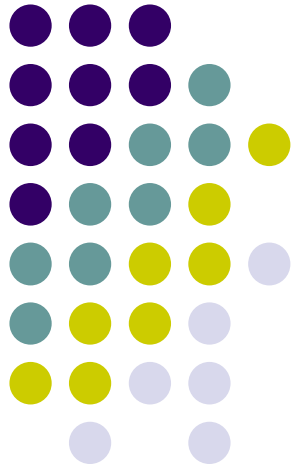
Análise de algoritmos

Introdução à Ciência da Computação 2

Baseado nos slides do Prof. Thiago A. S. Pardo



Algoritmo

- Noção geral: conjunto de instruções que devem ser seguidas para solucionar um determinado problema

- Cormen et al. (2002)

  - Qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores de **entrada** e produz algum valor ou conjunto de valores de **saída**

  - Ferramenta para resolver um problema computacional bem especificado

  - Assim como o hardware de um computador, constitui uma tecnologia, pois o desempenho total do sistema depende da escolha de um algoritmo eficiente tanto quanto da escolha de um hardware rápido

Algoritmo

- Comen et al. (2002)
  - Deseja-se que um algoritmo termine e seja correto

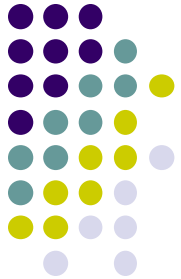
- Perguntas
  - Mas um algoritmo correto vai terminar, não vai?
  -

Exemplo

- Versão recursiva vs. iterativa do algoritmo de Fibonacci
  - Estimativa de tempo (Brassard e Bradley, 1996)

n	***10***	***20***	***30***	***50***	***100***
Recursão	8 ms	1 s	2 min	21 dias	10^9 anos
Iteração	1/6 ms	1/3 ms	1/2 ms	3/4 ms	1,5 ms

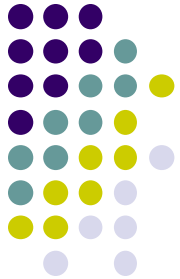
Recursos de um algoritmo



- 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

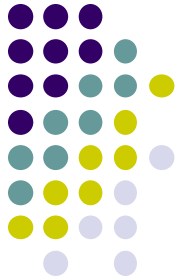


- Um algoritmo que soluciona um determinado problema, mas requer o processamento de um ano, não deve ser usado

- 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"

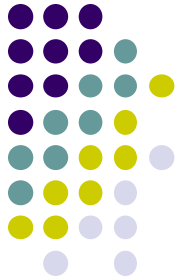
    - Você trocaria o SimpleX que roda em sua empresa pelo TripleX?

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
  - 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?

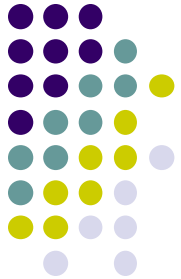
Análise de algoritmos



- A comunidade de computação começou a pesquisar formas de comparar algoritmos de forma independente de
  - Hardware
  - Linguagem de programação
  - Habilidade do programador

- Portanto, deseja-se comparar algoritmos e não programas
  - Área conhecida como "análise/complexidade de algoritmos"

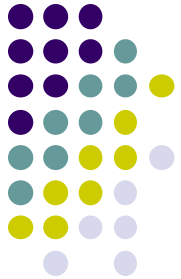
Eficiência de algoritmos



- 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.

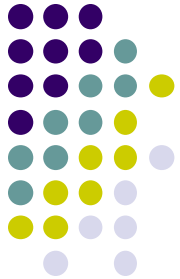
- Então, pode ser uma boa idéia estimar a eficiência de um algoritmo em função do tamanho do problema
  - Em geral, assume-se que “n” é o tamanho do problema, ou número de elementos que serão processados
  - E calcula-se o número de operações que serão realizadas sobre os n elementos

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?

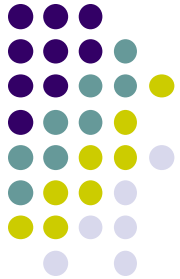
Exemplo: TripleX vs. SimpleX



- 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
  - $g(n)=1.000n$

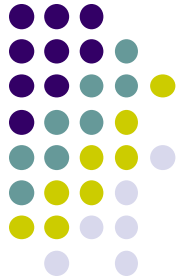
Exemplo: TripleX vs. SimpleX



- 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

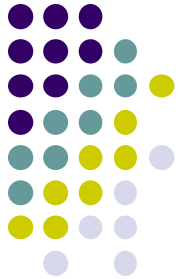


- Faça os cálculos do desempenho de cada algoritmo para cada tamanho de entrada

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

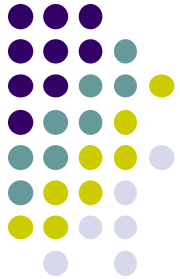


- Deve-se preocupar com a eficiência de algoritmos quando o **tamanho de n for grande**

- Definição: *a eficiência assintótica de um algoritmo descreve a eficiência relativa dele quando n torna-se grande*
  - *Pode não valer para quando a entrada for pequena*

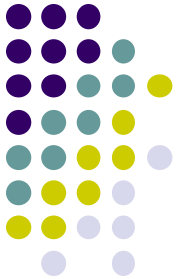
- Portanto, para comparar 2 algoritmos, determinam-se as taxas de crescimento de cada um: o algoritmo com menor taxa de crescimento rodará mais rápido quando o tamanho do problema for grande

Análise assintótica



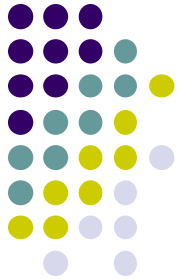
- Atenção
  - Algumas funções podem não crescer com o valor de n
    - Quais? Exemplo de um problema?
  - Também se pode aplicar os conceitos de análise assintótica para a quantidade de memória usada por um algoritmo
    - Mas não é tão útil, pois é difícil estimar os detalhes exatos do uso de memória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

Relembrando um pouco de matemática...



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

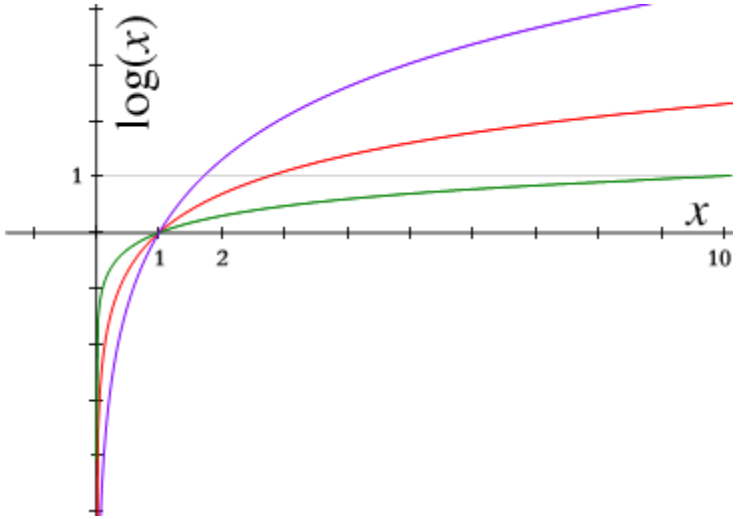
Relembrando um pouco de matemática...



- 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 $c>0$
  - $\log ab = \log a + \log b$
  - $\log a/b = \log a - \log b$
  - $\log(a^b) = b \log a$

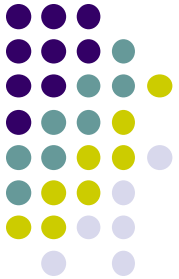
Relembrando um pouco de matemática...

- 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

Relembrando um pouco de matemática...



- Séries

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

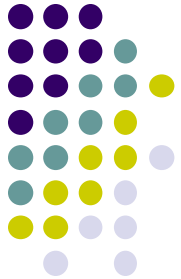
$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$$

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

Algumas notações

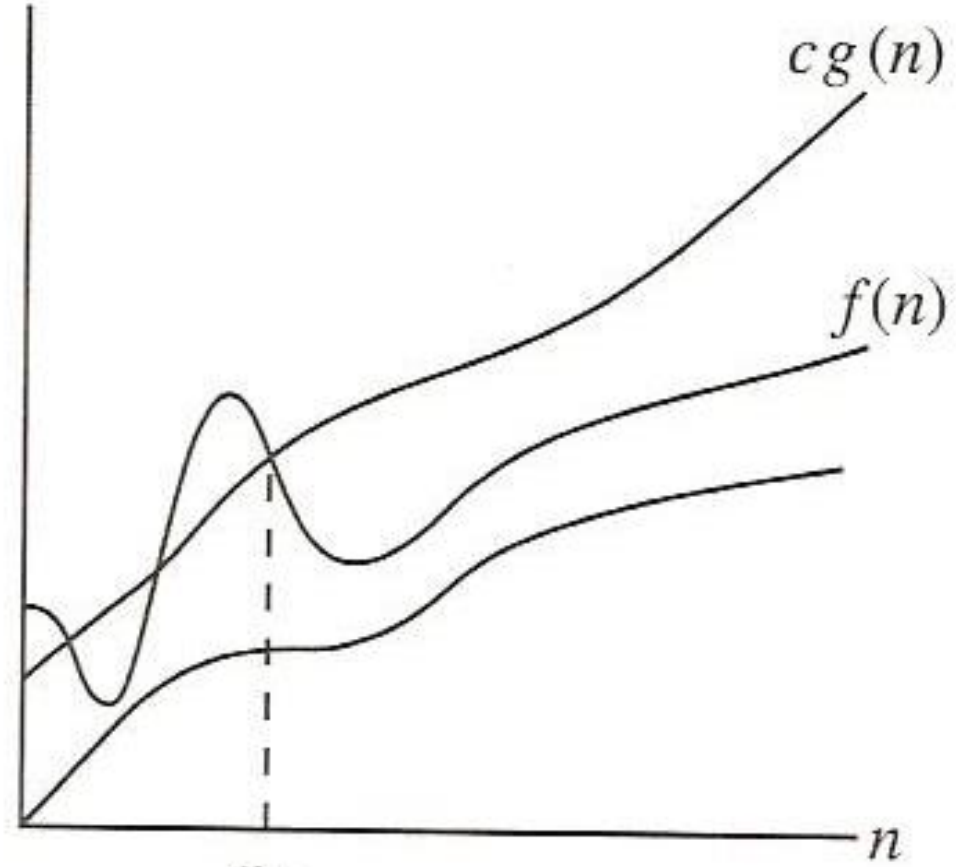
- Notações que usaremos na análise de algoritmos
  - $T(n) = O(f(n))$ (lê-se *big-oh*, *big-o* ou "da ordem de") se existirem constantes c e n_0 tal que $T(n) \leq c * f(n)$ quando $n \geq n_0$
    - A taxa de crescimento de T(n) é menor ou igual à taxa de f(n)
  - $T(n) = \Omega(f(n))$ (lê-se "ômega")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

Algumas notações



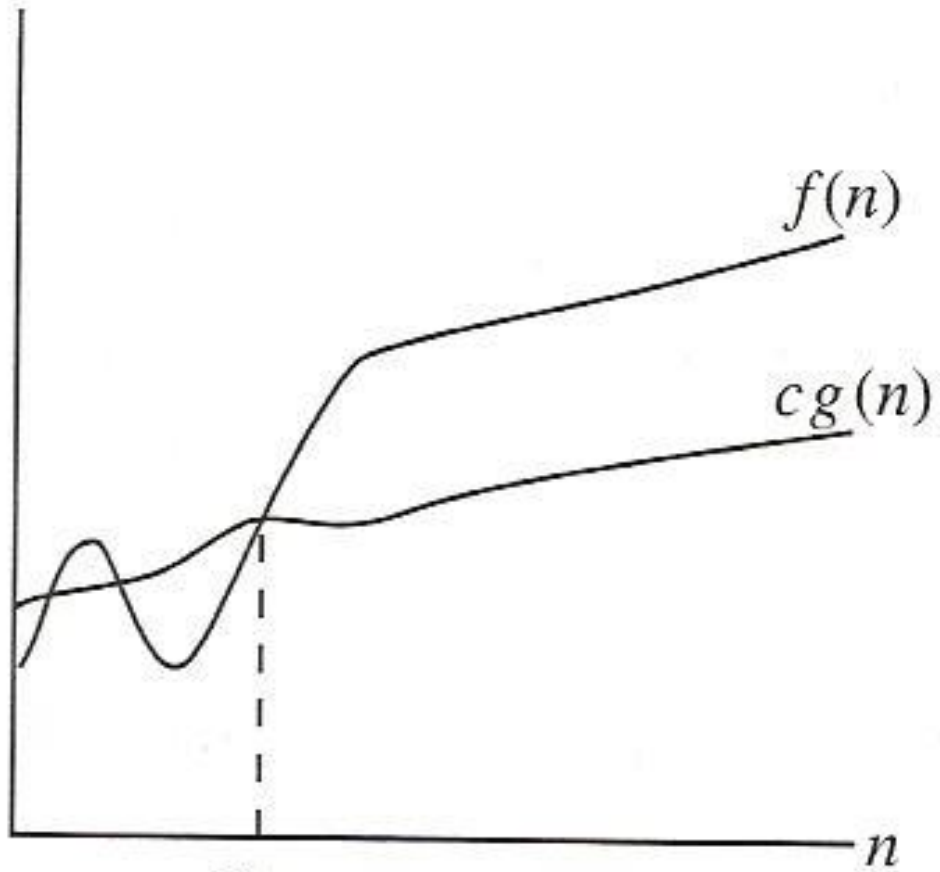
- Notações que usaremos na análise de algoritmos
  - $T(n) = \Theta(f(n))$ (lê-se “theta”) se e somente se $T(n) = O(f(n))$ e $T(n) = \Omega(f(n))$
    - A taxa de crescimento de $T(n)$ é igual à taxa de $f(n)$
  - $T(n) = o(f(n))$ (lê-se *little-oh* ou *little-o*) se e somente se $T(n) = O(f(n))$ e $T(n) \neq \Theta(f(n))$
    - A taxa de crescimento de $T(n)$ é menor do que a taxa de $f(n)$

Algumas notações



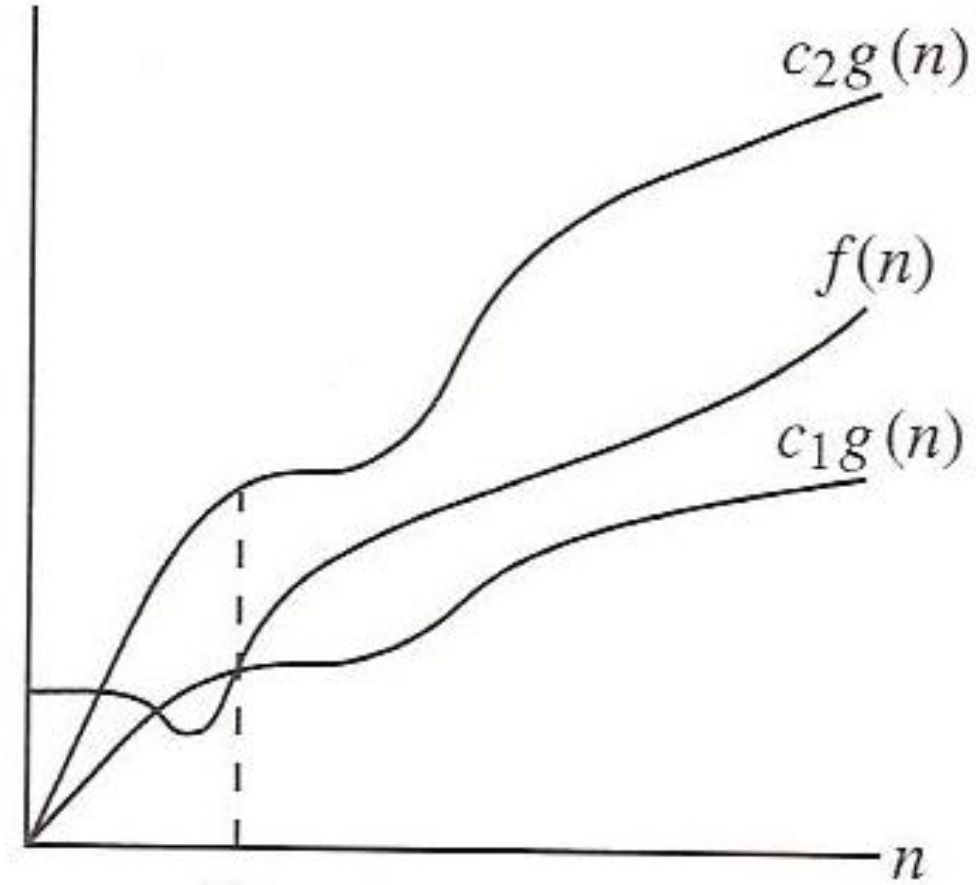
$f(n) = O(g(n))$

Algumas notações



$f(n) = \Omega(g(n))$

Algumas notações

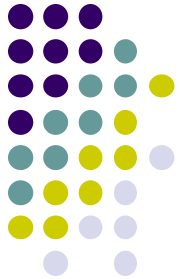


$f(n) = \Theta(g(n))$

Algumas notações

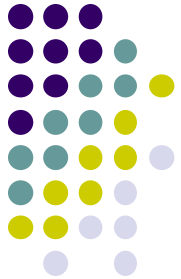
- O uso das notações permite comparar a taxa de crescimento das funções correspondentes aos algoritmos
  - Não faz sentido comparar pontos isolados das funções, já que podem não corresponder ao comportamento assintótico
  - Pode ser utilizado em outros contextos (não apenas para o tempo de execução)

Exemplo



- Para 2 algoritmos quaisquer, considere as funções de eficiência correspondentes 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
  - Existe uma constante c e um ponto n_0 a partir do qual T(n) é menor ou igual a c*f(n)
    - No nosso caso, T(n)=1.000n, f(n)=n^2, c=1 e n_0=1.000(ou, ainda, c=100 e n_0=10)
      - Dizemos que 1.000n=O(n^2)

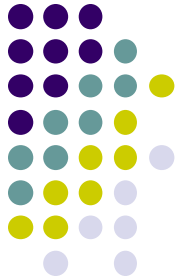
Mais algumas considerações



- Ao dizer que T(n) = O(f(n)), garante-se que T(n) cresce numa taxa não maior do que f(n), ou seja, f(n) é seu limite superior

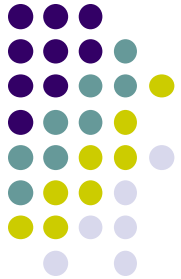
- Ao dizer que f(n) = Ω(T(n)), tem-se que T(n) é o limite inferior de f(n)

Exercícios conceituais



- Suponha que um dos algoritmo de ordenação mais utilizados, possui no melhor caso O(n). Podemos dizer que nenhum outro algoritmo poderá atingir uma complexidade melhor do que esta?

Exercícios conceituais



- Suponha que um dos algoritmo de ordenação mais utilizados, possui no melhor caso O(n). Podemos dizer que nenhum outro algoritmo poderá atingir uma complexidade melhor do que esta?
  - Para resolver o problema é necessária ler todos os valores. Portanto, qualquer algoritmo será no mínimo Ω(n).

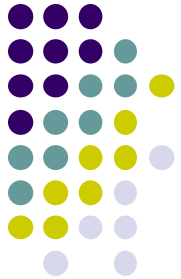
Exercícios conceituais

- Dois algoritmos A e B possuem complexidade n^2 e 200n, respectivamente. Você utilizaria o algoritmo A ao invés do B?

Exercícios conceituais

- Dois algoritmos A e B possuem complexidade n^2 e 200n, respectivamente. Você utilizaria o algoritmo A ao invés do B?
  - Sim, para entrada pequenas.

Exercícios conceituais



- O algoritmo a seguir é $O(n^2)$?

```
MaxMin(vetor v)
        max=v[1];
        min=v[1];
        para i=2 até n faça
                se v[1]> max então max=v[1]; fimse
                se v[1]< min então  min=v[1]; fimse
        fimpara;
fim.
```

Para pensar (em casa)

- Considere o problema de buscar um elemento em um conjunto ordenada do dados: $a_1 < a_2 < ... < a_n$
  - Mostre que o algoritmo é $\Omega(\log n)$

Exercício

- Mostrar que
  - $n^2 = O(n^3)$
  - $n^3 = \Omega(n^2)$

- Se $f(n)=n^2$ e $g(n)=2n^2$, então essas duas funções têm taxas de crescimento iguais
  - Provar que $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$

Exercício

- Provar que $6n^3 \neq \Theta(n^2)$

Exercício

- Provar que $f(n) = \frac{1}{2} n^2 - 3n = \Theta(n^2)$
  - Intuitivamente os termos de menor grau são desconsiderados na análise assintótica

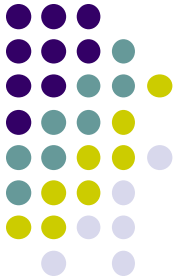
Exercício

- $2^{n+k} = O(2^n)$?
- $3^n = O(2^n)$?

Exercício

- Mostrar que $x^4-23x^3+12x^2+15x-21 = \Theta(x^4)$

Solução



It is clear that when $x \geq 1$,

$$x^4 - 23x^3 + 12x^2 + 15x - 21 \leq x^4 + 12x^2 + 15x \leq x^4 + 12x^4 + 15x^4 = 28x^4.$$

Also,

$$x^4 - 23x^3 + 12x^2 + 15x - 21 \geq x^4 - 23x^3 - 21 \geq x^4 - 23x^3 - 21x^3 = x^4 - 44x^3 \geq \frac{1}{2}x^4,$$

whenever

$$\frac{1}{2}x^4 \geq 44x^3 \Leftrightarrow x \geq 88.$$

Thus

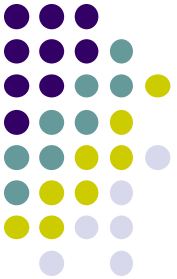
$$\frac{1}{2}x^4 \leq x^4 - 23x^3 + 12x^2 + 15x - 21 \leq 28x^4, \text{ for all } x \geq 88.$$

We have shown that $f(x) = x^4 - 23x^3 + 12x^2 + 15x - 21 = \Theta(x^4)$.

Exercício

- Seja f(x) = O(g(x)) e g(x) = O(h(x)).
  - Mostre que f(x) = O(h(x))

Solução



If $f(x) = O(g(x))$, then there are positive constants c_1 and n_0' such that

$$0 \leq f(n) \leq c_1\, g(n) \text{ for all } n \geq n_0',$$

and if $g(x) = O(h(x))$, then there are positive constants c_2 and n_0'' such that

$$0 \leq g(n) \leq c_2\, h(n) \text{ for all } n \geq n_0''.$$

Set $n_0 = \max(n_0', n_0'')$ and $c_3 = c_1\, c_2$. Then

$$0 \leq f(n) \leq c_1\, g(n) \leq c_1\, c_2\, h(n) = c_3\, h(n) \text{ for all } n \geq n_0.$$

Thus $f(x) = O(h(x))$.

Exercício

- Sejam f(n) e g(n) funções assintoticamente não negativas. Usando a definição básica da notação Θ, prove que

- max{ f(n), g(n) } = Θ(f(n) + g(n))

Demonstração

- Prova de O(.)

Supondo f maior

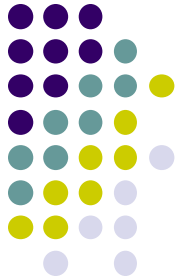
f <= c (f + g)

f (c – 1) + c g >= 0

Como f e g são positivos, basta (c-1) >= 0

Portanto c >= 1

Demonstração



- Prova de Ω(.)

Supondo f maior

f >= c (f + g)

c (f + g) <= f

c (f + g) <= g → porque g < f

c <= g / (f + g) <= g / (g + g) → porque g < f

c <= 1 / 2

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?

Demonstração

$T_1(n) + T_2(n) = \max (O(f(n)), O(g(n)))$

Demonstração

$T_1(n) + T_2(n) = \max(O(f(n)), O(g(n)))$

$T_1 <= c_1 f$ e $T_2 <= c_2 g$

Somando a desigualdade

$T_1 + T_2 <= c_1 f + c_2 g$, para $n > n_3 = \max\{n_1, n_2\}$

Supondo f maior

$T_1 + T_2 <= c_1 f + c_2 f \rightarrow T1 + T2 <= (c1+c2)f$ para $n > n_3$

Demonstração

$T_1(n) * T_2(n) = O(f(n) * g(n))$

$T_1 <= c_1\ f$ e $T_2 <= c_2\ g$

Multiplicando a desigualdade

$T_1 * T_2 <= c_1\ f * c_2\ g$, para $n > n_3 = \max\{n_1,n_2\}$

Supondo f maior

$T_1 * T_2 <= (c_1\ c_2)\ f\ g$, para $n > n_3$

Exercício

- Prove
  - 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

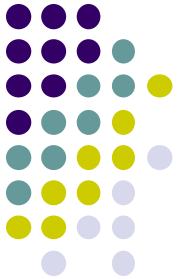
$$f(x) = a_n x^n + a_{n-1} x^{n-1} + ...a_1 x + a_0$$

seguindo a seguinte classificação em função do grau

- Grau 0: polinômio constante
- Grau 1: função afim (polinômio linear, caso $a_0 = 0$)
- Grau 2: polinômio quadrático
- Grau 3: polinômio cúbico

Se f(x)=0, tem-se o polinômio nulo

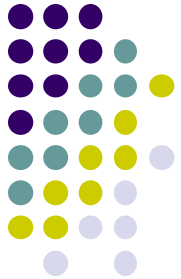
Demonstração



- Prova de Ω(.)
- Prova de O(.)

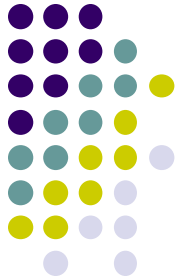
- Ver quadro

Taxas de crescimento



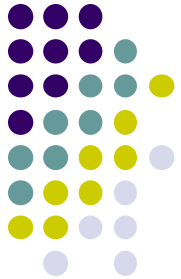
- Algumas regras
  - $\log^k n = O(n)$ para qualquer constante k, pois logaritmos crescem muito vagarosamente

Pergunta



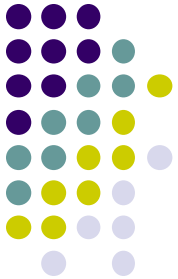
- Para qualquer algoritmo, pode-se dizer o que está abaixo?
  - $T(n) = O(\infty)$
  - $T(n) = \Omega(-\infty)$

Pergunta



- Para qualquer algoritmo, pode-se dizer o que está abaixo?
  - $T(n) = O(\infty)$
  - $T(n) = \Omega(-\infty)$
- Se sim, por que simplesmente não fazemos isso para todo algoritmo e pulamos para o próximo assunto da disciplina?

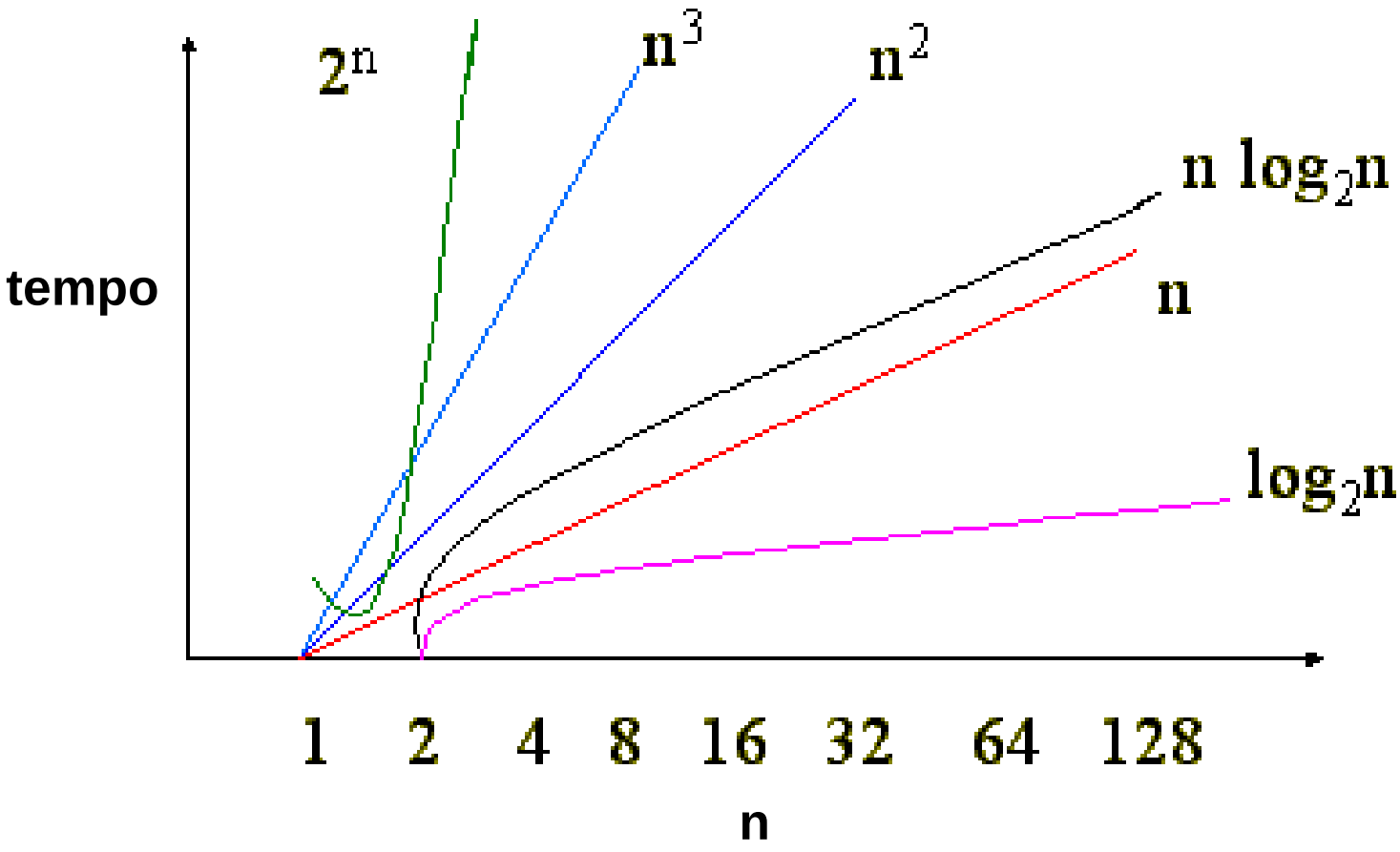
Funções e taxas de crescimento



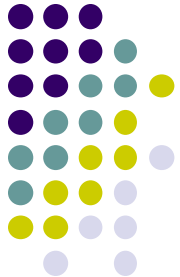
- As mais comuns

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

Funções e taxas de crescimento



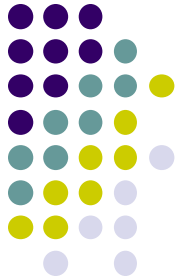
Taxas de crescimento



- Apesar de às 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



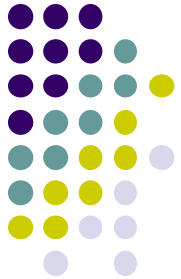
- Um algoritmo tradicional e muito utilizado é da ordem de $n^{1,5}$, enquanto um algoritmo novo proposto recentemente é da ordem de $n \log n$
  - $f(n)=n^{1,5}$
  - $g(n)=n \log n$

- Qual algoritmo você adotaria na empresa que está fundando?
  - Lembre-se que a eficiência desse algoritmo pode determinar o sucesso ou o fracasso de sua empresa

Exercício

- Uma possível solução
  - $f(n) = n^{1,5}$ ➔ $n^{1,5}/n = n^{0,5}$ ➔ $(n^{0,5})^2 = n$
  - $g(n) = n \log n$ ➔ $(n \log n)/n = \log n$ ➔ $(\log n)^2 = \log^2 n$
    - 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 pela empresa no momento

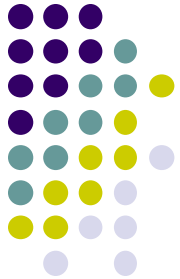
Análise de algoritmos



- Para proceder a uma análise de algoritmos e determinar as taxas de crescimento, necessitamos de um modelo de computador e das operações que executa

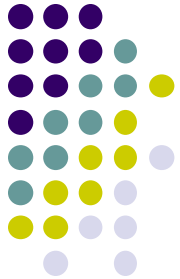
- Assume-se o uso de um computador tradicional, em que as instruções de um programa são executadas sequencialmente
  - Com memória infinita, por simplicidade

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

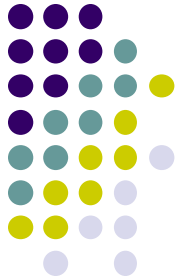


- Considera-se somente o algoritmo e suas entradas (de tamanho n)

- Para uma entrada de tamanho n, pode-se calcular $T_{melhor}(n)$, $T_{média}(n)$ e $T_{pior}(n)$, ou seja, o melhor tempo de execução, o tempo médio e o pior, respectivamente
  - Obviamente, $T_{melhor}(n) \leq T_{média}(n) \leq T_{pior}(n)$

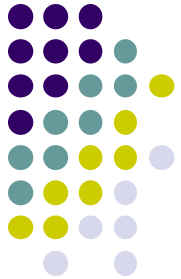
- Atenção: para mais de uma entrada, essas funções teriam mais de um argumento

Análise de algoritmos



- Geralmente, utiliza-se somente a análise do pior caso $T_{pior}(n)$, 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 subseqüência máxima
  - Dada uma seqüência de inteiros (possivelmente negativos) a_1, a_2, ..., 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_1 a a_3)

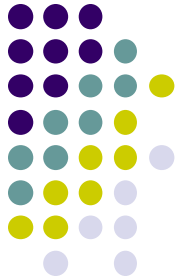
Soma da subseqüência máxima

- 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 subseqüência 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

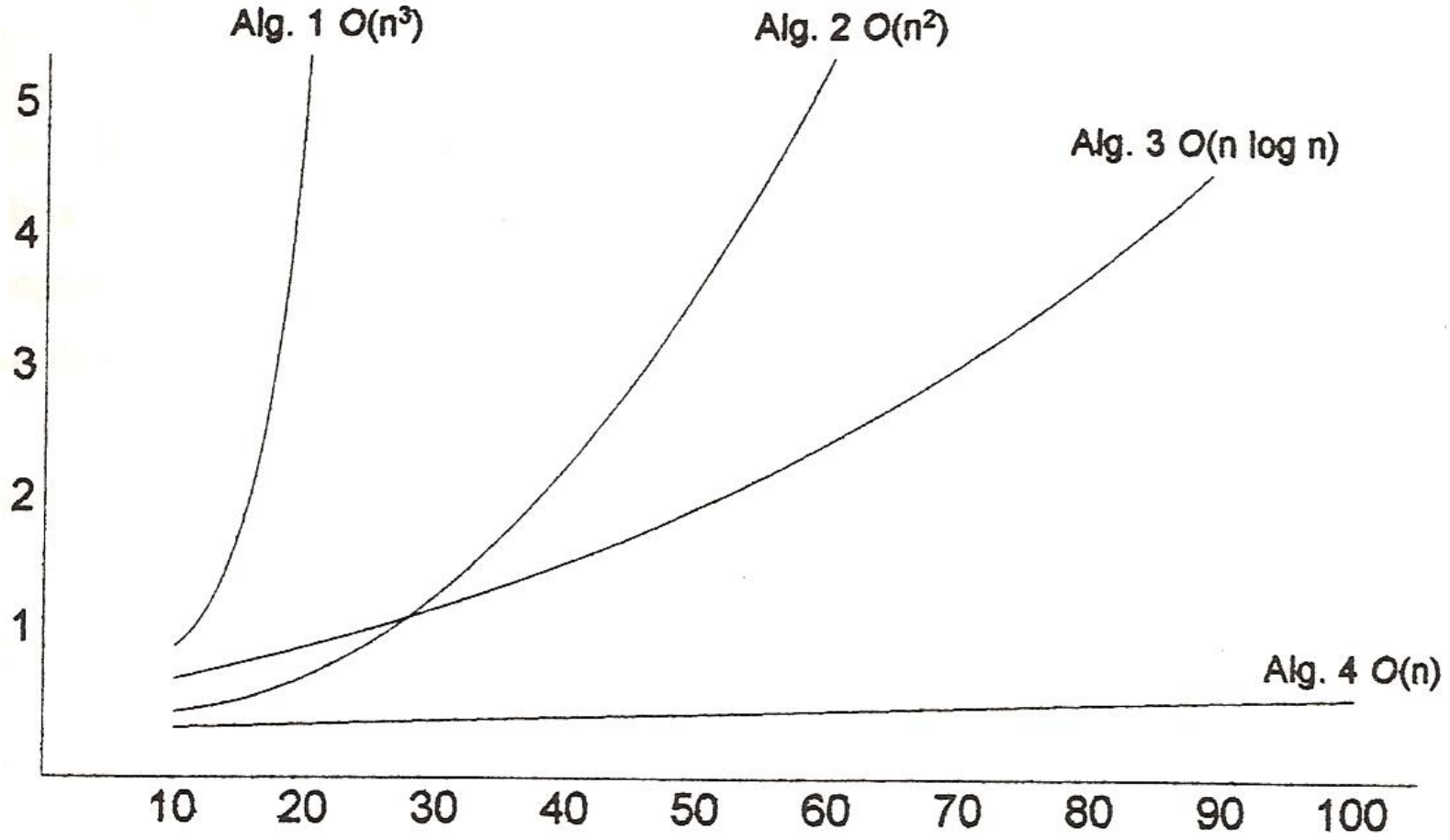


Gráfico (*n* x milisegundos) das taxas de crescimento dos 4 algoritmos com entradas entre 10 e 100.

Taxas de crescimento

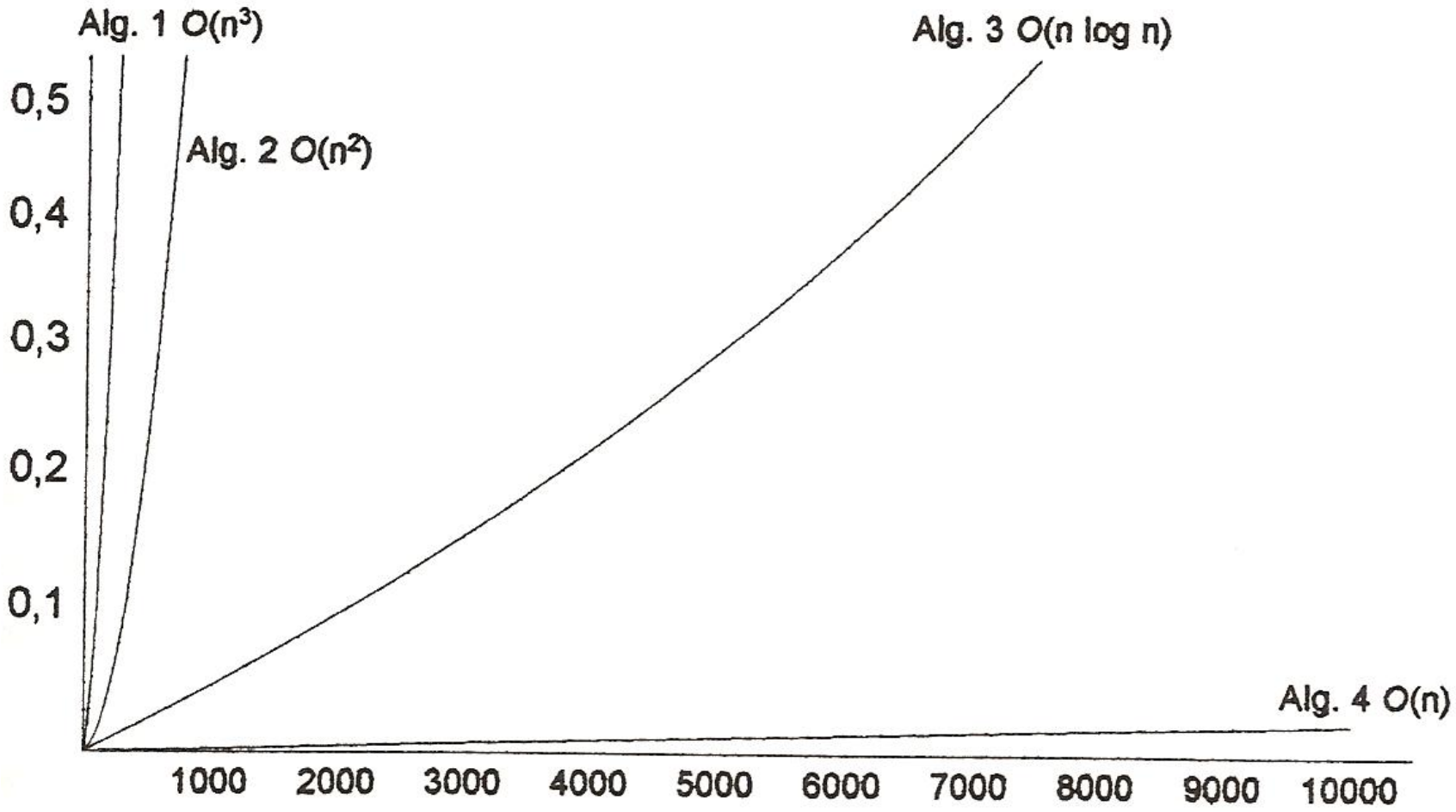


Gráfico (*n* x segundos) dos 4 algoritmos para entradas maiores