



# Algoritmo e Estrutura de Dados II

## COM-112

Aula 2

Vanessa Souza

O que se espera de um bom programador ?





# Fundamentação

---

Que ele saiba escolher a **melhor estrutura e algoritmo** para o seu problema

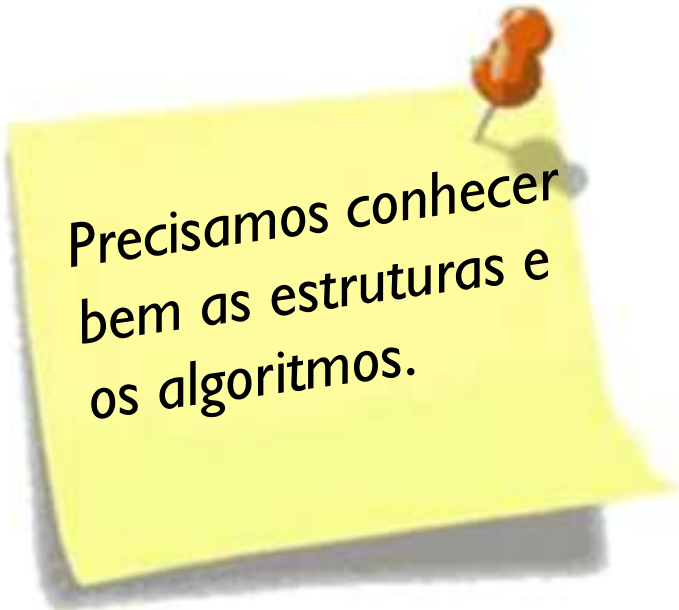




# Fundamentação

---

Que ele saiba escolher a melhor estrutura e algoritmo para o seu problema



Precisamos conhecer bem as estruturas e os algoritmos.





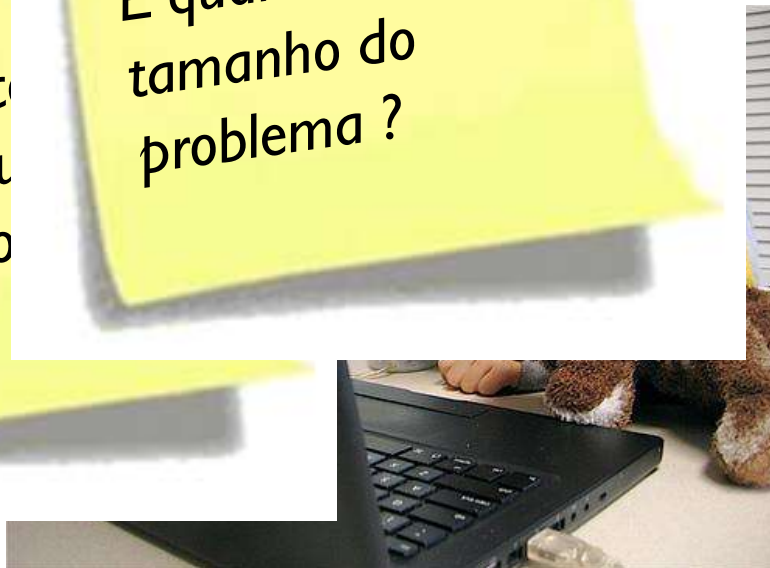
# Fundamentação

---

Que ele saiba escolher a melhor estrutura e algoritmo para o seu problema

Precisamos escolher  
bem as estruturas  
e os algoritmos

E quanto ao  
tamanho do  
problema?





# Fundamentação

---

**Reformulação:** Que ele saiba escolher a melhor estrutura e o algoritmo para uma dada **instância de um problema**.

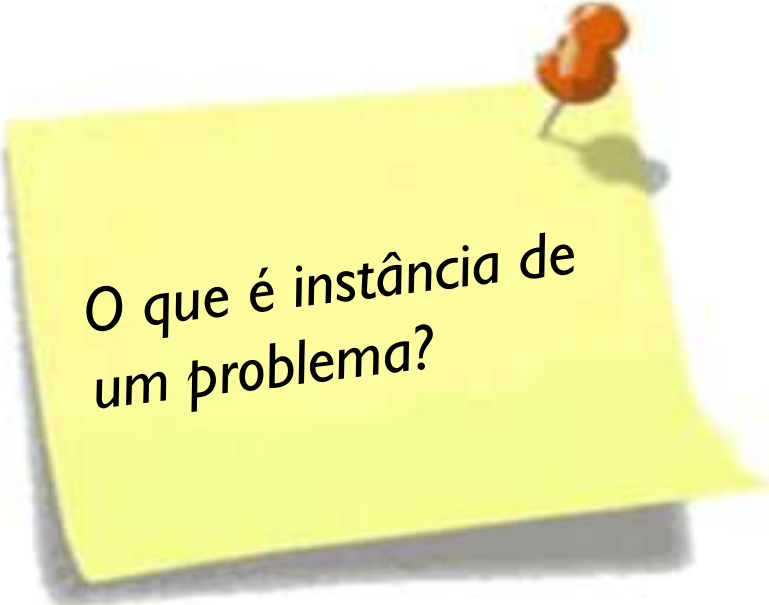




# Fundamentação

---

**Reformulação:** Que ele saiba escolher a melhor estrutura e o algoritmo para uma dada **instância de um problema**.



O que é instância de um problema?







# Instância

---

- ▶ Instância de um problema é um grupo particular de inserção de dados em que um programa é aplicado.
- ▶ Existem até algumas instâncias comuns que diversos programadores tentam resolver com menor tempo.
- ▶ Um problema é um conjunto de instâncias
  - ▶ Alguns algoritmos podem resolver apenas algumas instâncias.



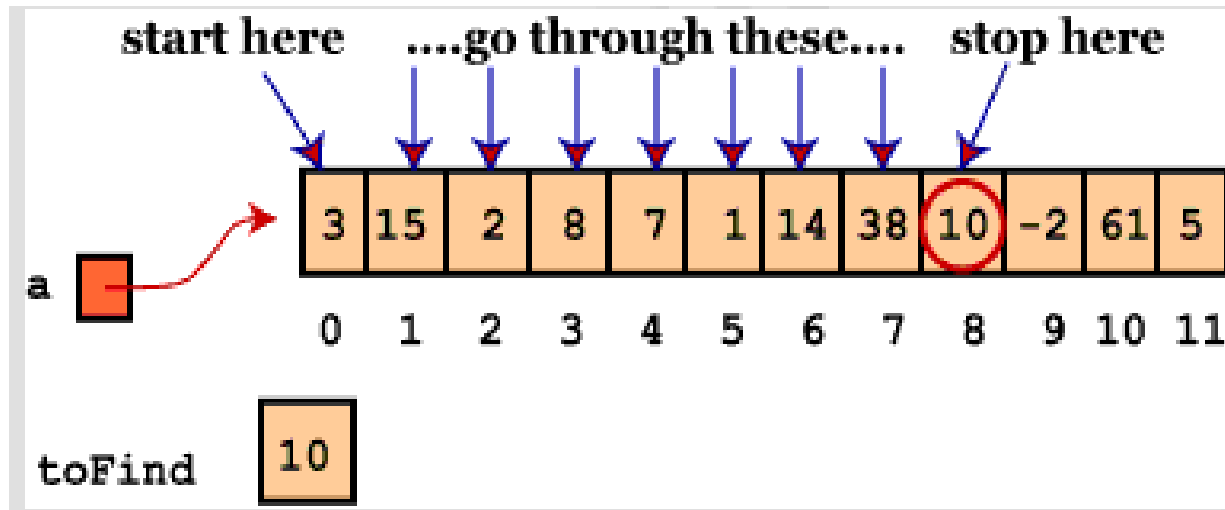




## Instância

---

- Considere uma pesquisa sequencial ou linear





# Características dos algoritmos

---

## ▶ **Correção**

- ▶ trabalhar corretamente quaisquer que sejam os dados de entrada dentro de um certo domínio

## ▶ **Eficiência**

- ▶ significa que as estruturas e os algoritmos devem ser rápidos e nunca usar recursos do computador superiores ao necessário
  - ▶ *Para algumas instâncias de problema, o tempo pode ser realmente grande...*



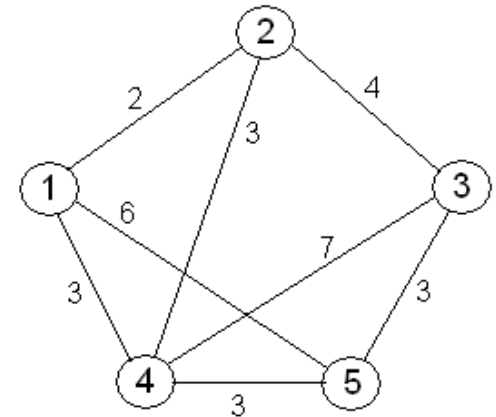


# Exemplos de problemas complexos

O **problema da mochila**, o objetivo é que se preencha a mochila com o maior valor possível, não ultrapassando o peso máximo.



O **problema do caixeiro-viajante** consiste na procura de um circuito que possua a menor distância, começando numa qualquer cidade, entre várias, visitando cada cidade precisamente uma vez e regressando à cidade inicial





Pensando um pouco ....

---

Quais os dois recursos de hardware mais importantes para a execução de um algoritmo?





# Pensando um pouco ....

---

Quais os dois recursos de hardware mais importantes para a execução de um algoritmo?

- Tempo de processamento;
- Quantidade de memória consumida;





# Pensando um pouco ....

---

Quais os dois recursos de hardware mais importantes para a execução de um algoritmo?

Tempo de processamento;

Quantidade de memória consumida;

Devemos, então, observar quanto de cada recurso nossos algoritmos consomem;

Temos que medir quanto de cada recurso nossos programas utilizam!

---





# COMPLEXIDADE





# Algoritmo

---

- Um algoritmo é um procedimento, consistindo de um conjunto de regras não ambíguas, as quais especificam, para cada entrada, uma seqüência finita de **operações**, terminando com uma saída correspondente.
- Um algoritmo resolve um problema quando, para qualquer entrada, produz uma resposta **correta**, se forem concedidos **tempo** e **memória** suficientes para sua execução.





# Análise de Algoritmos

---

- ▶ Análise de um algoritmo em particular
  - ▶ Qual é o custo de usar um dado algoritmo para resolver um problema específico?
    - ▶ Análise do número de vezes que cada parte do algoritmo deve ser executada, seguida pelo estudo da quantidade de memória necessária.
- ▶ Análise de uma classe de algoritmos
  - ▶ Qual é o algoritmo de menor custo possível para resolver um problema particular?
    - ▶ Todos os algoritmos são investigados e o resultado é comparado para identificar o melhor possível.





# Análise de Algoritmos

---

- ▶ Critérios de avaliação de um algoritmo
  - ▶ Quantidade de trabalho requerido;
  - ▶ Quantidade de memória utilizada;
  - ▶ Simplicidade;
  - ▶ Exatidão de resposta, etc.
- ▶ Como medir?
  - ▶ Execução do programa em um computador real – medir o tempo diretamente.
  - ▶ Desvantagem: os resultados são dependentes do computador.





# Complexidade

---

- ▶ Frequentemente, desejamos saber quão eficiente é um determinado algoritmo usado para resolver um problema, ou seja, quanto tempo (ou espaço de memória) será necessário para a execução do algoritmo.
- ▶ Isto é chamado de complexidade de tempo (ou de espaço) do algoritmo.





# Complexidade

---

- ▶ Medida da dificuldade (custo) para resolver um problema computacional.
- ▶ Custo: esforço computacional; quantidade de trabalho, em termos de tempo de execução ou da quantidade de memória requerida.
- **Objetivo do cálculo de complexidade:** avaliar o desempenho de um algoritmo e se possível melhorá-lo.





# Complexidade

---

- ▶ Por que analisar a complexidade dos algoritmos?
  - ▶ A preocupação com a complexidade de algoritmos é fundamental para projetar algoritmos eficientes.
  - ▶ Podemos desenvolver um algoritmo e depois analisar sua complexidade para verificar sua eficiência.
  - ▶ Melhor ainda é ter a preocupação de projetar algoritmos eficientes desde a sua concepção.
- ▶ Quanto maior a complexidade, menor a eficiência do algoritmo.





# Complexidade

---

- ▶ A Complexidade de um algoritmo consiste na quantidade de "trabalho" necessário para sua execução, expressa em função das operações fundamentais, as quais variam de acordo com o algoritmo, e em função do volume de dados.







# Complexidade

---

- ▶ Operações Fundamentais
  - ▶ Aritméticas
    - ▶ Soma, subtração, multiplicação, divisão, resto, piso, teto,...
  - ▶ Movimentação de Dados
    - ▶ Carregar, armazenar, copiar, atribuições
  - ▶ Controle
    - ▶ Desvio condicional e incondicional, chamada e retorno de sub-rotinas
- ▶ Cada uma dessas instruções demora um período constante.





# Complexidade

---

## ► Tipos de Complexidade

- Espacial - Este tipo de complexidade representa o espaço de memória usado para executar o algoritmo, por exemplo.
- Temporal - Este tipo de complexidade é o mais usado podendo dividir-se em dois grupos:
  - Tempo ( real ) necessário à execução do algoritmo.
  - Número de instruções necessárias à execução.





# Complexidade

---

## ► Exemplo

### ► O que faz esse código?

```
int buscaElemento(int vet[], int tam, int elemento)
{
    int i = 0;
    int quant = 0;

    for (i=0; i<tam; i++)
    {
        if (vet[i] == elemento)
            quant++;
    }
    return quant;
}
```





# Complexidade

## ► Exemplo

### ► O que faz esse código?

```
int buscaElemento(int vet[], int tam, int elemento)
{
    int i = 0;
    int quant = 0;
    for (i=0; i<tam; i++)
    {
        if (vet[i] == elemento)
            quant++;
    }
    return quant;
}
```

Custo	Número de vezes
c1	1
c2	1
c3	n
c4	n
c5	n
c6	1





# Complexidade

## ► Análise de Complexidade

```
int buscaElemento(int vet[], int tam, int elemento)
{
    int i = 0;
    int quant = 0;
    for (i=0; i<tam; i++)
    {
        if (vet[i] == elemento)
            quant++;
    }
    return quant;
}
```

Custo	Número de vezes
c1	1
c2	1
c3	n
c4	n
c5	n??
c6	1





# Complexidade

---

## ► Melhor Caso:

- Menor tempo de execução sobre todas as entradas de tamanho  $n$
- Quando o algoritmo tem o melhor desempenho, ou seja, faz o menor número de operações significativas

## ► Exemplo:

- Qual o melhor caso de um algoritmo que busca um elemento num vetor (sem repetição)?





# Complexidade

---

## ► Caso Médio:

- Média dos tempos de execução de todas as entradas de tamanho  $n$ .
- Na análise do caso esperado, supõe-se uma **distribuição de probabilidades** sobre o conjunto de entradas de tamanho  $n$  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.

## ► Exemplo:

- Qual o caso médio de um algoritmo que busca um elemento num vetor (sem repetição)?







# Complexidade

---

## ► Pior Caso:

- Maior tempo de execução sobre todas as entradas de tamanho  $n$ .
- Demonstra o pior desempenho que se pode esperar sobre todas as entradas com tamanho  $n$ .

## ► Exemplo:

- Qual o pior caso de um algoritmo que busca um elemento num vetor (sem repetição)?





## Complexidade

---

- ▶ Em geral, tempos mínimos são de pouca utilidade e tempos médios são difíceis de calcular (dependem de se conhecer a probabilidade de ocorrência das diferentes entradas para o algoritmo).
- ▶ Considera-se então a medida do pior caso de desempenho, que ocorre com a entrada mais desfavorável possível.





# Função de complexidade

---

- ▶ **Função de complexidade  $f(n)$ :** medida do custo para executar um algoritmo.
- ▶ A medida de custo depende principalmente do tamanho da entrada de dados → função do tamanho da entrada.
- ▶  **$f(n)$**  é a medida do tempo necessário para executar um algoritmo para um problema de tamanho  $n$ .
  - ▶ Deve ser especificado o conjunto de operações e seus custos de execuções.
  - ▶ Ignora-se o custo de algumas das operações e considera-se apenas as operações mais significativas.





# Complexidade

## ► Análise de Complexidade

```
int buscaElemento(int vet[], int tam, int elemento)
{
    int i = 0;
    int quant = 0;
    for (i=0; i<tam; i++)
    {
        if (vet[i] == elemento)
            quant++;
    }
    return quant;
}
```

Custo	Número de vezes
c1	1
c2	1
c3	n
c4	n
c5	n
c6	1

$$F(n) = c1*1 + c2*1 + c3*n + c4*n + c5*n + c6*1$$



# Complexidade

## ► Análise de Complexidade

```
int buscaElemento(int vet[], int tam, int elemento)
{
    int i = 0;
    int quant = 0;
    for (i=0; i<tam; i++)
    {
        if (vet[i] == elemento)
            quant++;
    }
    return quant;
}
```

Custo	Número de vezes
c1	1
c2	1
c3	n
c4	n
c5	n
c6	1

$$F(n) = (c3 + c4 + c5)n + c1 + c2 + c6$$



# Complexidade

## ► Análise de Complexidade

```
int buscaElemento(int vet[], int tam, int elemento)
{
    int i = 0;
    int quant = 0;
    for (i=0; i<tam; i++)
    {
        if (vet[i] == elemento)
            quant++;
    }
    return quant;
}
```

Custo	Número de vezes
c1	1
c2	1
c3	n
c4	n
c5	n
c6	1

$$F(n) = \underbrace{(c3 + c4 + c5)}_a n + \underbrace{c1 + c2 + c6}_b$$



# Complexidade

## ► Análise de Complexidade

```
int buscaElemento(int vet[], int tam, int elemento)
{
    int i = 0;
    int quant = 0;
    for (i=0; i<tam; i++)
    {
        if (vet[i] == elemento)
            quant++;
    }
    return quant;
}
```

Custo	Número de vezes
c1	1
c2	1
c3	n
c4	n
c5	n
c6	1

$$F(n) = an + b$$





# Complexidade Assintótica

---

- ▶ O parâmetro  $n$  fornece uma medida da dificuldade para se resolver o problema.
- ▶ Para valores suficientemente pequenos de  $n$ , qualquer algoritmo custa pouco para ser executado, mesmo os ineficientes.
- ▶ A **escolha do algoritmo** não é um problema crítico para problemas de tamanho pequeno.
- ▶ Logo, a análise de algoritmos é realizada para valores grandes de  $n \rightarrow$  **Complexidade Assintótica**





# Complexidade Assintótica

---

- ▶ Comportamento das funções de custo para  $n$  grandes.
- ▶ Limite do comportamento do custo quando  $n$  cresce.
- ▶ Notação :
  - ▶  $\Theta$  (Theta)
  - ▶  $O$  (big  $O$ )





# Complexidade Assintótica

---

- ▶ Quando  $n$  é muito grande, as constantes perdem significância, assim como as demais 'ordens' do polinômio.
- ▶ Exemplo:
  - ▶ Dada a função de complexidade  $F(n) = an^3 + bn^2 + cn + d$



# Complexidade Assintótica

---

- ▶ Quando  $n$  é muito grande, as constantes perdem significância, assim como as demais 'ordens' do polinômio.
- ▶ Exemplo:
  - ▶ Dada a função de complexidade  $F(n) = an^3 + bn^2 + cn + d$
  - ▶ Dizemos que sua complexidade no pior caso é dada por  $\Theta(n^3)$
  - ▶ Lê-se “*theta de n ao cubo*”
  - ▶ Ou, mais comumente,
  - ▶  $O(n^3)$  - Lê-se “*O de n ao cubo*”





# Complexidade

---

## ► Exemplo

- Se a operação fundamental for a soma, quantas vezes ela será executada?

```
int somaVetor(int tam, int vet[])  
{  
    int i;  
    int soma = 0;  
    for (i=0; i<tam; i++)  
    {  
        soma = soma + vet[i];  
    }  
    return(soma);  
}
```





# Complexidade

---

## ▶ Exemplo

- ▶ Se a operação fundamental for a soma, quantas vezes ela será executada?

$$\Theta(n)$$

- ▶ O tempo de execução (em instruções) variará conforme variar  $n$ , numa proporção linear.

```
int somaVetor(int tam, int vet[])  
{  
    int i;  
    int soma = 0;  
    for (i=0; i<tam; i++)  
    {  
        → soma = soma + vet[i];  
    }  
    return(soma);  
}
```





# Complexidade

---

## ► Exemplo 2

- Se a operação fundamental for a comparação, quantas vezes ela será executada?

```
void procuraElemento(int vet[], int tam, int elemento)
{
    int i = 0;
    int achou = 0;
    while ((i < tam) && (achou == 0))
    {
        → if (vet[i] == elemento)
        {
            printf("Elemento encontrado na posição %d", i);
            achou = 1;
        }
        i++;
    }
    return;
}
```





# Complexidade

---

## ► Exemplo 2

- Se a operação fundamental for a comparação, quantas vezes ela será executada?
- No melhor caso?

□ 1 vez ->  $\Theta(1)$

Complexidade constante

```
void procuraElemento(int vet[], int tam, int elemento)
{
    int i = 0;
    int achou = 0;
    while ((i < tam) && (achou == 0))
    {
        if (vet[i] == elemento)
        {
            printf("Elemento encontrado na posição %d", i);
            achou = 1;
        }
        i++;
    }
    return;
}
```





# Complexidade

---

## ► Exemplo 2

- Se a operação fundamental for a comparação, quantas vezes ela será executada?
- No pior caso?

□ n vezes ->  $\Theta(n)$  Complexidade linear

```
void procuraElemento(int vet[], int tam, int elemento)
{
    int i = 0;
    int achou = 0;
    while ((i < tam) && (achou == 0))
    {
        if (vet[i] == elemento)
        {
            printf("Elemento encontrado na posição %d", i);
            achou = 1;
        }
        i++;
    }
    return;
}
```



# Complexidade

---

## ▶ Exemplo 2

- ▶ Se a operação fundamental for a comparação, quantas vezes ela será executada?
- ▶ No caso médio?

□  $(1 + \text{tam})/2$  vezes  $\rightarrow \Theta(n)$

Complexidade linear

```
void procuraElemento(int vet[], int tam, int elemento)
{
    int i = 0;
    int achou = 0;
    while ((i < tam) && (achou == 0))
    {
        if (vet[i] == elemento)
        {
            printf("Elemento encontrado na posição %d", i);
            achou = 1;
        }
        i++;
    }
    return;
}
```



## Exemplos

---

- ▶  $g(n) = (n + 1)^2 \rightarrow g(n) = O(n^2)$
- ▶  $f(n) = 2n^2 + 5n + 1 \rightarrow f(n) = O(n^2)$
- ▶  $f(n) = n^2 - 1 \rightarrow f(n) = O(n^2)$
- ▶  $f(n) = n^3 - n^2 + 2 \rightarrow f(n) = O(n^3)$
- ▶  $g(n) = 3n^3 + 2n^2 + n \rightarrow g(n) = O(n^3)$
- ▶  $f(n) = 403 \rightarrow f(n) = O(1)$
- ▶  $f(n) = 5 + 2\log n + 3\log^2 n \rightarrow f(n) = O(\log^2 n)$
- ▶  $f(n) = 5 \cdot 2^n + 5n10 \rightarrow f(n) = O(2^n)$

▶ Obs.: Cada polinômio é  $O$  de  $n$  elevado à maior potência.



# Classes de Comportamento Assintótico

---

## ▶ $f(n) = O(1)$ (complexidade constante)

- ▶ O uso do algoritmo independe do tamanho de  $n$ .
- ▶ Neste caso as instruções do algoritmo são executadas um número fixo de vezes.

## ▶ $f(n) = O(\log n)$ (complexidade logarítmica)

- ▶ Ocorre tipicamente em algoritmos que resolvem um problema transformando-o em problemas menores.





# Classes de Comportamento Assintótico

---

## ▶ **$f(n) = O(n)$ (complexidade linear)**

- ▶ Em geral um pequeno trabalho é realizado sobre cada elemento de entrada.
- ▶ É a melhor situação possível para um algoritmo que tem que processar  $n$  elementos de entrada ou produzir  $n$  elementos de saída.
- ▶ Cada vez que  $n$  dobra de tamanho o tempo de execução dobra.

## ▶ **$f(n) = O(n \log n)$**

- ▶ Ocorre tipicamente em algoritmos que resolvem um problema quebrando-o em problemas menores, resolvendo cada um deles independentemente e depois unindo as soluções.





# Classes de Comportamento Assintótico

---

## ▶ **$f(n) = O(n^2)$ (complexidade quadrática)**

- ▶ Algoritmos desta ordem de complexidade ocorrem quando os itens de dados são processados aos pares, muitas vezes em um laço de repetição dentro de outro.
- ▶ Algoritmos deste tipo são úteis para resolver problemas de tamanhos relativamente pequenos.

## ▶ **$f(n) = O(n^3)$ (complexidade cúbica)**

- ▶ Algoritmos desta ordem de complexidade são úteis apenas para resolver pequenos problemas.





# Classes de Comportamento Assintótico

---

- ▶  **$f(n) = O(2^n)$  (complexidade exponencial)**

- ▶ Algoritmos desta ordem de complexidade geralmente não são úteis sob o ponto de vista prático.
- ▶ Eles ocorrem na solução de problemas quando se usa **força bruta** para resolvê-los.

- ▶  **$f(n) = O(n!)$  (complexidade exponencial)**

- ▶ Considerados algoritmos exponenciais, apesar de terem comportamento muito pior do que  $O(2^n)$ .
- ▶ Também usam força bruta para resolver o problema.





# Complexidade – Outras Funções

$$F(n) = a$$

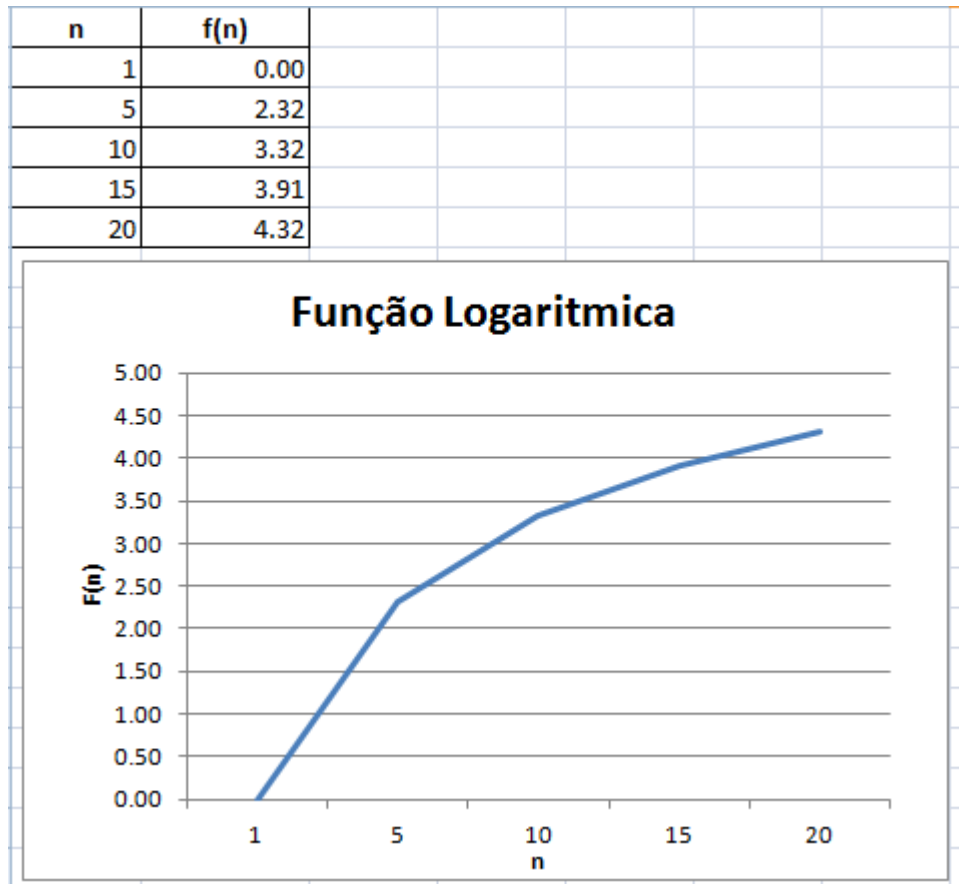
- ▶ Constante

$$F(n) = \log_2 n$$

- ▶ Logarítmica

$$F(n) = 2^n$$

- ▶ Exponencial







# Complexidade – Outras Funções

$$F(n) = a$$

- ▶ Constante

$$F(n) = \log_2 n$$

- ▶ Logarítmica

$$F(n) = 2^n$$

- ▶ Exponencial

n	f(n)
1	2
5	32
10	1024
15	32768
20	1048576





# Complexidade

---

- ▶ Partes do programa podem ter complexidades diferentes.
- ▶ A regra da soma  $O(f(n)) + O(g(n))$  pode ser usada para calcular o tempo de execução de uma sequência de trechos de programas.
- ▶ Suponha três trechos de programas, cujos tempos de execução são:  $O(n)$ ,  $O(n^2)$  e  $O(n \log n)$ .
  - ▶ Tempo dos dois primeiros trechos:  $O(\max(n; n^2)) = O(n^2)$ .
  - ▶ Tempo de execução dos três trechos:  
 $O(\max(n^2; n \log n)) = O(n^2)$ .





## Complexidade

---

- ▶ A avaliação analítica de um algoritmo pode ser feita com vistas a se obter uma estimativa do esforço de computação, não em termos de unidade de tempo propriamente, mais em termos de uma taxa de crescimento do tempo de execução em função do “tamanho do problema”, i.e., do tamanho da entrada.





# Complexidade

---

## ▶ Exercícios

### ▶ Estrutura de Dados : PILHA

- ▶ Qual a complexidade para inserir na pilha?
- ▶ Qual a complexidade para remover da pilha?





# Complexidade

---

## ▶ Exercícios

### ▶ Estrutura de Dados : FILA

- ▶ Qual a complexidade para inserir na fila?
- ▶ Qual a complexidade para remover da fila?





# Complexidade

---

## ▶ Exercícios

### ▶ Estrutura de Dados : LISTA

- ▶ Qual a complexidade para inserir na lista se ela for ordenada?
- ▶ Qual a complexidade para remover da lista?
- ▶ Qual a complexidade para pesquisar um elemento na lista?





# Complexidade

---

## ► Exercício

- Calcule a complexidade do seguinte trecho de código
  - Qual a função de complexidade?
  - Qual a complexidade assintótica?

```
algoritmo l (v: vetor; n: integer);  
início  
    para i ← 1 até n faça  
        início  
            soma ← v[1];  
            para j ← 2 até i faça  
                início  
                    soma ← soma + v[j]  
                fim  
            fim  
        fim  
    fim
```





## Comportamento assintótico

---

- ▶ Frequentemente, interessa-nos identificar um algoritmo **ótimo**.
  - ▶ Trata-se de um algoritmo, tal que o seu desempenho é sempre o melhor entre vários algoritmos que resolvem o mesmo problema.
- ▶ Essas ideias envolvem comparações entre funções **f** e **g**. Quando **f** é “melhor” do que **g**?
- ▶ Há vários critérios; os mais usados baseiam-se em comportamento assintótico.
  - ▶ Um algoritmo assintoticamente mais eficiente é melhor para todas as entradas, exceto para entradas relativamente pequenas.



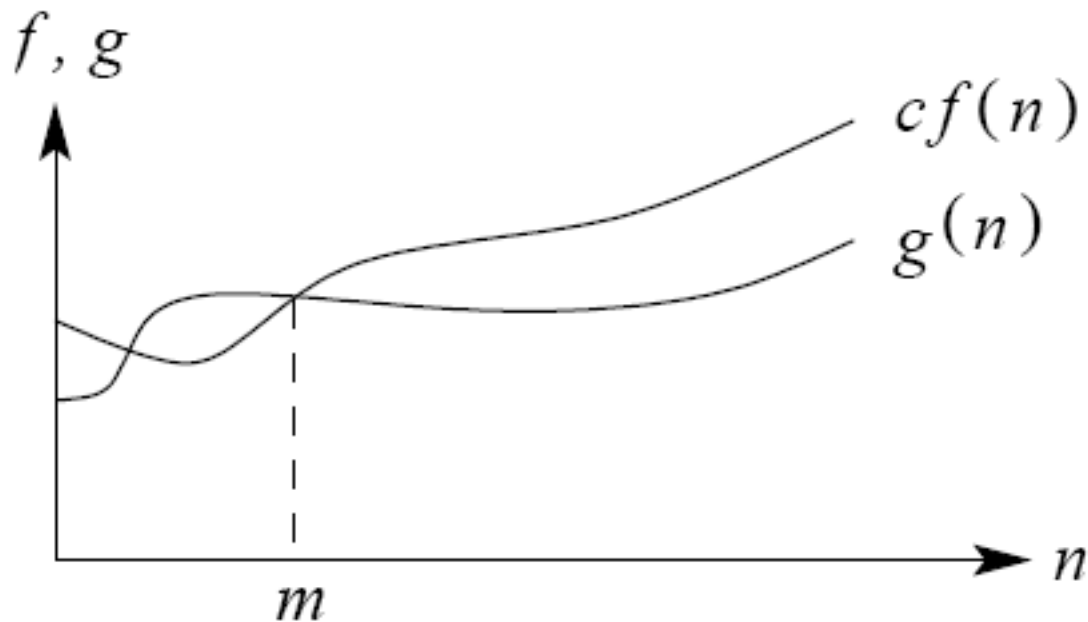




## Dominação assintótica

---

- Uma função  $f(n)$  **domina assintoticamente** outra função  $g(n)$  se existem duas constantes positivas  $c$  e  $m$  tais que, para  $n \geq m$ , temos  $|g(n)| \leq c \times |f(n)|$ .





## Dominação assintótica

---

- ▶ Exemplo: sejam  $g(n) = (n + 1)^2$  e  $f(n) = n^2$
- ▶ As funções  $g(n)$  e  $f(n)$  dominam assintoticamente uma a outra, desde que
  - ▶  $(n + 1)^2 \leq 4n^2$  para  $n \geq 1$
  - ▶  $n^2 \leq (n + 1)^2$  para  $n \geq 0$



## Notação O

---

- ▶ Para expressar que  $f(n)$  domina assintoticamente  $g(n)$ , escrevemos

$$g(n) = O(f(n))$$

- ▶ Lê-se  $g(n)$  é da ordem no máximo  $f(n)$ .



---

## Exercício

```
algoritmo3 (n: inteiro);  
inicio  
  para i de 1 até n faça  
    inicio  
      para j de i até 7*n faça  
        inicio  
          para k de 1 até 2*n faça  
            inicio  
              escreva("@");  
            fim  
          fim  
        fim  
      fim  
    fim  
  fim  
fim
```





## Para Casa

---



- ▶ Resenha

- ▶ Fazer uma resenha dos tópicos :

- ▶ 2.2 – Análise de Algoritmos
    - ▶ 3 – Crescimento de funções
    - ▶ Apêndice A – Somatórios

- ▶ Referência:

- ▶ CORMEN, Thomas H. et al. Algoritmos: teoria e prática – 2ed.

- ▶ Trazer na próxima aula

