Análise de algoritmos
Parte I

SCE-181 Introdução à Ciência da Computação II
Rosane Minghim

Material preparado por : Prof. Thiago A. S. Pardo
e modificado por R. Minghim

SCC - ICMC

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?
  - A afirmação está redundante?

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
    - 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?

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 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 <u>eficiência assintótica</u> de um algoritmo descreve a eficiência relativa dele 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 o impacto disso

Análise de algoritmos

- Existem basicamente 2 formas de estimar o tempo de execução de programas e decidir quais são os melhores
  - Empírica ou teoricamente
- É desejável e possível estimar qual o melhor algoritmo sem ter que executá-los
  - Função da análise de algoritmos

Calculando o tempo de execução

- Supondo que as operações simples demoram uma unidade de tempo para executar, considere o programa abaixo para calcular o resultado de $\sum_{i=1}^{n} i^3$

```
Início
declare soma_parcial numérico;
soma_parcial ← 0;
para i←1 até n faça
    soma_parcial←soma_parcial+i*i*i;
escreva(soma_parcial);
Fim
```

Calculando o tempo de execução

- Supondo que as operações simples demoram uma unidade de tempo para executar, considere o programa abaixo para calcular o resultado de $\sum_{i=1}^{n} i^3$

Início
declare soma_parcial numérico;
soma_parcial ← 0;
para i←1 até n faça
    soma_parcial←soma_parcial+i*i*i;
escreva(soma_parcial);
Fim

1 unidade de tempo

1 unidade para inicialização de i, n+1 unidades para testar se i≤n e n unidades para incrementar i = 2n+2

4 unidades (1 da soma, 2 das multiplicações e 1 da atribuição) executada n vezes (pelo comando "para") = 4n unidades

1 unidade para escrita

Calculando o tempo de execução

- Supondo que as operações simples demoram uma unidade de tempo para executar, considere o programa abaixo para calcular o resultado de $\sum_{i=1}^{n} i^3$

Início
declare soma_parcial numérico;
soma_parcial ← 0;
para i←1 até n faça
    ...parcial+i*i*i;

1 unidade de tempo

1 unidade para inicialização de i, n+1 unidades para testar se i≤n e n unidades para incrementar i = 2n+2

4 unidades (1 da soma, 2 das multiplicações e 1 da atribuição) executada n vezes (pelo comando "para") = 4n unidades

Custo total: somando tudo, tem-se 6n+4 unidades de tempo, ou seja, a função é **O(n)**

1 unidade para escrita

Calculando o tempo de execução

- Ter que realizar todos esses passos para cada algoritmo (principalmente algoritmos grandes) pode se tornar uma tarefa cansativa
- Em geral, como se dá a resposta em termos do *big-oh*, costuma-se desconsiderar as constantes e elementos menores dos cálculos
  - No exemplo anterior
    - A linha soma_parcial←0 é insignificante em termos de tempo
    - É desnecessário ficar contando 2, 3 ou 4 unidades de tempo na linha soma_parcial←soma_parcial+i*i*i
    - O que realmente dá a grandeza de tempo desejada é a repetição na linha para i←1 até n faça

Regras para o cálculo

- Repetições
  - O tempo de execução de uma repetição é pelo menos o tempo dos comandos dentro da repetição (incluindo testes) vezes o número de vezes que é executada

Regras para o cálculo

- Repetições aninhadas
  - A análise é feita de dentro para fora
  - O tempo total de comandos dentro de um grupo de repetições aninhadas é o tempo de execução dos comandos multiplicado pelo produto do tamanho de todas as repetições
  - O exemplo abaixo é $O(n^2)$

```
para i←0 até n faça
   para j←0 até n faça
            faça k←k+1;
```


Regras para o cálculo

- Comandos consecutivos
  - É a soma dos tempos de cada um, o que pode significar o máximo entre eles
  - O exemplo abaixo é $O(n^2)$, apesar da primeira repetição ser $O(n)$

```
para i←0 até n faça
   k←0;
para i←0 até n faça
   para j←0 até n faça
            faça k←k+1;
```


Regras para o cálculo

- Se... então... senão
  - Para uma cláusula condicional, o tempo de execução nunca é maior do que o tempo do teste mais o tempo do maior entre os comandos relativos ao <u>então</u> e os comandos relativos ao <u>senão</u>
  - O exemplo abaixo é $O(n)$

```
se i<j
   então i←i+1
   senão para k←1 até n faça
                  i←i*k;
```


Regras para o cálculo

- Chamadas a sub-rotinas
  - Uma sub-rotina deve ser analisada primeiro e depois ter suas unidades de tempo incorporadas ao programa/sub-rotina que a chamou

Operações Elementares

- As operações elementares (OE), são as medidas básicas de complexidade de algoritmos.

- Exemplos:
  - a++ → 2 OE (=,+).
  - b=a*5-Vetor[2*2] → 5 OE (=,*,-,[],*).
  - b+=soma(a,b<2) → 4 OE (+,=,soma,<).
  - C++ == E[1] AND B>3 → 6 OE (=,+,==,[],AND,>).

Exemplos

•Para o seguinte problema primeiro temos que identificar as operações elementares (OE) que são realizadas:

```
1  Procedure Buscar(VAR a:Vetor, c:Integer):Cardinal;
2    Var j:Cardinal
3
4    BEGIN
5      j:= 1;
6      WHILE (a[j]<c) AND (j<n) DO
7        j:=j+1
8      END
9      IF a[j]=c THEN
10       RETURN j
11     ELSE RETURN 0
12   END
13
14 End Buscar
```

- Linha 5: 1 OE (=)
- Linha 6: 4 OE (<,<,[],AND)
- Linha 7: 2 OE (+,=)
- Linha 9: 2 OE (IF,[])
- Linha 10: 1 OE (Return, só si a condição é cumprida)
- Linha 11: 1 OE (Return, só si a condição anterior é falsa)

Exemplos

- **Melhor Caso**
  - A linha 5 e 6 entrem só na primeira metade da condição, isso quer disser que a comparação a[j]<c vai ser falsa.
    - T1(n)=**1OE**(linha 5) + **2OE**(linha 6) = **3OE**
  - Da linha 9 à 11, vai fazer pelo menos 1 comparação e um Return.
    - T2(n)=**2OE**(linha 9) + **1OE**(Return) = **3OE**

```
5      j:= 1;
6      WHILE (a[j]<c) AND (j<n) DO
7        j:=j+1
8      END
9      IF a[j]=c THEN
10       RETURN j
11     ELSE RETURN 0
12   END
```

T(n)= T1(n)+T2(n)
T(n)=6OE

Exemplos

- **Pior Caso**:
  - Linha 5 = **1OE**.
  - Linha 2 é repetida **n-1** vezes até que a segunda condição seja cumprida.
  - Linha 9 até 11 = **3OE**.
  - Cada iteração do ciclo while composta pelas linhas 6 e 7 mas uma execução adicional do while (que avalia de novo si as condições do ciclo foram cumpridas).

```
5      j:= 1;
6      WHILE (a[j]<c) AND (j<n) DO
7        j:=j+1
8      END
9      IF a[j]=c THEN
10       RETURN j
11     ELSE RETURN 0
12   END
```

$T(n)=1+((\sum_{i=1}^{n-1}(4+2))+4)+3$
$T(n)=1+((6(n-1)+4)+3$
$T(n)=1+6n-6+4+3$
$T(n)=6n+2$

Exemplos

- **Pior Caso**:
  - **T(n)= 1 +** → Corresponde à **linha 5**, onde o custo é **1OE**.
  - **((Somatória desde i+1 até n-1)** → Corresponde à **linha 6**, já que se percorrera pelo menos n-1 vezes o ciclo até que a segunda condição seja TRUE.
    - **(4+2)** → As 4OE representam as condições dentro do while, e 2OE representam as condições dentro do ciclo while na **linha 7**.
    - **+4** → Representa as condições do while, já que cada loop ele volve a perguntar si a condição foi cumprida para terminar o ciclo.
  - Depois da somatória no **+3** → Representa às **linhas 9 até 11**, com 3OE.

```
5     j:= 1;
6     WHILE (a[j]<c) AND (j<n) DO
7        j:=j+1
8     END
9     IF a[j]=c THEN
10       RETURN j
11    ELSE RETURN 0
12  END
```

T(n)=1+ (($\sum_{i=1}^{n-1}(4+2)$)+4)+3

T(n)=1+ ((6(n-1)+4)+3

T(n)=1+6n-6+4+3

T(n)=6n+2

Exemplo

```
1  For i:=1 To n-1 do
2      For j:=n To i+1 By -1 do
3          If a[j-1]>a[j] THEN
4              temp:=a[j-1]
5              a[j-1]:=a[j]
6              a[j]:=temp
7          END
8      END
9  END
```

- 3 OE (=,+,n<=n-1)
- 3 OE (=,+,n<=i+1)
- 4 OE ([],-,|,[])
- 3 OE (=,-,[])
- 4 OE ([],=,-,[])
- 2 OE ([],=)
- 3 OE (incremento do for e comparação)
- 3 OE (comparação para incrementar)

- **Melhor Caso** → A condição na linha 3 será falsa, e as linhas 4 até 6 não serão executadas.
- **Pior Caso** → A condição na linha 3 seja verdadeira, e as linhas 4 até 6 serão executadas.

Exercício

- Estime quantas unidades de tempo são necessárias para rodar o algoritmo abaixo

```
Início
declare i e j numéricos;
declare A vetor numérico de n posições;
i←1;
enquanto i≤n faça
    A[i]←0;
    i←i+1;
para i←1 até n faça
    para j←1 até n faça
        A[i]←A[i]+i+j;
Fim
```


Relembrando um pouco de matemática...

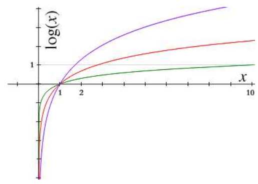
- Expoentes
  - $x^a x^b = x^{a+b}$
  - $x^a/x^b = x^{a-b}$
  - $(x^a)^b = x^{ab}$
  - $x^n+x^n = 2x^n$ (diferente de x^{2n})
  - $2^n+2^n = 2^{n+1}$

Relembrando um pouco de matemática...

- Logaritmos (usaremos a base 2, a menos que seja dito o contrário)
  - $x^a=b \rightarrow \log_x b=a$
  - $\log_a b = \text{lob}_c b/\log_c a$, 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

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

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

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

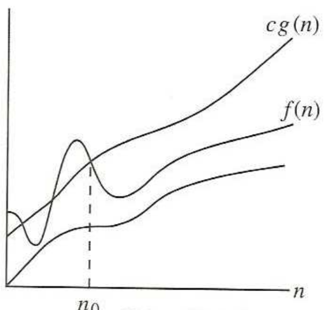
Algumas notações

- 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) ≤ c * f(n) quando n ≥ n_0
    - A taxa de crescimento de T(n) é menor ou igual à taxa de f(n)
  - T(n) = Ω(f(n)) (lê-se "ômega") se existirem constantes c e n_0 tal que T(n) ≥ c * f(n) quando n ≥ n_0
    - A taxa de crescimento de T(n) é maior ou igual à taxa de f(n)

Algumas notações

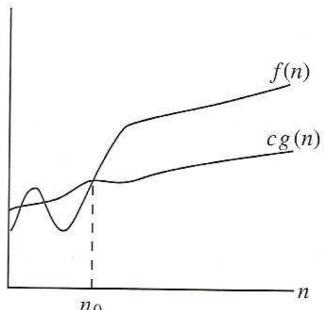
- Notações que usaremos na análise de algoritmos
  - T(n) = Θ(f(n)) (lê-se "theta") se e somente se T(n) = O(f(n)) e T(n) = Ω(f(n))
    - A taxa de crescimento de T(n) é igual à taxa de f(n)
  - T(n) = o(f(n)) (lê-se *little-oh* ou *little-o*) se e somente se T(n) = O(f(n)) e T(n) ≠ Θ(f(n))
    - A taxa de crescimento de T(n) é menor do que a taxa de f(n)

Algumas notações



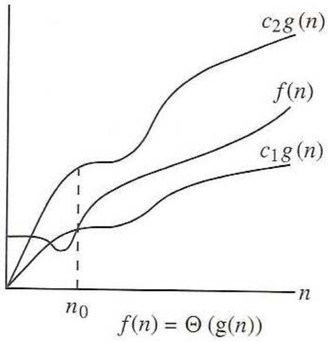
$f(n) = O(g(n))$

Algumas notações



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

Algumas notações



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 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
  - Segunda as notações anteriores, se existe um ponto n_0 a partir do qual c*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)

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)

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

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

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

Taxas de crescimento

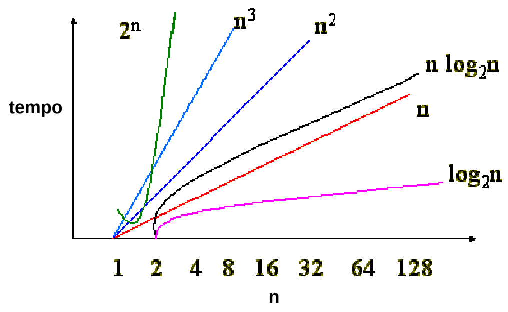
- Algumas regras
  - $\log^k n = O(n)$ para qualquer constante k, pois logaritmos crescem muito vagarosamente

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 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 em <u>duplas</u>

- 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 em duplas

- 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

Exercício

- Muito tempo atrás, o visir Sissa bem Dahir inventou o jogo de xadrez para o Rey Shirham da Índia.
- O Rey deu a possibilidade de selecionar sua recompensa, e Sissa lhe deu algumas opções:
  - Poderia ser recompensando com uma quantidade de trigo equivalente à plantação de trigo de seu reino por 2 anos, ou,
  - Poderia ser recompensado com uma quantidade de trigo que seria calculado da seguinte forma:
    - Um grão de trigo na primeira seção do tabuleiro de xadrez.
    - Dois grãos de trigo na segunda seção .
    - Quatro grãos na terceira seção , e assim por diante, até chegar na ultima seção.
  - O Rey selecionou a segunda opção. Quantos grão de trigo deu o Rey para Sissa?. Quál é a complexidade?

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_2 a a_4)

Soma da subseqüência máxima

- Há <u>muitos algoritmos</u> 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

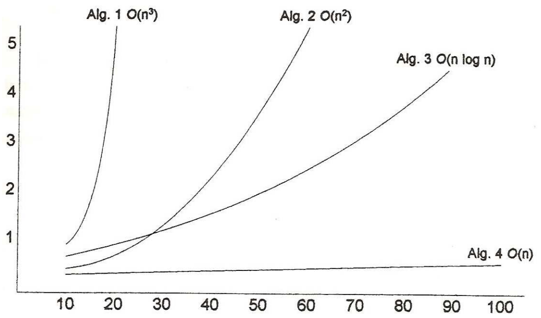


Gráfico (n x milisegundos) das taxas de crescimento dos 4 algoritmos com entradas entre 10 e 100.

Taxas de crescimento

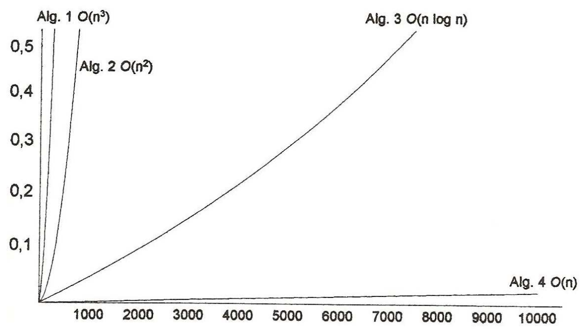


Gráfico (n x segundos) dos 4 algoritmos para entradas maiores

Exercício 1

- Com um algoritmo de função de custo temporal $f(n)= n^3$ se podem resolver problemas de tamanho k em 1 hora.
  - Até que tamanho poderemos resolver o mesmo problema,no mesmo tempo e com o mesmo algoritmo se tivéssemos um computador 1000 vezes mais rápido.
  - E si a função de custo fosse $f(n)= 2^n$

Solução 1

- F(n) representa o número de operações elementares feitas para o algoritmo de tamanho n.
- Cada operação precisa de um tempo t para ser feita, então temos que f(k)t é uma hora.
- Um computador 1000 vezes mais rápido vai demorar t/1000 para realizar cada operação Elemental.
  - A equação → $f(k)t = f(x)t / 1000$.
  - Quando $f(n)= n^3$ → $x=10*k$.
  - Quando $f(n)= 2^n$ → $x=k+\log_2 1000$.

Exercício 2

- Usando a definição de notação assintótica Θ, demonstre que $1024n^2 + 5n \in \Theta(n^2)$.
  - Primeiro há que encontrar um número n_0 e uma constante c>0 que cumpra que $1024n^2 + 5n <= cn^2$ para todo $n <= n_0$.
  - Para simplificar a equação só divide-se entre n^2 para obter $1024 + (5/n) <= c$.
    - $n_0 = 5$ e $c = 1025$.

Exercício 3

- Segundo o seguinte algoritmo, determine qual é a complexidade.

```
procedure prod_mat (n:integer)
    var
        i,j,k:integer
    begin
        for i:=1 to n do
            for j:=1 to n do begin
                C[i,j]:=0
                for k:=1 to n do
                    C[i,j]:=C[i,j]+A[i,k]*B[k,j]
                end
            end
        end
    end
end procedure
```

$O(n^3)$