Universidade Federal de Viçosa
Campus de Florestal

Algoritmos e Estruturas de Dados I (CCF 211)

Recursividade
(Cap 01 – Seção 1.3 Ziviani)

Profa.Thais R. M. Braga Silva
<thais.braga@ufv.br>

Conceito de Recursividade

- Fundamental em Matemática e Ciência da Computação
  - Um programa recursivo é um programa que chama a si mesmo
  - Uma função recursiva é definida em termos dela mesma
- Exemplos
  - Números naturais, Função fatorial
- Conceito poderoso
  - Define conjuntos infinitos com *comandos* finitos

Recursividade

- A recursividade é uma estratégia que pode ser utilizada sempre que o cálculo de uma função para o valor n, pode ser descrita a partir do cálculo desta mesma função para o termo anterior (n-1).

Exemplo – Função fatorial:

n! = n * (n-1) * (n-2) * (n-3) *....* 1

(n-1)! = (n-1) * (n-2) * (n-3) *....* 1

logo:

n! = n * (n-1)!

Função Recursiva

- Definição: dentro do corpo de uma função, chamar novamente a própria função
  - recursão direta: a função A chama a própria função A
  - recursão indireta: a função A chama uma função B que, por sua vez, chama A

Condição de parada

- Nenhum programa nem função pode ser exclusivamente definido por si
  - Um programa seria um loop infinito
  - Uma função teria definição circular
- Condição de parada
  - Permite que o procedimento pare de se executar

Recursividade – Uso de Memória

- Para cada chamada de uma função, recursiva ou não, os parâmetros e as variáveis locais são empilhados na pilha de execução.

Memória de Processo

- Internamente, quando qualquer chamada de função é feita dentro de um programa, é criado um **Registro de Ativação** na **Pilha de Execução** do programa
- O registro de ativação armazena os parâmetros e variáveis locais da função bem como o “ponto de retorno” no programa ou subprograma que chamou essa função.
- Ao final da execução dessa função, o registro é desempilhado e a execução volta ao subprograma que chamou a função

Exemplo – Fatorial Recursivo

```
int fat1(int n) {
 int r;
 if (n<=1)
  r = 1;
 else
  r = n*fat1(n-1);
 return r;
}
void main() {
 int f;
 f = fatX(4);
 printf(“%d”,f);
}
```

```
int fat2(int n) {
 if (n<=1)
  return 1;
 else
  return n * fat2(n-1);
}
```

Complexidade

- A complexidade de tempo do fatorial recursivo é O(n). (Em breve iremos ver a maneira de calcular isso usando **equações de recorrência**)
- Mas a complexidade de espaço também é O(n), devido a pilha de execução
- Já no fatorial não recursivo a complexidade de espaço é O(1)

```
Fat (int n) {
   int f;
   f = 1;
   while(n > 0){
     f = f * n;
     n = n – 1;
   }
   return f;
}
```

Recursividade

- Portanto, a recursividade nem sempre é a melhor solução, mesmo quando a definição matemática do problema é feita em termos recursivos

Quando vale a pena usar recursividade

- Recursividade vale a pena para Algoritmos complexos, cuja a implementação iterativa é complexa e normalmente requer o uso explícito de uma pilha
  - Dividir para Conquistar (Ex. Quicksort)
  - Caminhamento em Árvores (pesquisa, backtracking)

Análise de Complexidade O
(Comportamento Assintótico de Limite Superior)

- Define-se uma função de complexidade f(n) desconhecida
  - n mede o tamanho dos argumentos para o procedimento em questão
- Identifica-se a **equação de recorrência** T(n)**:**
  - Especifica-se T(n) como uma função dos termos anteriores
  - Especifica-se a condição de parada (e.g. T(1))

Análise da Função Fatorial

- Qual a equação de recorrência que descreve a complexidade da função fatorial?

$$\begin{cases} T(n) = 1 + T(n-1) \\ T(1) = 1 \end{cases}$$

```
int fat2(int n) {
 if (n<=1)
  return 1;
 else
  return n * fat2(n-1);
}
```

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

$T(n-1) = 1 + T(n-2)$

$T(n-2) = 1 + T(n-3)$

...

$T(2) = 1 + T(1)$

Análise de Funções Recursivas

- Além da análise de custo do tempo, deve-se analisar também o custo de espaço

- Qual a complexidade de espaço da função fatorial (qual o tamanho da pilha de execução)?

Outro Exemplo

- Considere a seguinte função:

```
Pesquisa(n)
{
(1)    if (n <= 1)
(2)        ‘ inspecione elemento ’ e termine;
       else
       {
(3)        para cada um dos n elementos ‘ inspecione elemento’;
(4)        Pesquisa(n/3) ;
       }
}
```

Outro Exemplo

- Qual a equação de recorrência?

  $T(n) = n + T(n/3)$

  $T(1) = 1$

- Resolva a equação de recorrência
  - Dicas:
    - Pode fazer a simplificação de que n será sempre divisível por 3
    - Somatório de uma PG finita: $a_1(1 - q^n)/(1-q)$

Resolvendo a Equação

- Substitui-se os termos $T(k)$, $k < n$, até que todos os termos $T(k)$, $k > 1$, tenham sido substituídos por fórmulas contendo apenas $T(1)$.

$$
\begin{aligned}
T(n) &= n + T(n/3) \\
T(n/3) &= n/3 + T(n/3/3) \\
T(n/3/3) &= n/3/3 + T(n/3/3/3) \\
&\vdots \qquad \vdots \\
T(n/3/3\cdots/3) &= n/3/3\cdots/3 + \overbrace{T(n/3\cdots/3)}^{1 \,\rightarrow\, n/3^K = 1 \,\rightarrow\, n = 3^K}
\end{aligned}
$$

Resolvendo a Equação

Considerando que $T(n/3^K) = T(1)$ temos:

$$T(n) = \sum_{i=0}^{K-1} (n/3^i) + T(1) = n \sum_{i=0}^{K-1} (1/3^i) + 1$$

Aplicando a fórmula do somatório de uma PG finita $(a_1 – q^n)/(1-q)$, temos:

$$n\,(1 – (1/3)^K)/(1 - 1/3) + 1$$
$$n\,(1 – (1/3^K))/(1 - 1/3) + 1$$
$$n\,(1 – (1/n))/(1 - 1/3) + 1$$
$$(n – 1)/(2/3) + 1$$
$$3n/2 – ½$$

O(n)

Exercício

- Crie uma função recursiva que calcula a potência de um número:
  - Como escrever a função para o termo n em função do termo anterior?
  - Qual a condição de parada?
- Qual a complexidade desta função?

Função de Potência Recursiva

```
int pot(int base, int exp)
{
  if (exp == 0)
    return 1;
  /* else */
  return (base*pot(base, exp-1));
}

Análise de complexidade:
  T(b,0) = 1;
  T(b,n) = 1 + T(b,n-1);
```

O(n)

Exercícios

- Implemente uma função recursiva para computar o valor de 2^n

- O que faz a função abaixo?

```
void f(int a, int b) { // considere a > b
    if (b == 0)
        return a;
    else
        return f(b, a % b);
}
```

Respostas

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

- Algoritmo de Euclides. Calcula o MDC (máximo divisor comum) de dois números a e b
  ```