Projeto e Análise de Algoritmos

Prof. Dr. Ednaldo B. Pizzolato

www.dc.ufscar.br/~ednaldo

ednaldo@dc.ufscar.br

COMPLEXIDADE

Complexidade

Exemplo

- Sejam 5 algoritmos A_1 a A_5 para resolver um mesmo problema, de complexidades diferentes. (Supomos que uma operação leva 1 ms para ser efetuada.)
- $T_k(n)$ é a complexidade ou seja o número de operações que o algoritmo efetua para n entradas

n	**A_1** $T_1(n)= n$	**A_2** $T_2(n)=n\log n$	**A_3** $T_3(n)=n^2$	**A_4** $T_4(n)=n^3$	**A_5** $T_5(n)=2^n$
16	0.016s	0.064s	0.256s	4s	1m4s
32	0.032s	0.16s	1s	33s	46 Dias
512	0.512s	9s	4m22s	1 Dia 13h	10^{137} Séculos

tempo necessário para o algoritmo em função de n entradas

Operações primitivas

- Atribuição de valores a variáveis
- Chamadas de métodos
- Operações aritméticas
- Comparação de dois números
- Acesso a elemento de um array
- Seguir uma referência de objeto (acesso a objeto)
- Retorno de um método

Exemplo de Análise de Algoritmo 1

arrayMax(A, n):

Entrada: array A com n ≥ 1 elementos inteiros

Saida: o maior elemento em A

```
tmpMax ← A[0]
for i←1 to n-1 do
 if tmpMax < A[i]
       tmpMax ← A[i]
return tmpMax
```

Exemplo de Análise de Algoritmo 1

código	custo	vezes
tmpMax <- A[0]	c1	1
for i ← 1 to n-1 do	c2	n
if tmpMax < A[i] then	c3	n-1
tmpMax ← A[i]	c4	n-1
return tmpMax	c5	1

$T_*(n)$ = c1*1+ c2*n + c3*(n-1) + c5 * 1 (melhor caso)

$T_*(n)$ = c1+ n*c2 + n*c3 – c3 + c5

se considerarmos os custos iguais, teremos:

$T_*(n)$ = c+ n*c + n*c – c + c = 2n*c + c (= **2n + 1** para c=1)

Exemplo de Análise de Algoritmo 1

código	custo	vezes
tmpMax <- A[0]	c1	1
for i ← 1 to n-1 do	c2	n
if tmpMax < A[i] then	c3	n-1
tmpMax ← A[i]	c4	n-1
return tmpMax	c5	1

$T^*(n)$ = c1*1+ c2*n + c3*(n-1) + c4*(n-1) + c5 * 1 (pior caso)
$T^*(n)$ = c1+ n*c2 + n*c3 – c3 + n*c4 – c4 + c5
se considerarmos os custos iguais, teremos:
$T^*(n)$ = c+ n*c + n*c – c + n*c – c + c = 3n*c (= **3n** para c=1)

$$T_*(n) \leq t(I) \leq T^*(n)$$

Observações sobre consumo de tempo:

- estimar consumo do algoritmo, independente do computador
- despreze constantes multiplicativas: *10 n é o mesmo que n*
- consumo de tempo é diferente para cada instância do problema
- agrupe instâncias por "tamanho"
- o conceito de tamanho de uma instância
- muitas instâncias têm o mesmo tamanho
- consumo de tempo no pior caso
- consumo de tempo no melhor caso

Exemplo de Análise de Algoritmo 2

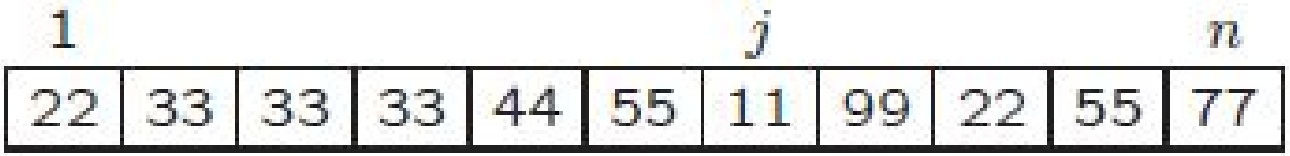
- Rearranjar um vetor em ordem crescente
- A[1 . . n] é crescente se A[1] ≤ · · · ≤ A[n]

22	33	33	33	44	55	11	99	22	55	77

11	22	22	33	33	33	44	55	55	77	99

Exemplo de Análise de Algoritmo 2

```
ORDENA-POR-INSERÇÃO (A, n)
1  para j ← 2 até n faça
2     chave ← A[j]
3     i ← j − 1
4     enquanto i ≥ 1 e A[i] > chave faça
5        A[i+1] ← A[i]
6        i ← i − 1
7     A[i+1] ← chave
```



Exemplo de Análise de Algoritmo 2

O algoritmo faz o que prometeu?

- Invariante: no início de cada iteração, A[1 . . j−1] é crescente
- Se vale na última iteração, o algoritmo está correto!

```
ORDENA-POR-INSERÇÃO (A, n)
1  para j ← 2 até (*) faça
2     chave ← A[j]
3     i ← j − 1
4     enquanto i ≥ 1 e A[i] > chave faça
5        A[i+1] ← A[i]
6        i ← i − 1
7     A[i+1] ← chave
```

- vale na primeira iteração
- se vale em uma iteração, vale na seguinte

Exemplo de Análise de Algoritmo 2

Quanto tempo consome?

- Suponha 1 unidade de tempo por linha

```
ORDENA-POR-INSERÇÃO (A, n)
1  para j ← 2 até n faça
2    chave ← A[j]
3    i ← j − 1
4    enquanto i ≥ 1 e A[i] > chave faça
5       A[i+1] ← A[i]
6       i ← i − 1
7    A[i+1] ← chave
```

Linha		total de unidades de tempo
1	=	n
2	=	n − 1
3	=	n − 1
4	≤	2+3+· · ·+n = (n − 1)(n+2)/2
5	≤	1+2+· · ·+(n−1) = n(n − 1)/2
6	≤	1+2+· · ·+(n−1) = n(n − 1)/2
7	=	n − 1

total ≤ **$3/2n^2 + 7/2n - 4$** unidades de tempo

Exemplo de Análise de Algoritmo 3

Encontrar a soma dos elementos positivos de um vetor A[1 . . n]

Uma instância do problema: Encontrar a soma dos elementos positivos do vetor (20; -30; 15; -10; 30; -20; -30; 30)

Exemplo de Análise de Algoritmo 3

```
SOMAPOSITIVOS (A; n)
1   s = 0
2   para i = 1 até n faça
3      se A[i] > 0
4         então s = s + A[i]
5   devolva s
```

O algoritmo está correto?

- testes só podem mostrar que o algoritmo está errado (????)
- análise pode provar que o algoritmo está correto.

O algoritmo está correto!

- Invariante: no começo de cada iteração
- s é a soma dos positivos de A[1 .. i-1]

No fim, s é a soma dos positivos de A[1 .. n].

Exemplo de Análise de Algoritmo 3

- Algoritmo recursivo

```
SOMAPOS (A;n)
1    se n = 0
2       então devolva 0
3       senão s = SOMAPOS (A; n - 1)
4    se A[n] > 0
5       então devolva s + A[n]
6       senão devolva s
```

T(n) : consumo de tempo no pior caso

- recorrência: T(n) = T(n - 1) + const
- T(n) = ?

Exemplo de Análise de Algoritmo 3

Observações sobre algoritmos recursivos

- Problemas com estrutura recursiva:
  - cada instância do problema contém uma instância menor do mesmo problema
- Algoritmo recursivo:
  - se a instância em questão é pequena resolva-a diretamente
  - Senão: reduza-a a uma instância menor do mesmo problema encontre solução S da instância menor use S para construir solução da instância original

Análise: Crescimento de Funções

- O tempo de execução geralmente dependente de um único parâmetro ***N***
  - ordem de um polinômio
  - tamanho de um arquivo a ser processado, ordenado, etc
  - ou medida abstrata do tamanho do problema a considerar (usualmente relacionado com o número de dados a processar)

- Quando há mais de um parâmetro
  - procura-se exprimir todos os parâmetros em função de um só
  - faz-se uma análise em separado para cada parâmetro

Análise: Crescimento de Funções

- Os Algoritmos têm tempo de execução proporcional a
  - ***1*** - muitas instruções são executadas uma só vez ou poucas vezes (se isto for verdade para todo o programa diz-se que o seu tempo de execução é constante)
  - ***Log N*** - tempo de execução é logarítmico (cresce ligeiramente à medida que ***N*** cresce) (quando ***N*** duplica ***log N*** aumenta, mas muito pouco; apenas duplica quando ***N*** aumenta para N^2)
  - ***N*** - tempo de execução é linear (típico quando algum processamento é feito para cada dado de entrada) (situação ótima quando é necessário processar ***N*** dados de entrada) (ou produzir ***N*** dados na saída)

Análise: Crescimento de Funções

- ***N log N*** - típico quando se reduz um problema em subproblemas, se resolve estes separadamente e se combinam as soluções (se ***N*** é 1 milhão ***N log N*** é perto de 20 milhões)

- N^2 - tempo de execução quadrático (típico quando é preciso processar todos os pares de dados de entrada) (prático apenas em pequenos problemas, ex: produto matriz - vetor)

- N^3 - tempo de execução cúbico (para $N = 100$, $N^3 = 1$ milhão, ex: produto de matrizes)

- 2^N - tempo de execução exponencial (provavelmente de pouca aplicação prática; típico em soluções de força bruta) (para $N = 20$, $2^N = 1$ milhão; ***N*** duplica, tempo passa a ser o quadrado)

RESOLUÇÃO DE ALGORITMOS

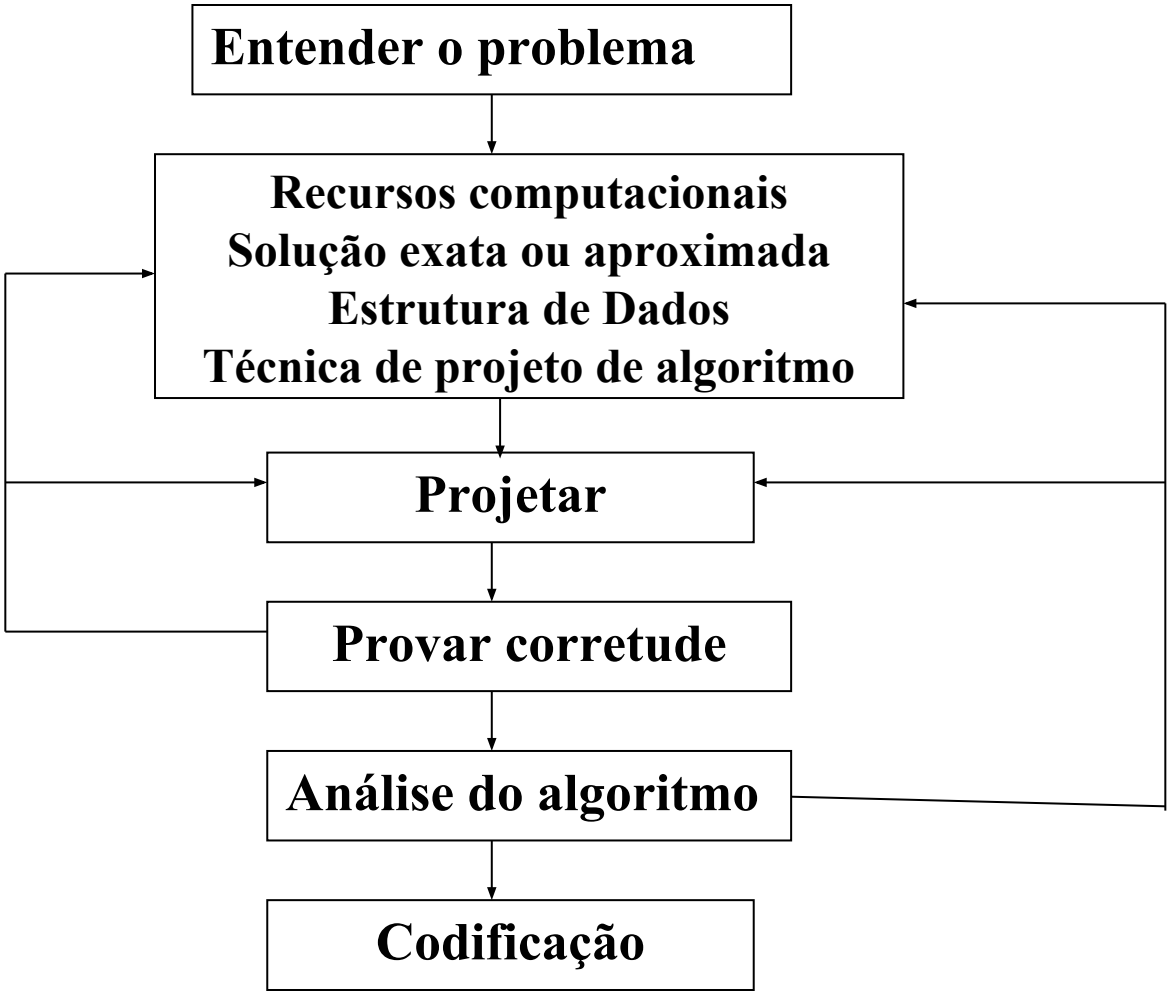
Fundamentos da resolução de problemas algorítmicos

- Passos para a resolução de um problema
  - Análise
  - Projeto de uma solução
  - Execução da solução
  - Verificação

Fundamentos da resolução de problemas algoritmicos

- Entendimento do problema
- Recursos computacionais
- Escolha entre solução exata ou aproximada
- Uso de técnica de projeto de algoritmo
- O projeto de algoritmos e a estrutura de dados
- Métodos para especificar um algoritmo
- Provando que o algoritmo está correto
- Analisando um algoritmo
- Codificando

Fundamentos da resolução de problemas algoritmicos



Fundamentos da resolução de problemas algoritmicos

- O que significa entender o problema?
  - Quais são os objetos do problema?
  - Quais as operações envolvidas/aplicadas aos objetos?

- O que contempla a questão sobre os recursos computacionais?
  - Como os objetos serão representados?
  - Como as operações serão implementadas?

Fundamentos da resolução de problemas algoritmicos

- Como provar que um algoritmos está correto?
  - Saídas corretas para cada entrada legítica de dados (levando-se em conta que a resposta tem que ser produzida em um tempo finito)
  - Baseada em fórmulas matemáticas corretas
  - Por indução

TIPOS DE PROBLEMAS

Tipos importantes de problemas

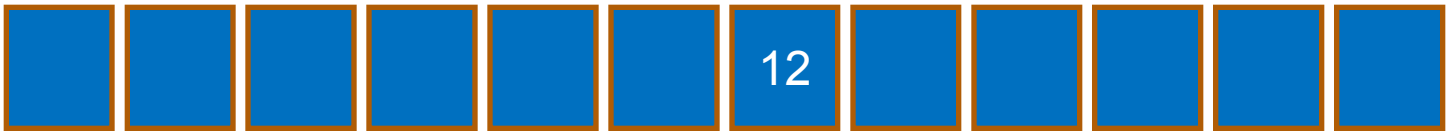
- Ordenação
- Busca
- Processamento de cadeias de caracteres
- Problemas de grafos
- Problemas combinatoriais
- Problemas geométricos
- Problemas numéricos

Ordenação

- Classificação
  - De estudantes
  - Resultados de busca na internet
  - Dicionários
  - Listas telefônicas

Buscas

- Encontrar uma determinada informação em uma lista;



- Encontrar uma configuração favorável para um determinado problema.

Processamento de cadeias de caracteres

- Achar uma determinada palavra em uma frase (string matching).
- Pode ser usado em palavras binárias (0 ou 1) ou “genéticas” (A,C,G,T).

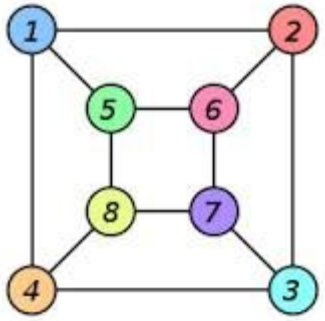
Grafos

- Transporte;
- Comunicações;
- Redes sociais e econômicas;
- Escalonamento de projetos; e
- Jogos

Grafos

- Algoritmos associados a grafos incluem:
  - Percorrer um grafo;
  - Caminho mais curto;
  - Ordenação topológica.

  - Traveling salesman problem
  - Graph-coloring problem



Combinatoriais

- Permutações
- Combinações
- Subsets

- 123
- 132
- 213
- 231
- 321
- 312

Combinatoriais

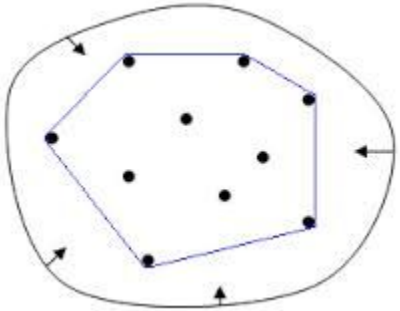
- Permutações
- Combinações
- Subsets

	Ana	Cintia	Luisa	Carla
João	x			
André			x	
Miguel		x		
Mauro				x

Geométricos

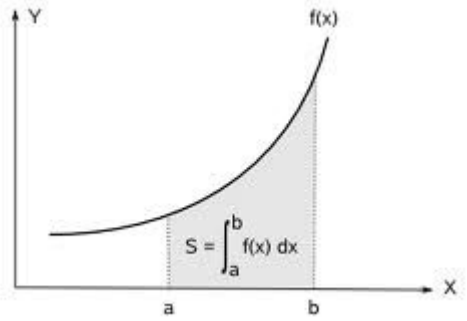
- Problemas relacionados com geometria como pontos, linhas e polígonos.

- Closest-pair problem
- Convex-hull problem



Numéricos

- Objetos matemáticos de natureza contínua:
  - Equações
  - Sistemas de equações
  - Integrais definidas
  - Funções
  - Números reais



THE END