Princípios de Desenvolvimento de Algoritmos
MAC122

Prof. Dr. Paulo Miranda
IME-USP

Funções Recursivas

Funções Recursivas

- **Definição:**
  - Uma função é dita **recursiva** quando dentro dela é feita uma ou mais chamadas a ela mesma.
  - A ideia é dividir um problema original um subproblemas menores de mesma natureza **(divisão)** e depois combinar as soluções obtidas para gerar a solução do problema original de tamanho maior **(conquista)**.
  - Os subproblemas são resolvidos recursivamente do mesmo modo em função de instâncias menores, até se tornarem problemas triviais que são resolvidos de forma direta, interrompendo a recursão.

Funções Recursivas

- O termo recursão é equivalente ao termo indução utilizado por matemáticos.

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

$$fibo(0) = 0$$
$$fibo(1) = 1$$
$$fibo(n) = fibo(n-1) + fibo(n-2) \quad \text{se } n>1$$

Funções Recursivas

- **Exemplo: Calcular o fatorial de um número.**
  - **Solução não recursiva**

```c
#include <stdio.h>

float fatorial(int n){
  float fat = 1.0;
  while(n>1){
    fat *= n;
    n--;
  }
  return fat;
}
int main(){
  float fat;
  fat = fatorial(6);
  printf("fatorial: %f\n",fat);
  return 0;
}
```

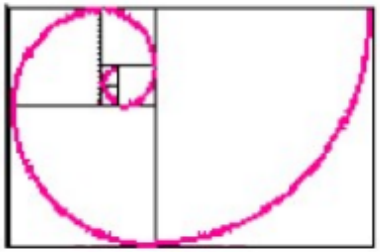
Funções Recursivas

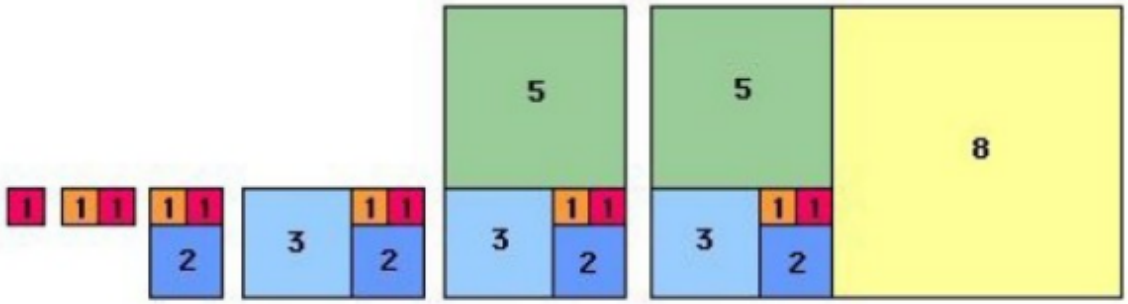
- **Exemplo: Calcular o fatorial de um número.**
  - **Solução recursiva: n! = n.(n-1)!**

```c
#include <stdio.h>

float fatorial(int n){
  if(n==0)       /* Caso trivial */
    return 1.0; /* Solução direta */

  return n*fatorial(n-1); /* Chamada recursiva */
}


int main(){
  float fat;
  fat = fatorial(6);
  printf("fatorial: %f\n",fat);
  return 0;
}
```

Funções Recursivas

- Cálculo da série de Fibonacci.





Funções Recursivas

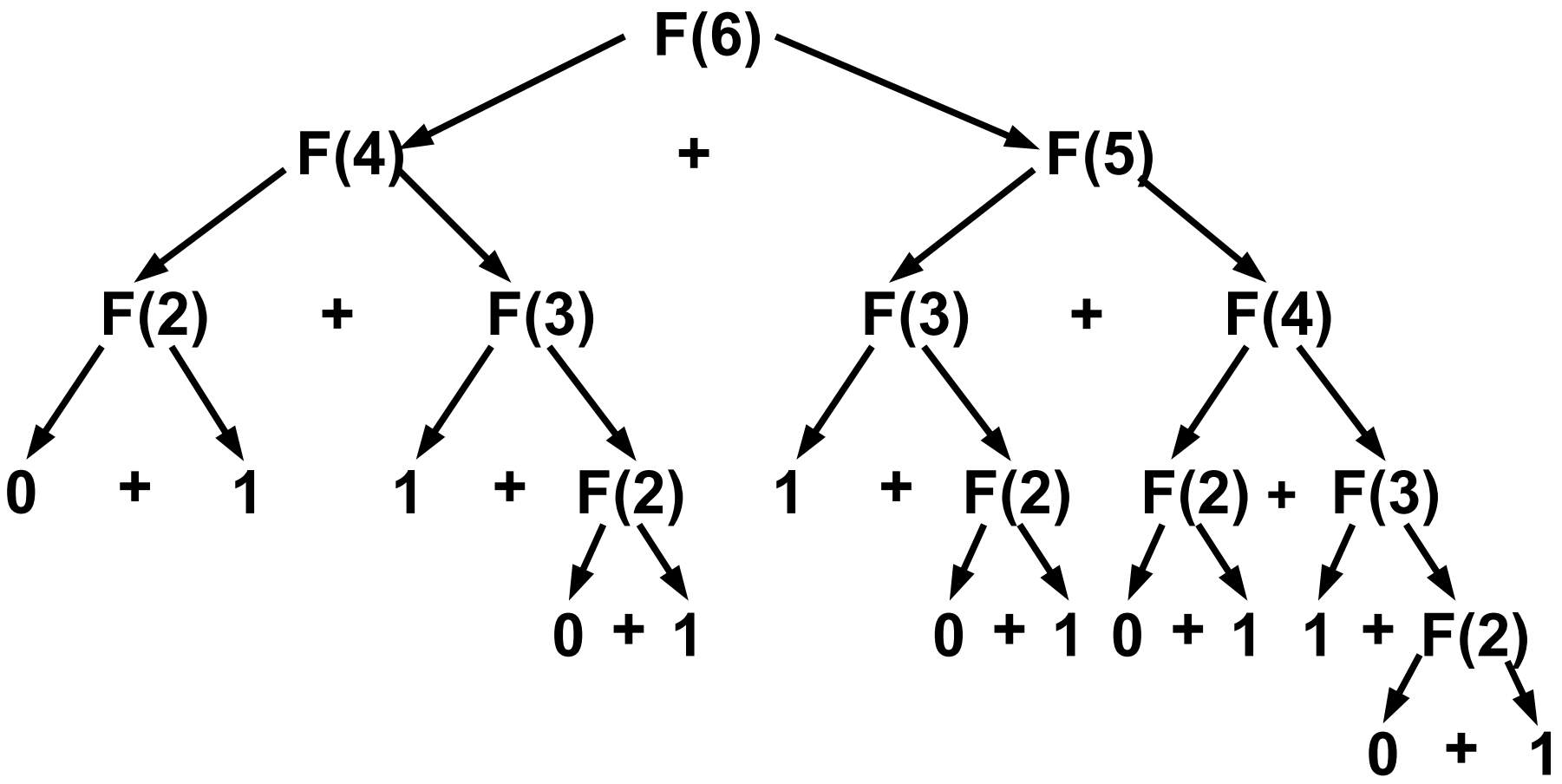
- Cálculo da série de Fibonacci.

```
/* 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;
}
```

Árvore de Recorrência

- A solução recursiva é ineficiente, pois recalcula várias vezes a solução para valores intermediários:
  - para **F(6)=fibo(6) = 8**, temos 2x**F(4)**, 3x**F(3)**, 5x**F(2)**.



Funções Recursivas

- **Exemplo: Calcular x elevado a n positivo.**
  - **Solução não recursiva**

```
#include <stdio.h>


float potencia(float x, int n){
  float pot=1.0;
  while(n>0){
    pot *= x;
    n--;
  }
  return pot;
}
```

Funções Recursivas

- **Exemplo: Calcular x elevado a n positivo.**
  - **Solução recursiva: x^n = x . x^(n-1)**

```
#include <stdio.h>


float potencia(float x, int n){
  if(n==0)          /* Caso trivial */
    return 1.0;     /* Solução direta */
  else
    return x*potencia(x, n-1); /*Chamada recursiva*/
}
```

Funções Recursivas

- **Exemplo: Calcular x elevado a n positivo.**
  - **Solução recursiva: x^n = x^(n/2). x^(n/2) = (x^(n/2))^2**

```
#include <stdio.h>

/* Função recursiva mais eficiente */
float potencia(float x, int n){
  float pot;

  if(n==0) return 1.0; /* Caso trivial */
  if(n%2==0){ /* Se n é par... */
    pot = potencia(x, n/2);
    return pot*pot;
  }
  else{ /* Se n é ímpar... */
    pot = potencia(x, n/2);
    return pot*pot*x;
  }
}
```

Funções Recursivas

- **Exemplo: Encontrar maior elemento de um vetor.**
  - **Solução recursiva**

```
#include <stdio.h>
int maiorinteiro(int v[], int n){
  int m;
  if(n==1) return v[0]; /* Caso trivial */
  else{
    m = maiorinteiro(v, n-1);
    if(m>v[n-1]) return m;
    else         return v[n-1];
  }
}
int main(){
  int max,v[5]={8,1,9,4,2};
  max = maiorinteiro(v, 5);
  printf("Max: %d\n",max);
  return 0;
}
```

Funções Recursivas

- **Exemplo: Imprimir elementos de um vetor.**
  - **Solução não recursiva**

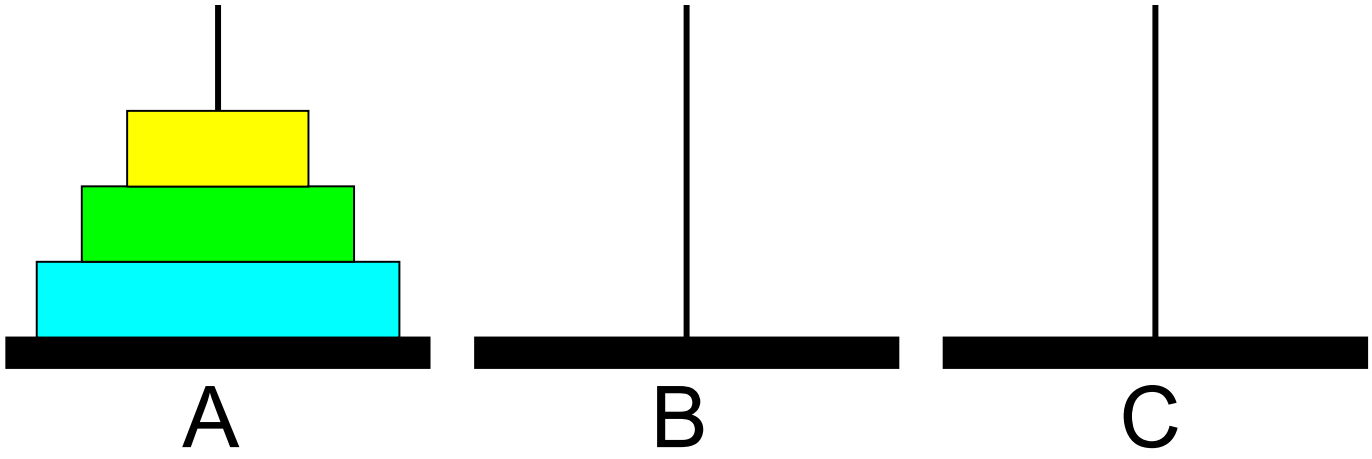
```c
#include <stdio.h>

void printvetor(int v[], int n){
  int i;
  for(i=0; i<n; i++)
    printf("%d ",v[i]);
}
```

  - **Solução recursiva**

```c
#include <stdio.h>

void printvetor(int v[], int n){
  if(n>1)
    printvetor(v, n-1);
  printf("%d ",v[n-1]);
}
```

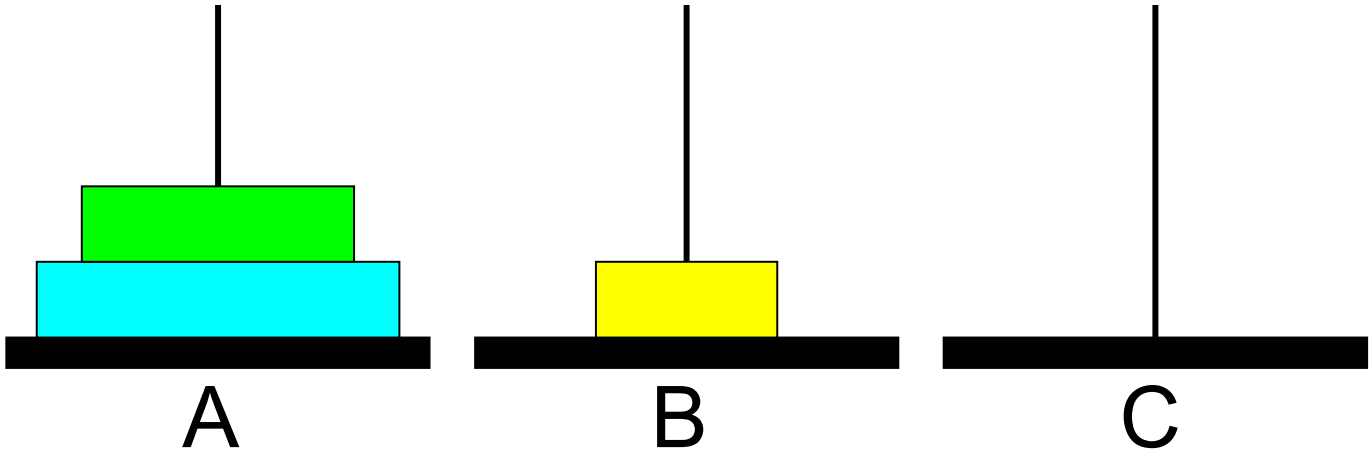
Funções Recursivas

- **Exemplo: Torre 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.



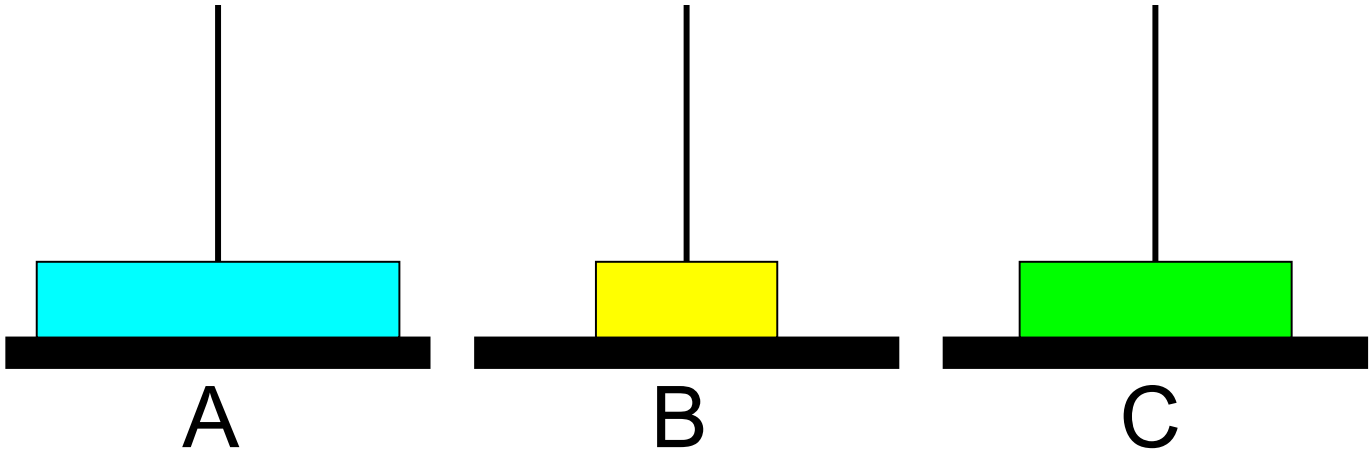
Funções Recursivas

- **Exemplo: Torre de Hanoi**
  - **<u>1° passo:</u>** Mover de A para B.



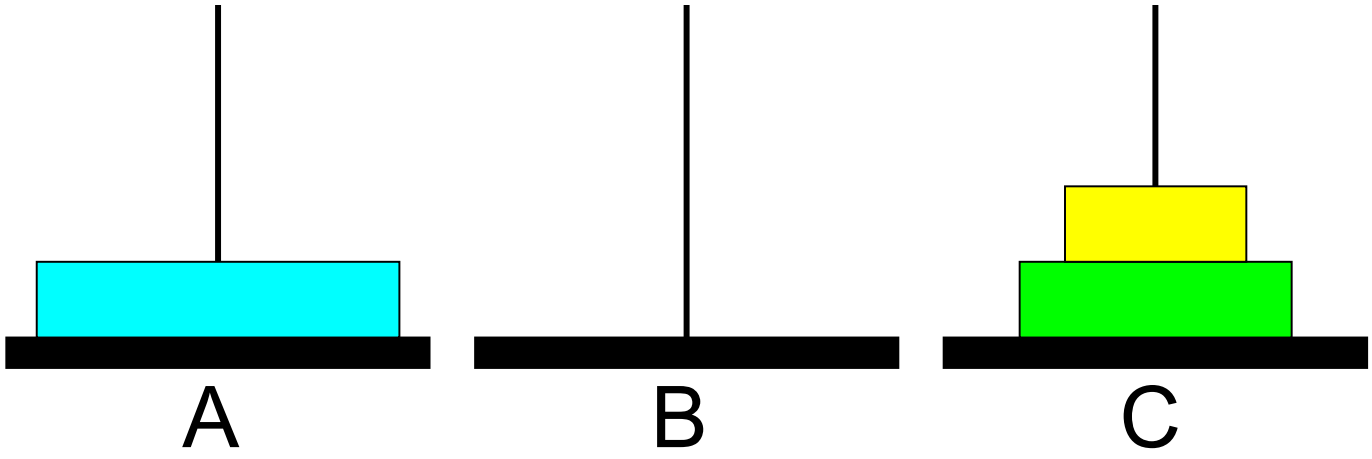
Funções Recursivas

- **Exemplo: Torre de Hanoi**
  - **2° passo:** Mover de A para C.



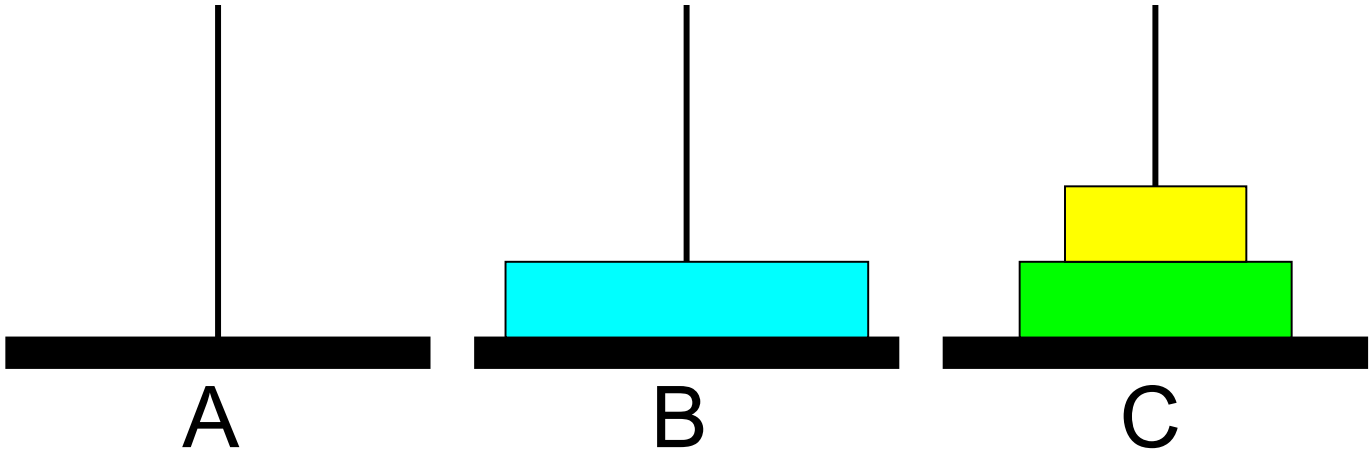
Funções Recursivas

- **Exemplo: Torre de Hanoi**
  - **3° passo:** Mover de B para C.



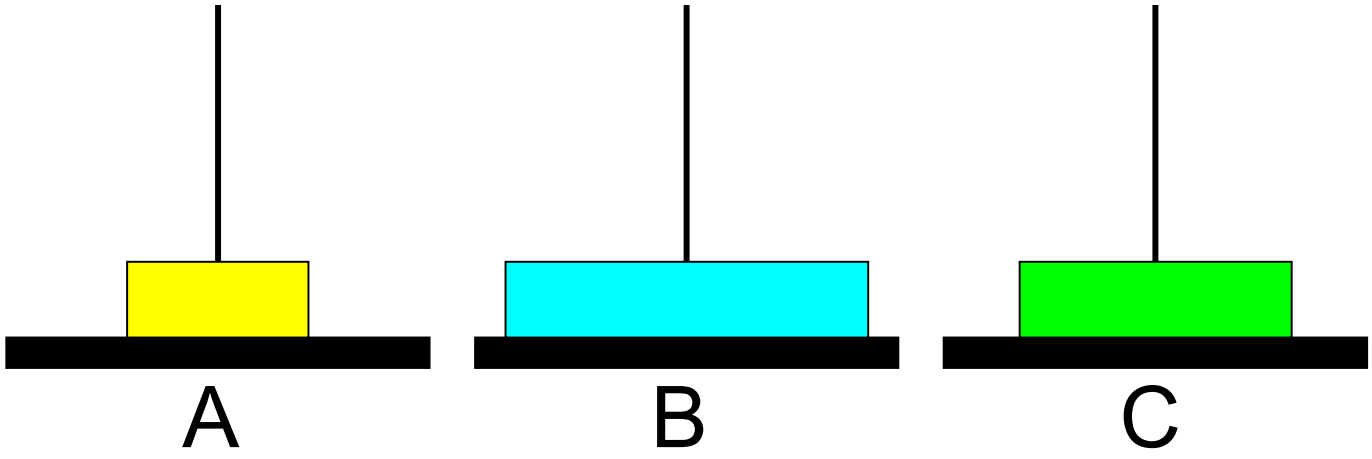
Funções Recursivas

- **Exemplo: Torre de Hanoi**
  - **4° passo:** Mover de A para B.



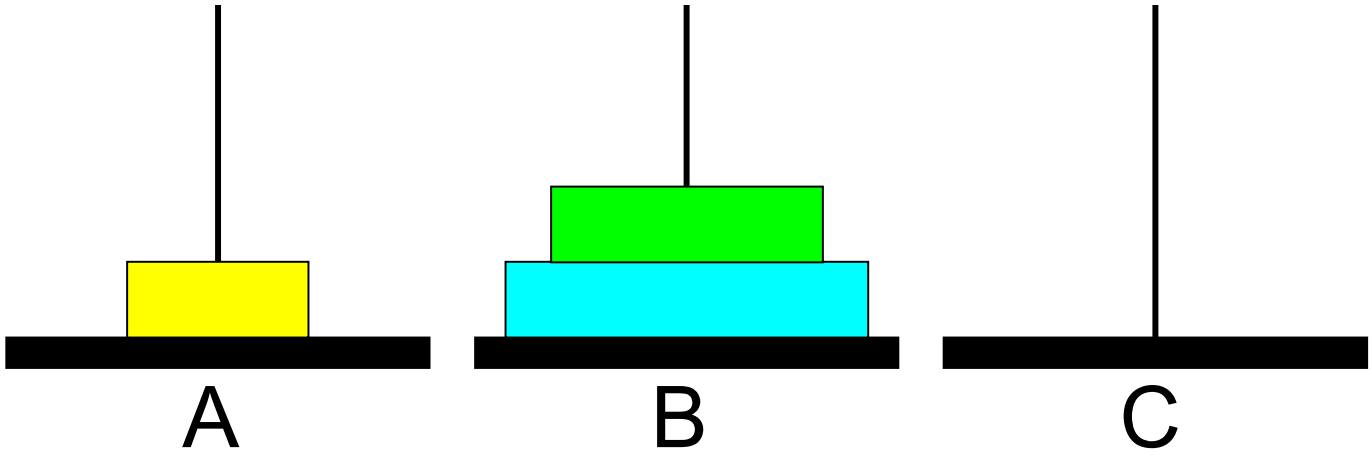
Funções Recursivas

- **Exemplo: Torre de Hanoi**
  - **5° passo:** Mover de C para A.



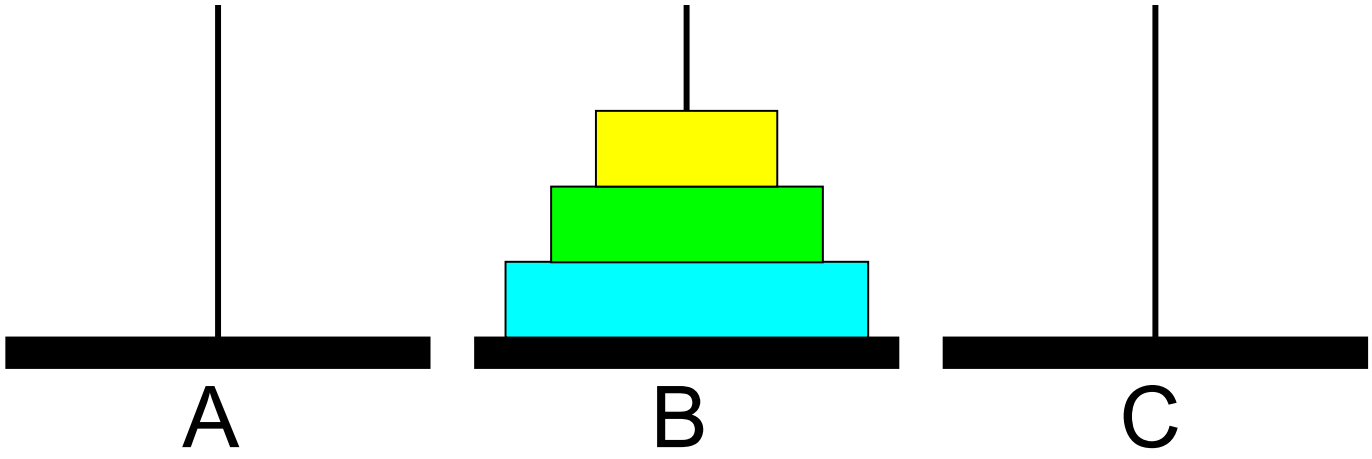
Funções Recursivas

- **Exemplo: Torre de Hanoi**
  - **6° passo:** Mover de C para B.



Funções Recursivas

- **Exemplo: Torre de Hanoi**
  - **7° passo:** Mover de A para B.



Funções Recursivas

- **Exemplo: Torre de Hanoi**

```c
#include <stdio.h>
void hanoi(char orig, char dest, char aux, int n){
  if(n == 1)
    printf("Move de %c para %c.\n", orig, dest);
  else{
    hanoi(orig, aux, dest, n-1);
    printf("Move de %c para %c.\n", orig, dest);
    hanoi(aux, dest, orig, n-1);
  }
}
int main(){
  int n;
  printf("Número de discos: ");
  scanf("%d",&n);
  hanoi('A', 'B', 'C', n);
  return 0;
}
```

Exemplo

- Torres de Hanoi.

```
/* Versão recursiva mais compacta: */

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

```
int main(){
  hanoi('A','B','C',3);
  return 0;
}
```

Exemplo

- Torres de Hanoi.

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

```
/* Versão recursiva mais compa

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

```
int main(){
  hanoi('A','B','C',3);
  return 0;
}
```

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.

```c
/* Fatorial com recursão tradicional: */

int fatorial_rec(int n){
  if(n==0)      /* Caso trivial */
    return 1;   /* Solução direta */
  else    /* n! = n.(n-1)!  */
    return (n*fatorial_rec(n-1));
}
```

```c
/* 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 funçã

```
/* Fatoria

int fatori
  if(n==0)
    return
  else   /
    return
}
```

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

```
/* Fatorial

/* ac é um acumula
int fatorial_caud      (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);
}
```

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 mais compacta: */

void hanoi(char orig,
           char dest,
           char aux,
           int n){
  if(n>0){
    hanoi(orig,aux,dest,n-1);
    printf("Mova de %c para %c\n",orig,dest);
    hanoi(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{
        [redacted]
      }
      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:
             [redacted]
        break;
        case chamada2:
             [redacted]
        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{
        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 retorno:
      if(PilhaVazia(p))
        a = saida;
      }
      }
      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 chamada1:
        printf("Mova de %c para %c\n",orig,dest);
        /* TorresHanoi(aux,dest,orig,n-1); */
        tmp  = orig;
        orig = aux;
        aux  = tmp;
        n--;
        a = entrada;
        break;
      case chamada2:
        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:
            [redacted]
          case chamada2:
            [redacted]
            break;
          }
        }
        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:
      if(n==0){
        /* 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:
          [redacted]
          break;
        case chamada2:
          [redacted]
          break;
        }
        k;
        saida:
      break;
      }
    }while(a!=saida);
    LiberaPilha(p);
}
```

Recursão indireta

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

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

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

Recursão indireta

- Transformação da notação infixa para pós-fixa.

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

   In2Pos(expr);

   return 0;
}
```

Recursão indireta

- Transformação da notação infixa para pós-fixa.

```
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();
}
```