Programação de Computadores

Raoni F. S. Teixeira

Aulas 16 e 17 - Recursão

1 Introdução

Suponha que desejamos criar um algoritmo para resolver um determinado problema. Usando o método de **recursão/indução**, a solução pode ser expressa da seguinte forma:

1. Primeiramente, definimos a solução para **casos base** (instâncias simples do problema).
2. Em seguida, definimos como resolver o problema para um **caso geral**, utilizando soluções para instâncias menores do mesmo problema.

Este é um conceito fundamental em computação que permite resolver problemas complexos dividindo-os em subproblemas mais simples.

1.1 Indução Matemática

Indução é uma técnica de demonstração matemática para proposições que envolvem números naturais. Seja $T(n)$ uma proposição que desejamos provar como verdadeira para todos os valores naturais n.

Para provar $T(n)$ por indução, basta demonstrar:

1. **Caso Base:** Provar que T é válida para $n = 1$ (ou outro valor inicial apropriado).
2. **Hipótese de Indução:** Assumir que T é válida para $n - 1$.
3. **Passo de Indução:** Assumindo que T é válida para $n - 1$, provar que T é válida para n.

1.2 Por Que a Indução Funciona?

- Mostramos que T é válida para um caso simples ($n = 1$).
- Com o passo da indução, mostramos que T é válida para $n = 2$.
- Como T é válida para $n = 2$, pelo passo de indução, T também é válida para $n = 3$.
- E assim por diante, estabelecendo a validade para todos os naturais.

1.3 Exemplo: Soma dos Primeiros n Naturais

Teorema: A soma $S(n)$ dos primeiros n números naturais é $S(n) = \frac{n(n+1)}{2}$.
Prova:

- **Base:** Para $n = 1$, temos $S(1) = \frac{1(1+1)}{2} = 1$.
- **Hipótese de Indução:** Assumimos que a fórmula é válida para $n-1$, ou seja, $S(n-1) = \frac{(n-1)n}{2}$.
- **Passo:** Devemos mostrar que é válida para n. Por definição, $S(n) = S(n-1) + n$. Por hipótese:

$$\begin{aligned} S(n) &= S(n-1) + n \\ &= \frac{(n-1)n}{2} + n \\ &= \frac{(n-1)n + 2n}{2} \\ &= \frac{n^2 - n + 2n}{2} \\ &= \frac{n^2 + n}{2} \\ &= \frac{n(n+1)}{2} \end{aligned}$$

2 Recursão em Funções

Definições recursivas de funções operam como o princípio matemático da indução: a solução é inicialmente definida para o(s) caso(s) base e estendida para o caso geral.

2.1 Exemplo: Fatorial

Problema: Calcular o fatorial de um número inteiro não negativo n.

A definição matemática de fatorial já é recursiva:

- **Caso base:** $0! = 1$
- **Caso recursivo:** $n! = n \times (n-1)!$ para $n \geq 1$

Note como aplicamos o princípio da indução:

- Sabemos a solução para um caso base ($n = 0$).
- Definimos a solução do problema geral em termos do mesmo problema, mas para um caso mais simples.

2.2 Implementação em C

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

Para solucionar o problema, fizemos uma chamada para a própria função. Por isso, esta função é chamada de **recursiva**.

> **Recursividade:** Geralmente permite uma descrição mais clara e concisa dos algoritmos, especialmente quando o problema é recursivo por natureza.

Podemos simplificar a função eliminando a variável auxiliar:

```
int fatorial(int n) {
    if (n == 0)
        return 1;
    return n * fatorial(n - 1);
}
```

3 Gerenciamento de Memória

Para entender como a recursão funciona, precisamos compreender como o sistema gerencia a memória durante chamadas de funções.

3.1 Divisão da Memória

A memória de um sistema computacional é dividida em três partes principais:

- **Espaço Estático:** Contém o código do programa e variáveis globais.
- **Heap:** Usado para alocação dinâmica de memória (`malloc/free`).
- **Pilha (Stack):** Usada para execução de funções e variáveis locais.

3.2 A Pilha de Execução

- Toda vez que uma função é invocada, suas variáveis locais e parâmetros são armazenados no topo da pilha em uma estrutura chamada **frame** ou **registro de ativação**.
- Quando uma função termina sua execução, seu frame é removido da pilha.
- A pilha cresce e diminui conforme funções são chamadas e retornam.

3.3 Exemplo de Gerenciamento

Considere o seguinte código:

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

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

int main() {
    int x = f2(2, 3);
    return 0;
}
```

A Figura 1 ilustra o estado da pilha durante a execução.

3.4 Chamadas Recursivas

No caso de chamadas recursivas, cada chamada corresponde a um novo frame na pilha:

- As execuções das chamadas de funções recursivas são feitas na pilha, assim como qualquer função.

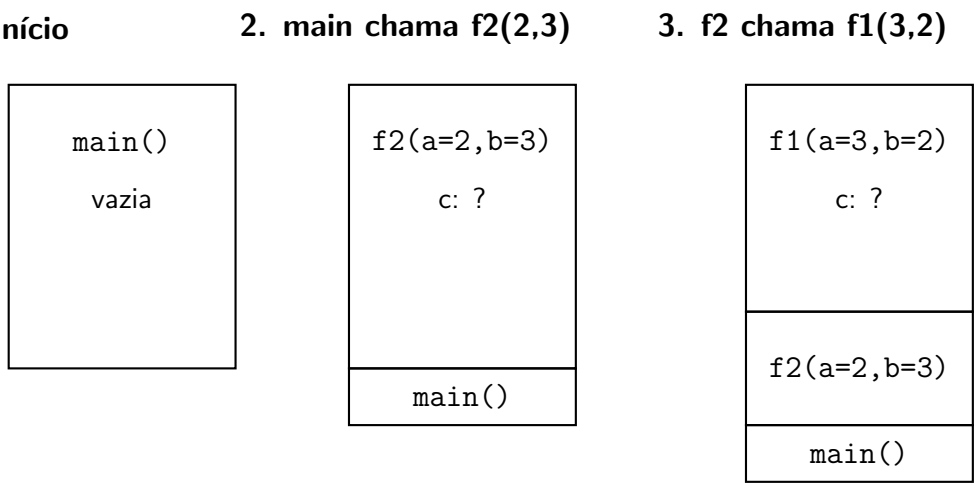

Figura 1: Estados da pilha durante a execução do programa.

- O último conjunto de variáveis alocadas na pilha (no topo) corresponde às variáveis da última chamada.
- Quando termina a execução de uma chamada, as variáveis locais são removidas da pilha.

3.5 Exemplo: fatorial(4)

Considere a chamada `fatorial(4)`. A Figura 2 mostra a evolução da pilha.

Cada chamada cria novas variáveis locais de mesmo nome (`n`). Em um dado instante, o nome `n` refere-se à variável local da função que está sendo executada naquele momento.

4 Recursão vs Iteração

4.1 Comparação

- **Clareza:** Soluções recursivas são geralmente mais concisas e próximas da definição matemática.
- **Memória:** Soluções iterativas geralmente consomem menos memória (não criam múltiplos frames).
- **Eficiência:** A cópia dos parâmetros a cada chamada recursiva gera custo adicional.

Pilha máxima:

`fatorial(n=0)` → retorna 1
`fatorial(n=1)` aguarda
`fatorial(n=2)` aguarda
`fatorial(n=3)` aguarda
`fatorial(n=4)` aguarda

↓

Retorna: 1 → 1 → 2 → 6 → 24

Figura 2: Pilha de execução para `fatorial(4)`.

4.2 Fatorial: Versão Iterativa

Para o fatorial, a solução iterativa é mais eficiente:

```
int fatorial(int n) {
    int fat = 1;
    for (int i = 2; i <= n; i++)
        fat = fat * i;
    return fat;
}
```

Esta versão usa memória constante e evita o overhead de múltiplas chamadas de função.

5 Exemplos Simples

5.1 Soma de Dois Números

Soma de dois números inteiros não negativos usando apenas incrementos e decrementos unitários:

```
int soma(int x, int y) {
    if (y == 0)
        return x;
    return soma(x + 1, y - 1);
}
```

Alternativamente:

```
int soma(int x, int y) {
    if (y == 0)
        return x;
    return soma(x, y - 1) + 1;
}
```

5.2 Multiplicação

Multiplicação de dois números inteiros positivos usando apenas somas:

```
int mult(int x, int y) {
    if (y == 1)
        return x;
    return mult(x, y - 1) + x;
}
```

Isto implementa: $x \times y = x \times (y - 1) + x$.

5.3 Potência

Cálculo de k^n onde n é um inteiro não negativo:

$$k^n = \begin{cases} 1 & \text{se } n = 0 \\ k \times k^{n-1} & \text{caso contrário} \end{cases}$$

```
int potencia(int k, int n) {
    if (n == 0)
        return 1;
    return k * potencia(k, n - 1);
}
```

5.4 Potência Eficiente

Podemos usar uma definição mais eficiente:

$$k^n = \begin{cases} 1 & \text{se } n = 0 \\ k^{n/2} \times k^{n/2} & \text{se } n > 0 \text{ e par} \\ k \times k^{\lfloor n/2 \rfloor} \times k^{\lfloor n/2 \rfloor} & \text{se } n > 0 \text{ e ímpar} \end{cases}$$

```
int potencia(int k, int n) {
    if (n == 0)
        return 1;
```

```
    int aux = potencia(k, n / 2);
    if (n % 2 == 0)
        return aux * aux;
    else
        return k * aux * aux;
}
```

Esta versão faz no máximo $\lfloor \log_2 n \rfloor + 2$ chamadas recursivas, comparado com n chamadas na versão anterior.

5.5 Máximo Divisor Comum

O algoritmo de Euclides:

$$\text{mdc}(x, y) = \begin{cases} x & \text{se } y = 0 \\ \text{mdc}(y, x \bmod y) & \text{se } y > 0 \end{cases}$$

Versão recursiva:

```
int mdc(int x, int y) {
    if (y == 0)
        return x;
    return mdc(y, x % y);
}
```

Versão iterativa:

```
int mdc(int x, int y) {
    int temp;
    while (y > 0) {
        temp = y;
        y = x % y;
        x = temp;
    }
    return x;
}
```

6 Exemplos com Vetores e Strings

6.1 Soma dos Elementos de um Vetor

Seja `v` um vetor de inteiros de tamanho `n`. Como calcular a soma de todos os elementos recursivamente?

Denotando por soma(v, n) a soma dos n primeiros elementos:

$$\text{soma}(v, n) = \begin{cases} 0 & \text{se } n = 0 \\ \text{soma}(v, n-1) + v[n-1] & \text{se } n > 0 \end{cases}$$

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

6.2 Maior Elemento de um Vetor

```
int max_vetor(int v[], int n) {
    if (n == 1)
        return v[0];

    int aux = max_vetor(v, n - 1);
    if (aux > v[n - 1])
        return aux;
    else
        return v[n - 1];
}
```

6.3 Palíndromo

Verificar se uma string é um palíndromo (lê-se igual de trás para frente):

```
#include <string.h>

int palindromo_rec(char s[], int inicio, int fim) {
    if (inicio >= fim)
        return 1;  // true

    if (s[inicio] != s[fim])
        return 0;  // false

    return palindromo_rec(s, inicio + 1, fim - 1);
}

int palindromo(char s[]) {
    return palindromo_rec(s, 0, strlen(s) - 1);
}
```

6.4 Inverter uma String

```
#include <string.h>

void inverter_rec(char s[], int inicio, int fim) {
    if (inicio >= fim)
        return;

    // Troca os caracteres
    char temp = s[inicio];
    s[inicio] = s[fim];
    s[fim] = temp;

    inverter_rec(s, inicio + 1, fim - 1);
}

void inverter(char s[]) {
    inverter_rec(s, 0, strlen(s) - 1);
}
```

7 Números de Fibonacci

A série de Fibonacci é: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Cada número é a soma dos dois anteriores:

$$\text{Fibonacci}(n) = \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2 \\ \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2) & \text{caso contrário} \end{cases}$$

7.1 Versão Recursiva

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

7.2 Problema de Eficiência

Esta implementação tem um problema sério de eficiência! Para calcular `fibonacci(n)`, a função realiza muitos cálculos repetidos:

- `fibonacci(5)`: 4 somas

- `fibonacci(10)`: 54 somas
- `fibonacci(20)`: 6.764 somas
- `fibonacci(50)`: 12.586.269.024 somas!

A função recursiva calcula `fibonacci(k)` múltiplas vezes para $k < n$. Por exemplo, para $n = 20$, `fibonacci(1)` é calculado 2584 vezes!

7.3 Versão Iterativa

A solução iterativa é muito mais eficiente:

```
int fibonacci(int n) {
    if (n <= 2)
        return 1;

    int fib1 = 1, fib2 = 1, fib_atual;
    for (int i = 3; i <= n; i++) {
        fib_atual = fib1 + fib2;
        fib1 = fib2;
        fib2 = fib_atual;
    }
    return fib_atual;
}
```

Esta versão calcula cada valor apenas uma vez, realizando exatamente $n - 2$ somas.

Lição: Nem sempre a solução recursiva é a melhor! Para Fibonacci, a versão iterativa é exponencialmente mais rápida.

8 Torre de Hanói

8.1 O Problema

Considere n discos de diâmetros diferentes colocados em um pino A. O problema da Torre de Hanói consiste em transferir todos os n discos do pino A (inicial) para o pino C (final), usando um pino B como auxiliar.

Regras:

- Apenas o disco do topo de um pino pode ser movido.
- Nunca um disco maior pode ficar sobre um disco menor.

8.2 Origem

O problema foi descrito em 1883 pelo matemático francês Édouard Lucas, baseado numa lenda hindu. Segundo a lenda, monges do templo de Kashi Vishwanath movem uma pilha de 64 discos de ouro segundo as regras. Quando todos os discos forem movidos, o mundo acabará.

8.3 Solução Recursiva

Vamos usar indução para resolver:

Teorema: É possível resolver o problema da Torre de Hanói com n discos.
Prova:

- **Base:** $n = 1$. Basta mover o único disco de A para C.
- **Hipótese:** Sabemos resolver o problema com $n - 1$ discos.
- **Passo:** Para resolver com n discos:
  1. Mova recursivamente os $n - 1$ primeiros discos de A para B (usando C como auxiliar).
  2. Mova o maior disco (que ficou em A) para C.
  3. Mova recursivamente os $n - 1$ discos de B para C (usando A como auxiliar).

8.4 Implementação

```
#include <stdio.h>

void hanoi(int n, char inicial, char final, char auxiliar) {
    if (n == 1) {
        printf("Mova␣o␣disco␣1␣do␣pino␣%c␣para␣o␣pino␣%c\n",
               inicial, final);
    } else {
        hanoi(n - 1, inicial, auxiliar, final);
        printf("Mova␣o␣disco␣%d␣do␣pino␣%c␣para␣o␣pino␣%c\n",
               n, inicial, final);
        hanoi(n - 1, auxiliar, final, inicial);
    }
}

int main() {
    hanoi(4, 'A', 'C', 'B');
    return 0;
}
```

8.5 Número de Movimentos

Seja $T(n)$ o número de movimentos necessários para mover n discos:

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 2T(n-1) + 1 & \text{se } n > 1 \end{cases}$$

Resolvendo: $T(n) = 2^n - 1$

- $T(3) = 7$ movimentos
- $T(4) = 15$ movimentos
- $T(5) = 31$ movimentos
- $T(64) = 18.446.744.073.709.551.615$ movimentos

Com 64 discos (lenda original), assumindo 1 segundo por movimento, seriam necessários aproximadamente 585 bilhões de anos!

9 Exercícios

Importante: Todas as funções devem ser implementadas sem comandos de repetição (`for`, `while`, etc.).

1. Escreva uma função recursiva que calcule a soma dos n primeiros números naturais: $S(n) = 1 + 2 + 3 + \cdots + n$.
2. Escreva uma função recursiva que calcule $n!$ (fatorial).
3. Escreva uma função recursiva que calcule o n-ésimo número da série de Fibonacci.
4. Escreva uma função recursiva que, dado um número inteiro positivo n, retorne o número de dígitos de n.
5. Escreva uma função recursiva que calcule a soma dos dígitos de um número inteiro não negativo.
6. Escreva uma função recursiva que, dado um número inteiro positivo n, retorne sua representação binária como uma string.
7. Escreva uma função recursiva que, dada uma string s e um caractere c, conte o número de ocorrências de c em s.

8. Escreva uma função recursiva que determine se uma string contém apenas dígitos (0-9).
9. Escreva uma função recursiva que inverta os elementos de um vetor (modificando o vetor original).
10. Escreva uma função recursiva que, dada uma lista de n números inteiros ordenados e um inteiro x, retorne o índice de x na lista usando busca binária, ou -1 se x não estiver na lista.
11. Implemente o algoritmo de ordenação *Merge Sort* de forma recursiva.
12. Implemente o algoritmo de ordenação *Quick Sort* de forma recursiva.
13. Escreva uma função recursiva que calcule todas as permutações de uma string.
14. Resolva o problema das N rainhas: coloque N rainhas em um tabuleiro $N \times N$ de xadrez de modo que nenhuma rainha ataque outra.
15. Implemente um resolvedor de Sudoku usando backtracking recursivo.

10 Resumo

Pontos-chave desta aula:

- Recursão divide problemas em subproblemas menores do mesmo tipo
- Toda função recursiva precisa de: (1) caso(s) base e (2) caso(s) recursivo(s)
- Cada chamada recursiva cria um novo frame na pilha
- Recursão geralmente resulta em código mais claro e conciso
- Soluções iterativas geralmente são mais eficientes em memória
- Nem sempre recursão é a melhor escolha (ex: Fibonacci)
- Verifique sempre se a recursão termina (convergência para caso base)
- Cuidado com cálculos repetidos em recursão (podem torná-la exponencial)