Algoritmos e Estruturas de Dados II

Tiago Oliveira Cunha
(tocunha@dcc.ufmg.br)

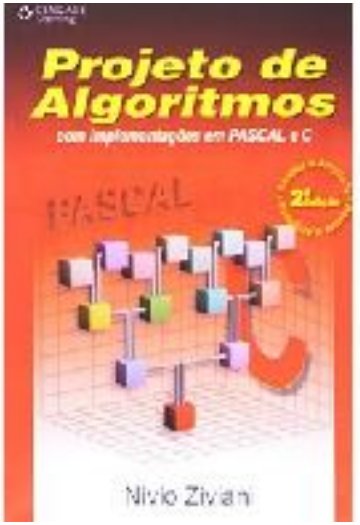
URL: www.dcc.ufmg.br/~tocunha (link teaching)

Aplicação de Algoritmos

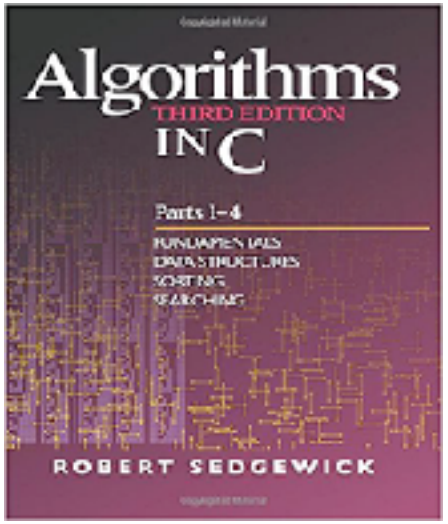
- Ordenação
- Pesquisa

Referências

- N. Ziviani. Projeto de Algoritmos com Implementações em Pascal e C. Cengage Learning, São Paulo, SP, 2011.



- R. Sedgewick. Algorithms in C, Parts 1-4 (Fundamental Algorithms, Data Structures, Sorting, Searching). Addison-Wesley, 1997.



Ementa

- Tipos Abstratos de Dados (TADs)
- Análise de Algoritmos

  O(n), O(n log n),)(n!), ...
- Estruturas de dados

  listas, filas e pilhas

Parte 1
prova: 26/09

- Métodos de ordenação

  quick, heap, merge, select, etc

Parte 2
prova: 07/11

- Métodos de pesquisa

  hash, árvores, árvores binárias, árvores digitais

Parte 3
prova: 07/12

Avaliação

- 3 provas (total 60 pontos)
- trabalhos práticos – 40 pontos (TP0 + 3 TPs)
  - Implementação
  - Documentação
  - Teste
- Listas de exercício e exercícios em sala de aula
  - Apenas contabilizados para quem obtiver >= 15 pontos nas provas

Regras Gerais

- Exame especial (nota >= 40 && frequencia >= 75%).
- Prova de reposição
  - Apenas para aqueles que perderam uma prova.
- Horário de atendimento:
  - (em breve)
- Presença:
  - Verificada por meio de chamada

Moddle / Website

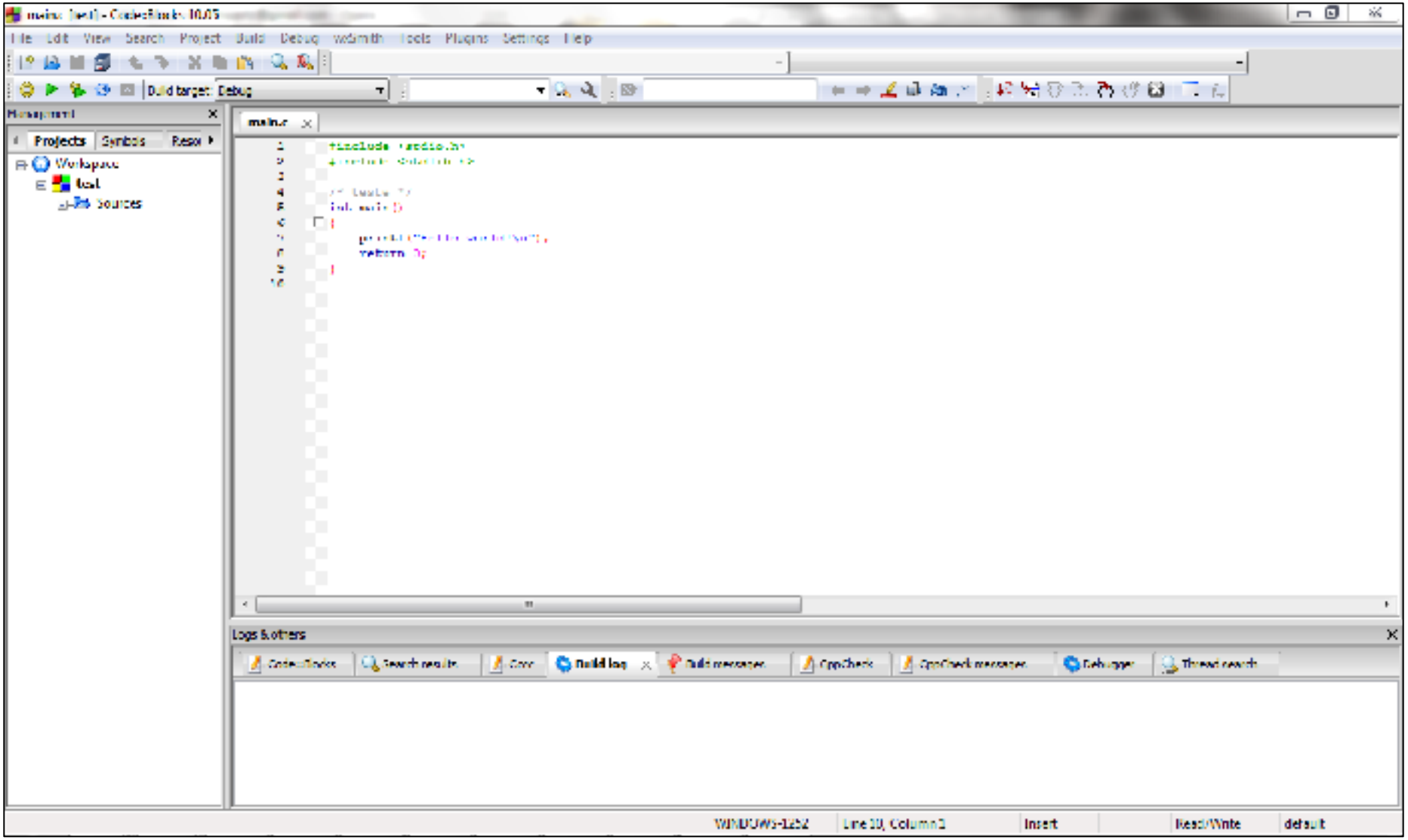
- Todas informações relacionadas ao curso, incluindo notas de aulas, estarão disponíveis no Moodle, fórum
- Sistema de submissão de trabalhos práticos

  (em breve)

Detalhes

- Linguagem: C
- Software Recomendado: CodeBlocks (www.codeblocks.org)
  - http://wiki.codeblocks.org/index.php?title=Creating_a_new_project
- Alta carga extra classe
- Não utilizar bibliotecas do windows
- Manual de instalação:
  - http://wiki.codeblocks.org/index.php?title=Installing_Code::Blocks

Code::blocks


File Edit View Search Project Build Debug wxSmith Tools Plugins Settings Help
Build target: Debug
Projects
Symbols
Workspace
Sources
main.c
Logs & others
Build log
WINDOWS-1252
Line 10, Column 2
Insert
Read/Write
default

Alternativa

- Sempre pode usar gcc + vi (emacs) + gdb

```
jightuse@debian:~$ gdb -q hello
Reading symbols from /home/jightuse/hello...done.
(gdb) b 1
Breakpoint 1 at 0x8048...: file hello.c, line 1.
(gdb) run
Starting program: /home/jightuse/hello

Breakpoint 1, main (argc=1, argv=0xbffff5f4) at hello.c:5
warning: Source file is more recent than executable.
5               int t = 10;
(gdb) s
7               printf("t = %d\n", t);
(gdb) set var t=100
(gdb) s
8               return 0;
(gdb) s
9       }
(gdb) s
0xb7e89e46 in __libc_start_main () from /lib/i686-linux-gnu/i686/cmov/libc.so.6
(gdb) s
Single stepping until exit from function __libc_start_main,
which has no line number information.
t = 100
Program exited normally.
(gdb) quit
jightuse@debian:~$
```

Tópicos

- Indentação
- Comentários
- Modularização
- Compilação e Debug
- Entrada e saída
- Vetores e Strings
- Passagem de parâmetros
- Structs

Boas Práticas

- Com um pequeno esforço extra, programas escritos em C podem se tornar mais legíveis e mais facilmente "debugáveis"
- No caso de disciplinas de programação, isso ainda ajuda no entendimento das idéias e na correção de trabalhos

Indentação

- É usual empregar TABS para indicar blocos de programação
  - Em geral, 1 tab equivale a 8 espaços, MAS NÃO USAR ESPAÇOS para alinhar
- Há vários estilos
- Quando o bloco é longo, é usual colocar um pequeno comentário após o fechamento indicando o que está sendo fechado

Indentação

- K&R: Kernighan & Ritchie

```
int main(int argc, char *argv[])
{
    ...
    while (x == y) {
        something();
        somethingelse();
        if (some_error)
            do_correct();
        else
            continue_as_usual();
    }
    finalthing();
    ...
}
```

Indentação

- 1TBS: One True Brace Style

```
//...
    if (x < 0) {
        printf("Negative");
        negative(x);
    } else {
        printf("Positive");
        positive(x);
    }
```

Indentação

- Allman

```
while(x == Y)
{
    something();
    somethingelse();
}
finalthing();
```

Comentários

- Importantes para compreensão do código
- Mais importantes em código mais complexo
- Úteis para explicar o conteúdo de variáveis, mas não substituem um bom critério de atribuição de nomes
- Não exagerar!

Comentários

- No início de cada módulo de código (arquivos .c, .h)
- Uma linha em cada função, explicando o que ela faz
  - Não é necessário explicar COMO ela faz, o código deve ser claro o bastante para permitir esse entendimento em uma função razoavelmente pequena
  - Se necessário, considerar a quebra em outras funções
- Comentário na linha da declaração, se necessário, para esclarecer o significado/o uso de uma variável
- Comentário na linha do fecha-chave, para ajudar a entender blocos e loops

Comentários

- No início de um bloco/arquivo fonte

```
/*
 *  earth.c
 *    by Cludoveu Davis (2007)
 *
 *  This program shows spherical texture mapping as in
 *    from http://local.wasp.uwa.edu.au/~pbourke/texture_colour/spheremap/
 *  Sphere drawing routine by Paul Bourke and Federico Dosil (JDL above)
 *  Texture loader, by Chris Leathley (http://members.iinet.net.au/~cleathley/)
 *  Hi-res texture images from http://textures.forrest.cz/cgi-bin/textureslib.cgi?page=maps&ss=3&p=0&re=
 *
 *  Interaction:
 *   x/X, y/Y, z/Z keys: rotate the object along the x, y and z axes, CCW (xyz) and CW (XYZ)
 *   r/R, s/S, t/T keys: scale the object along the x, y and z axes
 *   a/A, b/B, c/C keys: reposition camera along the x, y and z axes
 *   <, >: zoom
 *   .: reset zoom
 *   q: start animation
 *   Q: stop animation
 *   i: reset rotation
 *   I: reset camera position
 *   +/-: change animation speed
 *   =: animation speed reset
 *   Esc: exit program
 */

#define GLUT_DISABLE_ATEXIT_HACK
#include <stdlib.h>
#include <GL/glut.h>
#include "TextureLoader.h"
```

Comentários

- Em função (simples)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int quadrado(int x)
5  /* retorna o quadrado de um numero */
6  {
7      return (x * x);
8  }
9
```

Comentários

- Em funções (arquivo .h)

```
1   #include <stdic.h>
2   #include <stdlib.h>
3   #include "item.h"
4
5   typedef struct {
6           Apontador topo, fundo;
7           int tamanho;
0   } TipoPilha;
9
10  // FUNCOES BASICAS
11
12  void CriaPilhaVazia(TipoPilha *pilha); // Inicializa uma pilha
13
14  int PilhaVazia(TipoPilha pilha); // verifica se a pilha esta' vazia
15
16  void Empilha(TipoPilha *pilha, TipoItem x); // Empiha um item
17
18  void Desempilha(TipoPilha *pilha, TipoItem *x); // Desempilha um item
19
20  int TamanhoPilha(TipoPilha pilha); // retorna o tamanho de uma pilha
21
22  void ImprimePilha(TipoPilha pilha); // imprime o conteúdo da pilha
23
24
25
```

Comentários

- Em variáveis
- Em structs

```
/*****************************************************/
/* UNIVERSIDADE FEDERAL DE MINAS GERAIS              */
/* INSTITUTO DE CIENCIAS EXATAS                      */
/* DEPARTAMENTO DE CIENCIA DA COMPUTACAO             */
/*                                                   */
/* Doutorado em Ciencia da Computacao                */
/* Exame de Qualificacao 1997                        */
/* Simulador de Automatos Finitos                    */
/*                                                   */
/* Clodoveu Augusto Davis Jr. - fev/96               */
/*****************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>

/* GLOBAIS, CONSTANTES E ESTRUTURAS DE DADOS */
#define TRUE 1
#define FALSE 0

#define BOOLEAN int

/* numero maximo de estados */
#define MAXSTATES 100

/* lista de adjacencia - transicoes */
struct ADJ
{
    int state;              // estado
    char symbol;            // simbolo para transicao
    char output;            // simbolo de saida
    struct ADJ *link;       // pointer para outra transicao a partir desse estado
};

/* Estados */
struct STATE
{
    BOOLEAN final,          /* TRUE se o estado pertence a F */
    struct ADJ *adj;            /* lista de adjacencia */
};

/* Automato - vetor de estados */
struct STATE AUTOMATON[MAXSTATES];
```

Comentários

- No fim de um bloco de programa
- No código

```
47      while (!feof(in))
48      {
49          fgets(lin, 1024, in);
50
51          pl = strtok(lin, ".");
52          /* 0 = estado intermediario */
53          /* 1 = estado inicial */
54          /* 2 = estado final */
55          /* 3 = estado inicial e tambem final */
56          flag = atoi(pl);
57          AUTOMATON[i].final = ((flag == 2) || (flag == 3));
58
59          if (!flag == 1) || !flag == 3)) initial_state = i;
60          final state -= AUTOMATON[i].final;
61
62          pl = strtok(NULL, ',');
63          while (pl != NULL)
64              { if ((trans = (struct ADJ *)malloc(sizeof(struct ADJ))) == NULL) {
65                  fprintf(stderr, "ERRO: Memoria insuficiente para corpo do automato\n");
66                  exit(3);
67
68              /* simbolo de entrada - se nul , e'sao transicao-epsilon */
69              if (strcmp(pl, "eps") != 0)
70                  trans->symbol = pl[0];
71              else
72                  trans->symbol = '\0';
73
74              /* estado para o qual transita */
75              pl = strtok(NULL, ',');
76              trans->state = (int)atoi(pl);
77
78              /* simbolo de saida */
79              pl = strtok(NULL, ',');
80              trans->output = pl[0];
81
82              /* inserccao na lista de adjacencia */
83              trans->link = AUTOMATON[i].adj;
84              AUTOMATON[i].adj = trans;
85
86              pl = strtok(NULL, ',');
87          } //while pl
88          i++;
89      } //while !feof
```

Modularização

- Planejar a quebra de um programa em módulos
  - Um módulo não significa necessariamente um arquivo fonte; muitas vezes, é um par de arquivos `.c` / `.h`
  - Existe sempre um arquivo fonte para o programa principal (main), e outros para funções adicionais ou componentes
- Montar módulos especificamente para tipos abstratos de dados [aula: TAD]
- Procurar dar independência aos módulos, para que possam eventualmente ser reaproveitados

Modularização

meuprograma.c

```
#include <stdio.h>
#include <stdlib.h>

int main() {
...
}
```

Modularização

meuprograma.c

```
#include <stdio.h>
#include <stdlib.h>

int main() {
...
}
```

timer.c

```
#include <windows.h>

typedef struct {
    LARGE_INTEGER start;
    LARGE_INTEGER stop;
} stopWatch;

void StartTimer(stopWatch *timer)
{
...
}

void StopTimer(stopWatch * timer)
{
...
}
```

Modularização

meuprograma.c

```
#include <stdio.h>
#include <stdlib.h>

int main() {
...
}
```

Timer.h

```
#include <windows.h>

typedef struct {
    LARGE_INTEGER start;
    LARGE_INTEGER stop;
} stopWatch;

void StartTimer(stopWatch *);

void StopTimer(stopWatch *);
```

Timer.c

```
void StartTimer(stopWatch *timer)
{
...
}

void StopTimer(stopWatch * timer)
{
...
}
```

Modularização

meuprograma.c

```
#include <stdio.h>
#include <stdlib.h>
#include "timer.h"

int main() {
     stopWatch relogio;
...
     startTimer(&relogio);
...
     stopTimer(&relogio);
}
```

timer.h

```
#include <windows.h>

typedef struct {
    LARGE_INTEGER start;
    LARGE_INTEGER stop;
} stopWatch;

void StartTimer(stopWatch *);

void StopTimer(stopWatch *);
```

timer.c

```
void StartTimer(stopWatch *timer)
{
...
}

void StopTimer(stopWatch * timer)
{
...
}
```

Atenção: incluir timer.h e timer.c no projeto

Constantes e #define

- Não usar “números mágicos” no código
- Algumas constantes usadas no programa podem ter que ser modificadas, e é mais fácil fazer isso em um lugar só
- Sempre que for tornar o código mais legível, usar #define para dar um nome à constante
  - Em geral, nomes de constantes são em maiúsculas

```
#define PI 3.141592
#define MAX_VETOR 1000
```

Nomes de variáveis

- Algumas variáveis merecem nomes significativos: `MAX_VETOR`, `numClientes`, `listaAlunos`
- Variáveis auxiliares em geral recebem nomes curtos: `i`, `j`, `aux`, `x`
  - Cuidado para não fazer confusão
  - Não abusar: `i`, `ii`, `iii`, `aux1`, `aux2`, `aux3...`
  - Variáveis inteiras: `i`, `j`, `k`
  - Variáveis reais: `x`, `y`, `z`
  - Strings: `s1`, `s2`
  - Booleanas: nome do teste (`existe`, `valido`, `ocorre`)

Nomes de variáveis

- Estilos variados:
  - Só minúsculas (`i, num, conta`)
  - Só maiúsculas (constantes: `PI, E, MAX`)
  - CamelCase (`numMat, anguloEntrada`)
  - Indicação do tipo no início do nome (`iNum, iValor, fRaio, fAltura, dVolume`)
- Há quem prefira inserir comentários e usar nomes de variáveis em inglês, por ficar mais próximo da linguagem de programação

Organização e limpeza

- Procurar dar um aspecto organizado ao código, ajuda na compreensão
- Entender o código fonte como um instrumento de comunicação
- Comentar excessivamente código mal escrito não ajuda
- Dar nomes adequados a variáveis ajuda bastante

Parênteses e espaçamento

- Usar espaços antes de parênteses, depois de vírgulas, ao redor de operadores binários

```
if (x == 10) y = 5;
for (i = 0; i < 10; i++) {
        x += a;
        a = f(b);
}
```

- Cuidado com notações compactas demais, e com comandos embutidos em outros

```
if (x++ == b) y = 5;
```

Correção e robustez

- Testes: prever todo tipo de problema e variações na entrada de dados
  - Limites de vetores
  - Valores inteiros e de ponto flutuante
  - Contadores e incremento
  - Testes de fim de arquivo
  - Teste de disponibilidade de memória para alocação

Compilação

- LER as mensagens de erro e ENTENDER a origem do problema
- Warnings: indicam problemas potenciais, devem ser resolvidos
- Muitas vezes a mensagem de erro não reflete o que está ocorrendo
  - Observar a linha em que o erro foi indicado, a linha anterior, o bloco de código em que ocorreu, e o corpo da função em que ocorreu

Debugger

- Ajuda a acompanhar os valores das variáveis ao longo da execução
  - Observar o valor de variáveis (*watches*)
  - Posicionar pontos de interrupção (*breakpoints*)
  - Executar passo a passo
- Vide
  http://wiki.codeblocks.org/index.php?title=Debugging_with_Code::Blocks
- Documentação do CodeBlocks
  http://wiki.codeblocks.org/index.php?title=Main_Page