

Recursão

Aula 2

Diego Padilha Rubert

Faculdade de Computação
Universidade Federal de Mato Grosso do Sul

Algoritmos e Programação II

Conteúdo da aula

- 1 Motivação
- 2 Definição
- 3 Exemplos
- 4 Exercícios

- ▶ Conceito fundamental em computação
- ▶ Programas elegantes, mais curtos e poderosos
- ▶ Equivalência entre programas recursivos — não-recursivos (?)
- ▶ Memória

- ▶ Conceito fundamental em computação
- ▶ Programas elegantes, mais curtos e poderosos
- ▶ Equivalência entre programas recursivos — não-recursivos (?)
- ▶ Memória

- ▶ Conceito fundamental em computação
- ▶ Programas elegantes, mais curtos e poderosos
- ▶ Equivalência entre programas recursivos — não-recursivos (?)
- ▶ Memória

- ▶ Conceito fundamental em computação
- ▶ Programas elegantes, mais curtos e poderosos
- ▶ Equivalência entre programas recursivos — não-recursivos (?)
- ▶ Memória

Definição

- ▶ Alguns problemas têm uma **estrutura recursiva**: cada entrada do problema contém uma entrada menor do mesmo problema
- ▶ Estratégia:
 - se a entrada do problema é pequena então resolva-a diretamente;
 - senão,
 - reduza-a a uma entrada menor do mesmo problema,
 - aplique este método à entrada menor
 - e volte à entrada original.
- ▶ **Algoritmo recursivo, programa recursivo, função recursiva**
- ▶ Uma **função recursiva** é aquela que possui uma ou mais chamadas a si mesma (**chamada recursiva**)
- ▶ Toda função deve possuir ao menos uma chamada externa a ela. Se todas as chamadas à função são externas, então a função é dita **não-recursiva**

Definição

- ▶ Alguns problemas têm uma **estrutura recursiva**: cada entrada do problema contém uma entrada menor do mesmo problema
- ▶ Estratégia:
 - se a entrada do problema é pequena então resolva-a diretamente;
 - senão,
 - reduza-a a uma entrada menor do mesmo problema,
 - aplique este método à entrada menor
 - e volte à entrada original.
- ▶ Algoritmo recursivo, programa recursivo, função recursiva
- ▶ Uma **função recursiva** é aquela que possui uma ou mais chamadas a si mesma (**chamada recursiva**)
- ▶ Toda função deve possuir ao menos uma chamada externa a ela. Se todas as chamadas à função são externas, então a função é dita **não-recursiva**

Definição

- ▶ Alguns problemas têm uma **estrutura recursiva**: cada entrada do problema contém uma entrada menor do mesmo problema
- ▶ Estratégia:
 - se a entrada do problema é pequena então resolva-a diretamente;
 - senão,
 - reduza-a a uma entrada menor do mesmo problema,
 - aplique este método à entrada menor
 - e volte à entrada original.
- ▶ **Algoritmo recursivo, programa recursivo, função recursiva**
- ▶ Uma **função recursiva** é aquela que possui uma ou mais chamadas a si mesma (**chamada recursiva**)
- ▶ Toda função deve possuir ao menos uma chamada externa a ela. Se todas as chamadas à função são externas, então a função é dita **não-recursiva**

Definição

- ▶ Alguns problemas têm uma **estrutura recursiva**: cada entrada do problema contém uma entrada menor do mesmo problema
- ▶ Estratégia:
 - se a entrada do problema é pequena então resolva-a diretamente;
 - senão,
 - reduza-a a uma entrada menor do mesmo problema,
 - aplique este método à entrada menor
 - e volte à entrada original.
- ▶ **Algoritmo recursivo, programa recursivo, função recursiva**
- ▶ Uma **função recursiva** é aquela que possui uma ou mais chamadas a si mesma (**chamada recursiva**)
- ▶ Toda função deve possuir ao menos uma chamada externa a ela. Se todas as chamadas à função são externas, então a função é dita **não-recursiva**

Definição

- ▶ Alguns problemas têm uma **estrutura recursiva**: cada entrada do problema contém uma entrada menor do mesmo problema
- ▶ Estratégia:
 - se a entrada do problema é pequena então resolva-a diretamente;
 - senão,
 - reduza-a a uma entrada menor do mesmo problema,
 - aplique este método à entrada menor
 - e volte à entrada original.
- ▶ **Algoritmo recursivo, programa recursivo, função recursiva**
- ▶ Uma **função recursiva** é aquela que possui uma ou mais chamadas a si mesma (**chamada recursiva**)
- ▶ Toda função deve possuir ao menos uma chamada externa a ela. Se todas as chamadas à função são externas, então a função é dita **não-recursiva**

- ▶ Em geral, a toda função recursiva corresponde uma outra não-recursiva equivalente
- ▶ Correção de um algoritmo recursivo pode ser facilmente demonstrada usando indução matemática
- ▶ A implementação de uma função recursiva pode acarretar gasto maior de memória, já que durante o processo de execução da função muitas informações devem ser guardadas na pilha de execução

- ▶ Em geral, a toda função recursiva corresponde uma outra não-recursiva equivalente
- ▶ Correção de um algoritmo recursivo pode ser facilmente demonstrada usando indução matemática
- ▶ A implementação de uma função recursiva pode acarretar gasto maior de memória, já que durante o processo de execução da função muitas informações devem ser guardadas na pilha de execução

- ▶ Em geral, a toda função recursiva corresponde uma outra não-recursiva equivalente
- ▶ Correção de um algoritmo recursivo pode ser facilmente demonstrada usando indução matemática
- ▶ A implementação de uma função recursiva pode acarretar gasto maior de memória, já que durante o processo de execução da função muitas informações devem ser guardadas na pilha de execução

Problema

Dado um número inteiro $n \geq 0$, computar o fatorial $n!$.

Usamos uma fórmula que nos permite naturalmente escrever uma função recursiva para calcular $n!$:

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

Problema

Dado um número inteiro $n \geq 0$, computar o fatorial $n!$.

Usamos uma fórmula que nos permite naturalmente escrever uma função recursiva para calcular $n!$:

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

Exemplos

Uma solução.

```
/* Recebe um número inteiro n >= 0 e devolve o fatorial de n */  
int fat(int n)  
{  
    int result;  
  
    if (n <= 1)  
        result = 1;  
    else  
        result = n * fat(n-1);  
  
    return result;  
}
```

Outra solução.

```
/* Recebe um número inteiro n >= 0 e devolve o fatorial de n */  
int fat(int n)  
{  
    if (n <= 1)  
        return 1;  
    else  
        return n * fat(n-1);  
}
```

Exemplos

fat (3)

┌

fat (2)

┌

fat (1)

┌

devolve 1

└

devolve $2 \times 1 = 2 \times$ fat (1)

└

devolve $3 \times 2 = 3 \times$ fat (2)

└

Exemplos

fat (3)

┌

fat (2)

┌

fat (1)

┌

devolve 1

└

devolve $2 \times 1 = 2 \times$ **fat (1)**

└

devolve $3 \times 2 = 3 \times$ **fat (2)**

└

Exemplos

fat (3)

┌

fat (2)

┌

fat (1)

┌

devolve 1

└

devolve $2 \times 1 = 2 \times$ **fat (1)**

└

devolve $3 \times 2 = 3 \times$ **fat (2)**

└

Exemplos

fat (3)

┌

fat (2)

┌

fat (1)

┌

devolve 1

└

devolve $2 \times 1 = 2 \times$ **fat (1)**

└

devolve $3 \times 2 = 3 \times$ **fat (2)**

└

Exemplos

fat (3)

┌

fat (2)

┌

fat (1)

┌

devolve 1

└

devolve $2 \times 1 = 2 \times$ **fat (1)**

└

devolve $3 \times 2 = 3 \times$ **fat (2)**

└

Exemplos

fat (3)

┌

fat (2)

┌

fat (1)

┌

devolve 1

└

devolve $2 \times 1 = 2 \times$ **fat (1)**

└

devolve $3 \times 2 = 3 \times$ **fat (2)**

└

Problema

Dado um número inteiro $n > 0$ e uma sequência de n números inteiros armazenados em um vetor v , determinar um valor máximo em v .

Exemplos

```
/* Recebe um número inteiro n > 0 e um vetor v de números in-
   teiros com n elementos e devolve um elemento máximo de v */
int maximo(int n, int v[MAX])
{
    int aux;

    if (n == 1)
        return v[0];
    else {
        aux = maximo(n-1, v);
        if (aux > v[n-1])
            return aux;
        else
            return v[n-1];
    }
}
```

Como verificar que uma função recursiva está correta?

Passo 1: escreva *o que* a função deve fazer;

Passo 2: verifique se a função de fato faz o que deveria fazer quando a entrada é pequena;

Passo 3: imagine que a entrada é grande e suponha que a função fará a coisa certa para entradas menores; sob essa hipótese, verifique que a função faz o que dela se espera.

Como verificar que uma função recursiva está correta?

Passo 1: escreva *o que* a função deve fazer;

Passo 2: verifique se a função de fato faz o que deveria fazer quando a entrada é pequena;

Passo 3: imagine que a entrada é grande e suponha que a função fará a coisa certa para entradas menores; sob essa hipótese, verifique que a função faz o que dela se espera.

Como verificar que uma função recursiva está correta?

Passo 1: escreva *o que* a função deve fazer;

Passo 2: verifique se a função de fato faz o que deveria fazer quando a entrada é pequena;

Passo 3: imagine que a entrada é grande e suponha que a função fará a coisa certa para entradas menores; sob essa hipótese, verifique que a função faz o que dela se espera.

Como verificar que uma função recursiva está correta?

Passo 1: escreva *o que* a função deve fazer;

Passo 2: verifique se a função de fato faz o que deveria fazer quando a entrada é pequena;

Passo 3: imagine que a entrada é grande e suponha que a função fará a coisa certa para entradas menores; sob essa hipótese, verifique que a função faz o que dela se espera.

Proposição

A função **maximo** encontra um maior elemento em um vetor v com $n \geq 1$ números inteiros.

Prova

Indução na quantidade n de elementos do vetor v .

Se $n = 1$ é fácil.

Suponha que para qualquer valor inteiro positivo $m < n$ a função compute corretamente `maximo(m, v)`.

Suponha agora que temos um vetor v contendo $n > 1$ números inteiros.

Chamada externa `maximo(n, v)`, com $n > 1$. A função executa: `aux = maximo(n-1, v);`

Por hipótese de indução, `aux` contém um valor máximo para os $n - 1$ primeiros valores do vetor v . Então, a função decide quem é maior: `aux` ou `v[n-1]`.

Prova

Indução na quantidade n de elementos do vetor v .

Se $n = 1$ é fácil.

Suponha que para qualquer valor inteiro positivo $m < n$ a função compute corretamente `maximo(m, v)`.

Suponha agora que temos um vetor v contendo $n > 1$ números inteiros.

Chamada externa `maximo(n, v)`, com $n > 1$. A função executa: `aux = maximo(n-1, v);`

Por hipótese de indução, `aux` contém um valor máximo para os $n - 1$ primeiros valores do vetor v . Então, a função decide quem é maior: `aux` ou `v[n-1]`.

Prova

Indução na quantidade n de elementos do vetor v .

Se $n = 1$ é fácil.

Suponha que para qualquer valor inteiro positivo $m < n$ a função compute corretamente **maximo(m, v)**.

Suponha agora que temos um vetor v contendo $n > 1$ números inteiros.

Chamada externa **maximo(n, v)**, com $n > 1$. A função executa: `aux = maximo(n-1, v);`

Por hipótese de indução, **aux** contém um valor máximo para os $n - 1$ primeiros valores do vetor v . Então, a função decide quem é maior: **aux** ou **v[n-1]**.

Prova

Indução na quantidade n de elementos do vetor v .

Se $n = 1$ é fácil.

Suponha que para qualquer valor inteiro positivo $m < n$ a função compute corretamente `maximo(m, v)`.

Suponha agora que temos um vetor v contendo $n > 1$ números inteiros.

Chamada externa `maximo(n, v)`, com $n > 1$. A função executa: `aux = maximo(n-1, v);`

Por hipótese de indução, `aux` contém um valor máximo para os $n - 1$ primeiros valores do vetor v . Então, a função decide quem é maior: `aux` ou `v[n-1]`.

Exercícios

1. A n -ésima potência de um número x , denotada por x^n , pode ser computada recursivamente observando a seguinte fórmula:

$$x^n = \begin{cases} 1, & \text{se } n = 0, \\ x \cdot x^{n-1}, & \text{se } n > 1. \end{cases}$$

Considere neste exercício que x e n são números inteiros.

- (a) Escreva uma função não-recursiva com a seguinte interface:

```
int pot(int x, int n)
```

que receba dois números inteiros x e n e calcule e devolva x^n .

- (b) Escreva uma função recursiva com a seguinte interface:

```
int potR(int x, int n)
```

que receba dois números inteiros x e n e calcule e devolva x^n .

- (c) Escreva um programa que receba dois números inteiros x e n , com $n \geq 0$, e devolva x^n . Use as funções em (a) e (b) para mostrar os dois resultados.

Exercícios

```
#include <stdio.h>

/* Recebe um dois números inteiros x e n
   e devolve x a n-ésima potência */
int pot(int x, int n)
{
    int i, result;

    result = 1;
    for (i = 1; i <= n; i++)
        result = result * x;
    return result;
}

/* Recebe um dois números inteiros x e n
   e devolve x a n-ésima potência */
int potR(int x, int n)
{
    if (n == 0)
        return 1;
    else
        return x * potR(x, n-1);
}
```

Exercícios

```
/* Recebe dois números inteiros x e n e im-
   prime x a n-ésima potência chamando duas
   funções: uma não-recursiva e uma recursiva */
int main(void)
{
    int x, n;

    scanf("%d%d", &x, &n);
    printf("Não-resursiva: %d^%d = %d\n", x, n, pot(x, n));
    printf("Resursiva      : %d^%d = %d\n", x, n, potR(x, n));

    return 0;
}
```

2. O que faz a função abaixo?

```
void imprime_alguma_coisa(int n)
{
    if (n != 0) {
        imprime_alguma_coisa(n / 2);
        printf("%c", '0' + n % 2);
    }
}
```

Escreva um programa para testar a função `imprime_alguma_coisa`.

3. (a) Escreva uma função recursiva que receba dois números inteiros positivos e devolva o máximo divisor comum entre eles usando o algoritmo de Euclides.
- (b) Escreva um programa que receba dois números inteiros e calcule o máximo divisor comum entre eles. Use a função do item (a).

4. (a) Escreva uma função recursiva com a seguinte interface:

```
float soma(int n, float v[MAX])
```

que receba um número inteiro $n > 0$ e um vetor v de números com ponto flutuante com n elementos, e calcule e devolva a soma desses números.

- (b) Usando a função do item anterior, escreva um programa que receba um número inteiro n , com $n \geq 1$, e mais n números reais e calcule a soma desses números.

5. (a) Escreva uma função recursiva com a seguinte interface:

```
int soma_digitos(int n)
```

que receba um número inteiro positivo n e devolva a soma de seus dígitos.

- (b) Escreva um programa que receba um número inteiro n e imprima a soma de seus dígitos. Use a função do item (a).

6. A **sequência de Fibonacci** é uma sequência de números inteiros positivos dada pela seguinte fórmula:

$$\begin{cases} F_1 = 1, \\ F_2 = 1, \\ F_i = F_{i-1} + F_{i-2}, \quad \text{para } i \geq 3. \end{cases}$$

- (a) Escreva uma função recursiva com a seguinte interface:

```
int Fib(int i)
```

que receba um número inteiro positivo i e devolva o i -ésimo termo da sequência de Fibonacci, isto é, F_i .

- (b) Escreva um programa que receba um número inteiro $i \geq 1$ e imprima o termo F_i da sequência de Fibonacci. Use a função do item (a).

7. O **piso** de um número inteiro positivo x é o único inteiro i tal que $i \leq x < i + 1$. O piso de x é denotado por $\lfloor x \rfloor$.

Segue uma amostra de valores da função $\lfloor \log_2 n \rfloor$:

n	15	16	31	32	63	64	127	128	255	256
$\lfloor \log_2 n \rfloor$	3	4	4	5	5	6	6	7	7	8

- (a) Escreva uma função recursiva com a seguinte interface:

```
int piso_log2(int n)
```

que receba um número inteiro positivo n e devolva $\lfloor \log_2 n \rfloor$.

- (b) Escreva um programa que receba um número inteiro $n \geq 1$ e imprima $\lfloor \log_2 n \rfloor$. Use a função do item (a).

8. Considere o seguinte processo para gerar uma sequência de números. Comece com um inteiro n . Se n é par, divida por 2. Se n é ímpar, multiplique por 3 e some 1. Repita esse processo com o novo valor de n , terminando quando $n = 1$. Por exemplo, a sequência de números a seguir é gerada para $n = 22$:

22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

É conjecturado que esse processo termina com $n = 1$ para todo inteiro $n > 0$. Para uma entrada n , o **comprimento do ciclo de** n é o número de elementos gerados na sequência. No exemplo acima, o comprimento do ciclo de 22 é 16.

- (a) Escreva uma função não-recursiva com a seguinte interface:

```
int ciclo(int n)
```

que receba um número inteiro positivo n , mostre a sequência gerada pelo processo descrito acima na saída e devolva o comprimento do ciclo de n .

- (b) Escreva uma versão recursiva da função do item (a) com a seguinte interface:

```
int cicloR(int n)
```

que receba um número inteiro positivo n , mostre a sequência gerada pelo processo descrito acima na saída e devolva o comprimento do ciclo de n .

- (c) Escreva um programa que receba um número inteiro $n \geq 1$ e determine a sequência gerada por esse processo e também o comprimento do ciclo de n . Use as funções em (a) e (b) para testar.

9. Podemos calcular a potência x^n de uma maneira mais eficiente. Observe primeiro que se n é uma potência de 2 então x^n pode ser computada usando sequências de quadrados. Por exemplo, x^4 é o quadrado de x^2 e assim x^4 pode ser computado usando somente duas multiplicações ao invés de três. Esta técnica pode ser usada mesmo quando n não é uma potência de 2, usando a seguinte fórmula:

$$x^n = \begin{cases} 1, & \text{se } n = 0, \\ (x^{n/2})^2, & \text{se } n \text{ é par,} \\ x \cdot x^{n-1}, & \text{se } n \text{ é ímpar.} \end{cases} \quad (1)$$

- (a) Escreva uma função com interface:

```
int potencia(int x, int n)
```

que receba dois números inteiros x e n e calcule e devolva x^n usando a fórmula (1).

- (b) Escreva um programa que receba dois números inteiros a e b e imprima o valor de a^b .

