PCC104 - Projeto e Análise de Algoritmos

Marco Antonio M. Carvalho

Departamento de Computação
Instituto de Ciências Exatas e Biológicas
Universidade Federal de Ouro Preto

Conteúdo

Fonte

Este material é baseado nos livros

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
- S. Halim. *Competitive Programming*. 3rd Edition, 2013.
- Ian Parberry and William Gasarch. *Problems on Algorithms*. Second Edition, 2002.
- Ian Parberry *Lecture Notes on Algorithm Analysis and Complexity Theory*. Fourth Edition, 2001.

Licença

Recapitulando... Algoritmos Recursivos

Definição

Um algoritmo pode ser composto por funções, que, por sua vez, podem invocar outras funções.

Quando uma função invoca a si própria, a denominamos **função recursiva**.

É um conceito poderoso, pois define sucintamente conjuntos infinitos com instruções infinitas.

A idéia é aproveitar a solução de um ou mais subproblemas para resolver o problema original.

Recursão vs. Iteração

Algoritmos **recursivos** se opõem a algoritmos **iterativos**, em que a solução é construída de por meio de uma sequência de passos linear.

Algoritmos Recursivos

Princípio

A recursividade está intimamente relacionada ao princípio de indução matemática.

Parte-se da hipótese de que a solução para um problema de tamanho t pode ser obtida a partir da solução para o mesmo problema, porém, de tamanho $t - 1$.

Aplicações

Além das aplicações diretas, a recursividade é a base para os paradigmas *Backtracking*, Dividir e Conquistar e também está relacionado à Programação Dinâmica (*Top-Bottom*).

Algoritmos Recursivos

Projeto

Um algoritmo recursivo é composto, em sua forma mais simples, de uma **condição de parada** e de um **passo recursivo**.

Condição de Parada

Garante que a recursividade é finita, geralmente, definida sobre um **caso base**.

Por exemplo, a condição n>0 garante a parada com n positivo.

Passo Recursivo

Realiza as chamadas recursivas e processa os diferentes valores de retorno, quando adequado.

A idéia é associar um parâmetro n e realizar o passo recursivo sobre $n - 1$.

Observações

Para cada chamada de uma função, recursiva ou não, os parâmetros e as variáveis locais são empilhadas na pilha de execução.

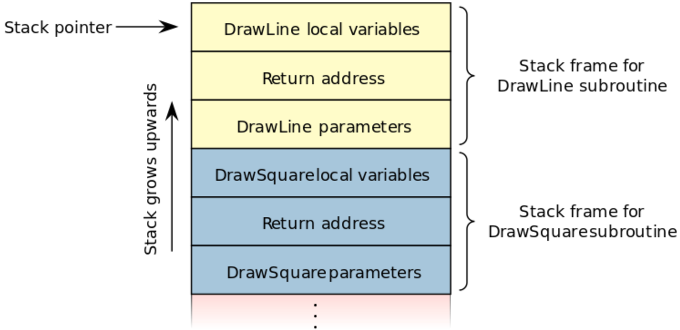
Internamente, quando qualquer chamada de função é feita dentro de um programa é criado um **registro de ativação** (ou *frame*) na **pilha de execução** do programa.

O registro de ativação armazena os parâmetros e variáveis locais da função, bem como o “ponto de retorno” no programa ou subprograma que chamou esta função.

Ao final da execução dessa função, o registro é desempilhado e a execução volta ao subprograma que chamou a função.

O tempo de execução é maior, devido ao *overhead* introduzido pelo gerenciamento das chamadas recursivas na **pilha de execução**.

Exemplo



Exemplo de pilha de execução para uma função hipotética DrawLine que é invocada por outra função hipotética DrawSquare.

Algoritmos Recursivos

Exemplo 1

Suponhamos que queremos imprimir recursivamente todos os primeiros cinco números inteiros.

```
int recursiveFunction(int n)
    se n < 5 então
        printf("%d\n", n);
        recursiveFunction(n + 1);
    fim
```

Exemplo 1

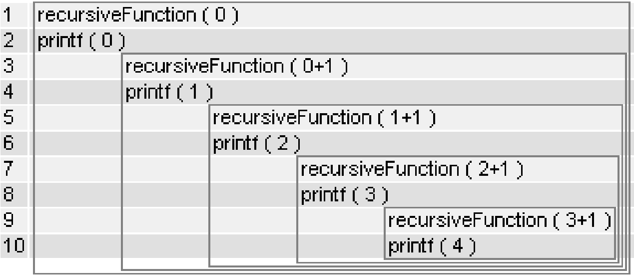


Ilustração da execução do exemplo.

Exemplo 2

Suponhamos que queremos imprimir recursivamente todos os primeiros cinco números inteiros, porém, em outra versão.

```
int recursiveFunction(int n)
    se n < 5 então
        recursiveFunction(n + 1);
        printf("%d\n", n);
    fim
```

Exemplo 2

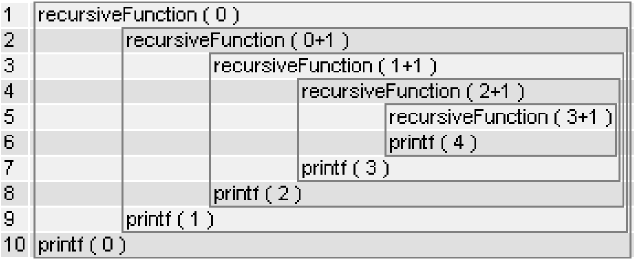


Ilustração da execução do exemplo.

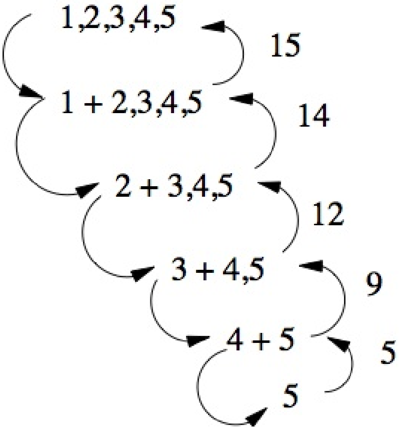
Algoritmos Recursivos

Exemplo 3

Suponhamos que queremos somar recursivamente todos os números inteiros entre m e n, inclusive.

```
int soma(int m, int n)
se m == n então
    return(m);
senão
    return m+soma(m+1, n);
fim
```

Exemplo 3



Árvore de recursão para $m = 1$ e $n = 5$.

Recursão em Cauda vs. Recursão Crescente

Na **Recursão Crescente** (ou funções crescentemente recursivas), depois de encerrada a chamada recursiva outras operações ainda são realizadas.

Na **Recursão em Cauda**, não existe processamento a ser realizado depois de encerrada a chamada recursiva, ou seja, a chamada recursiva é a última instrução a ser executada.

A recursão em cauda geralmente é mais rápida do que recursão crescente, uma vez que não é necessário armazenar todo o contexto na pilha de execução do programa.

Uma maneira de o compilador otimizar a recursão em cauda é reutilizar registros de ativação na pilha de execução, ao invés de criar novos.

Recursão em Cauda vs. Recursão Crescente

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

```
int somaCrescente(int x)
{
    if(x==1)
        return x;
    return x+somaCrescente(x-1);
}
```

Observações

Todo algoritmo recursivo possui uma versão iterativa equivalente, bastando utilizar uma pilha explícita.

Usualmente, os compiladores transformam funções recursivas de códigos-fonte em funções iterativas.

Quando se trata de recursividade em cauda, esta transformação é realizada com maior facilidade.

Algumas linguagens funcionais não possuem laços de repetição, apenas recursividade.

Com efeito, sem o *overhead* pela manutenção da pilha de execução em linguagens funcionais, a recursividade é uma operação muito rápida, tal que a diferença de desempenho entre códigos iterativos e recursivos é irrelevante.

Observações pt. 2

Se um problema é definido em termos recursivos, a implementação via algoritmos recursivos é facilitada.

Entretanto, isto não quer dizer que esta será a melhor solução.

É necessário estar atento ao **fator de ramificação** da recursão, ou seja, quantas chamadas recursivas serão feitas por vez.

Chance de loop infinito.

Erros de implementação fatalmente geram estouro da pilha de execução.

Estouro de Pilha

Normalmente, a pilha de execução possui uma região limitada de memória, geralmente no início do próprio programa.

Quando um há uso de mais memória do que o suportado, ocorre o estouro da pilha, que implica na suspensão da execução do programa.

Em recursividade, empilhar muitos parâmetros e variáveis locais pode causar este problema.

Recursão em Cauda vs. Estouro de Pilha

Algumas linguagens tiram proveito da recursão em cauda e não ocupam espaço na pilha de execução.

Desta maneira, funções recursivas em cauda não estouram a pilha, mesmo em *loop* infinito.

Recursivo vs Iterativo

```
int fibRec(int n)
{
    if(n<2)
        return 1;
    return fibRec(n-1)+fibRec(n-2);
}
```

```
int fibIt(int n)
{
    int i=1; fib=1; anterior=0;
    while (i < n) {
        temp = fib;
        fib = fib + anterior;
        anterior = temp;
        i++;
    }
    return fib;
}
```

Quando Não Utilizar Recursividade

Nem todo problema de natureza recursiva deve ser resolvido por um algoritmo recursivo.

Estes procedimentos podem ser caracterizados por $P \equiv$ **if** B **then** (S, P):

- P: Procedimento recursivo;
- B: Condição de continuidade da recursão;
- S: Conjunto de instruções a serem executadas.

Exemplo: **if** $i < n$ **then** $(i = i + 1;\ f * = i;\ P)$.

Tais problemas podem ser transformados pela eliminação da recursividade em cauda em uma versão iterativa $P \equiv (x = x_0;$ **while** B **do** $S)$.

Exemplo - Fibonacci

Os números de Fibonacci são definidos da seguinte maneira:

- $f_0 = 1$;
- $f_1 = 1$;
- $f_n = f_{n-1} + f_{n-2},\ \forall n > 2$.

Forma Fechada

A forma fechada para a recorrência é $f_n = \frac{1}{\sqrt{5}}[\phi^n - (-\phi)^{-n}]$, em que $\phi = \frac{\sqrt{5}+1}{2} \approx 1,618$ é a **razão áurea**.

Observação

Da mesma forma como utilizamos a forma fechada para resolver recorrências, podemos utilizá-las para evitar algoritmos recursivos (e suas versões iterativas)!

```
int fibRec(int n)
{
    if(n==0)
        return 0;
    return fibRec(n-1)+fibRec(n-2);
}
```

Exemplo - Fibonacci

O algoritmo proposto é extremamente ineficiente, pois recalcula repetidas vezes o mesmo valor.

A complexidade de tempo para calcular f_n, considerando as operações de adição, é $O(\phi^n)$.

A versão iterativa deste algoritmo possui complexidade de tempo $O(n)$.

Fibonacci Recursivo vs. Fibonacci Iterativo

n	20	30	50	100
Recursivo	1 s	2 m	21 dias	10^9 anos
Iterativo	1/3 ms	1/2 ms	3/4 ms	1,5 ms

Análise de Funções Recursivas

Além de analisar os algoritmos recursivos, também é necessário analisar as funções de recorrência para evitarmos casos excepcionais, como funções que crescem assustadoramente rápido, funções não bem definidas e funções que não se sabe se são bem definidas.

Função 91 de McCarthy – Bem Definida

$$M(n) = \begin{cases} n - 10, & \text{se } n > 100 \\ M(M(n+11)), & \text{se } n \leq 100 \end{cases}$$

$$\begin{aligned} M(99) &= (M(M(110))) \\ &= M(100) \\ &= M(101) \\ &= 91 \end{aligned}$$

Algoritmos Recursivos

Função de Ackermann

$$
\begin{aligned}
A(0,n) &= n+1 \\
A(m,0) &= A(m-1,1) \\
A(m,n) &= A(m-1, A(m,n-1))
\end{aligned}
$$

Análise

Esta função cresce assustadoramente rápido: $A(4,3) = A(3, 2^{65536} - 3)$.

$A(4,2)$ é maior do que o número de partículas do universo elevado a potência 200.

$A(5,2)$ não pode ser escrito como uma expansão decimal no universo físico.

Para valores de $m > 4$ e $n > 1$, os valores só podem ser expressos utilizando-se a própria notação.

Função Não Bem Definida

Seja a função $G : Z^+ \to Z$. Para todos os inteiros $n \geq 1$:

$$G(n) = \begin{cases} 1, & \text{se } n{=}1 \\ 1 + G(\frac{n}{2}), & \text{se } n \text{ é par} \\ G(3n-1), & \text{se } n \text{ é ímpar e } n > 1 \end{cases}$$

$$\begin{aligned} G(1) &= 1 \\ G(2) &= 1 + G(1) = 1 + 1 = 2 \\ G(3) &= G(8) = 1 + G(4) = 1 + (1 + G(2)) \\ &= 1 + (1 + 2) = 4 \\ G(4) &= 1 + G(2) = 1 + 2 = 3 \\ G(5) &= G(14) = 1 + G(7) = 1 + G(20) \\ &= 1 + (1 + (1 + G(5))) = 3 + G(5) \end{aligned}$$

Função Não Bem Definida

Seja a função $G : Z^{+} \rightarrow Z$. Para todos os inteiros $n \geq 1$:

$$G(n) = \begin{cases} 1, & \text{se } n\text{=1} \\ 1 + G(\frac{n}{2}), & \text{se } n \text{ é par} \\ G(3n-1), & \text{se } n \text{ é ímpar e } n > 1 \end{cases}$$

$$\begin{aligned} G(1) &= 1 \\ G(2) &= 1 + G(1) = 1 + 1 = 2 \\ G(3) &= G(8) = 1 + G(4) = 1 + (1 + G(2)) \\ &= 1 + (1 + 2) = 4 \\ G(4) &= 1 + G(2) = 1 + 2 = 3 \\ G(5) &= G(14) = 1 + G(7) = 1 + G(20) \\ &= 1 + (1 + (1 + G(5))) = 3 + G(5) \end{aligned}$$

Função Que Não Se Sabe Ser Bem Definida

Seja a função $H : Z^+ \rightarrow Z$. Para todos os inteiros $n \geq 1$:

$$H(n) = \begin{cases} 1, & \text{se } n\text{=1} \\ H(\frac{n}{2}), & \text{se } n \text{ é par} \\ H(3n+1), & \text{se } n \text{ é ímpar} \end{cases}$$

Porquê Não Sabemos?

Sabemos apenas que a função H é computável para todos os inteiros $n, 1 \leq n \leq 10^9$.

Comentários Finais

A recursividade é uma técnica natural para expressar algoritmos definidos em termos de recorrências.

Estes algoritmos são geralmente traduções de equações de recorrência em uma determinada linguagem de programação.

Deve ser analisada a eficiência dos algoritmos recursivos, principalmente o fator de ramificação e o crescimento da pilha de execução.

Também deve ser analisada a função de recorrência relacionada ao problema – ela pode não ser bem definida.

Em boa parte dos casos, a recursividade pura é utilizada mais como técnica conceitual do que como técnica computacional.

Exercícios

1. Implemente uma função recursiva para calcular 2^n.
2. Implemente uma função recursiva para calcular o MDC de dois números.

Dúvidas?