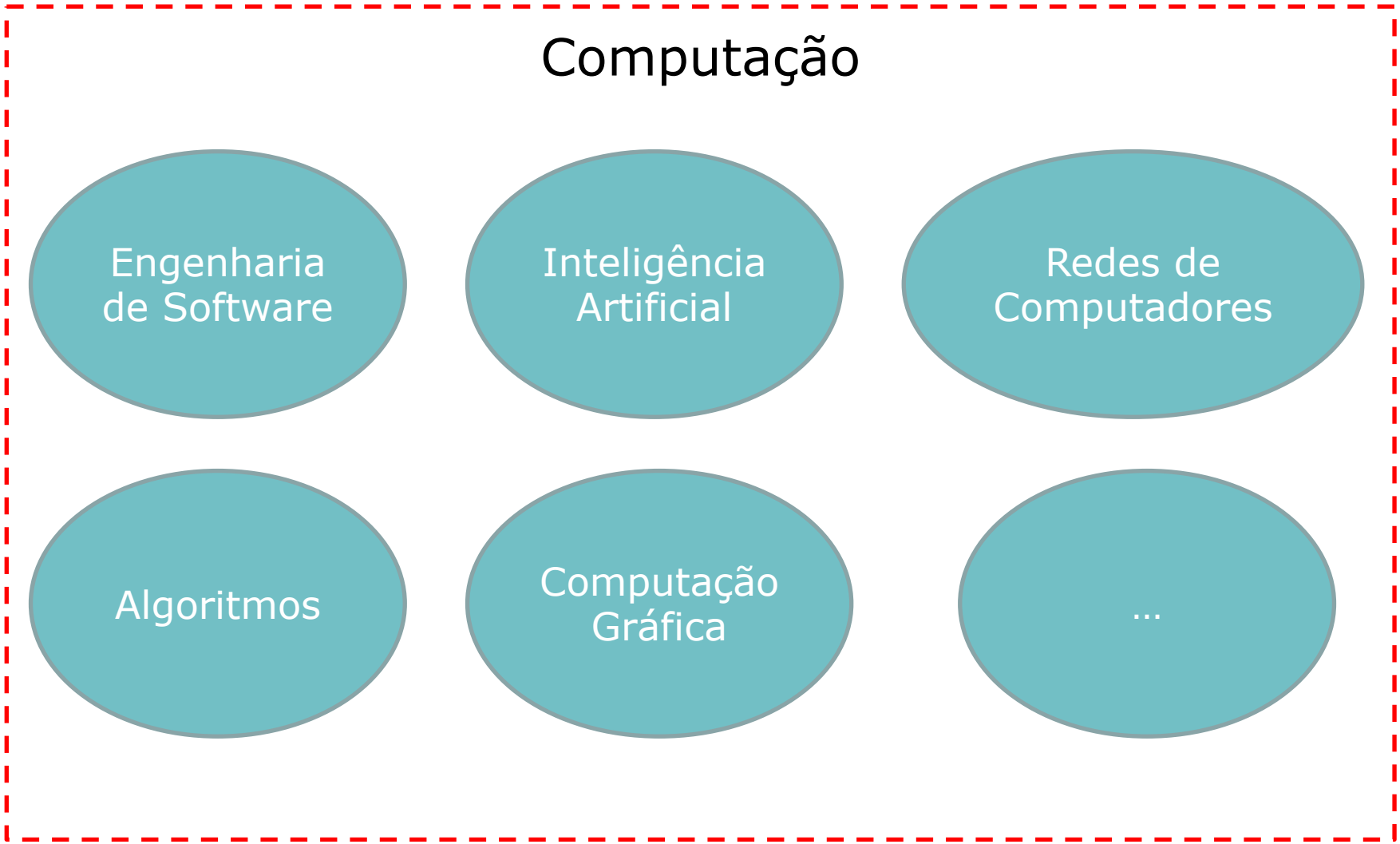
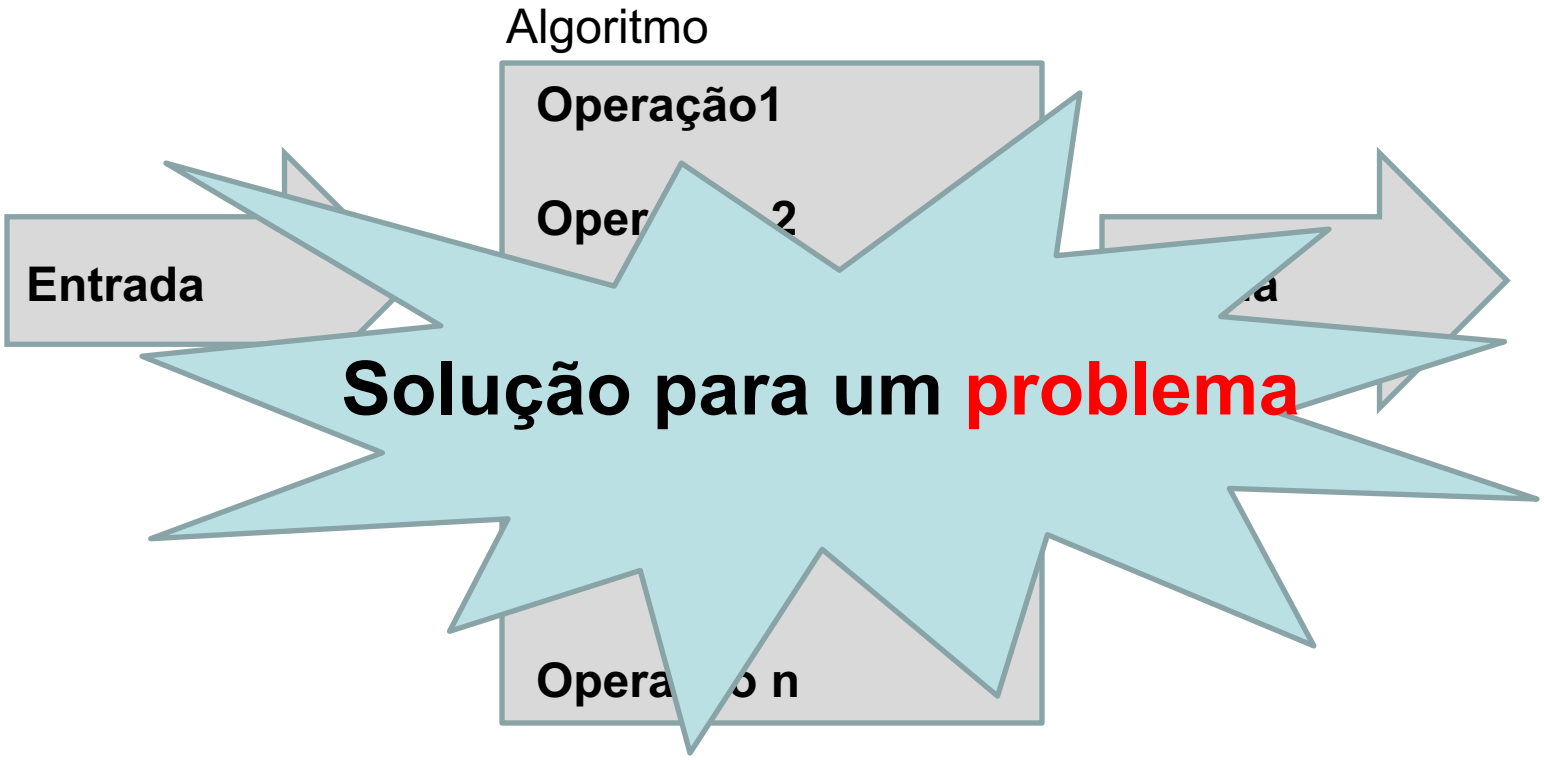
Estrutura de dados
Análise de Complexidade

Prof. Enyo José

Contexto


Computação
Engenharia de Software
Inteligência Artificial
Redes de Computadores
Algoritmos
Computação Gráfica
...

Algoritmo



Problema e instância

- Um **problema** é definido por:
  - Uma descrição dos parâmetros
  - Uma descrição das propriedades que a resposta deve satisfazer

- Uma instância para um problema fixa valores para todos parâmetros

Problema e instância

- Exemplo de problema:
  - Ordenar um conjunto de números em ordem crescente
  - **Entrada**: $a_1, a_2, \ldots, a_n$
  - **Saída**: $a'_1, a'_2, \ldots, a'_n$
  - **Propriedade**: A sequencia de saída é uma permutação dos valores de entrada tal que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

- Instâncias para este problema: 10 20 5 40 33 1 88

  15 12 3 77

Problema x Algoritmo/programa

Problema

Algoritmo/ Programa

Somar dois números inteiros
Entrada: n1 e n2
Saída: s
Propriedade: s é o resultado da soma de n1 e n2

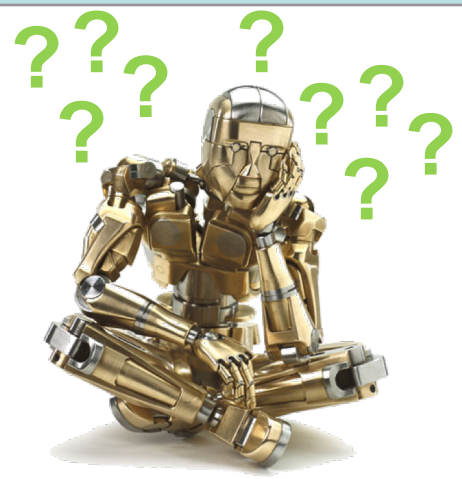
```
int soma (int x, int y){
   int soma = x + y;
   return soma;
}
```

```
int soma(int x, int y){
    return(x+y);
}
```

...

Algoritmo e Programa

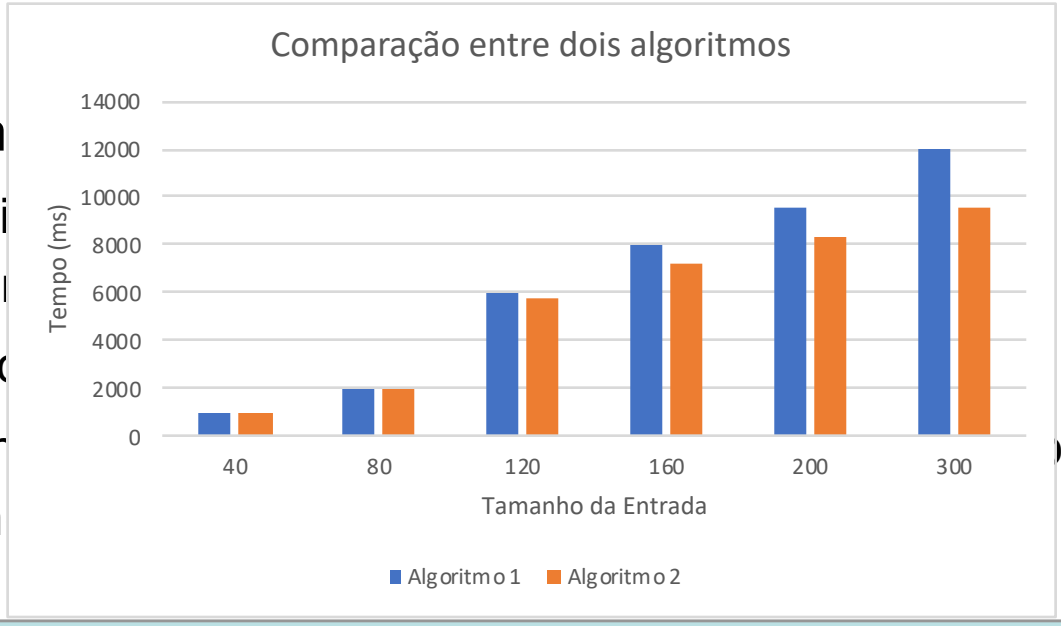
- Qual melhor solução?



- Vários critérios:
  - Facilidade de uso;
  - Documentação;
  - Velocidade de Execução e Tempo de resposta;

Como comparar dois algoritmos?

- Método empírico (Experimental)
  - Executar os programas e verificar o tempo utilizado para obter resposta para diferentes tamanhos da entrada
  - Pode variar com base em variáveis como computador, linguagem de programação, compilador...
  - Depende de ferramenta específica
  - Exemplo



- Métodos a
  - Determi que possa represe
  - Exempl
  - Indeper rogramação, compila

Análise de Algoritmos

- Auxilia na escolha do melhor algoritmo para um problema;
- Utiliza como base a **complexidade computacional**:
- Descritas por funções que tem como parâmetro o **tamanho da entrada**;

Vetor de 6 posições 

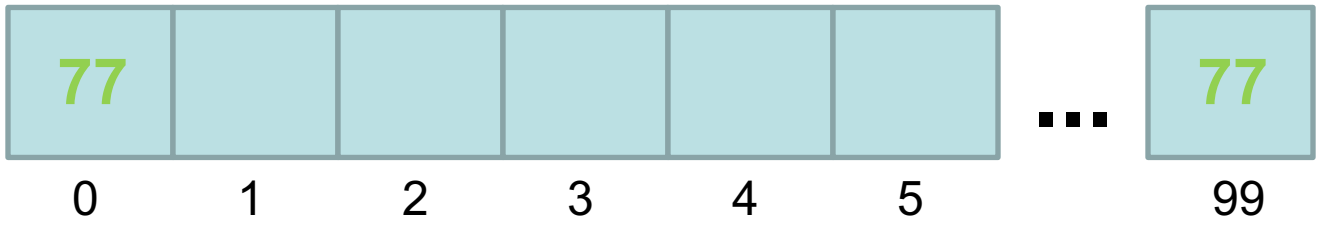
- Utiliza conceitos e modelos matemáticos;

Análise de Algoritmos

Complexidade do algoritmo = Trabalho = Número de operações efetuadas

Depende da entrada...

Problema: Encontrar um dado elemento em um vetor de n posições.
Algoritmo: Percorrer o vetor por cada posição até encontrar o elemento desejado, tendo como entrada o vetor e o elemento a ser procurado.
Situação 1: Entrada: vetor de 100 posições e elemento a ser procurado = **77**



Análise de Algoritmos

- Podemos observar que duas situações foram consideradas:
  - Um otimista, no qual o elemento é encontrado na 1ª posição. Este cenário é o ideal, pois representa o **menor** tempo de execução ou **limite inferior**;
  - Um cenário no qual o elemento encontra-se na ultima posição ou não existe foi demonstrado. Trata-se do **limite superior**, cenário que representa o **maior** tempo de execução;
  - Adicionalmente, o elemento poderia estar em qualquer uma das posições. Para esta situação é levada em consideração a média de todas as situações. Chamamos de **caso médio**;

Complexidade de Algoritmos

- Existem três escalas de complexidade:
  - Melhor Caso
  - Caso Médio
  - Pior Caso

- Nas três escalas, a função f(N) retorna a complexidade de um algoritmo com entrada de N elementos.

Complexidade de Algoritmos – Melhor Caso

- Definido pela letra grega Ω (Ômega)
- É o menor tempo de execução em uma entrada de tamanho N;
- É pouco usado, por ter aplicação em poucos casos.
- Ex.:
  - Se tivermos uma lista de N números e quisermos encontrar algum deles assume-se que a complexidade no melhor caso é f(N) = Ω (1), pois assume-se que o número estaria logo na cabeça da lista.

Complexidade de Algoritmos – Caso Médio

- Definido pela letra grega θ (Theta)
- Dos três, é o mais difícil de se determinar
- Deve-se obter a média dos tempos de execução de todas as entradas de tamanho N, ou baseado em probabilidade de determinada condição ocorrer
- No exemplo anterior:
  - A complexidade média é P(1) + P(2) + ... + P(N)
  - Para calcular a complexidade média, basta conhecer as probabilidades de P*i;*
  - P*i* = 1/N, 1 <= i <= N
  - Isso resulta em P(1/N) + P(2/N) + ... + P(N/N)
  - Que resulta em 1/N(1+2+...+N)
  - Que resulta em $\frac{1}{N}\left[\frac{N(N+1)}{2}\right]$
  - Que resulta em $f(N) = \theta\left[\frac{(N+1)}{2}\right]$

Complexidade de Algoritmos – Pior Caso

- Será o caso utilizado durante esse curso
- Representado pela letra grega ***O*** (O maiúsculo. Trata-se da letra grega ômicron maiúscula)
- É o método mais fácil de se obter. Baseia-se no maior tempo de execução sobre todas as entradas de tamanho N
- Ex.:
  - Se tivermos uma lista de N números e quisermos encontrar algum deles, assume-se que a complexidade no pior caso é O (N), pois assume-se que o número estaria, no pior caso, no final da lista. Outros casos adiante

Cálculo da Complexidade de Algoritmos (Análise dos passos)

- A complexidade de um algoritmo pode ser determinada a partir da complexidade de suas operações.
- De modo que algumas estruturas:
  - **Sequência** (De um modo geral possui peso 1) – Comando que é executado e o controle passa para o próximo comando; Ex: i:= 1;
  - **Seleção** (Possui peso 1) – Comando que ao ser executado permite desvios. Ex: if x = 10 then
  - Considerando-se o fragmento de código abaixo:

```
se cond então
   expresssão1
senão
   expressão2
fim se
```

  - o tempo de execução de um comando IF/THEN/ELSE nunca é maior do que o tempo de execução do teste condicional em si mais o tempo de execução da maior dentre as expressões expressão1 e expressão2.
  - Ou seja: se expressão1 é O(n3) e expressão2 é O(n), então o teste é O(n3) + 1 = O(n3).

Cálculo da Complexidade de Algoritmos (Análise dos passos)

- **Repetição** (Depende da quantidade de repetições) – O tempo de execução de um laço é no máximo o tempo de execução das instruções dentro do laço (incluindo os testes) vezes o número de iterações;
- **Aninhamento de Laços** – Analisar os mais internos. O tempo total de execução de uma instrução dentro de um grupo de laços aninhados é o tempo de execução da instrução multiplicado pelo produto dos tamanhos de todos os laços.

- **Chamada de Funções** – A análise é feita como no caso de laços aninhados. Para calcular a complexidade de um programa com várias funções, determina-se primeiro a complexidade de cada uma das funções. Desta forma, na análise, cada uma das funções é vista como uma instrução com a complexidade que foi calculada.

Exemplo 1 – Algoritmo de inversão de uma sequência

- Inverter os dados dentro de um vetor
  - **Entrada**: vetor (a_1, a_2, ..., a_n) e tamanho do vetor (n)
  - **Saída**: vetor (a'_1, a'_2, ..., a'_n)
  - **Propriedade**: A sequencia de saída é uma permutação dos valores de entrada tal que $a'_1 = a_n$, $a'_2 = a_{n-1}$, ..., $a'_n = a_1$

```
temp := 0                                  1
para i=1,...,  n/2  faça                   n/2
        temp := S[ i ]                     1
        S[ i ] := S[ n-i+1 ]               1
        S[ n-i+1 ] := temp                 1
```

$3n/2 + 1$

Simplificações

- Considerando que os algoritmos trabalharão com uma grande quantidade de dados (Tamanho grande)
  - Podemos simplificar as expressões, removendo partes da expressão menores que a maior parte como constantes multiplicativas ou aditivas

    Ex: $3n^2 + 2n + 1 \quad \sim> \quad n^2$

- Partindo deste princípio, expressões como $3n^2+5n+7$ e $12n^2+11n+32$ podem ser consideradas equivalentes

- Assim, podemos simplificar a análise do algoritmo, identificando a operação dominante (mais complexa)
  - Isso pode evitar a análise linha-por-linha do algoritmo

Complexidade de Algoritmos

- A partir da análise destas funções, os algoritmos podem ser classificados quanto a ordem:
  1. Complexidade Constante
  2. Complexidade Linear
  3. Complexidade Logarítmica
  4. *NlogN*
  5. Complexidade Quadrática
  6. Complexidade Cúbica
  7. Complexidade Exponencial

Complexidade Constante

- São os algoritmos de complexidade O(k), onde k é a quantidade de operações;
- Independe do tamanho N de entradas;
- É o único em que as instruções dos algoritmos são executadas num tamanho fixo de vezes;
- Ex.:

```
bool vazia(int a[], int n){
        bool vazia = a[0] == a[n-1];
        return vazia;
}
```

Complexidade Linear

- São os algoritmos de complexidade ***O***(n);
- Uma operação é realizada em cada elemento de entrada,
- Ex.: pesquisa de elementos em uma lista

```
int busca(int a[], int n, int x){
        int i = 1;
        int pos = -1;
        while(a[i] != x){
                i = i+1;
                if (i >= n){
                        pos = -1;
                }else{
                        pos = i;
                }
         }
          return pos;
}
```

Complexidade Logarítmica

- São os algoritmos de complexidade ***O***(logn);
- Ocorre tipicamente em algoritmos que dividem o problema em problemas menores;
- Ex.: O algoritmo de Busca Binária;

Algoritmo 1.5 Algoritmo de busca binária.

Chamada: BUSCABIN(p, r, S, x)
Entrada: índices p e r, conjunto S e valor x
Saída: se $x \in S$, índice de x em S; senão, valor -1
Requisito: o conjunto S deve estar ordenado em ordem crescente, ou seja, $S[i] < S[i+1]$, para todo $i \in \{p, p+1, \cdots, r-1\}$

```
1: se p ≤ r então
2:     b ← ⌊(p + r)/2⌋
3:     se S[b] = x então
4:         retorna b
5:     se S[b] > x então
6:         retorna BUSCABIN(p, b − 1, S, x)
7:     retorna BUSCABIN(b + 1, r, S, x)
8: retorna −1
```

Complexidade NlogN

- Como o próprio nome diz, são algoritmos que têm complexidade O(NlogN)
- Ocorre tipicamente em algoritmos que dividem o problema em problemas menores, porém juntando posteriormente a solução dos problemas menores

A maioria dos algoritmos de ordenação externa são de complexidade *logarítmica* ou *N Log N*

Complexidade Quadrática

- São os algoritmos de complexidade $O(N^2)$;
- Itens são processados aos pares, geralmente com um *loop* dentro do outro;
- Ex. neste exemplo Mat1, Mat2 e MatRes são matrizes de tamanho nxn:

```
for(int i=0; i< n; i++){
    for(int j=0; j< n; j++){
        MatRes[i][j] = Mat1[i][j] + Mat2[i][j];
    }
}
```

Complexidade Cúbica

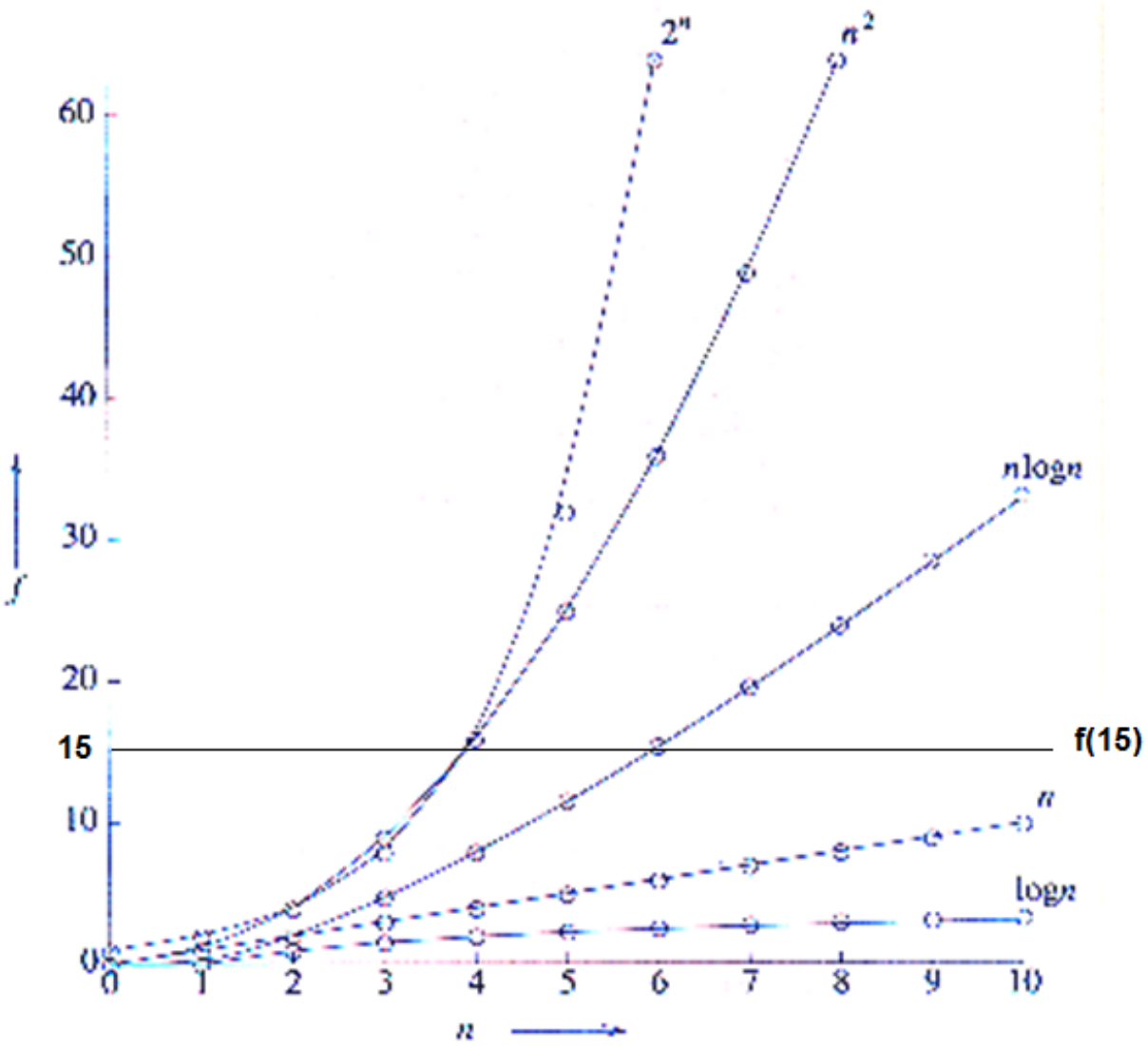
- São os algoritmos de complexidade $O(n^3)$
- Itens são processados três a três, geralmente com um *loop* dentro do outros dois
- Ex. neste exemplo Mat é uma matriz de tamanho nxn e vet é um vetor de tamanho n:

```
for(int i=0; i< n; i++){
    for(int j=0; j< n; j++){
            for(int k = 0; k < n; k++){
                    Mat[i][j] = Mat[i][j] + vet[k];
            }
    }
}
```

Complexidade Exponencial

- São os algoritmos de complexidade $O(2^N)$;
- Utilização de "Força Bruta" para resolvê-los (abordagem simples para resolver um determinado problema, geralmente baseada diretamente no enunciado do problema e nas definições dos conceitos envolvidos);
- Geralmente não são úteis sob o ponto de vista prático;

Comportamento das Ordens



Tempo de execução

Complexidade	n=5	n=10	n=50	n=100	n=1000	n=1000
O ($\log_2 n$)	0.0000023	0.0000033	0.0000056	0.0000066	0.0000099	0.000013
O (n)	0.000005	0.000010	0.000050	0.000100	0.001000	0.010000
O ($n.\log_2 n$)	0.000011	0.000033	0.000283	0.000684	0.009966	0.132877
O (n^2)	0.000025	0.000100	0.002500	0.010000	1s	1min 40s
O (n^3)	0.000125	0.001000	0.125000	1s	16min 40s	11dias 14h
O (n^5)	0.003125	0.100000	5min 13s	2h 47min	32 anos	∞
O (2^n)	0.000032	0.001024	35 anos	$4x10^{16}$ anos	∞	∞
O (3^n)	0.000243	0.059049	$2x10^{10}$ anos	∞	∞	∞

* Tabela extraída de Viana, Gerardo Valdísio Rodrigues,**Metaheurísticas e programação paralela em otimização combinatória**, Edições UFC, 1998.

Exemplo 2

```
para I de 1 até N faça
        S[I] ← X[I] + Y[I]
Fim para
```

- Complexidade n.

Exemplo 3

```
int Maximo (int v[], int n){
    int i, n;
    int max;
     if n = 0 error (“ Vetor vazio ”)
     else {
     max = v[0];
     for (i = 0; i < n; i++)
           if v[i] > max then max = v[i];
           }
     return max;
}
```

Exemplo 4

```
for (i=1; i<=n ; i++)
    for (j=1; j<=i ; j++)
        trecho com O(1)
```

- O laço interno é executado 1+2+3+...+n-1+n = n(n+1)/2.
- ***O(n*(n+1)/2)***, ou seja, ***O(0,5(n²+n))***, ou seja, ***O(n²)***

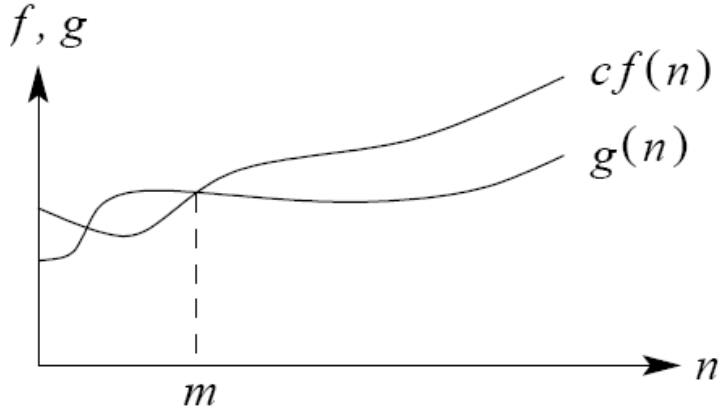
Comparação entre Algoritmos

- Para um problema podem existir vários algoritmos;
  - Como decidir qual melhor algoritmo?
  - A **complexidade dos algoritmos** pode ser utilizada como critério para esta decisão;

- Para determinarmos a complexidade de um algoritmo, as **operações** fundamentais do mesmo são analisadas e uma **função custo** é criada com base nestas **operações**.

- A partir desta função é possível traçar gráficos para representar o seu comportamento.

Comparação entre Algoritmos

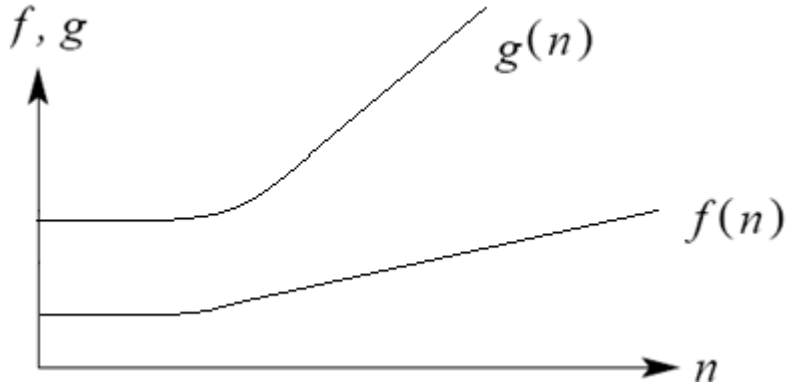


- E através destes gráficos é possível fazer uma comparação dos algoritmos através do **comportamento assintótico**.
- Comportamento assintótico = Estudo do crescimento das funções;

Comparação de Algoritmos

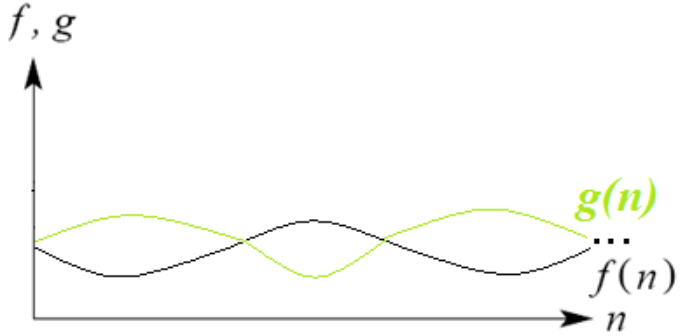
- O algoritmo mais **eficiente** é aquele que é melhor para as entradas;
- Uma função **domina assintoticamente** outra quando cresce mais rapidamente que a outra;
  - No gráfico uma função domina assintoticamente outra quando sua curva encontra-se acima da curva da outra função.

- Alguns casos podem ser observados:
  1. f sempre é inferior a g. Neste caso, o algoritmo f é melhor.



Comparação de Algoritmos

2. Às vezes f é superior e às vezes g é superior e os gráficos se interceptam em um número infinito de pontos. Empate



3. Se os gráficos se interceptam uma quantidade finita de vezes, o melhor é aquele que menor que o outro para valores grandes de n;