Complexidade de Algoritmos

UFSC / CTC / INE 5609 – Estruturas de Dados

Professora: Patricia D M Plentz

Plano de Aula

- Introdução
- Estrutura da Análise de Algoritmos
- Crescimento de Funções e Notação Assintótica
- Comparação de Funções
- Resumo

Introdução

- Nesta disciplina estamos interessados em utilizar boas estruturas de dados e algoritmos.
  - **Estruturas de dados** são formas sistemáticas de organizar e acessar dados;
  - **Algoritmos** são procedimentos para realizar alguma tarefa em um tempo finito.

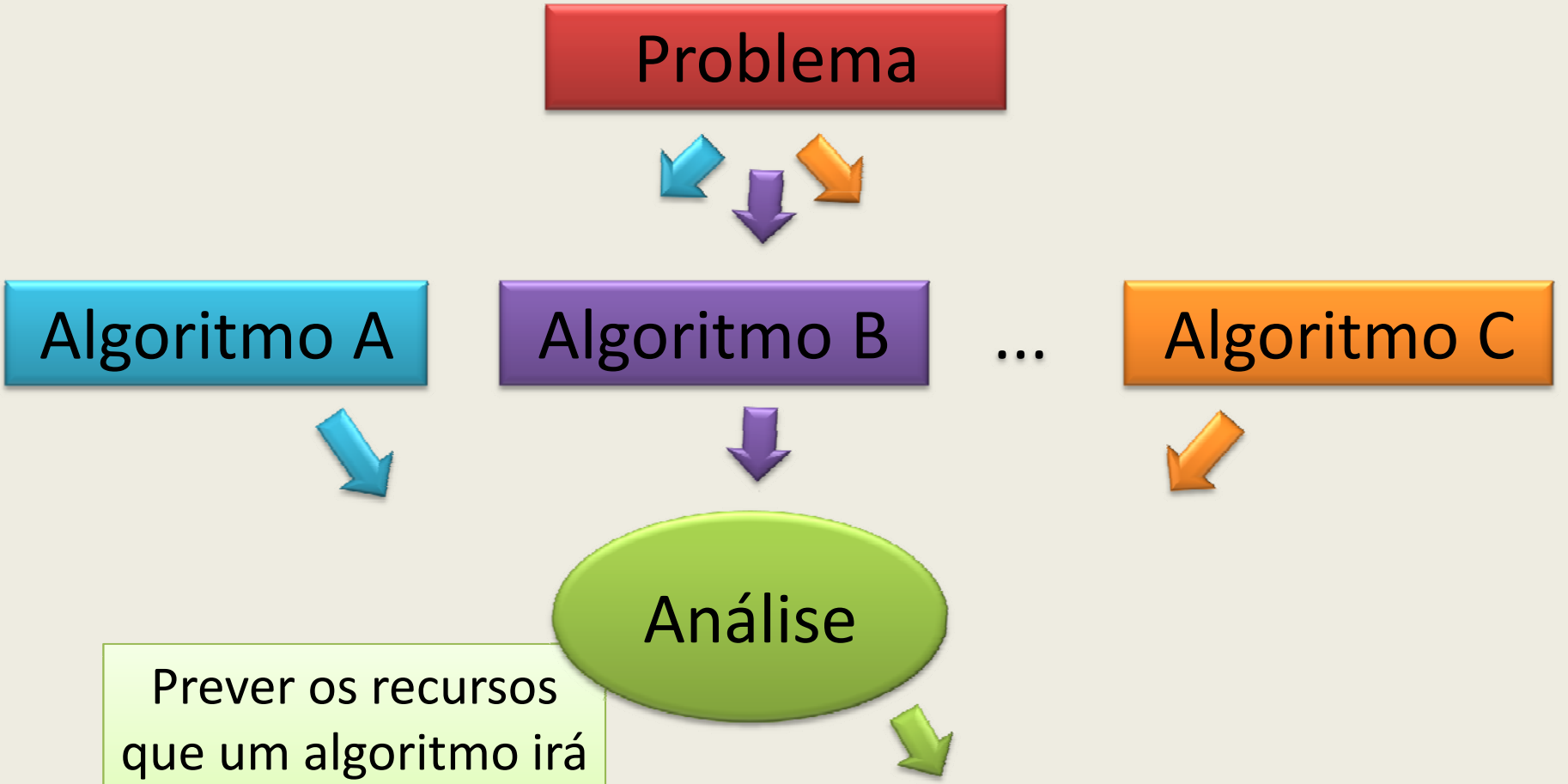
Esses conceitos são centrais para a computação

- Para sermos capazes de classificar estruturas de dados e algoritmos como **bons**, devemos ter maneiras precisas de analisá-los.
  - Uma ferramenta de análise pode, por exemplo, utilizar o **tempo de execução** de algoritmos e operações sobre estruturas de dados.

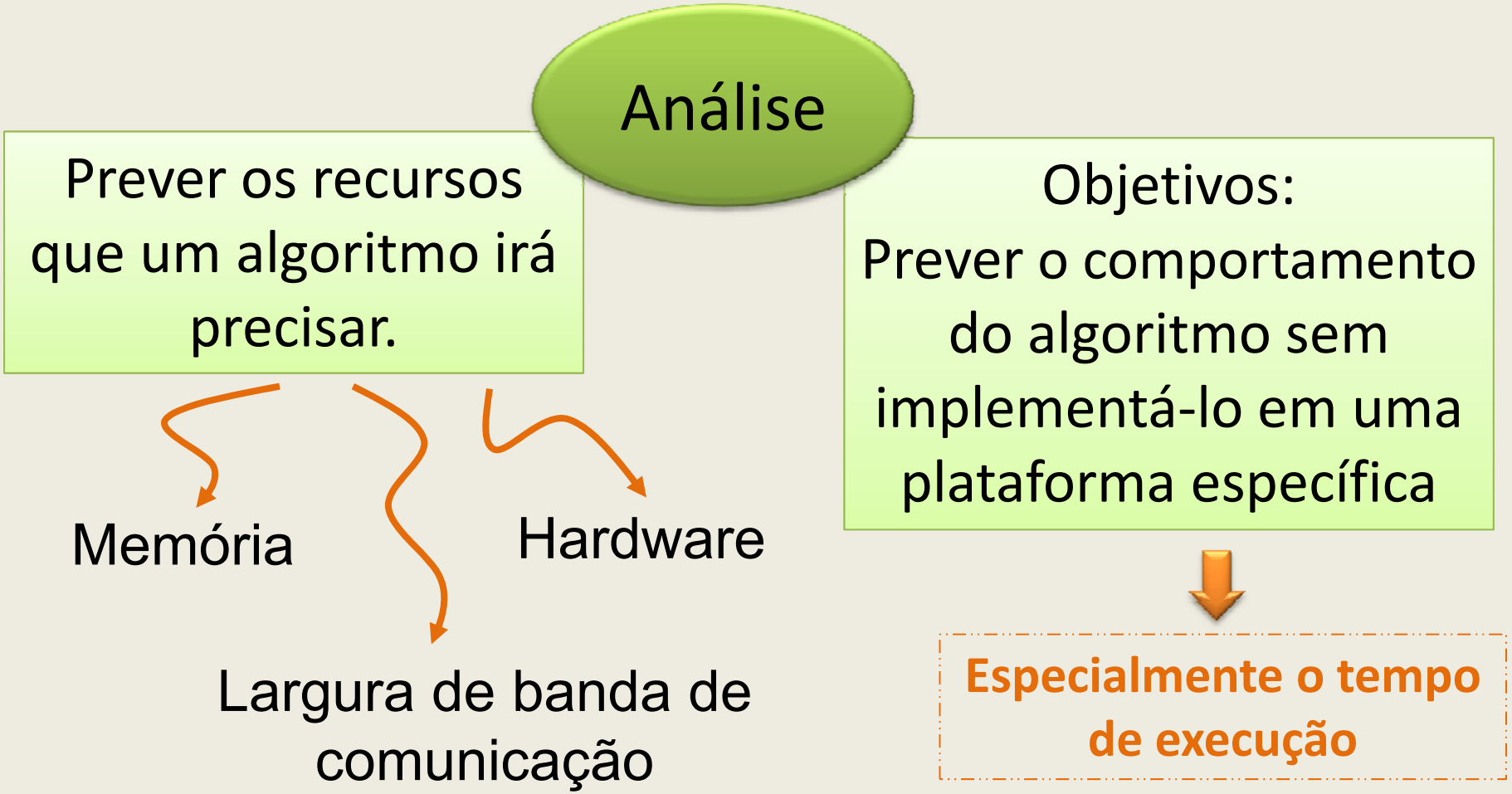
Como avaliar um algoritmo?

- Dado um algoritmo, podemos perguntar:
  - Ele fornece de fato uma solução para o problema em questão?
  - Quão eficiente é este algoritmo?
  - Qual o espaço em memória necessário para a execução do algoritmo?
  - Existe uma maneira melhor de resolver o problema em questão?

Introdução


Problema
Algoritmo A
Algoritmo B
...
Algoritmo C
Análise
Prever os recursos que um algoritmo irá precisar.
Algoritmo B é mais eficiente.

Introdução


Análise
Prever os recursos que um algoritmo irá precisar.
Memória
Hardware
Largura de banda de comunicação
Objetivos:
Prever o comportamento do algoritmo sem implementá-lo em uma plataforma específica
Especialmente o tempo de execução

Avaliação de Algoritmos

- Correção
- Eficiência Temporal
- Eficiência Espacial
- Otimalidade

- Análise Empírica
- Análise Teórica

Métodos

- **Análise Empírica**

  - Cria-se um algoritmo e executa-se testes usando um conjunto de dados (*casos de teste*).

- Entretanto:
  - **Não é** possível testar **todos** os casos :
    - ✓ Muito trabalhoso;
    - ✓ Algum caso será esquecido – caso que o algoritmo falha, caso em que o desempenho do algoritmo é particularmente muito bom ou muito ruim.

Métodos

- **Análise Teórica**

- Cria-se um algoritmo e, usando alguns critérios, realiza-se uma análise para entender seu comportamento:

  - Correção:
    - ✓ Ele fornece uma solução válida para o problema em questão?
  - Eficiência:
    - ✓ Quanto tempo o algoritmo gasta para finalizar sua execução?
    - ✓ Quanto de memória o algoritmo utiliza durante sua execução?

Critérios

- Eficiência Temporal

X

- Eficiência Espacial

Alguns anos atrás → **Atualmente**

Recursos **espaço** e **tempo** eram valiosos.

Espaço não é tão importante.

Tempo continua sendo importante.

- Quase todos os algoritmos levam mais tempo para serem executados sobre entradas maiores.
  - Torna-se interessante investigar a eficiência de algoritmos como função do parâmetro **N**

Por que não utilizar unidade de tempo?

Indica o tamanho da entrada do algoritmo

Métricas

Por que não utilizar unidade de tempo?

- Dependência de hardware
- Qualidade da implementação
- Compilador

Fatores Externos

- Como avaliar um algoritmo sem depender de ***fatores externos***?
  - Contar o número de vezes que cada operação do algoritmo é realizada.

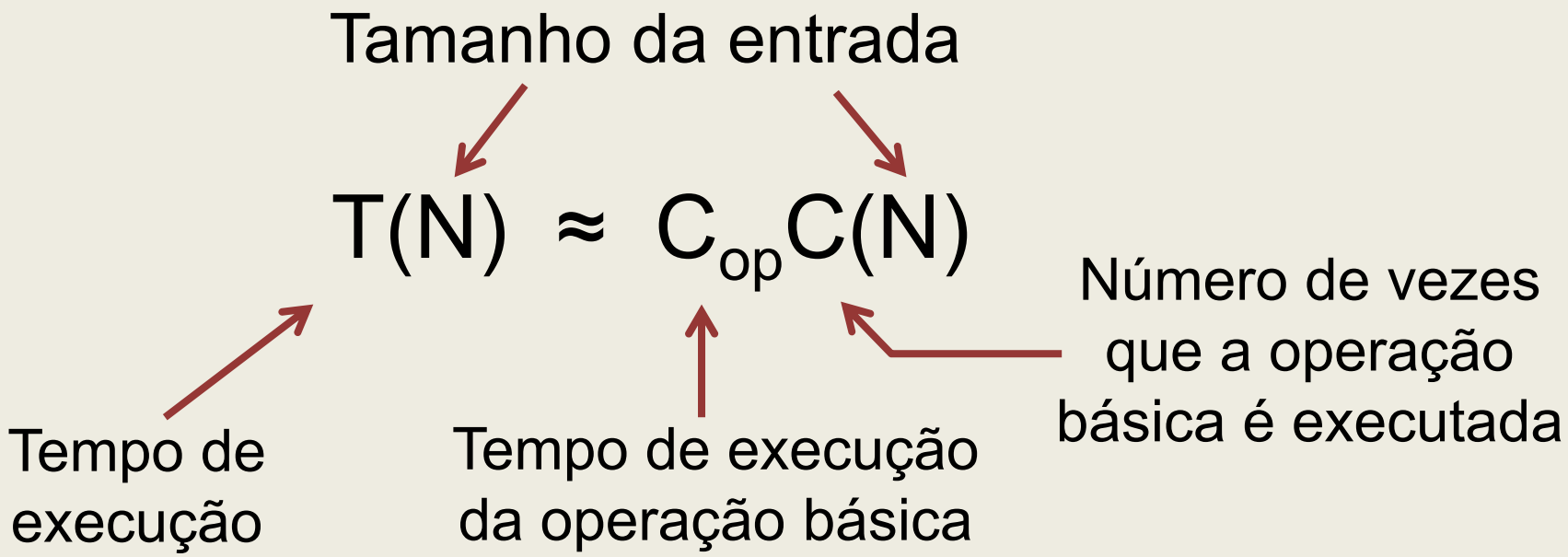
OU

  - Identificar a operação que **mais contribui para o tempo de execução total** do algoritmo, e contar o número de vezes que esta operação é realizada.

Operação Básica

Eficiência Temporal

- A *Eficiência Temporal* é analisada determinando o número de repetições de *operações básicas* como uma função do *tamanho da entrada (N)*.



Modelo Computacional
RAM (*Random Access Machine*)

- Com este modelo é possível medir o tempo de execução de um algoritmo contando o número de passos que ele gasta para uma dada instância do problema.
  - Não considera detalhes de hardware e de software.

Modelo Computacional
RAM (*Random Access Machine*)

- Modelo de Computação RAM
  - Cada operação simples (+, *, -, =, se) gasta exatamente 1 passo de tempo;
  - Laços e subrotinas não são considerados operações simples.
    - ✓São a composição de algumas operações de passo único;
  - Cada acesso à memória gasta exatamente um passo de tempo

Tamanho da Entrada e Operação Básica

Problema	Tamanho da Entrada	Operação Básica
Pesquisa por uma chave em uma lista de N itens	Número de itens na lista	Comparação de chaves
Multiplicação de duas matrizes de números de pontos flutuantes	Dimensões das matrizes (L x C) (L x C)	Multiplicação de ponto flutuante
Calcular a^N	N	Multiplicação de **a** N vezes
Grafos	Número de vértices e / ou arestas	Visitar um vértice ou atravessar uma aresta

Exemplo:
Tempo de Operação de um Algoritmo

- Pesquisa por K chaves em uma lista de N itens
  - Tamanho da entrada: N;
  - Número de itens: 10 N=10;
  - Operação Básica: comparação da chave com os itens;
  - C_{op}: 1 unidade de tempo;
  - C(N): N^2 ;

$$T(N) \approx C_{op}C(N)$$

Exemplo:
Tempo de Operação de um Algoritmo

$$T(N) \approx C_{op}C(N)$$

$$T(10) \approx 1 \,.\, (10)^2 = 100$$

Quanto tempo a mais levará a execução de um algoritmo se dobrarmos o tamanho de sua entrada? *N = 20* ?

Exemplo:
Tempo de Operação de um Algoritmo

$T(N) \approx C_{op}C(N)$

$T(20) \approx 1 \,.\, (20)^2 = 400$

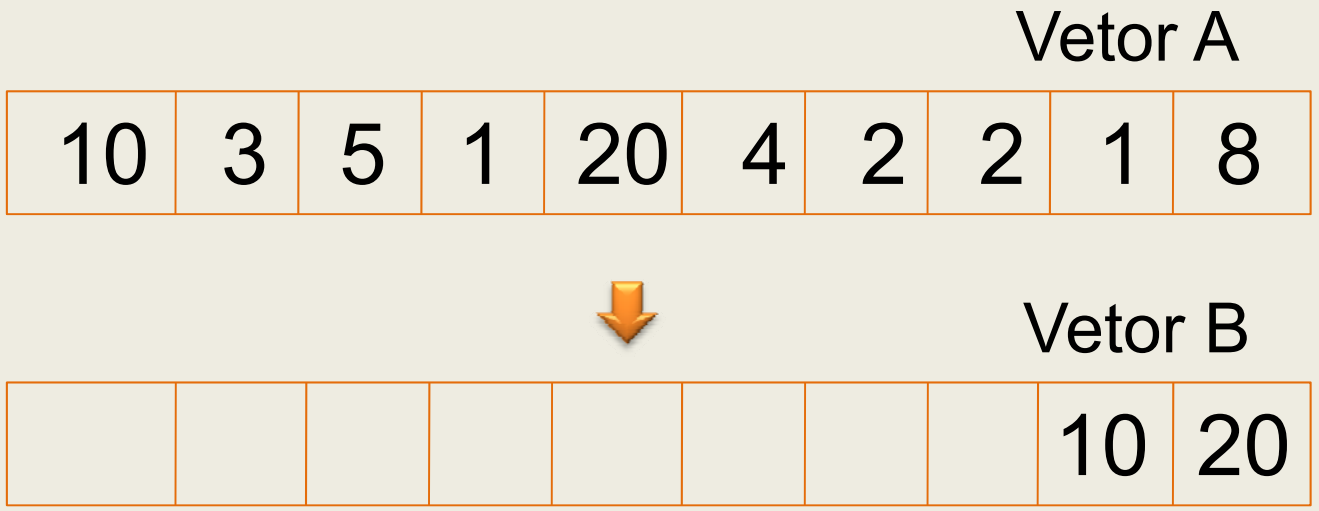
$T(10) \approx 100$

$T(20) \approx 400$

Relação
1 / 4

Análise da Complexidade de um Algoritmo

- Algoritmo de ordenação por inserção
  - Percorrer um vetor A (N = 10) desordenado buscando o maior elemento;
  - Inserir o elemento em um vetor B (N = 10) , de forma ordenada.



Análise da Complexidade de um Algoritmo

- Qual o número mínimo de operações?
  - *N op min* = 1 + 1 + 1 + 1 + ... + 1 = 10

- Qual o número máximo de operações?
  - *N op max* = N + (N -1) + (N -2) + (N -3) + ... + 1
  - *N op max* = 10 + 9 + 8 + 7 + ...+ 1 = 55

A soma de uma seqüência de N números de razão 1 é :

$$\frac{N(N+1)}{2} = \frac{10(10+1)}{2} = 55$$

Análise da Complexidade de um Algoritmo

A soma de uma seqüência de N números de razão 1 é :

$$\frac{N(N+1)}{2} \rightarrow \frac{N^2 + N}{2}$$

Elemento preponderante da equação*

** para um valor de N suficientemente grande*

Análise da Complexidade de um Algoritmo

- Complexidade de **Pior Caso** do Algoritmo
  - É a função definida pelo número máximo de passos gastos sobre qualquer instância de tamanho N.
- Complexidade de **Melhor Caso** do Algoritmo
  - É a função definida pelo número mínimo de passos gastos sobre qualquer instância de tamanho N.
- Complexidade de **Caso Médio** do Algoritmo
  - É a função definida pelo número médio de passos gastos sobre qualquer instância de tamanho N.

Análise da Complexidade de um Algoritmo

- Complexidade do algoritmo no **melhor caso** é igual a **N**.
- Complexidade do algoritmo no **pior caso** é aproximada por $\mathbf{N^2}$.
- O importante é estudar o ***pior caso***...

Exemplo de Pseudocódigo

Algoritmo: ***arrayMax(A, n)***

Entrada*: um arranjo A com n>= 1 elementos inteiros.*

Saída*: o maior elemento em A.*

```
currentMax = A[0];
for   i=1   to   n-1  do
      if currentMax < A[i] then
            currentMax = A[i];
return currentMax
```

```
public class ArrayMaxProgram {

    /**
     * Encontra o maior elemento em um arranjo A de inteiros
     */

    static int arrayMax(int[] A, int n){
        int currentMax = A[0];        // executado uma vez
        for(int i=1; i< n; i++)       // executado uma, duas, três vezes, etc.
            if(currentMax < A[i])     // executado n-1 vezes
                currentMax = A[i];    // executado uma vez
        return currentMax;
    }

    public static void main(String[] args) {
        int[] num = {10, 15, 3, 5, 56, 107, 22, 16, 85};
        for(int i = 0; i < num.length; i++)
            System.out.print(" The maximum element is "
                                        + arrayMax(num, num.length));
    }
}
```

Contando operações primitivas

- No exemplo arrayMax:
  - Inicializar a variável *currentMax* corresponde a duas operações primitivas (indexar um arranjo e atribuir um valor a uma variável) e é executada apenas uma vez, quando o algoritmo é iniciado.
    - ✓Ela contribui em duas unidades para a contagem
  - No início do laço FOR, o contador i é inicializado com o valor 1 – corresponde a uma operação primitiva.
  - Antes de entrar no laço FOR, a condição i < n é verificado - corresponde a uma operação primitiva (comparar dois números).

- Como o contador i é inicializado em 0 e incrementado em 1 no final de cada iteração, a comparação i < n é executada n vezes
  - ✓Colabora com *n* unidades para a contagem.
- O corpo do laço FOR é executado n-1 vezes. A cada iteração, A[i] é comparado com *currentMax* (duas operações primitivas – indexação e comparação)
  - ✓O valor de A[i] pode ser atribuído a currentMax (duas operações primitivas – indexação e atribuição);
  - ✓O contador i é incrementado (duas operações – soma e atribuição).
  - ✓A cada iteração do laço, quatro a seis operações primitivas são realizadas – o corpo do laço contribui para a contagem com um número de unidades que varia entre 4(n-1) e 6(n-1)

- Retornar o valor da variável *currentMax* corresponde a uma operação primitiva e é executada apenas uma vez.

- O número **mínimo** de operações primitivas T(n) executadas pelo algoritmo arrayMax:

$$2 + 1 + n + 4(n - 1) + 1 = 5n$$

- O número **máximo** de operações primitivas T(n) executadas pelo algoritmo arrayMax:

$$2 + 1 + n + 6(n - 1) + 1 = 7n - 2$$

- Considere:
  - *"a"* tempo da op. primitiva mais rápida
  - *"b"* tempo da op. primitiva mais lenta
- Seja *T(n)* o tempo real de execução do pior caso do algoritmo

$$a(7n-1) <= T(n) <= b(7n-1)$$

- Ou seja, o tempo T(n) está limitado pelas duas funções lineares.

Análise da Complexidade de um Algoritmo

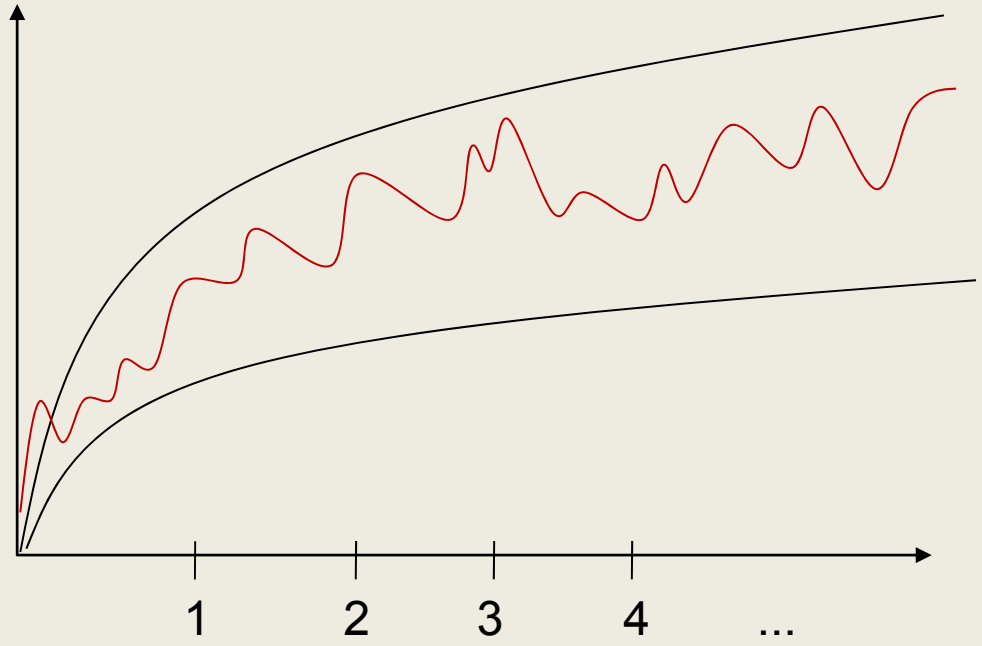
- Cada uma das complexidades de tempo (melhor e pior caso) define uma **função numérica**, representando tempo *versus* tamanho do problema.

Funções numéricas são difíceis de serem trabalhadas.

Para simplificar a análise da complexidade de um algoritmo utiliza-se a **Notação *O***

Notação O

- A Notação *O* permite fazer análise de complexidade de algoritmos considerando limites superiores e inferiores das funções numéricas.



- T(n) varia dentro da área definida pelo Melhor – Pior caso.
- Tempo cresce com o tamanho dos dados
- Não importa máquina nem implementação

Taxa de crescimento do tempo de execução

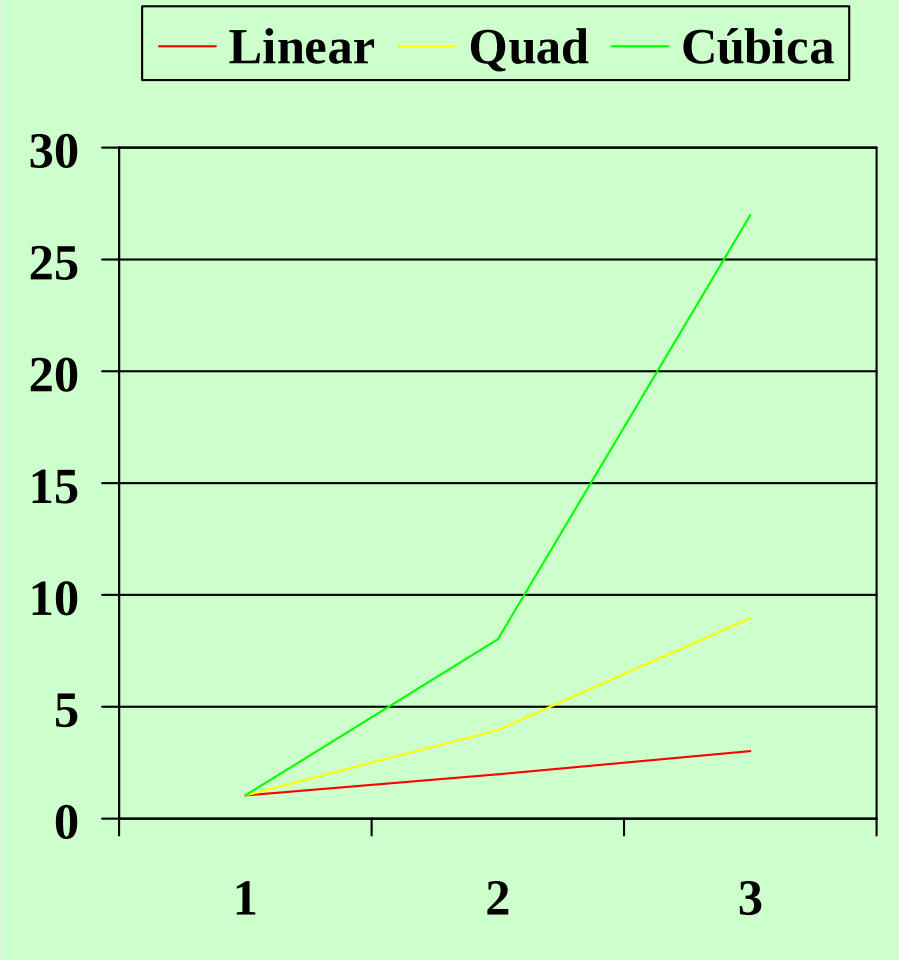
- Alterar o ambiente de hardware/software
  - Afeta *T(n)* com um fator constante
  - Mas não altera sua a taxa de crescimento

- Taxa de crescimento linear
  - propriedade intrínseca daquele algoritmo

Funções de taxas de crescimento

- Linear ~ n
- Quadrática ~ n^2
- Cúbica ~ n^3

- Taxa de crescimento não é afetada por
  - Fatores constantes
  - Termos de menores ordens



Classes Assintóticas

- Para fazer análise de complexidade de algoritmos, utiliza-se o comportamento assintótico da função do tempo “T” em relação ao tamanho da entrada do algoritmo “n”;
  - O comportamento assintótico de uma função f(n) pode ser encontrado nos casos em que

$$n \to \infty \quad e \quad n \to -\infty$$

Classes Assintóticas

- Os tipos de comportamento assintóticos são também chamados de **Classes Assintóticas**.
  - A definição do comportamento assintótico não define exatamente o número de operações, mas qual o comportamento esperado quando varia-se o tamanho da entrada.
  - Classes assintóticas são representadas pela notação ***O***.

Notação Big-O

- Dadas duas funções f(n) e g(n)
- Diz-se que

$$f(n) \text{ é } O(g(n))$$

- se

$$f(n) <= k*g(n), \text{ com } n >= n0$$

- Ou seja
  - se f é “menor” que g multiplicado por uma constante K, para um valor de n maior que um limite n0

- Quais funções estamos comparando?
  - f(n) e g(n) são algoritmos

- Para nosso uso:
  - f(n) é o algoritmo que estamos estudando
  - g(n) será uma função básica
    - ✓linear
    - ✓quadrática
    - ✓cúbica, etc

Classes Assintóticas

- Classes assintóticas:
  - Constante: $O(1)$
  - Logarítmica: $O(\lg(n))$ → representada na base 2
  - Linear: $O(n)$
  - Logarítmo-Linear: $O(n \lg(n))$
  - Quadrática: $O(n^2)$
  - Cúbica: $O(n^3)$
  - Exponencial: $O(c^n)$, onde c é uma constante

Notação Assintótica

- Notação Θ (*Theta*);
- Notação O (*O maiúsculo*);
- Notação Ω (*Ômega maiúsculo*);

Taxa de Crescimento Assintótica

- A taxa de crescimento assintótica é um modo de comparar funções, que ignora fatores constantes e entradas de tamanho pequeno.
  - **O(g(n))**: classes de funções f(n) que crescem menos rapidamente ou igual a g(n);
  - **Θ(g(n))**: classes de funções f(n) que crescem na mesma taxa de g(n);
  - **Ω (g(n))**: classes de funções f(n) que crescem pelo menos tão rapidamente quanto g(n).

Exemplo arrayMax

- O tempo de execução do algoritmo *arrayMax* para determinar o maior elemento de um arranjo de n inteiros é O(n).
  - O número de operações primitivas executadas pelo algoritmo é no máximo *7n-2*.
    - ✓Portanto, existe uma constante positiva **a** que depende da unidade de tempo do hardware e do software em que o algoritmo é implementado, compilado e executado, de tal forma que o tempo de execução de arrayMax para uma entrada de tamanho n é no **máximo *a(7n – 2)***.

Exemplo arrayMax

- Aplicamos a definição de O com c = 7a e $n_0 = 1$ e concluímos que o tempo de execução de arrayMax é **O(n)**

Notação Big-O

- f(n) é O(g(n)) significa que
  - a taxa de crescimento de f(n) não é maior que a taxa de crescimento de g(n)
- Então
  - se f(n) é O(n), quero dizer que
    - ✓ a taxa de crescimento de f(n) é menor ou igual a uma taxa de crescimento linear
  - se f(n) é $O(n^2)$, quero dizer que
    - ✓ a taxa de crescimento de f(n) é no máximo igual a uma taxa de crescimento quadrática

Regras do Big-O

- Se f(n) é um polinômio de grau d
  - f(n) é $O(n^d)$ ou seja
    - ✓joga-se fora termos de mais baixa ordem
    - ✓despreza-se os fatores constantes
- Use a menor classe de funções
  - 2n é O(n) em vez de 2n é $O(n^2)$
- Use a expressão de classe mais simples
  - 3n+5 é O(n) em vez de 3n+5 é O(3n)

Análise assintótica – como proceder

- Define o tempo de execução de um algoritmo
  - usando a notação Big-O
- Análise assintótica
  - encontra-se o pior caso com operações primitivas em função do tamanho da entrada
  - expressa-se esta função com notação Big-O
- Algoritmo arrayMax
  - pior caso: 7n-1 operações primitivas
  - arrayMax executa em tempo O(n)

Comparação entre algoritmos

- Faz-se a análise assintótica de cada algoritmo
- Compara-se diretamente os resultados
- Algumas funções típicas por ordem de crescimento:
  - $\sqrt{n} < \log n < n < n \log n < n^c < x^n < n! < n^n$
- Quanto mais rápido o crescimento, pior o desempenho do algoritmo

Eficiência Assintótica

- Maneira como o tempo de execução de um algoritmo aumenta com o tamanho da entrada, a medida que a entrada aumenta indefinidamente.

- Em geral:
  - Um algoritmo que é assintoticamente mais eficiente será a melhor escolha para todas as entradas (exceto as muito pequenas).

Análise dos Algoritmos de Algumas Estruturas

- **Pilha**
  - Criar – O(1)
  - Esvaziar – O(1)
  - Top – O(1)
  - Push – O(1)
  - Pop – O(1)

- **Fila**
  - Criar – O(1)
  - Entrar – O(1)
  - Sair – O(1)

- **Listas**
  - Criar – O(1)
  - Inserir – O(1)
  - Excluir – O(n)
  - Buscar – O(n)

- **Árvores**
  - Criar – O(1)
  - Inserir – O(h)
  - Percorrer – O(n)
  - Buscar – O(h)
  - Excluir – O(h)

Onde
n é o número de elementos
h é a altura da árvore

* Válido para árvores balanceadas

Referências Bibliográficas

- *Estruturas de Dados e Algoritmos em Java* – 2ª ed. Goodrich, Michael, T.; Tamassia, Roberto. Ed. Bookman.