SPOEDD1 – Estrutura de Dados

IFSP – Campus São Paulo

Antonio Ferreira Viana

2024

Recursividade

Recursividade

- Recursividade é uma técnica de programação na qual uma função (ou método) pode chamar a si mesma.
- Ela é bastante usada para simplificar a resolução de problemas que demandariam muitos passos.
- A idéia principal é que um problema pode ser particionado em problemas menores, ou de mais fácil resolução, até um ponto chamado de caso base, de onde não se pode mais particionar, considerado o caso trivial.
- Um exemplo muito utilizado é o cálculo do fatorial.

Recursividade

- O **fatorial** é um número natural inteiro positivo, sendo representado por **n!**, na matemática.

  O cálculo do fatorial de um número é obtido pela multiplicação desse número por todos os seus antecessores até chegar ao número 1.

  Assim: 3! = 3 * 2 * 1 = 6
  4! = 4 * 3 * 2 * 1 = 24 (1*2*3*4)
  5! = 5 * 4 * 3 * 2 * 1 = 120 (1*2*3*4*5)

  Além disso, por definição:

  1! = 1
  0! = 1
  n! = n * (n-1)!

Recursividade

- **Exemplo de implementação em C de uma função recursiva para cálculo do fatorial de um número:**

```
int  fatorial_recursivo (int n)
{
      if ( (n == 0) || (n == 1) )   /* CASO BASE ou de PARADA*/
            return 1;
      else                          /* CASO RECURSIVO */
            return ( n * fatorial_recursivo (n-1) );
}
```

Recursividade

- **Em geral, uma função definida recursivamente pode ser também definida de uma forma iterativa (através de estruturas de repetição).**
- **A definição recursiva é mais "declarativa" – explicita o que se pretende obter e não a forma como se obtém (ou seja, o algoritmo que é usado).**

Recursividade

- Por outro lado, uma definição iterativa, embora não permita uma compreensão tão imediata, é geralmente mais eficiente, dado que a implementação recursiva precisa registrar o estado atual do processamento para continuar de onde parou após a conclusão de cada nova execução, e isso consome tempo e memória.

Recursividade

- Implementação de funções para cálculo do fatorial de um número (uma recursiva e outra iterativa)

```c
/* Bibliotecas */
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <conio.h>
/* Funções */
int fatorial_recursivo(int n)
{
     if (n == 0 || n == 1)
          return 1;
     else
          return ( n * fatorial_recursivo (n-1) );
}

int fatorial_nao_recursivo (int n)
{
    int resultado, contador;
    resultado = 1;
    if ( n!=0 )
    {
        for(contador=n; contador >= 1; contador--)
        {
            resultado = resultado * contador;
        }
    }
    return resultado;
}
```

Recursividade

PROPOSTA:

Crie um programa em C para calcular a soma dos números inteiros entre 1 e n, onde n é fornecido pelo usuário como entrada, com n maior que zero.

Crie duas funções soma (uma recursiva e a outra não recursiva) que recebe como parâmetro de entrada o número lido.

NOTAR QUE:

soma_recursiva (n) = { 1, se n = 1 (solução trivial, caso base)
n + soma_recursiva (n-1), se n > 1 (solução recursiva)

Recursividade

- **Algumas operações matemáticas ou objetos matemáticas têm uma definição recursiva**
  - **Ex: fatorial, sequência de Fibonacci, palíndromos, etc...**
  - **ou podem ser vistos do ponto de vista da recursão:**
    - **multiplicação, divisão, exponenciação, etc...**

Isso nos permite projetar algoritmos para lidar com essas operações/objetos

Recursividade

- **Exponenciação**
- **Seja a é um número real e b é um número inteiro não-negativo**
  - **Se b = 0, então $a^b = 1$**
  - **Se b > 0, então $a^b = a \cdot a^{b-1}$**

```
double potencia (double a, int b) {
  if (b == 0)
    return 1;
  else
    return a * potencia(a, b-1);
}
```

Recursividade

- **Palíndromos**

Uma palavra é um palíndromo se ela é igual ao seu reverso (Exemplos: arara, osso, ovo, radar, etc.)

Matematicamente, uma palavra é palíndromo se:

- **ou tem zero letras (palavra vazia)**
- **ou tem uma letra**
- **ou é da forma αpα onde:**
  - **α é uma letra**
  - **p é um palíndromo**

Recursividade

- **Palíndromos**

Uma palavra é um palíndromo se ela é igual ao seu reverso (Exemplos: arara, osso, ovo, radar, etc.)

```
int eh_palindromo(char   *palavra ,   int ini,   int fim) {
    if (ini >= fim)
            return 1;
    int A = palavra[ini] == palavra[fim];
    int B = eh_palindromo(palavra , ini+1, fim-1);
    return   (A && B);
}
res = eh_palindromo(palavra , 0, strlen(palavra)-1);
```

Recursividade

- **Busca Binária**

Para buscar um valor **x** no vetor de **dados ordenado** **entre as posições left e right**

Casos base:

Se o intervalo for vazio (left > right), x não está no vetor

Se dados[m] == x, onde m = (left + right)/2

– Devolvemos m

Caso geral:

Se dados[m] < x, então x só pode estar entre m + 1 e right

– Devolvemos o resultado da chamada recursiva

Se dados[m] > x, então x só pode estar entre left e m - 1

– Devolvemos o resultado da chamada recursiva

Recursividade

- **Busca Binária**

```
int busca_binaria(int *dados, int l , int r , int x) {
    int m = (l+r)/2;
    if (l > r)
            return –1;          /* não encontrou */
    if (dados[m] == x)
            return m;
    else
            if (dados[m] < x)
                    return busca_binaria(dados, m + 1, r, x);
            else
                    return busca_binaria(dados, l, m – 1, x);
}
```

Recursividade

- **Comparando recursão e algoritmos iterativos**

Normalmente algoritmos recursivos são:

- **mais simples de entender**
- **menores e mais fáceis de programar**
- **mais “elegantes”**

Mas algumas vezes podem ser

- **muito ineficientes (quando comparados a algoritmos iterativos para o mesmo problema)**

Estratégia ideal:

1. **encontrar algoritmo recursivo para o problema**
2. **reescrevê-lo como um algoritmo iterativo**

Recursividade

- **Sequência de Fibonacci**

Em 1202, um matemático italiano conhecido como Fibonacci propôs o seguinte problema:

Um coelho macho e uma fêmea nascem no início do ano. Assumindo que:

- *Depois de dois meses de idade, o casal de coelhos produz um par misto (um macho e uma fêmea), e então outro par misto de coelhos a cada mês.*
- *Não ocorrem mortes durante o ano.*

Quantos coelhos teremos no fim do ano?

Recursividade

- **Sequência de Fibonacci**

Considere f_n o número de pares de coelhos no começo no mês n Assim, temos que:

$$f_0 = 1,\ f_1 = 1,\ f_2 = 2,\ f_3 = 3,\ f_4 = 5,\ f_5 = 8, ...$$

Assim, podemos observar que $\mathbf{f_n = f_{n-1} + f_{n-2}}$. Isto significa que todos os termos da sequencia (a partir de n = 2) são obtidos pela soma dos dois termos anteriores.

A sequência gerada a partir dessa construção, foi nomeada Sequência de Fibonacci, em homenagem ao matemático Leonardo Fibonacci, também conhecido como Leonardo de Pisa.

Recursividade

- **Sequência de Fibonacci**

Ao final de um ano (12 meses) teremos f_{12} = 144 casais de coelhos.

*Um problema é obter o valor $\boldsymbol{f_n}$ quando **n** é um número muito grande.*

Nesse caso a expressão não é muito viável, uma vez que precisamos calcular todos os termos anteriores a f_n.

Desafio: Como resolver essa situação?

Recursividade

- **Sequência de Fibonacci**

Elabore um programa recursivo e outro iterativo para calcular o número de casais de coelhos, sendo dado o valor de n. No caso, equivale a descobrir o n-ésimo elemento da Sequência de Fibonacci.

Caso tenha interesse, procure saber sobre a ocorrência da Sequência de Fibonacci na natureza.

Recursividade

Tarefas:

Elabore um programa recursivo e outro iterativo para converter um número na base dez para outro na base dois.

Recursividade

Tarefas:

Elabore um programa recursivo e outro iterativo para converter um número na base dez para outro na base dois.

Escreva uma função recursiva que determine quantas vezes um dígito K ocorre em um número natural N.

O máximo divisor comum (MDC) de dois números inteiros x e y pode ser calculado usando-se uma definição recursiva:

- MDC(x, y) = MDC(x − y, y), se x > y
- MDC(x,y) = MDC(y,x)
- MDC(x,x) = x

Recursividade

Tarefas:

Implemente uma função recursiva soma(n) que calcula o somatório dos n primeiros números inteiros.

Pode-se calcular o resto da divisão, MOD, de x por y, dois números inteiros positivos, usando-se a seguinte definição:

- MOD(x,y) = MOD(x - y, y) se x > y
- MOD(x,y) = x se x < y
- MOD(x,y) = 0 se x = y