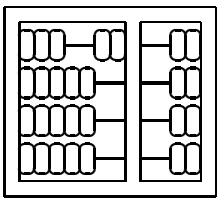

MC102 – ALGORITMOS E PROGRAMAÇÃO DE COMPUTADORES
Instituto de Computação — UNICAMP
1º Semestre de 2005
Turmas Q e T

Profa.: Amanda Meincke Melo
amanda.melo@ic.unicamp.br

Aula 25 - Recursão

1 Objetivos

- Compreender quando não usar recursão.
- Conhecer outros algoritmos recursivos.

2 Motivação

Em geral, programas recursivos são mais simples de escrever, analisar e entender. Entretanto, há momentos em que mesmo um cálculo tendo sido definido de forma recursiva, deve-se evitar a recursão em sua implementação computacional afim de poupar recursos (ex. memória, capacidade de processamento) e promover o desenvolvimento de soluções mais eficientes.

3 Quando não usar recursão

3.1 Chamada recursiva no início ou no fim da rotina

Cada chamada recursiva aloca memória para as variáveis locais e para os parâmetros (além da memória de controle). Assim, as funções recursivas para implementar os cálculos, a seguir, chegam a gastar espaço em memória proporcional ao número de chamadas recursivas realizadas. Já as funções iterativas para resolver estes mesmos cálculos gastam uma quantidade pequena e constante de memória local.

Exemplos:

- Cálculo do $\sum_{k=m}^{n}$ (Aula 24: implementações das definições 1 e 2)
- Cálculo de $n!$ (Aula 24)
- Cálculo de x^n (Aula 24)

Outro Exemplo: Busca Binária (ver notas de aula do **prof. Flávio Keidi Miyazawa**).

(**Recomendação:** Ilustrar chamadas recursivas com pilhas ou árvores de recursão)

3.2 Repetição de processamento

Em geral, cada chamada recursiva é independente uma da outra. Caso ocorram os mesmos cálculos para duas chamadas recursivas independentes, esses cálculos serão repetidos para cada chamada.

Se o número de cálculos repetidos for muito grande, a execução do programa pode ficar inviável.

Um exemplo é a implementação da função de Fibonacci, com base em sua definição recursiva:

$$Fibonacci(n) = \begin{cases} 0 & \text{se } n = 0, \\ 1 & \text{se } n = 1 \text{ e} \\ Fibonacci(n-1) + Fibonacci(n-2) & \text{se } n > 1. \end{cases}$$

Codificação recursiva da função de Fibonacci na linguagem C:

```
int Fibonacci (int n) {
   if (n == 0)
      return (0);
   else {
      if (n == 1)
         return (1);
      else
         return (Fibonacci (n-1) + Fibonacci (n-2));
   }
}
```

Codificação iterativa da função de Fibonacci na linguagem C:

```
int Fibonacci (int n) {
   int f0 = 0, fn = 1, i, temp;
   if (n == 0)
      return (f0);
   else {
      if (n == 1)
         return (fn);
      else {
         for (i = 2; i <= n; i++) {
            temp = fn;
            fn = fn + f0;
            f0 = temp;
         }
         return (fn);
      }
   }
}
```

Enquanto a versão recursiva causa um número exponencial de cálculos relativo ao parâmetro da função, a iterativa pode ser executada em tempo proporcional a esse mesmo parâmetro.

(**Recomendação:** Ilustrar chamadas recursivas com pilhas ou árvores de recursão)

4 Algoritmos Recursivos

4.1 Algoritmos de Ordenação

Quick Sort e Merge Sort estão entre os mais rápidos algoritmos que utilizam apenas comparações entre elementos para resolver o problema de ordenação. O algoritmo Quick Sort é aquele que apresenta o melhor tempo médio para ordenar seqüências em geral.

Tanto o Quick Sort quanto o Merge Sort são implementados recursivamente, utilizando a abordagem **divisão e consquista** (ou **indução forte**).

4.1.1 Quick Sort

Este algoritmo particiona a seqüência a ser ordenada em duas partes, de tal forma que todos os elementos da primeira parte são menores ou iguais aos da segunda. A seqüência é ordenada repetindo-se este processo de forma recursiva para cada parte.

```
void QuickSort (int *v, int inicio, int fim) {
   int p;

   if (inicio < fim) {
      p = Particiona (v, inicio, fim);
      QuickSort (v, inicio, p);
      QuickSort (v, p+1, fim);
   }
}
```

(**Recomendação:** Ilustrar chamadas recursivas com pilhas ou árvores de recursão)

4.1.2 Merge Sort

Este algoritmo subdivide a seqüencia a ser ordenada em duas partes, ordena cada parte de forma recursiva, e depois intercala as partes ordenadas.

```
int MergeSort (int *v, int inicio, int fim) {
   int meio;

   if (inicio < fim) {
      meio = (inicio + fim)/2;
      MergeSort (v, inicio, meio);
      MergeSort (v, meio+1, fim);
      Intercala (v, inicio, meio, fim);
   }
}
```

(**Recomendação:** Ilustrar chamadas recursivas com pilhas ou árvores de recursão)

Uma desvantagem deste algoritmo é a necessidade de memória auxiliar, do mesmo tamanho da seqüência a ser ordenada, na função de intercalação.

4.2 Torres de Hanoi

As Torres de Hanoi constituem um quebra-cabeça bastante antigo, formado de um conjunto de **n** discos de tamanhos diferentes e três pinos verticais, nos quais os discos podem ser encaixados.

A configuracão inicial consiste de todos os discos no pino 1 (um). O objetivo é mover todos os discos para o pino 3 (três), podendo utilizar o pino 2 (dois) como pino auxiliar.

As regras para resolver o problema são:

- Somente um disco pode ser movido de cada vez.
- Nenhum disco pode ser colocado sobre um disco menor que ele.
- Observando-se a regra 2, qualquer disco pode ser movido para qualquer pino.

Este problema pode ser resolvido de forma recursiva (ver notas de aula do **prof. Flávio Keidi Miyazawa**).

5 Referências

Rubio, J. A. "Recursividad en un Primer Curso de Computación", Departamento de Ciencias de la Computación, Universidad de Chile, Santiago, Chile.

Notas de Aula do **prof. Alexandre Falcão**, [online]: http://www.dcc.unicamp.br/ afalcao/mc102/

Notas de Aula do **prof. Flávio Keidi Miyazawa**.

Notas de Aula da **profa. M. Cecília F. Rubira**.