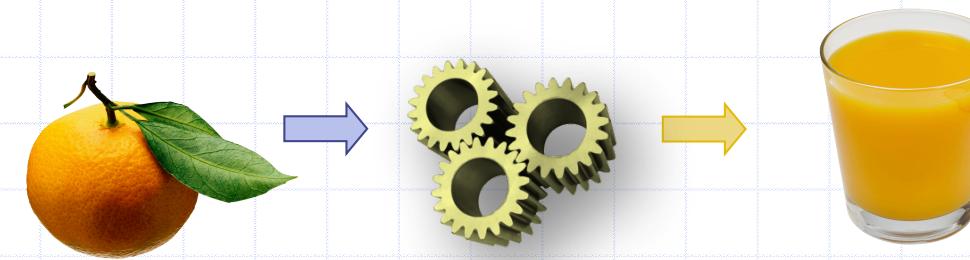


Análise de Algoritmos



Entrada

Algoritmo

Saída

Um **algoritmo** é um procedimento passo-a-passo,
para resolver um determinado problema em tempo finito

Referências

◆ Baseado no material que acompanha o livro:

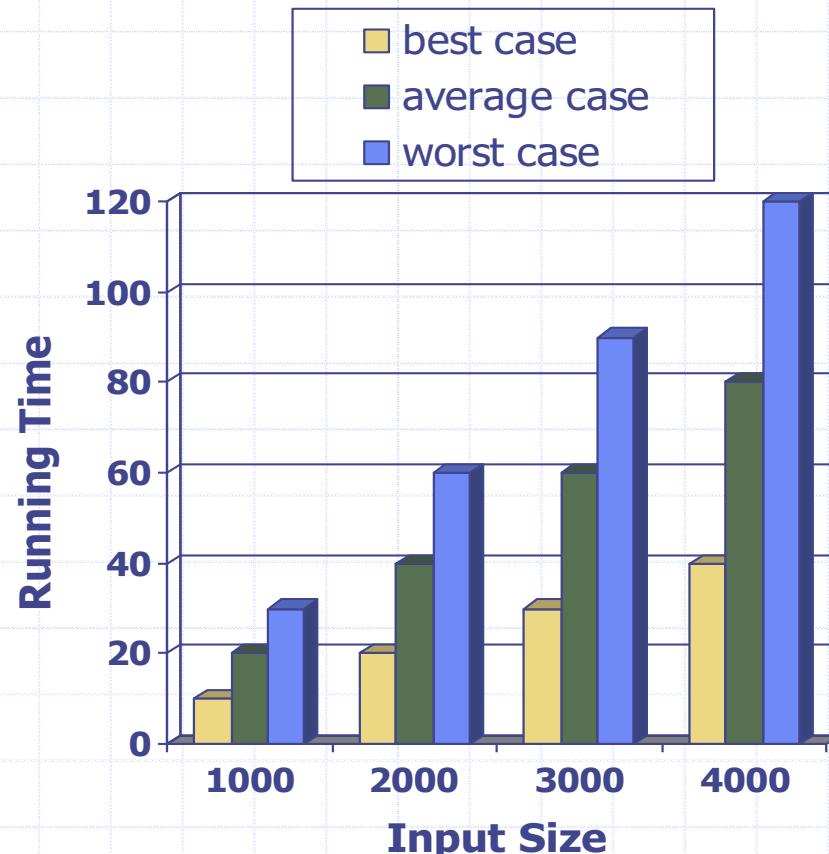
- Estruturas de dados e Algoritmos em Java
M. Goodrich e R. Tamassia, Bookman.
- Cap 4 - **Obs.:** repare que nós “pulamos” o cap 3, que é um capítulo novo da 4a edição

◆ Outras fontes:

- Introduction to Algorithms. T.H.Cormen, C.E.Leiserson, R.L.Rivest e Stein. McGraw-Hill.

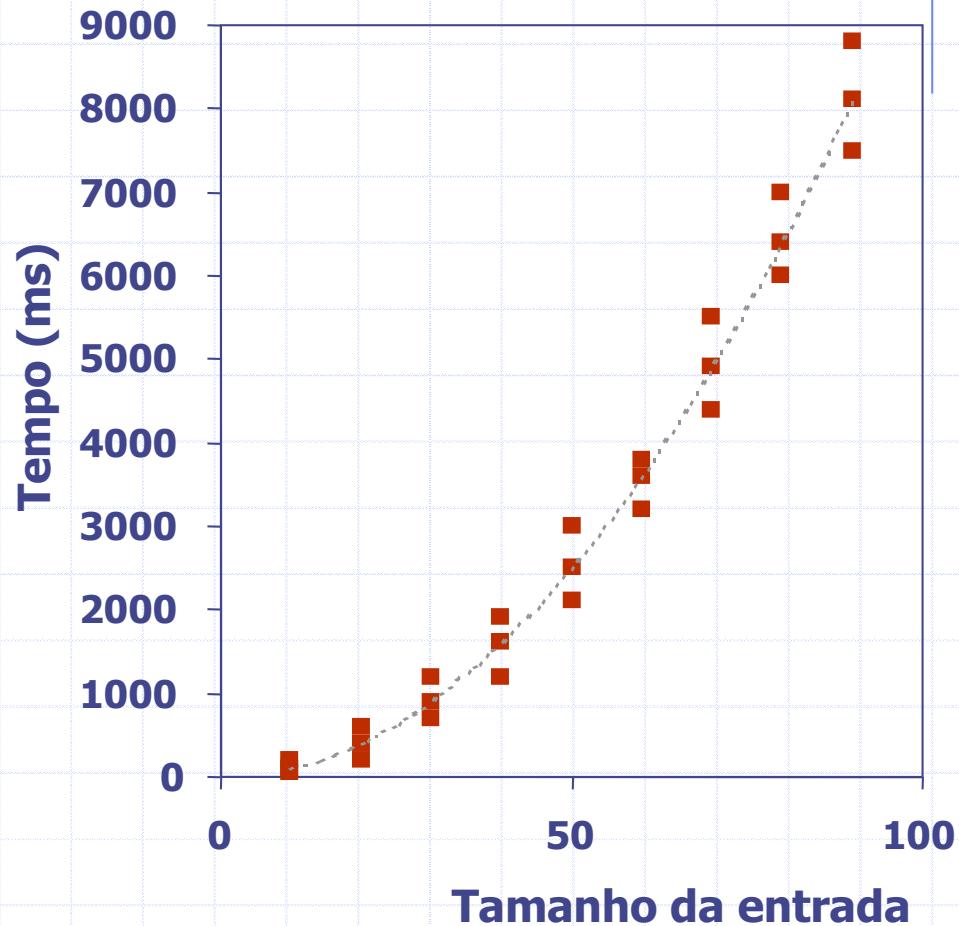
Tempo de execução

- ◆ A maioria dos algoritmos transforma objetos de entrada em objetos de saída
- ◆ O tempo de execução de um algoritmo tipicamente cresce à medida que aumentamos o tamanho de sua entrada
- ◆ O caso médio muitas vezes é difícil de se determinar
- ◆ Melhor considerar o tempo de execução do pior caso.
 - É mais fácil
 - Crucial para aplicações como jogos, finanças e robótica



Método experimental

- ◆ Escreva um programa que implementa um algoritmo
- ◆ Execute o programa variando a composição e o tamanho das entradas
- ◆ Use algum método como uma rotina do sistema operacional para obter medidas precisas do tempo de execução
- ◆ Esboce um gráfico dos resultados obtidos



Experimentos são Limitados

- ◆ é preciso implementar o algoritmo em alguma linguagem de programação (Qual? Porque?)
- ◆ os resultados podem não servir como indicativo do tempo de execução para outras entradas que não foram consideradas nos testes
- ◆ para comparar dois algoritmos, deveriam ser utilizados os mesmos ambientes de hardware e software

Análise Teórica

- ◆ baseada em uma descrição de alto nível do algoritmo, em vez de uma implementação
- ◆ caracteriza o tempo de execução como uma função do tamanho da entrada, n (bits)
- ◆ leva em consideração todas as possíveis entradas
- ◆ permite avaliar a rapidez de um algoritmo, independentemente do ambiente de hardware e/ou software



Pseudo-código (§ 3.2)

- ◆ Descrição de “alto nível” de um algoritmo (usa linguagem natural)
- ◆ Mais estruturada que um texto simples
- ◆ Menos detalhada que um programa
- ◆ Notação muito utilizada para descrever algoritmos
- ◆ Esconde detalhes de projeto dos programas

Exemplo: encontrar o valor máximo de um vetor

Algoritmo *arrayMax (A, n)*

Entrada: vetor A de n inteiros

Saída: elemento máximo de A

atualMax $\leftarrow A[0]$

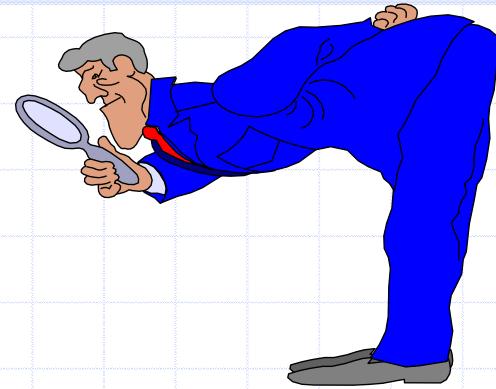
para $i \leftarrow 1$ até $n - 1$ **faça**

se $A[i] > atualMax$ **então**

atualMax $\leftarrow A[i]$

retorne *atualMax*

Pseudo-código



- ◆ Controle do fluxo de execução
 - se ... então ... [senão ...]
 - enquanto ... faça ...
 - repita ... até que ...
 - repita ... enquanto ...
 - para ... faça ...
 - Indentação substitui chaves
- ◆ Declaração de algoritmo
Algoritmo *metodo (arg [, arg...])*
Entrada: ...
Saída: ...

- ◆ Chamadas a métodos
var.metodo (arg [, arg...])
- ◆ Retorno de valores
retorne *expressão*
- ◆ Expressões
← **Atribuição**
(mesmo que = em Java, C, C++)
= **Teste de igualdade**
(mesmo que == in Java, C, C++)
- n^2 Sobrescritos e outros formatos matemáticos são permitidos

O Modelo RAM - *Random Access Machine*

- ◆ Uma única **CPU**
- ◆ Um conjunto potencialmente ilimitado de células de **memória**, que podem ser acessadas aleatoriamente, e podem armazenar um número arbitrário ou um caracter
- ◆ As células de memória são numeradas, e o tempo de acesso a cada célula é unitário (constante)

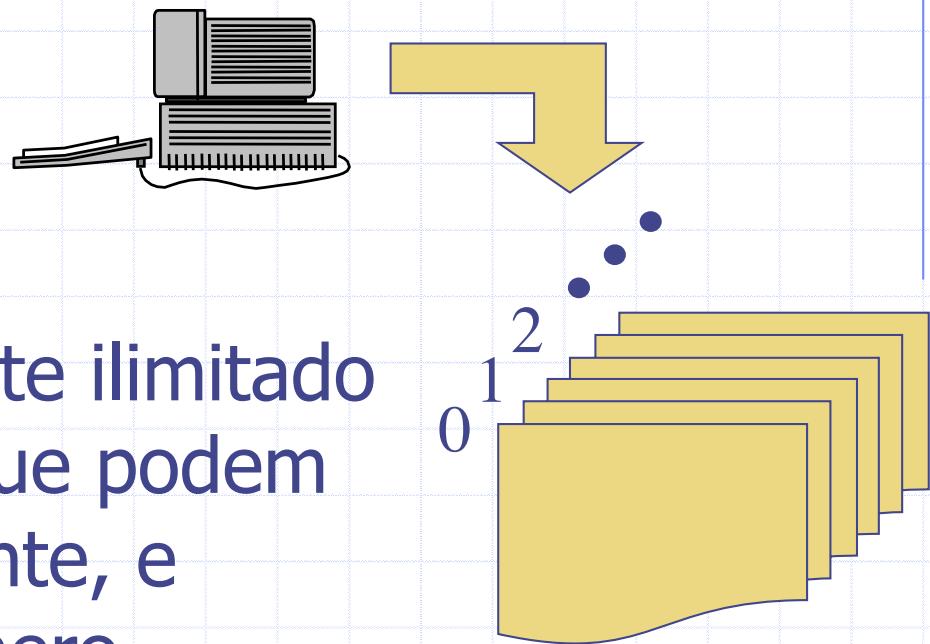
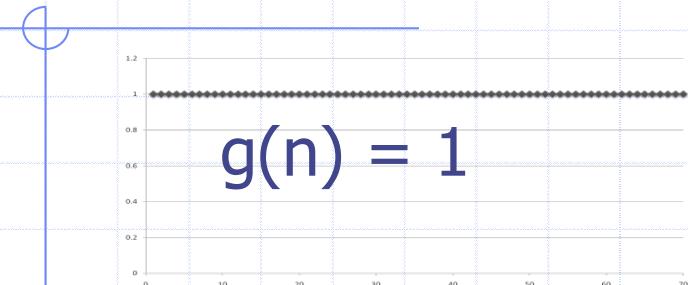
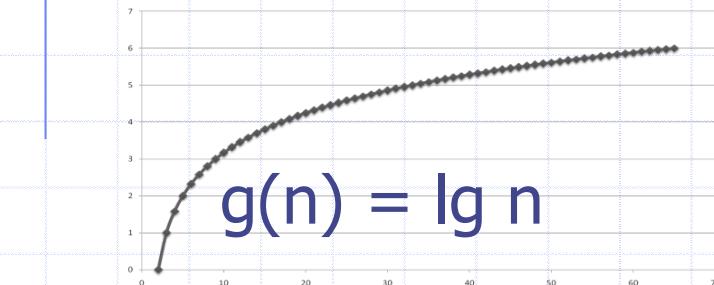


Gráfico da função usando escala “Normal”

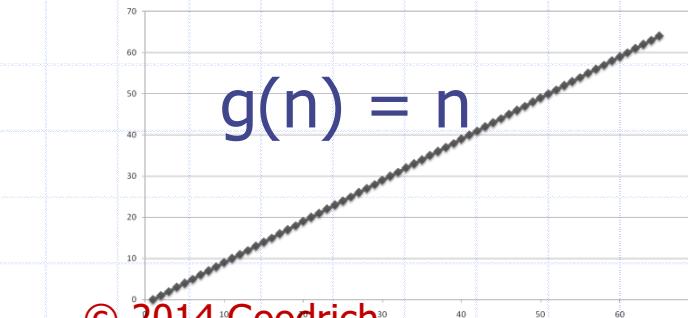
Slide by Matt Stallmann
included with permission.



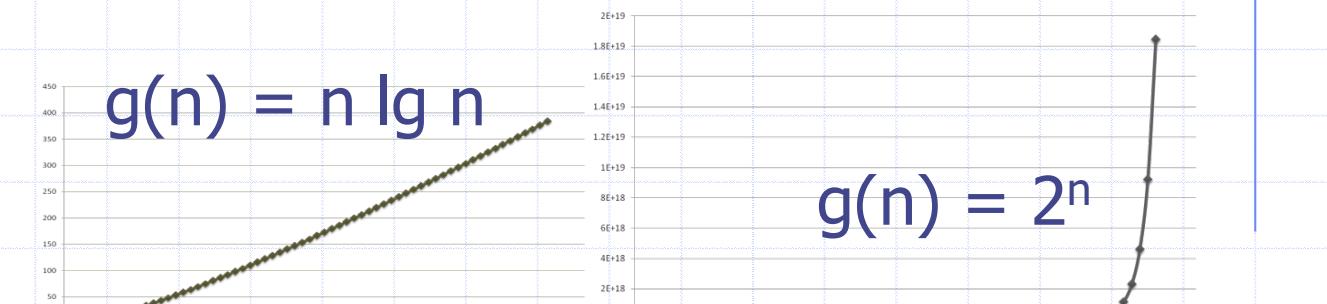
$$g(n) = 1$$



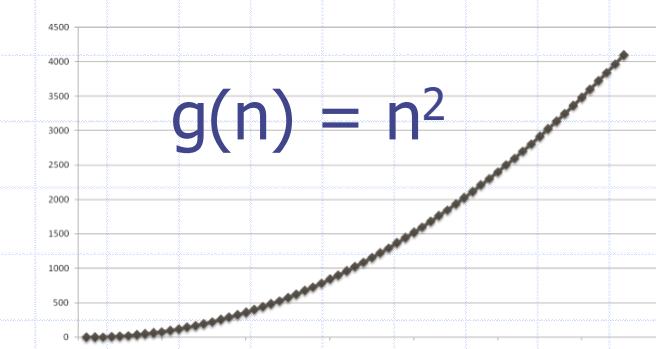
$$g(n) = \lg n$$



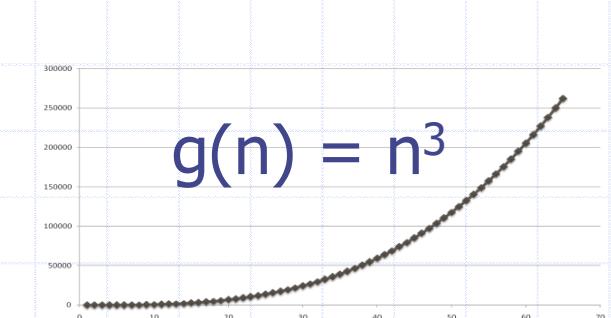
$$g(n) = n$$



$$g(n) = n \lg n$$



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

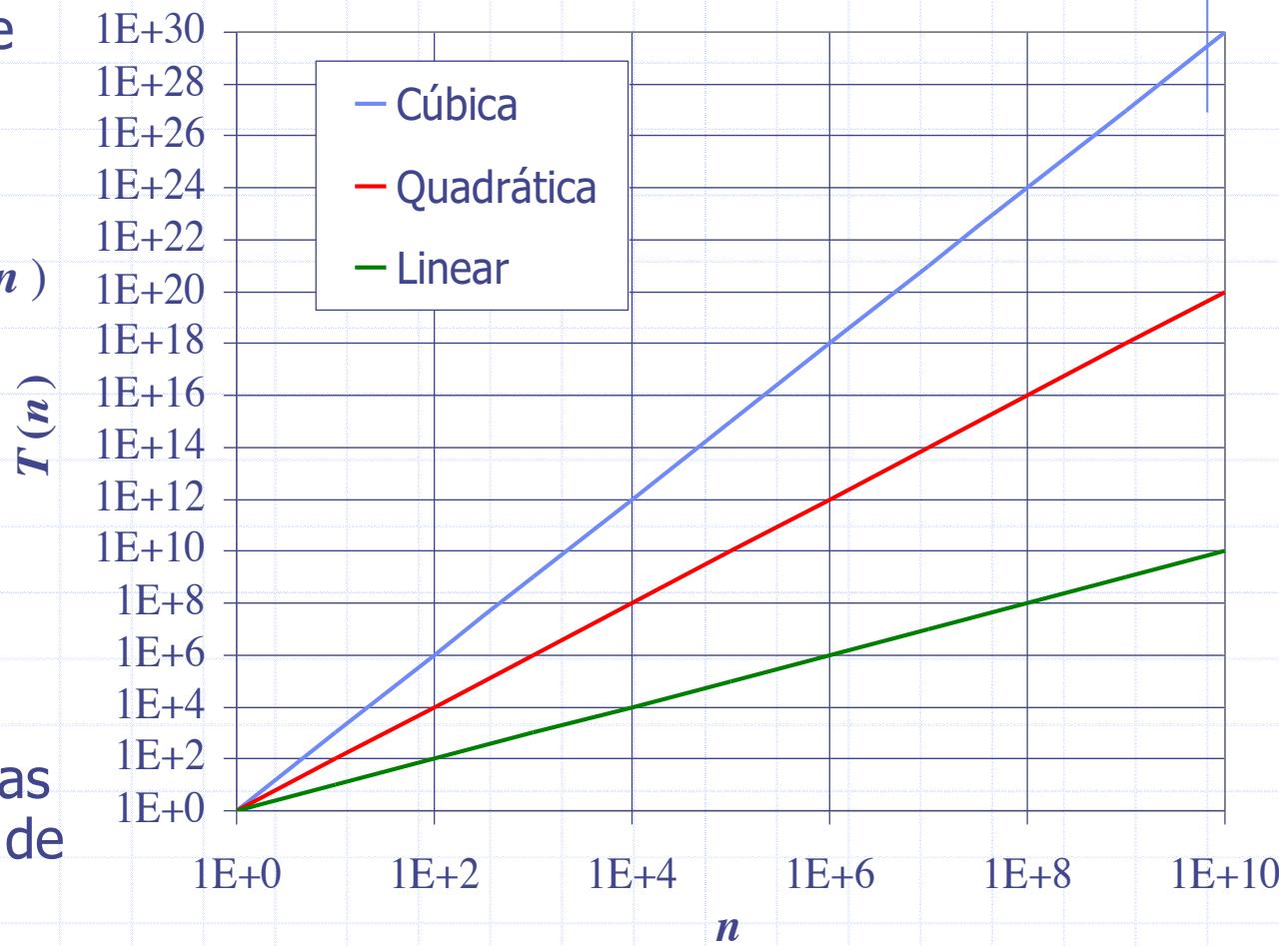


$$g(n) = n^3$$

Sete Funções Importantes (§ 3.3)

- ◆ Sete funções que aparecem freqüentemente na análise de algoritmos:
 - Constante ≈ 1
 - Logarítmica $\approx \log_2 n$ (que pode ser expressa como $\lg n$)
 - Linear $\approx n$
 - N-Log-N $\approx n \log_2 n$
 - Quadrática $\approx n^2$
 - Cúbica $\approx n^3$
 - Exponencial $\approx 2^n$

- ◆ No gráfico de escala logarítmica, a inclinação das linhas corresponde à taxa de crescimento da função



Operações Primitivas

- ◆ São ações simples executadas pelos algoritmos
- ◆ Aparecem no pseudo-código
- ◆ Não dependem da linguagem de programação
- ◆ Considera-se que todas as operações têm **tempo de execução constante** no modelo RAM

◆ Exemplos:

- Avaliar uma expressão
- Atribuir valor a uma variável
- Indexação de um vetor
- Chamada de um método/função
- Retornar de uma chamada de método/função
- Comparação de valores



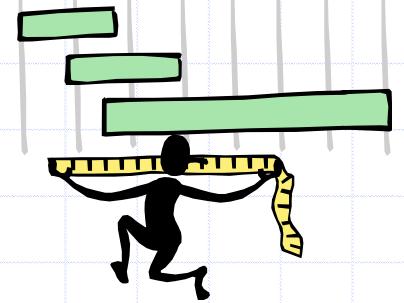
A Contagem de Operações Primitivas (§ 3.4)

- ◆ Inspecionando o pseudo-código, podemos determinar o número máximo de operações primitivas executadas por um algoritmo, em função do tamanho da entrada

```
Algoritmo arrayMax(A, n)
    atualMax  $\leftarrow A[0]
    para i  $\leftarrow 1$  até n – 1 faça
        se A[i] > atualMax então
            atualMax  $\leftarrow A[i]
        {incremente contador i }
    retorno atualMax$$ 
```

operations
2
<i>n</i>
$2(n - 1)$
$2(n - 1)$
$2n$
1
Total $7n - 1$

Estimando o Tempo de Execução



- ◆ O algoritmo *arrayMax* executa $7n - 1$ operações primitivas no pior caso. Suponhamos que:
 - a = Tempo gasto pela operação primitiva mais rápida
 - b = Tempo gasto pela operação primitiva mais lenta
- ◆ Seja $T(n)$ o tempo de pior caso de *arrayMax*. Então
$$a(7n - 1) \leq T(n) \leq b(7n - 1)$$
- ◆ Portanto, o tempo de execução $T(n)$ é limitado por duas **funções lineares** em n .

Outro Exemplo – Busca Binária

- ◆ O algoritmo compara o valor procurado com o elemento que está no centro do vetor A (posição K)
 - Se forem iguais, o algoritmo pára
 - Se forem diferentes, o algoritmo verifica se x pode estar na metade esquerda ou na metade direita do vetor, descartando a outra metade
- ◆ A busca prossegue na parte em que x pode ser encontrado.

Algoritmo *buscaBinaria(V, n, x)*

Entrada vetor A de n inteiros, e um valor x a ser procurado

Saída a posição onde x foi encontrado no vetor, ou -1, caso contrário

$i \leftarrow 0$

$s \leftarrow n - 1$

enquanto $i < s$ **faça**

$k \leftarrow (i + s)/2;$

se $x = A[k]$ **então**

retorne k

se $x > A[k]$ **então**

$i \leftarrow k + 1$

senão $s \leftarrow k - 1$

retorne (-1)

Análise – Busca Binária

- ◆ É preciso descobrir quantas vezes o algoritmo repete o *laço **enquanto** ...*
- ◆ Inicialmente, o intervalo de busca em **A** possui **N** elementos ($i \leftarrow 0$ e $s \leftarrow N-1$)
- ◆ Após 1 comparação, restarão **$N/2$** elementos
- ◆ Após 2 comparações, restarão **$N/4$** elementos
- ◆ ...
- ◆ Após quantas comparações, restarão **0** elementos?
 - A resposta é $\lceil \log_2 N \rceil$ (i.e. o menor inteiro maior ou igual a $\log_2 N$)

Análise da Busca Binária

Algoritmo $\text{buscaBinaria}(V, n, x)$

Entrada vetor A de n inteiros, e um valor x a ser procurado

Saída a posição onde x foi encontrado no vetor, ou -1

$i \leftarrow 0$

$s \leftarrow n - 1$

enquanto $i < s$ **faça**

$k \leftarrow (i + s)/2;$

se $x = A[k]$ **então**

retorne k

se $x > A[k]$ **então**

$i \leftarrow k + 1$

senão $s \leftarrow k - 1$

retorne (-1)

1

2

$\log_2 n + 1$

$\log_2 n$

$\log_2 n$

1

$\log_2 n$

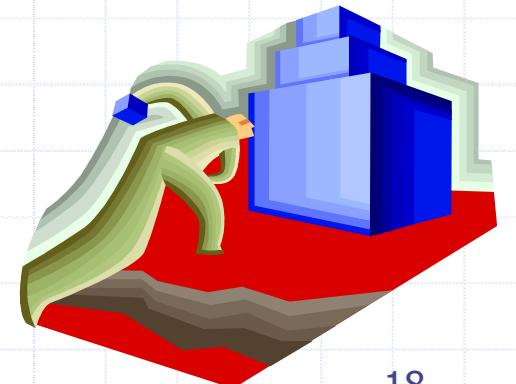
$\log_2 n$

1

Total: $5 \log_2 n + 4$

Taxa de crescimento de $T(n)$

- ◆ Mudar o ambiente de hardware/ software
 - Afeta $T(n)$ por um fator constante, mas
 - Não altera a taxa de crescimento de $T(n)$
- ◆ A taxa de crescimento linear de $T(n)$ é uma propriedade intrínseca do algoritmo
arrayMax



Por que a **taxa** interessa

if runtime is...	time for $n + 1$	time for $2n$	time for $4n$
$c \lg n$	$c \lg (n + 1)$	$c (\lg n + 1)$	$c(\lg n + 2)$
$c n$	$c(n + 1)$	$2c n$	$4c n$
$c n \lg n$	$\sim c n \lg n + c n$	$2c n \lg n + 2cn$	$4c n \lg n + 4cn$
$c n^2$	$\sim c n^2 + 2c n$	$4c n^2$	$16c n^2$
$c n^3$	$\sim c n^3 + 3c n^2$	$8c n^3$	$64c n^3$
$c 2^n$	$c 2^{n+1}$	$c 2^{2n}$	$c 2^{4n}$

Tempo de execução quadruplica quando o problema dobra de tamanho

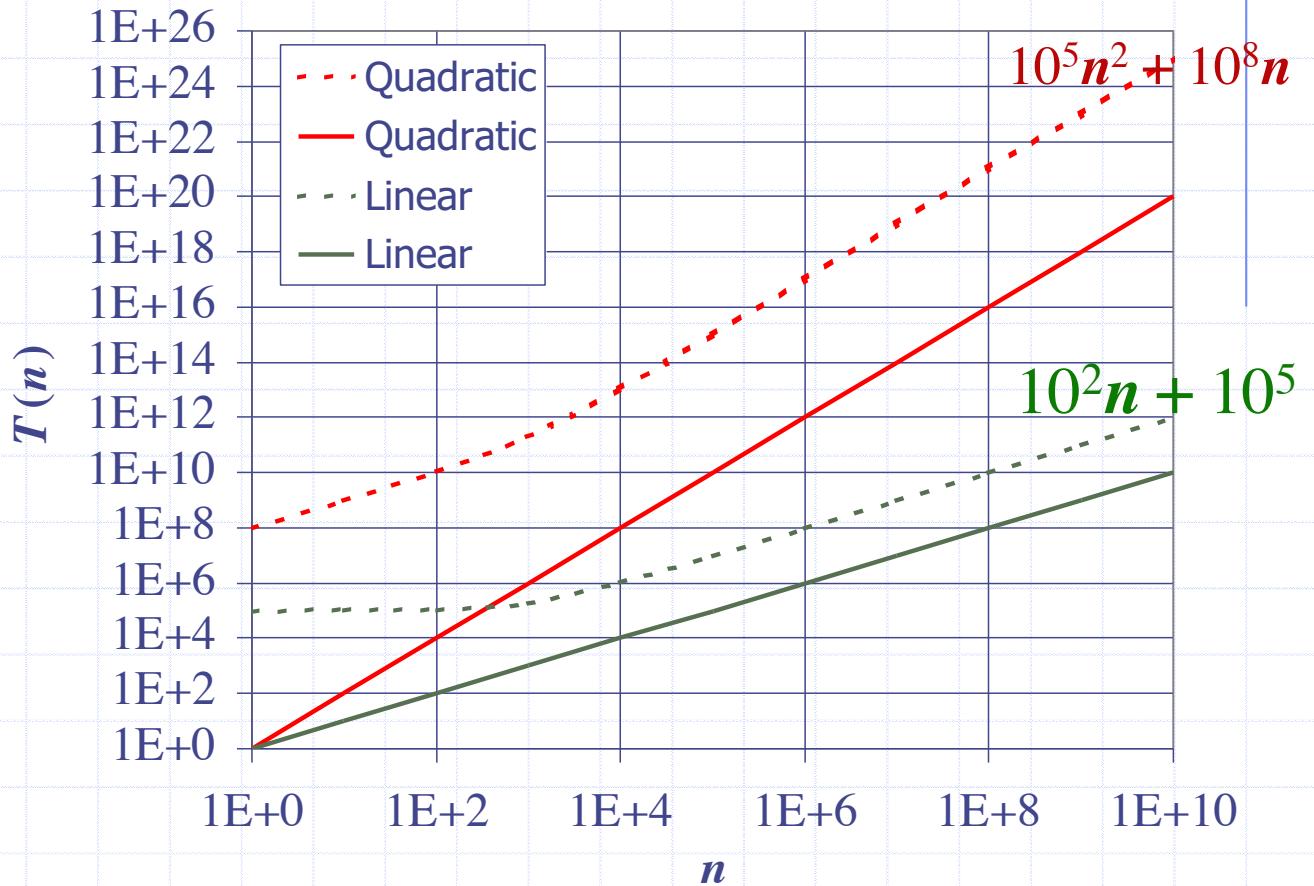
Fatores Constantes

◆ A taxa de crescimento não é afetada por

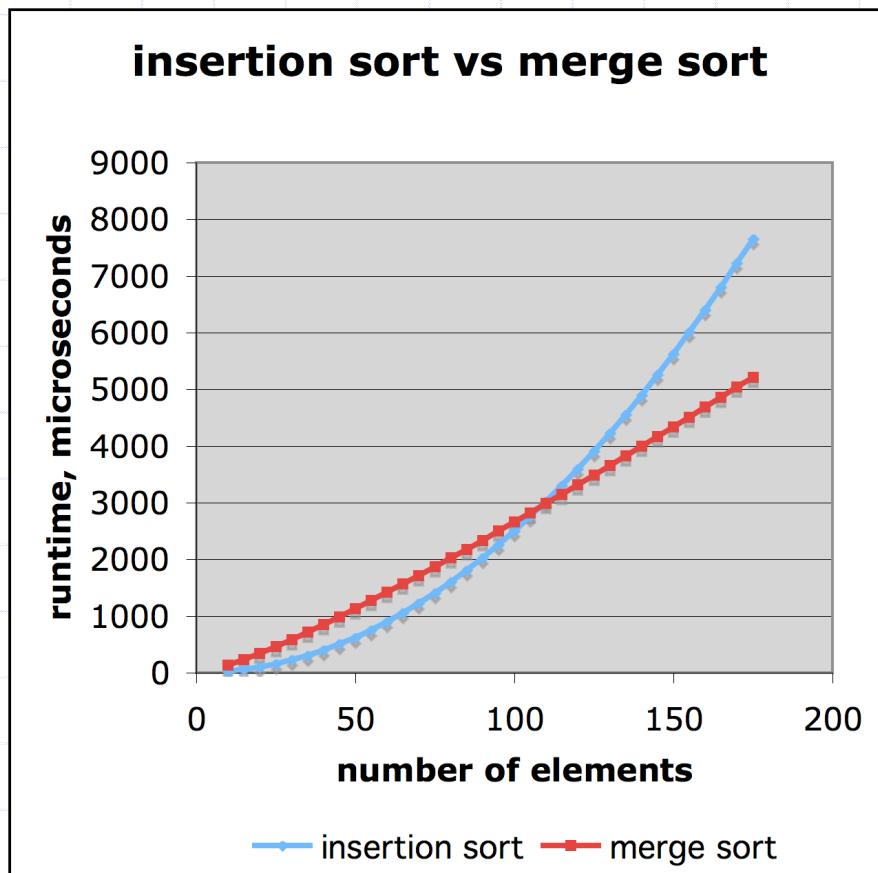
- fatores constantes ou
- termos de menor ordem

◆ Exemplos

- $100 n + 100000$ é uma função linear
- $100000 n^2 + 100000000 n$ é uma função quadrática



Comparação de Dois Algoritmos



insertion sort é
 $n^2 / 4$

merge sort é
 $2 n \lg n$

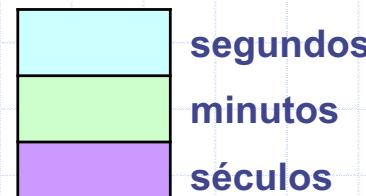
Ordenar 1 milhão de itens?

insertion sort leva
aprox. 70 horas
enquanto

merge sort leva
aprox 40 seconds

Essa máquina é lenta, mas se fosse
100 x + rápida, levaria 40 minutos
Contra menos de 0.5 segundos

$T(n)$ versus Tempo Cronológico



Características do hardware	
Número de Instruções executadas por Ciclo do relógio (IPC)	8
Frequência	3,00E+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

Simplificando a notação

- ◆ Termos constantes não são significativos
- ◆ Termos de menor ordem também não são significativos
- ◆ Portanto:
 - $4n^5 + 10n^3 + 100$ tem a mesma ordem de grandeza que n^5

Técnicas de prova

- ◆ **contra-exemplo:** para *justificar* uma afirmação do tipo “existe um elemento x no conjunto S que tem a propriedade P ” basta encontrar um x em S que tenha tal propriedade. Para mostrar que uma afirmação do tipo “todo elemento x no conjunto S tem a propriedade P ” é *falsa*, precisamos mostrar um x em S que não tenha tal propriedade. Um tal x é dito um *contra-exemplo*.

Técnicas de prova

◆ **contrapositivo:** para justificar a afirmação “se p é verdade, então q é verdade” podemos estabelecer a veracidade da afirmação equivalente “se q não é verdade, então p não é verdade”, *se a segunda alternativa for mais fácil de se provar.* Ex. se ab é ímpar, então a é ímpar ou b é par. Considere seu contrapositivo “se a é par e b é ímpar, então ab é par”. Suponha que $a = 2i$ para algum inteiro i . Nesse caso, $ab = (2i)b = 2(ib)$ que é par. Portanto, ab é par.

Técnicas de prova

◆ **contradição:** é uma técnica de demonstração por negação, como a contraposição. Estabelecemos que uma afirmação q é verdadeira supondo inicialmente que q é falsa, e mostrando que essa suposição nos leva a um absurdo (situação inconsistente). Portanto, q deve ser verdadeira originalmente.

Ex. “se ab é ímpar, então a e b são ímpares”
Suponha que ab é ímpar, mas que a é par ou b é par. Se a é par, então $a = 2i$ para algum inteiro i . Logo, $ab = (2i)b = 2(ib)$ que é par. Mas supomos que ab fosse ímpar. Absurdo. Analogamente para b par. Portanto, a e b devem ser ambos ímpares (necessariamente).

Técnicas de prova

◆ **indução (finita)**: esta técnica se resume em mostrar que para qualquer inteiro $n \geq 1$, existe uma sequência finita de implicações que inicia com um fato verdadeiro, $q(1)$, e leva à confirmação de que $q(n)$ é verdadeiro para todo n . Em geral, começamos uma demonstração por indução mostrando que $q(n)$ é verdadeiro para $n = 1$ (e possivelmente para outros valores $n = 2, 3, \dots, k$ para algum k constante). Isso se chama “*base da indução*”. A seguir, temos o “*passo de indução*”: mostramos que se $q(i)$ é verdadeiro para $i < n$, então, $q(n)$ é verdadeiro”. A combinação dessas duas partes completa a prova por indução (finita).

Técnicas de prova

♦ Ex. considere a sequência de Fibonacci: $F(1) = 1$, $F(2) = 2$ e $F(n) = F(n-1) + F(n-2)$ para $n > 2$. Prove que $F(n) < 2^n$.

Base: ($n \leq 2$) $F(1) = 1 < 2^1$, e $F(2) = 2 < 4 = 2^2$

Passo de indução: ($n > 2$) suponha que a afirmação é verdadeira para todo $n' < n$.

Considere $F(n)$. Como $n > 2$, então, $F(n) = F(n-1) + F(n-2)$. Além disso, $n-1 < n$ e $n-2 < n$. Logo, $F(n) < 2^{n-1} + 2^{n-2}$. Mas, $2^{n-1} + 2^{n-2} < 2^{n-1} + 2^{n-1} = 2 * 2^{n-1} = 2^n$. Isso completa a prova por indução (finita).

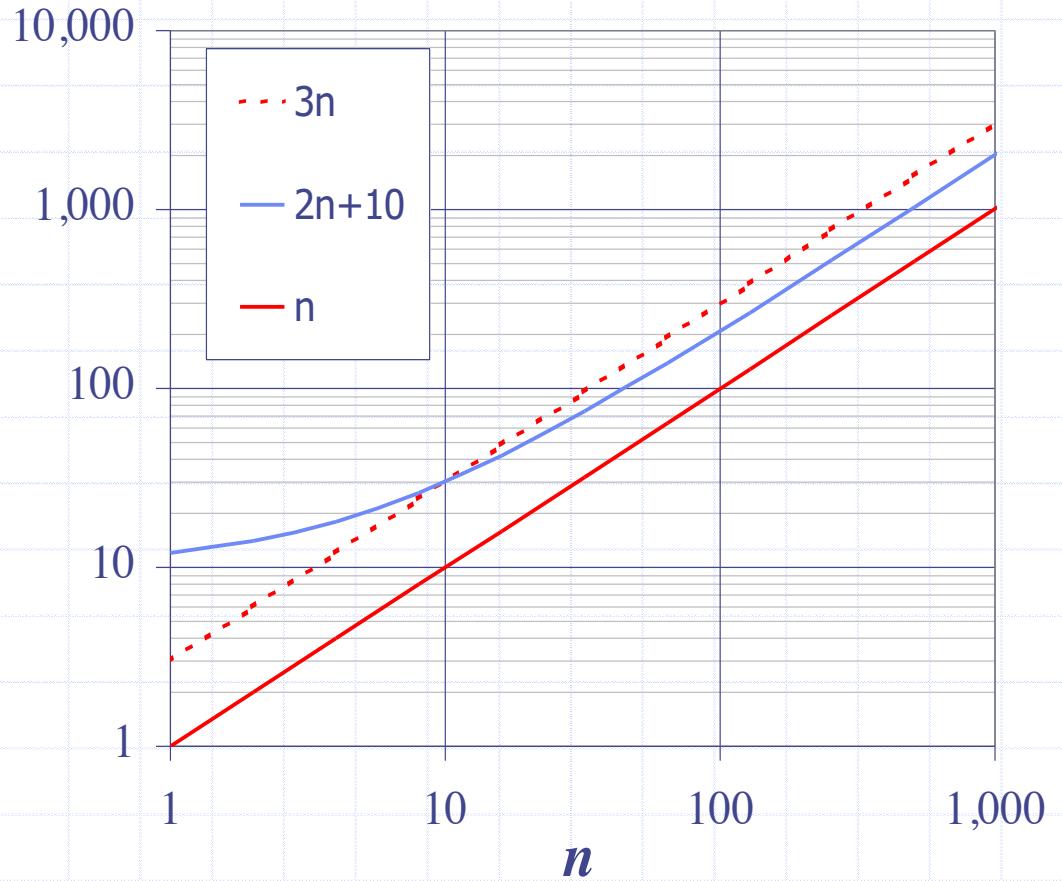
Notação O (§ 3.4)

◆ Dadas as funções $f(n)$ e $g(n)$, dizemos que $f(n)$ é $O(g(n))$ se existirem constantes inteiras e positivas c e n_0 tais que

$$f(n) \leq c.g(n) \text{ para } n \geq n_0$$

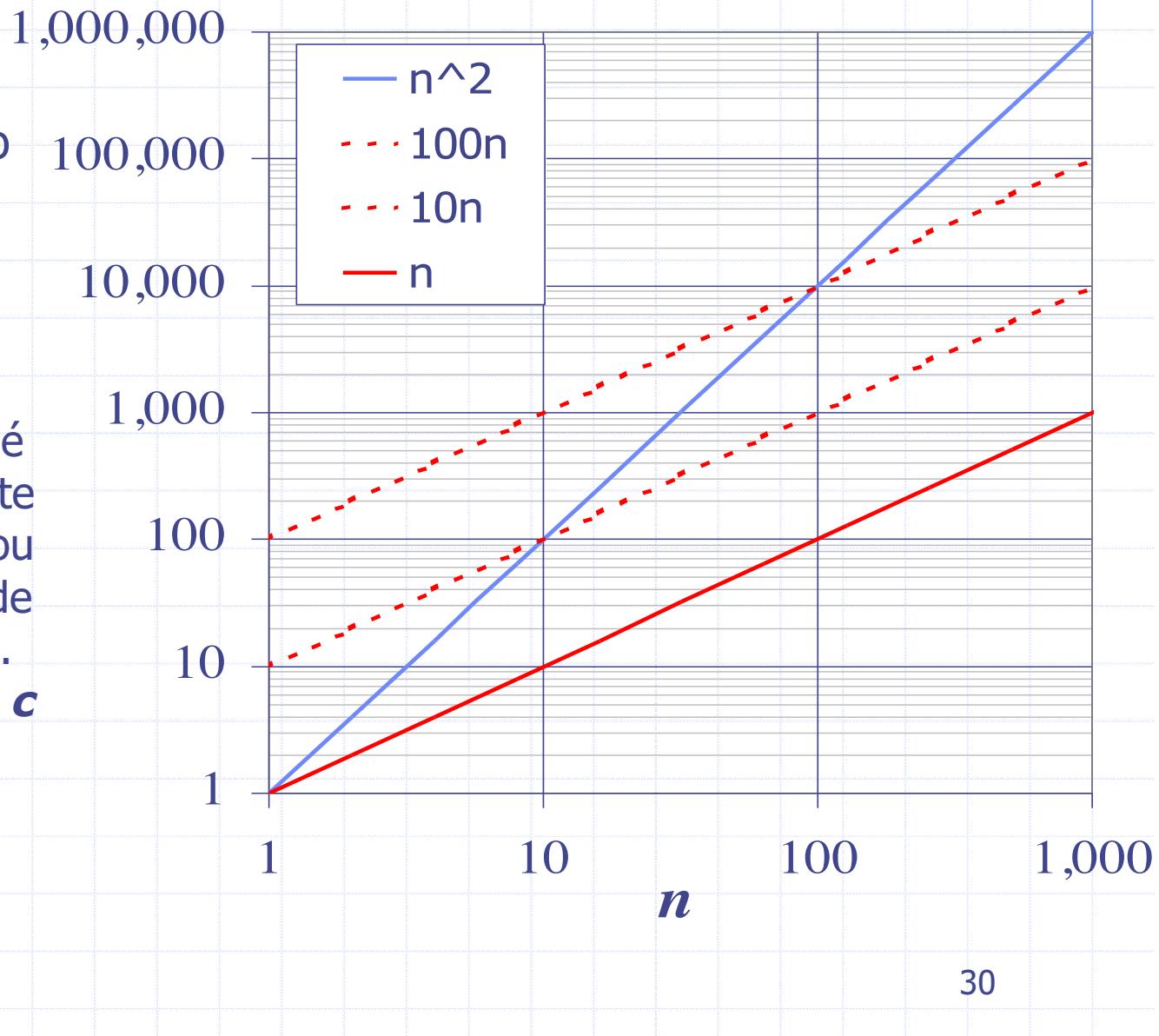
◆ Exemplo: $2n + 10$ é $O(n)$

- $2n + 10 \leq cn$
- $cn - 2n \geq 10$
- $(c - 2)n \geq 10$
- $n \geq 10/(c - 2)$
- considere $c = 3$ e $n_0 = 10$



Um exemplo

- ◆ Podemos dizer que a função n^2 é $O(n)$?
 - $n^2 \leq cn$
 - $n \leq c$
 - Para que a desigualdade acima possa ser satisfeita, é preciso achar uma constante c , que seja sempre maior ou igual a n , para todo valor de n (arbitrariamente grande). Mas isso é impossível, pois c deve ser constante.



Mais Exemplos do Conceito de O

- ◆ $7n-2$

$7n-2$ é $O(n)$

Existem $c > 0$ e $n_0 \geq 1$ tais que $7n-2 \leq c n$ para todo $n \geq n_0$??

isso é verdadeiro para $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$

$3n^3 + 20n^2 + 5$ é $O(n^3)$

Existem $c > 0$ e $n_0 \geq 1$ tais que $3n^3 + 20n^2 + 5 \leq c n^3$ para $n \geq n_0$??

isso é verdadeiro para $c = 4$ and $n_0 = 21$

- $3 \log n + 5$

$3 \log n + 5$ é $O(\log n)$

Existem $c > 0$ e $n_0 \geq 1$ tais que $3 \log n + 5 \leq c \log n$ para $n \geq n_0$??

Isso é verdadeiro para $c = 8$ e $n_0 = 2$



O e Taxa de Crescimento

◆ Afinal de contas, o que quer dizer " $f(n)$ é $O(g(n))$ "?

- Matematicamente isso significa que "a função $g(n)$ é um **limitante assintótico superior para $f(n)$** "
- Podemos interpretar isso como " $f(n)$ cresce menos, ou tanto quanto $g(n)$ ", e matematicamente

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

Evite abusar da notação: $f(n) = O(g(n))$

Alguns casos particulares de O

- ◆ Se $f(n)$ for um polinômio de grau d , então $f(n)$ é $O(n^d)$, i.e.,
 - Desconsidere os termos de menor ordem
 - Desconsidere fatores constantes
- ◆ Use a classe de funções mais próxima possível
 - “ $2n$ é $O(n)$ ” ao invés de “ $2n$ é $O(n^2)$ ”
- ◆ Use a expressão mais simples
 - “ $3n + 5$ é $O(n)$ ” ao invés de “ $3n + 5$ é $O(3n)$ ”

Análise Assintótica de Algoritmos

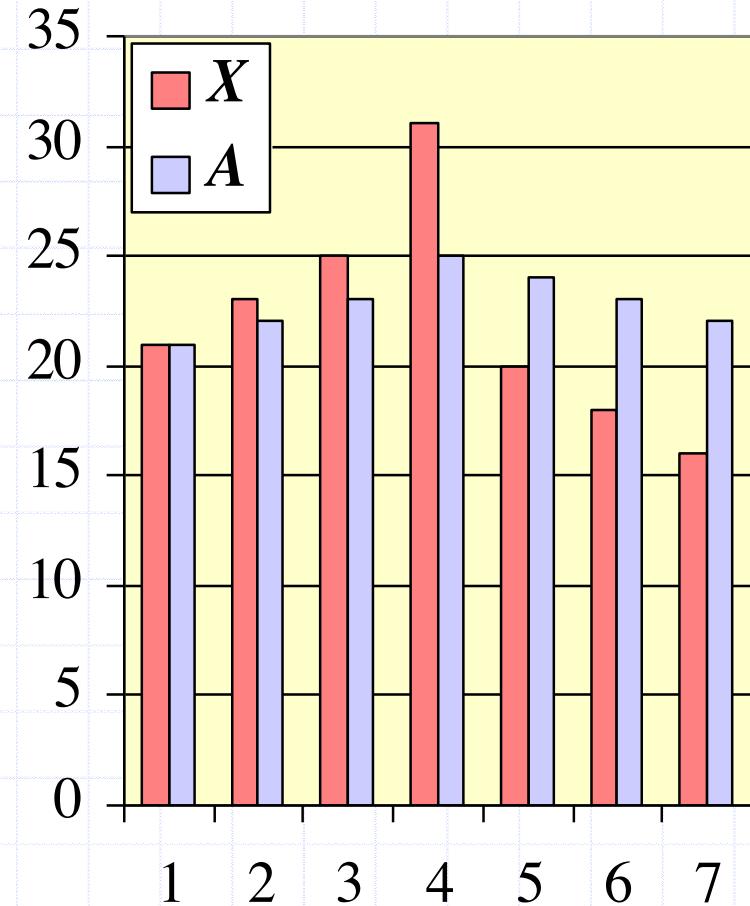
- ◆ A análise assintótica de um algoritmo determina o seu tempo de execução em notação O
- ◆ Para fazer a análise assintótica:
 - Nós encontramos o número de operações primitivas executado pelo algoritmo no pior caso (sempre em função do tamanho da entrada)
 - E escrevemos essa função com notação O
- ◆ Exemplo:
 - Nós determinamos que o algoritmo *arrayMax* executa no máximo $7n - 1$ operações primitivas
 - Então, dizemos que o algoritmo *arrayMax* “executa em tempo $O(n)$ ”
- ◆ Como os fatores constantes e termos de menor ordem já foram descartados, podemos ignorá-los

Calculando a Média dos Prefixos

- ◆ Vamos ilustrar a análise assintótica com dois algoritmos que fazem o cálculo da média dos prefixos
- ◆ A i -ésima média de prefixo de um vetor X é a média aritmética dos $(i + 1)$ primeiros elementos de X :

$$A[i] = (X[0] + X[1] + \dots + X[i])/(i+1)$$

- ◆ O cálculo do vetor A de médias dos prefixos de um outro vetor X possui aplicações em análise financeira



Média de Prefixo (Quadrático)

- ◆ O algoritmo abaixo calcula médias de prefixos em tempo quadrático, aplicando a definição

Algoritmo *prefixAverages1*(X, n)

Entrada vetor X de n inteiros

Saída vetor A de médias de prefixos de X #operações

$A \leftarrow$ novo vetor de n inteiros

n

para $i \leftarrow 0$ até $n - 1$ **faça**

n

$s \leftarrow X[0]$

n

para $j \leftarrow 1$ até i **faça**

$1 + 2 + \dots + (n - 1)$

$s \leftarrow s + X[j]$

$1 + 2 + \dots + (n - 1)$

$A[i] \leftarrow s / (i + 1)$

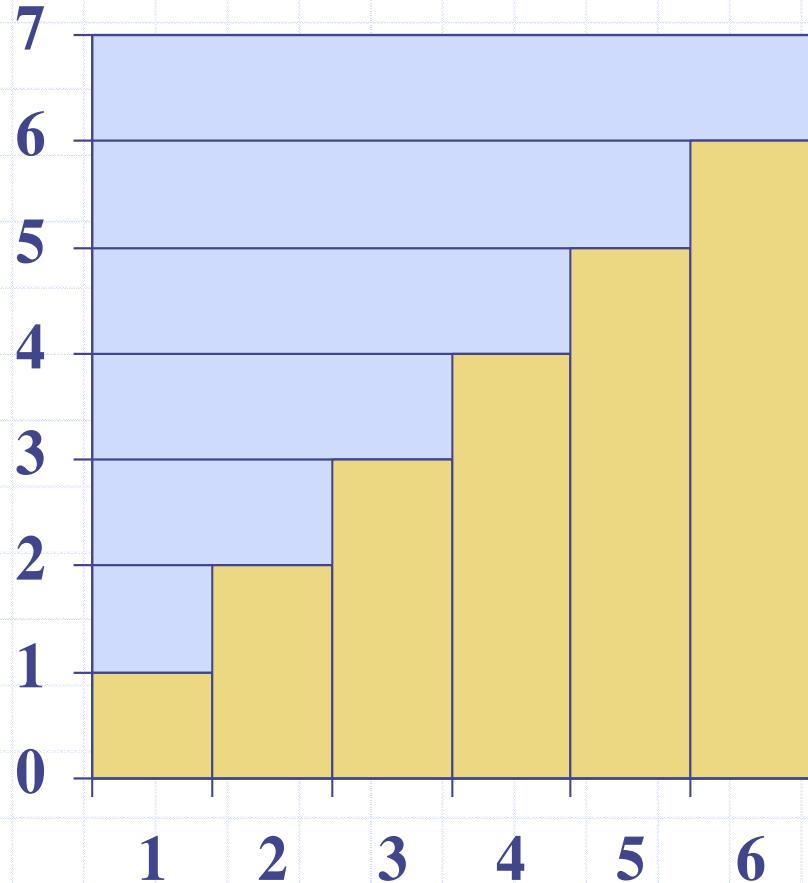
n

retorne A

1

Progressão Aritmética

- ◆ O tempo de execução de *prefixAverages1* é $O(1 + 2 + \dots + n)$
- ◆ A soma dos primeiros n inteiros é $n(n + 1) / 2$
 - Pode-se provar isso visualmente
- ◆ Portanto, o algoritmo *prefixAverages1* executa em tempo $O(n^2)$



Médias de Prefixos (Linear)

- ◆ O algoritmo abaixo calcula a média de prefixos em tempo linear, varrendo e somando de uma só vez

Algoritmo *prefixAverages2(X, n)*

Entrada vetor X de n inteiros

Saida vetor A de médias de prefixos de X #operations

$A \leftarrow$ novo vetor de n inteiros

n

$s \leftarrow 0$

1

para $i \leftarrow 0$ até $n - 1$ **faça**

n

$s \leftarrow s + X[i]$

n

$A[i] \leftarrow s / (i + 1)$

n

retorne A

1

- ◆ Algoritmo *prefixAverages2* executa em tempo $O(n)$

Pré-requisitos matemáticos

- ◆ Somatórios
- ◆ Logaritmos e Potências

- ◆ Técnicas de prova
- ◆ Probabilidade básica

◆ **propriedades de logaritmos:**

$$\log_b(xy) = \log_b x + \log_b y$$

$$\log_b(x/y) = \log_b x - \log_b y$$

$$\log_b x^a = a \cdot \log_b x$$

$$\log_b a = \log_x a / \log_x b$$

◆ **propriedades de potências:**

$$a^{(b+c)} = a^b a^c$$

$$a^{bc} = (a^b)^c$$

$$a^b / a^c = a^{(b-c)}$$

$$b = a^{\log_a b}$$

$$b^c = a^{c * \log_a b}$$

Notação:

$\lfloor x \rfloor$: o maior inteiro menor ou igual a x

$\lceil x \rceil$: o menor inteiro maior ou igual a x

Parentes do O



◆ Omega (Ω)

- $f(n)$ é $\Omega(g(n))$ se existir uma constante $c > 0$ e uma constante inteira $n_0 \geq 1$ tais que $f(n) \geq c.g(n)$ para todo $n \geq n_0$

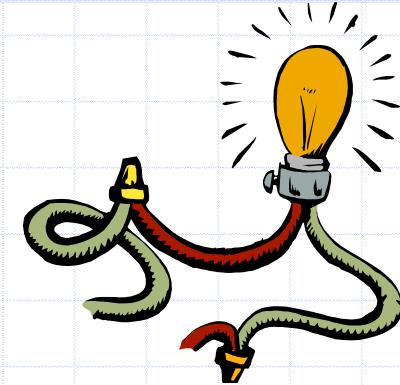
◆ Theta (Θ)

- $f(n)$ é $\Theta(g(n))$ se existirem constantes $c' > 0$ e $c'' > 0$ e uma constante inteira $n_0 \geq 1$ tais que $c'.g(n) \leq f(n) \leq c''.g(n)$ para todo $n \geq n_0$

Parentes do O

- ◆ **o** ('o' pequeno)
 - "f(n) é o(g(n))" se para **qualquer constante c > 0**, existe uma constante $n_0 > 0$ tal que $f(n) \leq c.g(n)$ para todo $n \geq n_0$.
- ◆ **ω** (omega pequeno)
 - "f(n) é ω(g(n))" se "g(n) é o(f(n))", ou seja, se para **qualquer constante c > 0**, existe uma constante $n_0 > 0$ tal que $g(n) \leq c.f(n)$ para todo $n \geq n_0$.

Trocando em miúdos ...



- $f(n)$ é $O(g(n))$ se $f(n)$ for assintoticamente **menor ou igual a** $g(n)$
- $f(n)$ é $\Omega(g(n))$ se $f(n)$ for assintoticamente **maior ou igual a** $g(n)$
- $f(n)$ é $\Theta(g(n))$ se $f(n)$ for assintoticamente **igual** a $g(n)$
- $f(n)$ é $o(g(n))$ se $f(n)$ for assintoticamente **menor que** $g(n)$
- $f(n)$ é $\omega(g(n))$ se $f(n)$ for assintoticamente **maior que** $g(n)$

Exemplos



- **$5n^2$ é $\Omega(n^2)$**

$f(n)$ é $\Omega(g(n))$ se existir uma constante $c > 0$ e uma constante inteira $n_0 \geq 1$ tais que $f(n) \geq c.g(n)$ para todo $n \geq n_0$

As constantes $c = 5$ e $n_0 = 1$ satisfazem as restrições.

- **$5n^2$ é $\Omega(n)$**

$f(n)$ é $\Omega(g(n))$ se existir uma constante $c > 0$ e uma constante inteira $n_0 \geq 1$ tais que $f(n) \geq c.g(n)$ for $n \geq n_0$

As constantes $c = 1$ e $n_0 = 1$ satisfazem as restrições.

- **$5n^2$ é $\Theta(n^2)$**

$f(n)$ é $\Theta(g(n))$ se é $\Omega(n^2)$ e $O(n^2)$. Nós já constatamos o primeiro caso. Para verificar o segundo lembre-se que $f(n)$ é $O(g(n))$ se existir uma constante $c > 0$ e uma constante inteira $n_0 \geq 1$ tais que $f(n) \leq c.g(n)$ para todo $n \geq n_0$

As constantes $c = 5$ e $n_0 = 1$ satisfazem as restrições.

Exemplos

- $f(n) = 12n^2 + 6n$ é $o(n^3)$ e também é $\omega(n)$.

Seja $c > 0$ uma constante. Se tomarmos $n_0 = (12+6)/c$ então, para $n \geq n_0$ teremos

$$c \cdot n^3 \geq 12n^2 + 6n^2 \geq 12n^2 + 6n.$$

Portanto, $f(n)$ é $o(n^3)$.

Para mostrar que $f(n)$ é $\omega(n)$ usamos $n_0 = c/12$, para uma dada constante c qualquer. Então, para $n \geq n_0$, teremos $12n^2 + 6n \geq 12n^2 + cn$.

Portanto, $f(n)$ é $\omega(n)$.