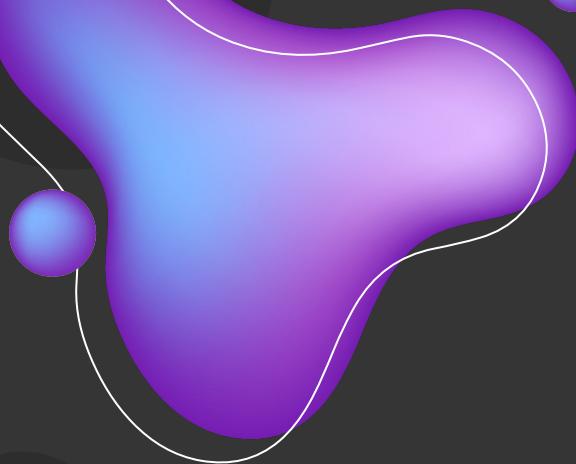
# Recursão

## □ Você deve utilizar a recursão quando:

- O problema é naturalmente recursivo (clareza) e a versão recursiva do algoritmo não gera ineficiência evidente, se comparada com a versão iterativa;
- O algoritmo se torna compacto, sem perda de clareza ou generalidade;
- É possível prever que o número de chamadas (e, consequentemente, a alocação na pilha) não vai provocar interrupção no processo.

## □ Você NÃO deve utilizar a recursão quando:

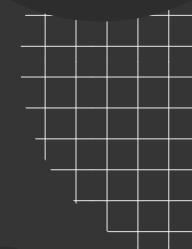
- A solução recursiva causa ineficiência, se comparada com a versão iterativa;
- O uso de recursão acarreta número maior de cálculos que a versão iterativa;
- Parâmetros consideravelmente grandes têm que ser passados por valor;
- Não é possível prever o número de chamadas que podem causar sobrecarga na pilha.



07

# Pilha

Último a entrar é o primeiro a sair



# Pilha

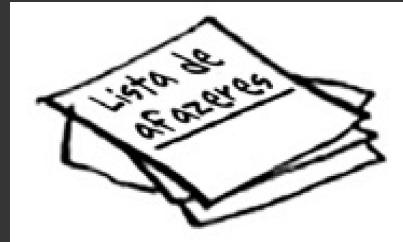
As **Pilhas** são estruturas de dados do tipo **LIFO (Last-In First-Out) em português (*Último-A-Entrar-Primeiro-A-Sair*)**, portanto nessa estrutura o último elemento a ser inserido, será o primeiro a ser retirado. Assim, uma pilha permite acesso a apenas um item de dados que é o último inserido. Para processar o penúltimo item inserido, deve-se remover o último.

A pilha é *um* conceito *muito* importante em programação e indispensável para entender a recursão que utiliza uma técnica que se chama **pilha de chamada (call stack)**.

# Pilha

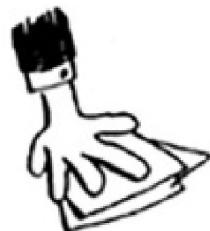
Suponha que você esteja fazendo um churrasco para os seus amigos. Você tem uma lista de afazeres em forma de uma pilha de notas adesivas.

Você se lembra de que, quando falamos de arrays e listas, também havia uma lista de afazeres? Podia adicionar itens em qualquer lugar da lista ou remover itens aleatórios. A pilha de notas adesivas é bem mais simples.

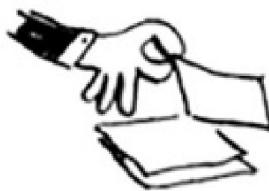


# Pilha

Quando você insere um item, ele é colocado no topo da pilha. Quando você lê um item, lê apenas o item do topo da pilha e ele é retirado da lista. Logo, sua lista de afazeres contém apenas duas ações: **empilhar** e **desempilhar**.



**PUSH**  
(ADICIONE UM NOVO  
ITEM AO TOPO)



**POP**  
(REMOVA O ITEM  
DO TOPO E LEIA-O)

# Pilha

Vamos ver como isso funciona na prática. A pilha é uma estrutura de dados simples. Você a tem usado esse tempo todo sem perceber!

- **A pilha de chamada**

Seu computador usa uma pilha interna denominada *pilha de chamada*. Vamos ver isto na prática. Aqui está um exemplo simples:

```
private void sauda(string nome)
{
    System.Console.WriteLine($"Olá, {nome} !");
    sauda2(nome);
    System.Console.WriteLine("preparando para dizer tchau...");
    tchau();
}
```

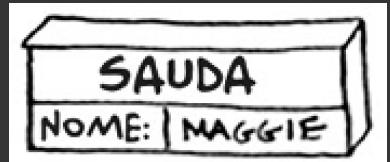
- **Esta função te cumprimenta e chama outras duas funções:**

```
private void sauda2(string nome)
{
    System.Console.WriteLine($"Como vai, {nome} ?");
}

private void tchau()
{
    System.Console.WriteLine("ok, tchau!");
}
```

# Pilha

Suponha que você chame sauda("maggie"). Primeiro, seu computador aloca uma caixa de memória para essa chamada. Agora, vamos usar a memória. A variável nome é setada para "maggie". Isso precisa ser salvo.



Cada vez que você faz uma chamada de função, seu computador salva na memória os valores para todas as variáveis. Depois disso, imprime olá, maggie!. Então, chama sauda2("maggie").



# Pilha

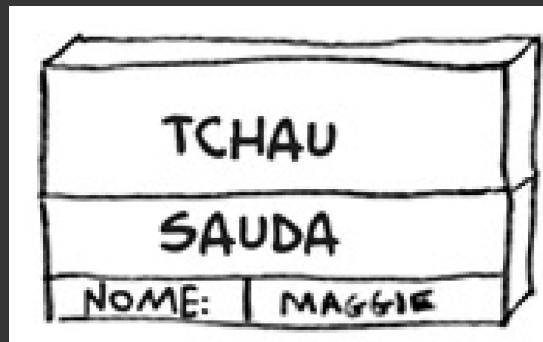
Novamente, seu computador aloca uma caixa de memória para essa chamada de função.

Seu computador está usando uma pilha para estas caixas. A segunda caixa é adicionada em cima da primeira. Você imprime "como vai maggie?". Então, retorna da chamada de função. Quando isso acontece, a caixa do topo da pilha é retirada.



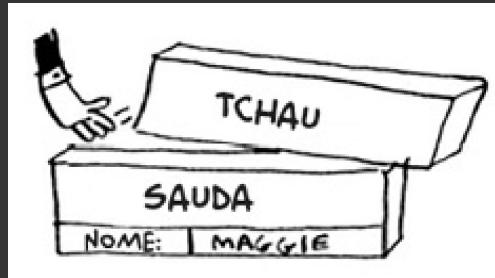
# Pilha

Agora, a caixa do topo da pilha aloca os valores da função sauda, o que significa que você retornou à função sauda. Quando você chamou a função sauda2, a função sauda ficou **parcialmente completa**. Esta é a grande ideia por trás desta seção: **quando você chama uma função a partir de outra, a chamada de função fica pausada em um estado parcialmente completo**. Todos os valores das variáveis para aquela função ainda estão armazenados na memória. Agora que você já utilizou a função sauda2, você está de volta na função sauda e pode continuar de onde parou. Primeiro, imprime "preparando para dizer tchau..." e então chama a função tchau.



# Pilha

Uma caixa para esta função é adicionada ao topo da pilha. Quando você imprimir ok, tchau!, retornará da chamada de função.



Agora, você está de volta à função sauda. Não há nada mais a ser feito, e você pode sair da função sauda também. Essa pilha usada para guardar as variáveis de múltiplas funções é denominada pilha de chamada.

# Pilha

## □ A pilha de chamada com recursão

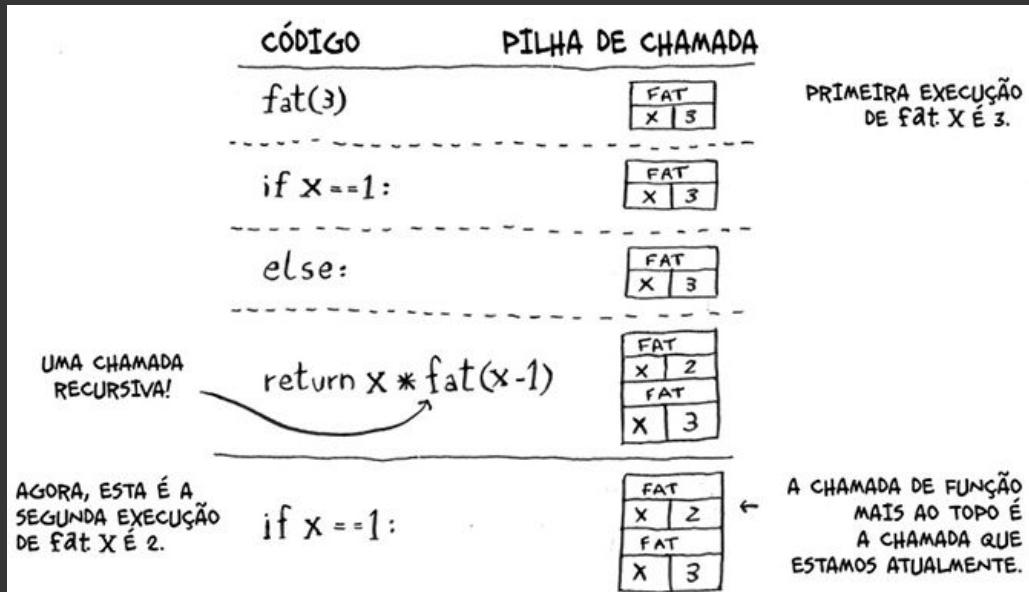
As funções recursivas também utilizam a pilha de chamada! Vamos analisar isto na prática com a função fatorial vista na sessão que falamos de recursão. `fat(5)` é escrita como  $5!$  e é definida da seguinte forma:  $5! = 5 * 4 * 3 * 2 * 1$ . De forma semelhante, `fat(3)` é  $3 * 2 * 1$ . Aqui está uma função recursiva para calcular a fatorial de um número:

```
public static int fat(int x)
{
    if (x == 0)
        return 1;

    return x * factorialComRecurso(x - 1);
}
```

# Pilha

Agora, você chama a função `fat(3)`. Vamos analisar esta pilha de chamada linha por linha e ver como ela se altera. Lembre-se, a caixa mais próxima ao topo lhe diz em qual chamada a função `fat` se encontra atualmente.



else:

return  $x * \text{fat}(x-1)$

if  $x == 1$ :

NOSSA, NÓS FIZEMOS  
TRÊS CHAMADAS À  
FUNÇÃO fat, MAS  
NÓS NÃO HAVÍAMOS  
FINALIZADO NENHUMA  
CHAMADA ATÉ AGORA!

return 1

FAT
X   2
FAT
X   3

FAT
X   1
FAT
X   2
FAT
X   3

FAT
X   1
FAT
X   2
FAT
X   3

FAT
X   1
FAT
X   2
FAT
X   3

PERCEBA QUE AMBAS AS  
CHAMADAS DE FUNÇÕES  
POSSUEM UMA VARIÁVEL  
X, MAS O VALOR DA  
VARIÁVEL X É DIFERENTE  
EM CADA UMA.

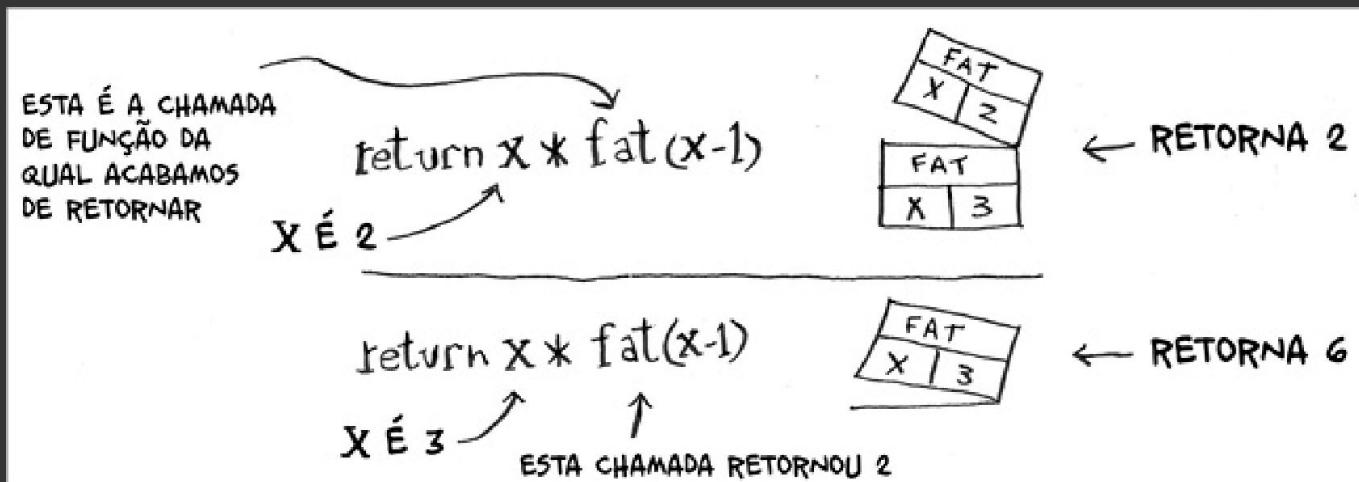
VOCÊ NÃO CONSEGUE  
ACESSAR ESTA VARIÁVEL  
X A PARTIR DESTA  
CHAMADA DE FUNÇÃO  
E VICE E VERSA.

ESTE É O PRIMEIRO ITEM A  
SER RETIRADO DA PILHA,  
O QUE SIGNIFICA QUE  
ESTA É A PRIMEIRA  
CHAMADA DA QUAL  
NÓS RETORNAMOS.

RETORNA 1

# Pilha

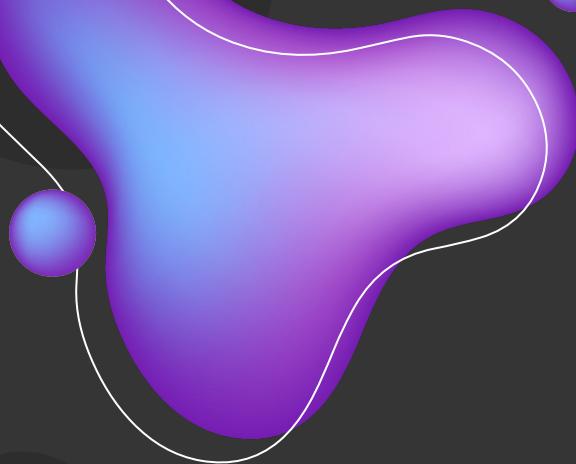
Repare que cada chamada para a função fat tem seu próprio valor de x. Você não consegue acessar a mesma função com outro valor de x.



# Pilhas

## □ Concluindo

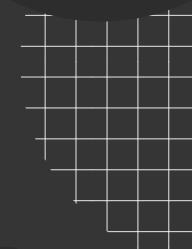
- Pilhas são estruturas de dados do tipo LIFO (Last-In First-Out) em português (Último-A-Entrar-Primeiro-A-Sair)
- Uma pilha tem duas operações: empilhar e desempilhar.
- Todas as chamadas de função vão para a pilha de chamada.
- A pilha de chamada pode ficar muito grande e ocupar muita memória.



08

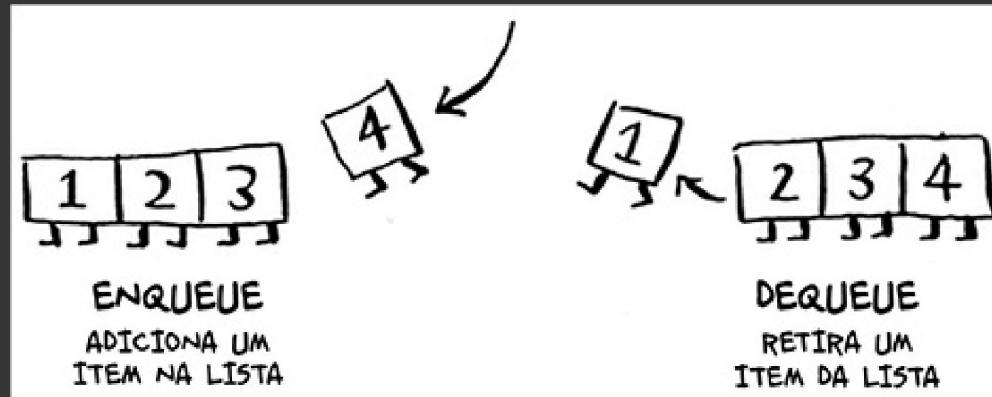
# Fila

Primeiro a entrar é o primeiro a sair



# Fila

Uma fila em estrutura de dados funciona exatamente como uma fila da vida real. Suponha que você e um amigo estejam em uma fila em uma parada de ônibus. Se você está antes dele na fila, entrará primeiro no ônibus. As filas funcionam da mesma maneira, tendo funcionamento similar ao das pilhas. Por isso não é possível acessar elementos aleatórios em uma fila. Em vez disso, apenas duas operações são possíveis: **enqueue (enfileirar)** e **dequeue (desenfileirar)**.



# Fila

Se você enfileirar dois itens na lista, o primeiro item adicionado será desenfileirado antes do segundo item. Isso pode ser utilizado em sua lista de pesquisas! Dessa forma, pessoas que foram adicionadas primeiro na lista serão desenfileiradas e verificadas primeiro.

A fila é uma estrutura de dados **FIFO (acrônimo para First In, First Out, que em português significa Primeiro a Entrar, Primeiro a Sair)**. Já a pilha é uma estrutura de dados **LIFO (Last In, First Out, que em português significa Último a Entrar, Primeiro a Sair)**.



# Fila

Precisamos controlar adequadamente as extremidades das filas, pois será nelas que faremos as manipulações. Imagine que você está chegando à fila do restaurante para comer. Onde você se posiciona? No fim da fila, certo? De outro lado, quando você estiver na iminência de ser atendido, estará no início da fila.

## Conceito

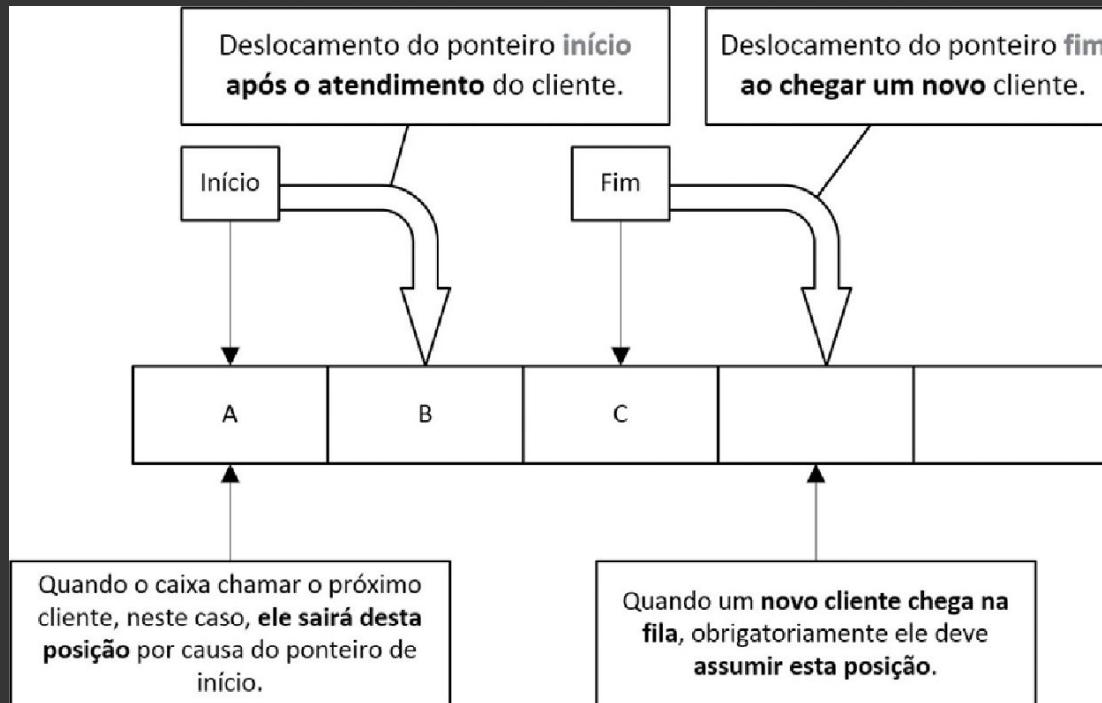
Toda informação que chega à fila é adicionada no **FIM**; toda informação a ser consumida pelo recurso é retirada do **INÍCIO** da fila. Lembre-se: você SEMPRE entra no fim da fila.

# Fila

Neste exemplo vamos imaginar que estamos em uma fila do restaurante da praça de alimentação, que funcionalidades você enxerga na manipulação dessa fila? É exatamente isso que você pensou: *entrar na fila e sair da fila*.

Para entrar na fila, você deve se posicionar imediatamente atrás da última pessoa que está nela e, nesse instante, se tornará o último da fila, posição que será procurada pelo próximo a entrar no fim da fila. Por outro lado, se você for o primeiro da fila, quando o caixa for liberado e chamar o próximo cliente, você sairá da fila e será atendido. Portanto, a pessoa que estiver atrás de você se tornará, naquele momento, o primeiro da fila, sendo a próxima a ser chamada. Isso também acontecerá na programação. Como é que você pensa em fazer isso?

# Fila



# Fila

## □ Deque

A estrutura de dados **deque (abreviação de double-ended queue ou “fila de duas pontas”)** é uma variação da fila que aceita inserção e remoção de elementos tanto do início quanto do final da fila.

Podemos comparar, novamente, com uma fila de pessoas em um guichê de atendimento: uma pessoa idosa que chega é atendida antes (ou seja, não pode ser colocada no fim da fila), ao mesmo tempo que uma pessoa que entrou no final da fila pode desistir de esperar e ir embora (nesse caso, não podemos esperar a pessoa chegar na frente da fila para retirá-la de lá).

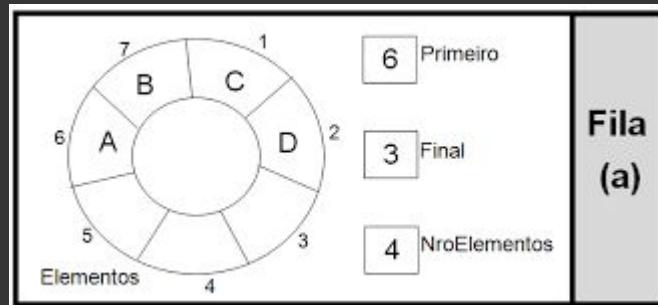
Uma outra forma de se entender a estrutura deque é como uma junção das estruturas de pilha e fila.

# Fila

## □ Fila Circular

Outra variação da fila é a **fila circular (circular queue)**, onde o último elemento é conectado com o primeiro elemento - como em um círculo:

A fila circular busca resolver uma limitação da fila linear, que é reaproveitar as posições vazias, geradas após a remoção.



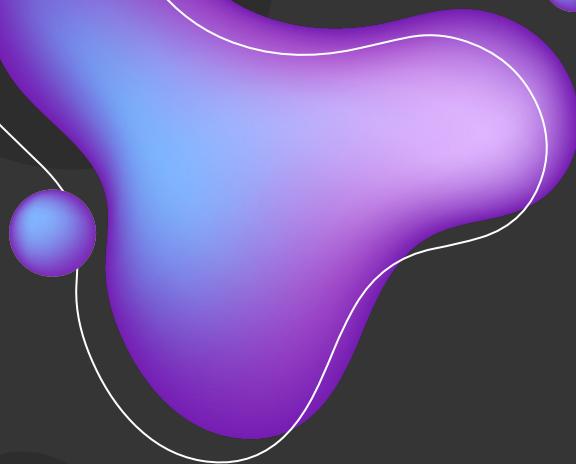
# Fila

## □ Conclusão

Um uso fácil de lembrar para a fila é justamente a fila de impressão dos sistemas operacionais: o último trabalho de impressão a ser adicionado à fila será o último a ser impresso.

Além disso, as requisições feitas a um servidor também são organizadas em fila para serem respondidas, e quando alternamos entre programas utilizando o atalho alt+tab, o sistema operacional faz o gerenciamento da ordem utilizando o princípio de fila circular.

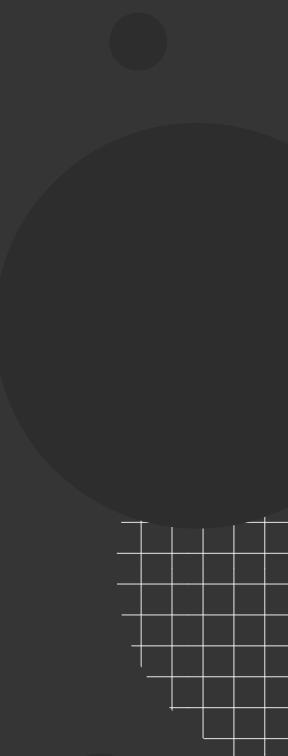
Uma dica é que a fila linear é ineficiente em certos casos em que os elementos são obrigados a mudar para os espaços vagos para executar a operação de inserção. Essa é a razão pela qual ele tende a desperdiçar o espaço de armazenamento enquanto a fila circular faz uso apropriado do espaço de armazenamento, pois os elementos são adicionados em qualquer posição, se houver um espaço vazio.



08

# Algoritmos de Ordenação

Passo a Passo dos principais algoritmos de ordenação



# Selection Sort

Para seguir nesta seção, você precisa ter compreendido arrays e listas, bem como a notação Big O. Suponha que você tenha um monte de músicas no seu computador. Para cada artista, você tem um contador de plays.

~♪~	CONTADOR DE PLAYS
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

# Selection Sort

Você quer ordenar uma lista de artistas, do artista mais tocado para o menos tocado, para que possa categorizar os seus artistas favoritos. Como pode fazer isso? Uma maneira seria pegar o artista mais tocado da lista de músicas e adicioná-lo a uma nova lista.

~♪~	CONTADOR DE PLAYS
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

1. RADIOHEAD É O ARTISTA MAIS TOCADO...

→

SORTED ♪	CONTADOR DE PLAYS
RADIOHEAD	156

2. ADICIONE-O EM UMA NOVA LISTA

# Selection Sort

Faça isso de novo para encontrar o próximo artista mais tocado.

~♪~	CONTADOR DE PLAYS	♪ SORTED ♪	CONTADOR DE PLAYS
KISHORE KUMAR	141	RADIOHEAD	156
THE BLACK KEYS	35	KISHORE KUMAR	141
NEUTRAL MILK HOTEL	94		
BECK	88		
THE STROKES	61		
WILCO	111		

1. KISHORE KUMAR É O PRÓXIMO ARTISTA MAIS TOCADO

2. PORTANTO, ELE É O PRÓXIMO ARTISTA ADICIONADO À NOVA LISTA

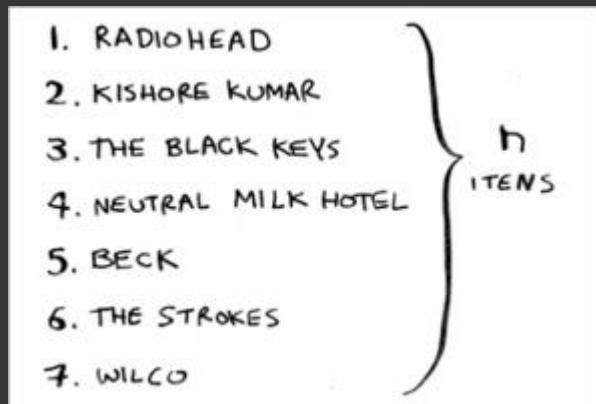
# Selection Sort

Continue fazendo isso e então você terminará com uma lista ordenada.

~♪~	CONTADOR DE PLAYS
RADIOHEAD	156
KISHORE KUMAR	141
WILCO	111
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
THE BLACK KEYS	35

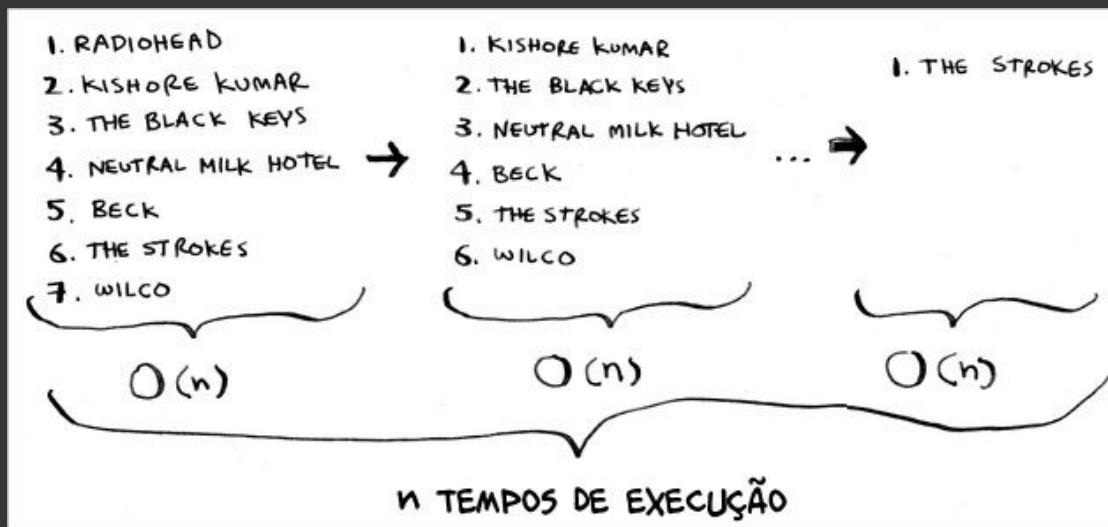
# Selection Sort

Vamos pensar como engenheiros da computação e avaliar quanto tempo isso demoraria a ser executado. Lembre-se de que o tempo de execução  $O(n)$  significa que você precisa passar por todos os elementos da lista uma vez. Por exemplo, executar uma pesquisa simples na lista de artistas significa olhar para cada artista uma vez.



# Selection Sort

Para encontrar o artista com o maior número de plays você precisa verificar cada item da lista. Isso tem tempo de execução  $O(n)$ , como você acabou de ver. Então você tem uma operação com tempo de execução  $O(n)$  e precisa repetir essa operação  $n$  vezes:



# Selection Sort

Este Algoritmo tem um tempo de execução  $O(n \times n)$  ou  $O(n^2)$ .

Algoritmos de ordenação são muito úteis. Agora você pode ordenar:

- Nomes em uma agenda telefônica.
- Datas de viagem.
- Emails (do mais novo ao mais antigo).

A ordenação por seleção é um algoritmo bom, mas não é muito rápido. O Quicksort é um algoritmo de ordenação mais rápido, que tem tempo de execução de apenas  $O(n \log n)$ .

# Selection Sort

## □ Exemplo de Código

```
public void SelectionSort()
{
    for(int i = 0; i < this.ultimaPosicao; i++)
    {
        var menorPosicao = i;

        for(int j = i + 1; j < this.ultimaPosicao + 1; j++)
        {
            var menorValor = this.valores[menorPosicao];
            var valorAtual = this.valores[j];

            if(menorValor > valorAtual)
                menorPosicao = j;
        }

        TrocaSelectionSort(i, menorPosicao);
    }
}

#region TrocaSelectionSort
private void TrocaSelectionSort(int i, int menorPosicao)
{
    var temp = this.valores[i];
    this.valores[i] = this.valores[menorPosicao];
    this.valores[menorPosicao] = temp;
}
#endregion
```

# Dividir para Conquistar

Esta seção focará na utilização destas suas novas habilidades aplicadas na resolução de problemas. Para isto, vamos explorar a técnica **dividir para conquistar (DC)**, uma técnica recursiva muito conhecida para resolução de problemas.

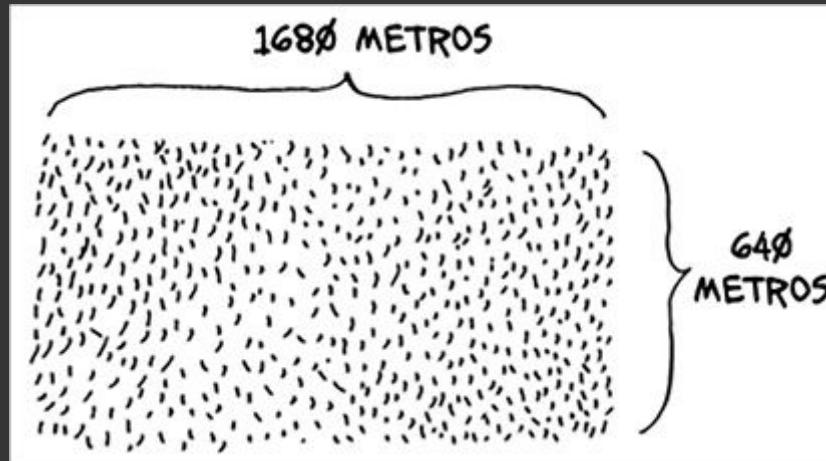
Este capítulo trata do ponto principal dos algoritmos, pois um algoritmo que consegue resolver apenas um tipo de problema não é muito útil. Assim, a técnica DC oferece uma nova maneira de pensar sobre a resolução de problemas, tornando-se mais uma alternativa em sua caixa de ferramentas.

Quando você se deparar com um problema novo, não terá motivos para ficar desnorteado. Em vez disso, poderá se perguntar “Será que posso resolver este problema usando a técnica de dividir para conquistar?”. Ao final desta seção você terá aprendido o um algoritmo que utiliza a técnica DC: o quicksort. O algoritmo quicksort é um algoritmo de ordenação muito mais rápido do que o algoritmo de ordenação por seleção.

# Dividir para Conquistar

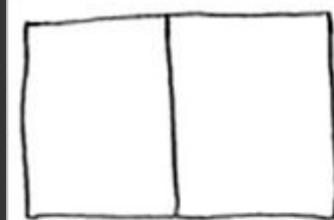
A técnica DC pode levar algum tempo para ser compreendida. Primeiro, mostrarei um exemplo visual. Depois, nos aprofundaremos no quicksort, um algoritmo de ordenação que utiliza DC.

Suponha que você seja um fazendeiro que tenha uma área de terra.

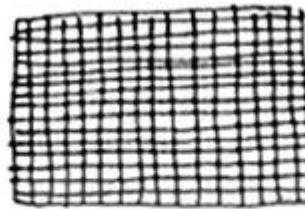


# Dividir para Conquistar

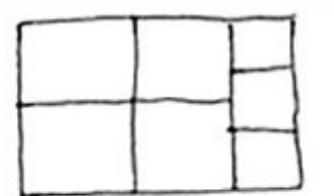
Você quer dividir sua fazenda em porções quadradas iguais, sendo que estas porções devem ter o maior tamanho possível. Assim, nenhuma destas alternativas funcionarão.



PORÇÕES NÃO  
SÃO QUADRADAS



PORÇÕES SÃO  
PEQUENAS DEMAIS



TODAS AS PORÇÕES  
DEVEM POSSUIR O  
MESMO TAMANHO

# Dividir para Conquistar

Como encontrará o maior tamanho possível para estes quadrados? Usando a estratégia DC! Os algoritmos DC são recursivos. Assim, para resolver um problema utilizando DC, você deve seguir dois passos:

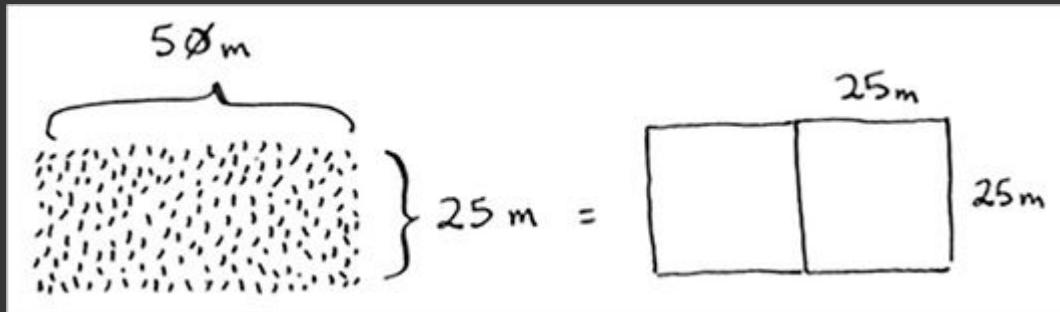
1. Descubra o caso-base, que deve ser o caso mais simples possível.
2. Diminua o seu problema até que ele se torne o caso-base.

Vamos usar DC para encontrar a solução deste problema. Qual é a maior largura que você pode usar?

Primeiro, descubra o caso-base. Seria mais fácil solucionar este problema se um dos lados fosse múltiplo do outro.

# Dividir para Conquistar

Suponha que um dos lados tenha 25 metros (m) e o outro tenha 50. Assim, o maior quadrado que você pode ter mede 25 m x 25 m. Você precisa de dois destes quadrados para dividir a porção de terra.



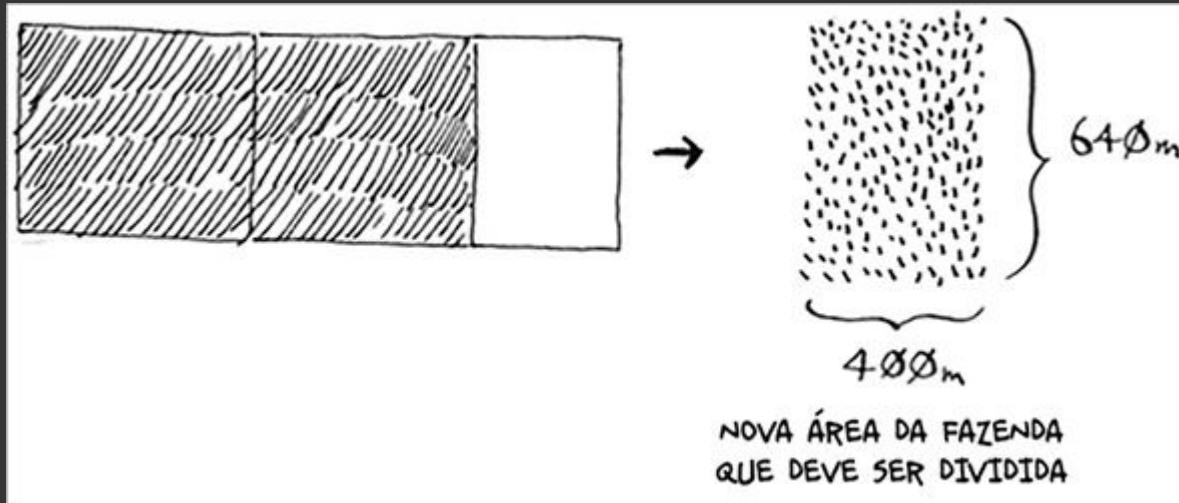
# Dividir para Conquistar

Agora você precisa descobrir o caso recursivo, e é aqui que a estratégia DC entra em ação. Seguindo a estratégia DC, a cada recursão você deve reduzir o seu problema. Então, como reduzir este problema? Vamos começar identificando os maiores quadrados que você pode utilizar.



# Dividir para Conquistar

Você pode posicionar dois quadrados de  $640 \times 640$  na fazenda e ainda continuará com uma porção de terra para ser dividida. Este é o momento “Aha!”. Você ainda tem um segmento da fazenda que deve ser dividido. Por que não aplica este mesmo algoritmo neste segmento?



# Dividir para Conquistar

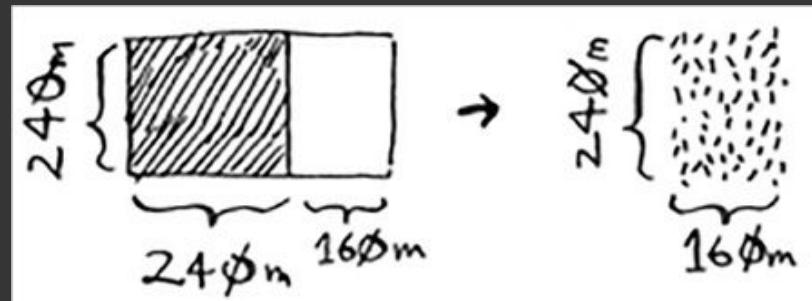
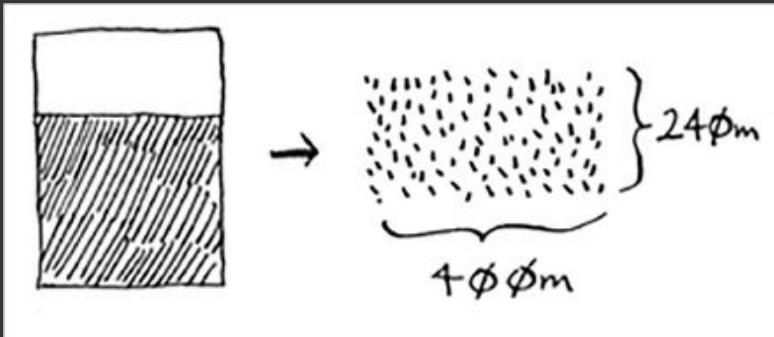
Você iniciou com uma porção de terra medindo  $1.680 \times 640$  que deveria ser dividida. Porém agora você precisa dividir um segmento menor, que mede  $640 \times 400$ . Caso encontre o maior quadrado que divide este segmento, ele será o maior quadrado que dividirá toda a fazenda. Você acabou de reduzir um problema de divisão de uma fazenda medindo  $1.680 \times 640$  para um problema de divisão de uma área medindo  $640 \times 400$ !

Vamos aplicar o mesmo algoritmo novamente. Começando com uma fazenda medindo  $640 \times 400$  m, o maior quadrado que você pode ter mede  $400 \times 400$  m.

# Dividir para Conquistar

E isso deixa você com um segmento menor do que  $400 \times 240$  m.

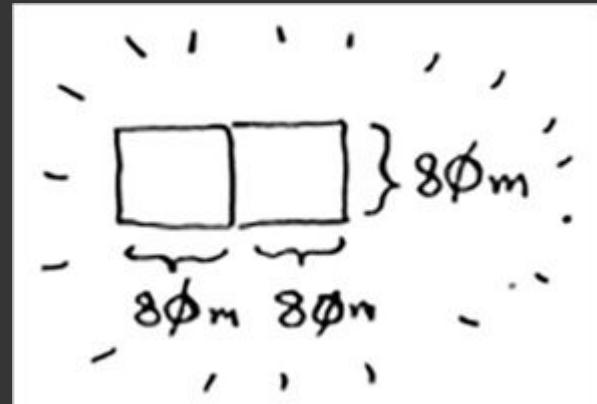
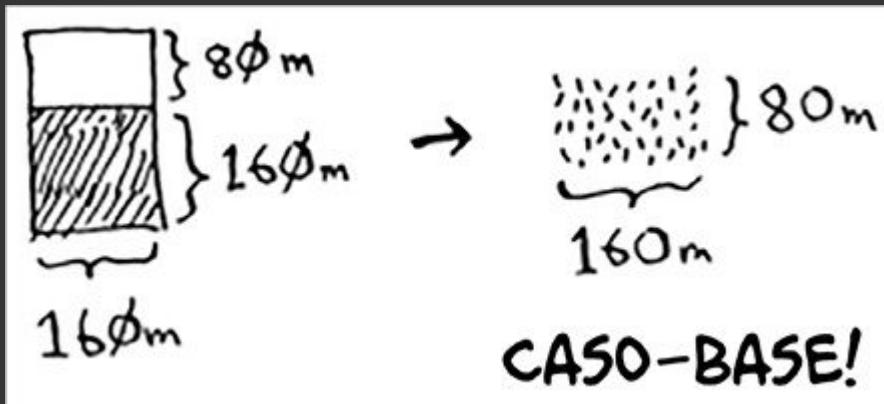
Você pode desenhar um quadrado neste segmento que lhe deixa com um segmento ainda menor, de  $240 \times 160$  m.



# Dividir para Conquistar

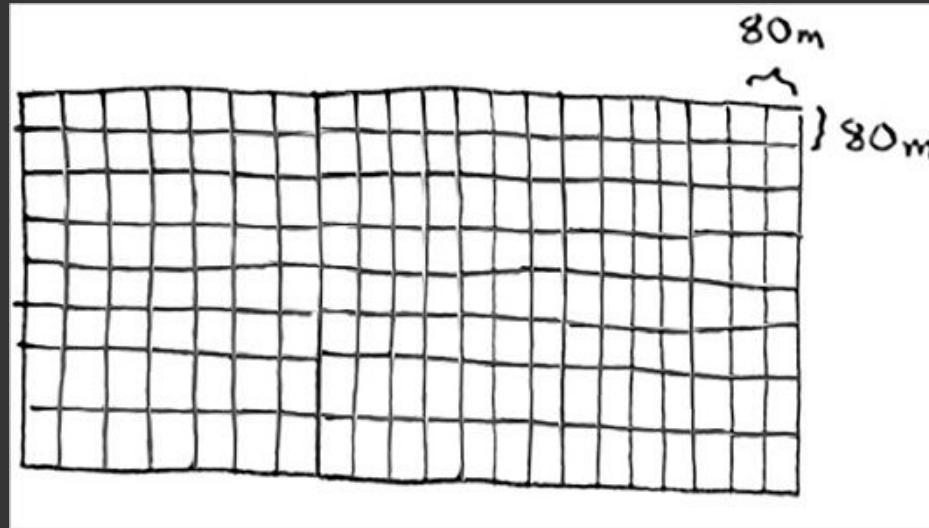
Então, você desenha um quadrado neste segmento para ter um segmento ainda menor.

Ei, você acabou de descobrir o caso-base, pois 80 é um múltiplo de 160. Se dividir este segmento em quadrados, não haverá segmentos sobrando!



# Dividir para Conquistar

Assim, para a fazenda original, o maior quadrado que você pode utilizar é  $80 \times 80$  m.



# Dividir para Conquistar

- Para recapitular, estes são os passos para aplicação da estratégia DC:

1. Descubra o caso-base, que deve ser o caso mais simples possível.
2. Descubra como reduzir o seu problema para que ele se torne o caso-base.

O algoritmo DC não é um simples algoritmo que você aplica em um problema, mas sim uma maneira de pensar sobre o problema. Vamos ver mais um exemplo.

```
var vetor = new int[2,4,6];
System.Console.WriteLine(soma(vetor));
```

```
1 reference
static int soma(int[] vetor)
{
    var total = 0;

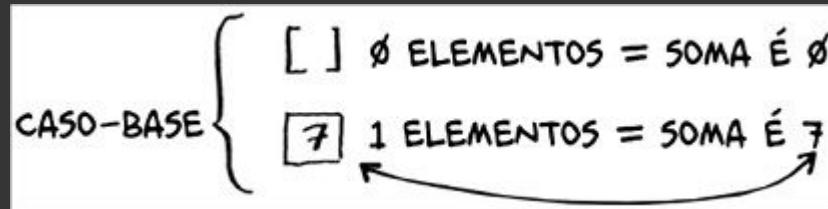
    for (int i = 0; i < vetor.Count; i++)
        total += vetor[i];

    return total;
}
```

# Dividir para Conquistar

Mas como isso poderia ser feito com uma função recursiva?

**Passo 1:** Descubra o caso-base. Qual é o array mais simples que você pode obter? Pense sobre o caso mais simples: se você tiver um array com 0 ou com 1 elemento, será muito simples calcular a soma.



Logo, esse é o caso-base.

# Dividir para Conquistar

**Passo 2:** Você deve chegar mais perto de um array vazio a cada recursão. Como pode reduzir o tamanho do seu problema? Esta é uma alternativa:

$$\text{SOMA} \left( \begin{array}{|c|c|c|} \hline 2 & 4 & 6 \\ \hline \end{array} \right) = 12$$

A soma deste array é igual a isto:

$$2 + \text{SOMA} \left( \begin{array}{|c|c|} \hline 4 & 6 \\ \hline \end{array} \right) = 2 + 1\emptyset = 12$$

# Dividir para Conquistar

Em ambos os casos o resultado é 12. Porém, na segunda versão, você está usando um array menor na função soma. Ou seja, você está diminuindo o tamanho do problema!

A sua função soma poderia funcionar assim:



# Dividir para Conquistar

- Aqui está um exemplo da função na prática:

AMBAS AS REPRESENTAÇÕES → SÃO IGUAIS

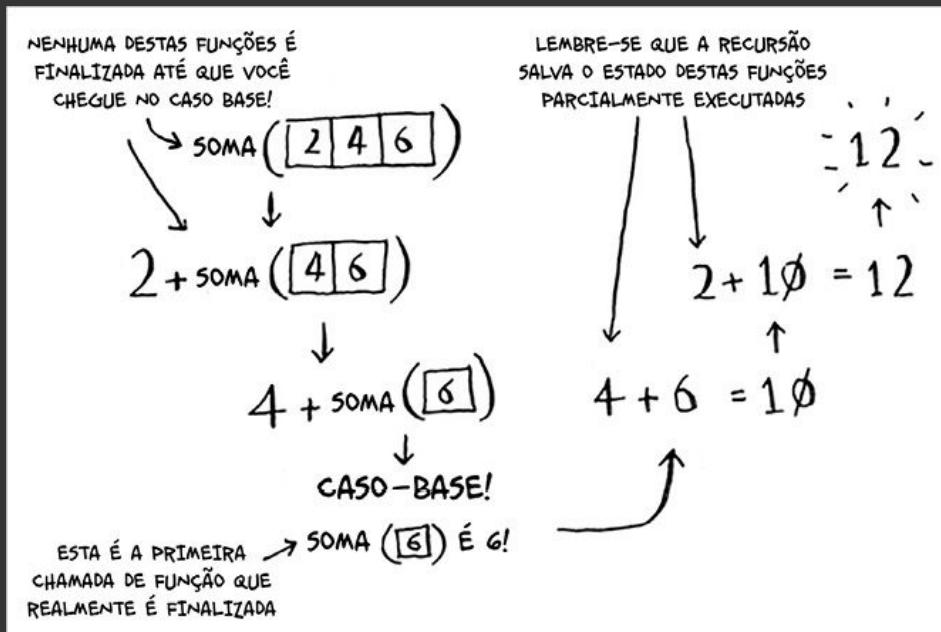
$$\begin{array}{c} \text{SOMA} (2 \boxed{4} 6) \\ \downarrow \\ 2 + \text{SOMA} (\boxed{4} 6) \\ \downarrow \\ 4 + \text{SOMA} (\boxed{6}) \\ \downarrow \\ \text{CASO-BASE!} \\ \text{SOMA} (\boxed{6}) \text{ É } 6! \end{array}$$

RESULTADO FINAL ✓

$$\begin{array}{c} 12 \\ \uparrow \\ 2 + 1\phi = 12 \\ \uparrow \\ 4 + 6 = 1\phi \end{array}$$

# Dividir para Conquistar

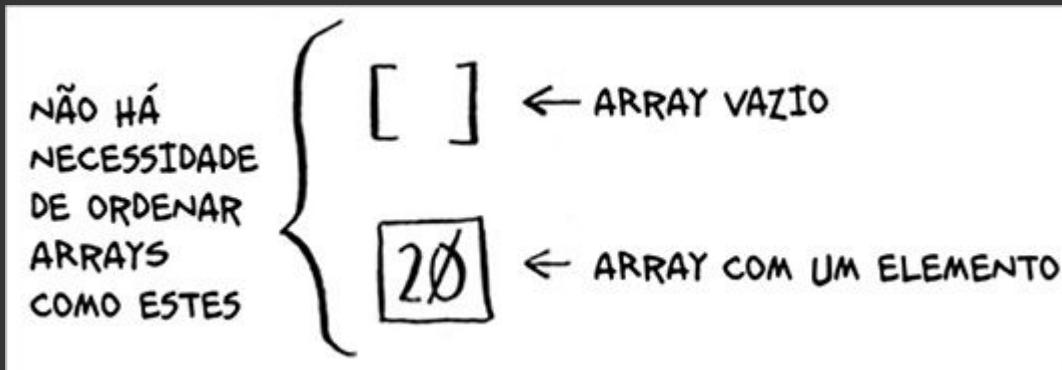
- Lembre-se de que a recursão tem memória dos estados anteriores.



# Quick Sort

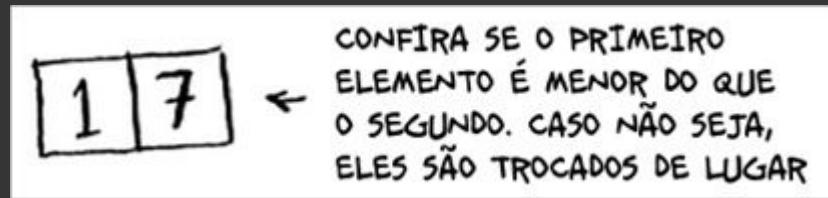
O quicksort é um algoritmo de ordenação. Este algoritmo é muito mais rápido do que a ordenação por seleção e é muito utilizado na prática. Por exemplo, a biblioteca-padrão da linguagem C tem uma função chamada qsort, que é uma implementação do quicksort. O algoritmo quicksort também utiliza a estratégia DC.

Vamos usar o quicksort para ordenar um array. Qual é o array mais simples que um algoritmo de ordenação pode ordenar? Bem, alguns arrays não precisam nem ser ordenados.



# Quick Sort

Arrays vazios ou arrays com apenas um elemento serão o caso-base. Você pode apenas retornar esses arrays como eles estão, visto que não há nada para ordenar.



E um array com três elementos?

33	15	10
----	----	----

# Quick Sort

Lembre-se, você está usando DC. Sendo assim, quer quebrar este array até que você chegue ao caso-base. Portanto o funcionamento do quicksort segue esta lógica: primeiro, escolha um elemento do array. Esse elemento será chamado de pivô.



Falaremos sobre como escolher um bom pivô mais tarde. Neste momento, vamos utilizar o primeiro item do array como pivô.

# Quick Sort

Assim, encontre os elementos que são menores do que o pivô e também os elementos que são maiores.

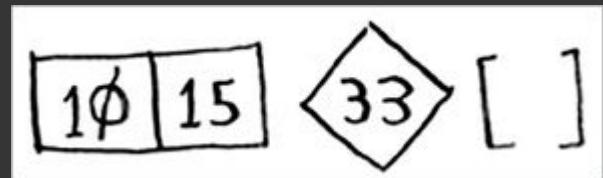


# Quick Sort

□ **Isso é chamado de particionamento. Desse modo, você tem:**

- Um subarray contendo todos os números menores do que o pivô.
- O pivô.
- Um subarray contendo todos os números maiores do que o pivô.

Os dois subarrays não estão ordenados, apenas particionados. Porém, se eles estivessem ordenados, a ordenação do array contendo todos os elementos seria simples.



# Quick Sort

Caso os subarrays estejam ordenados, poderá combiná-los desta forma:

**array esquerdo + pivô + array direito.** Consequentemente, terá um array ordenado. Neste caso, temos  $[10, 15] + [33] + [] = [10, 15, 33]$ , que é um array ordenado. Como você pode ordenar os subarrays? Bem, o caso-base do quicksort consegue ordenar arrays de dois elementos (o subarray esquerdo) e também arrays vazios (o subarray direito). Assim, se utilizar o quicksort em ambos os subarrays e então combinar os resultados, terá um array ordenado!

`quicksort([15, 10]) + [33] + quicksort( [] ) > [10, 15, 33]` ①

① Um array ordenado

# Quick Sort

Isto funcionará com qualquer pivô. Suponha que você tenha escolhido o número 15 como pivô.



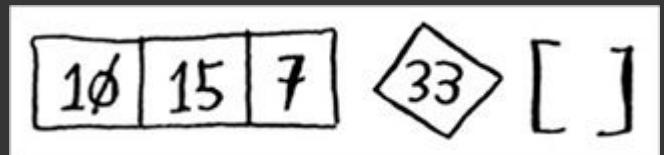
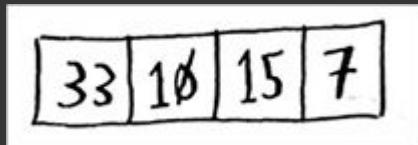
Ambos os subarrays contêm apenas um elemento, e você já sabe como ordenar este tipo de array. Logo, já sabe como ordenar um array de três elementos. Estes são os passos:

1. Escolha um pivô.
2. particione o array em dois subarrays, separando-os entre elementos menores do que o pivô e elementos maiores do que o pivô.
3. Execute o quicksort recursivamente em ambos os subarrays.

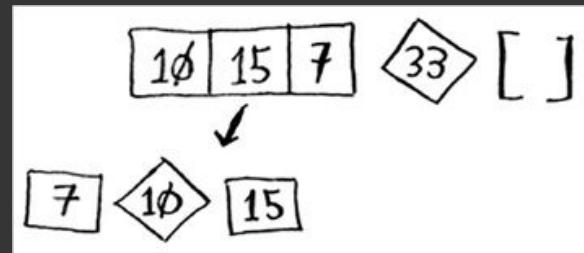
# Quick Sort

E quanto a um array de quatro elementos?

Suponha que, desta vez, você escolheu o número 33 como pivô.

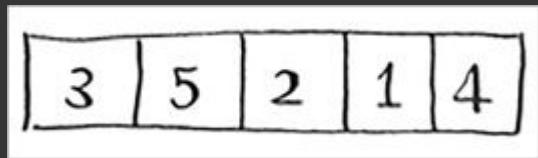


O array da esquerda contém três elementos, e você já sabe como ordenar arrays de três elementos: executando o quicksort recursivamente.

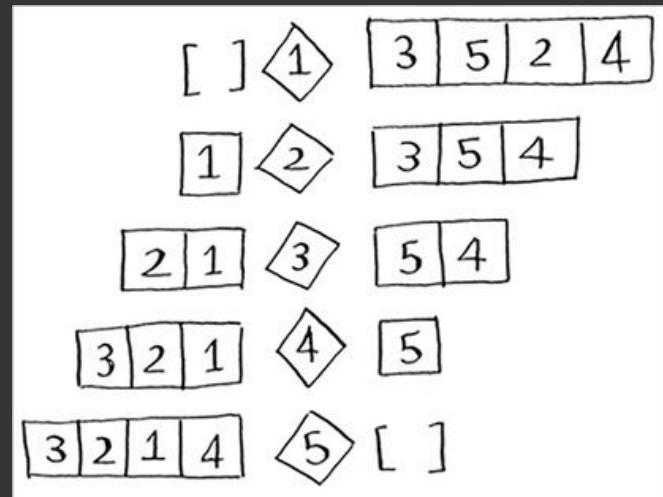


# Quick Sort

Agora pode ordenar arrays de quatro elementos. Sabendo ordenar arrays com quatro elementos, você consegue ordenar arrays com cinco elementos. Como? Suponha que tenha um array com cinco elementos.



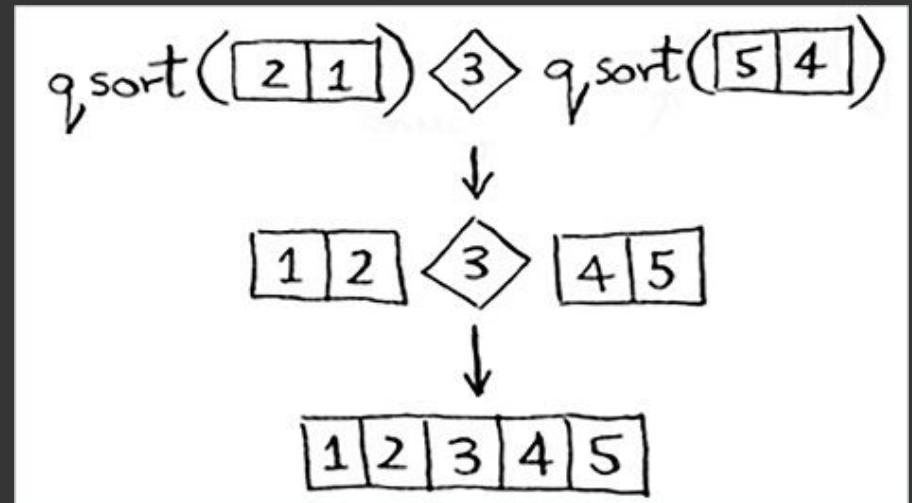
Estas são todas as maneiras pelas quais você pode particionar este array, dependendo do pivô que escolher.



# Quick Sort

Perceba que todos estes subarrays têm entre 0 a 4 elementos, e você já sabe como ordenar arrays de 0 a 4 elementos usando o quicksort! Logo, não importa o pivô que você escolher, pois você poderá executar o quicksort recursivamente em ambos os subarrays.

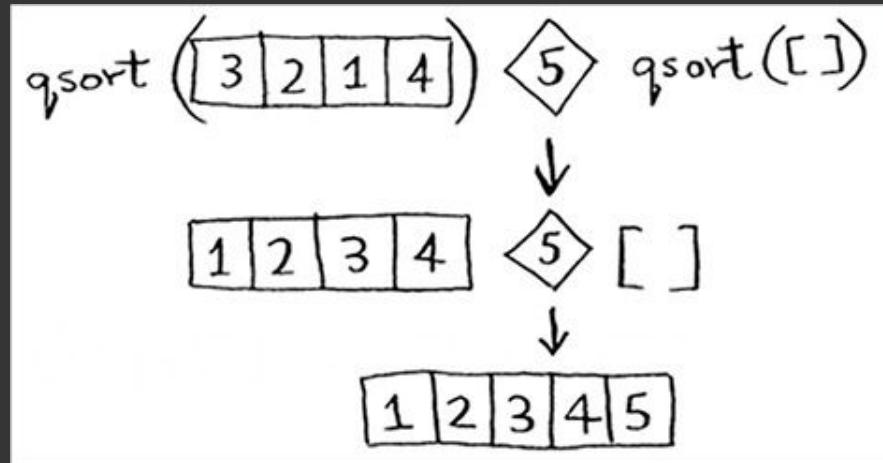
Por exemplo, imagine que você escolheu o número 3 como pivô. Você executa o quicksort nos subarrays.



# Quick Sort

Os subarrays são ordenados e então você os combina, obtendo um array ordenado. Isto funcionará mesmo que escolha o número 5 como pivô.

Isso funcionará, na verdade, com qualquer elemento como pivô. Agora você já consegue ordenar um array de cinco elementos. Usando a mesma lógica, conseguirá ordenar um array de seis elementos ou mais.



## □ Aqui está o código para o quicksort:

```
public void QuickSort(int inicio, int fim)
{
    if(inicio < fim)
    {
        var posicao = Particao(inicio, fim);

        // Esquerda
        QuickSort(inicio, posicao - 1);

        // Direita
        QuickSort(posicao + 1, fim);
    }
}
```

```
private int Particao(int inicio, int fim)
{
    var pivo = this.valores[fim];
    var i = inicio - 1;

    for (int j = inicio; j < fim; j++)
    {
        if (this.valores[j] <= pivo)
        {
            i += 1;
            TrocarPosicao(i, j);
        }
    }

    TrocarPivo(i, fim);

    return i + 1;
}
```

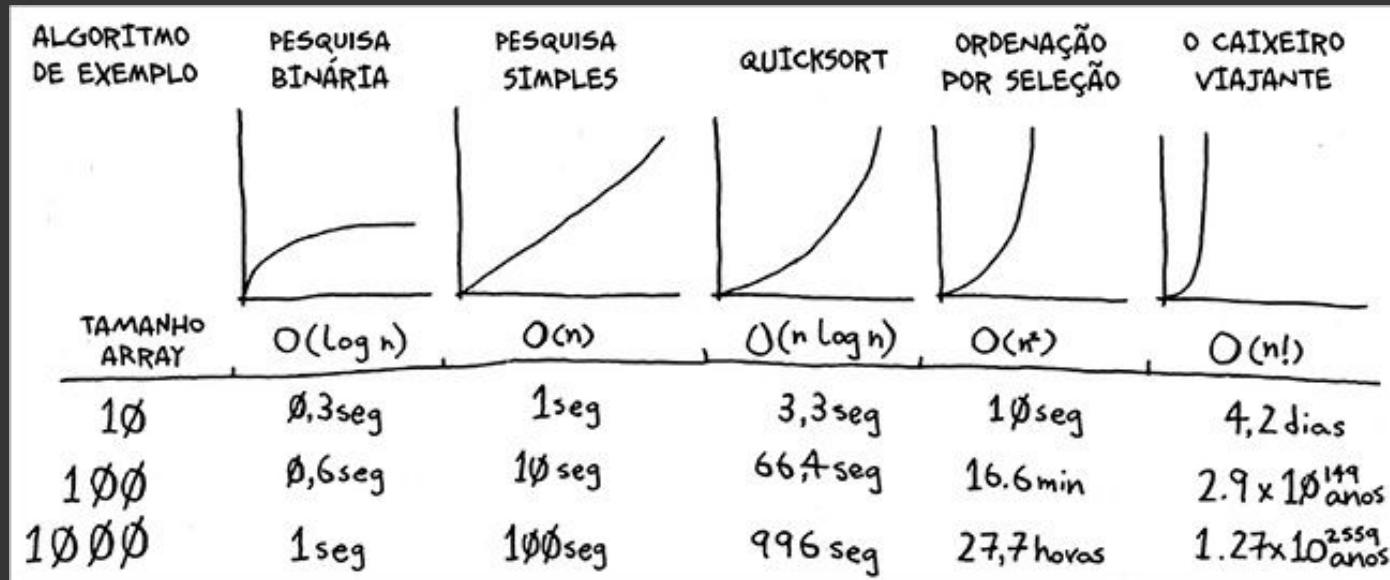
## □ Aqui está o código para o quicksort:

```
private void TrocarPosicao(int i, int j)
{
    var temp = this.valores[i];
    this.valores[i] = this.valores[j];
    this.valores[j] = temp;
}
```

```
private void TrocarPivo(int i, int fim)
{
    var temp = this.valores[i + 1];
    this.valores[i + 1] = this.valores[fim];
    this.valores[fim] = temp;
}
```

# Quick Sort

O algoritmo quicksort é único, pois sua velocidade depende do pivô escolhido. Antes de falarmos sobre quicksort, vamos analisar novamente os tempos de execução Big O mais comuns.



# Quick Sort

Os exemplos de tempos de execução contidos nestes gráficos são estimativas para um caso em que você executa dez operações por segundo. Estes gráficos não são precisos, mas servem apenas para fornecer um exemplo do quanto diferente são os tempos de execução. Na realidade, o seu computador é capaz de executar muito mais do que dez operações por segundo.

Há outro algoritmo de ordenação chamado merge sort, que tem tempo de execução  $O(n \log n)$ , o que é muito mais rápido! O algoritmo quicksort é um caso complicado. Na pior situação, o quicksort tem tempo de execução  $O(n^2)$ .

# Quick Sort

Ele é tão lento quanto a ordenação por seleção! Porém este é o pior caso possível. No caso médio, o quicksort tem tempo de execução  $O(n \log n)$ . E agora você pode estar se perguntando:

- O que significa pior caso e caso médio?
- Se o quicksort tem tempo de execução médio  $O(n \log n)$ , e o merge sort tem tempo de execução  $O(n \log n)$  sempre, por que não utilizar o merge sort? Não seria mais rápido?

# Merge Sort Vs Quick Sort

Suponha que você tenha esta simples função que imprime na tela todos os itens de uma lista:

```
def imprime_itens(lista):
    for item in lista:
        print item
```

Esta função analisa cada item da lista e o imprime. Como esta função passa por toda a lista uma vez, ela tem tempo de execução  $O(n)$ . Agora, imagine que você modificou esta função para que ela aguarde um segundo antes de imprimir um item:

```
from time import sleep
def imprime_itens2(lista):
    for item in lista:
        sleep(1)
        print item
```

# Merge Sort Vs Quick Sort

Antes de imprimir um item, ela espera por um segundo. Suponha que você imprima uma lista contendo cinco itens utilizando ambas as funções.

2	4	6	8	1Ø
---	---	---	---	----

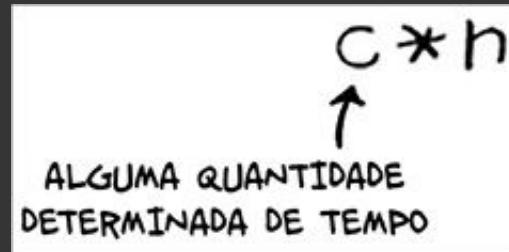


imprime\_itens: 2 4 6 8 1Ø

imprime\_itens 2: <AGUARDE> 2 <AGUARDE> 4 <AGUARDE> 6 <AGUARDE> 8 <AGUARDE> 1Ø

# Merge Sort Vs Quick Sort

Ambas as funções passam por toda a lista uma vez, portanto elas têm tempo de execução  $O(n)$ . Qual das duas você acha que será mais rápida na prática? Acho que a função imprime\_itens será muito mais rápida, visto que ela não aguarda um segundo antes de imprimir cada item. Assim, mesmo que ambas as funções tenham o mesmo tempo de execução na notação Big O, a função imprime\_itens acaba sendo mais rápida na prática. Quando você escreve algo na notação Big O, como  $O(n)$ , por exemplo, está querendo dizer isso.



# Merge Sort Vs Quick Sort

A letra c é uma quantidade determinada de tempo que o seu algoritmo leva para ser executado. Ela é chamada de constante. Pode ser, por exemplo, 10 milissegundos \* n para a função imprime\_itens contra 1 segundo \* n para a função imprime\_itens2.

Normalmente você ignora a constante, pois, caso dois algoritmos tenham tempos de execução Big O diferentes, a constante não importará. Vamos usar a pesquisa binária e a pesquisa simples como exemplos deste fato. Imagine que ambos os algoritmos contenham estas constantes.

$\frac{10\text{ms} * n}{\text{PESQUISA SIMPLES}}$	$\frac{1\text{seg} * \log n}{\text{PESQUISA BINÁRIA}}$
---	--

# Merge Sort Vs Quick Sort

Você pode pensar “Nossa! A pesquisa simples contém uma constante de 10 milissegundos, enquanto a pesquisa binária contém uma constante de um segundo. A pesquisa simples é muito mais rápida!”. Agora, suponha que esteja realizando uma busca em uma lista com 4 bilhões de elementos. A seguir, pode visualizar os tempos de execução desta busca.

PESQUISA SIMPLES	$10 \text{ ms} * 4 \text{ BILHÕES} = 463 \text{ dias}$
PESQUISA BINÁRIA	$1 \text{ seg} * 32 = 32 \text{ segundos}$

# Merge Sort Vs Quick Sort

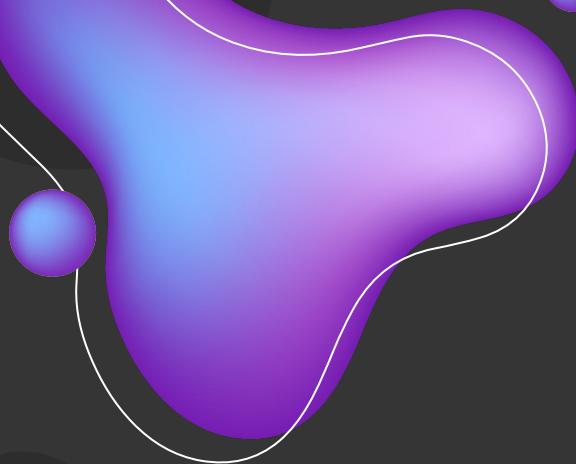
Como você pode ver, a pesquisa binária continua sendo muito mais rápida. A constante não causou diferença alguma no final das contas.

Porém, às vezes, a constante pode fazer diferença. O quicksort, comparado ao merge sort, é um exemplo disso. O quicksort tem uma constante menor do que o merge sort. Assim, como ambos têm tempo de execução  $O(n \log n)$ , o quicksort acaba sendo mais rápido. Além disso, o quicksort é mais rápido, na prática, pois ele funciona mais vezes no caso médio do que no pior caso.

# Merge Sort Vs Quick Sort

## □ Concluindo

- A estratégia DC funciona por meio da divisão do problema em problemas menores. Se você estiver utilizando DC em uma lista, o caso-base provavelmente será um array vazio ou um array com apenas um elemento.
- Se você estiver implementando o quicksort, escolha um elemento aleatório como o pivô. O tempo de execução médio do quicksort é  $O(n \log n)$ !
- A constante, na notação Big O, pode ser relevante em alguns casos. Esta é a razão pela qual o quicksort é mais rápido do que o merge sort.
- A constante dificilmente será relevante na comparação entre pesquisa simples e pesquisa binária, pois  $O(\log n)$  é muito mais rápido do que  $O(n)$  quando sua lista é grande.



09

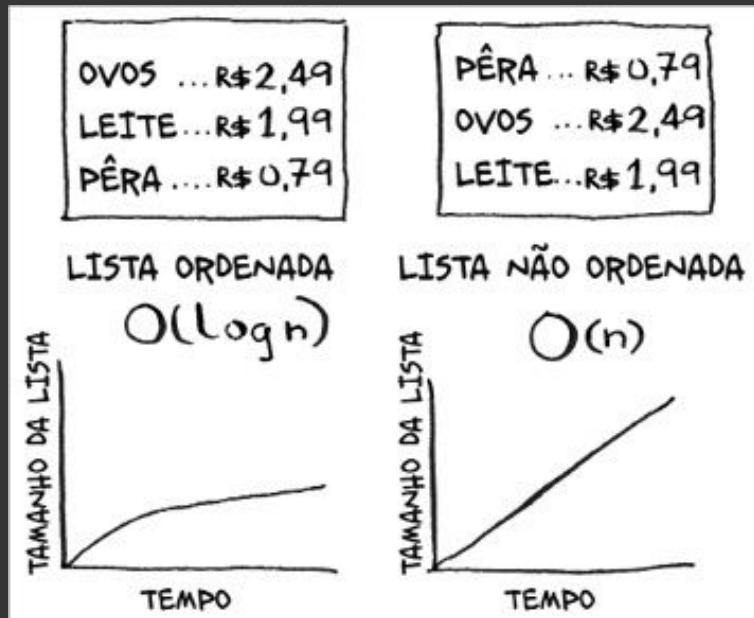
# Tabela Hash

Unindo o melhor dos mundos de Arrays e Listas Encadeadas



# Introdução

Imagine que você trabalha em um mercado. Quando um cliente compra um produto, é preciso conferir o preço deste produto em um caderno. Porém, se o caderno não estiver organizado alfabeticamente, você levará muito tempo analisando cada linha até encontrar o preço da maçã, por exemplo. Procurando desta forma você realizaria uma pesquisa simples, vista nas sessões anteriores, e por meio dela teria de analisar todas as linhas. Você lembra qual era o tempo de execução da pesquisa simples?  $O(n)$ . No entanto, se o caderno estivesse ordenado alfabeticamente, poderia executar uma pesquisa binária para encontrar o preço da maçã com um tempo de execução  $O(\log n)$ .



# Introdução

Vale lembrar que existe uma grande diferença entre um tempo de execução  $O(n)$  e  $O(\log n)$ ! Suponha que você conseguisse ler dez linhas do caderno por segundo. Na figura a seguir você pode ver quanto tempo levaria usando a pesquisa binária e a pesquisa simples.

# DE ITENS NO CADERNO	$O(n)$	$O(\log n)$
100	10 seg	1 seg ← VOCÊ PRECISA VERIFICAR $\log_2 100 = 7$ LINHAS
1000	1.66 min	1 seg ← VOCÊ PRECISA CHECAR $\log_2 1000 = 10$ LINHAS
10000	16.6 min	2 seg ← $\log_2 10000 = 14$ LINHAS = 2 seg

# Introdução

Você já sabe que a pesquisa binária é muito mais rápida. Porém, como um caixa de mercado, você já sabe que procurar o preço de mercadorias em um caderno é uma tarefa chata, mesmo que este caderno esteja ordenado, pois o cliente fica impaciente enquanto a procura pelo preço dos itens é realizada. Assim, o que você precisa é de um amigo que conheça todas as mercadorias e os seus preços, pois, dessa forma, não é necessário procurar nada: você pede para este seu amigo e ele informa o preço imediatamente.



# Introdução

A sua amiga Maggie pode dizer o preço com tempo de execução  $O(1)$  para todos os itens, não importando a quantidade de itens que compõem o caderno de preços. Dessa forma, ela é ainda mais rápida do que a pesquisa binária.

# DE ITENS NO CADERNO	PESQUISA SIMPLES	PESQUISA BINÁRIA	MAGGIE
100	$O(n)$	$O(\log n)$	$O(1)$
1000	10 seg	1 seg	INSTANTÂNEO
10000	1.6 min	1 seg	INSTANTÂNEO
100000	16.6 min	2 seg	INSTANTÂNEO

# Introdução

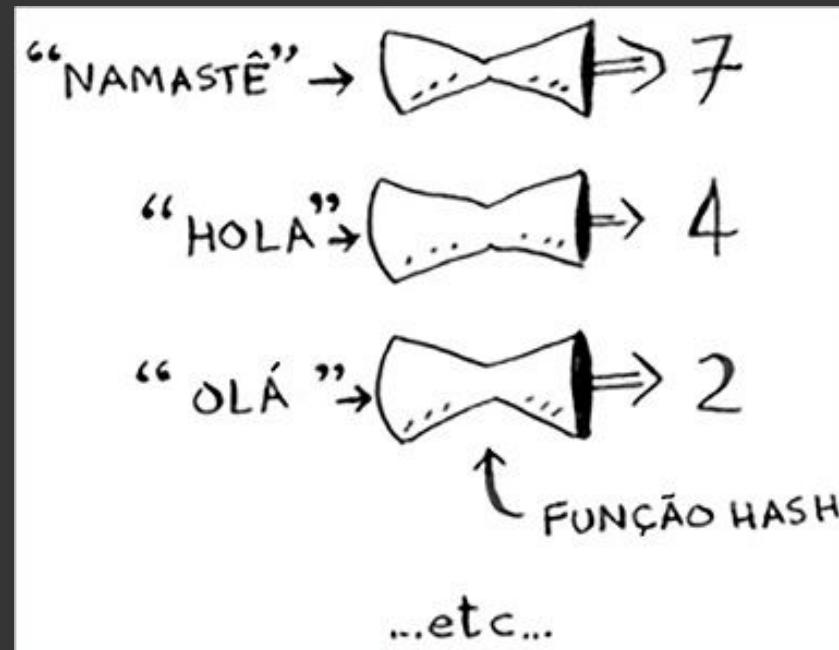
Que amiga maravilhosa! Mas, então, como você arranja uma “Maggie”? Agora, vamos voltar a falar de estruturas de dados. Você já conhece duas estruturas até agora: arrays e listas. Seria possível implementar este seu caderno de preços como um array.

(Ovos, 2.49)	(LEITE, 1.49)	(PÊRA, 0.79)
--------------	---------------	--------------

Cada item neste array é, na realidade, uma dupla de itens: um é o nome e o tipo do produto e o outro é o preço. Se ordenar este array por nome, será possível executar uma pesquisa binária para procurar o preço de um item. Logo, é possível pesquisar itens com tempo de execução  $O(\log n)$ . Entretanto, nós queremos encontrar itens com tempo de execução  $O(1)$ , ou seja, queremos uma “Maggie”, e é aí que entram as funções hash.

# Função Hash

Uma função hash é uma função na qual você insere uma string e, depois disso, a função retorna um número.



# Função Hash

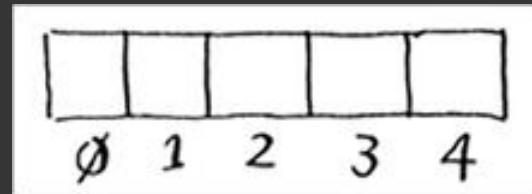
Em uma terminologia mais técnica, diríamos que uma função hash “mapeia strings e números”. Você pode pensar que não existe um padrão indicando qual número será retornado após a inserção de uma string, mas existem alguns requisitos para uma função hash:

- Ela deve ser consistente. Imagine que você insere a string “maçã” e recebe o número 4. Todas as vezes que você inserir “maçã”, a função deverá retornar o número “4”; caso contrário, sua tabela hash não funcionará corretamente.
- A função deve mapear diferentes palavras para diferentes números. Desta forma, uma função hash não será útil se ela sempre retornar “1”, independentemente da palavra inserida. No melhor caso, cada palavra diferente deveria ser mapeada e ligada a um número diferente.

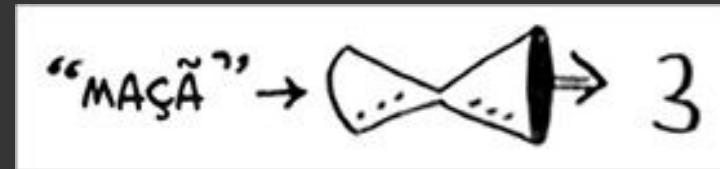
Então, uma função hash mapeia strings e as relaciona a números. Mas qual é a utilidade disso? Bem, você pode usar esta funcionalidade para criar a sua “Maggie”!

# Função Hash

Comece com um array vazio:

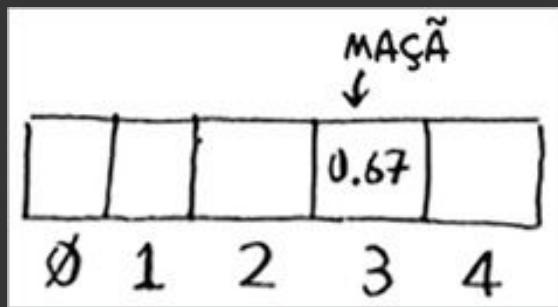


Você armazenará os preços das mercadorias neste array. Vamos adicionar o preço de uma maçã. Insira “maçã” na função hash.



# Função Hash

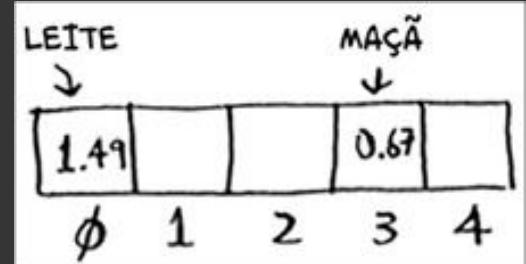
Ela retornará o valor “3”. Agora, vamos armazenar o preço da maçã no índice 3 do array.



Vamos adicionar o leite agora. Insira “leite” na função hash.

“LEITE” →

A função hash retornou “0”. Agora, armazene o preço do leite no índice 0.

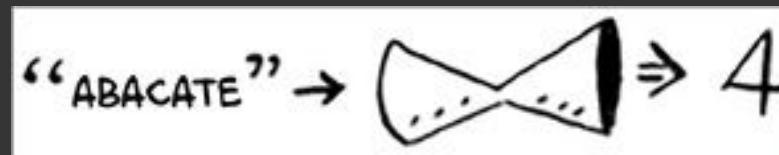


# Função Hash

Continue e, eventualmente, todo o array estará repleto de preços.

1.49	0.79	2.49	0.67	1.49
------	------	------	------	------

Agora, você poderá perguntar “Ei, qual é o preço de um abacate?” e não será necessário procurar o preço deste produto no array. Em vez disso, insira “abacate” na função hash. Ela informará o preço armazenado no índice 4.



# Função Hash

A função hash informará a posição exata em que o preço está armazenado. Assim, não precisará procurá-lo! Isto funciona porque:

- A função hash mapeia consistentemente um nome para o mesmo índice. Todas as vezes que você inserir “abacate”, ela retornará o mesmo número. Assim, a primeira execução da função hash servirá para identificar onde é possível armazenar o preço do abacate, e depois disso ela será utilizada para encontrar este valor armazenado.
- A função hash mapeia diferentes strings para diferentes índices. A string “abacate” é mapeada para o índice 4. A string “leite” é mapeada para o índice 0. Todas as diferentes strings são mapeadas para diferentes lugares do array onde você está armazenando os preços.
- A função hash tem conhecimento sobre o tamanho do seu array e retornará apenas índices válidos. Portanto, caso o seu array tenha cinco itens, a função hash não retornará 100, pois este valor não seria um índice válido do array.

# Função Hash

Você acabou de criar uma “Maggie”! Coloque uma função hash em conjunto com um array e você terá uma estrutura de dados chamada tabela hash. Uma tabela hash é a primeira estrutura de dados que tem uma lógica adicional aliada que você aprenderá, visto que arrays e listas mapeiam diretamente para a memória, porém as tabelas hash são mais inteligentes. Elas usam uma função hash para indicar, de maneira inteligente, onde armazenar os elementos.

As tabelas hash são, provavelmente, as mais úteis e complexas estruturas de dados que você aprenderá. Elas também são conhecidas como mapas hash, mapas, dicionários e tabelas de dispersão. Além disso, as tabelas hash são muito rápidas! Você se lembra da nossa discussão sobre arrays e listas encadeadas nas sessões anteriores? Você pode pegar um item de um array instantaneamente que as tabelas hash usarão arrays para armazenar os dados desse item; desta forma, elas são igualmente velozes.

# Tabela Hash

Tabelas hash são amplamente utilizadas. Nesta seção, vamos analisar alguns casos.

## □ Usando tabelas hash para pesquisas

O seu celular tem uma agenda telefônica integrada. Cada nome está associado a um número telefônico.

BADE MAMA → 581 660 9820

ALEX MANNING → 484 234 4680

JANE MARIN → 415 567 3579



# Tabela Hash

Imagine que você queira construir uma lista telefônica como esta. Será necessário mapear o nome das pessoas e associá-los a números telefônicos. Dessa forma, a sua lista telefônica deverá ter estas funcionalidades:

- Adicionar o nome de uma pessoa e o número de telefone associado a este nome.
- Inserir o nome de uma pessoa e receber o número telefônico associado a ela.

Este é um exemplo perfeito de situações em que tabelas hash podem ser usadas! As tabelas hash são ótimas opções quando:

- Você deseja mapear algum item com relação a outro.
- Você precisa pesquisar algo.

# Tabela Hash

Criar uma lista telefônica é uma tarefa simples. Primeiro, faça uma nova tabela hash para **lista\_telefonica**:

```
var lista_telefonica = new Hashtable();

lista_telefonica["jenny"] = 8675309;
lista_telefonica["emergency"] = 911;
```

É isso! Agora, digamos que queira encontrar o número de telefone da Jenny. Para isso, é necessário apenas informar a chave para a hash:

```
Console.WriteLine(lista_telefonica["jenny"]);
```

# Tabela Hash

Imagine que tivesse de fazer isto utilizando um array. Como faria? As tabelas hash tornam simples a modelagem de uma relação entre dois itens.

As tabelas hash são usadas para pesquisas em uma escala muito maior. Por exemplo, suponha que você queira acessar um website como o <http://adit.io>. O seu computador deve traduzir o nome adit.io para a forma de um endereço de IP.

ADIT.IO → 173.255.248.55

# Tabela Hash

Para cada website que você acessar, o endereço deverá ser traduzido para um endereço de IP.

Nossa, mapear o endereço de um website para um endereço IP? Isso parece o caso perfeito para a utilização de tabelas hash! Este processo é chamado de resolução DNS, e as tabelas hash são uma das maneiras pelas quais esta funcionalidade pode ser implementada.

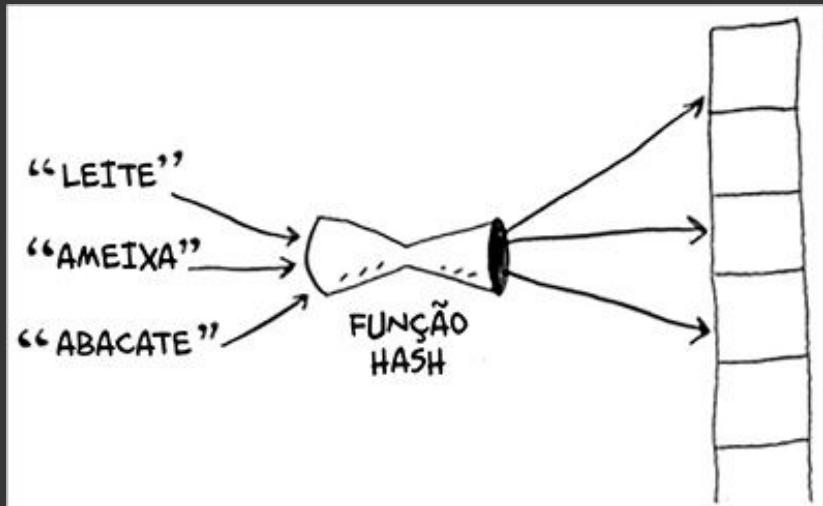
GOOGLE.COM → 74.125.239.133

FACEBOOK.COM → 173.252.120.6

SCRIBD.COM → 23.235.47.175

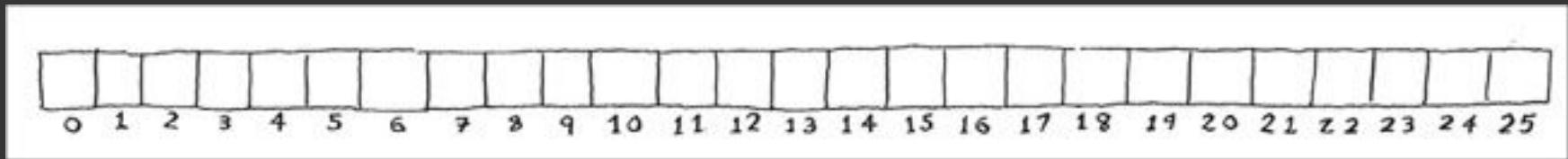
# Colisões

Como eu disse antes, a maioria das linguagens de programação contém tabelas hash, por isso você não precisa escrevê-las do zero. Sendo assim, não falarei muito sobre a estrutura das tabelas hash. No entanto, você precisa saber sobre o desempenho delas, e para isso precisa primeiro entender o que são colisões. Agora falaremos sobre colisões e desempenho. Primeiro, preciso dizer que estive contando uma pequena mentira. Disse que uma função hash sempre mapeia diferentes chaves para diferentes espaços em um array.

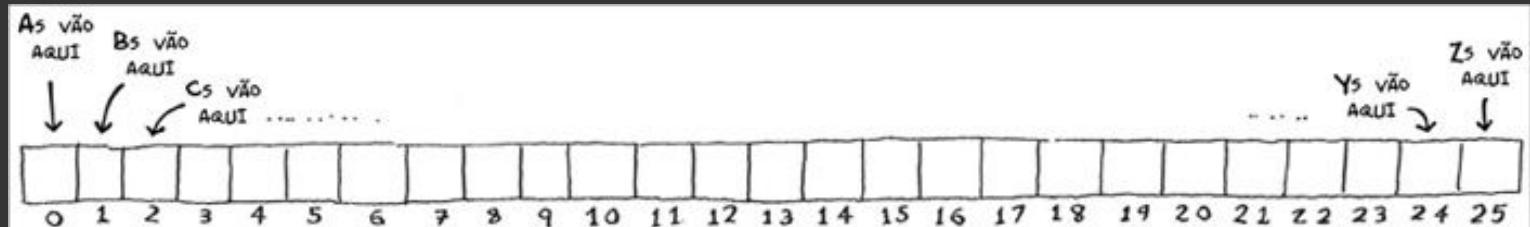


# Colisões

Na realidade, é praticamente impossível escrever uma função hash que faça isso. Vamos analisar este exemplo simples: suponha que você tenha um array que contenha 26 espaços.

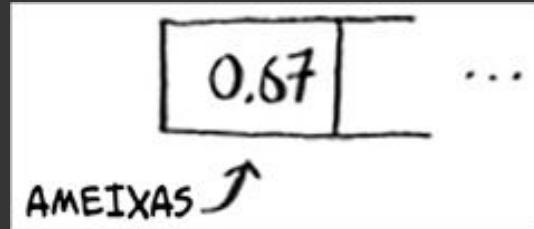


Sua função hash é bem simples: ela apenas indica um espaço do array alfabeticamente.

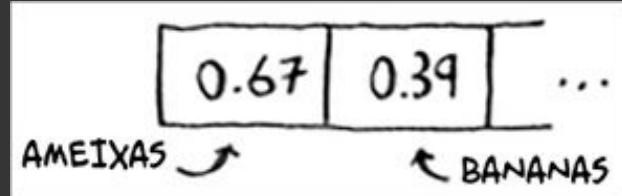


# Colisões

Talvez você já consiga ver o problema. Você quer inserir o preço de uma ameixa na sua hash. Assim, o primeiro espaço do array é indicado.

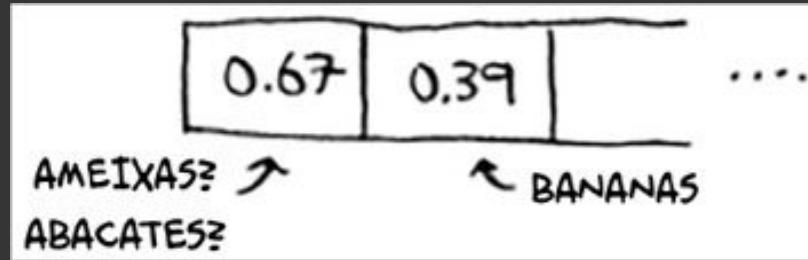


Depois, você quer inserir o preço das bananas. Então, o segundo espaço do array é indicado.



# Colisões

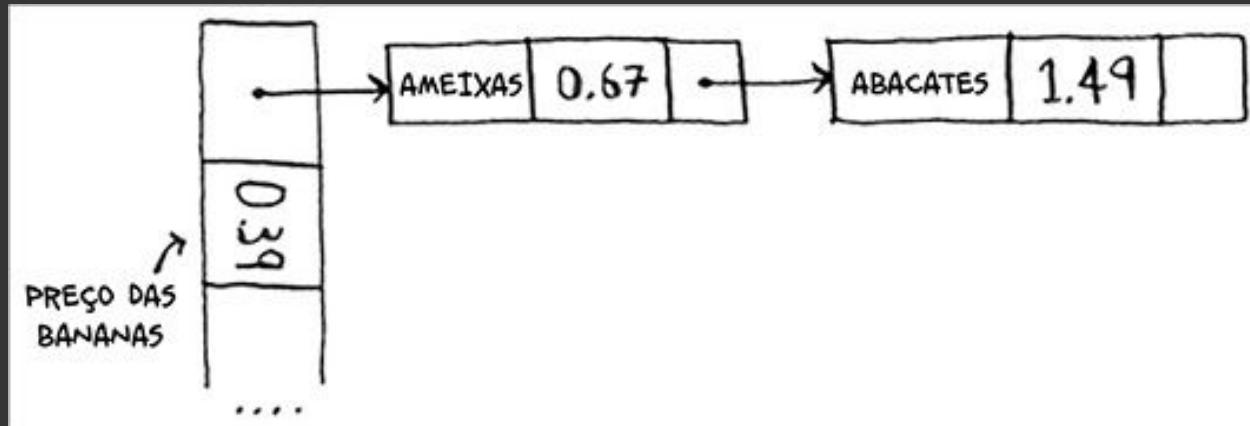
Tudo está indo tão bem! Mas agora quer inserir o preço dos abacates na sua hash. Então, o primeiro espaço do array é indicado novamente.



Ah, não! As ameixas já estão neste espaço! O que vamos fazer? Isso se chama colisão: duas chaves são indicadas para o mesmo espaço, e isto é um problema. Assim, se armazenar o preço dos abacates neste espaço, eles irão sobrescrever o preço das ameixas. Desta forma, da próxima vez que alguém perguntar o preço das ameixas, o preço dos abacates será informado!

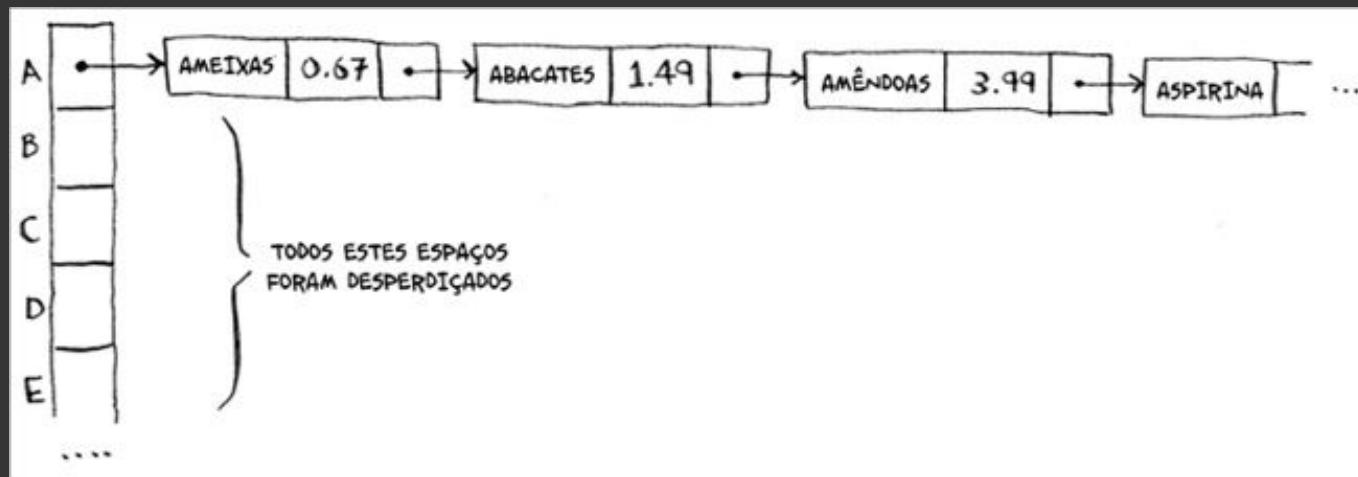
# Colisões

Colisões são um problema, e você precisa solucioná-las. Para isso há várias alternativas, e a mais simples é esta: se diversas chaves mapeiam para o mesmo espaço, inicie uma lista encadeada neste espaço. Neste exemplo, tanto a “ameixa” quanto o “abacate” são mapeados para o mesmo espaço. Logo, você deve iniciar uma lista encadeada neste espaço.



# Colisões

Caso você queira saber o preço das bananas, esta informação ainda será acessada de maneira rápida. Porém, se você quiser saber o preço das ameixas, essa informação será retornada de forma mais lenta, pois você precisará pesquisar na sua lista encadeada para encontrar “ameixas”. Se a lista encadeada for pequena, não haverá nenhum problema, pois você deverá pesquisar entre três ou quatro elementos. Mas imagine que você trabalha em um mercado onde são vendidos apenas produtos que iniciam com a letra A.



# Colisões

Ei, espere um pouco aí! Quase toda a tabela hash está vazia, exceto por um espaço, e neste espaço há uma lista encadeada gigante! Ou seja, cada elemento dessa tabela hash está contido nessa lista. Isso é tão ineficiente quanto colocar todos esses elementos apenas na lista encadeada, pois essa lista diminuirá o tempo de execução da sua tabela hash.

## □ **Aprendemos duas lições aqui:**

- A função hash é muito importante. Ela mapeia todas as chaves para um único espaço. Idealmente, a sua função hash mapearia chaves de maneira simétrica por toda a hash.
- Caso as listas encadeadas se tornem muito longas, elas diminuirão demais o tempo de execução da tabela hash. Porém elas não se tornarão muito longas se você utilizar uma boa função hash!

As funções hash são importantes, pois uma boa função hash cria poucas colisões.

# Desempenho

Você iniciou o assunto de tabelas hash em um supermercado, pois era necessário criar algo que fornecesse os preços dos produtos instantaneamente. Bem, as tabelas hash são muito rápidas.

	CASO MÉDIO	PIOR CASO
PROCURA	$O(1)$	$O(n)$
INSERÇÃO	$O(1)$	$O(n)$
REMOÇÃO	$O(1)$	$O(n)$
DESEMPENHO DAS TABELAS HASH		

# Desempenho

No caso médio, as tabelas hash têm tempo de execução  $O(1)$  para tudo. Assim,  $O(1)$  é chamado de tempo constante.

Você ainda não foi apresentado a tempo constante. Tempo constante não é algo que acontece instantaneamente, mas sim um tempo que continuará sempre o mesmo, independentemente de quão grande a tabela hash possa ficar. Por exemplo, você sabe que a pesquisa simples tem tempo de execução linear.

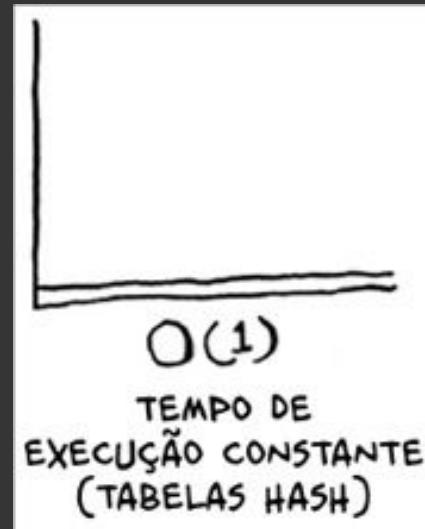


# Desempenho

A pesquisa binária é mais rápida, pois tem tempo de execução log:



A pesquisa binária é mais rápida, pois tem tempo de execução log:



# Desempenho

Percebe como é uma linha reta? Isso significa que não importa se a sua tabela hash tem 1 elemento ou 1 bilhão de elementos, pois o retorno da tabela hash sempre levará a mesma quantidade de tempo. Na verdade, você já viu tempo constante antes, pois retornar um item de um array leva um tempo constante. Novamente, não importa o tamanho do array, pois ele sempre levará a mesma quantidade de tempo para retornar um elemento.

No caso médio, as tabelas hash são muito rápidas. No pior caso, uma tabela hash tem tempo de execução  $O(n)$ , ou seja, tempo de execução linear para tudo, o que é bem lento.

# Desempenho

- Vamos comparar as tabelas hash com os arrays e com as listas.

	TABELAS HASH (CASO MÉDIO)	TABELAS HASH (PIOR CASO)	ARRAYS	LISTAS ENCADEADAS
BUSCA	$O(1)$	$O(n)$	$O(1)$	$O(n)$
INSERÇÃO	$O(1)$	$O(n)$	$O(n)$	$O(1)$
REMOÇÃO	$O(1)$	$O(n)$	$O(n)$	$O(1)$

# Desempenho

Preste atenção ao caso médio para as tabelas hash. As tabelas hash são tão velozes quanto os arrays na busca (pegar um valor em algum índice), e elas são tão velozes quanto as listas na inserção e na remoção de itens. Ou seja, ela é o melhor dos dois mundos! Porém, no pior caso, as tabelas hash são lentas em ambos os casos. Assim, é importante que você não opere no pior caso; para isso é preciso evitar colisões. Para evitar colisões são necessários

- Um baixo fator de carga.
- Uma boa função hash.

# Fator de Carga

O fator de carga de uma tabela hash é simples de calcular.

$$\frac{\text{NÚMERO DE ITENS NA TABELA HASH}}{\text{NÚMERO TOTAL DE ESPAÇOS}}$$

As tabelas hash utilizam um array para armazenamento, então você deve contar o número de espaços usados no array. Por exemplo, esta tabela hash tem fator de carga de 2/5, ou 0.4.



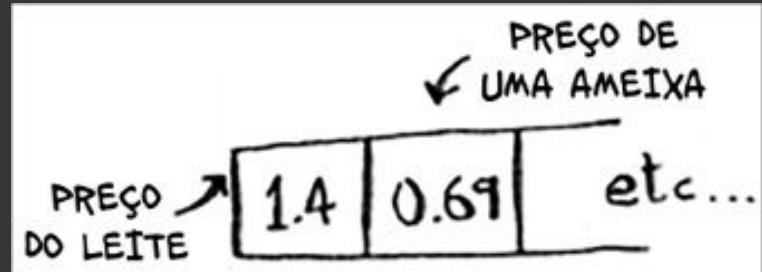
# Fator de Carga

Qual o fator de carga desta tabela hash?



Se você disse um terço, acertou. O fator de carga mede quantos espaços continuam vazios na sua tabela hash.

Suponha que você precise armazenar o preço de cem produtos em sua tabela hash, considerando que ela tenha cem espaços. Na melhor hipótese, cada item terá seu próprio espaço.



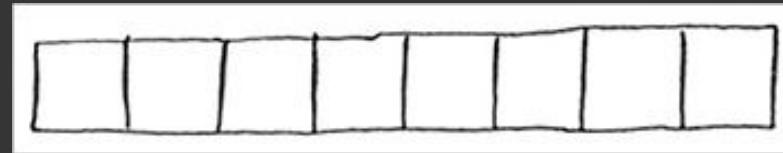
# Fator de Carga

Esta tabela hash tem um fator de carga de 1. E se a tabela hash tivesse apenas cinquenta espaços? Neste caso, ela teria um fator de carga de 2, sendo impossível que cada item tenha o seu próprio espaço, pois não existem espaços suficientes! Um fator de carga maior do que 1 indica que você tem mais itens do que espaços em seu array. Se o fator de carga começar a crescer, será necessário adicionar mais espaços em sua tabela hash. Isso se chama redimensionamento. Suponha, por exemplo, que você tenha esta tabela hash que está quase cheia:



# Fator de Carga

É necessário redimensionar esta tabela hash. Para isso, primeiro você deve criar um array maior. Empiricamente, definiu-se que este array deve ter o dobro do tamanho do array original.



Agora é necessário reinserir todos os itens nesta nova tabela hash utilizando a função hash:

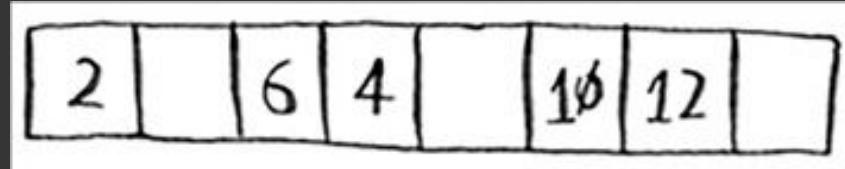


# Uma boa Função Hash

Esta nova tabela tem um fator de carga de  $\frac{3}{8}$  (três oitavos). Bem melhor! Com um fator de carga menor haverá menos colisões e sua tabela terá melhor desempenho.

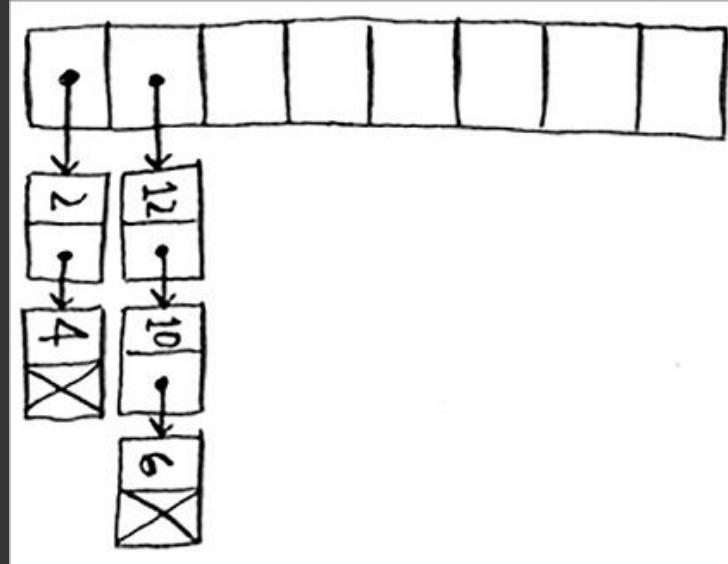
Uma boa regra geral é: redimensionar quando o fator de carga for maior do que 0.7. Você pode estar pensando “Redimensionar leva muito tempo!”, e isto é verdade. O redimensionamento é caro e não deve ser feito com frequência. No entanto, em média, as tabelas hash têm tempo de execução  $O(1)$ , mesmo com o redimensionamento.

- **Uma boa função hash distribui os valores no array simetricamente.**



# Uma boa Função Hash

- Uma função hash não ideal agrupa valores e produz diversas colisões.



# Uma boa Função Hash

Mas o que é uma boa função hash? Isso é algo com que você jamais deverá se preocupar, pois homens (e mulheres) velhos e barbudos sentam em quartos escuros e se preocupam com isso. Se você é muito curioso, dê uma olhada na família de funções SHA.

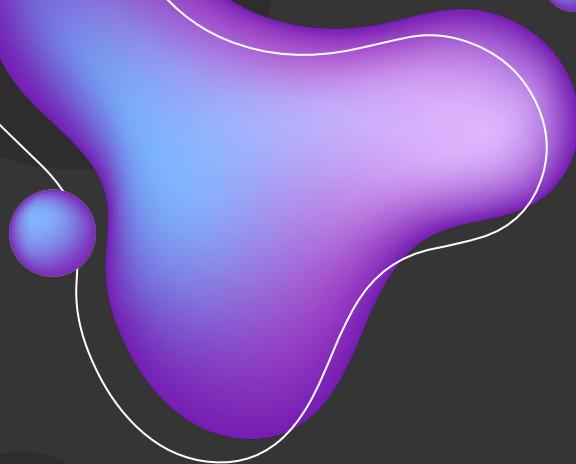
Você provavelmente nunca terá de implementar uma tabela hash, pois a linguagem de programação que você utiliza deve fornecer uma implementação desta funcionalidade.

# Uma boa Função Hash

## □ Concluindo

As tabelas hash são estruturas de dados poderosas, pois elas são muito rápidas e possibilitam a modelagem de dados de uma forma diferente. Logo você estará utilizando-as o tempo todo:

- Você pode fazer uma tabela hash ao combinar uma função hash com um array.
- Colisões são problemas. É necessário haver uma função hash que minimize colisões.
- As tabelas hash são extremamente rápidas para pesquisar, inserir e remover itens.
- Se o seu fator de carga for maior que 0.7, será necessário redimensionar a sua tabela hash.



10

# Árvores

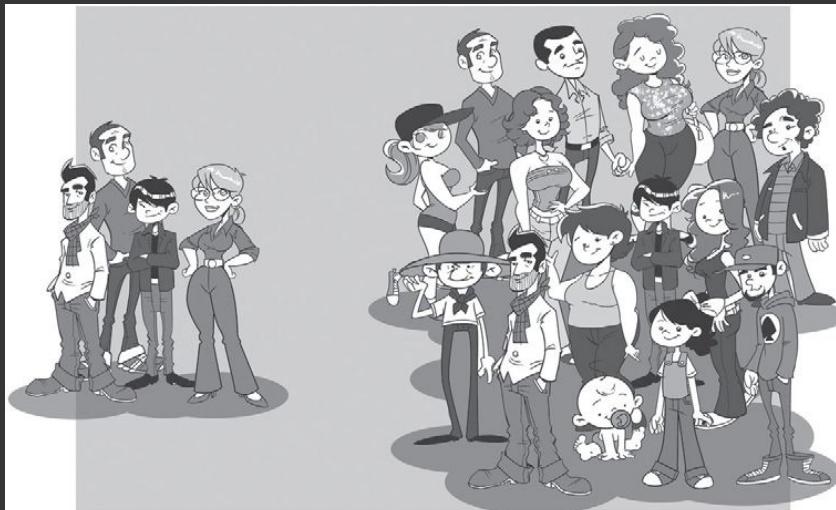
A estrutura de dados mais importante da computação



# Árvores

**Árvores** são estruturas em que os dados são dispostos de forma hierárquica. Nelas, os dados são armazenados em nós. Existe um nó principal, conhecido como raiz, a partir do qual surgem as ramificações, conhecidas como subárvore.

- Sua tarefa: localizar o José da Silva e Souza nas imagens apresentadas.



# Árvores

Observando a primeira imagem, percebemos que não seria difícil localizar determinada pessoa dentre as ali apresentadas. Essa tarefa seria realizada com facilidade e em curto período de tempo. Por outro lado, o mesmo não pode ser dito em relação à segunda imagem. Localizar uma pessoa no meio da multidão não é tarefa fácil e nem mesmo poderia ser atendida num curto período de tempo.

Precisaríamos, de alguma forma, estabelecer uma organização na multidão para que a tarefa proposta fosse viável num tempo aceitável. Sabemos que uma das formas de organização mais antigas é a hierárquica. Na organização hierárquica da multidão poderíamos estabelecer critérios que permitissem agrupar as pessoas de acordo com determinadas características. Isso reduziria sensivelmente o espaço de busca, uma vez que não seria necessária a busca nos agrupamentos cujos integrantes não tivessem as características da pessoa procurada.

# Árvores

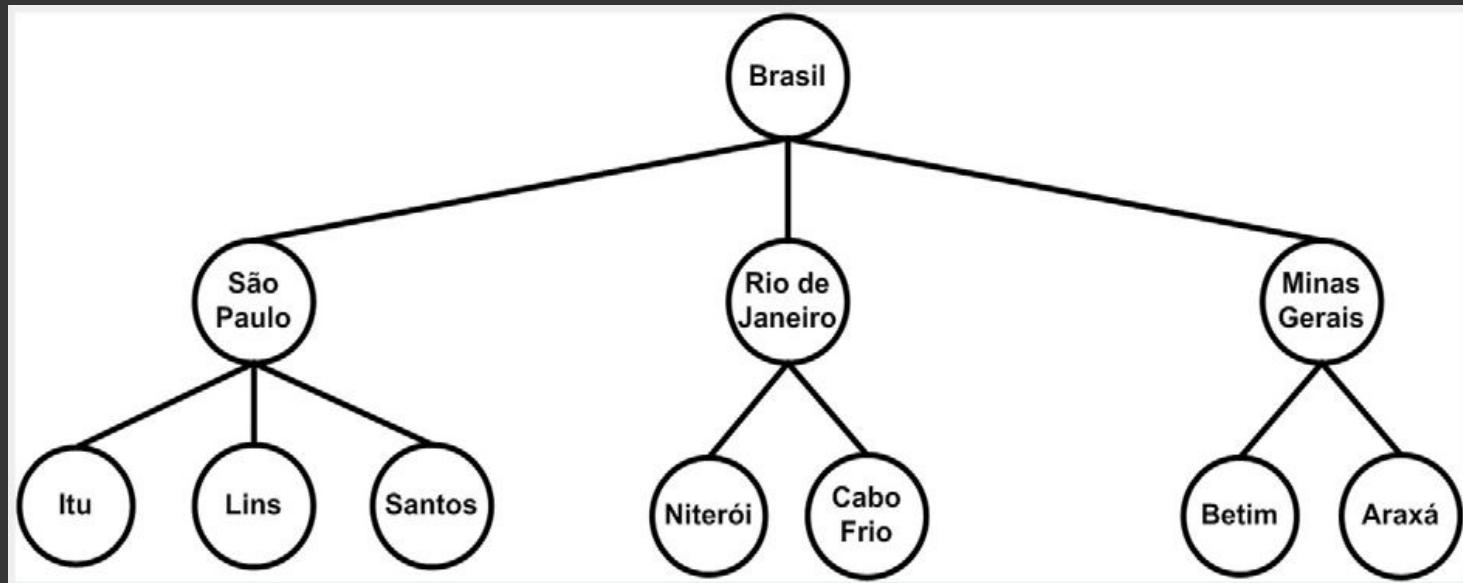
Foi nesse contexto que surgiu a estrutura de dados denominada árvore.

Nessa estrutura existe um elemento principal, que é o nó raiz. Pode haver outros nós associados a ele, os quais seriam seus filhos ou descendentes. Os nós filhos formam subárvores do nó pai ou antecessor. Essas subárvores apresentam a mesma estrutura de árvore definida, ou seja, os nós filhos do nó raiz poderão ter seus próprios descendentes, e assim por diante.

Para ilustrar esse conceito, a figura a seguir mostra a representação gráfica de uma árvore contendo nomes do nosso país, de alguns de seus estados e algumas cidades. Nessa figura temos como nó raiz aquele que contém o nome Brasil.

# Árvores

Como nós filhos temos os estados, que, por sua vez, têm como filhos os nós contendo os municípios.



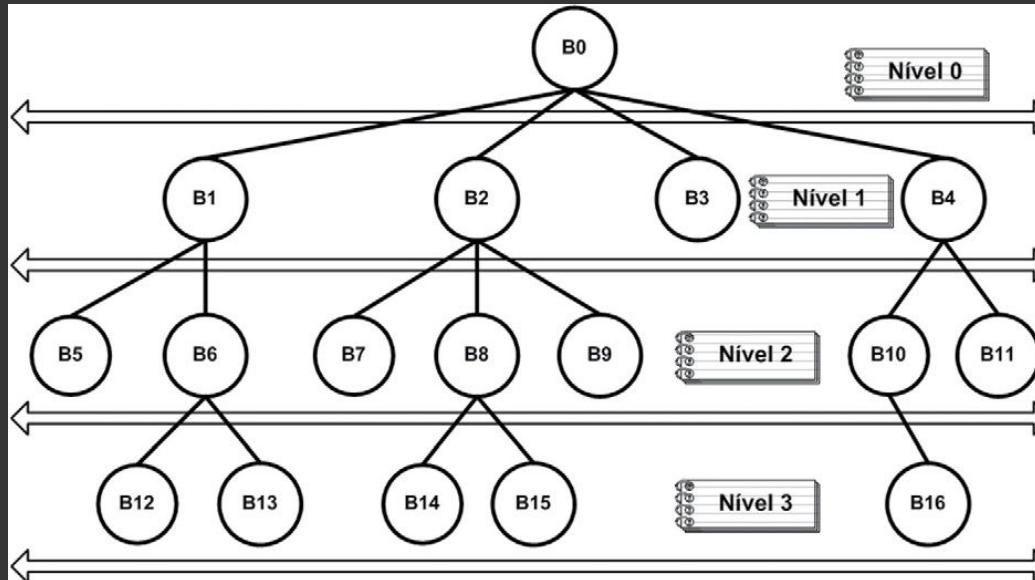
# Árvores

## □ Conceitos

- **Nó raiz:** principal elemento da árvore, não apresenta ancestrais, e todos os nós da árvore são seus descendentes (diretos ou indiretos);
- **Nó pai:** elemento que apresenta descendentes na árvore, podendo ter um ou mais filhos;
- **Nó filho:** elemento que descende de algum outro nó da árvore, tendo apenas um nó pai;
- **Nó folha:** nó que não apresenta descendentes;

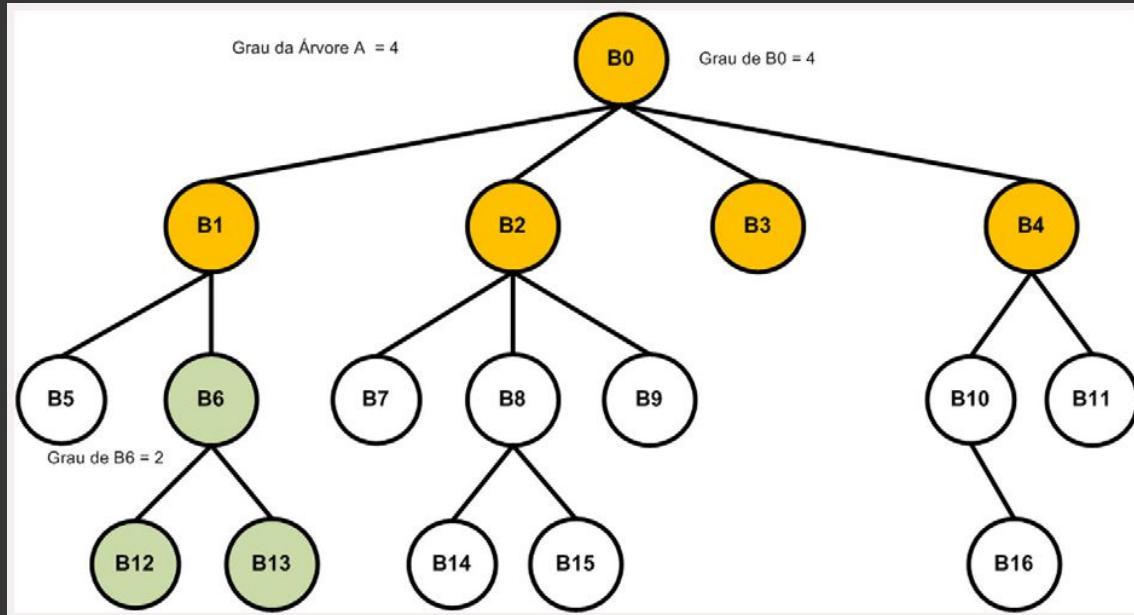
# Árvores

- **Nível de um nó:** o nível do nó raiz é 0 ( $N = 0$ ), e o nível dos nós restantes é igual ao nível do seu nó pai acrescido de 1. Para exemplificar, na Figura abaixo, o nível de B0 é 0, B2 é 1, B5 é 2, e B16 é 3;



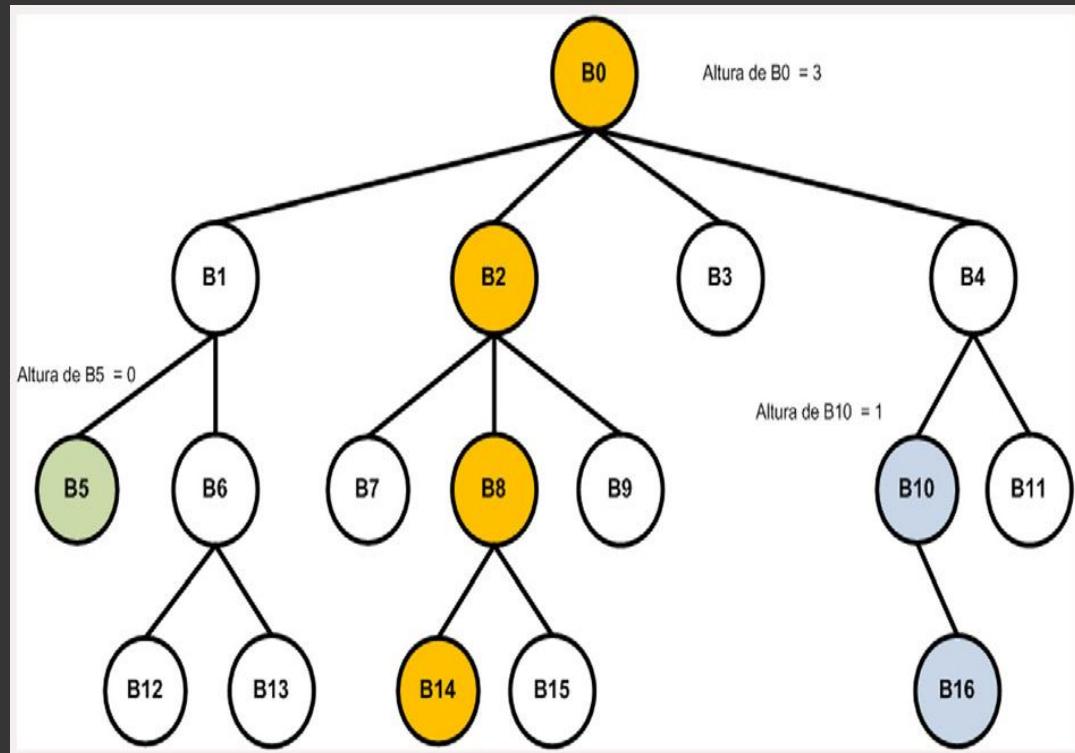
# Árvores

- **Grau de um nó:** o grau de determinado nó é dado pelo número de seus filhos. Para exemplificar, na figura abaixo, o grau do nó B0 é 4, e o do nó B6 é 2;



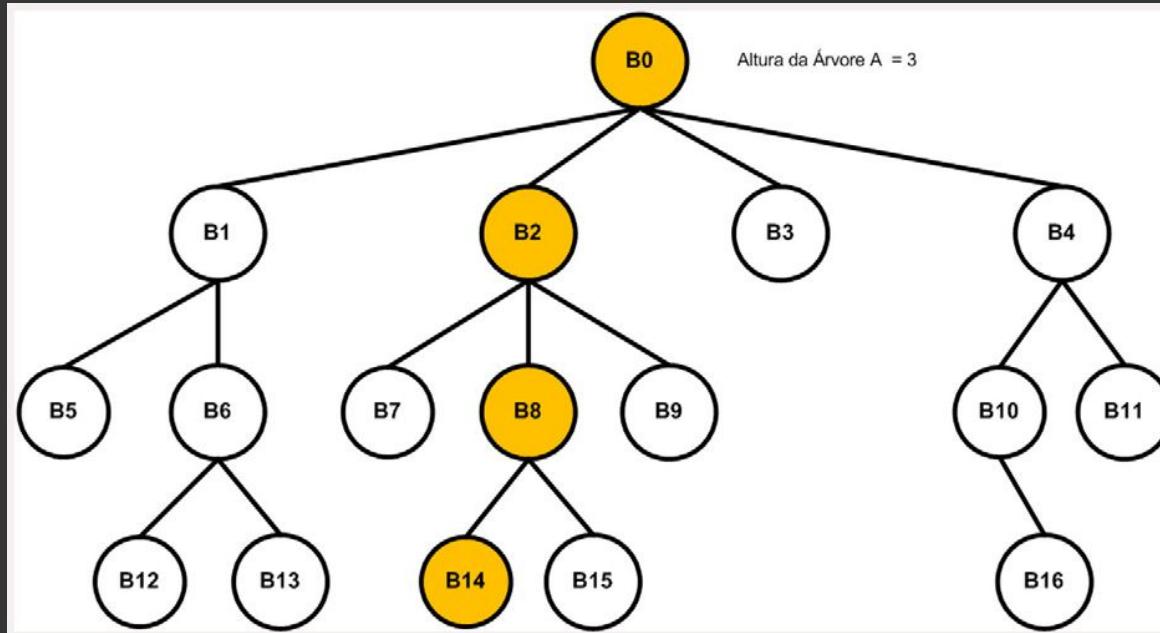
# Árvores

- **Grau da árvore:** uma árvore A tem grau igual ao grau máximo verificado para seus nós, ou seja, é igual ao do nó que apresenta mais filhos. Na árvore da Figura acima temos o grau da árvore A igual a 4;
- **Altura de um nó:** a altura de um nó  $B_i$  é igual à maior distância entre  $B_i$  e um nó folha que seja seu descendente. Para exemplificar, na Figura ao lado a altura de  $B_0$  é 3, a de  $B_5$  é 0, e a de  $B_{10}$  é 1;



# Árvores

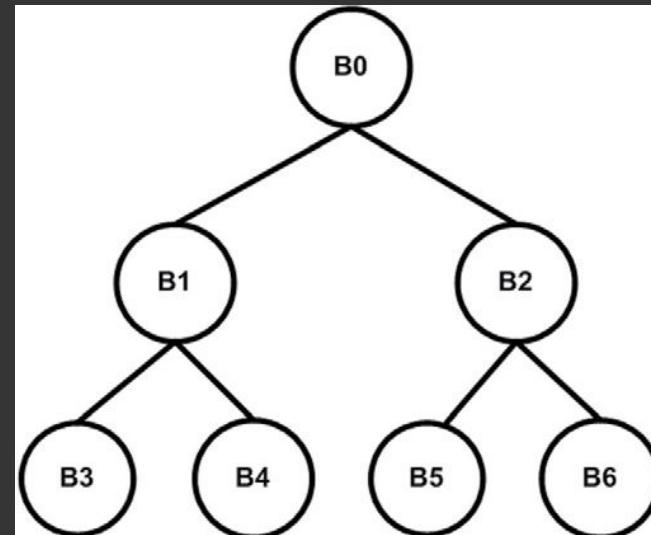
- **Altura de uma árvore:** a altura de uma árvore corresponde à altura do nó raiz. Na árvore da figura abaixo temos a altura da árvore A igual a 3.



# Árvores Binária

Numa estrutura de dados de árvores, quando restringimos a dois o número máximo de filhos para um nó, temos uma árvore binária. Uma árvore binária é caracterizada como um conjunto finito vazio (ou não) de nós. Esses nós são apresentados na forma de um nó raiz e seus descendentes, organizados em uma subárvore esquerda e uma subárvore direita.

- **Para melhor compreensão, observe a árvore binária ao lado**



# Árvores Binária

Nessa árvore temos B0 como nó raiz, B1 e B2 como nós filhos de B0, e assim por diante. Verificamos, também, que cada nó filho é um nó raiz da subárvore gerada a partir dele.

As árvores binárias herdam todos os conceitos existentes para a estrutura de dados árvore já vista, como: tipos de nó, nível de um nó ou árvore, grau de um nó ou árvore, e altura de um nó ou árvore. Da mesma forma, são aplicáveis as formas de representação estudadas.

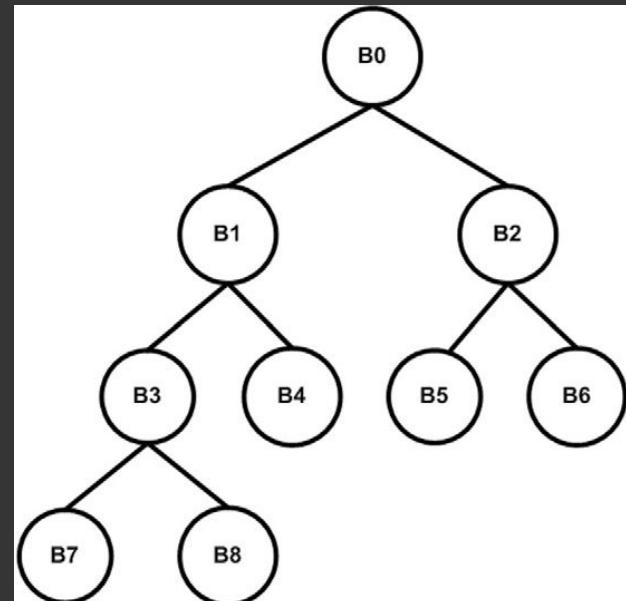
Ao considerarmos que em determinado nível  $N$  de uma árvore binária temos  $K$  nós, teremos no máximo  $2^N K$  nós no nível imediatamente subsequente ( $N + 1$ ). Dessa forma, como temos apenas 1 nó no nível 0, teremos no máximo 2 nós no nível 1, 4 no nível 2, e assim por diante.

# Árvores Binária

- Para melhor compreensão, observe a árvore binária ao lado

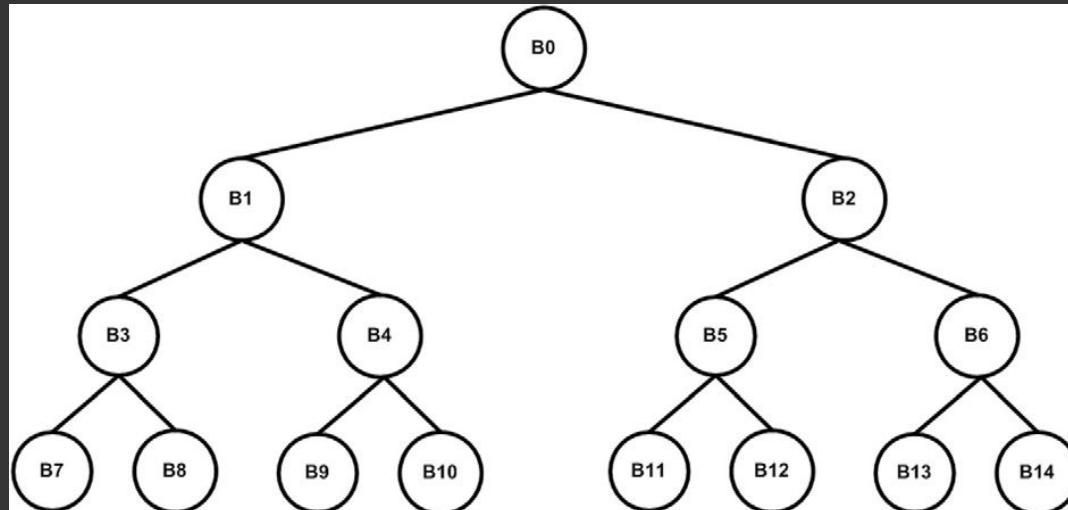
Dependendo da distribuição dos nós em uma árvore binária, teremos as seguintes classificações: árvore estritamente binária, árvore binária completa e árvore binária quase completa.

- **Árvore estritamente binária:** Nas situações em que, para todos os nós que não sejam nós folhas, não existem subárvore vazias, a árvore será denominada árvore estritamente binária. Um exemplo pode ser visto na Figura ao lado.



# Árvores Binária

- **Árvore binária completa:** Quando observamos a árvore da figura anterior, percebemos que nem todos os nós folhas apresentam o mesmo nível. Por outro lado, é possível que tenhamos uma árvore estritamente binária em que todos os nós folhas estejam no mesmo nível. Nessa situação temos uma árvore binária completa.

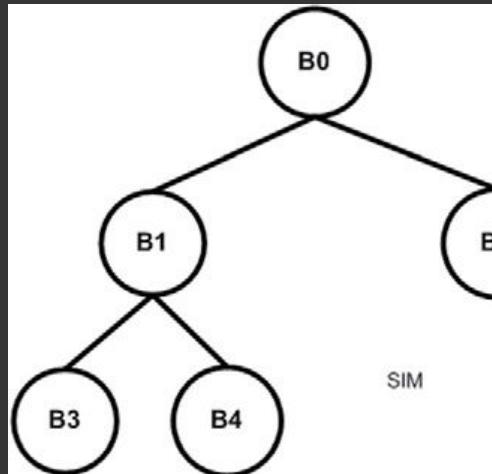


# Árvores Binária

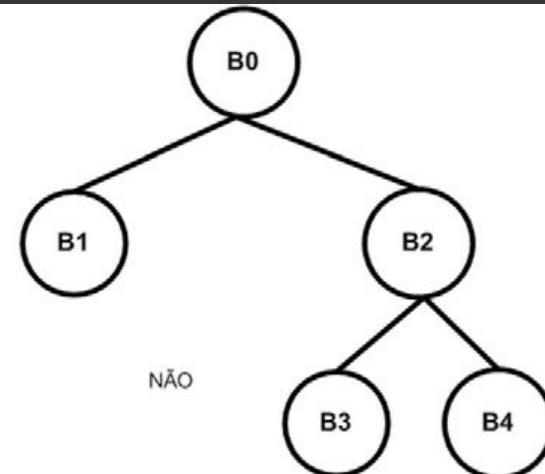
- **Árvore binária quase completa:** Uma árvore binária que atender às condições a seguir será considerada árvore binária quase completa:
  - Todos os nós folhas estão no nível N ou N-1;
  - Para todo nó  $B_n$  que possuir um descendente direito no nível N (nível máximo da árvore), todo descendente esquerdo de  $B_n$  deverá ser nó folha no nível N.

# Árvores Binária

Para compreender melhor, observe a figura abaixo. Na árvore (a) temos a caracterização de uma árvore binária quase completa, já na árvore (b) isso não ocorre, ou seja, ela não é uma árvore binária quase completa, por não atender à condição 2.



(a)



(b)

# Árvores Binária

## □ **Operações básicas numa árvore binária**

A manipulação de uma árvore binária se dá através de diferentes operações possíveis. Neste conteúdo, vamos focar as operações básicas de maior utilidade e difusão: percurso (também conhecido como travessia ou varredura), inserção e remoção.

Antes de começarmos a discutir e apresentar essas operações, necessitamos apresentar formas de declarar e criar uma árvore binária, que, como descrito, pode ser feita de forma estática ou de forma dinâmica. A forma estática consiste em apenas declarar um vetor e manipulá-lo como descrito no item “Implementação com Alocação Estática”, porém, devido ao alto desperdício de memória provocado por essa forma, ela não é a mais utilizada. Resta-nos, portanto, fazer uso da forma dinâmica, que, deste ponto em diante, será a forma padrão adotada.

# Árvores Binária

## □ Declaração de uma árvore binária

Consiste na definição do nó por meio de um registro. Os campos que compõem esse registro são: dado, que armazena o(s) elemento(s) de dado(s) do nó; que armazena a ligação com o nó filho à esquerda; e que armazena a ligação com o nó filho à direita e o valor do nó.

```
#region Propriedades da Classe
public int valor { get; set; }
public No esquerda { get; set; }
public No direita { get; set; }
#endregion

#region Construtor
public No(int valor)
{
    this.valor = valor;
    this.esquerda = null;
    this.direita = null;
}
#endregion
```

# Árvores Binária

## □ **Percorso de árvore binária**

Quando desejamos apresentar todos os elementos de uma árvore binária (independentemente de existir ou não alguma ordenação interna entre os elementos), é necessário estabelecermos um padrão para o percurso da árvore. No caso das árvores binárias, encontramos três formas básicas de percurso: **pré-ordem, em-ordem e pós-ordem**.

A distinção entre as formas de percurso se dá pela ordem em que os nós são visitados.

# Árvores Binária

## □ **Percorso em pré-ordem**

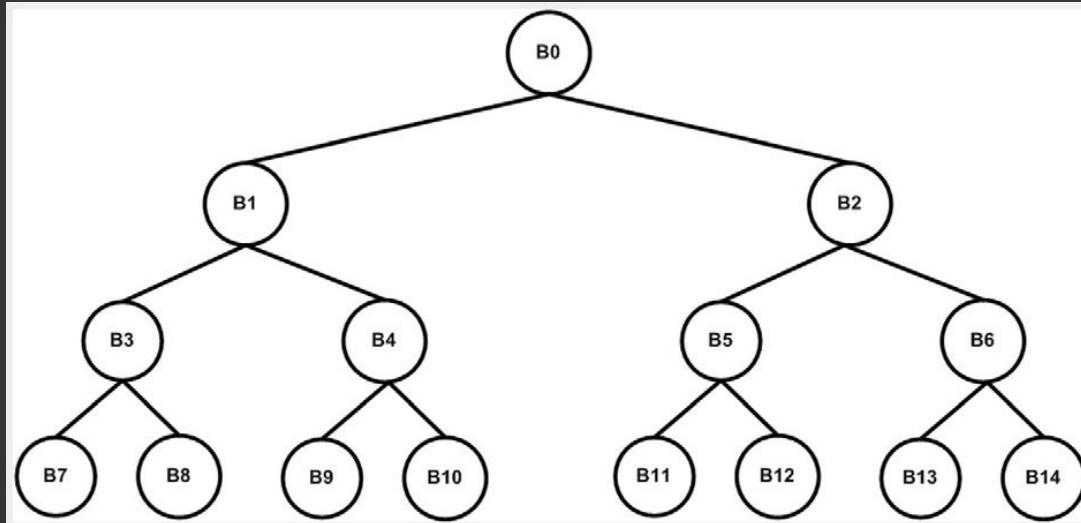
Nessa forma de percurso, os nós de uma árvore binária são visitados de forma recursiva na seguinte ordem:

- apresenta o elemento do nó visitado;
- passa para o elemento do nó filho à esquerda (subárvore à esquerda);
- passa para o elemento do nó filho à direita (subárvore à direita).

# Árvores Binária

Para ilustrar, tomemos novamente o exemplo da árvore binária apresentada na figura abaixo e vejamos como ficaria a ordem de apresentação dos elementos dessa árvore num percurso pré-ordem.

**B0-B1-B3-B7-B8-B4-B9-B10-B2-B5-B11-B12-B6-B13-B14**



# Árvores Binária

A seguir, veja a implementação da função **TravessiaPreOrdem**, que apresenta como saída os elementos da árvore binária, segundo os critérios já descritos.

```
#region Travessia Pré Ordem
// # raiz, esquerda, direita
2 references
public void TravessiaPreOrdem(No no)
{
    var NaoForNoFolha = no != null;

    if(NaoForNoFolha)
    {
        System.Console.WriteLine(no.valor);
        this.TravessiaPreOrdem(no.esquerda);
        this.TravessiaPreOrdem(no.direita);
    }
}
#endregion
```

# Árvores Binária

## □ **Percorso em em-ordem**

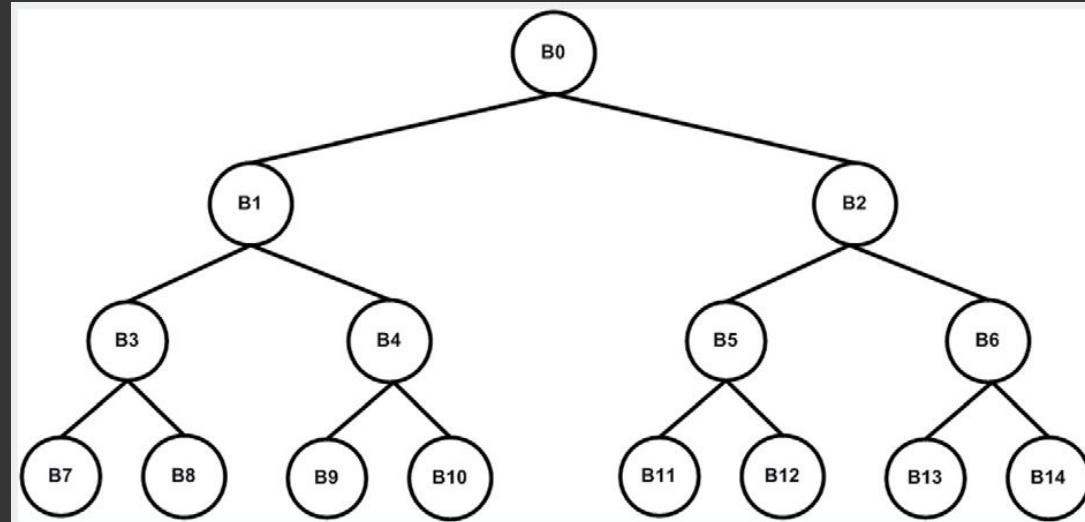
Nessa forma de percurso, os nós de uma árvore binária são visitados de forma recursiva na seguinte ordem:

- passa para o elemento do nó filho à esquerda (subárvore à esquerda);
- apresenta o elemento do nó visitado;
- passa para o elemento do nó filho à direita (subárvore à direita).

# Árvores Binária

Reproduzindo novamente a árvore da figura abaixo, vejamos como ficaria a ordem de apresentação dos elementos dessa árvore num percurso em-ordem.

**B7-B3-B8-B1-B9-B4-B10-B0-B11-B5-B12-B2-B13-B6-B14**



# Árvores Binária

A seguir, veja a implementação da função **TravessiaEmOrdem**, que apresenta como saída os elementos da árvore binária, segundo os critérios já descritos.

```
#region Travessia Em Ordem
// # esquerda, raiz, direita
2 references
public void TravessiaEmOrdem(No no)
{
    var NaoForNoFolha = no != null;

    if(NaoForNoFolha)
    {
        this.TravessiaEmOrdem(no.esquerda);
        System.Console.WriteLine(no.valor);
        this.TravessiaEmOrdem(no.direita);
    }
}
#endregion
```

# Árvores Binária

## □ **Percorso em pós-ordem**

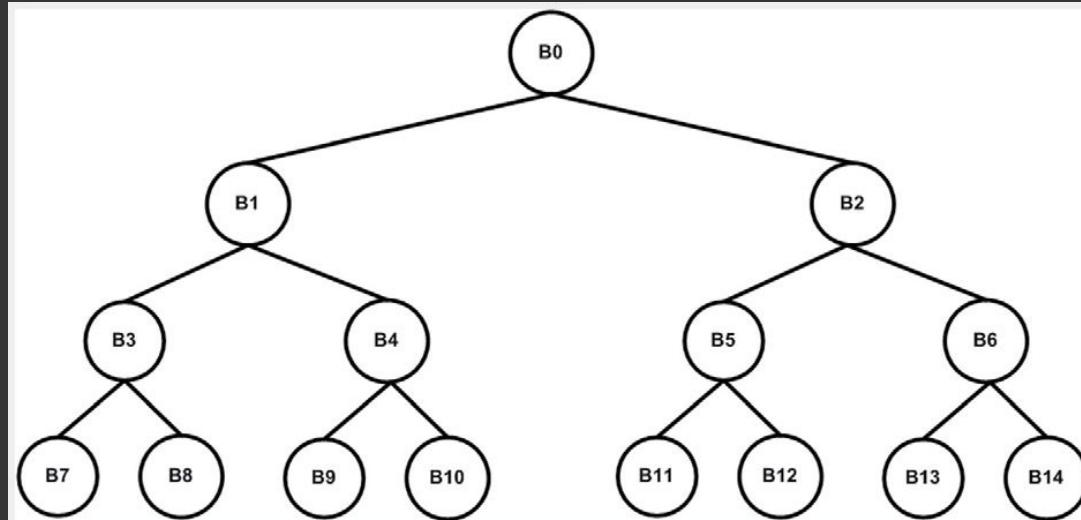
Nessa forma de percurso, os nós de uma árvore binária são visitados de forma recursiva na seguinte ordem:

- passa para o elemento do nó filho à esquerda (subárvore à esquerda);
- passa para o elemento do nó filho à direita (subárvore à direita);
- apresenta o elemento do nó visitado.

# Árvores Binária

Usando mais uma vez a reprodução da árvore da figura abaixo, vejamos como ficaria a ordem de apresentação dos elementos dessa árvore num percurso pós-ordem.

**B7-B8-B3-B9-B10-B4-B1-B11-B12-B5-B13-B14-B6-B2-B0**



# Árvores Binária

A seguir, veja a implementação da função **TravessiaPosOrdem**, que apresenta como saída os elementos da árvore binária, segundo os critérios já descritos.

```
#region Travessia Pós Ordem
// # esquerda, direita, raiz,
2 referências
public void TravessiaPosOrdem(No no)
{
    var NaoForNoFolha = no != null;

    if(NaoForNoFolha)
    {
        this.TravessiaPosOrdem(no.esquerda);
        this.TravessiaPosOrdem(no.direita);
        System.Console.WriteLine(no.valor);
    }
}
#endregion
```

# Árvores Binária

## □ Inserção numa árvore binária

Quando abordamos a inserção numa árvore binária, é possível adotar diferentes formas de preenchimento. Um exemplo seria inserir novos elementos da esquerda para a direita, preenchendo a árvore por nível, ou seja, escolhendo sempre o primeiro nó livre (vazio) mais à esquerda no nível ainda não totalmente preenchido. A dificuldade, nessa forma de preenchimento, é que não existe qualquer organização dos dados (ordenação) que permita, posteriormente, uma busca mais eficiente por determinado elemento pertencente à árvore.

Assim, para se usar a árvore binária como estrutura que auxilie na busca (árvore binária de busca), adotou-se que, para cada elemento da árvore, existe uma chave associada ao mesmo, além dos dados secundários desse elemento. Essa chave será a responsável por promover a ordenação necessária entre os elementos da árvore.

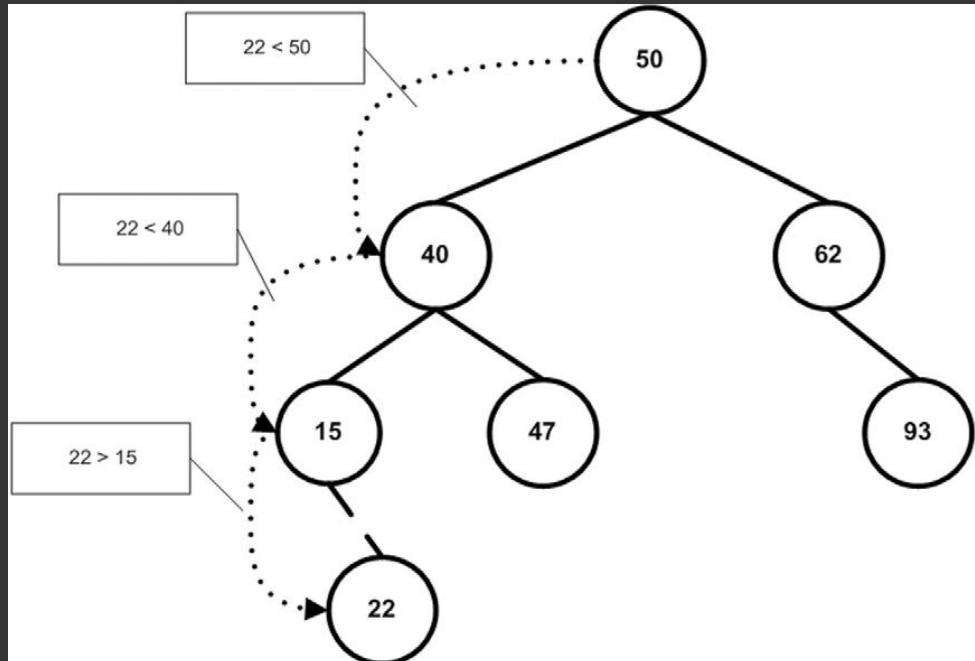
# Árvores Binária

No contexto de árvores binárias de busca, adotou-se também que os elementos com chaves menores que a contida num elemento que está armazenado no nó N da árvore serão inseridos como descendentes à esquerda de N na árvore. Consequentemente, os elementos com chaves maiores serão inseridos como descendentes à direita de N.

Para facilitar a exemplificação dos conceitos, neste conteúdo optamos por utilizar apenas o campo chave para caracterizar o elemento que será armazenado na árvore, desconsiderando a presença de dados secundários do elemento. Como consequência, temos que no campo dado (do registro que caracteriza um nó da árvore) vamos armazenar um valor que será também a própria chave de ordenação.

# Árvores Binária

Consideremos o exemplo da figura ao lado, no qual desejamos inserir a chave (valor) 22 na árvore binária apresentada. Notamos que, na localização do ponto para inserção de 22, primeiro vemos que 22 é menor que 50 e, a seguir, que 22 é menor que 40 (subárvore à esquerda); por fim, vemos que 22 é maior que 15 (subárvore à direita). Como 15 não apresenta subárvore à direita, 22 é inserido como filho à direita de 15.



# Árvores Binária

A seguir, veja a implementação da função **Inserir**, que insere a chave valor na árvore binária.

```
#region Inserir
public void Inserir(int valor)
{
    var novo = new No(valor);

    if(this.raiz == null)
        this.raiz = novo;
    else
    {
        InserirNovoElemento(novo, valor);
    }
}
#endregion
```

```
private void InserirNovoElemento(No novo, int valor)
{
    var atual = this.raiz;

    while (true)
    {
        var pai = atual;

        if(valor < atual.valor)
        {
            if(atual.esquerda == null)
            {
                pai.esquerda = novo;
                return;
            }
        }
        else
        {
            if(atual.direita == null)
            {
                pai.direita = novo;
                return;
            }
        }
    }
}
```

# Árvores Binária

## □ **Remoção numa árvore binária**

Para apresentar a operação de remoção numa árvore binária, adotaremos o mesmo padrão verificado na inserção, ou seja, nós filhos à esquerda contêm chaves menores do que a encontrada no nó pai, consequentemente, nós filhos à direita contêm chaves maiores que a existente no nó pai.

Diferentemente da inserção, a remoção traz dificuldades adicionais para sua operação. Quando removemos um elemento, para que a organização prévia seja mantida, torna-se necessária a reorganização da árvore binária.

# Árvores Binária

Dependendo da posição do elemento removido, diferentes ações podem ser necessárias. De forma geral, há três situações possíveis quanto ao nó em que se encontra o elemento a ser removido:

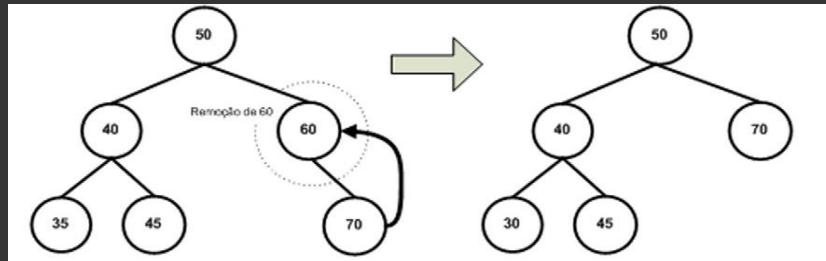
- **nó não apresenta subárvore à esquerda:** basta fazer com que o nó filho à direita passe a ser o nó pai;
- **nó não apresenta subárvore à direita:** basta fazer com que o nó filho à esquerda passe a ser o nó pai;
- **nó apresenta subárvore à esquerda e à direita:** deve ser deslocado, para a posição em que se encontra o nó Bi a ser removido, o nó com o elemento de maior chave da subárvore à esquerda de Bi.\*

# Árvores Binária

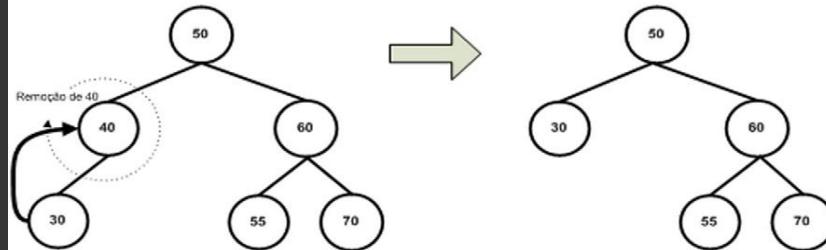
## □ Remoção numa árvore binária

Para exemplificar, observemos os exemplos da figura abaixo. Na parte (a) temos a remoção da chave (valor) 60 correspondendo à situação (1), descrita anteriormente. Na parte (b), a remoção da chave 40 corresponde à situação (2) e, por fim, na parte (c), a remoção da chave 38 corresponde à situação (3).

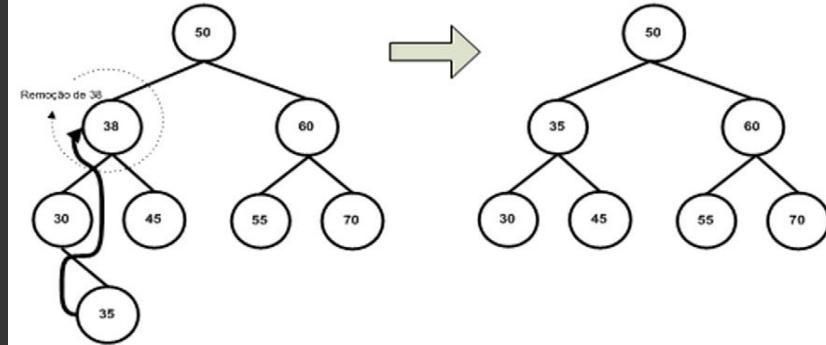
A seguir são apresentadas as funções que permitem a remoção de um elemento da árvore binária. A função `Remover` é a responsável por localizar e remover o elemento desejado.



(a)



(b)



# Árvores Binária

Veja a implementação da função **Remover**, que remove a chave valor na árvore binária.

```
#region Remover
0 references
public void Remover(int valor)
{
    #region Objetos ...
    #region Arvore Vazia ...
    #region Encontrar o Nó ...
        #region Remover Nó ...
    }
#endregion
```

# Árvores Binária

```
#region Objetos
var atual = this.raiz;
var pai = this.raiz;
var EhEsquerda = true;
#endregion

#region Arvore Vazia
if(this.raiz == null)
{
    System.Console.WriteLine("A árvore está vazia");
    return;
}
#endregion
```

```
#region Encontrar o Nó
while (atual.valor != valor)
{
    pai = atual;

    if(valor < atual.valor)
    {
        EhEsquerda = true;
        atual = atual.esquerda;
    }
    else
    {
        EhEsquerda = false;
        atual = atual.direita;
    }

    if (atual == null)
        return;
}
#endregion
```

```
// # Nó Folha
if (atual.esquerda == null && atual.direita == null)
{
    #region Nó folha
    if(atual == this.raiz)
    {
        this.raiz = null;
    }
    else if(EhEsquerda)
    {
        RemoverLigacaoNoFolha(pai, atual);
        pai.esquerda = null;
    }
    else
    {
        RemoverLigacaoNoFolha(pai, atual);
        pai.direita = null;
    }
    #endregion
}
```

```
//# Nó (1 filho)
else if(atual.direita == null)
{
    #region Filho está na Esquerda
    var EhUltimo = atual == this.raiz;

    if(EhUltimo)
        this.raiz = atual.esquerda;
    else if (EhEsquerda)
        pai.esquerda = atual.esquerda;
    else
        pai.direita = atual.esquerda;
    #endregion
}
else if(atual.esquerda == null)
{
    #region Filho está na Direita
    var EhUltimo = atual == this.raiz;

    if(EhUltimo)
        this.raiz = atual.direita;
    else if (EhEsquerda)
        pai.esquerda = atual.direita;
    else
        pai.direita = atual.direita;
    #endregion
}
```

```
//# Nó (2 filho)
else
{
    #region Nó (2 filho)
    var sucessor = GetSucessor(atual);

    if(atual == this.raiz)
        this.raiz = sucessor;
    else if(EhEsquerda)
        pai.esquerda = sucessor;
    else
        pai.direita = sucessor;

    sucessor.esquerda = atual.esquerda;
    #endregion
}
```

# Árvores Binária

## □ Eficiência na busca numa árvore binária

As operações básicas numa árvore binária (de busca) tem tempo proporcional à sua altura. Como a altura da árvore dependerá da quantidade N de chaves e de sua ordem de inserção na árvore, o tempo de resposta das operações básicas dependerá da quantidade e da distribuição das chaves pelas subárvores (subárvores com diferentes alturas). No pior caso, em termos de altura da árvore binária, teríamos as inserções das chaves de forma que a altura da árvore fosse N-1, ou seja, um tempo de execução das operações básicas O(N).

Numa situação ideal de distribuição de N chaves pelas subárvores (árvore binária completa), porém, as operações básicas, no pior caso, seriam executadas num tempo O(logN).

# Árvores Balanceadas

Quando armazenamos dados em uma estrutura de dados árvore, se não houver preocupação com a distribuição homogênea desses dados pelos galhos (subárvores), podemos ter árvores que crescem apenas para um lado, o que traz prejuízos quando se imagina utilizá-las como estruturas que ofereçam maior eficiência na busca e na recuperação de dados.

Agora que já vimos o funcionamento básico da estrutura de dados de árvores, bem como as vantagens que essa estrutura traz quando utilizada em tarefas de busca. Foi possível também entender que árvores com altura menor possibilitam, em média, uma redução no número de acessos aos nós necessários para se localizar um elemento ali armazenado.

Assim, agora nosso objetivo passará a ser estudar árvores nas quais é mantida uma distribuição homogênea dos elementos armazenados pelas subárvores. Para isso, vamos conceituar o balanceamento em estrutura de dados em árvore e apresentar o funcionamento das árvores binárias平衡adas (aqui representadas pelas árvores AVL).

# Árvores Balanceadas

## □ Árvores Binárias Balanceadas: Árvores AVL

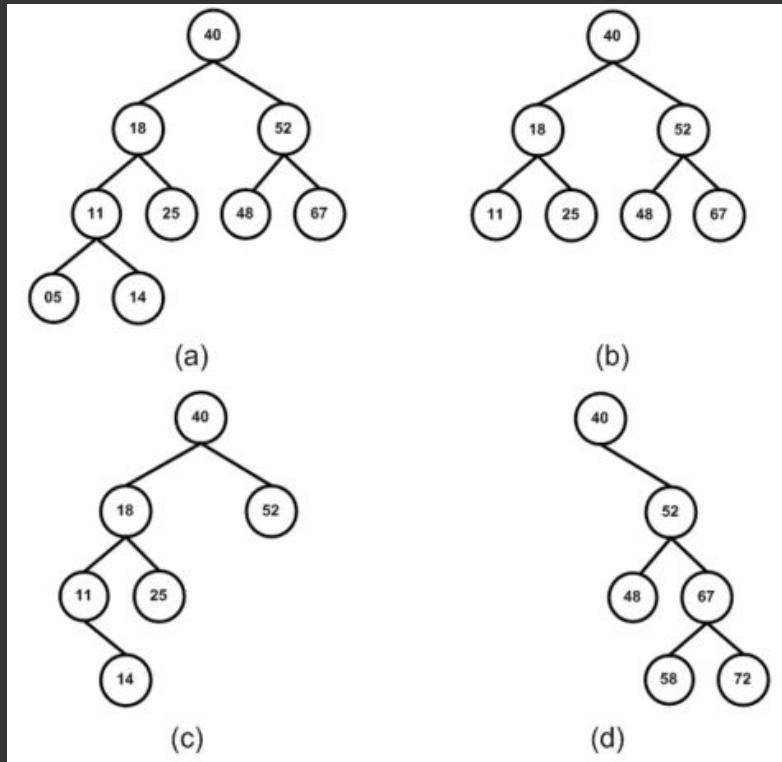
Todos os conceitos e definições de árvores binárias vistos anteriormente valem para as árvores binárias平衡adas (que, daqui em diante, serão tratadas apenas como árvores AVL). As árvores AVL são árvores binárias em que a distribuição dos elementos é feita respeitando determinadas condições que vão garantir o balanceamento dessa árvore. Numa árvore AVL, o balanceamento é definido a partir das alturas das subárvores nela existentes, uma curiosidade é que o termo AVL é proveniente dos nomes de seus criadores, os matemáticos russos Georgy Adelson-Velsky e Yevgeniy Landis.

Nesse tipo de árvore, a diferença entre as alturas das subárvores esquerda e direita de qualquer nó é de no máximo 1, ou seja, se a altura da subárvore esquerda é  $N$ , então, a altura da subárvore direita será igual a  $N$ ,  $N-1$  ou  $N+1$ .

# Árvores Balanceadas

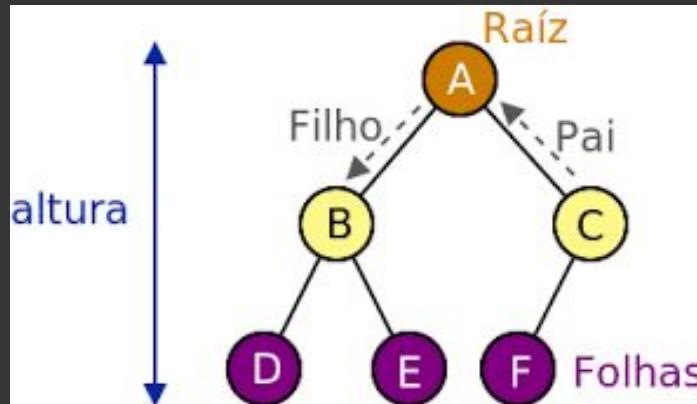
Para ilustrar, vejamos as árvores da figura ao lado.

Tanto a árvore (a) quanto a árvore (b) são árvores AVL, uma vez que as alturas das subárvore esquerda e direita nunca diferem de 1 entre elas. Já as árvores (c) e (d) não são árvores AVL, pois na árvore (c) a altura da subárvore esquerda do nó raiz difere de 2 da altura de sua subárvore direita, e na árvore (d) a altura da subárvore direita do nó raiz difere de 3 da altura de sua subárvore esquerda.



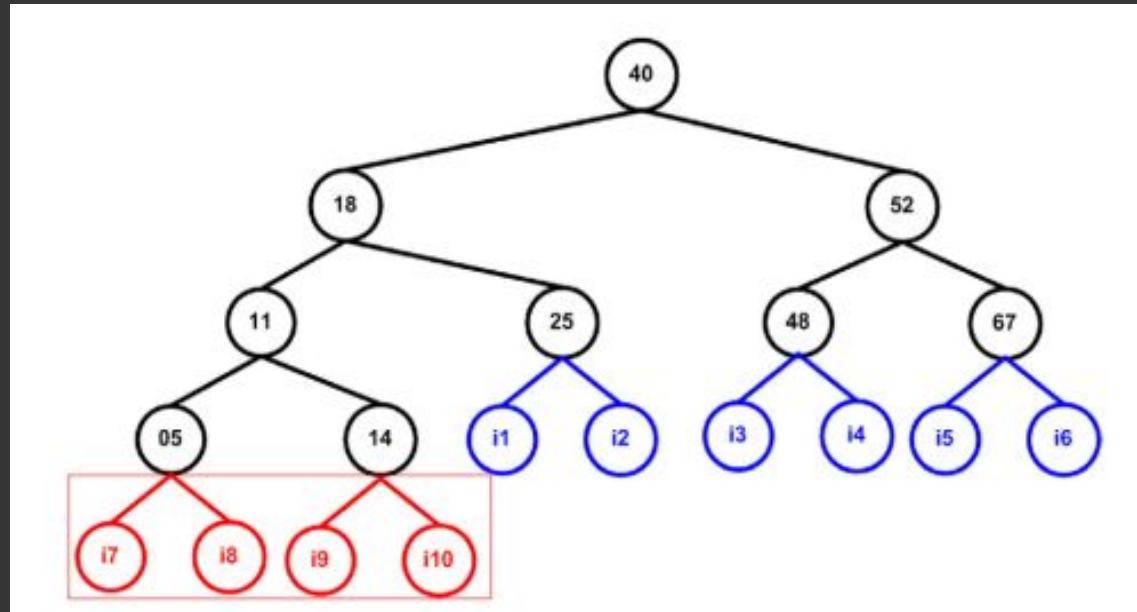
# Árvores Balanceadas

Embora as operações básicas de percurso (busca), inserção e remoção em árvores AVL sejam as mesmas de uma árvore binária (busca) qualquer, para que seja garantido o balanceamento, torna-se necessário que, após a inserção ou remoção de um elemento, seja verificado se a condição de平衡amento por altura se mantém e se a ordenação das chaves não é afetada.



# Árvores Balanceadas

Na figura abaixo é possível observar quais inserções, nessa árvore, manteriam o balanceamento.



# Árvores Balanceadas

É possível verificar que as inserções individuais i1, i2, i3, i4, i5 ou i6 levariam a situações em que o balanceamento seria mantido. Por outro lado, as inserções individuais i7, i8, i9 ou i10, provocariam um desbalanceamento na árvore.

Agora vamos conferir os valores que manteriam a árvore balanceada e quais levariam ao desbalanceamento em casos de exclusão dos valores, conferindo os resultados a seguir:

- **Remoções que não promoveriam o desbalanceamento: 05-11-14-18-48-52-67**
- **Remoções que promoveriam o desbalanceamento: 25-40**

Pelo exemplo, percebemos que nas situações em que se tem o desbalanceamento da árvore a partir de uma inserção ou remoção, algo precisa ser feito para restabelecer a diferença de 1 entre as alturas das subárvore esquerda e direita dos nós da árvore, sem que a ordenação das chaves seja afetada.

# Árvores Balanceadas

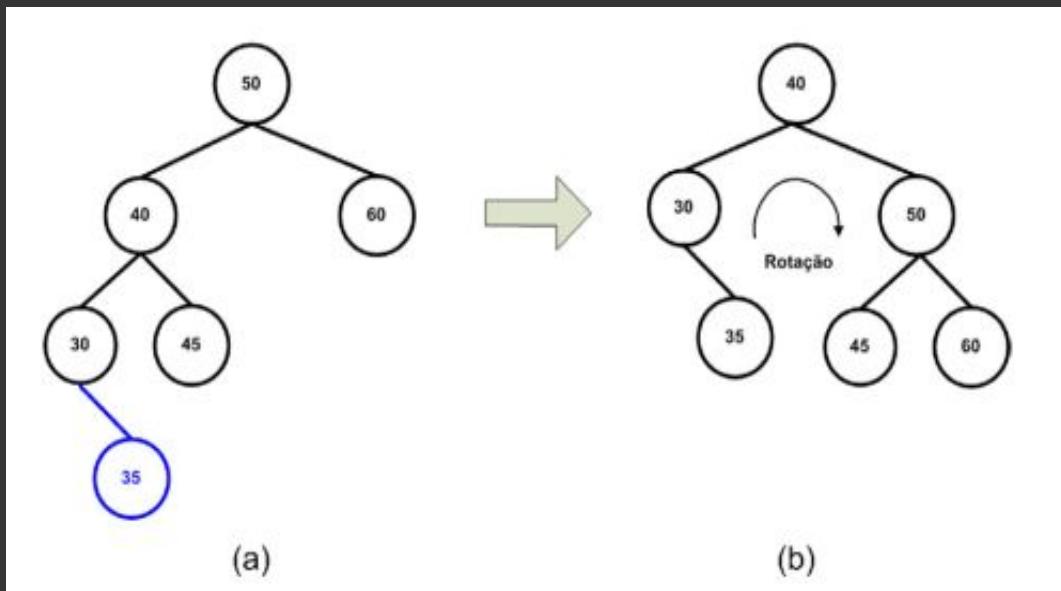
É aqui que entra o conceito de rotação de nós numa árvore, que objetiva o restabelecimento do balanceamento da árvore após a inserção ou a remoção de um elemento que tenha provocado um desequilíbrio na distribuição dos elementos, de modo que a condição de balanceamento não seja mais atendida.

Para compreender melhor esse conceito, tomemos o exemplo da árvore da figura abaixo (a). Note que a inserção da chave 35 na árvore leva a seu desbalanceamento, uma vez que a diferença de altura entre as subárvore do nó raiz é superior a 1. Para restabelecer o balanceamento, poderia ser feita uma rotação dos nós, conforme ilustrado na figura abaixo (b).

# Árvores Balanceadas

- De modo geral, podemos resumir as rotações de nós em árvores AVL em:

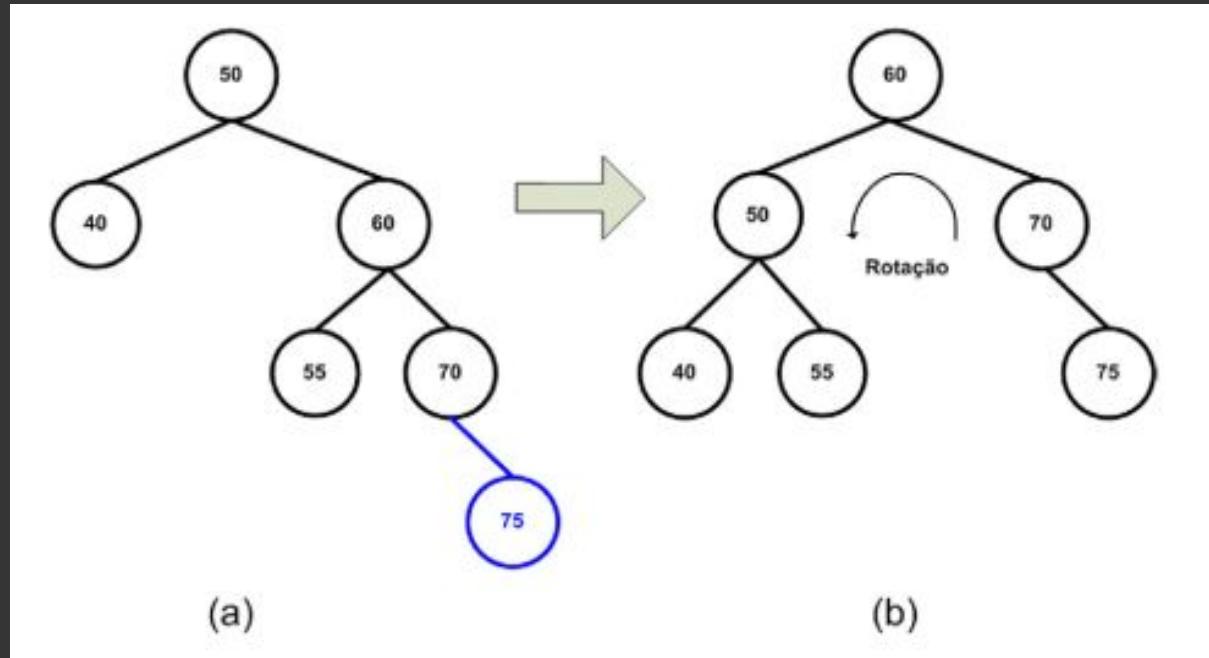
- **rotação à direita:** Neste tipo de rotação, o nó raiz da subárvore é deslocado para a posição de seu nó filho à direita, que, por sua vez, continua a ser o nó filho à direita do nó deslocado. O nó filho à direita do nó filho à esquerda da raiz é deslocado para ser o nó filho à esquerda do nó raiz deslocado. O nó filho à esquerda do nó raiz ocupa seu antigo lugar, passando a ser a nova raiz.



# Árvores Balanceadas

Para ilustrar a rotação à esquerda, vejamos a figura ao lado.

Aplicando na árvore (a) as ações definidas para essa rotação, temos a árvore (b), que atende tanto a manutenção da ordenação entre as chaves quanto a condição de balanceamento por altura.

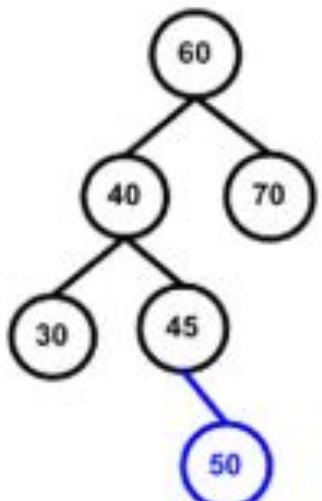


# Árvores Balanceadas

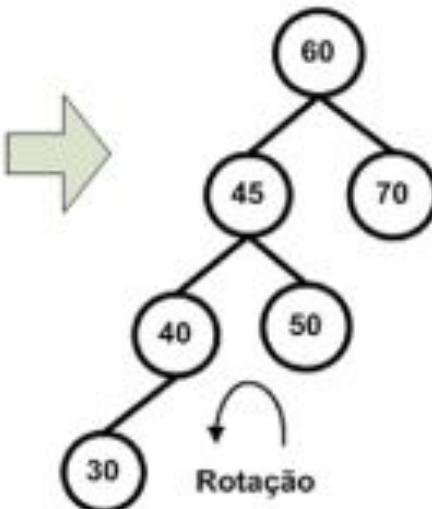
Existem situações nas quais necessitamos de duas rotações seguidas (uma à direita e outra à esquerda, ou uma à esquerda e outra à direita) para que seja restabelecido o balanceamento da árvore. Essas rotações em sequência são conhecidas como “rotações duplas”.

Na Figura a seguir temos uma situação em que é necessária a aplicação de uma “rotação dupla” para restabelecer o balanceamento da árvore. Ao tentar inserir a chave 50 na árvore do exemplo (a), teremos que a diferença de altura entre as subárvore direita e esquerda da raiz será 2 (árvore desbalanceada). Nesse caso, uma simples rotação não seria suficiente para restabelecer o balanceamento. Assim, é necessário primeiro aplicar uma rotação à esquerda, a partir do nó com chave 40, obtendo a árvore do exemplo (b) e, na sequência, uma rotação à direita a partir da raiz, obtendo a árvore do exemplo (c), que atenderá novamente às propriedades de uma árvore AVL.

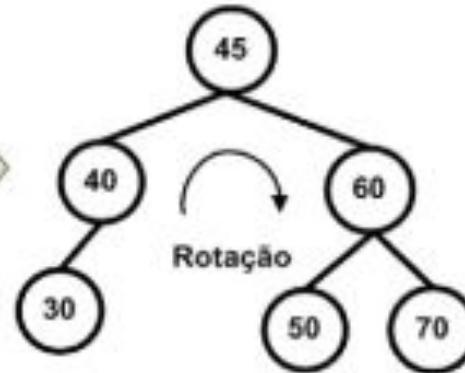
# Árvores Balanceadas



(a)



(b)



(c)

# Árvores Balanceadas

De forma resumida, podemos dizer que as operações de inserção e de remoção de nós numa árvore AVL são as mesmas usadas em árvores binárias; a diferença é que, após a realização dessas operações, torna-se necessário verificar a manutenção do balanceamento por altura. No caso de se identificar um desbalanceamento na árvore, ou seja, se a diferença for maior que 1 (ou menor que -1) entre as alturas das subárvores direita e esquerda, torna-se necessária a aplicação de rotações à direita, à esquerda ou “rotações duplas”, dependendo do caso, para que se restabeleça o balanceamento da árvore. Lembramos que as rotações devem preservar a ordenação das chaves armazenadas na árvore.

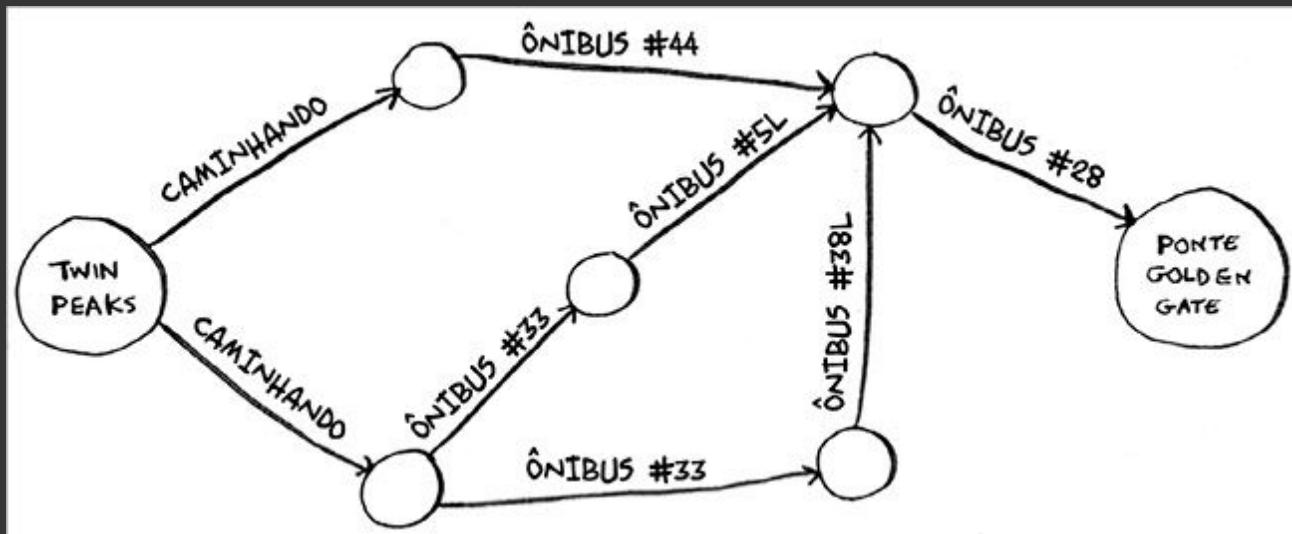
# 11

## Grafos

Relações entre os objetos de um determinado conjunto

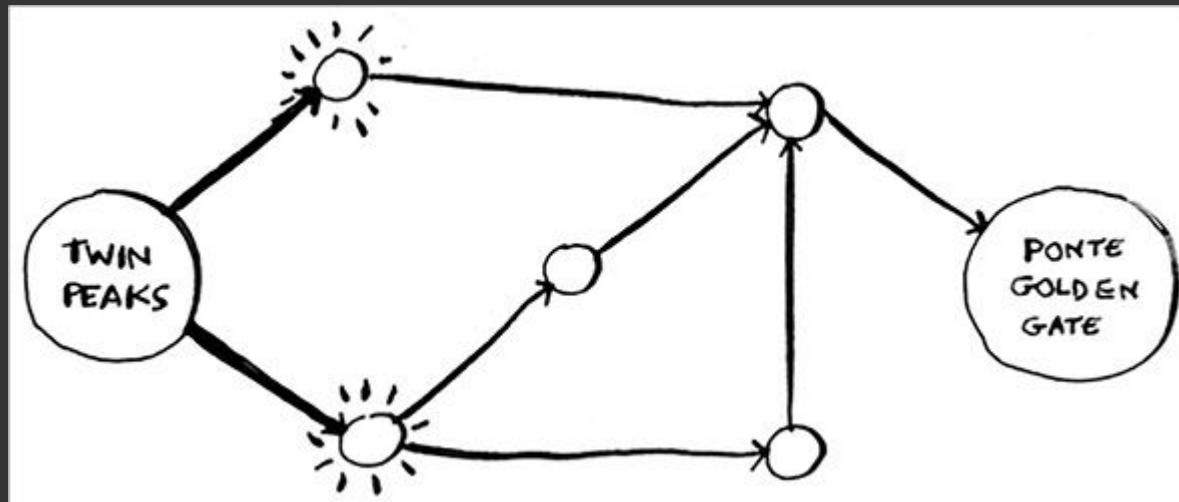
# Grafos

Suponha que você esteja em San Francisco e queira ir das Twin Peaks (duas montanhas localizadas no centro da cidade) até a ponte Golden Gate. Você pretende chegar lá de ônibus, porém quer fazer transferência de um ônibus para outro o menor número de vezes possível. Suas opções são:



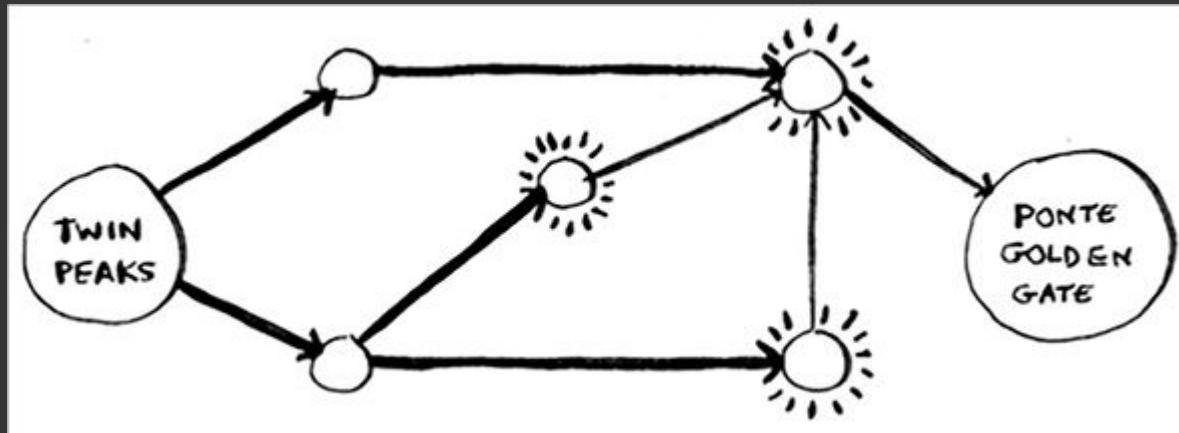
# Grafos

Qual algoritmo você propõe para encontrar o caminho com o menor número de etapas? Bem, você consegue chegar ao seu destino com uma etapa? Aqui estão todos os lugares para os quais é possível chegar com uma etapa:



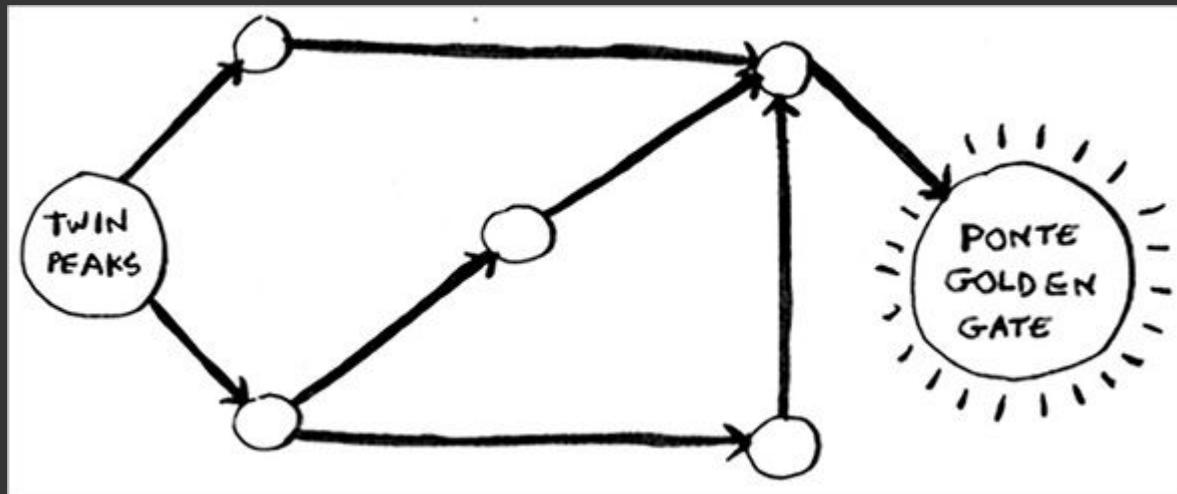
# Grafos

A ponte não está destacada, logo não é possível chegar lá com uma etapa. E com duas etapas?



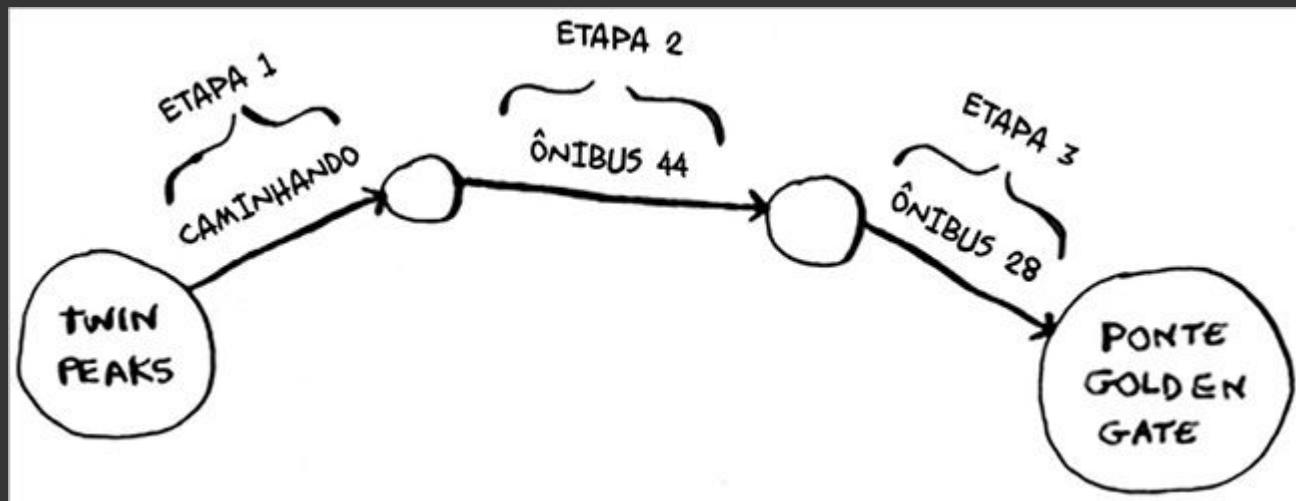
# Grafos

Mais uma vez, a ponte não está destacada, logo você não pode chegar lá com duas etapas. E com três etapas?



# Grafos

Ahá! Agora a ponte Golden Gate está destacada. Então, são necessárias três etapas para ir da Twin Peaks até a ponte por meio dessa rota.



# Grafos

Existem outras rotas que levam você até a ponte, mas elas são mais longas (quatro etapas). O algoritmo descobriu que o caminho mais curto até a ponte demanda três etapas. Esse tipo de problema é chamado de **problema do caminho mínimo**. Neste problema, você sempre tentará achar o caminho mínimo para algo, como por exemplo a rota mais curta até a casa de seu amigo, ou também o número mínimo de movimentos para dar xeque-mate em um jogo de xadrez. O algoritmo que resolve problemas de caminho mínimo é a **pesquisa em largura**.

Para descobrir como ir da Twin Peaks até a ponte Golden Bridge existem duas etapas:

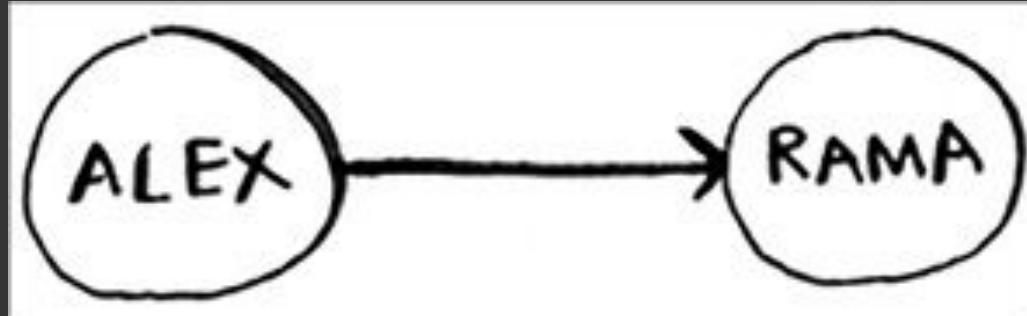
- Modele o problema utilizando grafos.
- Resolva o problema utilizando a pesquisa em largura.

Agora, falarei sobre o que são grafos. Depois, abordarei a pesquisa em largura em mais detalhes.

# Grafos

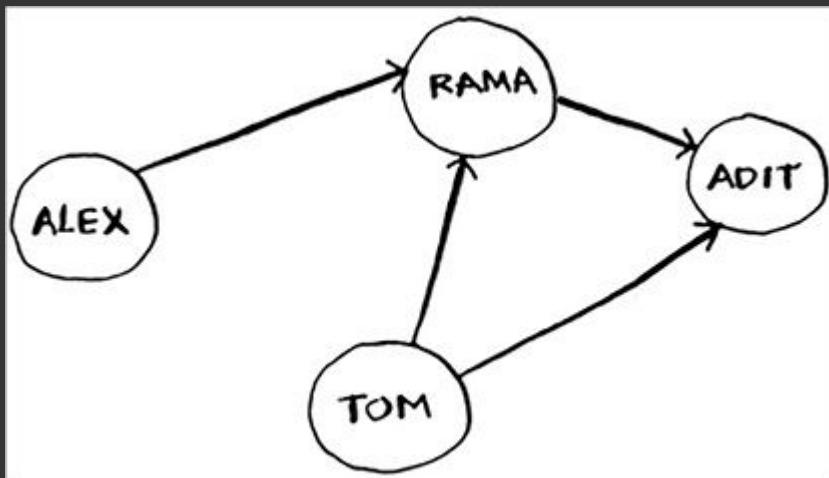
## □ O que é um grafo ?

Um modelo de grafo é um conjunto de conexões. Por exemplo, suponha que você e seus amigos estejam jogando pôquer e que você queira descrever quem deve dinheiro a quem. Você poderia dizer “Alex deve dinheiro à Rama”.

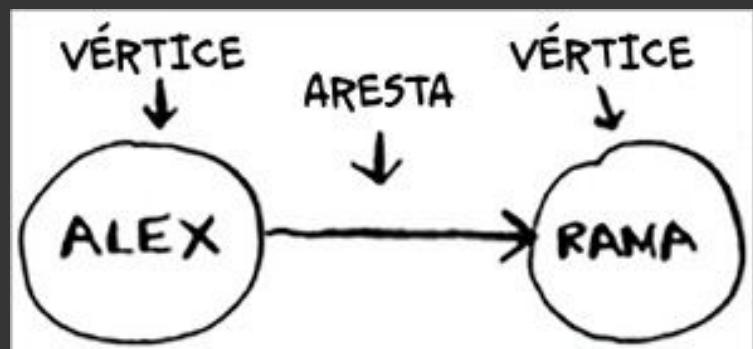


# Grafos

O grafo completo poderia ser algo do tipo:



Alex deve dinheiro à Rama, Tom deve dinheiro à Adit, e assim por diante. Cada grafo é constituído de vértices e arestas.



# Grafos

E isso é tudo! Grafos são formados por vértices e arestas, e um vértice pode ser diretamente conectado a muitos outros vértices, por isso os chamamos de vizinhos. Neste grafo, Rama é vizinha de Alex. Já Adit não é vizinho de Alex, pois eles não estão diretamente conectados, mas Adit é vizinho de Rama e de Tom.

Os grafos são uma maneira de modelar como eventos diferentes estão conectados entre si. Agora vamos ver a pesquisa em largura na prática.

# Grafos

## □ Pesquisa em largura

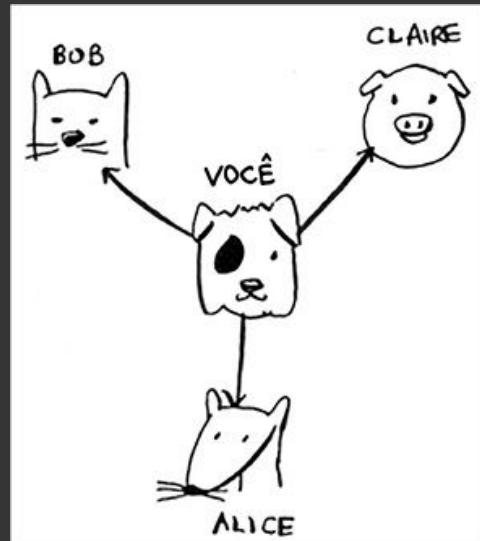
Nós conhecemos um algoritmo de pesquisa nas seções anteriores: a pesquisa binária. A pesquisa em largura é um tipo diferente de algoritmo, pois utiliza grafos. Este algoritmo ajuda a responder a dois tipos de pergunta:

- **1:** Existe algum caminho do vértice A até o vértice B?
- **2:** Qual o caminho mínimo do vértice A até o vértice B?

Você já viu a pesquisa em largura em ação uma vez quando calculou a rota mais curta do Twin Peaks até a ponte Golden Gate. Essa pergunta foi do tipo 2: “Qual é o caminho mínimo?”. Agora vamos analisar o algoritmo em mais detalhes, e você fará uma pergunta do tipo 1: “Existe um caminho?”.

# Grafos

Vamos supor que você seja o dono de uma fazenda de mangas e esteja procurando um vendedor de mangas que possa vender a sua colheita. Você conhece algum vendedor de mangas no Facebook? Bem, você pode procurar entre seus amigos.

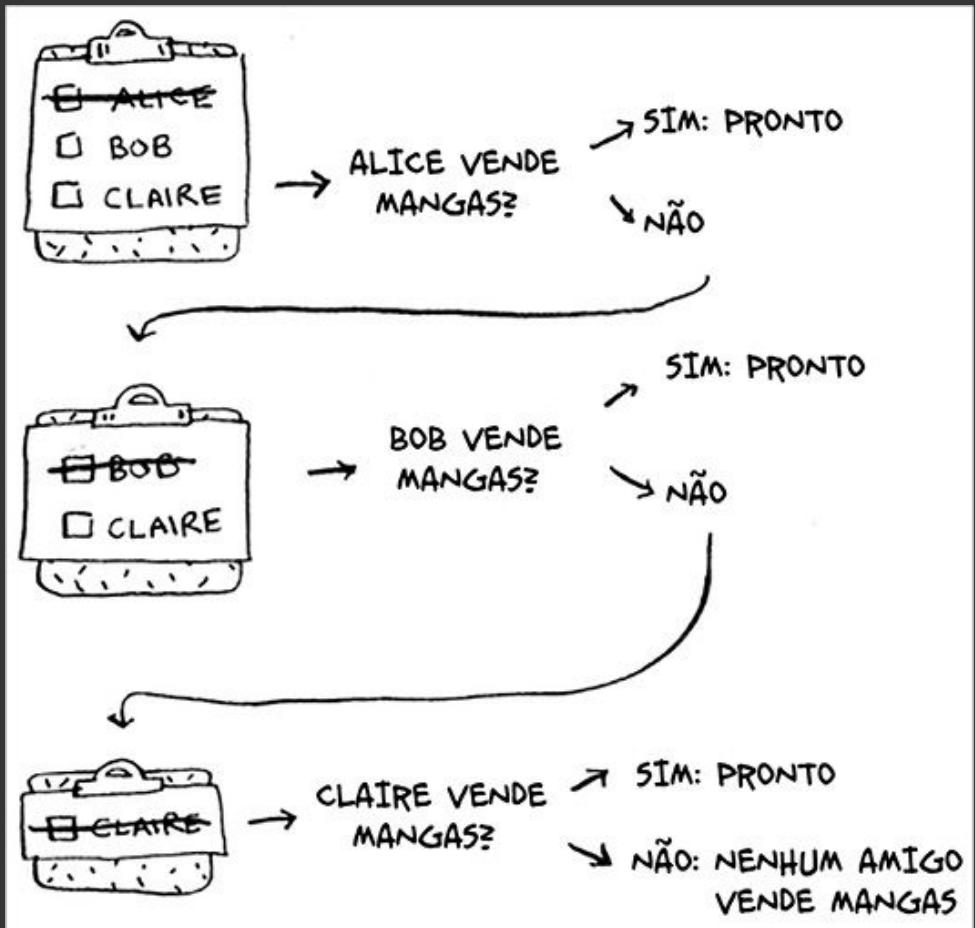


# Grafos

Essa pesquisa é bem direta. Primeiro, faça uma lista de amigos para pesquisar.

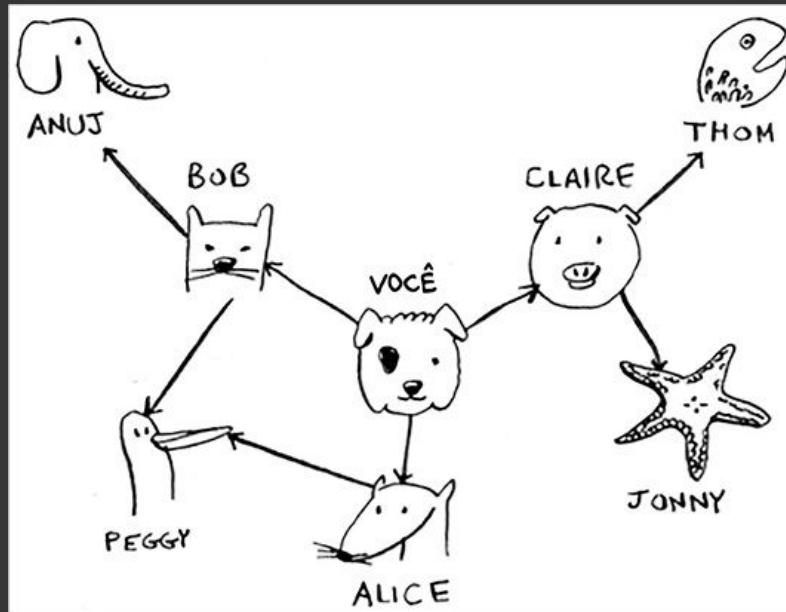


Agora vá até cada pessoa da lista e verifique se esta pessoa vende mangas.



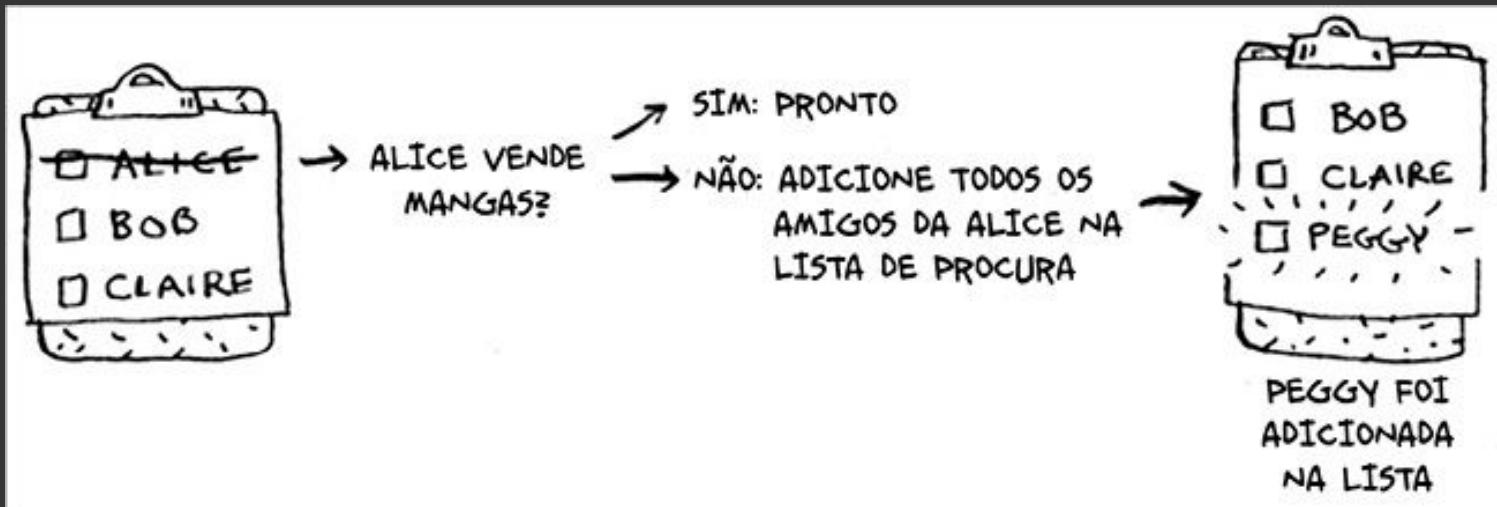
# Grafos

Imagine que nenhum de seus amigos é um vendedor de mangas. Então, será necessário pesquisar entre os amigos dos seus amigos.



# Grafos

Cada vez que você pesquisar uma pessoa da lista, todos os amigos dela serão adicionados à lista.



# Grafos

Dessa maneira você não pesquisa apenas entre os seus amigos, mas também entre os amigos deles. Lembre-se de que o objetivo é encontrar um vendedor de mangas em sua rede. Então, se Alice não é uma vendedora de mangas, você adicionará também os amigos dela à lista. Isso significa que, eventualmente, pesquisará entre os amigos dela e entre os amigos dos amigos, e assim por diante. Com esse algoritmo você pesquisará toda a sua rede até que encontre um vendedor de mangas. Isto é o algoritmo da pesquisa em largura em ação.

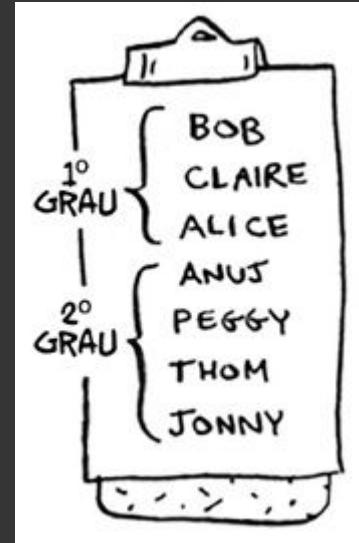
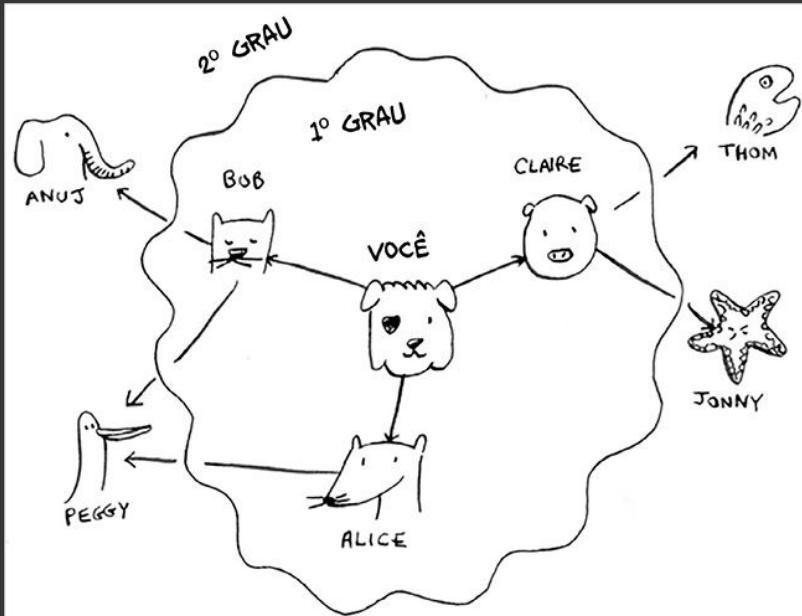
## □ Encontrando o caminho mínimo

Relembrando, existem dois tipos de pergunta que a pesquisa em largura responde:

- **1:** Existe um caminho do vértice A até o vértice B? (Existe um vendedor de manga na minha rede?)
- **2:** Qual o caminho mínimo do vértice A até o vértice B? (Quem é o vendedor de manga mais próximo?)

# Grafos

Você já sabe a resposta para a pergunta 1. Agora, vamos tentar responder a pergunta 2. Você consegue encontrar o vendedor de mangas mais próximo? Por exemplo, seus amigos são conexões de primeiro grau e os amigos deles são conexões de segundo grau.



# Grafos

Você preferiria uma conexão de primeiro grau em vez de uma conexão de segundo grau, e uma conexão de segundo grau a uma de terceiro grau, e assim por diante.

Portanto, não se deve pesquisar nenhuma conexão de segundo grau antes de você ter certeza de que não existe uma conexão de primeiro grau com um vendedor de mangas. Bem, a pesquisa em largura já faz isso! O funcionamento da pesquisa em largura faz com que a pesquisa a partir do ponto inicial. Dessa forma, você verificará as conexões de primeiro grau antes das conexões de segundo grau. Pergunta rápida: Quem será verificado primeiro, Claire ou Anuj? Resposta: Claire é uma conexão de primeiro grau e Anuj é uma conexão de segundo grau, logo Claire será verificada antes de Anuj.

Outra maneira de ver isso é sabendo que conexões de primeiro grau são adicionadas à pesquisa antes de conexões de segundo grau.

# Grafos

Você apenas segue a lista e verifica se a pessoa é uma vendedora de mangas. As conexões de primeiro grau serão procuradas antes das de segundo grau, e, dessa forma, você encontrará o vendedor de mangas mais próximo.

Assim, a pesquisa em largura não encontra apenas um caminho entre A e B, ela encontra o caminho mais curto.

Repare que isso só funciona se você procurar as pessoas na mesma ordem em que elas foram adicionadas. Ou seja, se Claire foi adicionada à lista antes de Anuj, deve-se pesquisar Claire antes de Anuj. O que acontece se você pesquisar Anuj antes de Claire, sendo que ambos são vendedores de mangas? Bem, Anuj é um contato de segundo grau enquanto Claire é um contato de primeiro grau, o que fará com que o vendedor de mangas encontrado não seja o mais próximo. Portanto é necessário pesquisar as pessoas na ordem em que elas foram adicionadas, para isso existe uma estrutura de dados específica que já vimos que é a fila.

# Grafos

## □ Tempo de Execução

Se você procurar um vendedor de mangas em toda a sua rede, cada aresta (lembre-se de que aresta é a seta ou a conexão entre uma pessoa e outra) será analisada. Portanto o tempo de execução é, no mínimo,  $O(\text{número de arestas})$ .

Além disso, também será mantida uma lista com as pessoas já verificadas. Adicionar uma pessoa à lista leva um tempo constante:  $O(1)$ . Fazer isso para cada pessoa terá tempo de execução  $O(\text{número de pessoas})$  no total. Assim, a pesquisa em largura tem tempo de execução  $O(\text{número de pessoas} + \text{número de arestas})$ , que é frequentemente descrito como  $O(V+A)$  ( $V$  para número de vértices,  $A$  para número de arestas).

# Grafos

## □ Concluindo

- A pesquisa em largura lhe diz se há um caminho de A para B.
- Se esse caminho existir, a pesquisa em largura lhe dará o caminho mínimo.
- Se você tem um problema do tipo “encontre o menor X”, tente modelar o seu problema utilizando grafos e use a pesquisa em largura para resolvê-lo.
- Grafos não direcionados não contêm setas, e a relação acontece nos dois sentidos.

# 12

## Classes de Problemas

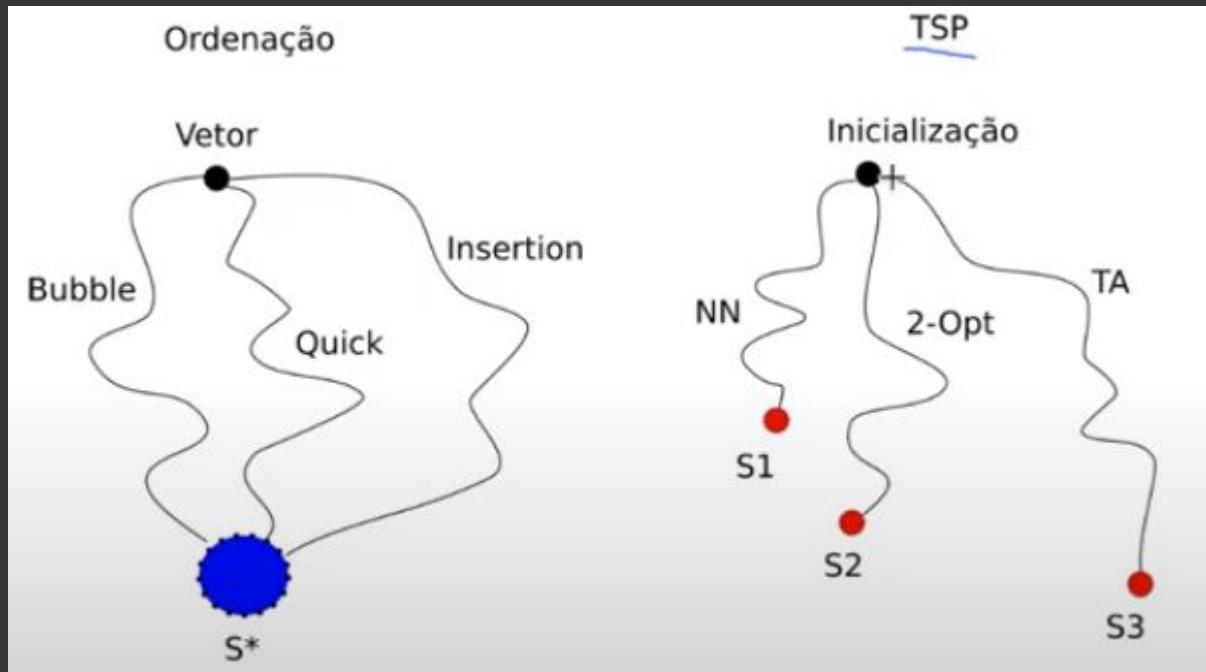
A pergunta de 1 milhão de dólares P é igual NP ??

# Classes de Problemas

Na teoria da complexidade computacional, existem vários tipos de problemas como **problemas de ordenação** que podem ser resolvidos através de vários algoritmos como o Bubble Sort, Quick Sort e Insertion Sort, e o que ocorre é que em alguns problemas, todos os algoritmos chegam na mesma solução considerada ótima, independente da entrada. Em problemas mais complexos isso não ocorre.

Observe na imagem a seguir que para o problema de ordenação independente do algoritmo que for utilizado o resultado sempre será uma solução ótima, mesmo que para cada algoritmo o número de passos executados sejam diferentes, porém isso não acontece no segundo caso que é o **problema do caixeiro viajante (traveling salesman problem)** que vimos nas seções anteriores que para cada algoritmo utilizado, o resultado será diferente, ou seja, não existe garantia que exista um algoritmo resolve o problema em uma solução considerada ótima para todas as entradas.

# Classes de Problemas



# Classes de Problemas

## □ Tipos de Problemas

- **Problemas de otimização:** Começamos com problemas de otimização, ou seja, problemas que pedem o mínimo ou o máximo de alguma coisa.
- **Problemas de busca:** Intuitivamente, o problema consiste em encontrar uma estrutura Y em um objeto X. Um algoritmo é dito para resolver o problema se ele se comporta da seguinte forma: se pelo menos uma estrutura correspondente existe, então uma ocorrência desta estrutura é emitida, caso contrário, o algoritmo pára com uma saída apropriada ("Item não encontrado" ou qualquer outra mensagem do gênero).
- **Problemas de decisão:** é um problema computacional onde a resposta para cada instância é sim ou não. Um exemplo de um problema de decisão é o teste de primalidade: "Dado um inteiro positivo n, determine se n é primo."

# Classes de Problemas

Como os problemas de decisão não são mais difíceis que os de busca, e estes não são mais difíceis que os de otimização, é suficiente tratar dos problemas de decisão. Se provarmos que um dado problema de decisão não tem um algoritmo razoavelmente rápido, ficará demonstrado também que os correspondentes problemas de busca e de otimização não têm algoritmos razoavelmente rápidos.

Para alguns problemas, obter a solução ótima é um desafio e isso incentivou os cientistas da computação a separar os problemas computacionais em algumas classes e agora vamos ver as principais classes de problemas computacionais.

# Classes de Problemas

## □ Classe “P”

**Classe P** é o acrônimo em inglês para **Tempo polinomial determinístico** (**Deterministic Polynomial time**) que denota o conjunto de problemas que podem ser resolvidos em **tempo polinomial determinístico** para todas as entradas.

**Problema determinístico** é aquele problema **fácil de prever** que se consegue uma solução exata a cada passo, qualquer problema deste conjunto pode ser resolvido por um algoritmo com **tempo de execução  $O(n^k)$ , onde k é uma constante**).

# Classes de Problemas

## □ Alguns Exemplos de Problemas da Classe “P”

- **Divisor-Comum-Grande:** Dados números naturais  $m$ ,  $n$  e  $k$ , decidir se existe um divisor comum de  $m$  e  $n$  que seja maior ou igual a  $k$ .
- **Caminho-Curto:** Dados vértices  $r$  e  $s$  de um grafo e um número  $k$ , decidir se existe um caminho de  $r$  a  $s$  que tenha comprimento menor ou igual a  $k$ .
- **Intervalos-Disjuntos:** Dada uma coleção de intervalos na reta e um número natural  $k$ , decidir se a coleção tem uma subcoleção disjunta com  $k$  ou mais intervalos.

Resumindo, podemos dizer que a classe P é uma classe de problemas que existe um algoritmo polinomial para resolvê-lo. São problemas **tratáveis** do ponto de vista computacional.

# Classes de Problemas

## □ Classe “NP”

A **classe NP** é o conjunto dos problemas que são **polinomialmente verificáveis**, ou seja, dada uma cadeia de entrada é possível verificar se ela pertence à linguagem ou não em tempo polinomial. Por exemplo ao tentar adivinhar uma senha aleatória pelo algoritmo de força bruta, testando todas as combinações de caracteres levaríamos um **tempo exponencial** em uma Máquina de Turing determinística, porém para testar se uma senha é a correta ou não, levaria tempo polinomial, ou seja, esta classe de problema pode ser **provado** em **tempo polinomial**.

Cuidado! NP não é abreviatura de não polinomial mas sim de **nondeterministic polynomial time**. um **algoritmo não determinístico** é um algoritmo **difícil de prever** e dada uma certa entrada, pode apresentar comportamentos diferentes em diferentes execuções, isso faz com que em alguns casos pode ser executado em tempo polinomial ou tempo exponencial, tudo vai depender das escolhas que o algoritmo faz durante a execução.

# Classes de Problemas

## □ Dificuldades Envolvidas em “NP”

Devido ao fato de existirem muitos problemas importantes nesta classe, existe um esforço intensivo para encontrar algoritmos para resolver os problemas NP em um tempo que seja polinomial em relação ao tamanho da entrada. Contudo, existe um grande número de problemas NP que resiste a tais tentativas, parecendo requerer um tempo super-polinomial. Se estes problemas realmente não podem ser resolvidos em tempo polinomial é uma das grandes questões abertas na ciência da computação

Uma importante noção neste contexto é o conjunto de problemas de decisão NP-Completo, o qual é um subconjunto de NP e pode ser informalmente descrito como os mais difíceis problemas em NP. Se existe um algoritmo em tempo polinomial para um deles, então haverá um algoritmo em tempo polinomial para todos os problemas em NP. Devido a isto, e a falha das pesquisas em encontrar um algoritmo polinomial para qualquer problema NP-completo, uma vez que foi provado que um problema pode ser caracterizado como NP-completo este é um sinal largamente aceito que um algoritmo polinomial para este problema não deve existir.

# Classes de Problemas

## □ Alguns Exemplos de Problemas da Classe “NP”

- **Isomorfismo de grafos:** determinar se dois grafos podem ser desenhados de forma idêntica.
- **Caixeiro viajante:** queremos saber se existe uma rota de qualquer comprimento que passe através de todos os nós de uma certa rede.
- **Roteamento de Veículos:** queremos alocar uma frota veicular para o atendimento de um conjunto de consumidores. É semelhante ao problema do caixeiro viajante, mas possui mais de um veículo para entregas e coletas de mercadorias.

# Classes de Problemas

## □ Classe “NP-Completo”

A classe de complexidade NP-Completo é o subconjunto dos problemas NP de tal modo que todo problema em NP se pode reduzir, com uma **redução de tempo polinomial**, a um dos problemas NP-completo. Pode-se dizer que os problemas de NP-completo são os problemas mais difíceis de NP e muito provavelmente não estão na classe de complexidade P.

A razão é que se conseguisse encontrar uma maneira de resolver qualquer problema NP-completo rapidamente (em tempo polinomial), então poderiam ser utilizados algoritmos para resolver todos problemas NP rapidamente. Como exemplo de um problema NP-completo está o problema da soma dos subconjuntos que pode ser enunciado conforme segue: dado um conjunto S de inteiros, determine se há algum conjunto não vazio de S cujos elementos somem zero.

# Classes de Problemas

É fácil de verificar se uma resposta é correta, mas não se conhece uma solução significativamente mais rápida para resolver este problema do que testar todos os subconjuntos possíveis, até encontrar um que cumpra com a condição. Na prática, o conhecimento de NP-completo pode evitar que se desperdice tempo tentando encontrar um algoritmo de tempo polinomial para resolver um problema quando esse algoritmo não existe.

Um problema **P** em **NP** também está em **NPC, Se e somente se** todos os outros problemas em NP podem ser transformados em P em **tempo polinomial**.

Problemas NP-completo são estudados porque a habilidade de rapidamente verificar soluções para um problema (NP) parece correlacionar-se com a capacidade de resolver rapidamente esse problema (P). Não é sabido se todos os problemas em NP podem ser rapidamente resolvidos, isso é chamado de problema **P versus NP**.

# Classes de Problemas

Mas se qualquer problema em NP-completo pode ser resolvido rapidamente, então todo problema em NP também pode ser, por causa da definição de NP-completo afirma que todo problema em NP deve ser rapidamente redutível para todo problema em NP-completo (ou seja, pode ser reduzido em tempo polinomial). Por causa disso, é geralmente falado que os problemas NP-completo são mais difíceis que os problemas NP em geral.

Ninguém ainda conseguiu determinar conclusivamente se os problemas NP-completo são de fato resolvíveis em tempo polinomial, fazendo deste um dos maiores problemas não solucionados da matemática. O **Clay Mathematics Institute** oferece **1 milhão de dólares** de recompensa para quem fizer uma prova formal que **P = NP** ou que **P ≠ NP**.

# Classes de Problemas

## □ Redução

O que significa um problema ser "reduzido" a outro? Simplesmente que existe um processo capaz de transformar um problema em outro, solucionar o outro, e transformar a solução encontrada numa solução para o problema original. Exemplo:

- **Qual é o mínimo múltiplo comum entre 6 e 15?**

Em vez de resolver esse problema diretamente, pode-se transformar esse problema em um problema de máximo divisor comum:

- **$\text{mmc}(x, y) = \text{abs}(x * y) / \text{mdc}(x, y)$**

# Classes de Problemas

Resolver o problema  $\text{mdc}(6, 15)$  (muito mais "fácil" dado o algoritmo de Euclides ou, se a questão for eficiência, o MDC Binário) e, de posse da solução do mesmo (3), obter a solução final:

- **$\text{mmc}(6, 15) = \text{abs}(6 * 15) / \text{mdc}(6, 15) = 90 / 3 = 30$**

Às vezes a importância da redução é apenas teórica, em outras pode-se ter benefícios concretos em fazê-lo aproveitar uma implementação já pronta, por exemplo. Desde é claro que o processo de se converter o problema a solução não seja mais custoso que simplesmente solucionar o problema original.

# Classes de Problemas

## □ E como mostrar que um problema é NP-completo ?

- 1- Primeiro mostramos que o problema **está em NP**.
- 2- Depois encontramos um problema que **já é NP-Completo** e o **reduzimos polinomialmente** para o nosso, e com isso automaticamente garantimos que o nosso problema é NP-Completo.

## □ Exemplo:

- Supomos que o nosso problema é o da **MOCHILA**
- Garantimos que o problema da **MOCHILA está em NP**.
- Conseguimos desenvolver um algoritmo que reduz o problema **SUBSET-SUM** para o problema da **MOCHILA** em **tempo polinomial**.
- Sabemos que **SUBSET-SUM** é **NP-Completo**.
- Portanto, o problema da **MOCHILA** é **NP-Completo**.

# Classes de Problemas

## □ Alguns Exemplos de Problemas da Classe “NP-Completo”

- **Problema de satisfazibilidade booleana (SAT):** Foi o primeiro problema identificado como pertencente à classe de complexidade NP-completo. O problema de satisfatibilidade booleana é o problema de determinar se existe uma determinada valoração para as variáveis de uma determinada fórmula booleana tal que esta valoração satisfaça esta fórmula em questão.
- **Problema da mochila:** é um problema de otimização combinatória. O nome dá-se devido ao modelo de uma situação em que é necessário preencher uma mochila com objetos de diferentes pesos e valores. O objetivo é que se preencha a mochila com o maior valor possível, não ultrapassando o peso máximo.

# Classes de Problemas

- **Problema do caixeiro viajante:** é um problema que tenta determinar a menor rota para percorrer uma série de cidades (visitando uma única vez cada uma delas), retornando à cidade de origem. Ele é um problema de otimização NP-difícil inspirado na necessidade dos vendedores em realizar entregas em diversos locais (as cidades) percorrendo o menor caminho possível, reduzindo o tempo necessário para a viagem e os possíveis custos com transporte e combustível.
- **Problema do clique:** refere-se a qualquer problema que possui como objetivo encontrar subgrafos completos ("cliques") em um grafo. Como exemplo, o problema de encontrar conjuntos de nós em que todos os elementos estão conectados entre si.

# Classes de Problemas

Em geral, pode-se dizer que um problema é NP-Completo se ele se **reduz** a qualquer outro problema NP-Completo, não necessariamente o SAT (o problema do clique é muito popular nesse sentido).

Na prática, a grande discussão é que os problemas NP-completo são a chave para se determinar se **P = NP** ou **P ≠ NP**. Se em algum momento um problema NP não puder ser resolvido em tempo polinomial, nenhum problema NP-Completo pode ser resolvido em tempo polinomial. Por outro lado, se algum problema NP-Completo puder ser resolvido em tempo polinomial, podemos afirmar que **P = NP**.

O problema se  $P = NP$  é considerado por muitos **o maior problema não resolvido na ciência da computação**, a maioria dos cientistas da computação acredita que **P ≠ NP**, e a razão para isso é que depois de décadas de estudo sobre estes problemas não foi possível achar um algoritmo polinomial que decida qualquer um dos mais de 3000 problemas NP-Completos conhecidos.

# Classes de Problemas

## □ Classe “NP-Difícil”

NP-difícil na teoria da complexidade computacional, é uma classe de problemas que são, informalmente, "Pelo menos tão difíceis quanto os problemas mais difíceis em NP" e não necessariamente está em NP, ou seja, existem problemas nesta classe que **nem se quer existe** um algoritmo que verifica se uma solução é válida.

## □ Como consequências da definição, temos que (note que estas afirmações não são definições):

- O problema H é pelo menos tão difícil quanto L, porque H pode ser usado para resolver L;

# Classes de Problemas

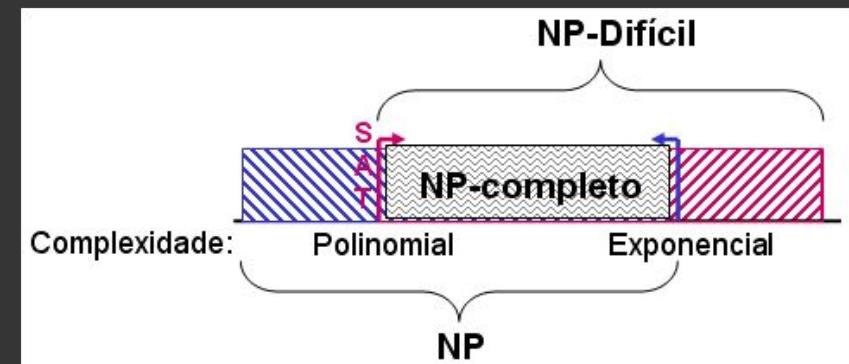
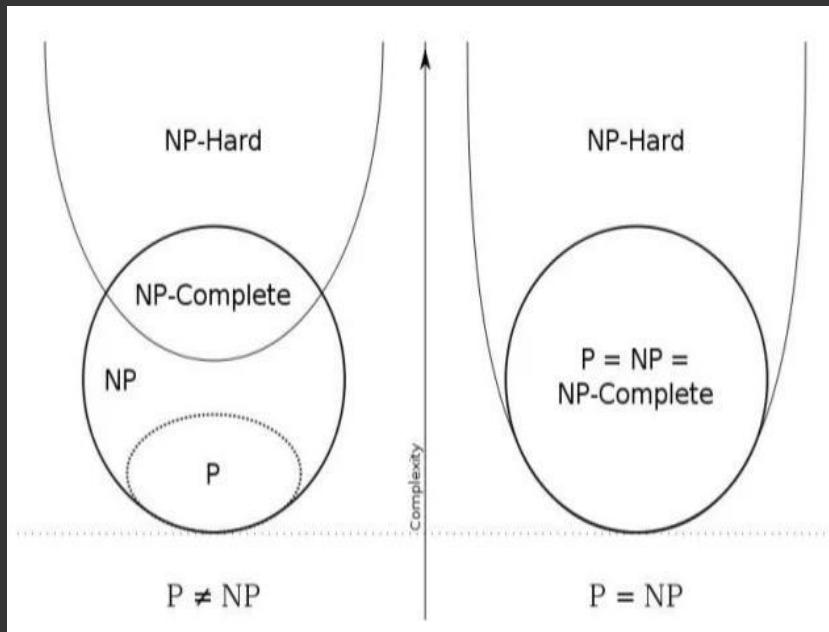
- Como L é NP-completo e, portanto, o mais difícil na classe NP, também o problema H é pelo menos tão difícil quanto um NP, mas H não tem que estar em NP e, consequentemente, não tem de ser um problema de decisão (mesmo que seja um problema de decisão, ele não precisa estar em NP);
- Uma vez que os problemas NP-completos transformam-se uns aos outros por redução um para muitos em tempo polinomial todos os problemas NP-Completos podem ser resolvidos em tempo polinomial por uma redução para H, Assim, todos os problemas em NP reduzem para H, note-se, porém, que isso envolve a combinação de duas transformações diferentes: de problemas de decisão NP-Completos para o problema NP-completo L por transformação polinomial, e de L para H pela redução em tempo polinomial.

# Classes de Problemas

- Se existe um algoritmo polinomial para qualquer problema NP-difícil, então existem algoritmos polinomiais para todos os problemas em NP, e, portanto,  $P = NP$ .
- Se um problema de otimização H tem uma versão de decisão NP-completo L, então H é NP-difícil.

Um erro comum é pensar que NP em NP-difícil representa não-polinomial. Embora seja amplamente suspeito de que não existem algoritmos de tempo polinomial para problemas NP-difíceis, isto nunca foi provado. Além disso, a classe NP contém também todos os problemas que podem ser resolvidos em tempo polinomial.

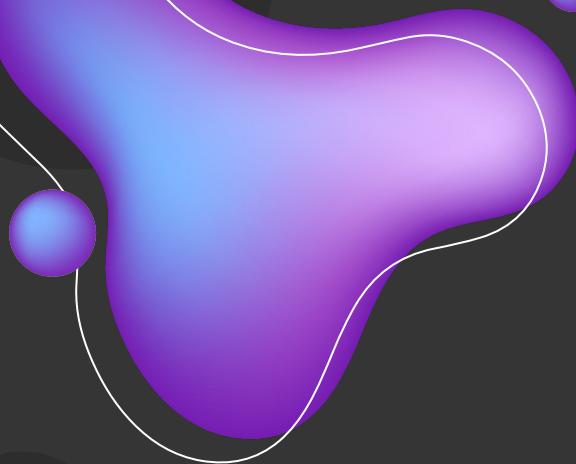
# Classes de Problemas



# Classes de Problemas

## □ Concluindo

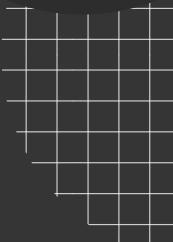
- A **Classe P** é o conjunto de problemas que podem ser **resolvidos em tempo polinomial** para todas as entradas.
- A **Classe NP** é o conjunto dos problemas que são **polinomialmente verificáveis**, mas difíceis de serem solucionáveis, ou seja, dada uma solução é possível verificar se ela é válida em tempo polinomial.
- A **Classe NP-Completo** é o subconjunto dos problemas mais difíceis em NP, de tal modo que todo problema em NP pode reduzir, com uma **redução de tempo polinomial**, a um dos problemas NP-completo.
- A **Classe NP-Difícil** é uma classe de problemas, pelo menos tão difíceis quanto os problemas mais difíceis em NP e não necessariamente está em NP, ou seja, existem problemas nesta classe que **nem se quer existe** um algoritmo que verifica se uma solução é válida.



13

# Programação Dinâmica

Resolvendo problemas complexos de forma mais eficiente



# Programação Dinâmica

A **Programação Dinâmica** é uma técnica para construção de algoritmos para resolução de problemas computacionais, em especial os de **otimização combinatória**. Ela é aplicável a problemas nos quais a solução ótima pode ser computada a partir da solução ótima previamente calculada e memorizada de forma a evitar recálculo de outros subproblemas que, sobrepostos, compõem o problema original.

Nesta sessão iremos aplicar esta técnica em um problema muito famoso conhecido como **problema da mochila**.

O **problema da mochila** é um problema de otimização combinatória. O nome dá-se devido ao modelo de uma situação em que é necessário preencher uma mochila com objetos de diferentes pesos e valores. O objetivo é que se preencha a mochila com o maior valor possível, não ultrapassando o peso máximo.

# Programação Dinâmica

Você é um ladrão com uma mochila que consegue carregar apenas 16 quilos. Você tem três itens disponíveis para colocar dentro da sua mochila.

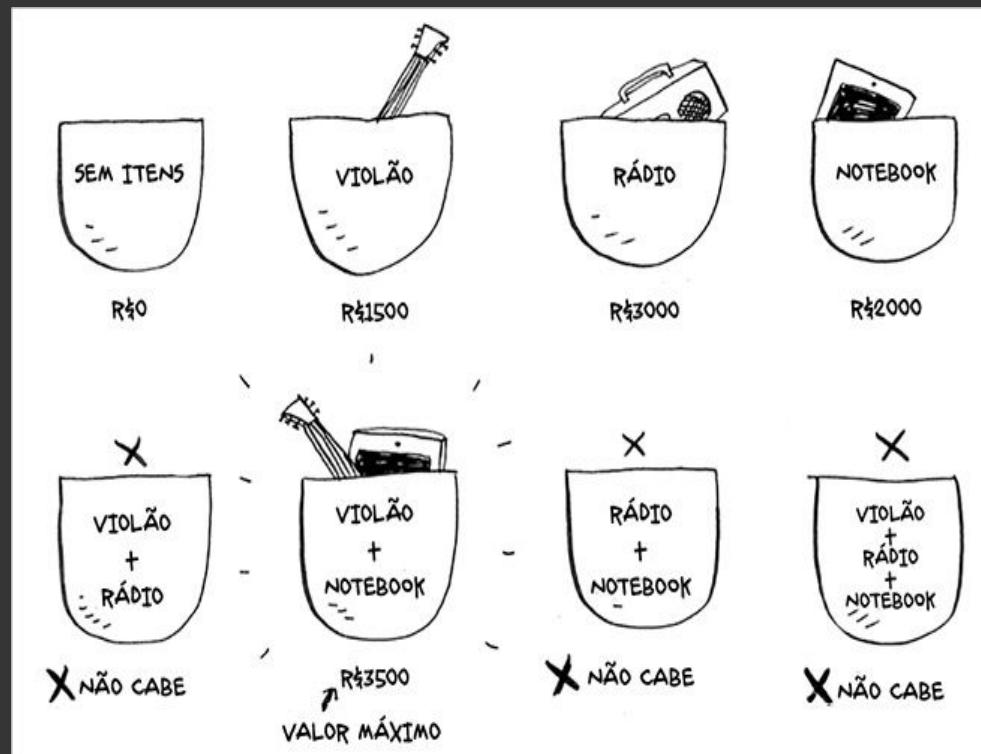
Quais itens você deveria roubar para maximizar o valor roubado?



# Programação Dinâmica

## □ A solução simples

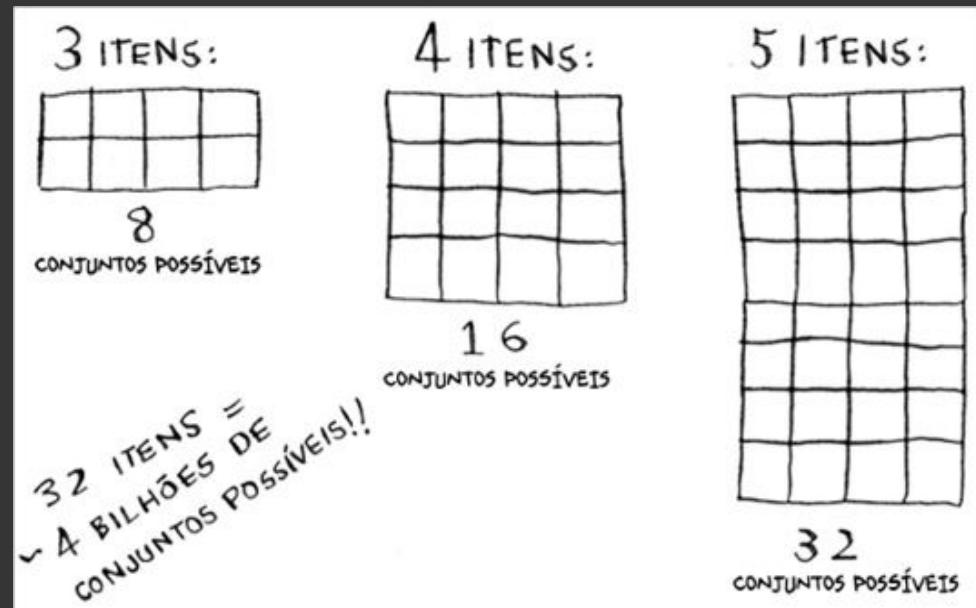
O algoritmo mais simples é o seguinte: você deve testar para todos os conjuntos de itens possíveis e descobrir qual conjunto maximizará o valor roubado.



# Programação Dinâmica

Isto funciona, mas é uma solução muito lenta, pois para três itens você deverá calcular oito conjuntos possíveis. Para quatro itens, são 16 conjuntos. Cada item adicionado dobrará o número de cálculos. Este algoritmo tem tempo de execução  $O(2^n)$ , é muito, muito lento.

Esta solução não é prática para qualquer número razoável de itens. Esta solução é próxima o suficiente da solução ideal, mas talvez não seja a própria.

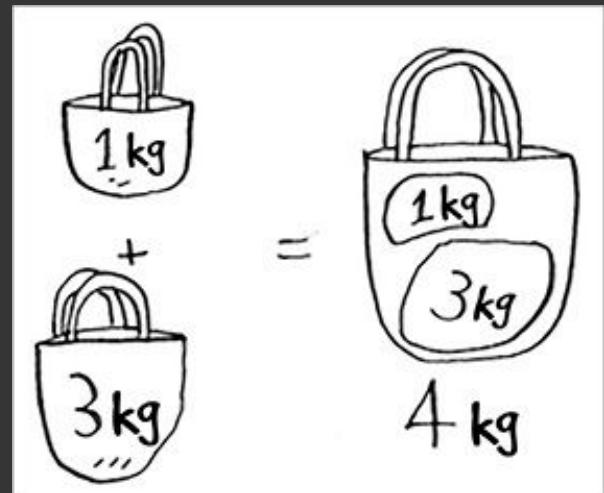


# Programação Dinâmica

Para calcular a **solução ideal** iremos utilizar a técnica de **programação dinâmica**.

Vamos observar como o algoritmo da programação dinâmica funciona. Ele começa com a resolução de subproblemas e vai escalando-os até resolver o problema geral.

No problema da mochila, você começaria resolvendo o problema para mochilas menores (ou “submochilas”) e iria escalando estes problemas até resolver o problema original.



# Programação Dinâmica

Vamos começar mostrando o algoritmo na prática. Cada algoritmo de programação dinâmica começa com uma tabela. Aqui está a tabela para o problema da mochila.

AS COLUNAS SÃO AS CAPACIDADES DE CADA MOCHILA, DE 1 KG A 4 KG

1 2 3 4

1	2	3	4

UMA LINHA PARA CADA ITEM DOS QUAIS PODEMOS ESCOLHER

VIOLÃO  
RÁDIO  
NOTEBOOK

# Programação Dinâmica

As linhas da tabela são os itens e as colunas são as capacidades das mochilas, com valores de 1 quilo, 2 quilos, 3 quilos e 4 quilos. Você precisa destes valores porque eles auxiliarão na resolução dos subproblemas.

A tabela começa vazia, mas você preencherá cada célula dela. Quando a tabela for preenchida, a resposta do problema terá sido encontrada!

	1	2	3	4
VIOLÃO				
RÁDIO				
NOTEBOOK				

# Programação Dinâmica

## □ A linha do violão

Esta é a linha do violão, isso indica que você está tentando colocá-la na sua mochila. Em cada célula, uma decisão simples será tomada: Você roubará ou não o violão? Lembre-se de que você está tentando encontrar o conjunto de itens perfeito para roubar, o qual maximizará o valor do roubo.

A primeira célula indica uma capacidade de peso para mochila igual a 1 quilo. O violão pesa exatamente isso, o que nos confirma que ele cabe na mochila! Assim, o valor desta célula é R\$ 1.500 e ela contém um violão. Vamos começar a preencher a tabela.

	1	2	3	4
VIOLÃO	R\$1500 V			
RÁDIO				
NOTEBOOK				

# Programação Dinâmica

Cada célula da tabela conterá uma lista de todos os itens que cabem na mochila.

Vamos para a próxima célula, que tem uma capacidade de 2 quilos. Bom, com certeza o violão cabe!

	1	2	3	4
VIOLÃO	R\$1500 V	R\$1500 V		
RÁDIO				
NOTEBOOK				

# Programação Dinâmica

E fazemos o mesmo para o restante desta linha. Lembre-se: esta é a primeira linha, portanto você tem apenas o violão para escolher, pois estamos considerando os outros itens como indisponíveis para o roubo.

	1	2	3	4
VIOLÃO	R\$1500 V	R\$1500 V	R\$1500 V	R\$1500 V
RÁDIO				
NOTEBOOK				

# Programação Dinâmica

Você provavelmente está confuso em relação ao porquê de utilizarmos mochilas com capacidades de 1 quilo, 2 quilos e assim por diante, quando o problema especificou que a sua mochila tem capacidade para 16 quilos.

Você se lembra de quando eu disse que a programação dinâmica inicia com problemas menores e os resolve até chegar ao problema geral? Você está resolvendo subproblemas que o ajudarão a resolver o problema especificado. Continue lendo com atenção, e as coisas começarão a fazer mais sentido.

Agora, sua tabela deve estar assim:

	1	2	3	4
VIOLÃO	R\$1500 V	R\$1500 V	R\$1500 V	R\$1500 V
RÁDIO				
NOTEBOOK				

# Programação Dinâmica

Lembre-se de que estamos tentando maximizar o valor contido na mochila. Esta linha representa o melhor palpite atual para este máximo. Assim, agora, de acordo com esta linha, se você tivesse uma mochila com capacidade de 4 quilos, o valor máximo que você poderia roubar seria R\$ 1.500. Você sabe que esta não é a solução final. Ao adentrarmos no algoritmo, teremos estimativas mais refinadas.

	1	2	3	4
VIOLÃO	R\$1500 V	R\$1500 V	R\$1500 V	R\$1500 V
RÁDIO				
NOTEBOOK				

NOSSO MELHOR PALPITE ATUAL SOBRE O QUE O LADRÃO DEVERIA ROUBAR: O VIOLÃO, QUE CUSTA R\$1500

# Programação Dinâmica

## □ A linha do rádio

Vamos preencher a próxima linha, a qual é relativa ao rádio. Agora que você está na segunda linha, pode roubar tanto o rádio quanto o violão. Em cada linha, será possível roubar o item relativo àquela linha e todos os itens das linhas anteriores. Porém o notebook ainda não é uma possibilidade. Vamos começar com a primeira célula, relativa a uma mochila com capacidade de 1 quilo. O valor máximo atual que você pode roubar com uma mochila de 1 quilo é R\$ 1.500.

MÁXIMO ATUAL PARA UMA MOCHILA COM CAPACIDADE PARA 1 kg				
1	2	3	4	
VIOLÃO	R\$1500 V	R\$1500 V	R\$1500 V	R\$1500 V
RÁDIO				
NOTEBOOK				
NOVO MÁXIMO PARA UMA MOCHILA COM CAPACIDADE PARA 1 kg				

# Programação Dinâmica

Você deveria roubar o rádio? Você tem uma mochila com capacidade de 1 quilo. O rádio pode ser colocado dentro dela? Não, ele é muito pesado! E como você não consegue roubar o rádio, o palpite para maximizar o roubo com uma mochila de 1 quilo continua sendo R\$ 1.500.

	1	2	3	4
VIOLÃO	R\$1500 V	R\$1500 V	R\$1500 V	R\$1500 V
RÁDIO	R\$1500 V			
NOTEBOOK				

# Programação Dinâmica

A mesma situação se repete nas próximas duas células, pois a mochila tem capacidade de 2 quilos e 3 quilos, respectivamente. Ou seja, o valor máximo continua sendo R\$ 1.500.

	1	2	3	4
VIOLÃO	R\$1500 V	R\$1500 V	R\$1500 V	R\$1500 V
RÁDIO	R\$1500 V	R\$1500 V	R\$1500 V	
NOTEBOOK				

# Programação Dinâmica

O rádio não pode ser roubado, então o seu palpite continua o mesmo. E se você tiver uma mochila com capacidade para 4 quilos? Ahá! O rádio finalmente pode ser roubado! O valor máximo antigo era R\$ 1.500, mas, com a possibilidade de roubar o rádio, o valor se torna R\$ 3.000! Ou seja, vamos roubar o rádio.

	1	2	3	4
VIOOLÃO	R\$1500 V	R\$1500 V	R\$1500 V	R\$1500 V
RÁDIO	R\$1500 V	R\$1500 V	R\$1500 V	R\$3000 ' ' ' R
NOTEBOOK				

# Programação Dinâmica

Você acabou de atualizar a sua estimativa! Se você tiver uma mochila com capacidade para 4 quilos, conseguirá roubar itens que valem R\$ 3.000. É possível visualizar na tabela que a sua estimativa está sendo incrementada.

	1	2	3	4
VIOLÃO	R\$1500 V	R\$1500 V	R\$1500 V	R\$1500 V
RÁDIO	R\$1500 V	R\$1500 V	R\$1500 V	R\$3000 R
NOTEBOOK				

← ESTIMATIVA ANTIGA

← NOVA ESTIMATIVA

← ESTIMATIVA FINAL

# Programação Dinâmica

## □ A linha do notebook

Vamos fazer o mesmo com o notebook! Ele pesa 3 quilos, portanto não será possível roubá-lo com a mochila de 1 quilo e 2 quilos. Assim, a estimativa para as primeiras duas células continua sendo R\$ 1.500.

	1	2	3	4
VIOLÃO	R\$1500 V	R\$1500 V	R\$1500 V	R\$1500 V
RÁDIO	R\$1500 V	R\$1500 V	R\$1500 V	R\$3000 R
NOTEBOOK	R\$1500 V	R\$1500 V		

# Programação Dinâmica

Com a mochila de 3 quilos, a estimativa antiga para o valor máximo era de R\$ 1.500. Porém você pode escolher roubar o notebook agora, que vale R\$ 2.000. Logo, o novo máximo estimado é R\$ 2.000!

	1	2	3	4
VIOLÃO	R\$1500 V	R\$1500 V	R\$1500 V	R\$1500 V
RÁDIO	R\$1500 V	R\$1500 V	R\$1500 V	R\$3000 R
NOTEBOOK	R\$1500 V	R\$1500 V	R\$2000 ' ' N	

# Programação Dinâmica

Com uma mochila de 4 quilos as coisas ficam interessantes. Esta é uma parte importante, pois a estimativa atual é de R\$ 3.000. Você pode colocar o notebook na mochila, mas ele vale apenas R\$ 2.000.

R\$ 3000 vs R\$ 2000

Humm, isso não é tão bom quanto a outra estimativa. Mas espere aí! O notebook pesa apenas 3 quilos, o que deixa a mochila com capacidade para mais 1 quilo. Ou seja, você pode colocar mais alguma coisa que pesa 1 quilo na mochila.

R\$ 3000 vs  $(R\$ 2000 + \frac{? ? ?}{1kg \text{ DE } ESPA\tilde{\o} \text{ LIVRE}})$

# Programação Dinâmica

Qual o valor máximo que você consegue colocar em uma mochila com 1 quilo livre?  
Bem, você já calculou isso.

VALOR MÁXIMO → PARA 1 KG

1	2	3	4
R\$1500 V	R\$1500 V	R\$1500 V	R\$1500 V
↓ R\$1500	↓ R\$1500	↓ R\$1500	R\$3000 R
V	V	V	... R\$2000 ' ' ' N

# Programação Dinâmica

De acordo com a última estimativa, é possível colocar um violão em 1 quilo de espaço livre, sabendo que ele vale R\$ 1.500. Portanto a comparação real é a seguinte:

$$\begin{matrix} \text{R\$ 3000} & \text{vs} & (\text{R\$ 2000} + \text{R\$ 1500}) \\ \text{RÁDIO} & & \text{NOTEBOOK} & \text{VIOLÃO} \end{matrix}$$

Você pode estar se perguntando por que estivemos calculando os valores máximos para mochilas menores. Espero que agora tudo faça sentido! Quando você tem espaço sobrando, é possível usar as respostas dos subproblemas para descobrir o que colocar no espaço livre.

Assim, a melhor opção é levar o notebook + violão, com valor de R\$ 3.500.

# Programação Dinâmica

- A tabela final ficará assim:

	1	2	3	4
VIOLÃO	R\$1500 V	R\$1500 V	R\$1500 V	R\$1500 V
RÁDIO	R\$1500 V	R\$1500 V	R\$1500 V	R\$3000 R
NOTEBOOK	R\$1500 V	R\$1500 V	R\$2000 N	R\$3500 NV

← A RESPOSTA!

# Programação Dinâmica

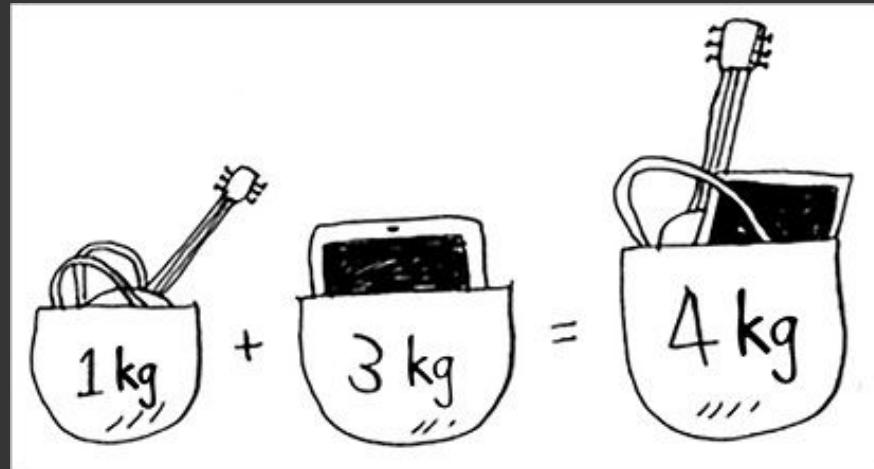
E ali está a resposta: O valor máximo que caberá na mochila é R\$ 3.500, referentes a um violão e um notebook!

Você pode achar que usei uma fórmula diferente para calcular o valor da última célula, mas isso é impressão, pois pulei algumas complexidades desnecessárias enquanto preenchia os valores das células anteriores. Cada célula é calculada com a mesma fórmula, que pode ser vista a seguir:

LINHA COLUNA  
↓ ↓  
CÉLULA[i][j] = MÁXIMO DE {  
1. O MÁXIMO ANTERIOR(VALOR NA CÉLULA[i-1][j])  
VS  
2. VALOR DO ITEM ATUAL + VALOR DO ESPAÇO RESTANTE  
↑  
CÉLULA[i-1][j - PESO DO ITEM]}

# Programação Dinâmica

Você pode usar esta fórmula em cada célula da tabela e deverá encontrar uma tabela igual a esta demonstrada aqui. Você se lembra de quando falei sobre resolver subproblemas? Combinamos as soluções de dois subproblemas para resolver um problema maior.



# Programação Dinâmica

## □ Perguntas frequentes sobre o problema da mochila:

Talvez você ainda pense nessa solução como se fosse algum tipo de mágica. Pois bem, nesta seção trataremos de algumas perguntas frequentes.

- **O que acontece se você adicionar um item?**

Imagine que há um quarto item que você pode roubar, o qual você não havia percebido. Suponha que este item seja um iPhone.

Você deve calcular novamente tudo para levar este item em consideração?

Claro que não. Lembre-se: a programação dinâmica continua construindo progressivamente a sua estimativa.



# Programação Dinâmica

- Até agora, estes são os valores máximos.

	1	2	3	4
VIOLÃO	R\$1500 V	R\$1500 V	R\$1500 V	R\$1500 V
RÁDIO	R\$1500 V	R\$1500 V	R\$1500 V	R\$3000 R
NOTEBOOK	R\$1500 V	R\$1500 V	R\$2000 N	R\$3500 N V

# Programação Dinâmica

O que significa que para uma mochila de 4 quilos você conseguirá roubar um total de R\$ 3.500 em itens. Você achou que este era o valor máximo final, mas agora vamos adicionar uma linha para o iPhone.

	1	2	3	4
VIOLÃO	R\$1500 V	R\$1500 V	R\$1500 V	R\$1500 V
RÁDIO	R\$1500 V	R\$1500 V	R\$1500 V	R\$3000 R
NOTEBOOK	R\$1500 V	R\$1500 V	R\$2000 N	R\$3500 N V
IPHONE				

↑  
NOVA RESPOSTA

# Programação Dinâmica

E agora temos o valor máximo atualizado! Tente preencher esta linha nova antes de prosseguir. Vamos começar com a primeira célula. O iPhone pode ser levado com uma mochila de 1 quilo. O valor máximo antigo era R\$ 1.500, mas o iPhone custa R\$ 2.000. Assim, levaremos o iPhone.

	1	2	3	4
VIOLÃO	R\$1500 V	R\$1500 V	R\$1500 V	R\$1500 V
RÁDIO	R\$1500 V	R\$1500 V	R\$1500 V	R\$3000 R
NOTEBOOK	R\$1500 V	R\$1500 V	R\$2000 N	R\$3500 N V
IPHONE	R\$2000 I			

# Programação Dinâmica

Na próxima célula é possível levar o iPhone e o violão.

R\$1500	R\$1500	R\$1500	R\$1500
V	V	V	V
R\$1500	R\$1500	R\$1500	R\$3000
V	V	V	R
R\$1500	R\$1500	R\$2000	R\$3500
V	V	N	N V
R\$2000	R\$3500		
I	I V		

# Programação Dinâmica

Para a célula 3, não há opção melhor do que levar o iPhone e o violão novamente, então deixe esta célula como está.

Na última célula as coisas ficam interessantes. O valor máximo atual é R\$ 3.500, mas você pode roubar o iPhone e ainda ter 3 quilos de espaço sobrando.

$$\begin{array}{l} \text{R\$3500 vs } \left( \text{R\$2000 + } \frac{\text{? ? ?}}{3\text{kg DE ESPAÇO LIVRE}} \right) \\ \text{NOTEBOOK + VIOLÃO} \qquad \qquad \qquad \text{IPHONE} \end{array}$$

Estes 3 quilos valem R\$ 2.000, sendo R\$ 2.000 do iPhone + R\$ 2.000 do subproblema antigo, totalizando R\$ 4.000! Ou seja, temos um novo máximo!

# Programação Dinâmica

- Aqui está a tabela final:

**Pergunta:** O valor da coluna poderá diminuir? Isto é possível?

**Resposta:** Não. A cada iteração, você armazenará a estimativa máxima atual. A estimativa nunca poderá ficar abaixo do que ela já é!

R\$1500	R\$1500	R\$1500	R\$1500
V	V	V	V
R\$1500	R\$1500	R\$1500	R\$3000
V	V	V	R
R\$1500	R\$1500	R\$2000	R\$3500
V	V	N	N V
R\$2000	R\$3500	R\$3500	R\$4000
I	I V	I V	I N

↑  
NOVA RESPOSTA

# Programação Dinâmica

## □ Programação Dinâmica Vs Divisão e Conquista:

Vocês devem ter percebido que a **Programação Dinâmica** tem o mesmo princípio que a técnica que vimos anteriormente que é a **Divisão e Conquista** que trata em **dividir o problema geral em problemas menores**, porém sua estratégia e implementação do conceito é diferente, vamos recapitular cada uma das técnicas que aprendemos até aqui e discutir suas diferenças:

- **Divisão e Conquista:** É a **técnica de construir/projetar algoritmos** que envolve **recursão**, a idéia é **dividir a instância do problema** em 2 ou mais instâncias menores (**divisão**), e resolvê-la de forma recursiva e **combinar as soluções** em uma solução da instância original (**conquista**).

**Exemplos:** Merge Sort (Ordenação), Quick Sort (Ordenação), Karatsuba (Multiplicação de Inteiros Grandes), Strassen (Multiplicação de Matrizes) e Contagem de Imersões em Vetor.

# Programação Dinâmica

## □ Programação Dinâmica Vs Divisão e Conquista:

- **Programação Dinâmica:** É uma **técnica construir/projetar algoritmos**, para resolver alguns tipos de problemas de forma **mais performática**, esses problemas normalmente tem como características calcular ou processar a **mesma operação diversas vezes** armazenando as soluções anteriores em uma tabela (geralmente implementada como uma matriz ou uma tabela hash), de forma que nosso algoritmo **não precise recalcular a mesma operação** diversas vezes, a técnica de armazenar as soluções de subproblemas é conhecida como memorização.

**Exemplos:** Sequência de Fibonacci, Problema da Mochila, Árvore de Busca Binária Otimizada, Problema do Subset-sum, Multiplicação de Cadeia de Matrizes.

# Programação Dinâmica

## □ A programação dinâmica é mais eficiente do que divisão e conquista?

Divisão e Conquista, funciona melhor quando todos os subproblemas são independentes. Então, escolha a partição que torna o algoritmo mais eficiente e simplesmente combine soluções para resolver todo o problema. A programação dinâmica é necessária quando os subproblemas são dependentes; não sabemos onde dividir o problema.

## □ Exemplo de uso da programação dinâmica?

A programação dinâmica é principalmente uma otimização sobre recursão simples. Por exemplo, se escrevermos solução recursiva simples para Números de Fibonacci, obtemos complexidade de tempo exponencial e se a optimizarmos armazenando soluções de subproblemas, a complexidade de tempo se reduz a linear.

# Programação Dinâmica

## ❑ Quais são as desvantagens da programação dinâmica?

Muitas vezes, o valor de saída é armazenado e nunca é utilizado nos próximos subproblemas durante a execução. Isso leva à utilização desnecessária da memória.

## ❑ É programação dinâmica ascendente ou descendente?

Os problemas de programação dinâmica podem ser resolvidos usando abordagens de baixo para cima ou de cima para baixo. Geralmente, a abordagem de baixo para cima usa a técnica de tabulação, enquanto a abordagem de cima para baixo usa a técnica de recursão (com memorização).

# Programação Dinâmica

## □ Diferenças:

A principal **diferença** entre **dividir e conquistar** e a **programação dinâmica** é que dividir e conquistar **combina as soluções dos subproblemas** para obter a solução do problema principal, enquanto a programação dinâmica usa o **resultado dos subproblemas** para encontrar a solução ótima do problema principal.

# Programação Dinâmica

## □ Concluindo

- A programação dinâmica é útil quando você está tentando otimizar algo em relação a um limite.
- Você pode utilizar a programação dinâmica quando o problema puder ser dividido em subproblemas discretos.
- Todas as soluções em programação dinâmica envolvem uma tabela.
- Os valores nas células são, geralmente, o que você está tentando otimizar.
- Cada célula é um subproblema, então pense sobre como é possível dividir este subproblema em outros subproblemas.
- Não existe uma fórmula única para calcular uma solução em programação dinâmica.

# Conclusão

As estruturas de dados são essenciais em projetos de desenvolvimento de algoritmos e programas em geral. Esse é um campo de estudo que deve fazer parte da formação de todo programador, pois só assim os profissionais serão realmente capazes de resolver problemas.

Ao estudarmos algoritmos, podemos aprender técnicas de análise que nos permitem comparar e encontrar soluções baseadas unicamente em suas próprias características.

Como programadores, além de nossa capacidade de resolver problemas, também precisaremos conhecer e entender técnicas para avaliação de soluções. Ao final, sempre há muitas maneiras de resolver um problema e espero que este conteúdo te ajudará a entender e desenvolver algoritmos mais eficientes no dia a dia.

# Obrigado!

Alguma Pergunta ?

- ★ willian\_brito00@hotmail.com
- ★ www.linkedin.com/in/willian-ferreira-brito
- ★ github.com/willian-brito

