Noções de Complexidade

Notação *O* e Análise Assintótica

ALGORITMOS E ESTRUTURA DE DADOS I

Profa. Andréa Aparecida Konzen
Escola Politécnica - PUCRS

Baseado no material cedido pelas professoras Isabel Manssour e Daniela Amaral e nas referências bibliográficas

Agenda

1. Análise e Complexidade de Algoritmos
2. Análise Assintótica
3. Notação Big O
4. Diferentes Classes de Complexidade

Um **algoritmo** é um conjunto finito de instruções a serem executadas para resolver um problema

Ao fazer uma análise de algoritmos deve-se considerar várias características

Quais critérios podemos usar para avaliar um algoritmo?

Podemos considerar critérios de análise de um algoritmo:

- tempo de processamento
- espaço de memória ocupado
- tamanho do código desenvolvido
- eficácia da solução
- legibilidade de código fonte
- precisão de processamento de dados
- manutenibilidade do código
- taxa de possibilidade de reuso

Entre outros...

Dentre tantos critérios, como identificar o melhor desempenho?

Complexidade de algoritmos

A **complexidade** de um algoritmo é determinada pelos recursos de **tempo** e **espaço** exigidos.

Assim, deve-se **prever** o **crescimento dos recursos exigidos por** um **algoritmo à medida que** o **tamanho dos dados de entrada** crescem. Um algoritmo **mais eficiente** **exige** **menos recursos**.

tempo

gasto na execução é outra característica de desempenho.

O quão eficiente é um algoritmo?

espaço

ocupado é uma característica de desempenho.

O quanto o processamento do algoritmo ocupa da memória?

tempo

O quão eficiente é um algoritmo?

Eficiência

Uma operação com o menor desperdício de recursos.

Fazer as coisas de maneira otimizada, utilizando o mínimo de recursos possíveis para alcançar um determinado objetivo.

Então... Como analisar a eficiência de algoritmos em relação ao tempo?

Tempo de processamento **depende** de uma série de fatores:

- hardware
- software
- tamanho
- tipo da entrada de dados

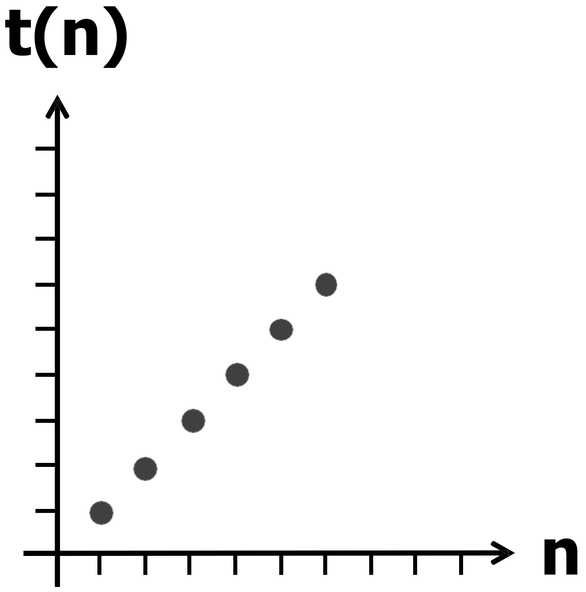
Assim, <u>não se pode considerar o tempo de execução em medidas tradicionais de tempo</u>

Como resolver isso?

usando...

Ordem de grandeza

Uma **notação científica** para representação da **complexidade** em relação ao **tempo de processamento** ***versus*** a **entrada de dados**.



Então...

Análise assintótica

- Metodologia utilizada para descrever e comparar o desempenho de algoritmos

- Descreve o comportamento geral do algoritmo, conforme os dados crescem

- Refere-se ao **crescimento do número de operações,** conforme **cresce o número de elementos processados**

Análise de Algoritmos e Complexidade

- Essa análise independe de linguagem de programação
- **Pode ser feita diretamente sobre o pseudocódigo** e consiste em **contar quantas operações primitivas** são executadas
- A medição é feita por **ordem de grandeza** que se refere a **quantidade de passos executados pelo algoritmo**

PSEUDOCÓDIGO

```
Iniciar soma como 0
    Para cada número de 1 a 10
    Adicionar número à soma
    Fim do loop
    Imprimir soma
```

LINGUAGEM JAVA

```java
public class SomaNumeros {
    public static void main(String[] args) {
        int soma = 0;
        for (int numero = 1; numero <= 10; numero++) {
            soma += numero;
        }
        System.out.println(soma);
    }
}
```

LINGUAGEM PYTHON

```python
soma = 0
for numero in range(1, 11):
    soma += numero
print(soma)
```

Análise assintótica

Tipos de Complexidade

Pior

$O(n!)$	Fatorial
$O(a^n)$	Exponencial
$O(n^3)$	Cúbica
$O(n^2)$	Quadrática
$O(n \log n)$	Linearítmico
$O(n)$	Linear
$O(\log n)$	Logarítmica
$O(1)$	Constante

Melhor

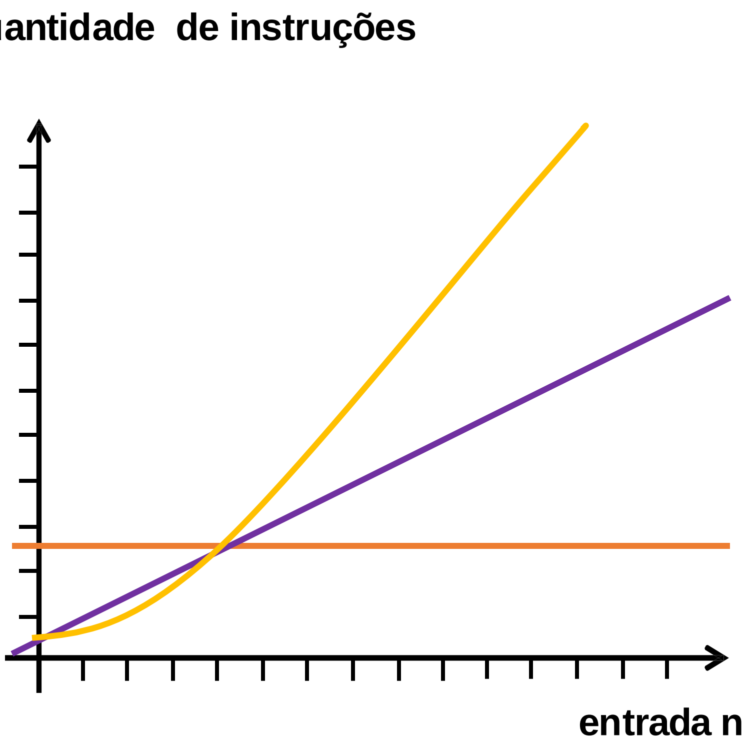
Operações Primitivas

Operações básicas e fundamentais que servem como blocos de construção para algoritmos e programas

Exemplos

- Aritméticas: Adição, Subtração, Multiplicação, Divisão
- Lógicas: E lógico (AND), Ou lógico (OR)
- Comparações: Igualdade (==), Diferente (!=), Maior que (>), Menor que (<)
- Atribuições: Atribuição de valor (=),
- Controle de fluxo: Instruções condicionais (if, else), Laços (for, while)
- Manipulação de memória: Acesso a variáveis, Alocação e desalocação de memória

Quanto mais rapidamente crescer o número de operações para processar os itens, **pior é o desempenho** do algoritmo. Mas, determinar o **tempo médio** é difícil. Portanto, para análise de algoritmos quase sempre se quer saber **o limite superior**, ou seja, **o pior caso**.



Complexidade Computacional:

•A complexidade computacional de um algoritmo determina o crescimento do número de operações em função do tamanho do *input* (n).

•Algoritmos com complexidade **O(1)** ou **O(log n)** são mais eficientes porque o número de operações cresce muito lentamente à medida que n aumenta.

•Algoritmos com complexidade **O(n)**, **O(n log n)**, **O(n²)**, ou pior (como **O(2^n)** ou **O(n!)**) têm um crescimento muito mais rápido no número de operações, tornando-os menos eficientes.

Impacto do Crescimento Rápido:

- Quando um algoritmo possui uma **alta complexidade**, como **$O(n^2)$**, o **número de operações** necessárias para processar os itens **cresce quadraticamente**. Ou seja, se o tamanho do input dobrar, o número de operações quadruplicará.

- Para complexidades exponenciais, como **$O(2^n)$**, um pequeno aumento no tamanho do input resulta em um aumento exponencial no número de operações, tornando o algoritmo impraticável para inputs relativamente pequenos.

Desempenho Prático:

- Na prática, algoritmos com complexidade alta exigem mais tempo de processamento e mais recursos computacionais (memória, CPU), o que resulta em pior desempenho.
- Gerando um **problema para grandes conjuntos de dados**, onde o tempo de execução pode se tornar impraticável.

Portanto, quanto mais rapidamente cresce o número de operações para processar os itens, pior é o desempenho do algoritmo, devido ao aumento significativo na quantidade de recursos necessários para realizar essas operações.

Este é um princípio fundamental na análise de algoritmos e é essencial para projetar soluções eficientes para problemas computacionais.

Assim, adota-se a...

Notação Big O

Referente a **complexidade de tempo** (*Time Complexity*), ou seja, tempo de execução de um algoritmo que está relacionado ao número de operações executadas

Representa **sempre o pior caso** do algoritmo

Usa-se a letra **O** seguida de uma função sobre **n** que descreva esse crescimento do algoritmo

Mas, por que representa **sempre o pior caso** do algoritmo?

Por que representar **sempre o pior caso** do algoritmo?

A notação Big O é focada no pior caso do algoritmo por várias razões:

1. **Garantia de Desempenho**: Em situações críticas, como sistemas em tempo real, é essencial garantir que o algoritmo não ultrapasse um determinado limite de tempo.
O pior caso oferece uma garantia sólida de desempenho máximo.

Por que representar **sempre o pior caso** do algoritmo?

2. Análise Simplificada: Considerar o pior caso simplifica a análise, pois evita a necessidade de considerar todos os possíveis cenários de entrada.

Facilita a comparação entre diferentes algoritmos.

Por que representa **sempre o pior caso** do algoritmo?

3. Consistência: Focar no pior caso assegura que a análise seja consistente, independentemente dos dados específicos de entrada. É útil ao comparar algoritmos diferentes, onde alguns podem ter desempenho consistentemente ruim para certos tipos de dados

Por que representa **sempre o pior caso** do algoritmo?

4. Prevenção de Ataques: Em contextos de segurança, analisar o pior caso pode ajudar a identificar vulnerabilidades onde um algoritmo pode ser explorado deliberadamente com dados que causam o pior desempenho

Ao ver expressões, normalmente se pensa em valores pequenos. Mas, na análise assintótica de algoritmos, deve-se ignorar os valores pequenos e concentrar-se nos valores elevados de n.

As funções:

n^2

$(3/2)n^2$

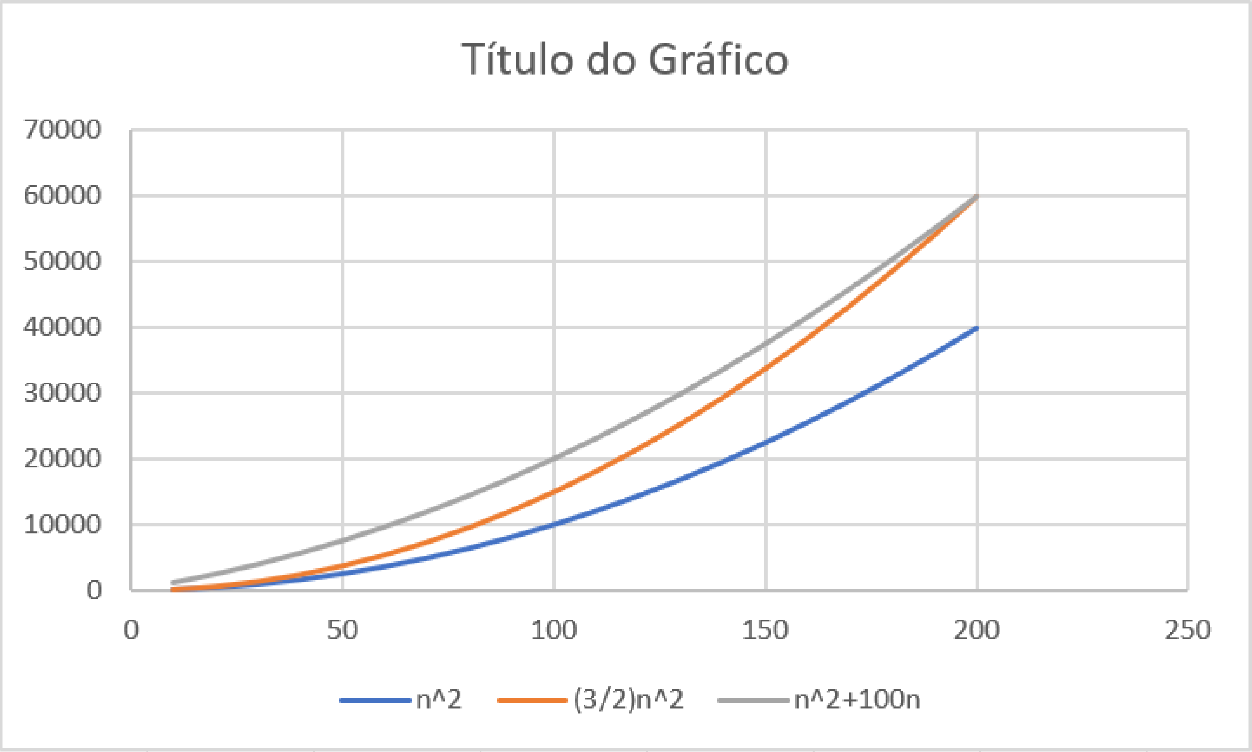
n^2+100n

Crescem da mesma forma?

Análise de Algoritmos e Complexidade

2	n	n^2	(3/2)n^2	n^2+100n
3	10	100	150	1100
4	20	400	600	2400
5	30	900	1350	3900
6	40	1600	2400	5600
7	50	2500	3750	7500
8	60	3600	5400	9600
9	70	4900	7350	11900
10	80	6400	9600	14400
11	90	8100	12150	17100
12	100	10000	15000	20000
13	110	12100	18150	23100
14	120	14400	21600	26400
15	130	16900	25350	29900
16	140	19600	29400	33600
17	150	22500	33750	37500
18	160	25600	38400	41600
19	170	28900	43350	45900
20	180	32400	48600	50400
21	190	36100	54150	55100
22	200	40000	60000	60000

Sim!



Conforme n aumenta, a expressão $3/2\ n^2$ eventualmente ultrapassa n2+100n. Isso ocorre porque o termo quadrático se torna dominante em comparação ao termo linear em valores altos de n

ESCOLA POLITÉCNICA
PUCRS

A tabela e o gráfico na imagem mostram a relação entre diferentes valores de n e três diferentes expressões matemáticas:

A tabela é organizada em cinco colunas:

1. **n**: Representa o valor de n, variando de 10 a 200, incrementado de 10 em 10.
2. **n^2**: O valor de n elevado ao quadrado.
3. **$3/2\ n^2$**: O valor de n elevado ao quadrado e multiplicado por 32\frac{3}{2}23.
4. **$n^2 + 100n$**: O valor de n elevado ao quadrado somado a 100 vezes o valor de n.

Exemplo

Quando $n = 10$:

- $n^2 = 100$
- $\frac{3}{2}n^2 = 150$
- $n^2 + 100n = 1100$

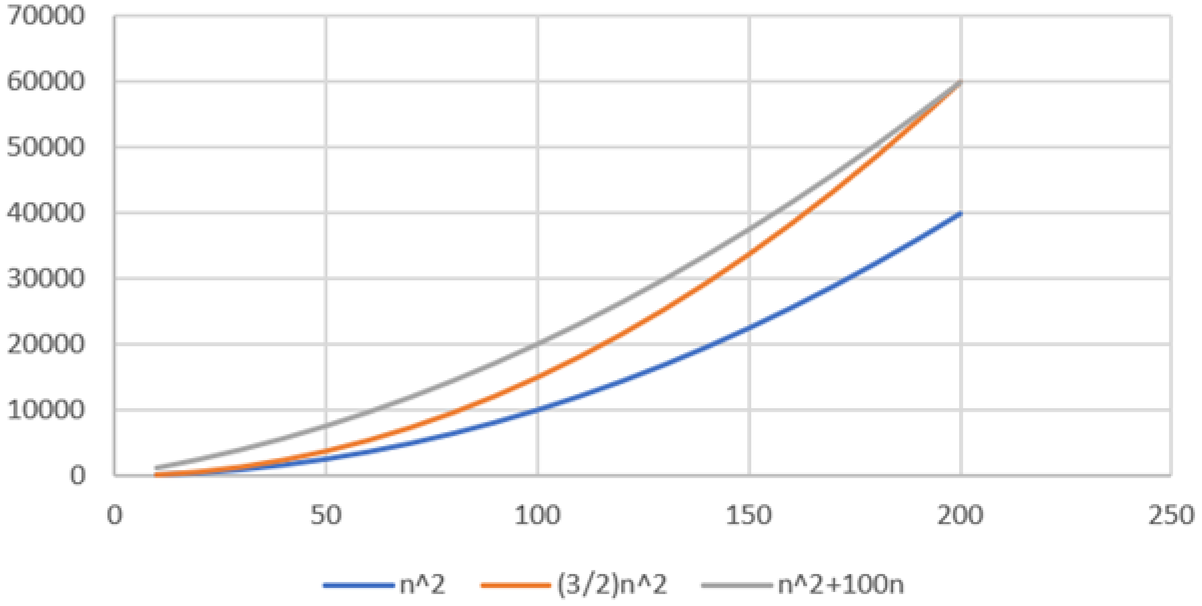
Quando $n = 100$:

- $n^2 = 10000$
- $\frac{3}{2}n^2 = 15000$
- $n^2 + 100n = 20000$

Gráfico

O gráfico ilustra essas três expressões matemáticas com o eixo x representando os valores de n e o eixo y representando os valores calculados das expressões

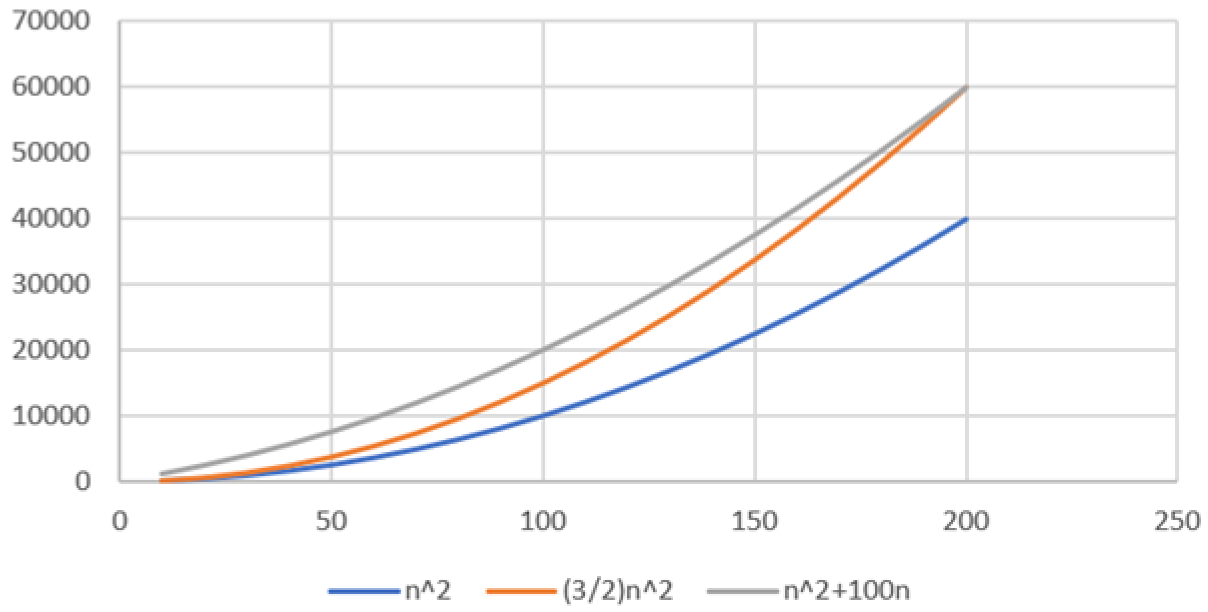
- **Linha Azul**: Representa n^2
- **Linha Laranja**: Representa $3/2n^2$
- **Linha Cinza**: Representa n^2+100n



n^2 (linha azul) cresce de forma quadrática, o que é esperado para uma função n^2.

$\frac{3}{2}n^2$ (linha laranja) cresce mais rapidamente que n^2, pois está sendo multiplicada por $\frac{3}{2}$.

$n^2 + 100n$ (linha cinza) também cresce quadraticamente, mas a presença do termo linear $100n$ faz com que seu crescimento inicial seja mais rápido que n^2, especialmente em valores menores de n. Para valores muito grandes de n, o termo n^2 domina, e o crescimento se aproxima mais de n^2.



À medida que n aumenta, a função n^2 cresce muito mais rapidamente do que a função 100n.
Isso ocorre porque a taxa de crescimento de uma função quadrática é proporcional ao quadrado do valor de n, enquanto a taxa de crescimento de uma função linear é proporcional ao valor de n em si.

Eventualmente, para valores suficientemente grandes de n, o termo quadrático n^2 supera o termo linear 100n por uma margem significativa, tornando-se o termo dominante na expressão n^2+100n

Contagem de Operações

Contar o número de operações para atribuir a cada posição `vet[i]` de um **arranjo unidimensional** o resultado de `i*2`

```
public static int[] multiplicavetor() {
    int vet[] = new int[10];
    for(int i = 0; i<vet.length; i++)
        vet[i] = i * 2;
    return vet;
}
```

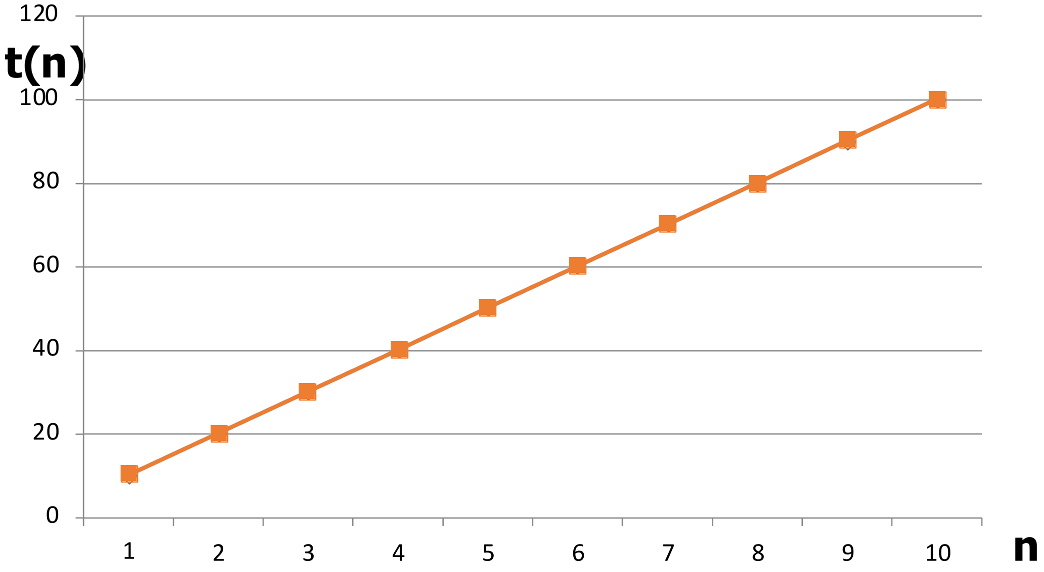
Operações de multiplicação, atribuição e acesso às posições do arranjo

Executado n vezes, onde n=10

Contagem de Operações

Contar o número de operações para atribuir para cada posição `vet[i]` de um **arranjo Unidimensional** o resultado de `i*2`

```
public static int[] multiplicavetor() {
    int vet[] = new int[10];
    for(int i = 0; i<vet.length; i++)
        vet[i] = i * 2;
    return vet;
}
```



Cresce de maneira linear!

Contar o número de operações para atribuir para cada posição `mat[i,j]` de um **arranjo Bidimensional** o resultado de `i*j`

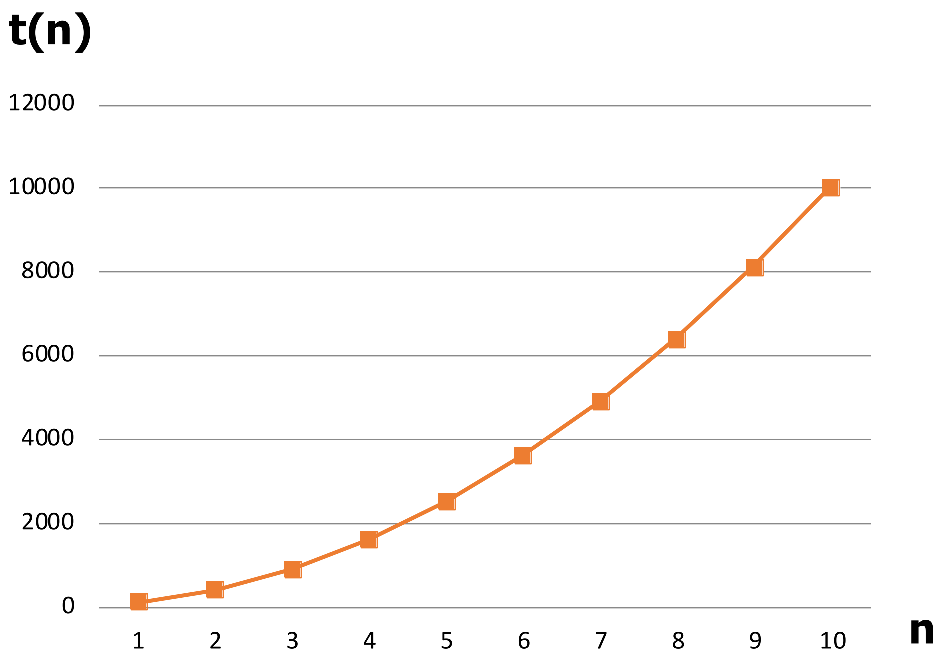
```
public static int[][] multiplicamatriz() {
    int mat[][] = new int[10][10];
    for (int i = 0; i < mat.length; i++) {
        for (int j = 0; j < mat.length; j++) {
            mat[i][j] = i * j;
        }
    }
    return mat;
}
```

Operação de multiplicação, atribuição e acesso as posições do arranjo

Executado n*n vezes, onde n=10

Contar o número de operações para atribuir para cada posição `mat[i,j]` de um **arranjo Bidimensional** o resultado de `i*j`

```java
public static int[][] multiplicamatriz() {
    int mat[][] = new int[10][10];
    for (int i = 0; i < mat.length; i++) {
        for (int j = 0; j < mat.length; j++) {
            mat[i][j] = i * j;
        }
    }
    return mat;
}
```



Cresce de maneira quadrática

Propriedades da notação O

Permite **ignorar os fatores constantes e** os **termos de menor ordem**, focando nos principais componentes da função que afetam seu crescimento

Deve-se descrever a função **O** em termos simples:

Se $f(n)$ é um polinômio de grau d, isto é,

$$f(n) = a_0 + a_1 n + \ldots + a_d n^d$$

e $a_d > 0$, então $f(n)$ é $O(n^d)$

Para medir a complexidade de tempo, em relação ao passo a passo para análise do polinômio (slide anterior), deve-se:

1. Ignorar as constantes
2. Focar nas repetições do algoritmos
3. Verificar a complexidade dos métodos de terceiros
4. Priorizar o termo de maior grau
5. Considerar sempre o pior caso

- Portanto, o termo de mais alto grau em um polinômio é o termo que determina a taxa de crescimento assintótico do polinômio

Por quê?

Exemplo 1:

$$5n^2 + 3n \log n + 2n + 5$$

Qual é o termo mais alto?

Qual é o termo mais alto?

$$5n^2 + 3n \log n + 2n + 5$$

$$\cancel{5}n^2 + \cancel{3}n \log n + \cancel{2}n + \cancel{5}$$

$$n^2 + n \log n + n$$

Independende das constantes, pois os valores relevantes são as variáveis que são n. O maior termo dará a complexidade.

Qual é o termo mais alto?

$$5n^2 + 3n \log n + 2n + 5$$

$$\cancel{5}n^2 + \cancel{3}n \log n + \cancel{2}n + \cancel{5}$$

$$\boxed{n^2} + n \log n + n$$

- Portanto, o termo de mais alto grau em um polinômio é o termo que determina a taxa de crescimento assintótico do polinômio

Veja aqui...

$O(n!)$	Fatorial
$O(a^n)$	Exponencial
$O(n^3)$	Cúbica
$O(n^2)$	Quadrática
$O(n \log n)$	Linearítmico
$O(n)$	Linear
$O(\log n)$	Logarítmica
$O(1)$	Constante

Pior

Melhor

Exemplo 2:

$$20n^3 + 10n \log n + 5$$

Qual é o termo mais alto?

Qual é o termo mais alto?

$$20n^3 + 10n \log n + 5$$

$$\cancel{20}n^3 + \cancel{10}n \log n + \cancel{5}$$

$$n^3 + n \log n$$

Qual é o termo mais alto?

$$20n^3 + 10n \log n + 5$$

$$\cancel{20}n^3 + \cancel{10}n \log n + \cancel{5}$$

$$\boxed{n^3} + n \log n$$

- Portanto, o termo de mais alto grau em um polinômio é o termo que determina a taxa de crescimento assintótico do polinômio

Veja aqui...

Notação	Classe	
$O(n!)$	Fatorial	Pior
$O(a^n)$	Exponencial	
$O(n^3)$	**Cúbica**	
$O(n^2)$	Quadrática	
$O(n \log n)$	Linearítmico	
$O(n)$	Linear	
$O(\log n)$	Logarítmica	
$O(1)$	Constante	Melhor

Exemplo 3:

$$3 \log n + 2n + 2$$

Qual é o termo mais alto?

Qual é o termo mais alto?

$$3 \log n + 2n + 2$$

$$\cancel{3} \log n + \cancel{2}n + \cancel{2}$$

$$\log n + n$$

Qual é o termo mais alto?

$$3 \log n + 2n + 2$$

$$\cancel{3} \log n + \cancel{2}n + \cancel{2}$$

$$\log n + \boxed{n}$$

n é maior que log n, porque … *Olhar a tabela de referência*

- Portanto, o **termo de mais alto grau em um polinômio é o termo que determina a taxa de crescimento assintótico do polinômio**

Veja aqui...

Pior

O(n!)	Fatorial
$O(a^n)$	Exponencial
$O(n^3)$	Cúbica
$O(n^2)$	Quadrática
O(n log n)	Linearítmico
O(n)	Linear
O(log n)	Logarítmica
O(1)	Constante

Melhor

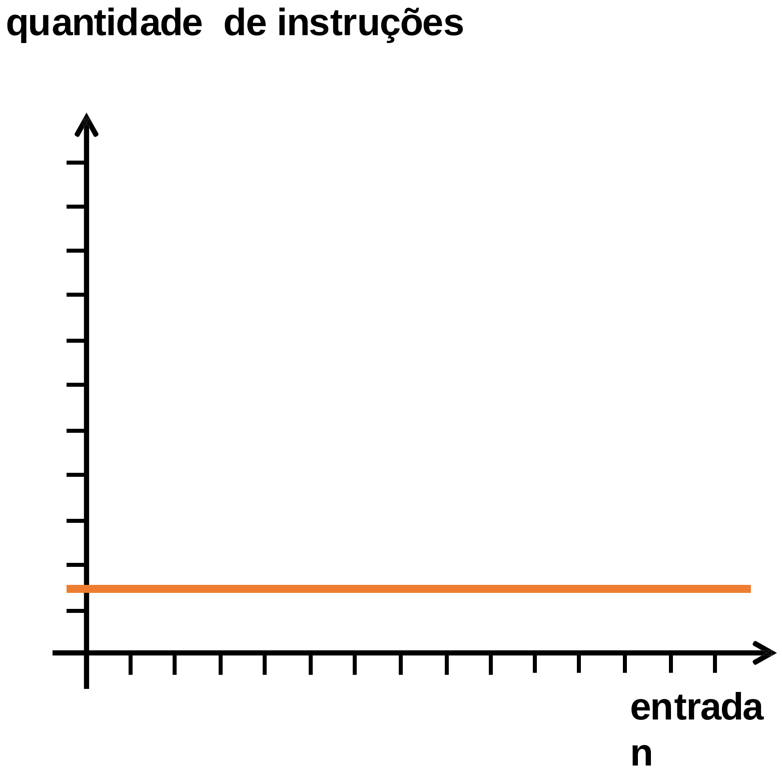
Então: Uma **classe de complexidade** é uma forma de agrupar algoritmos que apresentam complexidade similar. A **Notação Big O** adota as seguintes classes:

$O(1)$	**Constante**	$O(\log n)$	**Logarítmica**
$O(n)$	**Linear**	$O(n \log n)$	**Linearítmico**
$O(n^2)$	**Quadrática**	$O(a^n)$	**Exponencial**
$O(n^3)$	**Cúbica**	$O(n!)$	**Fatorial**

O(1) **Constante**

Função mais simples que representa um algoritmo de complexidade constante, ou seja, **não há crescimento do número de operações** depende do volume de dados de entrada (n). Assim, não importa o valor de n, sempre será igual ao valor da constante c.

Exemplos: acesso direto a um elemento de uma matriz ou uma função que recebe um arranjo de inteiros e retorna o valor do primeiro elemento multiplicado por 2.



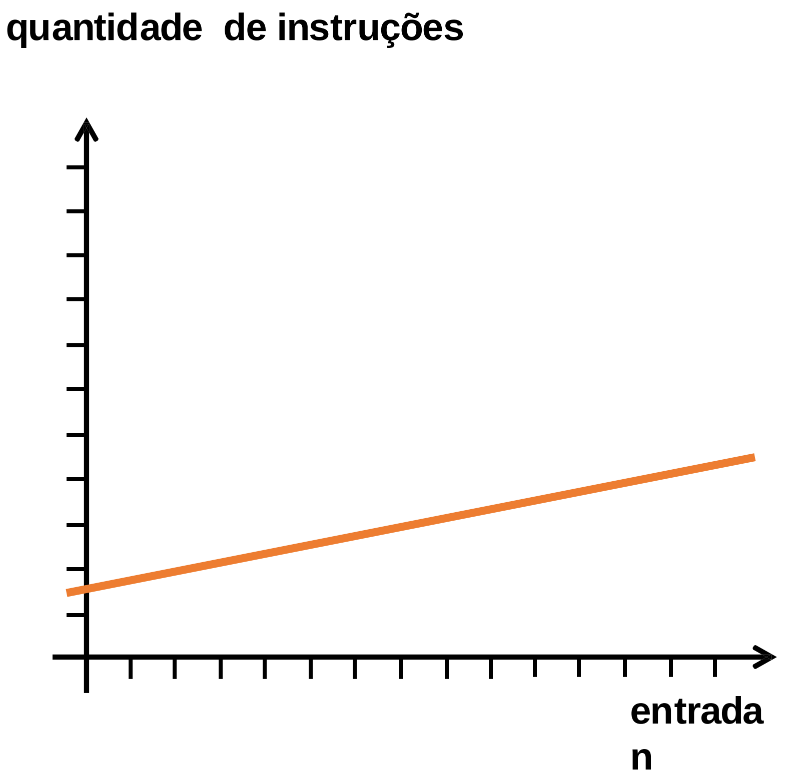
$$f(n) = c$$

O(n) Linear

Um algoritmo de complexidade linear, onde o crescimento no número de operações é diretamente proporcional ao crescimento do número de itens.

Se dobra o valor de n, dobra o consumo de recursos.

Exemplos: localizar um elemento em uma lista e a busca em uma matriz unidimensional não ordenada.

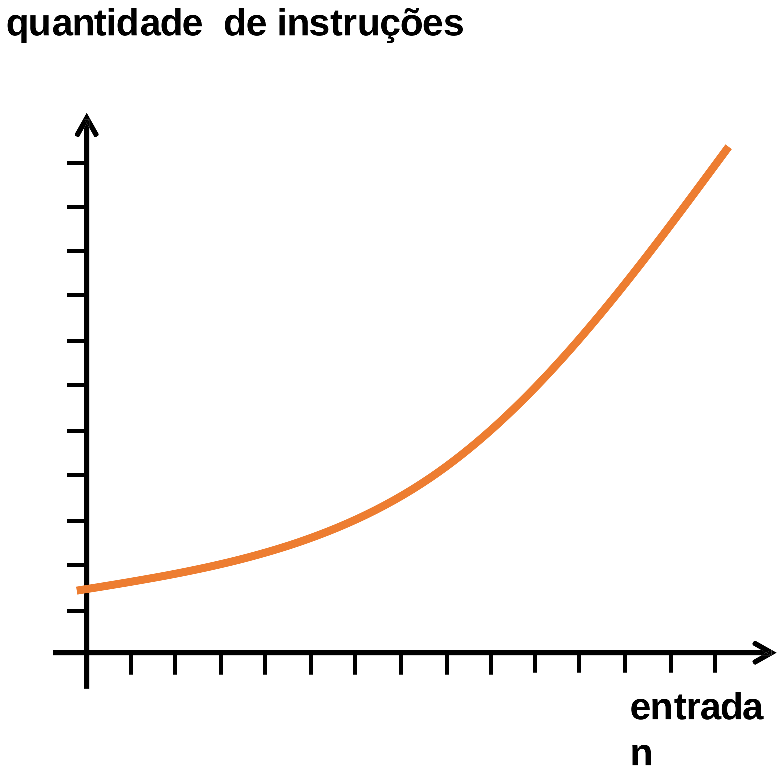


$$f(n) = n$$

$O(n^2)$ Quadrática

Um algoritmo de complexidade $O(n^2)$ é factível, mas tende a se tornar muito ruim quando a quantidade de dados é suficientemente grande. Função polinomial com expoente 2 não cresce de forma abrupta, mas dificulta o uso em problemas grandes.

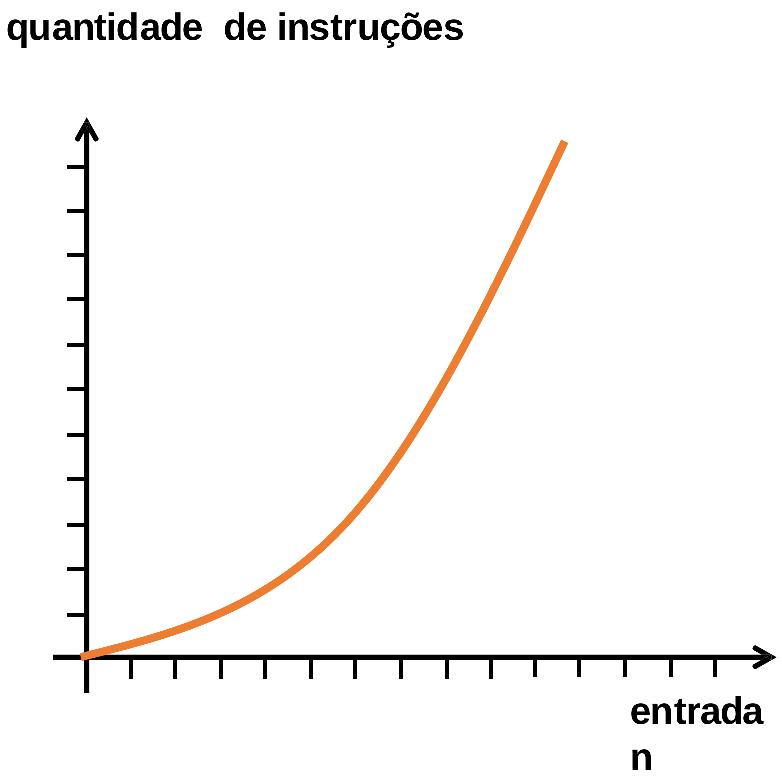
Exemplos: Ordenação com o algoritmo bubblesort e algoritmos que têm dois laços encadeados, como, por exemplo, o processamento de itens em uma matriz bidimensional.



$$f(n) = n^2$$

$O(n^3)$ **Cúbica**

Aparece com menos frequência e são geralmente para resolver problemas relativamente pequenos.



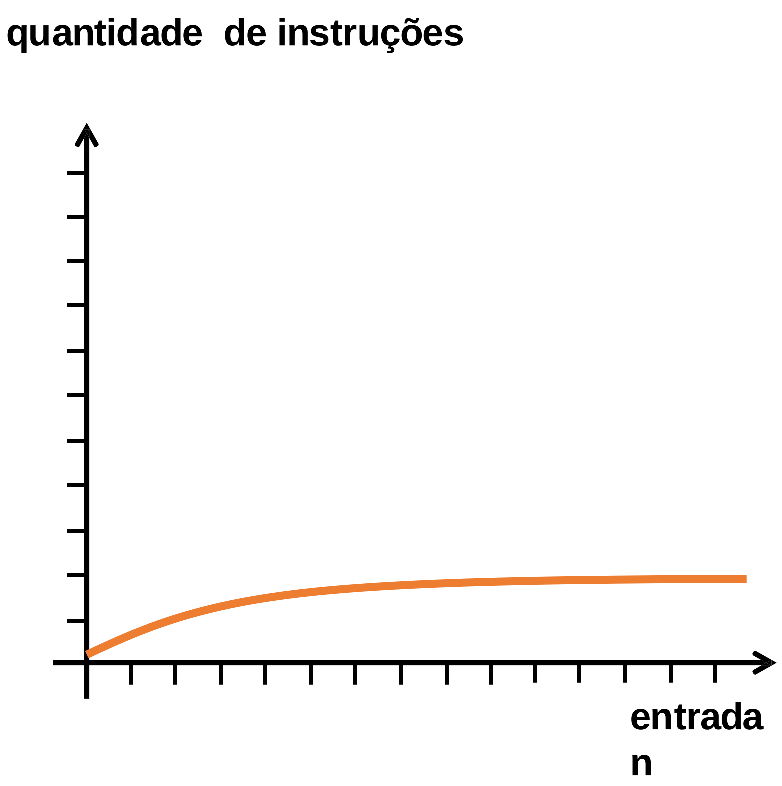
$$f(n) = n^3$$

Exemplos: multiplicar matrizes.

O(log n) Logarítmica

O número de operações realizadas para solução do problema **não cresce da mesma forma que n**. Se dobra o valor de n, o incremento do consumo é **bem menor.** Ou seja, é aquele cujo crescimento do número de operações é menor do que o do número de itens.

Exemplo: conversão de número decimal para binário, pesquisa binária e busca em árvores binárias ordenadas.

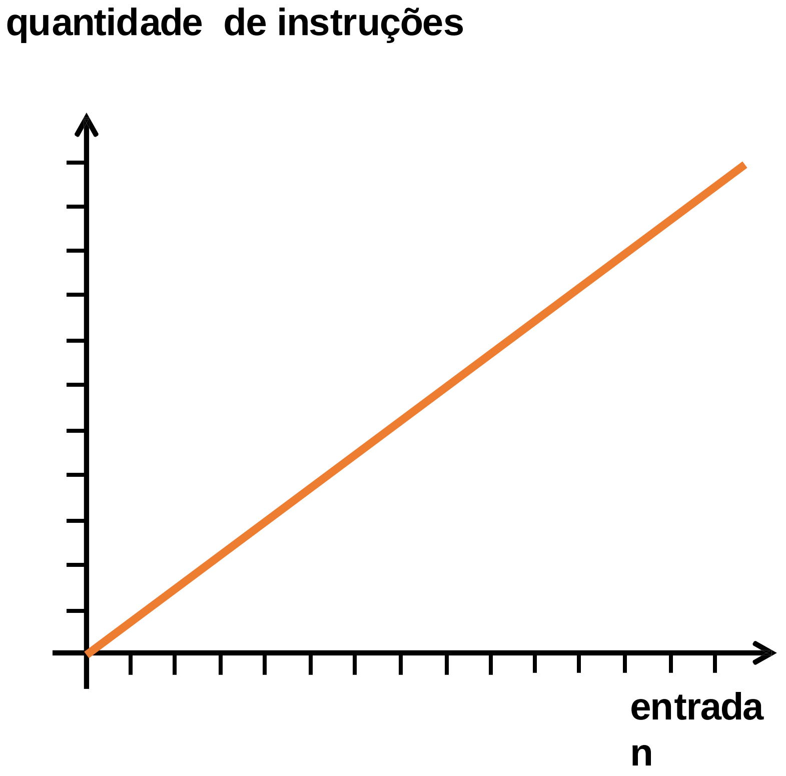


f(n) = log n

O(n log n) Linearítmico

Também chamado de sub-quadrático ou super-linear, atribui para uma entrada n o valor de **n multiplicado pelo logaritmo de base 2 de n**. Cresce mais rápido que a função linear, mas é melhor do que o quadrático. Geralmente é até onde se consegue otimizar algoritmos que são quadráticos.

Exemplo: algoritmos de ordenação como *mergesort* e *heapsort*.

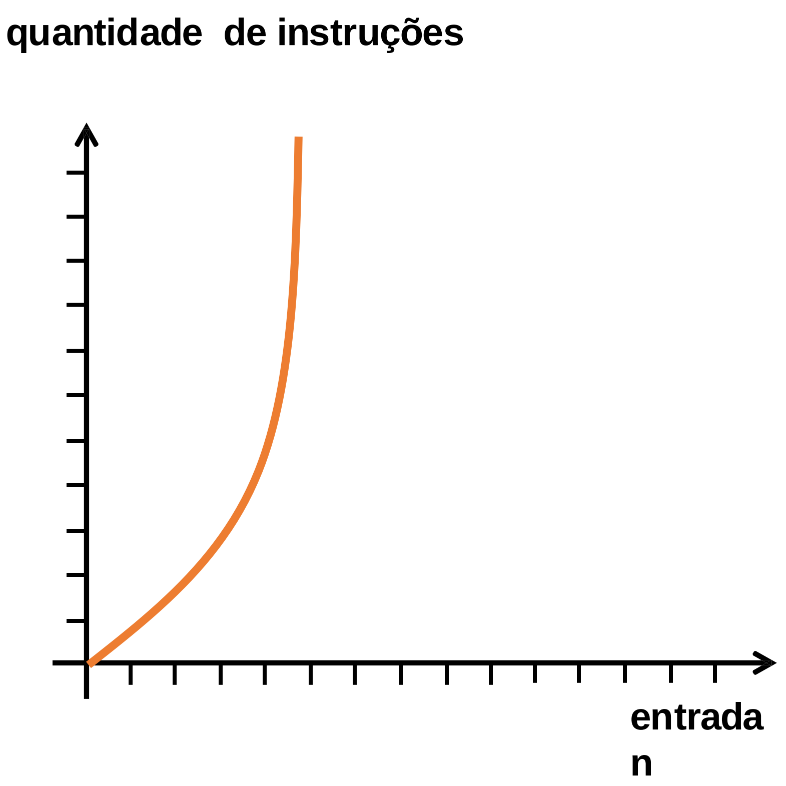


$$f(n) = n \log n$$

$O(a^n)$ Exponencial

Típico de algoritmos que fazem muitas combinações, do tipo “força bruta” para resolver um problema. São considerados ruins, pois o número de instruções cresce muito rapidamente, ou seja, de maneira exponencial.

Exemplos: quebrar senhas com força bruta e listar todos os subconjuntos de um conjunto S e algoritmos que fazem busca em árvores binárias não ordenadas, por exemplo.

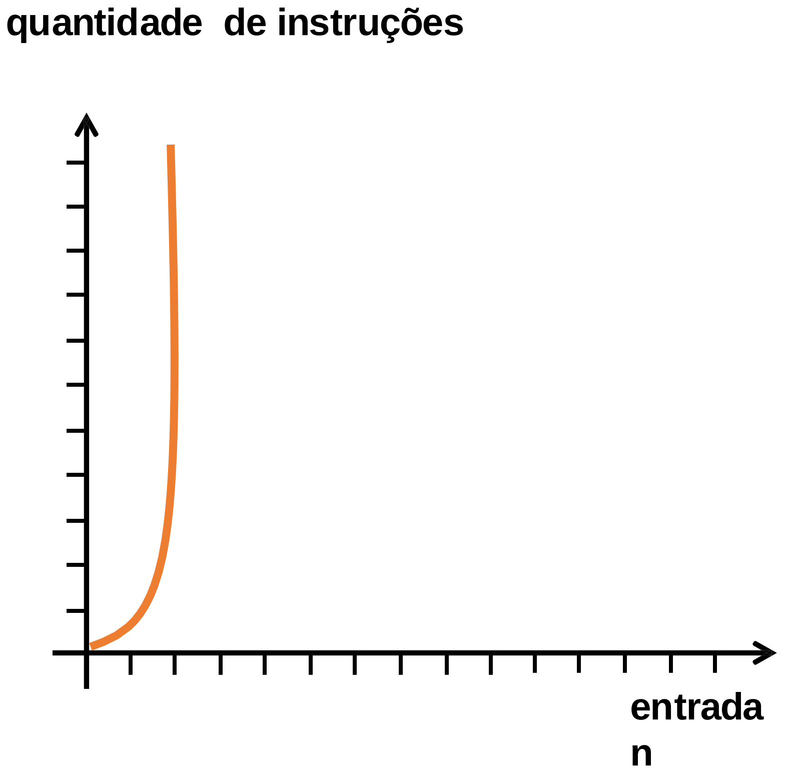


$$f(n) = a^n$$

O(n!) Fatorial

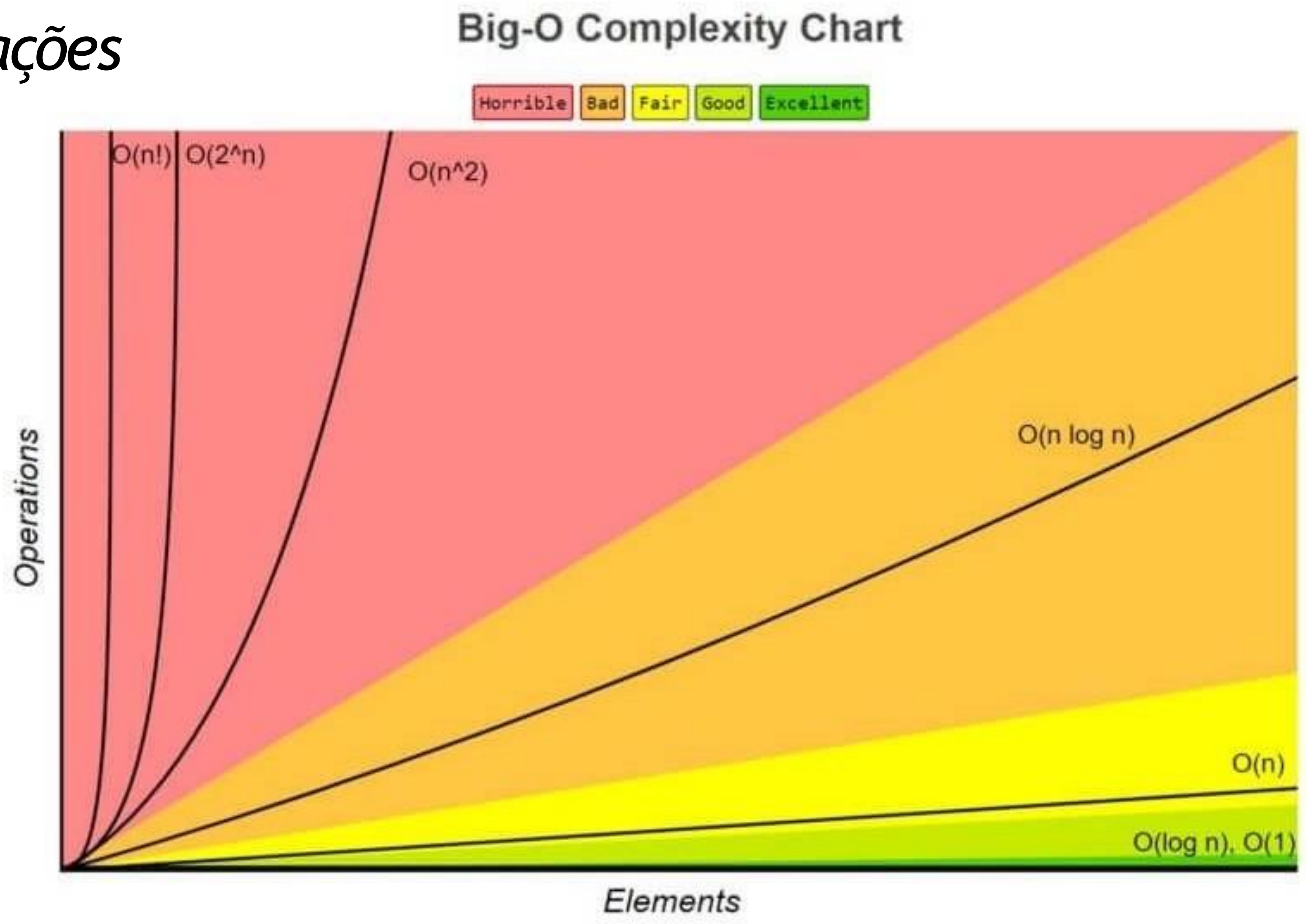
O número de instruções executadas cresce muito rapidamente para um pequeno crescimento do número de itens processados. Dentre os ilustrados é o pior comportamento para um algoritmo, pois rapidamente o processamento se torna inviável.

Exemplos: implementação do Problema do Caixeiro Viajante ou de um algoritmo que gera todas as possíveis permutações de uma lista.



$$f(n) = n!$$

Comparações



Contar o número de operações das funções listadas a seguir.

```
public static int funcao01(int n) {
    int r = 0;
    for(int i=0; i<n; i++) {    n
        r = r + 1;
    }
    return r;
}
```

para n=10

i → 0 1 2 3 4 5 6 7 8 9

para n=20

i → 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

O(n)

```java
public static int funcao02(int n) {
    int cont = 0;
    for(int i=0; i<n; i++) {
        for (int j = 0; j < n; j++)
            cont++;
    }
    return cont;
}
```

n

n

Ex: para n=10

i → *0 1 2 3 4 5 6 7 8 9*
↓
j *0 0 0 0 0 0 0 0 0 0 0 0*
1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9 9

$O(n) * O(n) = O(n^2)$

```java
public static int funcao03(int n) {
    int r = 0;
    for(int i=1; i<n; i++) {
        for (int j = i + 1; j<n; j++) {
            r = r + 2;
        }
    }
    return r;
}
```

n

n

Ex: para n=10

i 0 1 2 3 4 5 6 7 8 9
j 1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 9
3 4 5 6 7 8 9
4 5 6 7 8 9
5 6 7 8 9
6 7 8 9
7 8 9
8 9
9

$O(n) * O(n) = O(n^2)$

```java
public static int funcao05(int n) {
    int r = 0;
    for(int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            for (int k=0; k<n; k++) {
                r = r + 1;
            }
        }
    }
    return r;
}
```

n

n

$O(n) * O(n) * O(n) = O(n^3)$