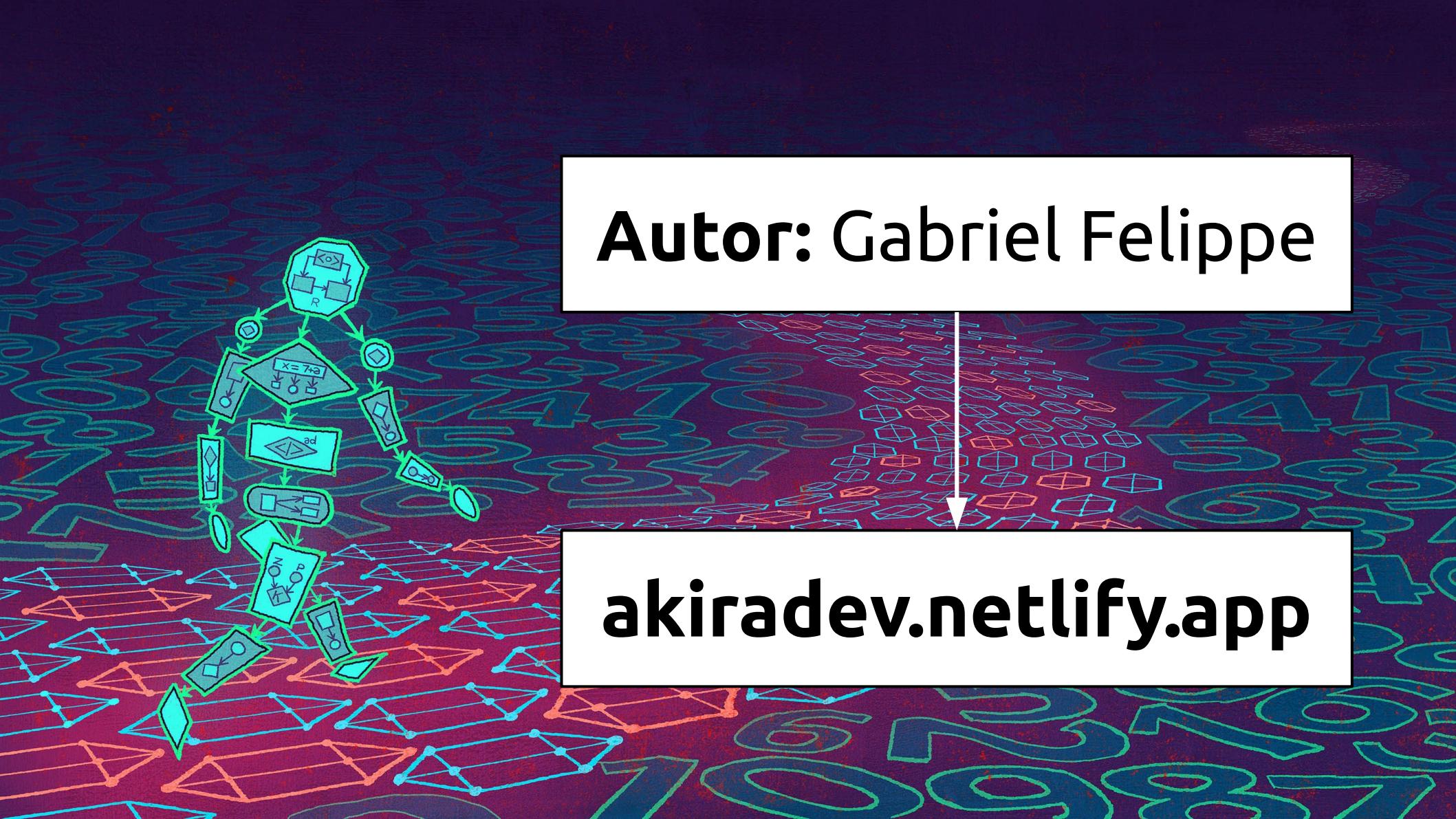


0
1
0
1
1
1

# Estruturas de Dados

&



# Autor: Gabriel Felippe

[akiradev.netlify.app](https://akiradev.netlify.app)

# Introdução

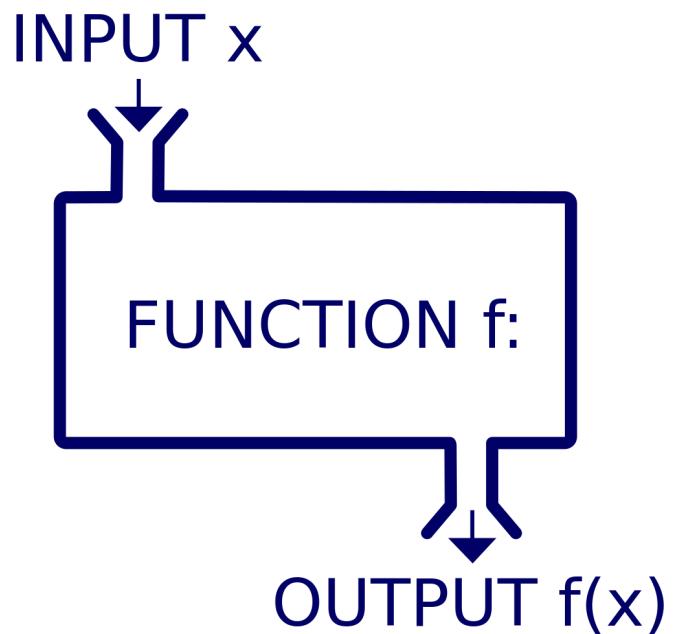
---

- Em **matemática** e **ciência da computação**, um **algoritmo** é uma sequência finita de instruções bem definidas e implementáveis por computador, normalmente para resolver uma classe de problemas ou realizar um cálculo.
- Os algoritmos são sempre inequívocos e são usados como especificações para realizar cálculos, processamento de dados, raciocínio automatizado e outras tarefas.
- Como um método eficaz, um algoritmo pode ser expresso em uma quantidade finita de **espaço** e **tempo** e em uma linguagem formal bem definida para calcular uma **função**.

# Introdução

---

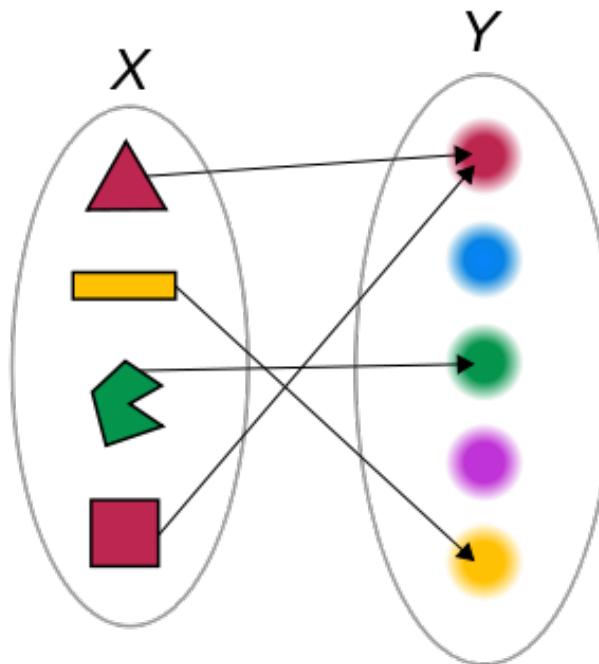
- ♦ Em **matemática**, uma **função** é uma **relação binária** entre dois conjuntos que associa a cada elemento do primeiro conjunto a exatamente um elemento do segundo conjunto.
- ♦ Representação esquemática de uma função descrita metaforicamente como uma "máquina" ou "black box" que para cada **input** produz um **output** correspondente.



# Introdução

---

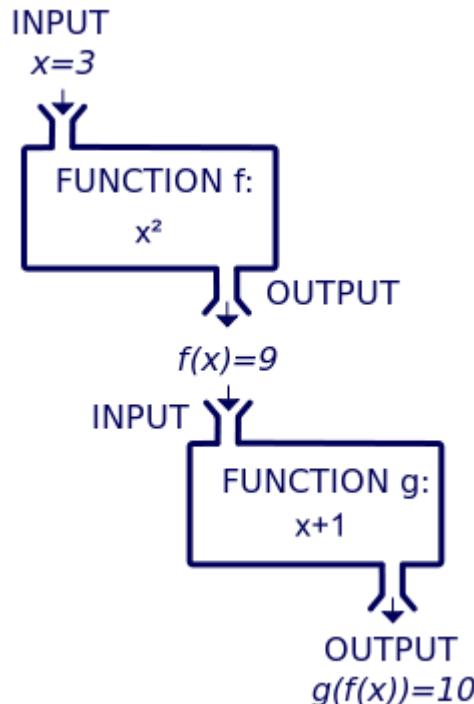
- ♦ Por exemplo, podemos ter uma função que associa qualquer uma das quatro formas coloridas ( $X$ ) à sua cor ( $Y$ ):



# Introdução

- ◆ Uma função composta  $g(f(x))$  pode ser visualizada como a combinação de duas "máquinas":

$$\begin{aligned}f(x) &= x^2 \\f(3) &= 3^2 \\f(3) &= 9\end{aligned}$$



$$\begin{aligned}g(x) &= x + 1 \\g(9) &= 9 + 1 \\g(10) &= 10\end{aligned}$$

# Introdução

---

- ♦ A própria palavra **algoritmo** deriva do nome do matemático do século IX, **Muhammad ibn Mūsā al-Khwārizmī**, cujo nisba (identificando-o como de Khwarazm) foi latinizado como **Algoritmi**.

O conceito de algoritmo existe desde a antiguidade. Algoritmos aritméticos, como um algoritmo de divisão, eram usados pelos antigos matemáticos da Babilônia c. 2500 AC e matemáticos egípcios c. 1550 AC. Mais tarde, os matemáticos gregos usaram algoritmos em 240 aC na **sieve of Eratosthenes** para encontrar números primos e o algoritmo euclidiano para encontrar o maior divisor comum de dois números. Matemáticos árabes como al-Kindi no século 9 usavam algoritmos criptográficos para quebra de código, com base na análise de frequência.



# Introdução

---

- ♦ Informalmente, um **algoritmo** é qualquer procedimento computacional bem definido que assume algum valor, ou conjunto de valores, como **input** e produz algum valor, ou conjunto de valores, como **output**.
- ♦ Também podemos ver um algoritmo como uma ferramenta para resolver um **problema computacional** bem especificado. O enunciado do problema especifica em termos gerais a relação desejada de **input/output**.
- ♦ O algoritmo descreve um **procedimento computacional** específico para alcançar essa relação de **input/output**.

# Introdução

---

- ◆ Por exemplo, podemos precisar ordenar uma sequência de números em ordem crescente. Este problema surge freqüentemente na prática e fornece um terreno fértil para a introdução de muitas técnicas de design padrão e ferramentas de análise.
- ◆ Aqui está como definimos formalmente o **problema de ordenação**:
- ◆ **Input:** Uma sequência de  $n$  números  $\{a_1, a_2, \dots, a_n\}$
- ◆ **Output:** Uma permutação (reordenamento)  $\{a'_1, a'_2, \dots, a'_n\}$  da sequência de input, tal que  $a'_1 \leq a'_2 \leq \dots \leq a'_n$

# Introdução

---

- ♦ Por exemplo, dada a sequência de input {13, 42, 27, 8, 3, 19, 1} um algoritmo de ordenação retorna como output a sequência {1, 3, 8, 13, 19, 27, 42}.
- ♦ Essa sequência de input é chamada de **instância** do problema de ordenação.
- ♦ Em geral, uma **instância de um problema** consiste do **input** (satisfazendo quaisquer restrições impostas na definição do problema) necessário para calcular uma solução para o problema.

# Introdução

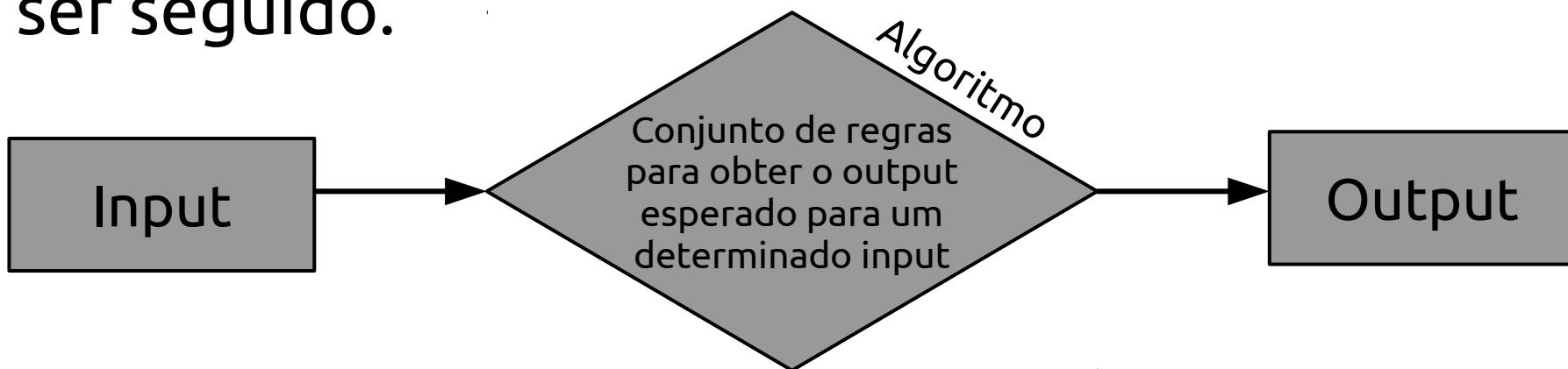
---

- Como muitos programas o usam como uma etapa intermediária, a **ordenação** é uma **operação fundamental** na **ciência da computação**.
- Como resultado, temos um grande número de bons algoritmos de ordenação à nossa disposição.
- Qual algoritmo é o melhor para uma determinada aplicação depende - entre outros fatores - do número de itens a serem ordenados, até que ponto os itens já estão ordenados de alguma forma, possíveis restrições sobre os valores dos itens, a arquitetura do computador e o tipo de dispositivos de armazenamento a serem usados: memória principal, discos ou até mesmo fitas.

# Introdução

---

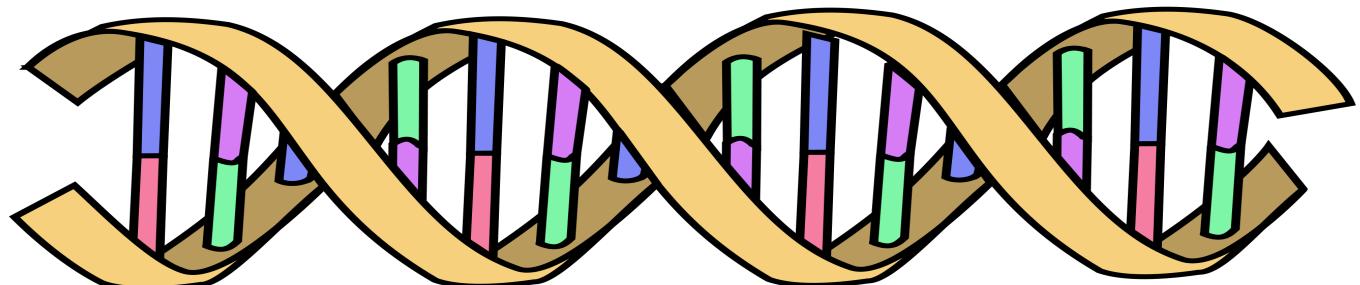
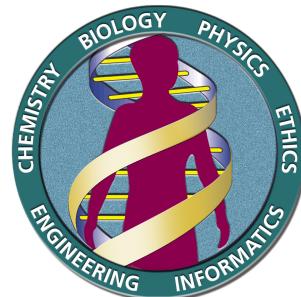
- ♦ Um **algoritmo** pode ser especificado em português, como um programa de computador ou até mesmo como um projeto de hardware.
- ♦ O único requisito é que a especificação deve fornecer uma descrição precisa do procedimento computacional a ser seguido.



# Tipos de Problemas

---

- ◆ Que tipos de problemas são resolvidos por algoritmos?
- ◆ O **Human Genome Project** fez um grande progresso em direção às metas de identificar todos os 100.000 genes no DNA humano, determinar as sequências dos 3 bilhões de pares de bases químicas que compõem o DNA humano, armazenar essas informações em bancos de dados e desenvolver ferramentas para análise de dados.



# Tipos de Problemas

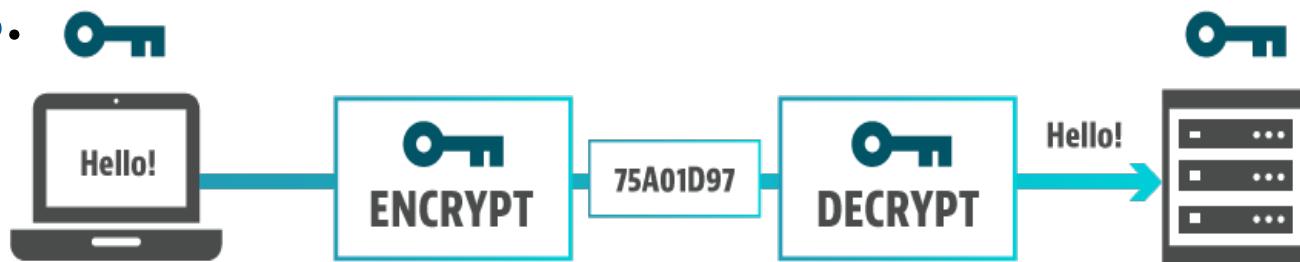
---

- ◆ A **Internet** permite que pessoas em todo o mundo acessem e recuperem rapidamente grandes quantidades de informações.
- ◆ Com a ajuda de algoritmos inteligentes, sites na Internet são capazes de gerenciar e manipular esse grande volume de dados.
- ◆ Exemplos de problemas que fazem uso essencial de algoritmos incluem encontrar boas rotas nas quais os dados irão viajar e usar um **search engine** para encontrar rapidamente as páginas nas quais residem informações específicas.

# Tipos de Problemas

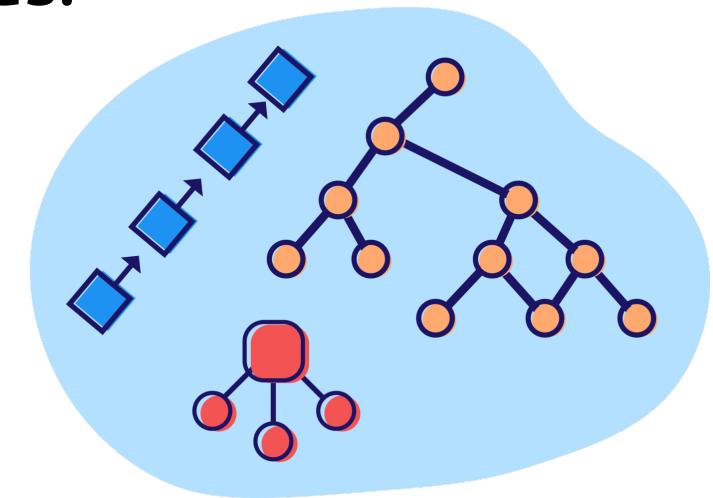
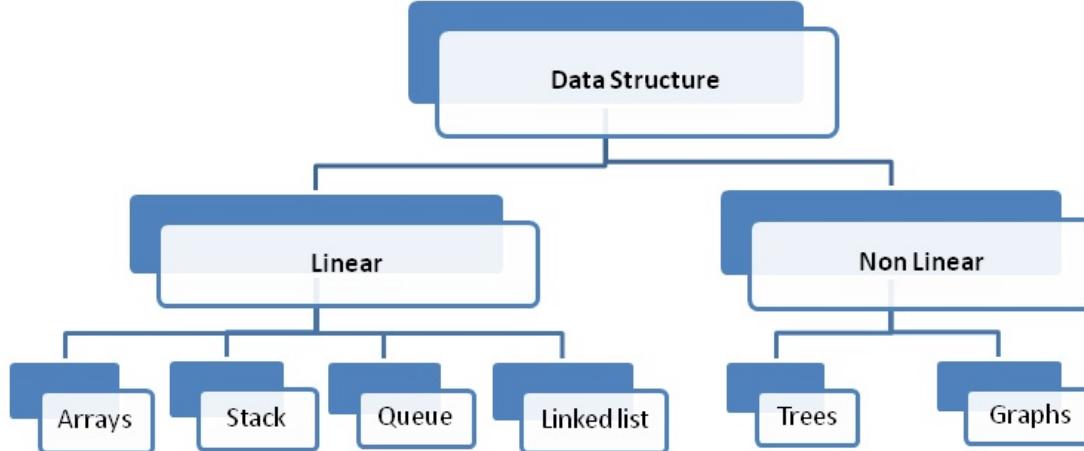
---

- O **comércio eletrônico** permite que bens e serviços sejam negociados e trocados eletronicamente e depende da privacidade das informações pessoais, como números de cartão de crédito, senhas e extratos bancários.
- As principais tecnologias usadas no comércio eletrônico incluem criptografia de chave pública e assinaturas digitais, que são baseadas em **algoritmos numéricos** e **teoria dos números**.



# Estruturas de Dados

- Uma **estrutura de dados** é uma forma de **armazenar** e **organizar** dados para facilitar o acesso e as modificações.
- Nenhuma estrutura de dados única funciona bem para todos os fins, por isso é importante conhecer os pontos fortes e as limitações de várias delas.



# Problemas Difíceis

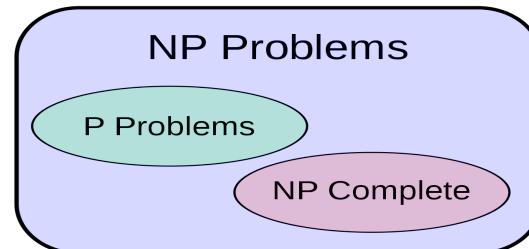
---

- ♦ Um assunto muito importante no estudo dos algoritmos é a respeito da **eficiência de algoritmos**.
- ♦ Nossa medida usual de eficiência é a velocidade, ou seja, quanto tempo um algoritmo leva para produzir seu resultado.
- ♦ Existem alguns problemas, entretanto, para os quais nenhuma solução eficiente é conhecida.
- ♦ Existe um subconjunto interessante desses problemas, que são conhecidos como **NP-complete**.

# NP-Complete

---

- ◆ Por que os problemas NP-complete são interessantes?
- ◆ Primeiro, embora nenhum algoritmo eficiente para um problema NP-complete tenha sido encontrado, ninguém jamais provou que um algoritmo eficiente para um não possa existir.
- ◆ Em outras palavras, ninguém sabe se existem ou não algoritmos eficientes para problemas NP-complete.



# NP-Complete

---

- ♦ Em segundo lugar, o conjunto de problemas NP-complete tem a propriedade notável de que, se existe um algoritmo eficiente para qualquer um deles, então existem algoritmos eficientes para todos eles.
- ♦ Terceiro, vários problemas NP-complete são semelhantes, mas não idênticos, a problemas para os quais conhecemos algoritmos eficientes. Os cientistas da computação estão intrigados em como uma pequena mudança na definição do problema pode causar uma grande mudança na eficiência do algoritmo mais conhecido.

# NP-Complete

---

- ♦ Como um exemplo concreto, considere uma empresa de entrega com um depósito central. A cada dia, ela carrega cada caminhão de entrega no depósito e os envia para entregar mercadorias em vários endereços.
- ♦ No final do dia, cada caminhão deve voltar ao depósito para que esteja pronto para ser carregado no dia seguinte.
- ♦ Para reduzir custos, a empresa deseja selecionar uma ordem de paradas de entrega que produza a menor distância geral percorrida por cada caminhão.
- ♦ Este é o conhecido "problema do caixeiro viajante" e é NP-complete.

# Paralelismo

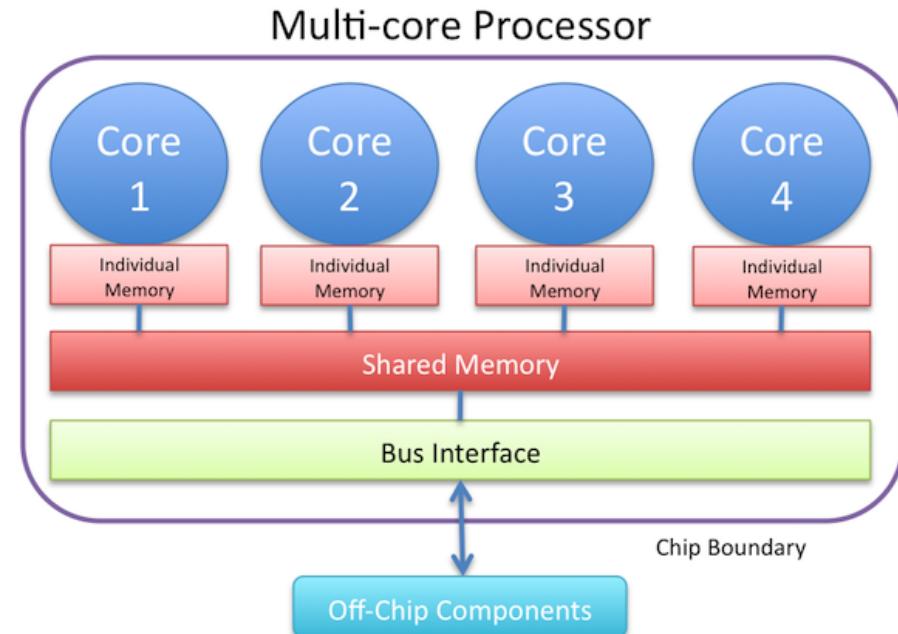
---

- ♦ Por muitos anos, pudemos contar com a **velocidade do clock** do **processador** aumentando a uma taxa constante.
- ♦ Limitações físicas apresentam um obstáculo fundamental para velocidades de clock sempre crescentes, no entanto: como a densidade de energia aumenta superlinearmente com a velocidade de clock, os chips correm o risco de derreter quando suas velocidades de clock se tornam altas o suficiente.
- ♦ Para realizar mais cálculos por segundo, portanto, os chips estão sendo projetados para conter não apenas um, mas vários "**núcleos**" de processamento.

# Paralelismo

- ♦ Podemos comparar esses computadores multicore a vários computadores sequenciais em um único chip; em outras palavras, eles são um tipo de “computador paralelo”.

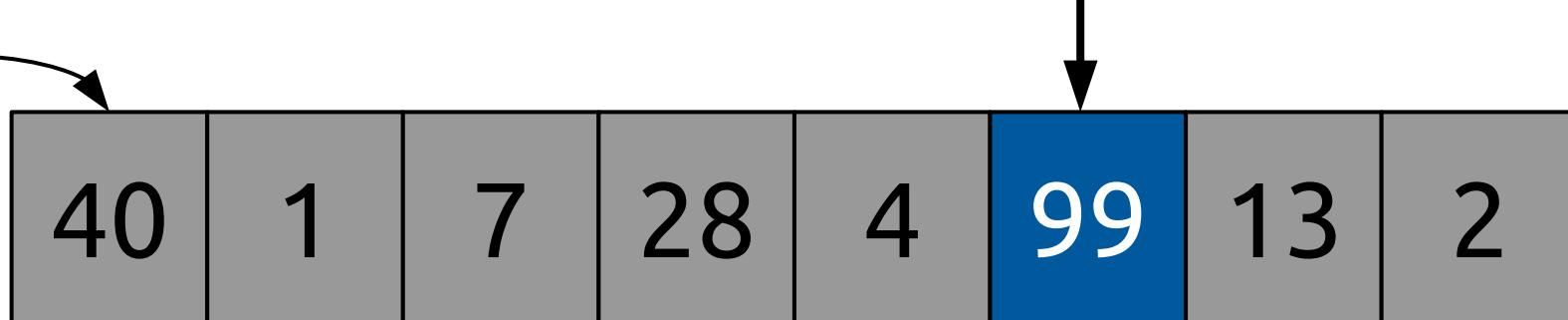
“A fim de obter o melhor desempenho de computadores de vários núcleos precisamos projetar algoritmos com o paralelismo em mente”.



# Exemplo de Algoritmo

---

- ♦ Um dos algoritmos mais simples é encontrar o maior número em uma lista de números de ordem aleatória.
- ♦ Encontrar a solução requer olhar para todos os números da lista.



Olhar cada elemento

# Exemplo de Algoritmo

---

- ♦ A partir disso segue um algoritmo simples, que pode ser expresso em uma **descrição de alto nível** em prosa portuguesa, como:
  1. Se não houver números no conjunto, não haverá o número mais alto.
  2. Suponha que o primeiro número do conjunto seja o maior número do conjunto.
  3. Para cada número restante no conjunto: se este número for maior que o maior número atual, considere este número como o maior número no conjunto.
  4. Quando não houver mais números no conjunto para iterar, considere o maior número atual como o maior número do conjunto.

# Exemplo de Algoritmo

- Descrição (quase) formal: escrito em prosa, mas muito mais próximo da linguagem de alto nível de um programa de computador, o seguinte é a codificação mais formal do algoritmo em **pseudocódigo** ou **código pidgin**:

**Algoritmo** MaiorNúmero

**Input:** Uma lista de números L.

**Output:** O maior número na lista L.

```
if L.size = 0 return null
largest ← L[0]
for each item in L, do
    if item > largest, then
        largest ← item
return largest
```

"return" termina o algoritmo  
e produz o respectivo valor.

" $\leftarrow$ " denota atribuição.  
Por exemplo, "largest  $\leftarrow$  item"  
significa que o valor do maior muda para o valor do item.

# Exemplo de Algoritmo

- Podemos também facilmente implementar o algoritmo para encontrar o maior número na linguagem **Python**:

```
def maior_número(lista):
    if len(lista) == 0:
        return None
    maior = lista[0]
    for item in lista:
        if item > maior:
            maior = item
    return maior

lista = [40, 1, 7, 28, 4, 99, 13, 2]
print(maior_número(lista)) # 99
```



# O Algoritmo Insertion Sort

---

- ◆ Nosso primeiro algoritmo de ordenação irá resolver o problema de ordenação introduzido na introdução desta apresentação.
- ◆ **Input:** Uma sequência de  $n$  números  $\{a_1, a_2, \dots, a_n\}$
- ◆ **Output:** Uma permutação (reordenamento)  $\{a'_1, a'_2, \dots, a'_n\}$  da sequência de input, tal que  $a'_1 \leq a'_2 \leq \dots \leq a'_n$
- ◆ Os números que desejamos ordenar também são conhecidos como **chaves**.
- ◆ Embora, conceitualmente, estejamos ordenando uma sequência, o **input** chega até nós na forma de um **array** com  $n$  elementos.

# O Algoritmo Insertion Sort

---

- ♦ **Insertion Sort** é um algoritmo eficiente para ordenar um pequeno número de elementos.
- ♦ O Insertion Sort funciona da mesma forma que muitas pessoas ordenam uma mão de cartas de baralho.
- ♦ Começamos com a mão esquerda vazia e as cartas voltadas para baixo na mesa. Em seguida, removemos uma carta de cada vez da mesa e o inserimos na posição correta na mão esquerda.
- ♦ Para encontrar a posição correta para uma carta, comparamos com cada uma das cartas já na mão, da direita para a esquerda. Em todos os momentos, as cartas mantidas na mão esquerda são ordenadas.

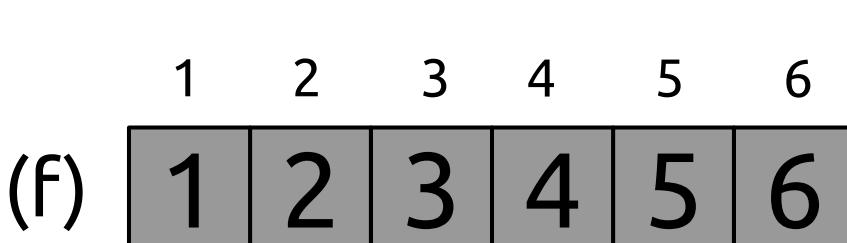
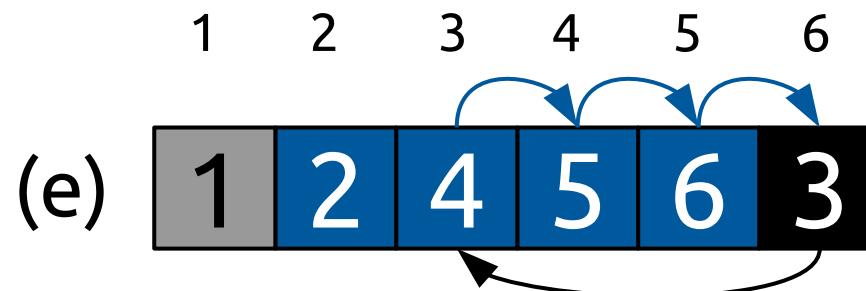
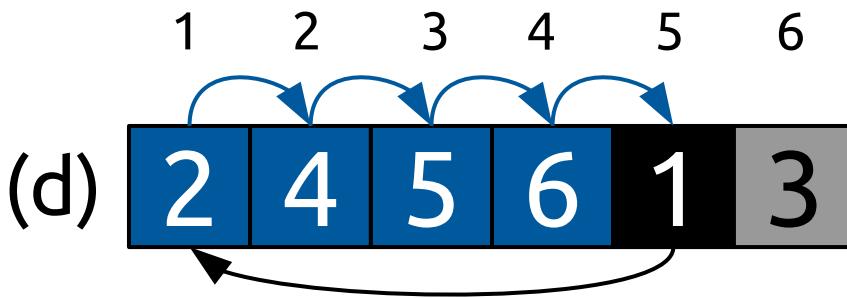
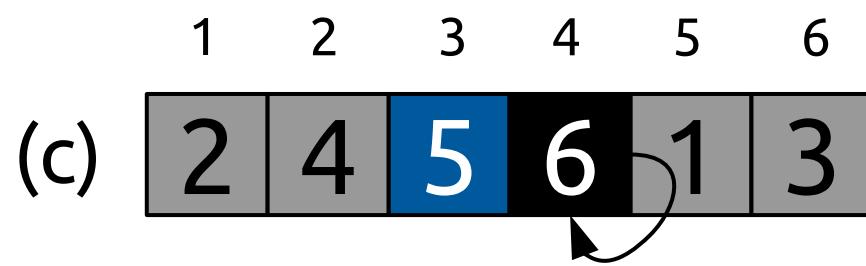
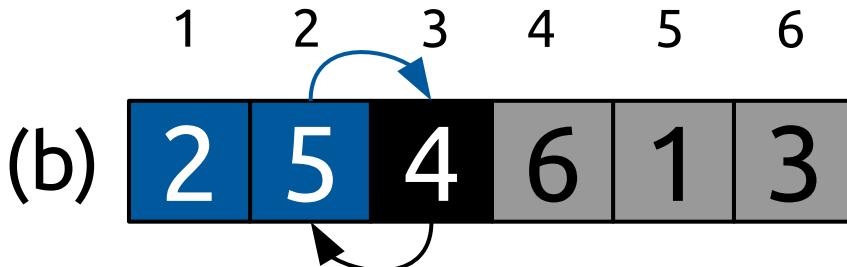
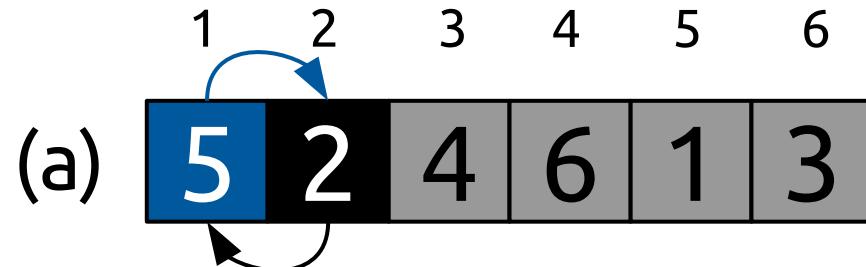


# O Algoritmo Insertion Sort

---

- Ilustraremos as operações do algoritmo Insertion Sort no array  $A = \{5, 2, 4, 6, 1, 3\}$ .
- Os **índices** do array aparecem acima dos retângulos e os **valores** armazenados nas posições do array aparecem dentro dos retângulos.
- **(a) – (e)**: As iterações do **loop for** das linhas 1–8. Em cada iteração, o retângulo preto contém a chave tirada de  $A[j]$ , que é comparada com os valores em retângulos azuis à sua esquerda no teste da linha 5.
- As setas azuis mostram os valores do array movidos uma posição para a direita na linha 6 e as setas pretas indicam para onde a chave se move na linha 8. **(f)**: O array ordenado final.

# O Algoritmo Insertion Sort



# O Algoritmo Insertion Sort

---

- ◆ Apresentamos o **pseudocódigo** para o algoritmo Insertion Sort, ao qual recebe como parâmetro um array **A[1...n]** contendo uma sequência de comprimento **n** que deve ser ordenada. (No código, o número **n** de elementos em A é denotado por **A.length**)
- ◆ O algoritmo ordena os números de input **in place**: ele reorganiza os números dentro do array **A**, com no máximo um número constante deles armazenado fora do array a qualquer momento.
- ◆ O array de input **A** contém a sequência de output ordenada quando o procedimento Insertion Sort é concluído.

# O Algoritmo Insertion Sort

---

- ♦ Pseudocódigo para o algoritmo Insertion Sort:

```
1 for j = 2 to A:length
2   key = A[j]
3   // Insere A[j] na sequência ordenada A[1...j - 1]
4   i = j - 1
5   while i > 0 and A[j] > key
6     A[i + 1] = A[i]
7     i = i - 1
8   A[i + 1] = key
```

# O Algoritmo Insertion Sort

---

- ◆ A ilustração apresentada mostra como o algoritmo funciona para o array  $A = \{5, 2, 4, 6, 1, 3\}$
- ◆ O índice  $j$  indica a “carta atual” sendo inserida na mão.
- ◆ No início de cada iteração do loop **for**, que é indexado por  $j$ , o subarray consistindo dos elementos  $A[1\dots j - 1]$  constitui a mão atualmente ordenada, e o subarray restante  $A[j + 1\dots n]$  corresponde à pilha de cartas ainda na mesa.
- ◆ Na verdade, os elementos  $A[1\dots j - 1]$  são os elementos originalmente nas posições 1 até  $j - 1$ , mas agora ordenados.

# O Algoritmo Insertion Sort

---

- ♦ Declaramos essas propriedades de  $A[1 \dots j - 1]$  formalmente como um **invariante de loop**.
- ♦ Na ciência da computação, uma invariante de loop é uma propriedade de um loop de programa que é verdadeira antes (e depois) de cada iteração. É uma afirmação lógica.
  - No início de cada iteração do loop for das linhas 1-8, o subarray  $A[1 \dots j - 1]$  consiste dos elementos originalmente em  $A[1 \dots j - 1]$ , mas em ordem ordenada.

# O Algoritmo Insertion Sort

---

- ♦ Usamos invariantes de loop para nos ajudar a entender por que um algoritmo está correto. Devemos mostrar três aspectos sobre um invariante de loop:
  - **Inicialização:** É **true** antes da primeira iteração do loop.
  - **Manutenção:** Se for **true** antes de uma iteração do loop, ele permanecerá **true** antes da próxima iteração.
  - **Término:** Quando o loop termina, o invariante nos dá uma propriedade útil que ajuda a mostrar que o algoritmo está correto.

# O Algoritmo Insertion Sort

---

- Quando as **duas primeiras propriedades** são mantidas, a invariante do loop é verdadeira antes de cada iteração do loop. (Claro, estamos livres para usar fatos estabelecidos além do próprio invariante do loop para provar que o invariante do loop permanece true antes de cada iteração).
- Observe a semelhança com a indução matemática, onde para provar que uma propriedade é válida, você prova um **caso base** e uma **etapa indutiva**. Aqui, mostrando que o invariante é válido antes da primeira iteração corresponde ao caso base, e mostrando que o invariante é válido de iteração para iteração corresponde ao passo indutivo.

# O Algoritmo Insertion Sort

---

- ♦ A **terceira propriedade** é talvez a mais importante, uma vez que estamos usando o invariante de loop para mostrar a correção.
- ♦ Normalmente, usamos a invariante do loop junto com a condição que causou o encerramento do loop. A propriedade de terminação difere de como geralmente usamos a indução matemática, na qual aplicamos a etapa indutiva infinitamente; aqui, paramos a “indução” quando o loop termina.
- ♦ A seguir veremos como essas propriedades são válidas para o algoritmo Insertion Sort.

# O Algoritmo Insertion Sort

---

- ♦ **Inicialização:** Começamos mostrando que o invariante do loop é válido antes da primeira iteração do loop, quando  $j = 2$ .
- ♦ O subarray  $A[1\dots j - 1]$ , portanto, consiste apenas no único elemento  $A[1]$ , que é de fato o original elemento em  $A[1]$ .
- ♦ Além disso, esse subarray é ordenado (trivialmente, é claro), o que mostra que a invariante do loop é mantida antes da primeira iteração do loop.

# O Algoritmo Insertion Sort

---

- **Manutenção:** A seguir, abordamos a segunda propriedade: mostrando que cada iteração mantém a invariante do loop.
- Informalmente, o corpo do loop for funciona movendo  $A[j - 1]$ ,  $A[j - 2]$ ,  $A[j - 3]$  e assim por diante em uma posição para a direita até encontrar a posição adequada para  $A[j]$  (linhas 4-7) , em cujo ponto ele insere o valor de  $A[j]$  (linha 8).
- O subarray  $A[1\dots j]$  consiste então nos elementos originalmente em  $A[1\dots j]$ , mas ordenados. Incrementar  $j$  para a próxima iteração do loop for preserva a invariante do loop.
- Um tratamento mais formal da segunda propriedade exigiria que declarássemos e mostrássemos um laço invariante para o loop **while** das linhas 5–7.

# O Algoritmo Insertion Sort

---

- **Término:** Finalmente, examinamos o que acontece quando o loop termina.
- A condição que faz com que o loop for termine é que  $j > A.length = n$ .
- Como cada iteração do loop aumenta  $j$  em 1, devemos ter  $j = n + 1$  naquele momento.
- Substituindo  $n + 1$  por  $j$  na formulação de invariante de loop, temos que o subarray  $A[1...n]$  consiste nos elementos originalmente em  $A[1...n]$ , mas ordenados.
- Observando que o subarray  $A[1...n]$  é o array inteiro, concluímos que o array inteiro está ordenado. Portanto, o algoritmo está correto.

# O Algoritmo Insertion Sort

- A seguir temos o algoritmo Insertion Sort em [Python](#):

```
def insertion_sort(array):
    for j in range(1, len(array)):
        key = array[j]
        i = j

        while i > 0 and array[i - 1] > key:
            array[i] = array[i - 1]
            i = i - 1

        array[i] = key

array = [5, 2, 4, 6, 1, 3]
insertion_sort(array)
print(f'Array ordenado: {array}') # Array ordenado: [1, 2, 3, 4, 5, 6]
```



# Analisando Algoritmos

---

- Analisar um algoritmo passou a significar prever os recursos que o algoritmo requer.
- Ocasionalmente, recursos como memória, largura de banda de comunicação ou hardware de computador são a principal preocupação, mas na maioria das vezes é o tempo computacional que queremos medir.
- Geralmente, ao analisar vários algoritmos candidatos para um problema, podemos identificar um mais eficiente.
- Essa análise pode indicar mais de um candidato viável, mas muitas vezes podemos descartar vários algoritmos inferiores no processo.

# Analisando Algoritmos

---

- ♦ Antes de podermos analisar um algoritmo, devemos ter um modelo da tecnologia de implementação que usaremos, incluindo um modelo para os recursos dessa tecnologia e seus custos.
- ♦ Para a maior parte desta apresentação, assumiremos um modelo genérico de computação com um **processador** e **random-access-machine** (RAM) como nossa tecnologia de implementação e compreenderemos que nossos algoritmos serão implementados como programas de computador.
- ♦ No **modelo RAM**, as instruções são executadas uma após a outra, sem operações simultâneas.

# Analisando Algoritmos

---

- ♦ O **modelo RAM** contém instruções comumente encontradas em computadores reais:
  - **Aritmética** (como adicionar, subtrair, multiplicar, dividir, resto, *floor*, *ceiling*);
  - **Movimentação de dados** (carregar, armazenar, copiar);
  - **Controle** (ramificação condicional e incondicional, chamada de sub-rotina e retorno);
- ♦ Cada uma dessas instruções leva um **tempo** constante.

# Analisando Algoritmos

---

- Os tipos de dados no modelo RAM são **inteiros** e **ponto flutuante** (para armazenar números reais).
- Analisar até mesmo um algoritmo simples no modelo de RAM pode ser um desafio.
- As ferramentas matemáticas necessárias podem incluir **combinatória**, **teoria da probabilidade**, **destreza algébrica** e a capacidade de identificar os termos mais significativos em uma fórmula.
- Como o comportamento de um algoritmo pode ser diferente para cada **input** possível, precisamos de um meio para resumir esse comportamento em fórmulas simples e de fácil compreensão.

# Analisando Algoritmos

---

- ♦ No modelo RAM, não tentamos modelar a hierarquia de memória que é comum em computadores contemporâneos. Ou seja, não modelamos caches ou memória virtual.
- ♦ Embora normalmente selecionemos apenas um modelo de máquina para analisar um determinado algoritmo, ainda temos muitas opções ao decidir como expressar nossa análise.
- ♦ Gostaríamos de uma maneira que seja simples de escrever e manipular, mostre as características importantes dos requisitos de recursos de um algoritmo e suprima detalhes tediosos.

# Análise do Insertion Sort

---

- ◆ O tempo gasto pelo procedimento **Insertion Sort** depende do **input**: ordenar mil números leva mais tempo do que ordenar três números.
- ◆ Além disso, o Insertion Sort pode levar diferentes quantidades de tempo para ordenar duas sequências de input do mesmo tamanho, dependendo de quão quase ordenadas elas já estão.
- ◆ Em geral, o tempo gasto por um algoritmo aumenta com o tamanho do input, por isso é tradicional descrever o **tempo de execução** de um programa em função do **tamanho de seu input**.

# Análise do Insertion Sort

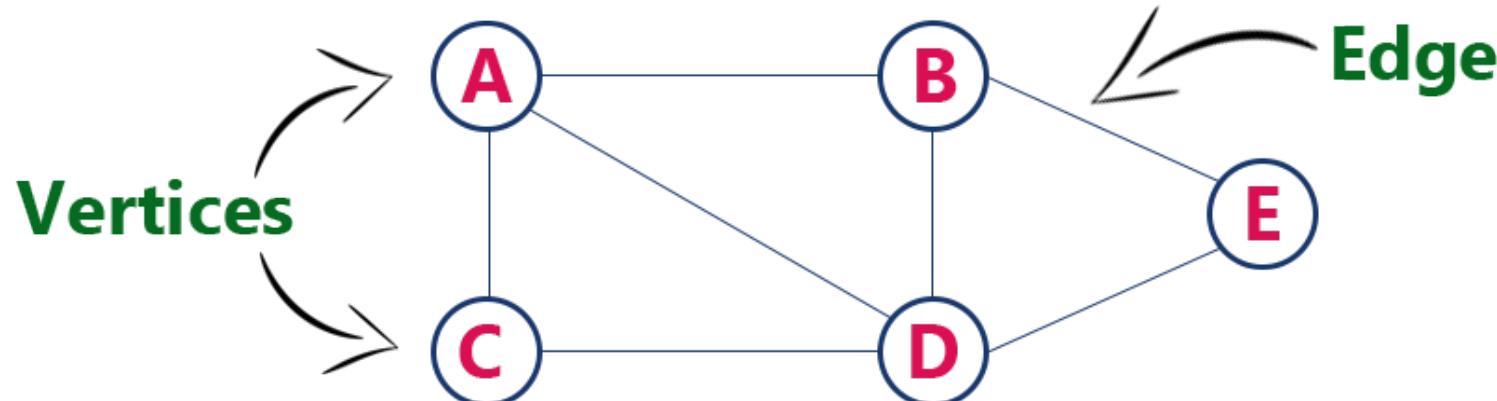
---

- ♦ A melhor noção para o **tamanho do input** depende do problema que está sendo estudado.
- ♦ Para muitos problemas, como **ordenação** ou **computação de transformações discretas de Fourier**, a medida mais natural é o **número de itens no input**.
- ♦ Por exemplo, o tamanho do array **n** para ordenação.
- ♦ Para muitos outros problemas, como a multiplicação de dois inteiros, a melhor medida do tamanho de input é o número total de bits necessários para representar a entrada em notação binária comum.

# Análise do Insertion Sort

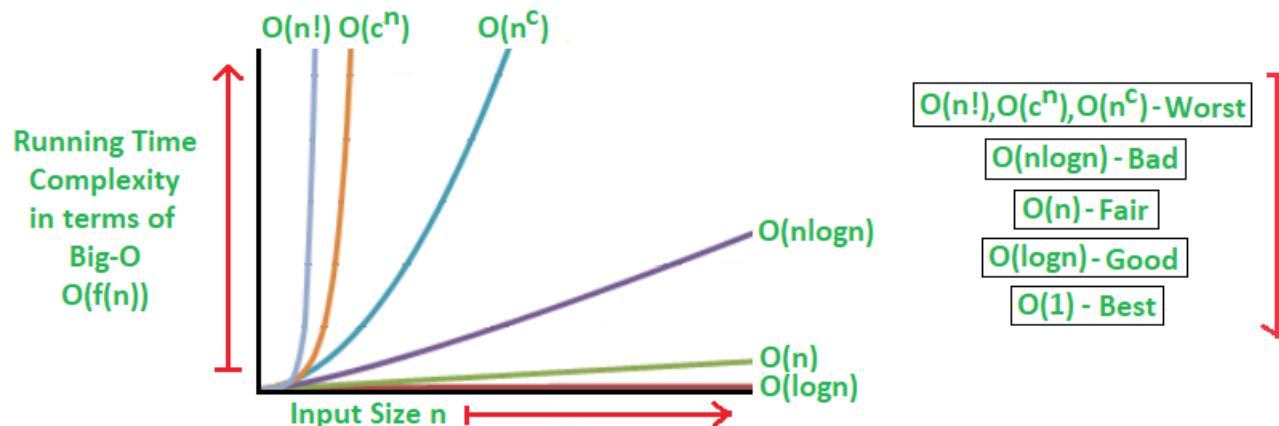
---

- ♦ Às vezes, é mais apropriado descrever o tamanho do input com dois números em vez de um.
- ♦ Por exemplo, se o input para um algoritmo é um **Grafo**, o tamanho do input pode ser descrito pelo número de vértices e arestas no gráfico.



# Análise do Insertion Sort

- ♦ O **tempo de execução** de um algoritmo em um determinado input é o número de operações primitivas ou “etapas” executadas.
- ♦ É conveniente definir a noção de “etapa” para que seja o mais independente da máquina possível.



# Análise do Insertion Sort

---

- ♦ Por enquanto, vamos adotar a seguinte visão:
  - É necessário um período de tempo constante para executar cada linha de nosso pseudocódigo.
  - Uma linha pode levar um período de tempo diferente do que outra linha, mas devemos assumir que cada execução da  $i$ -ésima linha leva um tempo  $c_i$ , onde  $c_i$  é uma **constante**.
  - Esse ponto de vista está de acordo com o modelo RAM e também reflete como o pseudocódigo seria implementado na maioria dos computadores reais.

# Análise do Insertion Sort

---

- ♦ Começamos apresentando o procedimento **Insertion Sort** com o “custo” de tempo de cada instrução e o número de vezes que cada instrução é executada.
- ♦ Para cada  $j = 2, 3, \dots, n$ , onde  $n = A.length$ , deixamos  $t_j$  denotar o número de vezes que o teste do loop **while** (na linha 5) é executado para aquele valor de  $j$ .
- ♦ Quando um loop **for** ou **while** sai da maneira usual (ou seja, devido ao teste no cabeçalho do loop), o teste é executado uma vez mais do que o corpo do loop. Presumimos que os **comentários** não são instruções executáveis e, portanto, não levam tempo.

# Análise do Insertion Sort

- ◆ Pseudocódigo para o algoritmo Insertion Sort:

```
1 for j = 2 to A:length
2   key = A[j]
3   // Insere A[j] na sequência ordenada A[1...j - 1]
4   i = j - 1
5   while i > 0 and A[j] > key
6     A[i + 1] = A[i]
7     i = i - 1
8   A[i + 1] = key
```

Custo	Execuções
1 → $c_1$	n
2 → $c_2$	$n - 1$
3 → 0	$n - 1$
4 → $c_4$	$n - 1$
5 → $c_5$	$\sum_{j=2}^n t_j$
6 → $c_6$	$\sum_{j=2}^n (t_j - 1)$
7 → $c_7$	$\sum_{j=2}^n (t_j - 1)$
8 → $c_8$	$n - 1$

# Análise do Insertion Sort

---

- ♦ O tempo de execução do algoritmo é a soma dos tempos de execução de cada instrução executada; uma instrução que leva  $c_i$  etapas para ser executada e executa  $n$  vezes contribuirá com  $c_i n$  para o tempo total de execução.
- ♦ Para calcular  $T(n)$ , o tempo de execução do Insertion Sort em um input de  $n$  valores, somamos os produtos das colunas de **custo** e **tempos**, obtendo a fórmula a seguir.

# Análise do Insertion Sort

---

- ◆ Fórmula:

$$T(n) = c_1 n + c_2(n - 1) + c_4(n - 1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) \\ + c_8(n - 1)$$

- ◆ Mesmo para inputs de um determinado tamanho, o tempo de execução de um algoritmo pode depender de qual input desse tamanho é fornecido.
- ◆ Por exemplo, no Insertion Sort, o melhor caso ocorre se o array já estiver ordenado.

# Análise do Insertion Sort

---

- ♦ Para cada  $j = 2, 3, \dots, n$ , descobrimos então que  $A[i] \leq \text{key}$  na linha 5 quando  $i$  tem seu valor inicial de  $j - 1$ . Assim,  $t_j = 1$  para  $j = 2, 3, \dots, n$ , e o melhor caso de tempo de execução é:

$$T(n) = c_1n + c_2(n - 1) + c_4(n - 1) + c_5(n - 1) + c_8(n - 1)$$

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

- ♦ Podemos expressar esse tempo de execução como um  $an + b$  para as constantes  $a$  e  $b$  que dependem dos custos de instrução  $c_i$ ; É, portanto, uma **função linear** de  $n$ .

# Análise do Insertion Sort

---

- ♦ Se o array estiver em ordem inversa de ordenação - ou seja, em ordem decrescente - o pior caso ocorre.
- ♦ Devemos comparar cada elemento  $A[j]$  com cada elemento em todo o subarray ordenado  $A[1...j - 1]$ , e assim  $t_j = j$  para  $j = 2, 3, \dots, n$ . Notar que:

$$\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}$$

# Análise do Insertion Sort

---

- ♦ Nós então descobrimos que, no **pior caso**, o tempo de execução do Insertion Sort é:

$$T(n) = c_1 n + c_2(n - 1) + c_4(n - 1) + c_5 \left( \frac{n(n + 1)}{2} - 1 \right)$$

$$+ c_6 \left( \frac{n(n - 1)}{2} \right) + c_7 \left( \frac{n(n - 1)}{2} \right) + c_8(n - 1)$$

$$T(n) = \left( \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left( c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n$$
$$- (c_2 + c_4 + c_5 + c_8)$$

# Análise do Insertion Sort

---

- ♦ Podemos expressar esse tempo de execução de pior caso como  $an^2 + bn + c$  para as constantes  $a$ ,  $b$  e  $c$  que, novamente, dependem dos custos de instrução  $c_i$ ; É, portanto, uma **função quadrática** de  $n$ .
- ♦ Normalmente, como no **Insertion Sort**, o **tempo de execução** de um algoritmo é fixo para um determinado **input**, embora mais adiante veremos alguns algoritmos “aleatórios” interessantes cujo comportamento pode variar mesmo para um input fixo.

# Análise de Pior Caso e Caso Médio

---

- ♦ Em nossa análise do **Insertion Sort**, examinamos o melhor caso, em que o array de input já estava ordenado, e o pior caso, em que o array de input estava ordenado reversamente.
- ♦ Entretanto, geralmente nos concentraremos em encontrar apenas o tempo de execução do **pior caso**, ou seja, o tempo de execução mais longo para qualquer input de tamanho  $n$ .
- ♦ Oferecemos **três razões** para esta orientação.

# Análise de Pior Caso e Caso Médio

---

- ♦ O **pior caso** de tempo de execução de um algoritmo nos dá um limite superior no tempo de execução para qualquer input.
- ♦ No caso de tempo de execução, o pior caso de complexidade de tempo indica o tempo de execução mais longo executado por um algoritmo, dado qualquer input de tamanho  $n$ , e assim garante que o algoritmo terminará no período de tempo indicado.
- ♦ Sabê-lo fornece uma garantia de que o algoritmo nunca vai demorar mais.

# Análise de Pior Caso e Caso Médio

---

- ♦ Para alguns algoritmos, o **pior caso** ocorre com bastante frequência.
- ♦ Por exemplo, ao pesquisar um banco de dados por uma informação específica, o pior caso dos algoritmos de pesquisa geralmente ocorre quando a informação não está presente no banco de dados.
- ♦ Em algumas aplicações, as buscas por informações ausentes podem ser frequentes.

# Análise de Pior Caso e Caso Médio

---

- O “caso médio” é quase tão ruim quanto o pior caso.
- Suponha que escolhemos aleatoriamente  $n$  números e apliquemos o [Insertion Sort](#).
- Quanto tempo leva para determinar onde em um subarray  $A[1\dots j - 1]$  para inserir o elemento  $A[j]$ ?
- Em média, metade dos elementos em  $A[1\dots j - 1]$  são menores que  $A[j]$  e metade dos elementos são maiores.
- Em média, portanto, verificamos metade do array  $A[1\dots j - 1]$  e, portanto,  $t_j$  é cerca de  $j/2$ .
- O tempo médio de execução resultante acaba sendo uma função quadrática do tamanho de input, assim como o tempo de execução do pior caso.

# Ordem de Crescimento

---

- Usamos algumas abstrações simplificadoras para facilitar nossa análise do procedimento Insertion Sort.
- Primeiro, ignoramos o custo real de cada instrução, usando as constantes  $c_i$  para representar esses custos.
- Então, observamos que mesmo essas constantes nos fornecem mais detalhes do que realmente precisamos: expressamos o tempo de execução do pior caso como  $an^2 + bn + c$  para algumas constantes  $a$ ,  $b$  e  $c$  que dependem dos custos de instrução  $c_i$ .
- Assim, ignoramos não apenas os custos reais das instruções, mas também os custos abstratos  $c_i$ .
- Faremos agora mais uma abstração simplificadora: é a **taxa de crescimento**, ou **ordem de crescimento**, do tempo de execução que realmente nos interessa.

# Ordem de Crescimento

---

- Portanto, consideramos apenas o termo principal de uma fórmula (por exemplo,  $an^2$ ), uma vez que os termos de ordem inferior são relativamente insignificantes para grandes valores de  $n$ .
- Também ignoramos o coeficiente constante dos termos principais, uma vez que os fatores constantes são menos significativos do que a taxa de crescimento na determinação da eficiência computacional para grandes inputs.
- Para o Insertion Sort, quando ignoramos os termos de ordem inferior e o coeficiente constante do termo líder, ficamos com o fator de  $n^2$  do termo líder.
- Escrevemos então que Insertion Sort tem um tempo de execução de pior caso de  $\Theta(n^2)$  (pronuncia-se “Theta de n-ao quadrado”).

# Ordem de Crescimento

---

- ♦ Normalmente consideramos um algoritmo mais eficiente do que outro se seu tempo de execução de pior caso tiver uma ordem de crescimento inferior.
- ♦ Devido a fatores constantes e termos de ordem inferior, um algoritmo cujo tempo de execução tem uma ordem de crescimento superior pode levar menos tempo para pequenos inputs do que um algoritmo cujo tempo de execução tem uma ordem de crescimento inferior.
- ♦ Mas para inputs grandes o suficiente, um algoritmo  $\Theta(n^2)$ , por exemplo, será executado mais rapidamente no pior caso do que um algoritmo  $\Theta(n^3)$ .

# Projetando Algoritmos

---

- ♦ Podemos escolher entre uma ampla variedade de técnicas de design de algoritmos.
- ♦ Para o Insertion Sort, usamos uma abordagem **incremental**: tendo ordenado o subarray  $A[1\dots j - 1]$ , inserimos o único elemento  $A[j]$  em seu lugar apropriado, resultando no subarray ordenado  $A[1\dots j]$ .
- ♦ Um **algoritmo incremental** recebe uma sequência de input e encontra uma sequência de soluções que são construídas de forma incremental enquanto se adaptam às mudanças no input.

# Dividir e Conquistar

---

- ♦ Muitos algoritmos úteis têm estrutura **recursiva**: para resolver um determinado problema, eles chamam a si mesmos recursivamente uma ou mais vezes para lidar com subproblemas intimamente relacionados.
- ♦ Esses algoritmos normalmente seguem uma abordagem de **dividir e conquistar**: eles dividem o problema em vários subproblemas que são semelhantes ao problema original, mas menores em tamanho, resolvem os subproblemas recursivamente e, em seguida, combinam essas soluções para criar uma solução para o problema original.

# Dividir e Conquistar

---

- ♦ O paradigma **dividir e conquistar** envolve três etapas em cada nível da recursão:
  - **Dividir** o problema em vários subproblemas que são instâncias menores do mesmo problema.
  - **Conquistar** os subproblemas resolvendo-os recursivamente. Se os tamanhos dos subproblemas forem pequenos o suficiente, no entanto, resolva os subproblemas de maneira direta.
  - **Combinar** as soluções para os subproblemas na solução para o problema original.

# O Algoritmo Merge Sort

---

- ♦ O algoritmo **Merge Sort** segue de perto o paradigma **dividir e conquistar**. Intuitivamente, ele funciona da seguinte maneira:
  - **Divide:** Divida a sequência de  $n$  elementos a ser ordenada em duas subsequências de  $n/2$  elementos cada.
  - **Conquistar:** Ordene as duas subsequências recursivamente usando o Merge Sort.
  - **Combinar:** Mescle as duas subsequências ordenadas para produzir a resposta ordenada final.

# O Algoritmo Merge Sort

---

- ◆ A recursão “chega ao fundo” quando a sequência a ser ordenada tem comprimento 1, caso em que não há trabalho a ser feito, visto que toda sequência de comprimento 1 já está ordenada.
- ◆ A operação principal do algoritmo Merge Sort é a mesclagem de duas sequências ordenadas na etapa de “combinação”.
- ◆ Nós unimos ao chamar o procedimento auxiliar **MERGE(A, p, q, r)**, onde A é um array e p, q e r são índices no array, tal que  $p \leq q < r$ .

# O Algoritmo Merge Sort

---

- ◆ O procedimento assume que os subarrays  $A[p...q]$  e  $A[q + 1...r]$  estão ordenados.
- ◆ Ele mescla eles para formar um único subarray ordenado que substitui o subarray atual  $A[q...r]$ .
- ◆ Nosso procedimento **MERGE** leva tempo  $\Theta(n)$ , onde  $n = r - p + 1$  é o número total de elementos sendo mesclados e funciona da seguinte maneira.
- ◆ Voltando ao nosso **tema de jogo de cartas**, suponha que temos duas pilhas de cartas viradas para cima em uma mesa.

# O Algoritmo Merge Sort

---

- Cada pilha é ordenada, com as cartas menores no topo.
- Queremos unir as duas pilhas em um único pilha de output ordenada, que deve estar voltada para baixo sobre a mesa.
- Nossa etapa básica consiste em escolher a menor das duas cartas no topo das pilhas voltadas para cima, removendo-a de sua pilha (o que expõe uma nova carta do topo) e colocando esta carta voltada para baixo na pilha de output.
- Repetimos essa etapa até que uma pilha de input esteja vazia, momento em que apenas pegamos a pilha de input restante e colocamos com a face para baixo na pilha de output.
- Computacionalmente, cada etapa básica leva um tempo constante, uma vez que estamos comparando apenas as duas cartas do topo. Como executamos no máximo  $n$  etapas básicas, a união leva tempo  $\Theta(n)$ .

# O Algoritmo Merge Sort

---

- O pseudocódigo a seguir implementa a ideia anterior, mas com uma variação adicional que evita ter que verificar se alguma pilha está vazia em cada etapa básica.
- Colocamos no fundo de cada pilha uma carta **sentinela**, que contém um valor especial que usamos para simplificar nosso código.
- Aqui, usamos  $\infty$  como valor de sentinela, de modo que sempre que uma carta com  $\infty$  for exposta, ela não possa ser a carta menor, a menos que ambas as pilhas tenham suas cartas sentinela expostas.
- Mas quando isso acontece, todas as cartas não-sentinela já foram colocadas na pilha de output.
- Como sabemos de antemão que exatamente  $r - p + 1$  cartas serão colocadas na pilha de output, podemos parar depois de realizar todas as etapas básicas.

# MERGE(A, p, q, r)

---

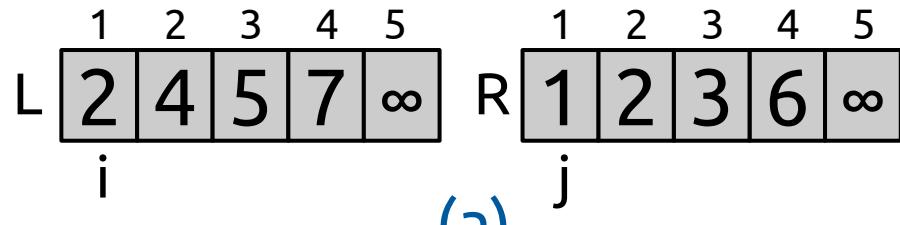
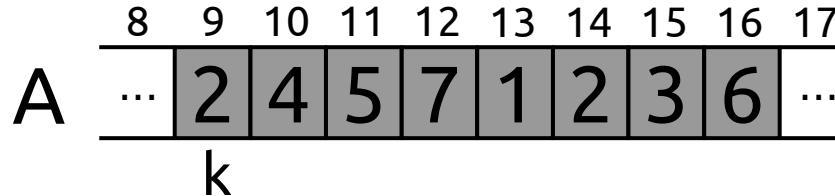
```
01 n1 = q - p + 1
02 n2 = r - q
03 let L[1...n1 + 1] and R[1...n2 + 1] ser novos arrays
04 for i = 1 to n1
05   L[i] = A[p + i - 1]
06 for j = 1 to n2
07   R[j] = A[q + j]
08 L[n1 + 1] = ∞
09 R[n2 + 1] = ∞
10 i = 1
11 j = 1
12 for k = p to r
13   if L[i] ≤ R[j]
14     A[k] = L[i]
15     i = i + 1
16   else A[k] = R[j]
17     j = j + 1
```

# O Algoritmo Merge Sort

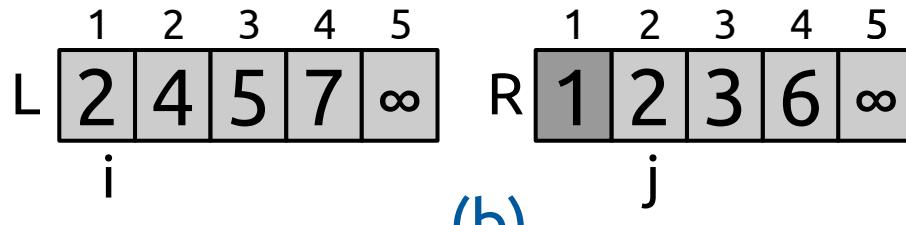
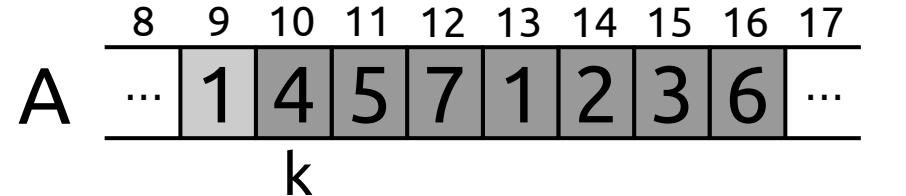
---

- Em detalhes, o procedimento **MERGE** funciona da seguinte maneira:
  - **Linha 1** computa o comprimento  $n_1$  do subarray  $A[p \dots q]$ , e a **linha 2** computa o comprimento  $n_2$  do subarray  $A[q + 1 \dots r]$ .
  - Criamos os arrays  $L$  e  $R$  (“left” e “right”), de comprimentos  $n_1 + 1$  e  $n_2 + 1$ , respectivamente, na **linha 3**; a posição extra em cada array manterá o sentinela.
  - O loop **for** das **linhas 4–5** copia o subarray  $A[p \dots q]$  em  $L[1 \dots n_1]$ , e o loop **for** das **linhas 6–7** copia o subarray  $A[q + 1 \dots r]$  em  $R[1 \dots n_2]$ .
  - As **linhas 8–9** colocam os sentinelas nas extremidades dos arrays  $L$  e  $R$ .
  - As **linhas 10–17** ilustradas na figura a seguir, executam as etapas básicas  $r - p + 1$ , mantendo uma **invariante de loop**.

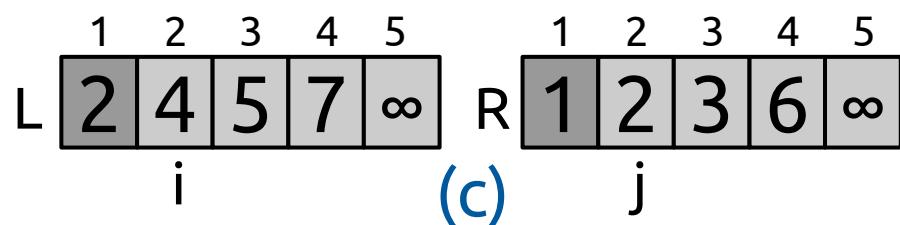
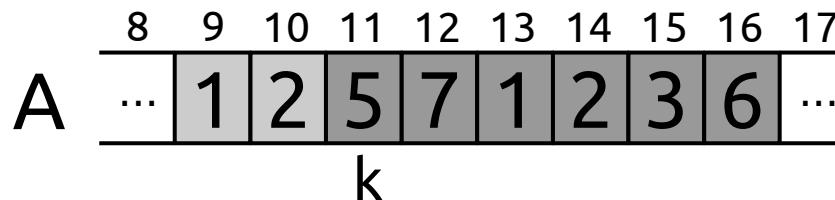
# O Algoritmo Merge Sort



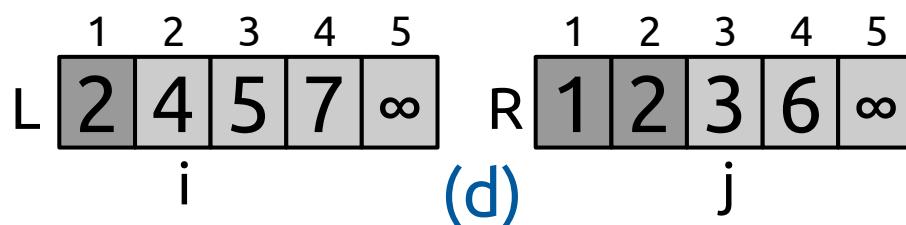
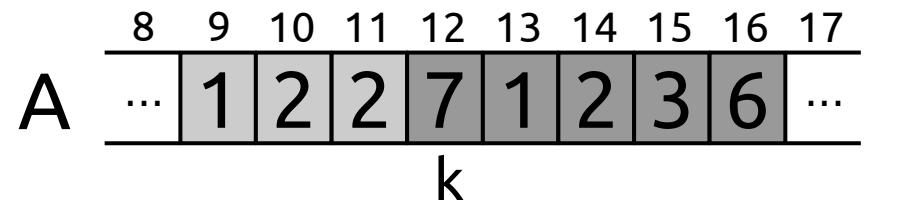
(a)



(b)

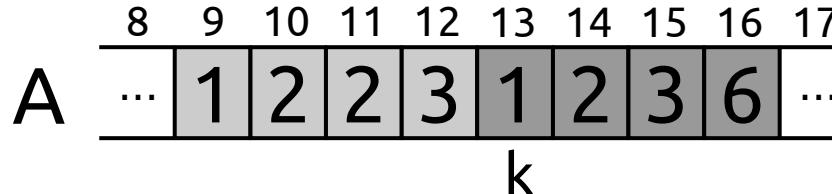


(c)

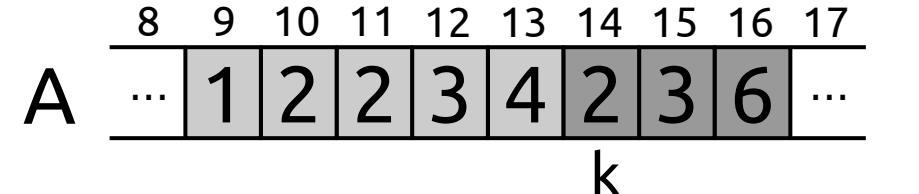


(d)

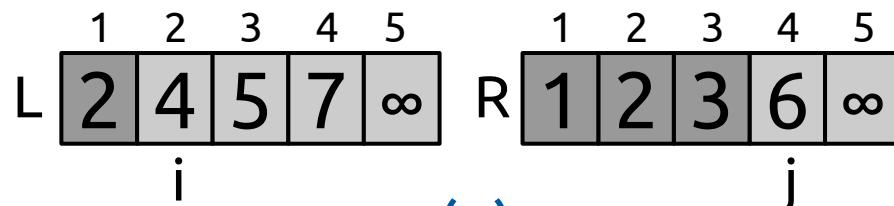
# O Algoritmo Merge Sort



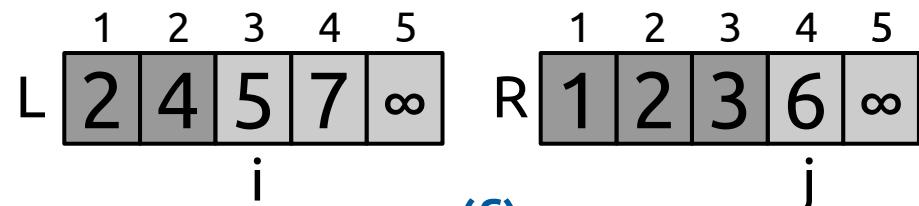
(e)



(f)



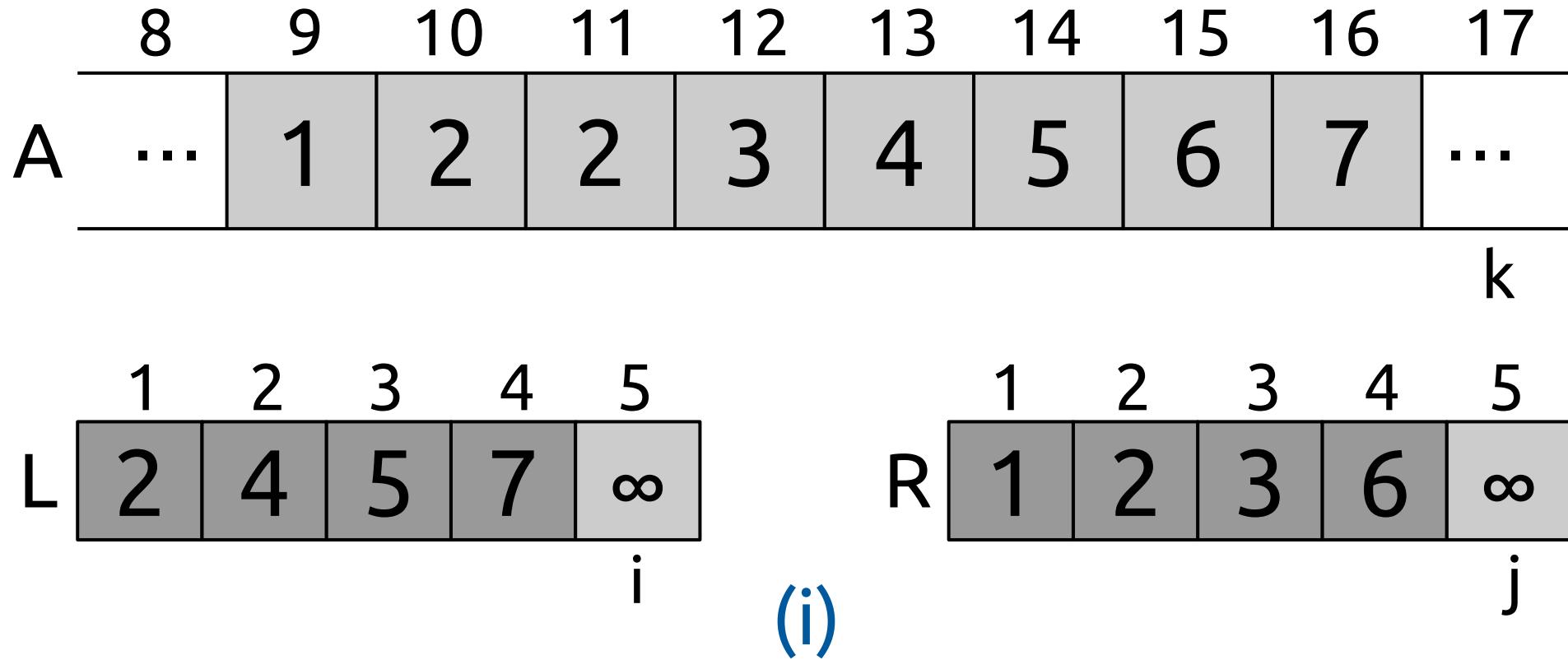
(g)



(h)

# O Algoritmo Merge Sort

---



# O Algoritmo Merge Sort

---

- Figura: A operação das **linhas 10-17** na chamada **MERGE(A, 9, 12, 16)**, quando o subarray **A[9...16]** contém a sequência {**2, 4, 5, 7, 1, 2, 3, 6**}. Depois de copiar e inserir sentinelas, o array **L** contém {**2, 4, 5, 7,  $\infty$** }, e o array **R** contém {**1, 2, 3, 6,  $\infty$** }. As posições levemente sombreadas em **A** contêm seus valores finais, e as posições levemente sombreadas em **L** e **R** contêm valores que ainda não foram copiados de volta para **A**. Juntas, as posições levemente sombreadas sempre compreendem os valores originalmente em **A[9...16]**, junto com as duas sentinelas.

# O Algoritmo Merge Sort

---

- ♦ Figura: As posições fortemente sombreadas em A contêm valores que serão copiados, e as posições fortemente sombreadas em L e R contêm valores que já foram copiados de volta para A. (a) - (h) Os arrays A, L e R e seus respectivos índices k, i e j antes de cada iteração do loop das linhas 12–17. (i) Os arrays e índices na término. Neste ponto, o subarray em A[9...16] é ordenado, e as duas sentinelas em L e R são os únicos dois elementos nesses arrays que não foram copiados em A.

# O Algoritmo Merge Sort

---

- Devemos lembrar que as **linhas 10-17** executam as etapas básicas  $r - p + 1$ , mantendo a seguinte **invariante de loop**:
  - No início de cada iteração do loop **for** das **linhas 12–17**, o subarray  $A[p \dots k - 1]$  contém os  $k - p$  menores elementos de  $L[1 \dots n_1 + 1]$  e  $R[1 \dots n_2 + 1]$ , ordenados.
  - Além disso,  $L[i]$  e  $R[j]$  são os menores elementos de seus arrays que não foram copiados de volta para  $A$ .
- Devemos mostrar que esse invariante de loop é válido antes da primeira iteração do loop **for** das **linhas 12–17**, que cada iteração do loop mantém o invariante e que o invariante fornece uma propriedade útil para mostrar a correção quando o loop termina.

# O Algoritmo Merge Sort

---

- ♦ **Inicialização:** Antes da primeira iteração do loop, temos  $k = p$ , de forma que o subarray  $A[p \dots k - 1]$  está vazio.
- ♦ Este subarray vazio contém os menores elementos  $k - p = 0$  de  $L$  e  $R$ , e como  $i = j = 1$ , ambos  $L[i]$  e  $R[j]$  são os menores elementos de seus arrays que não foram copiados de volta para  $A$ .

# O Algoritmo Merge Sort

---

- ♦ **Manutenção:** Para ver que cada iteração mantém o laço invariante, vamos primeiro supor que  $L[i] \leq R[j]$ . Então  $L[i]$  é o menor elemento ainda não copiado de volta para  $A$ .
- ♦ Como  $A[p\dots k - 1]$  contém os  $k - p$  menores elementos, após a **linha 14** copiar  $L[i]$  em  $A[k]$ , o subarray  $A[p\dots k]$  conterá os  $k - p + 1$  menores elementos.
- ♦ Incrementar  $k$  (na atualização do laço for) e  $i$  (na **linha 15**) restabelece a invariante do laço para a próxima iteração. Se, em vez disso,  $L[i] > R[j]$ , então as **linhas 16–17** executam a ação apropriada para manter o laço invariante.

# O Algoritmo Merge Sort

---

- ♦ **Término:** No término,  $k = r + 1$ .
- ♦ Pelo invariante do loop, o subarray  $A[p \dots k - 1]$ , que é  $A[p \dots r]$ , contém os  $k - p = r - p + 1$  menores elementos de  $L[1 \dots n_1 + 1]$  e  $R[1 \dots n_2 + 1]$ , ordenados.
- ♦ Os arrays  $L$  e  $R$  juntos contêm  $n_1 + n_2 + 2 = r - p + 3$  elementos.
- ♦ Todos, exceto os dois maiores, foram copiados de volta para  $A$ , e esses dois maiores elementos são as sentinelas.

# O Algoritmo Merge Sort

---

- ♦ Para ver que o procedimento **MERGE** é executado em tempo  $\Theta(n)$ , onde  $n = r - p + 1$ , observe que cada uma das linhas 1–3 e 8–11 leva um tempo constante, os loops **for** das linhas 4–7 levam tempo  $\Theta(n_1 + n_2) = \Theta(n)$ , e há  $n$  iterações do loop **for** das linhas 12–17, cada uma das quais leva um tempo constante.
- ♦ Agora podemos usar o procedimento **MERGE** como uma sub-rotina no algoritmo Merge Sort.

# O Algoritmo Merge Sort

---

- ◆ O procedimento **MERGE-SORT(A, p, r)** ordena os elementos no subarray **A[p...r]**.
- ◆ Se  $p \geq r$ , o subarray tem no máximo um elemento e, portanto, já está ordenado.
- ◆ Caso contrário, a etapa de divisão simplesmente calcula um índice **q** que divide **A[p...r]** em dois subarrays: **A[p...q]**, contendo **[n/2]** elementos, e **A[q + 1...r]**, contendo **[n/2]** elementos.

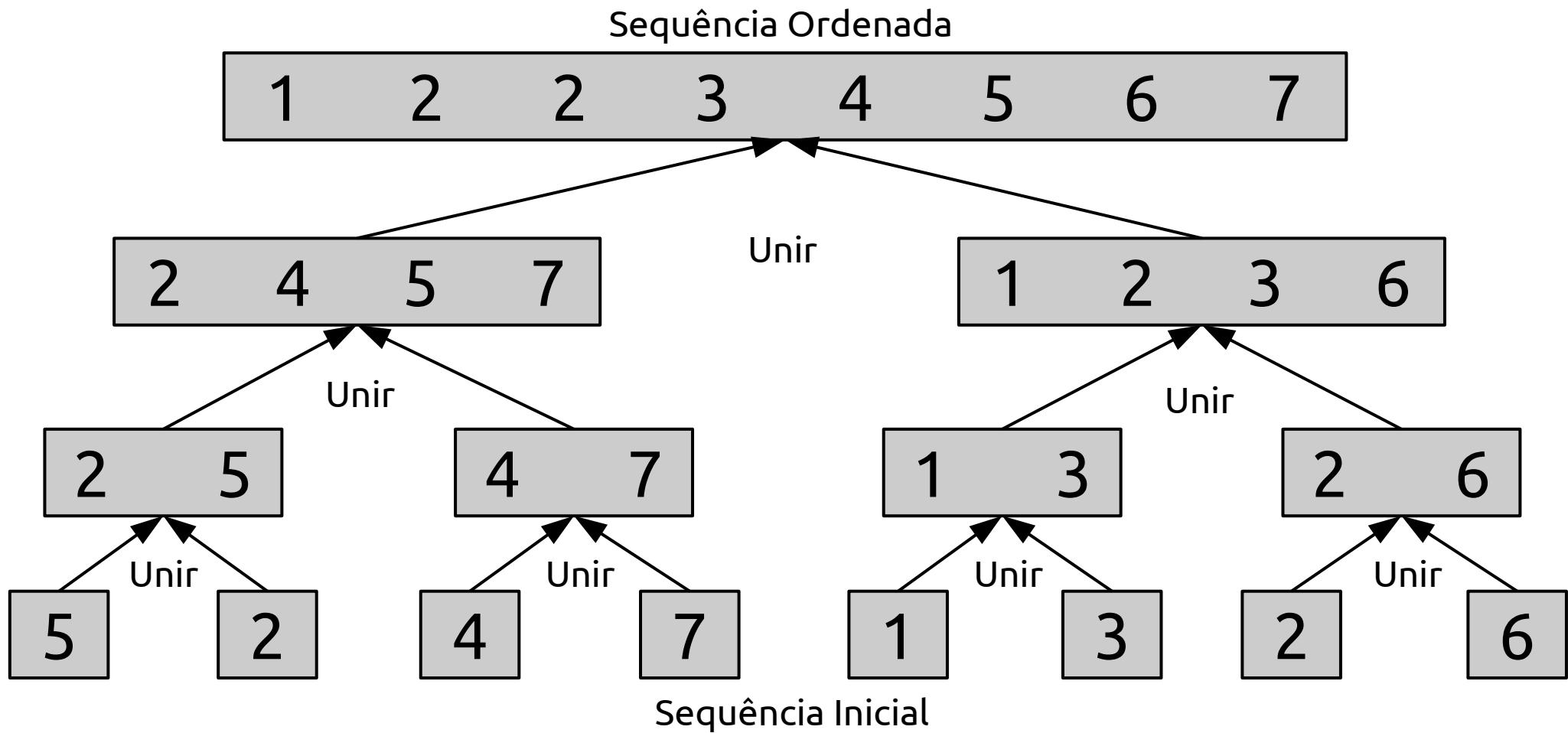
```
MERGE-SORT(A, p, r)
1 if p < r
2   q = [(p + r)/2]
3   MERGE-SORT(A, p, q)
4   MERGE-SORT(A, q+1, r)
5   MERGE(A, p, q, r)
```

# O Algoritmo Merge Sort

---

- ♦ Para ordenar a sequência inteira  $A = \{A[1], A[2], \dots, A[n]\}$  fazemos a chamada inicial  $\text{MERGE-SORT}(A, 1, A.length)$ , onde mais uma vez  $A.length = n$ .
- ♦ A figura a seguir ilustra a operação do procedimento de baixo para cima (bottom-up) quando  $n$  é uma potência de 2.
- ♦ O algoritmo consiste em unir pares de sequências de 1 item para formar sequências ordenadas de comprimento 2, unir pares de sequências de comprimento 2 para formar sequências ordenadas de comprimento 4 e assim por diante, até que duas sequências de comprimento  $n/2$  sejam unidas para formar a sequência final ordenada de comprimento  $n$ .

# O Algoritmo Merge Sort



# O Algoritmo Merge Sort

- ♦ A seguir temos o procedimento **MERGE** em Python:

```
def merge(A,p,q,r):  
    n1 = q - p + 1  
    n2 = r - q  
    L = [0] * (n1)  
    R = [0] * (n2)
```

```
    for i in range(0, n1):  
        L[i] = A[p + i]
```

```
    for j in range(0, n2):  
        R[j] = A[q + 1 + j]
```

```
i = 0  
j = 0  
k = p
```

```
while i < n1 and j < n2:
```

```
    if L[i] <= R[j]:
```

```
        A[k] = L[i]
```

```
        i += 1
```

```
else:
```

```
    A[k] = R[j]
```

```
    j += 1
```

```
    k += 1
```

```
while i < n1:
```

```
    A[k] = L[i]
```

```
    i += 1
```

```
    k += 1
```

```
while j < n2:
```

```
    A[k] = R[j]
```

```
    j += 1
```

```
    k += 1
```

# O Algoritmo Merge Sort

- ◆ A seguir temos o procedimento **MERGE-SORT** em **Python**:

```
def merge_sort(A,p,r):
    if p < r:
        q = (p+(r-1))//2
        merge_sort(A, p, q)
        merge_sort(A, q+1, r)
        merge(A, p, q, r)

array = [5, 2, 4, 7, 1, 3, 2, 6]
merge_sort(array,0,len(array)-1)
print(array) # [1, 2, 2, 3, 4, 5, 6, 7]
```



# Analisando Algoritmos de Divisão e Conquista

---

- ♦ Quando um algoritmo contém uma chamada recursiva para si mesmo, podemos frequentemente descrever seu tempo de execução por uma **equação de recorrência** ou **recorrência**, que descreve o tempo de execução geral em um problema de tamanho  $n$  em termos do tempo de execução em inputs menores.
- ♦ Podemos então usar ferramentas matemáticas para resolver a recorrência e fornecer limites para o desempenho do algoritmo.

# Analisando Algoritmos de Divisão e Conquista

---

- ♦ Uma recorrência para o tempo de execução de um algoritmo de divisão e conquista decorre das três etapas do paradigma básico.
- ♦ Como antes, deixamos  $T(n)$  ser o tempo de execução de um problema de tamanho  $n$ .
- ♦ Se o tamanho do problema for pequeno o suficiente, digamos  $n \leq c$  para alguma constante  $c$ , a solução direta leva um tempo constante, que escrevemos como  $\Theta(1)$ .

# Analisando Algoritmos de Divisão e Conquista

---

- ♦ Suponha que nossa divisão do problema produza  $a$  subproblemas, cada um dos quais o tamanho é  $1/b$  do original.
- ♦ Para Merge Sort,  $a$  e  $b$  são 2, porém existem algoritmos de divisão e conquista em que  $a \neq b$ .
- ♦ Leva tempo  $T(n/b)$  para resolver um subproblema de tamanho  $n/b$ , portanto, leva tempo  $aT(n/b)$  para resolver um número  $a$  deles.

# Analizando Algoritmos de Divisão e Conquista

---

- ♦ Se tomarmos  $D(n)$  tempo para dividir o problema em subproblemas e  $C(n)$  tempo para combinar as soluções para os subproblemas na solução para o problema original, obtemos a recorrência:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{caso contrário} \end{cases}$$

# Análise do Merge Sort

---

- Embora o pseudocódigo para **MERGE-SORT** funcione corretamente quando o número de elementos não é par, nossa análise baseada em recorrência é simplificada se assumirmos que o tamanho original do problema é uma potência de  $2$ .
- Cada etapa de divisão então produz duas subsequências de tamanho exatamente  $n/2$ .
- Raciocinamos da seguinte forma para configurar a recorrência para  $T(n)$ , o pior caso de tempo de execução do Merge Sort em  $n$  números.
- O Merge Sort em apenas um elemento leva um tempo constante. Quando temos  $n > 1$  elementos, dividimos o tempo de execução da seguinte maneira.

# Análise do Merge Sort

---

- ♦ **Dividir:** A etapa de divisão apenas calcula o meio do subarray, o que leva um tempo constante. Assim,  $D(n) = \Theta(1)$ .
- ♦ **Conquistar:** Resolvemos recursivamente dois subproblemas, cada um de tamanho  $n/2$ , que contribui com  $2T(n/2)$  para o tempo de execução.
- ♦ **Combinar:** Já observamos que o procedimento **MERGE** em um subarray de  $n$  elementos leva tempo  $\Theta(n)$  e, portanto,  $C(n) = \Theta(n)$ .

# Análise do Merge Sort

---

- ◆ Quando adicionamos as funções  $D(n)$  e  $C(n)$  para a análise do Merge Sort, estamos adicionando uma função que é  $\Theta(n)$  e uma função que é  $\Theta(1)$ .
- ◆ Essa soma é uma função linear de  $n$ , ou seja,  $\Theta(n)$ .
- ◆ Adicionando ele ao termo  $2T(n/2)$  da etapa "conquistar" dá a recorrência para o pior caso de tempo de execução  $T(n)$  do Merge Sort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

# Análise do Merge Sort

---

- É possível usar o "master theorem", que pode ser utilizado para mostrar que  $T(n)$  é  $\Theta(n \log n)$ , onde  $\log n$  representa  $\log_2 n$ . Como a função de logaritmo cresce mais lentamente do que qualquer função linear, para inputs grandes o suficiente, o Merge Sort, com seu tempo de execução  $\Theta(n \log n)$ , supera o Insertion Sort, cujo tempo de execução é  $\Theta(n^2)$ , no pior caso.
- Não precisamos do master theorem para entender intuitivamente por que a solução para a recorrência anterior é  $T(n) = \Theta(n \log n)$ . Vamos reescrever a recorrência como:

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/b) + cn & \text{if } n > 1 \end{cases}$$

# Análise do Merge Sort

---

- Onde a constante  $c$  representa o tempo necessário para resolver problemas de tamanho 1, bem como o tempo por elemento do array das etapas de divisão e combinação.
- A figura que apresentaremos mostra como podemos resolver a recorrência.
- Por conveniência, assumimos que  $n$  é uma potência exata de 2. A parte (a) da figura mostra  $T(n)$ , que expandimos na parte (b) em uma árvore equivalente que representa a recorrência. O termo  $cn$  é a raiz (o custo incorrido no nível superior de recursão) e as duas subárvore da raiz são as duas recorrências menores  $T(n/2)$ . A parte (c) mostra que esse processo foi levado um passo adiante, expandindo  $T(n/2)$ . O custo incorrido em cada um dos dois subnós no segundo nível de recursão é  $cn/2$ . Continuamos expandindo cada nó na árvore dividindo-o em suas partes constituintes, conforme determinado pela recorrência, até que os tamanhos do problema caiam para 1, cada um com um custo de  $c$ . A parte (d) mostra a árvore de recursão resultante.

# Análise do Merge Sort

---

- ♦ Em seguida, adicionamos os custos em cada nível da árvore.
- ♦ O nível superior tem custo total  $cn$ , o próximo nível inferior tem custo total  $c(n/2) + c(n/2) = cn$ , o nível seguinte tem custo total  $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$  e assim por diante.
- ♦ Em geral, o nível  $i$  abaixo do topo tem  $2^i$  nós, cada um contribuindo com um custo de  $c(n/2^i)$ , de modo que o nível  $i$ -ésimo abaixo do topo tem custo total  $2^i c(n/2^i)$ .
- ♦ O nível inferior tem  $n$  nós, cada um contribuindo com um custo de  $c$ , para um custo total de  $cn$ .

# Análise do Merge Sort

---

- O número total de níveis da árvore de recursão na figura que vamos apresentar é  $\log n + 1$ , onde  $n$  é o número de folhas, correspondendo ao tamanho do input.
- Um argumento indutivo informal justifica essa afirmação. O caso base ocorre quando  $n = 1$ , caso em que a árvore tem apenas um nível.
- Como  $\log 1 = 0$ , temos que  $\log n + 1$  nos dá o número correto de níveis.
- Agora assuma como hipótese indutiva que o número de níveis de uma árvore de recursão com  $2^i$  folhas é  $\log 2^i + 1 = i + 1$  (uma vez que para qualquer valor de  $i$ , temos aquele  $\log 2^i = i$ ). Como estamos assumindo que o tamanho do input é uma potência de 2, o próximo tamanho de input a ser considerado é  $2^{i+1}$ .

# Análise do Merge Sort

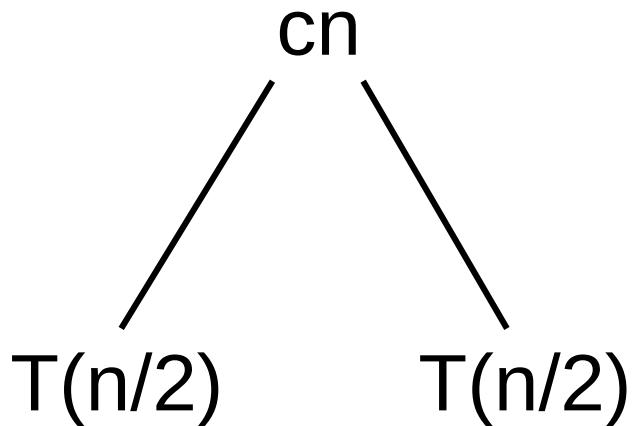
---

- Uma árvore com  $n = 2^{i+1}$  folhas tem um nível a mais do que uma árvore com  $2^i$  folhas e, portanto, o número total de níveis é  $(i + 1) + 1 = \log 2^{i+1} + 1$ .
- Para calcular o custo total representado pela recorrência, simplesmente somamos os custos de todos os níveis.
- A **árvore de recursão** tem  $\log n + 1$  níveis, cada um custando  $cn$ , para um custo total de  $cn(\log n + 1) = cn \log n + cn$ .
- Ignorar o termo de ordem inferior e a constante  $c$  fornece o resultado desejado de  $\Theta(n \log n)$ .

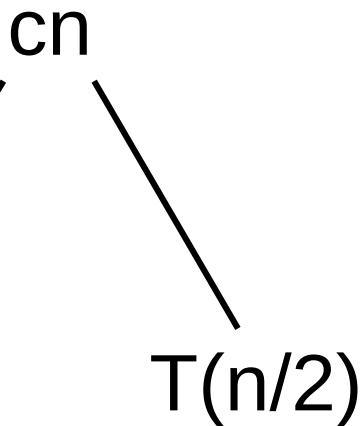
# Árvore de Recursão

---

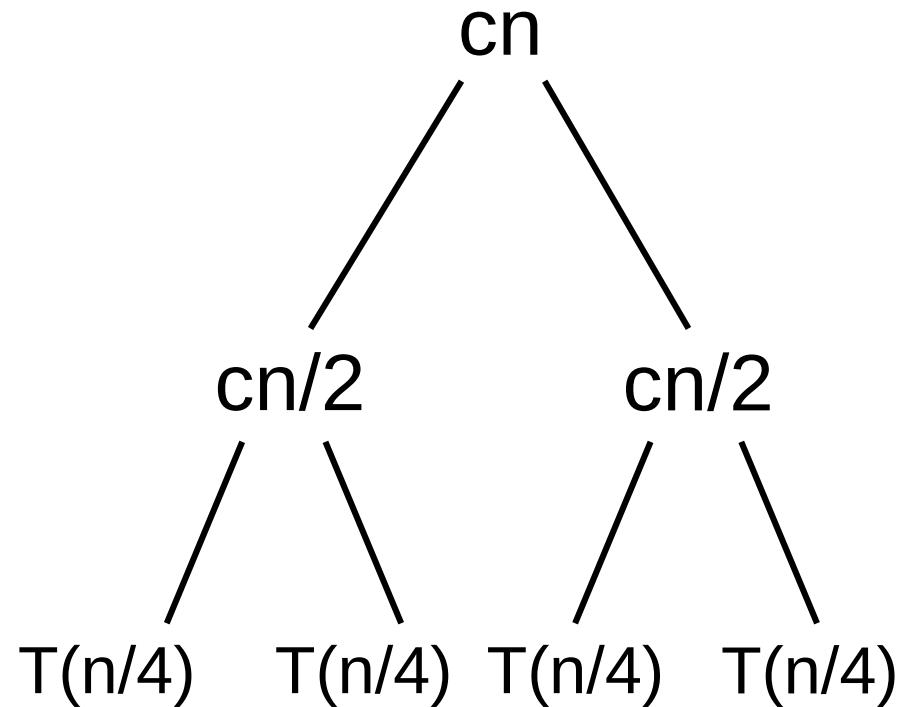
$T(n)$



(a)

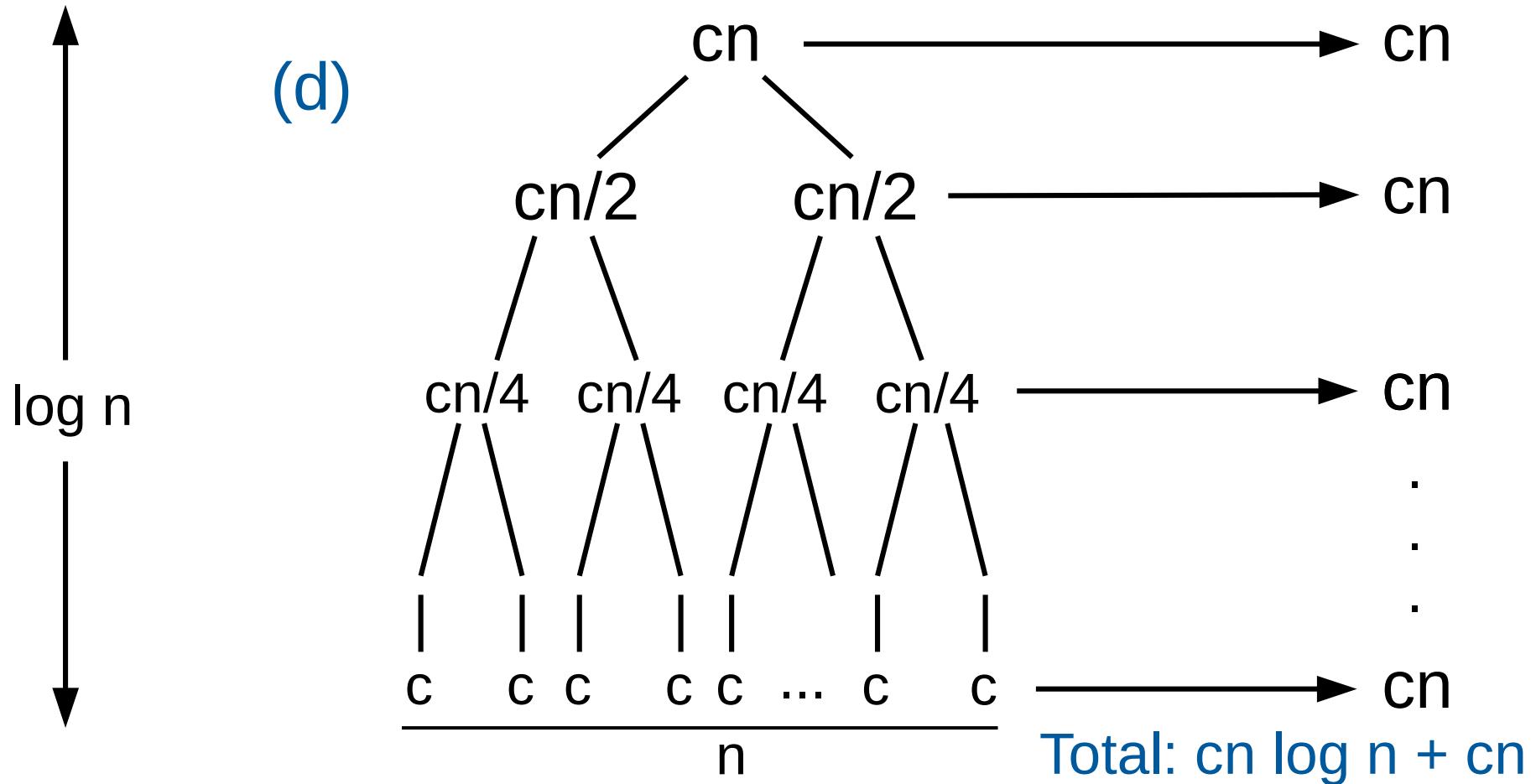


(b)



(c)

# Árvore de Recursão



# Árvore de Recursão

---

- ◆ A figura mostra como construir uma árvore de recursão para a recorrência  $T(n) = 2T(n/2) + cn$ .
- ◆ A parte (a) mostra  $T(n)$ , que se expande progressivamente em (b) - (d) para formar a árvore de recursão.
- ◆ A árvore totalmente expandida na parte (d) tem  $\log n + 1$  níveis (ou seja, tem altura  $\log n$ , conforme indicado) e cada nível contribui com um custo total de  $cn$ .
- ◆ O custo total, portanto, é  $cn \log n + cn$ , que é  $\Theta(n \log n)$ .

# Crescimento de Funções

---

- A **ordem de crescimento** do tempo de execução de um algoritmo fornece uma caracterização simples da eficiência do algoritmo e também nos permite comparar o desempenho relativo de algoritmos alternativos.
- Uma vez que o tamanho de input  $n$  torna-se grande o suficiente, o Merge Sort, com seu tempo de execução de pior caso  $\Theta(n \log n)$ , supera o Insertion Sort, cujo tempo de execução no pior caso é  $\Theta(n^2)$ .
- Embora possamos às vezes determinar o tempo exato de execução de um algoritmo, como fizemos para o Insertion Sort anteriormente, a precisão extra geralmente não vale o esforço de computá-la.

# Crescimento de Funções

---

- Para inputs grandes o suficiente, as constantes multiplicativas e os termos de ordem inferior de um tempo de execução exato são dominados pelos efeitos do próprio tamanho do input.
- Quando olhamos para tamanhos de input grandes o suficiente para tornar relevante apenas a ordem de crescimento do tempo de execução, estamos estudando a eficiência **assintótica** dos algoritmos.
- Ou seja, estamos preocupados em como o tempo de execução de um algoritmo aumenta com o tamanho do input no limite, à medida que o tamanho do input aumenta sem limite.
- Normalmente, um algoritmo assintoticamente mais eficiente será a melhor escolha para todos os inputs, exceto inputs muito pequenos.

# Notação Assintótica

---

- As notações que usamos para descrever o tempo de execução assintótico de um algoritmo são definidas em termos de funções cujos domínios são o conjunto de números naturais  $N = \{0, 1, 2, \dots\}$ .
- Essas notações são convenientes para descrever a função de tempo de execução de pior caso  $T(n)$ , que geralmente é definida apenas em tamanhos de inputs inteiros.
- Às vezes achamos conveniente, entretanto, abusar da notação assintótica de várias maneiras.
- Por exemplo, podemos estender a notação ao domínio dos números reais ou, alternativamente, restringí-lo a um subconjunto dos números naturais.

# Notação Assintótica, Funções e Tempos de Execução

---

- Usaremos notação assintótica principalmente para descrever os tempos de execução dos algoritmos, como quando escrevemos que o pior caso de execução do Insertion Sort é  $\Theta(n^2)$ .
- A notação assintótica realmente se aplica às funções, entretanto. Lembre-se de que caracterizamos o tempo de execução de pior caso do Insertion Sort como  $an^2 + bn + c$ , para algumas constantes  $a$ ,  $b$  e  $c$ .
- Ao escrever que o tempo de execução do Insertion Sort é  $\Theta(n^2)$ , abstraímos alguns detalhes desta função.
- Como a notação assintótica se aplica a funções, o que estávamos escrevendo como  $\Theta(n^2)$  era a função  $an^2 + bn + c$ , que, nesse caso, caracterizou o pior caso de tempo de execução do Insertion Sort.

# Notação Assintótica, Funções e Tempos de Execução

---

- É importante lembrar que a **notação assintótica** pode ser aplicada a funções que caracterizam algum outro aspecto dos algoritmos (a **quantidade de espaço** que eles usam, por exemplo), ou mesmo a funções que não têm absolutamente nada a ver com algoritmos.
- Mesmo quando usamos a notação assintótica para aplicar ao tempo de execução de um algoritmo, precisamos entender a que tempo de execução estamos falando.
- Freqüentemente, entretanto, desejamos caracterizar o tempo de execução, não importa qual seja o input. Em outras palavras, frequentemente desejamos fazer uma declaração geral que cubra todos os inputs, não apenas o pior caso.

# Notação $\Theta$

---

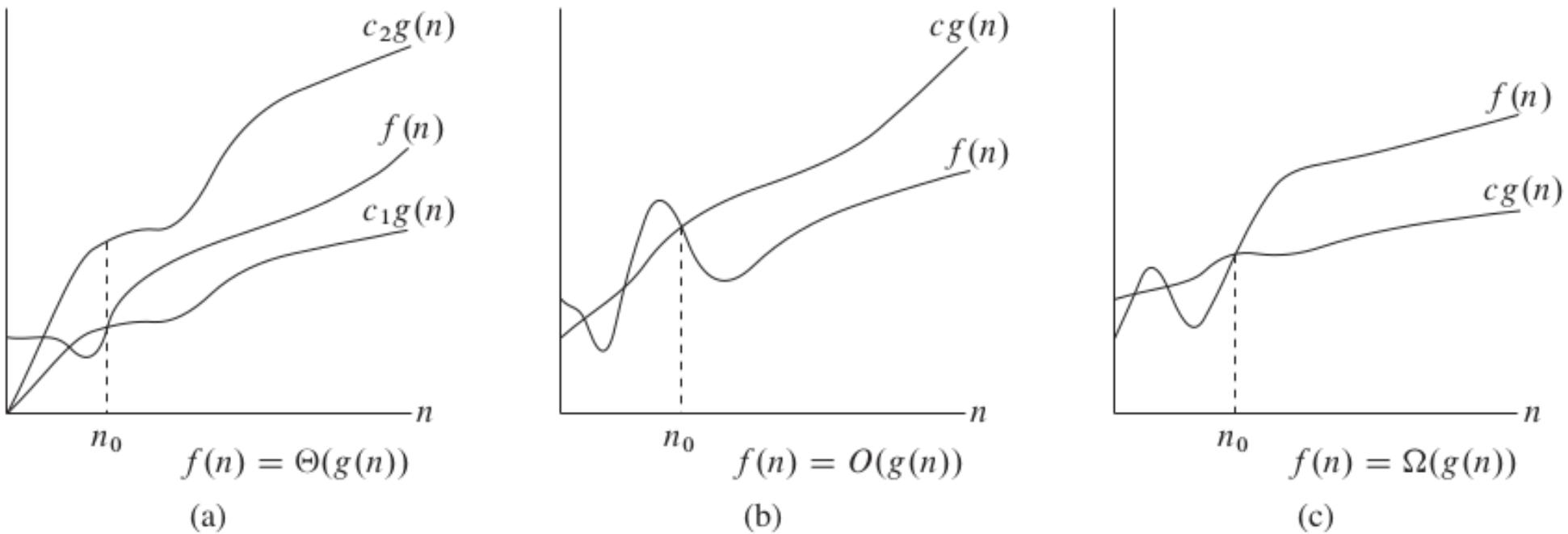
- Anteriormente, descobrimos que o pior caso de tempo de execução do Insertion Sort é  $T(n) = \Theta(n^2)$ . Vamos definir o que essa notação significa. Para uma determinada função  $g(n)$ , denotamos por  $\Theta(g(n))$  o conjunto de funções:
- $\Theta(g(n)) = \{ f(n) : \text{existem constantes positivas } c_1, c_2 \text{ e } n_0 \text{ tal que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ para todos } n \geq n_0 \}$ .
- Uma função  $f(n)$  pertence ao conjunto  $\Theta(g(n))$  se existirem constantes positivas  $c_1$  e  $c_2$  de modo que ela possa ser “comprimida” entre  $c_1 g(n)$  e  $c_2 g(n)$ , para  $n$  suficientemente grande. Como  $\Theta(g(n))$  é um conjunto, poderíamos escrever “ $f(n) \in \Theta(g(n))$ ” para indicar que  $f(n)$  é membro de  $\Theta(g(n))$ . Em vez disso, geralmente escreveremos “ $f(n) = \Theta(g(n))$ ” para expressar a mesma noção.

# Notação Assintótica

---

- A figura que apresentaremos mostra gráficos das notações  $\Theta$ ,  $O$ ,  $\Omega$ .
- Em cada parte, o valor de  $n_0$  mostrado é o valor mínimo possível; qualquer valor maior também funcionaria.
- (a) A notação  $\Theta$  limita uma função a fatores constantes. Escrevemos  $f(n) = \Theta(g(n))$  se existem constantes positivas  $n_0$ ,  $c_1$  e  $c_2$  tais que em e à direita de  $n_0$ , o valor de  $f(n)$  sempre fica entre  $c_1g(n)$  e  $c_2g(n)$  inclusive.
- (b) A notação  $O$  fornece um limite superior para uma função dentro de um fator constante. Escrevemos  $f(n) = O(g(n))$  se houver constantes positivas  $n_0$  e  $c$  tais que em e à direita de  $n_0$ , o valor de  $f(n)$  sempre fica em ou abaixo de  $cg(n)$ .
- (c) A notação  $\Omega$  dá um limite inferior para uma função dentro de um fator constante. Escrevemos  $f(n) = \Omega(g(n))$  se houver constantes positivas  $n_0$  e  $c$  tais que em e à direita de  $n_0$ , o valor de  $f(n)$  sempre está em ou acima de  $cg(n)$ .

# Notação Assintótica



# Notação Assintótica

---

- ◆ A figura (a) fornece uma imagem intuitiva das funções  $f(n)$  e  $g(n)$ , onde  $f(n) = \Theta(g(n))$ .
- ◆ Para todos os valores de  $n$  em e à direita de  $n_0$ , o valor de  $f(n)$  está em ou acima de  $c_1 g(n)$  e em ou abaixo de  $c_2 g(n)$ .
- ◆ Em outras palavras, para todo  $n \geq n_0$ , a função  $f(n)$  é igual a  $g(n)$  dentro de um fator constante. Dizemos que  $g(n)$  é um **limite assintoticamente restrito** para  $f(n)$ .

# Notação Assintótica

---

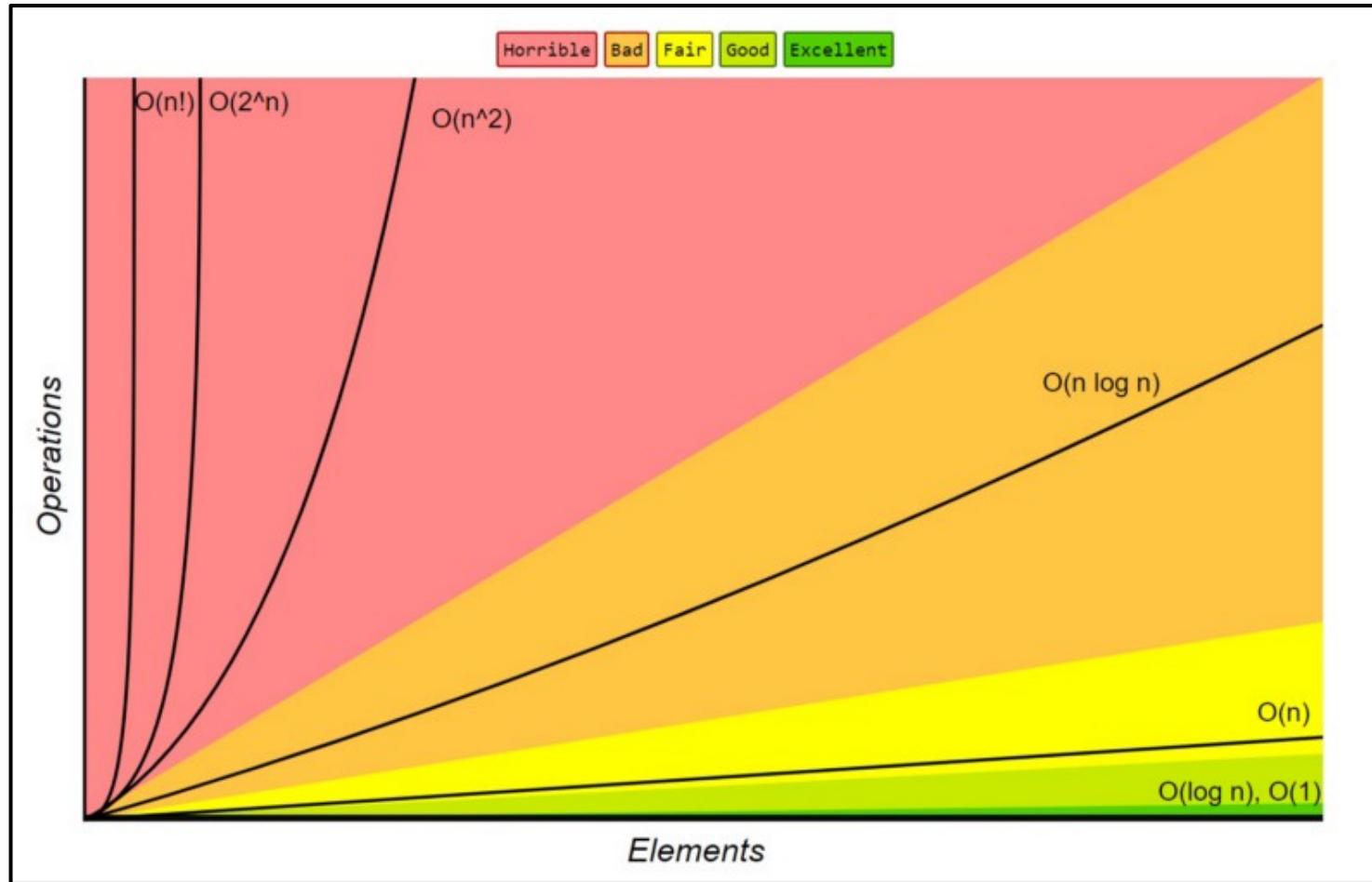
- A definição de  $\Theta(g(n))$  requer que todo membro  $f(n) \in \Theta(g(n))$  seja assintoticamente não negativo, isto é, que  $f(n)$  seja não negativo sempre que  $n$  for suficientemente grande.
- Uma função assintoticamente positiva é aquela que é positiva para todos os  $n$  suficientemente grandes.
- Consequentemente, a própria função  $g(n)$  deve ser assintoticamente não negativa, ou então o conjunto  $\Theta(g(n))$  está vazio.
- Devemos, portanto, assumir que toda função usada na notação  $\Theta$  é assintoticamente não negativa.

# Ordens Comuns de Funções

---

- ♦ Apresentaremos uma lista de **classes de funções** que são comumente encontradas ao analisarmos o tempo de execução de um algoritmo.
- ♦ Em cada caso,  $c$  é uma constante positiva e  $n$  aumenta sem limite.
- ♦ As funções de crescimento mais lento geralmente são listadas primeiro.

# Ordens Comuns de Funções



# Ordens Comuns de Funções

---

- ◆ **Notação:**  $O(1)$
- ◆ **Nome:** Constante
- ◆ **Exemplo:** Determinar se um número binário é par ou ímpar, usar uma tabela de pesquisa de tamanho constante

# Ordens Comuns de Funções

---

- ◆ **Notação:**  $O(\log \log n)$
- ◆ **Nome:** Logarítmico duplo
- ◆ **Exemplo:** Número de comparações gastas para encontrar um item usando a pesquisa de interpolação em um array ordenado de valores uniformemente distribuídos

# Ordens Comuns de Funções

---

- ◆ **Notação:**  $O(\log n)$
- ◆ **Nome:** Logarítmico
- ◆ **Exemplo:** Encontrar um item em um array ordenado com uma pesquisa binária ou uma árvore de pesquisa balanceada, bem como todas as operações em um Heap Binomial

# Ordens Comuns de Funções

---

- ◆ **Notação:**  $O(n)$
- ◆ **Nome:** Linear
- ◆ **Exemplo:** Encontrar um item em uma lista não ordenada ou em um array não ordenado; adicionando dois inteiros de  $n$  bits por ripple carry

# Ordens Comuns de Funções

---

- ◆ **Notação:**  $O(n \log n)$
- ◆ **Nome:** Loglinear
- ◆ **Exemplo:** Executando fast Fourier transform; algoritmos de ordenação como: heapsort e merge sort

# Ordens Comuns de Funções

---

- ◆ **Notação:**  $O(n^2)$
- ◆ **Nome:** Quadrático
- ◆ **Exemplo:** Algoritmos de ordenação simples como bubble sort, selection sort e insertion sort, (pior caso) ligado a alguns algoritmos de classificação geralmente mais rápidos, como quicksort, shellsort e tree sort

# Ordens Comuns de Funções

---

- ◆ **Notação:**  $O(n^c)$
- ◆ **Nome:** Polinomial ou algébrico
- ◆ **Exemplo:** Análise gramatical adjacente à árvore; correspondência máxima para gráficos bipartidos; encontrar o determinante com decomposição LU

# Ordens Comuns de Funções

---

- ◆ **Notação:**  $O(c^n)$   $c > 1$
- ◆ **Nome:** Exponencial
- ◆ **Exemplo:** Encontrar a solução (exata) para o *travelling salesman problem* usando programação dinâmica; determinar se duas declarações lógicas são equivalentes usando a pesquisa de força bruta

# Ordens Comuns de Funções

---

- ◆ **Notação:**  $O(n!)$
- ◆ **Nome:** Fatorial
- ◆ **Exemplo:** Resolvendo o *travelling salesman problem* por meio de busca de força bruta; gerar todas as permutações irrestritas de um *poset*; encontrar o determinante com a expansão de Laplace; enumerando todas as partições de um conjunto

# Classificação de Algoritmos

---

- ♦ Existem várias maneiras de **classificar algoritmos**, cada uma com seus próprios méritos:
  - Por implementação;
  - Por paradigma ou metodologia de design;
  - Problemas de otimização;
  - Por campo de estudo;
  - Por complexidade;
- ♦ Alguns problemas podem ter vários algoritmos de complexidade diferente, enquanto outros problemas podem não ter algoritmos ou algoritmos eficientes conhecidos. Também existem mapeamentos de alguns problemas para outros problemas.

# Classificação por Implementação

---

- **Recursão:** Um **algoritmo recursivo** é aquele que invoca (faz referência a) a si mesmo repetidamente até que uma determinada condição (também conhecida como condição de terminação) corresponda, que é um método comum à programação funcional.
- Os algoritmos iterativos usam construções repetitivas como loops e, às vezes, estruturas de dados adicionais como pilhas para resolver os problemas fornecidos.
- Alguns problemas são naturalmente adequados para uma implementação ou outra. Por exemplo, as torres de Hanói são bem conhecidas por meio da implementação recursiva. Cada versão recursiva tem uma versão iterativa equivalente (mas possivelmente mais ou menos complexa) e vice-versa.

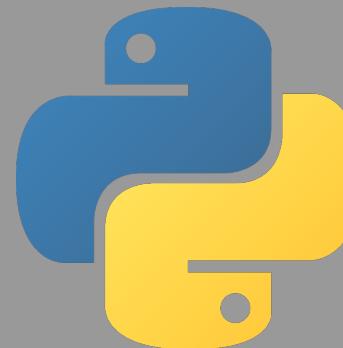
# Algoritmo de Euclides

---

- Em matemática, o **algoritmo de Euclides** é um método simples e eficiente de encontrar o **máximo divisor comum** entre dois números inteiros diferentes de zero.
- A seguir temos uma implementação recursiva do algoritmo de Euclides em **Python**:

```
def gcd(a, b):
    if b == 0:
        return a
    elif a > b:
        return gcd(a-b,b)
    else:
        return gcd(a,b-a)

print(gcd(50,25)) # 25
```



# Classificação por Implementação

---

- **Lógico:** Um algoritmo pode ser visto como dedução lógica controlada. Esta noção pode ser expressa como: **Algoritmo = lógica + controle.**
- O componente lógico expressa os axiomas que podem ser usados no cálculo e o componente de controle determina a maneira como a dedução é aplicada aos axiomas.
- Esta é a base para o paradigma de **programação lógica**. Em linguagens de programação de lógica pura, o componente de controle é fixo e os algoritmos são especificados fornecendo apenas o componente lógico.
- O apelo dessa abordagem é a **semântica elegante**: uma mudança nos axiomas produz uma mudança bem definida no algoritmo.

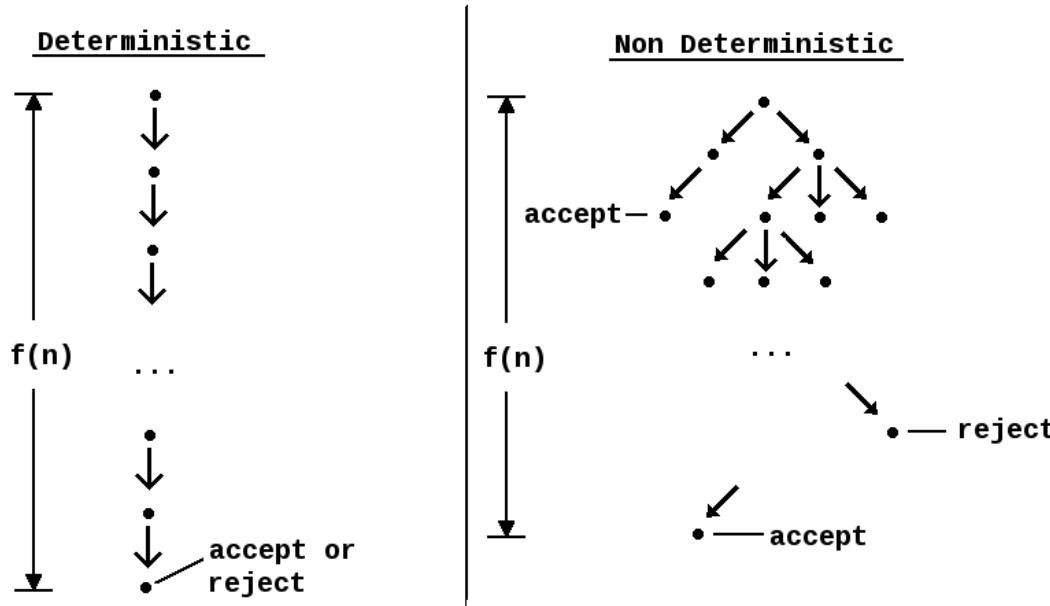
# Classificação por Implementação

---

- **Serial, paralelo ou distribuído:** Os algoritmos são geralmente discutidos com a suposição de que os computadores executam uma instrução de um algoritmo por vez. Esses computadores às vezes são chamados de computadores seriais. Um algoritmo projetado para tal ambiente é chamado de algoritmo serial, em oposição a algoritmos paralelos ou algoritmos distribuídos.
- Algoritmos paralelos tiram vantagem de arquiteturas de computador em que vários processadores podem trabalhar em um problema ao mesmo tempo, enquanto algoritmos distribuídos utilizam várias máquinas conectadas a uma rede de computadores. Algoritmos paralelos ou distribuídos dividem o problema em subproblemas mais simétricos ou assimétricos e coletam os resultados novamente. O consumo de recursos em tais algoritmos não é apenas os ciclos do processador em cada processador, mas também a sobrecarga de comunicação entre os processadores.
- Alguns algoritmos de ordenação podem ser parallelizados com eficiência, mas sua sobrecarga de comunicação é custosa. Os algoritmos iterativos são geralmente parallelizáveis. Alguns problemas não têm algoritmos paralelos e são chamados de problemas inherentemente seriais.

# Classificação por Implementação

- **Determinístico ou não-determinístico:** Algoritmos determinísticos resolvem o problema com decisão exata em cada etapa do algoritmo, enquanto algoritmos não determinísticos resolvem problemas por meio de suposições, embora suposições típicas sejam tornadas mais precisas com o uso de **heurísticas**.



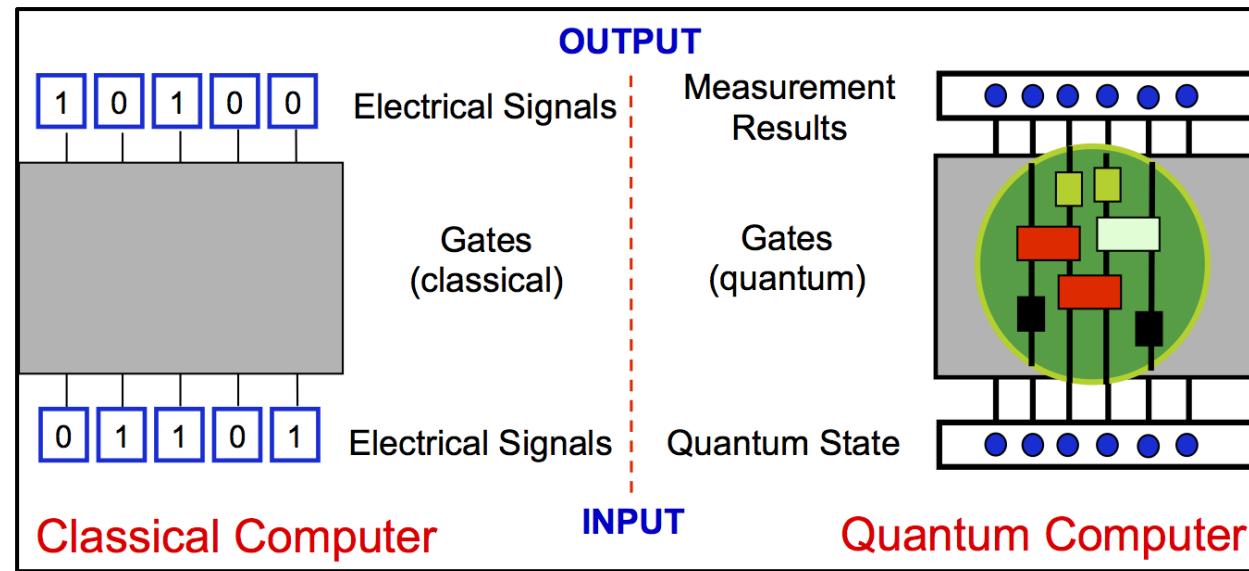
# Classificação por Implementação

---

- **Exato ou aproximado:** Enquanto muitos algoritmos chegam a uma solução exata, os algoritmos de aproximação buscam uma aproximação mais próxima da solução verdadeira.
- A aproximação pode ser alcançada usando uma estratégia determinística ou aleatória. Esses algoritmos têm valor prático para muitos problemas difíceis.
- Um dos exemplos de algoritmo aproximado é o [problema da mochila](#), onde existe um conjunto de itens dados. Seu objetivo é embalar a mochila para obter o valor total máximo. Cada item tem algum peso e algum valor. O peso total que pode ser carregado não é mais do que um número fixo X. Portanto, a solução deve considerar os pesos dos itens, bem como seu valor.

# Classificação por Implementação

- **Algoritmo Quântico:** Eles são executados em um modelo realista de **computação quântica**. O termo é geralmente usado para aqueles algoritmos que parecem inerentemente quânticos, ou usam algum recurso essencial da computação quântica, como **superposição quântica** ou **emaranhamento quântico**.



# Classificação por Paradigma ou Design

---

- Outra forma de classificar algoritmos é por sua **metodologia de design ou paradigma**. Existe um certo número de paradigmas, cada um diferente do outro. Além disso, cada uma dessas categorias inclui muitos tipos diferentes de algoritmos. Alguns paradigmas comuns são:
  - Força bruta ou busca exaustiva;
  - Divisão e Conquista;
  - Busca e enumeração;
  - Algoritmo Randomizado;
  - Redução de complexidade;
  - Backtracking;

# Classificação por Paradigma ou Design

---

- **Força bruta ou busca exaustiva:** Na ciência da computação, a busca de força bruta ou busca exaustiva, também conhecida como gerar e testar, é uma técnica de solução de problemas muito geral e paradigma algorítmico que consiste em enumerar sistematicamente todos os candidatos possíveis para a solução e verificar se cada candidato satisfaz a declaração do problema.
- Um algoritmo de força bruta que encontra os divisores de um número natural  $n$  enumeraria todos os inteiros de 1 a  $n$  e verificaria se cada um deles divide  $n$  sem resto.
- Embora uma busca de força bruta seja simples de implementar e sempre encontrará uma solução (se houver), os custos de implementação são proporcionais ao número de soluções candidatas - que em muitos problemas práticos tende a crescer muito rapidamente conforme o tamanho do problema aumenta (explosão combinatória).
- Portanto, a busca de força bruta é normalmente usada quando o tamanho do problema é limitado ou quando há heurísticas específicas do problema que podem ser usadas para reduzir o conjunto de soluções candidatas a um tamanho administrável.

# Classificação por Paradigma ou Design

---

- **Dividir e conquistar:** Um algoritmo de divisão e conquista reduz repetidamente uma instância de um problema a uma ou mais instâncias menores do mesmo problema (geralmente recursivamente) até que as instâncias sejam pequenas o suficiente para resolver facilmente.
- Um exemplo de divisão para conquistar é o Merge Sort. A ordenação pode ser feita em cada segmento de dados depois de dividir os dados em segmentos menores e a ordenação de todos os dados pode ser obtida na fase de conquista, mesclando os segmentos.
- Uma variante mais simples de dividir e conquistar é chamada de algoritmo de **diminuir e conquistar**, que resolve um subproblema idêntico e usa a solução desse subproblema para resolver o problema maior.
- Dividir para conquistar divide o problema em vários subproblemas e, portanto, o estágio de conquista é mais complexo do que os algoritmos de diminuir e conquistar. Um exemplo de algoritmo de diminuição e conquista é o [algoritmo de busca binária](#).

# Classificação por Paradigma ou Design

---

- ♦ **Busca e enumeração:** Muitos problemas (como jogar xadrez) podem ser modelados como problemas em **grafos**.
- ♦ Um **algoritmo de exploração de grafo** especifica regras para se mover em um grafo e é útil para tais problemas.
- ♦ Em ciência da computação, a travessia de grafo (também conhecida como pesquisa de grafo) refere-se ao processo de visitar (verificar e / ou atualizar) cada vértice em um grafo. Essas travessias são classificadas pela ordem em que os vértices são visitados.

# Classificação por Paradigma ou Design

---

- **Algoritmo Randomizado:** Esses algoritmos fazem algumas escolhas aleatoriamente (ou pseudo-aleatoriamente).
- Eles podem ser muito úteis para encontrar soluções aproximadas para problemas onde encontrar soluções exatas pode ser impraticável.
- Para alguns desses problemas, sabe-se que as aproximações mais rápidas devem envolver alguma aleatoriedade. Se algoritmos randomizados com complexidade de tempo polinomial podem ser os algoritmos mais rápidos para alguns problemas, é uma questão em aberto conhecida como **problema P versus NP**. Existem duas grandes classes de tais algoritmos:
  1. Os **algoritmos de Monte Carlo** retornam uma resposta correta com alta probabilidade. Por exemplo, RP é a subclasse desses que são executados em tempo polinomial.
  2. Os **algoritmos de Las Vegas** sempre retornam a resposta correta, mas seu tempo de execução é limitado apenas probabilisticamente, por exemplo, ZPP.

# Classificação por Paradigma ou Design

---

- **Redução de complexidade:** Esta técnica envolve resolver um problema difícil, transformando-o em um problema mais conhecido para o qual temos (esperançosamente) **algoritmos assintoticamente otimizados**.
- O objetivo é encontrar um algoritmo de redução cuja complexidade não seja dominada pelo algoritmo reduzido resultante.
- Por exemplo, um algoritmo de seleção para encontrar a mediana em uma lista não ordenada envolve primeiro a ordenação da lista (a parte custosa) e, em seguida, extraír o elemento do meio na lista ordenada (a parte de baixo custo). Essa técnica também é conhecida como **transformar e conquistar**.

# Classificação por Paradigma ou Design

---

- **Backtracking:** Backtracking é um algoritmo geral para encontrar todas (ou algumas) soluções para alguns problemas computacionais, notadamente **problemas de satisfação de restrição**, que cria candidatos para as soluções de forma incremental e abandona um candidato ("backtracks") assim que determina que o candidato não pode possivelmente ser completado para uma solução válida.
- O exemplo clássico do uso de backtracking é o **quebra-cabeça de oito rainhas**, que pede todos os arranjos de oito rainhas de xadrez em um tabuleiro de xadrez padrão para que nenhuma rainha ataque qualquer outra. Na abordagem de **backtracking** comum, as candidatas parciais são arranjos de  $k$  rainhas nas primeiras  $k$  linhas do tabuleiro, todas em diferentes linhas e colunas. Qualquer solução parcial que contenha duas rainhas que se atacam mutuamente pode ser abandonada.

# Problemas de Otimização

---

- ♦ Para **problemas de otimização**, há uma classificação mais específica de algoritmos; um algoritmo para tais problemas pode cair em uma ou mais das categorias gerais descritas anteriormente, bem como em uma das seguintes:
  - Programação Linear;
  - Programação Dinâmica;
  - O método ganancioso;
  - O método heurístico;

# Problemas de Otimização

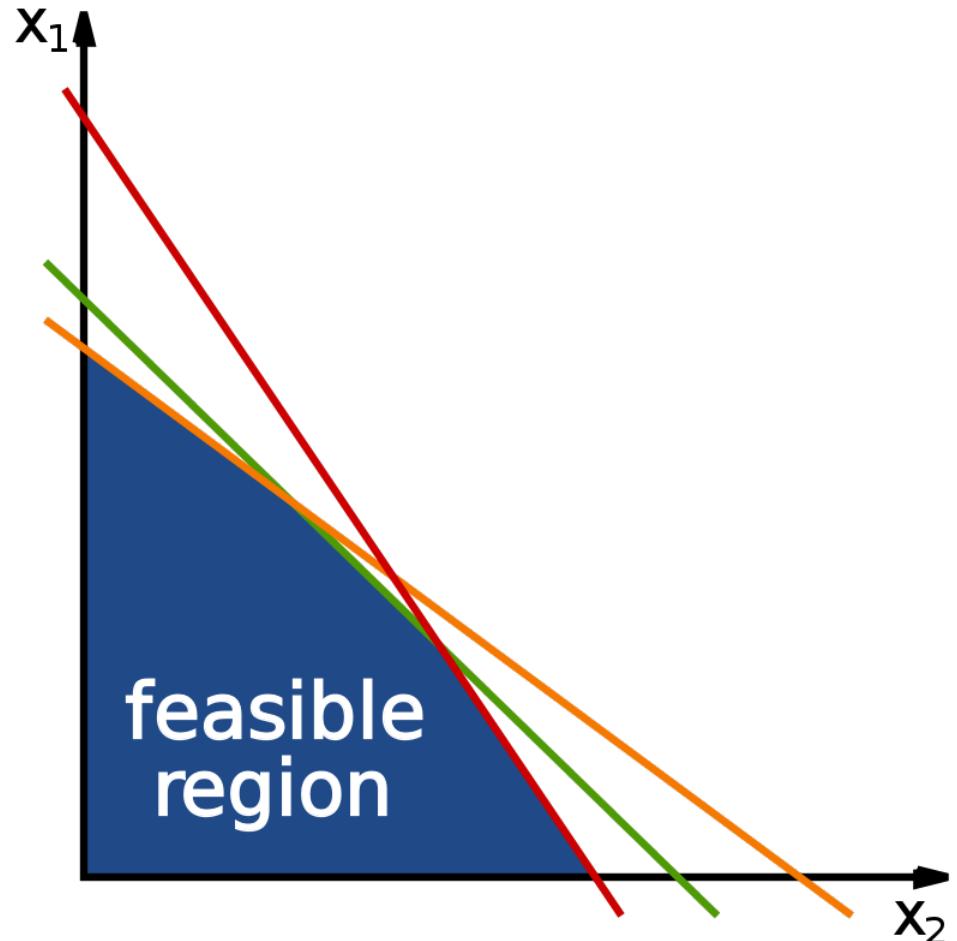
---

- ◆ **Programação Linear:** Ao procurar soluções otimizadas para uma função linear limitada a restrições lineares de igualdade e desigualdade, as restrições do problema podem ser usadas diretamente na produção das soluções otimizadas. Existem algoritmos que podem resolver qualquer problema nesta categoria, como o popular **algoritmo simplex**.
- ◆ Os problemas que podem ser resolvidos com a programação linear incluem o **problema de fluxo máximo** para **grafos direcionados**.
- ◆ Mais formalmente, a programação linear é uma técnica para a otimização de uma **função objetivo linear**, sujeita à igualdade linear e restrições de desigualdade linear. Sua região viável é um **politopo** convexo, que é um conjunto definido como a interseção de muitos **meios espaços** finitos, cada um dos quais definido por uma desigualdade linear.

# Problemas de Otimização

---

Em um **problema de programação linear**, uma série de restrições lineares produz uma **região convexa viável** de valores possíveis para essas variáveis. No caso de duas variáveis, essa região tem a forma de um polígono simples convexo.



# Problemas de Otimização

---

- **Programação Dinâmica:** Quando um problema mostra **subestruturas otimizadas** - significando que a solução otimizada para um problema pode ser construída a partir de soluções otimizadas para subproblemas - e **subproblemas sobrepostos**, significando que os mesmos subproblemas são usados para resolver muitas instâncias de problema diferentes, uma abordagem mais rápida chamada programação dinâmica evita recomputação de soluções que já foram calculados.
- Por exemplo, o **algoritmo Floyd–Warshall**, o caminho mais curto para uma meta de um vértice em um **grafo ponderado** pode ser encontrado usando o caminho mais curto para a meta de todos os vértices adjacentes.
- Programação dinâmica e **memoization** caminham juntas.

# Problemas de Otimização

---

- **Programação Dinâmica:** A principal diferença entre a programação dinâmica e dividir e conquistar é que os subproblemas são mais ou menos independentes em dividir e conquistar, enquanto os subproblemas se sobrepõem na programação dinâmica.
- A diferença entre a programação dinâmica e a recursão direta está no armazenamento em cache ou na **memoização** de chamadas recursivas.
- Quando os subproblemas são independentes e não há repetição, a **memoização** não ajuda; portanto, a programação dinâmica não é uma solução para todos os problemas complexos.
- Ao usar **memoização** ou manter uma tabela de subproblemas já resolvidos, a programação dinâmica reduz a natureza exponencial de muitos problemas à complexidade polinomial.

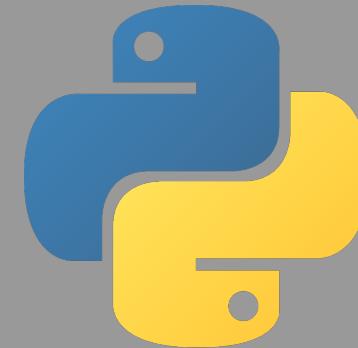
# Calculando Números de Fibonacci

- A seguir temos um exemplo de um algoritmo recursivo escrito em **Python** que calcula **números de Fibonacci** usando **memoization**:

```
cache = {0: 0, 1: 1, 2: 1, 3: 2}
```

```
def fibonacci_recursive(n):
    if cache.get(n, None) is None:
        if n % 2:
            f1 = fibonacci_recursive((n - 1) / 2)
            f2 = fibonacci_recursive((n + 1) / 2)
            cache[n] = f1 ** 2 + f2 ** 2
        else:
            f1 = fibonacci_recursive(n / 2 - 1)
            f2 = fibonacci_recursive(n / 2)
            cache[n] = (2 * f1 + f2) * f2
    return cache[n]
```

```
print(fibonacci_recursive(100)) # 354224848179261915075
```



# Problemas de Otimização

---

- **O método ganancioso:** Um **algoritmo guloso** ou **ganancioso** é semelhante a um algoritmo de programação dinâmica, pois funciona examinando subestruturas, neste caso não do problema, mas de uma determinada solução.
- Esse algoritmos começam com alguma solução, que pode ser fornecida ou construída de alguma forma, e a melhora fazendo pequenas modificações.
- Para alguns problemas eles podem encontrar a solução otimizada, enquanto para outros eles param em **ótimos locais**, isto é, em soluções que não podem ser melhoradas pelo algoritmo, mas não são otimizadas.
- O uso mais popular de algoritmos gulosos é para encontrar a **minimal spanning tree** onde encontrar a solução ideal é possível com este método. **Huffman Tree**, **Kruskal**, **Prim**, **Sollin** são algoritmos gananciosos que podem resolver este problema de otimização.

# Problemas de Otimização

---

- **O método heurístico:** Em problemas de otimização, **algoritmos heurísticos** podem ser usados para encontrar uma solução próxima da solução otimizada nos casos em que encontrar a solução otimizada é impraticável.
- Esses algoritmos funcionam ficando cada vez mais próximos da solução ideal à medida que progridem.
- Em princípio, se executado por um período infinito de tempo, eles encontrarão a solução otimizada. Seu mérito é que eles podem encontrar uma solução muito próxima da solução otimizada em um tempo relativamente curto.
- Esses algoritmos incluem **local search**, **tabu search**, **simulated annealing** e **algoritmos genéticos**.
- Alguns deles, como o **simulated annealing**, são algoritmos não determinísticos, enquanto outros, como o **tabu search**, são determinísticos.

# Classificação por Campo de Estudo

---

- ◆ Cada campo da ciência tem seus próprios problemas e precisa de algoritmos eficientes.
- ◆ Problemas relacionados em um campo são freqüentemente estudados juntos.
- ◆ Alguns exemplos de classes são: **algoritmos de busca, algoritmos de ordenação, algoritmos de mesclagem, algoritmos numéricos, algoritmos de grafo, algoritmos de string, algoritmos geométricos computacionais, algoritmos combinatórios, algoritmos médicos, machine learning, criptografia, algoritmos de compressão de dados e parsing techniques.**

# Classificação por Complexidade

---

- Os algoritmos podem ser classificados pela quantidade de tempo que precisam para serem concluídos em comparação com seu tamanho de input:
  - **Tempo constante:** se o tempo necessário para o algoritmo for o mesmo, independente do tamanho do input. Por exemplo: um acesso a um elemento de um array.
  - **Tempo logarítmico:** se o tempo for uma função logarítmica do tamanho do input. Por exemplo: algoritmo de busca binária.
  - **Tempo linear:** se o tempo for proporcional ao tamanho do input. Por exemplo: a travessia de uma lista.
  - **Tempo polinomial:** se o tempo for uma potência do tamanho do input. Por exemplo: o algoritmo bubble sort tem complexidade de tempo quadrática.
  - **Tempo exponencial:** se o tempo for uma função exponencial do tamanho do input. Por exemplo: Pesquisa de força bruta.

# Design de Algoritmos

---

- O **design de algoritmo** se refere a um método ou processo matemático para a solução de problemas e engenharia de algoritmos.
- O design de algoritmos é parte de muitas teorias de solução de pesquisa operacional, como programação dinâmica e divisão e conquista.
- As técnicas para projetar e implementar projetos de algoritmo também são chamadas de padrões de projeto de algoritmo, com exemplos incluindo o padrão de método de template e o padrão decorator.
- Um dos aspectos mais importantes do design de algoritmo é a eficiência dos recursos (tempo de execução, uso de memória); a notação Big O é usada para descrever o crescimento do tempo de execução de um algoritmo conforme o tamanho de seu input aumenta.

# Design de Algoritmos

---

- ♦ Etapas típicas no desenvolvimento de algoritmos incluem:
  1. Definição do problema
  2. Desenvolvimento de um modelo
  3. Especificação do algoritmo
  4. Projetando um algoritmo
  5. Verificar a exatidão do algoritmo
  6. Análise de algoritmo
  7. Implementação do algoritmo
  8. Testes do algoritmo
  9. Preparação de documentação

# Expressando Algoritmos

---

- Como já vimos, os algoritmos podem ser expressos em muitos tipos de notação, incluindo **linguagens naturais**, **pseudocódigo**, **fluxogramas**, **diagramas de drakon**, **linguagens de programação** ou **tabelas de controle** (processadas por interpretadores).
- Expressões de linguagem natural de algoritmos tendem a ser prolixas e ambíguas e raramente são usadas para algoritmos complexos ou técnicos.
- Pseudocódigo, fluxogramas, diagramas de drakon e tabelas de controle são formas estruturadas de expressar algoritmos que evitam muitas das ambigüidades comuns nas declarações baseadas na linguagem natural.
- As linguagens de programação destinam-se principalmente a expressar algoritmos em uma forma que pode ser executada por um computador, mas também são freqüentemente usadas como uma forma de definir ou documentar algoritmos.

# Expressando Algoritmos

---

- As representações de algoritmos podem ser classificadas em três níveis aceitos de descrição da **máquina de Turing**, como segue:

## 1. Descrição de alto nível

"... Prosa para descrever um algoritmo, ignorando os detalhes de implementação. Nesse nível, não precisamos mencionar como a máquina gerencia sua fita ou cabeçote. "

## 2. Descrição de implementação

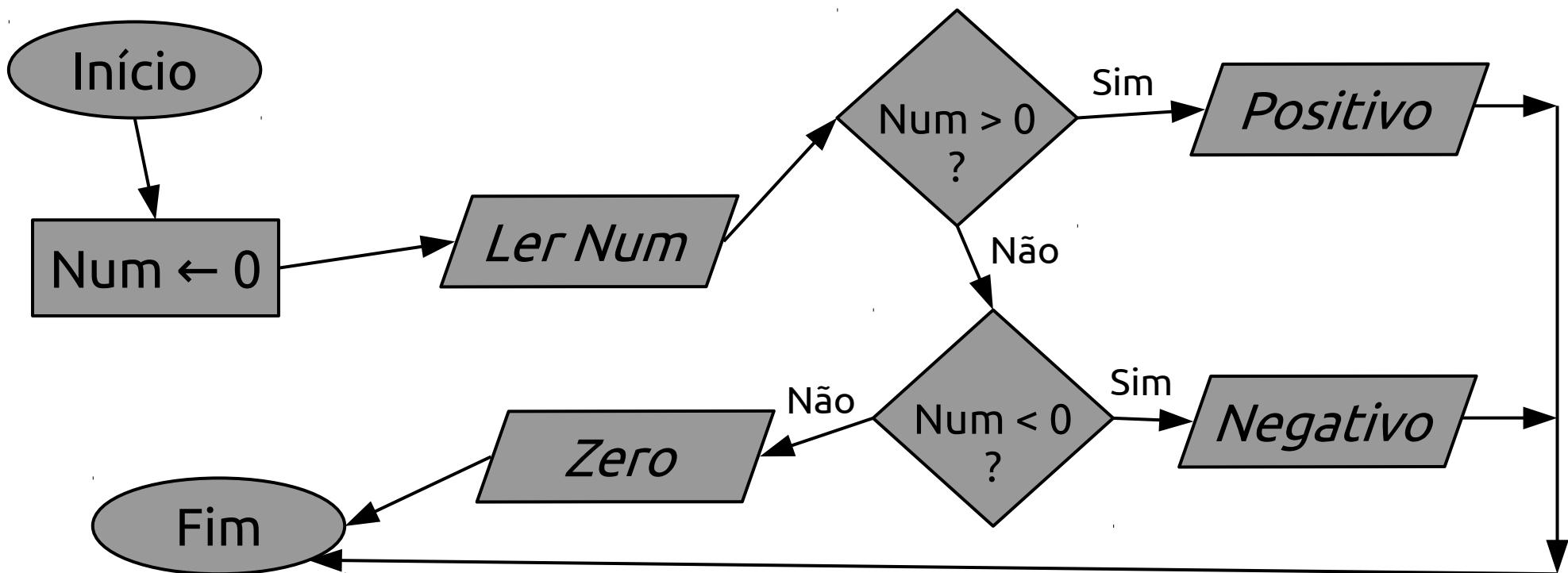
"... A prosa costumava definir a forma como a máquina de Turing usa seu cabeçote e como armazena dados em sua fita. Neste nível, não fornecemos detalhes de estados ou função de transição. "

## 3. Descrição formal

Mais detalhado, "nível mais baixo", fornece a "tabela de estados" da máquina de Turing.

# Fluxograma

- Podemos usar um **fluxograma** para expressar um algoritmo que verifica se um número é positivo, negativo ou zero.



# O Algoritmo Heapsort

---

- Na ciência da computação, o heapsort é um algoritmo de [ordenação baseado em comparação](#).
- Heapsort pode ser entendido como a versão aprimorada da [árvore de busca binária](#).
- Ele não cria um nó como no caso da árvore de busca binária, em vez disso, ele constrói o [heap](#) ajustando a posição dos elementos dentro do próprio array.
- Nesse método, uma estrutura em árvore chamada heap é usada, em que heap é um tipo de [árvore binária](#).
- Uma árvore binária balanceada ordenada é chamada de [Min-heap](#), onde o valor na raiz de qualquer subárvore é menor ou igual ao valor de qualquer um de seus filhos.
- Uma árvore binária balanceada ordenada é chamada de [Max-Heap](#), em que o valor na raiz de qualquer subárvore é maior ou igual ao valor de qualquer um de seus filhos.

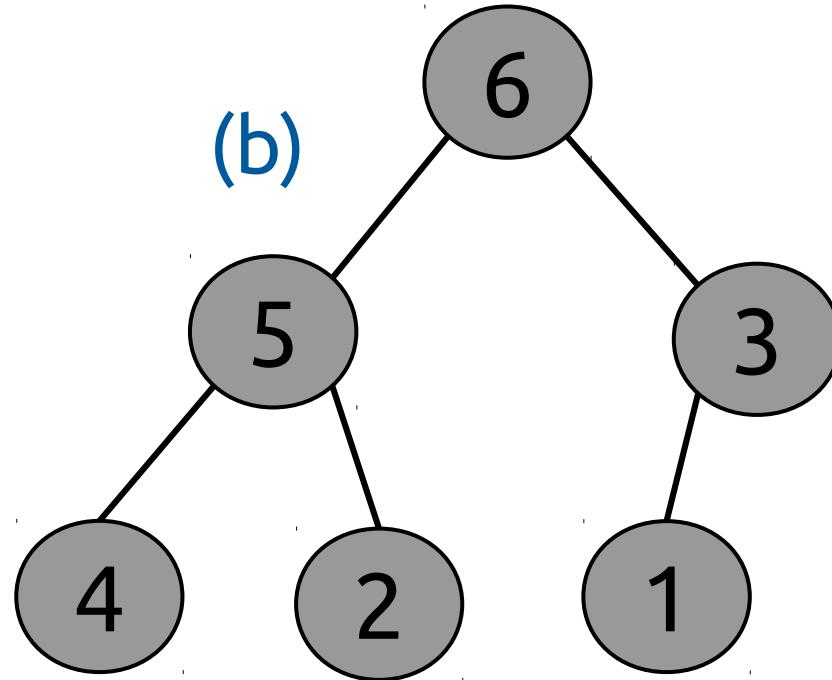
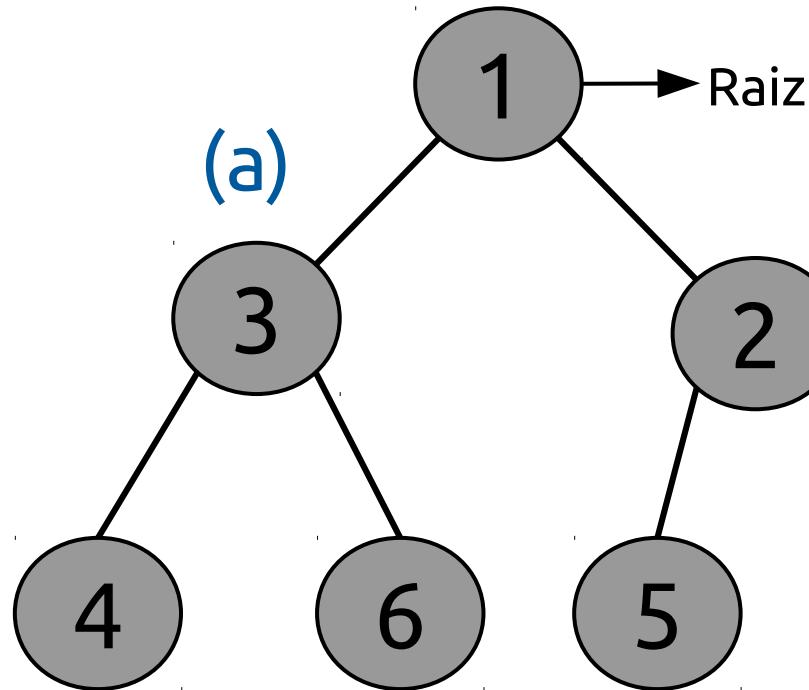
# O Algoritmo Heapsort

---

- Vamos então entender o que é um **heap**. Um heap é uma estrutura de dados em árvore que satisfaz as seguintes propriedades:
  1. **Propriedade da forma:** Heap é sempre uma árvore binária completa, o que significa que todos os níveis de uma árvore são totalmente preenchidos. Não deve haver um nó com apenas um filho. Cada nó, exceto as folhas, deve ter dois filhos, então apenas um heap é chamado como uma árvore binária completa.
  2. **Propriedade heap:** Todos os nós são maiores ou iguais ou menores ou iguais a cada um de seus filhos. Isso significa que se o nó pai for maior do que o nó filho, ele é chamado de heap máximo. Considerando que, se o nó pai for menor do que o nó filho, ele é chamado de heap mínimo.

# O Algoritmo Heapsort

- A seguir temos o exemplo de um **Min-Heap (a)** e **Max-Heap (b)**:



Não é necessário que os dois filhos estejam em alguma ordem. às vezes, o valor do filho esquerdo pode ser maior do que o valor do filho certo e, em outras ocasiões, pode ser o contrário.

# O Algoritmo Heapsort

---

- ♦ Basicamente, existem duas fases envolvidas na ordenação de elementos usando o algoritmo **heapsort**, elas são as seguintes:
  1. Primeiro, começamos com a construção de um heap ajustando os elementos do array.
  2. Depois que o heap for criado, repetidamente elimine o elemento raiz do heap deslocando-o para o final do array e, em seguida, armazene a estrutura do heap com os elementos restantes.

# O Algoritmo Heapsort

---

- Suponha que um array consista em **N** elementos distintos na memória, então o algoritmo heapsort funciona da seguinte maneira:
  1. Para começar, um heap é construído movendo os elementos para sua posição adequada no array. Isso significa que conforme os elementos são percorridos do array, a raiz, seu filho esquerdo e seu filho direito são preenchidos, respectivamente, formando uma árvore binária.
  2. Na segunda fase, o elemento raiz é eliminado do heap movendo-o para o final do array.
  3. Os elementos equilibrados podem não formar um heap. Portanto, novamente as etapas 1 e 2 são repetidas para o balanço dos elementos. O procedimento é continuado até que todos os elementos sejam eliminados.

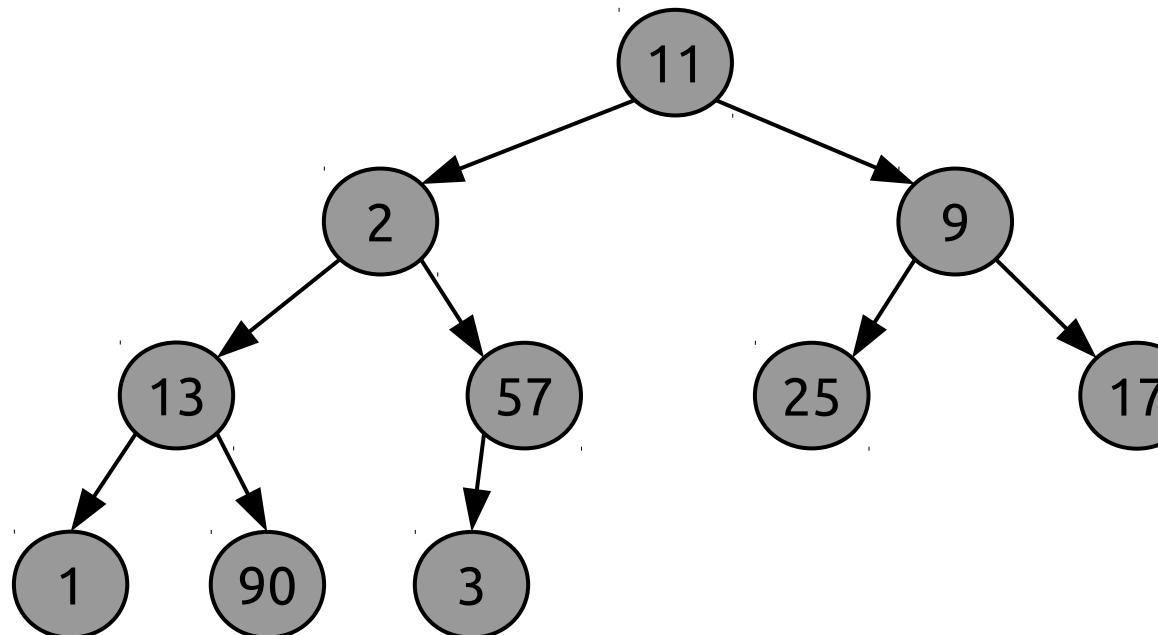
# O Algoritmo Heapsort

---

- ♦ Ao eliminar um elemento do heap, precisamos diminuir o valor de índice máximo do array em um. Os elementos são eliminados em ordem decrescente para um heap máximo e em ordem crescente para um heap mínimo.
- ♦ Considere o seguinte exemplo:
  - Suponha que o array a ser ordenado contenha os seguintes elementos: {11, 2, 9, 13, 57, 25, 17, 1, 90, 3}.
  - A primeira etapa agora é criar um heap a partir dos elementos do array.
  - Para isso, considere uma árvore binária que pode ser construída a partir dos elementos do array, o elemento zero seria o elemento raiz e os filhos esquerdo e direito de qualquer elemento seriam avaliados em  $i$ ,  $2 * i + 1$  e  $2 * i + 2$ º índice respectivamente.

# O Algoritmo Heapsort

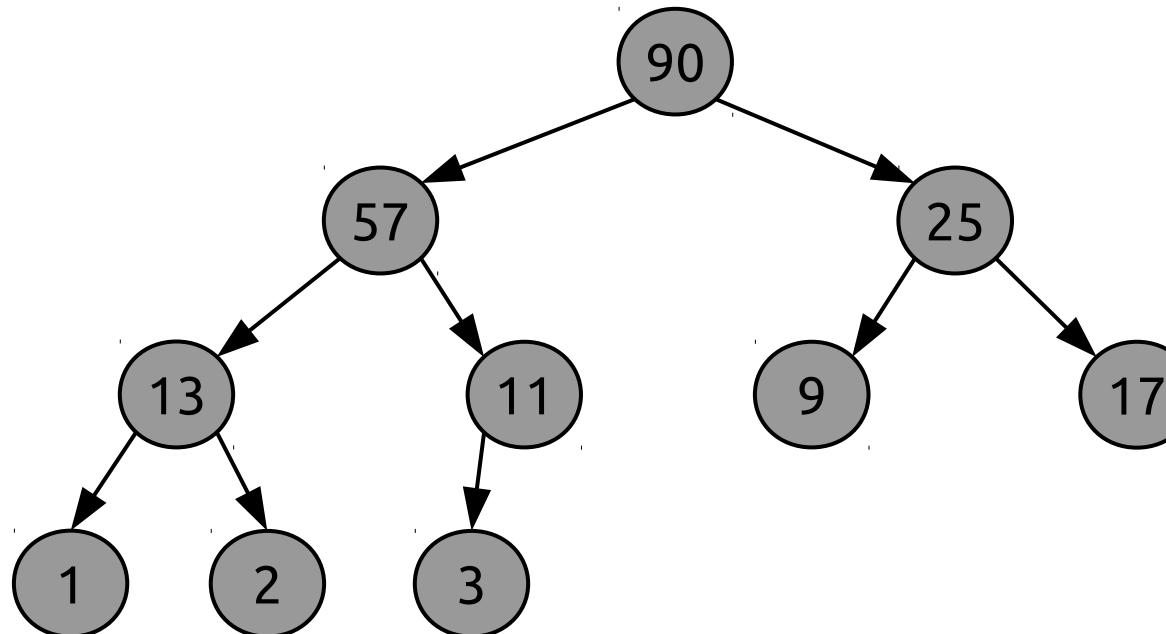
- O diagrama a seguir mostra a versão da árvore binária do array:



11	2	9	13	57	25	17	1	90	3
----	---	---	----	----	----	----	---	----	---

# O Algoritmo Heapsort

- ◆ Construímos o heap a partir da árvore binária anterior:



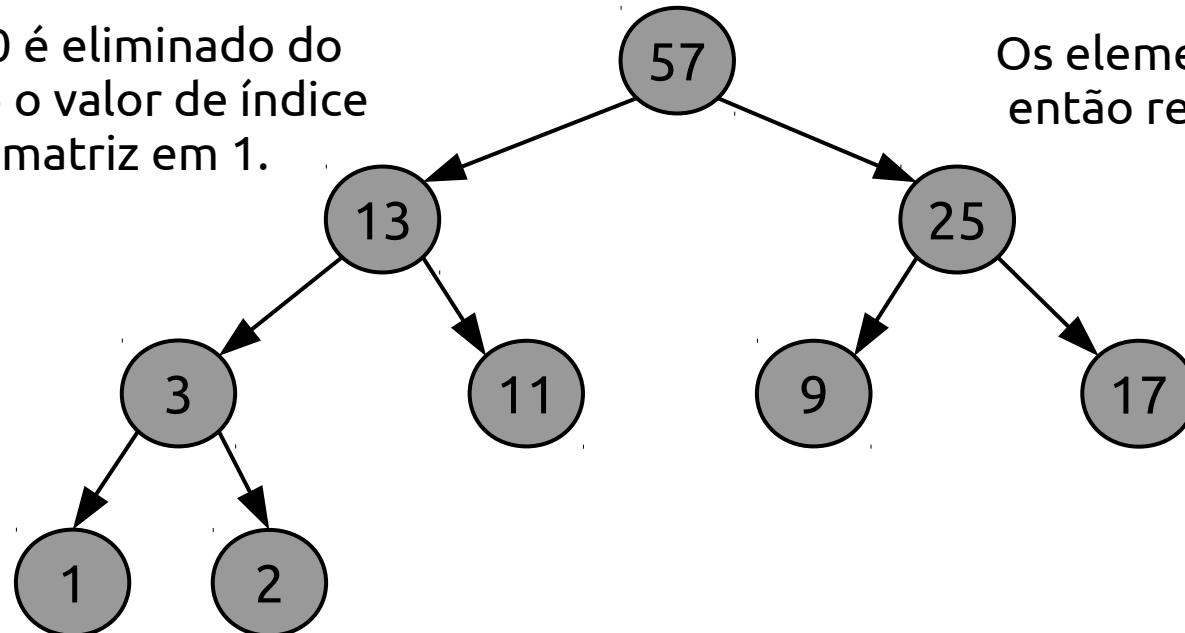
90	57	25	13	11	9	17	1	2	3
----	----	----	----	----	---	----	---	---	---

# O Algoritmo Heapsort

- Agora, a raiz 90 é movido para o último local trocando-o por 3.

Finalmente, 90 é eliminado do heap reduzindo o valor de índice máximo da matriz em 1.

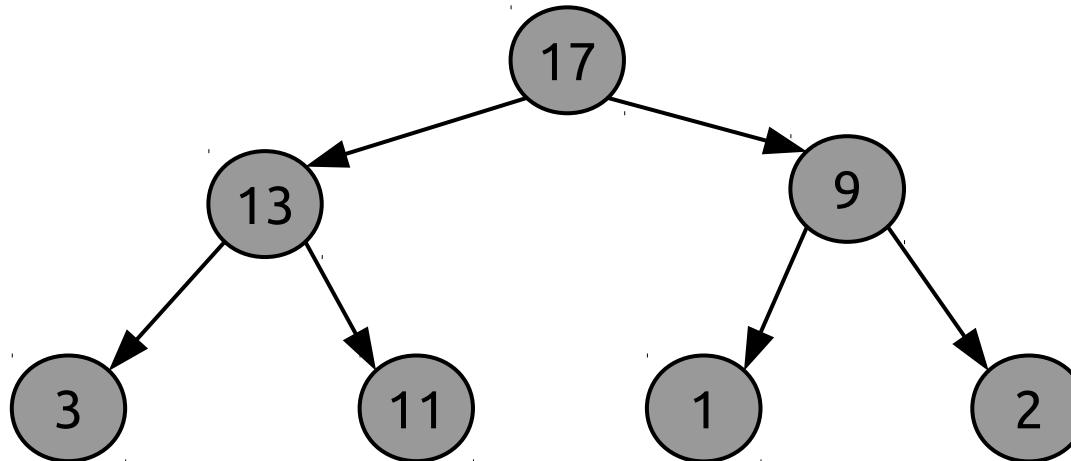
Os elementos de equilíbrio são então reorganizados no heap.



57	13	25	3	11	9	17	1	2	90
----	----	----	---	----	---	----	---	---	----

# O Algoritmo Heapsort

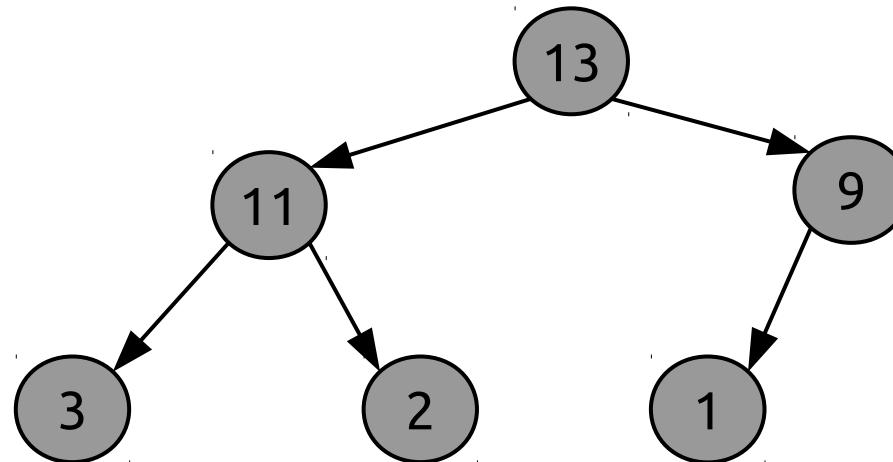
- Da mesma forma, quando a raiz atual 57 é trocada por 2 e eliminada do heap, reduzindo o valor de índice máximo do array em 1. Os elementos de equilíbrio são então reorganizados no heap.



17	13	9	3	11	1	2	25	57	90
----	----	---	---	----	---	---	----	----	----

# O Algoritmo Heapsort

- Seguindo a mesma abordagem, as seguintes fases são seguidas até que o array totalmente ordenado seja alcançado.

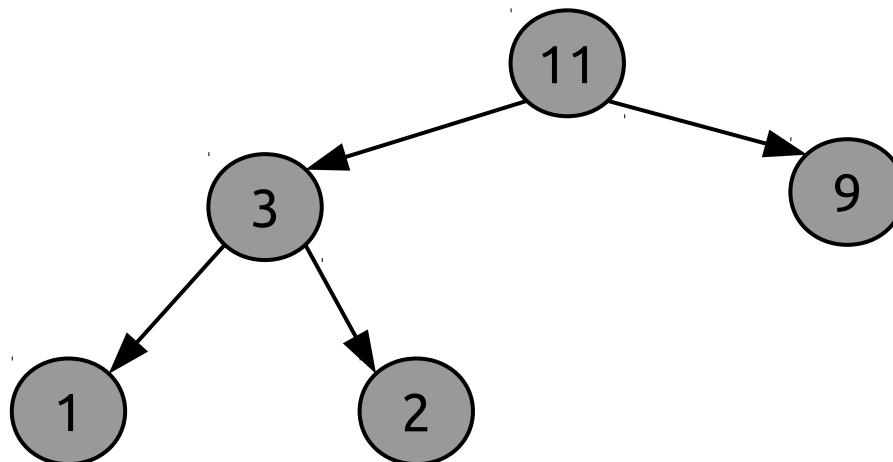


13	11	9	3	2	1	17	25	57	90
----	----	---	---	---	---	----	----	----	----

# O Algoritmo Heapsort

---

- Seguindo a mesma abordagem, as seguintes fases são seguidas até que o array totalmente ordenado seja alcançado.

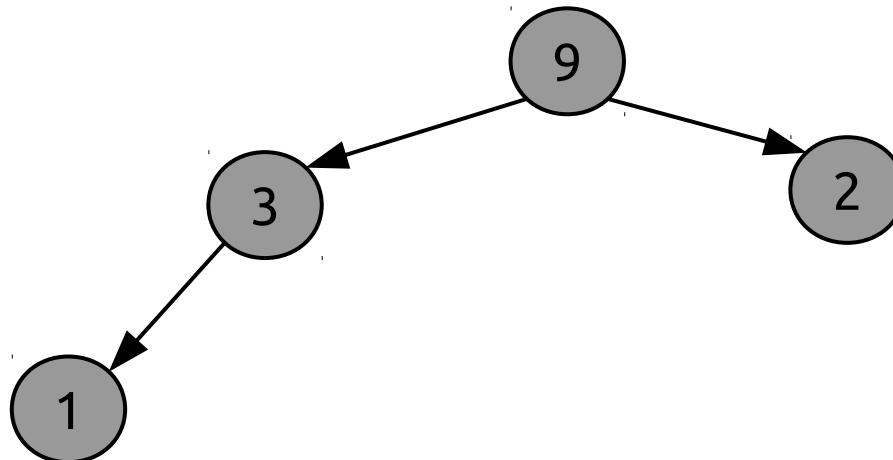


11	3	9	1	2	13	17	25	57	90
----	---	---	---	---	----	----	----	----	----

# O Algoritmo Heapsort

---

- Seguindo a mesma abordagem, as seguintes fases são seguidas até que o array totalmente ordenado seja alcançado.

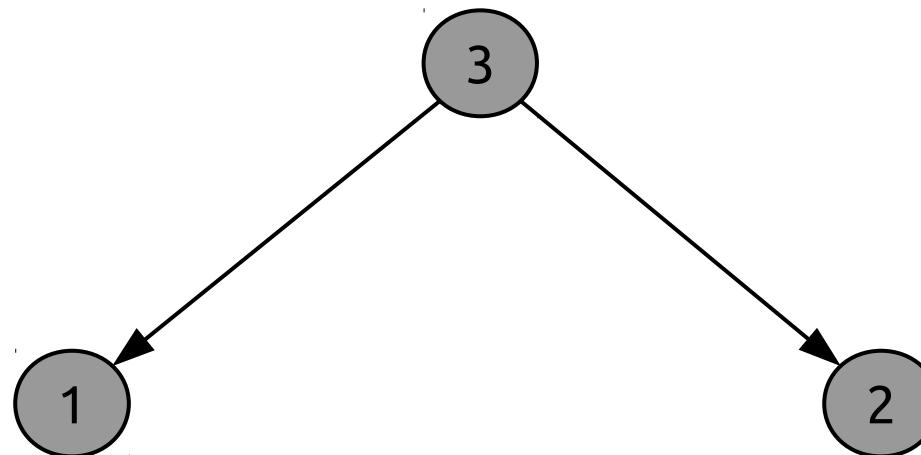


9	3	2	1	11	13	17	25	57	90
---	---	---	---	----	----	----	----	----	----

# O Algoritmo Heapsort

---

- Seguindo a mesma abordagem, as seguintes fases são seguidas até que o array totalmente ordenado seja alcançado.

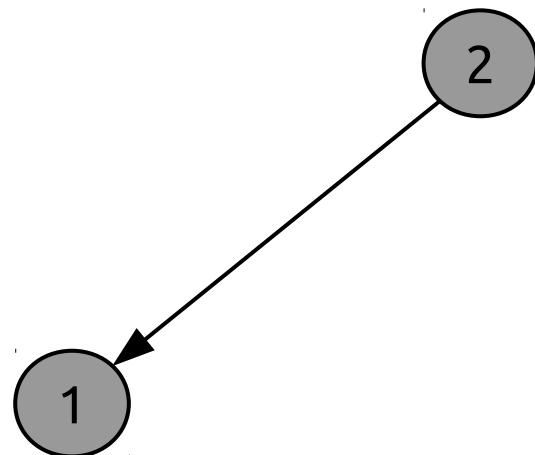


3	1	2	9	11	13	17	25	57	90
---	---	---	---	----	----	----	----	----	----

# O Algoritmo Heapsort

---

- Seguindo a mesma abordagem, as seguintes fases são seguidas até que o array totalmente ordenado seja alcançado.



2	1	3	9	11	13	17	25	57	90
---	---	---	---	----	----	----	----	----	----

# O Algoritmo Heapsort

---

- ◆ O array finalmente está ordenando através do algoritmo heapsort.

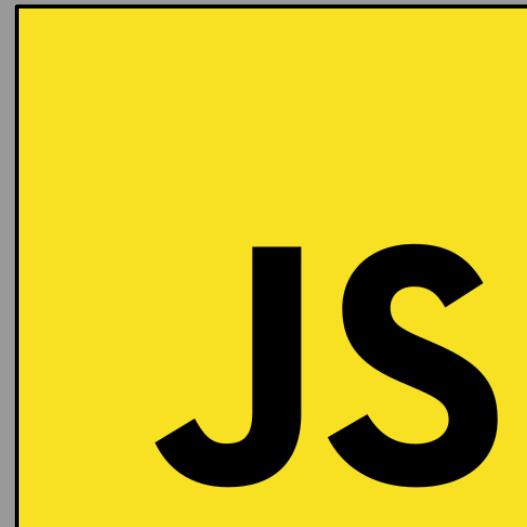
1

1	2	3	9	11	13	17	25	57	90
---	---	---	---	----	----	----	----	----	----

# O Algoritmo Heapsort

- ◆ A seguir temos a implementação do procedimento **MaxHeap** na linguagem **JavaScript**:

```
function maxHeap(arr, i) {  
    const left = 2 * i + 1  
    const right = 2 * i + 2  
    let max = i  
    if (left < arrLength && arr[left] > arr[max]) {  
        max = left  
    }  
    if (right < arrLength && arr[right] > arr[max]) {  
        max = right  
    }  
    if (max != i) {  
        swap(arr, i, max)  
        maxHeap(arr, max)  
    }  
}
```



# O Algoritmo Heapsort

- ♦ E por fim, o procedimento **Heapsort**:

```
function heapSort(arr) {  
    arrLength = arr.length  
  
    for (let i = Math.floor(arrLength / 2); i >= 0; i -= 1) {  
        maxHeap(arr, i)  
    }  
  
    for (i = arr.length - 1; i > 0; i--) {  
        swap(arr, 0, i)  
        arrLength--  
        maxHeap(arr, 0)  
    }  
    return  
}
```



A função **swap** é responsável por trocar o valor de dois valores

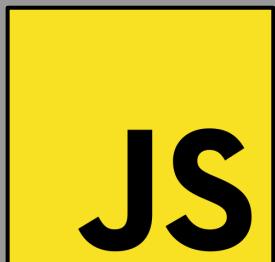
```
function swap(arr, indexA, indexB) {  
    const temp = arr[indexA]  
    arr[indexA] = arr[indexB]  
    arr[indexB] = temp  
}
```

# O Algoritmo Heapsort

---

- ♦ Finalmente podemos testar o algoritmo Heapsort:

```
let arrLength  
const array = [11, 2, 9, 13, 57, 25, 17, 1, 90, 3]  
heapSort(array)  
console.log(array)  
// [ 1, 2, 3, 9, 11, 13, 17, 25, 57, 90 ]
```

A yellow square containing the letters "JS" in black.

# O Algoritmo Heapsort

---

- ◆ Heapsort é um algoritmo **in-place**.
- ◆ Sua implementação típica não é estável, mas pode se tornar **estável**.
- ◆ Complexidade de tempo: a complexidade de tempo de heapify é  **$O(\log n)$** .
- ◆ A complexidade de tempo de criar e construir o heap é  **$O(n)$**  e a complexidade de tempo geral de Heapsort é  **$O(n \log n)$** .
- ◆ O algoritmo de Heapsort tem usos limitados porque **Quicksort** e **Mergesort** são melhores na prática. No entanto, a própria estrutura de dados Heap é enormemente usada.

# O Algoritmo Quicksort

---

- ♦ O **algoritmo quicksort** tem um tempo de execução de pior caso de  $\Theta(n^2)$  em um array de input de  $n$  números.
- ♦ Apesar desse tempo de execução lento no pior caso, o quicksort costuma ser a melhor escolha prática para ordenar porque é notavelmente eficiente na média: seu tempo de execução esperado é  $\Theta(n \log n)$ , e os fatores constantes ocultos na notação  $\Theta(n \log n)$  são bastante pequenos.
- ♦ Também tem a vantagem de ordenação **in place**, e funciona bem mesmo em ambientes de memória virtual.

# O Algoritmo Quicksort

---

- O Quicksort, assim como o Merge Sort, aplica o paradigma **dividir e conquistar**. Aqui está o processo de divisão e conquista de três etapas para ordenar um subarray típico  $A[p \dots r]$ :
  - **Dividir:** Particione (reorganize) o array  $A[p \dots r]$  em dois (possivelmente vazios) subarrays  $A[p \dots q - 1]$  e  $A[q + 1 \dots r]$  de modo que cada elemento de  $A[p \dots q - 1]$  seja menor que ou igual a  $A[q]$ , que por sua vez é menor ou igual a cada elemento de  $A[q + 1 \dots r]$ . Calcule o índice  $q$  como parte deste procedimento de particionamento.
  - **Conquistar:** Ordene os dois subarrays  $A[p \dots q - 1]$  e  $A[q + 1 \dots r]$  por chamadas recursivas para o Quicksort.
  - **Combine:** Como os subarrays já estão ordenados, nenhum trabalho é necessário para combiná-los: o array inteiro  $A[p \dots r]$  agora está ordenado.

# O Algoritmo Quicksort

---

- ♦ O seguinte procedimento implementa quicksort:

```
QUICKSORT(A, p, r)
1 if p < r
2   q = PARTITION(A, p, r)
3   QUICKSORT(A, p, q - 1)
4   QUICKSORT(A, q + 1, r)
```

- ♦ Para ordenar inteiramente um array **A**, a chamada inicial é **QUICKSORT(A, 1, A.length)**.

# O Algoritmo Quicksort

- ♦ A chave do algoritmo é o procedimento **PARTITION**, que reorganiza o subarray **A[p...r]** **in place**.

```
PARTITION(A, p, r)
```

```
1 x = A[r]
2 i = p - 1
3 for j = p to r - 1
4   if A[j] ≤ x
5     i = i + 1
6     exchange A[i] with A[j]
7   exchange A[i + 1] with A[r]
8 return i + 1
```

# O Algoritmo Quicksort

---

- ♦ A figura que vamos apresentar, mostrará como **PARTITION** funciona em um array de 8 elementos.
- ♦ **PARTITION** sempre seleciona um elemento  $x = A[r]$  como um elemento **pivô** em torno do qual particionar o subarray  $A[p \dots r]$ .
- ♦ Conforme o procedimento é executado, ele particiona o array em quatro regiões (possivelmente vazias).
- ♦ No início de cada iteração do loop for nas **linhas 3-6**, as regiões satisfazem certas **propriedades**.

# O Algoritmo Quicksort

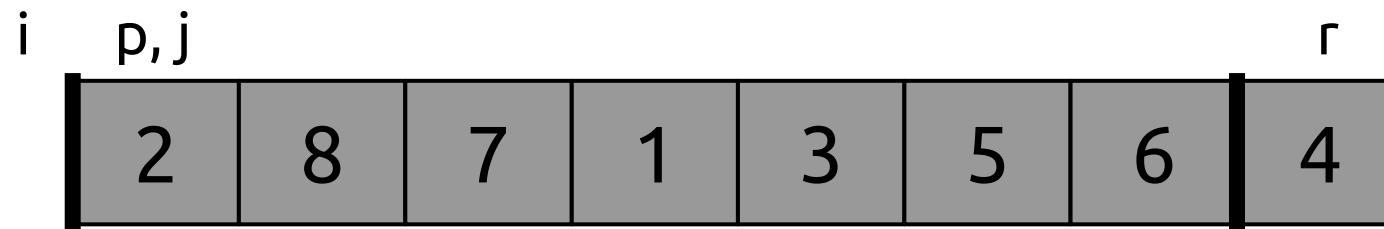
---

- ♦ Declaramos essas propriedades como um invariante de loop.
- ♦ No início de cada iteração do loop das **linhas 3-6**, para qualquer índice **k** de array:
  1. **if**  $p \leq k \leq i$ , then  $A[k] \leq x$ .
  2. **if**  $i + 1 \leq k \leq j - 1$ , then  $A[k] > x$ .
  3. **if**  $k = r$ , then  $A[k] = x$ .

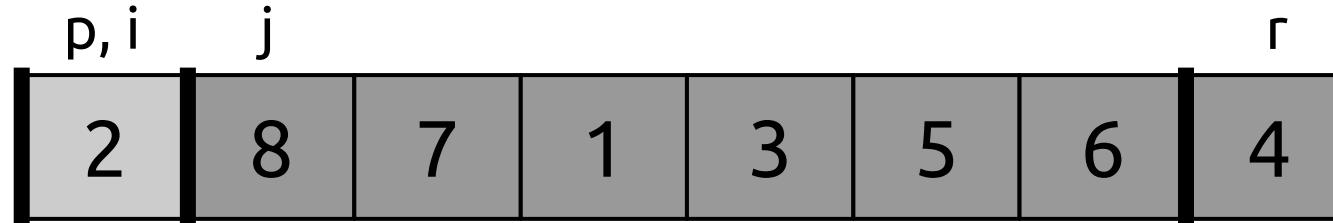
# O Algoritmo Quicksort

---

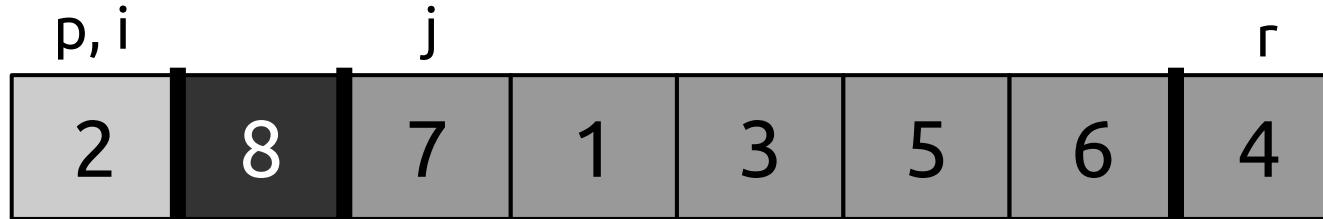
(a)



(b)



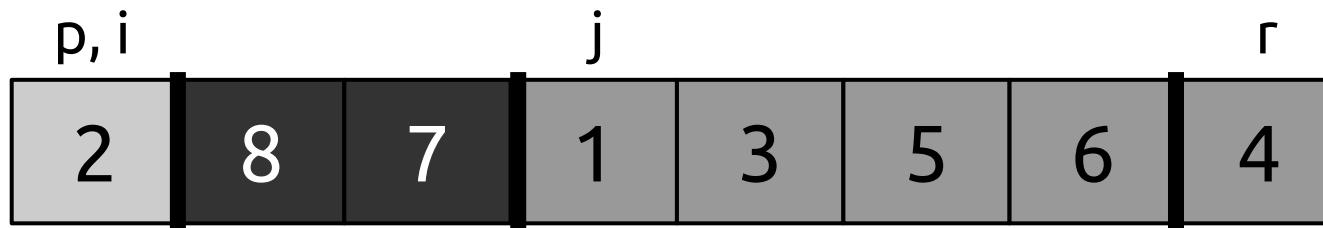
(c)



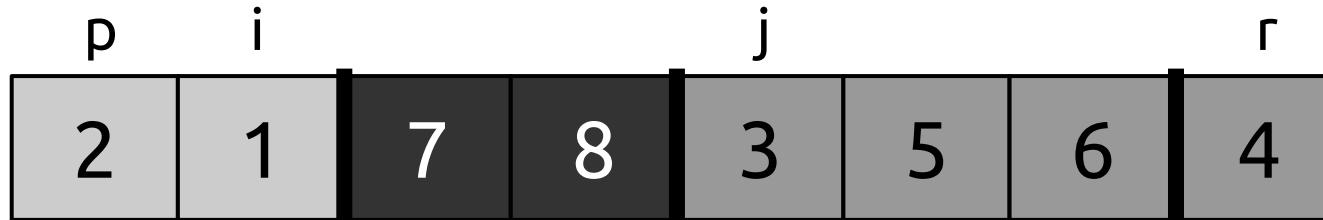
# O Algoritmo Quicksort

---

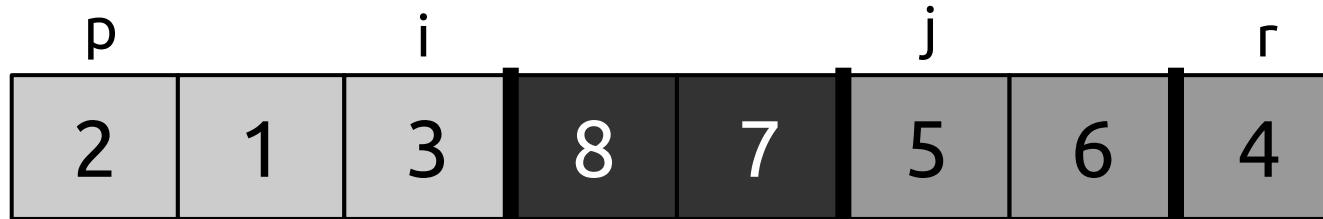
(d)



(e)

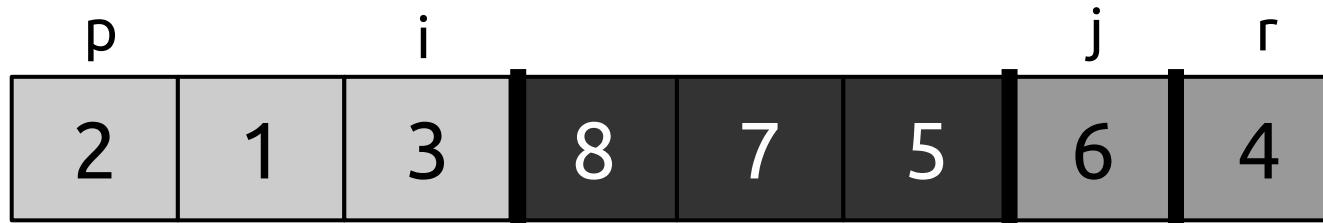


(F)

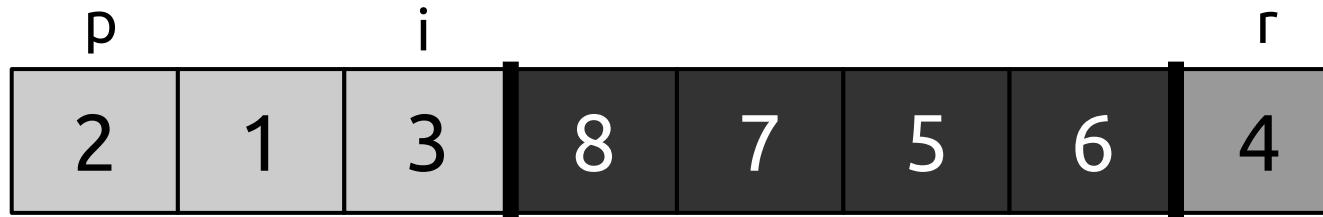


# O Algoritmo Quicksort

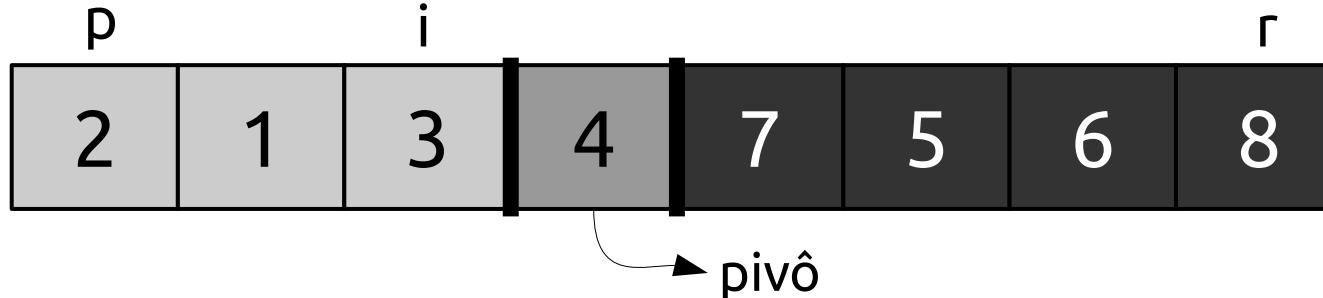
(g)



(h)



(i)



# O Algoritmo Quicksort

---

- ♦ **Figura:** A operação de **PARTITION** em um array de amostra. A entrada de array  $A[r]$  torna-se o pivô elemento  $x$ . Os elementos do array levemente sombreados estão todos na primeira partição com valores **não maiores** que  $x$ . Elementos fortemente sombreados estão na segunda partição com valores **maiores** que  $x$ . Os elementos sem sombra ainda não foram colocados em uma das duas primeiras partições e o não-sombreado final é o **pivô  $x$** .

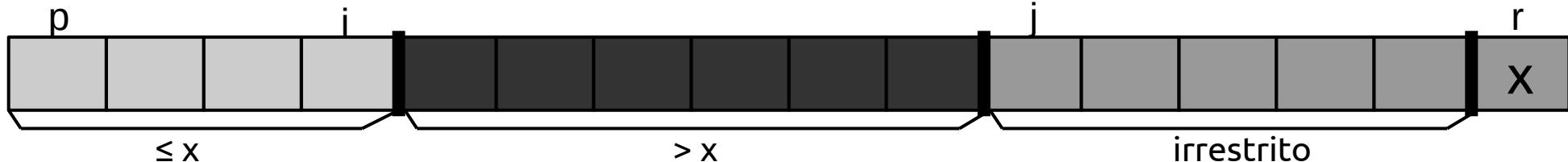
# O Algoritmo Quicksort

---

- ♦ **Figura:** (a) O array inicial e as configurações de variáveis. Nenhum dos elementos foi colocado em nenhuma das duas primeiras partições. (b) O valor 2 é “trocado consigo mesmo” e colocado na partição de valores menores. (c) - (d) Os valores 8 e 7 são adicionados à partição de valores maiores. (e) Os valores 1 e 8 são trocados e a partição menor cresce. (f) Os valores 3 e 7 são trocados e a partição menor cresce. (g) - (h) A partição maior cresce para incluir 5 e 6, e o loop termina. (i) Nas linhas 7–8, o elemento pivô é trocado para que fique entre as duas partições.

# O Algoritmo Quicksort

- Os índices entre  $j$  e  $r - 1$  não são cobertos por nenhum dos três casos, e os valores nessas entradas não têm relação particular com o pivô  $x$ .
- Precisamos mostrar que esse **invariante de loop** é verdadeira antes da primeira iteração, que cada iteração do loop mantém a invariante e que a invariante fornece uma propriedade útil para mostrar a exatidão quando o loop termina.
- **Figura:** As quatro regiões mantidas pelo procedimento **PARTITION** em um subarray  $A[p \dots r]$ . Os valores em  $A[p \dots i]$  são todos menores ou iguais a  $x$ , os valores em  $A[i + 1 \dots j - 1]$  são todos maiores que  $x$ , e  $A[r] = x$ . O subarray  $A[j \dots r - 1]$  pode assumir qualquer valor.



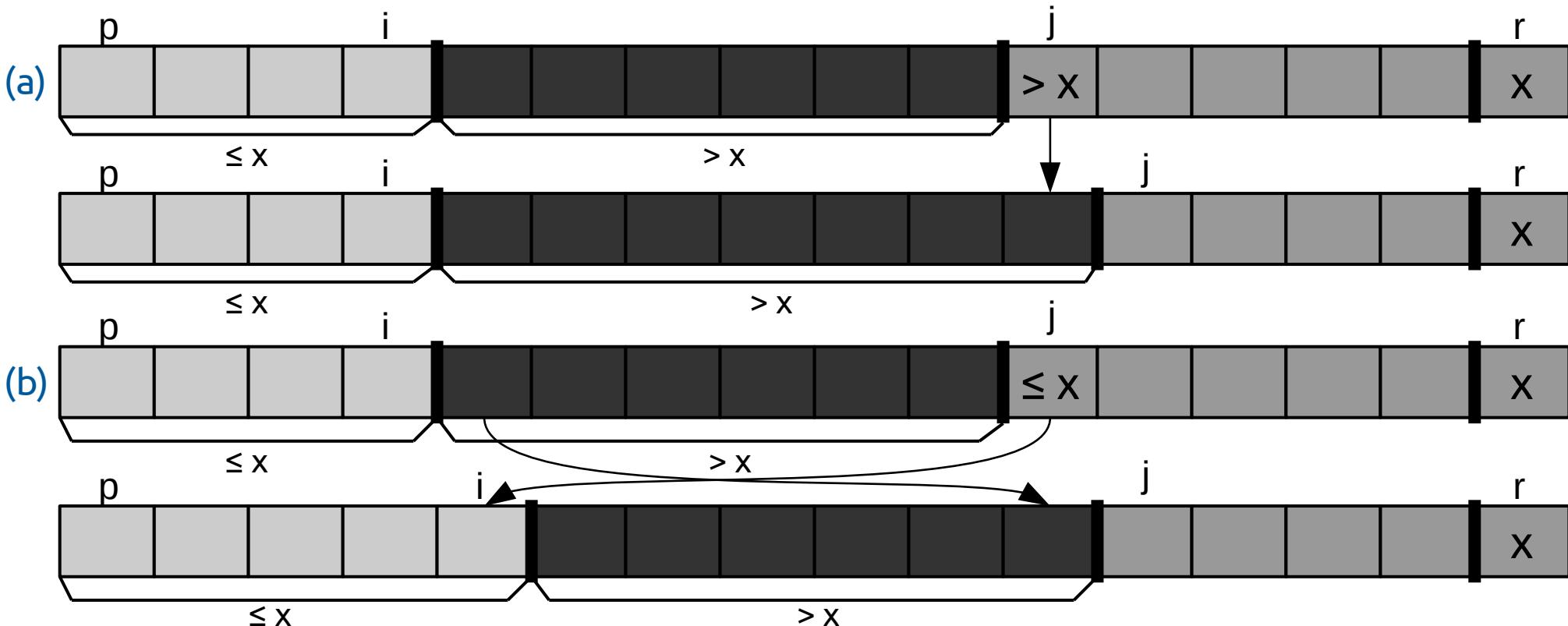
# O Algoritmo Quicksort

---

- **Inicialização:** Antes da primeira iteração do loop,  $i = p - 1$  e  $j = p$ . Como nenhum valor está entre  $p$  e  $i$  e nenhum valor está entre  $i + 1$  e  $j - 1$ , as duas primeiras condições da invariante do loop são trivialmente satisfeitas. A atribuição na linha 1 satisfaz a terceira condição.
- **Manutenção:** Como mostra a figura a seguir, consideramos dois casos, dependendo do resultado do teste na linha 4. (a) mostra o que acontece quando  $A[j] > x$ ; a única ação no loop é incrementar  $j$ . Depois que  $j$  é incrementado, a condição 2 é válida para  $A[j - 1]$  e todas as outras entradas permanecem inalteradas. (b) mostra o que acontece quando  $A[j] \leq x$ ; o loop incrementa  $i$ , troca  $A[i]$  e  $A[j]$  e, a seguir, incrementa  $j$ . Por causa da troca, agora temos aquele  $A[i] \leq x$ , e a condição 1 é satisfeita. Da mesma forma, também temos que  $A[j - 1] > x$ , uma vez que o item que foi trocado em  $A[j - 1]$  é, pela invariante do loop, maior que  $x$ .

# O Algoritmo Quicksort

- Os dois casos para uma iteração do procedimento **PARTITION**. (a) if  $A[j] > x$ , a única ação é incrementar  $j$ , o que mantém a invariante do loop. (b) if  $A[j] \leq x$ , o índice  $i$  é incrementado,  $A[i]$  e  $A[j]$  são trocados, e então  $j$  é incrementado. Novamente, a invariante do loop é mantida.



# O Algoritmo Quicksort

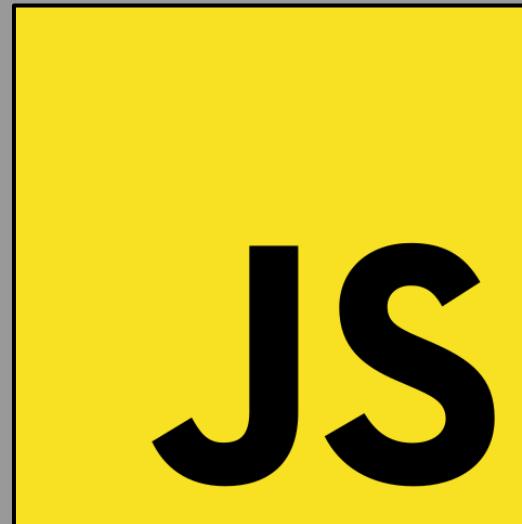
---

- **Término:** No término,  $j = r$ . Portanto, cada entrada no array está em um dos três conjuntos descritos pela invariante, e particionamos os valores no array em **três conjuntos**: aqueles menores ou iguais a  $x$ , aqueles maiores que  $x$ , e um conjunto único contendo  $x$ .
- As duas linhas finais de **PARTITION** terminam trocando o elemento pivô com o elemento mais à esquerda maior que  $x$ , movendo assim o pivô em seu lugar correto no array particionado e, em seguida, retornando o novo índice do pivô. O output de **PARTITION** agora satisfaz as especificações fornecidas para a etapa de divisão. Na verdade, ele satisfaz uma condição um pouco mais forte: após a linha 2 de **QUICKSORT**,  $A[q]$  é estritamente menor que todos os elementos de  $A[q + 1 \dots r]$ .
- O tempo de execução de **PARTITION** no subarray  $A[p \dots r]$  é  $\Theta(n)$ , onde  $n = r - p + 1$ .

# O Algoritmo Quicksort

- ♦ A seguir temos o procedimento **PARTITION** implementado na linguagem **JavaScript**:

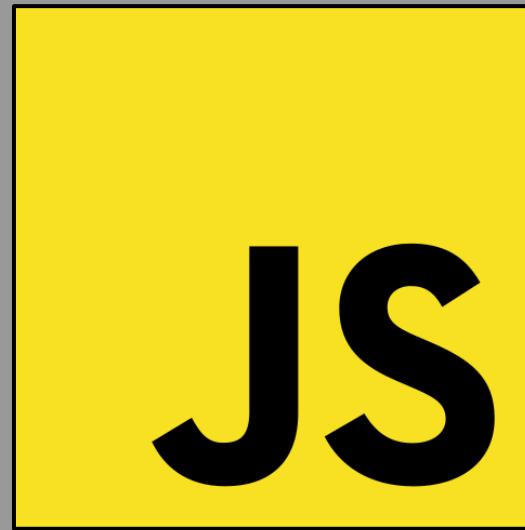
```
function partition(arr, p, r){  
    const pivot = arr[r];  
    let i = p;  
    for (let j = p; j < r; j++) {  
        if (arr[j] < pivot) {  
            [arr[j], arr[i]] = [arr[i], arr[j]];  
            i++;  
        }  
    }  
    [arr[i], arr[r]] = [arr[r], arr[i]]  
    return i;  
};
```



# O Algoritmo Quicksort

- E finalmente o procedimento que implementa **Quicksort**:

```
function quickSort(arr, p, r) {  
    if (p >= r) {  
        return;  
    }  
  
    let i = partition(arr, p, r);  
  
    quickSort(arr, p, i - 1);  
    quickSort(arr, i + 1, r);  
}  
  
array = [2, 8, 7, 1, 3, 5, 6, 4]  
quickSort(array, 0, array.length - 1)  
console.log(array) // [ 1, 2, 3, 4, 5, 6, 7, 8 ]
```



# O Algoritmo Quicksort

---

- Análise de **pior caso** do Quicksort: A partição mais desequilibrada ocorre quando uma das sublistas retornadas pela rotina de particionamento é de tamanho  $n - 1$ . Isso pode ocorrer se o pivô for o menor ou maior elemento na lista, ou em algumas implementações (por exemplo, o esquema de partição Lomuto) quando todos os elementos são iguais. Se isso acontecer repetidamente em cada partição, cada chamada recursiva processará uma lista de tamanho um a menos que a lista anterior. Conseqüentemente, podemos fazer  $n - 1$  chamadas aninhadas antes de chegarmos a uma lista de tamanho 1. Isso significa que a árvore de chamadas é uma cadeia linear de  $n - 1$  chamadas aninhadas. A enésima chamada faz trabalho  $O(n - i)$  para realizar a partição, e  $\sum_{i=0}^n (n - i) = O(n^2)$ , então, nesse caso, o quicksort leva  $O(n^2)$ .

# O Algoritmo Quicksort

---

- Análise de **melhor caso** do Quicksort: No caso mais equilibrado, cada vez que realizamos uma partição, dividimos a lista em duas partes quase iguais. Isso significa que cada chamada recursiva processa uma lista com metade do tamanho. Conseqüentemente, podemos fazer apenas chamadas aninhadas  $\log_2 n$  antes de chegarmos a uma lista de tamanho 1. Isso significa que a profundidade da árvore de chamadas é  $\log_2 n$ . Mas não há duas chamadas no mesmo nível da árvore de chamadas processando a mesma parte da lista original; portanto, cada nível de chamadas precisa de apenas tempo  $O(n)$  todas juntas (cada chamada tem alguma sobrecarga constante, mas como há apenas  $O(n)$  chamadas em cada nível, isso é incluído no fator  $O(n)$ ). O resultado é que o algoritmo usa apenas tempo  $O(n \log n)$ .

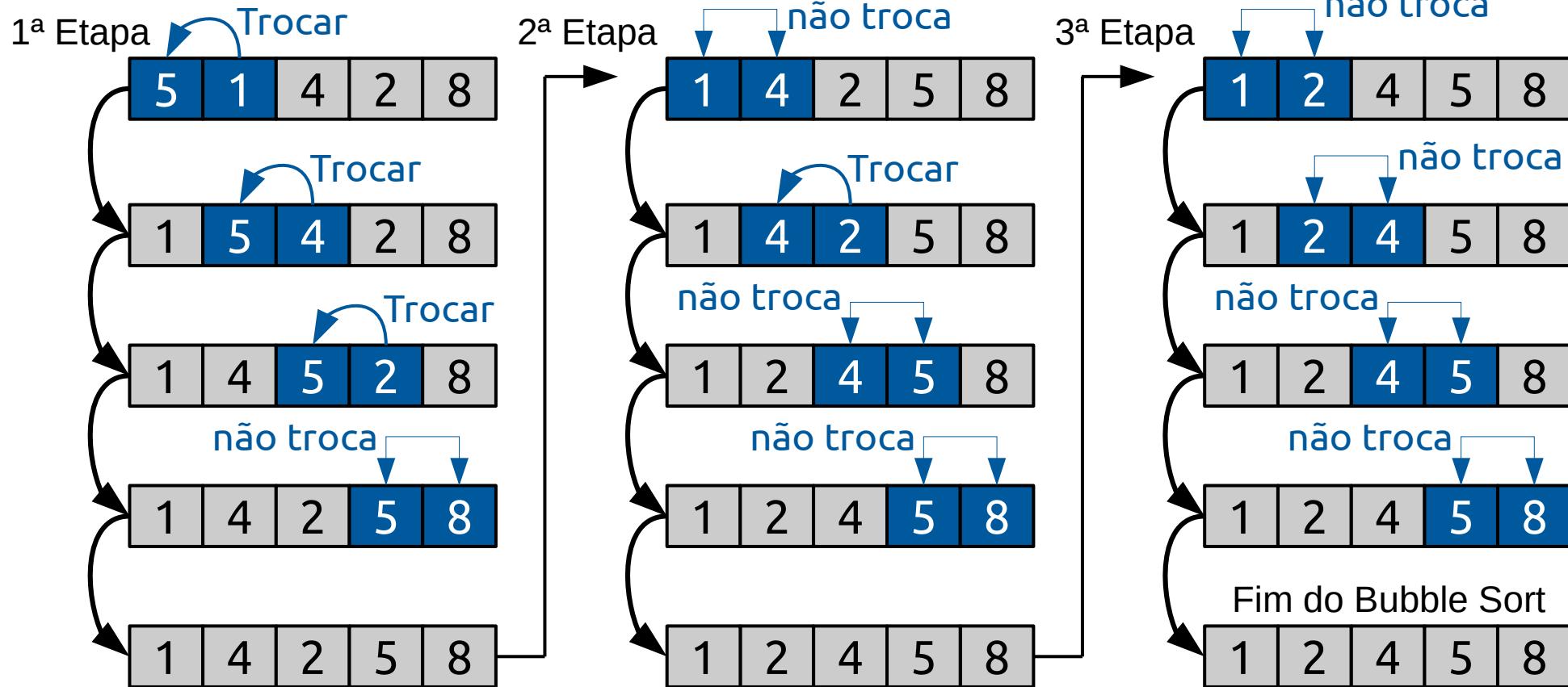
# O Algoritmo Bubble Sort

---

- O **Bubble Sort** é um algoritmo de ordenação simples que percorre repetidamente uma lista, **compara** elementos adjacentes e os **troca** se estiverem na ordem errada.
- A passagem pela lista é repetida até que a lista seja ordenada.
- O algoritmo, que é do tipo de comparação, é denominado pela maneira como os elementos menores ou maiores "borbulham" para o topo da lista.
- Este algoritmo simples tem um desempenho insatisfatório no uso no mundo real e é usado principalmente como uma ferramenta educacional.

# O Algoritmo Bubble Sort

- Funcionamento do Bubble Sort:

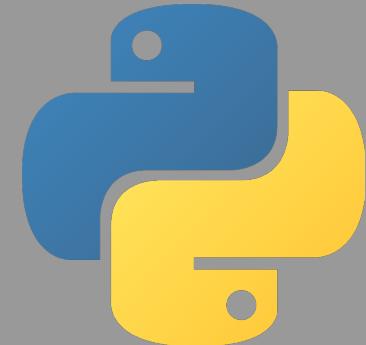


# O Algoritmo Bubble Sort

- ♦ A seguir temos um exemplo do Bubble Sort em Python:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n-1):
        for j in range(n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

arr = [5, 1, 4, 2, 8]
bubble_sort(arr)
print(arr) # [1, 2, 4, 5, 8]
```



# O Algoritmo Bubble Sort

---

- ♦ Bubble Sort tem um pior caso e uma complexidade média de  $O(n^2)$ , onde  $n$  é o número de itens sendo ordenados.
- ♦ A maioria dos algoritmos de ordenação práticos tem um pior caso substancialmente melhor ou complexidade média, geralmente  $O(n \log n)$ .
- ♦ Mesmo outros algoritmos de ordenação  $O(n^2)$ , como Insertion Sort, geralmente são executados mais rapidamente do que o Bubble Sort e não são mais complexos.
- ♦ Portanto, Bubble Sort não é um algoritmo de ordenação prático.

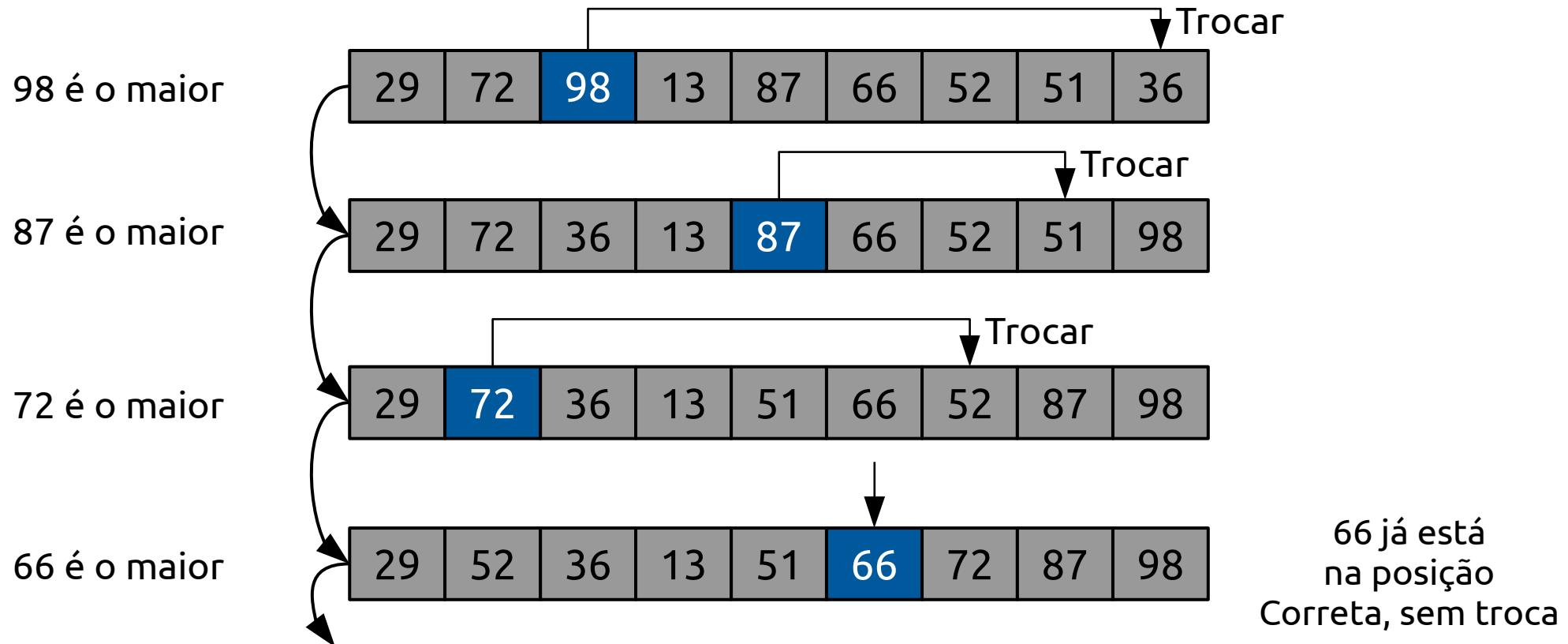
# O Algoritmo Selection Sort

---

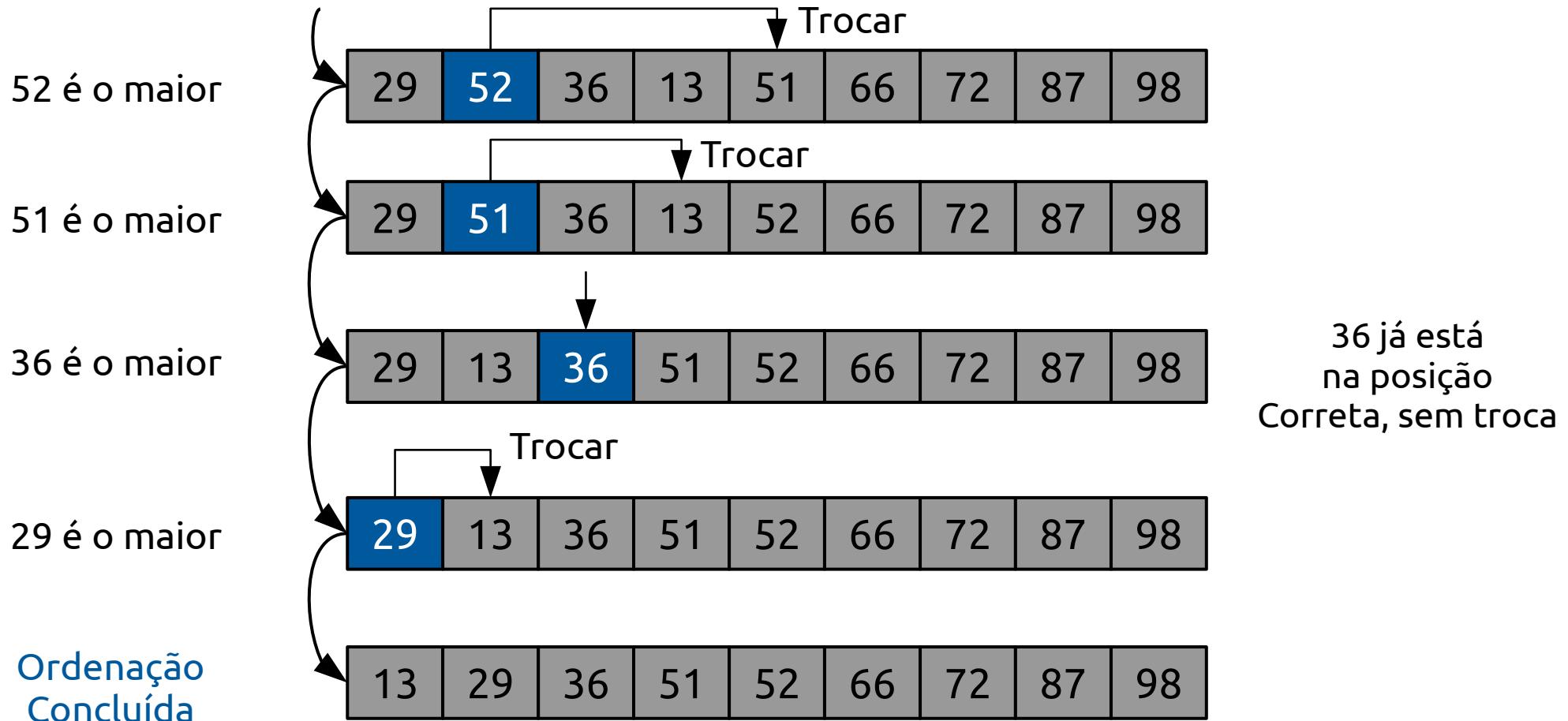
- Selection Sort é um algoritmo de ordenação por comparação **in-place**.
- Ele tem uma complexidade de tempo  **$O(n^2)$** , o que o torna ineficiente em listas grandes e geralmente tem um desempenho pior do que o seu semelhante Insertion Sort.
- O Selection Sort melhora o Bubble Sort fazendo apenas uma troca para cada passagem pela lista.
- Para fazer isso, Selection Sort procura o maior valor à medida que faz uma passagem e, após concluir a passagem, coloca-o no local adequado. Tal como acontece com Bubble Sort, após a primeira passagem, o maior item está no lugar correto. Após a segunda passagem, o próximo maior está no lugar. Esse processo continua e requer  **$n - 1$**  passagens para ordenar  $n$  itens, uma vez que o item final deve estar no lugar após a  **$(n - 1)^a$**  passagem.

# O Algoritmo Selection Sort

- A figura mostra todo o processo de ordenação. Em cada passagem, o maior item restante é selecionado e colocado em seu local apropriado.



# O Algoritmo Selection Sort



# O Algoritmo Selection Sort

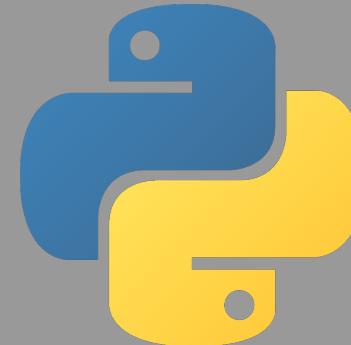
- ♦ A seguir temos um exemplo do Selection Sort em [Python](#):

```
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i

        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j

        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

```
array = [29, 72, 98, 13, 87, 66, 52, 51, 36]
selection_sort(array)
print(array) # [13, 29, 36, 51, 52, 66, 72, 87, 98]
```



# Classificação dos Algoritmos de Ordenação

---

- Os algoritmos de ordenação são frequentemente classificados por:
  - **Complexidade computacional** (pior, médio e melhor comportamento) em função do tamanho da lista  $n$ . Para algoritmos de ordenação serial típicos, o bom comportamento é  $O(n \log n)$ , com a ordenação paralela em  $O(\log_2 n)$ , e o mau comportamento é  $O(n^2)$ .
  - **Complexidade computacional de swaps** (para algoritmos "in-place").
  - **Uso de memória** (e uso de outros recursos do computador): Em particular, alguns algoritmos de ordenação são "in-place". Estritamente, uma ordenação in-place precisa apenas de  $O(1)$  memória além dos itens que estão sendo ordenados; às vezes  $O(\log n)$  memória adicional é considerada "in-place".
  - **Recursão**: Alguns algoritmos são recursivos ou não recursivos, enquanto outros podem ser ambos (por exemplo, Merge Sort).

# Classificação dos Algoritmos de Ordenação

---

- Os algoritmos de ordenação são frequentemente classificados por:
  - **Estabilidade**: algoritmos de ordenação estáveis mantêm a ordem relativa dos registros com chaves iguais (ou seja, valores).
  - Se eles são ou não de **comparação**. Uma ordenação de comparação examina os dados apenas comparando dois elementos com um operador de comparação.
  - **Adaptabilidade**: Se a pré-ordenação do input afeta ou não o tempo de execução. Algoritmos que levam isso em consideração são conhecidos por serem **adaptativos**.
  - Se o algoritmo é **serial** ou **paralelo**.

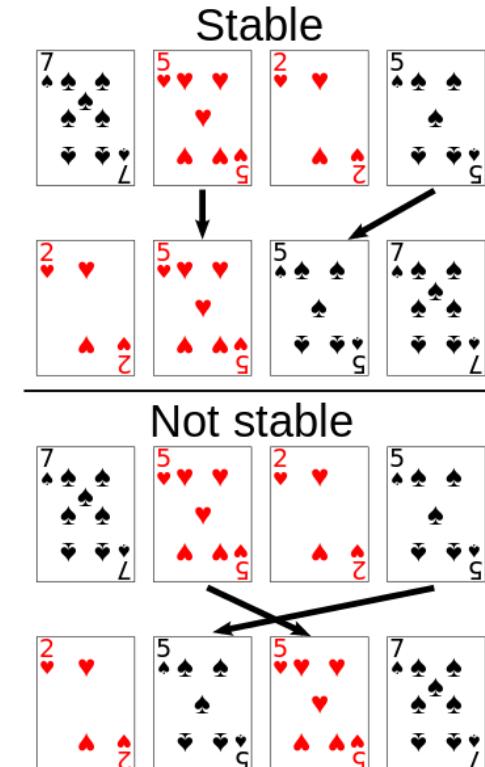
# Algoritmos In-Place

---

- Em ciência da computação, um **algoritmo in-place** é um algoritmo que transforma o input sem nenhuma estrutura de dados auxiliar.
- No entanto, uma pequena quantidade de espaço de armazenamento extra é permitida para variáveis auxiliares.
- O input é geralmente substituída pelo output à medida que o algoritmo é executado.
- Um algoritmo in-place atualiza sua sequência de input apenas por meio da substituição ou troca de elementos.

# Estabilidade

- ♦ Os algoritmos de **ordenação estável** ordenam os elementos repetidos na mesma ordem em que aparecem no input.
- ♦ Ao ordenar alguns tipos de dados, apenas parte dos dados é examinada ao determinar a ordem de ordenação. Por exemplo, no exemplo de ordenação de cartas à direita, as cartas estão sendo ordenadas por seu valor e seu naipe está sendo ignorado. Isso permite a possibilidade de várias versões diferentes ordenadas corretamente da lista original.



# Algoritmos de Busca

---

- Em ciência da computação, um **algoritmo de busca** é qualquer algoritmo que resolve o problema de busca, ou seja, para recuperar informações armazenadas em alguma estrutura de dados, ou calculadas no espaço de busca de um domínio de problema, seja com valores discretos ou contínuos. As aplicações específicas de algoritmos de busca incluem:
  - Problemas na otimização combinatória
  - Problemas na satisfação de restrição
- O algoritmo de busca apropriado geralmente depende da estrutura de dados que está sendo pesquisada e também pode incluir conhecimento prévio sobre os dados. Algumas estruturas de banco de dados são especialmente construídas para tornar os algoritmos de busca mais rápidos ou mais eficientes, como uma **search tree**, **hash map** ou um **índice de banco de dados**.

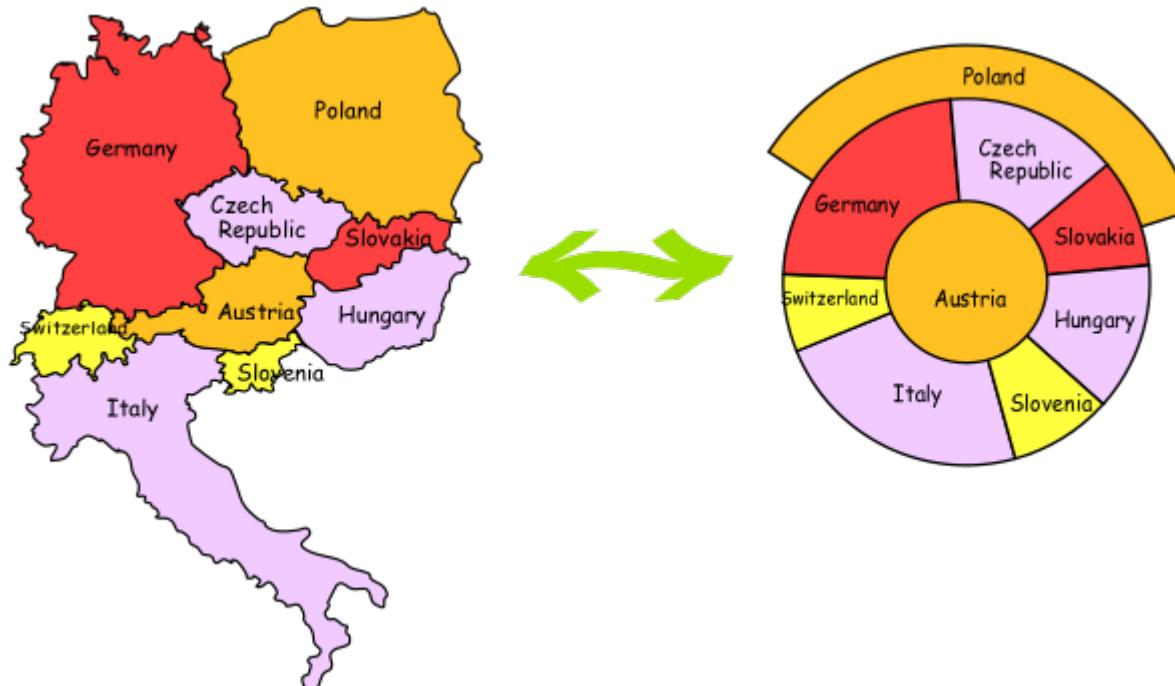
# Problemas na Otimização Combinatória

---

- ◆ Problemas na otimização combinatória, incluem:
  - O **problema de roteamento de veículos**, uma forma de problema de caminho mais curto
  - O **problema da mochila**: dado um conjunto de itens, cada um com um peso e um valor, determine o número de cada item a incluir em uma coleção de modo que o peso total seja menor ou igual a um determinado limite e o valor total seja o máximo possível.
  - O **problema de agendamento da enfermeira**

# Problemas na Satisfação de Restrição

- Problemas na satisfação de restrição, incluem:
  - O problema da coloração do mapa
  - Preenchendo um sudoku ou um quebra-cabeça de palavras cruzadas



# Problemas de Busca

---

- Outros problemas de busca incluem:
  - Na **teoria dos jogos** e especialmente na teoria combinatória dos jogos, escolher o melhor movimento a ser feito a seguir (como com o **algoritmo minmax**)
  - Encontrar uma combinação ou senha de todo o conjunto de possibilidades
  - Fatorar um número inteiro (um problema importante na **criptografia**)
  - Recuperar um registro de um banco de dados
  - Otimizar um processo industrial, como uma reação química, alterando os parâmetros do processo (como temperatura, pressão e pH)
  - Encontrar o valor máximo ou mínimo em uma lista ou array
  - Verificar se um determinado valor está presente em um conjunto de valores

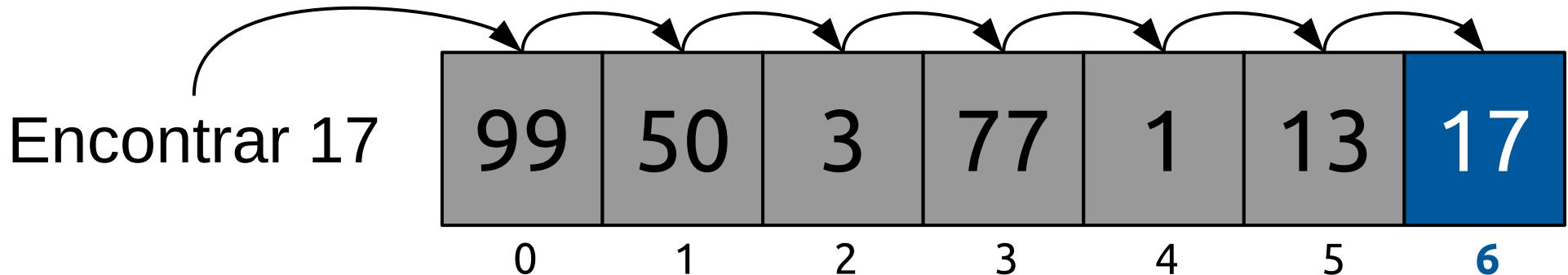
# Linear Search

---

- Na ciência da computação, uma **busca linear** ou **busca sequencial** é um método para localizar um elemento em uma lista. Ele verifica sequencialmente cada elemento da lista até que uma correspondência seja encontrada ou toda a lista tenha sido pesquisada.
- Uma pesquisa linear é executada no **pior tempo linear** e faz no máximo **n comparações**, onde **n** é o comprimento da lista.
- Se cada elemento tiver a mesma probabilidade de ser pesquisado, a pesquisa linear terá um caso médio de  **$n+1/2$**  comparações, mas o caso médio pode ser afetado se as probabilidades de pesquisa para cada elemento variar. A pesquisa linear raramente é prática porque outros algoritmos e esquemas de pesquisa, como o algoritmo de busca binária e tabelas de hash, permitem uma pesquisa significativamente mais rápida para tudo, exceto listas curtas.

# Linear Search

- Uma abordagem simples para fazer uma busca linear é:
  - Comece a partir do elemento mais à esquerda de array e, um por um, compare  $x$  (elemento a ser encontrado) com cada elemento do array.
  - Se  $x$  corresponde a um elemento, retornar o índice.
  - Se  $x$  não corresponder a nenhum dos elementos, retornar -1.

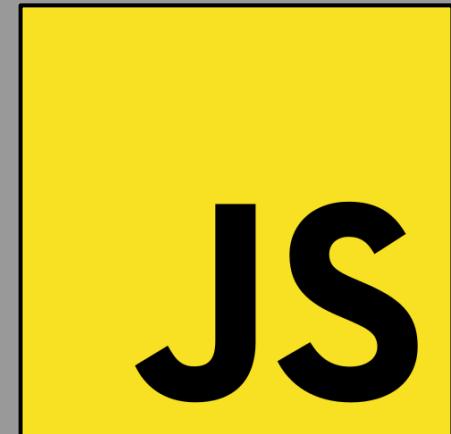


# Linear Search

- O algoritmo **Linear Search** pode ser facilmente implementado **JavaScript** ou qualquer outra linguagem:

```
function linearSearch(array, item) {  
    for(const [i, element] of array.entries()) {  
        if (element === item) {  
            return i  
        }  
    }  
    return -1  
}
```

```
let i = linearSearch([99, 50, 3, 77, 1, 13, 17], 17)  
console.log("Elemento se encontra no índice: " + i)  
// Elemento se encontra no índice: 6
```



# Binary Search

---

- Em ciência da computação, a busca binária, também conhecida como **busca de meio intervalo**, **busca logarítmica** ou **divisão binária**, é um algoritmo de busca que encontra a posição de um valor-alvo dentro de um **array ordenado**.
- A pesquisa binária compara o valor de destino com o elemento do meio do array. Se não forem iguais, a metade em que o alvo não pode estar é eliminada e a busca continua na metade restante, novamente tomando o elemento do meio para comparar com o valor alvo, e repetindo até que o valor alvo seja encontrado. Se a pesquisa terminar com a metade restante vazia, o alvo não está no array.

# Binary Search

---

- A busca binária é executada em **tempo logarítmico** no **pior caso**, fazendo  **$O(\log n)$**  comparações, onde **n** é o número de elementos no array.
- A busca binária é mais rápida do que a busca linear, exceto para pequenos arrays.
- No entanto, o array deve ser ordenado primeiro para poder aplicar a busca binária. Existem estruturas de dados especializadas projetadas para busca rápida, como **tabelas hash**, que podem ser pesquisadas com mais eficiência do que a busca binária.
- Porém, a pesquisa binária pode ser usada para resolver uma gama mais ampla de problemas, como encontrar o próximo menor ou o próximo maior elemento no array em relação ao destino, mesmo se estiver ausente do array.

# Binary Search

- ♦ Vejamos um exemplo ilustrado da busca binária:

Buscar 45	0	1	2	3	4	5	6	7	8
	6	12	17	23	38	45	77	84	90

	Low	High	Mid	
#1	0	8	4	$\text{Mid} = \frac{\text{Low} + \text{High}}{2}$

0 1 2 3 4 5 6 7 8

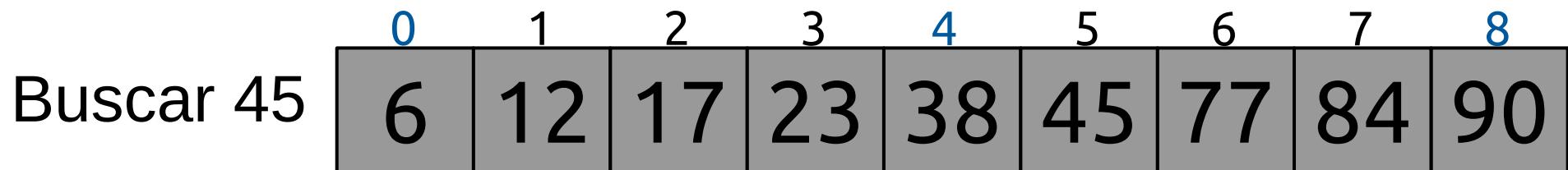
6	12	17	23	38	45	77	84	90
---	----	----	----	----	----	----	----	----

Low                          Mid                          High

$38 < 45$   
 $\text{Low} = \text{Mid} + 1 = 5$

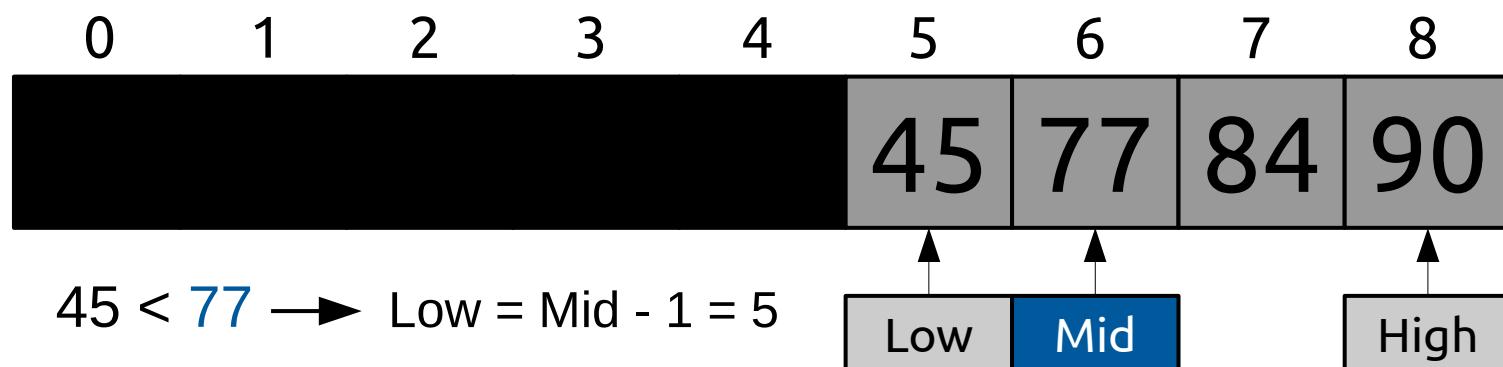
# Binary Search

- Vejamos um exemplo ilustrado da busca binária:



	Low	High	Mid
#1	0	8	4
#2	5	8	6

$$\text{Mid} = \frac{\text{Low} + \text{High}}{2}$$



# Binary Search

- Vejamos um exemplo ilustrado da busca binária:

	0	1	2	3	4	5	6	7	8
Buscar 45	6	12	17	23	38	45	77	84	90

	Low	High	Mid
#1	0	8	4
#2	5	8	6
#3	5	5	5

$$\text{Mid} = \frac{\text{Low} + \text{High}}{2}$$



45 = 45 → Busca concluída!

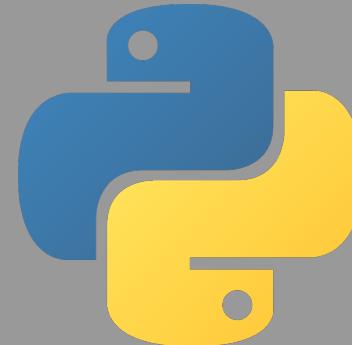
Low Mid High

# Binary Search

- A seguir temos uma versão do algoritmo Binary Search em Python:

```
def binary_search(arr,item):
    first = 0
    last = len(arr) - 1
    found = False
    while first <= last and not found:
        mid = (first + last)//2
        if arr[mid] == item:
            found = True
        else:
            if item < arr[mid]:
                last = mid - 1
            else:
                first = mid + 1
    return item, found
```

```
print(binary_search([6,12,17,23,38,45,77,84,90],45)) # (45, True)
print(binary_search([6,12,17,23,38,45,77,84,90],1)) # (1, False)
```



# Sieve of Eratosthenes

- Em matemática, a **sieve of Eratosthenes** é um algoritmo ancestral para encontrar todos os números primos até um determinado limite.

Ele faz isso marcando iterativamente como compostos (ou seja, não primos) os múltiplos de cada primo, começando com o primeiro número primo, 2. Os múltiplos de um dado primo são gerados como uma sequência de números começando daquele primo, com diferença constante entre eles que é igual a esse primo.

	Prime numbers									
	2	3	4	5	6	7	8	9	10	
11	2	3	4	5	6	7	8	9	10	
21	11	12	13	14	15	16	17	18	19	20
31	21	22	23	24	25	26	27	28	29	30
41	31	32	33	34	35	36	37	38	39	40
51	41	42	43	44	45	46	47	48	49	50
61	51	52	53	54	55	56	57	58	59	60
71	61	62	63	64	65	66	67	68	69	70
81	71	72	73	74	75	76	77	78	79	80
91	81	82	83	84	85	86	87	88	89	90
101	91	92	93	94	95	96	97	98	99	100
111	101	102	103	104	105	106	107	108	109	110
120	111	112	113	114	115	116	117	118	119	

# Sieve of Eratosthenes

---

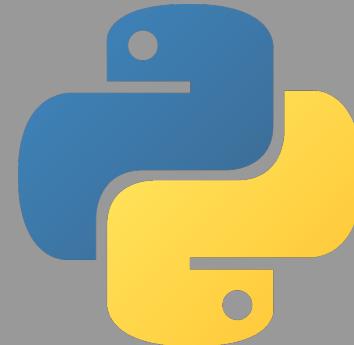
- Um **número primo** é um número natural que tem exatamente dois divisores de número naturais distintos: o número **1** e **ele mesmo**.
- Para encontrar todos os números primos menores ou iguais a um dado inteiro **n** pelo método de Eratosthenes:
  1. Crie uma lista de inteiros consecutivos de **2** a **n**: (**2, 3, 4, ..., n**).
  2. Inicialmente, faça **p** igual a **2**, o menor número primo.
  3. Enumere os múltiplos de **p** contando em incrementos de **p** de **2p** a **n** e marque-os na lista (serão **2p, 3p, 4p, ...**; o próprio **p** não deve ser marcado).
  4. Encontre o menor número na lista maior que **p** que não esteja marcado. Se esse número não existir, pare. Caso contrário, seja **p** agora igual a esse novo número (que é o próximo primo) e repita a partir do passo 3.
  5. Quando o algoritmo termina, os números restantes não marcados na lista são todos os primos abaixo de **n**.
- A ideia principal aqui é que todo valor dado a **p** será primo, porque se fosse composto, seria marcado como um múltiplo de algum outro primo menor. Observe que alguns dos números podem ser marcados mais de uma vez (por exemplo, 15 será marcado para 3 e 5).

# Sieve of Eratosthenes

- A seguir temos uma versão do Sieve of Eratosthenes em Python:

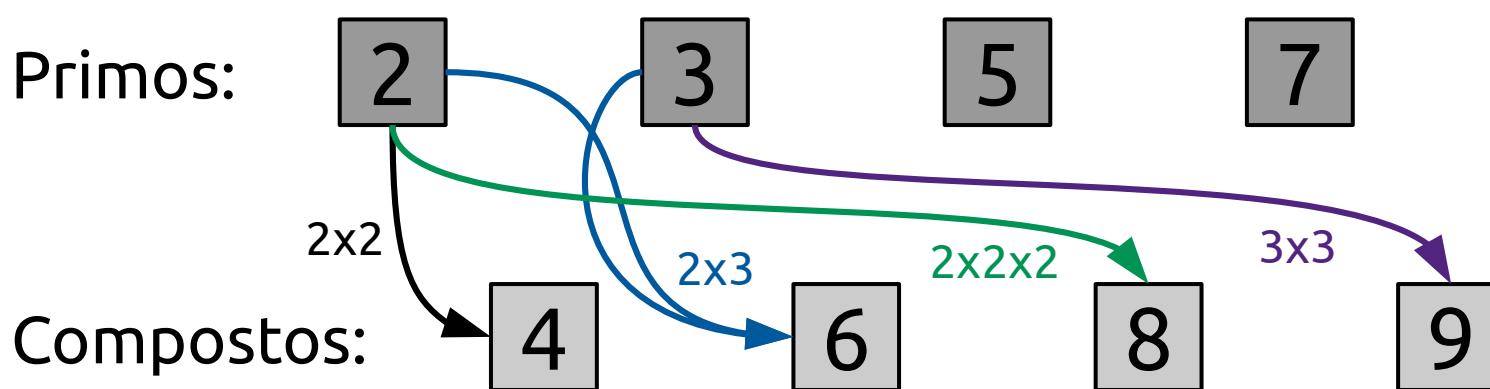
```
def primes_sieve(n):
    not_prime, primes = set(), []
    for i in range(2, n):
        if i in not_prime:
            continue
        for j in range(i*i, n, i):
            not_prime.add(j)
        primes.append(i)
    return primes

print(primes_sieve(50));
# [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```



# Fatores Primos de um Número

- Como vimos anteriormente, um Número **Primo** é um número inteiro maior que 1 que não pode ser obtido pela multiplicação de outros números inteiros.
- Os primeiros números primos são: 2, 3, 5, 7, 11, 13, 17, 19 e 23.
- Se pudermos fazê-lo multiplicando outros números inteiros, o Número é **Composto**. Por exemplo:

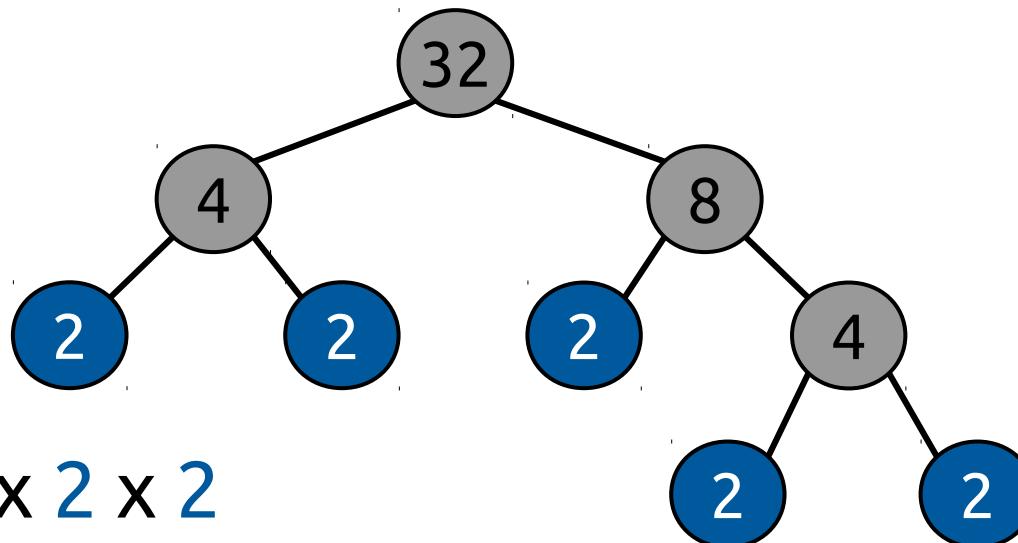


# Fatores Primos de um Número

- ♦ "Fatores" são os números que multiplicamos para obter outro número:  $2 \times 3 = 6$

2  
Fator      Fator

- ♦ Podemos definir uma árvore factorial de um número:



$$32 = 2 \times 2 \times 2 \times 2 \times 2$$

# Fatores Primos de um Número

---

- Dado um número  $n$ , vamos escrever uma função eficiente para imprimir todos os fatores primos de  $n$ .
- Por exemplo, se o número de entrada for 12, a saída deve ser “2 2 3”. E se o número de entrada for 555, o output deve ser “3 5 37”.
- A seguir estão as etapas para encontrar todos os fatores primos:
  1. Enquanto  $n$  é divisível por 2, imprima 2 e divida  $n$  por 2.
  2. Após a etapa 1,  $n$  deve ser ímpar. Agora inicie um loop de  $i = 3$  até a raiz quadrada de  $n$ . Enquanto  $i$  divide  $n$ , imprima  $i$  e divida  $n$  por  $i$ . Depois que  $i$  falhar em dividir  $n$ , incremente  $i$  por 2 e continue.
  3. Se  $n$  for um número primo e maior que 2, então  $n$  não se tornará 1 acima de duas etapas. Então imprima  $n$  se for maior que 2.

# Fatores Primos de um Número

- ♦ Implementação em Python:

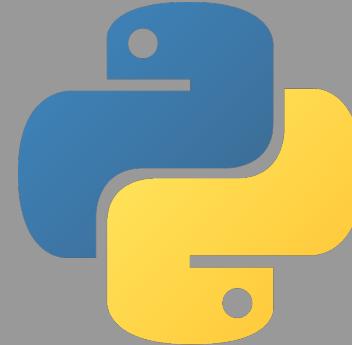
```
from math import sqrt

def prime_factors(n):
    while n % 2 == 0:
        print(2,end=" ")
        n /= 2

    for i in range(3,int(sqrt(n))+1,2):
        while n % i == 0:
            print(i,end=" ")
            n /= i

    if n > 2:
        print(int(n),end=" ")

prime_factors(555) # 3 5 37
```



# Breadth-First Search (BFS)

---

- Breadth-first search (BFS) é um algoritmo para percorrer ou buscar estruturas de dados de **árvore** ou **grafo**.
- Ele começa na raiz da árvore (ou algum nó arbitrário de um grafo, às vezes referido como uma 'chave de pesquisa') e explora todos os nós vizinhos na profundidade atual antes de passar para os nós no próximo nível de profundidade.
- Ele usa a estratégia oposta de **depth-first search**, que em vez disso explora o ramo do nó o máximo possível antes de ser forçado a retroceder e expandir outros nós.
- O BFS e sua aplicação na localização de componentes conectados de grafos foram inventados em 1945 por Konrad Zuse, em sua rejeitada tese de doutorado sobre a linguagem de programação **Plankalkül**, mas não foi publicada até 1972. Ele foi reinventado em 1959 por Edward F. Moore, que a usou para encontrar o caminho mais curto para sair de um labirinto.

# Breadth-First Search (BFS)

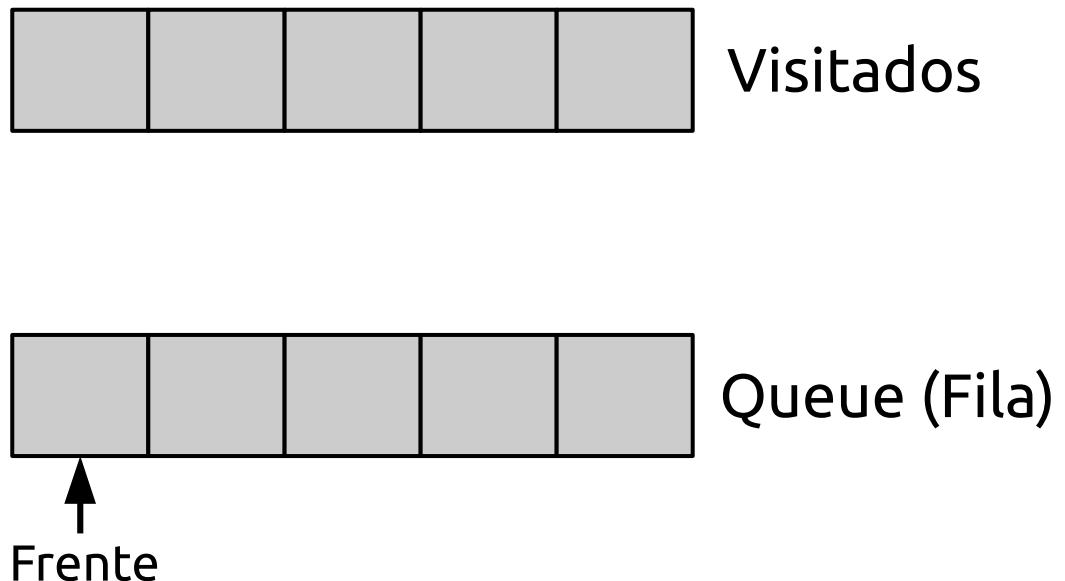
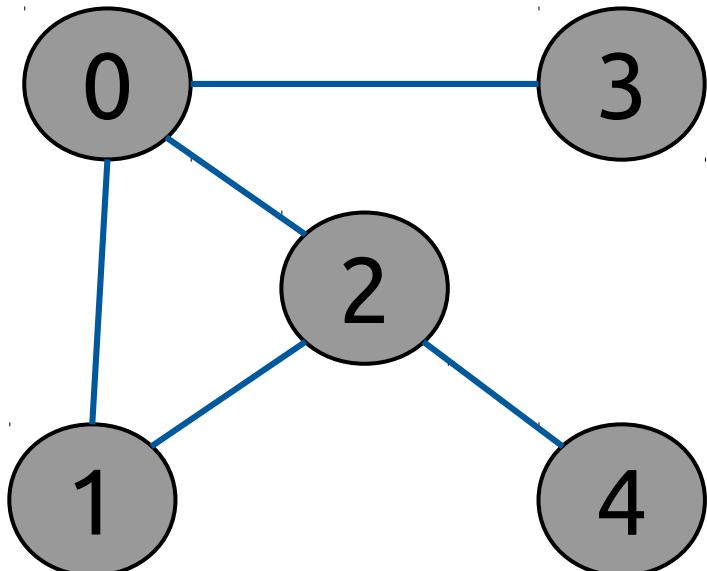
---

- Uma implementação BFS padrão coloca cada vértice do grafo em uma de duas categorias:
  1. Visitado
  2. Não Visitado
- O objetivo do algoritmo é marcar cada vértice como visitado, evitando ciclos.
- O algoritmo funciona da seguinte maneira:
  1. Comece colocando qualquer um dos vértices do grafo no final de uma fila (**queue**).
  2. Pegue o item da frente da fila e adicione-o à lista de visitados.
  3. Crie uma lista dos nós adjacentes desse vértice. Adicione aqueles que não estão na lista de visitados no final da fila.
  4. Continue repetindo as etapas 2 e 3 até que a fila esteja vazia.
- O grafo pode ter duas partes desconectadas diferentes, portanto, para ter certeza de que cobriremos todos os vértices, também podemos executar o algoritmo BFS em cada nó.

# Breadth-First Search (BFS)

- Vamos ver como o algoritmo de primeira pesquisa de amplitude funciona com um exemplo. Usamos um grafo não direcionado com 5 vértices.

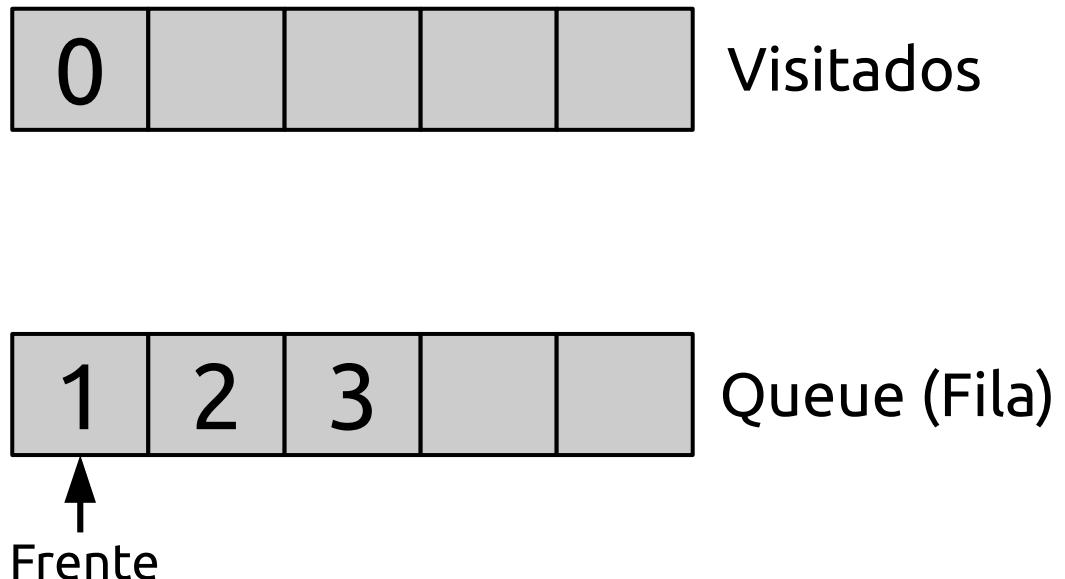
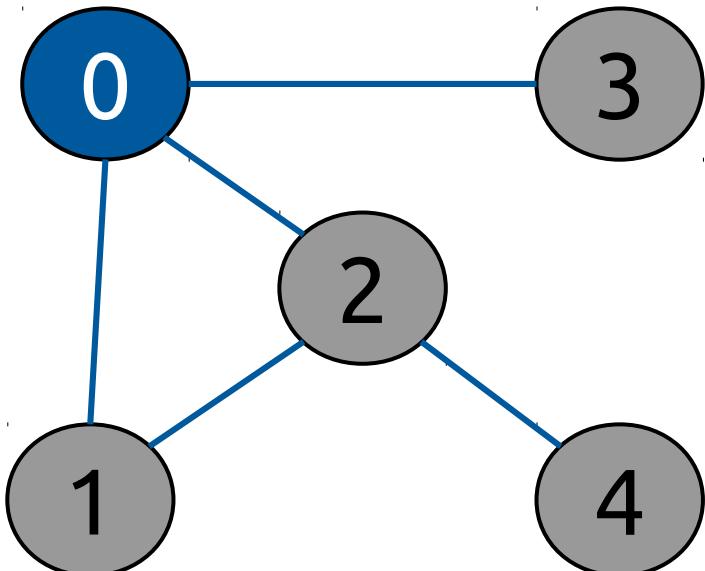
Grafo não direcionado com 5 vértices



# Breadth-First Search (BFS)

- Começamos a partir do vértice 0, o algoritmo BFS começa colocando-o na lista de Visitados e colocando todos os seus vértices adjacentes na Queue.

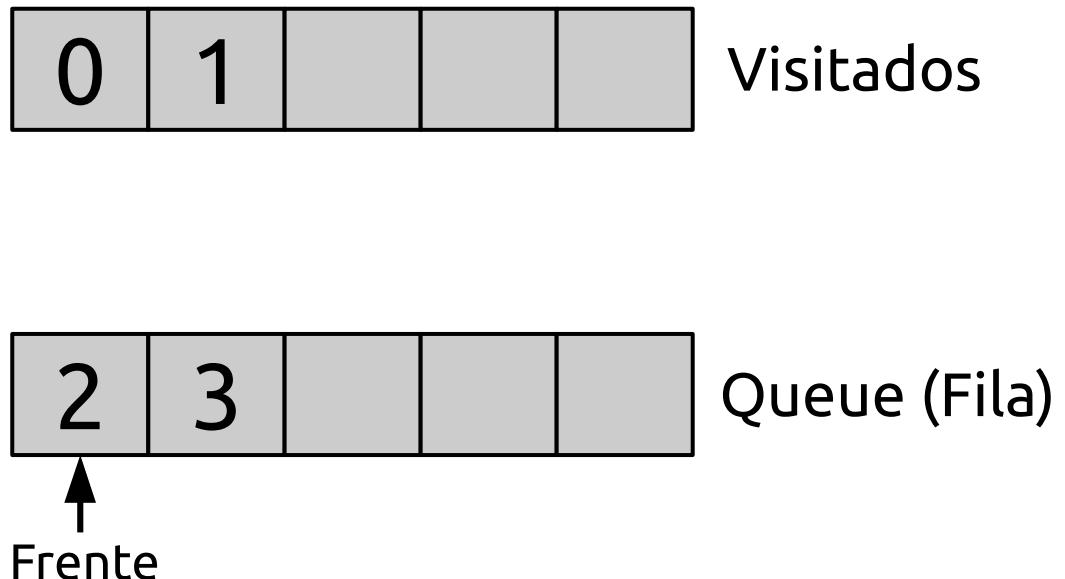
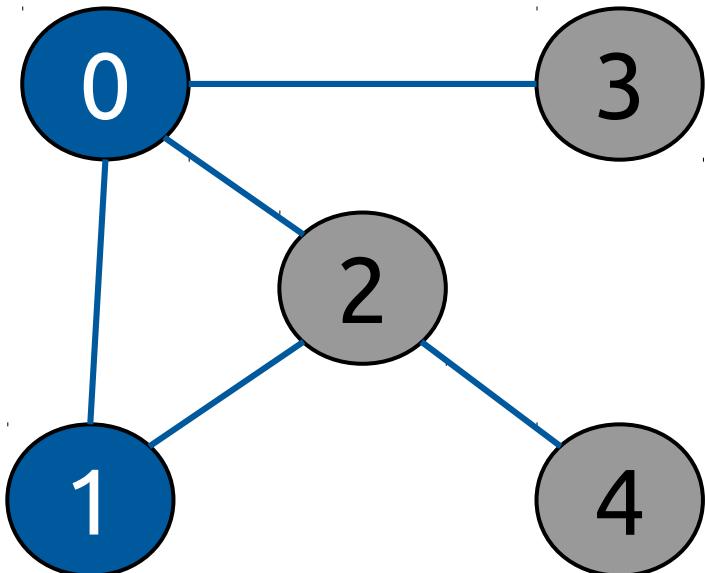
Visite o vértice inicial e adicione seus vértices adjacentes à fila



# Breadth-First Search (BFS)

- Em seguida, visitamos o elemento na frente da Queue, ou seja, 1 e vamos para seus nós adjacentes. Como 0 já foi visitado, visitamos 2 em seu lugar.

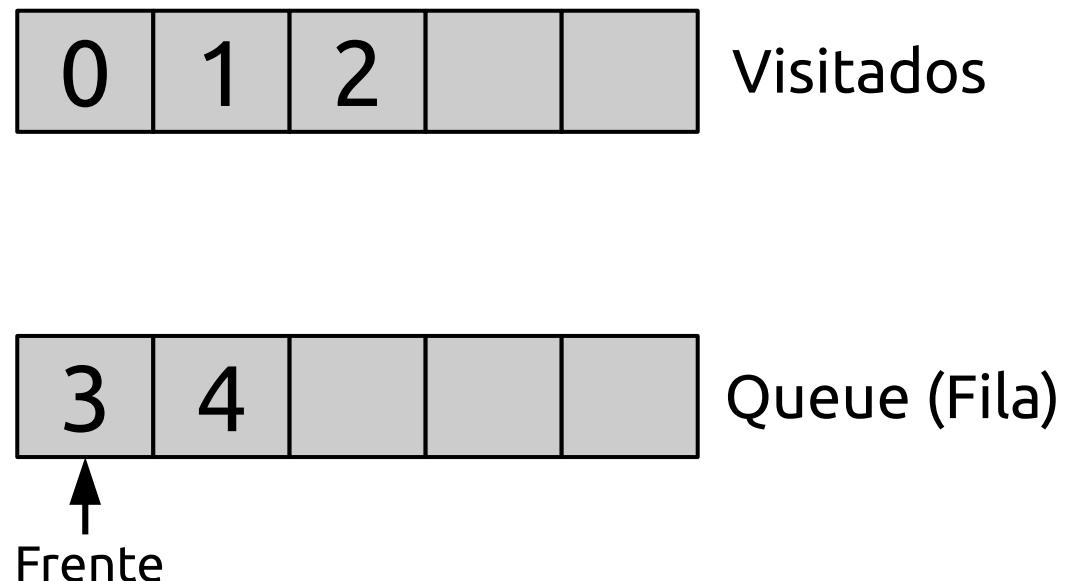
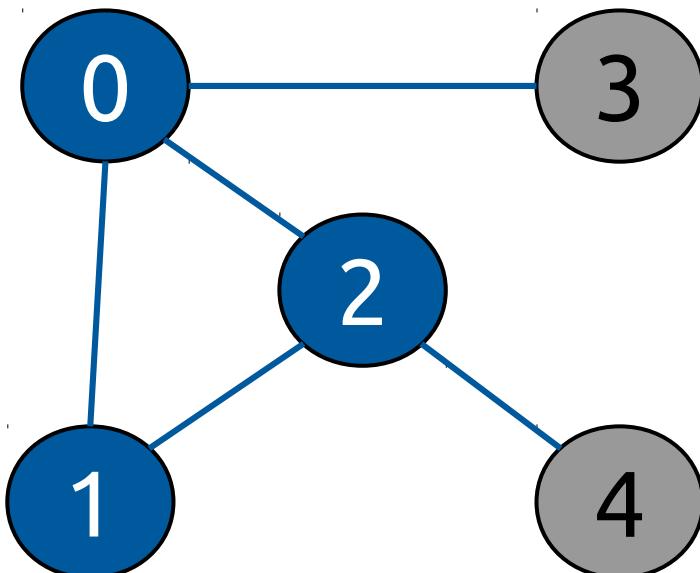
Visite o primeiro vizinho do nó inicial  
0, que é 1



# Breadth-First Search (BFS)

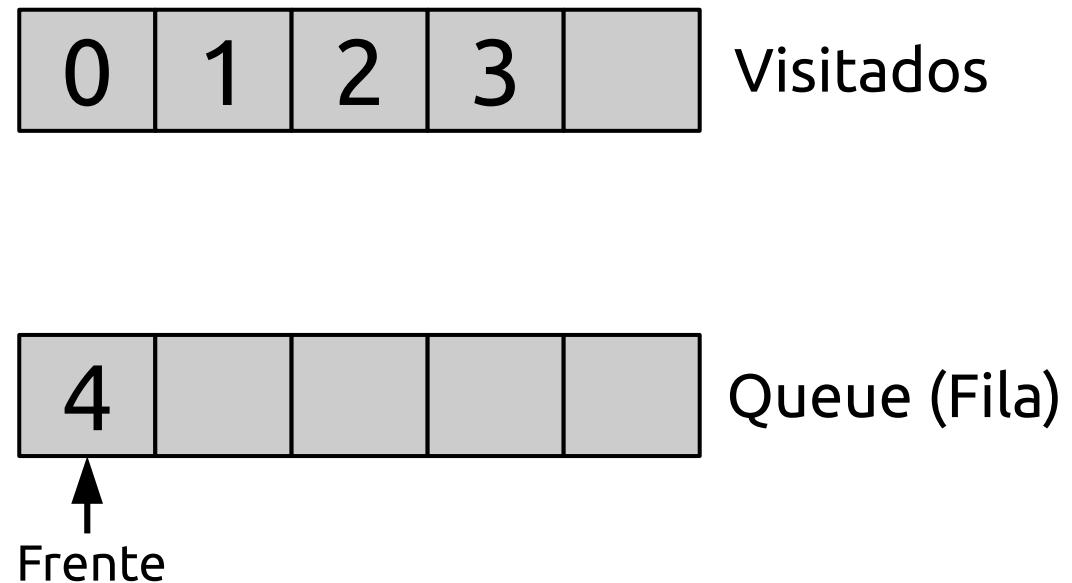
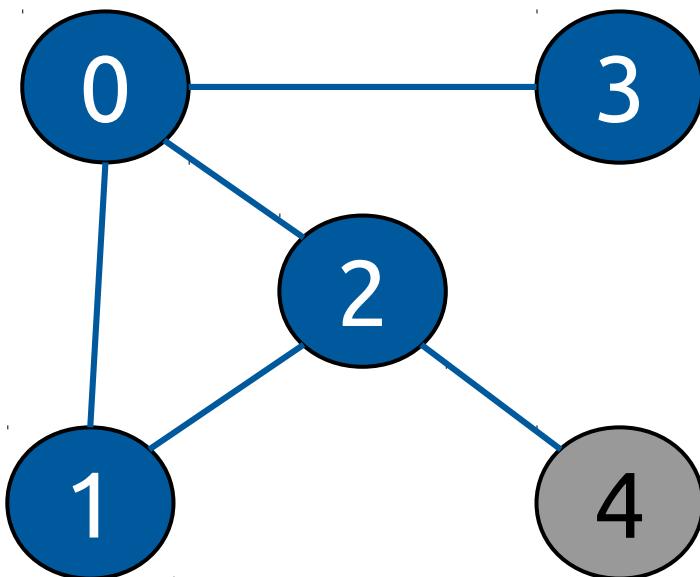
- O vértice **2** tem um vértice adjacente não visitado em **4**, portanto, adicionamos esse vértice ao final da Queue e visitamos **3**, que está na frente da Queue.

Visita 2, que foi adicionada à Queue anteriormente para adicionar seus vizinhos



# Breadth-First Search (BFS)

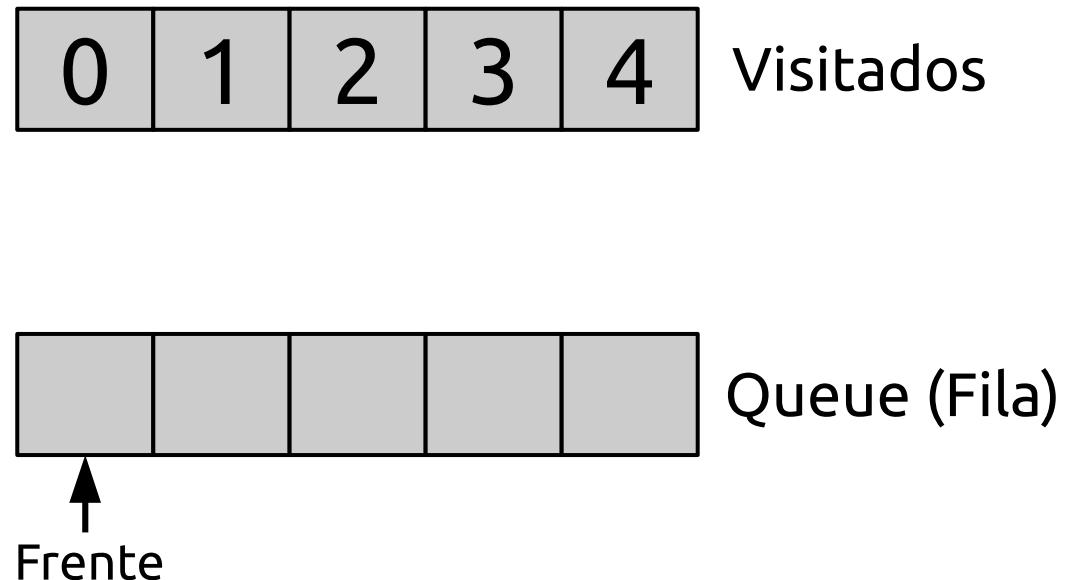
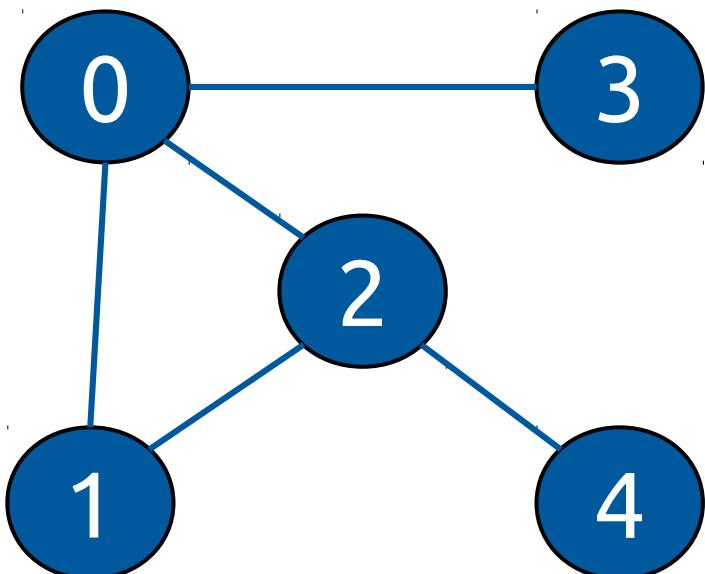
- ◆ 4 permanece na Queue:



# Breadth-First Search (BFS)

- ◆ Apenas **4** permanece na Queue uma vez que o único nó adjacente de **3**, ou seja, **0** já foi visitado. Nós o visitamos.

Como a Queue está vazia, concluímos a Breadth First Traversal do grafo.



# Breadth-First Search (BFS)

- A seguir temos uma versão do algoritmo BFS em **Python**:

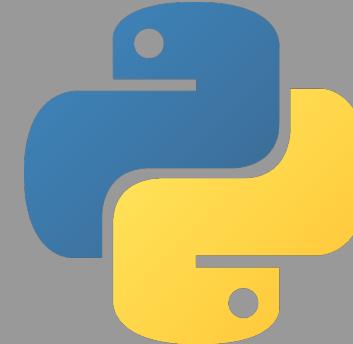
```
import collections

def bfs(graph, root):
    visited, queue = set(), collections.deque([root])
    visited.add(root)

    while queue:
        vertex = queue.popleft()
        print(str(vertex) + " ", end="")

        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)

graph = {0:[1, 2, 3], 1:[0, 2], 2:[0 ,1, 4], 3:[0], 4:[2]}
bfs(graph, 0) # 0 1 2 3 4
```



# Algoritmo de Huffman

---

- ◆ A codificação Huffman é uma técnica de compactação de dados para reduzir seu tamanho sem perder nenhum dos detalhes.
- ◆ Foi desenvolvido pela primeira vez por David Huffman.
- ◆ A codificação Huffman é geralmente útil para compactar os dados nos quais há caracteres de ocorrência frequente.
- ◆ O projeto foi publicado no artigo de 1952 intitulado “A Method for the Construction of Minimum-Redundancy Codes”.

# Algoritmo de Huffman

---

- Para entendermos como a **codificação de Huffman** funciona, suponha que a string abaixo seja enviada por uma rede.

B	C	A	A	D	D	D	C	C	A	C	A	C	A	C
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

String Inicial

- Cada caractere ocupa **8 bits**. Há um total de **15** caracteres na string acima. Portanto, um total de  $8 * 15 = 120$  bits são necessários para enviar esta string.
- Usando a técnica de codificação de Huffman, podemos comprimir a string para um tamanho menor.

# Algoritmo de Huffman

---

- A codificação de Huffman primeiro cria uma **árvore** usando as frequências do caractere e, em seguida, gera o código para cada caractere.
- Depois que os dados são codificados, eles precisam ser decodificados. A decodificação é feita usando a mesma árvore.
- A codificação Huffman evita qualquer ambigüidade no processo de decodificação usando o conceito de **código de prefixo**, isto é. um código associado a um caractere não deve estar presente no prefixo de nenhum outro código. A árvore criada auxilia na manutenção da propriedade.
- A codificação de Huffman é feita com a ajuda das etapas a seguir.

# Algoritmo de Huffman

---

1. Calcule a frequência de cada caractere na string:

Frequência da String

1	6	5	3
B	C	A	D

2. Ordene os caracteres em ordem crescente de frequência.  
Eles são armazenados em uma priority queue Q:

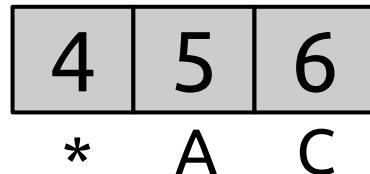
Caracteres ordenados de acordo com a frequência

1	3	5	6
B	C	A	C

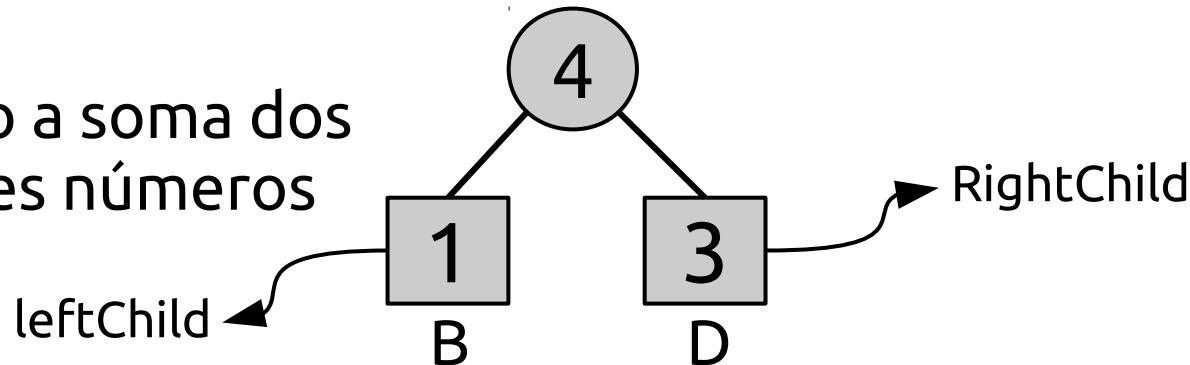
# Algoritmo de Huffman

3. Faça cada caractere único como um nó folha.

4. Crie um nó vazio z. Atribua a frequência mínima ao filho esquerdo de z e atribua a segunda frequência mínima ao filho direito de z. Defina o valor de z como a soma das duas frequências mínimas acima.



Obtendo a soma dos menores números

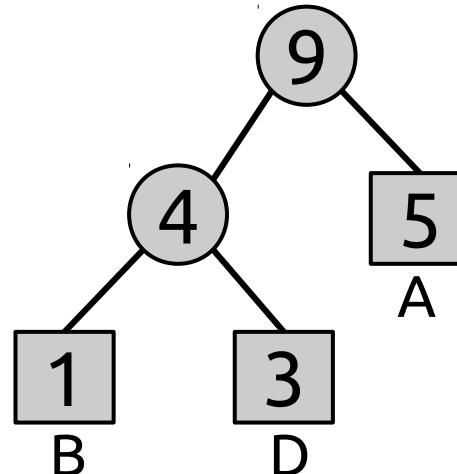


# Algoritmo de Huffman

---

5. Remova essas duas frequências mínimas de **Q** e adicione a soma à lista de frequências (\* denota os nós internos na figura anterior).
6. Insira o nó **z** na árvore.
7. Repita as etapas 3 a 5 para todos os caracteres.

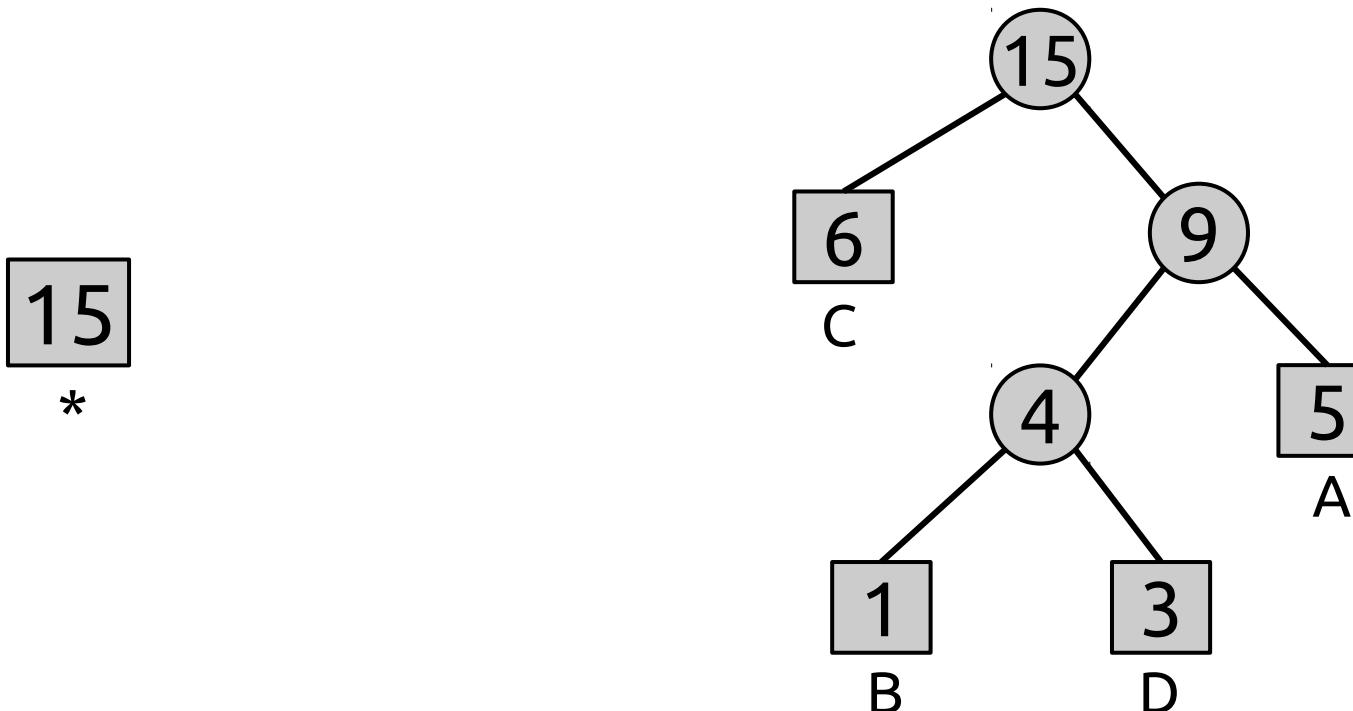
6	9
C	*



# Algoritmo de Huffman

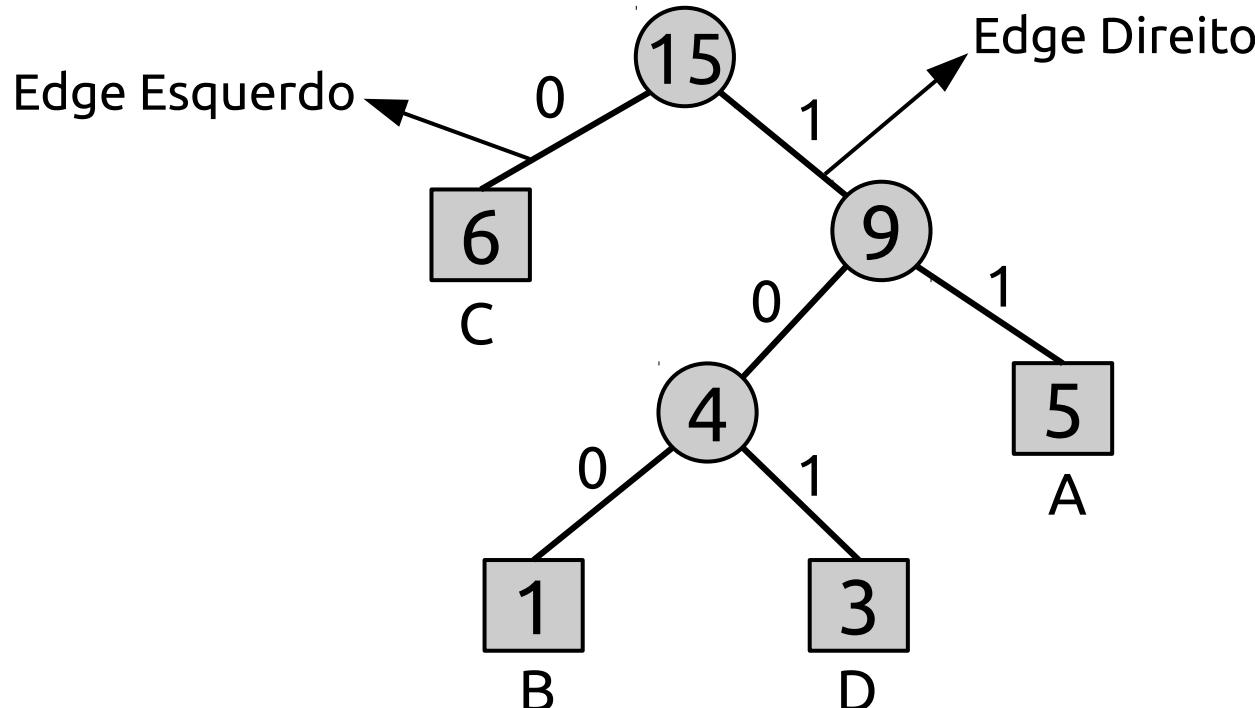
---

- ♦ Repita as etapas 3 a 5 para todos os caracteres.



# Algoritmo de Huffman

8. Para cada nó não folha, atribua 0 ao edge esquerdo e 1 ao edge direito.



# Algoritmo de Huffman

---

- Para enviar a nossa string pela rede, temos que enviar a árvore, bem como o código compactado anteriormente. O tamanho total é dado pela tabela abaixo.

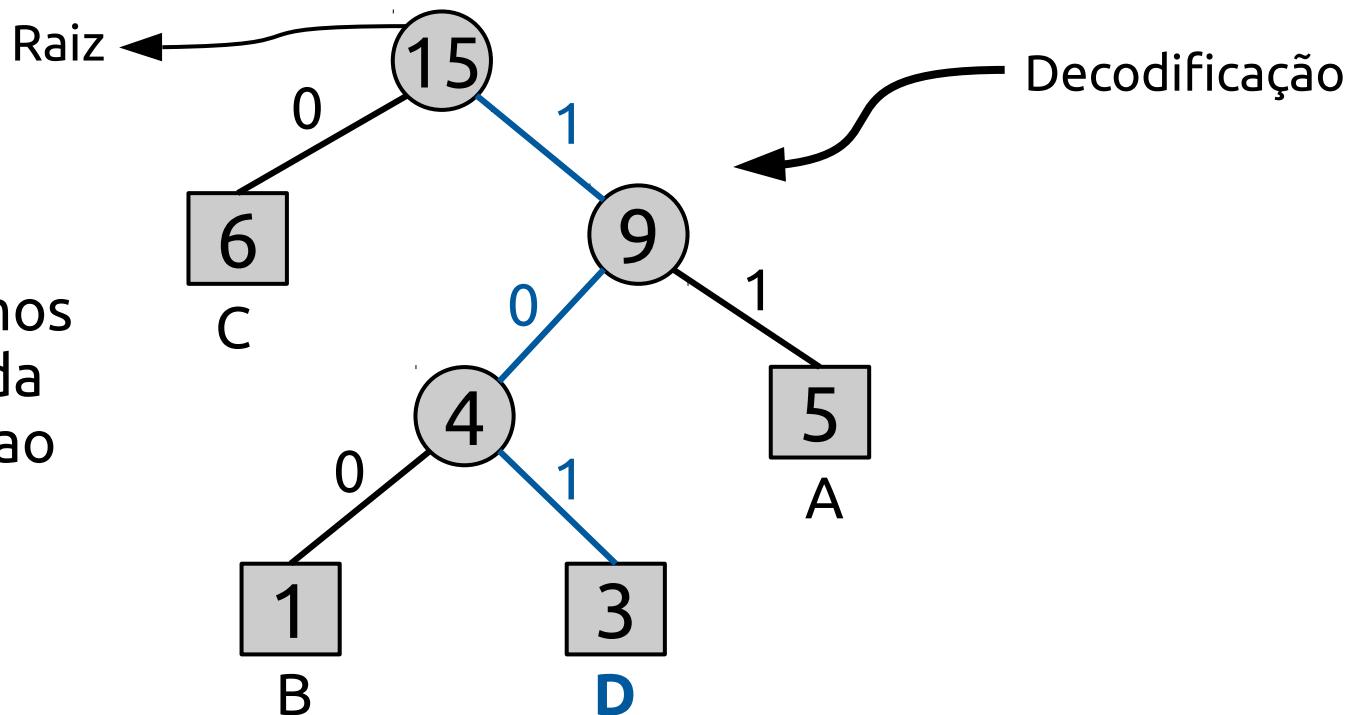
Sem codificação, o tamanho total da string era de 120 bits.  
Após a codificação, o tamanho é reduzido para  $32 + 15 + 28 = 75$ .

Caráter	Frequência	Código	Tamanho
A	5	11	$5*2 = 10$
B	1	100	$1*3 = 3$
C	6	0	$6*1 = 6$
D	3	101	$3*3 = 9$
$4 * 8 = 32$ bits	15 bits		

# Algoritmo de Huffman

- Para decodificar o código, podemos pegar o código e percorrer a árvore para encontrar o caractere.

Seja 101 para ser decodificado, podemos atravessar a partir da raiz como na figura ao lado.



# Algoritmo de Huffman

---

- ♦ Algoritmo de codificação de Huffman:

crie uma priority queue **Q** consistindo em cada caractere único.  
ordene-os então em ordem crescente de suas frequências.  
para todos os caracteres únicos:

crie um **newNode**

extraia o valor mínimo de **Q** e atribua-o a **leftChild** de **newNode**

extraí o valor mínimo de **Q** e atribuí-lo a **rightChild** de **newNode**

calcule a soma desses dois valores mínimos e atribua-o ao valor de **newNode**

insira este **newNode** na árvore

return **rootNode**

# Algoritmo de Huffman

---

- ♦ Complexidade da codificação de Huffman:
  - A complexidade de tempo para codificar cada caractere exclusivo com base em sua frequência é  $O(n \log n)$ .
  - A extração da frequência mínima da priority queue ocorre  $2 * (n - 1)$  vezes e sua complexidade é  $O(\log n)$ . Portanto, a complexidade geral é  $O(n \log n)$ .
- ♦ Aplicativos da codificação Huffman:
  - A codificação de Huffman é usada em formatos de compressão convencionais como GZIP, BZIP2, PKZIP, etc.
  - Para transmissões de texto e fax.

# Torre de Hanói

---

- ◆ O quebra-cabeça da Torre de Hanói foi inventado pelo matemático francês Edouard Lucas em 1883.



# Torre de Hanói

---

- Ele foi inspirado por uma lenda que fala de um templo hindu onde o quebra-cabeça foi apresentado a jovens sacerdotes.
- No início dos tempos, os sacerdotes recebiam **três varas** e uma **pilha de 64 discos de ouro**, cada disco um pouco menor do que o que estava embaixo.
- A tarefa deles era transferir todos os 64 discos de uma das três varas para outra, com duas restrições importantes.
- Eles só podiam mover um disco por vez e nunca poderiam colocar um disco maior em cima de um menor.
- Os sacerdotes trabalharam com muita dedicação, dia e noite, movendo um disco a cada segundo. Quando eles terminassem seu trabalho, dizia a lenda, o templo viraria pó e o mundo desapareceria.

# Torre de Hanói

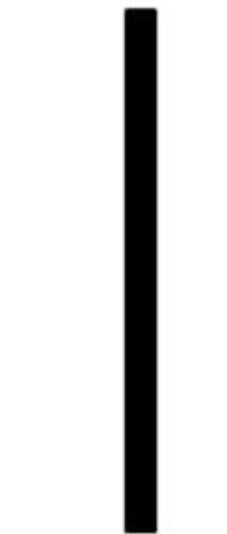
---

- ♦ O número de movimentos necessários para mover corretamente uma torre de 64 discos é  $2^{64} - 1 = 18.446.744.073.709.551.615$ .
- ♦ A uma taxa de um movimento por segundo, isso significa  $584.942.417.355$  anos!
- ♦ A seguir temos um exemplo de arranjo de discos para a Torre de Hanói, observe que, conforme as regras especificam, os discos em cada vara são empilhados de forma que os discos menores fiquem sempre em cima dos discos maiores.

# Torre de Hanói

---

- ♦ Neste caso, nosso objetivo é mover todos os discos da vara “Start” para a vara “Goal”.



# Torre de Hanói

---

- É possível resolver este problema usando **Recursão**.
- Vamos pensar sobre esse problema de baixo para cima. Suponha que você tenha uma torre de cinco discos, originalmente na vara um.
- Se você já sabe como mover uma torre de quatro discos para a vara dois, poderia facilmente mover o disco inferior para a vara três e, em seguida, mover a torre de quatro da vara dois para a vara três.
- Mas e se você não souber mover uma torre de altura quatro? Suponha que você soubesse como mover uma torre de altura três para a vara três; então seria fácil mover o quarto disco para o pino dois e mover os três do pino três para cima dele. Mas e se você não souber como mover uma torre de três? Que tal mover uma torre de dois discos para a vara dois e, em seguida, mover o terceiro disco para a vara três e, em seguida, mover a torre de altura dois em cima dele? Mas e se você ainda não souber fazer isso? Certamente você concordaria que mover um único disco para o pino três é fácil, trivial, você pode até dizer. Isso soa como um **base case** em formação.

# Torre de Hanói

---

- Aqui está um esboço de alto nível de como mover uma torre da vara de “Start” para a vara “Goal”, usando uma vara intermediária:
  1. Mova uma torre de altura-1 para uma vara intermediária, usando a vara “Goal”.
  2. Mova o disco restante para a vara “Goal”.
  3. Mova a torre de altura-1 da vara intermediária para a vara “Goal” usando a vara original.
- Contanto que sempre obedeçamos a regra de que os discos maiores permanecem na parte inferior da pilha, podemos usar as três etapas acima recursivamente, tratando quaisquer discos maiores como se eles nem estivessem lá. A única coisa que falta no esboço acima é a identificação de um base case. O problema mais simples da Torre de Hanói é uma torre de um disco. Nesse caso, precisamos mover apenas um único disco para seu destino final. Uma torre de um disco será nosso base case.

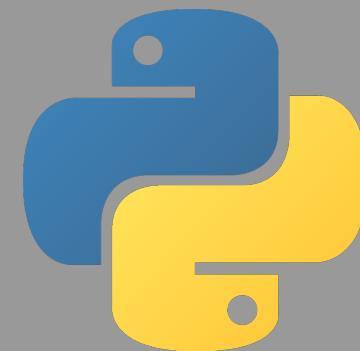
# Torre de Hanói

- ♦ O algoritmo a seguir nos fornece a solução recursiva para o problema da Torre de Hanói.

```
def move_tower(height,start,middle,goal):
    if height >= 1:
        move_tower(height-1,start,goal,middle)
        move_disk(height,start,goal)
        move_tower(height-1,middle,start,goal)

def move_disk(n,sp,gp):
    print(f"Movendo disco {n} em {sp} para {gp}")

discos = 3
move_tower(discos,"A","B","C")
```



# Torre de Hanói

---

- A chave para a simplicidade do algoritmo é que fazemos duas chamadas recursivas diferentes, uma na linha 3 e uma segunda na linha 5.
- Na linha 3, movemos todos, exceto o disco inferior da torre inicial, para uma vara intermediária.
- A próxima linha simplesmente move o disco inferior para seu local de descanso final.
- Então, na linha 5, movemos a torre da vara intermediária para o topo do disco maior.
- O base case é detectado quando a altura da torre é 0; neste caso, não há nada a fazer, então a função **move\_tower** simplesmente retorna. O importante a lembrar sobre como lidar com o base case dessa maneira é que simplesmente retornar de **move\_tower** é o que finalmente permite que a função **move\_disk** seja chamada.
- Você pode fazer experimentos com a Torre de Hanói neste game interativo:  
<https://www.mathsisfun.com/games/towerofhanoi.html>

# Paradoxo do Aniversário

---

- ◆ Quantas pessoas devem estar em uma sala para fazer a probabilidade de 100% de que pelo menos duas pessoas na sala façam aniversário no mesmo dia?
- ◆ **Resposta:** 367 (já que existem 366 aniversários possíveis, incluindo 29 de fevereiro).
- ◆ A pergunta acima era simples, vejamos esta: Quantas pessoas devem estar em uma sala para ocorrer a probabilidade de 50% de que pelo menos duas pessoas na sala façam aniversário no mesmo dia?
- ◆ **Resposta:** 23.
- ◆ O número é surpreendentemente muito baixo. Na verdade, precisamos de apenas 70 pessoas para fazer a probabilidade de 99,9%.
- ◆ Vamos discutir a fórmula generalizada.

# Paradoxo do Aniversário

---

- ♦ Qual é a probabilidade de que duas pessoas entre  $n$  façam aniversário no mesmo dia?
- ♦ Suponha que a probabilidade de que duas pessoas em uma sala com  $n$  tenham o mesmo aniversário seja  $P(\text{mesmo})$ .
- ♦  $P(\text{mesmo})$  pode ser facilmente avaliado em termos de  $P(\text{diferente})$ , onde  $P(\text{diferente})$  é a probabilidade de que todos tenham aniversário diferente.
- ♦  $P(\text{mesmo}) = 1 - P(\text{diferente})$
- ♦  $P(\text{diferente})$  pode ser escrito como  $1 \times (364/365) \times (363/365) \times (362/365) \times \dots \times (1 - (n - 1)/365)$

# Paradoxo do Aniversário

---

- ♦ A questão é: como obtivemos a expressão anterior?
- ♦ Pessoas do primeiro ao último podem obter aniversários na seguinte ordem, para que todos os aniversários sejam distintos:
  - A primeira pessoa pode fazer aniversário entre 365.
  - A segunda pessoa deve ter um aniversário diferente da primeira pessoa.
  - A terceira pessoa deve ter um aniversário diferente das duas primeiras.
  - ...
  - A enésima pessoa deve ter um aniversário diferente de qualquer uma das pessoas anteriormente consideradas ( $n-1$ ).

# Paradoxo do Aniversário

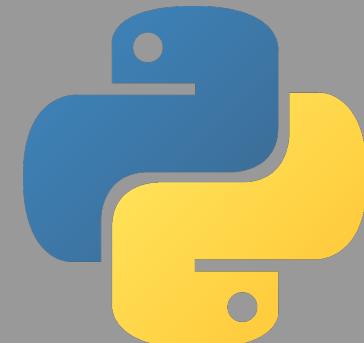
---

- A expressão anterior pode ser aproximada usando a [Série de Taylor](#).
- A seguir temos um programa para aproximar o número de pessoas para uma determinada probabilidade.

```
from math import sqrt, log, ceil

def calcular(p):
    return ceil(sqrt(2*365*log(1/(1-p)))) 

print(calcular(0.50)) # 23
```



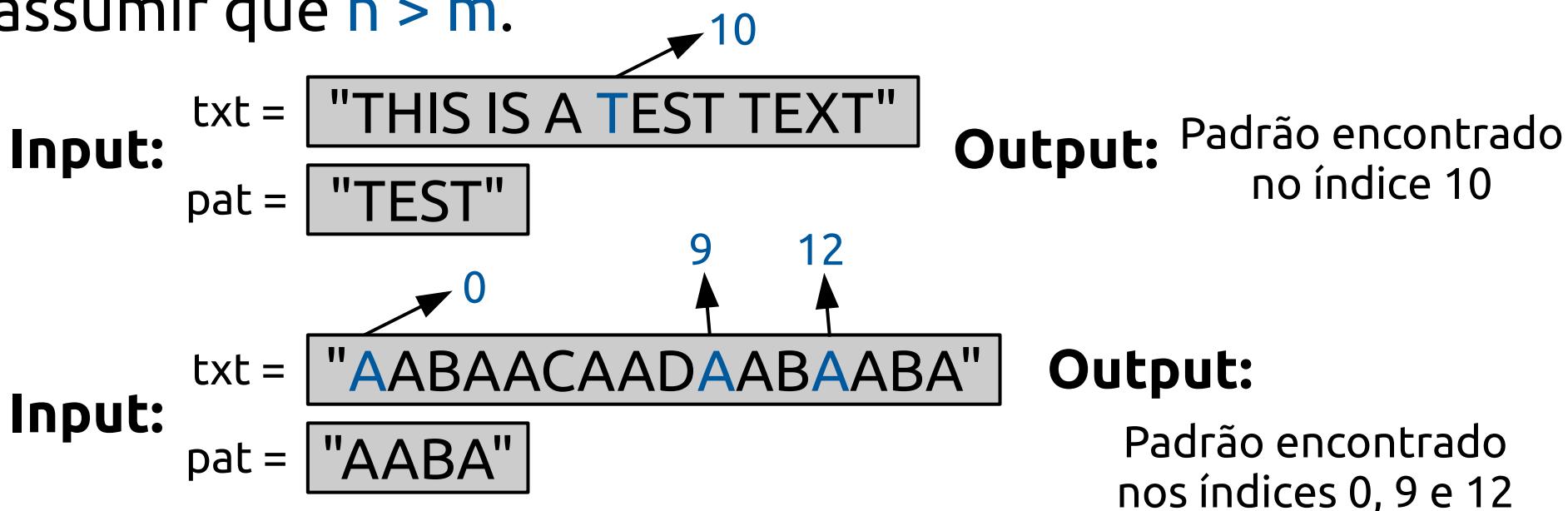
# O Algoritmo Knuth–Morris–Pratt

---

- Na ciência da computação, o algoritmo de busca de strings Knuth–Morris–Pratt (ou algoritmo KMP) procura ocorrências de uma "palavra"  $W$  dentro de uma "string de texto" principal  $S$ , empregando a observação de que, quando ocorre uma incompatibilidade, a própria palavra incorpora informações suficientes para determinar onde a próxima correspondência poderia começar, evitando assim o reexame dos caracteres combinados anteriormente.
- O algoritmo foi concebido por James H. Morris e descoberto independentemente por Donald Knuth "algumas semanas depois" da teoria dos autômatos. Morris e Vaughan Pratt publicaram um relatório técnico em 1970. Os três também publicaram o algoritmo em conjunto em 1977.

# O Algoritmo Knuth–Morris–Pratt

- Problema: Dado um texto `txt[0..n-1]` e um padrão `pat[0..m-1]`, escreva uma função `pesquisa(pat,txt)` que imprime todas as ocorrências de `pat` em `txt`. Podemos assumir que `n > m`.



# O Algoritmo Knuth–Morris–Pratt

---

- A pesquisa de padrões é um problema importante na ciência da computação.
- Quando fazemos uma busca por uma string no [notepad](#) / [arquivo word](#) ou [navegador](#) ou [banco de dados](#), algoritmos de busca de padrões são usados para mostrar os resultados da busca.
- O algoritmo de correspondência KMP usa a propriedade degenerativa (padrão com os mesmos subpadrões aparecendo mais de uma vez no padrão) do padrão e melhora a complexidade do pior caso para  [\$O\(n\)\$](#) .
- A ideia básica por trás do algoritmo do KMP é: sempre que detectamos uma incompatibilidade (após algumas correspondências), já conhecemos alguns dos caracteres no texto da próxima janela.
- Aproveitamos essas informações para evitar a correspondência de caracteres que sabemos que corresponderão de qualquer maneira.

# O Algoritmo Knuth–Morris–Pratt

- Vamos considerar o exemplo abaixo para entender isso:

Visão geral da correspondência

Texto = "AAAAAABAAAABA"

Padrão = "AAAAA"

Encontramos uma correspondência. Isso é o mesmo que **Naive String Matching**.

Comparamos a primeira janela do Texto com o Padrão

Texto = "AAAAAABAAAABA"

Padrão = "AAAAA" Posição inicial

Na próxima etapa, compararemos a próxima janela de Texto com Padrão.

Texto = "AAAAA**A**BAAAABA"

Padrão = "AAAAA**A**" O padrão mudou uma posição

# O Algoritmo Knuth–Morris–Pratt

---

- É aqui que o **KMP** faz a otimização sobre o **Naive**. Nessa segunda janela, comparamos apenas o quarto **A** do padrão com o quarto caractere da janela atual de texto para decidir se a janela atual corresponde ou não.
- Já que sabemos que os três primeiros caracteres corresponderão de qualquer maneira, pulamos a correspondência dos três primeiros caracteres.
- Uma questão importante surge da explicação acima, como saber quantos caracteres devem ser ignorados.
- Para saber isso, pré-processamos o padrão e preparamos um array **lps** de inteiros que nos informa a contagem de caracteres a serem ignorados.

# O Algoritmo Knuth–Morris–Pratt

---

- O algoritmo KMP pré-processa **pat** e constrói um array auxiliar **lps** de tamanho **m** (igual ao tamanho do padrão) que é usado para pular caracteres durante a correspondência.
- O nome **lps** indica o **longest proper prefix which is also suffix**. Um prefixo adequado é o prefixo com string inteira não permitida. Por exemplo, os prefixos de “ABC” são “”, “A”, “AB” e “ABC”. Os prefixos adequados são “”, “A” e “AB”. Os sufixos da string são “”, “C”, “BC” e “ABC”.
- Procuramos **lps** em subpadrões. Mais claramente, nos concentramos em subcadeias de padrões que são prefixo e sufixo.
- Para cada subpadrão **pat[0...i]** onde **i = 0 a m-1**, **lps[i]** armazena o comprimento do prefixo adequado de correspondência máxima que também é um sufixo do subpadrão **pat[0...i]**.

**lps[i]** = o prefixo adequado mais longo de **pat[0...i]**  
que também é um sufixo de **pat[0...i]**.

# O Algoritmo Knuth–Morris–Pratt

---

- Observação:  $\text{lps}[i]$  também pode ser definido como o prefixo mais longo, que também é o sufixo adequado. Precisamos usar corretamente em um lugar para garantir que toda a substring não seja considerada. Exemplos de construção  $\text{lps}$ :

Para o padrão **“AAAAA”**  $\text{lps} = [0, 1, 2, 3]$

Para o padrão **“ABCDE”**  $\text{lps} = [0, 0, 0, 0, 0]$

Para o padrão **“AABAACAAABAA”**  $\text{lps} = [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]$

Para o padrão **“AAACAAAAAAC”**  $\text{lps} = [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]$

Para o padrão **“AAABAAA”**  $\text{lps} = [0, 1, 2, 0, 1, 2, 3]$

# O Algoritmo Knuth–Morris–Pratt

---

- Ao contrário do algoritmo **Naive**, em que deslizamos o padrão por um e comparamos todos os caracteres em cada deslocamento, no KMP usamos um valor de **lps** para decidir os próximos caracteres a serem correspondidos.
- A ideia é não corresponder a um caractere que sabemos que corresponderá de qualquer maneira.
- Como usar **lps** para decidir as próximas posições (ou saber um número de caracteres a serem pulados)?
  - Começamos a comparação de **pat[j]** com  $j = 0$  com caracteres da janela de texto atual.
  - Continuamos correspondendo os caracteres **txt[i]** e **pat[j]** e continuamos incrementando  $i$  e  $j$  enquanto **pat[j]** e **txt[i]** continuam correspondendo.

# O Algoritmo Knuth–Morris–Pratt

---

- Quando vemos uma **incompatibilidade**:
  - Sabemos que os caracteres  $\text{pat}[0\dots j-1]$  correspondem a  $\text{txt}[i-j\dots i-1]$  (observe que  $j$  começa com 0 e o incrementa apenas quando há uma correspondência).
  - Também sabemos (da definição acima) que  $\text{lps}[j-1]$  é a contagem de caracteres de  $\text{pat}[0\dots j-1]$  que são prefixo e sufixo apropriados.
  - Dos dois pontos acima, podemos concluir que não precisamos combinar esses caracteres  $\text{lps}[j-1]$  com  $\text{txt}[i-j\dots i-1]$  porque sabemos que esses caracteres corresponderão de qualquer maneira. Vamos considerar o exemplo a seguir para entender essa ideia.

# O Algoritmo Knuth–Morris–Pratt

---

txt = "AAAAAABAAABA"

pat = "AAAAA"

lps = {0, 1, 2, 3}

i = 0, j = 0

txt = "AAAAAABAAABA"

pat = "AAAAA"

txt[i] and pat[j] correspondem,  
executar i++, j++

i = 1, j = 1

txt = "AAAAAABAAABA"

pat = "AAA"

txt[i] and pat[j] correspondem,  
executar i++, j++

i = 2, j = 2

txt = "AAAAAABAAABA"

pat = "AAA"

txt[i] and pat[j] correspondem,  
executar i++, j++

# O Algoritmo Knuth–Morris–Pratt

---

i = 3, j = 3

txt = "AAAAAABAAABA"

pat = "AAAAA"

txt[i] and pat[j] correspondem,  
executar i++, j++

i = 4, j = 4

Como j == M, imprimir padrão  
encontrado e resetar j,  
j = lps[j-1] = lps[3] = 3

Aqui, ao contrário do algoritmo **Naive**, não combinamos os primeiros três caracteres desta janela. O valor de **lps[j-1]** (na etapa acima) nos deu o índice do próximo caractere a ser correspondido.

i = 4, j = 3

txt = "AAAAAABAAABA"

pat = "AAAAA"

txt[i] and pat[j] correspondem,  
executar i++, j++

i = 5, j = 4

Como j == M, imprimir padrão  
encontrado e resetar j,  
j = lps[j-1] = lps[2] = 2

Ao contrário do algoritmo **Naive**, não combinamos os primeiros três caracteres desta janela. O valor de **lps[j-1]** (na etapa acima) nos deu o índice do próximo caractere a ser correspondido.

# O Algoritmo Knuth–Morris–Pratt

---

i = 5, j = 3

txt = "AAAAAA**B**AAABA"

pat = "AAAA**A**"

txt[i] and pat[j] não correspondem,  
e j > 0, mudar apenas j  
j = lps[j-1] = lps[2] = 2

i = 5, j = 2

txt = "AAAAAA**B**AAABA"

pat = "AAAA**A**"

txt[i] and pat[j] não correspondem,  
e j > 0, mudar apenas j  
j = lps[j-1] = lps[1] = 1

i = 5, j = 1

txt = "AAAAAA**B**AAABA"

pat = "A**AAA**"

txt[i] and pat[j] não correspondem,  
e j > 0, mudar apenas j  
j = lps[j-1] = lps[0] = 0

i = 5, j = 0

txt = "AAAAAA**B**AAABA"

pat = "**A**AAA"

txt[i] and pat[j] não correspondem,  
e j é 0, executamos então i++

# O Algoritmo Knuth–Morris–Pratt

---

$i = 6, j = 0$

$txt =$  "AAAAAAB**A**AABA"

$pat =$  "**AAAA**"

$txt[i]$  and  $pat[j]$  correspondem,  
executar  $i++, j++$

$i = 7, j = 1$

"AAAAAABAA**A**ABA"

"**AAAA**"

$txt[i]$  and  $pat[j]$  correspondem,  
executar  $i++, j++$

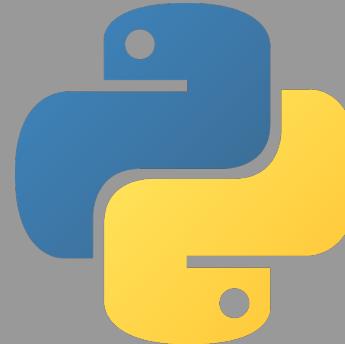
E assim o Algoritmo  
continua...

# O Algoritmo Knuth–Morris–Pratt

- ♦ A seguir temos o código que implementa o algoritmo KMP:

```
def KMP(pat,txt):
    M = len(pat)
    N = len(txt)
    lps = [0]*M
    j = 0
    compute_lps_array(pat, M, lps)

    i = 0
    while i < N:
        if pat[j] == txt[i]:
            i += 1
            j += 1
        if j == M:
            print(f"Encontrado padrão no índice {str(i-j)}")
            j = lps[j-1]
        elif i < N and pat[j] != txt[i]:
            if j != 0:
                j = lps[j-1]
            else:
                i += 1
```

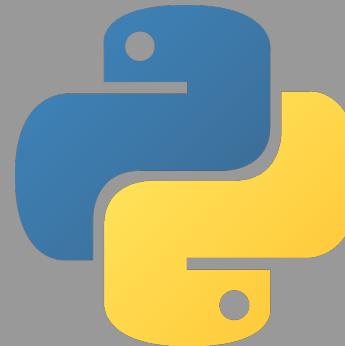


# O Algoritmo Knuth–Morris–Pratt

- E o código que computa o array lps:

```
def compute_lps_array(pat, M, lps):
    len = 0
    lps[0]
    i = 1

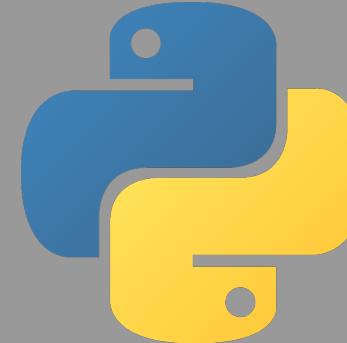
    while i < M:
        if pat[i]== pat[len]:
            len += 1
            lps[i] = len
            i += 1
        else:
            if len != 0:
                len = lps[len-1]
            else:
                lps[i] = 0
                i += 1
```



# O Algoritmo Knuth–Morris–Pratt

- Finalmente podemos testar o algoritmo KMP:

```
# t = texto, p = padrão
t1 = "ABABDABACDABABCABAB"
p1 = "ABABCABAB"
KMP(p1,t1)
t2 = "AAAAAA"
p2 = "AA"
KMP(p2,t2)
t3 = "ABBBCAABCC"
p3 = "AB"
KMP(p3,t3)
```



# **Lista de Algoritmos para Estudos**

---



<https://github.com/tayllan/awesome-algorithms>

<https://www.geeksforgeeks.org/fundamentals-of-algorithms>

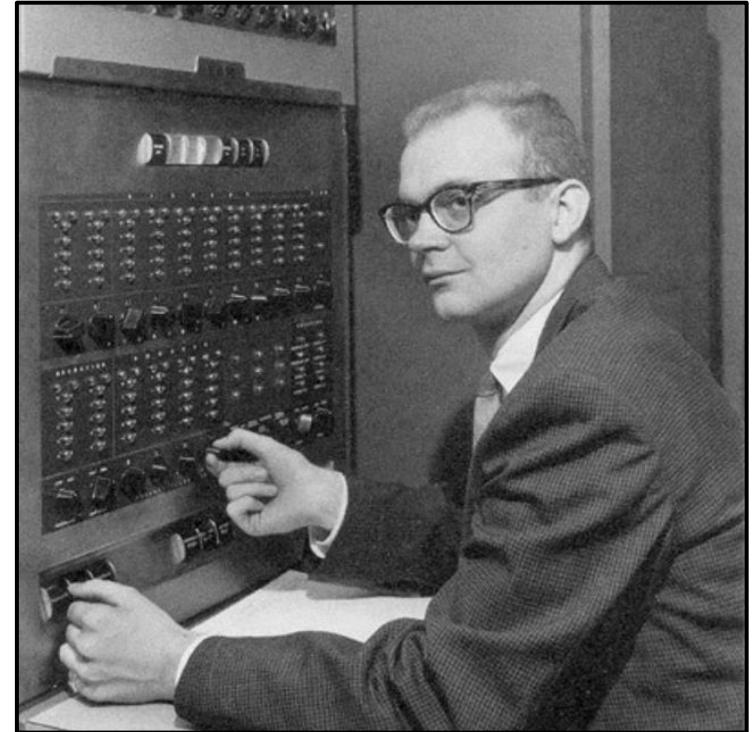
[https://en.wikipedia.org/wiki/List\\_of\\_algorithms](https://en.wikipedia.org/wiki/List_of_algorithms)

<https://github.com/trekhleb/javascript-algorithms>

# Construindo Algoritmos

---

“The best programs are written so that computing machines can perform them quickly and so that human beings can understand them clearly. A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts, to communicate the way that an algorithm works and to convince a reader that the results will be correct.” **Donald Knuth**



# Estruturas de Dados

---

- Em nossa breve introdução, vimos que uma **estrutura de dados** é um formato de **organização**, **gerenciamento** e **armazenamento** de dados que permite **acesso** e **modificação** eficientes.
- Mais precisamente, uma estrutura de dados é uma coleção de valores de dados, os relacionamentos entre eles e as funções ou operações que podem ser aplicadas aos dados.
- Uma estrutura de dados é normalmente classificada em duas categorias:
  - Estrutura de Dados Lineares
  - Estrutura de Dados Não Lineares

# Estruturas de Dados Lineares

---

- ♦ As **estrutura de dados lineares** tem elementos de dados organizados de maneira sequencial e cada elemento membro é conectado ao seu elemento anterior e seguinte.
- ♦ Essa conexão ajuda a percorrer uma estrutura de dados linear em um único nível e em uma única execução.
- ♦ Essas estruturas de dados são fáceis de implementar, pois a memória do computador também é sequencial.
- ♦ Exemplos de estruturas de dados lineares são [Lista](#), [Queue \(Fila\)](#), [Stack \(Pilha\)](#), [Array](#), etc.

# Estruturas de Dados Não Lineares

---

- Uma **estrutura de dados não linear** não tem sequência definida para conectar todos os seus elementos e cada elemento pode ter vários caminhos para se conectar a outros elementos.
- Essas estruturas de dados suportam armazenamento em vários níveis e geralmente não podem ser percorridas em uma única execução.
- Essas estruturas de dados não são fáceis de implementar, mas são mais eficientes na utilização da memória do computador.
- Exemplos de estruturas de dados não lineares são [Árvore](#), [Binary Search Tree](#), [Grafos](#), etc.

# Uso das Estruturas de Dados

---

- Estruturas de dados servem como base para **Abstract Data Types** (ADT).
- O **ADT** define a forma lógica do tipo de dados. A estrutura de dados implementa a forma física do tipo de dados.
- Diferentes tipos de estruturas de dados são adequados para diferentes tipos de aplicações e alguns são altamente especializados para tarefas específicas.
- Por exemplo, bancos de dados relacionais comumente usam **B-tree indexes** para recuperação de dados, enquanto implementações de compilador geralmente usam **hash tables** para procurar identificadores.

# Uso das Estruturas de Dados

---

- As **estruturas de dados** fornecem um meio de gerenciar grandes quantidades de dados com eficiência para usos como grandes bancos de dados e serviços de indexação da Internet.
- Normalmente, **estruturas de dados eficientes** são essenciais para projetar **algoritmos eficientes**.
- Alguns métodos de design formal e linguagens de programação enfatizam as estruturas de dados, ao invés de algoritmos, como o principal fator de organização no design de software.
- As estruturas de dados podem ser usadas para organizar o armazenamento e a recuperação das informações armazenadas na memória principal e na memória secundária.

# Implementação das Estruturas de Dados

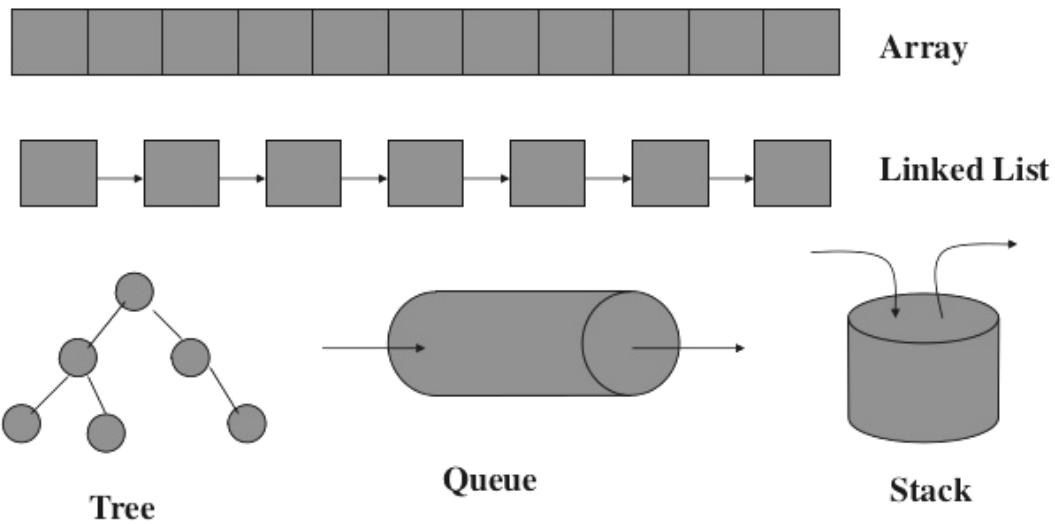
---

- As estruturas de dados geralmente são baseadas na capacidade de um computador de buscar e armazenar dados em qualquer lugar de sua memória, especificado por um **ponteiro** - uma **string de bits**, representando um **endereço de memória**, que pode ser armazenada na memória e manipulada pelo programa.
- O **array** e as estruturas de dados de **registro** são baseadas na computação dos endereços de itens de dados com operações aritméticas, enquanto as estruturas de dados vinculadas (**linked**) são baseadas no armazenamento de endereços de itens de dados dentro da própria estrutura.
- A implementação de uma estrutura de dados geralmente requer a escrita de um conjunto de procedimentos que criam e manipulam instâncias dessa estrutura.
- A eficiência de uma estrutura de dados não pode ser analisada separadamente dessas operações. Essa observação motiva o conceito teórico de um **abstract data type**, uma estrutura de dados que é definida indiretamente pelas operações que podem ser realizadas sobre ela e as propriedades matemáticas dessas operações (incluindo seu custo de **espaço** e **tempo**).

# Exemplos de Estruturas de Dados

- Existem vários tipos de estruturas de dados, geralmente construídas sobre **tipos de dados primitivos** mais simples:

- Arrays
- Linked Lists
- Stacks (Pilhas)
- Queues (Filas)
- Maps & Hash Tables
- Grafos
- Trees (Árvores)
- Binary Trees & Binary Search Trees
- Self-balancing Trees (AVL Trees, Red-Black Trees, Splay Trees)
- Heaps
- Tries



**Lista detalhada:** [https://en.wikipedia.org/wiki/List\\_of\\_data\\_structures](https://en.wikipedia.org/wiki/List_of_data_structures)

# Arrays

---

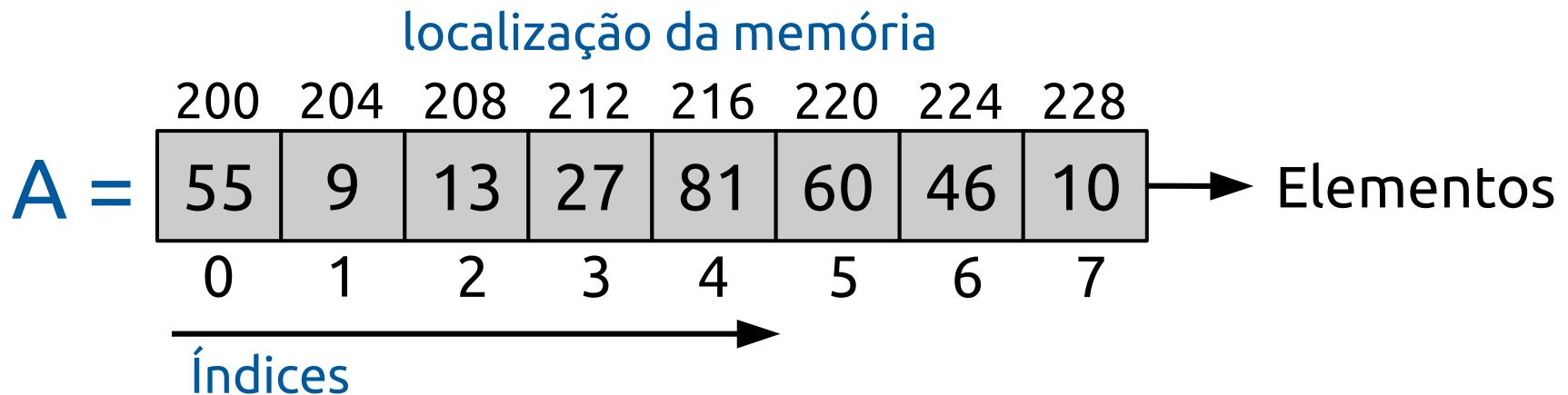
- Um **array** é uma estrutura de dados que consiste em uma coleção de elementos (valores ou variáveis), cada um identificado por pelo menos um índice de array ou chave.
- Um array é armazenado de forma que a posição de cada elemento possa ser calculada a partir de sua **tupla de índice** por uma fórmula matemática.
- O tipo mais simples de estrutura de dados é um array linear, também chamada de array unidimensional.
- Por exemplo, um array de 10 variáveis inteiras de **32 bits** (4 bytes), com índices de **0** a **9**, pode ser armazenada como 10 **words** nos endereços de memória **2000, 2004, 2008, ..., 2036**, (em hexadecimal: **0x7D0, 0x7D4, 0x7D8, ..., 0x7F4**) para que o elemento com índice  $i$  tenha o endereço  $2000 + (i \times 4)$ .
- O endereço de memória do primeiro elemento de um array é chamado de primeiro endereço ou endereço base.
- Os arrays estão entre as estruturas de dados mais antigas e importantes e são usados por quase todos os programas.

# Arrays

---

- ◆ Propriedades dos Arrays:
  - Os valores dos elementos são colocados em ordem e acessados por seu índice de 0 ao comprimento do array-1.
  - Um array é um bloco contínuo de memória.
  - Geralmente são feitos de elementos do mesmo tipo (depende da linguagem de programação).
  - O acesso e a adição de elementos são rápidos; busca e exclusão não são feitas em  $O(1)$ .

# Arrays



Complexidade

$$\begin{aligned} A[0] &= 55 \\ A[1] &= 9 \\ A[7] &= 10 \end{aligned}$$

Inserir no Fim	Acesso	Busca	Remoção
$O(1)$	$O(1)$	$O(n)$	$O(n)$

# Linked Lists

---

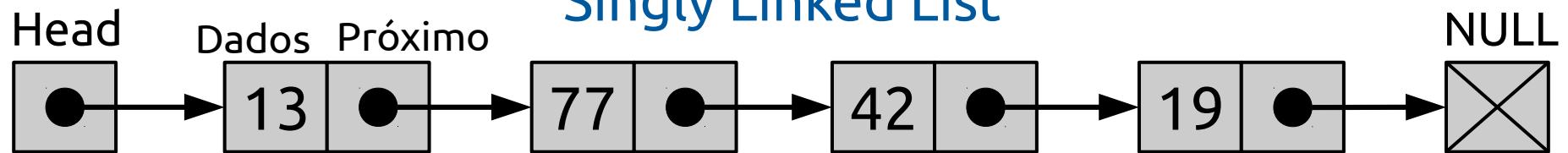
- Uma **Linked List** é uma coleção linear de elementos de dados cuja ordem não é dada por sua colocação física na memória.
- Em vez disso, cada elemento **aponta** para o próximo.
- É uma estrutura de dados que consiste em uma coleção de **nós** que juntos representam uma **sequência**.
- Em sua forma mais básica, cada nó contém: **dados** e uma **referência** (em outras palavras, um link) para o próximo nó na sequência.
- Essa estrutura permite a inserção ou remoção eficiente de elementos de qualquer posição na sequência durante a iteração.

# Linked Lists

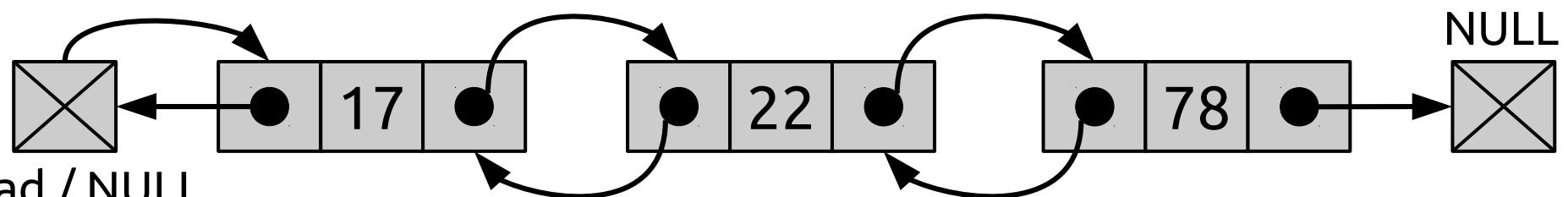
---

- Variantes mais complexas adicionam links, permitindo uma inserção ou remoção mais eficiente de nós em posições arbitrárias.
- Uma desvantagem das Linked Lists é que o tempo de acesso é linear.
- O acesso mais rápido, como o acesso aleatório, não é viável.
- Os arrays têm melhor cache locality em comparação com Linked Lists.
- O principal benefício de uma Linked List em relação a um array convencional é que os elementos dela podem ser facilmente inseridos ou removidos sem realocação ou reorganização de toda a estrutura, porque os itens de dados não precisam ser armazenados de forma contígua na memória ou no disco.
- Linked Lists permitem a inserção e remoção de nós em qualquer ponto da lista, e permitem fazer isso com um número constante de operações, mantendo o link anterior ao link sendo adicionado ou removido na memória durante a travessia dela.

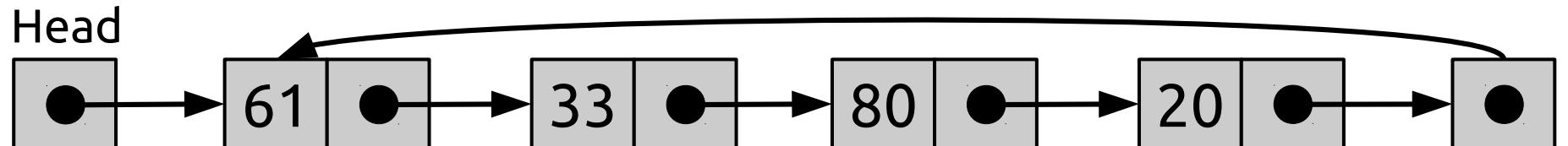
# Linked Lists



Doubly Linked List



Circular Linked List



# Linked Lists

---

- **Singly linked list:** Contêm nós que têm um campo de dados, bem como um campo 'próximo', que aponta para o próximo nó na linha de nós. As operações que podem ser realizadas em singly linked lists incluem inserção, exclusão e passagem.
- **Doubly linked list:** Em cada nó contém, além do link do próximo nó, um segundo campo de link apontando para o nó 'anterior' na sequência. Os dois links podem ser chamados de 'para frente' e 'para trás', ou 'próximo' e 'anterior'.
- **Circular linked list:** No último nó de uma lista, o campo de link geralmente contém uma referência nula, um valor especial é usado para indicar a falta de outros nós. Uma convenção menos comum é apontar para o primeiro nó da lista; nesse caso, a lista é considerada 'circular'.

# Linked Lists

---

- Propriedades das Linked Lists:
  - Os elementos não são armazenados em um bloco contíguo de memória.
  - Perfeito para um excelente gerenciamento de memória (usar ponteiros implica em uso de memória dinâmica).
  - Inserção e exclusão são rápidas; acessar e pesquisar elementos são feitos em tempo linear.

## Complexidade

Inserir	Acesso	Busca	Remoção
$O(1)$	$O(n)$	$O(n)$	$O(1)$

# Stacks

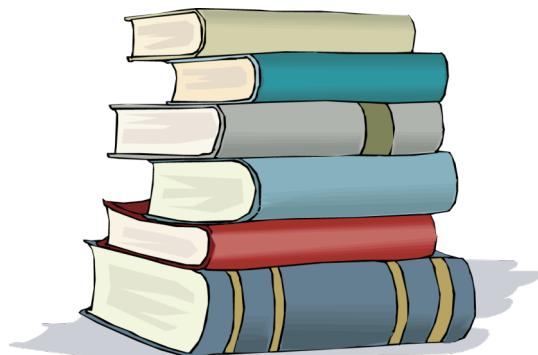
---

- ♦ Uma **stack** é um **abstract data type** que serve como uma coleção de elementos, com duas operações principais principais:
  - **Push**, que adiciona um elemento à coleção.
  - **Pop**, que remove o elemento adicionado mais recentemente que ainda não foi removido.
- ♦ A ordem em que os elementos saem de uma pilha dá origem ao seu nome alternativo, **LIFO** (last in, first out) (último a entrar, primeiro a sair).

# Stacks

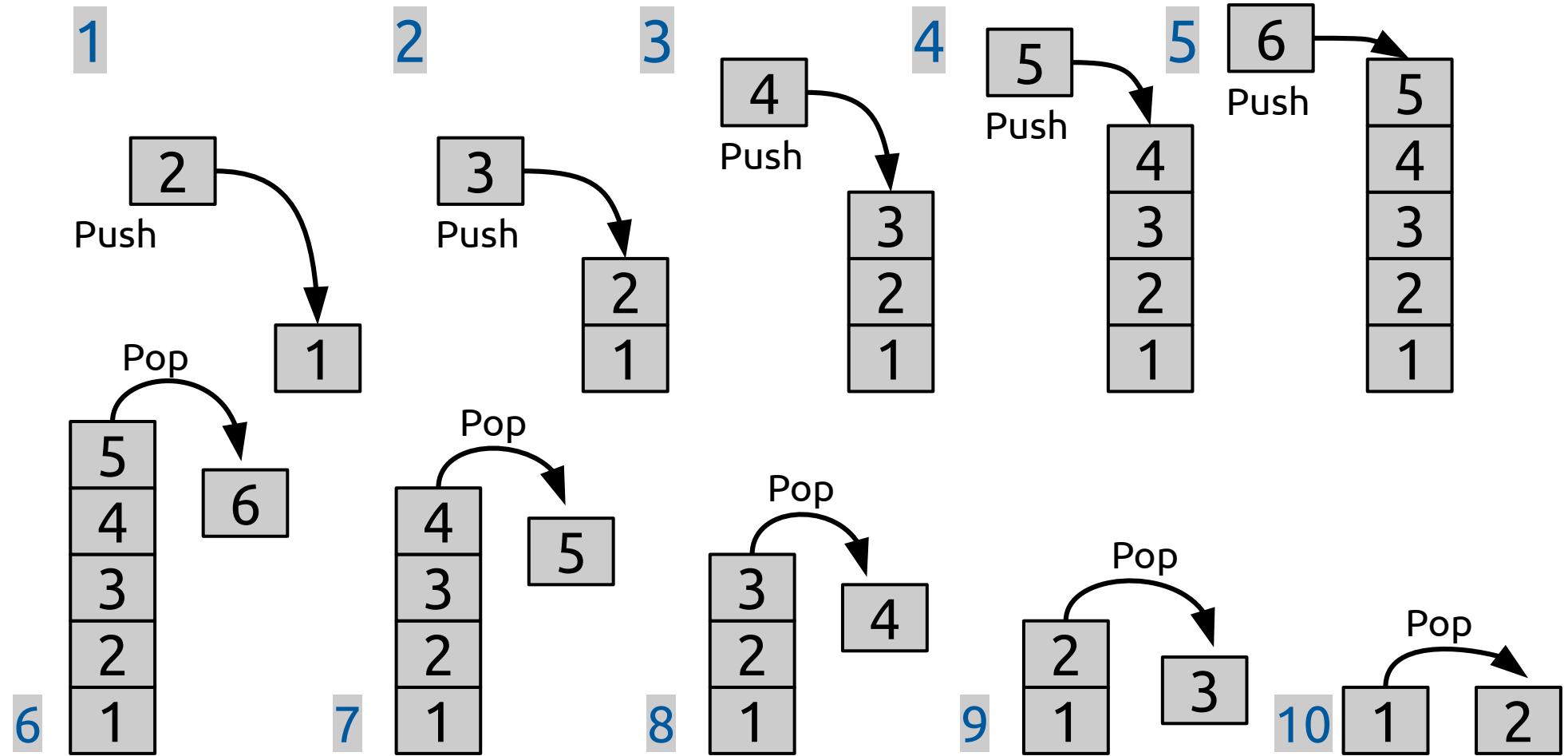
---

- ◆ O nome "**stack**" (pilha) para esse tipo de estrutura vem da analogia com um conjunto de itens físicos empilhados uns sobre os outros.
- ◆ Essa estrutura torna mais fácil retirar um item do topo da stack, enquanto chegar a um item mais profundo na stack pode exigir a retirada de vários outros itens primeiro.



Semelhante a uma pilha de livros, adicionar ou remover só é possível no topo.

# Stacks



# Stacks

---

- Propriedades das Stacks:

- Você só pode acessar o último elemento de cada vez (o que está no topo).
- Uma desvantagem é que, uma vez que você remova os elementos do topo para acessar outros elementos, seus valores serão perdidos da memória da stack.
- O acesso de outros elementos é feito em tempo linear; qualquer outra operação está em **O(1)**. **Complexidade**

Push	Pop	Top	Acesso
O(1)	O(1)	O(1)	O(n)

# Queues

---

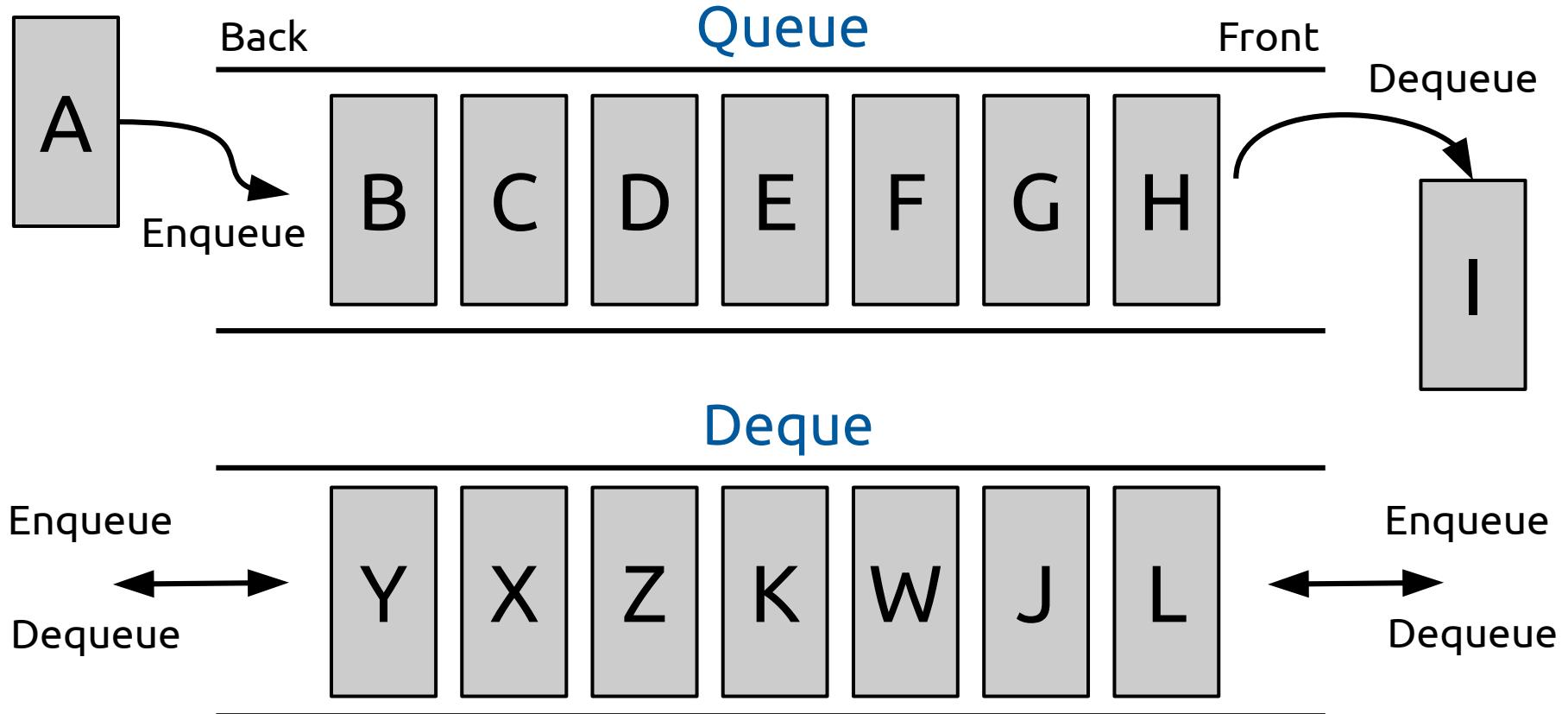
- Uma **Queue** (fila) é uma coleção de entidades que são mantidas em uma sequência e podem ser modificadas pela adição de entidades em uma extremidade da sequência e a remoção de entidades da outra extremidade da sequência.
- Por convenção, o final da sequência na qual os elementos são adicionados é chamado de **parte traseira**, **tail** ou **rear** da queue, e o final na qual os elementos são removidos é chamado de **head** ou **início** da fila, analogamente às palavras usadas quando as pessoas fazem fila para esperar por bens ou serviços.
- A operação de adicionar um elemento à parte traseira da fila é conhecida como **enqueue** (enfileirar), e a operação de remover um elemento da frente é conhecida como **dequeue** (retirar da fila). Outras operações também podem ser permitidas.

# Queues

---

- ◆ As operações de uma Queue tornam-na uma estrutura de dados primeiro a entrar, primeiro a sair (**first-in-first-out**), (**FIFO**).
- ◆ Em uma estrutura de dados FIFO, o primeiro elemento adicionado à fila será o primeiro a ser removido.
- ◆ Isso é equivalente ao requisito de que, uma vez que um novo elemento seja adicionado, todos os elementos que foram adicionados antes devem ser removidos antes que o novo elemento possa ser removido.
- ◆ As Queues fornecem serviços em ciência da computação, transporte e pesquisa operacional em que várias entidades, como dados, objetos, pessoas ou eventos, são armazenados e mantidos para processamento posterior. Nesses contextos, a fila desempenha a função de um **buffer**.
- ◆ Outro uso de Queues é na implementação do algoritmo **breadth-first search**.

# Queues



Double-ended queue (abreviado como deque) generaliza uma Queue, nela os elementos podem ser adicionados ou removidos da frente (head) ou de trás (tail).

# Queues

---

- ♦ Propriedades das Queues:

- Podemos acessar diretamente apenas o elemento "mais antigo" introduzido.
- A pesquisa de elementos removerá todos os elementos acessados da memória da Queue.
- Remover / adicionar elementos ou obter a frente da fila é feito em tempo constante. A pesquisa é linear.

## Complexidade

Enqueue	Dequeue	Front	Search
$O(1)$	$O(1)$	$O(1)$	$O(n)$

# Hash Table

---

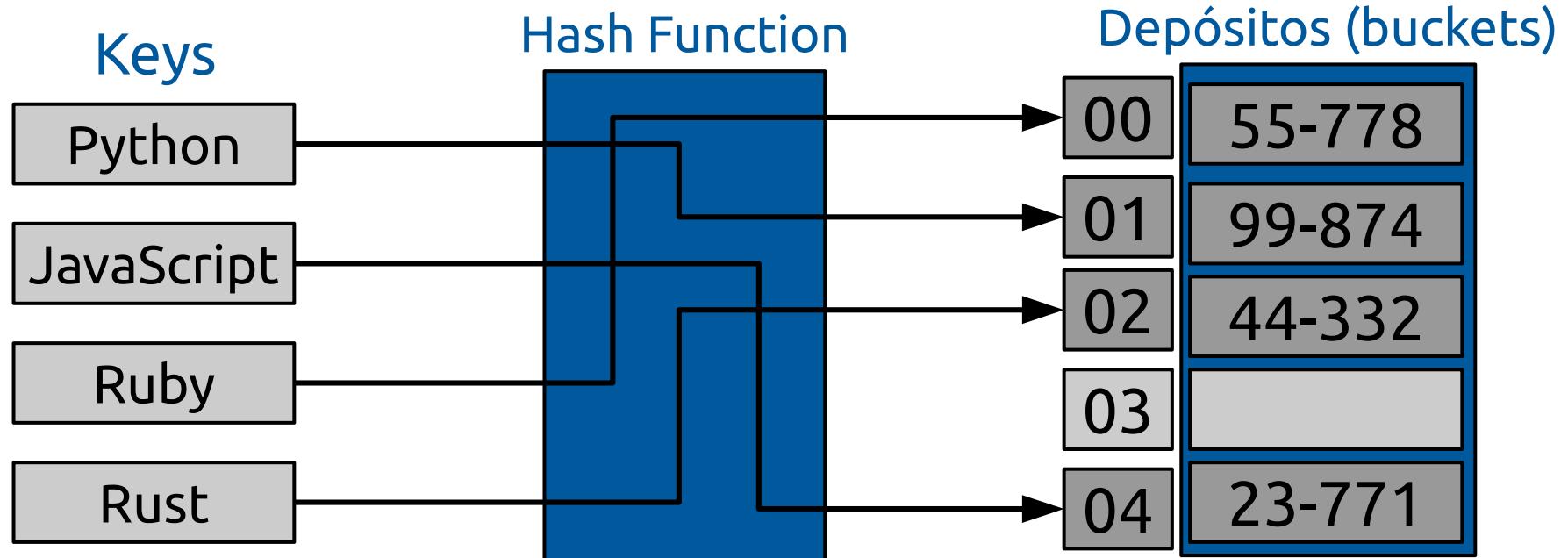
- ◆ Uma **Hash Table** (também conhecida como Hash Map) é uma estrutura de dados que implementa um abstract data type de **array associativo**, uma estrutura que pode mapear **chaves** para **valores**.
- ◆ Uma Hash Table usa uma **função hash** para calcular um **índice**, também chamado de código hash, em um array de depósitos ou slots, a partir dos quais o valor desejado pode ser encontrado.
- ◆ Durante a pesquisa, a **chave** é “hashed” e o hash resultante indica onde o valor correspondente está armazenado.
- ◆ Idealmente, a função hash atribuirá cada chave a um depósito exclusivo, mas a maioria dos designs de hash table emprega uma função hash imperfeita, que pode causar **colisões** de hash onde a função hash gera o mesmo índice para mais de uma chave.

# Hash Table

---

- Em uma Hash Table bem dimensionada, o custo médio (número de instruções) para cada consulta é independente do número de elementos armazenados na tabela.
- Muitos designs de Hash Table também permitem inserções e exclusões arbitrárias de **pares de valores-chave**, a um custo médio constante (**amortizado**) por operação.
- Em muitas situações, as hash tables são, em média, mais eficientes do que as árvores de pesquisa ou qualquer outra estrutura de pesquisa de tabela.
- Por esse motivo, elas são amplamente usados em muitos tipos de software de computador, especialmente para **arrays associativas**, **indexação de banco de dados**, **caches** e **conjuntos**.

# Hash Table



Inserção	Acesso	Remoção	Pesquisa
$O(1)$	$O(1)$	$O(1)$	$O(1)$

# Hash Table

---

- ◆ Propriedades das Hash Tables:
  - As chaves são exclusivas (sem duplicatas).
  - Resistência à colisão: deve ser difícil encontrar duas entradas diferentes com a mesma chave.
  - Resistência da pré-imagem: dado um valor  $H$ , deve ser difícil encontrar uma chave  $x$ , tal que  $h(x) = H$ .
  - Resistência da segunda pré-imagem: dada uma chave e seu valor, deve ser difícil encontrar outra chave com o mesmo valor.

# Grafos

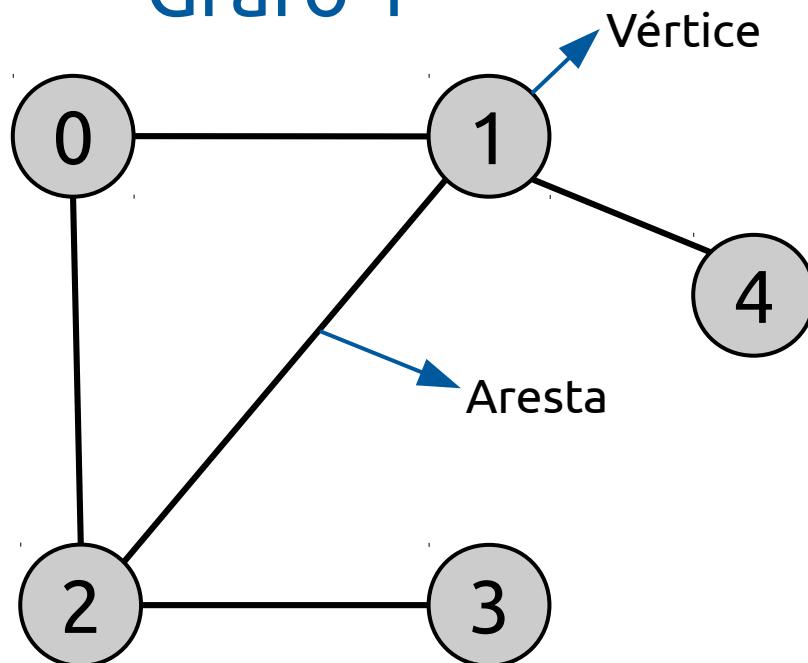
---

- Um **grafo** é um **abstract data type** que se destina a implementar o **grafo não-direcionado** e os conceitos de **grafo direcionado** do campo da teoria dos grafos dentro da matemática.
- Uma estrutura de dados de grafo consiste em um conjunto finito (e possivelmente mutável) de **vértices** (também chamados de nós ou pontos), junto com um conjunto de pares não ordenados desses vértices para um grafo não-direcionado ou um conjunto de pares ordenados para um gráfico direcionado.
- Esses pares são conhecidos como **arestas** (também chamados de links ou linhas) e, para um grafo direcionado, também são conhecidos como **setas**. Os vértices podem fazer parte da estrutura do grafo ou podem ser entidades externas representadas por índices inteiros ou referências.

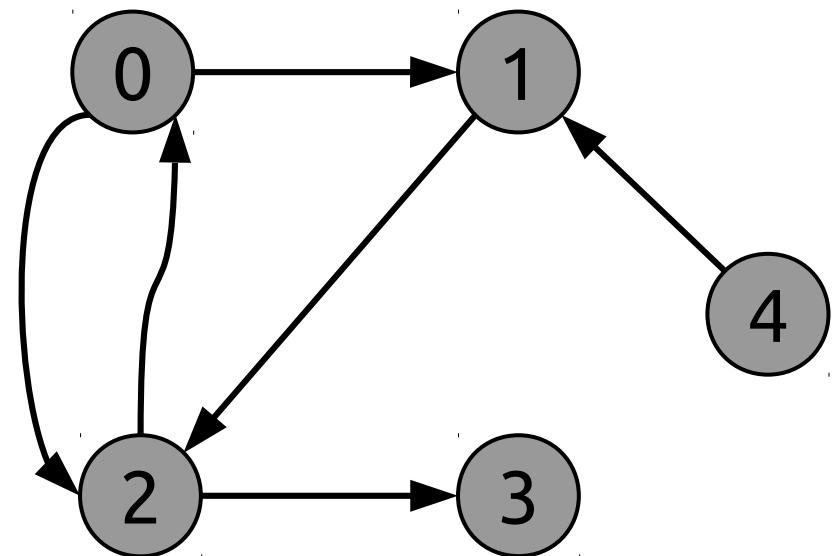
# Grafos

---

Grafo 1



Grafo 2



Não-Direcionado

Direcionado

# Grafos

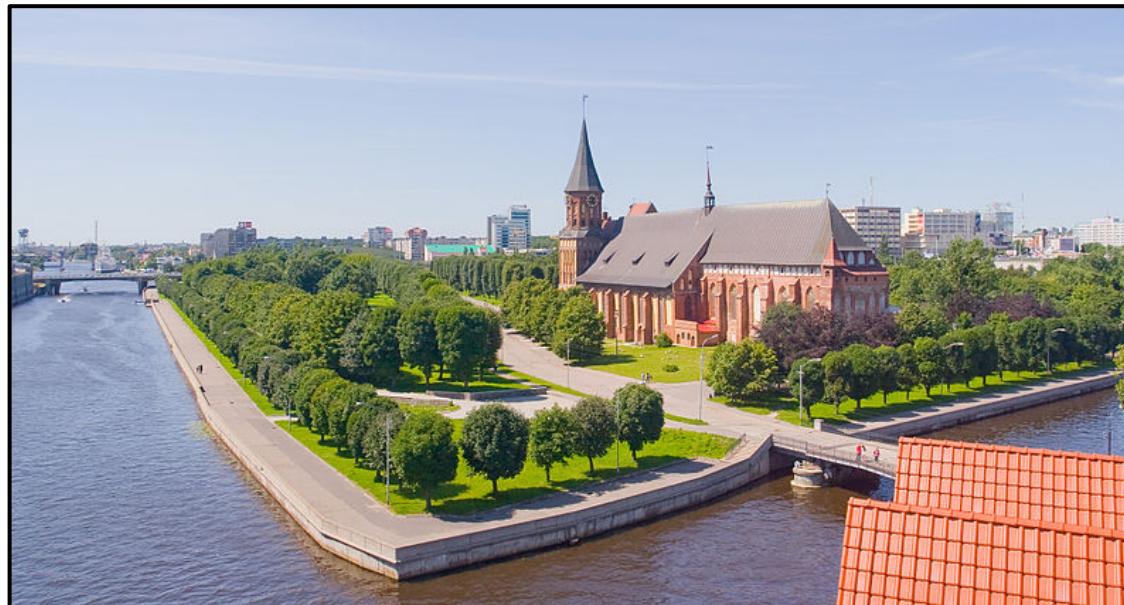
---

- As operações básicas fornecidas por uma estrutura de dados de grafo **G** geralmente incluem:
  - **adjacente(G, x, y)**: testa se há uma aresta do vértice **x** para o vértice **y**;
  - **vizinhos(G, x)**: lista todos os vértices **y** de forma que haja uma aresta do vértice **x** ao vértice **y**;
  - **add\_vertex(G, x)**: adiciona o vértice **x**, se não estiver lá;
  - **remove\_vertex(G, x)**: remove o vértice **x**, se estiver lá;
  - **add\_edge(G, x, y)**: adiciona a aresta do vértice **x** ao vértice **y**, se não estiver lá;
  - **remove\_edge(G, x, y)**: remove a aresta do vértice **x** para o vértice **y**, se estiver lá;
  - **get\_vertex\_value(G, x)**: retorna o valor associado ao vértice **x**;

# As Sete Pontes de Königsber

---

- As [Sete Pontes de Königsberg](#) é um problema historicamente notável em matemática.
- Sua resolução negativa por [Leonhard Euler](#) em 1736 lançou as bases da **teoria dos grafos** e prefigurou a ideia de **topologia**.



# As Sete Pontes de Königsber

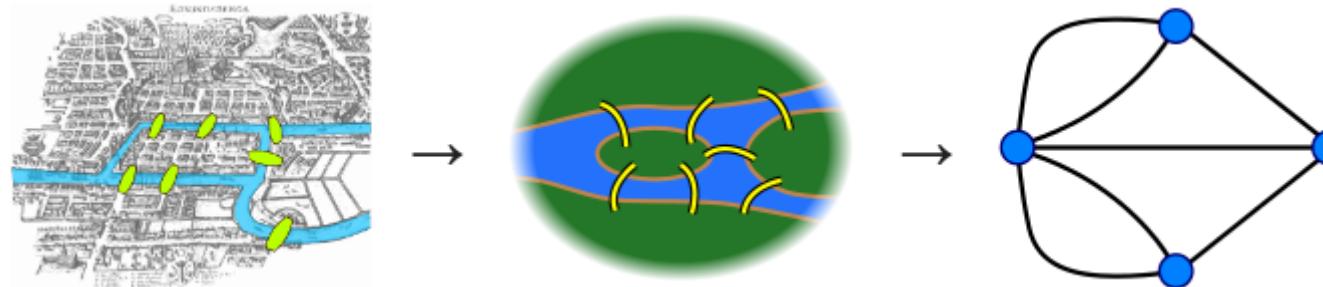
---

- A cidade de **Königsberg**, na Prússia (agora Kaliningrado, Rússia) foi estabelecida em ambos os lados do rio Pregel e incluía duas grandes ilhas - Kneiphof e Lomse - que estavam conectadas entre si, ou às duas partes do continente da cidade, por **sete pontes**.
- O problema era planejar um passeio pela cidade que cruzasse cada uma dessas pontes uma vez e apenas uma vez.
- Por meio de especificar a tarefa lógica de forma inequívoca, soluções envolvendo qualquer:
  - Alcançando uma ilha ou banco continental que não seja por meio de uma das pontes;
  - Acessar qualquer ponte sem cruzar para a outra extremidade;
- são explicitamente inaceitáveis.

# As Sete Pontes de Königsber

---

- Euler provou que o problema não tem solução.
- Primeiro, Euler apontou que a escolha da rota dentro de cada massa de terra é irrelevante.
- A única característica importante de uma rota é a sequência de pontes cruzadas.
- Isso lhe permitiu reformular o problema em termos abstratos (estabelecendo as bases da **teoria dos grafos**), eliminando todas as características, exceto a lista de massas de terra e as pontes que as conectam.



# As Sete Pontes de Königsber

---

- Em termos modernos, substitui-se cada massa de terra por um "vértice" ou nó abstrato, e cada ponte por uma conexão abstrata, uma "aresta", que serve apenas para registrar qual par de vértices (massas de terra) está conectado por aquela ponte. A estrutura matemática resultante é um grafo.
- Uma vez que apenas as informações de conexão são relevantes, a forma das representações pictóricas de um grafo pode ser distorcida de qualquer maneira, sem alterar o gráfico em si.
- Apenas a existência (ou ausência) de uma aresta entre cada par de nós é significativa. Por exemplo, não importa se as arestas desenhadas são retas ou curvas, ou se um nó está à esquerda ou à direita de outro.

# As Sete Pontes de Königsber

---

- A seguir, Euler observou que (exceto nos pontos finais do passeio), sempre que alguém entra em um vértice por uma ponte, sai do vértice por uma ponte.
- Em outras palavras, durante qualquer caminhada no grafo, o número de vezes que alguém entra em um vértice não-terminal é igual ao número de vezes que o deixa.
- Agora, se cada ponte foi atravessada exatamente uma vez, segue-se que, para cada massa de terreno (exceto para as escolhidas para o início e o fim), o número de pontes que tocam essa massa de terreno deve ser par (metade delas, na travessia particular, será percorrida "em direção" à massa de terra; a outra metade, "para longe" dela).

# As Sete Pontes de Königsber

---

- No entanto, todas as quatro massas de terra no problema original são tocadas por um número ímpar de pontes (uma é tocada por 5 pontes e cada uma das outras três é tocada por 3). Visto que, no máximo, duas massas de terra podem servir como pontos finais de uma caminhada, a proposição de uma caminhada atravessando cada ponte uma vez leva a uma contradição.
- Em linguagem moderna, Euler mostra que a possibilidade de um passeio por um grafo, percorrendo cada aresta exatamente uma vez, depende dos graus dos nós.
- O grau de um nó é o número de arestas que o tocam.
- O argumento de Euler mostra que uma condição necessária para o caminhar da forma desejada é que o grafo seja conectado e tenha exatamente zero ou dois nós de grau ímpar. Essa condição também se mostra suficiente - um resultado afirmado por Euler e mais tarde provado por Carl Hierholzer. Essa caminhada é agora chamada de caminho de Euler em sua homenagem. Além disso, se houver nós de grau ímpar, qualquer caminho Euleriano começará em um deles e terminará no outro. Como o grafo correspondente ao Königsberg histórico tem quatro nós de grau ímpar, ele não pode ter uma trajetória Euleriana.

# Trees

---

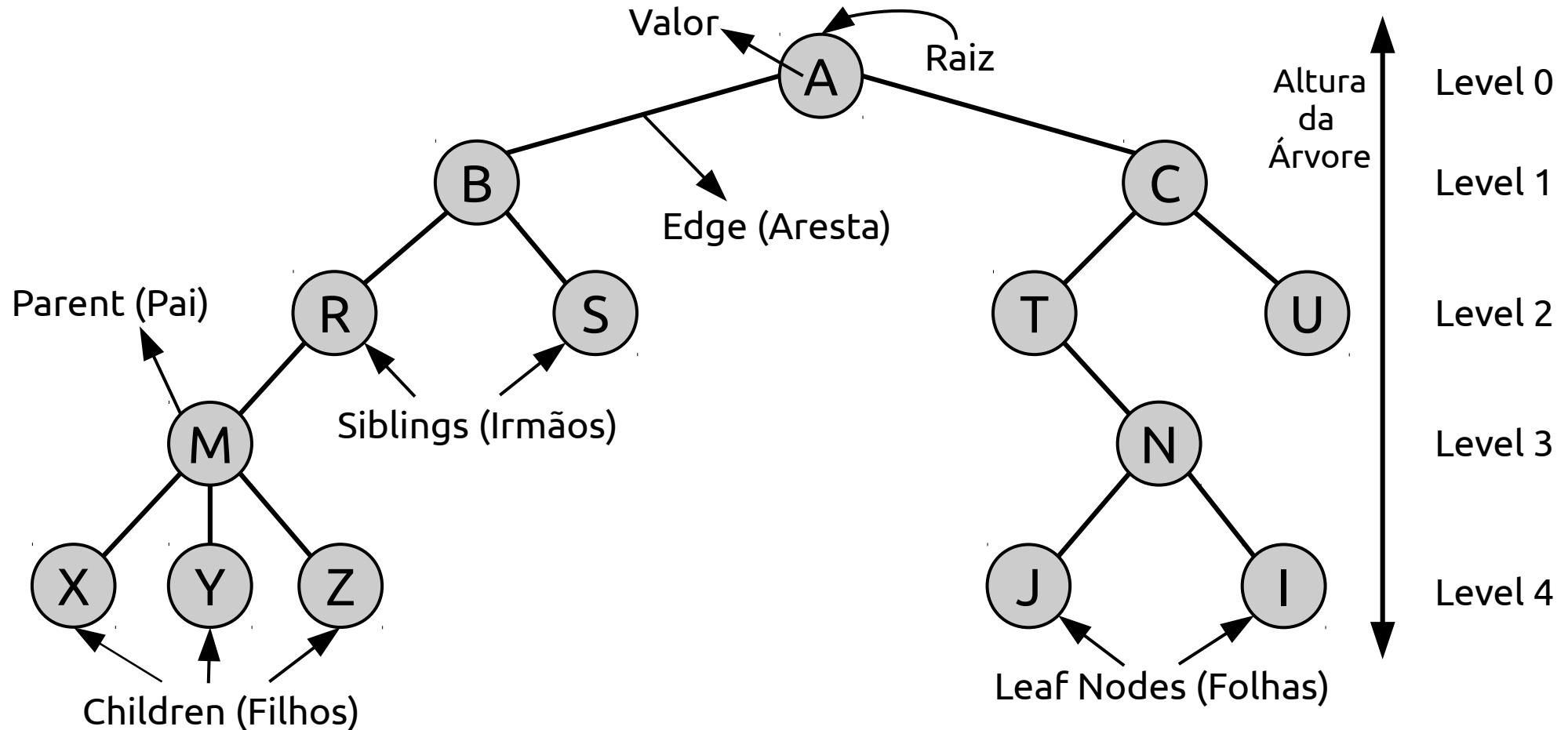
- ◆ Uma **Tree** (árvore) é um abstract data type amplamente usado que simula uma **estrutura de árvore hierárquica**, com um valor **raiz** e subárvore de **filhos** com um **nó pai**, representados como um conjunto de **nós vinculados**.
- ◆ Uma estrutura de dados em árvore pode ser definida recursivamente como uma **coleção de nós** (começando em um **nó raiz**), onde cada nó é uma estrutura de dados que consiste em um valor, juntamente com uma lista de referências a nós (os "filhos"), com o restrições de que nenhuma referência é duplicada e nenhuma aponta para a raiz.

# Trees

---

- Alternativamente, uma árvore pode ser definida abstratamente como um todo (globalmente) como uma árvore ordenada, com um valor atribuído a cada nó.
- Ambas as perspectivas são úteis: enquanto uma árvore pode ser analisada matematicamente como um todo, quando realmente representada como uma estrutura de dados, ela é geralmente representada e trabalhada separadamente por nó (ao invés de um conjunto de nós e uma lista adjacente de arestas entre nós , como se pode representar um dígrafo, por exemplo).
- Por exemplo, olhando para uma árvore como um todo, pode-se falar sobre "o nó pai" de um determinado nó, mas em geral, como uma estrutura de dados, um determinado nó contém apenas a lista de seus filhos, mas não contém uma referência para seu pai (se houver).

# Trees



# Trees

---

- Usos comuns das Árvores:
- Representando dados hierárquicos, como:
  - **Abstract syntax trees** para linguagens de computador.
  - **Parse trees** para linguagens humanas.
  - **Document Object Models** de documentos **XML** e **HTML**.
  - Documentos **JSON** e **YAML** sendo processados.
- As árvores de pesquisa armazenam dados de uma maneira que torna possível um algoritmo de pesquisa eficiente por meio da travessia da árvore:
  - Uma **árvore de pesquisa binária** é um tipo de **árvore binária**.
- Representando listas ordenadas de dados.
- Como um fluxo de trabalho para composição de imagens digitais para efeitos visuais.
- Armazenamento de árvores **Barnes-Hut** usadas para simular galáxias.

# Trees

---

- ◆ Propriedades das Árvores:

- A raiz não tem pai.
- As folhas não têm filhos.
- O comprimento da cadeia entre a raiz e um nó  $x$  representa o nível em que  $x$  está situado.
- A altura de uma árvore é o nível máximo dela (4 em nosso exemplo).
- O método mais comum para percorrer uma árvore é o DFS (Depth-first search), mas também podemos usar o BFS (Breadth-first search).

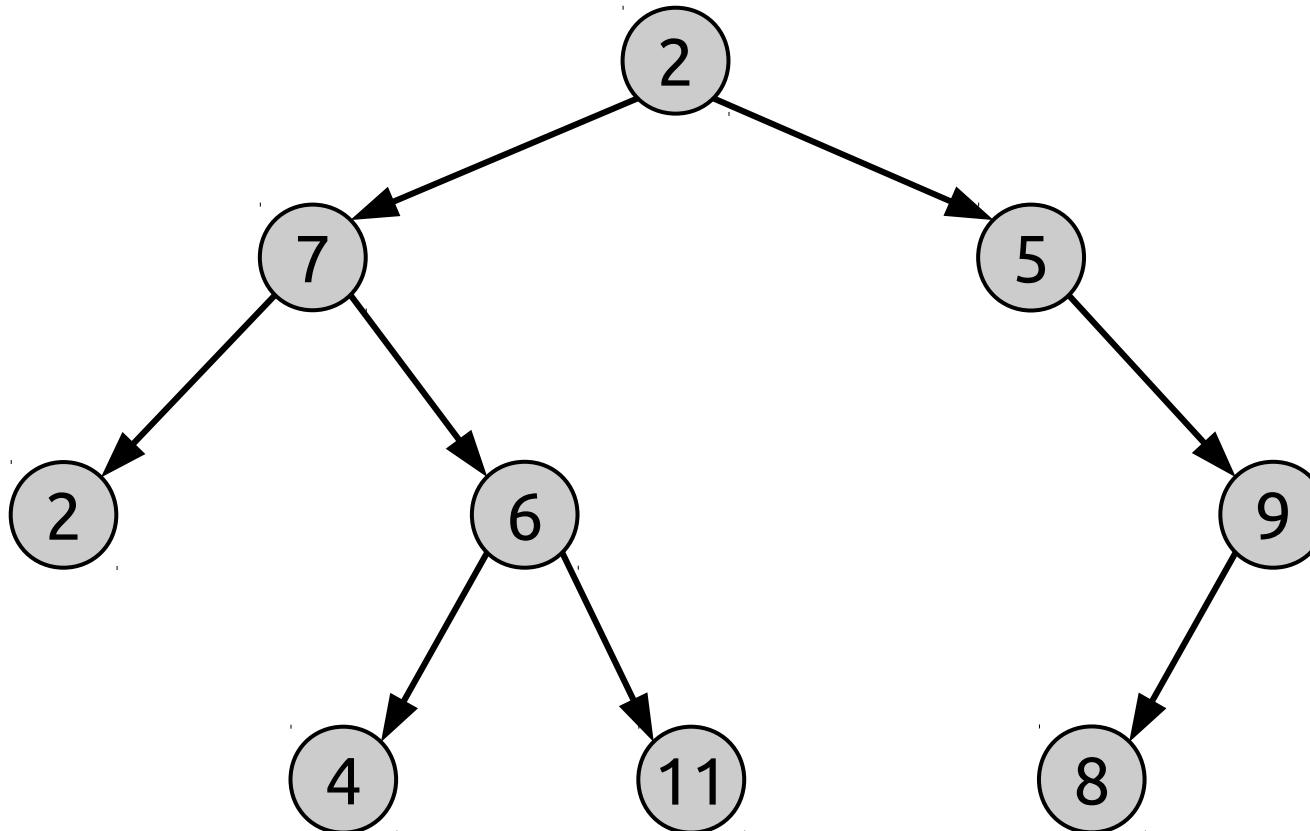
# Binary Trees

---

- ◆ Uma **binary tree** (árvore binária) é uma estrutura de dados em árvore na qual cada nó tem no máximo dois filhos, chamados de **filho esquerdo** e **filho direito**.
- ◆ Uma definição recursiva usando apenas noções da teoria de conjuntos é que uma árvore binária (não vazia) é uma tupla  $(L, S, R)$ , onde  $L$  e  $R$  são árvores binárias ou o conjunto vazio e  $S$  é um conjunto singleton contendo a raiz.
- ◆ Alguns autores permitem que a árvore binária também seja o conjunto vazio.

# Binary Trees

---



# Binary Trees

---

- ◆ Tipos de árvores binárias:
  - Uma árvore binária **rooted** possui um nó raiz e cada nó tem no máximo dois filhos.
  - Uma árvore binária **completa** é uma árvore em que cada nó tem 0 ou 2 filhos.
  - Uma árvore binária **perfeita** é uma árvore binária na qual todos os nós internos têm dois filhos e todas as folhas têm a mesma profundidade ou mesmo nível.
  - Uma árvore binária **balanceada** é uma estrutura de árvore binária na qual as subárvores esquerda e direita de cada nó diferem em altura em não mais do que 1.

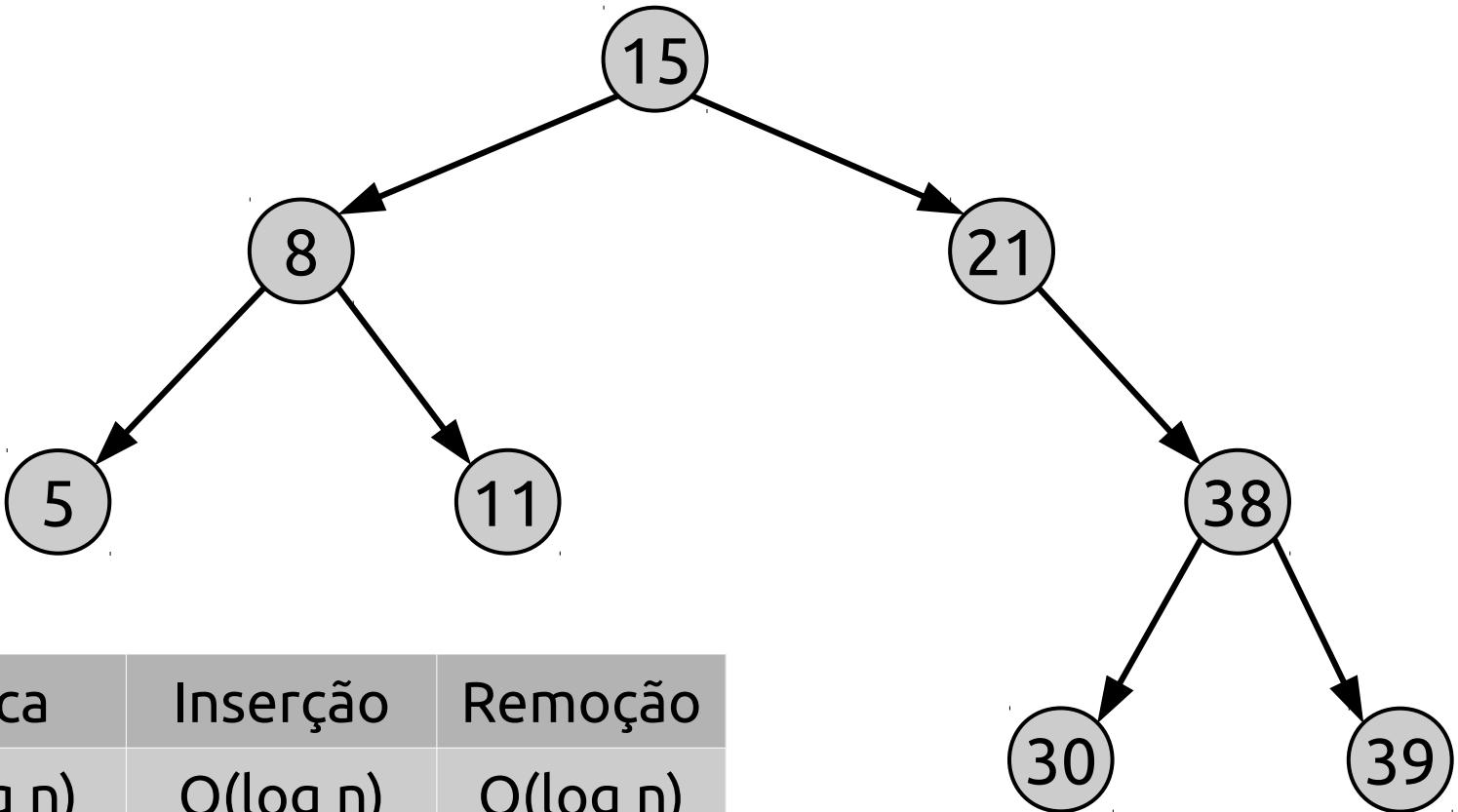
# Binary Search Trees

---

- Uma **binary search tree** (BST), também chamada de árvore binária ordenada, é uma árvore binária **rooted** em que cada um dos nós internos armazena uma chave maior do que todas as chaves na subárvore esquerda do nó e menor do que na subárvore direita.
- **Binary search trees** permitem pesquisa binária para busca, adição e remoção rápida de itens de dados e podem ser usadas para implementar conjuntos dinâmicos e tabelas de pesquisa.
- A ordem dos nós em um **BST** significa que cada comparação pula cerca da metade da árvore restante, portanto, toda a pesquisa leva um tempo proporcional ao logaritmo binário do número de itens armazenados na árvore. Isso é muito melhor do que o tempo linear necessário para localizar itens por chave em um **array** (não ordenado), mas mais lento do que as operações correspondentes em **hash tables**.

# Binary Search Trees

---



# Binary Search Trees

---

- Propriedades das Binary Search Trees:
  - Existem três tipos de travessias Depth-first search (DFS) para Binary Trees:
    - **Preorder** ([raiz](#), [esquerda](#), [direita](#));
    - **Inorder** ([esquerda](#), [raiz](#), [direita](#));
    - **Postorder** ([esquerda](#), [direita](#), [raiz](#)); tudo feito em tempo  $O(n)$ ;
  - a travessia inorder nos dá todos os nós da árvore em ordem crescente;
  - o nó mais à esquerda é o valor mínimo na BST e o nó mais à direita é o máximo;
  - Uma BST tem as vantagens de um array ordenado, mas a desvantagem da inserção logarítmica - todas as suas operações são feitas em tempo  $O(\log n)$ .

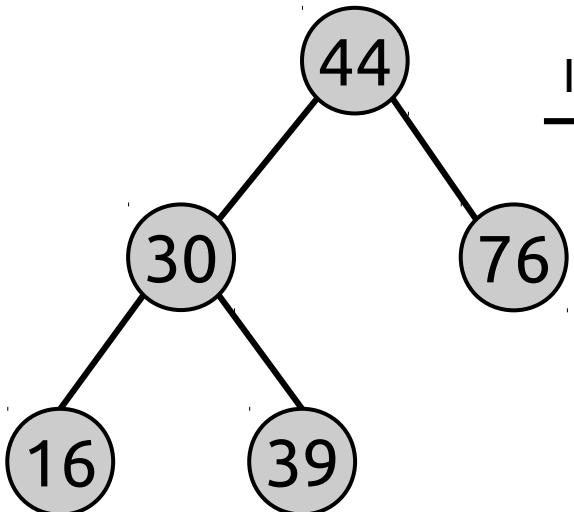
# AVL Trees

---

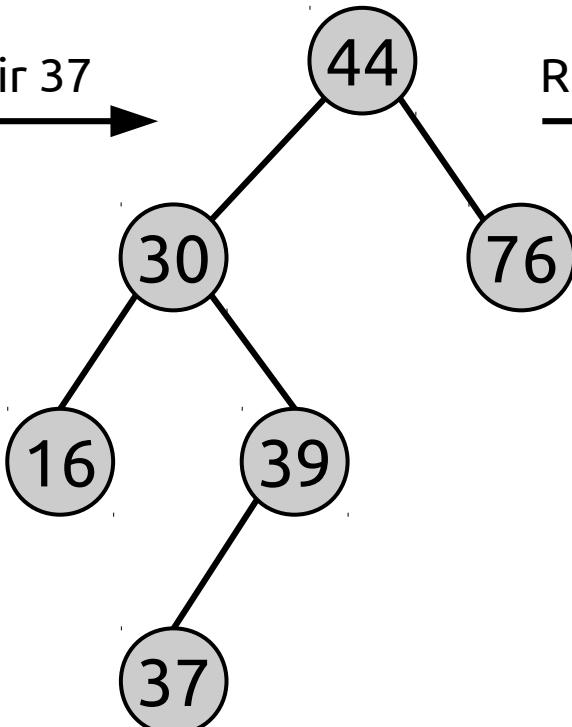
- Uma AVL tree (nomeada em homenagem aos inventores [Adelson-Velsky](#) e [Landis](#)) é uma árvore de busca binária com equilíbrio automático.
- Foi a primeira estrutura de dados desse tipo a ser inventada.
- Em uma árvore AVL, as alturas das duas subárvore filhas de qualquer nó diferem em no máximo um; se a qualquer momento eles diferirem em mais de um, o rebalanceamento é feito para restaurar essa propriedade.
- Consulta, inserção e exclusão levam tempo  $O(\log n)$  tanto na média quanto no pior caso, onde  $n$  é o número de nós na árvore antes da operação. As inserções e exclusões podem exigir que a árvore seja rebalanceada por uma ou mais [rotações](#) de árvore.

# AVL Trees

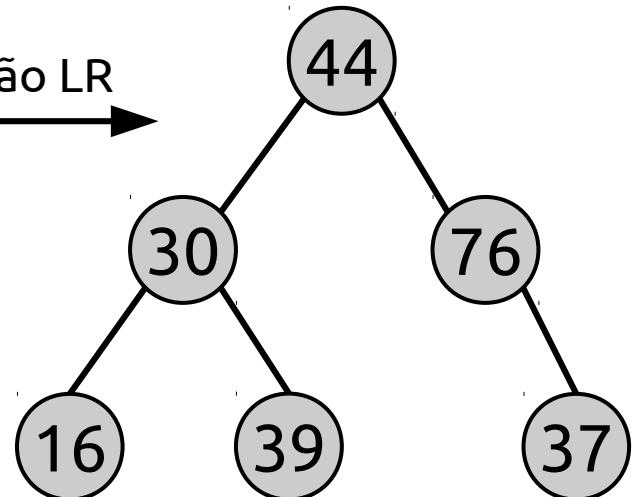
AVL Tree inicial



AVL Tree desequilibrada



AVL Tree equilibrada



Inserir 37

Rotação LR

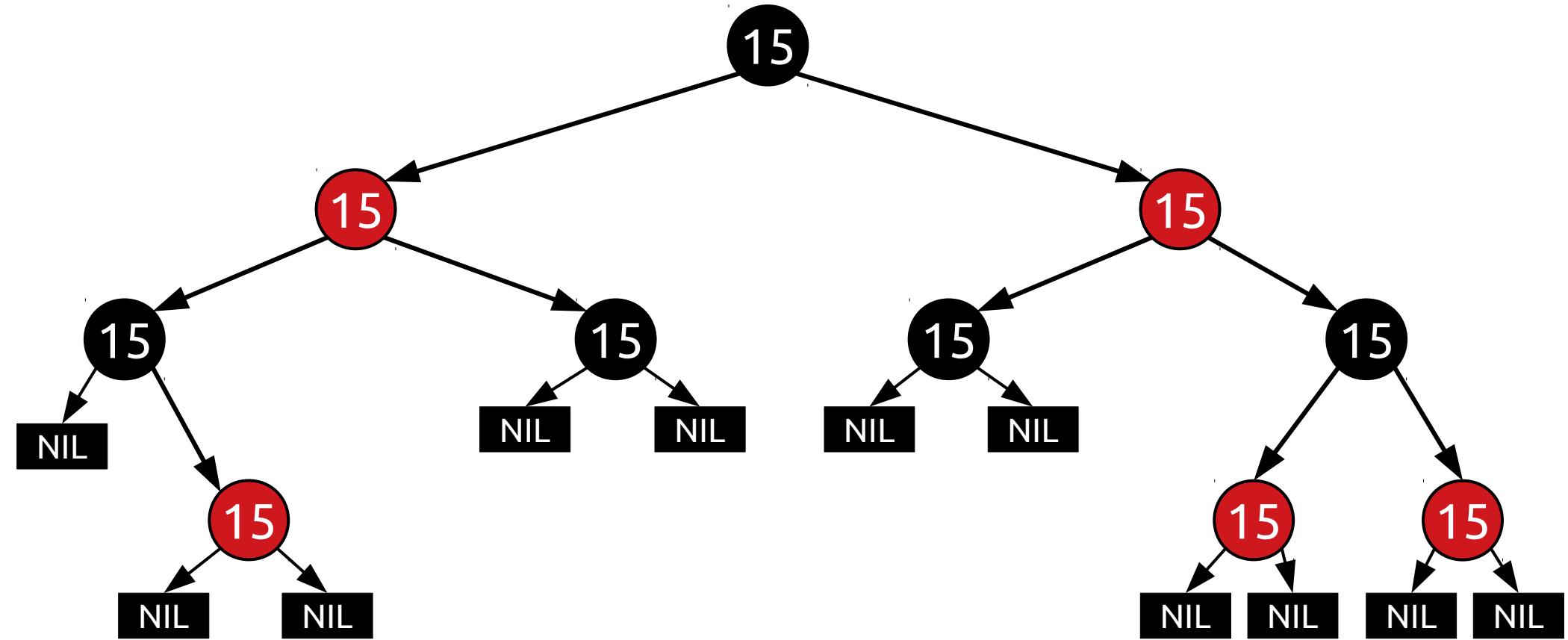
# Red-black Trees

---

- Uma **red-black tree** é uma espécie de árvore de busca binária com equilíbrio automático.
- Cada nó armazena um bit extra que representa a "cor" ("vermelho" ou "preto"), usado para garantir que a árvore permaneça equilibrada durante as inserções e exclusões.
- Quando a árvore é modificada, a nova árvore é reorganizada e "repintada" para restaurar as propriedades de coloração que restringem o quanto desequilibrada a árvore pode se tornar no pior caso.
- As propriedades são projetadas de forma que essa reorganização e recoloração possam ser executadas com eficiência.
- O re-balanceamento não é perfeito, mas garante a busca no tempo  **$O(\log n)$** , onde  $n$  é o número de nós da árvore. As operações de inserção e exclusão, junto com o rearranjo da árvore e recoloração, também são realizadas em tempo  **$O(\log n)$** .

# Red-black Trees

---



# Red-black Trees

---

- Além dos requisitos impostos a uma **binary search tree**, o seguinte deve ser satisfeito por uma **red–black tree**:
  - Cada nó é **vermelho** ou **preto**.
  - Todas as folhas **NIL** (figura anterior) são consideradas pretas.
  - Se um nó for vermelho, seus dois filhos serão pretos.
  - Cada caminho de um determinado nó para qualquer uma de suas folhas NIL descendentes passa pelo mesmo número de nós pretos.
  - **Árvores vermelho-pretas**, como todas as árvores de busca binárias, permitem acesso sequencial bastante eficiente (por exemplo, travessia **inorder**, isto é: na ordem Esquerda-Raiz-Direita) de seus elementos. Mas eles também suportam acesso direto assintoticamente ideal por meio de um percurso da raiz para a folha, resultando em tempo de pesquisa **O(log n)**.

# Tries

---

- Um **trie**, também chamado de **árvore digital** ou **árvore de prefixo**, é um tipo de árvore de pesquisa, uma estrutura de dados em árvore usada para localizar chaves específicas em um conjunto.
- Essas chaves são, na maioria das vezes, cadeias de caracteres, com links entre nós definidos não pela chave inteira, mas por caracteres individuais.
- Para acessar uma chave (para recuperar seu valor, alterá-lo ou removê-lo), o trie é percorrido em profundidade, seguindo os links entre os nós, que representam cada caractere da chave.
- Ao contrário de uma árvore de pesquisa binária, os nós no trie não armazenam sua chave associada. Em vez disso, a posição de um nó no trie define a chave com a qual está associado. Isso distribui o valor de cada chave pela estrutura de dados e significa que nem todo nó tem necessariamente uma chave associada.

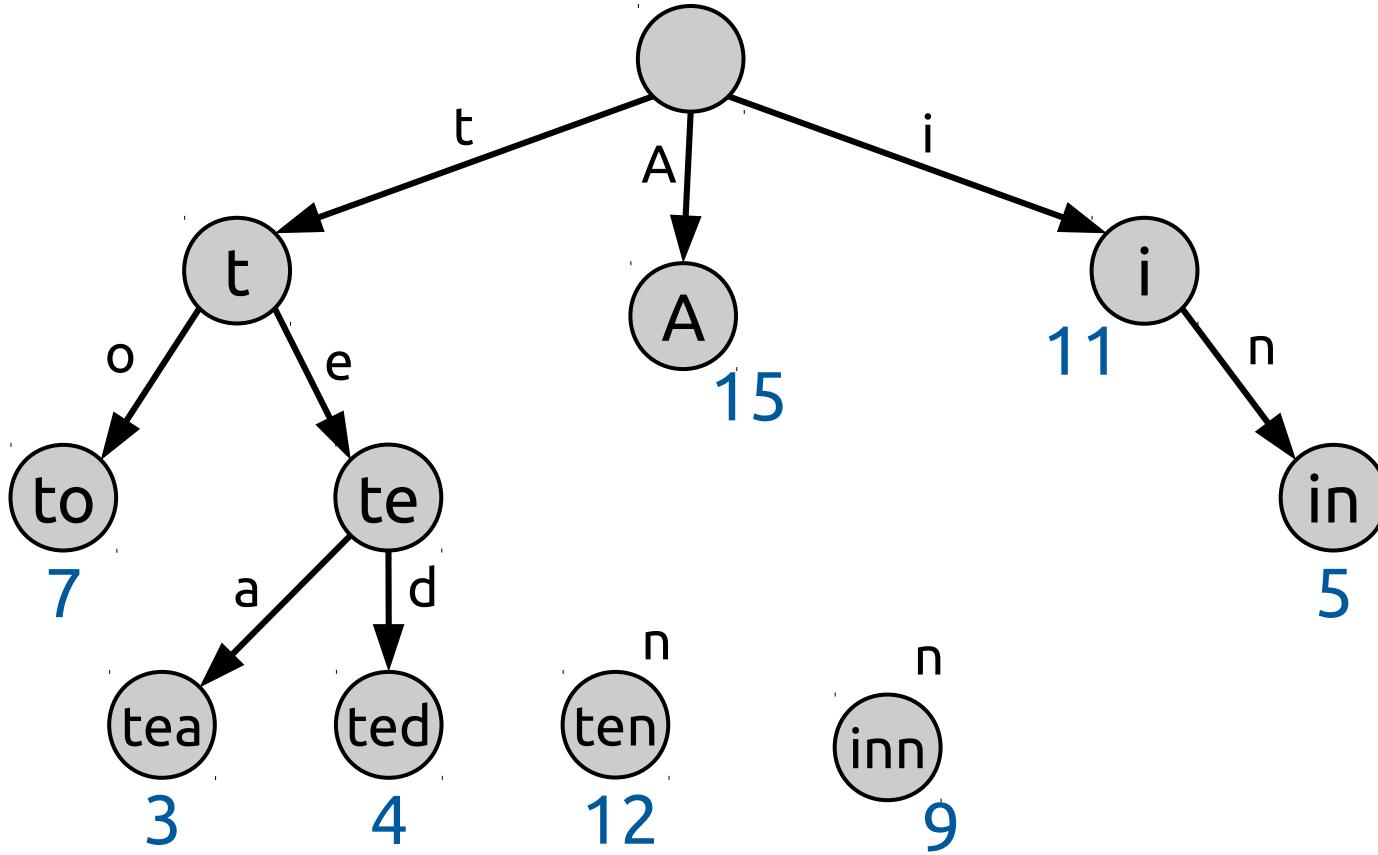
# Tries

---

- Todos os filhos de um nó têm um prefixo comum da string associada a esse nó pai, e a raiz está associada à string vazia.
- Essa tarefa de armazenar dados acessíveis por seu prefixo pode ser realizada de forma otimizada para memória, empregando uma **radix tree**.
- Embora as tries possam ser codificadas por cadeias de caracteres, elas não precisam ser. Os mesmos algoritmos podem ser adaptados para listas ordenadas de qualquer tipo subjacente, por exemplo, permutações de dígitos ou formas.
- A seguir temos um trie para as chaves "A", "to", "tea", "ted", "ten", "i", "in" e "inn". Cada palavra completa em inglês possui um valor inteiro arbitrário associado a ela.

# Tries

---



# Tries

---

- Propriedades das Tries:
  - Tem uma associação de valor-chave; a chave geralmente é uma palavra ou um prefixo dela, mas pode ser qualquer lista ordenada;
  - A raiz tem uma string vazia como chave;
  - a diferença de comprimento entre o valor de um nó e os valores de seus filhos é 1; dessa forma, os filhos da raiz armazenarão um valor de comprimento 1; como conclusão, podemos dizer que um nó  $x$  de um nível  $k$  tem um valor de comprimento  $k$ ;
  - A complexidade de tempo das operações de inserção / pesquisa é  $O(L)$ , onde  $L$  é o comprimento da chave, que é muito mais rápido do que o  $O(\log n)$  de um Binary Search Tree, mas comparável a uma hash table;

# Considerações Finais

---

“Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.” **Rob Pike**

“Algorithms + Data Structures = Programs”

**Niklaus Wirth**

# Mais Detalhes

---



<https://github.com/the-akira/IntroComp>

# Referências

---

- ◆ Introduction to Algorithms, 3rd Edition (The MIT Press)
- ◆ <https://en.wikipedia.org/wiki/Algorithm>
- ◆ [https://en.wikipedia.org/wiki/Sorting\\_algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm)
- ◆ <https://www.programiz.com>
- ◆ <https://www.geeksforgeeks.org>
- ◆ <https://www.mathsisfun.com>
- ◆ <https://www.w3resource.com/python-exercises>
- ◆ <https://runestone.academy/runestone/books/published/pythonds/index.html>