Fundamentos de Programação
Recursividade - Introdução

Dainf - UTFPR

Profa. Leyza Baldo Dorini

Recursividade

Um objeto é dito recursivo se pode ser definido em termos de si próprio.

Componentes da Recursão

Toda recursão é composta por

- Um caso base, uma instância do problema que pode ser solucionada facilmente. Por exemplo, é trivial fazer a soma de uma lista com um único elemento.
- Uma ou mais chamadas recursivas, onde o objeto define-se em termos de sí próprio, tentando convergir para o caso base. A soma de uma lista de n elementos pode ser definida a partir da lista da soma de $n - 1$ elementos.

Como definir recursivamente a soma abaixo?

$$\sum_{k=m}^{n} k = m + (m+1) + \cdots + (n-1) + n$$

Primeira definição recursiva

$$\sum_{k=m}^{n} k = \begin{cases} m & \text{se } n = m \\ \displaystyle\sum_{k=m}^{n-1} k + n & \text{se } n > m \end{cases}$$

Recursão na matemática

Segunda definição recursiva

$$\sum_{k=m}^{n} k = \begin{cases} m & \text{se } n = m \\ m + \displaystyle\sum_{k=m+1}^{n} k & \text{se } n > m \end{cases}$$

```c
int soma(int m, int n) {
  if (m == n)
    return n;
  else
    return m + soma(m+1, n);
}

main() {
  int m, n, s;

  printf("Digite m e n: ");
  scanf("%d %d", &m, &n);
  s = soma(m, n);
  printf("Soma de %d a %d = %d\n", m, n, s);
}
```

Pilha de execução

Quando uma função é chamada:

- é preciso inicializar os parâmetros da função com os valores passados como argumento;
- o sistema precisa saber onde reiniciar a execução do programa.

Informações de cada função (variáveis retorno) devem ser guardadam até a função acabar a execução.

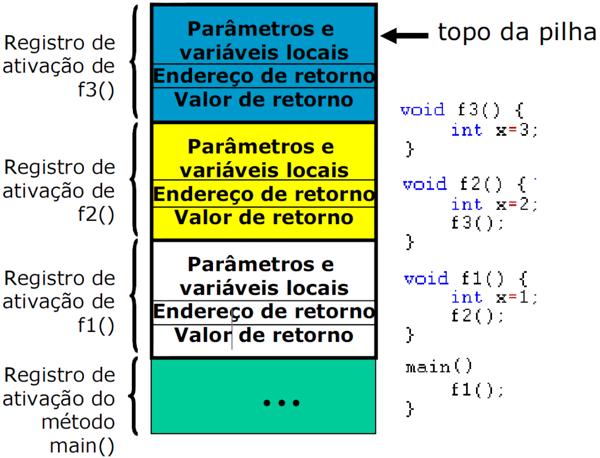
- cada chamada de função o sistema reserva espaço para parâmetros, variáveis locais e valor de retorno.

Mas como diferenciar as variáveis de cada chamada? - Registro de ativação

Registro de ativação

- Área de memória que guarda o estado de uma função, ou seja: variáveis locais, valores dos parâmetros, endereço de retorno (instrução após a chamada do método corrente) e valor de retorno;
- Registro de ativação são criados em uma pilha em tempo de execução;
- Existe um registro de ativação (um nó na pilha) para cada método;
- Quando um método é chamado é criado um registro de ativação para ele, que é empilhado na pilha;
- Quando o método finaliza sua execução o registro de ativação desse método é desalocado.

Ilustração



Registro de ativação de f3()
Registro de ativação de f2()
Registro de ativação de f1()
Registro de ativação do método main()
Parâmetros e variáveis locais
Endereço de retorno
Valor de retorno
Parâmetros e variáveis locais
Endereço de retorno
Valor de retorno
Parâmetros e variáveis locais
Endereço de retorno
Valor de retorno
...
topo da pilha
void f3() {
int x=3;
}
void f2() {
int x=2;
f3();
}
void f1() {
int x=1;
f2();
}
main()
f1();
}

Em resumo...

- A cada chamada de função o sistema reserva espaço para parâmetros, variáveis locais e valor de retorno.

main	s
	ret: ??
soma	m: 5
	n: 10
	ret: ??
soma	m: 6
	n: 10
	...

Estouro de pilha de execução

"To understand recursion you must first understand recursion."

- O que acontece se a função não tiver um caso base?
- O sistema de execução não consegue implementar infinitas chamadas. Lembre-se, somente Chuck Norris conta até o infinito.

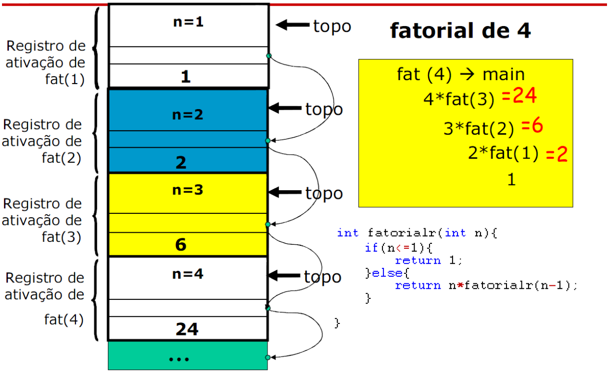
```c
void recInfinita(int n) {
  if (n % 10000 == 0)
    printf("n = %d\n", n);
  recInfinita(n+1);
}

main() {
  recInfinita(0);
}
```

Exemplo clássico: fatorial!

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n(n-1)! & \text{se } n > 0 \end{cases}$$

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

Exemplo: fatorial




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

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Empilha `fatorial(4)`

`fatorial(4)`

`-> return 4*fatorial(3)`

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Empilha `fatorial(3)`

`fatorial(3)`	`-> return 3*fatorial(2)`
`fatorial(4)`	`-> return 4*fatorial(3)`

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Empilha `fatorial(2)`

`fatorial(2)`	`-> return 2*fatorial(1)`
`fatorial(3)`	`-> return 3*fatorial(2)`
`fatorial(4)`	`-> return 4*fatorial(3)`

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Empilha fatorial(1)

fatorial(1)	-> return 1*fatorial(0)
fatorial(2)	-> return 2*fatorial(1)
fatorial(3)	-> return 3*fatorial(2)
fatorial(4)	-> return 4*fatorial(3)

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Empilha fatorial(0)

fatorial(0)	-> return 1 (caso trivial)
fatorial(1)	-> return 1*fatorial(0)
fatorial(2)	-> return 2*fatorial(1)
fatorial(3)	-> return 3*fatorial(2)
fatorial(4)	-> return 4*fatorial(3)

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Desempilha fatorial(0)

fatorial(0)	-> return 1 (caso trivial)
fatorial(1)	-> return 1***fatorial(0)**
fatorial(2)	-> return 2*fatorial(1)
fatorial(3)	-> return 3*fatorial(2)
fatorial(4)	-> return 4*fatorial(3)

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Desempilha `fatorial(1)`

fatorial(1)	-> return 1*1
fatorial(2)	-> return 2***fatorial(1)**
fatorial(3)	-> return 3*fatorial(2)
fatorial(4)	-> return 4*fatorial(3)

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Desempilha `fatorial(2)`

fatorial(2)	-> return 2*(1*1)
fatorial(3)	-> return 3*fatorial(2)
fatorial(4)	-> return 4*fatorial(3)

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Desempilha `fatorial(3)`

fatorial(3)	`-> return 3*(`**2*1*1**`)`
fatorial(4)	`-> return 4*`**fatorial(3)**

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Desempilha `fatorial(4)`

`fatorial(4)` `-> return 4*`**`(3*2*1*1)`**

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Resultado = 24

Função Fibonacci

- A função Fibonacci retorna o n-ésimo número da seqüência: 1, 1, 2, 3, 5, 8, 13, ...

- Os dois primeiros termos são iguais a 1 e cada um dos demais números é a soma dos dois números imediatamente anteriores.

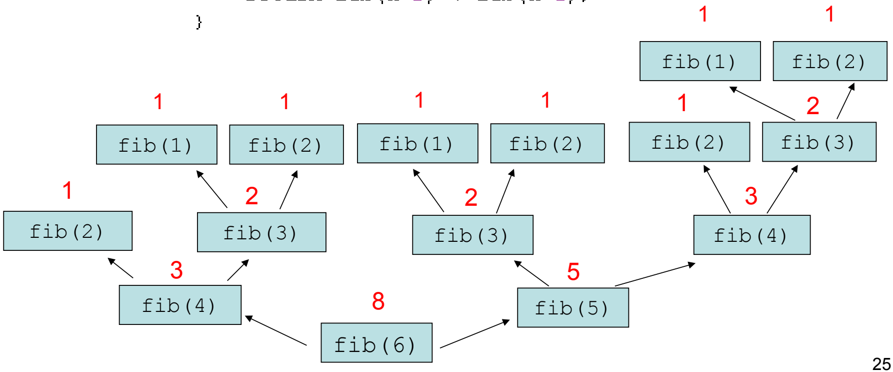
- Sendo assim, o n-ésimo número *fib(n)* é dado por:

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

Função Fibonacci

- Veja uma implementação recursiva para esta função:

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



Função Fibonacci

- A função recursiva para cálculo do n-ésimo termo da seqüência é extremamente ineficiente, uma vez que recalcula o mesmo valor várias vezes

Observe agora uma versão iterativa da função fib:

```
int fib(int n)
{
  int i,a,b,c;
  if ((n == 1) || (n == 2))
    return 1;
  else
  {
    a = 0;
    b = 1;
    for (i = 3; i <= n; i++)
    {
      c = a + b;
      a = b;
      b = c;
    }
    return c;
  }
}
```

Função Fibonacci

- O livro dos autores Brassard e Bradley (*Fundamentals of Algorithmics*, 1996, pág. 73) apresenta um quadro comparativo de tempos de execução das versões iterativa e recursiva:

n	10	20	30	50	100
recursivo	8 ms	1 s	2 min	21 dias	10^9 anos
iterativo	0.17 ms	0.33 ms	0.50 ms	0.75 ms	1,50 ms

- Portanto: um algoritmo recursivo nem sempre é o melhor caminho para se resolver um problema.
- No entanto, a recursividade muitas vezes torna o algoritmo mais simples.

Potência

Exemplo: potência!

$$x^n = \begin{cases} \dfrac{1}{x^{(-n)}} & \text{se } n < 0 \\ 1 & \text{se } n = 0 \\ x.x^{(n-1)} & \text{se } n > 0 \end{cases}$$

```
double pot(double x, int n) {
  if (n == 0) return 1;
  else if (n < 0)
         return 1/pot(x, -n);
       else
         return x*pot(x, n-1);
}
```

Maior elemento de um vetor

Problema: encontrar o valor do maior elemento do vetor `v[0... n-1]` (com n elementos)

```c
//iterativo
int maior(int v[], int n)
{
    int x = v[0], i;
    for (i=1; i< n; i++)
        if (x < v[i])
            x = v[i];
    return x;
}
```

Maior elemento de um vetor

```
//recursivo
int maior_r(int v[], int n)
{
    int x;
    if (n==1)
        return v[0];

    x = maior_r(v, n-1);
    if (x > v[n-1])
        return x;
    else
        return v[n-1];
}
```

Busca binária

```c
int bb(int vetor[], int ini, int fim, int pesq){
    int meio;

    if(ini>fim)
        return -1;
    else{
        meio = (ini+fim)/2;

        if(vetor[meio]==pesq)
            return meio;
        else if (vetor[meio]<pesq)
            return bb(vetor,meio+1,fim,pesq);
        else
            return bb(vetor,ini,meio-1,pesq);
    }
}
```

Questões de desempenho

- É sempre mais simples usar recursão? Lembre-se que todo algoritmo recursivo pode ser transformado para um iterativo.
- É mais eficiente? As implementações recursivas tendem a ser mais custosas tanto em termos de memória como também de CPU, já que é alocada uma pilha de processamento e de memória muito grande.

Questões de desempenho

```
double pot(double x, int n) {
  double res = 1;
  int i;

  if (n < 0)
    for (i = 1; i <= -n; i++)
      res *= 1/x;
  else
    for (i = 1; i <= n; i++)
      res *= x;
  return res;
}
```

Fibonacci

Exemplo clássico: sequência de Fibonacci!

$$n! = \begin{cases} fib(0) & 0 \\ fib(1) & 1 \\ n(n-1)! & \text{se } n > 0 \end{cases}$$

```
//A função recebe n >= 0 e devolve F(n).
long fibonacci(int n) {

    if (n <= 1) return n;

    return fibonacci(n-1) + fibonacci(n-2);
}
```