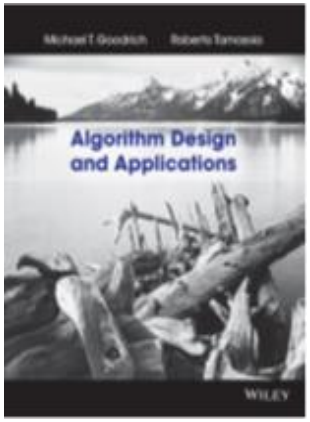
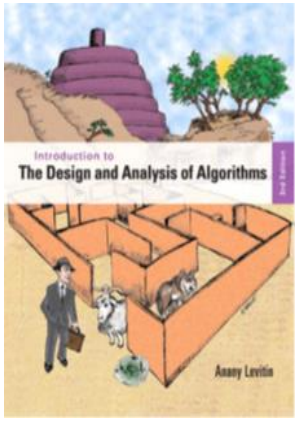
Unidade 1 - Introdução à Análise de Algoritmos
Modelo de Knuth

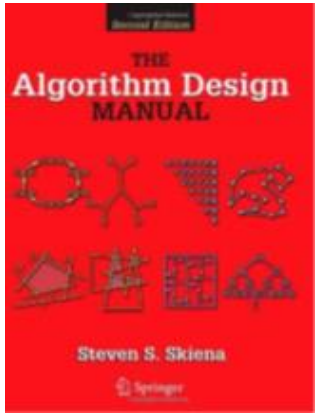
Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP
aparecidovfreitas@gmail.com

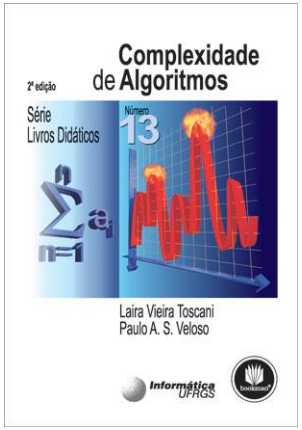
Bibliografia

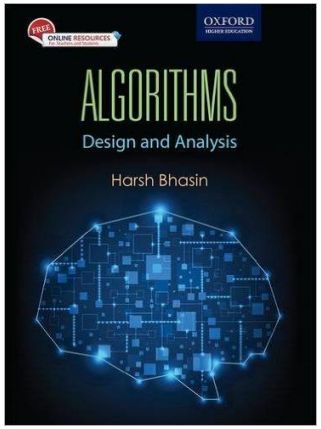
- Algorithm Design and Applications – Michael T. Goodrich, Roberto Tamassia, Wiley, 2015
- Introduction to the Design and Analysis of Algorithms – Anany Levitin, Pearson, 2012
- The Algorithm Design Manual – Steven S. Skiena, Springer, 2008
- Complexidade de Algoritmos – Série Livros Didáticos – UFRGS
- Algorithms – Design and Analysis – Harsh Bhasin – Oxford University Press - 2015

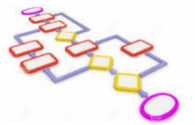








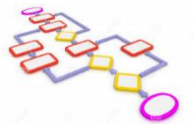




Em primeiro lugar...

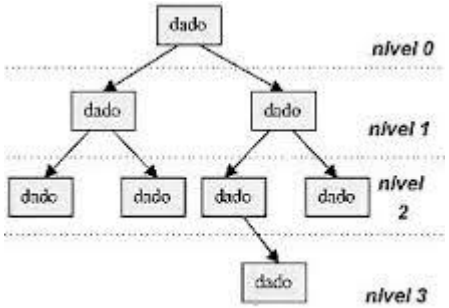
O que é Algoritmo?

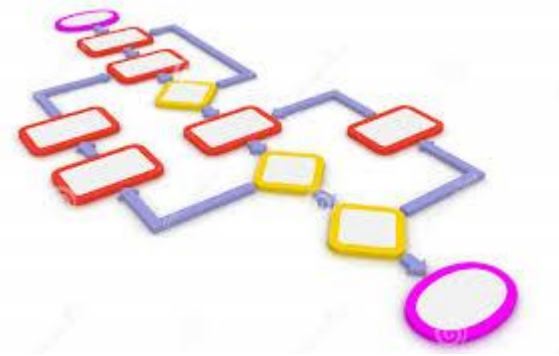
O que são Estruturas de Dados?

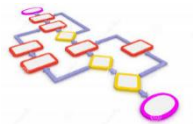


Algoritmo

- Procedimento **passo-a-passo** para a execução de alguma tarefa em uma quantidade **finita** de tempo;
- Uma estrutura de dados é uma forma organizada de acessar **dados**;
- Esses conceitos são **fundamentais** para a Computação.







Análise de Algoritmos

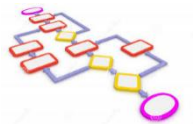
- Dado um algoritmo, podemos executá-lo em uma dada máquina para um determinado conjunto de dados;
- Porém, tal conhecimento é restrito e válido apenas para aquela situação;
- Para avaliarmos o comportamento do algoritmo precisamos analisá-lo nos **casos gerais** (para várias **instâncias**)…

Análise de Algoritmos

- Em geral, o programador ao terminar o teste de um algoritmo em um programa fica feliz pois o programa pode ter executado **bem** e com bom **desempenho**!

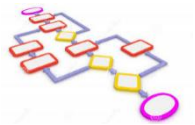
Mas ...

- Será que o comportamento do programa será satisfatório **para outras instâncias do problema**?



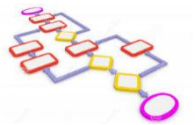
Eficiência de um Algoritmo

- Apesar de haver várias questões importantes para se analizar em um algoritmo, em geral interessa-se mais pelo seu **desempenho**.
- Isto particularmente se aplica à problemas que tem **alta** complexidade computacional.



O que se pode analisar?

- Desempenho
- Espaço ocupado de memória
- Comprimento total do código
- Corretismo
- Legibilidade
- Robustez



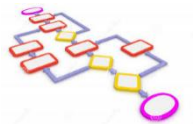
Como se avalia o desempenho de um algoritmo?

Medição de Algoritmos

Será que a medição direta é viável?

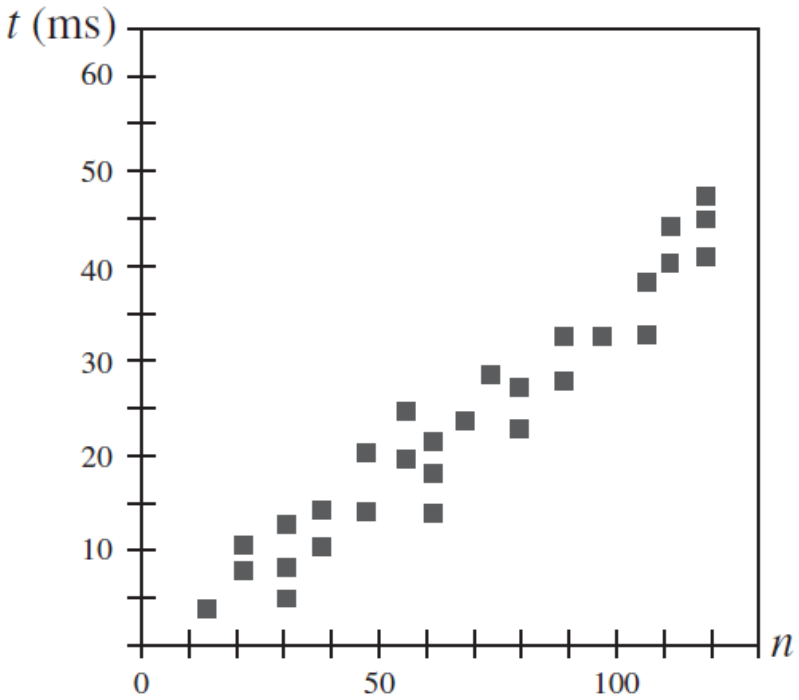
Medição de Algoritmos

- Dependência do compilador;
- Dependência de Hardware;
- Dependência do Sistema Operacional;
- Quantidade de memória disponível;
- Espaço disponível em disco.

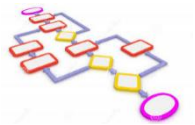


Desempenho de Algoritmos

- Em geral, o tempo de execução de um algoritmo **aumenta** com o tamanho da entrada (**instância**);



Fonte: Algorithm, Design and Applications, Tamassia, Goodrich, 2015



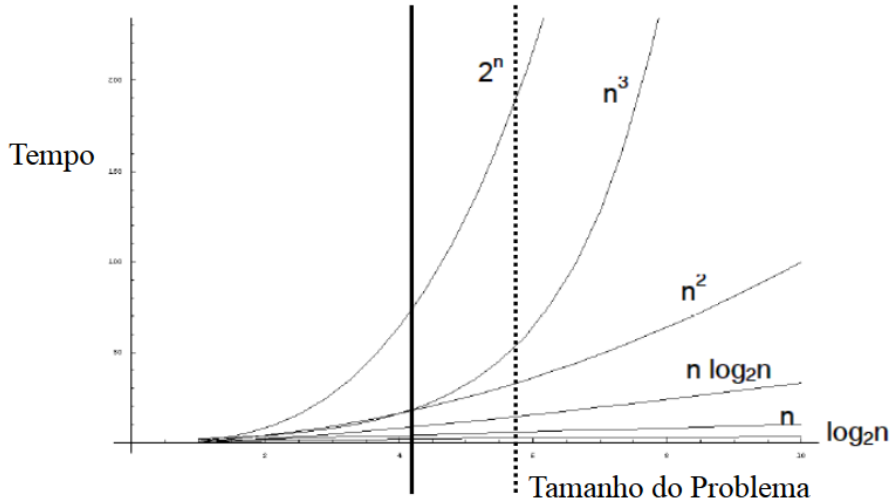
Experimentar (executar) algoritmos e obter os tempos de execução são úteis, mas pode haver limitações?

Limitações do método experimental

- Experimentos podem ser feitos em um **conjunto limitado** de entradas e podem não ser **representativos**;
- É difícil comparar-se algoritmos sem que os **ambientes** de **software** e **hardware** sejam idênticos;
- É necessário implementar e **executar** um algoritmo para se obter o tempo de execução de forma experimental.

Método Analítico

- Embora o método experimental tem um importante papel em análise de algoritmos, quando tratado de forma isolada **não** é suficiente;
- É necessário um método analítico que:
  - ✓ Considere **todas** as entradas possíveis;
  - ✓ Seja **independente** de ambientes de hardware e software;
  - ✓ Seja obtido **sem** a execução do algoritmo.



Modelo de Knuth

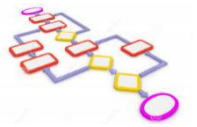
- Modelo matemático, desenvolvido por **Donald Knuth**, Stanford University, 1968;
- Baseia-se na **contabilização** do conjunto de **operações** executadas pelo algoritmo;
- Associa-se um custo à cada operação executada;
- Em geral, ignora-se o custo de algumas operações e se contabiliza apenas as operações mais significativas;
- As operações mais significativas são denominadas **operações básicas**;

Stanford University

Modelo Detalhado

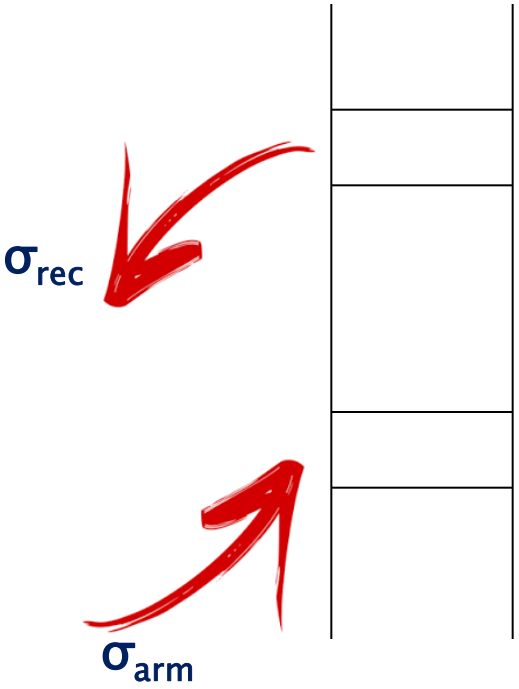
- O tempo de execução de um algoritmo será calculado pela somatória do tempo necessário para a execução das operações básicas;

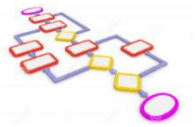
- O tempo de processamento das operações básicas é definido por um conjunto de **axiomas**.



Axioma 1

- Os tempos requeridos para recuperar um operando da memória e para armazenar o resultado na memória são constantes: σ_{rec} , σ_{arm} , respectivamente.





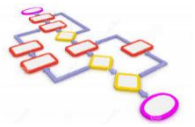
Exemplo - Axioma 1

Qual o tempo de processamento da atribuição?

y = x ;

- ✓ Será necessário recuperar-se em memória o conteúdo da variável x. Este tempo é σ_{rec} .
- ✓ O tempo requerido para se armazenar o valor na variável y é σ_{arm} .

Resposta: $\sigma_{rec} + \sigma_{arm}$



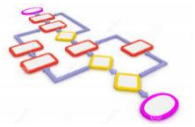
Outro Exemplo - Axioma 1

Qual o tempo de processamento do comando ?

y = 1 ;

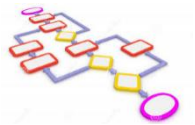
- ✓ A constante 1 (chamada de literal numérico) também precisa ser armazenada na memória (tabela de literais gerada pelo compilador).
- ✓ Assim, o custo para se recuperar em memória a constante 1 também será σ_{rec} .
- ✓ O tempo requerido para se armazenar o valor na variável y é σ_{arm}.

Resposta: $\sigma_{rec} + \sigma_{arm}$



Axioma 2

- ✓ Os tempos necessários para se realizar operações aritméticas elementares, tais como: adição, subtração, multiplicação, divisão e comparação são todos constantes.
- ✓ Estes tempos são denotados por: σ_+, σ_-, σ_x, $\sigma_/$, $\sigma_\leq$, respectivamente.



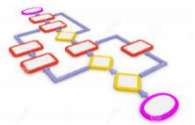
Exemplo – Axioma 2

Qual o tempo de processamento da atribuição ?

y = y + 1 ;

- ✓ Neste caso, temos a necessidade de recuperar dois valores em memória: y e 1.
- ✓ Assim, o tempo para se recuperar estes valores será dado por: $2\sigma_{rec}$.
- ✓ O tempo para se efetuar a soma é σ_{+}.
- ✓ O tempo requerido para se armazenar o resultado na variável y é σ_{arm}.

Resposta: $2\sigma_{rec} + \sigma_{+} + \sigma_{arm}$



Axioma 3

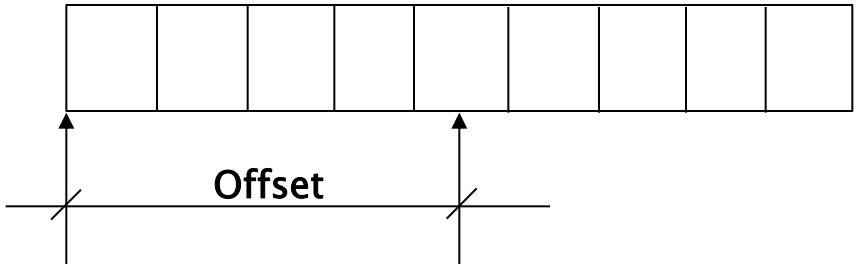
- O tempo necessário para se chamar um método é constante: $\sigma_{chamada}$ e o tempo necessário para se retornar de um método é constante: $\sigma_{retorno}$.
- Quando um método é chamado, algumas operações de bastidores são necessárias: save de endereços de retorno, chaveamento de contexto, etc.
- Estas operações são desfeitas no momento de retorno.

Axioma 4

- O tempo necessário para se passar um parâmetro a um método é o mesmo tempo para se armazenar um valor em memória: σ_{arm}.
- Conceitualmente, o esforço computacional necessário para o tratamento da passagem de parâmetros é o mesmo que se atribuir ao parâmetro formal do método o valor do argumento.

Operações com índices de Arrays

- Em geral, os elementos de um array são armazenados em locais contíguos de memória.
- Assim, dado o endereço do primeiro elemento do array, uma simples operação de adição é suficiente para se determinar o endereço de um elemento arbitrário do array.



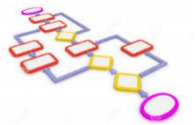
Axioma 5

- O tempo requerido para o cálculo do endereço advindo de uma operação de índice de um array, por exemplo, a[i], é constante: $\sigma_.$;
- Esse tempo não inclui o tempo para calcular a expressão do índice, nem inclui o tempo de acesso (ou seja, o tempo de recuperação ou armazenamento) ao elemento do array;
- Exemplo: $y = a[i] \Rightarrow$ Tempo: $3\sigma_{rec} + \sigma_. + \sigma_{arm}$
- Serão necessárias três recuperações: a primeira para recuperar a (o endereço base do array), a segunda para recuperar i e a terceira para recuperar o elemento a[i].

Modelo Simplificado de KNUTH

Modelo Simplificado

- O modelo detalhado fornece uma boa previsão do desempenho de execução de um algoritmo;
- No entanto, tal modelo é oneroso e trabalhoso;
- No modelo simplificado, eliminamos a dependência de tempo, considerando-se um tempo constante e igual ao ciclo do processador (**T=1**);
- Assim, neste modelo contabiliza-se apenas a quantidade de operações efetuadas pelo algoritmo.



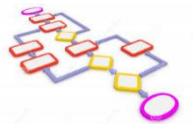
Modelo Simplificado - Axioma 1

- O total de ciclos de processador requeridos para recuperar um operando da memória e para armazenar o resultado na memória são constantes (1 ciclo = 1 operação para recuperar e 1 ciclo = 1 operação para armazenar)

Exemplo: **Qual o total de operações para executar a atribuição?**

$$y = x ; \quad (\sigma_{rec} + \sigma_{arm})$$

Resposta: 1 + 1 = 2 operações



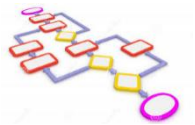
Modelo Simplificado - Axioma 2

- As operações necessárias para se realizar operações aritméticas elementares, tais como: adição, subtração, multiplicação, divisão e comparação são todas constantes e iguais a 1 ciclo cada (1 operação).

Exemplo: Qual o total de operações para processar a atribuição?

$$y = y + 1 \; ; \quad (2\sigma_{rec} + \sigma_{+} + \sigma_{arm})$$

Resposta: 4 operações



Modelo Simplificado – Axioma 3

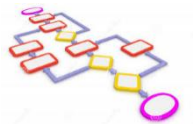
- Gasta-se 1 ciclo de processador (1 operação) para se chamar um método e 1 ciclo (1 operação) para se providenciar o retorno do método.

Resposta: 2 operações

Modelo Simplificado – Axioma 4

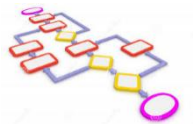
- Gasta-se 1 ciclo de processador (1 operação) para se passar um parâmetro a um método.

Resposta: 1 operação



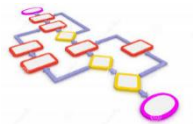
Modelo Simplificado – Axioma 5

- y = a[i] => Tempo: $3\sigma_{rec} + \sigma_{.} + \sigma_{arm}$
- Serão necessárias três recuperações: a primeira para recuperar **a** (o endereço base do array), a segunda para recuperar **i** e a terceira para recuperar o elemento a[i] ;
- Teremos, portanto um total de 5 operações básicas.



Função de Complexidade

- Para medir o custo de execução de um algoritmo é comum definir uma função de custo ou função de complexidade **f** ;
- Função de complexidade de tempo: **f(n)** mede o tempo necessário para executar um algoritmo em um problema de tamanho **n** ;
- Função de complexidade de espaço: **f(n)** mede a memória necessária para executar um algoritmo em um problema de tamanho **n** ;
- A complexidade de tempo na realidade não representa tempo diretamente, mas o número de vezes que determinada operação, considerada relevante, é executada.



Exemplo

Usando o modelo simplificado, apresente a equação que define o tempo de processamento do algoritmo que calcula a somatória de uma série aritmética simples:

$$\sum_{i=1}^{n} i$$

Exemplo

O algoritmo pode ser implementado pelo código abaixo:

```
package maua;
import java.util.Scanner;

public class Somatoria {
        public static int Soma (int n){

            int resultado = 0;
            for (int i = 1; i <= n; ++i)
              resultado += i;
             return resultado;
         }
        public static void main(String[] args) {

              Scanner Input = new Scanner (System.in);

              System.out.print("Entre com o valor de n: ");

              int n = Input.nextInt();

              int resposta = Soma(n) ;

              System.out.println("Soma(" + n + ")= " + Soma(n));
        }
}
```

```
public static int Soma (int n){

2       int resultado = 0;
3       for (int i = 1; i <= n; ++i)
4         resultado += i;
5        return resultado;
     }
```

```
public static void main(String[] args) {

        ...

1       int resposta = Soma(n) ;

        ...
}
```

```
public static void main(String[] args) {

    ...

1   int resposta = Soma(n) ;

    ...
}
```

- 1 – operação para se recuperar a variável n
- 1 – operação para passagem do argumento n à função Soma
- 1 – operação para chamada da função Soma
- Σ op $_{Soma(n)}$ – total de operações do método Soma
- 1 – operação de retorno da função Soma
- 1 σ_{arm} – operação de armazenamento na variável resposta

$$5 + \Sigma \text{ op } _{Soma(n}$$

```
public static int Soma (int n){

    int resultado = 0;
    for (int i = 1; i <= n; ++i)
      resultado += i;
    return resultado;
}
```

2

2 operações

```
public static int Soma (int n){

    int resultado = 0;
3a  for (int i = 1; i <= n; ++i)
      resultado += i;
    return resultado;
}
```

Código	Tempo
int i = 1	2 operações

- Este tempo corresponde à primeira parte do código for que representa a etapa de inicialização;
- É executado uma única vez antes da primeira iteração do loop.

```
public static int Soma (int n){

    int resultado = 0;
3b  for (int i = 1; i <= n; ++i)
      resultado += i;
    return resultado;
}
```

Código	Tempo
i <= n	3 x (n+1) operações

- Este tempo corresponde ao teste de término do loop;
- É executado antes do início de cada iteração do loop;
- O número de vezes em que o teste de término do loop é feito é um a mais que o número de vezes em que o corpo do loop é executado.

```
public static int Soma (int n){

    int resultado = 0;
3c  for (int i = 1; i <= n; ++i)
      resultado += i;
    return resultado;
}
```

Código	Tempo
++i	4 x (n) operações

- Este tempo corresponde ao terceiro elemento do for, o passo de incremento do contador do loop. Equivale a i = i +1;
- É executado uma vez a cada iteração do loop. Portanto, n vezes.

```
public static int Soma (int n){

    int resultado = 0;
    for (int i = 1; i <= n; ++i)
4 ➡   resultado += i;
     return resultado;
}
```

Código	Tempo
resultado += i	4 x (n) operações

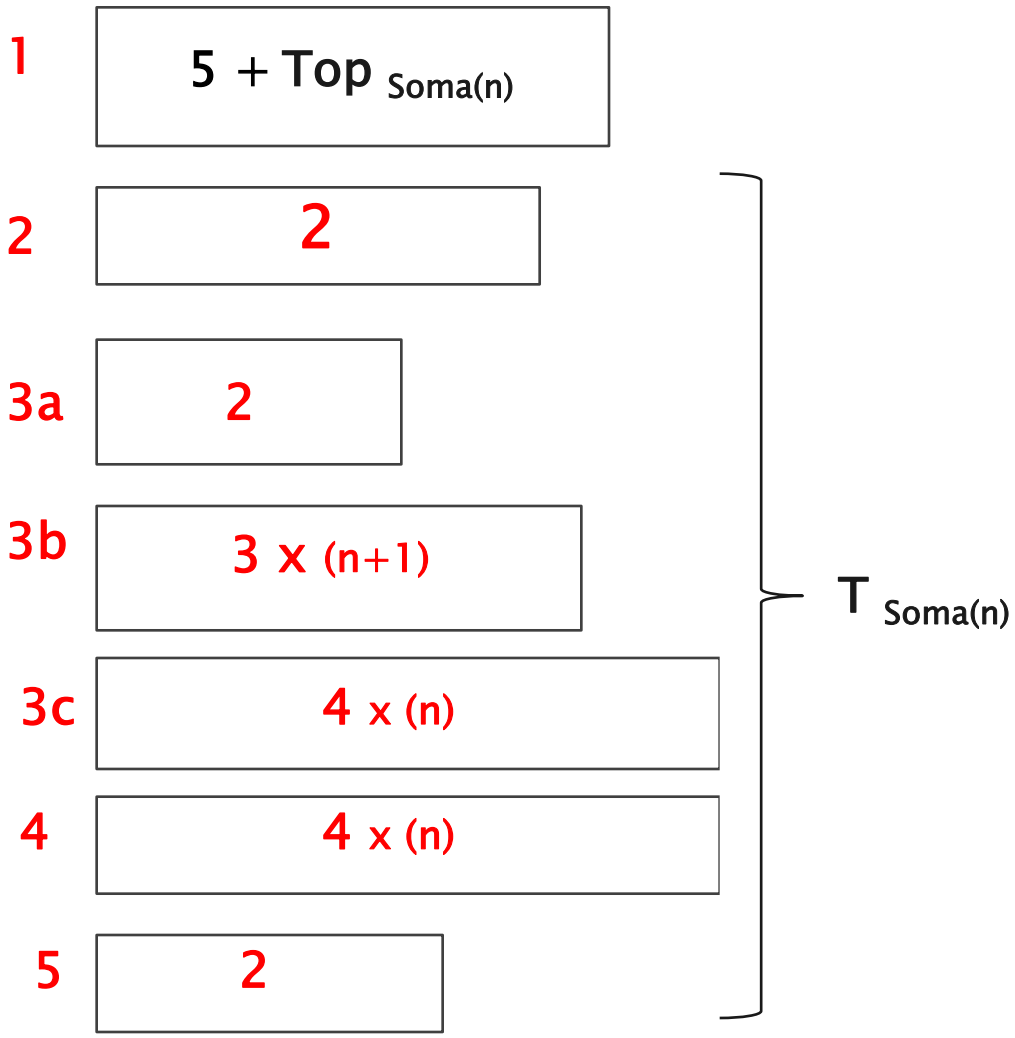
- Este tempo corresponde ao corpo do loop;
- É executado n vezes.

```
public static int Soma (int n){

    int resultado = 0;
    for (int i = 1; i <= n; ++i)
      resultado += i;
5   return resultado;
    }
```

Código	Tempo
return resultado;	2 operações

- Este tempo corresponde ao retorno da variável resultado;
- A variável é lida na memória e armazenada na pilha (registro de ativação).

Contagem Total de Operações



Contagem Total de Operações

1	$5 + \Sigma op_{Soma(n)}$
2	2
3a	2
3b	3 x (n+1)
3c	4 x (n)
4	4 x (n)
5	2

$5 + \Sigma op_{Soma(n)}$

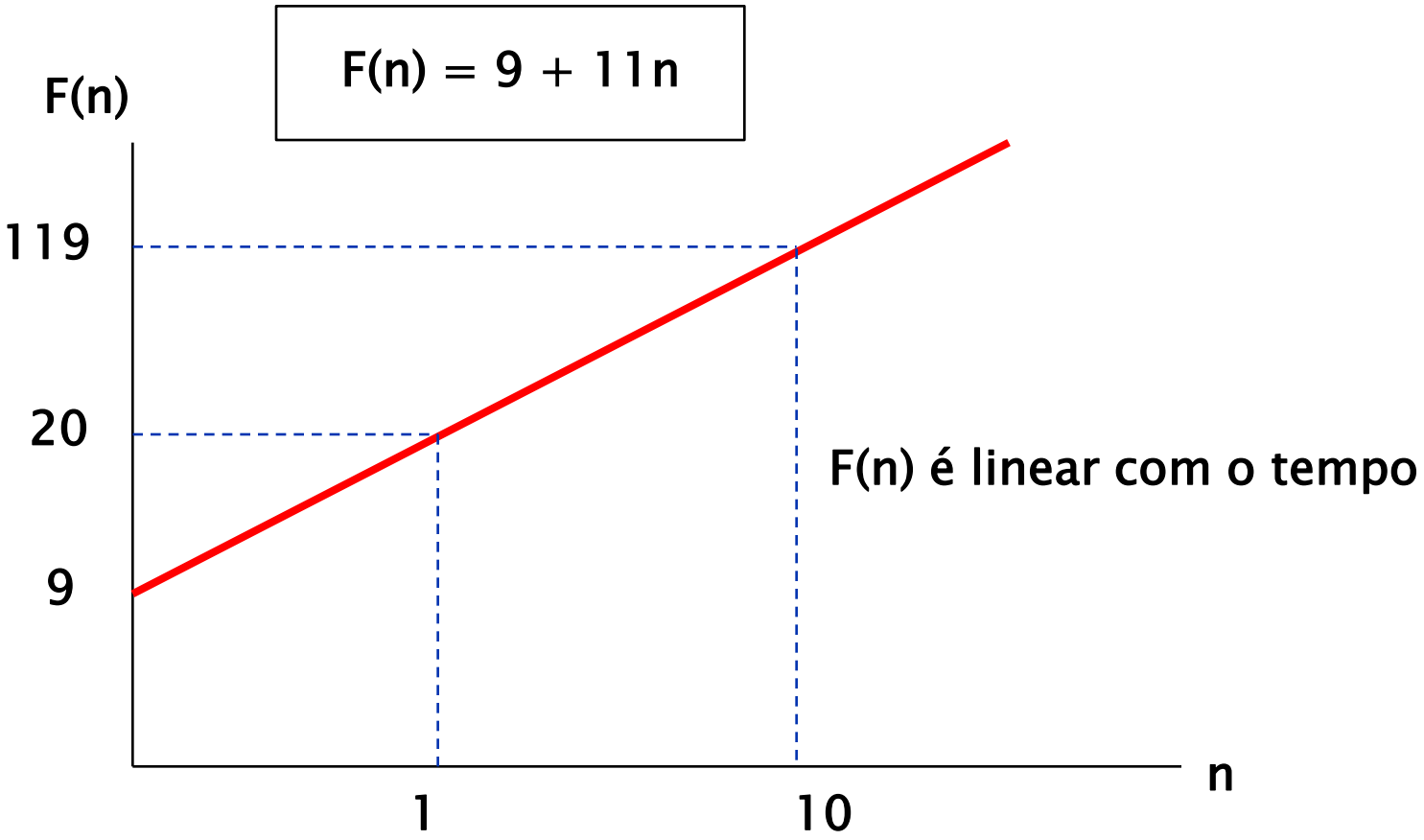
$5 + (9 + 11n)$

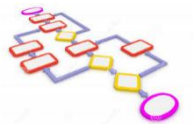
$14 + 11n$

$F(n) = 6 + 3(n+1) + 8n$

$F(n) = 6 + 3n + 3 + 8n$

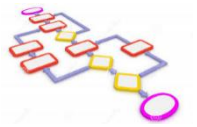
$F(n) = 9 + 11n$


F(n) = 9 + 11n
F(n)
119
20
9
1
10
n
F(n) é linear com o tempo



Função de Complexidade

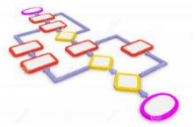
- Para medir o custo de execução de um algoritmo é comum definir uma função de custo ou função de complexidade f.
- Função de complexidade de tempo: f(n) mede o tempo necessário para executar um algoritmo em um problema de tamanho n .
- Função de complexidade de espaço: f(n) mede a memória necessária para executar um algoritmo em um problema de tamanho n.
- A complexidade de tempo na realidade não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada.



Exemplo: Maior elemento de um array de inteiros

- Seja V um array de n elementos inteiros, n >= 1.

```
public class Max {

   public static void main(String[] args) {

         int[] V = { 2,5,6,4,2,10,2,3,5,8,1,2 };
         // vetor tem 12 elementos

         int Max=V[0],contador = 0;

         for (int i=1; i< V.length; i++) {
                  if (V[i] > Max) Max = V[i]; contador++;
         }

         System.out.println("Maior Valor do Vetor V = " + Max);
         System.out.println("Contador = " + contador );
   }

}
```

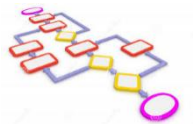


Exemplo: Maior elemento de um array de inteiros

Trace de Execução

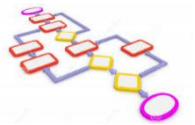
```
Maior Valor do Array V = 10
Contador = 11
```

- ✓ O vetor tem **12** elementos e foram executadas **11** comparações
- ✓ Assim, se o array tiver **n** elementos, serão executadas **n-1** comparações . . .



Exemplo: Maior elemento de um array de inteiros

- Seja f um função de complexidade tal que f(n) corresponda ao número de comparações entre os elementos de V, considerando V com n elementos;
- f(**n**) = **n – 1**, para n > 0;
- Logo, **n – 1** comparações são necessárias.

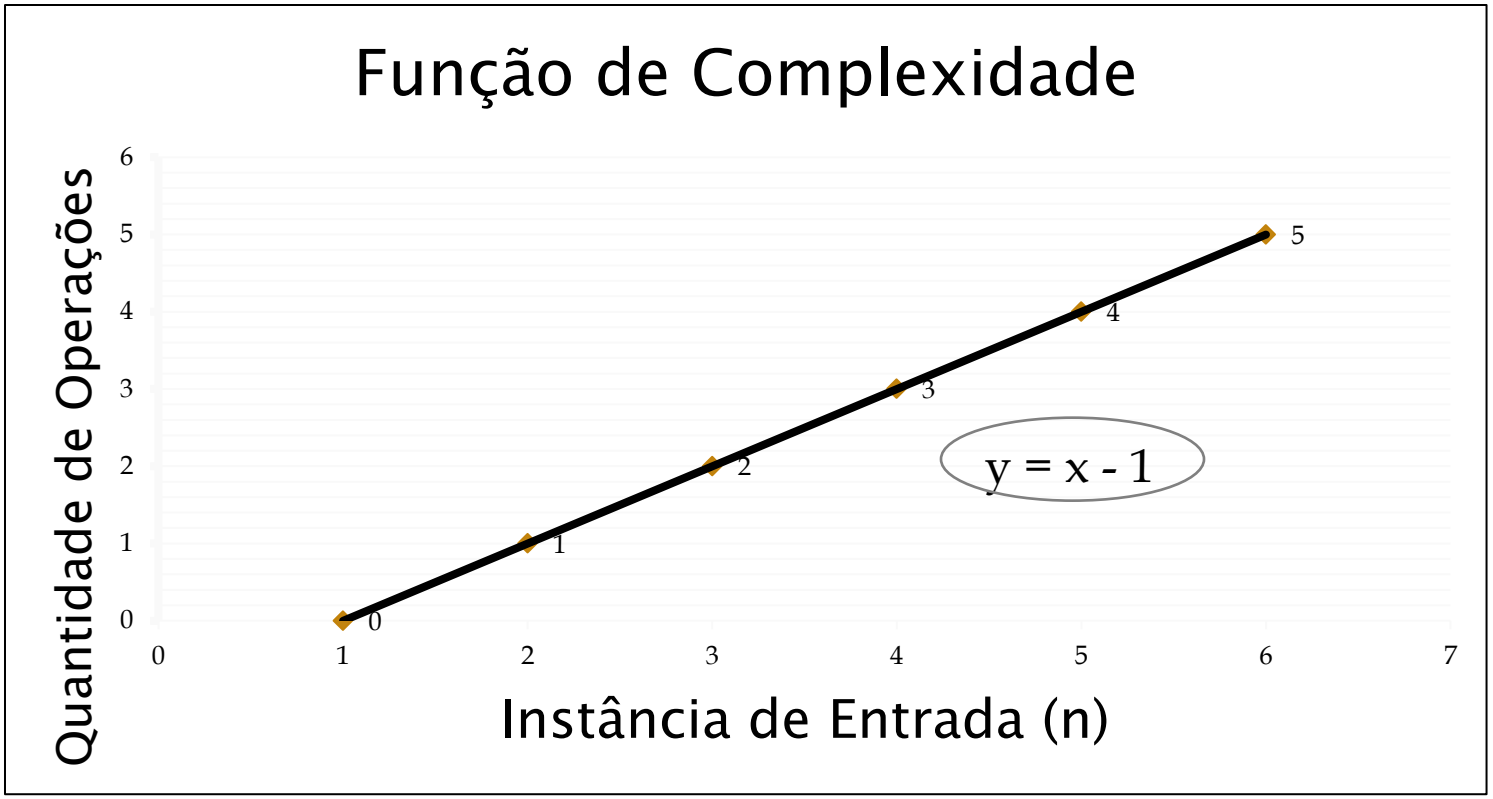


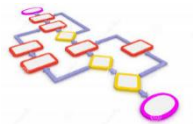
Exemplo: Maior elemento de um array de inteiros

Função de Complexidade

n	f(n) = n- 1
1	0
2	1
3	2
4	3
10	9
20	19

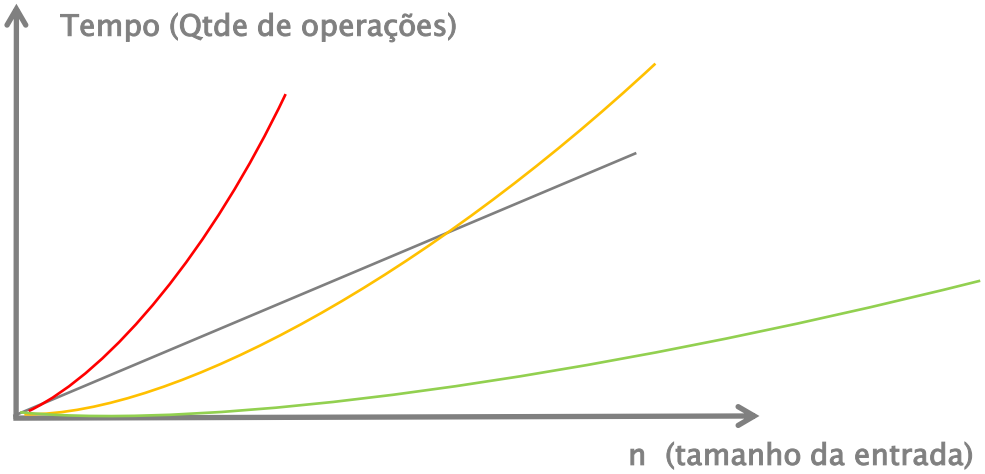
Exemplo: Maior elemento de um vetor de inteiros





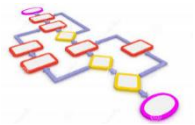
Tamanho da entrada de dados (Instância)

- A medida do custo de execução de um algoritmo depende principalmente do tamanho da entrada dos dados;
- É comum considerar o tempo de execução de um programa como uma função do tamanho da entrada.



Tamanho da entrada de dados

- No caso do método **max** do programa do exemplo, o custo é proporcional à entrada de dados submetida ao algoritmo;
- Já para um algoritmo de ordenação isso não ocorre: se os dados de entrada já estiverem quase ordenados, então o algoritmo irá trabalhar menos.

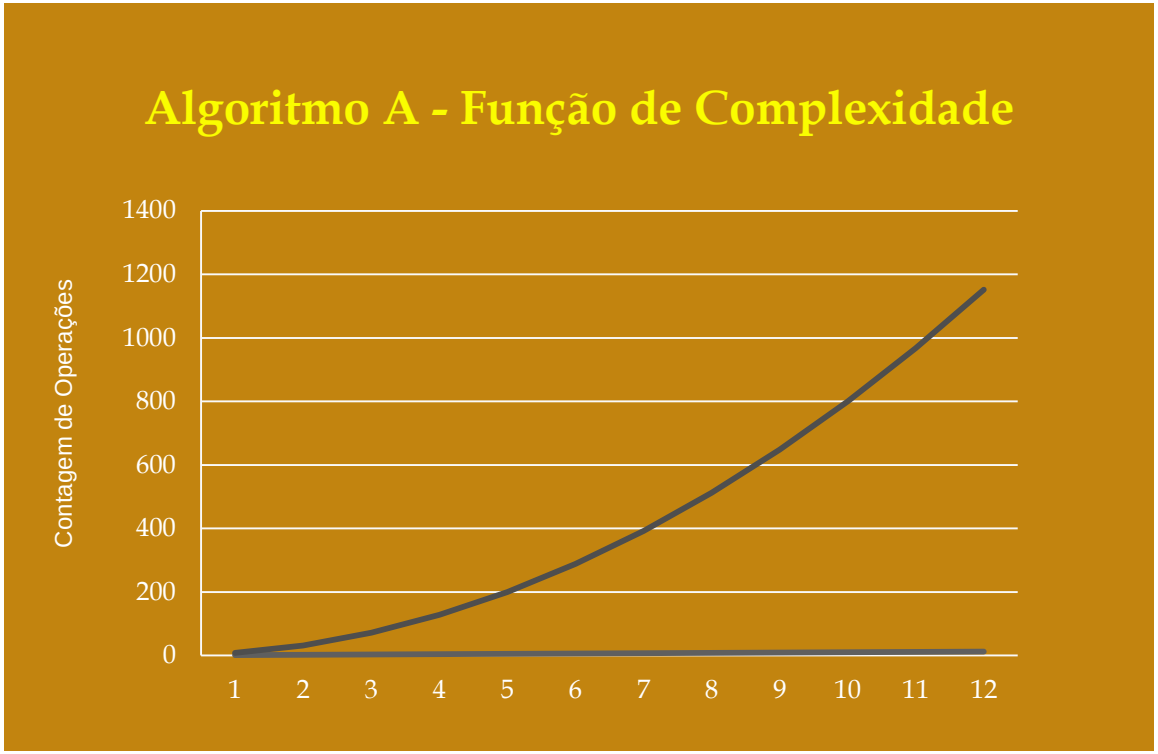


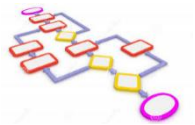
Atividade 1

Considere dois algoritmos **A** e **B** com complexidades respectivamente iguais a $8n^2$ e n^3.
Qual o maior valor de n, para o qual o algoritmo B é mais eficiente que o algoritmo A ?

Função de Complexidade $8n^2$

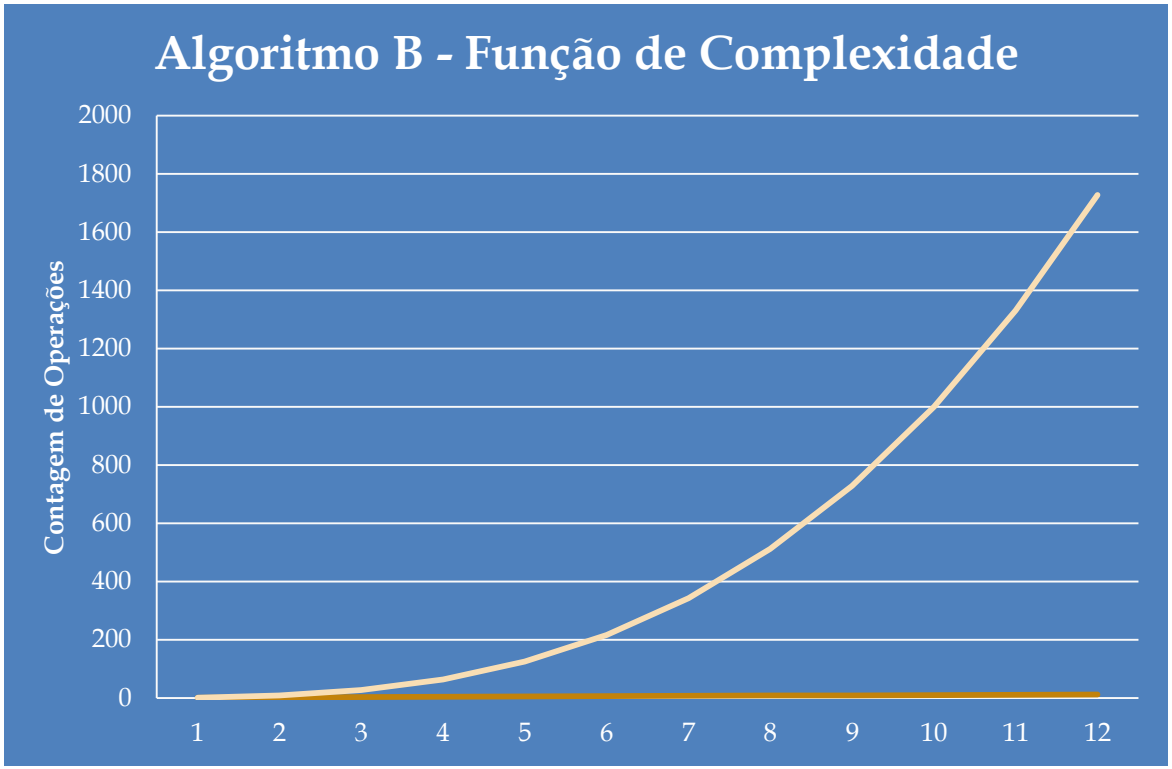
n	$8n^2$
1	8
2	32
3	72
4	128
5	200
6	288
7	392
8	512
9	648
10	800
11	968
12	1152



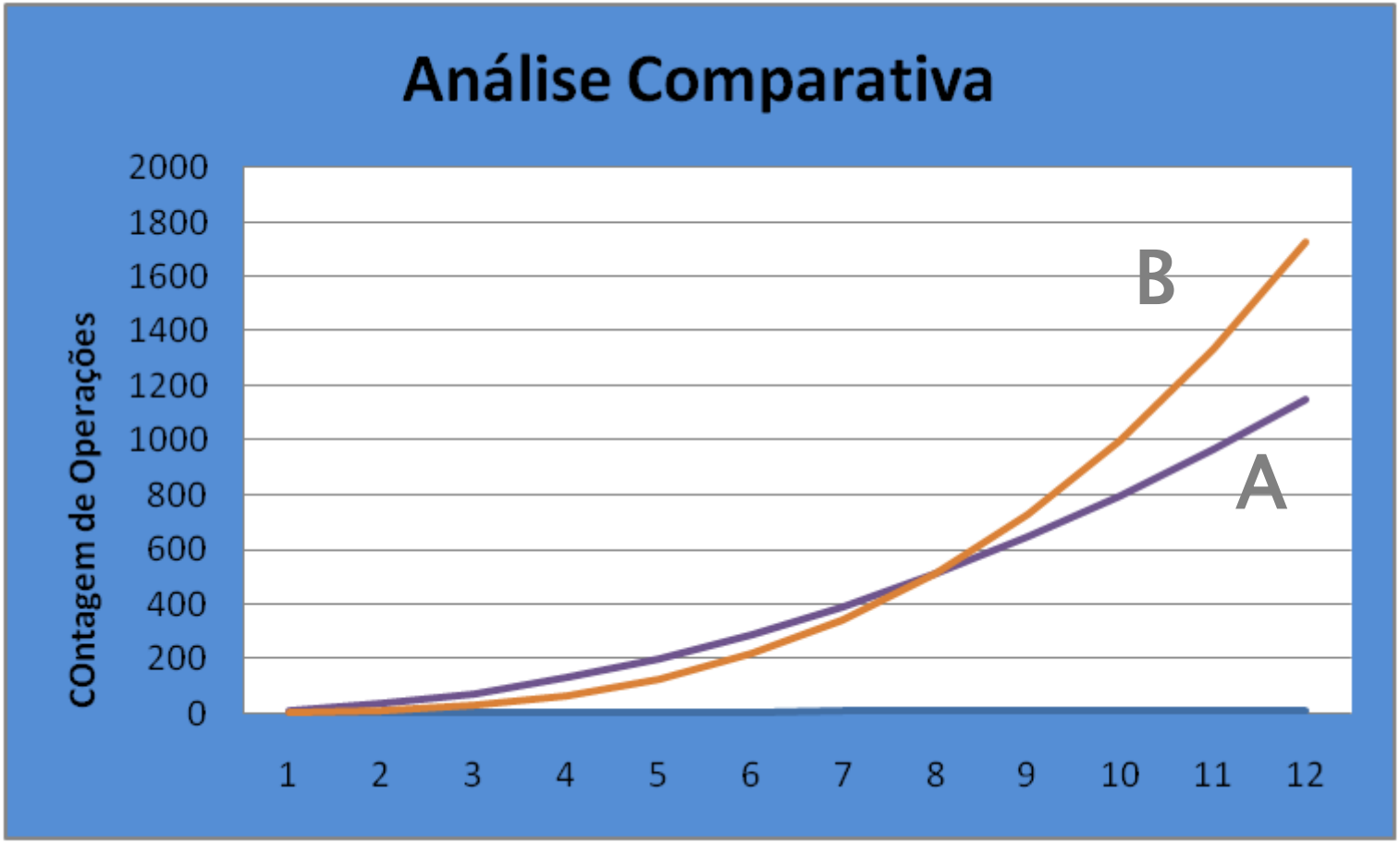


Função de Complexidade n^3

n	n^3
1	1
2	8
3	27
4	64
5	125
6	216
7	343
8	512
9	729
10	1000
11	1331
12	1728

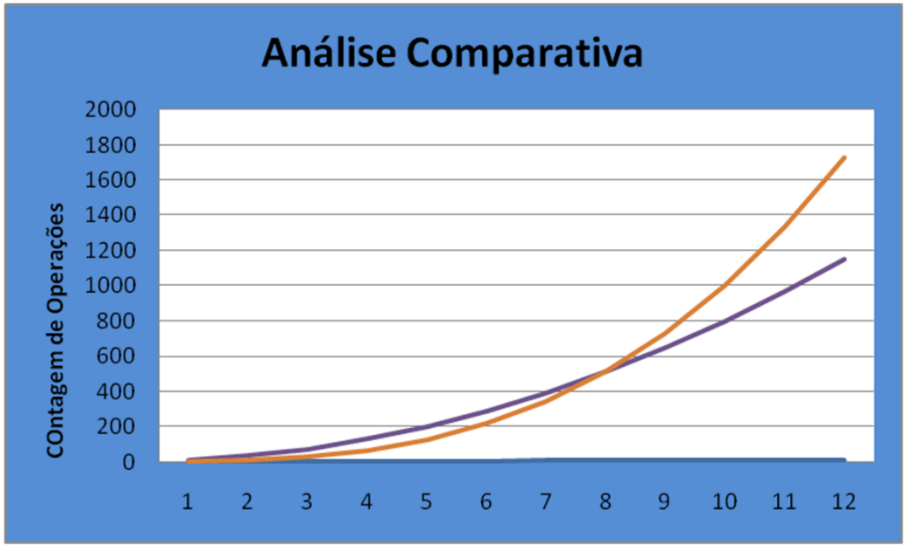


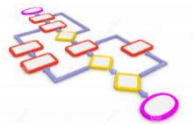
Análise Gráfica



Atividade 1

- ✓ Até um determinado valor de instância de entrada, o algoritmo B é melhor.
- ✓ A partir de um certo valor de n, o algoritmo A passa a ser melhor.
- ✓ O ponto de equilíbrio ocorre quando: $n^3 = 8n^2 => n^3 - 8n^2 = 0 => n^2 (n - 8) = 0$
- ✓ Assim, n = 0 ou n-8 = 0. Portanto, o ponto de equilíbrio é **n = 8**
- ✓ Assim, o algoritmo B é mais eficiente até n = 7.

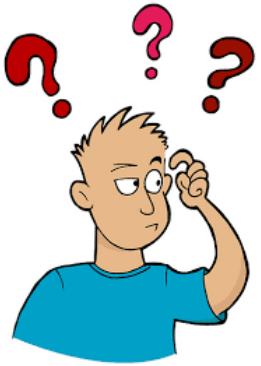




Atividade - 2

a) Um algoritmo tem complexidade $2n^2$. Num certo computador, num tempo t, o algoritmo resolve um problema de tamanho 25. Imagine agora que se tenha disponível um computador 100 vezes mais rápido. Qual o tamanho máximo de problema que o mesmo algoritmo resolve no mesmo tempo t no computador mais rápido ?

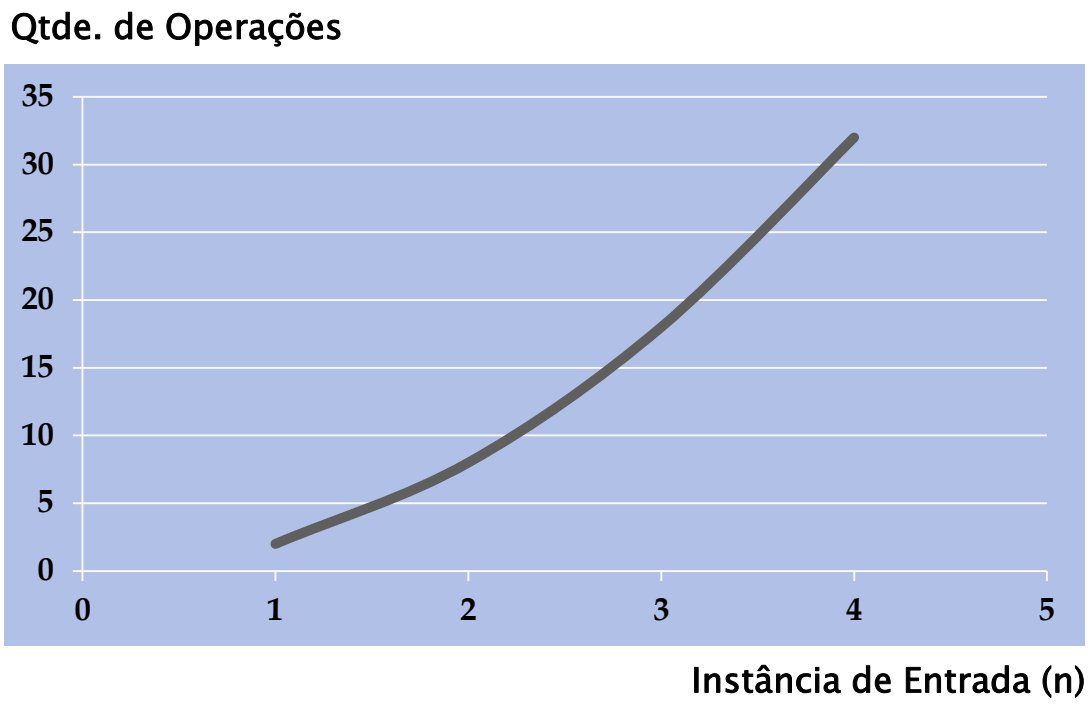
b) Considere o mesmo problema para um algoritmo de complexidade 2^n.

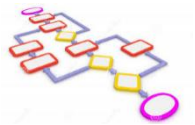


Atividade – 2a

Função de Complexidade $2n^2$

n	$2n^2$
1	2
2	8
3	18
4	32
5	50
10	200
25	1250





Atividade – 2a

- ✓ Analisando-se o comportamento da função de complexidade, pode-se afirmar que para **n**=25, são necessárias 1250 operações;
- ✓ Estas operações são executadas no computador antigo em um determinado tempo **t** ;
- ✓ No computador novo as operações são executadas **100** vezes mais rápidas ;
- ✓ Assim, no computador novo (no mesmo tempo t) pode-se executar 1250*100 = **125.000** operações.

Atividade – 2a

- ✓ Como o algoritmo é o mesmo (tanto no computador novo quando no antigo), a função de complexidade é a mesma ($f(n) = 2n^2$).
- ✓ Assim, no mesmo tempo t, o computador novo executa 125.000 operações, o que representa (certamente) uma instância maior.
- ✓ Teremos então: $f(n) = 2n^2 \Rightarrow 125000 = 2n^2$

$$\Rightarrow 62500 = n^2$$

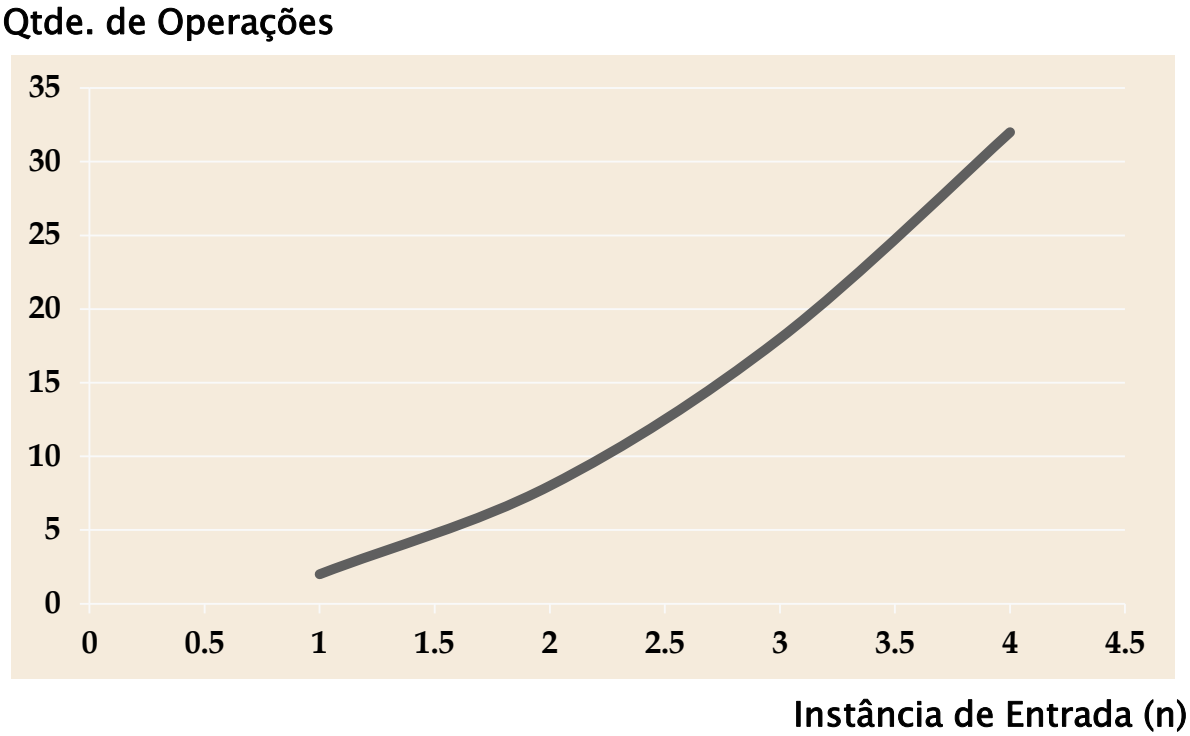
$$\Rightarrow \quad n = 250$$

- ✓ Resposta: O tamanho máximo de problema que o mesmo algoritmo resolve no tempo t (no computador mais rápido) é 250.
- ✓ Observação: Embora o computador mais novo seja 100 vezes mais rápido, o tamanho do problema aumentou apenas 10 vezes (de 25 para 250).

Atividade – 2b

Função de Complexidade 2^n

n	2^n
1	2
2	4
3	8
4	16
5	32
10	1024
25	33554432



Atividade – 2b

- ✓ Analisando-se o comportamento da função de complexidade, podemos afirmar que para n=25, são necessárias 33.554.432 operações.
- ✓ Estas operações são executadas no computador antigo em um determinado tempo t.
- ✓ No computador novo as operações são executadas **100** vezes mais rápidas.
- ✓ Assim, no computador novo (no mesmo tempo t) podem-se executar 33.554.432 *100 = 3.355.443.200 operações.

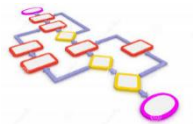
Atividade – 2b

- ✓ Como o algoritmo é o mesmo (tanto no computador novo quando no antigo), a função de complexidade é a mesma ($f(n) = 2^n$).
- ✓ Assim, no mesmo tempo t, o computador novo executa 3.355.443.200 operações, o que representa (certamente) uma instância maior.
- ✓ Teremos então: $f(n) = 2^n \Rightarrow 2^n = 3355443200$

$$\log_2 2^n = \log_2 3355443200$$

$$n = 31$$

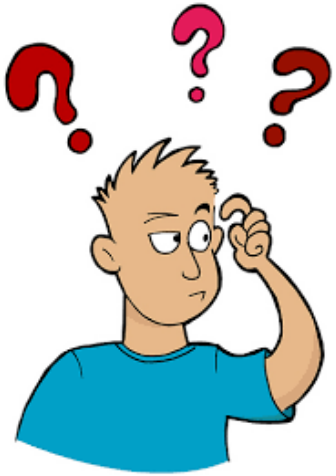
- ✓ Resposta: O tamanho máximo de problema que o mesmo algoritmo resolve no tempo t (no computador mais rápido) é 31.
- ✓ Observação: Embora o computador mais novo seja 100 vezes mais rápido, o tamanho do problema aumentou apenas 1,24 vezes (de 25 para 31) .

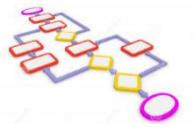


Atividade - 3

a) Suponha que uma empresa utiliza um algoritmo de complexidade n^2 que, em um tempo t, na máquina disponível, resolve um problema de tamanho x. Suponha que o tamanho do problema a ser resolvido aumentou em 20%, mas o tempo de resposta deve ser mantido. Para isso, a empresa pretende trocar a máquina por uma mais rápida. Qual percentual de melhoria no tempo de execução das operações básicas é necessário para atingir sua meta?

b) Suponha que no problema anterior, ainda se queira reduzir em 50% o tempo de resposta. Qual a melhoria esperada para a nova máquina?





Atividade – 3a

Máquina Velha

Qtde. de Operações	Tempo de cada Operação	Tempo Total
x^2	t_v	$tt_v = x^2 . t_v$

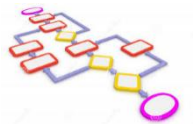
Máquina Nova

Qtde. de Operações	Tempo de cada Operação	Tempo Total
$(1.2x)^2$	t_n	$tt_n = 1.44x^2 . t_n$

Atividade – 3a

- ✓ O problema afirma que o tempo total da máquina nova deve ser igual ao tempo total da máquina velha;
- ✓ Assim, $tt_n = tt_v$
- ✓ Portanto, como: $tt_v = x^2 . t_v$ e $tt_n = 1.44.x^2 . t_n$
- ✓ Teremos: $x^2 . t_v = 1.44 . x^2 . t_n$
- ✓ Portanto: $t_v = 1.44 . t_n$
- ✓ Assim: $t_v = 1.44 . t_n \Rightarrow tv = 1.t_n + 0.44\, t_n \Rightarrow t_v = t_n + 44/100 . t_n$

A máquina velha é 44% mais lenta que a nova, ou
A máquina nova é 44% mais rápida que a velha



Atividade – 3b

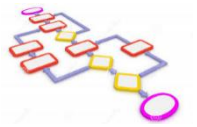
Máquina Velha		
Qtde. de Operações	Tempo de cada Operação	Tempo Total
x^2	t_v	$tt_v = x^2 . t_v$

Máquina Nova		
Qtde. de Operações	Tempo de cada Operação	Tempo Total
$(1.2x)^2$	t_n	$tt_n = 1.44x^2 . t_n$

Atividade – 3b

- ✓ O problema afirma que o tempo total da máquina nova deve ser 50% inferior ao tempo total da máquina velha.
- ✓ Assim, tt_n = (50%) tt_v => $tt_n = 0{,}5\ tt_v$
- ✓ Portanto, como: $tt_v = x^2 . t_v$ e $tt_n = 1{,}44.x^2 . t_n$
- ✓ Teremos: $1{,}44 . x^2 . t_n = 0{,}5 . x^2 . t_v$
- ✓ Portanto: $1{,}44 . t_n = 0.5 . t_v$
- ✓ Assim: $2{,}88 . t_n = t_v => t_v = t_n + 1{,}88 . t_n => t_v = t_n + 188/100 . t_n$

A máquina velha é 188% mais lenta que a nova, ou
A máquina nova é 188% mais rápida que a velha



Melhor Caso, Pior Caso e Caso Médio

- ✓ **Melhor caso**: menor tempo de execução sobre todas as entradas de tamanho n.
- ✓ **Pior caso**: maior tempo de execução sobre todas as entradas de tamanho n.
- ✓ **Caso médio** (ou caso esperado): média dos tempos de execução de todas as entradas de tamanho n.