Algoritmos e Lógica de Programação

Profa. Eliane Oliveira Santiago

Modularização

É a divisão do algoritmo em módulos ou sub-rotinas, para que o problema dividido em subproblemas possa ser facilmente interpretado e desenvolvido.

Temos dois tipos de módulos:

procedimentos e **funções**.

Necessidades de modularização

Os principais benefícios para o uso da modularização são:

- a divisão do algoritmo em módulos, para que ele seja melhor interpretado e desenvolvido; e
- a reutilização de código.

Os módulos acabam sendo mais simples e menos complexos.

A divisão em módulos facilita o entendimento parcial do problema e entendendo todos os problemas parciais, no final, teremos entendido o problema maior.

A maioria dos módulos pode ser vista como um minialgoritmo, ou seja, com entrada de dados, processamento desses dados e saída de resultados.

Modularização
dividir o problema em subproblemas.

Para construir os módulos, primeiro precisamos analisar o problema e dividi-lo em partes principais, que são os módulos; depois precisamos analisar os módulos obtidos para verificar se a divisão está coerente. Se algum módulo ainda estiver complexo, devemos dividi-lo também em outros submódulos. Por fim, precisamos analisar todos os módulos e o problema geral para garantir que o algoritmo mais os módulos resolvam o problema de forma simples.

Esse processo é chamado de **Método de Refinamento Sucessivo**, pois o problema complexo é dividido em problemas menores e, a partir do resultado dos problemas menores, teremos o resultado do problema complexo.

Procedimento sem parâmetros de entrada

```
procedimento nomeProcedimento()
Var
     //declaração de variáveis
início
     //bloco de comandos
fimprocedimento
```

Exemplo de Procedimento sem parâmetros de entrada

```
procedimento imprimaDados()
início
     escreva(“Aula de Algoritmos”)
     escreva(“Profa. Eliane”)
fimprocedimento
```

Chamada de Procedimentos

```
Algoritmo "Exemplo"


Var
Inicio
    imprimirDados()

Fimalgoritmo
```

Chamada de Procedimento sem parâmetros de entrada

```
Início
      nomeProcedimento()
Fimalgoritmo
```

Procedimentos

Os procedimentos são aqueles módulos que executam um conjunto de comandos sem retorno para o módulo que o chamou.

```
procedimento <nome do módulo>([lista de parâmetros])
início
      <comandos>;
fim_procedimento
```

A lista de parâmetros poderá ser vazia.

Exemplo: procedimento menu()

procedimento imprimir_dados(n1, n2 : inteiro)

Procedimentos

```
procedimento imprimirDados()
início
  disc, fac : Cadeia
  disc ← "Lógica de Programação”
  fac ← ”Fatec”
      escreva("nome da disciplina => " , disc);
      escreva("nome da instituição de ensino =>" , fac);
fim_procedimento
```

Neste exemplo, **imprimir_dados** é o nome do procedimento. O módulo parece um mini-algoritmo com declaração de variáveis, entrada de dados para essas variáveis e a saída de resultados. O que difere um algoritmo de um módulo é justamente sua simplicidade.

Procedimentos

```
procedimento imprimir_dados() //void imprimir_dados(){..}
início
   disc, fac : Cadeia //o tipo Cadeia é correspondente ao String, mas não existe o pseudocódigo
   disc ← "Lógica de Programação”
   fac ← ”Fatec”
       escreva("nome da disciplina => " , disc);
       escreva("nome da instituição de ensino =>" , fac);
fim_procedimento
```

Neste exemplo, **imprimir_dados** é o nome do procedimento. O módulo parece um mini-algoritmo com declaração de variáveis, entrada de dados para essas variáveis e a saída de resultados. O que difere um algoritmo de um módulo é justamente sua simplicidade.

Funções

As funções são aqueles módulos que executam um conjunto de comando e retornam algum dado para o módulo que o chamou.

```
função <nome_da_função>([lista_parametros]) : <tipo_de_retorno>
var
início
     <comandos>;
     retornar <valor de retorno>;
fimfunção
```

Funções

```
funcao imprimir_dados() : Cadeia
var
 disc, fac, mens : Cadeia
início
     disc ← "Lógica de Programação”
     fac ← ”Fatec”
     mens ← "nome da disciplina => " + disc
            + ”nome da instituição de ensino => "
            + fac
      retorne mens;
fimfunção
```

Note que o tipo de retorno do módulo função **imprimir_dados** é **alfanumérico** e o retorno do módulo é o conteúdo da variável **mens** do tipo **alfanumérico**, ou seja, do mesmo tipo de dados, isso é obrigatório.

Variável global

Uma **variável global** é aquela declarada no início do algoritmo e pode ser utilizada por qualquer parte desse algoritmo, seja nos comandos do próprio algoritmo, bem como, dentro de qualquer módulo que pertença ao algoritmo.

Nesse caso, **sua declaração é feita apenas uma única vez,** não sendo permitido que o mesmo nome de variável seja declarado dentro de qualquer outra parte do algoritmo, por exemplo, dentro de qualquer outro módulo.

Variável local

Uma **variável local** é aquela declarada dentro de algum bloco, por exemplo, dentro de um módulo.

Nesse caso, **essa variável é válida e reconhecida somente dentro do bloco em que foi declarada**. Assim, o mesmo nome de variável pode ser declarado dentro de diferentes módulos (procedimentos ou funções), pois serão reconhecidas como uma nova variável.

Escopo de variáveis: local ou global

O escopo de uma variável é onde, dentro do algoritmo, uma variável é válida ou pode ser reconhecida.

Por exemplo, se declararmos uma variável **x** no início de um algoritmo, essa variável **x** pode ser usada e alterada em qualquer parte desse algoritmo e em nenhum momento declarada novamente, ou seja, ela é única no algoritmo inteiro. Mas é muito importante ter o controle dessas variáveis globais, justamente porque elas podem ser alteradas a qualquer momento.

No entanto, se declararmos, usarmos e alterarmos uma variável **y** dentro do módulo soma, poderemos também declará-la, utilizá-la e alterá-la dentro do módulo subtração, do módulo divisão e do módulo multiplicação, se assim o desejarmos. Nesse caso, não precisando tomar os cuidados necessários que uma variável global precisa.

Parâmetros

O uso de argumentos passados como parâmetros em módulos, sejam eles funções ou procedimentos é muito comum.

Os exemplos vistos anteriormente para procedimentos e funções estão sem o uso de parametrização de módulos, por isso, após o nome dos módulos, os parênteses estão vazios.

É dentro dos parênteses que os argumentos são passados como parâmetros aos módulos.

```
<tipo de retorno><nome do módulo> (<var1>,  <var2> : tipo1, <var3> :<tipo2>)
início_módulo
      <comandos>
      <retorne> valor
fim_módulo;
```

Parametrização em procedimentos
(valores de entrada pro módulo)

A quantidade de argumentos que pode ser passada como parâmetro não é determinada, pondendo ser um único argumento ou uma quantidade finita de argumentos.

```
procedimento imprimir_dados (disc, fac : Cadeia)
início
    escreva("nome da disciplina => " , disc)
    escreva("nome da instituição de ensino => " , fac)
fimprocedimento
```

Neste exemplo, não é mais necessária a declaração das variáveis **disc** e **fac** dentro do módulo **Dados**, pois esses dados estão declarados no cabeçalho do módulo como argumentos que receberão dados que serão passados como parâmetros na chamada desse módulo.

Parametrização em funções
(valores de entrada pro módulo)

A quantidade de argumentos que pode ser passada como parâmetro não é determinada, pondendo ser um único argumento ou uma quantidade finita de argumentos.

função divisao(x: **real**, y: **inteiro**) **: real**
var
    resultado **: real**
início
    resultado ← x/y
    retorne resultado
fim_procedimento

TODA FUNÇÃO RETORNA ALGUM VALOR

A função divisão retorna a variável resultado. Como esta variável é do tipo real, a função retorna um valor do tipo real.

Neste exemplo, não é mais necessária a declaração das variáveis **disc** e **fac** dentro do módulo **Dados**, pois esses dados estão declarados no cabeçalho do módulo como argumentos que receberão dados que serão passados como parâmetros na chamada desse módulo.

Chamada de procedimento
Exemplos

```
//chamada de procedimento          ⇔ voz de comando
imprimir_dados()
imprimir_dados("LPA", "Fatec")

//chamada de função   // ⇔ responde a uma pergunta
a ← divisao(10, 5)
leia(x,y)       //x e y variáveis inteiras
se(soma(x,y)>100) então
       //faça alguma coisa.
Fimse
a ← pot(4,3)          // ⇔ Quanto é 4 elevado a 3 (4³)?
b ← fatorial(8)       // ⇔ Qual é o fatorial de 8?
```

Resumo

Procedimentos

//sem parâmetros de entrada

procedimento nome()

:

fimprocedimento

//com parâmetros de entrada

procedimento nome(x: inteiro)

:

fimprocedimento

Funções

//sem parâmetros de entrada

função nomeFuncao() : **tipo**

var valor : tipo

:

retorne valor

fimfuncao

//com parâmetros de entrada

função nomeFuncao(x: inteiro) : **tipo**

var valor : tipo

:

retorne valor

fimfuncao

Bibliografias

BÁSICA

- GOMES, Ana Fernanda A. Campos, Edilene Aparecida V. Fundamentos da Programação de Computadores – Algoritmos, Pascal e C/C++. Prentice Hall, 2007.
- CARBONI, Irenice de Fátima. Lógica de Programação. Thomson.
- XAVIER, Gley Fabiano Cardoso. Lógica de Programação - Cd-rom. Senac São Paulo – 2007.

COMPLEMENTAR

- FORBELLONE, André Luiz Villar. Eberspache, Henri Frederico. Lógica de Programação – A construção de Algoritmos e Estrutura de Dados. Makron Books, 2005.
- LEITE, Mário - Técnicas de Programação – Brasport - 2006.
- PAIVA, Severino – Introdução à Programação – Ed. Ciência Moderna – 2008.
- PAULA, Everaldo Antonio de. SILVA, Camila Ceccatto da. Lógica de Programação –Viena – 2007.
- CARVALHO, Fábio Romeu, ABE, Jair Minoro.Tomadas de decisão com ferramentas da lógica paraconsistente anotada: Método Paraconsistente de Decisão (MPD), Editora Edgard Blucher Ltda. - 2012.

LPA - Intervalo

- Façam a função que retorna o teste de primalidade

Dado um n, retorne Verdadeiro se n é primo e falso, caso contrário.

```c
#include <stdio.h>

#define VERDADEIRO 1
#define FALSO 0

int verificaPrimo(int x);

int main()
{
    int limInf,limSup;
    printf("Entre com o limite inferior para verificacao: ");
    scanf(" %d", &limInf);
    printf("Entre com o limite superior para verificacao: ");
    scanf(" %d", &limSup);

    for(int numVer=limInf;numVer<=limSup;numVer++){
        if(verificaPrimo(numVer)==VERDADEIRO){
            printf("\n%d eh primo.", numVer);
        /*}else //if(verificaPrimo(numVer)==FALSO){
            printf("\n%d nao eh primo.", numVer);*/
        }
    }
    return 0;
}

int verificaPrimo(int x){
    int primo=FALSO;
    if(x!=1){
        primo=VERDADEIRO;
```

LPA – INTERVALO - Retornaremos às 21h

Exercício: Escrever um programa modularizado para resolver os itens abaixo:

1. Procedimento para escrever o menu
2. Funções para retornar os cálculos abaixo:

Código	Produto Notável	Fórmula
1	Quadrado da diferença de dois números	(a – b) * (a – b)
2	Quadrado da soma de dois números	(a + b) * (a + b)
3	soma do quadrado de dois números	a * a + b * b
4	Diferença do quadrado de dois números	a * a – b * b
5	produto da soma com a diferença de dois números	(a – b) * (a + b)

	Figura geométrica	Fórmula
6	quadrado	lado * lado
7	triângulo	(base * altura) / 2
8	retângulo	base * altura
9	trapézio	((Base maior + base menor) * altura) / 2

Funções Recursivas

Uma função que chama a si própria para uma instância menor do problema.

Função não recursiva

$S(n) = 1 + 2 + 3 + 4 + \ldots + n$

$S(1) = 1$

$S(2) = 1+2$

$S(3) = 1+2+3$

$S(4) = 1+2+3+4$

$S(5) = 1+2+3+4+5$

$S(6) = 1+2+3+4+5+6$

$S(7) = S(6) + 7$

$S(n) = S(n-1) + n$

```
funcao S(n: inteiro) : inteiro
var
   soma : inteiro
inicio
   soma<-0
   enquanto (n>0) faca
         soma <- soma + n
         n<-n-1
   fimenquanto
   retorne soma
fimfuncao
```

Função Recursiva

$S(n) = 1 + 2 + 3 + 4 + \ldots + n$

$S(1) = 1$

$S(2) = 1+2$

$S(3) = 1+2+3$

$S(4) = 1+2+3+4$

$S(5) = 1+2+3+4+5$

$S(6) = 1+2+3+4+5+6$

$S(7) = S(6) + 7$

$S(n) = S(n-1) + n$

```
funcao S(n: inteiro) : inteiro
inicio
    se (n=1) então
        retorne n
    senao
        retorne S(n-1) + n
    fimse
fimfuncao
```

Função Recursiva

$S(n) = 1 + 2 + 3 + 4 + \ldots + n$

$S(1) = 1$

$S(2) = 1+2$

$S(3) = 1+2+3$

$S(4) = 1+2+3+4$

$S(5) = 1+2+3+4+5$

$S(6) = 1+2+3+4+5+6$

$S(n) = S(n-1) + n$

Função Fatorial Recursiva

F (n) = n * Fatorial(n-1)

F(0) = 1

F(1) = 1

F(2) = 1* 2

F(3) = 1* 2*3

F(4) = 1*2*3*4

F(5) = 1*2*3*4*5

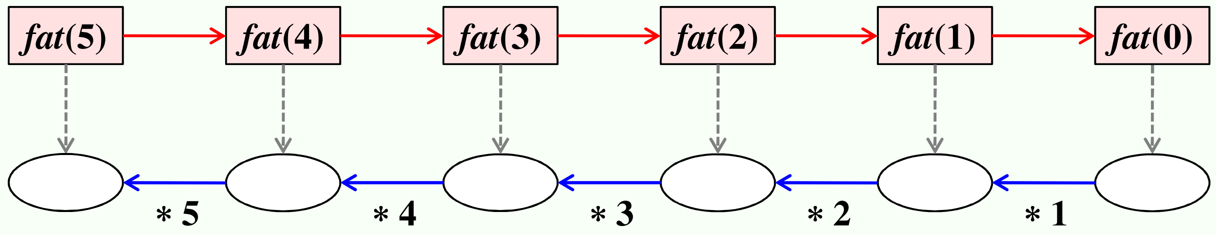
F(6) = 1*2*3*4*5*6

F(7) = S(6)*7

F(n) = S(n-1) * n

```
funcao F(n: inteiro) : inteiro
inicio
    se ((n=1) ou (n=0)) então
        retorne 1
    senao
        retorne F(n-1) * n
    fimse
fimfuncao
```

Exemplo 6. Simulação por fluxo de execução para `fat(5)`



Função Fatorial Recursiva

F (5) = 5 * F(4)

F(4) = 4 * F(3)

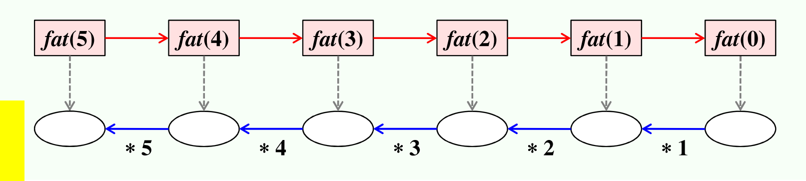
F(3) = 3 * F(2)

F(2) = 2 * F(1)

F(1) = 1

```
funcao F(n: inteiro) : inteiro
inicio
   se ((n=1) ou (n=0)) então
      retorne 1
   senao
      retorne n* F(n-1)
   fimse
fimfuncao
```

Exemplo 6. Simulação por fluxo de execução para `fat(5)`



Intervalo: Retornaremos às 10h10

Função Fatorial Não Recursiva

```
funcao fatorial(n : inteiro) : inteiro
var
   fat, i : inteiro
Inicio
   fat<-1
   para i de 1 ate n passo 1 faÁa
      fat<- fat*I
   fimpara
   retorne fat
Fimfuncao

//Bloco do Programa Principal
Var
  x : inteiro
Inicio
  x<-4
  escreva(x, "! = ", fatorial(x))
Fimalgoritmo
```

Escreva uma função para calcular o somatório S(n) = 1 + 2 + 3 + ... + n

```
funcao S(n: inteiro) : inteiro
var
   soma : inteiro
inicio
   soma<-0
   enquanto (n>0) faca
         soma <- soma + n
         n<-n-1
   fimenquanto
   retorne soma
fimfuncao
```

```
var
     x: inteiro

X <- S(5) //X = 0+5+4+3+2+1
```

Exercícios

1. Escreva um programa para mostrar todos os subconjuntos de um conjunto de n elementos (sem repetir nenhum conjunto).

   Fundamentação teórica:

Dado o conjunto A={1,2,3}, o conjunto das partes de A é

P(A)= $\{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\},\{2,3\}, \{1,2,3\}\}$

Calculando o k-ésimo número de Fibonacci usando Recursão Linear

```
algoritmo LinearFibonacci(k)
        Entrada: um inteiro não-negativo k
        Saída: um par de números Fibonacci(Fk +Fk-1)
        se k <=1 então
                retorna (k,0)
        senão
                (i,j) ← LinearFibonacci(k-1)
                retorna(i+j,i)
fimalgoritmo
```

Calculando o k-ésimo número de Fibonacci usando Recursão

```
Algoritmo Fib(k)
var
   k: inteiro
inicio
   se (k <=1) então
       retorne (k)
   senão
       retorne Fib(k-1) + Fib(k-2)
Fimalgoritmo
```

Sequência de Fibonacci

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89

Fib(1) = 1

Fib(2) = 1

Fib(3) = Fib(2) + Fib(1)

Fib(3) = 1 + 1

Fib(4) = Fib(3) + Fib(2)

2 + 1

//Saída: um par de números Fibonacci(F_k +F_{k-1})

O Que faz o algoritmo abaixo?

```
public static int maior(int v[], int inicio, int fim) {
        int meio = (inicio+fim)/2;
        int n1, n2;
        if(meio>inicio){
                n1=maior(v, inicio, meio);
                n2=maior(v, meio+1, fim);
        } else{
                n1=v[inicio];
                n2=v[fim];
        }
        if(n1>n2) return n1; else return n2;
}
```

Solução do Vinicius

```
Algoritmo "Fibonacci_solucaoVinicius”
// escreve a sequencia de Fibonacci de 1 ate n
procedimento Fib(n : inteiro)
var
   f1, f2 : inteiro
inicio
   f1<-1
   f2<-1
   enquanto (f1<=n) faca
       f2 <- f1-f2
       escreva(f2)
       f1<-f1+f2
   fimenquanto
fimprocedimento
//bloco de programa principal
 Var
 Inicio
   Fib(90)
 Fimalgoritmo
```

Algoritmo recursivo que encontra o maior elemento de um vetor de inteiros sem usar nenhum laço.

```
public static int maior(int v[], int inicio, int fim) {
        int meio = (inicio+fim)/2;
        int n1, n2;
        if(meio>inicio){
                n1=maior(v, inicio, meio);
                n2=maior(v, meio+1, fim);
        } else{
                n1=v[inicio];
                n2=v[fim];
        }
        if(n1>n2) return n1; else return n2;
}
```

```c
int potencia(int x, int n){
    int pot = 1;
    while(n>0){
        pot = pot * x;
        n=n-1;
    }
    return pot;
}

int potenciaRecursiva(int x, int n){
    if (n==0) return 1;
    return x*potencia(x, n-1);
}

int main()
{
    printf("\n 2 elevado a 5 = %d", potencia(2,5));
    printf("\n 2 elevado a 5 = %d", potenciaRecursiva(2,5));
    return 0;
}
```

Crie a função recursiva e não recursiva para efetuar o cálculo da Série Harmônica

Crie a função recursiva $h(n)$, que calcula a soma dos $n \geq 1$ primeiros termos da série harmônica $(1 + 1/2 + 1/3 + \ldots + 1/n)$.