Linguagens de Programação II

Recursividade

Dany Sanchez Dominguez

dsdominguez@gmail.com
Sala 1 – NBCGIB

Definição de Recursividade

- Um objeto é denominado recursivo quando sua definição é parcialmente feita em termos dele mesmo.
- Uma função ou procedimento é dito recursivo (ou apresenta recursividade) se for definido em termos de si próprio.
- Em programação, a recursividade é um mecanismo útil e poderoso que permite a uma função chamar a si mesma direta ou indiretamente.

Exemplo - Fatorial

- Definição matemática ($n!$):

$$n! = \begin{cases} \prod_{i=0}^{n-1} (n-i) = n \times (n-1) \times \cdots \times (2) \times (1), & \text{se } n > 0 \\ 1 \quad , & \text{se } n = 0 \end{cases}$$

- Podemos expressar o fatorial dos primeiros números naturais na forma:

$$0! = 1 \qquad 3! = 3 \times 2 \times 1$$

$$1! = 1 \qquad 4! = 4 \times 3 \times 2 \times 1$$

$$2! = 2 \times 1 \qquad 5! = 5 \times 4 \times 3 \times 2 \times 1$$

Exemplo - Fatorial

- Seguindo o procedimento anterior é impossível listar a fórmula de fatorial para cada número natural (inviável para valores grandes).
- A definição anterior define um conjunto *infinito* com comandos *infinitos*.
- Chamaremos a definição anterior como definição não-recursiva de fatorial.
- A definição não-recursiva pode ser utilizada para criarmos um algoritmo iterativo para o calculo do fatorial.

Exemplo - Fatorial

- Fatorial Iterativo:

```
int fatorial_ite(int);

int fatorial_ite(int n){
  int i, result;

  result = 1;
  for(i=n;i>0;i--)
    result *= i;

  return result;
}
```

- Execução do algoritmo.

Exemplo - Fatorial

- Vamos a examinar detalhadamente a definição de fatorial:

$$0! = 1$$

$$1! = 1 \longrightarrow 1! = 1 \times 0!$$

$$2! = 2 \times 1 \longrightarrow 2! = 2 \times 1!$$

$$3! = 3 \times 2 \times 1 \longrightarrow 3! = 3 \times 2!$$

$$4! = 4 \times 3 \times 2 \times 1 \longrightarrow 4! = 4 \times 3!$$

$$5! = 5 \times 4 \times 3 \times 2 \times 1 \longrightarrow 5! = 5 \times 4!$$

- Conclusão: $n! = n \times (n-1)! \quad \text{para} \quad n > 0$

Exemplo - Fatorial

- Definição recursiva ($n!$):

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

- Usando a definição anterior é possível expressar o fatorial de qualquer número natural,
- A definição anterior é recursiva!!!
- Define uma função em termos dela mesma (aparentemente circular)
- Define um *conjunto infinito* com *comandos finitos*

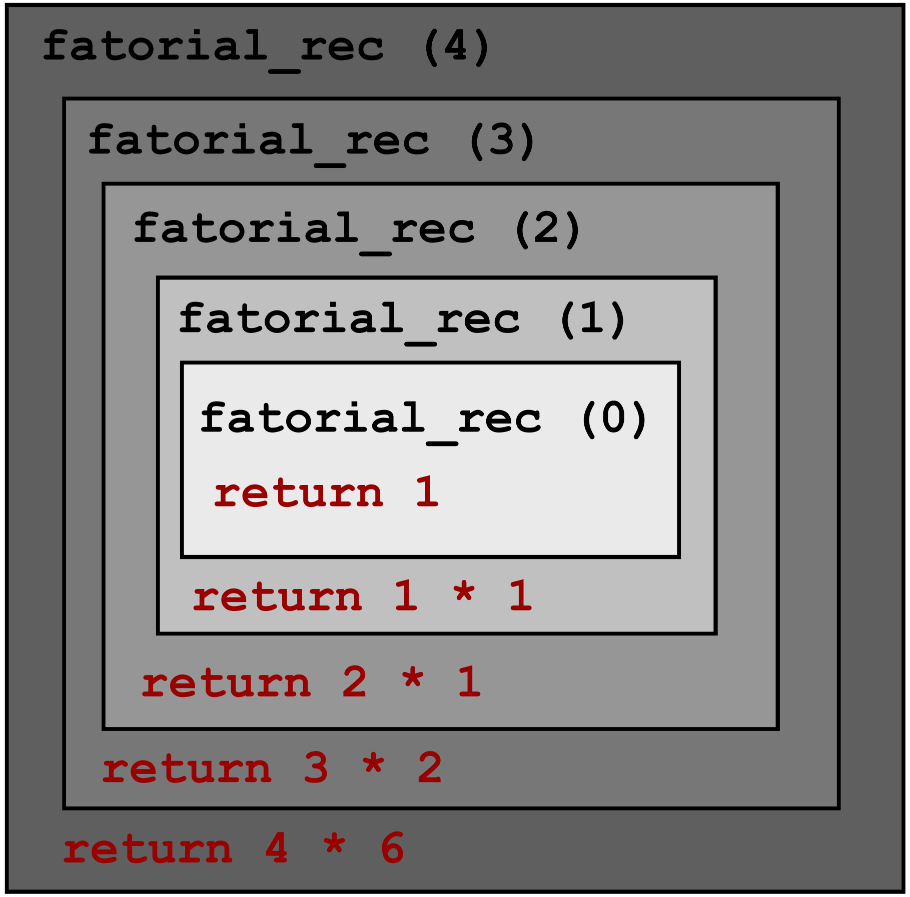
Exemplo - Fatorial

- Usando a definição anterior podemos escrever um algoritmo recursivo para o cálculo do fatorial.
- Fatorial Recursivo:

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

Exemplo - Fatorial

- Execução do algoritmo recursivo para $n = 4$.



Endereço de memória	Células de memória
1024	n=4
1056	n=0
1088	n=2
1120	
1152	n=1
1184	
1216	n=3
1238	

Recursividade

- A recursividade pode ser utilizada quando um problema puder ser definido em termos de si próprio,
- Uma função é dita recursiva se ela contém pelo menos uma chamada explícita ou implícita a si própria,
- Uma função recursiva chama ela mesma, mas com outros parâmetros,
- Algoritmos recursivos são baseados no paradigma de dividir e conquistar,

Recursividade

- As técnicas de dividir e conquistar, preconizam obter a solução de um problema complexo, dividindo este em problemas mais simples,
- A idéia básica de um algoritmo recursivo consiste em diminuir sucessivamente o problema em um (ou vários) problema(s) menor(es) ou mais simples,
- Em cada etapa da recursão o universo do problema é diminuído,

Recursividade

- O problema é simplificado até que o tamanho do mesmo permita resorve-lô de forma direta (sem recorrer a recursividade),
- Quando o problema é resolvido diretamente, é porque a condição de parada foi alcançada,
- A condição de parada ou casso básico permite finalizar o processo recursivo,
- Exemplo de fatorial, casso básico para $n = 0$,

Recursividade

- Procedimentos recursivos, que não atingem a condição de parada, levam a programas infinitos que estouram a memória do computador,
- Um algoritmo recursivo é formado por:
  - condição de parada: o problema é resolvido diretamente (casso básico),
  - outros comandos: resolvem o problema chamando novamente a função recursiva.

Problemas associados a recursão

- Recursões que não terminam!!!
- Um requisito fundamental é que a chamada recursiva esteja sujeita à uma condição booleana que em um determinado momento irá se tornar falsa e não mais ativará a recursão (condição de parada!).
- Um subprograma recursivo deve ter obrigatoriamente uma condição que controla sua execução ou término, sob pena de produzir uma computação interminável.

Ex. – Seqüência de Fibonacci

- Os estudos sobre a seqüência de Fibonacci datam do século XI, e foram feitas por Leonardo de Pisa (Fibonacci=filius Bonacci),

- A mesma tem aplicações teoricas e práticas na medida que muitos padrões da natureza parecem segui-la,

Ex. – Seqüência de Fibonacci

- Entre as principais aplicações temos:
  - Estudo genealógico de coelhos
  - Estudo genealógico de abelhas
  - Comportamento da luz
  - Comportamento de átomos
  - Crescimento de plantas
  - Ascenção e queda em bolsas de valores
  - Probabilidade e Estatística

Ex. – Seqüência de Fibonacci

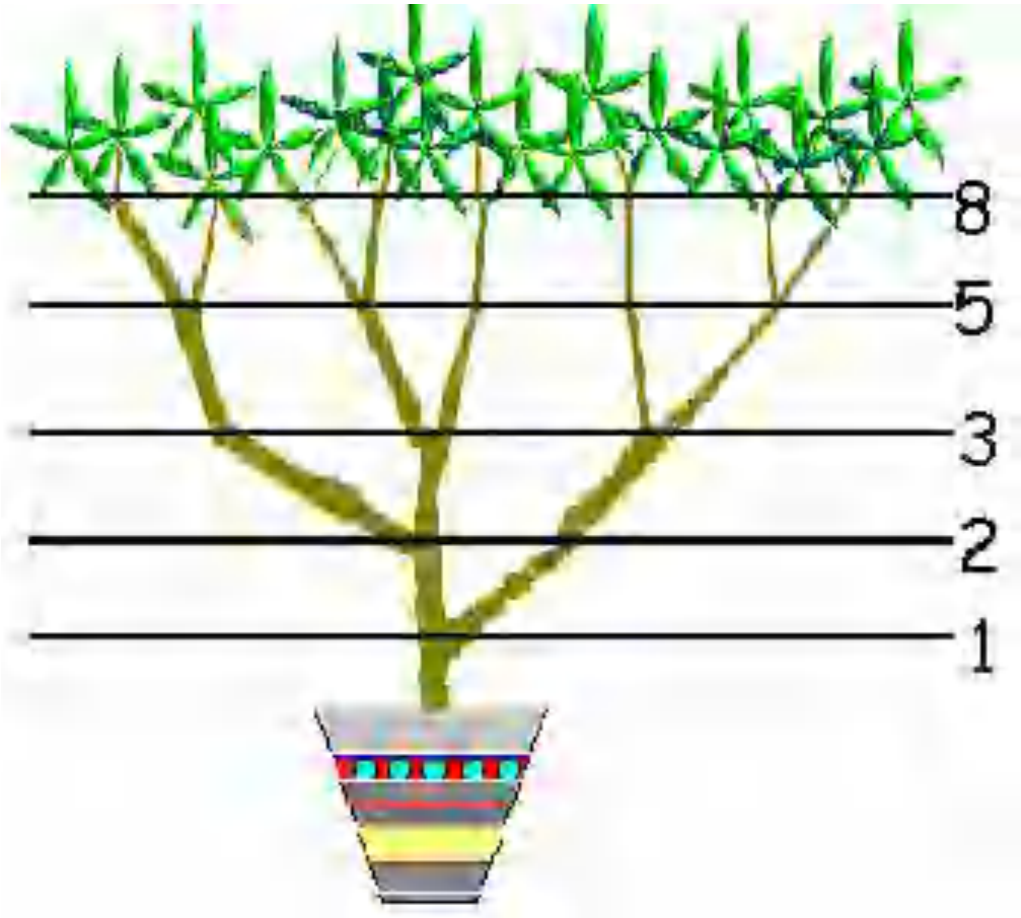
- A seqüência de Fibonacci é definida como:
  1. uma sequência de números inteiros onde cada elemento é a soma dos dois números anteriores,
  2. os dois primeiro elementos são iguais a 1.

- Os primeiros valores da seqüência:

$$\{1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ \cdots\}$$

Ex. – Seqüência de Fibonacci

- Aplicações, crescimento de plantas (nro. galhos):



Ex. – Seqüência de Fibonacci

- Definição recursiva da seqüência de Fibonacci:

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

- A seqüência de Fibonacci é definida recursivamente em função de duas instancias anteriores da própria seqüência,
- porém a condição de parada contempla dois casos bases, $n=1$ e $n=2$.

Ex. – Seqüência de Fibonacci

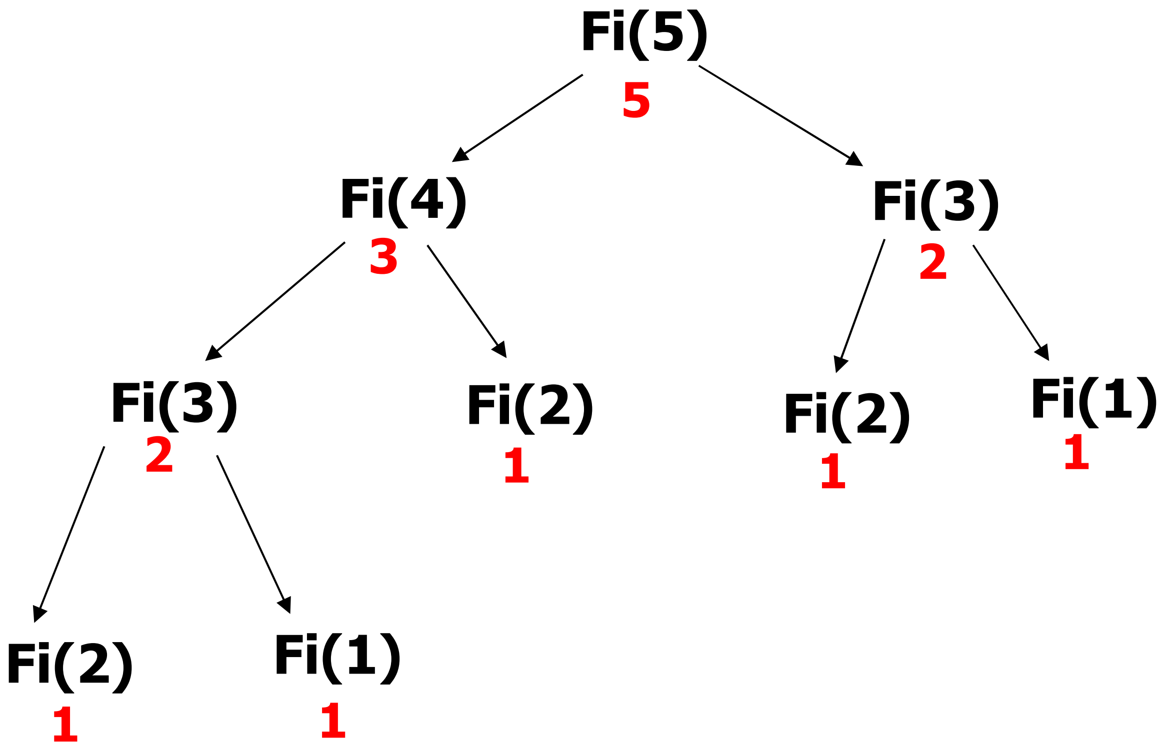
- Algoritmo recursivo para calcular o n-esimo termo da seqüência de Fibonacci:

```
int fibonacci_rec(int);

int fibonacci_rec(int n){
  if (n==1 || n==2) //condição de parada
    return 1;
  else
    return fibonacci_rec(n-1)+fibonacci_rec(n-2);
}
```

Ex. – Seqüência de Fibonacci

- Arvore de recurssão do algoritmo para $n = 5$



Ex. – Seqüência de Fibonacci

- Arvore de recurssão são úties para analizar e comprender o funcionamento de algoritmos recursivos,
- Note que para cada chamada com $n>2$ duas novas chamadas recursivas são feitas,
- O número de chamadas cresce exponencialmente,
- Para $n=5$ são feitas 9 chamadas a função,
- Adicionalmente $Fi(3)$ e $Fi(1)$ são chamadas três vezes e $Fi(2)$ é chamada duas vezes,

Ex. – Seqüência de Fibonacci

- A implementação recursiva para calcular o n-ésimo elemento da seqüência de Fibonacci mostrou-se muito ineficiente,
- É possível obtermos um algoritmo iterativo para fazer essa tarefa?
- Escreva um algoritmo iterativo para calcular o n-esimo termo da seqüência.

Ex. – Seqüência de Fibonacci

- Fibonacci iterativo:

```
int fibonacci_ite(int n){
  int i, F1, F2, F;

  if (n==1 || n==2)
    return 1;

  F1 = F2 = 1;
  for(i=2;i<n;i++){
    F = F1 + F2;
    F2 = F1;
    F1 = F;
  }
  return F;}
```

Ex. – Seqüência de Fibonacci

- Fibonacci iterativo:

```
int fibonacci_ite(int n){
  int i, F1, F2, F;

  F = F1 = F2 = 1;
  for(i=2;i<n;i++){
    F = F1 + F2;
    F2 = F1;
    F1 = F;
  }
  return F;}
```

Recursivo vs Iterativo

- Para cada algoritmo recursivo existe um algoritmo iterativo que executa a mesma tarefa,
- Algoritmos recursivos são mais elegantes e reproduzem claramente as definições matemáticas,
- Algoritmos iterativos são mais eficientes,
- Algoritmos recursivos quase sempre consomem mais recursos (memória e processador), sobrecarga pelas chamadas a função.

Recursivo vs Iterativo

- algoritmos recursivos são mais difíceis de serem depurados, especialmente quando for alta a profundidade de recursão,
- Para alguns problemas (recursivos por natureza) acharmos um algoritmo iterativo é uma tarefa díficil.

Ex. – Cálculo de Potência

- Escreva uma função recursiva que implemente a operação a^n onde n é um número inteiro positivo.

$$a^n = a \times a \times a \times \cdots \times a$$

$$a^n = a \times a^{(n-1)}$$

$$a^n = \begin{cases} a \times a^{(n-1)} & n > 0 \\ 1 & n = 0 \end{cases}$$

definição recursiva (para $n > 0$)

condição de parada (para $n = 0$)

Ex. – Cálculo de Potência

```
float potencia_rec(float, int);

float potencia_rec(float a, int n){
  if (n==0)
    return 1;
  else
    return a*potencia_rec(a, n-1);
}
```

- Construa a árvore de recursão para o algoritmo anterior considerando $a = 2$ e $n = 4$.

Ex. – Cálculo de Potência

- Árvore de recursão $a=2$ e $n=4$

$$
\begin{tikzcd}
\text{pot\_rec(2, 4)} \arrow[d] & 16 \\
\text{pot\_rec(2, 3)} \arrow[d] & 8 \\
\text{pot\_rec(2, 2)} \arrow[d] & 4 \\
\text{pot\_rec(2, 1)} \arrow[d] & 2 \\
\text{pot\_rec(2, 0)} & 1
\end{tikzcd}
$$

Ex. – Torres de Hanoi

- *Durante o reinado do imperador Fo Hi, existia um templo que marcaria o centro do universo. Dentro deste templo, alguns monges moviam discos de ouro entre 3 torres de diamante. Deus colocou 64 discos de ouro em uma das torres na hora da criação do universo. Diz-se que, quando os monges completarem a tarefa de transportar todos os discos para a terceira torre, o universo terminará. Quanto tempo demoraram os monges para completar a tarefa?*

Retorno

Ex. – Torres de Hanoi

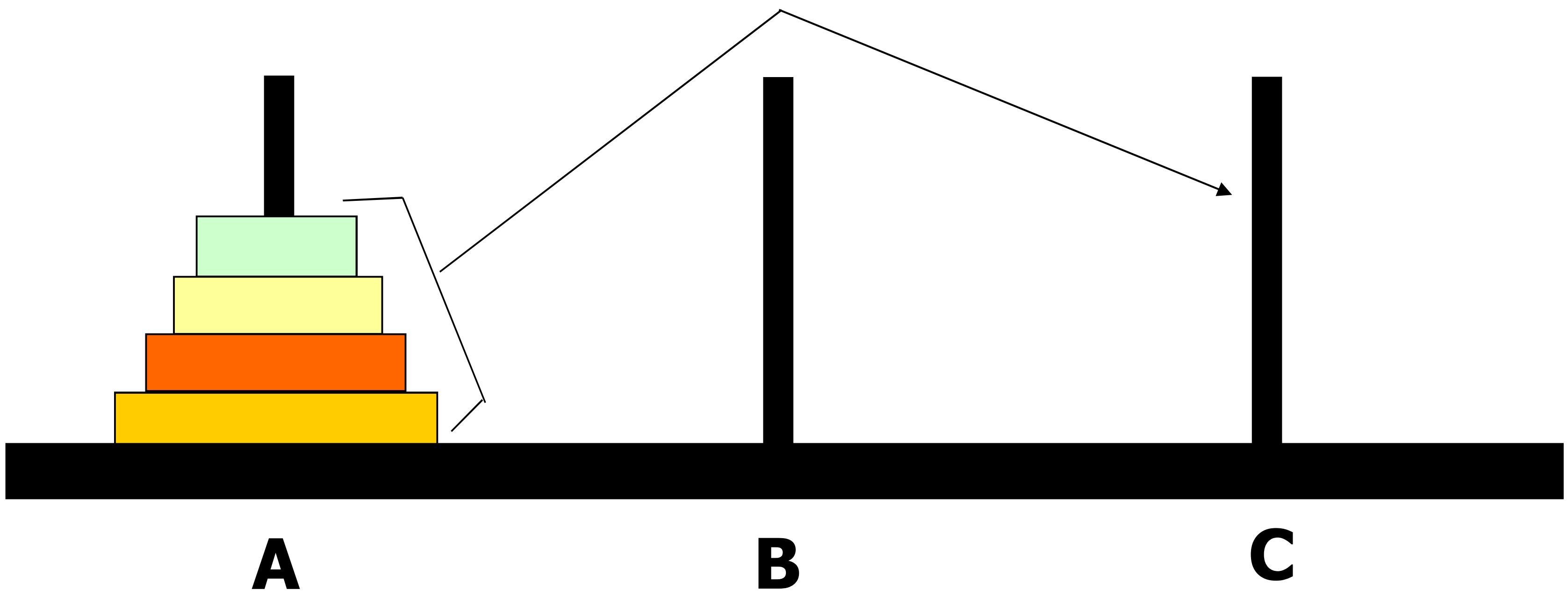
- Problema criado pelo matemático francês Edouard Lucas, em 1883;
- 3 torres (**A**, **B** e **C**);
- n discos de vários tamanhos na torre **A** (colocados em ordem decrescente segundo o tamanho);
- Objetivo: transportar os n discos da torre **A** para a torre **B**;

Ex. – Torres de Hanoi

- Regras:
  - nunca colocar um disco maior sobre um disco menor;
  - pode-se mover um único disco por vez;
  - nunca colocar um disco em outro lugar que não numa das três hastes.

Ex. – Torres de Hanoi

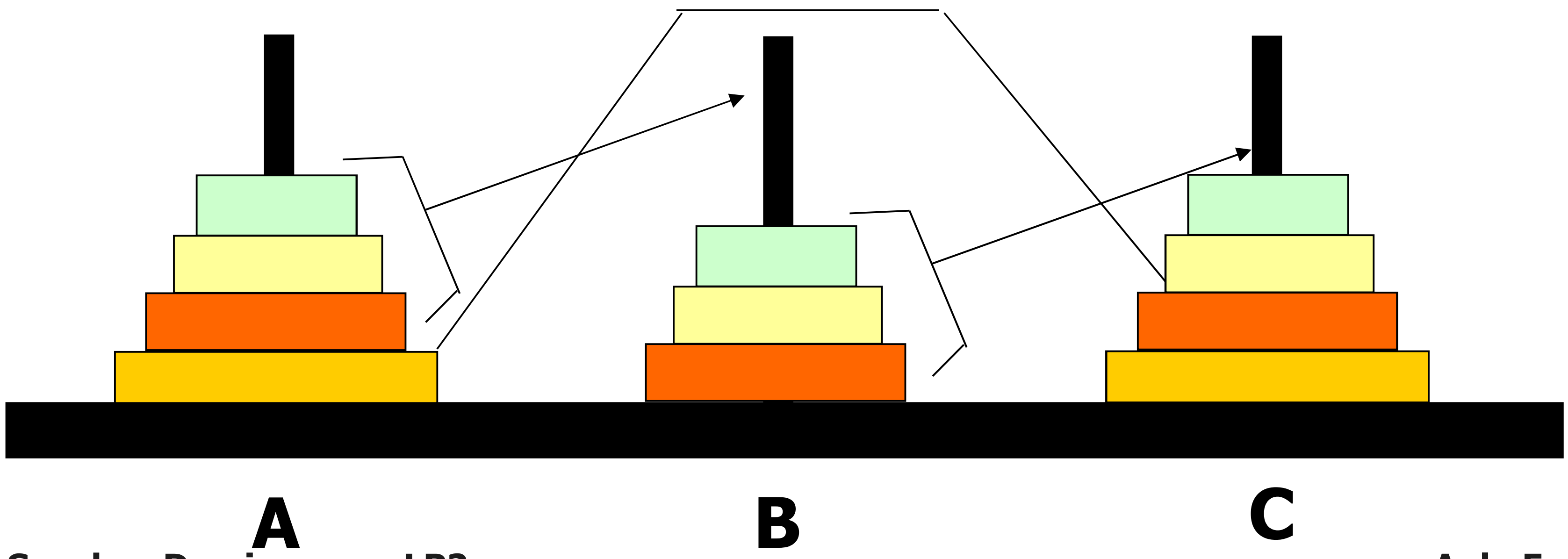
- Problema para 4 discos:



- Problema principal: Mover 4 discos da torre A para a torre C.

Ex. – Torres de Hanoi

- Recursão: simplificar o problema.
- Mover 4 discos da torre A para a torre C
  - Mover 3 discos de A para B
  - Mover 1 disco de A para C
  - Mover 3 discos de B para C



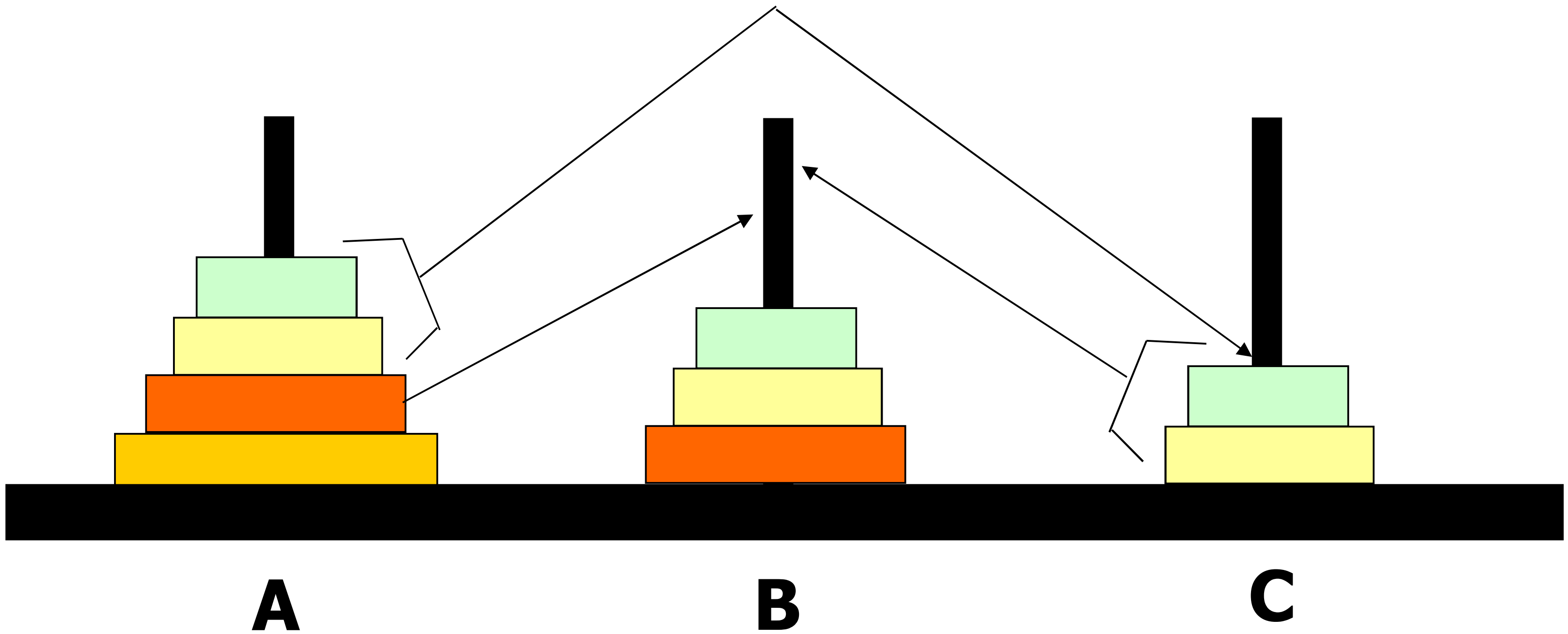
Ex. – Torres de Hanoi

- Logo o problema principal:
  - Mover 4 discos da torre A para a torre C
- Se transforma em dois problemas menores:
  - mover 3 discos da torre A para a torre B;
  - mover 1 disco da torre A para a torre C;
  - mover 3 discos da torre B para a torre C.

- Mas, como mover 3 discos?

Ex. – Torres de Hanoi

- Mover 3 discos da torre A para a torre B
  - Mover 2 discos de A para C
  - Mover 1 disco de A para B
  - Mover 2 discos de C para B



Ex. – Torres de Hanoi

- Função recursiva geral, mover n discos da torre A para a torre C
  - mover (n-1) discos, da torre A para a torre B
  - mover 1 disco da torre A para a torre C
  - mover (n-1) discos, da torre B para a torre C

- Condição de parada:
  - se numero de discos = 1, faz o movimento diretamente

Ex. – Torres de Hanoi

- Função recursiva:

```
void torres_hanoi(int, char, char, char);

void torres_hanoi(int n, char orig, char dest, char aux){
  if (n==1){
    printf("Mover disco de %c para %c\n", orig, dest);
    return;
  }

  torres_hanoi(n-1, orig, aux, dest);
  torres_hanoi(1, orig, dest, aux);
  torres_hanoi(n-1, aux, dest, orig);
}
```

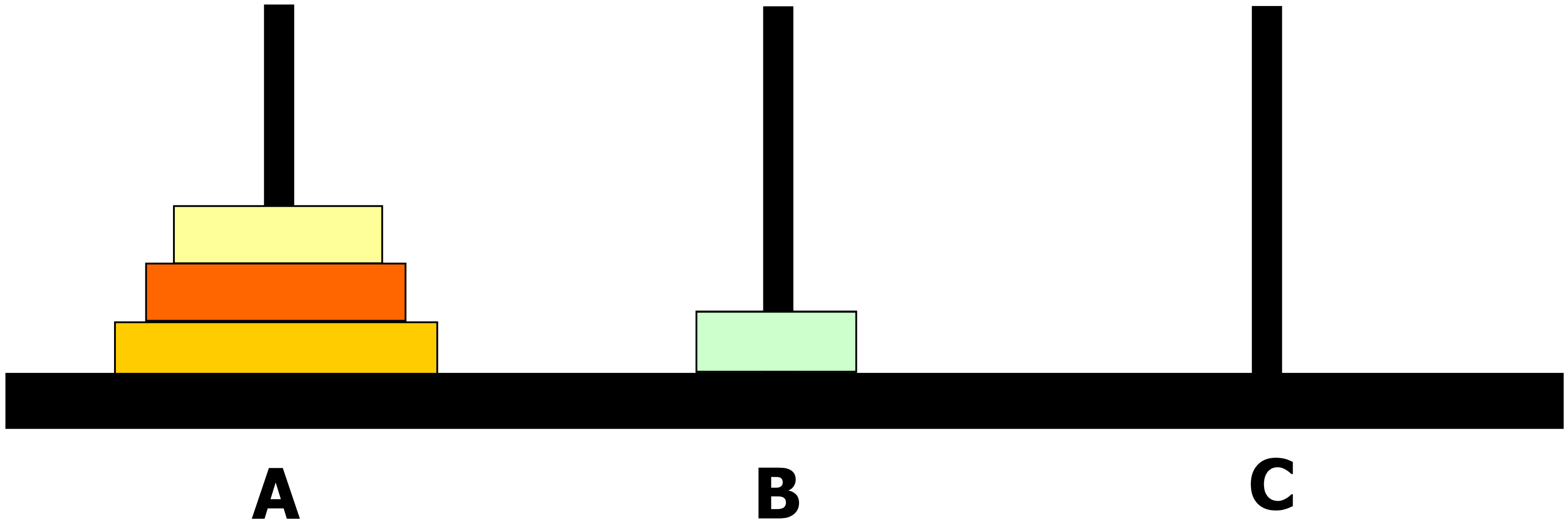
Ex. – Torres de Hanoi

- Saída:

```
Mover disco de A para B
Mover disco de A para C
Mover disco de B para C
Mover disco de A para B
Mover disco de C para A
Mover disco de C para B
Mover disco de A para B
Mover disco de A para C
Mover disco de B para C
Mover disco de B para A
Mover disco de C para A
Mover disco de B para C
Mover disco de A para B
Mover disco de A para C
Mover disco de B para C
```

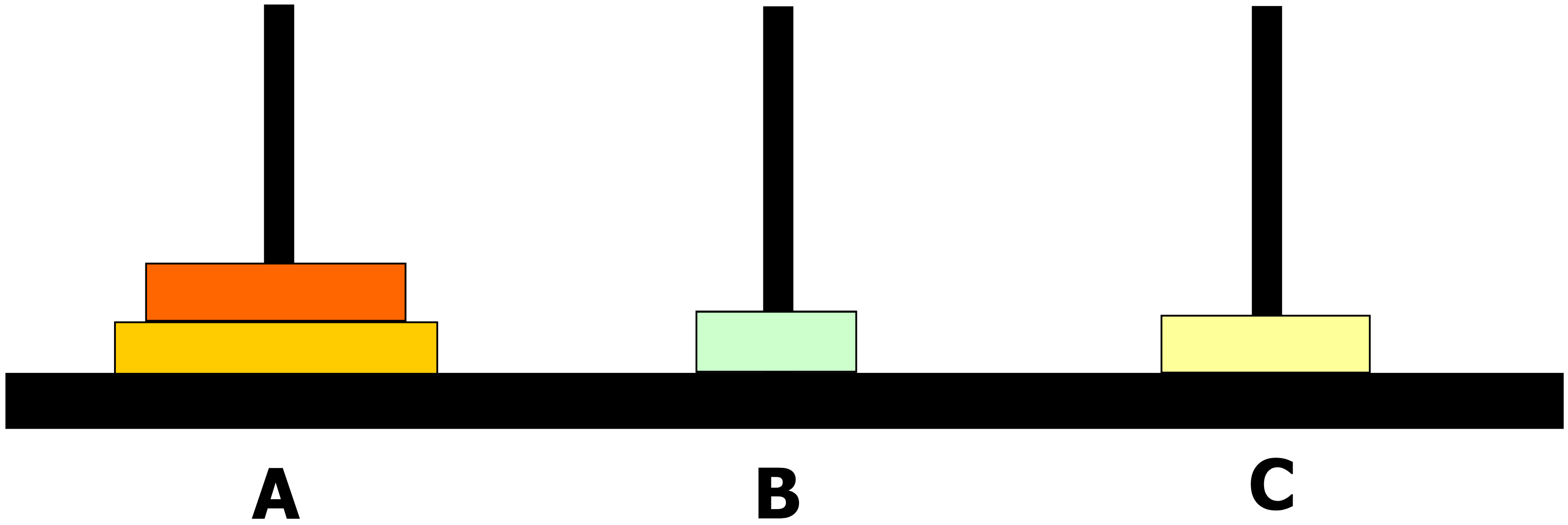
Ex. – Torres de Hanoi

- **Mover disco de A para B**



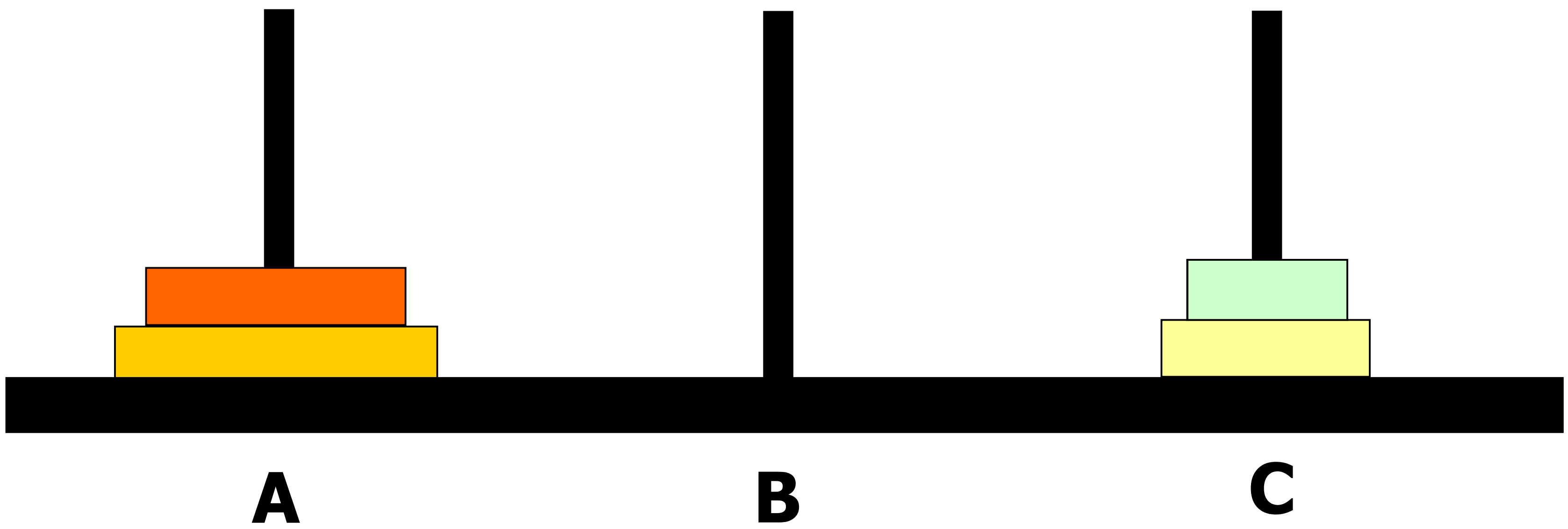
Ex. – Torres de Hanoi

- **Mover disco de A para C**



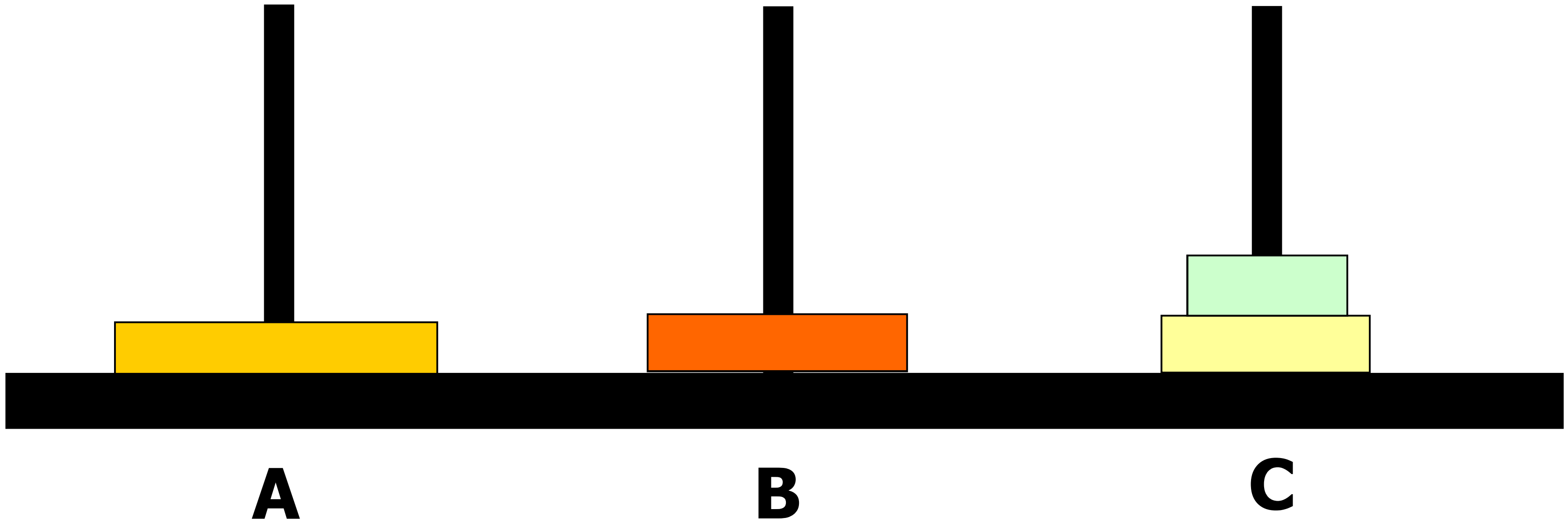
Ex. – Torres de Hanoi

- Mover disco de B para C



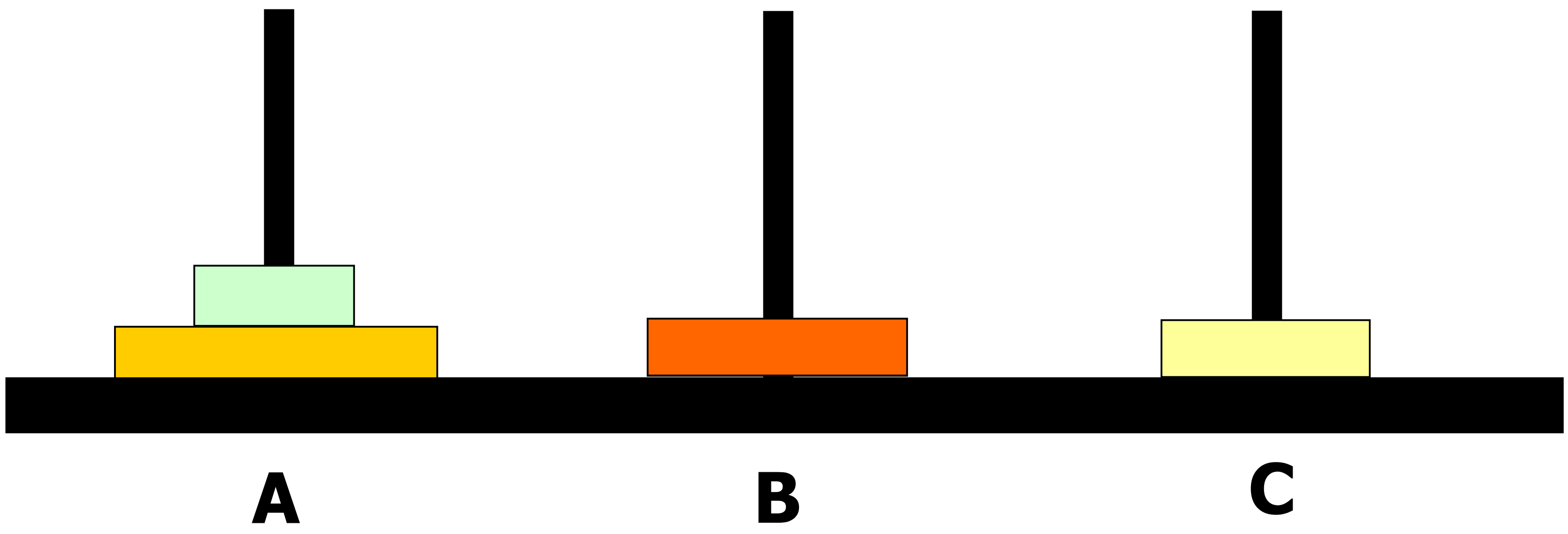
Ex. – Torres de Hanoi

- **Mover disco de A para B**



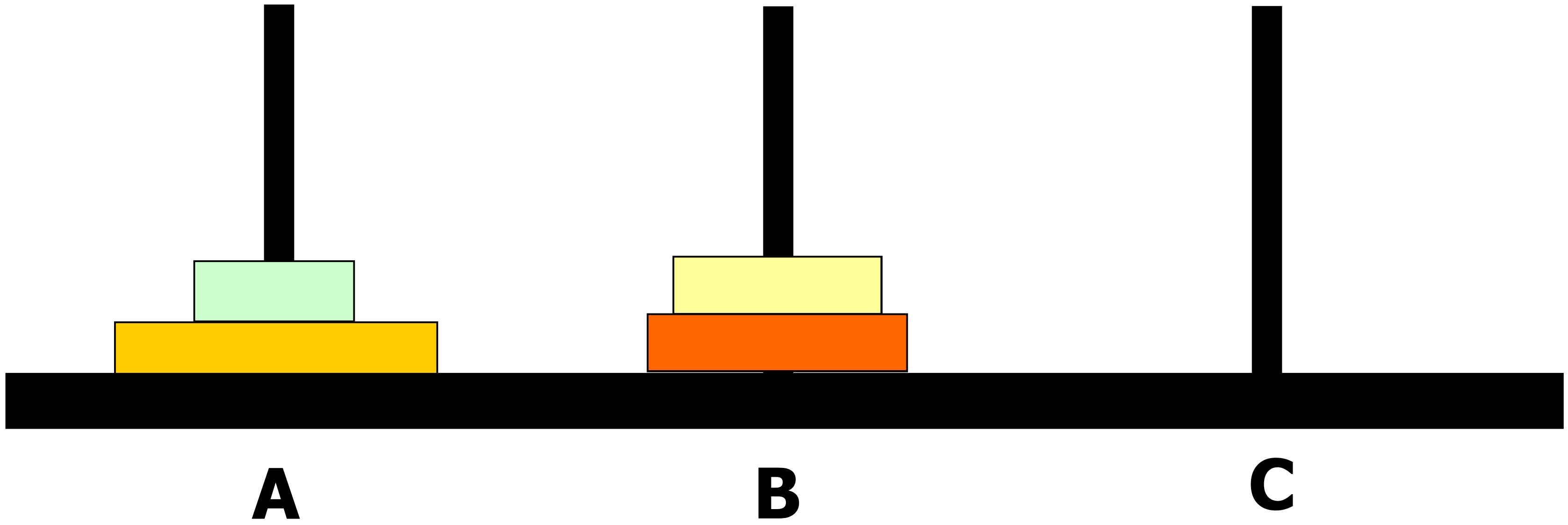
Ex. – Torres de Hanoi

- **Mover disco de C para A**



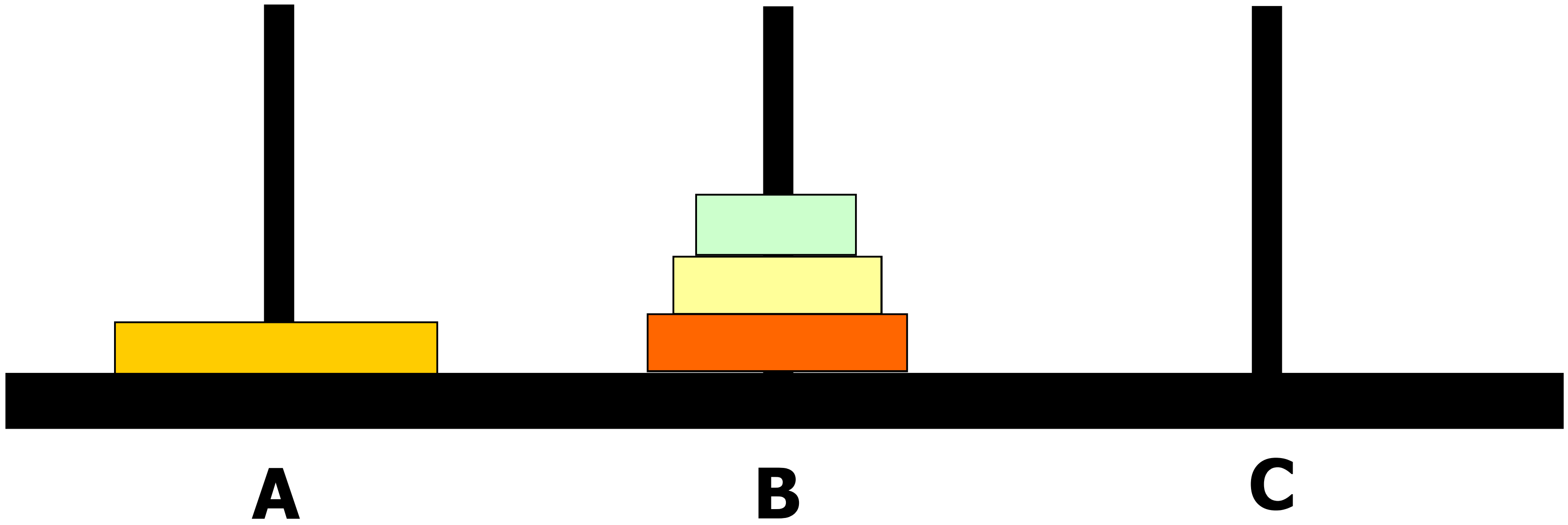
Ex. – Torres de Hanoi

- Mover disco de C para B



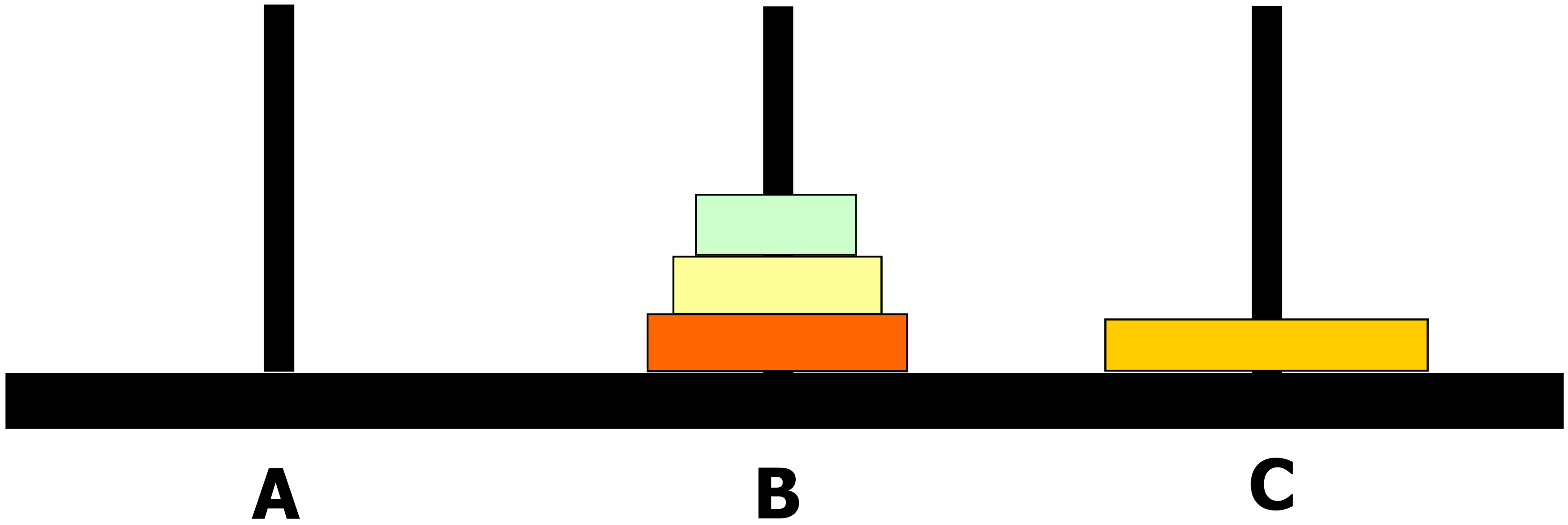
Ex. – Torres de Hanoi

- Mover disco de A para B



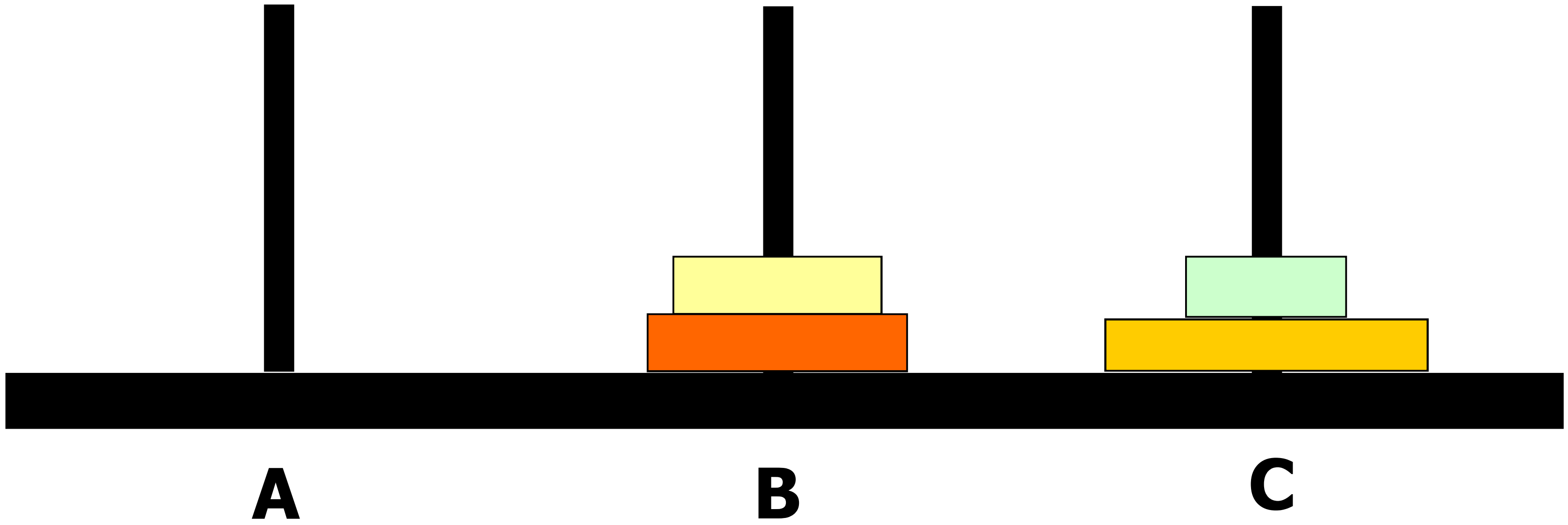
Ex. – Torres de Hanoi

- **Mover disco de A para C**



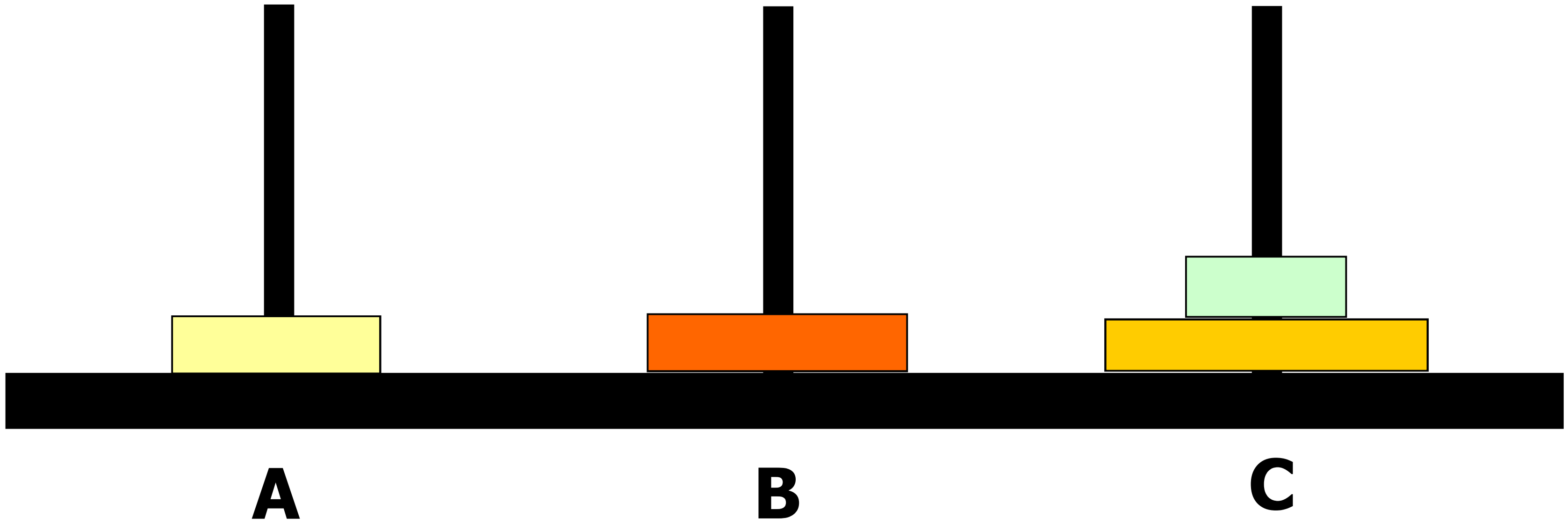
Ex. – Torres de Hanoi

- Mover disco de B para C



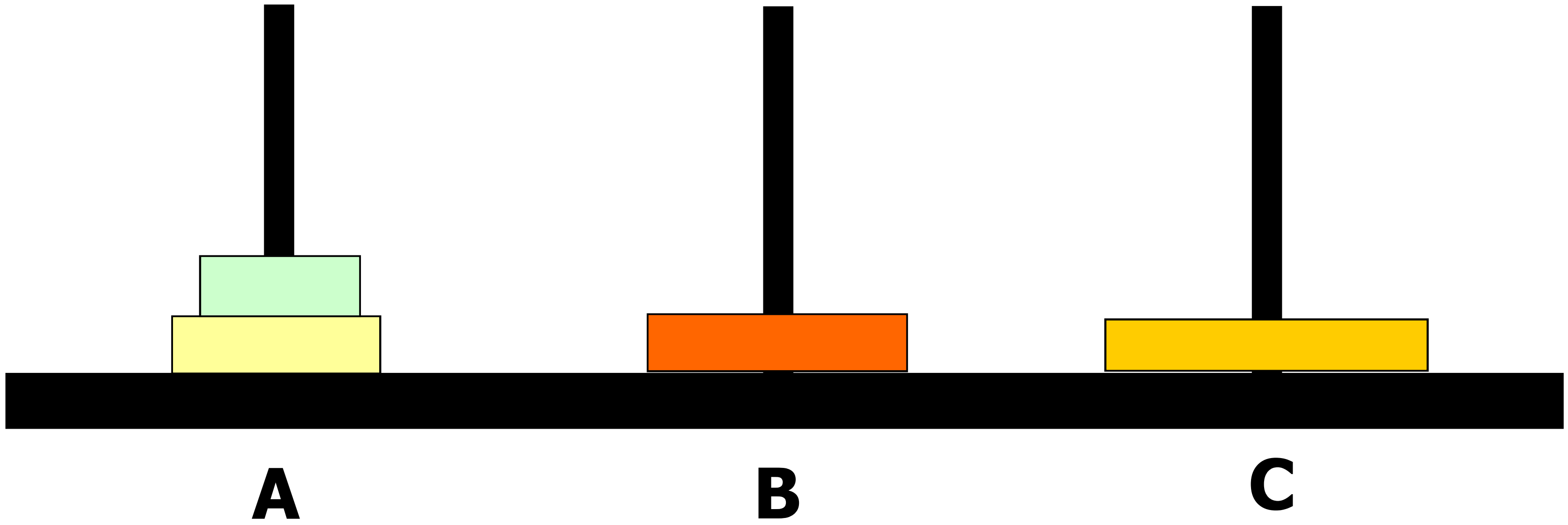
Ex. – Torres de Hanoi

- **Mover disco de B para A**



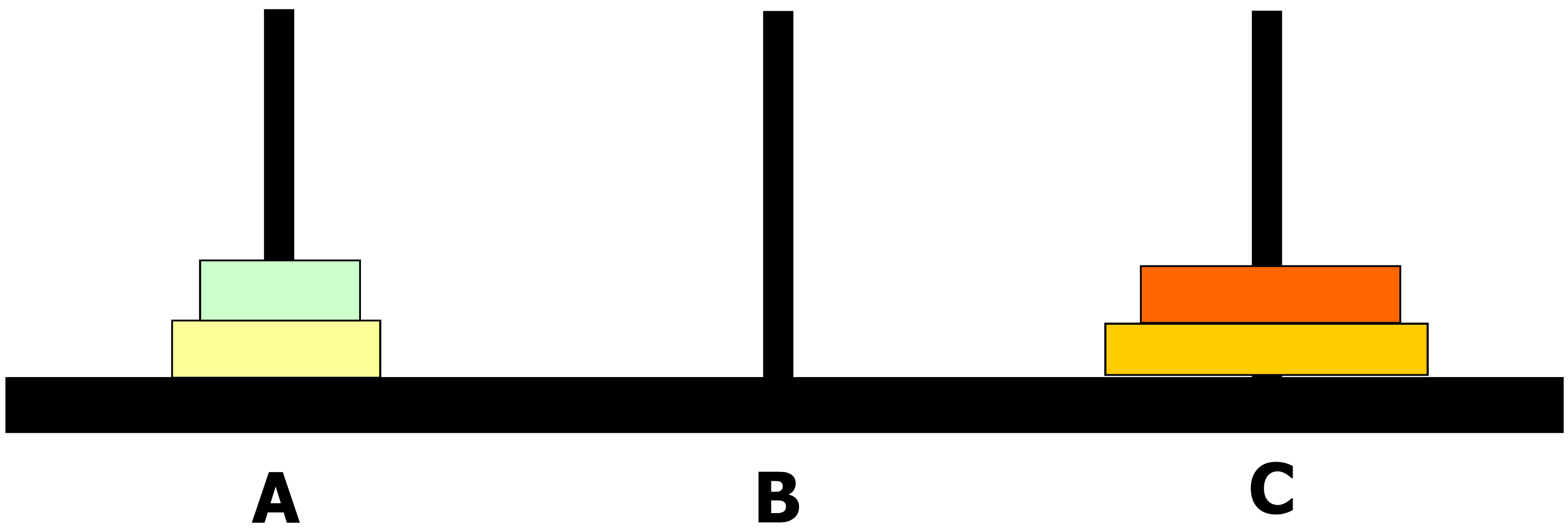
Ex. – Torres de Hanoi

- **Mover disco de C para A**



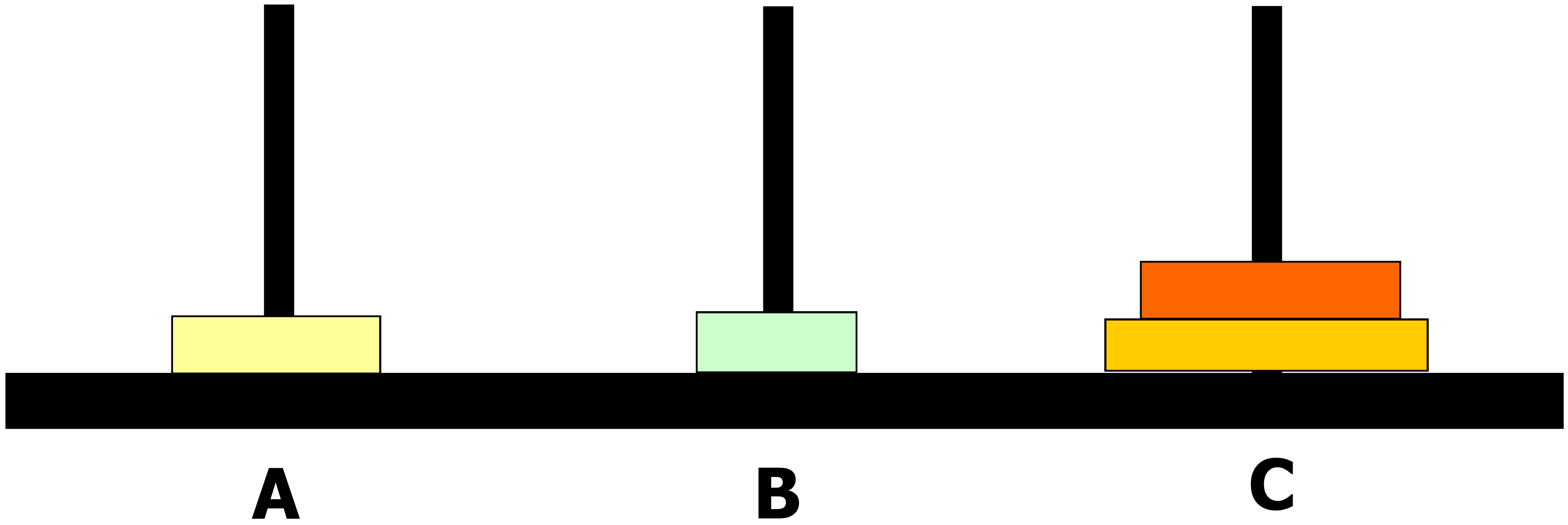
Ex. – Torres de Hanoi

- **Mover disco de B para C**



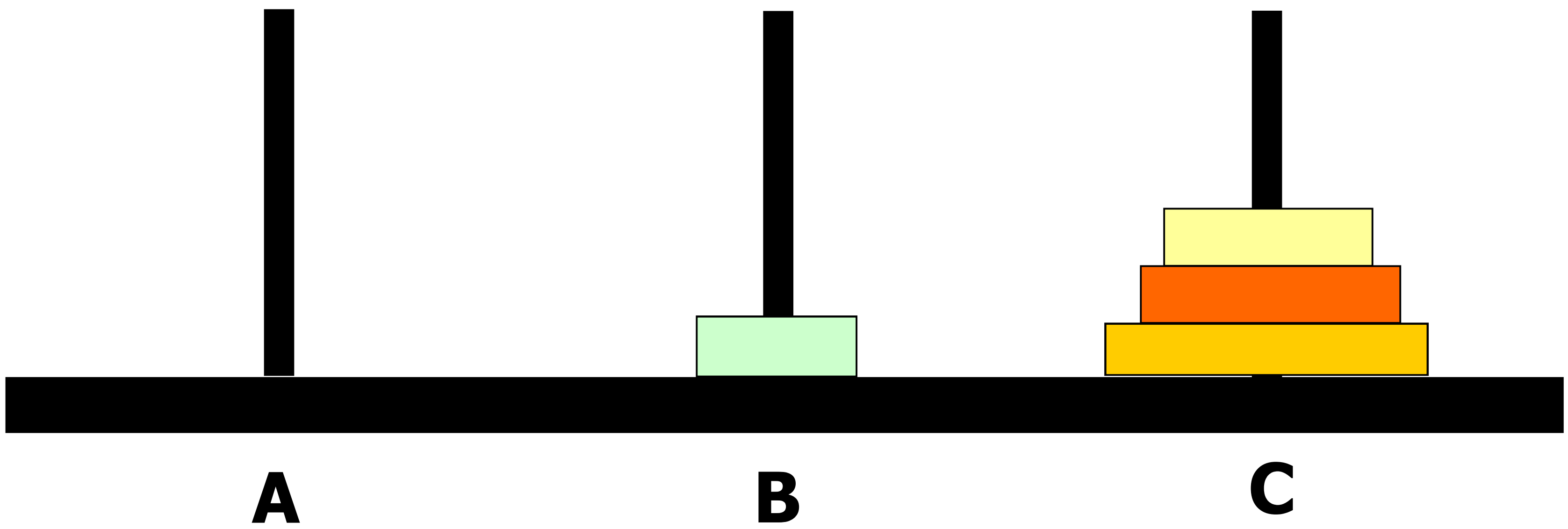
Ex. – Torres de Hanoi

- Mover disco de A para B



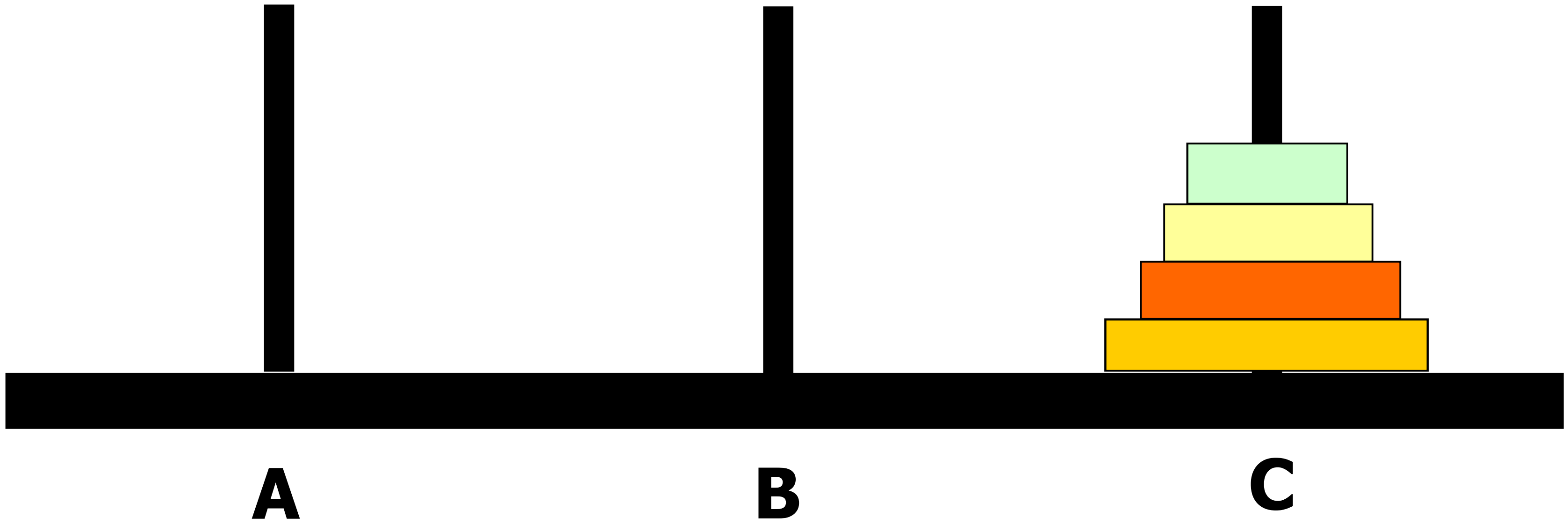
Ex. – Torres de Hanoi

- `Mover disco de A para C`



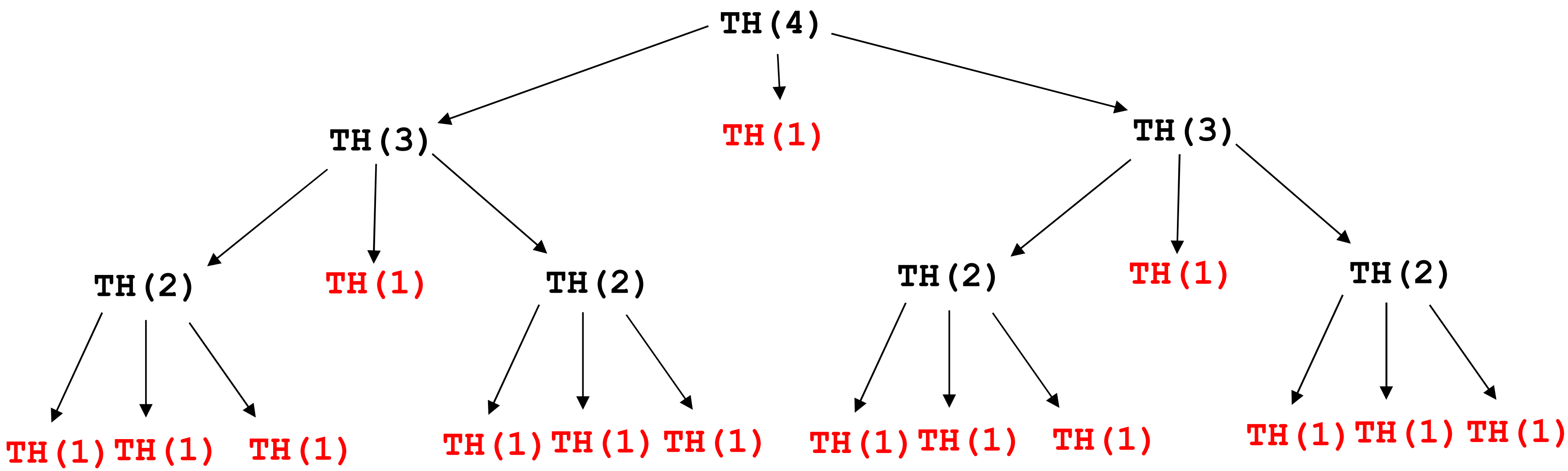
Ex. – Torres de Hanoi

- **Mover disco de B para C**



Ex. – Torres de Hanoi

- Árvore de recursão:



- Número de movimentos = 15
- Profundidade da recursão: 4
- Número de movimentos = 2^n-1

Ex. – Torres de Hanoi

- Retornando a lenda dos monges ... (enlace)
- Quantidade de movimentos necessários para acabar o universo: 2^{64}-1 = 18446744073709551616,

- Assumindo que os monges realizem 1 movimento por segundo, e não cometam erros, eles irão levar um total de

  18446744073709551616 ≈ 585,000,000,000 anos

Ex. – Comprimento de string

- Escreva uma função que receba uma string e retorne seu comprimento, sua função não pode utilizar a função `strlen()` da biblioteca padrão e não pode conter nenhum laço (`for`, `while`, `do..while`).
- **Sugestão**: Utilize recursividade, e considere que se retirarmos o primeiro caractere da string o tamanho total é 1 + tamanho do resto da string.

Ex. – Comprimento de string

```
int strlen_rec(char *);

int main(){
  char str[] = "Recursividade";

  printf("Comprimento: %d\n", strlen_rec(str));

  system("PAUSE");
  return 0;
}

int strlen_rec(char * ch){
  if (!(*ch))  //condição de parada
    return 0;
  else
    return 1 + strlen_rec(++ch);
}
```

Ex. – Busca seqüencial

- Escreva uma função recursiva que receba os elementos de um vetor e um valor a ser procurado, a função deve retornar 1 se o valor estiver no vetor 0 se não estiver.

- **Sugestão**: A busca em N elementos significa olhar para o 1º elemento. Se não for o elemento procurado, procurar nos N-1 elementos restantes.

Ex. – Busca seqüencial

- Parâmetros da função:
  - Valor a ser procurado (val)
  - Elementos do vetor (v)
  - Extremo esquerdo do conjunto (ee)
  - Extremo direito do conjunto (ed)
- Condição de parada:
  - Valor encontrado
  - Extremo esquerdo > Extremo direito

Ex. – Busca seqüencial

```
int busca_rec(int, int, int, int *);

int busca_rec(int ee, int ed, int val, int *v){
  if (ee>ed)
    return 0;
  if (v[ee] == val)
    return 1;
  return busca_rec(ee+1, ed, val, v);
}
```

Comentários

- Não se aprende recursividade sem praticar
- Para montar um algoritmo recursivo:
  - Defina pelo menos um caso básico (condição de parada);
  - Quebre o problema em problemas menores, definindo o(s) caso(s) com recursão(ões)
  - Fazer o teste de finitude, isto é, certificar-se de que as sucessivas chamadas recursivas levam obrigatoriamente, e numa quantidade finita de vezes, ao(s) caso(s) básico(s)

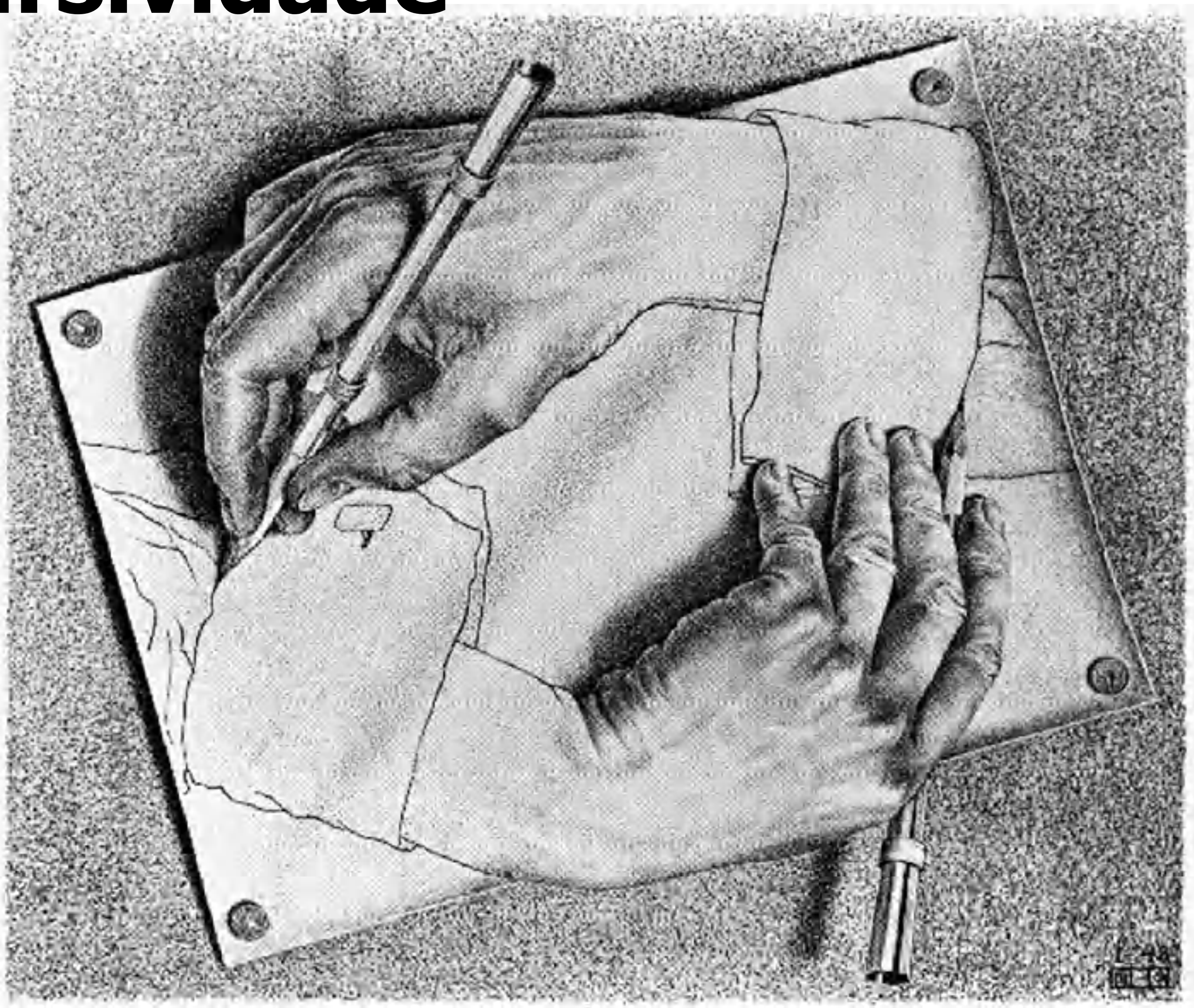
Comentários

- Vantagens:
  - Redução do tamanho do código fonte,
  - Permite descrever algoritmos de forma mais clara e concisa,
  - Os algoritmos recursivos normalmente são mais compactos, mais legíveis e mais fáceis de serem compreendidos,
  - Reproduzem claramente as definições matemáticas

Comentários

- Desvantagens:
  - Redução do desempenho de execução devido ao tempo para gerenciamento de chamadas
  - Dificuldades na depuração de programas recursivos, especialmente se a recursão for muito profunda

Recursividade



Maurits Cornelis Escher (1898-1972)
Matemático e Artista Plástico
Fonte: http://www.mcescher.com/