

MC102 – Algoritmos e Programação de Computadores

Instituto de Computação

UNICAMP

Primeiro Semestre de 2015

Roteiro

- 1 Recursão
- 2 Fatorial
- 3 Gerenciamento de memória
- 4 Recursão \times Iteração
- 5 Soma dos elementos de um vetor
- 6 Números de Fibonacci
- 7 Exemplos simples de recursão
- 8 Exemplos de recursão com vetores e strings
- 9 Torre de Hanói
- 10 Exercícios



Recursão

- Suponha que desejamos criar um algoritmo para resolver um determinado problema.
- Usando o método de recursão/indução, a solução de um problema pode ser expressa da seguinte forma:
 - ▶ Primeiramente, definimos a solução para casos base.
 - ▶ Em seguida, definimos como resolver o problema para um caso geral, utilizando soluções para instâncias menores do problema.

Indução

- Indução é uma técnica de demonstração matemática em que algum parâmetro da proposição a ser demonstrada envolve números naturais.
- Seja T uma proposição que desejamos provar como verdadeira para todos valores naturais n .
- Ao invés de provar diretamente que T é válido para todos os valores de n , basta provar os itens 1 e 3, listados a seguir:
 - 1) *Caso Base: Provar que T é válido para $n = 1$.*
 - 2) *Hipótese de Indução: Assumimos que T é válido para $n - 1$.*
 - 3) *Passo de Indução: Assumindo que T é válido para $n - 1$, devemos provar que T é válido para n .*

- Por que a indução funciona? Por que as duas condições são suficientes?
 - ▶ Mostramos que T é válido para um caso simples como $n = 1$.
 - ▶ Com o passo da indução, mostramos que T é válido para $n = 2$.
 - ▶ Como T é válido para $n = 2$, pelo passo de indução, T também é válido para $n = 3$ e assim por diante.

Exemplo

Teorema

A soma $S(n)$ dos primeiros n números naturais é $S(n) = n(n+1)/2$.

Prova:

- *Base:* Para $n = 1$, temos que $S(1) = n(n+1)/2 = 1(1+1)/2 = 1$.
- *Hipótese de Indução:* Vamos assumir que a fórmula é válida para $n - 1$, ou seja, $S(n - 1) = (n - 1)n/2$.
- *Passo:* Devemos mostrar que é válido para n . Por definição, $S(n) = S(n - 1) + n$. Por hipótese, $S(n - 1) = (n - 1)n/2$, logo:

$$\begin{aligned} S(n) &= S(n - 1) + n \\ &= (n - 1)n/2 + n \\ &= (n^2 - n)/2 + n \\ &= (n^2 + n)/2 \\ &= n(n + 1)/2 \end{aligned}$$



Recursão

- Definições recursivas de funções operam como o princípio matemático da indução visto anteriormente, ou seja, a solução é inicialmente definida para o(s) caso(s) base e estendida para o caso geral.

Fatorial

- Problema: calcular o fatorial de um número inteiro não negativo (N).
- Qual é o caso base?
 - ▶ $0! = 1$
- Qual é o passo indutivo?
 - ▶ $N! = N \times (N - 1)!$
- Este problema é trivial pois a própria definição de fatorial é recursiva.

- Portanto, a solução do problema pode ser expressa da seguinte forma:
 - ▶ Se $N = 0$ então $0! = 1$.
 - ▶ Se $N \geq 1$ então $N! = N \times (N - 1)!$.
- Note como aplicamos o princípio da indução:
 - ▶ Sabemos a solução para um caso base ($N = 0$).
 - ▶ Definimos a solução do problema geral em termos do mesmo problema, mas para um caso mais simples.

Fatorial

```
long int fatorial(int N) {  
    int X;  
    long int Y;  
  
    /* caso base */  
    if (N == 0)  
        return 1;  
    else {  
        X = N - 1;  
        Y = fatorial(X);  
        return N * Y;  
    }  
}
```

Fatorial

- Para solucionar o problema, faz-se uma chamada para a própria função, por isso, esta função é chamada de recursiva.
- Recursividade geralmente permite uma descrição mais clara e concisa dos algoritmos, especialmente quando o problema é recursivo por natureza.

Gerenciamento de memória

- Devemos entender como é feito o controle sobre as variáveis locais em chamadas recursivas.
- A memória de um sistema computacional é dividida em três partes:
 - ▶ *Espaço Estático*: contém as variáveis globais e código do programa.
 - ▶ *Heap*: para alocação dinâmica de memória.
 - ▶ *Pilha*: para execução de funções.

Gerenciamento de memória

- Toda vez que uma função é invocada, suas variáveis locais são armazenadas no topo da pilha.
- Quando uma função termina a sua execução, suas variáveis locais são removidas da pilha.

Gerenciamento de memória

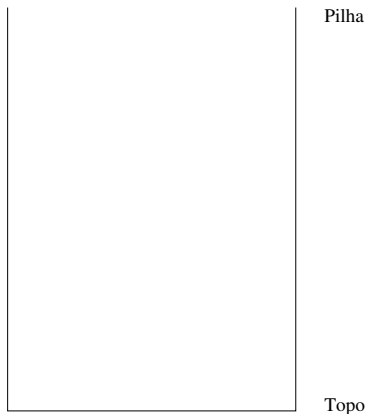
```
int f1(int a, int b) {  
    int c = 5;  
    return (c + a + b);  
}
```

```
int f2(int a, int b) {  
    int c;  
    c = f1(b, a);  
    return c;  
}
```

```
int main() {  
    f2(2, 3);  
  
    return 0;  
}
```

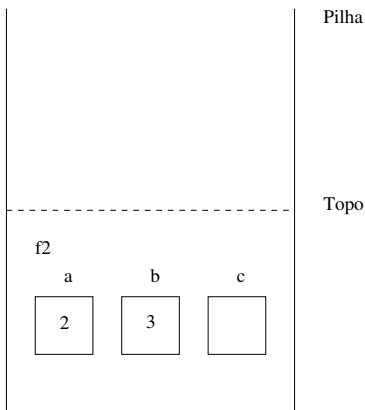
Gerenciamento de memória

O programa começa a execução pela função `main`, que não aloca nenhuma variável local. Sendo assim, inicialmente, a pilha está vazia.



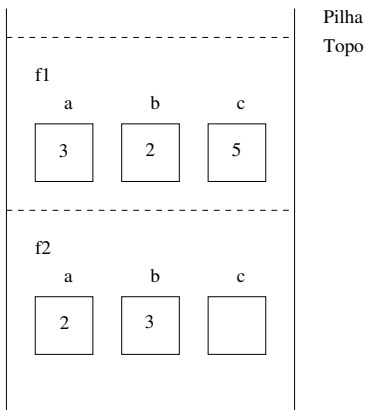
Gerenciamento de memória

A função `main` invoca a função `f2` e as variáveis locais desta são alocadas no topo da pilha.



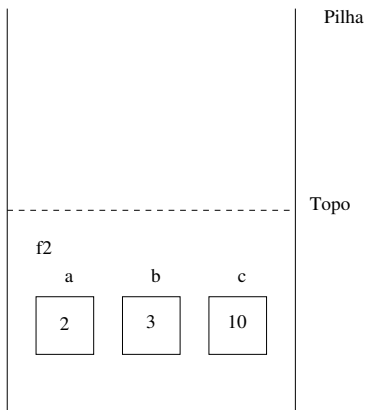
Gerenciamento de memória

A função f2 invoca a função f1 e as variáveis locais desta são alocadas no topo da pilha sobre as de f2.



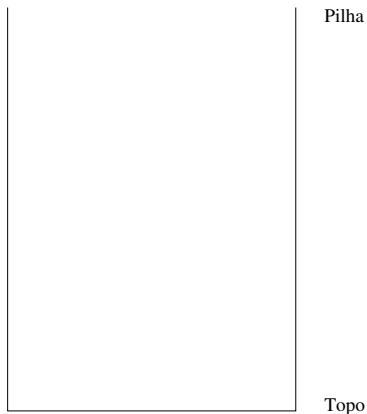
Gerenciamento de memória

A função `f1` termina, devolvendo 10. As variáveis locais de `f1` são removidas da pilha.



Gerenciamento de memória

Finalmente, a função $f2$ termina a sua execução devolvendo 10. Suas variáveis locais são removidas da pilha.



Gerenciamento de memória

- No caso de chamadas recursivas é como se cada chamada correspondesse à chamada de uma função distinta.
- As execuções das chamadas de funções recursivas são feitas na pilha, assim como qualquer função.
- O último conjunto de variáveis alocadas na pilha, que está no topo, corresponde às variáveis da última chamada da função.
- Quando termina a execução de uma chamada da função, as variáveis locais desta são removidas da pilha.

Gerenciamento de memória

Considere novamente a solução recursiva para se calcular o fatorial e assumamos que seja feita a chamada `fatorial(4)`.

```
long int fatorial(int N) {  
    int X;  
    long int Y;  
  
    if (N == 0)  
        return 1;  
    else {  
        X = N - 1;  
        Y = fatorial(X);  
        return N * Y;  
    }  
}
```

Gerenciamento de memória

- Cada chamada da função `fatorial` cria novas variáveis locais de mesmo nome (`N`, `X` e `Y`).
- Portanto, múltiplas variáveis (`N`, `X` e `Y`) podem existir em um dado momento.
- Em um dado instante, o nome `N` (ou `X` ou `Y`) refere-se à variável local ao corpo da função que está sendo executada naquele instante.
- A seguir, veremos o estado da pilha de execução para a chamada `fatorial(4)`.

Gerenciamento de memória

N X Y

4	*	*

N X Y

3	*	*
4	3	*

N X Y

2	*	*
3	2	*
4	3	*

N X Y

1	*	*
2	1	*
3	2	*
4	3	*

N X Y

0	*	*
1	0	*
2	1	*
3	2	*
4	3	*

N X Y

1	0	1
2	1	*
3	2	*
4	3	*

N X Y

2	1	1
3	2	*
4	3	*

N X Y

3	2	2
4	3	*

N X Y

4	3	6

N X Y

N X Y

Retorna 24

Gerenciamento de memória

- Note que as variáveis X e Y são desnecessárias.

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

Recursão × Iteração

- Soluções recursivas são geralmente mais concisas do que as iterativas.
- Soluções iterativas em geral consomem menos memória do que as soluções recursivas.
- Cópia dos parâmetros a cada chamada recursiva é um custo adicional para as soluções recursivas.

Recursão × Iteração

No caso da função fatorial, a solução iterativa é mais eficiente:

```
long int fatorial(int n) {  
    long int fat = 1;  
    int i;  
  
    for (i = 1; i <= n; i++)  
        fat = fat * i;  
  
    return fat;  
}
```

Soma dos elementos de um vetor

- Seja um vetor v de inteiros de tamanho n .
- Queremos saber a soma de todos os seus elementos.
- Como podemos descrever este problema de forma recursiva?
- Vamos denotar por $\text{soma}(v, n)$ a soma n primeiros elementos de um vetor v . Com isso, temos:
 - ▶ Se $n = 0$ então $\text{soma}(v, 0) = 0$.
 - ▶ Se $n > 0$ então $\text{soma}(v, n) = \text{soma}(v, n-1) + v[n-1]$.

Soma dos elementos de um vetor

```
int soma(int v[], int n) {  
    if (n == 0)  
        return 0;  
    else  
        return soma(v, n - 1) + v[n - 1];  
}
```

Soma dos elementos de um vetor

- O método recursivo sempre termina, pois:
 - ▶ Existe um caso base bem definido.
 - ▶ A cada chamada recursiva, usamos um valor menor de n .

Soma dos elementos de um vetor

Neste caso, a solução iterativa também é melhor que a recursiva:

```
int soma(int v[], int n) {  
    int soma = 0, i;  
  
    for (i = 0; i < n; i++)  
        soma = soma + v[i];  
  
    return soma;  
}
```

Números de Fibonacci

- A série de Fibonacci é a seguinte:
 - ▶ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,
- Suponha que queremos determinar qual é o n -ésimo número da série de Fibonacci, que denotaremos por $\text{Fibonacci}(n)$.
- Como descrever o n -ésimo número de Fibonacci de forma recursiva?

Números de Fibonacci

- Temos os seguintes casos base:
 - ▶ Se $n = 1$ ou $n = 2$ então $\text{Fibonacci}(n) = 1$.
- Sendo assim, podemos computar $\text{Fibonacci}(n)$ como:
 - ▶ $\text{Fibonacci}(n) = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)$.

Números de Fibonacci (versão recursiva)

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

Números de Fibonacci (versão iterativa)

```
unsigned long int Fibonacci(int n) {  
    unsigned long int anterior, atual, temp;  
    int i;  
  
    anterior = 1;  
    atual = 1;  
  
    for (i = 3; i <= n; i++) {  
        temp = atual;  
        atual = atual + anterior;  
        anterior = temp;  
    }  
  
    return atual;  
}
```

Números de Fibonacci

- Para $n > 2$, quantas somas são necessárias para calcular $\text{Fibonacci}(n)$ usando uma estratégia iterativa?
 - ▶ São necessárias $n - 2$ somas.
- Para $n > 2$, quantas somas são realizadas para calcular $\text{Fibonacci}(n)$ usando a função recursiva vista anteriormente?
 - ▶ $\text{Fibonacci}(3)$: 1 soma.
 - ▶ $\text{Fibonacci}(4)$: 2 somas.
 - ▶ $\text{Fibonacci}(5)$: 4 somas.
 - ▶ $\text{Fibonacci}(6)$: 7 somas.
 - ▶ $\text{Fibonacci}(7)$: 12 somas.
 - ▶ $\text{Fibonacci}(8)$: 20 somas.
 - ▶ $\text{Fibonacci}(9)$: 33 somas.
 - ▶ $\text{Fibonacci}(10)$: 54 somas.
 - ▶ $\text{Fibonacci}(20)$: 6.765 somas.
 - ▶ $\text{Fibonacci}(30)$: 832.040 somas.
 - ▶ $\text{Fibonacci}(40)$: 102.334.155 somas.
 - ▶ $\text{Fibonacci}(50)$: 12.586.269.025 somas.
 - ▶ São realizadas $\text{Fibonacci}(n) - 1$ somas.

Números de Fibonacci

- A versão iterativa calcula uma única vez todos os valores de $\text{Fibonacci}(k)$, para $1 \leq k \leq n$, enquanto a versão recursiva calcula várias vezes cada um deles.
- Por exemplo, para $n = 20$, os números de vezes que a função recursiva calcula $\text{Fibonacci}(k)$, para $1 \leq k \leq 20$, são os seguintes:

$\text{Fibonacci}(1)$: 2584	$\text{Fibonacci}(11)$: 55
$\text{Fibonacci}(2)$: 4181	$\text{Fibonacci}(12)$: 34
$\text{Fibonacci}(3)$: 2584	$\text{Fibonacci}(13)$: 21
$\text{Fibonacci}(4)$: 1597	$\text{Fibonacci}(14)$: 13
$\text{Fibonacci}(5)$: 987	$\text{Fibonacci}(15)$: 8
$\text{Fibonacci}(6)$: 610	$\text{Fibonacci}(16)$: 5
$\text{Fibonacci}(7)$: 377	$\text{Fibonacci}(17)$: 3
$\text{Fibonacci}(8)$: 233	$\text{Fibonacci}(18)$: 2
$\text{Fibonacci}(9)$: 144	$\text{Fibonacci}(19)$: 1
$\text{Fibonacci}(10)$: 89	$\text{Fibonacci}(20)$: 1

- Para calcular o valor de $\text{Fibonacci}(n)$, a função recursiva calcula o valor de $\text{Fibonacci}(k)$, para $2 \leq k \leq n$, $\text{Fibonacci}(n-k+1)$ vezes.

Resumindo

- Recursão é uma técnica para se criar algoritmos em que:
 - ▶ Devemos descrever soluções para casos base.
 - ▶ Assumindo a existência de soluções para casos mais simples, mostramos como obter solução para o caso mais complexo.
- Algoritmos recursivos geralmente são mais claros e concisos.
- Devemos avaliar a clareza de código \times eficiência do algoritmo.

Soma

Soma de dois números inteiros não negativos, x e y , usando apenas incrementos ($++$) e decrementos unitários ($--$).

```
int soma(int x, int y) {  
    if (y == 0)  
        return x;  
    else  
        return soma(++x, --y);  
}
```

Multiplicação

Multiplicação de dois números inteiros positivos, x e y , usando apenas somas ou subtrações.

```
int mult(int x, int y) {  
    if (y == 1)  
        return x;  
    else  
        return mult(x, y - 1) + x;  
}
```


Soma de valores pares

Soma de todos os inteiros positivos pares menores ou iguais a um valor inteiro n .

```
int somapar(int n) {  
    if (n <= 0)  
        return 0;  
    else  
        if (n % 2 == 0)  
            return somapar(n - 2) + n;  
        else  
            return somapar(n - 1);  
}
```

Soma de valores pares

Soma de todos os inteiros positivos pares menores ou iguais a um valor inteiro n .

```
int somapar2(int n) {  
    if (n <= 0)  
        return 0;  
    else  
        return somapar2(n - 2) + n;  
}
```

```
int somapar(int n) {  
    if (n % 2 == 0)  
        return somapar2(n);  
    else  
        return somapar2(n - 1);  
}
```

Produtório

Cálculo do produtório $\prod_{i=m}^n i = m \times (m+1) \times (m+2) \times \cdots \times n$, tal que m e n são inteiros tais que $m \leq n$.

```
int produtorio(int m, int n)
    if (m == n)
        return m;
    else
        return m * produtorio(m + 1, n);
}
```

Produtório

Cálculo do produtório $\prod_{i=m}^n i = m \times (m+1) \times (m+2) \times \cdots \times n$, tal que m e n são inteiros tais que $m \leq n$.

```
int produtorio(int m, int n)
    if (m == n)
        return m;
    else
        return produtorio(m, n - 1) * n;
}
```

Potência

Cálculo do valor de k^n , onde n é um número inteiro não negativo.

$$k^n = \begin{cases} 1, & \text{se } n = 0 \\ k \times k^{n-1}, & \text{caso contrário} \end{cases}$$

```
double potencia(double k, int n) {  
    if (n == 0)  
        return 1;  
    else  
        return k * potencia(k, n - 1);  
}
```

Potência

A solução iterativa é mais eficiente, já que não há o custo adicional de criação e remoção de variáveis locais na pilha.

```
double potencia(double k, int n) {  
    double p = 1;  
    int i;  
  
    for (i = 1; i <= n; i++)  
        p = p * k;  
  
    return p;  
}
```

Podemos redefinir k^n (sendo n um inteiro não negativo) da seguinte forma:

$$k^n = \begin{cases} 1, & \text{se } n = 0 \\ k^{n/2} \times k^{n/2}, & \text{se } n \text{ for positivo e par} \\ k \times k^{\lfloor n/2 \rfloor} \times k^{\lfloor n/2 \rfloor}, & \text{se } n \text{ for positivo e ímpar} \end{cases}$$

Note que definimos a solução do caso mais complexo em termos de casos mais simples. Usando esta definição, podemos implementar uma função iterativa ou recursiva, ambas mais eficientes que as versões anteriores.

Potência

```
double potencia(double k, int n) {  
  
    if (n == 0)  
        return 1;  
    else {  
  
        if (n % 2 == 0)  
            return potencia(k, n / 2) * potencia(k, n / 2);  
        else  
            return k * potencia(k, n / 2) * potencia(k, n / 2);  
    }  
}
```


Potência

```
double potencia(double k, int n) {  
    double aux;  
  
    if (n == 0)  
        return 1;  
    else {  
        aux = potencia(k, n / 2);  
  
        if (n % 2 == 0)  
            return aux * aux;  
        else  
            return k * aux * aux;  
    }  
}
```

- Na nova versão do algoritmo, a cada chamada recursiva, o valor de n é dividido por 2. Ou seja, a cada chamada recursiva, o valor de n é reduzido, pelo menos, para a metade.
- Usando divisões inteiras, faremos no máximo $\lfloor \log_2 n \rfloor + 2$ chamadas recursivas.
- Por outro lado, a função iterativa original executa o laço n vezes.

Soma dos dígitos de um inteiro

Soma dos dígitos de um número inteiro positivo.

```
int soma_digitos(int n) {  
    if (n < 10)  
        return n;  
    else  
        return soma_digitos(n / 10) + (n % 10);  
}
```

Máximo Divisor Comum

O algoritmo de Euclides para o cálculo do Máximo Divisor Comum entre dois números inteiros não negativos x e y pode ser resumido na seguinte fórmula:

$$\text{mdc}(x, y) = \begin{cases} x, & \text{se } y = 0 \\ \text{mdc}(y, x \% y), & \text{se } y > 0 \end{cases}$$

Máximo Divisor Comum (versão iterativa)

```
int mdc(int x, int y) {  
    int aux;  
  
    while (y > 0) {  
        aux = y;  
        y = x % y;  
        x = aux;  
    }  
  
    return x;  
}
```

Máximo Divisor Comum (versão recursiva)

```
int mdc(int x, int y) {  
    if (y == 0)  
        return x;  
    else  
        return mdc(y, x % y);  
}
```

Maior elemento de um vetor

Maior elemento de um vetor v de $n > 0$ números inteiros.

```
int max(int v[], int n) {  
  
    if (n == 1)  
        return v[0];  
    else {  
  
        if (max(v, n - 1) > v[n - 1])  
            return max(v, n - 1);  
        else  
            return v[n - 1];  
    }  
}
```

Maior elemento de um vetor

Maior elemento de um vetor v de $n > 0$ números inteiros.

```
int max(int v[], int n) {  
    int x;  
  
    if (n == 1)  
        return v[0];  
    else {  
        x = max(v, n - 1);  
  
        if (x > v[n - 1])  
            return x;  
        else  
            return v[n - 1];  
    }  
}
```


Maior elemento de um vetor

Maior elemento de um vetor v de $n > 0$ números inteiros.

```
int max(int v[], int n) {  
    int x;  
  
    if (n == 1)  
        return v[0];  
    else {  
        x = max(v + 1, n - 1);  
  
        if (x > v[0])  
            return x;  
        else  
            return v[0];  
    }  
}
```

Número de caracteres de uma string

Calcula número de caracteres de uma string.

```
int strlen(char *s) {  
    if (*s == '\\0')  
        return 0;  
    else  
        return strlen(s + 1) + 1;  
}
```

Comparação de strings

Compara duas strings e retorna 0 se as strings são iguais, um valor negativo se a primeira string é lexicograficamente menor que a segunda ou um valor positivo se a primeira string é lexicograficamente maior que a segunda.

```
int strcmp(char *s, char *t) {  
    if (*s != *t)  
        return *s - *t;  
    if ((*s == '\\0') && (*t == '\\0'))  
        return 0;  
    else  
        return strcmp(s + 1, t + 1);  
}
```

Comparação de strings

Compara duas strings e retorna 0 se as strings são iguais, um valor negativo se a primeira string é lexicograficamente menor que a segunda ou um valor positivo se a primeira string é lexicograficamente maior que a segunda.

```
int strcmp(char *s, char *t) {  
    if (*s != *t)  
        return *s - *t;  
    if (*s == '\\0')  
        return 0;  
    else  
        return strcmp(s + 1, t + 1);  
}
```

Comparação de strings

Compara duas strings e retorna 0 se as strings são iguais, um valor negativo se a primeira string é lexicograficamente menor que a segunda ou um valor positivo se a primeira string é lexicograficamente maior que a segunda.

```
int strcmp(char *s, char *t) {  
    if ((*s != *t) || (*s == '\0'))  
        return *s - *t;  
    else  
        return strcmp(s + 1, t + 1);  
}
```

Busca de um caractere em uma string

Função que busca um caractere em uma string e retorna o ponteiro para ele, caso encontre, ou NULL, caso não.

```
char *strchr(char *s, char c) {  
    if (*s == c)  
        return s;  
    else  
        if (*s == '\\0')  
            return NULL;  
        else  
            return strchr(s + 1, c);  
}
```

Busca de um caractere em uma string

Função que busca um caractere em uma string e retorna o ponteiro para ele, caso encontre, ou NULL, caso não.

```
char *strchr(char *s, char c) {  
    if (*s == c)  
        return s;  
  
    if (*s == '\\0')  
        return NULL;  
  
    return strchr(s + 1, c);  
}
```

Cópia de strings

Função que retorna uma cópia da string *t* na string *s*.

```
void strcpy(char *s, char *t) {  
    *s = *t;  
  
    if (*s)  
        strcpy(s + 1, t + 1);  
}
```


Palíndromo

Verifica se uma string é um palíndromo.

```
int palindromo(char *s, int n) {  
    if (n <= 1)  
        return 1;  
    else  
        if (s[0] != s[n - 1])  
            return 0;  
        else  
            return palindromo(s + 1, n - 2);  
}
```

Palíndromo

Verifica se uma string é um palíndromo.

```
int palindromo(char *s, int n) {  
    if (n <= 1)  
        return 1;  
  
    if (s[0] != s[n - 1])  
        return 0;  
  
    return palindromo(s + 1, n - 2);  
}
```

Impressão de uma string em ordem inversa

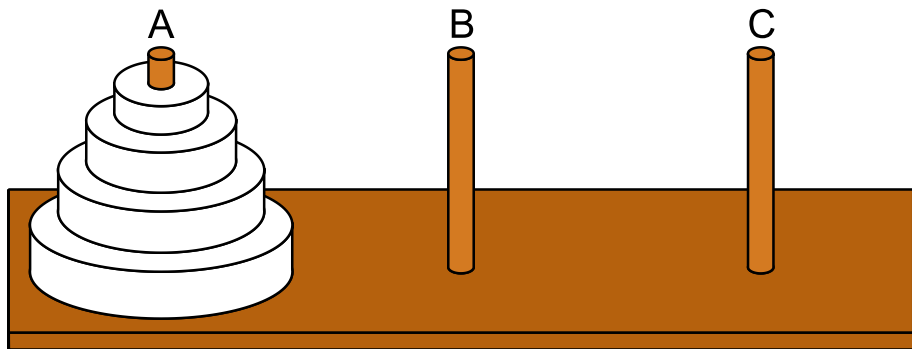
Imprime um string em ordem inversa.

```
void imprime_inversa(char *s) {  
    if (*s) {  
        imprime_inversa(s + 1);  
        printf("%c", *s);  
    }  
}
```

Torre de Hanói

- Considere n discos de diâmetros diferentes colocados em um pino A.
- O problema da Torre de Hanói consiste em transferir os n discos do pino A (inicial) para o pino C (final), usando um pino B como auxiliar.
- Entretanto, deve-se respeitar algumas regras:
 - ▶ Apenas o disco do topo de um pino pode ser movido.
 - ▶ Nunca um disco de diâmetro maior pode ficar sobre um disco de diâmetro menor.
- O problema foi descrito pela primeira vez no ocidente em 1883 pelo matemático francês Édouard Lucas, baseado numa lenda hindu, onde Brahma havia ordenado que os monges do templo de Kashi Vishwanath movessem uma pilha de 64 discos de ouro, segundo as regras previamente descritas.
- Quando todos os discos tivessem sido movidos, o mundo acabaria.

Torre de Hanói



Torre de Hanói

- Vamos usar indução para obter um algoritmo para este problema.

Teorema

É possível resolver o problema da Torre de Hanói com n discos.

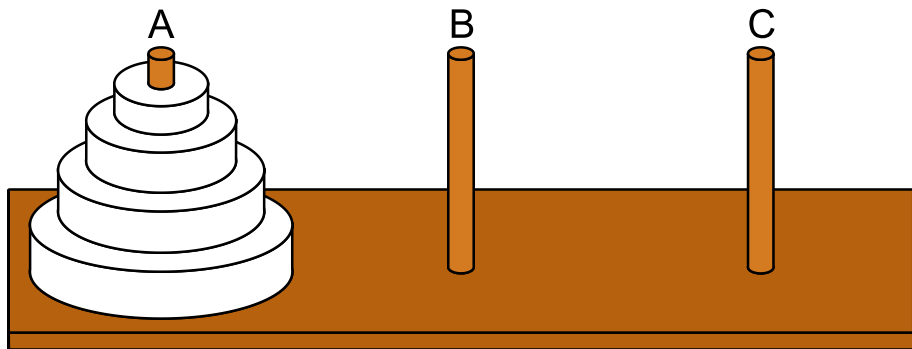
Torre de Hanói

Prova:

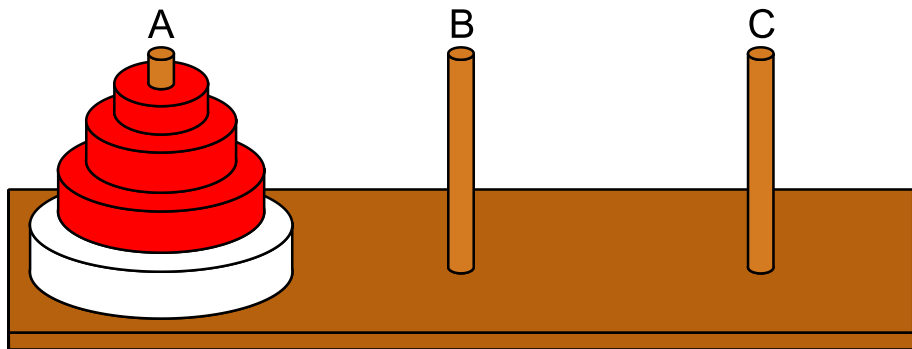
- Base da Indução: $n = 1$. Neste caso, temos apenas um disco. Basta mover este disco do pino A para o pino C.
- Hipótese de Indução: Sabemos como resolver o problema quando há $n - 1$ discos.
- Passo de Indução: Devemos resolver o problema para n discos assumindo que sabemos resolver o problema com $n - 1$ discos.
- Por hipótese de indução, sabemos mover os $n - 1$ primeiros discos do pino A para o pino B usando o pino C como auxiliar.
- Depois de movermos estes $n - 1$ discos, movemos o maior disco (que continua no pino A) para o pino C.
- Novamente, pela hipótese de indução, sabemos mover os $n - 1$ discos do pino B para o pino C usando o pino A como auxiliar.
- Com isso, temos uma solução para o caso em que há n discos.



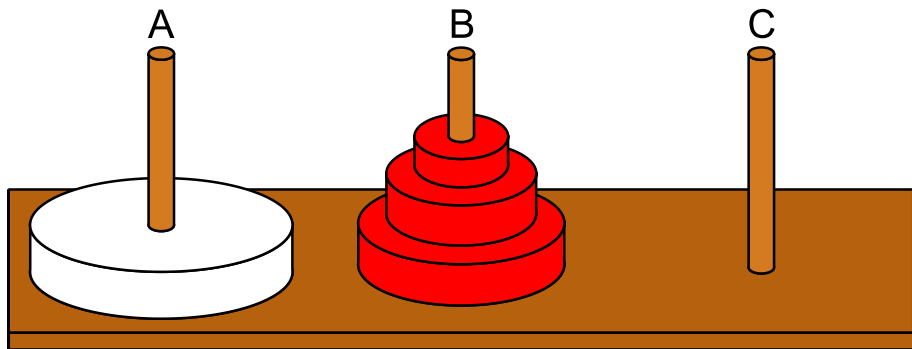
Torre de Hanói



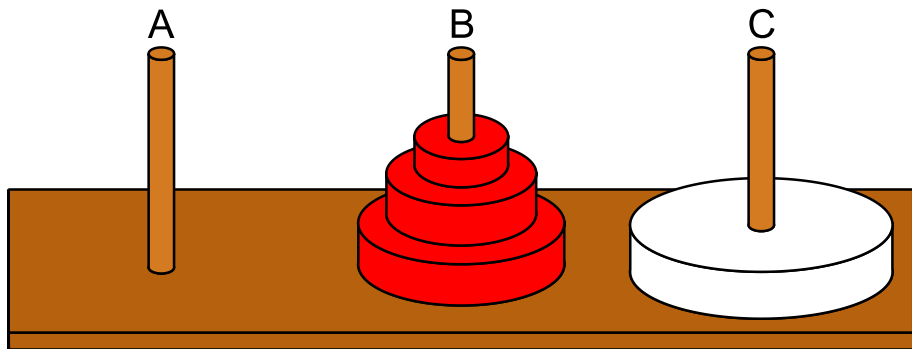
Torre de Hanói



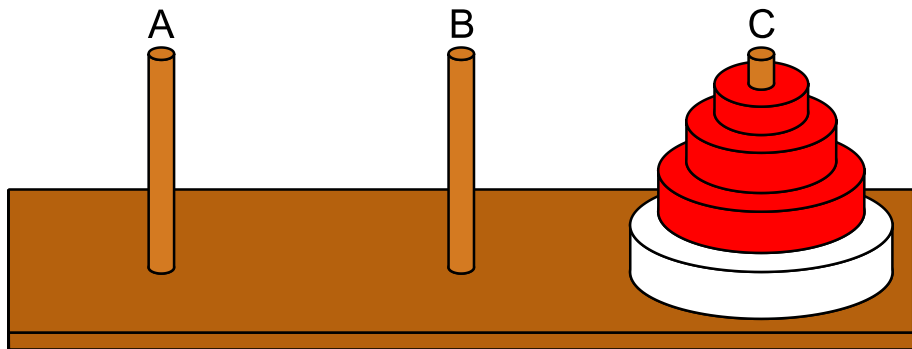
Torre de Hanói



Torre de Hanói



Torre de Hanói



Torre de Hanói

- Como solucionar o problema de forma recursiva:
 - ▶ Se $n = 1$ então mova o único disco de A para C.
 - ▶ Caso contrário ($n > 1$) desloque de forma recursiva os $n - 1$ primeiros discos de A para B, usando C como auxiliar.
 - ▶ Mova o último disco de A para C.
 - ▶ Mova, de forma recursiva, os $n - 1$ discos de B para C, usando A como auxiliar.

Torre de Hanói

```
#include <stdio.h>

void hanoi(int n, char inicial, char final, char auxiliar) {
    if (n == 1)
        printf("Mova o disco %d do pino %c para o pino %c\n", n, inicial, final);
    else {
        hanoi(n - 1, inicial, auxiliar, final);
        printf("Mova o disco %d do pino %c para o pino %c\n", n, inicial, final);
        hanoi(n - 1, auxiliar, final, inicial);
    }
}

int main() {
    int n;

    printf("Entre com o numero de discos: ");
    scanf("%d", &n);

    hanoi(n, 'A', 'C', 'B');

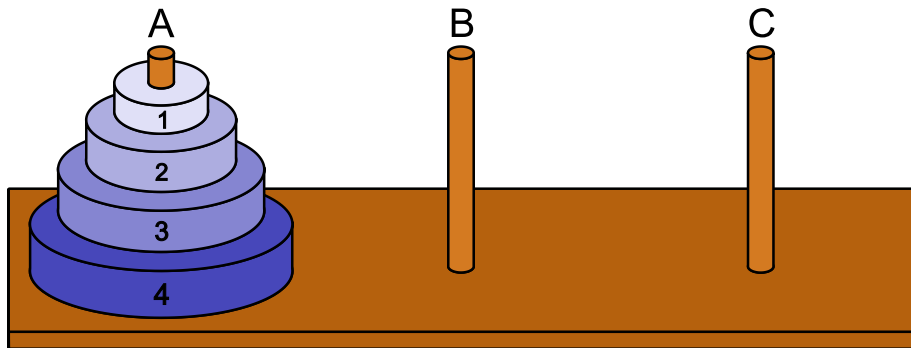
    return 0;
}
```

Torre de Hanói

- Solução para 4 discos:

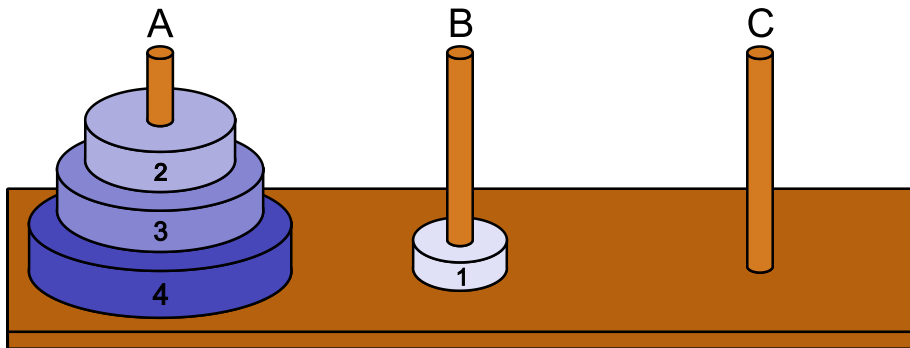
```
Mova o disco 1 do pino A para o pino B
Mova o disco 2 do pino A para o pino C
Mova o disco 1 do pino B para o pino C
Mova o disco 3 do pino A para o pino B
Mova o disco 1 do pino C para o pino A
Mova o disco 2 do pino C para o pino B
Mova o disco 1 do pino A para o pino B
Mova o disco 4 do pino A para o pino C
Mova o disco 1 do pino B para o pino C
Mova o disco 2 do pino B para o pino A
Mova o disco 1 do pino C para o pino A
Mova o disco 3 do pino B para o pino C
Mova o disco 1 do pino A para o pino B
Mova o disco 2 do pino A para o pino C
Mova o disco 1 do pino B para o pino C
```

Torre de Hanói



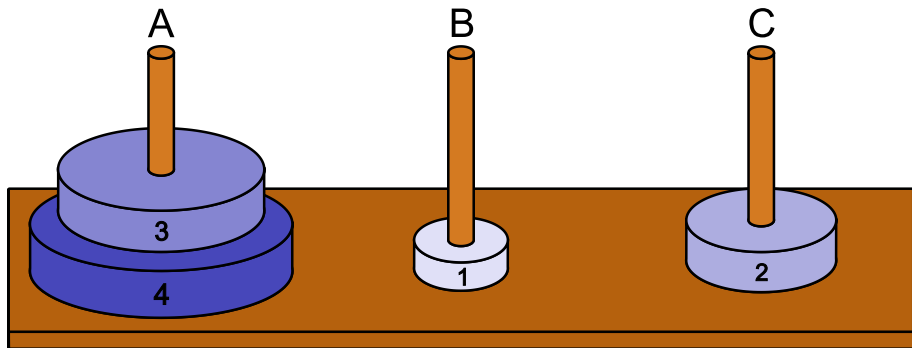
Configuração Inicial

Torre de Hanói



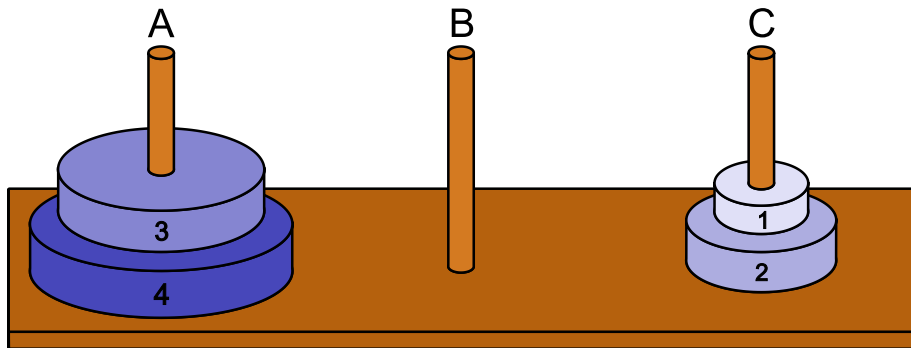
Mova o disco 1 do pino A para o pino B

Torre de Hanói



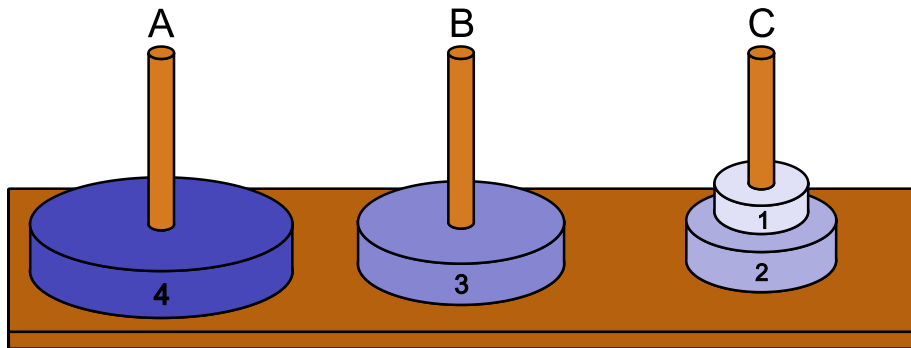
Mova o disco 2 do pino A para o pino C

Torre de Hanói



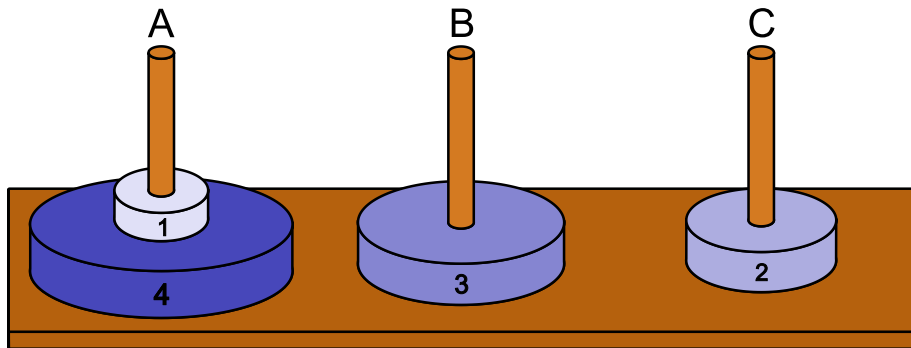
Mova o disco 1 do pino B para o pino C

Torre de Hanói



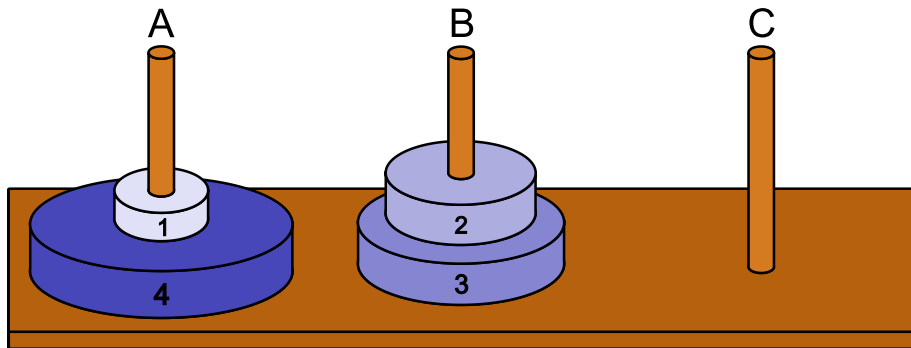
Mova o disco 3 do pino A para o pino B

Torre de Hanói



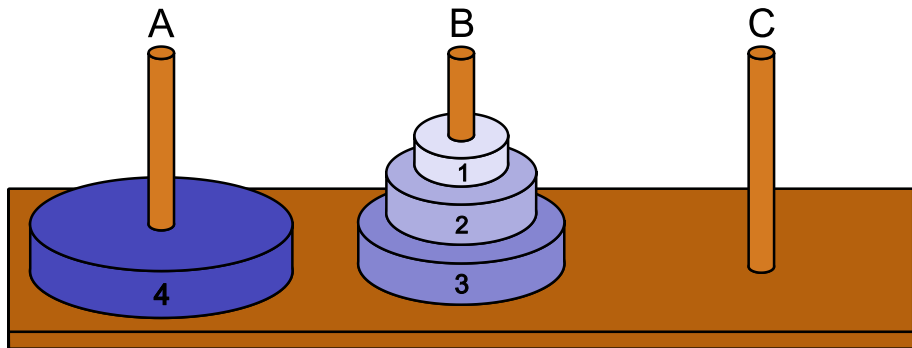
Mova o disco 1 do pino C para o pino A

Torre de Hanói



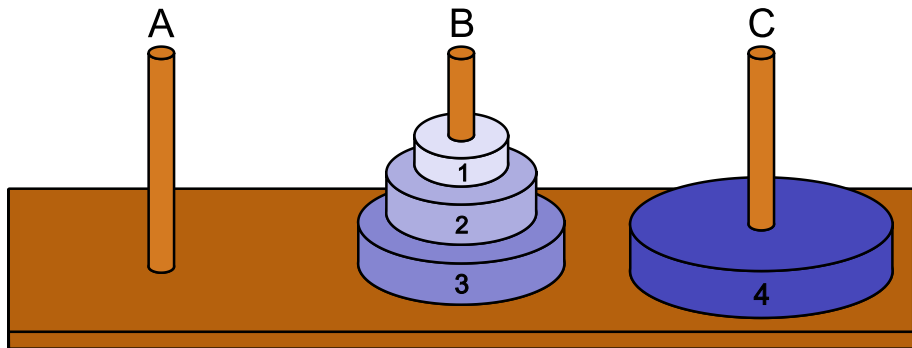
Mova o disco 2 do pino C para o pino B

Torre de Hanói



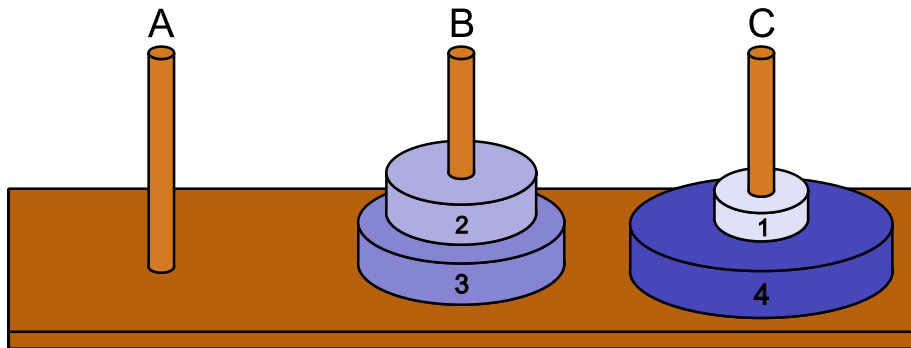
Mova o disco 1 do pino A para o pino B

Torre de Hanói



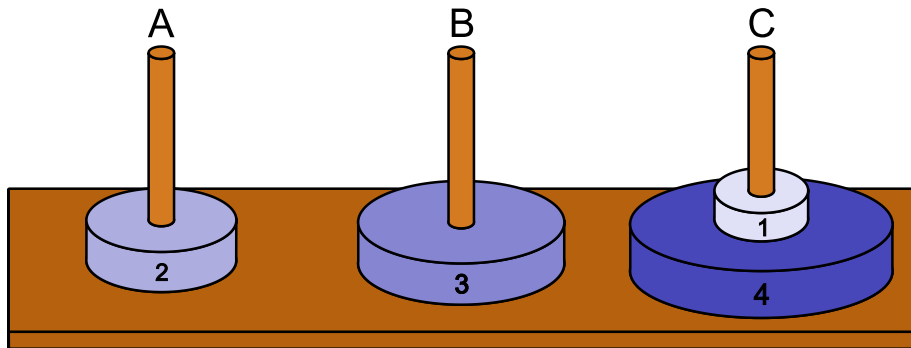
Mova o disco 4 do pino A para o pino C

Torre de Hanói



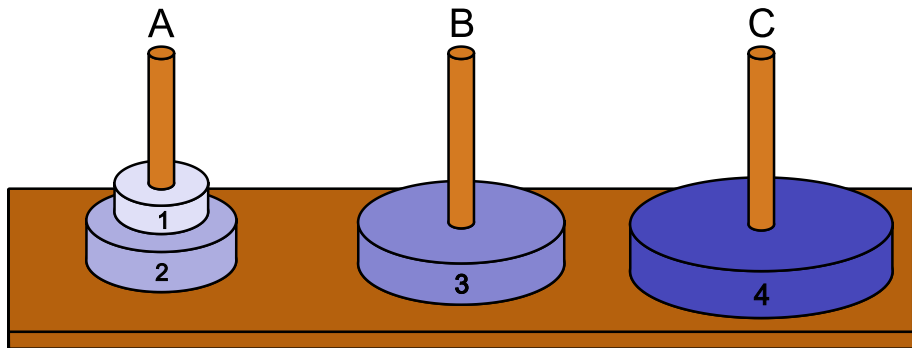
Mova o disco 1 do pino B para o pino C

Torre de Hanói



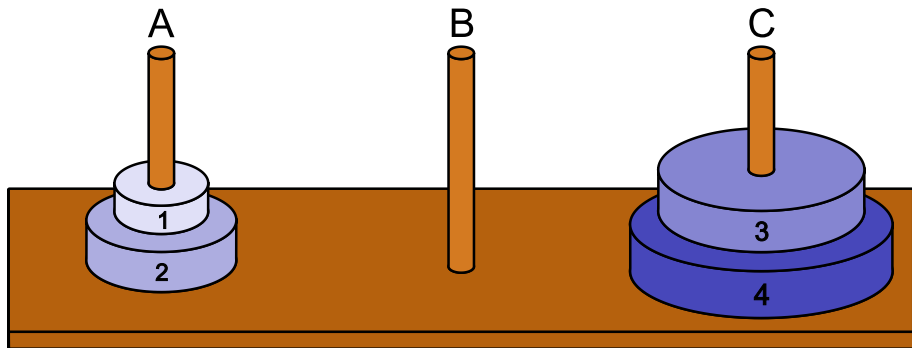
Mova o disco 2 do pino B para o pino A

Torre de Hanói



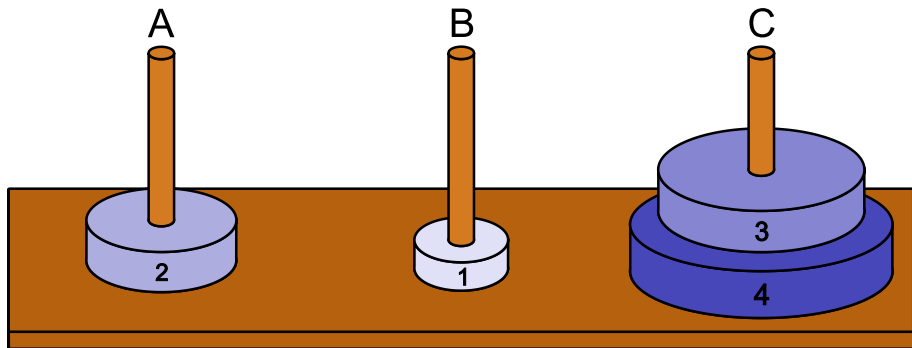
Mova o disco 1 do pino C para o pino A

Torre de Hanói



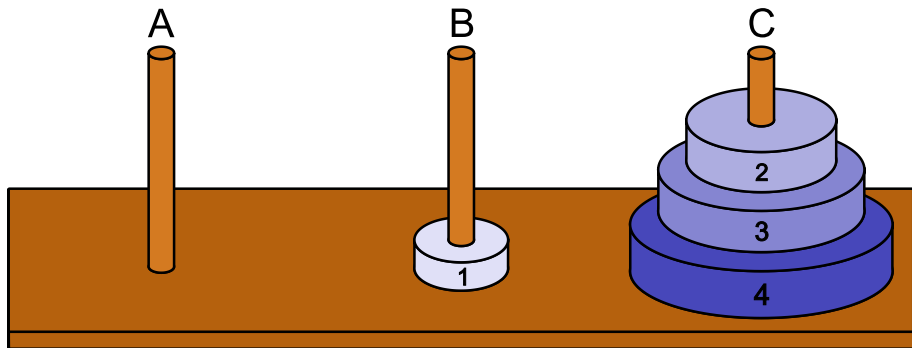
Mova o disco 3 do pino B para o pino C

Torre de Hanói



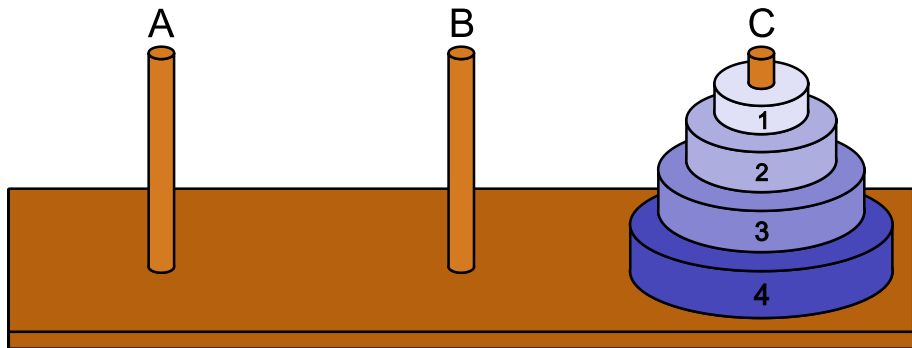
Mova o disco 1 do pino A para o pino B

Torre de Hanói



Mova o disco 2 do pino A para o pino C

Torre de Hanói



Mova o disco 1 do pino B para o pino C

Torre de Hanói

- Seja $T(n)$ o número de movimentos necessários para mover uma pilha de n discos.
- Claramente temos que:
 - ▶ $T(1) = 1$
 - ▶ $T(n) = 2T(n-1) + 1$
- O que nos permite deduzir que:
 - ▶ $T(2) = 2T(1) + 1 = 3$
 - ▶ $T(3) = 2T(2) + 1 = 7$
 - ▶ $T(4) = 2T(3) + 1 = 15$
 - ▶ $T(5) = 2T(4) + 1 = 31$
 - ▶ ...
 - ▶ $T(n) = 2^n - 1$
- No caso de 64 discos são necessários 18.446.744.073.709.551.615 movimentos ou, aproximadamente, 585 bilhões de anos, se cada movimento puder ser feito em um segundo.

Exercícios

- O que será impresso pelo programa abaixo?

```
#include <stdio.h>

void imprime(int v[], int i, int n);

int main() {
    int vet[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    imprime(vet, 0, 10);
    printf("\n");
}

void imprime(int v[], int i, int n) {
    if (i < n) {
        printf("%d ", v[i]);
        imprime(v, i + 1, n);
    }
}
```

Exercícios

- O que será impresso pelo programa abaixo?

```
#include <stdio.h>

void imprime(int v[], int i, int n);

int main() {
    int vet[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    imprime(vet, 0, 10);
    printf("\n");
}

void imprime(int v[], int i, int n) {
    if (i < n) {
        imprime(v, i + 1, n);
        printf("%d ", v[i]);
    }
}
```

Exercícios

- O que será impresso pela chamada `imprimir(5)`?

```
void imprimir(int i) {  
    int j;  
  
    if (i > 0) {  
        imprimir(i - 1);  
        for (j = 1; j <= i; j++)  
            printf("*");  
        printf("\n");  
    }  
}
```

Exercícios

- Escreva uma função recursiva que, dado um número real x e um inteiro k qualquer, calcule o valor de x^k .
- Escreva uma função recursiva que, dado um número inteiro positivo n , imprima a representação binária de n .
- Escreva uma função recursiva que, dada uma string s e um caractere c , conte o número de ocorrência do caractere c na string s .
- Escreva uma função recursiva que, dado um vetor v de n números inteiros ($n \geq 1$), determine o menor e o maior valor armazenado no vetor.
- Escreva uma função recursiva que, dado um vetor v de n números inteiros ordenados ($n \geq 1$) e um inteiro x , retorne, usando uma busca binária, o índice de x no vetor ou o valor -1 , caso x não pertença ao vetor.

Exercícios

- Escreva versões recursivas das funções `indiceMenor` e `insertion`, estudadas na aula sobre ordenação.
- Usando as funções acima, escreva versões recursivas para o Selection Sort e para o Insertion Sort.
- Importante: todas as funções acima não devem usar qualquer comando de repetição (`for`, `while`, etc).