Universidade Federal de Ouro Preto
Campus João Monlevade

CSI 488 – Algoritmos e Estruturas de Dados I

Recursividade

Prof. Mateus Ferreira Satler

Índice

1. Introdução

- Uma das regras básicas para definir novos objetos ou conceitos é que a definição deve conter somente termos que tenham já sido definidos ou que sejam óbvios.

- Exemplo:
  - π é definido com sendo a razão entre a circunferência de um círculo e seu diâmetro.

- Isso equivale ao seguinte conjunto de instruções:
  1. Obter a circunferência de um círculo e seu diâmetro;
  2. Dividir o primeiro pelo último;
  3. Chamar o resultado de π

1. Introdução

- Por outro lado, existem muitos conceitos que se **autodefinem**.

- Tais definições são chamadas **definições recursivas** e são usadas primordialmente para se definir conjuntos infinitos.

- Exemplo:

  - Ancestralidade de uma pessoa:

    - Os pais de uma pessoa são seus antepassados;
    - Os pais de qualquer antepassado são também antepassados da pessoa em consideração.

1. Introdução

- A recursividade é 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.

- Exemplo:
  - Números Naturais $\mathbb{N}$
    - $0 \in \mathbb{N}$
    - Se $n \in \mathbb{N}$, então $(n+1) \in \mathbb{N}$
    - Não há outros objetos no conjunto $\mathbb{N}$

  - $\mathbb{N}$ consiste dos seguintes itens:
    - $0, 0+1, 0+1+1, 0+1+1+1, \ldots$

2. Indução

- Técnica de demonstração matemática na qual algum parâmetro da proposição a ser demonstrada envolve números naturais.

- Se desejamos provar uma proposição **T** como verdadeira para todos os números naturais, utilizamos a indução para não precisar provar **T** para cada um dos números naturais.

- A indução permite provar que proposição **T** é válida para todos os naturais de forma rápida.

2. Indução

- Para provar que uma proposição **T** é válida para todos os naturais através da por indução, basta executar os passos à seguir:
  - **Passo base**: Provar que T é válida para n = 1.
  - **Hipótese de indução**: Assumir que T é válida para n-1
  - **Passo indutivo**: Partindo-se de que T é válida para n-1, provar que T é valida para n.

2. Indução

- **Teorema:** A soma dos n primeiros números naturais é

$$S(n) = \frac{n * (n+1)}{2}$$

  - **Base:** Para n = 1, devemos mostrar que $S(1) = 1$
  - Temos que: $S(1) = \frac{n * (n+1)}{2} = \frac{1 * (1+1)}{2} = \frac{2}{2} = 1$
  - **Hip. de Indução:** Assume-se

$$S(n-1) = \frac{(n-1) * ((n-1)+1)}{2} = \frac{(n-1) * n}{2}$$

2. Indução

- **Passo indutivo:** Deve-se mostrar que

$$S(n) = \frac{n * (n+1)}{2}$$

- Temos que: $S(n) = S(n-1) + n = \dfrac{(n-1) * n}{2} + n$

$$S(n) = \frac{(n-1) * n}{2} + \frac{2n}{2} = \frac{n^2 - n + 2n}{2}$$

$$S(n) = \frac{n^2 + n}{2} = \frac{n * (n+1)}{2}$$

cqd (como queríamos demostrar)

2. Indução

- Por que a indução funciona?
  - Constrói-se a prova da proposição T para n = 1.
  - O passo de indução é uma fórmula genérica para se provar a proposição T para n a partir da prova para n-1.
  - A partir da prova de T para n = 1, utiliza-se o passo de indução para se obter a prova de T para n = 2.
  - Novamente, utiliza-se do passo de indução para construir a prova de T para n = 3, a partir de prova de T para n = 2.
  - Aplica-se o passo de indução até se obter a prova de T para n.

3. Recursão

- A definição recursiva de uma função funciona como o princípio matemático da indução.

- A ideia consiste em:
  - Definir a resposta da função para um caso base.
  - Definir como construir a resposta para um caso geral (n) com base em respostas de casos menores (n-1)

- A recursão também pode ser vista como a definição matemática de uma função por casos.

3. Recursão

Função fatorial

- 1! = 1
- 2! = 1 * 2 <==> 1! * 2 = 2
- 3! = (1 * 2) * 3 <==> 2! * 3 = 6
- 4! = (1 * 2 * 3) * 4 <==> 3! * 4 = 24
- 5! = (1 * 2 * 3 * 4) * 5 <==> 4! * 5 = 120
- ...
- (n-1)! = (1 * 2 * 3 * ... * n-2) * n-1 <==> (n-2)! * (n-1)
- n! = (1 * 2 * 3 * ... * n-1) * n <==> (n-1)! * n

3. Recursão

- A definição matemática da função fatorial é:

$$n! = \begin{cases} 1 & \text{se n} = 1 \\ n * (n-1)! & \text{se n} > 1 \end{cases}$$

3. Recursão

- **Função fatorial:** definição indutiva
  - Qual é o caso base e o passo da indução para a função fatorial?
  - **Base:** Se n é igual a 1, o fatorial de n é 1, ou seja, 1! == 1.
  - **Hipótese:** Assume-se que se sabe calcular o fatorial de n-1, ou seja, (n-1) é conhecido!
  - **Passo indutivo:** Expressa-se como calcular o fatorial de n utilizando o fatorial de n-1, que é hipoteticamente conhecido.
    - Isto é feito da seguinte forma: n! = n * (n-1)!

3. Recursão

- Codificação recursiva da função fatorial
  - **Base**: Caso o código receba 1 como parâmetro, deverá retornar 1, que é o valor de 1!.
  - **Hipótese**: Assume-se que se conhece como calcular fat (n-1).
  - **Passo indutivo**: Codifica-se o caso genérico utilizando-se a chamada fat(n-1) para compor a resposta

3. Recursão

- Codificação recursiva da função fatorial
  - **Base**: Caso o código receba 1 como parâmetro, deverá retornar 1, que é o valor de 1!.
  - **Hipótese**: Assume-se que se conhece como calcular fat (n-1).
  - **Passo indutivo**: Codifica-se o caso genérico utilizando-se a chamada fat(n-1) para compor a resposta

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

$$n! = \begin{cases} 1 & \text{se n} = 1 \\ n * (n-1)! & \text{se n} > 1 \end{cases}$$

3. Recursão

- Observações:

  - Para solucionar um problema, faz-se uma chamada para a própria função, com um parâmetro menor.

  - Por este motivo, a função que codifica um problema de forma indutiva é chamada função recursiva.

  - A recursividade geralmente permite uma descrição mais clara e curta de algoritmos, especialmente para problemas que são naturalmente recursivos.

3. Recursão

- Como o fatorial não está definido para zero e para números negativos, devemos considerar, do ponto de vista computacional, que para estes valores o resultado da função fatorial também cai no caso base.

- Ou seja, o código deve ser:

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

3. Recursão

- Memória:
  - Toda vez que uma função é chamada, suas variáveis locais são armazenadas no topo da pilha.
  - Quando uma função termina, suas variáveis locais são removidas da pilha.
  - A execução de uma função deixa no topo da pilha o resultado da função.
  - Cada chamada de uma função recursiva é uma nova chamada de função no topo da pilha.

3. Recursão

```
int f1(int a, int b){
     int c = 5;
     return (c + a + b);
}

int f2(int a, int b){
     int c;
     c = f1(b, a);
     return c;
}

int main(){
     int x= f2(2, 3);
}
```

Topo da Pilha

a = 3	b = 2	c = 5
a = 2	b = 3	c = f1(3,2)
x = f2(2, 3)		

3. Recursão

- Um programa iterativo raramente tem muitas funções que chamam funções.
- Um programa recursivo pode ter muitas chamadas recursivas de uma função.
- Estas chamadas recursivas podem facilmente utilizar muita memória devido às copias desnecessárias de variáveis locais.
- Se as chamadas recursivas criarem muitas cópias de variáveis, este programa será menos eficiente que sua versão iterativa.

3. Recursão

- O programa iterativo a seguir é mais eficiente que a versão recursiva, devido ao grande número de cópias locais.

```
int fat(int n){
    int fatorial = 1;
    for(int i = 1; i <= n; i++)
        fatorial = fatorial * i;
    return fatorial;
}
```

3. Recursão

- Criamos algoritmos recursivos:
  - Definindo o resultado de casos base
  - Assumindo a solução para casos menores
  - Construindo a solução do caso geral utilizando as soluções de casos menores

- Algoritmos recursivos são mais claros e concisos.

- Algoritmos recursivos podem rapidamente ocupar toda a memória.

4. Referências

- Material de aula do Prof. Ricardo Anido, da UNICAMP: http://www.ic.unicamp.br/~ranido/mc102/
- Material de aula da Profa. Virgínia F. Mota: https://sites.google.com/site/virginiaferm/home/disciplinas
- DEITEL, P; DEITEL, H. *C How to Program*. 6a Ed. Pearson, 2010.