

Princípios de Desenvolvimento de Algoritmos MAC122

Prof. Dr. Paulo Miranda
IME-USP

Funções Recursivas – parte B

Recursão

- Uma rotina é *recursiva* quando a sua definição envolve uma chamada a ela mesma.
- Neste sentido, o termo recursão é equivalente ao termo indução utilizado por matemáticos.

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

$$\text{fibo}(0) = 0$$

$$\text{fibo}(1) = 1$$

$$\text{fibo}(n) = \text{fibo}(n-1) + \text{fibo}(n-2) \quad \text{se } n>1$$

Exemplo

- Cálculo da função fatorial.

```
/* Fatorial com recursão: */  
  
int fatorial_rec(int n) {  
    if(n==0)          /* Caso trivial:  */  
        return 1;    /* Solução direta */  
    else              /*  $n! = n \cdot (n-1)!$  */  
        return (n*fatorial_rec(n-1));  
}
```

```
/* Fatorial com repetição: */  
  
int fatorial(int n) {  
    int i,fat;  
  
    fat = 1;  
    for(i=1; i<=n; i++)  
        fat *= i;  
  
    return fat;  
}
```

Exemplo

- Cálculo da série de Fibonacci.

```
/* Versão recursiva: */
```

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

```
/* Versão não recursiva: */
```

```
int fibo(int n) {  
    int f0,f1,f2,k;  
    f0 = 0;  
    f1 = 1;  
    for(k=1; k<=n; k++) {  
        f2 = f0+f1;  
        f0 = f1;  
        f1 = f2;  
    }  
    return f0;  
}
```

Exemplo

- Análise de eficiência das funções *Fibonacci* calculando o número de operações de soma $S(n)$ realizadas:

- Versão recursiva

$$S(n) = \begin{cases} 0 & \text{se } n \leq 1 \\ S(n-1) + S(n-2) + 1 & \text{se } n > 1 \end{cases}$$

$$S(n) = \text{fibo}(n+1) - 1$$

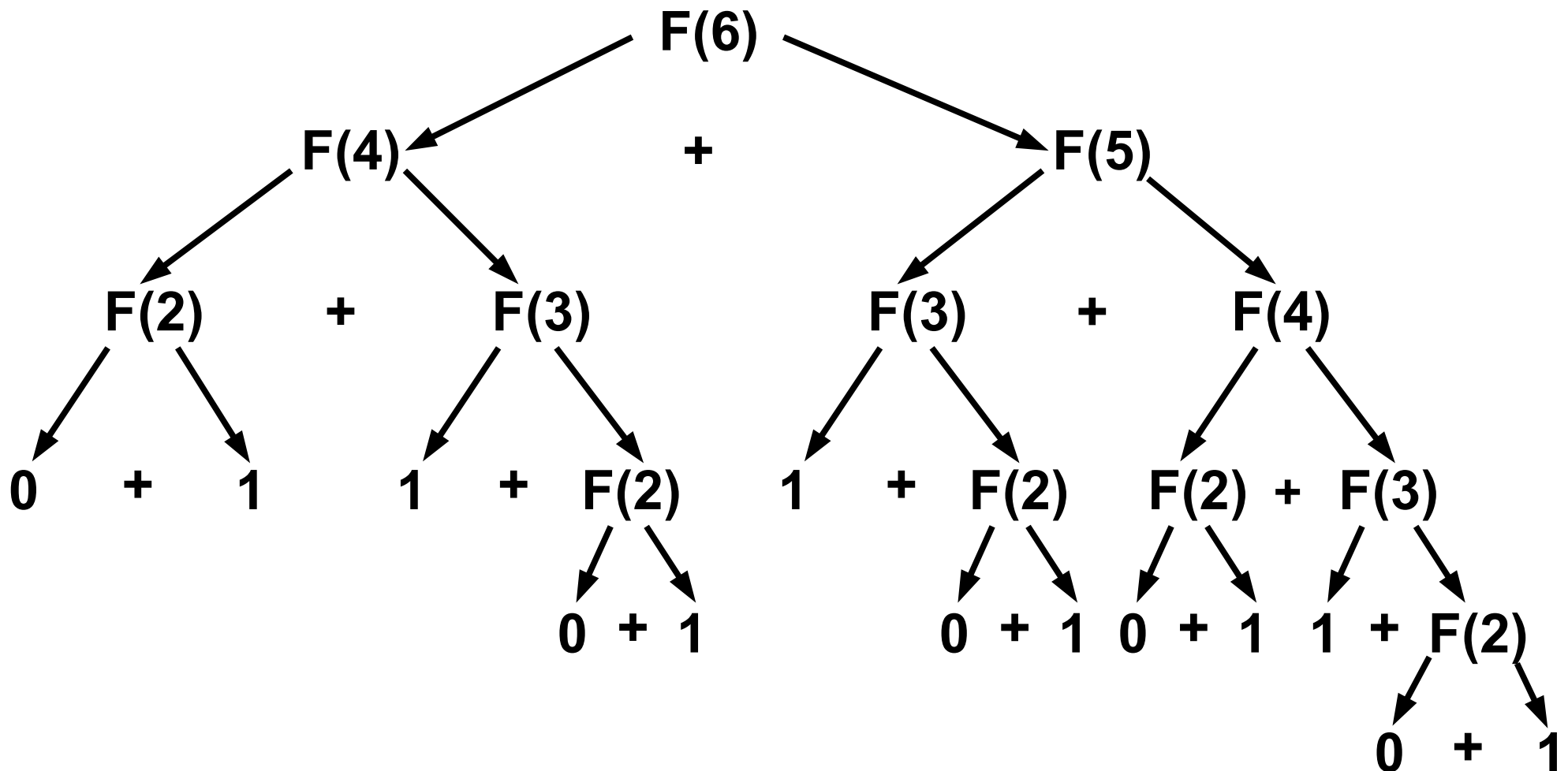
$$S(n) \approx 1.6^n / 1.4$$

- Versão não recursiva - $S(n) = n$

- Neste caso não faz sentido utilizar a versão recursiva.

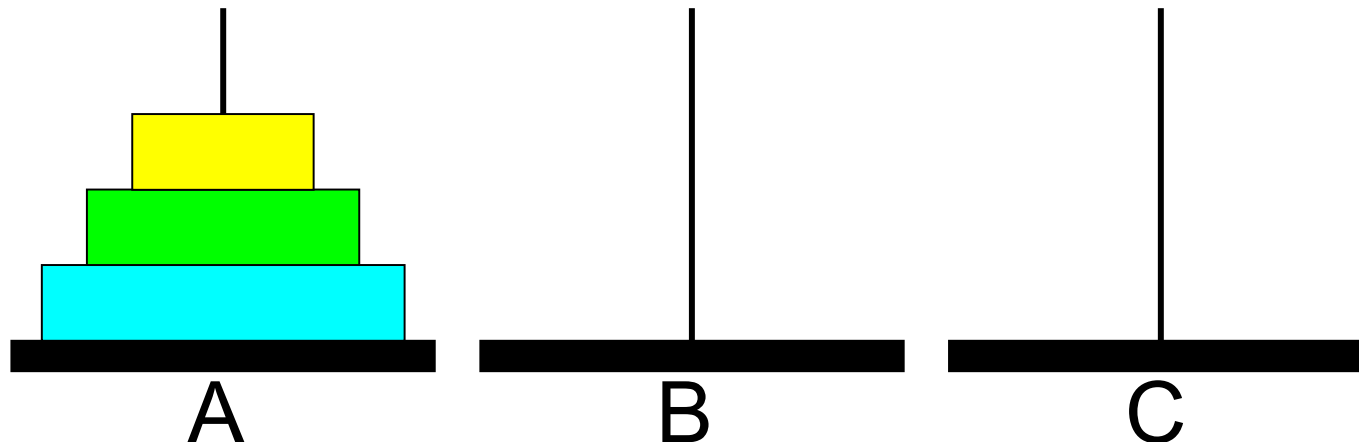
Árvore de Recorrência

- A solução recursiva é ineficiente, pois recalcula várias vezes a solução para valores intermediários:
 - para **$F(6)=\text{fib}(6) = 8$** , temos $2 \times F(4)$, $3 \times F(3)$, $5 \times F(2)$.



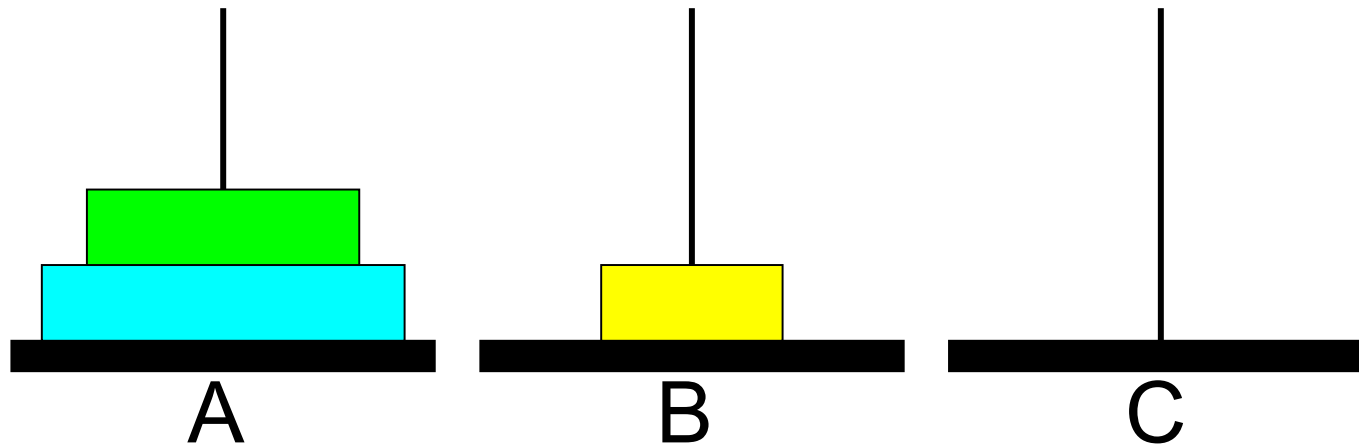
Exemplo

- Torres de Hanoi.
 - São dados um conjunto de N discos com diferentes tamanhos e três bases A, B e C.
 - O problema consiste em imprimir os passos necessários para transferir os discos da base A para a base B, usando a base C como auxiliar, nunca colocando discos maiores sobre menores.



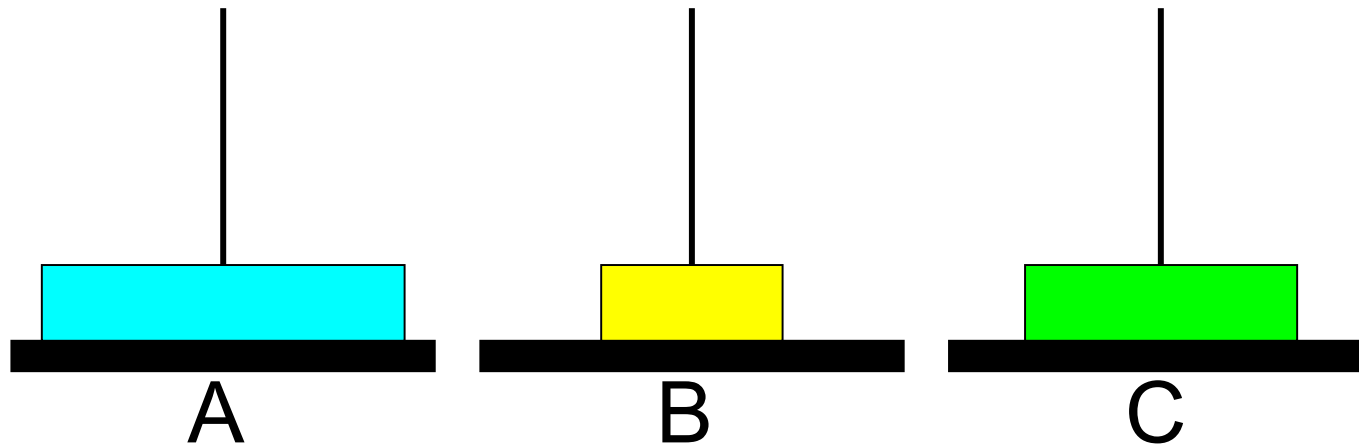
Exemplo

- Torres de Hanoi.
 - 1º passo: Mover de A para B.



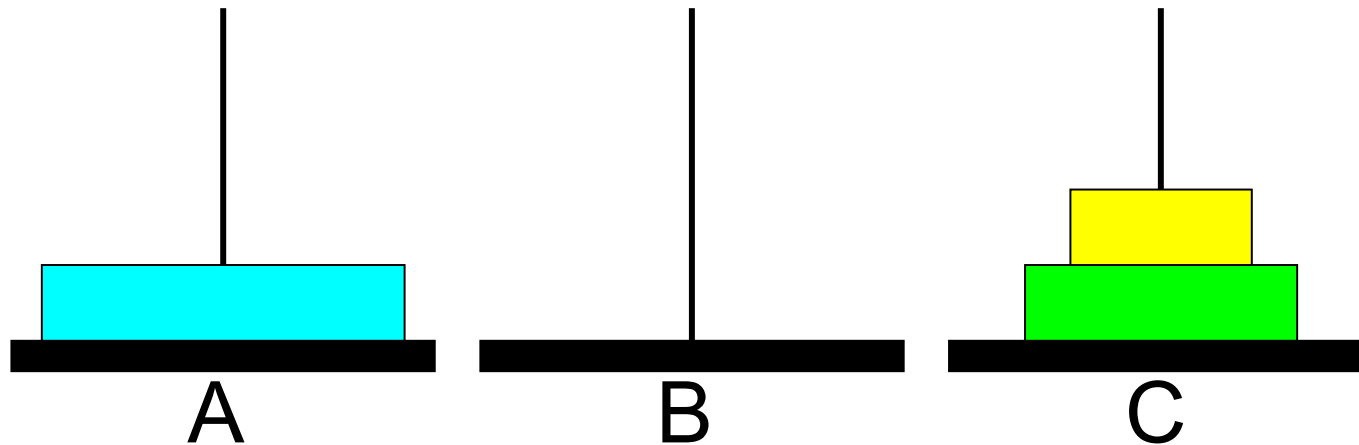
Exemplo

- Torres de Hanoi.
 - 2º passo: Mover de A para C.



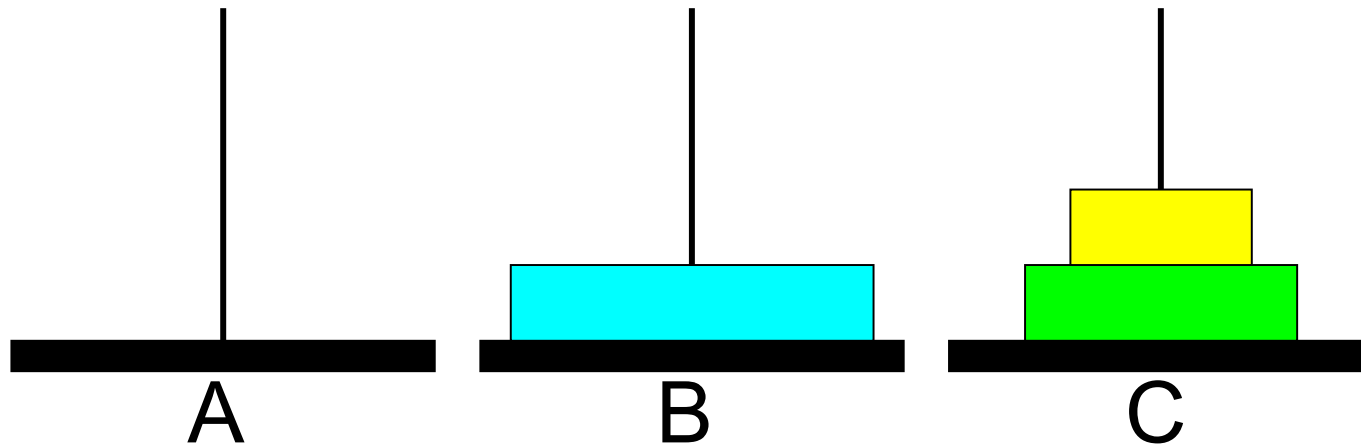
Exemplo

- Torres de Hanoi.
 - 3º passo: Mover de B para C.



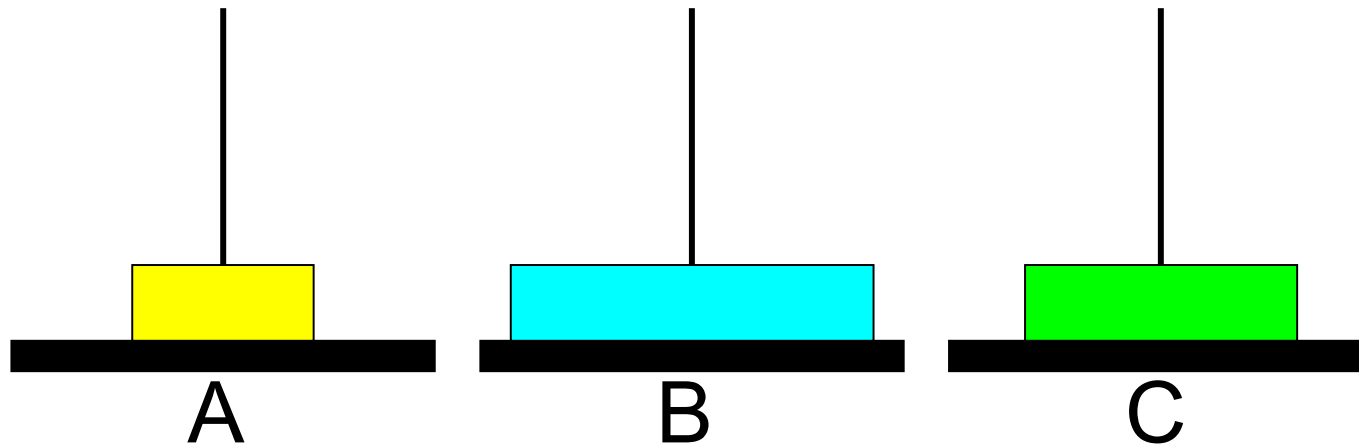
Exemplo

- Torres de Hanoi.
 - 4º passo: Mover de A para B.



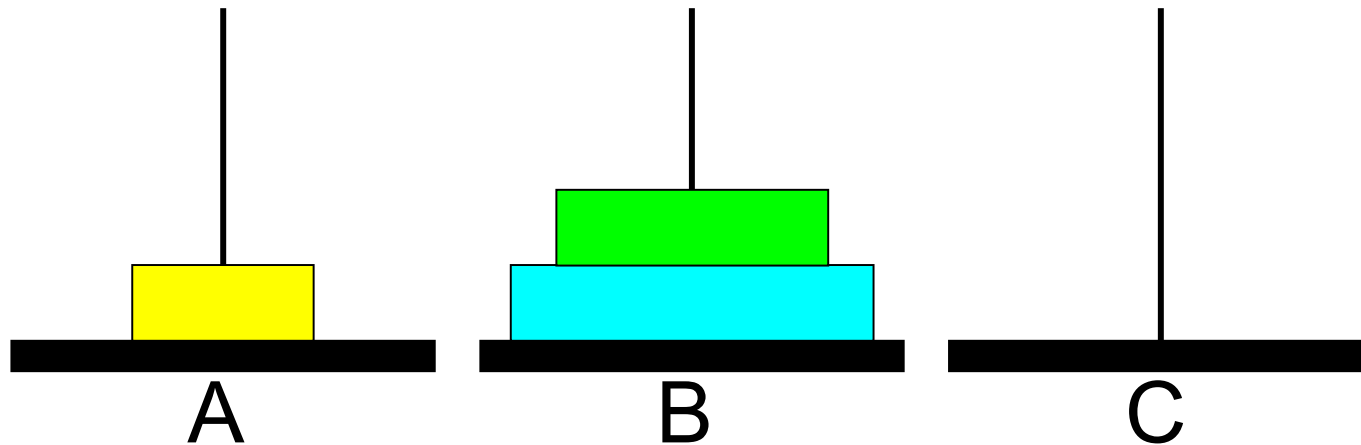
Exemplo

- Torres de Hanoi.
 - 5º passo: Mover de C para A.



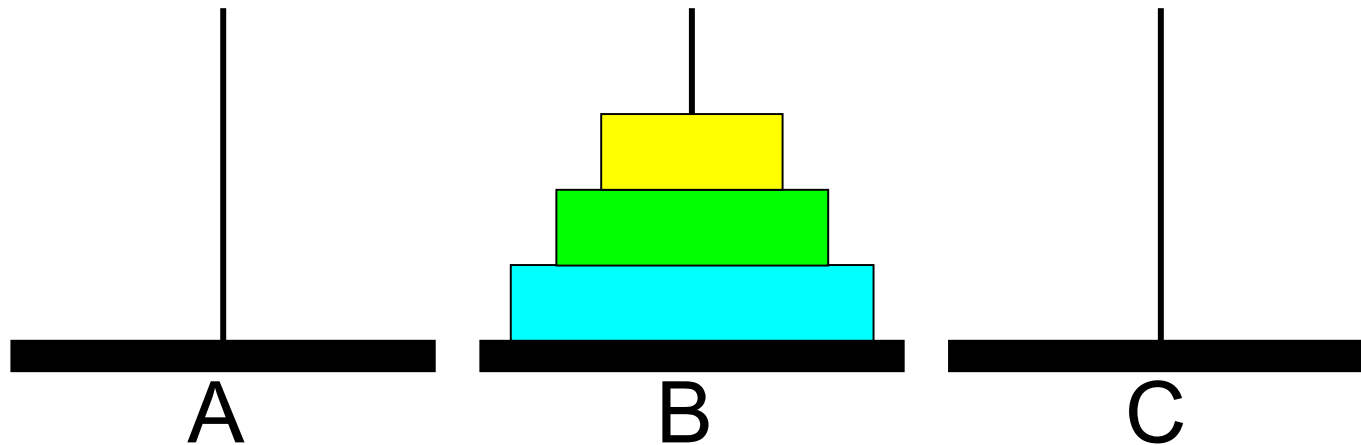
Exemplo

- Torres de Hanoi.
 - 6º passo: Mover de C para B.



Exemplo

- Torres de Hanoi.
 - 7º passo: Mover de A para B.



Exemplo

- Torres de Hanoi.

```
/* Versão recursiva: */

void TorresDeHanoi(char orig,
                  char dest,
                  char aux,
                  int n) {
    if (n > 0) {
        TorresDeHanoi(orig, aux, dest, n - 1);
        printf("Mova de %c para %c\n", orig, dest);
        TorresDeHanoi(aux, dest, orig, n - 1);
    }
}
```

```
int main() {
    TorresDeHanoi('A', 'B', 'C', 3);

    return 0;
}
```

Exemplo

- Torres de Hanoi.

```
/* Versão recursiva: */

void TorresDeHanoi(char orig,
                  char dest,
                  char aux,
                  int n) {

    if (n > 0) {
        TorresDeHanoi(orig, aux, dest, n - 1);
        printf("Mova de %c para %c\n", orig, dest);
        TorresDeHanoi(aux, dest, orig, n - 1);
    }
}
```

```
int main() {
    TorresDeHanoi('A', 'B', 'C', 3);

    return 0;
}
```

O número de movimentos necessários para mover n discos é $2^n - 1$.

Tipos de Recursão

- Existem três tipos:

1) Recursividade direta:

- A função tem uma chamada explícita a si própria.

2) Recursividade indireta (ou mútua):

- Duas ou mais rotinas dependem mutuamente uma da outra (ex: função A chama a função B, que por sua vez chama a função A).

3) Recursividade em cauda (tail):

- A chamada recursiva é a última instrução a ser executada (não existem operações pendentes).

Recursão em cauda

- Cálculo da função fatorial.

```
/* Fatorial com recursão tradicional: */  
  
int fatorial_rec(int n) {  
    if(n==0)      /* Caso trivial */  
        return 1; /* Solução direta */  
    else          /*  $n! = n \cdot (n-1)!$  */  
        return (n*fatorial_rec(n-1));  
}
```

```
/* Fatorial com recursão em cauda: */  
  
/* ac é um acumulador. */  
int fatorial_caudaAux(int n, int ac) {  
    if(n==0) return ac;  
    return fatorial_caudaAux(n-1, n*ac);  
}  
  
int fatorial_cauda(int n) {  
    return fatorial_caudaAux(n, 1);  
}
```

Recursão em cauda

- Cálculo da função fatorial

```
/* Fatorial
```

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

```
/* Fatorial
```

```
/* ac é um acumulador */
```

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

```
int fatorial_cauda(int n){  
    return fatorial_caudaAux(n, 1);  
}
```

```
int fatorial_caudaAux(int n, int ac){  
    inicio:  
    if(n==0) return ac;  
    else{  
        ac *= n;  
        n--;  
        goto inicio;  
    }  
}
```

Pilha explícita

- Sempre é possível eliminar o uso da recursão e substituí-la pelo uso de uma **pilha explícita**.
 - As funções obtidas ficam mais complexas,
 - porém, em geral, são mais eficientes.
 - Evita chamadas e retornos,
 - Evita criação de registros de ativação,
 - Permitem código mais customizado.

Pilha explícita

- Torres de Hanoi.

```
/* Versão recursiva: */

void TorresDeHanoi(char orig,
                   char dest,
                   char aux,
                   int n) {
    if (n > 0) {
        TorresDeHanoi(orig, aux, dest, n - 1);
        printf("Mova de %c para %c\n", orig, dest);
        TorresDeHanoi(aux, dest, orig, n - 1);
    }
}
```

Pilha explícita


- Torres de Hanoi com pilha explícita.
 - Definição da estrutura de pilha utilizada.

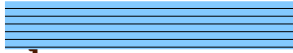
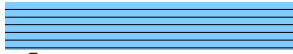
```
typedef enum chamadas {chamada1,chamada2} chamada;  
  
typedef struct _elemPilha{  
    chamada ch;  
    char    orig;  
    char    dest;  
    char    aux;  
    int     n;  
} ElemPilha;  
  
typedef enum acoes {entrada,retorno,saida} acao;
```

Pilha explícita

```
void TorresDeHanoi_pilha(char orig,
                          char dest,
                          char aux,
                          int n){

    Pilha p = CriaPilha();
    acao a = entrada;
    ElemPilha S;
    char tmp;

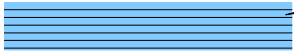
    do{
        switch(a){
            case entrada:
                if(n==0){
                    a = retorno;
                }
                else{

                }
                break;
            case retorno:
                if(PilhaVazia(p))
                    a = saida;
```

```
        else{
            S = Desempilha(p);
            orig = S.orig;
            dest = S.dest;
            aux = S.aux;
            n = S.n;
            switch(S.ch){
                case chamada1:

                break;
                case chamada2:

                break;
            }
        }
        break;
        case saida:
            break;
    }
    while(a!=saida);
    LiberaPilha(p);
}
```

Pilha explícita

```
void TorresDeHanoi_pilha
```

```
Pilha p = CriaPilha()  
acao a = entrada;  
ElemPilha S;  
char tmp;
```

```
do{  
    switch(a){  
        case entrada:  
            if(n==0){  
                a = retorno;  
            }  
            else{  
                  
            }  
            break;  
        case retorno:  
            if(PilhaVazia(p))  
                a = saida;
```

```
S.ch    = chamada1;  
S.orig  = orig;  
S.dest  = dest;  
S.aux   = aux;  
S.n     = n;  
Empilha(p, S);
```

```
/* TorresHanoi(orig,aux,dest,n-1); */  
tmp  = dest;  
dest = aux;  
aux  = tmp;  
n--;
```

```
        break;  
        case saida:  
            break;  
    }  
}while(a!=saida);  
LiberaPilha(p);  
}
```


Pilha explícita

```
void TorresDeHanoi_pilha(char orig,  
                           char dest,  
                           char aux,  
                           int n) {
```

```
Pilha p = CriaPilha();
acao a = entrada;
ElemPilha S;
char tmp;
```

```
do{
    switch (a) {
        case entrada:
```

```
printf("Mova de %c para %c\n",orig,dest);
/* TorresHanoi(aux,dest,orig,n-1); */
tmp  = orig;
orig = aux;
aux  = tmp;
n--;
a = entrada;
```

```
a = saída;
```

```
else{
    S = Desempilha(p);
    orig = S.orig;
    dest = S.dest;
    aux   = S.aux;
    n      = S.n;
    switch(S.ch){
    case chamada1:
```

```
break;
case chamada2:
    // ...
break;
```

aida:

```
!=saida) ;  
lha (p) ;
```

1

Pilha explícita

```
void TorresDeHanoi_pilha(char orig,
                          char dest,
                          char aux,
                          int n){

    Pilha p = CriaPilha();
    acao a = entrada;
    ElemPilha S;
    char tmp;

    do{
        switch(a){
        case entrada:
            if(n==0){
                a = retorno;
            }
            else /* não existem operações pendentes.*/
                break;
        case retorno:
            if(PilhaVazia(p))
                a = saida;
```

```
        else{
            S = Desempilha(p);
            orig = S.orig;
            dest = S.dest;
            aux = S.aux;
            n = S.n;
            switch(S.ch){
            case chamada1:
                break;
            case chamada2:
                break;
            }
        }while(a!=saida);
        LiberaPilha(p);
    }
```

Recursão indireta

- Transformação da notação infixada para pós-fixada.

```
int main() {  
    char expr[]="a* (b+c) * (d-g) *h";  
  
    In2Pos(expr) ;  
  
    return 0 ;  
}
```

Recursão indireta

- Transformação da notação infixa para pós-fixa.

```
typedef enum boolean {false,true} bool;

void Erro() {
    printf("Erro\n");
    exit(-1);
}

void In2Pos(char *expr) {
    int i;
    i = 0;
    Expressao(expr, &i);
    printf("\n");

    if(expr[i]!='\0') Erro();
}
```

Recursão indireta

- Transformação da notação infixa para pós-fixa.

```
void Expressao(char *expr, int *i){
    char op;
    bool fim=false;

    Termo(expr, i);
    do{
        op = expr[*i];
        if(op=='+' || op=='-'){
            (*i)++;
            Termo(expr, i);
            printf("%c",op);
        }
        else fim = true;
    }while(!fim);
}
```

Recursão indireta

- Transformação da notação infixa para pós-fixa.

```
void Termo(char *expr, int *i){
    char op;
    bool fim=false;

    Fator(expr, i);
    do{
        op = expr[*i];
        if(op=='*' || op=='/'){
            (*i)++;
            Fator(expr, i);
            printf("%c", op);
        }
        else fim = true;
    }while(!fim);
}
```

Recursão indireta

- Transformação da notação infixa para pós-fixa.

```
void Fator(char *expr, int *i) {
    char c;

    c = expr[*i];

    if(c>='a' && c<='z') {
        printf("%c", c);
        (*i)++;
    }
    else if(c=='(') {
        (*i)++;
        Expressao(expr, i);
        if(expr[*i]==')')
            (*i)++;
        else Erro();
    }
    else Erro();
}
```