MC 102 turma Z - 2sem2012
Linguagem C
Funções Recursivas

Prof. Fernando Vanini

IC - Unicamp

Introdução

Na apresentação sobre alocação dinâmica de memória, foi mostrada a uma função para inserir um elemento numa lista:

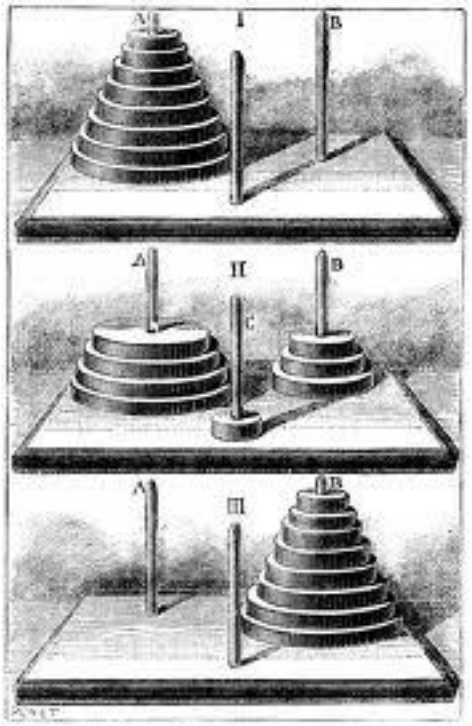
```
struct Aluno *insere2( struct Aluno* a,
                                struct Aluno* lista){
      *a.proximo = NULL;
      if(lista == NULL) return a;
      insere2(a,lista.proximo);
      return lista;
   }
```

Esse é um exemplo de função recursiva: uma função que durante sua execução, faz uma chamada a si própria (direta ou indiretamente).

Funções recursivas podem ser muito úteis para resolver problemas cuja definição é descrita de forma *recursiva*. Uma lista, por exemplo, pode ser descrita como : uma lista é vazia ou é formada por um elemento seguido de uma lista.

A função acima se baseia nessa descrição de lista para inserir um novo elemento na lista: se a lista é vazia, basta criar uma lista com o novo elemento; caso contrário, inserimos o novo elemento na lista que se segue ao primeiro elemento.

Um exemplo clássico: as torres de Hanói



‘Torres de Hanói ‘ é um tipo de quebra-cabeça formado por três pinos e uma série de discos de tamanhos diferentes, que podem ser encaixados em qualquer um dos pinos. Inicialmente os discos são dispostos num dos pinos em ordem decrescente de diâmetro (de baixo para cima), formando uma ‘torre’. O objetivo do quebra cabeças é mover essa torre inteira de um pino para outro, obedecendo às seguintes regras:

- mover apenas um disco por vez.
- um movimento consiste em retirar o disco no topo de uma torre e colocá-lo no topo de outra torre.
- um disco não pode ser colocado em cima de outro menor que ele.

Um exemplo clássico: As torres de Hanói

Um caso simples: para uma torre com dois discos, do pino1, para o pino 3 a sequência de movimentos seria:

1. mover disco de 1 para 2
2. mover disco de 1 para 3
3. mover disco de 2 para 3

A sequência acima pode ser a base para um caso genérico: uma torre com *n* discos nada mais é que uma torre formada por um disco sob uma torre com *n-1* pinos menores que ele. Nessa linha, para mover uma torre com *n* discos, a sequência de 'movimentos' pode ser inspirada na sequencia acima:

1. mover torre de tamanho *n-1* de 1 para 2
2. disco de 1 para 3
3. mover torre de tamanho *n-1* de 2 para 3

Notar que

- se (*n-1*) for igual a zero, não é necessário fazer nada.
- os passos 1 e 3 são 'recursivos'

As torres de Hanói: o programa

```
void moveDisco(int a, int b){
      printf("mover disco de %d para %d\n",a,b);
}

void moveTorre(int n, int a, int b, int c){
    if(n > 0){
        moveTorre(n-1,a,c,b);
        moveDisco(a,c);
        moveTorre(n-1,b,a,c);
    }
}
```

Outro exemplo (1)

Exercício resolvido em sala, com a participação da classe:

Escreva a função 'separa()' que tem os seguintes parâmetros:
um vetor de inteiros v
índices i e j, inteiros que delimitam um intervalo nesse vetor (i >= j).
A função deve rearranjar os elementos do vetor da seguinte forma:
supondo w o valor do último elemento do intervalo (v[j])
- o trecho v[i] .. v[k-1] deve conter os valores de v menores que w
- v[k] deve conter o valor w
- 0 trecho v[k+1] .. v[j] deve conter os valores de v maiores ou iguais a w.

Suponha disponível a função troca(int v[], int i, int j); que troca o valor de v[i] com v[j].

Outro exemplo (2)

A solução feita em sala:

```
int separa(int v[], int i, int j){
    int jj = j;
    int k = v[j];
    while(1){
        while(v[i] < k)  i++;
        while(v[j] >= k) j--;
        if(i < j) troca(v,i,j);
        else {
            troca(v,i,jj);
            return i;
        }
    }
}
```

Outro exemplo (3)

Algumas observações interessantes sobre o trecho de um vetor processado por **`separa()`**:

- se o trecho tiver 3 elementos ou menos, o trecho está ordenado
- após uma chamada a

  k = separa(v,0,n-1);

onde *n* é o tamanho do vetor, para ordenar esse vetor, é suficiente ordenar o primeiro trecho (de 0 a k-1) e o segundo trecho (de k+1 a n-1). Isso acontece porque após a chamada a separa(), o elemento v[k] não muda de posição ao se ordenar o vetor (os 'anteriores' são menores que ele, e os 'seguintes' são maiores).

Outro exemplo (4)

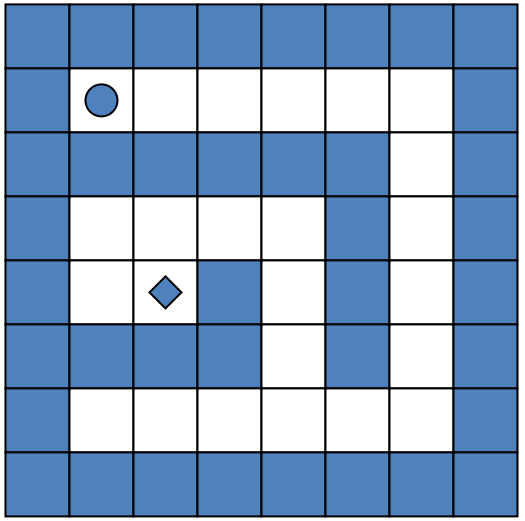
O método de ordenação:

```
void ordena(int v[],int i, int j){
    if(i >= j) return;
    int k = separa(v,i,j);
    ordena(v,i,k-1); ordena(v,k+1,j);
}
```

Esse método de ordenação é conhecido como ‘quicksort’, foi inventado na década de 50 por Anthony Hoare e na média é dos métodos mais eficientes para ordenação de um vetor.

Recursão e retrocesso

- Existem problemas nos quais várias alternativas devem ser ‘tentadas’.
- Se uma alternativa não é bem sucedida, voltar ao ‘estado anterior’ e tentar a próxima alternativa.
- Exemplo clássico: Procurar um caminho num labirinto



Recursão e retrocesso

- Idéia geral: a estrutura geral de uma função para resolver esse tipo de problema segue um padrão semelhante ao mostrado abaixo

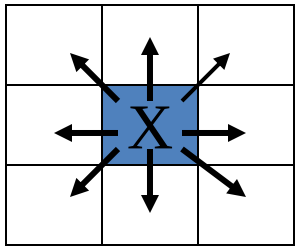
```
void tenta( alternativa ){
  if( alternativa é viável  {
      marca alternativa selecionada
      if( atingiu objetivo )  encontrou solução;
      else  para toda alternativa <alt> disponível no estado atual
            tenta( <alt> );
      'desmarca' alternativa selecionada
  }
}
```

Um exemplo (labirinto)

```
/* matriz que define o 'labirinto'*/
int m[8][8];

/* dx e dy demarcam as 'células vizinhas' */
int dx[8] = {-1,-1,-1,0,0,1,1,1};
int dy[8] = {-1,0,1,-1,1,-1,0,1};

/* a função tenta() */
void tenta(int i, int j){
int k;
  if(m[i][j] == ' '){          /* alternativa é viável ?    */
    m[i][j] = 'x';             /* marcar o caminho          */
    if((i==DI)&&(j==DJ))       /* atingiu o objetivo ?      */
        imprime();
    else {                     /* tentar todas alternativas */
      for(k=0; k < 8; k++) tenta(i+dx[k],j+dy[k]);
    }
    m[i][j] = ' ';             /* desfazer a marca          */
  }
}
```



Um exemplo (labirinto)

```
/* inicia a matriz m */
void inicia(){
  int i,j;
  for(i=0; i < 8; i++)
    for(j=0; j < 8; j++) m[i][j] = ' ';
  for(i=0; i < 8; i++){  /* 'bordas' do labirinto */
    m[i][0] = '#';
    m[i][7] = '#';
    m[7][i] = '#';
    m[0][i] = '#';
  }
  for(i=0; i < 6; i++){ /* 'paredes' internas */
    m[2][i] = '#';
    m[4][i+2] = '#';
  }
  m[5][2] = '#';
  m[5][4] = '#';
}
```

Finalizando

- Funções recursivas são úteis em situações onde
  - O problema pode ser expressado de forma recursiva (como no caso das listas ligadas e das Torres de Hanói)
  - O problema exige que se tente ou experimente várias alternativas e cada alternativa é também expressada de forma recursiva (é o caso do exemplo do labirinto).

Finalizando

- Dada uma função recursiva, é sempre possível escrever uma função equivalente não recursiva
  - Se a chamada recursiva ocorre uma única vez, é relativamente simples substituir o trecho onde a mesma ocorre por um comando repetitivo (while, for ou do-while).
  - Se a função faz mais de uma chamada recursiva, a eliminação da recursão ainda é possível, mas pode ser mais difícil e nem sempre vale a pena.
- Funções recursivas têm um custo associado
  - A chamada de uma função implica na alocação de memória para suas variáveis locais e dados de controle. Chamadas recursivas não são diferentes e também alocam memória.
- Outro exemplo feito em sala: lab01
  - Foi disponibilizada na página do curso a solução para a primeira atividade de laboratório, usando o método das tentativas baseadas em chamadas recursivas.