Recursão

Disciplina de Programação de Computadores I
Universidade Federal de Ouro Preto

Agenda

- Indução
- Função recursiva através da definição indutiva
- Função recursiva através da definição matemática por casos
- Recursão e memória
- Exercícios

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.

Prova por 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**.

Exemplo de Indução (I)

- **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

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

Exemplo de Indução (II)

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

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

Temos que:

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

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

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

cqd (como queríamos demonstar)

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.

Recursão (I)

- 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.

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)!

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

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

Definição matemática da função fatorial

- A definição matemática da função fatorial é:

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

- Utilizamos estes casos ao codificar a função fatorial em C:

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

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.

Fatorial e números negativos

- 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:

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

Chamada de funções e a 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.

Memória com chamadas de função

```
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)		

Memória com chamadas recursivas de função

```
long fat(long n){
    long n_ant, fat_ant;
    if(n == 1){
        return 1;
    }else{
        n_ant = n-1;
        fat_ant = fat(n_ant);
        return (n * fat_ant);
    }
}
int main(){
  int resp = fat(4);
}
```

Topo da Pilha

n = 1		
n = 2	n_ant = 1	fat_ant = fat(1)
n = 3	n_ant = 2	fat_ant = fat(2)
n = 4	n_ant = 3	fat_ant = fat(3)
resp = fat(4)		

Recursão e uso de memória

- 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.

Recursão e uso de memória: Exemplo

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

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

```
long fat(long n){
    long n_ant, fat_ant;
    if(n == 1){
        return 1;
    }else{
        n_ant = n-1;
        fat_ant = fat(n_ant);
        return (n * fat_ant);
    }
}
```

Exemplo: Soma de elementos de um vetor

- Crie uma função para calcular a soma S dos elementos de um vetor v.
  - **Base:** Se o vetor v tem 1 elemento, então S(1) = v[1].
  - **Hipótese:** Assume-se que a soma dos n-1 primeiros elementos do vetor seja S(n-1).
  - **Passo indutivo:** A soma dos n elementos será o elemento v[n] mais a soma dos n-1 primeiros elementos, ou seja, S(n) = v[n] + S(n-1).

Soma de elementos: Correção de índices e código

- Como C conta elementos de vetor a partir de 0, vamos corrigir os índices anteriores:
  - **Base:** S(1) = v[0].
  - **Passo indutivo:** S(n) = v[n-1] + S(n-1)
- Geramos o código a seguir:

```
int soma(int v[], int n){
    if(n == 1)
        return v[0];
    else
        return v[n-1] + soma(v, n-1);
}
```

Soma de elementos: pilha de memória

```
int soma(int v[], int n){
    if(n == 1)
        return v[0];
    else
        return v[n-1] + soma(v, n-1);
}
main(){
  int x, v[]={1,2,3};
  x = soma(v,3);
  return 0;
  }
```

v = [1,2,3]	n = 1	v[0]
v = [1,2,3]	n = 2	v[1] + soma(v,1)
v = [1,2,3]	n = 3	v[2] + soma(v,2)
soma([1,2,3], 3)		

Exemplo: Série de Fibonacci

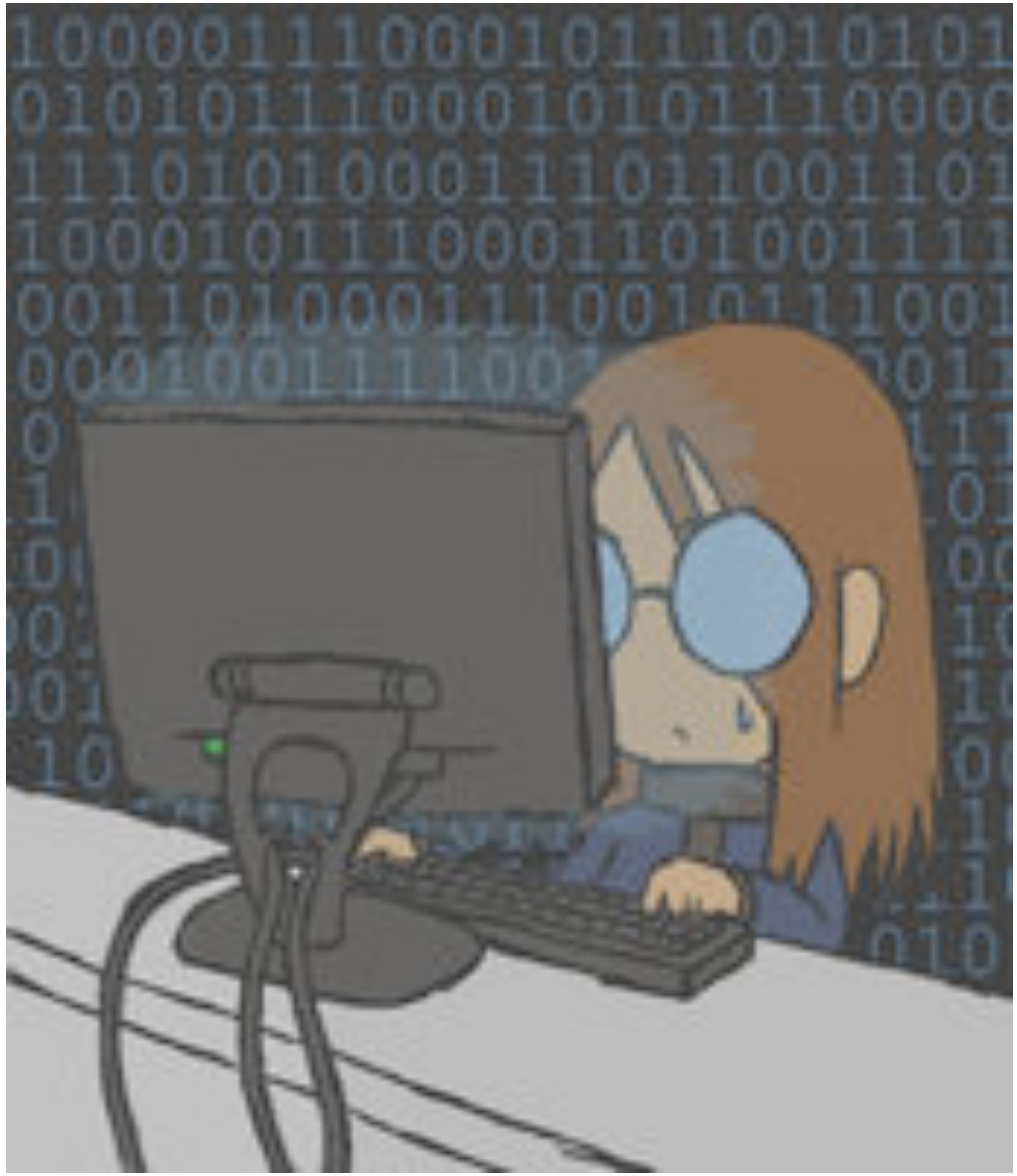
- Codifique um programa recursivo que calcule o n-ésimo elemento da série de fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, ...
  - **Base:** fib(1) == 0 e fib(2) == 1.
  - **Hipótese de indução:** Admite-se conhecer fib(n-1) e fib(n-2)
  - **Passo indutivo:** fib(n) = fib(n-1) + fib(n-2)

Exemplo: Código recursivo para série de Fibonacci

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

Resumo de 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.



Exercícios

O que será impresso?

```
void imprime(int v[], int i, int n){
   if(i==n){
        printf("%d, ", v[i]);
    }else{
        imprime(v, i+1, n);
        printf("%d, ", v[i]);
    }
}
int main(){
    int vet[] = {1,2,3};
    imprime(vet, 0, 2);
    printf("\n");
}
```

Série de Fibonacci: pilha de memória

- Mostre o estado da pilha de memória durante a execução da função fib para a chamada fib(3).

Referências Bibliográficas

- 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.