Recursão

Profª. Rose Yuri Shimizu

Roteiro

Definição

- É a propriedade daquilo que pode se repetir várias vezes
- Dependência entre os elementos do conjunto
  - Elemento atual depende da determinação de um elemento anterior ou posterior
- Condição de parada: necessária para terminar a recursão
- **Exemplos de recursões matemáticas**

Fatorial $n! = \begin{cases} 1, & \text{se } n = 0 \\ n.(n-1)!, & \text{se } n \geq 1 \end{cases}$

Fibonacci $f(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ f(n-1) + f(n-2), & \text{se } n \geq 2 \end{cases}$

Roteiro

1. Recursão

2. Recursão na programação

Algoritmos Recursivos

- São implementados através de funções:
  - Que invocam a si mesmos
  - Chamadas de funções recursivas
- Contribuem na implementação de algoritmos complexos em códigos mais compactos
- Sistemas atuais possibilitam uma execução eficiente das chamadas de função recursivas
  - Stacks: empilhamento das funções
  - Compiladores eficientes: otimizações

Algoritmos Recursivos

Execução

- Comportamento de uma pilha
- Cada iteração: dados são empilhados, inclusive o endereço de quem chamou a função (para onde retornar)
- Última iteração:
  - Último invocado termina o seu processamento
  - É retirado da pilha e o topo da pilha retoma sua execução
- Processo de desempilhamento continua até a base da pilha
- Assim, o invocador inicial pode finalmente terminar seu processamento

Algoritmos Recursivos

Fatorial iterativo $n! = \begin{cases} 1, & \text{se } n = 0 \\ n.(n-1)!, & \text{se } n \geq 1 \end{cases}$

```c
    //fatorial iterativo
    int n = 3;
    int t = 1;
    while(n>0) {
        t *= n;
        n--;
    }
    printf("%d! = %d\n", n, t); //n? t?

```

Algoritmos Recursivos

Fatorial recursivo $n! = \begin{cases} 1, & \text{se } n = 0 \\ n.(n-1)!, & \text{se } n \geq 1 \end{cases}$

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

```

```
//chamada de funcao
int x = fat(3);

```

Algoritmos Recursivos

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

```

```
//chamada de funcao
int x = fat(3);
/* stack
 chamada 1
|---------|
|         |
|         |
|         |
|         |
|         |
|         |
|         |
|  fat(3) |
x |    ?    |
|---------|
*/
```

Algoritmos Recursivos

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

```

```
    //chamada de funcao
    int x = fat(3);
    /* stack
     chamada 2
    |-----------|
    |           |
    |           |
    |           |
    |           |
    |           |
    |   fat(2)  |
  n |     3     |
    |   fat(3)  |
  x |     ?     |
    |-----------|
    */
```

Algoritmos Recursivos

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

```

```
    //chamada de funcao
    int x = fat(3);
    /*  stack
     chamada 3
    |-----------|
    |           |
    |           |
    |           |
    |   fat(1)  |
  n |     2     |
    |   fat(2)  |
  n |     3     |
    |   fat(3)  |
  x |     ?     |
    |-----------|
    */
```

Algoritmos Recursivos

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

```
//chamada de funcao
int x = fat(3);
/*  stack
 chamada 4
|-----------|
 |          |
 |  fat(0)  |
n|    1     |
 |  fat(1)  |
n|    2     |
 |  fat(2)  |
n|    3     |
 |  fat(3)  |
x|    ?     |
|-----------|
*/
```

Algoritmos Recursivos

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

```
    //chamada de funcao
    int x = fat(3);
   /*
    retorno 1    retorno 2    retorno 3    retorno 4    original
   |---------|  |---------|  |---------|  |---------|  |---------|
 n |    0    |  |         |  |         |  |         |  |         |
   | fat(0)  |  |         |  |         |  |         |  |         |
 n |    1    |  |   1*1   |  |         |  |         |  |         |
   | fat(1)  |  | fat(1)  |  |         |  |         |  |         |
 n |    2    |  |    2    |  |   2*1   |  |         |  |         |
   | fat(2)  |  | fat(2)  |  | fat(2)  |  |         |  |         |
 n |    3    |  |    3    |  |    3    |  |   3*2   |  |         |
   | fat(3)  |  | fat(3)  |  | fat(3)  |  | fat(3)  |  |         |
 x |    ?    |  |    ?    |  |    ?    |  |    ?    |  |    6    |
   |---------|  |---------|  |---------|  |---------|  |---------|
    */
```

Algoritmos Recursivos

Fibonacci iterativo - F(n) = F(n-1) + F(n-2)
F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2)

```
int n = 4;
int f, f1, f2;
f2 = 0;  //F(0)
f1 = 1;  //F(1)
f = n;   //F(n) para n = 0 ou 1

for(int i=2; i<=n; i++)
{
  f = f1 + f2; //Calculando F(n)

  //Calculando F(n-1) e F(n-2) para a próxima iteração
  f2 = f1; //próximo F(n-2) = atual F(n-1)
  f1 = f;  //próximo F(n-1) = atual F(n)
}
```

Algoritmos Recursivos

Fibonacci iterativo - F(n) = F(n-1) + F(n-2)
F(0) = 0
F(1) = 1

```
int n = 4;
int f, f1, f2;
f2 = 0;   //<-
f1 = 1;   //<-
f = n;    //<- ??

for(int i=2; i<=n; i++)
{
   f = f1 + f2;

   f2 = f1;
   f1 = f;
}
```

i	f	f2	f1
-	4	0	1

Algoritmos Recursivos

Fibonacci iterativo - F(n) = F(n-1) + F(n-2)
F(0) = 0
F(1) = 1
F(2) = F(1) + F(0) = 1 + 0 = 1

```
int n = 4;
int f, f1, f2;
f2 = 0;
f1 = 1;
f = n;

for(int i=2; i<=n; i++) //<-
{
  f = f1 + f2; //<-

  f2 = f1; //<-
  f1 = f;  //<-
}
```

i	f	f2	f1
-	4	0	1
2	**?**	**?**	**?**

Algoritmos Recursivos

Fibonacci iterativo - F(n) = F(n-1) + F(n-2)
F(0) = 0
F(1) = 1
F(2) = F(1) + F(0) = 1 + 0 = 1

```
int n = 4;
int f, f1, f2;
f2 = 0;
f1 = 1;
f = n;

for(int i=2; i<=n; i++)
{
  f = f1 + f2; //<-

  f2 = f1; //<-
  f1 = f;  //<-
}
```

i	f	f2	f1
-	4	0	1
2	**1**	**1**	**1**

Algoritmos Recursivos

Fibonacci iterativo - F(n) = F(n-1) + F(n-2)
F(0) = 0
F(1) = 1
F(2) = F(1) + F(0) = 1 + 0 = 1
F(3) = F(2) + F(1) = 1 + 1 = 2

```c
int n = 4;
int f, f1, f2;
f2 = 0;
f1 = 1;
f = n;

for(int i=2; i<=n; i++) //<-
{
   f = f1 + f2; //<-

   f2 = f1; //<-
   f1 = f;  //<-
}
```

i	f	f2	f1
-	4	0	1
2	1	1	1
3	**?**	**?**	**?**

Algoritmos Recursivos

Fibonacci iterativo - F(n) = F(n-1) + F(n-2)
F(0) = 0
F(1) = 1
F(2) = F(1) + F(0) = 1 + 0 = 1
F(3) = F(2) + F(1) = 1 + 1 = 2

```
int n = 4;
int f, f1, f2;
f2 = 0;
f1 = 1;
f = n;

for(int i=2; i<n; i++)
{
   f = f1 + f2; //<-

   f2 = f1; //<-
   f1 = f;  //<-
}
```

i	f	f2	f1
-	4	0	1
2	1	1	1
3	**2**	**1**	**2**

Algoritmos Recursivos

Fibonacci iterativo - F(n) = F(n-1) + F(n-2)

F(0) = 0
F(1) = 1
F(2) = F(1) + F(0) = 1 + 0 = 1
F(3) = F(2) + F(1) = 1 + 1 = 2
F(4) = F(3) + F(2) = 2 + 1 = 3

```
int n = 4;
int f, f1, f2;
f2 = 0;
f1 = 1;
f = n;

for(int i=2; i<=n; i++) //<-
{
   f = f1 + f2; //<-

   f2 = f1; //<-
   f1 = f;  //<-
}
```

i	f	f2	f1
-	4	0	1
2	1	1	1
3	2	1	2
4	**?**	**?**	**?**

Algoritmos Recursivos

Fibonacci iterativo - F(n) = F(n-1) + F(n-2)
F(0) = 0
F(1) = 1
F(2) = F(1) + F(0) = 1 + 0 = 1
F(3) = F(2) + F(1) = 1 + 1 = 2
F(4) = F(3) + F(2) = 2 + 1 = 3

```
int n = 4;
int f, f1, f2;
f2 = 0;
f1 = 1;
f = n;

for(int i=2; i<=n; i++)
{
   f = f1 + f2; //<-

   f2 = f1; //<-
   f1 = f;  //<-
}
```

i	f	f2	f1
-	4	0	1
2	1	1	1
3	2	1	2
4	**3**	2	3

Algoritmos Recursivos

Fibonacci é inerentemente/naturalmente recursivo
Fibonacci recursivo

$$f(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ f(n-1) + f(n-2), & \text{se } n \geq 2 \end{cases}$$

Como aplicar a recursividade?

Algoritmos Recursivos

Fibonacci recursivo $f(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ f(n-1) + f(n-2), & \text{se } n \geq 2 \end{cases}$

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

//chamada de funcao
int a = fib(3);
```

Algoritmos Recursivos

Validade dos algoritmos Recursivos

- A sequencia recursiva precisa ser finita
- Podemos utilizar a indução matemática para provar sua validade
- Método da indução finita: provar propriedades que são verdadeiras para uma sequência de objetos
  1. Passo base
     (ex.: T é válido para $n = 1$)
  2. Passo indutivo ou hipótese da indução
     (ex.: para todo $n > 1$, se T é válido para $n - 1$, então T é válido para n)
- Podemos simplificar garantindo:
  - Caso base (condição de parada)
  - Em cada chamada o valor da função recursiva tenda para o alcance da condição de parada (garantindo o término da recursão)

Algoritmos Recursivos

Exemplo da página do prof. Paulo Feofiloff

```c
int max(int n, int v[]) {
  if (n == 1) return v[0];
  else {
    int x = max(n-1, v);
    //x is largest in v[0..n-2]

    if (x > v[n-1]) return x;
    else return v[n-1];
  }
}
/* v[3] -> 77 88 66
    max(3, v)
    | max(2, v)
    | | max(1, v)
    | | |_ returns 77
    | |_ returns 88
    |_ returns 88
*/
```

```c
int max(int i, int n, int v[]) {
  if (i == (n-1)) return v[i];
  else {
    int x = max(i+1, n, v);
    //x is largest in v[i..n-1]

    if (x > v[i]) return x;
    else return v[i];
  }
}
/* v[6] -> 100 99 77 88 66 87
    max(1, 4, v)
    | max(2, 4, v)
    | | max(3, 4, v)
    | | |_ returns 88
    | |_ returns 88
    |_ returns 99
*/
```

Algoritmos Recursivos

Analise

Exemplo do livro do Sedgewick

```
    //algoritmo euclidiano : encontrar o maximo divisor comum
    int mdc(int m, int n) {
        if(n==0) return m;
        return mdc(n, m%n); //% resto = mod
    }

```

Análise simplificada:

1. Devem garantir, explicitamente, o caso base
   - Para $n = 0$, o retorno é m
2. Cada chamada deve tratar argumentos progressivamente menores
   - Um número t divide m e n se e somente se t divide n e $m\%n$ pois m é igual ($m\%n$) mais um múltiplo (t) de n ($5 = (5\%2) + 2 * 2$)
   - Próximo passo, o parâmetro n é reduzido ao resto da divisão entre os argumentos de entrada

Algoritmos Recursivos

```
mdc(25, 9) = mdc(9, 25%9)
    mdc(9, 7) = mdc(7, 9%7)
        mdc(7, 2) = mdc(2, 7%2)
            mdc(2, 1) = mdc(1, 2%1)
                mdc(1, 0) = 1
```

```
mdc(20, 24) = mdc(20, 24%20)
    mdc(20, 4) = mdc(4, 20%4)
        mdc(4, 0) = 4
```

Algoritmos Recursivos

Exemplo do livro do Sedgewick

```
    //resolver expressao matematica com notacao prefixa
    //* + 7 * * 4 6 + 8 9 5 = (7+((4*6)*(8+9)))*5
    char *a; int i=0;
    int eval(){
        int x=0;
        while(a[i] == ' ') i++; //procura por operadores e digitos
        if(a[i] == '+') {
            i++;
            return eval()+eval();
        }
        if(a[i] == '*') {
            i++;
            return eval()*eval();
        }

        //equivalente numerico de uma sequencia de caracteres
        while((a[i] >= '0') && (a[i] <= '9'))
            //calcula o decimal, centena ... + valor numerico
            x = 10*x + (a[i++]-'0'); //tabela ascii

        return x;
    }
```

Algoritmos Recursivos

```
    resolver expressao matematica com notacao prefixa
    * + 7 * * 4 6 + 8 9 5 = (7+((4*6)*(8+9)))*5

    |eval() * + 7 * * 4 6 + 8 9 5
    |    |eval() + 7 * * 4 6 + 8 9
    |    |    |eval() 7
    |    |    |eval() * * 4 6 + 8 9
    |    |    |    |eval() * 4 6
    |    |    |    |    |eval() 4
    |    |    |    |    |eval() 6
    |    |    |    |    return  4 * 6 = 24
    |    |    |    |eval() + 8 9
    |    |    |    |    |eval() 8
    |    |    |    |    |eval() 9
    |    |    |    |    return 8 + 9 = 17
    |    |    |    return 24 * 17 = 408
    |    |    return 7 + 408 = 415
    |    |eval() 5
    |    return 5*415 = 2075
```

Algoritmos Recursivos

Analise

Exemplo do livro do Sedgewick

```
//n > 0
int puzzle(int n) {
    if(n==1) return 1; //condição de parada

    if(n%2 == 0) {
        return puzzle(n/2); //diminui a entrada:
                            //    tende para parda
    } else {
        return puzzle(3*n+1); //cuidado com o aumento
                              //alcança a condição de parada?
                              //aumento arbritário da profundidade da pilha
    }
}

```

Usos da recursividade - observações

- Se uma instância for pequena: use força bruta, resolva diretamente
- Senão, reduza em instância menores do mesmo problema
- Resolva por partes e volte para instância original
- Essa é a técnica da "divisão e conquista"
  - Resolva os subproblemas para resolver o problema
  - Consiste em:
    - Dividir o problema em partes menores
    - Encontrar as soluções das partes
    - Combinando-as para obter a solução global (conquista)

```
    divisao_conquista(d) {
        se simples
            calculo_direto(d)
        senao
            combina(divisao_conquista(decompoe(d))
    }
```

  - O custo computacional geralmente é determinada pela relação da recorrência (profundidade da pilha)
  - Tende a algoritmos mais eficientes
  - Auxilia em problemas mais complexos, dividindo em problemas menores
  - Facilita a paralelização na fase da conquista

Usos da recursividade - observações

Sobre o algoritmo abaixo:

- O que faz?
- Funciona?
- É mais eficiente?

```
int funcao1(int a[], int l, int r) {
    int u, v;
    int m = (l+r)/2;
    u = funcao1(a, l, m);
    v = funcao1(a, m+1, r);
    if(u>v) return u;
    return v;
}
```

Usos da recursividade - observações

```c
int max(int a[], int l, int r) {
    int u, v;
    int m = (l+r)/2;
    if(l==r) return a[l];
    u = max(a, l, m);
    v = max(a, m+1, r);
    if(u>v) return u;
    return v;
}

```

```
                 9
          /              \
        9                 7
     /    \            /     \
    9       \         6       \
   / \       |       / \       |
  9   \      4      6   \      7
 /\    |    /\     /\    |    /\
9  6   3   4  1   6  3   5   2  7
```

Usos da recursividade - observações

- Cuidado com estouro de pilhas: técnicas como a recursão de cauda (tail call - chamada recursiva é a última instrução a ser executada - função mdc) e otimizações na compilação (gcc -O2)
- Algoritmos tendem a ter forte dependência entre os valores
- Pode ser aplicado em problemas de:
  - Planejamento de caminhos em robotica
  - Problemas de tentativa e erro (*backtracking*: errou? volta e tenta outra solução)
  - Compiladores (analisadores léxicos)
  - **Manipulação das estrutura de dados** (formas de armazenamento de dados)
  - **Algoritmos de pesquisas, ordenação**

Usos da recursividade - observações

```c
#include <stdio.h>

void recursiveFunction1(int num) {
    if (num > 0)
        recursiveFunction1(num - 1);
    printf("%d\n", num);
}

void recursiveFunction2(int num) {
    printf("%d\n", num);
    if (num > 0)
        recursiveFunction2(num - 1);
}

int main()
{
    //int 4 bytes -> 8 MB = 8000 KB = 8000000 B
    //recursiveFunction1(8000000);
    recursiveFunction2(8000000);
    return 0;
}
```

Usos da recursividade - observações

```
Teste com gcc teste.c
          gcc -O1 teste.c
          gcc -O2 teste.c

Observe o assembly
          gcc -S teste.c
          cat teste.s
          gcc -S -O2 teste.c
          cat teste.s
```