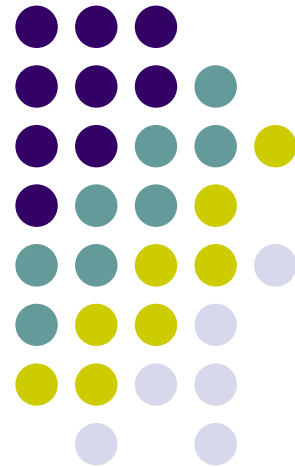


Análise de algoritmos

Parte I





Procedimento X Algoritmo

- **Procedimento**: sequência *finita* de *instruções*, que são operações *claramente descritas*, e que podem ser executadas *mecanicamente*, em tempo *finito*.
 - *mecanicamente*: não há dúvidas sobre o que deve ser feito;
 - *em tempo finito*: não há dúvidas de que a tarefa correspondente à instrução pode, em qualquer caso, ser levada até sua conclusão.
- Exemplo 1:

1. se $i = 0$, pare e diga “sim”.
2. diminua o valor de i de duas unidades.
3. vá para 1.

Note que se i for originalmente ímpar ou negativo, o procedimento não pára, embora cada instrução seja finita, conforme a definição. Esse procedimento não poderia servir de instrução para outro procedimento, pois não seria garantido seu término.



Procedimento X Algoritmo

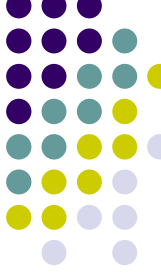
- Exemplo 2:

1. se $i = 0$, pare e diga “sim”.
2. se $i < 0$, pare e diga “não”.
3. diminua o valor de i de duas unidades.
4. vá para 1.

O procedimento acima pára e diz “sim” se o número inteiro i for par e não negativo; pára e diz “não” nos demais casos.

- **Algoritmo**: é um procedimento que sempre pára, quaisquer que sejam os valores de suas entradas.

Procedimento, Algoritmo e Computabilidade



- Para estudar o processo de computação do ponto de vista teórico, com a finalidade de caracterizar o que é ou não computável, é necessário introduzir um **modelo matemático** que represente o que se entende por computação.
- Cada matemático tem seu modelo favorito: máquinas de Turing, funções recursivas parciais, baseados em algoritmos de Markov, em linguagens de programações convencionais, etc.
- Conjectura de Alonzo Church: Todos os modelos razoáveis do processo de computação, definidos e por definir, são equivalentes (**Tese de Church**).
- Neste curso, usaremos a **Máquina de Turing (MT)** como modelo de computabilidade, ou seja, *Uma MT será capaz de executar uma função se e somente se essa função for computável.*

Recursos usados por um algoritmo



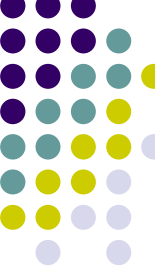
- Uma vez que um algoritmo está pronto/disponível, é importante determinar os **recursos necessários** para sua execução
 - Tempo
 - Memória
- Qual o principal quesito? Por que?



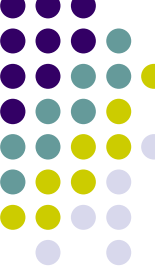
Análise de algoritmos

- Um algoritmo que soluciona um determinado problema, mas requer o processamento de **um ano**, não deve ser usado
- O que dizer de uma afirmação como a abaixo?
 - “Desenvolvi um novo algoritmo chamado TripleX que leva 14,2 segundos para processar 1.000 números, enquanto o método SimpleX leva 42,1 segundos”
- Você trocaria o SimpleX que roda em sua empresa pelo TripleX?

Análise de algoritmos



- A afirmação tem que ser examinada, pois há diversos fatores envolvidos
 - Características da máquina em que o algoritmo foi testado
 - Quantidade de memória
 - Linguagem de programação
 - Compilada vs. interpretada
 - Alto vs. baixo nível
 - Implementação pouco cuidadosa do algoritmo SimpleX vs. “super” implementação do algoritmo TripleX
 - Quantidade de dados processados
 - Se o TripleX é mais rápido para processar 1.000 números, ele também é mais rápido para processar quantidades maiores de números, certo? Mas, quanto mais rápido?



Análise de algoritmos

- A comunidade de computação começou a pesquisar formas de comparar algoritmos de forma independente de
 - Hardware
 - Linguagem de programação
 - Habilidade do programador
- Portanto, quer-se comparar **algoritmos** e não **programas**
 - Área conhecida como “análise/complexidade de algoritmos”



Eficiência de algoritmos

- Sabe-se que
 - Processar 10.000 números leva mais tempo do que 1000 números
 - Cadastrar 10 pessoas em um sistema leva mais tempo do que cadastrar 5
 - Etc.
- Então, pode ser uma boa idéia estimar a eficiência de um algoritmo em função do tamanho do problema
 - Em geral, assume-se que “n” é o tamanho do problema ou da entrada, ou ainda o número de elementos que serão processados
 - E calcula-se o número de operações que serão realizadas sobre os n elementos, ou seja, em função de n.



Eficiência de algoritmos

- O melhor algoritmo é aquele que requer menos operações sobre a entrada, pois é o mais rápido
 - O tempo de execução do algoritmo pode variar em diferentes máquinas, mas o número de operações é uma boa medida de desempenho de um algoritmo
- De que **operações** estamos falando?
- Toda operação leva o **mesmo tempo**?



Exemplo: TripleX vs. SimpleX

- TripleX: para uma entrada de tamanho n , o algoritmo realiza n^2+n operações
 - Pensando em termos de função: $f(n)=n^2+n$
- SimpleX: para uma entrada de tamanho n , o algoritmo realiza $1.000n$ operações
 - $g(n)=1.000n$



Exemplo: TripleX vs. SimpleX

- Faça os cálculos do desempenho de cada algoritmo para cada tamanho de entrada

Tamanho da entrada (n)	1	10	100	1.000	10.000
$f(n)=n^2+n$					
$g(n)=1.000n$					



Exemplo: TripleX vs. SimpleX

- Faça os cálculos do desempenho de cada algoritmo para cada tamanho de entrada

Tamanho da entrada (n)	1	10	100	1.000	10.000
$f(n)=n^2+n$	2	110	10.100	1.001.000	100.010.000
$g(n)=1.000n$	1.000	10.000	100.000	1.000.000	10.000.000

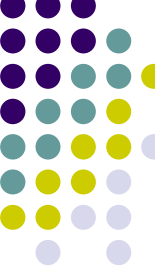
- A partir de $n=1.000$, $f(n)$ mantém-se maior e cada vez mais distante de $g(n)$
 - Diz-se que $f(n)$ cresce mais rápido do que $g(n)$



Análise assintótica

- Deve-se preocupar com a eficiência de algoritmos quando o tamanho de n for **grande**
- Definição: a eficiência assintótica de um algoritmo descreve sua eficiência relativa quando n torna-se grande
- Portanto, para **comparar** 2 algoritmos, determinam-se as **taxas de crescimento** de cada um: o algoritmo com menor taxa de crescimento rodará mais rápido quando o tamanho do problema for grande

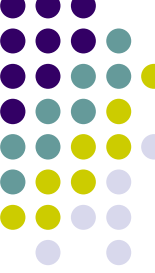
Análise assintótica



- Atenção

- Algumas funções podem não crescer com o valor de n
 - Quais?
- Também se pode aplicar os conceitos de análise assintótica para a quantidade de memória usada por um algoritmo
 - Mas não é tão útil, pois é difícil estimar os detalhes exatos do uso de memória e seu impacto

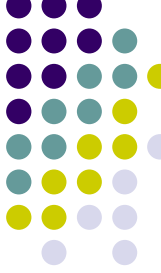
Relembrando um pouco de matemática...



- Exponentes

- $x^a x^b = x^{a+b}$
- $x^a / x^b = x^{a-b}$
- $(x^a)^b = x^{ab}$
- $x^n + x^n = 2x^n$ (diferente de x^{2n})
- $2^n + 2^n = 2^{n+1}$

Relembrando um pouco de matemática...



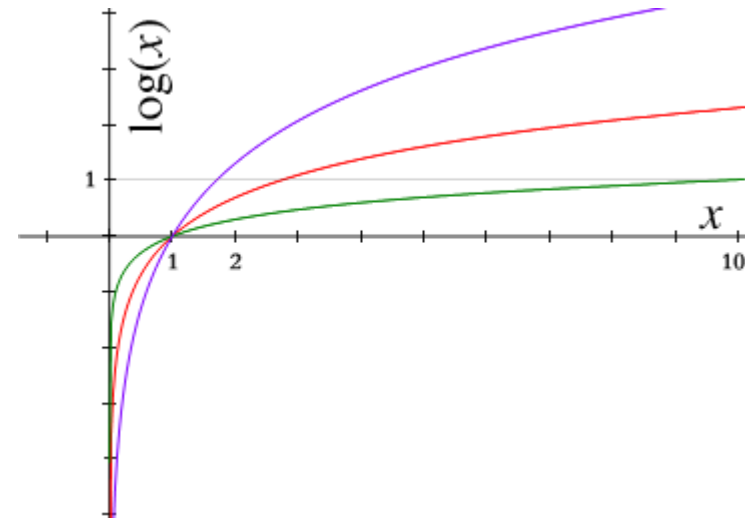
- **Logaritmos** (usaremos a base 2, a menos que seja dito o contrário)
 - $x^a=b \rightarrow \log_x b=a$
 - $\log_a b = \log_c b / \log_c a$, se $c>0$
 - $\log ab = \log a + \log b$
 - $\log a/b = \log a - \log b$
 - $\log(a^b) = b \log a$

Relembrando um pouco de matemática...



- **Logaritmos** (usaremos a base 2, a menos que seja dito o contrário)
 - E o mais importante
 - $\log x < x$ para todo $x > 0$

- Alguns valores
 - $\log 1=0$, $\log 2=1$,
 $\log 1.024=10$,
 $\log 1.048.576=20$



Exemplo para várias bases

Função exponencial vs. logarítmica



- Na palma da mão direita



Relembrando um pouco de matemática...



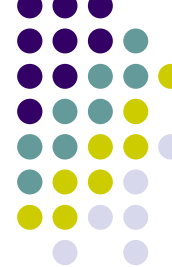
- Séries (demonstre a igualdade)

$$\sum_{i=0}^n 2^i = 2^{n+1} - 1$$

$$\sum_{i=0}^n a^i = \frac{a^{n+1} - 1}{a - 1}$$

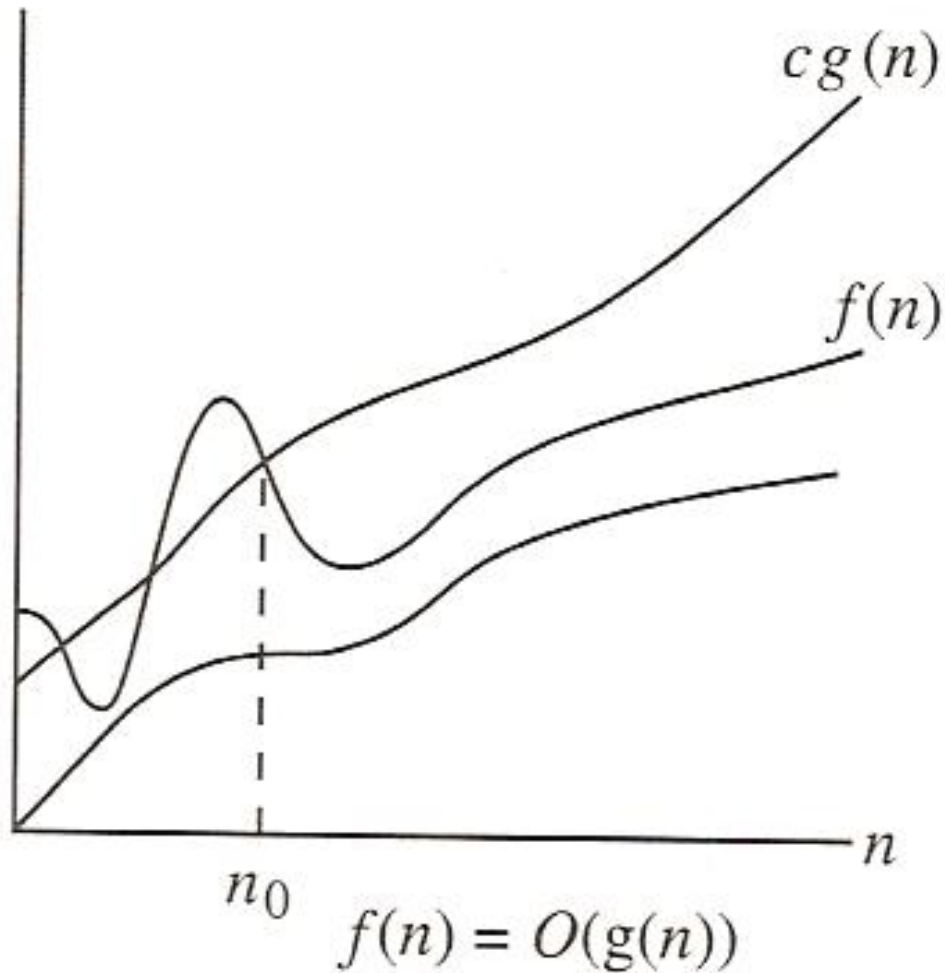
$$\sum_{i=1}^n i = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$$

Algumas notações



- Notações que usaremos na análise de algoritmos
 - $T(n) = O(f(n))$ (lê-se *big-oh*, *big-o* ou “da ordem de”) se existirem constantes c e n_0 tal que $T(n) \leq c * f(n)$ quando $n \geq n_0$
 - A taxa de crescimento de $T(n)$ é menor ou igual à taxa de $f(n)$. $T(n)$ é limitada superiormente por $f(n)$.

Algumas notações

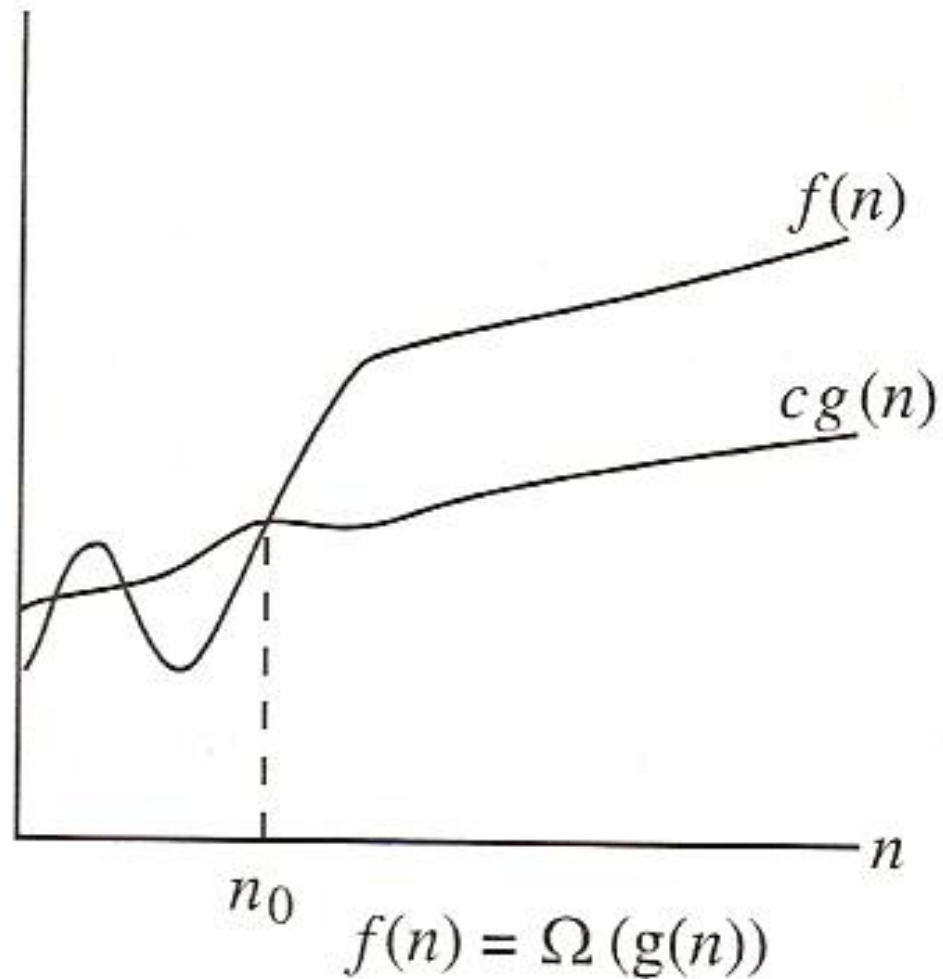


Algumas notações



- Notações que usaremos na análise de algoritmos
 - $T(n) = \Omega(f(n))$ (lê-se “ômega”) se existirem constantes c e n_0 tal que $T(n) \geq c * f(n)$ quando $n \geq n_0$
 - A taxa de crescimento de $T(n)$ é maior ou igual à taxa de $f(n)$. $T(n)$ é limitada inferiormente por $f(n)$.

Algumas notações

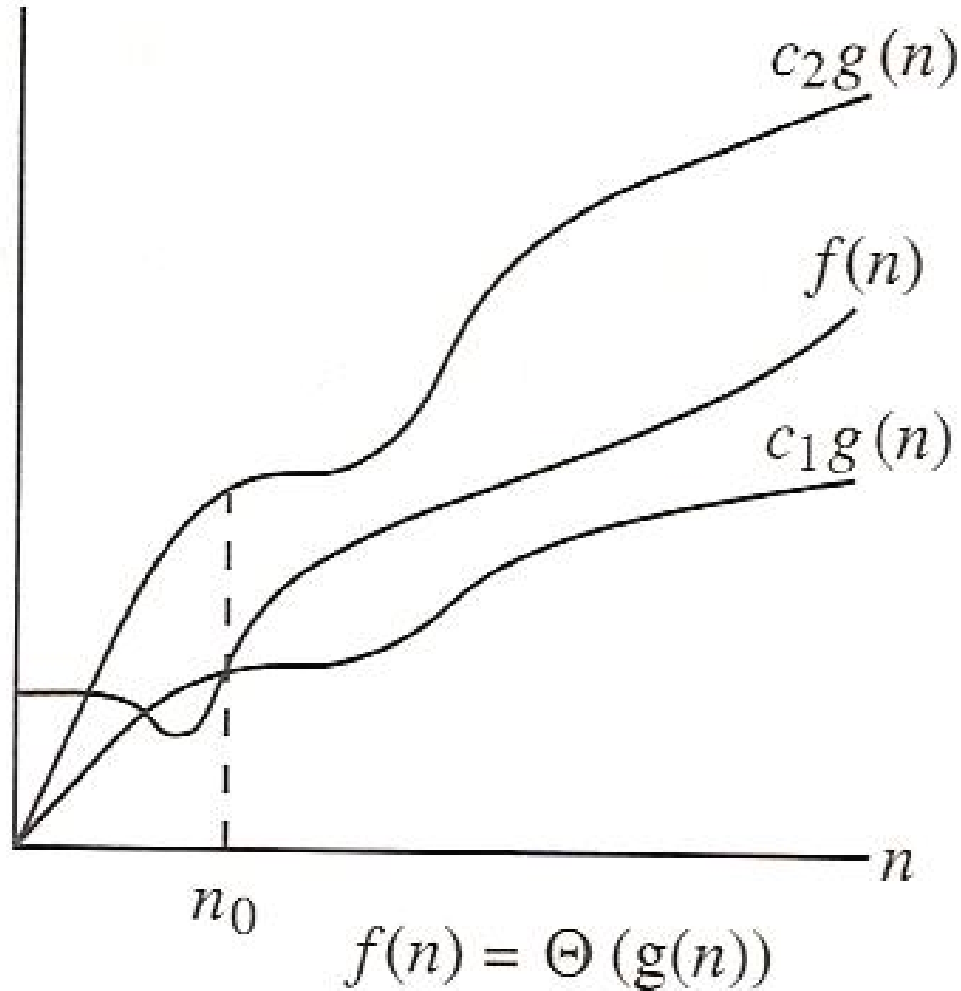


Algumas notações



- Notações que usaremos na análise de algoritmos
 - $T(n) = \Theta(f(n))$ (lê-se “theta”) se e somente se $T(n) = O(f(n))$ e $T(n) = \Omega(f(n))$
 - A taxa de crescimento de $T(n)$ é igual à taxa de $f(n)$

Algumas notações





Algumas notações

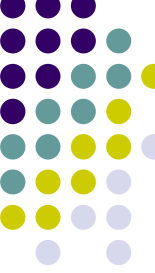
- O uso das notações permite **comparar a taxa de crescimento** das funções correspondentes aos algoritmos
 - **Não faz sentido comparar pontos isolados** das funções, já que podem não corresponder ao comportamento assintótico

Exemplo



• Para 2 algoritmos quaisquer, considere as funções de eficiência correspondentes a $1.000n$ e n^2

- A primeira é maior do que a segunda para valores pequenos de n
- A segunda cresce mais rapidamente e eventualmente será uma função maior, sendo que o ponto de mudança é $n=1.000$
- Segundo as notações anteriores, se existe um ponto n_0 a partir do qual $c \cdot f(n)$ é sempre pelo menos tão grande quanto $T(n)$, então, se os fatores constantes forem ignorados, $f(n)$ é pelo menos tão grande quanto $T(n)$
 - No nosso caso, $T(n)=1.000n$, $f(n)=n^2$, $n_0=1.000$ e $c=1$ (ou, ainda, $n_0=10$ e $c=100$)
 - Dizemos que $1.000n=O(n^2)$



Outros exemplos

- A função n^3 cresce mais rapidamente que n^2
 - $n^2 = O(n^3)$
 - $n^3 = \Omega(n^2)$
- Se $f(n)=n^2$ e $g(n)=2n^2$, então essas duas funções têm taxas de crescimento iguais
 - Portanto, $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$

Taxas de crescimento



- Algumas regras
 - Se $T_1(n) = O(f(n))$ e $T_2(n) = O(g(n))$, então
 - $T_1(n) + T_2(n) = \max(O(f(n)), O(g(n)))$
 - $T_1(n) * T_2(n) = O(f(n) * g(n))$
 - Para que precisamos desse tipo de cálculo?

Taxas de crescimento



- Algumas regras

- Se $T(x)$ é um polinômio de grau n , então

- $T(x) = \Theta(x^n)$

- Relembrando: um polinômio de grau n é uma função que possui a forma abaixo

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots a_1 x + a_0$$

segundo a seguinte classificação em função do grau

- Grau 0: polinômio constante
- Grau 1: polinômio linear
- Grau 2: polinômio quadrático
- Grau 3: polinômio cúbico

Se $f(x)=0$, tem-se o polinômio nulo

Taxas de crescimento



- Algumas regras
 - $\log^k n = O(n)$ para qualquer constante k , pois logaritmos crescem muito vagarosamente

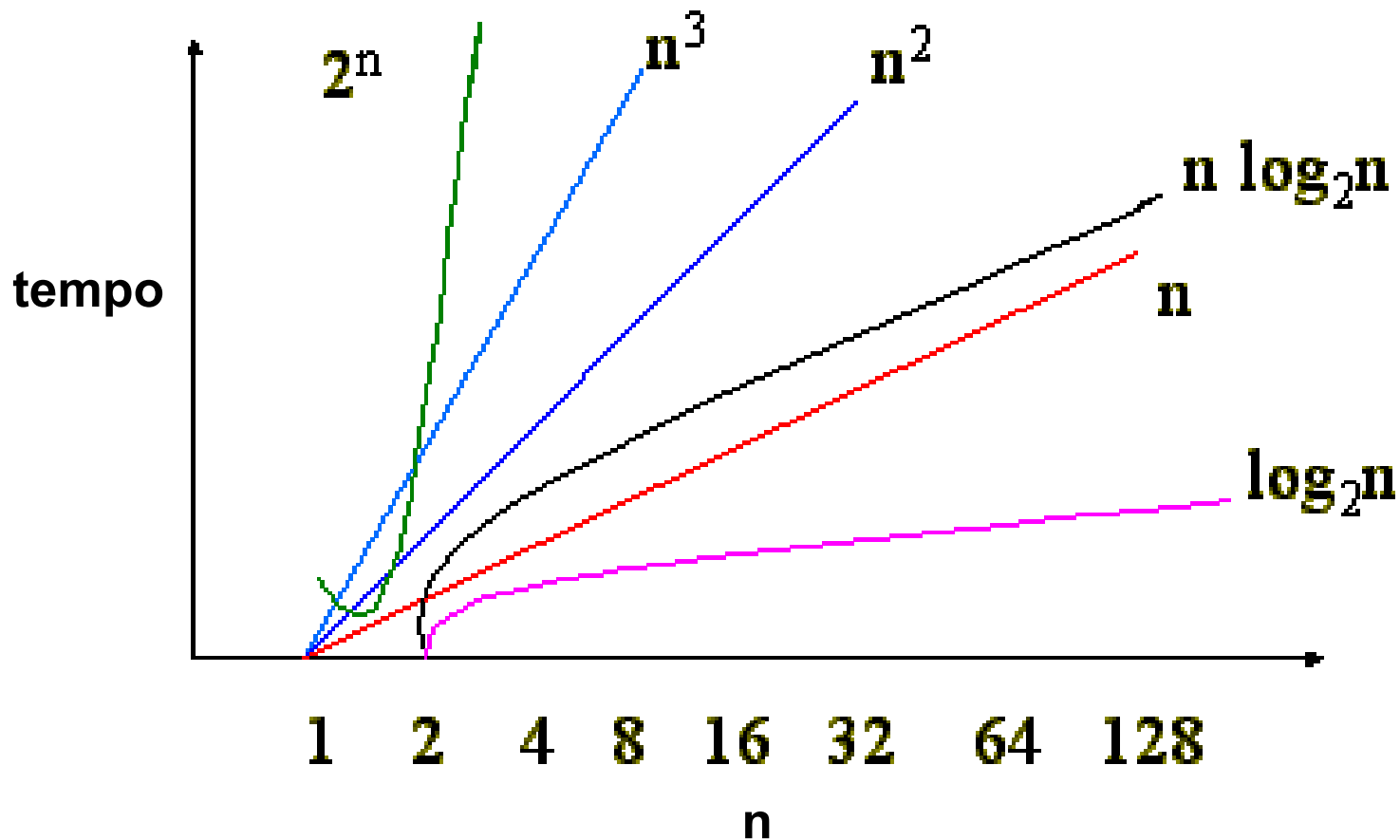
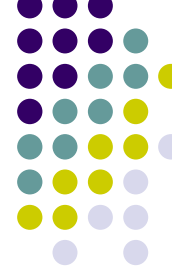
Funções e taxas de crescimento



- As mais comuns

Função	Nome
c	constante
$\log n$	logarítmica
$\log^2 n$	log quadrado
n	linear
$n \log n$	quadrática
n^2	
n^3	cúbica
2^n	exponencial
a^n	

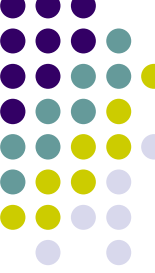
Funções e taxas de crescimento



Relações entre as Ordens

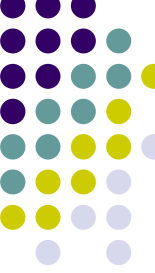


Função de Complexidade	Tamanho do problema n			
	10	10^2	10^3	10^4
$\log_2 n$	3.3	6.6	10	13.3
n	10	10^2	10^3	10^4
$n \log_2 n$	$0.33 \cdot 10^2$	$0.17 \cdot 10^3$	10^4	$1.3 \cdot 10^5$
n^2	10^2	10^4	10^6	10^8
n^3	10^3	10^6	10^9	10^{12}
2^n	1024	$1.3 \cdot 10^{30}$	$> 10^{100}$	$> 10^{100}$
$n!$	$3 \cdot 10^6$	$> 10^{100}$	$> 10^{100}$	$> 10^{100}$



Taxas de crescimento

- Apesar de, as vezes, ser importante, **não se costuma incluir constantes ou termos de menor ordem em taxas de crescimento**
 - Queremos medir a taxa de crescimento da função, o que torna os “termos menores” irrelevantes
 - As constantes também dependem do tempo exato de cada operação; como ignoramos os custos reais das operações, ignoramos também as constantes
- Não se diz que $T(n) = O(2n^2)$ ou que $T(n) = O(n^2+n)$
 - Diz-se apenas $T(n) = O(n^2)$



Exercício

- Um algoritmo tradicional e muito utilizado é da ordem de $n^{1,5}$, enquanto um algoritmo novo proposto recentemente é da ordem de $n \log n$
 - $f(n)=n^{1,5}$
 - $g(n)=n \log n$
- Qual algoritmo você adotaria?
 - Lembre-se que a eficiência desse algoritmo pode determinar o sucesso ou o fracasso de aplicação



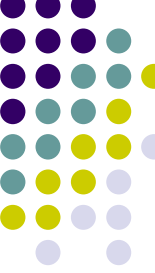
Exercício em duplas

- Uma possível solução

- $f(n) = n^{1,5} \quad \rightarrow \quad n^{1,5}/n = n^{0,5} \quad \rightarrow \quad (n^{0,5})^2 = n$

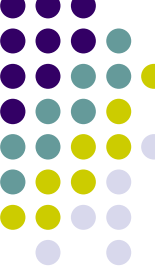
- $g(n) = n \log n \quad \rightarrow \quad (n \log n)/n = \log n \quad \rightarrow \quad (\log n)^2 = \log^2 n$

- Como n cresce mais rapidamente do que qualquer potência de \log , temos que o algoritmo novo é mais eficiente e, portanto, deve ser o adotado. Faça uma tabela de valores para as duas funções e comprove!



Análise de algoritmos

- Para proceder a uma análise de algoritmos e determinar as taxas de crescimento, necessitamos de um **modelo de computador** e das **operações** que executa
- Assume-se o uso de um **computador tradicional**, em que as **instruções** de um programa são executadas sequencialmente
 - Com **memória infinita**, por simplicidade



Análise de algoritmos

- Repertório de **instruções simples**: soma, multiplicação, comparação, atribuição, etc.
 - Por simplicidade e viabilidade da análise, assume-se que **cada instrução demora exatamente uma unidade de tempo** para ser executada
 - Obviamente, em situações reais, isso pode não ser verdade: a leitura de um dado em disco pode demorar mais do que uma soma
 - Operações complexas, como inversão de matrizes e ordenação de valores, não são realizadas em uma única unidade de tempo, obviamente: devem ser analisadas em partes



Análise de algoritmos

- Considera-se somente o algoritmo e suas entradas (de tamanho n)
- Para uma entrada de tamanho n , pode-se calcular $T_{\text{melhor}}(n)$, $T_{\text{média}}(n)$ e $T_{\text{pior}}(n)$, ou seja, o melhor tempo de execução, o tempo médio e o pior, respectivamente
 - Obviamente, $T_{\text{melhor}}(n) \leq T_{\text{média}}(n) \leq T_{\text{pior}}(n)$
- Atenção: para mais de uma entrada, essas funções teriam mais de um argumento

Análise de algoritmos



- Geralmente, utiliza-se somente a análise do pior caso $T_{\text{pior}}(n)$, pois ela fornece os limites para todas as entradas, incluindo particularmente as entradas ruins
- Logicamente, muitas vezes, o tempo médio pode ser útil, principalmente em sistemas executados rotineiramente
 - Por exemplo: em um sistema de cadastro de alunos como usuários de uma biblioteca, o trabalho difícil de cadastrar uma quantidade enorme de pessoas é feito somente uma vez; depois, cadastros são feitos de vez em quando apenas
- Dá mais trabalho calcular o tempo médio
- O melhor tempo não tem muita utilidade

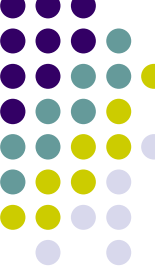


Pergunta

- Idealmente, para um algoritmo qualquer de ordenação de vetores com n elementos:
 - Qual a configuração do vetor que você imagina que provavelmente geraria o melhor tempo de execução?
 - E qual geraria o pior tempo?

Exemplo

- Soma da subsequência máxima
 - Dada uma seqüência de inteiros (possivelmente negativos) a_1, a_2, \dots, a_n , encontre o valor da máxima soma de quaisquer números de elementos consecutivos; se todos os inteiros forem negativos, o algoritmo deve retornar 0 como resultado da maior soma
 - Por exemplo, para a entrada -2, 11, -4, 13, -5 e -2, a resposta é 20 (soma de a_2 a a_4)





Soma da subsequência máxima

- Há muitos algoritmos propostos para resolver esse problema
- Alguns são mostrados abaixo juntamente com seus tempos de execução

Algoritmo	1	2	3	4
Tempo	$O(n^3)$	$O(n^2)$	$O(n \log n)$	$O(n)$
Tamanho da entrada				
n = 10	0.00103	0.00045	0.00066	0.00034
n = 100	0.47015	0.01112	0.00486	0.00063
n = 1.000	448.77	1.1233	0.05843	0.00333
n = 10.000	ND*	111.13	0.68631	0.03042
n = 100.000	ND	ND	8.0113	0.29832

*ND = Não Disponível



Soma da subsequência máxima

- Deve-se notar que
 - Para entradas pequenas, todas as implementações rodam num piscar de olhos
 - Portanto, se somente entradas pequenas são esperadas, não devemos gastar nosso tempo para projetar melhores algoritmos
 - Para entradas grandes, o melhor algoritmo é o 4
 - Os tempos não incluem o tempo requerido para leitura dos dados de entrada
 - Para o algoritmo 4, o tempo de leitura é provavelmente maior do que o tempo para resolver o problema: característica típica de algoritmos eficientes

Taxas de crescimento

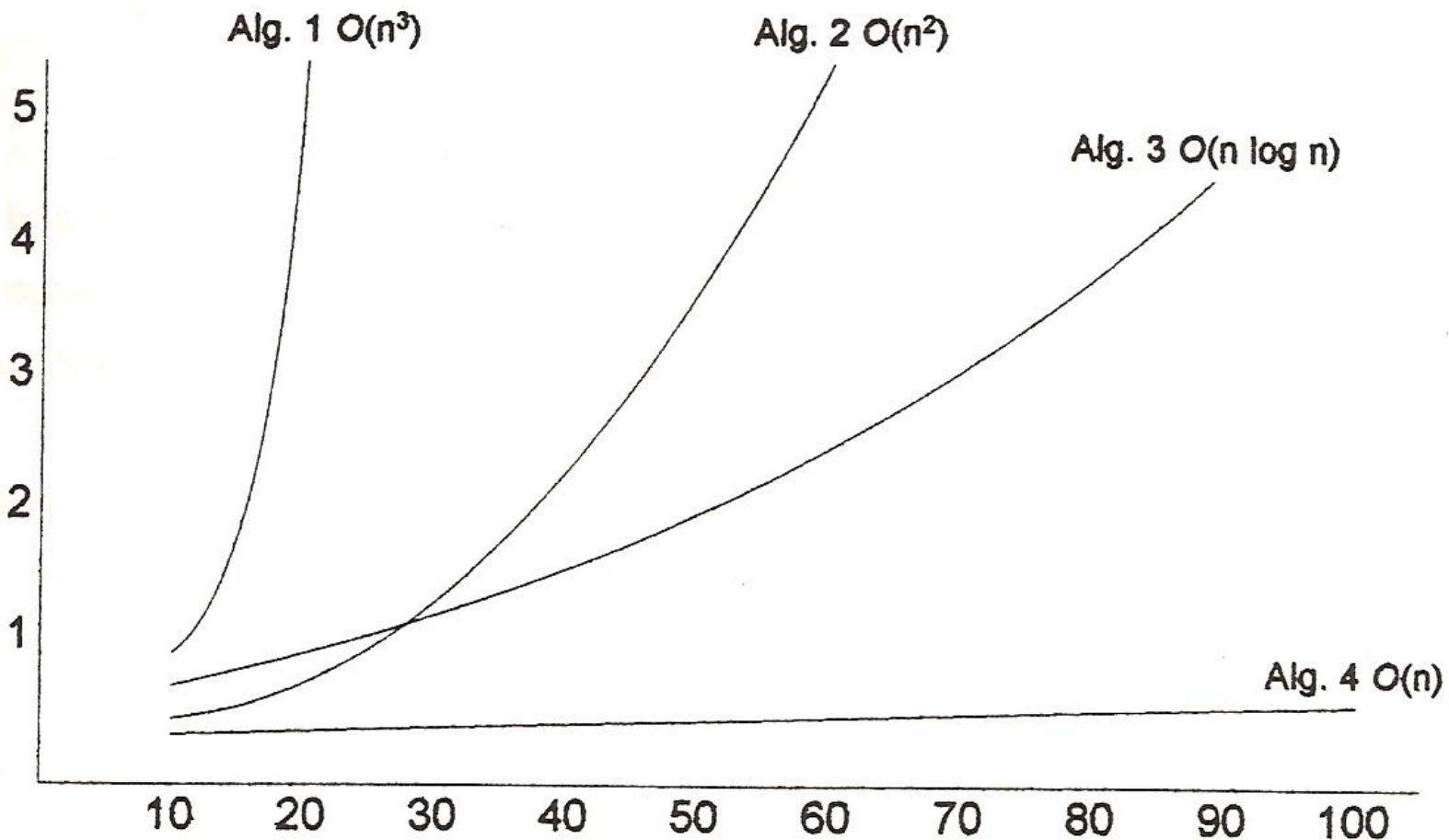


Gráfico (n x milisegundos) das taxas de crescimento dos 4 algoritmos com entradas entre 10 e 100

Taxas de crescimento

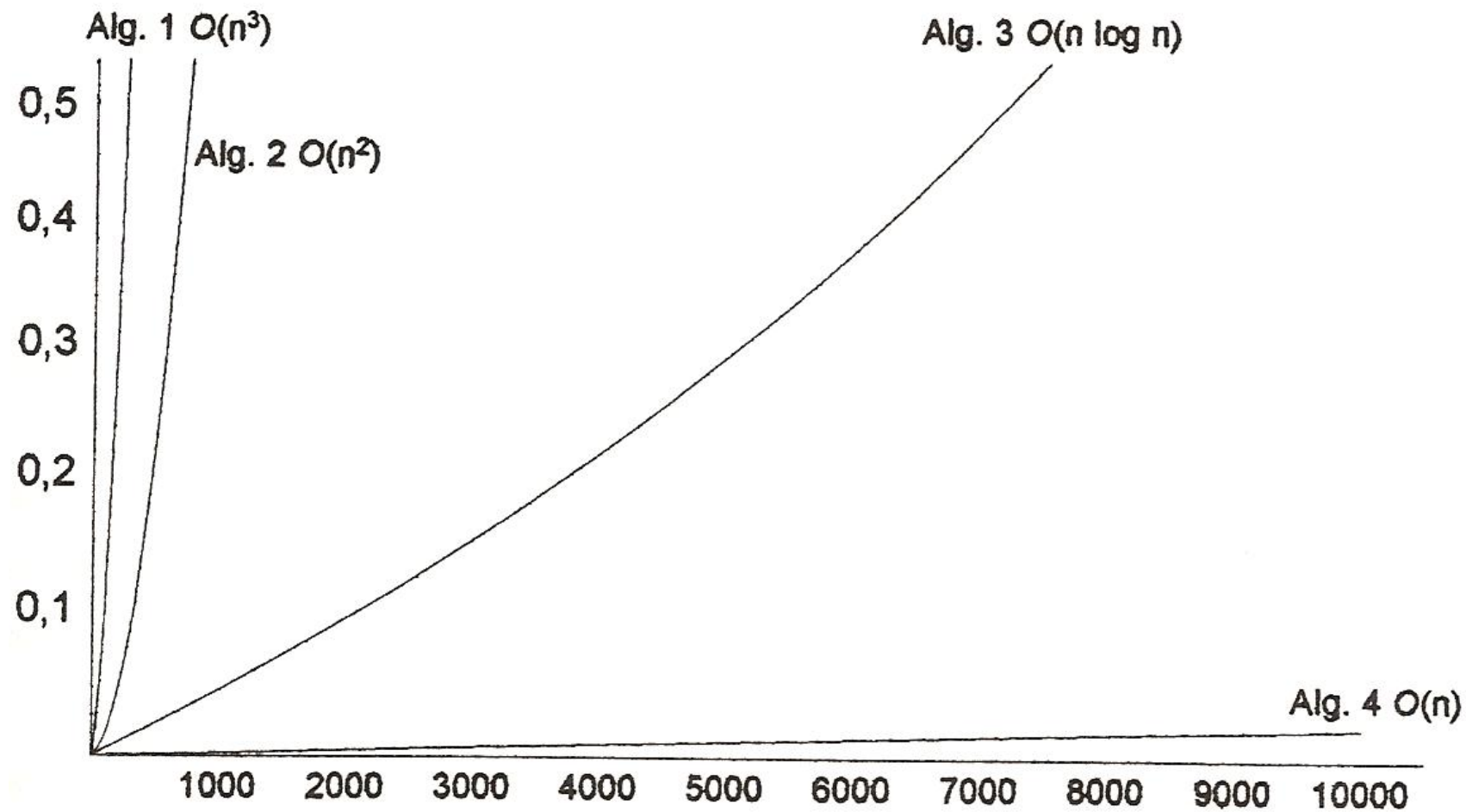
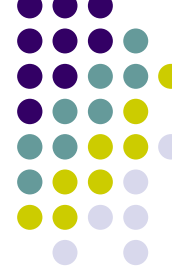


Gráfico (n x segundos) dos 4 algoritmos para entradas maiores