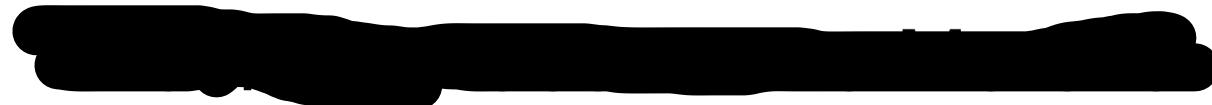


Comportamento Assintótico de Funções



Adaptado dos slides da
Prof^a. Renata Oliveira

A notação “O”

- A ordem de crescimento do tempo de execução de um algoritmo nos dá uma caracterização da sua eficiência e nos permite comparar o desempenho relativo com outros algoritmos.
- Quando nos concentramos em tamanhos grandes de entrada, que torna relevante apenas a ordem de crescimento do tempo de execução do algoritmo, estamos estudando a **EFICIÊNCIA ASSINTÓTICA** do mesmo, ou seja, estamos preocupados em como o tempo de execução de um algoritmo cresce com o tamanho da entrada crescendo infinitamente.

Ou ainda: Qual o comportamento do algoritmo para entradas cada vez maiores?

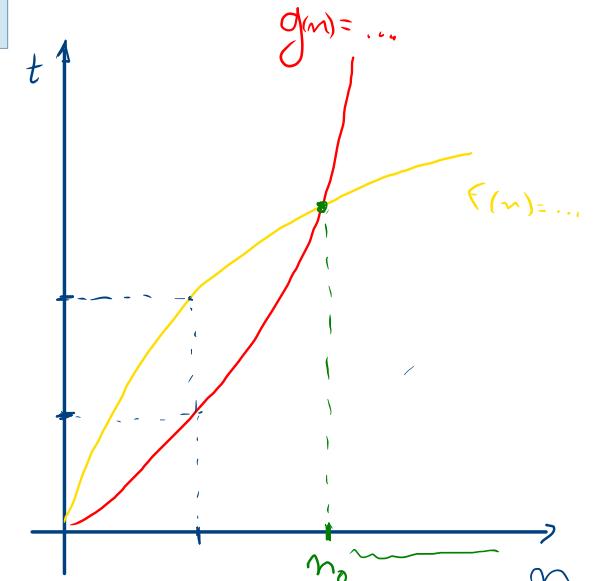
A notação “O”

- A ordem de magnitude de um algoritmo é dada pela soma das frequências de todas as suas declarações. Ela pode ser extraída diretamente do algoritmo, independente da máquina ou linguagem.
- Existe uma notação matemática adequada para tratar da ordem de magnitude. É chamada de notação O (“O” grande ou Big “O”), e foi sugerida por Knuth.

Definição

- Uma função $g(n)$ domina assintoticamente outra função, $f(n)$, se existem constantes c e n_0 (positivas) tais que:

$$\forall n \geq n_0 \quad |f(n)| \leq c |g(n)|$$



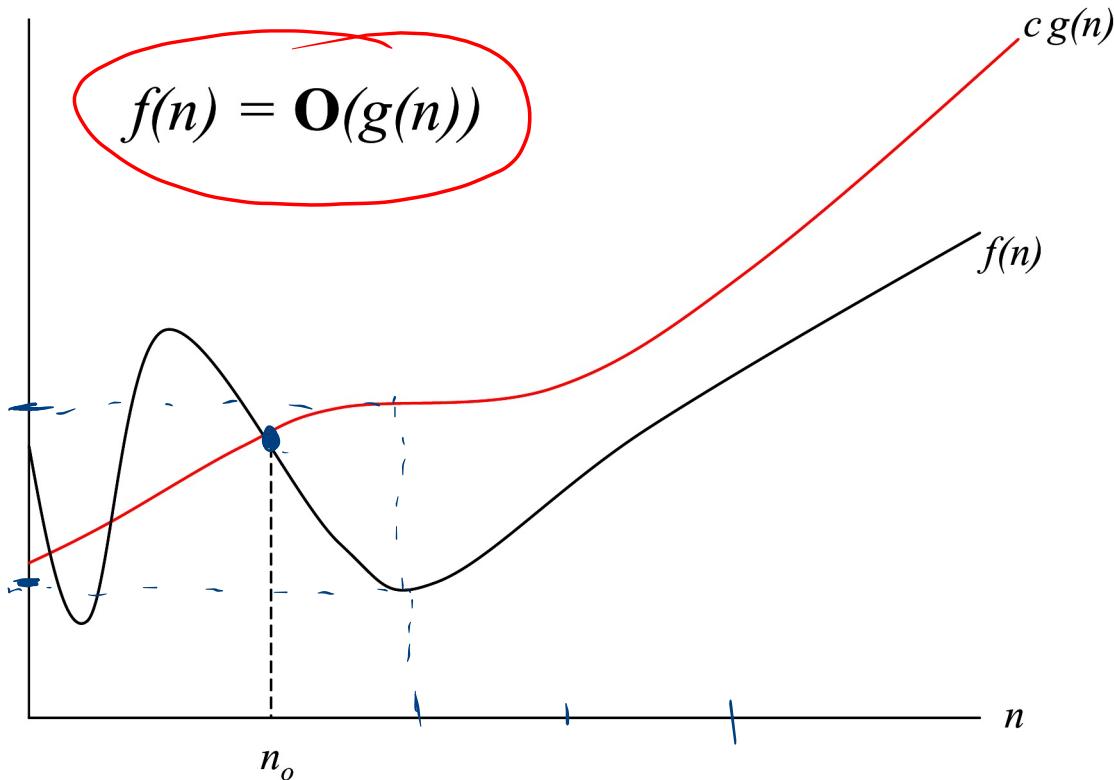
- Neste caso dizemos que $f(n)$ é $O(g(n))$ e escrevemos $f(n) = O(g(n))$

$$|f(n)| \leq c \cdot |g(n)|$$

$$f(n) \in O(g(n))$$

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

Definição



Dizemos que $g(n)$ domina assintoticamente $f(n)$, ou que $g(n)$ é o limite assintótico superior para $f(n)$

Exemplo

- Suponha que $f(n) = (3/2)n^2 + (7/2)n - 4$ e que $g(n) = n^2$. A tabela abaixo sugere que $f(n) \leq 2 g(n)$ para $n \geq 6$ e, portanto, $f(n) = O(g(n))$

n	f(n)	g(n)
0	4	0
1	1	1
2	9	4
3	20	9
4	34	16
5	51	25
6	71	36 *
7	94	49 ↓
8	120	64

$$F(m) = \frac{3}{2}m^2 + \frac{7}{2}m - 4$$

$$g(m) = m^2$$

$$\underline{C=2}$$

$$\boxed{|f(n)| \leq 2 |g(n)| \\ m \geq 6}$$

Exemplo

- Mostre, usando a definição, que $3n^2+n$ é $O(n^2)$

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

$$g(n) = n^2$$

$|f(n)| \leq c |g(n)|$
 $f(n) \in O(g(n))$

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

$$|f(n)| \leq c |g(n)|$$

$$3n^2 + n \leq c \cdot n^2$$

Supondo $n=1$ e $c=4$

OK!

$$3 \cdot (1)^2 + 1 = 4 \leq 4 = 4 \cdot (1)^2$$

Hipótese: $n = K$

$$3K^2 + K \leq 4K^2$$

Passo Indutivo: $n = K+L$

$$3(K+L)^2 + (K+L) \leq 4(K+L)^2$$

aplicar a
hip.

$$\begin{aligned} 3(K+L)^2 + (K+L) &= 3K^2 + 6K + 3 + K + L = (3K^2 + K) + (6K + 4) \leq 4K^2 + 6K + 4 \dots \\ \dots &\leq 4K^2 + 8K + 4 = 4(K+L)^2 \end{aligned}$$

$$n^2 \in O((n+1)^2)$$

$$F(n) \in O(g(n))$$

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

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

$$|n^2| \leq c |(n+1)^2|$$

TAREFA

$$2m^2 + 3 \in O(m^2)$$

$$m^2 \in O(2m^2 + 3)$$

Análise de Algoritmos

- A análise de um algoritmo conta apenas algumas operações “elementares” (por exemplo, comparações, atribuições, multiplicações, acessos ao disco, etc).
- A medida de custo ou medida de complexidade relata o crescimento assintótico da operação considerada.
- Se f é uma função de complexidade para um algoritmo F , então $\underline{O}(f)$ é considerada a complexidade assintótica ou o comportamento assintótico do algoritmo F.
- A relação de dominação assintótica permite comparar funções de complexidade.

Teorema

- Se $A(n) = a_m n^m + \dots + a_1 n + a_0$ é um polinômio de grau m , então $A(n) = O(n^m)$.

$$A = a_m n^m + b \cdot n^2 + c \cdot n + d$$

$$A = \underline{O(n^m)}$$

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

n	$2n^2$	n	1	$2n^2 + n + 1$
1	2	1	1	4
2	8	2	1	11
3	18	3	1	22
4	32	4	1	37
⋮	⋮	⋮	⋮	⋮
10	200	10	1	211

$$O(n^2)$$

Exemplo:

- A função de complexidade de um algoritmo de ordenação A é dada por

$$f(n) = \frac{n^3 + 2n^2 + n + 1}{5} = \frac{n^3}{5} + \frac{2n^2}{5} + \frac{n}{5} + \frac{1}{5}$$

- Logo, dizemos que este algoritmo tem complexidade $O(n^3)$.
- Isto quer dizer que para valores muito grandes de n poderíamos desconsiderar a outra parte do polinômio ($2n^2 + n + 1$) pois ela não influenciaria tanto no crescimento de $f(n)$.

Importante:

- Um algoritmo é $O(g(n))$ significa que se ele está rodando em um computador com valores crescentes de n , os tempos de execução resultantes serão sempre menores que $|g(n)|$.
- Quando tentamos determinar a ordem de complexidade de $f(n)$, estamos procurando pela menor $g(n)$ tal que $f(n) = O(g(n))$.

$F(n)$ é $O(g(n))$

$|F(n)| \leq C * |g(n)|$

Propriedades:

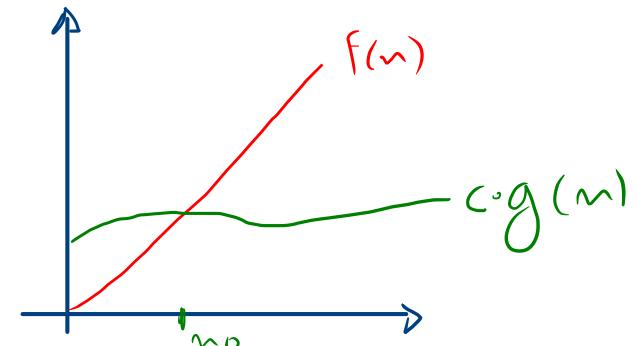
- $f(n) = O(f(n))$ $f(n) = n^2$
 $n^2 = O(n^2)$
- $c \cdot O(f(n)) = O(f(n))$ $c = \text{constante}$ $2 \cdot O(f(n)) = O(f(n))$
- $O(f(n) + O(f(n))) = O(f(n))$ $O(n^2 + O(n^2)) = O(n^2)$
- $O(O(f(n))) = O(f(n))$
- $O(f(n) + g(n)) = O(\max(f(n), g(n)))$ $O(n^3 + n^2) = O(n^3)$
- $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$ $O(n^3) \cdot O(n^2) = O(n^3 \cdot n^2)$
- $O(f(n) \cdot g(n)) = f(n) \cdot O(g(n)) = g(n) \cdot O(f(n))$

Outras notações assintóticas

- Notação Omega (Ω):

Limite assintótico inferior $f(n) = \Omega(g(n))$ se existirem constantes positivas c e n_0 tais que:

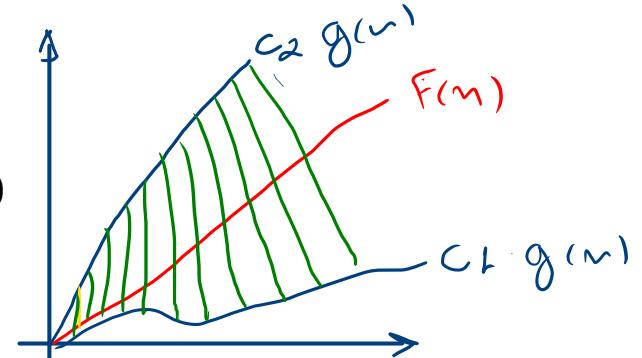
$$|f(n)| \geq c |g(n)|, \quad \forall n \geq n_0$$



- Notação Teta (Θ):

$f(n) = \Theta(g(n))$ se existirem constantes positivas c_1 , c_2 , n_0 tais que:

$$c_1 |g(n)| \leq |f(n)| \leq c_2 |g(n)|, \quad \forall n \geq n_0$$



Propriedades das funções O, θ, Ω

- Considere $\# = O, \theta, \Omega$
 - $f(n) = \#(g(n))$ e $g(n) = \#(h(n)) \Rightarrow f(n) = \#(h(n))$
 - $f(n) = \#(f(n))$
 - $f(n) = \theta(g(n))$ se e somente se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$
- Resultados Importantes:
 - $f(n)$ é $O(g(n)) \Rightarrow 0 \leq \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$
 - $f(n)$ é $\Omega(g(n)) \Rightarrow 0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq \infty$
 - $f(n)$ é $\theta(g(n)) \Rightarrow 0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$

Exercícios

1. Mostre, por definição, que :

a) \sqrt{n} é $O(n^2)$

b) n^2 é $O((n+1)^2)$

2. João desenvolveu um algoritmo cuja função de complexidade é $f(n) = 2n^2 + 3$. Mostre, por definição, que $f(n)$ é $O(n^2)$.

$$f(n) \in O(n^2)$$

$$|2n^2 + 3| \leq C * n^2$$

Exercícios

- José, na análise de quatro algoritmos para criptografia de dados, A_1 , A_2 , A_3 e A_4 encontrou as seguintes expressões para o número de operações matemáticas realizadas:

$$A_1: f(n) = \log n^{10}$$

$$A_2: f(n) = 2n$$

$$A_3: f(n) = 10n$$

$$A_4: f(n) = \sqrt[4]{n}$$

Ajude-o a demonstrar o seguinte

- A_1 é $\Theta(\log n)$
- A_2 é $\Omega(n^m)$ para $m > 1$
- A_3 é $O(n \log n)$
- A_4 é $\Omega(\ln n)$

Classes de comportamento assintótico

- $f = O(1)$ complexidade constante

- O uso do algoritmo independe de n .

- $O(c)$ para $c \in \mathbb{R}$ é $O(1)$

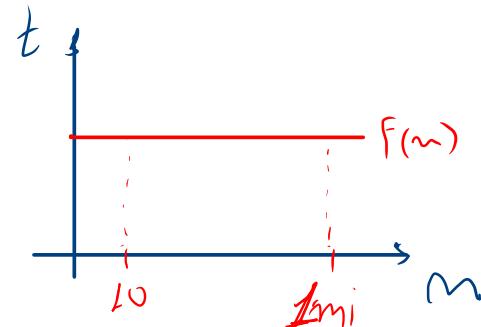
- Comandos de atribuição, leitura, escrita.

- $f = O(\log(n))$ complexidade logarítmica

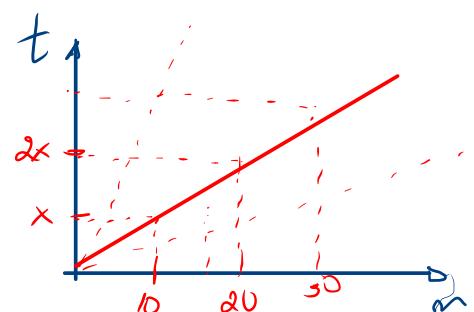
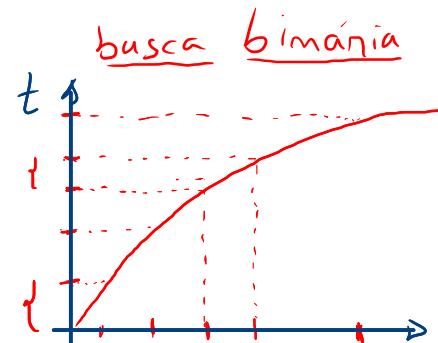
- O custo de aplicar o algoritmo a problemas com n suficientemente grande pode ser limitado por uma função do tipo $K \cdot \log n$ $K \in \mathbb{R}$.

- $f = O(n)$ complexidade linear

- O custo do algoritmo cresce linearmente com o tamanho da entrada.



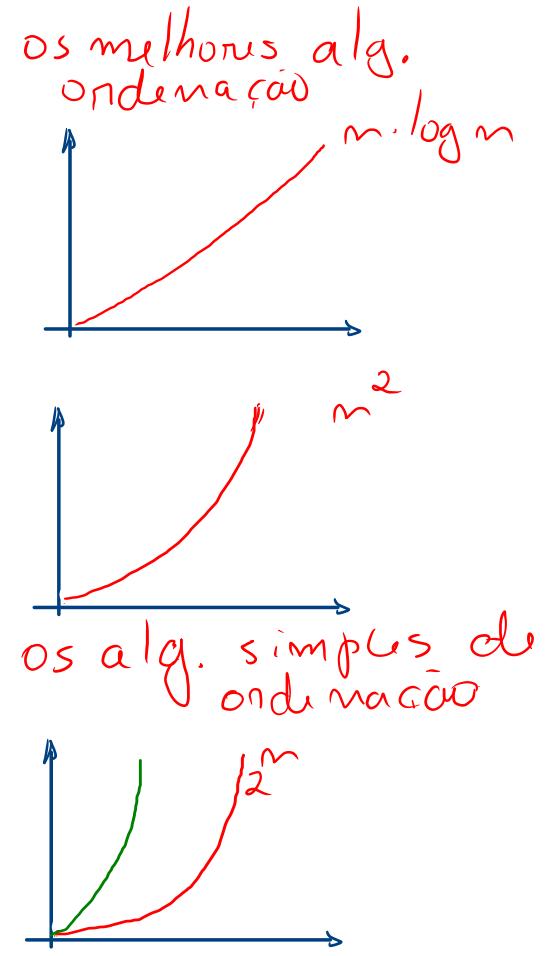
b.seq = 6 comp
b.bim = 2 comp
(10)



Classes de comportamento assintótico

- $f = O(n \log n)$
- $f = O(n^2)$ complexidade quadrática
- $f = O(n^3)$ complexidade cúbica
- $f = O(cn)^c$, constante, complexidade exponencial
simplex
- $f = O(n!)$ complexidade fatorial

$O(1) < O(\log \log n) < O(\log n) < O(n^{1/10}) < O(\sqrt{n}) <$
 $O(n) < \dots < O(n \log n) < O(n^2) < O(n^5) < O(2^n) <$
 $O(n!)$



Análise de algoritmos: Alguns princípios

1. O tempo de execução de um comando de atribuição de leitura e ou de escrita pode ser considerado O(1).
2. O tempo de execução de uma SEQUÊNCIA de comandos é determinado pelo maior tempo de execução de qualquer comando da sequência
3. O tempo de execução de um comando de DECISÃO é composto pelo tempo dos comandos executados dentro da condição, mais o tempo para avaliar a condição.
4. O tempo para executar um laço é a soma do tempo de execução do corpo do laço mais o tempo de avaliar a condição de parada, multiplicado pelo número de iterações do laço.

$$F(n) = n + \log n + 10 \\ = O(n)$$

$$\begin{array}{l} c_1 : O(\log n) \\ c_2 : O(3) \\ c_3 : O(n) * \\ c_4 : O(5) \\ c_5 : O(2) \end{array} \left\{ \begin{array}{l} O(n) \\ \hline \end{array} \right.$$

$$\begin{array}{l} \text{if } (<\text{condição}>) O(x) \\ \quad \quad \quad \left\{ \begin{array}{l} O(x) \\ \text{else} \\ \quad \quad \quad \left\{ \begin{array}{l} O(x) \\ \end{array} \right. \end{array} \right. \\ \end{array}$$

$$\begin{array}{l} \text{while } (<\text{condição}>) \{ \\ \quad \quad \quad \left\{ \begin{array}{l} O(1) \\ \end{array} \right. \\ \quad \quad \quad \left\{ \begin{array}{l} \text{m vezes} \\ \sum_{i=1}^m O(1) = O(1) + O(1) + \dots \\ \end{array} \right. \end{array} \right\} O(m)$$

$$\begin{array}{l} \text{for } (L \rightarrow m) \{ \\ \quad \quad \quad \left\{ \begin{array}{l} O(\log m) \\ \end{array} \right. \\ \quad \quad \quad \left\{ \begin{array}{l} O(m \cdot \log m) \\ \end{array} \right. \\ \} \end{array} \right\} O(m \cdot \log m)$$

Análise de algoritmos: Alguns princípios

5. PROGRAMAS COM PROCEDIMENTOS NÃO RECURSIVOS

- Cada procedimento é tratado separadamente, iniciando pelos procedimentos que não chamam outros procedimentos.
- A seguir, computar o tempo dos procedimentos que chamam os procedimentos que não chamam outros procedimentos, utilizando os tempos já avaliados.
- Repetir o processo até chegar no procedimento principal.

```
main(...){  
    : } O(6)  
    Função1(), O(n)  
    : } O(log n)  
    Função2(), O(n)  
    : } O(n)  
}
```

$$f(n) = O(10) + O(n) +
O(\log n) + O(10)
+ O(n)$$

$$\begin{aligned} f(n) &= O(n) \\ &= 2n + \log n \\ &= 3n + \log n \\ &= \frac{5}{2}n + 6\log n + 20 \end{aligned}$$

Exemplo 1: Encontrar o maior e o menor elemento de um vetor

```
void maxmin(A:vetor;)
{
    int i;          O(1)
    max = A[0];    O(1)
    min = A[0];    O(1)
    for (i = 1; i < n;i++)
    {
        if (A[i] > max) max = A[i];
        if (A[i] < min) min = A[i];
    }
}
```

1
 2
 3
 4
 5
 6
 7
 8
 9

$O(1) + \underbrace{O(n)}_{\text{for loop}} + O(1)$

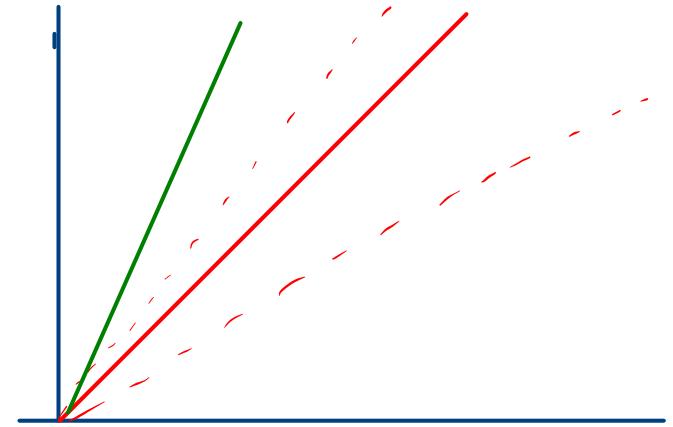
$O(1) + \underbrace{O(n)}_{\text{if statements}} + O(1)$

$O(1) + O(1) + O(1)$

$(n-1) * O(1) + 3 * O(1)$

$$F(n) = 3(n-1) + 3 = \underline{\underline{O(n)}}$$

n	$3(n-1) + 3$	$3(n-1)$	$3n$
10	12	9	10
100	102	99	100
1000	1002	999	1000



```
... main (...) {
```

```
    For (i = 2; i <= m - 2; i++) {
```

```
        For (j = 1; j < m + 2; j++) {
```

```
            if (A[i] < A[j]) { O(1)
```

⋮

```
            } O(1)
```

For inner ~ O

```
}
```

```
}
```

```
}
```

$$F(m) = \sum_{i=2}^{m-2} \left(\sum_{j=L}^{m+L} 1 \right) = (m-3) * \overbrace{(m+1) + (m+1) + (m+1) + \dots + (m+1)}^{(m-3)} = (m-3) * (m+1) = m^2 + m - 3m - 3 = m^2 - 2m - 3 = O(m^2)$$

$$L \rightarrow m+1 = m+1$$

$$L \rightarrow m = m$$

$$L \rightarrow m-L = n-L$$

$$1 \rightarrow m-2 = m-2$$

$$2 \rightarrow m-2 = m-3$$

$$= O(m^2)$$

... main (...) {

For ($i = 1$; $i \leq m - 1$; $i++$) {

For ($j = i + 1$; $j < m$; $j++$) {

if ($A[i] < A[j]$) {

⋮
⋮

$O(1)$

}

}

}

$$F(m) = \sum_{i=1}^{m-1} \left(\sum_{j=i+1}^{m-1} 1 \right) = \begin{matrix} i=1 & 2 & 3 & \dots & m-3 & m-2 & m-1 \\ (m-2) & + (m-3) & + (m-4) & + \dots & + 2 & + 1 & + 0 \end{matrix}$$
$$= \frac{m(m+1)}{2} - (m-1) - m = \frac{m^2 + m - 4m + 2}{2} = \frac{m^2 - 3m + 2}{2} = O(m^2)$$

1: For interval
2: $m-1 = m-2$

2: $3 \rightarrow m-1$

⋮
⋮

penúltima:

$i = m-2$

$j = m-1$

1 vez

última:

$i = m-1$
 $j = m \rightarrow (m-1)$

PA: $1 + 2 + 3 + \dots + (m-2) + (\cancel{m-1}) + \cancel{m} = \frac{m(m+1)}{2} - (m-1) - m =$

Exemplo 2: Ordenar os elementos de um vetor

```
void Ordena (vetor A)
1 {
2     int i, j, min, x;
3     for (i = 1; i <= n; i++)
4     {
5         min = i;
6         for (j = (i+1); j <= n; j++) {
7             if (A[j-1] < A[min-1])
8                 min := j;
9             x = A[min-1];
10            A[min-1] = A[i-1];
11            A[i-1] = x;
12        }
13    }
```

COMPARAÇÕES

$$\begin{aligned} C(n) &= \sum_{i=1}^n \left(\sum_{j=i+1}^n (1) \right) = \\ &= \underset{i=1}{(n-1)} + \underset{2}{(n-2)} + \underset{3}{(n-3)} + \dots + \underset{(n-2)}{2} + \underset{(n-1)}{1} + 0 \\ &= 0 + 1 + 2 + 3 + 4 + 5 + \dots + (n-2) + (n-1) \quad (+n) \\ &= \frac{n(n+1)}{2} - n = \frac{n^2 + n - 2n}{2} = \frac{n^2 - n}{2} \end{aligned}$$

$$C(n) = \frac{1}{2}n^2 - \frac{1}{2}n = O(n^2)$$

$$1 \rightarrow m = m$$

$$2 \rightarrow m = m - 1$$

Exemplo 2: Ordenar os elementos de um vetor

```
void Ordena (vetor A)
1  {
2      int i, j, min, x;
3      for (i = 1; i <= n; i++)
4      {
5          min = i;
6          for ( j = (i+1); j <= n; j++ ){
7              if (A[j-1] < A[min-1] )
8                  min := j;
9              x = A[min-1];
10             A[min-1] = A[i-1];
11             A[i-1] = x;
12         }
13     }
```

MOVIMENTAÇÕES

$$M(n) = 3n \quad O(n)$$

$$\left. \begin{array}{l} \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array} \right\} T_{no ca} = 3m o v.$$

Exemplo 2: ~~Ordenar~~ os elementos de um vetor

```
void Ordena (vetor A)
1  {
2      int i, j, min, x;
3      for (i = 1; i <= n; i++)
4      {
5          min = i;
6          for ( j = (i+1); j < n; j++ )
7              if (A[j-1] < A[min-1] )
8                  min := j;
9          x = A[min-1];
10         A[min-1] = A[i-1];
11         A[i-1] = x;
12     }
13 }
```

Análise de algoritmos: Alguns princípios

6. PROGRAMAS COM PROCEDIMENTOS RECURSIVOS

- Para cada procedimento recursivo, associar uma função de custo $f(n)$ desconhecida (n mede o tamanho dos argumentos)
- Obter uma relação de recorrência.
- Calcular a complexidade da relação de recorrência.

Recursão

- Uma definição na qual o item que está sendo definido aparece como parte da definição é chamada definição indutiva ou definição recursiva.
- Definições recursivas são compostas de duas partes:
 - Uma base, onde alguns casos simples do item que está sendo definido são dados explicitamente. $0! = 1$
 - Um passo indutivo ou recursivo, onde outros casos do item que está sendo definido são dados em termos dos casos anteriores. $N! = N * (N - 1)!$

Sequências Recursivas

- Uma sequência S é uma lista de objetos que são enumerados segundo alguma ordem; existe o 1º objeto, um segundo e assim por diante.
- $S(k)$ denota o K-ésimo objeto da sequência
- Uma sequência é definida recursivamente, explicitando-se seu primeiro valor (ou seus primeiros valores) e, a partir daí, definindo-se outros valores na sequência em termos dos valores iniciais.

1 1 2
 $s_{(1)}$ $s_{(2)}$ $s_{(3)}$

Exemplo

- Seja a sequência S definida recursivamente por:

$$\begin{cases} S(1) = 2 \\ S(n) = \underline{2S(n-1)} \text{ para } n \geq 2 \end{cases}$$

- Qual o valor de S(1)? E de S(2)? E de S(3)? E de S(5)?

$$S(1) = 2 = 2^{\textcircled{1}}$$

$$S(2) = 2 * S(1) = 4 = 2^2$$

$$S(3) = 2 * S(2) = 8 = 2^3$$

$$S(4) = 2 * S(3) = 16 = 2^{\textcircled{4}}$$

$$S(5) = 2 * S(4) = 32 = 2^5$$

$$\vdots$$
$$S(n) = 2^{\textcircled{n}}$$

$$O(2^n)$$

Relações de Recorrência

- As relações de recorrência são utilizadas para descrever algoritmos recursivos.
- Uma recorrência é uma equação ou desigualdade que descreve uma função em termos de entradas menores.
- Existem três métodos principais para resolver relações de recorrência.
 - Método da Substituição
 - Método da Iteração
 - Método Master ou Mestre

Método da Substituição

$$\sum 3^{i-1}$$

- “Chute Inicial” Pode ser muito difícil de ser encontrado
- Indução matemática para provar que o chute estava correto.

$$\begin{cases} T(n) = 3T(n-1) + 1 & n > 1 \\ T(1) = 1 & n = 1 \end{cases}$$

```
void F(int m) {  
    if (m == 1)  
        return 1;  
    else  
        return F(m-1) + F(m-1) + F(m-1) + 1;  
}
```

$$\begin{aligned} T(1) &= 1 \\ T(2) &= 3 \cdot T(1) + 1 = 3 \cdot 1 + 1 = 4 = (3^2 - 1)/2 = \\ T(3) &= 3 \cdot T(2) + 1 = 3 \cdot 4 + 1 = 13 = (3^3 - 1)/2 \\ T(4) &= 3 \cdot T(3) + 1 = 3 \cdot 13 + 1 = 40 = (3^4 - 1)/2 \\ T(5) &= 3 \cdot T(4) + 1 = 3 \cdot 40 + 1 = 121 \\ T(6) &= 3 \cdot T(5) + 1 = 3 \cdot 121 + 1 = 364 \\ &\vdots \\ T(m) &= \frac{3^m - 1}{2} \end{aligned}$$

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

Fazendo $n = 1$

Hipótese $n = k+1$

$$T(k+1) = \frac{3^{k+1} - 1}{2}$$

Substituição

$$\begin{cases} T(m) = T(m-1) + 1 & m > 1 \\ T(1) = 1 & m = 1 \end{cases}$$

$$T(1) = 1$$

$$T(2) = 1 + 1 = 2$$

$$T(3) = 2 + 1 = 3$$

$$T(4) = 3 + 1 = 4$$

$$T(5) = 4 + 1 = 5$$

•

⋮

•

$$\boxed{T(n) = n}$$

```
void Funcao(int n)
{
    if (n == 1)
        return 1;
    else
        return Funcao(n-1) + 1;
}
```

$$\begin{cases} T(n) = T(n-1) + n & n > 1 \\ T(1) = 1 & n = 1 \end{cases}$$

$$T(n) = \sum_{i=1}^n \frac{1}{2}$$

$$T(1) = 1 = 0*1 + 1$$

$$T(2) = 1 + 2 = 3 \quad \frac{1}{2}*2 + 2$$

$$T(8) = 28 + 8 = 36$$

$$T(3) = 3 + 3 = 6$$

$$T(9) = 36 + 9 = 45$$

$$T(4) = 6 + 4 = 10$$

$$T(n) = \frac{n}{2} + n$$

$$T(5) = 10 + 5 = 15$$

$$T(6) = 15 + 6 = 21$$

$$T(7) = 21 + 7 = 28$$

$$T(n) = \frac{n^2 + n}{2}$$