Universidade Federal do Vale do São Francisco
Curso de Engenharia de Computação

Introdução a Algoritmos – Parte 08

(Baseado no Material do Prof. Marcelo Linder)

Prof. Jorge Cavalcanti
jorge.cavalcanti@univasf.edu.br
www.univasf.edu.br/~jorge.cavalcanti
www.twitter.com/jorgecav

Recursividade

Conceitos

- Alguns problemas são definidos com base nos mesmos, ou seja, podem ser descritos por instâncias do próprio problema.
- Para tratar estas classes de problemas, utiliza-se o conceito de recursividade.
- Um módulo recursivo é um módulo que em sua seção de comandos chama a si mesmo.
- Uma grande vantagem da recursividade é o fato de gerar uma redução no tamanho do algoritmo, permitindo descrevê-lo de forma mais clara e concisa.

Recursividade

Conceitos

- Porém, todo cuidado é pouco ao se fazer módulos recursivos. A primeira coisa a se providenciar é um critério de parada, o qual vai determinar quando o módulo deverá parar de chamar a si mesmo.
- Este cuidado impede que o módulo se chame infinitas vezes.
- Para todo algoritmo recursivo existe um outro correspondente iterativo (não recursivo), que executa a mesma tarefa.
- Implementar um algoritmo recursivo, partindo de uma definição recursiva do problema, em uma linguagem de programação de alto nível como C é simples e quase imediato, pois o seu código é praticamente transcrito para a sintaxe da linguagem

Recursividade

- A definição de recursividade (recursão) aplica-se a funções e procedimentos. Por isso, vale (re)lembrar os seus conceitos:
  - Função: é um módulo que produz um único valor de saída. Ela pode ser vista como uma expressão que é avaliada para um único valor, sua saída, assim como uma função em Matemática.
  - Procedimento: é um tipo de módulo usado para várias tarefas, não produzindo valores de saída.

- Como a diferença entre função e procedimento é sutil, utilizaremos os termos funções e procedimentos de forma indiscriminada neste conteúdo.

Recursividade

- Até agora, foram vistos exemplos de procedimentos chamados genericamente de iterativos. Recebem este nome pois a repetição de processos neles inclusos fica explícita, através do uso de laços.
- Um exemplo de procedimento iterativo, para cálculo do fatorial de um número n, pode ser visto a seguir:

```
Função Fatorial (n)
    fat ←1
    para i ←1 até n faça
        fat ← fat * i
    fimpara
    retorna fat
    fimfunção
```

Recursividade

- Um exemplo de um problema passível de definição recursiva é a operação de multiplicação efetuada sobre números naturais.
- Podemos definir a multiplicação em termos da operação mais simples de adição.
  - No caso

$$A * B$$

  - Pode ser definido como

$$A + A * (B - 1)$$

- Precisamos agora especificar um critério de parada. Qual seria?

$$A * 0 = 0$$

Recursividade

- Com base no que vimos podemos, também, definir um módulo recursivo que implemente a operação de multiplicação com base na operação de adição:

```
funcao multiplicar (A: inteiro; B: inteiro): inteiro
inicio
        se (B=0) entao
           retorne (0)
        senao
           retorne (A + multiplicar (A, B-1))
        fimse
fimfuncao
```

Recursividade

```
algoritmo "exemplo recursividade"
var
   a, b, res: inteiro
   funcao multiplicar (A: inteiro; B: inteiro): inteiro
   inicio
         se (B=0) entao
            retorne (0)
         senao
            retorne (A + multiplicar (A, B-1))
         fimse
   fimfuncao
inicio
   repita
      escreva ("Multiplicando (valor natural): ")
      leia (a)
   ate (a>=0)
   repita
      escreva ("Multiplicador (valor natural): ")
      leia (b)
   ate (b>=0)
   res <- multiplicar(a,b)
   escreva (a," *",b," =",res)
fimalgoritmo
```

Recursividade

- Estratégia para a definição recursiva de uma função:
    1. Dividir o problema em problemas menores do mesmo tipo
    2. Resolver os problemas menores (dividindo-os em problemas ainda menores, se necessário)
    3. Combinar as soluções dos problemas menores para formar a solução final
- Ao dividir o problema sucessivamente em problemas menores eventualmente os casos simples são alcançados:
    - Não podem ser mais divididos
    - Suas soluções são definidas explicitamente

Recursividade

- **Exemplo - Função Fatorial (!)**

- Esta função é um dos exemplos clássicos de recursividade e, por isso, de citação quase obrigatória. Eis sua definição recursiva:

$$n! = \begin{cases} 1 & , \text{se } n \leq 1 \\ n * (n-1)! & , \text{caso contrário} \end{cases}$$

Dado um número inteiro n ≥ 0, computar o fatorial n!.

- Usamos uma fórmula que nos permite naturalmente escrever uma função recursiva para calcular n! :

Recursividade

- Por exemplo, calculando o fatorial de 6:

6! = 6 x 5!

fatorial 5 * 6

(fatorial 4 * 5) * 6

((fatorial 3 * 4) * 5) * 6

(((fatorial 2 * 3) * 4) * 5) * 6

((((fatorial1 * 2) * 3) * 4) * 5) * 6

(((((fatorial 0 * 1) * 2) * 3) * 4) * 5) * 6

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

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

(((2 * 3) * 4) * 5) * 6

((6 * 4) * 5) * 6

(24 * 5) * 6

120 * 6

720

Recursividade

- Outro exemplo: pela definição, valor de 4! é calculado como:

4!=4*3!=4*(3*2!)=4*(3*(2*1!))=4*(3*(2*(1*0!)))=4*(3*(2*(1*1)))=24

- Note que função é chamada recursivamente com argumento decrescente até chegar ao caso trivial (0!), cujo valor é 1.
- Este caso trivial (condição de parada) encerra a sequência de chamadas recursivas. A sequência de chamadas é melhor ilustrada abaixo:

4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1! **Cada chamada é empilhada**
1! = 1 * 0!
<u>0! = 1 (Condição de parada)</u>
1! = 1
2! = 2 **Cada chamada é desempilhada**
3! = 6
4! = 24

Recursividade

```
algoritmo "fatorial recursivo"
var
   n: inteiro
   funcao fatorial (num: inteiro): inteiro
   inicio
      se (num=0) entao
         retorne (1)
      senao
         retorne (num * fatorial(num-1))
      fimse
   fimfuncao
inicio
   escreva("Digite o número que você deseja saber o fatorial: ")
   leia (n)
   se (n>=0) entao
      escreva ("O fatorial do número ",n," é ",fatorial(n))
   senao
      escreva("Não existe fatorial de números negativos!")
   fimse
fimalgoritmo
```

Recursividade

- Um outro exemplo muito utilizado de problema que possui uma definição recursiva é a geração da série de Fibonacci:

{0,1,1,2,3,5,8,13,21,34, ...}

Essa série pode ser definida como:

$$\mathbf{Fib(n)} = \begin{cases} \mathbf{0} & \mathbf{, se\ n = 0} \\ \mathbf{1} & \mathbf{, se\ n = 1} \\ \mathbf{Fib(n-1) + Fib\ (n-2)} & \mathbf{, se\ n>1} \end{cases}$$

- Uma função recursiva que recebe a posição do elemento na série e retorna seu valor é:

```
funcao fibonacci (i: inteiro): inteiro
inicio
        se (i=1) entao
          retorne (0)
        senao
          se (i=2) entao
             retorne (1)
          senao
             retorne (fibonacci(i-1) + fibonacci(i-2))
          fimse
     fimse
fimfuncao
```

Recursividade

- Com base no que foi exposto, podemos visualizar algumas desvantagens da utilização de recursividade, como:
  - O consumo de memória necessário para a troca de contexto.
  - Redução do desempenho de execução devido ao tempo para gerenciamento de chamadas.
  - Dificuldades na depuração de programas recursivos, especialmente se a recursão for muito profunda.

Recursividade

- Fora os problemas mencionados, gerados pela recursão, qual seria outro problema proveniente da recursão evidenciado na função recursiva apresentada para o cálculo do valor de um elemento da série de Fibonacci com base na sua posição?

- O cálculo do mesmo elemento da série n vezes.

fibonacci(5)

fibonacci(4) + fibonacci(3)

fibonacci(3) + fibonacci(2)

fibonacci(2) + fibonacci(1)

fibonacci(2) + fibonacci(1)

Recursividade

Obs.: Mesmo problemas que possuem uma definição recursiva também podem ser solucionados de forma imperativa. Um exemplo disso é o cálculo do valor de um elemento da série de Fibonacci com base na sua posição através da função imperativa abaixo:

```
funcao fibonacci (i: inteiro): inteiro
var a, b:inteiro
inicio
      se (i=1) entao
         retorne (0)
      senao
         se (i=2) entao
           retorne (1)
       senao
            a<-0
            b<-1
        enquanto (i-2<>0) faca
               b<-b+a
               a<-b-a
               i<-i-1
        fimenquanto
            retorne (b)
         fimse
      fimse
fimfuncao
```

Recursividade

- Assim como a série de Fibonacci existem outras sequencias definidas por recorrência, ou seja, onde um valor da sequencia é definido em termos de um ou mais valores anteriores, o que é denominado de relação de recorrência.

Exercício:

Estabeleça a relação de recorrência presente na sequencia abaixo e construa uma função recursiva que recebe a posição do elemento na série e retorna seu valor.

$$S = \{ 2, 4, 8, 16, 32 \ldots \}$$

Recursividade

A sequencia S é definida por recorrência por

1. S(1) = 2
2. S(n) = 2 * S(n-1) para n>=2

Uma função recursiva que recebe a posição do elemento na série e retorna seu valor é:

```
funcao func (p: inteiro): inteiro
inicio
se (p=1) entao
        retorne (2)
    senao
retorne (2*func(p-1))
fimse
fimfuncao
```

Recursividade

Exercício 1

Elabore um algoritmo recursivo capaz de receber como parâmetro um número natural e calcula a potência de base 2 desse número, usando somente um caso base e multiplicação.

Ex. n=3, 2^3 => 8.

Caso base: $2^0 = 1$

Caso Recursivo: $2^n = 2 * 2^{(n-1)}$

Recursividade

Exercício 2

Elabore um algoritmo recursivo capaz de receber como parâmetro um número natural, na base decimal, convertê-lo para sua representação na base binária, retornando o resultado desta operação na saída padrão.

Recursividade

Exercício 3

Elabore um módulo recursivo que receba dois números inteiros, como parâmetros, e retorne o resultado do somatório de todos os números contidos no intervalo aberto delimitado pelos números fornecidos. Em seguida, construa um algoritmo que se utilize de forma eficaz do módulo elaborado.

Recursividade

Exercício 4

Elabore um algoritmo com módulo recursivo que resolva a seguinte relação de recorrência:

1. S(1) = 5
2. S(n) = S(n-1) + 5 para n>=2

Recursividade

Exercício 5

Elabore um algoritmo com módulo recursivo que resolva o seguinte somatório:

$$\sum_{i=1}^{n} (2 * i^2 + 2 * i + 8)$$