Recursividade

Liana Duenha

Faculdade de Computação
Universidade Federal de Mato Grosso do Sul

Algoritmos e Programação II

Segundo Semestre de 2015
Ciência da Computação e Engenharia da Computação

Conteúdo da aula:

- Definições
- Recorrências
- Exercícios

Runção Recursiva

Uma função recursiva é aquela que é definida em relação a si mesma.

Exemplo:

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

Função Recursiva em C

Em C, uma função recursiva é aquela que, direta ou indiretamente, chama a si mesma.

Exemplo:

```
int fatorial(int n)
{
   // se a condição de parada é satisfeita,
   // a recursão termina
1. if (n<=1) return 1;
   // caso contrário, há uma chamada recursiva
2. return n*fatorial(n-1);
}
```

A(s) chamada(s) externa(s) é (são) realizada(s) fora da função recursiva.

```
printf ("O fatorial de %d é %d:", n, fatorial(n));
```

Vantagens e Desvantagens no uso de Recursividade

- **vantagens:** descrição mais clara e concisa, quando o problema tem natureza recursiva ou usa estruturas de dados recursivas.
- **desvantagens:** depuração, gasto de memória (pilha).
- Uma função recursiva tem: uma ou mais chamadas externas, uma ou mais chamadas recursivas, uma ou mais condições de parada.
- Erro comum: esquecer ou errar a implementação da condição de parada. O que ocorre nesse caso?

Recursividade e o Uso de Pilha

- A cada chamada de uma função, recursiva ou não, os parâmetros, as variáveis locais e o endereço de retorno são armazenados na pilha de execução (dizemos que são "empilhados").
- Se a recursão nao pára, a pilha fica cheia e o programa é finalizado por falta de espaço.
- Na grande maioria dos casos, um erro de estouro de pilha decorre de algum erro de lógica de programação.

Retirando a Recursão

- A cada função recursiva corresponde uma não-recursiva que realiza a mesma computação.
- **Exercício:** Realize a mesma computação de `fatorial` usando um algoritmo não recursivo.

Segundo Exemplo: Fibonnacci

Como segundo exemplo, segue a definição da sequência de Fibonacci usando uma equação de recorrência:

- $f_0 = 0$,
- $f_1 = 1$
- $f_n = f_{n-1} + f_{n-2}$

Segundo Exemplo: Fibonnacci

Uma função recursiva para determinar o *n*-ésimo número da sequência de Fibonacci pode ser projetada diretamente a partir da equação de recorrência, como abaixo:

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

Desempenho

- Embora a natureza deste problema seja nitidamente recursiva, a solução recursiva apresentada não apresenta um bom desempenho na prática.
- Por que?
- Enquanto um algoritmo iterativo para resolver o mesmo problema devolve a resposta para $n = 100$ em poucos milissegundos, o método recursivo demoraria 10^9 anos para encontrar o mesmo valor.

Recorrências

- Uma recorrência é uma equação ou desigualdade que descreve uma função em termos de seu valor calculado sobre entradas menores.
- **Exemplo:** Seja $T(n)$ que expressa a quantidade de operações realizadas e consequentemente, a complexidade de tempo da função `fatorial` (abaixo).

```
int fatorial(int n)
{
1.      if (n<=1) return 1;
2.      return n*fatorial(n-1);
}
```

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

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

Resolvendo $T(n)$:

$$T(n) = 1 + T(n-1) = 1 + 1 + T(n-2) =$$
$$1 + 1 + 1 + T(n-3) = 1 + 1 + 1 + 1 + T(n-4) = \ldots$$

Após um total de k etapas como estas, teremos $T(n) = \sum_k 1 + T(n-k)$, e quando $k = n - 1$ temos $T(n) = n - 1 + 1 = n$. Assim, a função de complexidade que expressa o número de operações realizadas pela função `fatorial` é $T(n) = n$ e podemos dizer que a função tem complexidade linear.