NumeroComplexo.cpp

```cpp
#include <iostream>
#include "NumeroComplexo.h"

using namespace std;

NumeroComplexo inicializa(int real, int img){
  NumeroComplexo num;
  num.real = real;
  num.img = img;
  return num;
}

void imprime (NumeroComplexo num){
  cout <<  "real " << num.real << " imaginário " <<
  num.img << endl;
}

void copia (NumeroComplexo *dst, NumeroComplexo src){
  dst->real = src.real;
  dst->img = src.img;
}

```

NumeroComplexo.cpp

```
NumeroComplexo soma(NumeroComplexo a, NumeroComplexo b){
  NumeroComplexo temp;
  temp.real = a.real + b.real;
  temp.img = a.img + b.img;
  return temp;
}

int ehReal(NumeroComplexo num){
  return num.img == 0;
}
```

Agenda

- Unidade 1: Conceito de algoritmos, estruturas de dados e programas
- Tipos de dados e tipos abstratos de dados
- **Como medir o tempo de execução de um programa**
- Introdução técnicas de análise de algoritmos

Introdução

Problema

Algoritmo A | **Algoritmo B** … **Algoritmo N**

Análise

O algoritmo B é o mais eficiente

O que é análise de algoritmos

- Segundo Cormen (2002), é a previsão dos recursos de que o algoritmo necessitará.
- Memória.
- Largura de banda de comunicação.
- Hardware de computação.
- Tempo de computação.

Estrutura de Análise

- Como analisar a eficiência de algoritmos?
  - Eficiência temporal: indica quão rápido um algoritmo em questão é executado
  - Eficiência espacial: está relacionada com o espaço extra que o algoritmo necessita

Estrutura da Análise

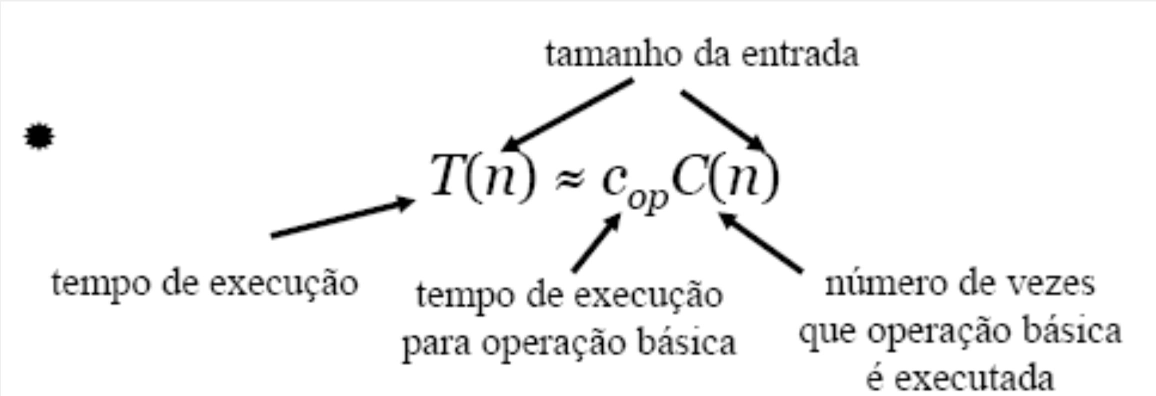
- Antigamente, ambos recursos – tempo e espaço – eram importantes
- Atualmente, a quantidade extra de espaço requerida por um algoritmo não é tão importante, ainda que exista um diferença entre memória principal, secundária e cache
- Por outro lado, o tempo continua sendo importante
  - Problemas cada vez mais complexos
  - Abordaremos somente eficiência temporal

Modelo RAM

- Instruções executadas uma após a outra, sem operações concorrentes.
- Instruções encontradas em computadores reais, tais como instruções aritméticas, de movimentação de dados e de controle.

Tempo de execução

- Eficiência temporal é analisada determinando o número de repetições de operações básicas com uma função do tamanho de entrada
- Operação básica: a operação que contribui mais para o tempo de execução do algoritmo



Tempo de Execução

- Atenção, pois, C(n) não contém informação sobre operações que não são básicas.
- C(n) é calculado com aproximação
- A constante c_{op} também é uma aproximação.
- Somente é possível fazer uma estimativa do tempo de execução do algoritmo se n for extremamente grande ou pequeno.

Tamanho da entrada

- Quase todos os algoritmos levam mais tempo para ser executado sobre entrada maiores
- Assim, é obvio investigar a eficiência de algoritmos como função do parâmetro n que indica o tamanho da entrada do algoritmo

Tempo de Execução

- Por que não utilizar unidade de tempo?
  - Dependência do hardware, qualidade da implementação, compilador, etc
- Métrica que não dependa de fatores externos. Alternativas
  - Contar o número de vezes que cada operação do algoritmo realiza
  - Identificar a operação mais importante do algoritmo(operação básica), a operação que mais contribui para o tempo de execução total e contar o número de vezes que a operação é realizada

Tempo de execução

- Geralmente, a operação que consome mais tempo está no laço mais interno do algoritmo

Tamanho da Entrada e Operação Básica

Problema	Medida do Tamanho da Entrada	Operação Básica
Busca por uma chave em uma lista de n itens	Número de itens na lista	Comparação de chaves
Multiplicação de duas matrizes de números de pontos flutuantes	Dimensões das matrizes	Multiplicação de Ponto Flutuante
Calcular a^n	n	Multiplicação de Ponto Flutuante
Grafos	Número de vértices e/ou arestas	Visitar um vértice ou atravessar uma aresta

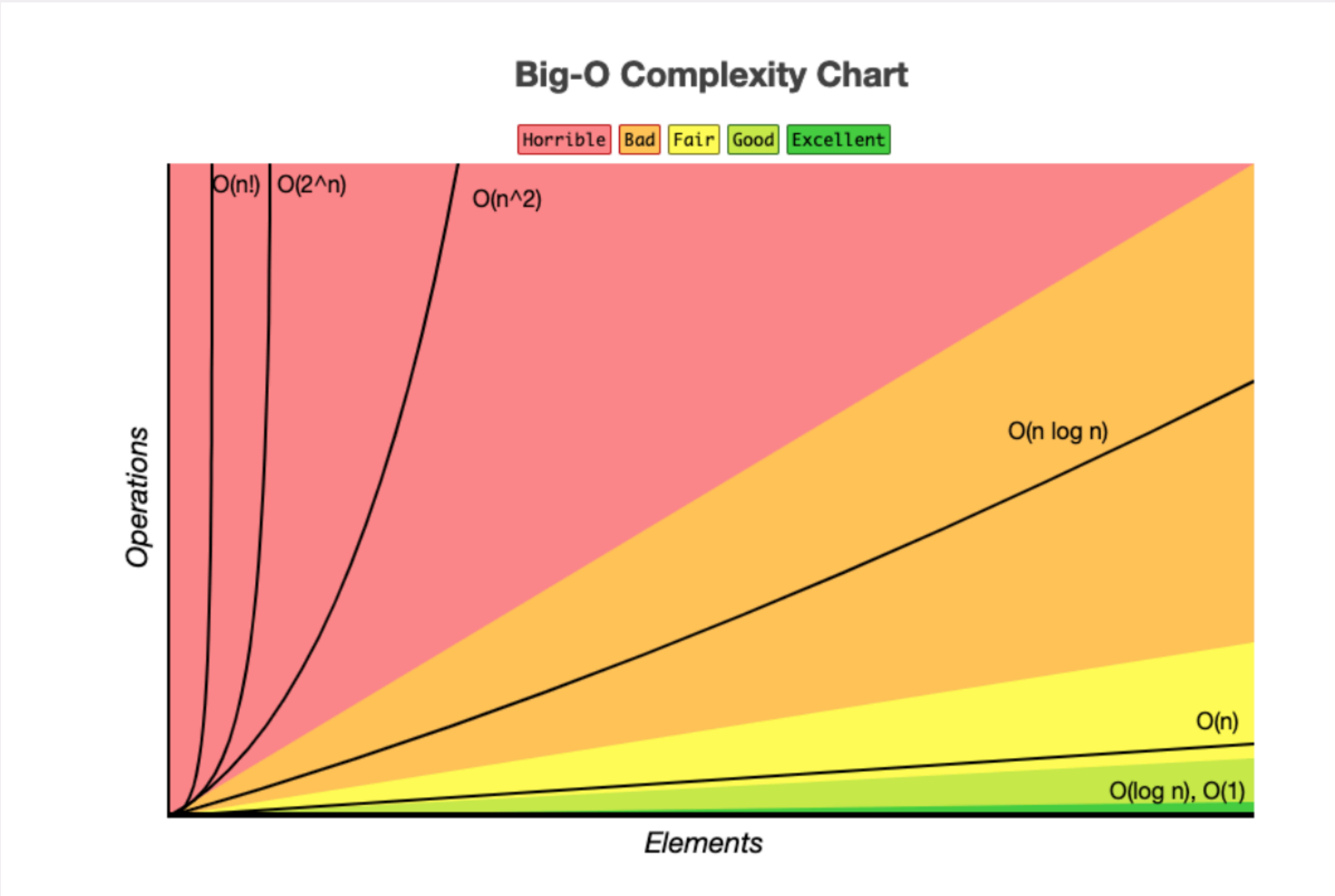
Ordens de Crescimento

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

- A função logarítmica cresce mais lentamente
- As funções exponenciais e fatoriais crescem rapidamente

Ordens de Crescimento



Tempo de execução

- O tempo de execução de um algoritmo aumenta proporcionalmente ao tamanho da entrada n do problema.
- Escolha do algoritmo pelo desempenho em entradas grandes.

Crescimento de Funções

- Suponha que Fulano fez um algoritmo max, chamado de maxFulano.
- Suponha que Beltrano fez outro algoritmo max, chamado maxBeltrano.
- Como saber qual dos 2 algoritmos é mais eficiente?

Crescimento de Funções

- Podemos cronometrar o tempo?
  - Sim! Mas não é uma boa ideia. Por que?
  - Porque temos que medir o tempo de maxFulano e maxBeltrano para 1 elemento, 2 elementos, ..., 10 elementos, ..., 100 elementos, ..., 1000 elementos
  - O resultado não pode depender do compilador, do hardware, do sistema operacional etc.
- O que é feito em geral
  - Define-se o tamanho da entrada como um número
  - Exemplo. Para max, o tamanho da entrada é quantidade de elementos: n

Crescimento de Funções

- O que é feito em geral:
  - Definem-se as operações mais lentas (mais caras computacionalmente).
- Tipicamente: os comandos de comparação (por exemplo, maximo < a_i)
  - Dependendo do algoritmo, pode-se eleger outras operações como a mais cara. Por exemplo, a operação de multiplicação

Crescimento de Funções

- O que feito em geral:
  - Calcula-se, para o algoritmo a ser analisado, quantas operações caras são executadas dado uma entrada de tamanho n.
  - Exemplo. Suponha que o algoritmo maxFulano executa uma quantidade de comparações de acordo com a função $f(n) = n^2 + 2n + 1$.
  - Ou seja, se passarmos 50 elementos para maxFulano, ele fará $f(50) = 50^2 + 2 . 50 + 1 = 2601$ comparações.
  - Como descobre-se f(n)? Não se preocupe ainda. Assuma que é dado de graça.

Crescimento de Funções

- Resumindo: a unidade de comparação entre 2 algoritmos é uma função que calcula o número de operações caras dado o tamanho de uma entrada
  - Pode-se também analisar quanto de memória é usado por tamanho de entrada

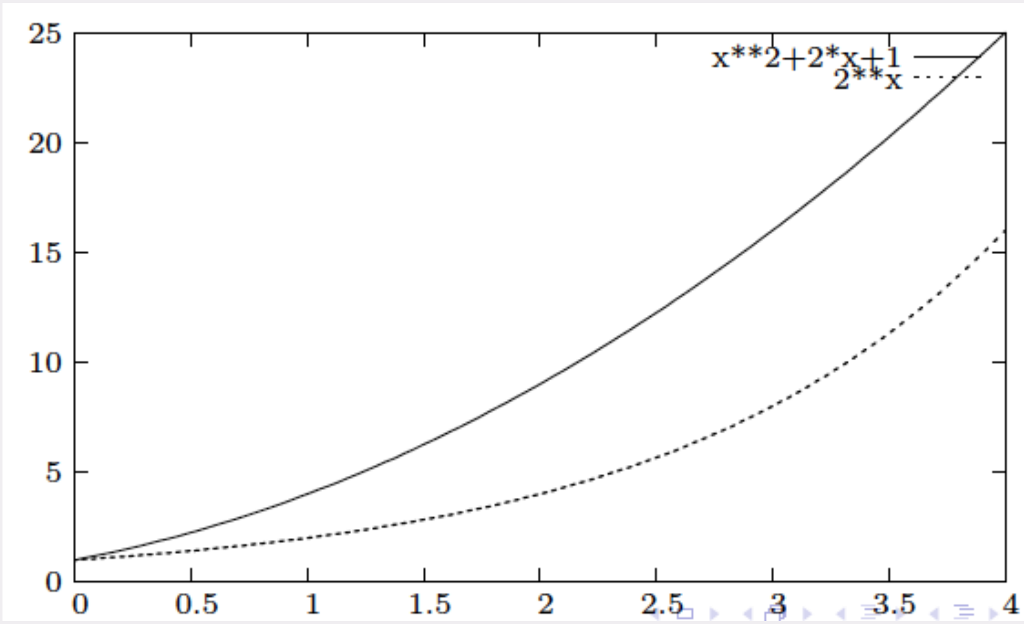
- Esta função define a complexidade do algoritmo.

Crescimento de Funções

- Seja a complexidade de $\max_{Fulano}$ $f(n)=n^2+2n+1$
- Seja a complexidade de maxBeltrano $g(n) = 2^n$
- Qual dos 2 é mais eficiente?
- Exercício: faça o gráfico de f(n) e g(n) para $0 \leq n \leq 4$

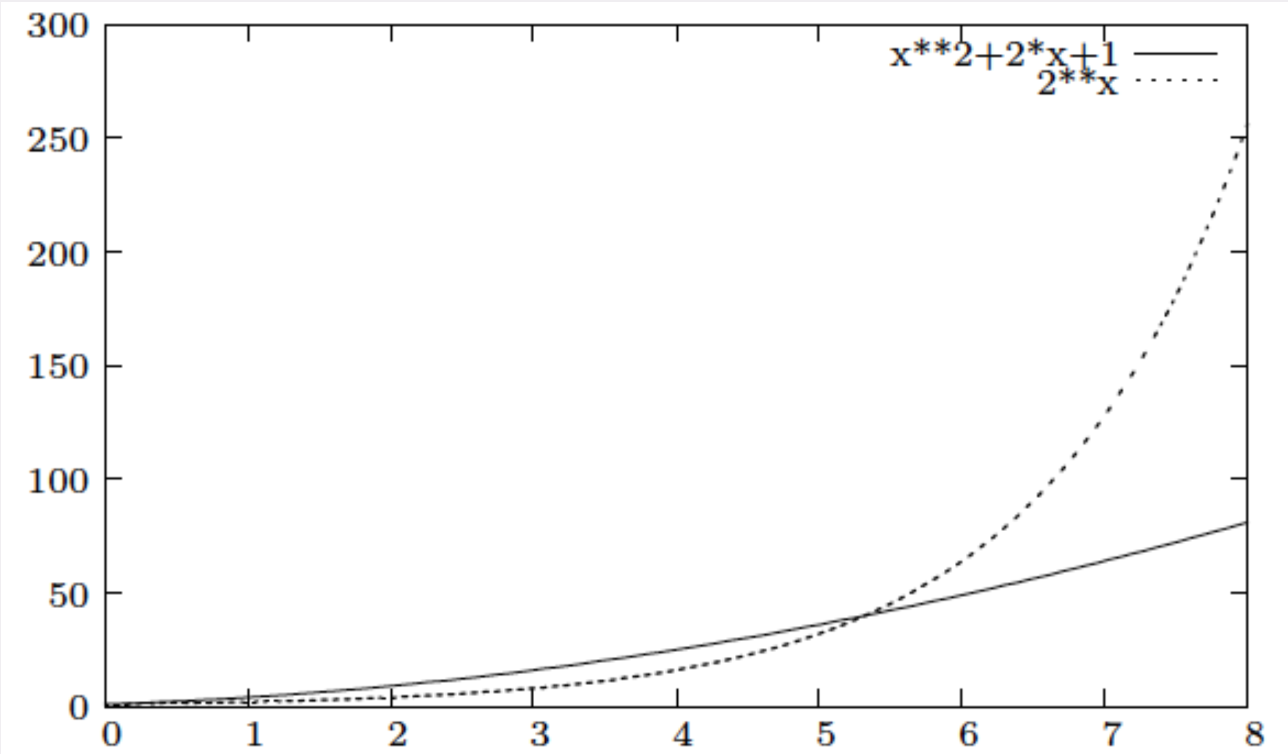
Crescimento de Funções

- Seja a complexidade de maxFulano f(n)=n2+2n+1
- Seja a complexidade de maxBeltrano g(n) = 2n
- Qual dos 2 é mais eficiente?
- Exercício: faça o gráfico de f(n) e g(n) para 0 ≤ n ≤ 4



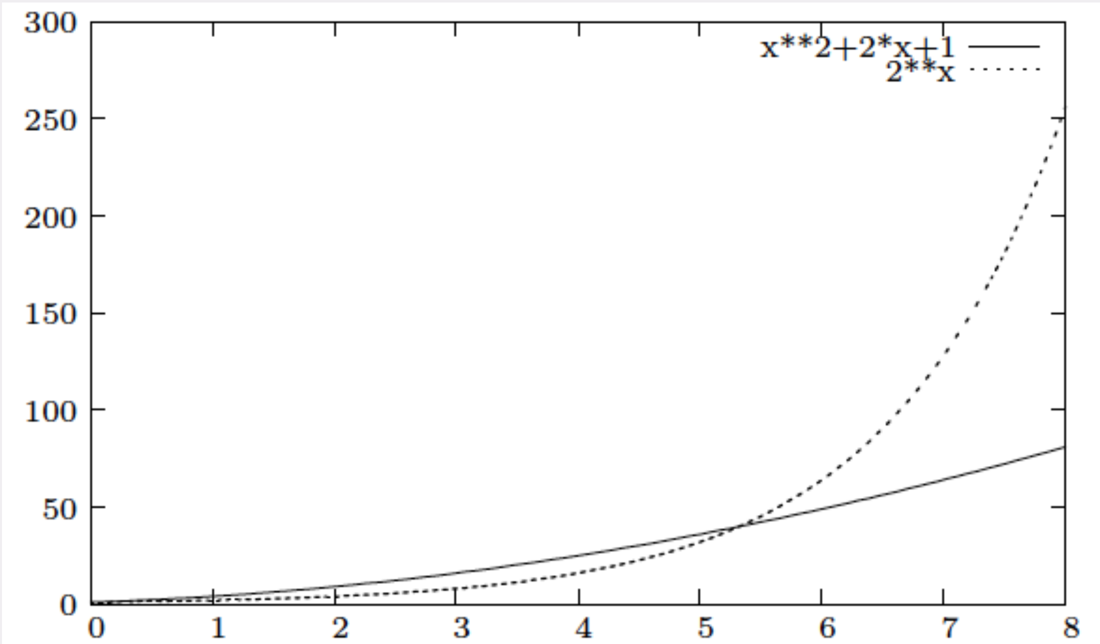
Crescimento de Funções

- Veja o gráfico para $0 \leq n \leq 8$
- Quem é mais eficiente? $f(n) = n^2 + 2n + 1$ ou $g(n) = 2^n$?



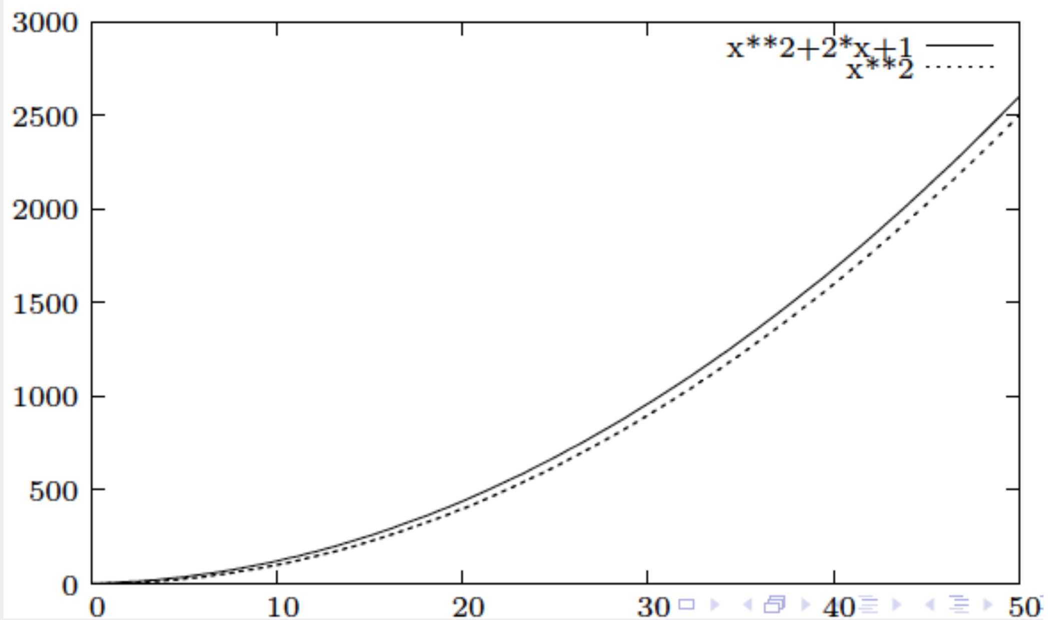
Crescimento de Funções

- Veja o gráfico para $0 \leq n \leq 8$
- Quem é mais eficiente? f(n) = n2 + 2n + 1 ou g(n) = 2n?
- A longo prazo, f(n) é mais eficiente
- Longo prazo significa para todo $n > 6$



Crescimento de Funções

- Veja o gráfico para $0 \leq n \leq 50$
- Quem é mais eficiente? $f(n) = n^2+2n+1$ ou $g(n)=n^2$?
- Não há uma clara vantagem a longo prazo
- Eles se diferenciam por uns trocados
- Neste caso, os 2 algoritmos são considerados equivalentes



Crescimento de Funções – Notação Big-O

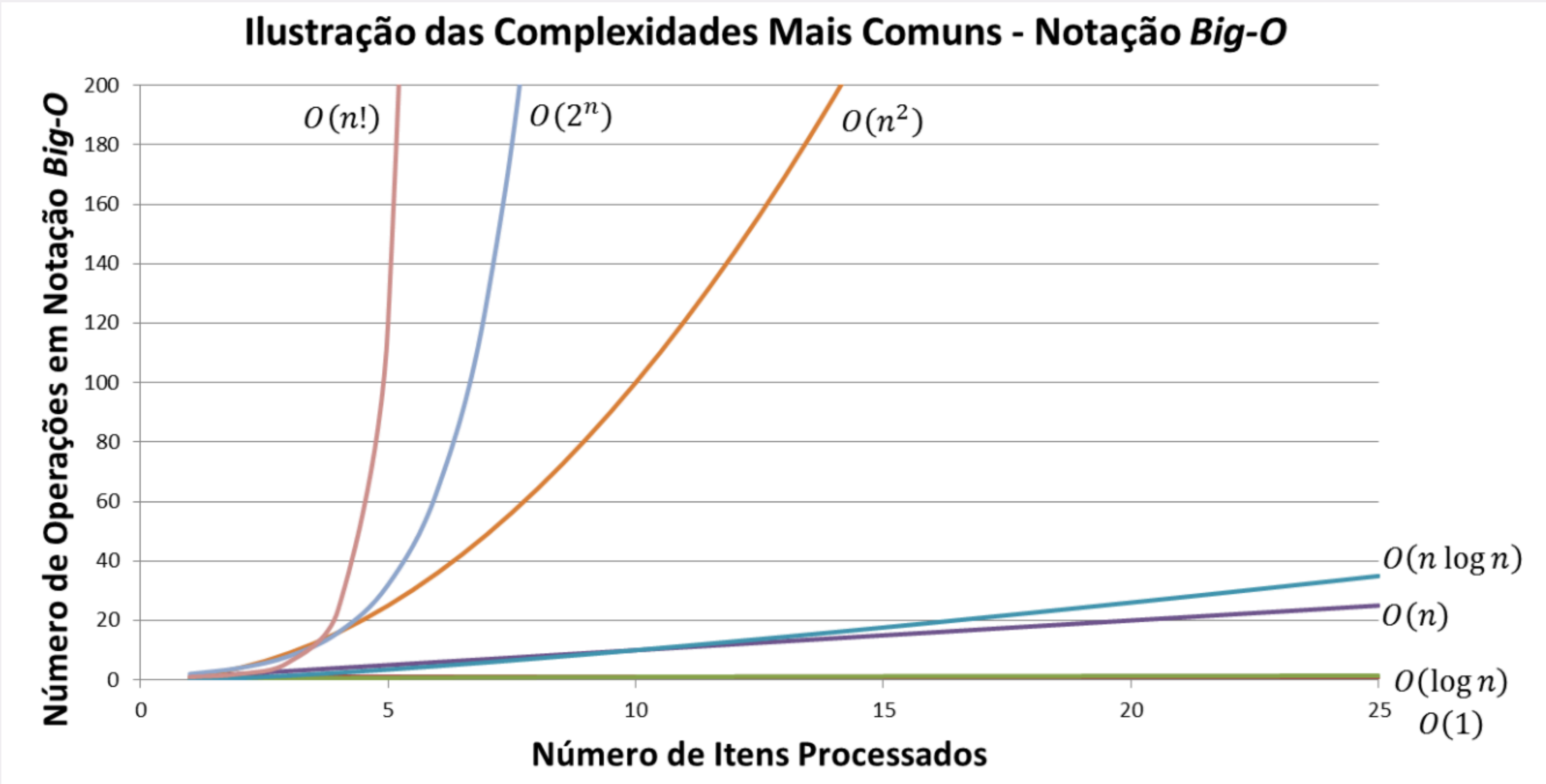
$\boxed{\begin{matrix} 1 \\ \vdots \end{matrix}}$ é melhor que $\boxed{\begin{matrix} log\ \ n \\ \vdots \end{matrix}}$ é melhor que $\boxed{\begin{matrix} n \\ \vdots \end{matrix}}$ ...

... $\boxed{\begin{matrix} nlog\ \ n \\ \vdots \end{matrix}}$ é melhor que $\boxed{\begin{matrix} n^2 \\ \vdots \end{matrix}}$ é melhor que $\boxed{\begin{matrix} n^3 \\ \vdots \end{matrix}}$ ...

... $\boxed{\begin{matrix} 2^n \\ \vdots \end{matrix}}$ é melhor que $\boxed{\begin{matrix} n! \\ \vdots \end{matrix}}$

Obs. Existem mais classes de complexidade que nesta figura!
Por exemplo, $\sqrt{n}$, n^4, n^5 etc.

Crescimento de Funções – Notação Big-O



Crescitmento de Funções-Conclusões

- Em geral, um algoritmo que assintoticamente mais eficiente será a melhor escolha para todas as entradas, exceto as muito pequenas.
- A notação Big-O é usada para estimar o número de operações que um algoritmo precisa realizar dado o crescimento dos seus dados de entrada.

Crescitmento de Funções-Conclusões

- Com ela é possível determinar se é prático usar um algoritmo, ou mesmo comparar dois algoritmos para determinar qual é o mais eciente.
- Exemplo. Um algoritmo usa $7n^2$ operações e outro usa n^3 operações, com a notação Big-O concluímos que o primeiro realiza um número menor de operações quando n é grande.

Melhor Caso, Pior Caso e Caso Médio

- Melhor caso: menor tempo de execução sobre todas as entradas de tamanho n.
- Pior caso: maior tempo de execução sobre todas as entradas de tamanho n.
- Se f é uma função de complexidade baseada na análise de pior caso, o custo de aplicar o algoritmo nunca é maior do que f(n).
- Caso médio (ou caso esperado): média dos tempos de execução de todas as entradas de tamanho n.

Melhor Caso, Pior Caso e Caso Médio

- Na análise do caso esperado, uma distribuição de probabilidades sobre o conjunto de entradas de tamanho n é suposta e o custo médio é obtido com base nessa distribuição.
- A análise do caso médio é geralmente muito mais difícil de obter do que as análises do melhor e do pior caso.
- É comum supor uma distribuição de probabilidades em que todas as entradas possíveis são igualmente prováveis.
- Na prática isso nem sempre é verdade.

Perguntas?