Recursão

Prof^{a}. Rose Yuri Shimizu

Roteiro

1. Recursão

2. Recursão na programação

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

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

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

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

```

```c
    //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 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)
    - Que cada chamada decremente o valor da função recursiva, que garanta o alcance da condição de parada (garantindo a stack e o término da recursão)

Algoritmos Recursivos

Exemplo da página do prof. Paulo Feofiloff

```
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[0..3] -> 77 88 66
     max(3,v)
     | max(2,v)
     | | max(1,v)
     | | |__________ returns 77
     | |____________ returns 88
     |______________ returns 88
*/
```

Corretude por prova indutiva:

Passo base: para $n = 1$, o maior é v[0]

Passo indutivo: para $n \geq 1$,

- Sub-vetores menores de v: $v[0..n-2]$ e $v[n-1]$
- Se $max(n-2, v)$ retorna o máximo de $v[0..n-2]$
- Então, após a instrução $x = max(n-1, v)$, x é o maior valor de $v[0..n-2]$
- Portanto, a solução é o maior entre x e $v[n-1]$ (outra parte do vetor)
  - Conforme computação da função
- Conclui-se então, que a função *max* retorna o maior elemento de um dado vetor

Algoritmos Recursivos

Versão mais genérica da função max: intervalo do vetor

```
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];
  }
}
```

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

```
int puzzle(int n) {
    if(n==1) return 1;
    if(n%2 == 0) {
        return puzzle(n/2);
    } else {
        return puzzle(3*n+1);
    }
}
```

```
//aumenta a recursao aumentando o custo da funcao
puzzle(3*n+1)
```

Usos da recursividade - recomendações

- 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 - recomendações

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

Usos da recursividade - recomendações

- Pode ser aplicado em problemas de:
  - Operações de multiplicacao de matrizes
  - Planejamento de caminhos em robotica movel
  - 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**