Análise de Algoritmos

Prof. Luiz Gustavo Almeida Martins

Introdução

- **Análise de algoritmos:** área de estudo cujo foco é a eficiência dos algoritmos

Introdução

- **Análise de algoritmos:** área de estudo cujo foco é a eficiência dos algoritmos

- **Algoritmo:** um **conjunto finito e bem definido de ações** para a obtenção de uma solução para um determinado tipo de problema
  - Transforma os valores de **entrada** em valores de **saída**

Introdução

- **Análise de algoritmos:** área de estudo cujo foco é a eficiência dos algoritmos

- **Algoritmo:** um **conjunto finito e bem definido de ações** para a obtenção de uma solução para um determinado tipo de problema
  - Transforma os valores de **entrada** em valores de **saída**

- ""Um algoritmo é **correto** se, para cada instância de entrada, ele pára com a saída correta ou informa que não há solução para aquela entrada" (Cormen et al., 2002)

Introdução

- **Análise de algoritmos:** área de estudo cujo foco é a eficiência dos algoritmos

- **Algoritmo:** um **conjunto finito e bem definido de ações** para a obtenção de uma solução para um determinado tipo de problema
  - Transforma os valores de **entrada** em valores de **saída**

- ""Um algoritmo é **correto** se, para cada instância de entrada, ele pára com a saída correta ou informa que não há solução para aquela entrada" (Cormen et al., 2002)

- Questões:
  - Um algoritmo correto sempre termina?

Introdução

- **Análise de algoritmos:** área de estudo cujo foco é a eficiência dos algoritmos

- **Algoritmo:** um **conjunto finito e bem definido de ações** para a obtenção de uma solução para um determinado tipo de problema
  - Transforma os valores de **entrada** em valores de **saída**

- ""Um algoritmo é **correto** se, para cada instância de entrada, ele pára com a saída correta ou informa que não há solução para aquela entrada" (Cormen et al., 2002)

- Questões:
  - Um algoritmo correto sempre termina?
  - Só existe um algoritmo correto para um dado problema?

Exemplo

- Dado um mapa rodoviário, determinar a melhor rota entre os pontos *A* e *B*:
  - O número de rotas pode ser enorme
  - Diversas estratégias podem ser utilizadas para obter a melhor rota

Exemplo

- Dado um mapa rodoviário, determinar a melhor rota entre os pontos *A* e *B*:
  - O número de rotas pode ser enorme
  - Diversas estratégias podem ser utilizadas para obter a melhor rota

Exemplo

- Dado um mapa rodoviário, determinar a melhor rota entre os pontos *A* e *B*:
  - O número de rotas pode ser enorme
  - Diversas estratégias podem ser utilizadas para obter a melhor rota

Exemplo

- Dado um mapa rodoviário, determinar a melhor rota entre os pontos *A* e *B*:
  - O número de rotas pode ser enorme
  - Diversas estratégias podem ser utilizadas para obter a melhor rota

Eficiência

- Algoritmos diferentes são capazes de tratar um mesmo problema, mas **não** necessariamente **com a mesma eficiência**

- Essas diferenças de eficiência podem:
  - **Ser irrelevantes** para um **número pequeno de elementos** processados
  - **Aumentar proporcionalmente** com o crescimento no número de elementos
  - Existem problemas **intratáveis**, ou seja, para os quais não se conhece uma solução eficiente (***NP-difícil e NP-completo***)

- Deve-se considerar a **eficiência** de um algoritmo ao desenvolver seu software

- **Questão:**
  - Um algoritmo que permite obter uma solução em 3 anos é eficiente?

Eficiência

- ""Um algoritmo corresponde a uma descrição de um padrão de comportamento , expresso em termos de um conjunto finito de ações." (Dijkstra, 1971)

- **Ex:** Considere a instrução ***c = a + b***

  - Existe um padrão de comportamento nessa instrução, mesmo que ela seja realizada com valores diferentes para ***a*** e ***b***

  - Envolve 2 operações básicas (soma e atribuição)

Eficiência

- O projeto de algoritmos é fortemente influenciado pelo **estudo de seus comportamentos**

- Uma vez que o problema foi analisado e que as decisões sobre o projeto foram finalizadas é hora do analista **estudar as várias opções de algoritmos**

Eficiência

- Na análise de algoritmos exitem dois tipos de problemas bem distintos (Knuth, 1971):
  - **Análise de um algoritmo em particular:** Qual é o custo de usar um dado algoritmo para resolver um problema específico?
  - **Análise de uma classe de algortimos:** Qual o algoritmo de menor custo possível para resolver um problema particular?

Eficiência

- Uma das perguntas comuns em entrevistas no Google:
  - Qual a maneira mais eficiente de ordenar um milhão de inteiros de 32 bits?

- Para responder esta pergunta é importante associar o **algoritmo** a sua **eficiência**
- Eficiência está relacionada com o **custo computacional (complexidade)** do algoritmo

- **Análise de algoritmos**:
  - Definir a métrica de comparação
  - Usar uma metodologia para medir a complexidade
  - Comparar o custo dos algoritmos para determinar o mais eficiente

Complexidade computacional

- A complexidade computacional é uma medida que indica o **custo de se aplicar um algoritmo**
  - Associado ao consumo dos recursos computacionais:
    - **Tempo de execução** (**+ comum**)
    - **Espaço de armazenamento** utilizado
    - **Tráfego** gerado em uma rede de computadores
    - Etc.
- O custo computacional de um algorimto é dado **em função do tamanho da entrada** a ser processada

Exemplos de medidas de complexidade

- **Complexidade de tempo**:
  - Quando se mede o **tempo** necessário para executar um algoritmo para uma entrada de tamanho ***n***
    - O tempo de execução representa o número de vezes que uma determinada operação considerada relevante é executada
    - **NÃO** precisa representar o tempo de execução efetivo (real), mas sua ordem de grandeza (valor relativo)
- **Complexidade de espaço**:
  - Quando se mede a **quantidade de memória** necessária para se executar uma entrada de tamanho ***n***

Análise de algoritmos

- Permitir mapear um **problema** em uma **classe de algoritmos**
  - Encontrar a "**melhor**" **escolha** envolve a **comparação entre os algoritmos**, com base em seu **custo (eficiência)**

Análise de algoritmos

- Permitir mapear um **problema** em uma **classe de algoritmos**
  - Encontrar a "**melhor**" **escolha** envolve a **comparação entre os algoritmos**, com base em seu **custo (eficiência)**

- Podem ser usadas 2 abordagens:

  - **Análise matemática:** avalia as propriedades do algoritmo
    - Permite um estudo formal e prévio

Análise de algoritmos

- Permitir mapear um **problema** em uma **classe de algoritmos**
  - Encontrar a "**melhor**" **escolha** envolve a **comparação entre os algoritmos**, com base em seu **custo (eficiência)**

- Podem ser usadas 2 abordagens:

  - **Análise matemática:** avalia as propriedades do algoritmo
    - Permite um estudo formal e prévio

  - **Análise empírica ou experimental:** avalia o programa gerado
    - Permite comprovar a complexidade obtida na análise formal

Análise de algoritmos

- Permitir mapear um **problema** em uma **classe de algoritmos**
  - Encontrar a "**melhor**" **escolha** envolve a **comparação entre os algoritmos**, com base em seu **custo (eficiência)**

- Podem ser usadas 2 abordagens:

  - **Análise matemática:** avalia as propriedades do algoritmo
    - Permite um estudo formal e prévio

  - **Análise empírica ou experimental:** avalia o programa gerado
    - Permite comprovar a complexidade obtida na análise formal

  - Ambas são importantes

Análise empírica

- Um algoritmo é analisado a partir da **execução direta de seu programa** correspondente
  - Custo computacional é obtido através da avaliação da execução de uma versão implementada do algoritmo
  - Normalmente envolve várias execuções com diferentes entradas

Análise empírica

- **Vantagens:**
  - Permite avaliar o desempenho em uma determinada configuração do sistema
    - Ex: comparação de desempenho entre computadores, sistema operacionais e/ou linguagens de programação
  - Considera custos não aparentes
    - **Ex:** alocação dinâmica
  - Permite uma aferição mais precisa para o conjunto de dados testados

Análise empírica

- **Desvantagens e limitações:**
  - Exige a implementação do algoritmo
    - Experiência do programador pode influenciar no resultado
  - Envolve testar o algoritmo com diferentes entradas
  - Análise é feita sobre um conjunto limitado de dados
    - Influenciado pela natureza dos dados:
      - Dados reais, aleatórios (caso médio) ou ""perversos" (pior caso)
  - Afetada pelo hardware, sistema operacional, linguagem de programação, etc.
    - **Ex:** processos concorrentes no momento da avaliação

Análise empírica

- Código comumente usado para computar o tempo total de execução de um algoritmo

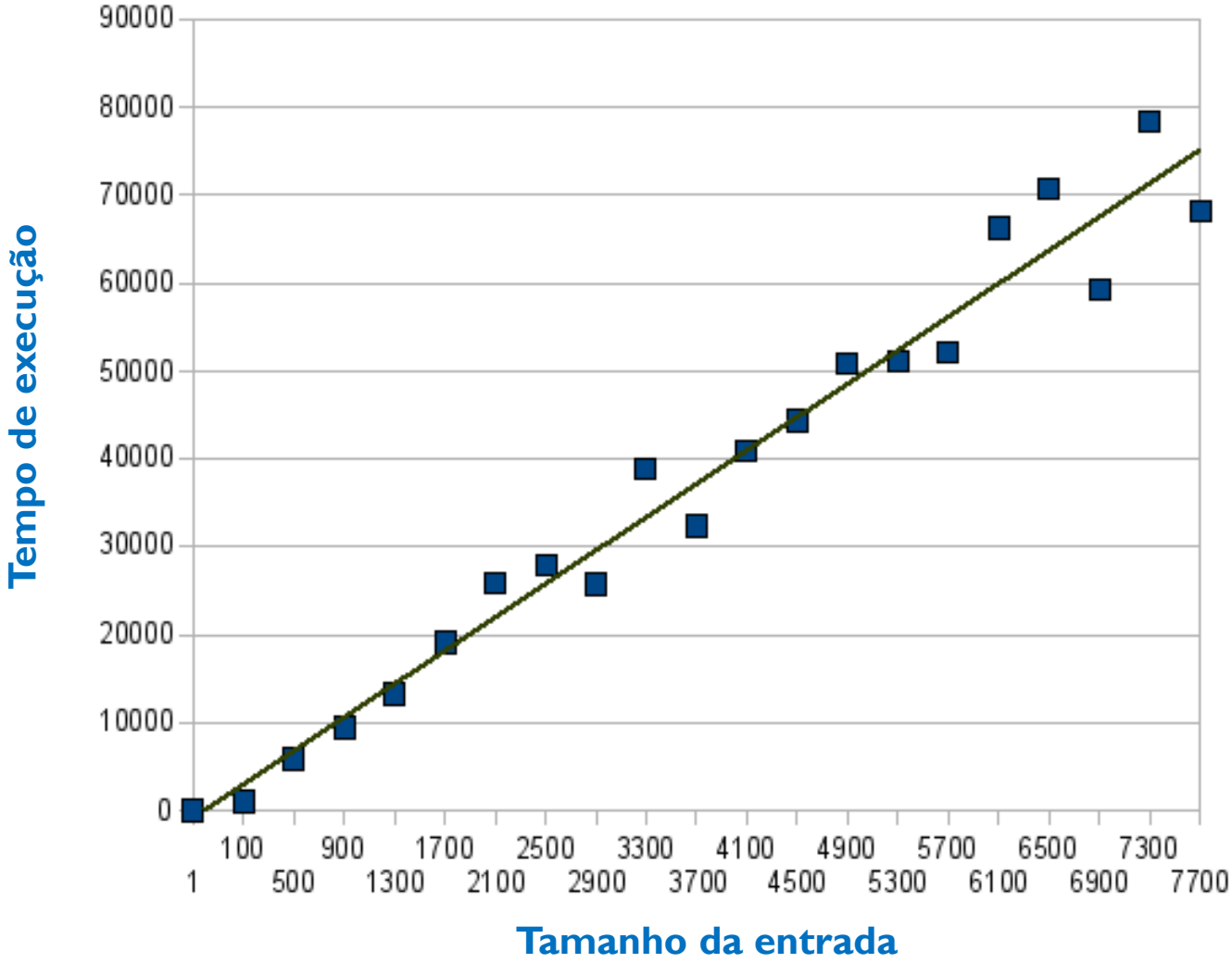
```
#include <stdio.h>
#include <time.h>

int main (void) {
   time_t t1, t2, total;
   t1 = time(NULL); // retorna hora atual do sistema

   /* algoritmo */

   t2 = time(NULL);
   total = difftime(t2,t1); // retorna a diferença t2-t1
   printf("\n\nTotal: %ld seg.\n", total);
   return 0;
}
```

Resultado de uma análise empírica



Análise matemática

- Permite o **estudo formal** de um algoritmo ao **nível conceitual**
  - Avalia a ideia do algoritmo
  - Algoritmo não precisa estar implementado

- Faz uso de um **computador idealizado** e simplificações que destacam o **custo dominante** do algoritmo
  - Estima o desempenho do algoritmo **independentemente** da máquina
  - Determina uma **função de custo ou complexidade** (modelo matemático)

- **Vantagens:**
  - Ignora detalhes de ""baixo nível"
  - Permite expressar a relação entre os dados de entrada e o custo computacional necessário para o processamento
    - Comportamento em função do crescimento do conjunto de entrada

Análise matemática

- Contabiliza o **nº de passos básicos** necessários para executar o algoritmo em função do **tamanho da entrada**
  - Reduz a análise ao **número de operações executadas**

- **Passos básicos:**
  - Referem-se às instruções simples que podem ser executadas diretamente pelo processador (ou algo próximo disto):
    - **Ex:** operações aritméticas, comparações, atribuições, etc.
  - Assumi-se que possuem o mesmo tempo de processamento (**mesmo custo**)

- **Tamanho da entrada** depende do problema
  - Está associado ao número de elementos processados
    - **Ex:** número de elementos em um arranjo, lista, árvore, etc.
  - Também pode ser a magnetude de um argumento de entrada
    - **Ex:** valor do número passado para a função fatorial

Análise matemática

- Considere a afirmação abaixo:

> *"Desenvolvi um novo algoritmo chamado TripleX que leva 14,2 segundos para processar 1.000 números, enquanto o método SimpleX leva 42,1 segundos."*

- **Questões:**
  - Você trocaria o SimpleX que já roda na sua empresa pelo TripleX?

Análise matemática

- Considere a afirmação abaixo:

> *"Desenvolvi um novo algoritmo chamado TripleX que leva 14,2 segundos para processar 1.000 números, enquanto o método SimpleX leva 42,1 segundos."*

- **Questões:**
  - Você trocaria o SimpleX que já roda na sua empresa pelo TripleX?
  - Será que o TripleX também é mais rápido para processar quantidades maiores que 1000 números?

Eficiência: TripleX vs. SimpleX

- Considere:
  - Um conjunto de entrada de tamanho ***n***

- **TripleX** **:** realiza $n^2 + n$ operações no processamento
  - **Função de custo:** $t(n) = n^2 + n$

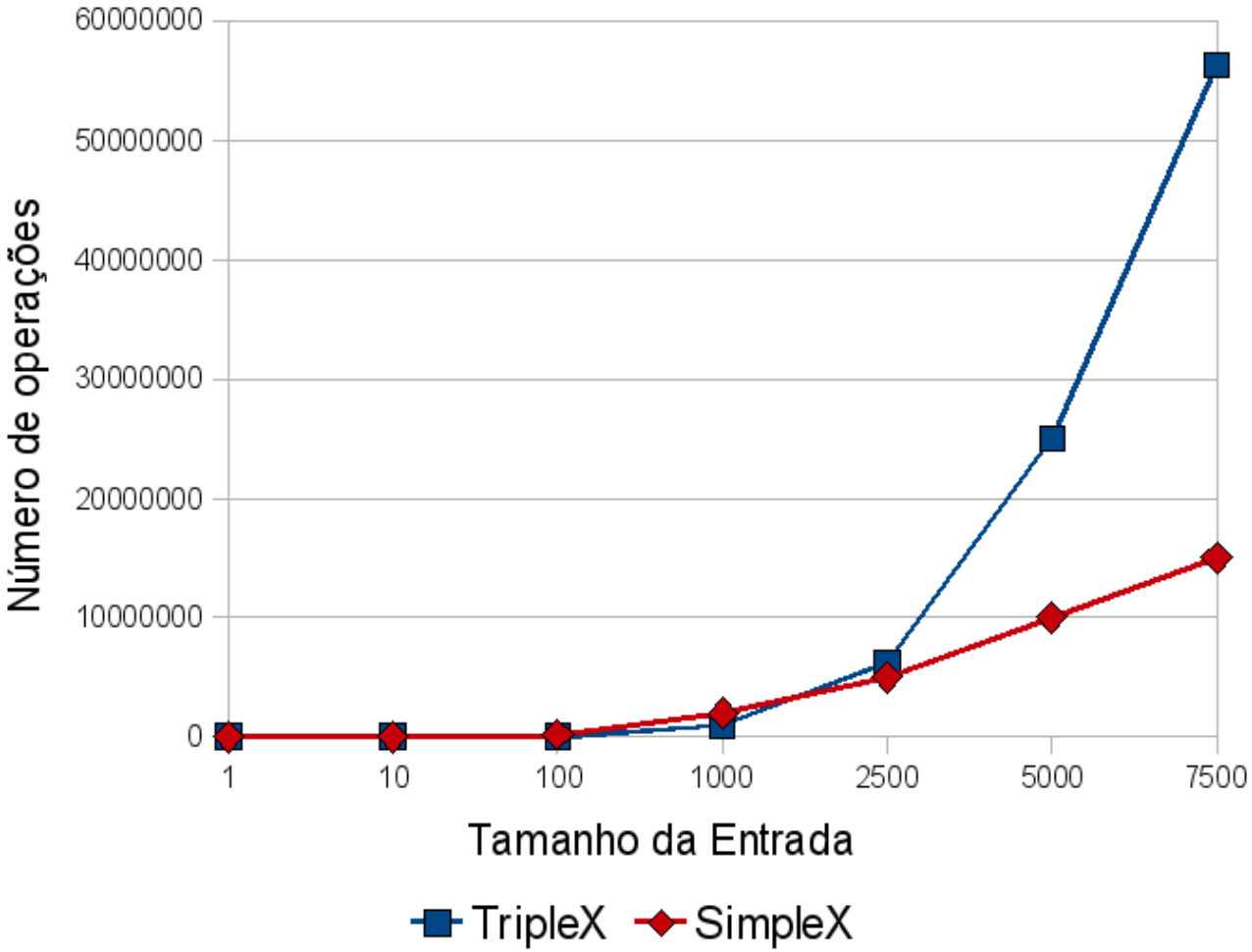
- **SimpleX** **:** realiza $2000n$ operações no processamento
  - **Função de custo:** $s(n) = 2000 * n$

Eficiência: TripleX vs. SimpleX

- Calculando o número de operações em função da entrada:

n	1	10	100	1.000	10.000
$t(n) = n^2+n$	2	110	10.100	1.001.000	100.010.000
$s(n) = 2000*n$	2.000	20.000	200.000	2.000.000	20.000.000

Eficiência: TripleX vs. SimpleX



Medida de Tempo de Execução

- **Exemplo 1**: Construa um algoritmo para a função:

int Max(int* A, int n)

Esse algoritmo deve retornar o maior elemento de um vetor de inteiros ***A*** de tamanho ***n*** (sendo ***n*** > 0)

- Seja ***f*** a função de custo tal que ***f(n)*** é o número de operações sobre os ***n*** elementos do vetor ***A***

Qual é f(n)????

Contagem de instruções

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

Contagem de instruções

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

Acesso a A[0]
+
Atribuição
=
2 instruções

Contagem de instruções

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

f(n) = 2

Contagem de instruções

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

f(n) = 2

Inicialização do for:
atribuição + comparação = 2

Contagem de instruções

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

f(n) = 2 + 2

Iteração do for:
incremento (2) +
comparação = 3

Contagem de instruções

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

$f(n) = 2 + 2 + 3(n-1)$

Iteração do for:
incremento (2) +
comparação = 3
(repete **n-1 vezes**)

Contagem de instruções

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

f(n) = 2 + 2 + 3(n-1) + 2 (n-1)

instr. de seleção:
acesso a A[i] + comparação = 2
(repete **n-1 vezes**)

Contagem de instruções

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

$f(n) = 2 + 2 + 3(n-1) + 2(n-1)$

acesso a A[i] + atribuição = 2

Contagem de instruções

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

f(n) = 2 + 2 + 3(n-1) + 2(n-1)

acesso a A[i] + atribuição = 2 (**depende do teste condicional**)

Contagem de instruções

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

$f(n) = 2 + 2 + 3(n-1) + 2(n-1)$

acesso a A[i] + atribuição = 2 (**depende do teste condicional**)

Melhor caso: 2 x **0**

Contagem de instruções

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

f(n) = 2 + 2 + 3(n-1) + 2(n-1)

acesso a A[i] + atribuição = 2 (**depende do teste condicional**)

Melhor caso: 2 x **0**

Pior caso: 2 x **(n-1)**

Contagem de instruções

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

$f(n) = 2 + 2 + 3(n-1) + 2(n-1)$

acesso a A[i] + atribuição = 2 (**depende do teste condicional**)

Melhor caso: 2 x **0**
Pior caso: 2 x **(n-1)**
Média: 2 x **(n-1)/2**

Contagem de instruções

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

f(n) = 2 + 2 + 3(n-1) + 2(n-1) + 2(n-1)

acesso a A[i] + atribuição = 2 (**depende do teste condicional**)

Melhor caso: 2 x **0**
Pior caso: 2 x **(n-1)**
Média: 2 x **(n-1)/2**

Contagem de instruções

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

$f(n) = 2 + 2 + 3(n-1) + 2(n-1) + 2(n-1) + 1$

retorno da função
=
1 instrução

Contagem de instruções

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

Análise do pior caso:

$f(n) = 2 + 2 + 3(n-1) + 2(n-1) + \mathbf{2(n-1)} + 1$

$$\boldsymbol{f(n) = 7(n-1) + 5}$$

Contagem das instruções dominantes

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

Simplificação:

Na análise matemática podemos nos concentrar nas **operações que dominam o processo**

Contagem das instruções dominantes

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

Simplificação:

Na análise matemática podemos nos concentrar nas **operações que dominam o processo**

Forte relação com os laços (*loops*) e seus aninhamentos

Contagem das instruções dominantes

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

f(n) pode ser representada em função do **número de comparações**

Contagem das instruções dominantes

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

$f(n) = n-1$, para $n > 0$

f(n) pode ser representada em função do **número de comparações**

Contagem das instruções dominantes

```
int  Max(int* A, int n)
inicio
    declare i e temp como inteiro;
    temp = A[0];
    para i = 1 até  i < n passo de 1
        se  temp < A[i]
            temp = A[i];
        fim_se
    fim_para
    returna temp;
fim
```

$f(n) = n-1$, para $n > 0$

Nota: nesse caso a função de custo é **uniforme**, ou seja, é independente da organização interna dos dados no vetor

Medida de Tempo de Execução

Teorema: qualquer algoritmo para encontrar o maior elemento de um conjunto com n elementos, $n > 0$, faz pelo menos $n-1$ comparações.

Prova: cada um dos $n-1$ elementos têm que ser avaliados, por meio de comparações, para se saber se ele é maior que algum outro elemento. Logo $n-1$ comparações são necessárias. Não importa a ordem dos números, sempre serão necessária $n-1$ comparações.

Medida de Tempo de Execução

Não é possível melhorar o algoritmo

Algoritmo já pode ser implementado

```
int Max(int* A, int n)
{
    int I, temp;
    temp = A[0];
    for (i = 1; i < n; i++)
        if ( temp < A[i])
            temp = A[i];
    return temp;
}
```

Medida de Tempo de Execução

- **Exemplo 2**: escreva o algoritmo para a função:

void MaxMin(int *A, int n, int *max, int *min)

o qual encontra o maior e o menor elemento de um dado vetor ***A*** de tamanho ***n*** > *0*.

- Seja ***f*** a função de custo, tal que ***f(n)*** é o número de operações entre os ***n*** elementos de ***A***

Qual é f(n)????

Medida de Tempo de Execução

```
void MaxMin1(int *A, int n, int *max, int *min)
{
    int i;
    *max = A[0]; *min = A[0];
     for (i=1; i<n; i++) {
       if A[i] > *max
                  *max = A[i];
       if A[i] < *Min
                  *min = A[i];
    }
}
```

Medida de Tempo de Execução

```
void MaxMin1(int *A, int n, int *max, int *min)
{
    int i;
    *max = A[0]; *min = A[0];
     for (i=1; i<n; i++) {
       if A[i] > *max
                    *max = A[i];
       if A[i] < *Min
                    *min = A[i];
    }
}
```

Medida de Tempo de Execução

```
void MaxMin1(int *A, int n, int *max, int *min)
{
    int i;
    *max = A[0]; *min = A[0];
     for (i=1; i<n; i++) {
       if A[i] > *max
                *max = A[i];
       if A[i] < *Min
                *min = A[i];
    }
}
```

f(n) = 2(n-1)

Medida de Tempo de Execução

- Pequena modificação na função ***MinMax()***

```
void MaxMin2(int *A, int n, int *max, int *min)
{
    int i;
    *max = A[0]; *min = A[0];
     for (i=1; i<n; i++)
       if (A[i] > *max)
                  *max = A[i];
      else  if A[i] < *Min
                  *min = A[i];
}
```

Medida de Tempo de Execução

- Pequena modificação na função ***MinMax()***

```
void MaxMin2(int *A, int n, int *max, int *min)
{
    int i;
    *max = A[0]; *min = A[0];
     for (i=1; i<n; i++)
       if (A[i] > *max)
                  *max = A[i];
      else  if A[i] < *Min
                  *min = A[i];
}
```

Qual é *f(n)*?

Medida de Tempo de Execução

- Pequena modificação na função ***MinMax()***

```
void MaxMin2(int *A, int n, int *max, int *min)
{
    int i;
    *max = A[0]; *min = A[0];
     for (i=1; i<n; i++)
       if (A[i] > *max)
                  *max = A[i];
      else  if A[i] < *Min
                  *min = A[i];
}
```

Qual é *f(n)*?

Depende da organização dos dados no vetor

Medida de Tempo de Execução

- Análise de acordo com a organização dos dados:
  - **Melhor caso:** os dados do vetor em ordem crescente

$$f(n) = n-1$$

Medida de Tempo de Execução

- Análise de acordo com a organização dos dados:
  - **Melhor caso:** os dados do vetor em ordem crescente

$$f(n) = n-1$$

  - **Pior caso:** os dados do vetor em ordem descrescente

$$f(n) = 2(n-1)$$

Medida de Tempo de Execução

- Análise de acordo com a organização dos dados:
  - **Melhor caso:** os dados do vetor em ordem crescente

$$f(n) = n-1$$

  - **Pior caso:** os dados do vetor em ordem descrescente

$$f(n) = 2(n-1)$$

  - **Caso médio**:

$$f(n) = (n-1 + 2(n-1))/2 = 3n/2 – 3/2$$

Medida de Tempo de Execução

- **Ainda dá para melhorar o algoritmo?**
  - Comparar os elementos de ***A*** aos pares, separando em dois subconjuntos, um para os maiores entre os pares e outra para os menores entre os pares

$$f_{pares}(n) = n/2$$

  - O maior elemento é obtido do subconjunto de máximos

$$f_{maior}(n) = n/2 - 1$$

  - O menor elemento é obtido do subconjunto de mínimos

$$f_{menor}(n) = n/2 - 1$$

$$f(n) = 3n/2 - 2 \text{ , para } n > 0$$

Medida de Tempo de Execução

```
void MaxMin3(int *A, int n, int *max,
int *min)
{
    int i, fim = n;
   if (n % 2 == 1) //  n é impar
      fim = n-1;
   if (A[0] > A[1]) {
            *min = A[1]; *max = A[0]; }
  else {*min = A[0]; *max = A[1]; }

//continua
```

```
i= 2;
   while(i < fim)   {
        if (A[i] > A[i+1])
        {
           if (A[i] > *max)  *max = A[i];
           if (A[i+1] < *min) *min = A[i+1];
        }
      else {
              if (A[i] < *min)  *min = A[i];
             if (A[i+1] > *max) *max = A[i+1];
             }
      i = i+2;
   }
   if ( fim == n-1)  //  nro impar de elementos
       if (A[n-1] > *max) *max = A[n-1];
       else if ((A[n-1] < *min) *min = A[n-1];
}
```

Comparação entre os algoritmos

Os três algorimos	***f(n)***		
	Melhor caso	Pior caso	Caso médio
MaxMin1	2(n-1)	2(n-1)	2(n-1)
MaxMin2	n-1	2(n-1)	3n/2 – 3/2
MaxMin3	3n/2 - 2	3n/2 - 2	3n/2 - 2

Análise do problema MinMax

- É possível construir uma solução com uma função de complexidade menor?

Análise do problema MinMax

- O problema pode ser modelado por um quádrupla (a,b,c,d), sendo:
  - **a**: Nº de elementos que nunca foram comparados
  - **b**: Nº de elementos foram vencedores e nunca perderam em comparações realizadas
  - **c**: Nº de elementos foram perdedores e nunca venceram em comparações realizadas
  - **d**: Nº de elementos que foram vencedores ou perdedores em comparações realizadas.

Análise do problema MinMax

- E por seis regras:
  - (a-2,b+1,c+1,d) se a >=2 {dois elementos de a são comparados}
  - (a-1,b+1,c,d) ou
  - (a-1,b,c+1,d) ou { um elemento de a comparado com um de b ou c}
  - (a-1,b,c,d+1) se a >= 1
  - (a,b-1,c,d+1) se b >= 2 {dois elementos de b são comparados}
  - (a,b,c-1,d+1) se c >= 2 {dois elementos de c são comparados}

Análise do problema MinMax

- **Cenário inicial:** ***(n , 0 , 0 , 0)***
- **Cenário desejado (final):** ***(0 , 1 , 1 , n-2)***

Análise do problema MinMax

- **Cenário inicial:** ***(n , 0 , 0 , 0)***
- **Cenário desejado (final):** ***(0 , 1 , 1 , n-2)***
- Levar ***a*** para **0** requer ***n/2*** comparações: ***(0 , n/2 , n/2 , 0)***

Análise do problema MinMax

- **Cenário inicial:** ***(n , 0 , 0 , 0)***
- **Cenário desejado (final):** ***(0 , 1 , 1 , n-2)***
- Levar ***a*** para **0** requer ***n/2*** comparações: ***(0 , n/2 , n/2 , 0)***
- Reduzir ***b*** para ***1*** precisa de ***(n/2) - 1*** comparações: ***(0 , 1 , n/2 , n/2-1)***

Análise do problema MinMax

- **Cenário inicial:** ***(n , 0 , 0 , 0)***
- **Cenário desejado (final):** ***(0 , 1 , 1 , n-2)***
- Levar ***a*** para **0** requer ***n/2*** comparações: ***(0 , n/2 , n/2 , 0)***
- Reduzir ***b*** para ***1*** precisa de ***(n/2) - 1*** comparações: ***(0 , 1 , n/2 , n/2-1)***
- Reduzir ***c*** para ***1*** precisa de ***(n/2) - 1*** comparações: ***(0 , 1 , 1 , n-2)***

Análise do problema MinMax

- **Cenário inicial:** ***(n , 0 , 0 , 0)***
- **Cenário desejado (final):** ***(0 , 1 , 1 , n-2)***
- Levar ***a*** para **0** requer ***n/2*** comparações: ***(0 , n/2 , n/2 , 0)***
- Reduzir ***b*** para ***1*** precisa de ***(n/2) - 1*** comparações: ***(0 , 1 , n/2 , n/2-1)***
- Reduzir ***c*** para ***1*** precisa de ***(n/2) - 1*** comparações: ***(0 , 1 , 1 , n-2)***
- **Conclusão: o menor custo do algoritmo é:**

$$f(n) = n/2 + n/2 - 1 + n/2 - 1 = 3n/2 - 2$$

Contagem de instruções

- **Problema:**

  Nem sempre a contagem das instruções é trivial

Contagem de instruções

- **Problema:**

  Nem sempre a contagem das instruções é trivial

  Pode envolver a resolução de séries matemáticas

Contagem de instruções

- **Problema:**

  Nem sempre a contagem das instruções é trivial

  Pode envolver a resolução de séries matemáticas

  Ex: algoritmos de ordenação simples

```
for (x=0; x<n-1; x++)
    for (y=x+1; y<n-1; y++)
```

Contagem de instruções

- **Problema:**

  Nem sempre a contagem das instruções é trivial

  Pode envolver a resolução de séries matemáticas

  Ex: algoritmos de ordenação simples

```
for (x=0; x<n-1; x++)
    for (y=x+1; y<n-1; y++)
```

Nº de iterações variável (progressão)

Revisão de matemática

- Séries

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$$

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$$

Análise assintótica

- **O custo** para obter uma solução para um dado problema **aumenta com o tamanho *n* da entrada**
  - Para *n* pequeno, **a escolha** do algoritmo **não é crítica**
  - Análise deve focar quando *n* assume **valores elevados**
- Análise estuda o **comportamento assintótico** da função
  - Representa o **limite** de comportamento do custo à medida que *n* cresce (**eficiência com base na ordem de crescimento**)
    - Comportamento da função quando *n* tende a infinito
  - Reduz o problema a uma **resposta menos precisa**, mas **fácil de derivar e de interpretar**

Comportamento assintótico

- Comparação de algoritmos envolve determinar suas ordens de crescimento (**eficiência assintótica**)
  - O algoritmo com a **menor** ordem de crescimento deverá executar mais rápido

- Algumas “consequências” desse tipo de análise:
  - Usa um **modelo de máquina único** com as operações básicas

  - Eficiência do algoritmo pode estar relacionada à detalhes dos dados de entrada além do seu tamanho
    - **Ex:** organização dos dados no vetor
    - Análise de diferentes cenários: **melhor caso**, **pior caso** e **caso médio**

Comportamento assintótico

- **Melhor caso**: não é uma boa análise
  - Nivela os algoritmos por baixo
  - O custo pode não ser alcançado na prática

Comportamento assintótico

- **Melhor caso**: não é uma boa análise
  - Nivela os algoritmos por baixo
  - O custo pode não ser alcançado na prática

- **Caso médio**: é o ideal (intuitivamente) em algumas situações
  - Seu cálculo pode não ser uma tarefa trivial
  - É preciso conhecer a **distribuição de probabilidade** típica da entrada e utilizar teoria da probabilidade
    - Descrição da chance de uma variável aleatória assumir um determinado valor

Comportamento assintótico

- **Melhor caso**: não é uma boa análise
  - Nivela os algoritmos por baixo
  - O custo pode não ser alcançado na prática

- **Caso médio**: é o ideal (intuitivamente) em algumas situações
  - Seu cálculo pode não ser uma tarefa trivial
  - É preciso conhecer a **distribuição de probabilidade** típica da entrada e utilizar teoria da probabilidade
    - Descrição da chance de uma variável aleatória assumir um determinado valor

- **Pior caso**: recomendado e mais utilizado
  - Fácil de identificar
  - Representa o **limite superior** do tempo de execução em função entrada
    - Nenhuma execução do algoritmo será pior que isto
  - Ocorre com frequência em muitos algoritmos

Comportamento assintótico

- Considere o custo de execução de um algoritmo:
  - **Pior caso:** $2+3n+2+3\times(2n)+1 = \mathbf{9n+5}$ instruções
  - **Melhor caso:** $2+3n+2+2\times(2n)+1 = \mathbf{7n+5}$ instruções

Comportamento assintótico

- Considere o custo de execução de um algoritmo:
  - **Pior caso:** $2+3n+2+3\times(2n)+1 = \mathbf{9n+5}$ instruções
  - **Melhor caso:** $2+3n+2+2\times(2n)+1 = \mathbf{7n+5}$ instruções

- Suponha ainda que:
  - t_1 é o tempo gasto pela instrução mais rápida
  - t_2 é o tempo gasto pela instrução mais lenta

Comportamento assintótico

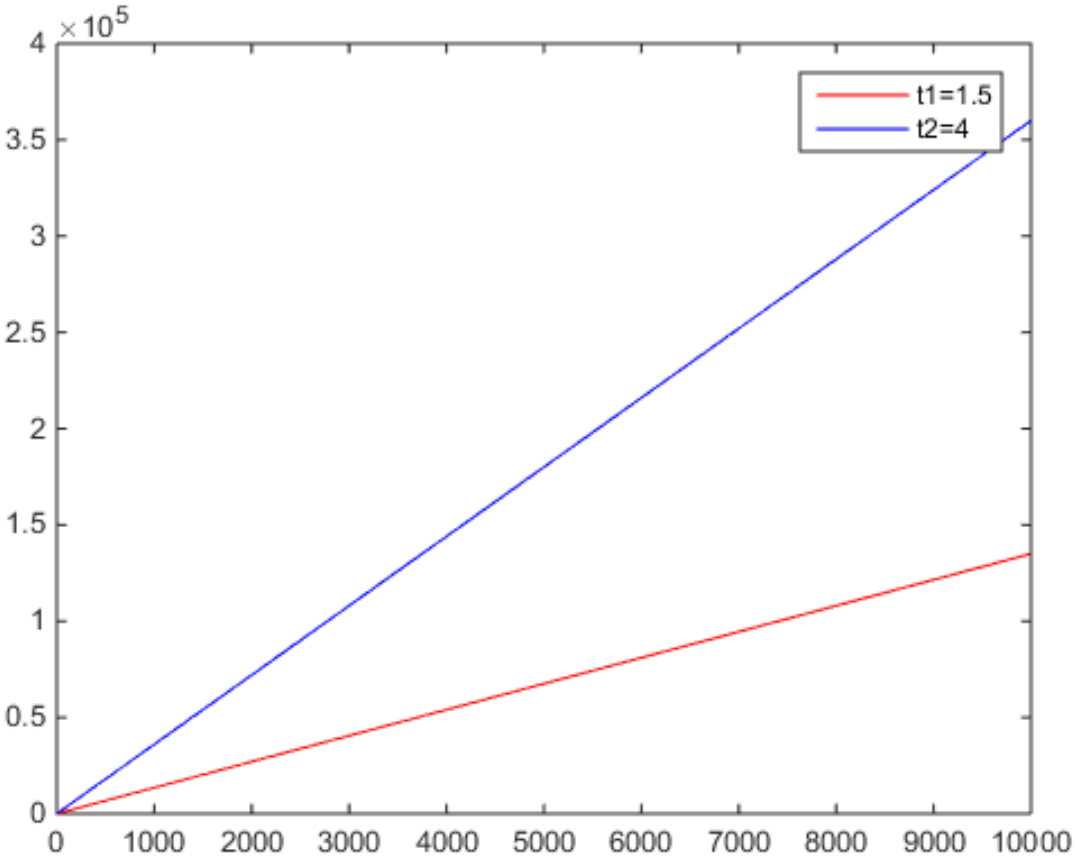
- Considere o custo de execução de um algoritmo:
  - **Pior caso:** $2+3n+2+3\times(2n)+1 = \mathbf{9n+5}$ instruções
  - **Melhor caso:** $2+3n+2+2\times(2n)+1 = \mathbf{7n+5}$ instruções

- Suponha ainda que:
  - t_1 é o tempo gasto pela instrução mais rápida
  - t_2 é o tempo gasto pela instrução mais lenta

- O tempo de execução no pior caso $T(n)$ é limitado por duas **funções lineares**

$$t_1(9n+5) \leq T(n) \leq t_2(9n+5)$$

Gráfico do custo de execução



Exemplo: t_1 = 1,5 e t_2 = 4,0

Comportamento assintótico

- **Questão:** o que podemos concluir sobre a eficiência do algoritmo analisado quando *n* aumenta?

Comportamento assintótico

- **Questão:** o que podemos concluir sobre a eficiência do algoritmo analisado quando *n* aumenta?
  - A função que caracteriza a complexidade computacional do algoritmo é de ordem **linear**

Comportamento assintótico

- **Questão:** o que podemos concluir sobre a eficiência do algoritmo analisado quando *n* aumenta?
  - A função que caracteriza a complexidade computacional do algoritmo é de ordem **linear**
  - **Fatores constantes** podem ser desprezados
    - Não se alteram a medida que *n* aumenta
    - Não afetam o comportamento da taxa de crescimento

Comportamento assintótico

- **Questão:** o que podemos concluir sobre a eficiência do algoritmo analisado quando *n* aumenta?
  - A função que caracteriza a complexidade computacional do algoritmo é de ordem **linear**
  - **Fatores constantes** podem ser desprezados
    - Não se alteram a medida que *n* aumenta
    - Não afetam o comportamento da taxa de crescimento
  - **Simplificação:** $T(n) = 9n+5 \approx n$

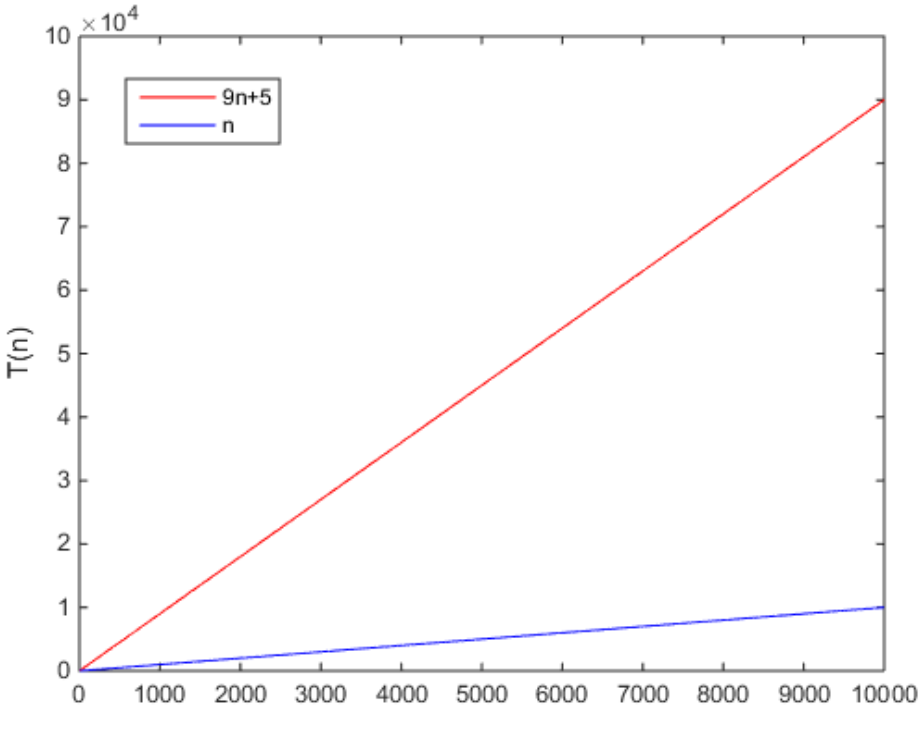
Comportamento assintótico

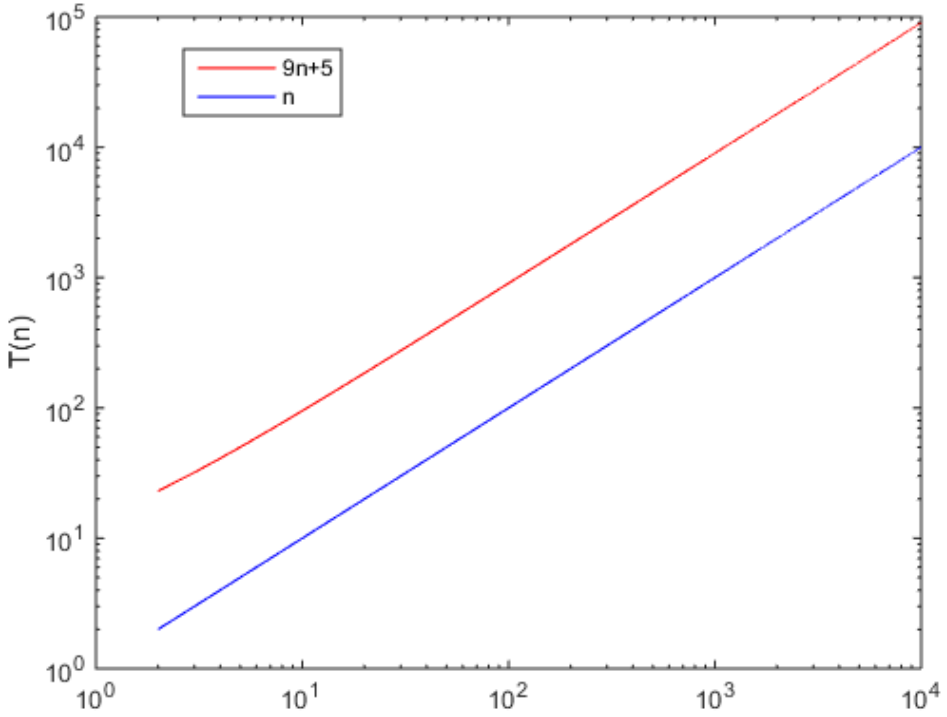
- **Questão:** o que podemos concluir sobre a eficiência do algoritmo analisado quando *n* aumenta?
  - A função que caracteriza a complexidade computacional do algoritmo é de ordem **linear**

T(n)	1	10	100	1.000	10.000
9n+5	14	95	905	9.005	90.005
9n	9	90	900	9.000	90.000
n	1	10	100	1.000	10.000

Gráfico do comportamento assintótico

- **Fatores constantes** não afetam a forma de crescimento





OBS: *a escala logarítmica permite visualizar a taxa de crescimento (**crescimento similar**)*

Comportamento assintótico

- **Fatores constantes** não afetam a ordem de complexidade da função de custo do algoritmo
  - Não são alterados com o aumento no tamanho da entrada (n)
  - Mudanças no ambiente de execução (HW e SW) afetam a taxa de crescimento por um **fator constante**

Comportamento assintótico

- **Fatores constantes** não afetam a ordem de complexidade da função de custo do algoritmo
  - Não são alterados com o aumento no tamanho da entrada (*n*)
  - Mudanças no ambiente de execução (HW e SW) afetam a taxa de crescimento por um **fator constante**

- **Termos de menor ordem** também não afetam o comportamento do crescimento
  - **Crescem mais lentamente** a medida que *n* aumenta

Comportamento assintótico

- **Fatores constantes** não afetam a ordem de complexidade da função de custo do algoritmo
  - Não são alterados com o aumento no tamanho da entrada (*n*)
  - Mudanças no ambiente de execução (HW e SW) afetam a taxa de crescimento por um **fator constante**

- **Termos de menor ordem** também não afetam o comportamento do crescimento
  - **Crescem mais lentamente** a medida que *n* aumenta

- Apenas o termo que representa o **comportamento assintótico** da função de custo do algoritmo é mantido
  - **Termo de maior ordem** domina o comportamento de *f(n)* quando *n* tende ao infinito

Comportamento assintótico

- **Influência dos termos de menor ordem:**

T(n)	1	10	100	1.000
n^2	1	100	10.000	1.000.000
n^2+n	2	110	10.100	1.001.000
Δ	100%	10%	1%	0,1%

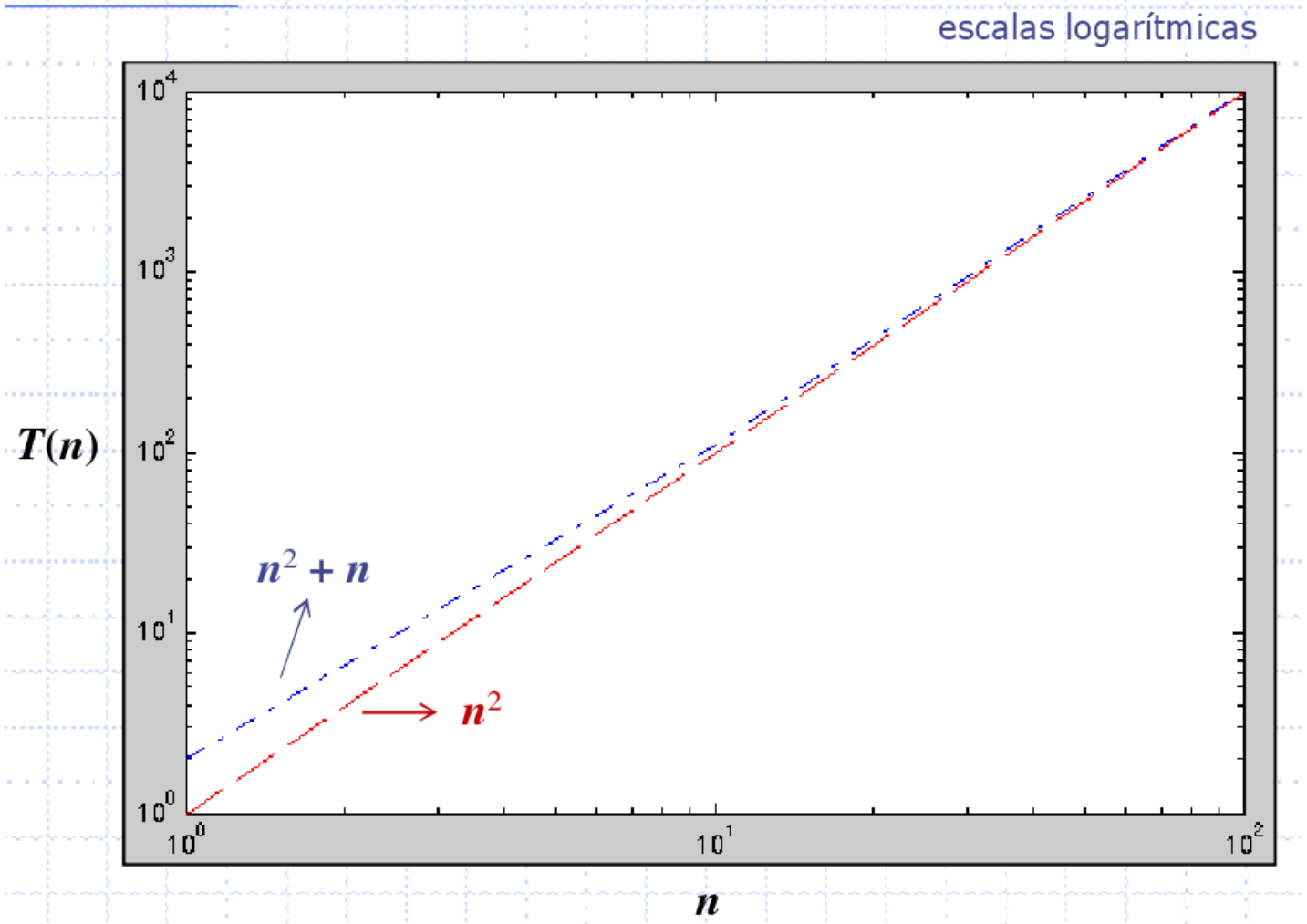
Comportamento assintótico

- **Influência dos termos de menor ordem:**

T(n)	1	10	100	1.000
n^2	1	100	10.000	1.000.000
n^2+n	2	110	10.100	1.001.000
Δ	100%	10%	1%	0,1%

diminui a medida que *n* aumenta

Comportamento assintótico



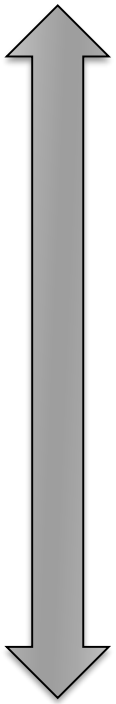
Fonte da figura: notas de aula do Prof. Ricardo Campello

Classes de comportamento assintótico

- As principais **classes de comportamento assintótico** são representadas pelas seguintes funções:
  - Constante (≈ 1)
  - Logarítmica ($\approx \log_a n$)
  - Linear ($\approx n$)
  - Log linear ($\approx n \times \log_a n$)
  - Quadrática ($\approx n^2$)
  - Cúbica ($\approx n^3$)
  - Exponencial ($\approx a^n$)
  - Fatorial ($\approx n!$)

Classes de comportamento assintótico

- As principais **classes de comportamento assintótico** são representadas pelas seguintes funções:
  - Constante (≈ 1)
  - Logarítmica (≈ $log_a\ n$)
  - Linear (≈ n)
  - Log linear (≈ $n \times log_a\ n$)
  - Quadrática (≈ n^2)
  - Cúbica (≈ n^3)
  - Exponencial (≈ a^n)
  - Fatorial (≈ $n!$)



< custo
+ eficiente

> custo
- eficiente

Classes de comportamento assintótico

- **Constante**
  - Independe do tamanho de *n*
  - Instruções executadas um número fixo de vezes
  - **Ex:** Remover o topo de uma pilha (desempilhar)

- **Logarítmica**
  - Típica de algoritmos que resolvem um problema transformando-o em problemas menores
  - **Ex:** busca binária

- **Linear**
  - Melhor cenário para problemas que precisam processar *n* entradas para obter *n* saídas
  - Uma certa qtde. de operações é aplicada em cada elemento de entrada
  - **Ex:** Obter o maior elemento de um vetor

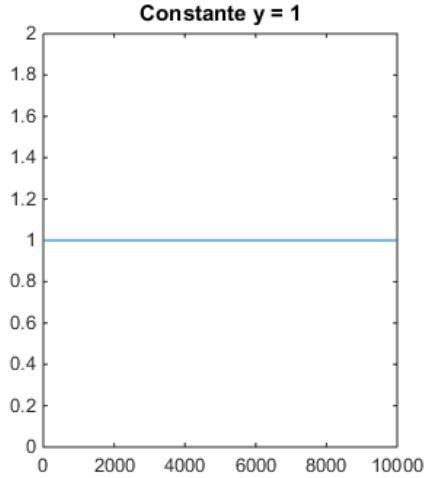
Classes de comportamento assintótico

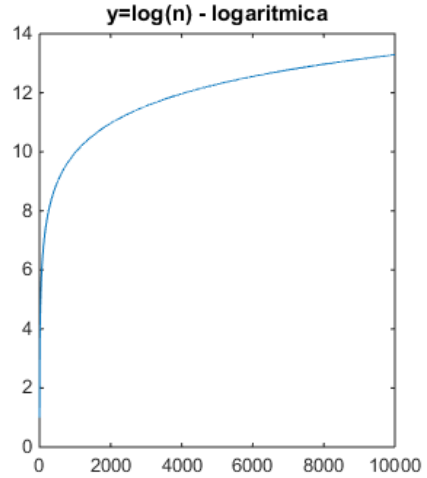
- **Log linear**
  - Engloba algoritmos que resolvem um problema transformando-o em problemas menores
    - Resolve cada um de forma independente e mescla as soluções
  - **Ex:** *quick sort*

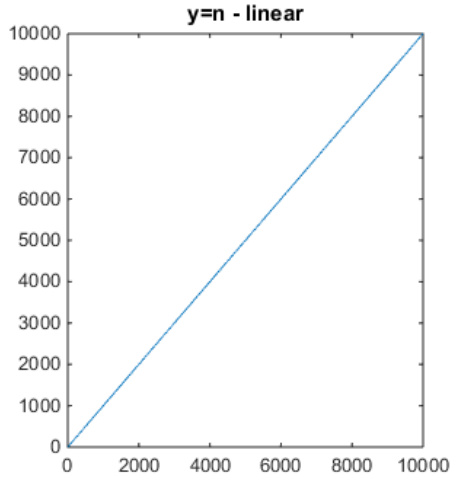
- **Quadrática e cúbica**
  - Ocorre quando os dados de entrada são processados através de laços aninhados e relacionados
    - Potência está relacionada à quantidade de laços aninhados
  - **Ex:** *bubble sort* (n^2) e multiplicação de matrizes (n^3)

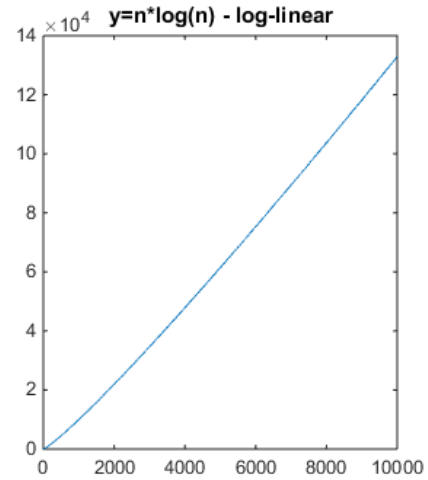
- **Exponencial e fatorial**
  - Geralmente envolvem soluções baseadas em "**força bruta**"
    - Não são viáveis na prática
  - Fatorial é uma exponencial com comportamento muito pior

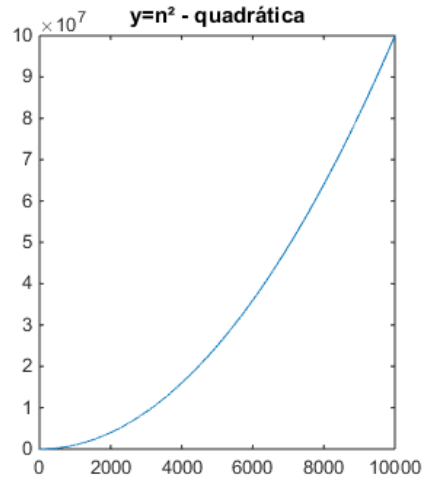
Classes de comportamento assintótico

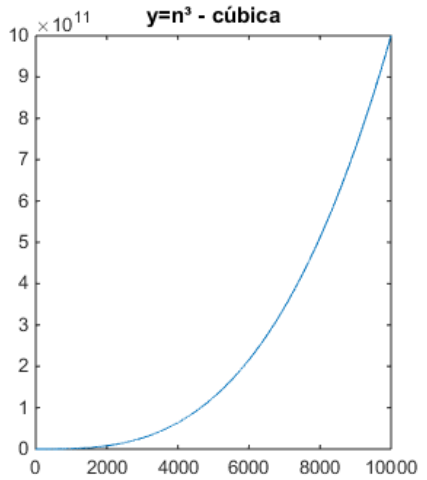












Observe a ordem de grandeza no eixo y

Classes e tempo de execução

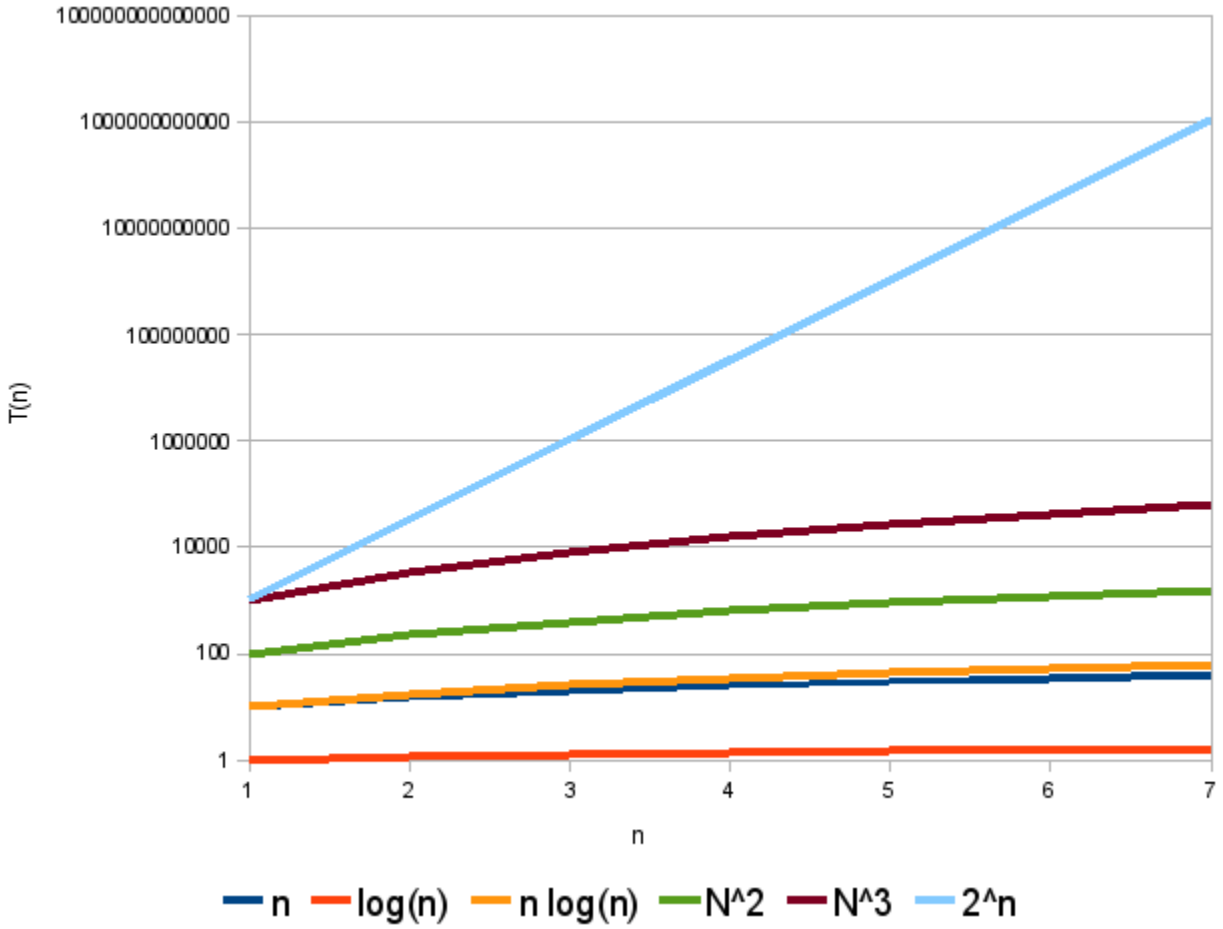
segundos
minutos
séculos

Características Aproximadas do Hardware	
Número de Instruções executadas por Ciclo do relógio (IPC)	8
Freqüência (1 / período do ciclo em min.)	3E+09
No. de Instruções por minuto	24E+09

$T(n)$	$n = 20$	$n = 40$	$n = 60$	$n = 80$
n	5,3E-08	1,1E-07	1,6E-07	2,1E-07
$n \log n$	2,3E-07	5,7E-07	9,5E-07	1,3E-06
n^2	1,1E-06	4,3E-06	9,6E-06	1,7E-05
n^3	2,1E-05	1,7E-04	5,8E-04	1,4E-03
2^n	2,8E-03	48,9	1,0	1,0E+06
3^n	0,2	5,4E+08	1,9E+18	6,6E+27

Fonte da figura: notas de aula do Prof. Ricardo Campello

Gráfico comparativo das classes assintóticas



Exemplo: em escala logarítmica – para ser possível visualizar todas as função ao mesmo tempo

Exercícios

- Qual a classe de comportamento assintótico para cada uma das funções de custo abaixo:

$f_1(n) = n^2 + 10n + 5$

$f_2(n) = 10n^2 + 5n + n^3 + 1000$

$f_3(n) = 5347$

$f_4(n) = 5n \log n + 10$

$f_5(n) = 5n + \log n$

$f_6(n) = 3n(n-1) + 10$

$f_7(n) = 3n^2 + 2^n$

Exercícios

- Qual a classe de comportamento assintótico para cada uma das funções de custo abaixo:

$f_1(n) = n^2 + 10n + 5$	$f_1(n) \approx \mathbf{n^2}$
$f_2(n) = 10n^2 + 5n + n^3 + 1000$	$f_2(n) \approx \mathbf{n^3}$
$f_3(n) = 5347$	$f_3(n) \approx \mathbf{1}$
$f_4(n) = 5n \log n + 10$	$f_4(n) \approx \mathbf{n \log n}$
$f_5(n) = 5n + \log n$	$f_5(n) \approx \mathbf{n}$
$f_6(n) = 3n(n-1) + 10$	$f_6(n) \approx \mathbf{n^2}$
$f_7(n) = 3n^2 + 2^{n+1}$	$f_7(n) \approx \mathbf{2^n}$

Exercícios

- Determine a função de custo do algoritmo abaixo, em função do tempo de execução, e sua classe de comportamento assintótico.

```
Inicio
   i, j, A[n]: inteiro
   i = 1;
   enquanto (i < n) faca
       A[i] = 0;
       i = i + 1;
   fim_enquanto
   para i = 1 ate n faca
       para j = 1 ate n faca
                A[i] = A[i] + (i*j);
        fim_para
   fim_para
fim
```

Exercícios

- Determine a função de custo do algoritmo abaixo, em função do tempo de execução, e sua classe de comportamento assintótico.

```
Inicio
   i, j, A[n]: inteiro
   i = 1;
   enquanto (i < n) faca
       A[i] = 0;
       i = i + 1;
   fim_enquanto
   para i = 1 ate n faca
       para j = 1 ate n faca
              A[i] = A[i] + (i*j);
     fim_para
   fim_para
fim
```

$f_1(n) = 1$

Exercícios

- Determine a função de custo do algoritmo abaixo, em função do tempo de execução, e sua classe de comportamento assintótico.

```
Inicio
   i, j, A[n]: inteiro
   i = 1;                          ---------->   f1(n) = 1
   enquanto (i < n) faca
       A[i] = 0;                                 f2(n) = 5n
       i = i + 1;
   fim_enquanto
   para i = 1 ate n faca
       para j = 1 ate n faca
              A[i] = A[i] + (i*j);
     fim_para
   fim_para
fim
```

Exercícios

- Determine a função de custo do algoritmo abaixo, em função do tempo de execução, e sua classe de comportamento assintótico.

```
Inicio
   i, j, A[n]: inteiro
   i = 1;                                  → f1(n) = 1
   enquanto (i < n) faca                   ┐
       A[i] = 0;                           │ f2(n) = 5n
       i = i + 1;                          │
   fim_enquanto                            ┘
   para i = 1 ate n faca                   ┐
       para j = 1 ate n faca               │ f3(n) = 6n
              A[i] = A[i] + (i*j);         │
     fim_para                              │ f4(n) = 2n x f3(n)
   fim_para                                │       = 12n²
fim                                        ┘
```

Exercícios

- Determine a função de custo do algoritmo abaixo, em função do tempo de execução, e sua classe de comportamento assintótico.

```
Inicio
   i, j, A[n]: inteiro
   i = 1;
   enquanto (i < n) faca
       A[i] = 0;
       i = i + 1;
   fim_enquanto
   para i = 1 ate n faca
       para j = 1 ate n faca
              A[i] = A[i] + (i*j);
     fim_para
   fim_para
fim
```

$f_1(n) = 1$

$f_2(n) = 5n$

$f_3(n) = 6n$

$f_4(n) = 2n \times f_3(n)$
$= 12n^2$

$f(n) = f_1(n)+f_2(n)+f_4(n)$
$= 12n^2 + 5n + 1$
$\approx n^2$

Exercícios

- Determine a função de custo do algoritmo abaixo, em função do tempo de execução, e sua classe de comportamento assintótico.

```
Inicio
   i, j, A[n]: inteiro
   i = 1;
   enquanto (i < n) faca
       A[i] = 0;
       i = i + 1;
   fim_enquanto
   para i = 1 ate n faca
       para j = 1 ate n faca
              A[i] = A[i] + (i*j);
     fim_para
   fim_para
fim
```

$f_1(n) = 1$

$f_2(n) = 5n$

$f_3(n) = 6n$

$f_4(n) = 2n \times f_3(n)$
$= 12n^2$

$f(n) = f_1(n)+f_2(n)+f_4(n)$
$= 12n^2 + 5n + 1$
$\approx n^2$

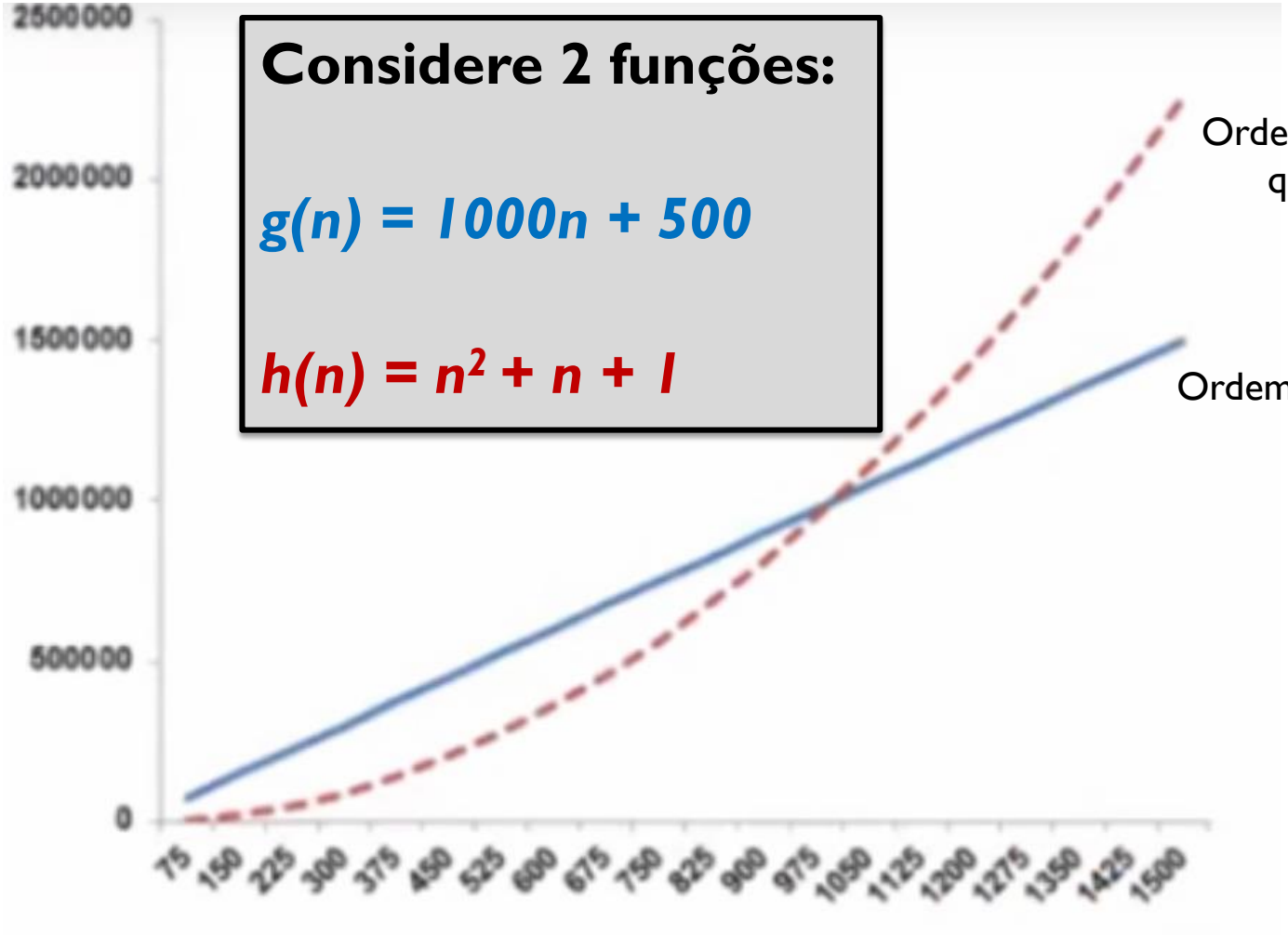
facilmente obtido pela **contagem dos laços aninhados**

Domínio assintótico

- **Revisão:**
  - O comportamento assintótico de *f(n)* representa o **limite de comportamento** da função de custo à medida que *n* cresce

- **Definição informal de domínio:**
  - Uma função *f(n)* **domina assintoticamente** outra função *g(n)* se a partir de um determinado valor de *n* (***n* ≥ *m***), temos ***g(n) ≤ f(n)***

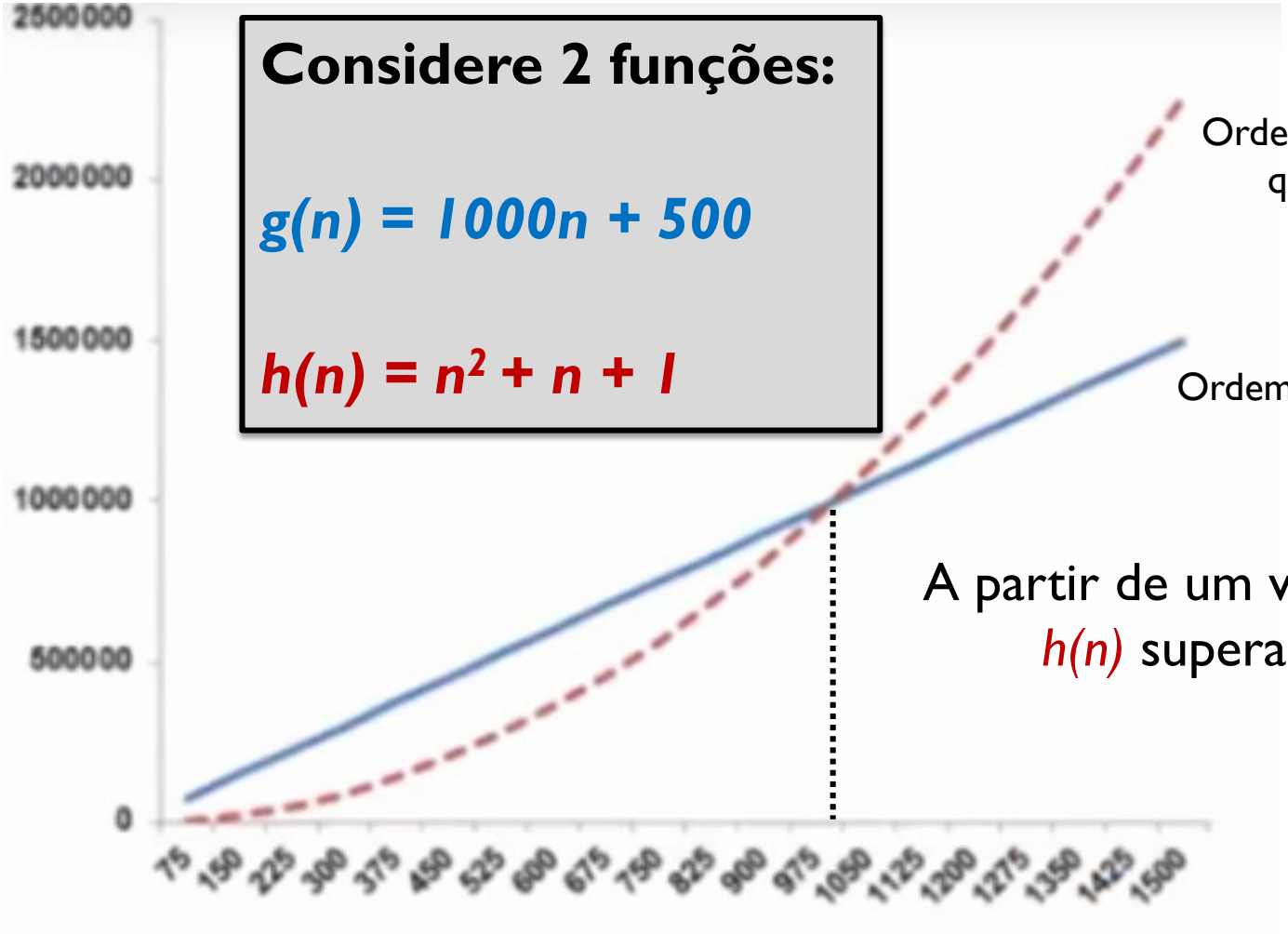
Exemplo



Ordem de crescimento quadrático (n^2)

Ordem de crescimento linear (n)

Exemplo



Considere 2 funções:

$g(n) = 1000n + 500$

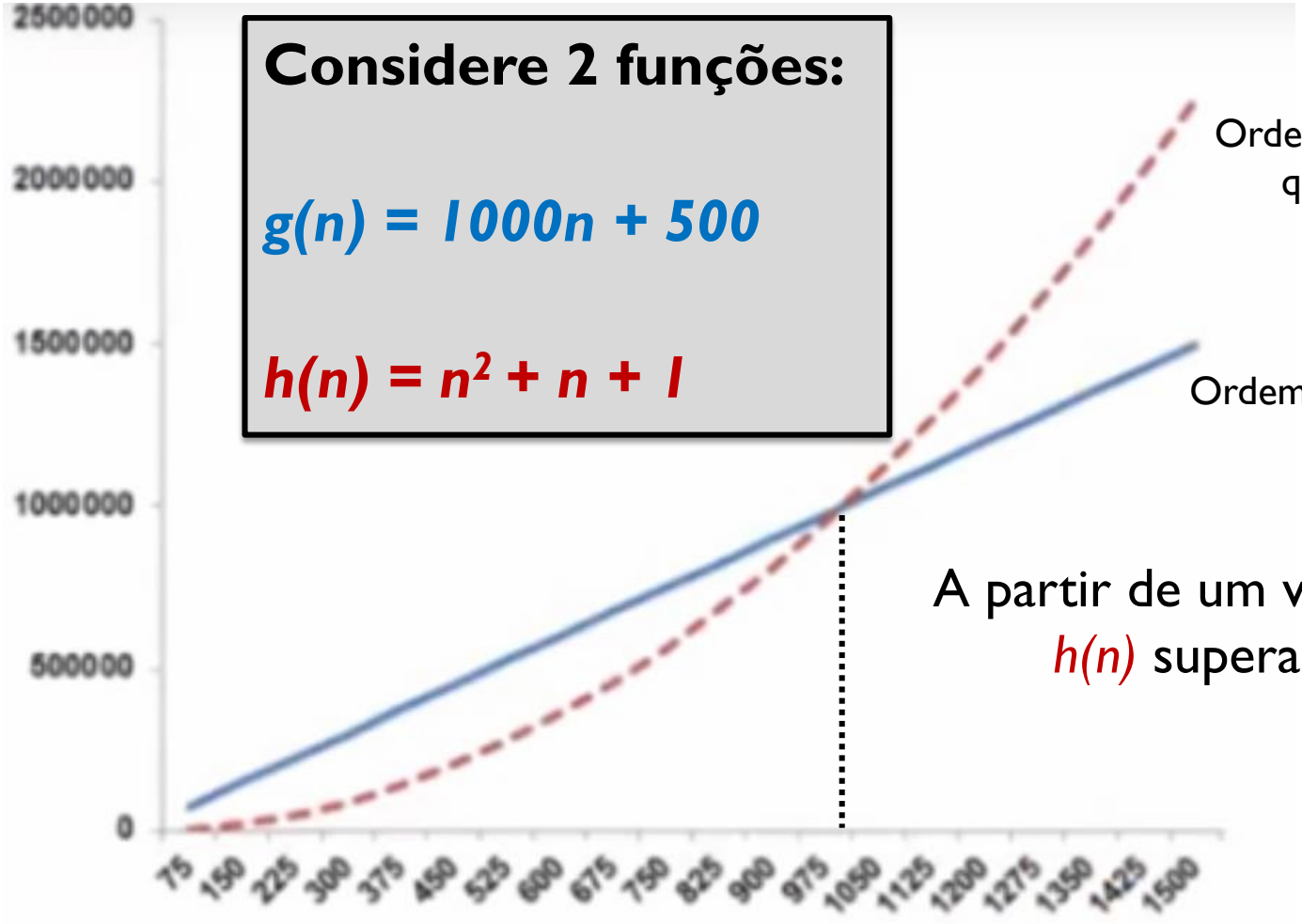
$h(n) = n^2 + n + 1$

Ordem de crescimento quadrático (n^2)

Ordem de crescimento linear (n)

A partir de um valor de n, $h(n)$ supera $g(n)$

Exemplo



Portanto, podemos dizer que *h(n)* domina **assintoticamente** *g(n)*

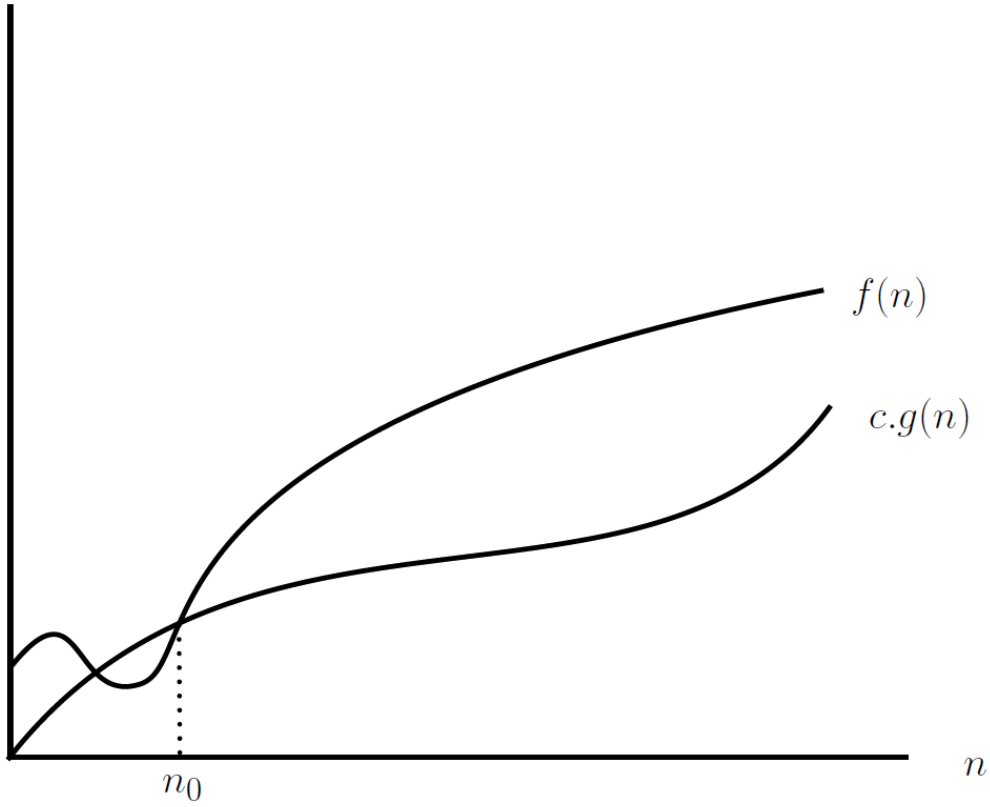
Notação Ω (big-ômega)

- Usada para denotar o custo do algoritmo no **melhor caso** possível
  - Indica o **limite assintótico inferior** para expressar algo que tenha "**pelo menos**" um dado comportamento

- **Definição:**
  - Para uma dada função ***g(n)***, denotamos ***Ω(g(n))*** como o conjunto de funções:

Ω(g(n)) **=** { ***f(n)*** se existem constantes positivas *c* e *m*, tal que: ***c.g(n)*** ***≥*** ***f(n)***, para todo ***n ≥ m*** }

Notação Ω (big-ômega)



$f(n) = \Omega(g(n))$

Para todos os valores de n à direita de n_0, o valor de $f(n)$ reside em $c \cdot g(n)$ ou acima desse.

Exemplo: $3n^2 + n = \Omega(n)$

- Podemos pensar nessa equação como sendo $3n^2 + n \geq \Omega(n)$
- Ou seja, a taxa de crescimento de $3n^2 + n$ é maior ou igual à taxa de n

Notação Ω (big-ômega)

- **Exemplo:**
  - $\sqrt{n} = \Omega(\log n)$

Notação Ω (big-ômega)

- **Exemplo:**
  - $\sqrt{n} = \Omega(\log n)$
- **Podemos ler:**

"raiz de *n* é **pelo menos** ômega de log *n*"

Notação Ω (big-ômega)

- **Exemplo:**
  - $\sqrt{n} = \Omega(\log n)$
- **Podemos ler:**

  "raiz de *n* é **pelo menos** ômega de log *n*"

- Isto vale para um ***n* suficientemente grande** ($n \geq n_0$)

Notação θ (theta)

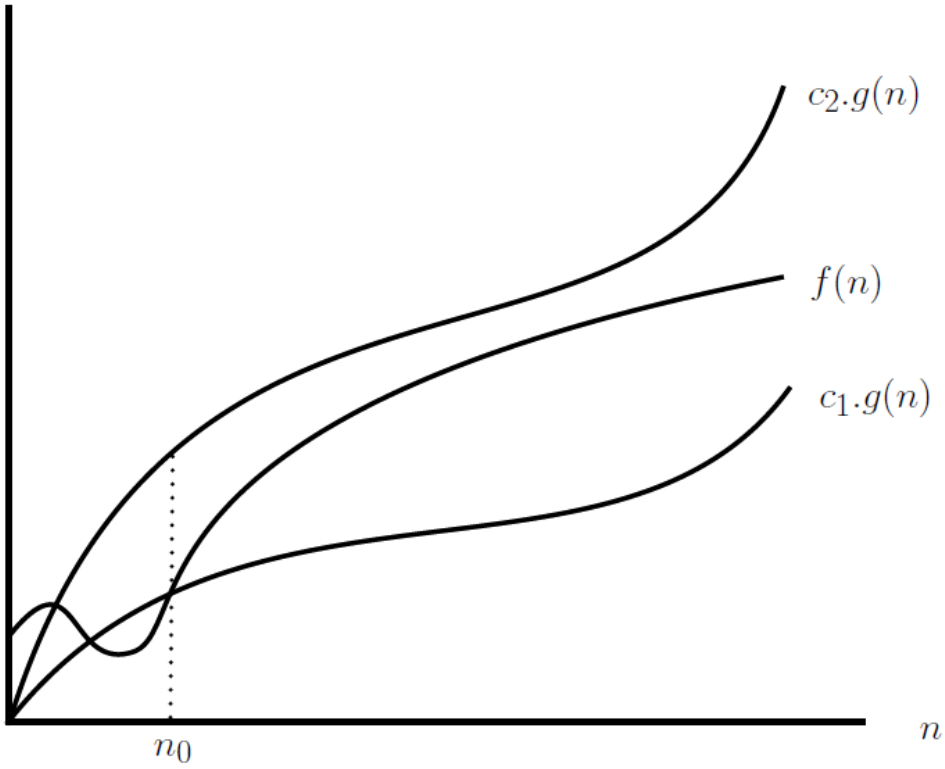
- Usada para denotar o **caso médio** da função custo de um algoritmo

- **Definição:**
  - Para uma dada função ***g(n)***, denotamos ***θ(g(n))*** como o conjunto de funções:

  θ(g(n)) **=** { ***f(n)*** se existem constantes positivas c_1, c_2 e m, tal que: $c_1.g(n) \leq f(n) \leq c_2.g(n)$, para todo $n \geq m$ }

- Basta multiplicar ***g(n)*** pelas constantes c_1 e c_2 para obter os limites superior e inferior para *f(n)*

Notação θ (theta)



$f(n) = \Theta(g(n))$

- Para todos os valores de n à direita de n_0, o valor de $f(n)$ reside em $c_1 g(n)$ ou acima dele e em $c_2 g(n)$ ou abaixo desse.
- Para todo $n > 0, f(n) = g(n)$ dentro de um fator constante.
- $g(n)$ é um **limite "estreito"** para $f(n)$.

Notação θ (theta)

- Foi dito que poderíamos descartar os termos de mais baixa ordem e coeficientes do termo de mais alta ordem. Para mostrar formalmente que, por exemplo,

$$\frac{1}{2}n^2 - 3n = \Theta(n^2)$$

  - Definiremos constantes positivas c_1, c_2 e n_0 tais que:

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2, \text{ para todo } n \geq n_0$$

Dividindo por n^2:

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

Notação θ (theta)

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

- A desigualdade do lado direito pode ser considerada válida para $n \geq 1$ escolhendo $c_2 \geq {}^1/_2$, e a do lado esquerdo pode ser considerada válida para $n \geq 7$ escolhendo $c_1 \geq {}^1/_{14}$.

- Para $c_2 = {}^1/_2$, $n = 7$ e $c_1 = {}^1/_{14}$, temos:

$$\frac{1}{2}n^2 - 3n = \Theta(n^2)$$

Notação θ (theta)

- Também é possível mostrar que $6n^3 \neq \Theta(n^2)$, para isso devemos encontrar:

$$c_1 n^2 \leq 6n^3 \leq c_2 n^2$$

- Dividindo os termos por n^2, temos:

$$c_1 \leq 6n \leq c_2$$

Notação θ (theta)

$$c_1 \leq 6n \leq c_2$$

- E assim chegamos a: $n \leq \frac{c_2}{6}$
  - mas c_2 é constante e n não poderia ser suficientemente grande
  - O valor de n está limitado pela constante $\frac{c_2}{6}$, não sendo possível a análise assintótica (entrada tendendo ao infinito)

- Assim, por contradição, provamos que $6n^3 \neq \Theta(n^2)$,

Notação O (big-oh)

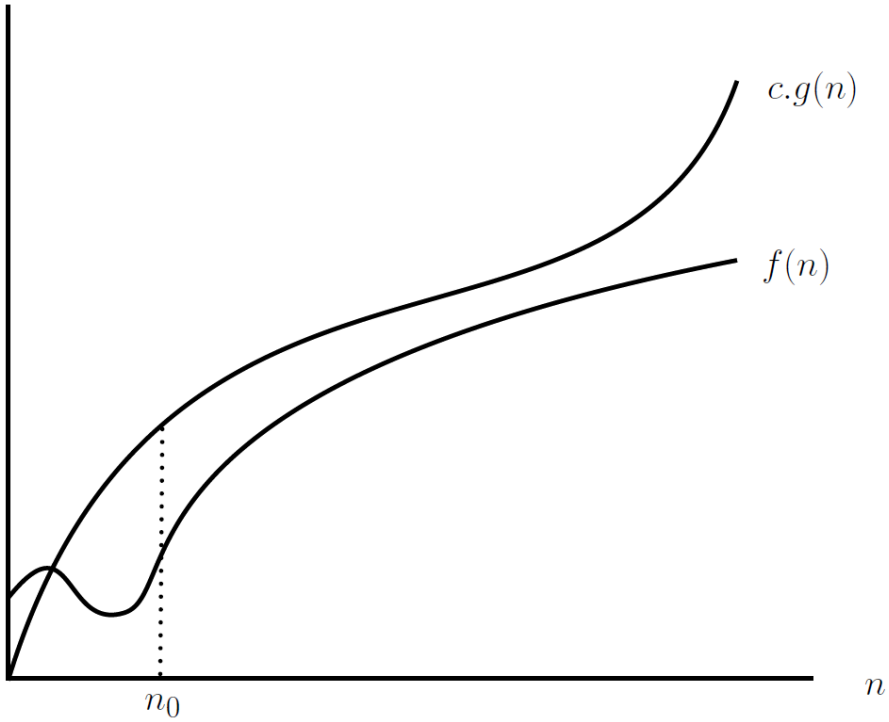
- Usada para denotar o custo do algoritmo no **pior caso** possível
  - Indica o **limite assintótico superior**

- É a notação mais conhecida e utilizada
  - Caso mais fácil de identificar
  - Não precisa estar próxima, nem ser um limite estreito para *f(n)*

- **Definição:**
  - Para uma dada função ***g(n)***, denotamos ***O(g(n))*** como o conjunto de funções:

O(g(n)) = { ***f(n)*** se existem constantes positivas *c* e *m*, tal que: ***f(n)*** ≤ ***c.g(n)***, para todo ***n*** ≥ ***m*** }

Notação O (big-oh)



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

- Para todos os valores de n à direita de n_0, o valor de $f(n)$ reside em $c \cdot g(n)$ ou abaixo desse.
- Formalmente, a função $g(n)$ representa um **limite superior** para $f(n)$
- Exemplo: $2n^2 = O(n^3)$
  - Podemos pensar nessa equação como sendo $2n^2 \leq O(n^3)$ ou $2n^2 \in O(n^3)$
  - A taxa de crescimento de $2n^2$ é menor ou igual à taxa de n^3

Notação O (big-oh)

- Exemplo 1: $2n + 10$ é $O(n)$

  - Podemos realizar uma manipulação para encontrar c e n_0:

$$2n + 10 \leq c \cdot n$$

$$c \cdot n - 2n \geq 10$$

$$(c - 2)n \geq 10$$

$$n \geq \frac{10}{c - 2}$$

  - A afirmação é válida para $c = 3$ e $n_0 = 10$.

Notação O (big-oh)

- Exemplo 2: n^2 é $O(n)$
  - É preciso encontrar c que seja sempre maior ou igual a n para todo valor de um n_0:

$$n^2 \leq c \cdot n \Rightarrow$$

$$n \leq c$$

- É impossível pois c deve ser constante.

Notação O (big-oh)

- Exemplo 3:

$$3n^3 + 20n^2 + 5 \text{ é } O(n^3)$$

  - É preciso encontrar $c > 0$ e $n_0 \geq 1$ tais que $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ para $n \geq n_0$

  - Como $3n^3 + 20n^2 + 5 \leq (3 + 20 + 5) \cdot n^3$, podemos tomar $c = 28$ e qualquer $n_0 > 1$

Notação O (big-oh)

- Exemplo 4:

$$3 \log n + 5 \text{ é } O(\log\ n)$$

- É preciso encontrar $c > 0$ e $n_0 \geq 1$ tais que
$3 \log +5 \leq c \cdot \log n$ para todo $n \geq n_0$

- Note que $3 \log n + 5 \leq (3 + 5) \cdot \log n$ se $n > 1$ $(\log 1 = 0)$

- Basta tomar, por exemplo, $c = 8$ e qualquer $n_0 = 2$

Notação O (big-oh)

- Exemplo 5:

$$2^{n+2} \text{ é } O(2^n)$$

  - É preciso $c > 0$ e $n_0 \geq 1$ tais que $2^{n+2} \leq c \cdot 2^n$ para todo $n \geq n_0$
  - Note que $2^{n+2} = 2^n * 2^2 = 4 \cdot 2^n$
  - Assim, basta tomar, por exemplo, $c = 4$ e qualquer n_0

Operações com a notação O (big-Oh)

- Se $T1(n) = O(\mathbf{f(n)})$ e $T2(n) = O(\mathbf{g(n)})$, então:
  - $c \times O(f(n)) = O(f(n))$
  - $O(O(f(n)) = O(f(n))$
  - $O(f(n)) + O(f(n)) = O(f(n))$
  - $O(f(n)) + O(g(n)) = O(max(f(n),g(n)))$

códigos sequenciais

  - $O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$
  - $f(n) \times O(g(n)) = O(f(n) \times g(n))$

códigos aninhados

Notação assintótica

- ...Algumas regras
  - Se $T(x)$ é um polinômio de grau n, então: $T(x) = \Theta(x^n)$
- Relembrando
  - um polinômio de grau *n* é uma função na forma:
    - $f(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \cdots + a_0 \cdot x + a_0$
  - classificação em função do grau
    - 0: constante
    - 1: linear
    - 2:quadrático
    - 3:cúbico

Exercício

- Um algoritmo tradicional e muito utilizado possui complexidade $n^{1,5}$, enquanto um algoritmo novo proposto é da ordem de $n \log n$:
  - $\mathrm{f}(n) = n^{1,5}$
  - $\mathrm{g}(n) = n \log n$
- Qual algoritmo adotar?

Exercício

- Um algoritmo tradicional e muito utilizado possui complexidade $n^{1,5}$, enquanto um algoritmo novo proposto é da ordem de $n \log n$:
  - $\mathrm{f}(n) = n^{1,5}$
  - $\mathrm{g}(n) = n \log n$
- Qual algoritmo adotar?
- **Uma possível solução:**
  - $f(n) = \frac{n^{1,5}}{n} = n^{0,5} \Rightarrow (n^{0,5})^2 = n$
  - $g(n) = \frac{n \log n}{n} = \log n \Rightarrow (\log n)^2 = log^2 n$
- Como n cresce mais rapidamente do que qualquer potência de $\log$, o algoritmo novo é mais eficiente.

Dicas de análise na prática

- Se $f(n)$ for um polinômio de grau d então $f(n)$ é $O(n^d)$
  - despreze os termos de menor ordem
  - despreze os fatores constantes

- Use a menor classe de funções possível
  - $2n$ é $O(n)$, ao invés de $O(2n)$

- Use a expressão mais simples
  - $3n + 5$ é $O(n)$, ao invés de $O(3n + 5)$

Dicas de análise na prática

- **Repetições:** o tempo de execução é pelo menos o tempo dos comandos dentro da repetição multiplicada pelo número de vezes que é executada.
  - o exemplo abaixo é *O*(n)

```
para i de 1 ate n faca
    a = a*i
```

Dicas de análise na prática

- **Repetições:** o tempo de execução é pelo menos o tempo dos comandos dentro da repetição multiplicada pelo número de vezes que é executada.
  - o exemplo abaixo é $O(n)$

```
para i de 1 ate n faca
    a = a*i
```

- **Repetições aninhadas:** análise feita de dentro para fora
  - O tempo total é o tempo de execução dos comandos multiplicado pelo produto do tamanho de todas as repetições.
  - o exemplo abaixo é $O(n^2)$

```
para i de 1 ate n faca
    para j de 0 ate n-1 faca
        a = a*(i+j)
```

Dicas de análise na prática

- **Condições:** o tempo nunca é maior do que o tempo do teste mais o tempo do maior entre os comandos dentro do bloco do "então" e do "senão"
  - o exemplo abaixo é $O(n)$

```
se (a < b) entao
        a = a + 1
senao
        para i de 1 ate n-1 faca
                a = a*i
```

Dicas de análise na prática

- **Condições:** o tempo nunca é maior do que o tempo do teste mais o tempo do maior entre os comandos dentro do bloco do "então" e do "senão"
  - o exemplo abaixo é $O(n)$

```
se (a < b) entao
        a = a + 1
senao
        para i de 1 ate n-1 faca
                a = a*i
```

- **Chamadas à sub-rotinas:**
  - Analisa a eficiência da sub-rotina
  - Incorpora seu custo ao programa que a chamou

Considerações finais

- Quando a análise assintótica não é indicada:
  - Interesse em **entradas relativamente pequenas**
  - Para funções com **fatores constantes com relevância prática**

Considerações finais

- Quando a análise assintótica não é indicada:
  - Interesse em **entradas relativamente pequenas**
  - Para funções com **fatores constantes com relevância prática**

- **Ex:** considere a função custo de dois algoritmos:
  - $f(n) = 10^{100}n$
  - $g(n) = 10n \log n$

Considerações finais

- Quando a análise assintótica não é indicada:
  - Interesse em **entradas relativamente pequenas**
  - Para funções com **fatores constantes com relevância prática**

- **Ex:** considere a função custo de dois algoritmos:
  - $f(n) = 10^{100}n$
  - $g(n) = 10n \log n$

- **Pela análise assintótica: O(*f(n)*) ≤ O(*g(n)*)**

  - **Nota:** 10^{100} é o número estimado como o limite superior para a quantidade de átomos no universo observável

$$10n \log n > 10^{100}n \quad \text{apenas para} \quad \boldsymbol{n > 2^{10^{99}}}$$

Exercícios

1. Descreva a notação O (apresentando a classe e as constantes *c* e *m*) para seguintes funções de custo:

   $f_1(n) = 5n^{5/2} + n^{2/5}$

   $f_2(n) = 6 \log 2n + 9n$

   $f_3(n) = 3n4 + 2n \log^2 n$

1. Suponha que o algoritmo presente na parte <...> tenha complexidade *5n+2*. Calcule a complexidade de tempo de execução do trecho de algoritmo apresentado a seguir:

```
k = 1;
enquanto K <= n faça
   <...>
   k = K+1;
fim_enquanto
```

Exercícios

3. Suponha que a complexidade de um algoritmo qualquer seja $5n^2 + n$. Considerando que uma iteração nesse algoritmo leva algo em torno de 1 nanossegundo (10^{-9} s), quanto tempo ele levará para processar uma entrada com 1000 elementos?

4. Considere 3 algoritmos para um mesmo problema, os quais foram testados usando 2 conjuntos de entrada (n = 10 e n = 100), como apresentado na tabela. Determine a complexidade assintótica de cada algoritmo e indique qual é o melhor e o pior. Também indique qual é o menor tamanho do conjunto de entrada necessário para o melhor algoritmo obter o menor tempo de execução.

Cj. de entrada	Algoritmo 1	Algoritmo 2	Algoritmo 3
n = 10	1	1/100	1/1000
n = 100	10	1	1

Bibliografia

- Slides adaptados do material do Prof. Dr. Bruno Travençolo, da Profa. Dra. Denise Guliato e do Prof. Dr. Ricardo Campello

- BACKES, A. Linguagem C Descomplicada: portal de vídeo-aulas para estudo de programação. Disponível em: https://programacaodescomplicada.wordpress.com/indice/estrutura-de-dados/

- CORMEN, T.H. et al. Algoritmos: Teoria e Prática, Campus, 2002

- ZIVIANI, N. Projeto de algoritmos: com implementações em Pascal e C (2ª ed.), Thomson, 2004

- MORAES, C.R. Estruturas de Dados e Algoritmos: uma abordagem didática (2ª ed.), Futura, 2003

Bibliografia

- FEOFILOFF, P. **Minicurso de Análise de Algoritmos**, 2010. Disponível em: http://www.ime.usp.br/~pf/livrinho-AA/

- DOWNEY, A. B**. Analysis of algorithms** (Cap. 2), Em: Computational Modeling and Complexity Science. Disponível em: http://www.greenteapress.com/compmod/html/book003.html

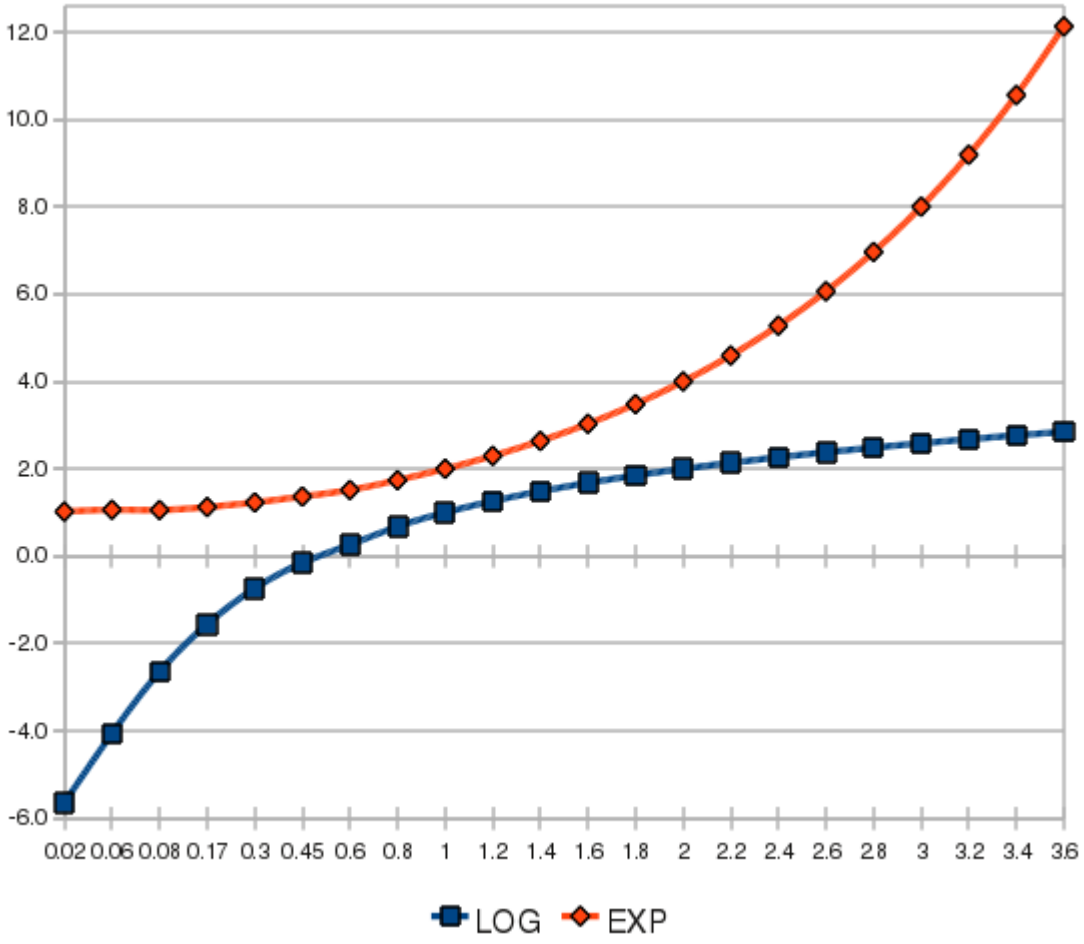
- ROSA, J. L. **Notas de Aula de Introdução a Ciência de Computação II**. Universidade de São Paulo. Disponível em: http://coteia.icmc.usp.br/mostra.php?ident=639

SLIDES EXTRAS

Revisão de matemática

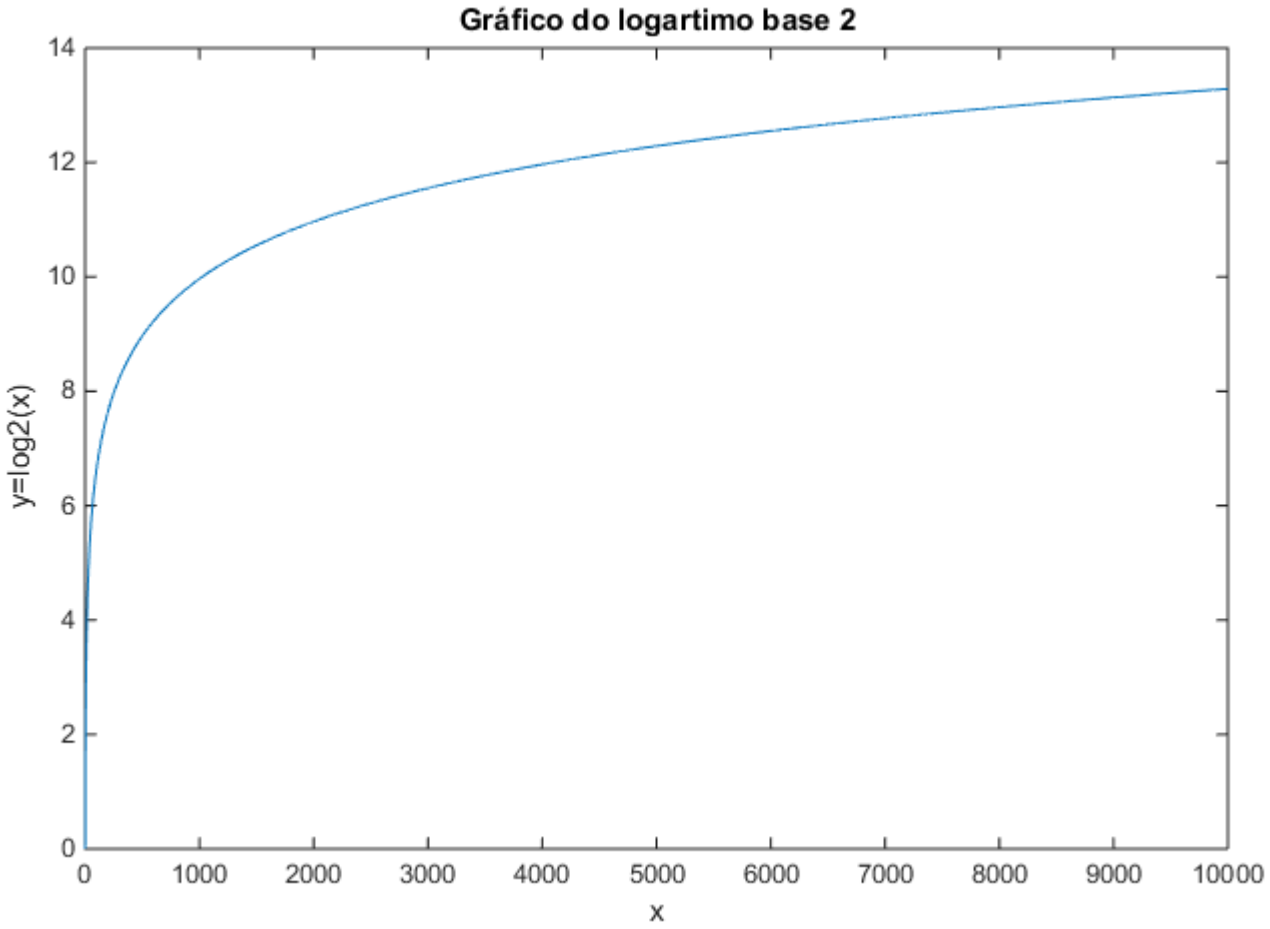
- Expoentes
  - $x^a\ x^b = x^{a+b}$
  - $x^a / x^b = x^{a-b}$
  - $(x^a)^b = x^{ab}$
  - $x^n + x^n = 2x^n$ (e não x^{2n})
  - $2^n + 2^n = 2^{n+1}$
- Logaritmos (por padrão, base 2)
  - $(x^a = b) \rightarrow (\log_x b = a)$
  - $\log_a b = \log_c b \,/\, \log_c a$ para $c > 0$
  - $\log ab = \log a + \log b$
  - $\log a/b = \log a - \log b$
  - $\log(a^b) = \mathrm{b} \log a$
  - $\log x < x$ para todo $x > 0$
  - Notação: $\lg n = \log_2 n$ (logaritmo binário)

Função logarítmica X exponencial



Exponencial: $y = 2^x$ e **Logarítmica:** $y = log_2\ x$

Sobre os logaritmos



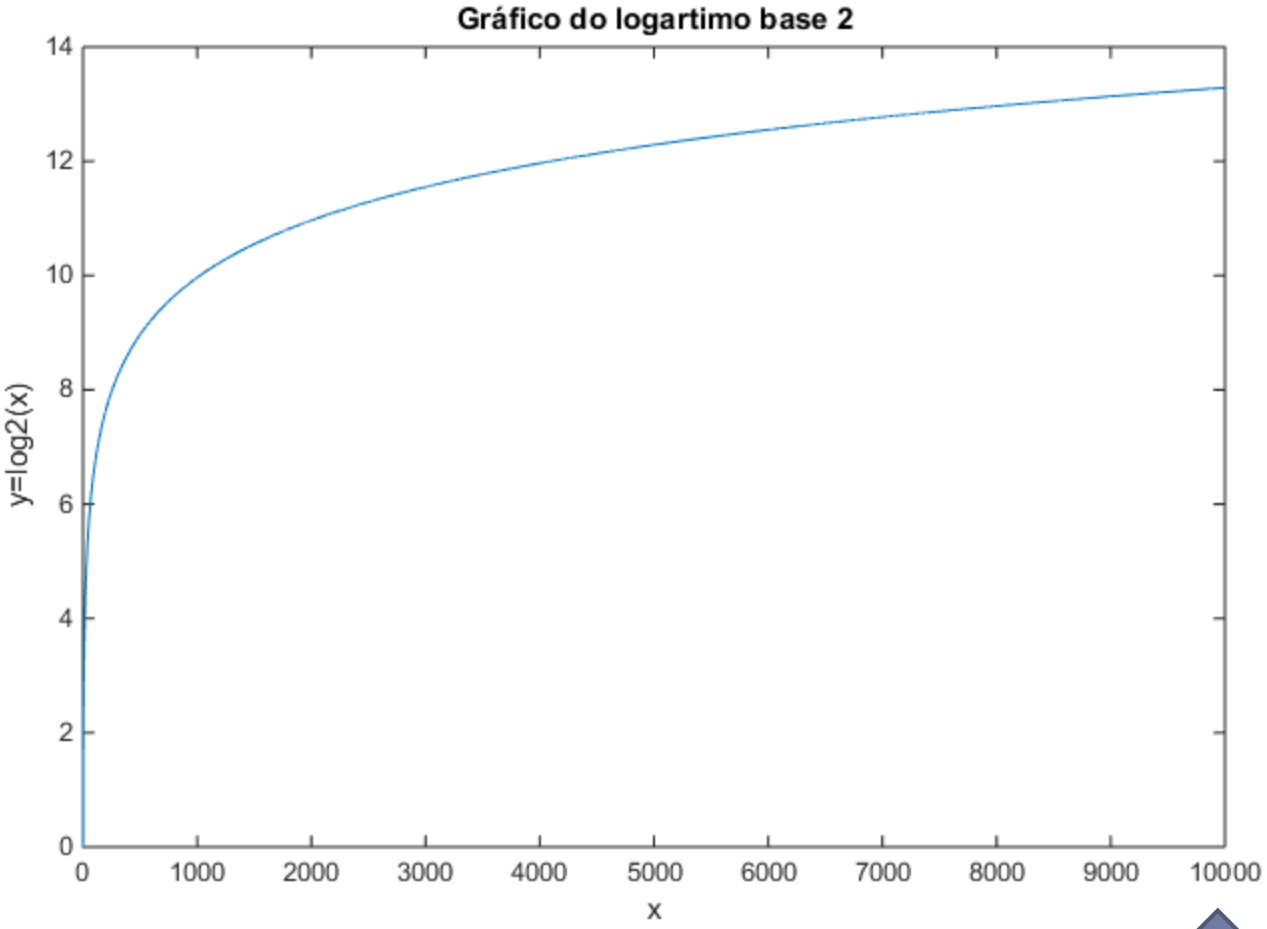
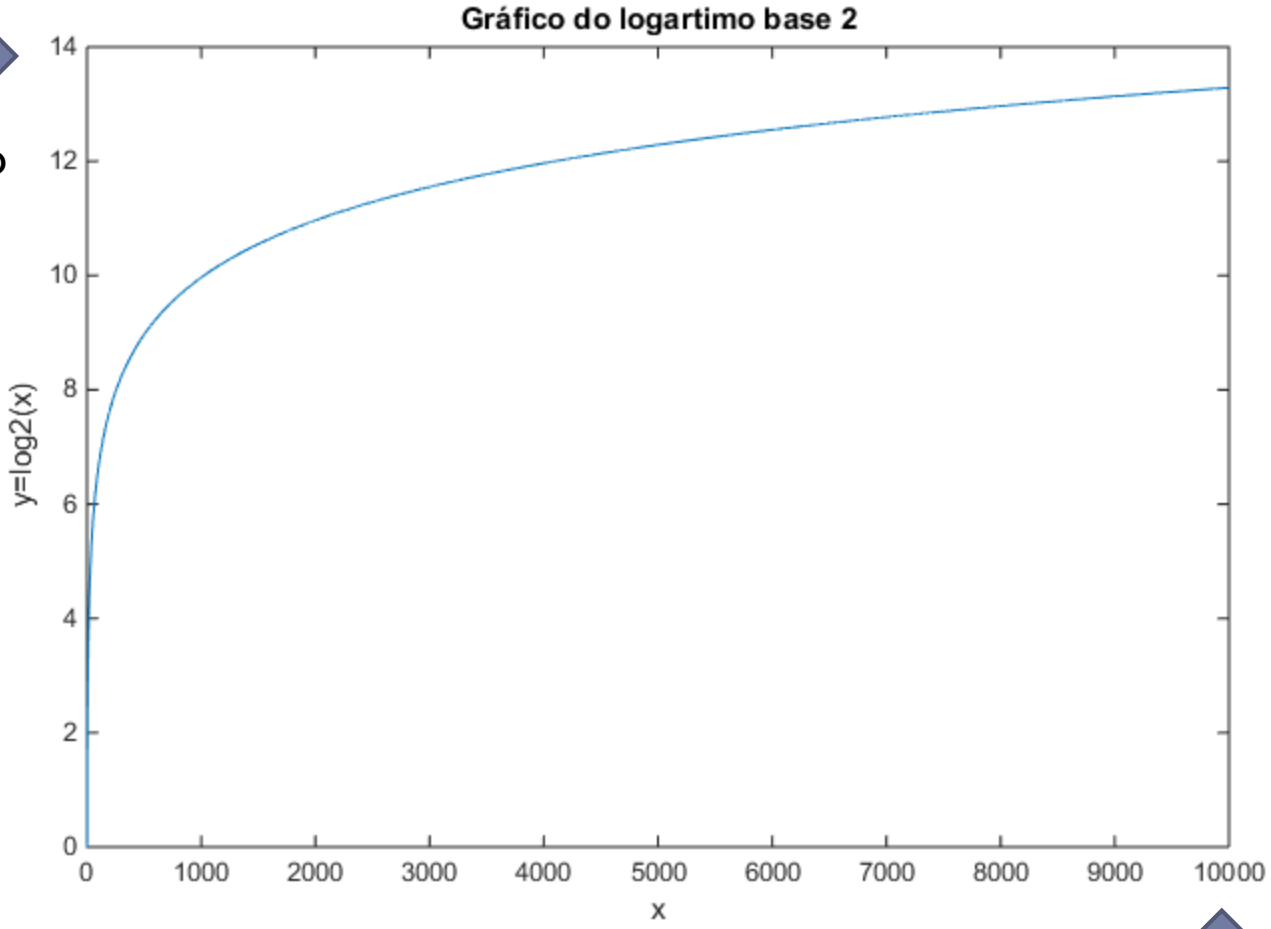


Gráfico do logartimo base 2
y=log2(x)
x
0
2
4
6
8
10
12
14
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

Valor alto

Valor baixo



Valor alto

Logaritmo

- O logaritmo nos fornece um número que é o expoente de um outro número
  - Seu valor é baixo e cresce bem lentamente

Logaritmo

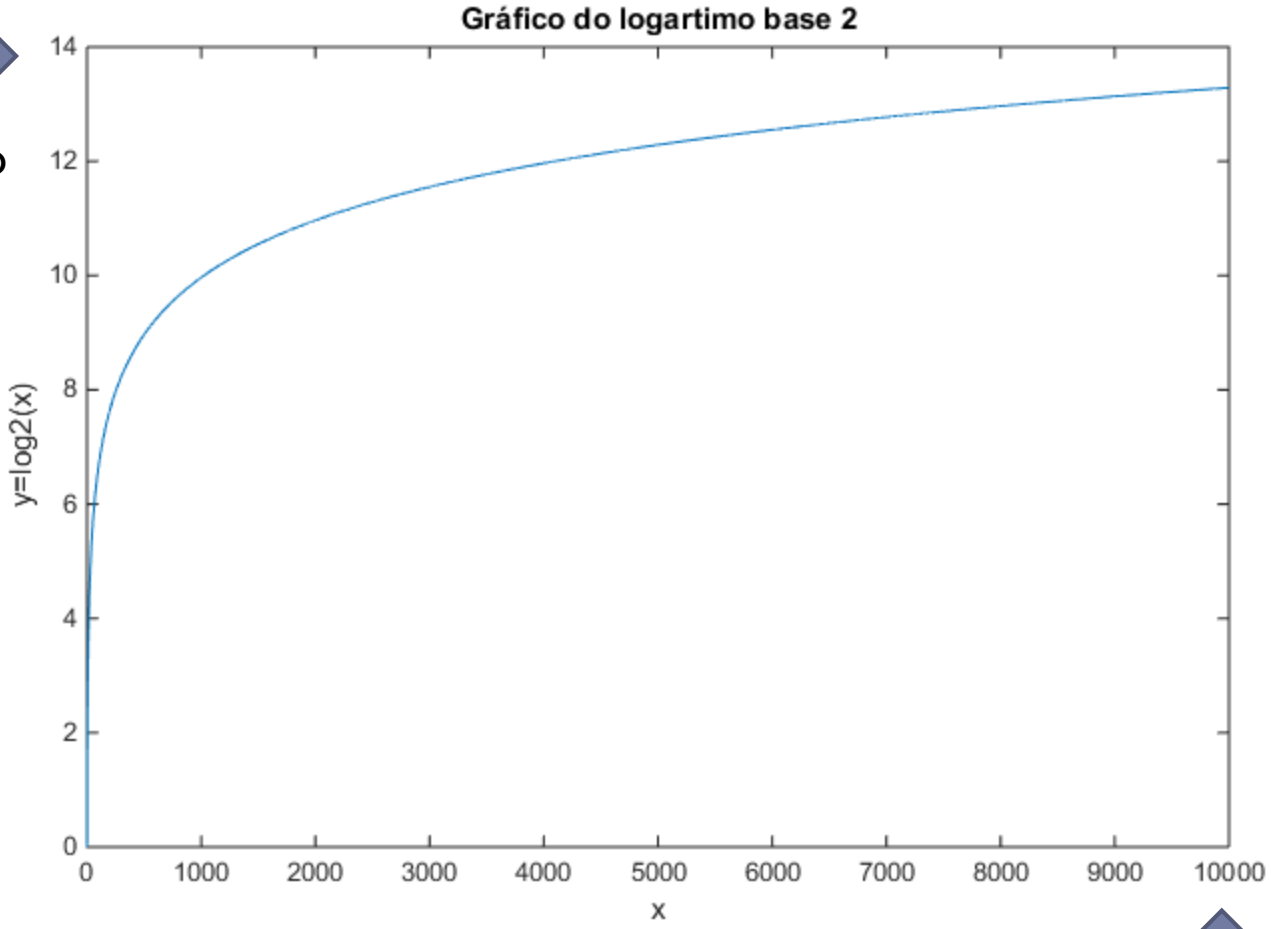
- O logaritmo nos fornece um número que é o expoente de um outro número
  - Seu valor é baixo e cresce bem lentamente
  - Qual é esse outro número??

Logaritmo

- O logaritmo nos fornece um número que é o expoente de um outro número
  - Seu valor é baixo e cresce bem lentamente
  - Qual é esse outro número??

A base do logaritmo

Valor baixo

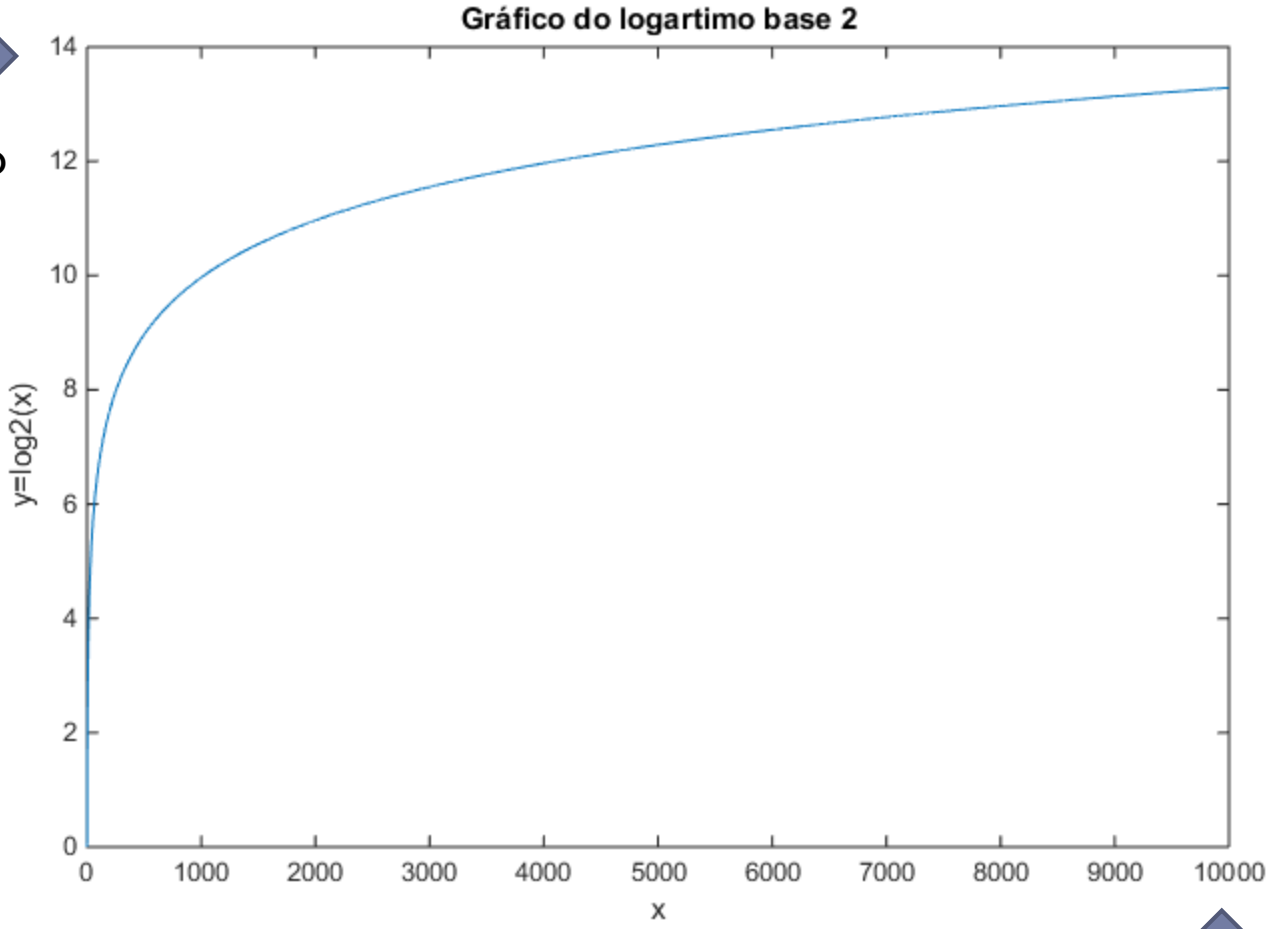


$$2^y = x \Leftrightarrow y = \log_2 x$$

A base é 2

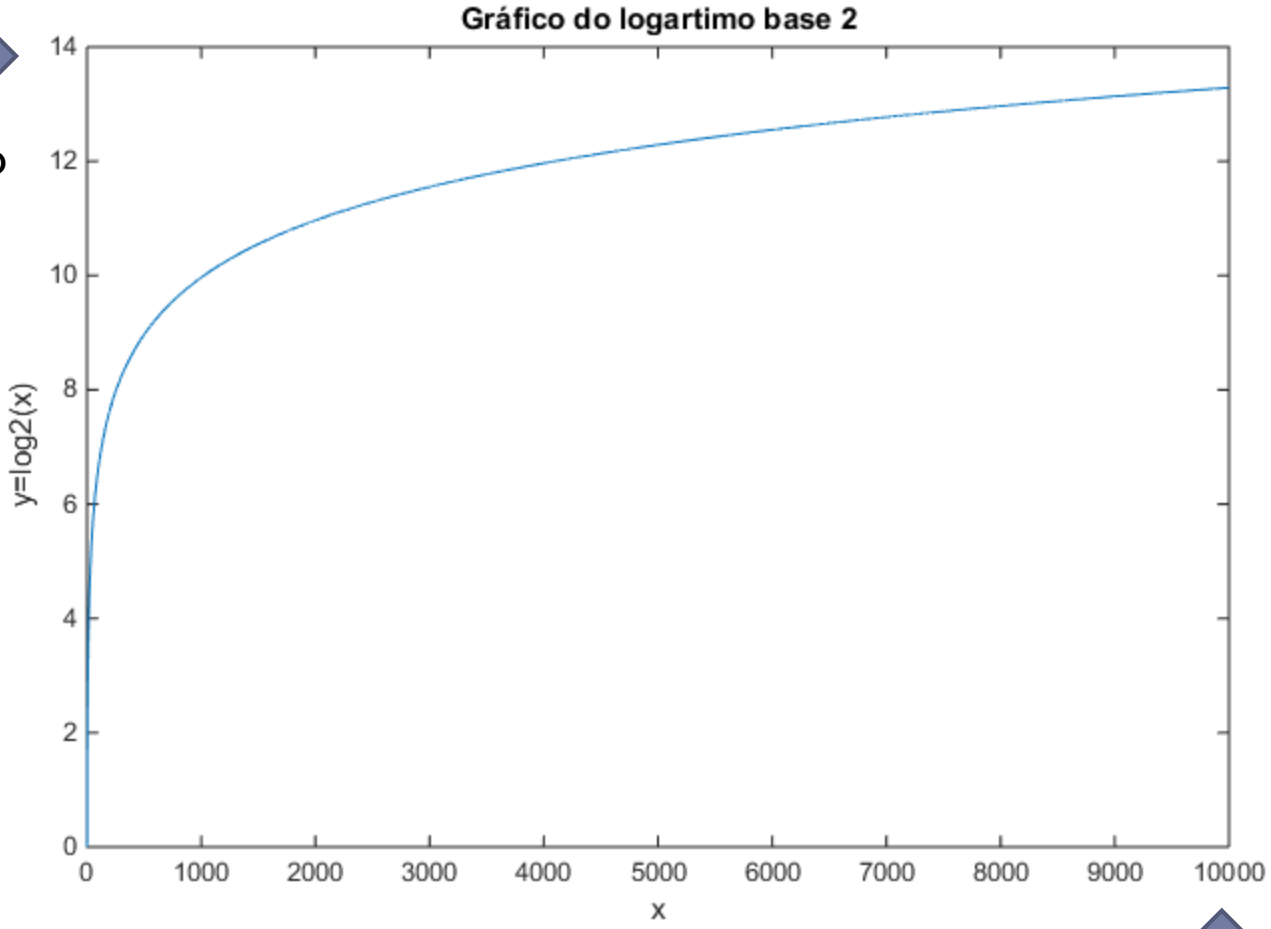
Valor alto

Valor baixo



$$2^y = \boxed{x}$$

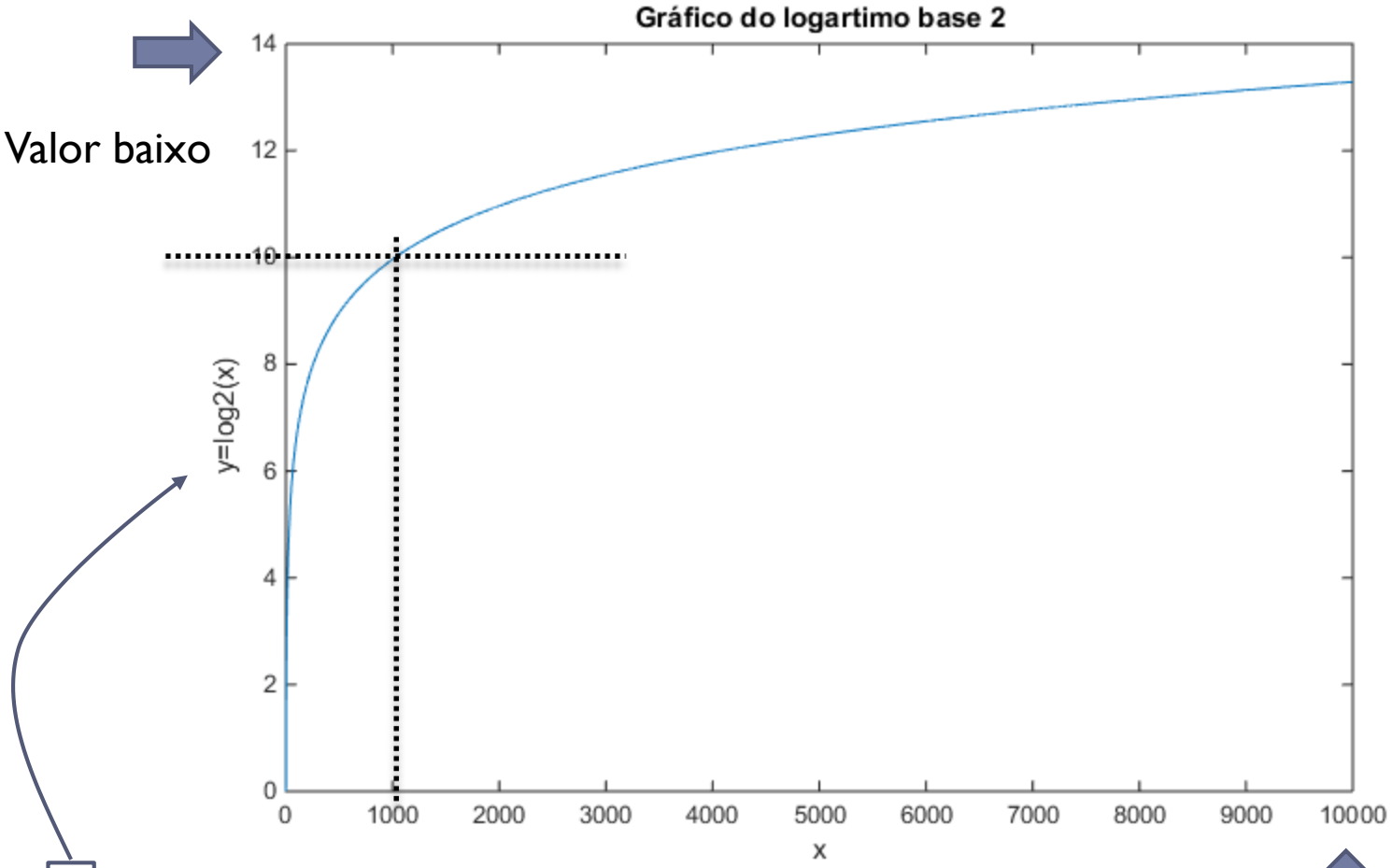
Valor alto



Valor baixo

$$2^{y} = x$$

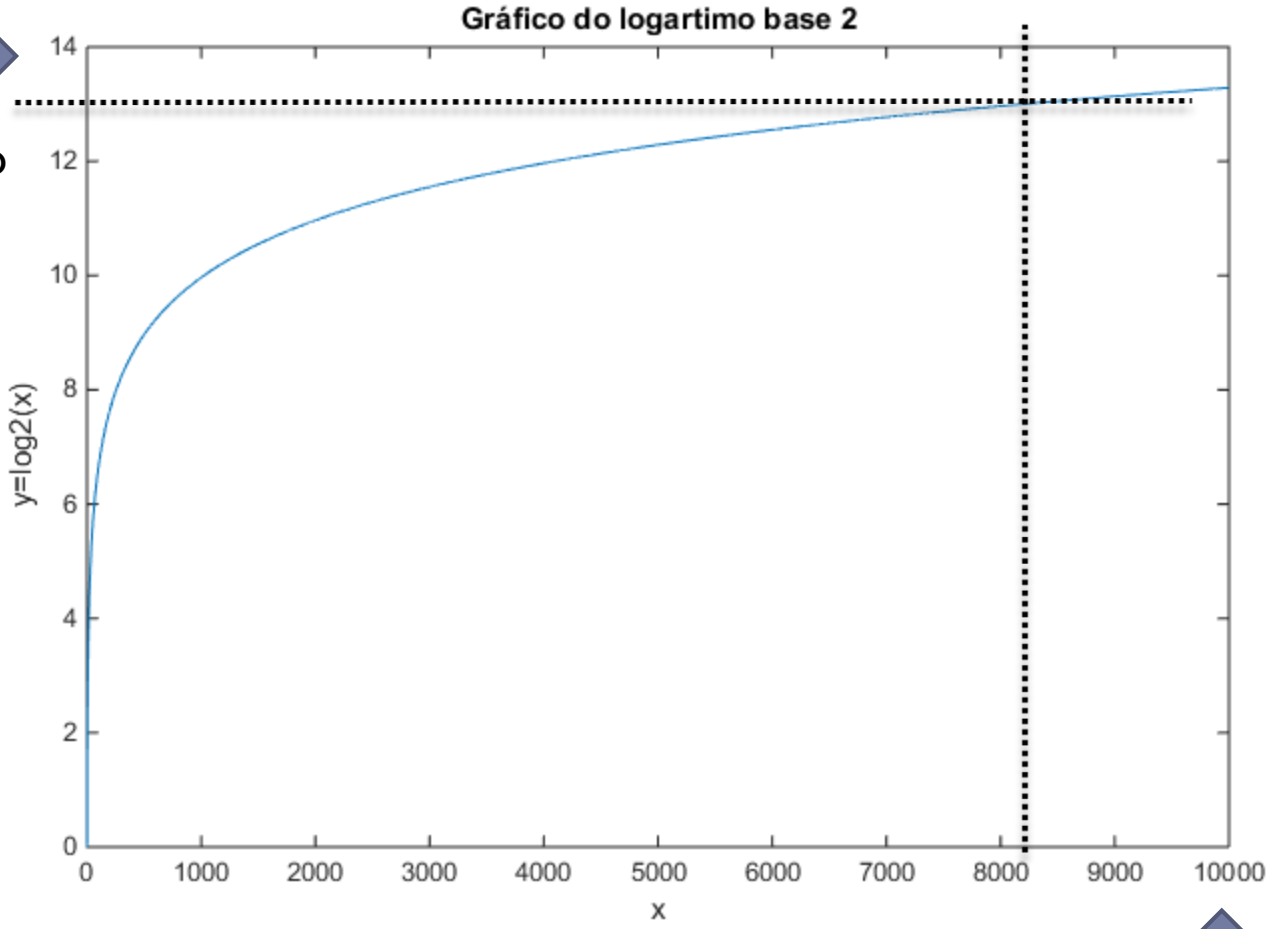
Valor alto



Valor baixo

$$2^{y} = x$$

Valor alto



Valor baixo

$2^{y} = x$

Valor alto

Uso do logaritmo

- Dado um valor, usamos *log* quando queremos descobrir qual o expoente que resulta neste número

Uso do logaritmo

- Dado um valor, usamos *log* quando queremos descobrir qual o expoente que resulta neste número
  - **Ex:** Qual é o expoente de 2 que resulta em 1538?

$$2^y = 1538 \rightarrow y = ?$$

Uso do logaritmo

- Dado um valor, usamos *log* quando queremos descobrir qual o expoente que resulta neste número
  - **Ex:** Qual é o expoente de 2 que resulta em 1538?

$$2^y = 1538 \rightarrow \; y = ?$$

  - Sabemos que $2^{10} = 1024$ e $2^{11} = 2048$, logo: $10 < y < 11$

Uso do logaritmo

- Dado um valor, usamos *log* quando queremos descobrir qual o expoente que resulta neste número
  - **Ex:** Qual é o expoente de 2 que resulta em 1538?

$$2^y = 1538 \rightarrow y = ?$$

  - Sabemos que $2^{10} = 1024$ e $2^{11} = 2048$, logo: $10 < y < 11$
  - Usando logaritmo, temos:

$$y = \log_2 1538 = 10{,}5868$$

ou seja

$$2^{10{,}5868} = 1538$$