PANC: Projeto e Análise de Algoritmos

Aula 06: Algoritmos Recursivos: Problemas e Análise da Complexidade

Breno Lisi Romano

http://sites.google.com/site/blromano

Instituto Federal de São Paulo – IFSP São João da Boa Vista
Bacharelado em Ciência da Computação – 3º Semestre

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Sumário

- Revisão de Conteúdo
- Algoritmos Recursivos
  - Definições e Exemplos Simples
- Recursão em Cauda vs. Recursão Crescente
- Recursão vs. Iteração
- Análise de Problema Recursivos
  - Lógica
  - Demonstrações
  - Análise da Complexidade
- Recorrências
- Quando não utilizar Recursividade

Recapitulando... (1)

- Para determinar a **complexidade de tempo T(n):**
  - Considerar que **entradas (n) aumentam indefinidamente**
  - É uma **análise teórica** → Não considera aspectos de hardware
  - Os **termos** de **mais baixa ordem** e **constantes** são **ignorados**
- Utilizam-se **05 notações: O, Θ, Ω, o, ω**
  - **f(n)** ∈ **O(g(n))** se existir n0 e c tal que **f(n) ≤ c g(n)**, para todo n ≥ n0
  - **f(n)** ∈ **Ω(g(n))** se existir n0 e c tal que **f(n) ≥ c g(n)**, para todo n ≥ n0
  - **f(n)** ∈ **Θ(g(n))** se existir n0, c1 e c2 tal que **c1g(n) ≤ f(n) ≤ c2g(n)**, para todo n ≥ n0
  - **f(n)** ∈ **o(g(n))** se existir c e n0 tal que **f(n) < c g(n)**, para todo n ≥ n0
  - **f(n)** ∈ **ω(g(n))** se existir c e n0 tal que **f(n) > c g(n)** para todo n ≥ n0
- **Relação de Dominância** de Funções Tipicamente Utilizadas:
  - $n! \gg 2^n \gg n^3 \gg n^2 \gg n.\lg n \gg n \gg \lg n \gg 1$

Recapitulando... (2)

- Paradigma de **Divisão e Conquista**:
  - **Objetivo**: Resolver **problemas** no qual tentamos **simplificar a solução do problema** original **dividindo-o em subproblemas** menores e **resolvendo-os** (ou "conquistando-os") separadamente
    - **Dividir** o problema original em **subproblemas** – normalmente com a metade (ou algo próximo disto) do tamanho do problema original, porém com a mesma estrutura
    - **Conquistar**, ou determinar a solução dos subproblemas, comumente, de maneira recursiva – que agora se tornam mais "fáceis"
    - Se necessário, **combinar** as soluções dos subproblemas para produzir a **solução completa** para o problema original
- Precisamos praticar a **terminologias apropriadas** para **representar** a complexidade **T(n)** de algoritmos recursivos → **AULA DE HOJE**

Algoritmos Recursivos (1)

- 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 de instruções finitas**
- A ideia é **aproveitar a solução** de um ou mais subproblemas com **estrutura semelhante para resolver o problema original**
- Geralmente adota-se a **recursividade** para **auxiliar na aplicação do paradigma de Divisão e Conquista**
- **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 (2)

- **Projeto:**
  - Um algoritmo recursivo é composto, em sua forma mais simples, de uma **condição de parada** e de um **passo recursivo**
- **Passo Recursivo:**
  - Realiza as **chamadas recursivas** e **processa os diferentes valores de retorno**, quando adequado
  - A **ideia** é **associar** um **parâmetro n** e **realizar o passo recursivo sobre n−1** (ou outra fração de n)
- **Condição de Parada:**
  - Garante que a **recursividade é finita**, geralmente, definida sobre um **caso base**
  - Por exemplo, a condição de Parada do *MergeSort* é quando o Array possui tamanho n = 1
  - Outro exemplo, a condição n > 0 garante a parada com n positivo

Algoritmos Recursivos (3)

- **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 (4)

- 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**
- O **registro** de **ativação armazena** os **parâmetros** e **variáveis locais** da função, bem como o **"ponto de retorno"** no programa 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**

Algoritmos Recursivos – Exemplo 01

- **Problema:** Suponhamos que queremos imprimir recursivamente todos os primeiros cinco números inteiros
- **Solução:**

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

- **Ilustração da Execução na Pilha de Memória:**

1	recursiveFunction (0)				
2	printf (0)				
3		recursiveFunction (0+1)			
4		printf (1)			
5			recursiveFunction (1+1)		
6			printf (2)		
7				recursiveFunction (2+1)	
8				printf (3)	
9					recursiveFunction (3+1)
10					printf (4)

Algoritmos Recursivos – Exemplo 2

- **Problema:** Suponhamos que queremos imprimir recursivamente todos os primeiros cinco números inteiros, porém, em outra versão
- **Solução:**

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

- **Ilustração da Execução na Pilha de Memória:**

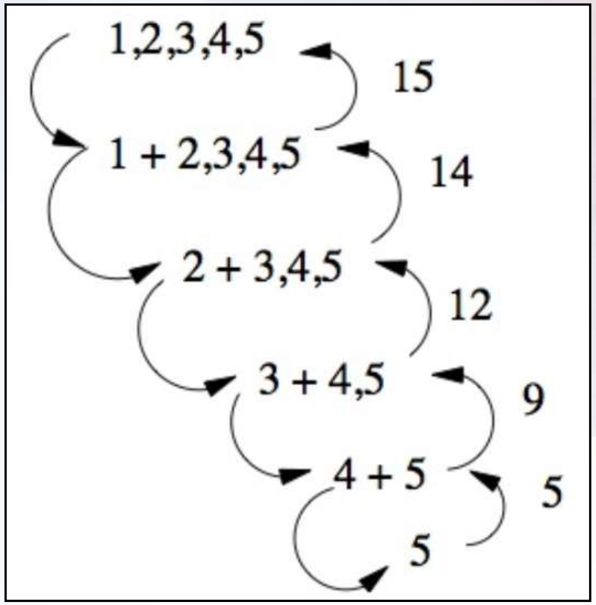
1	recursiveFunction (0)				
2		recursiveFunction (0+1)			
3			recursiveFunction (1+1)		
4				recursiveFunction (2+1)	
5					recursiveFunction (3+1)
6					printf (4)
7				printf (3)	
8			printf (2)		
9		printf (1)			
10	printf (0)				

Algoritmos Recursivos – Exemplo 3

- **Problema:** Suponhamos que queremos somar recursivamente todos os números inteiros entre m e n, inclusive
- **Solução:**

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

- **Ilustração da Execução utilizando uma Árvore de Recursão para m=1 e n=5:**



Algoritmos Recursivos – Cauda vs. Crescente (1)

- 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

Algoritmos Recursivos – Cauda vs. Crescente (2)

- Exemplo para Comparação da **Recursão em Cauda e 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);
}
```

Problema 01: Fatorial (1)

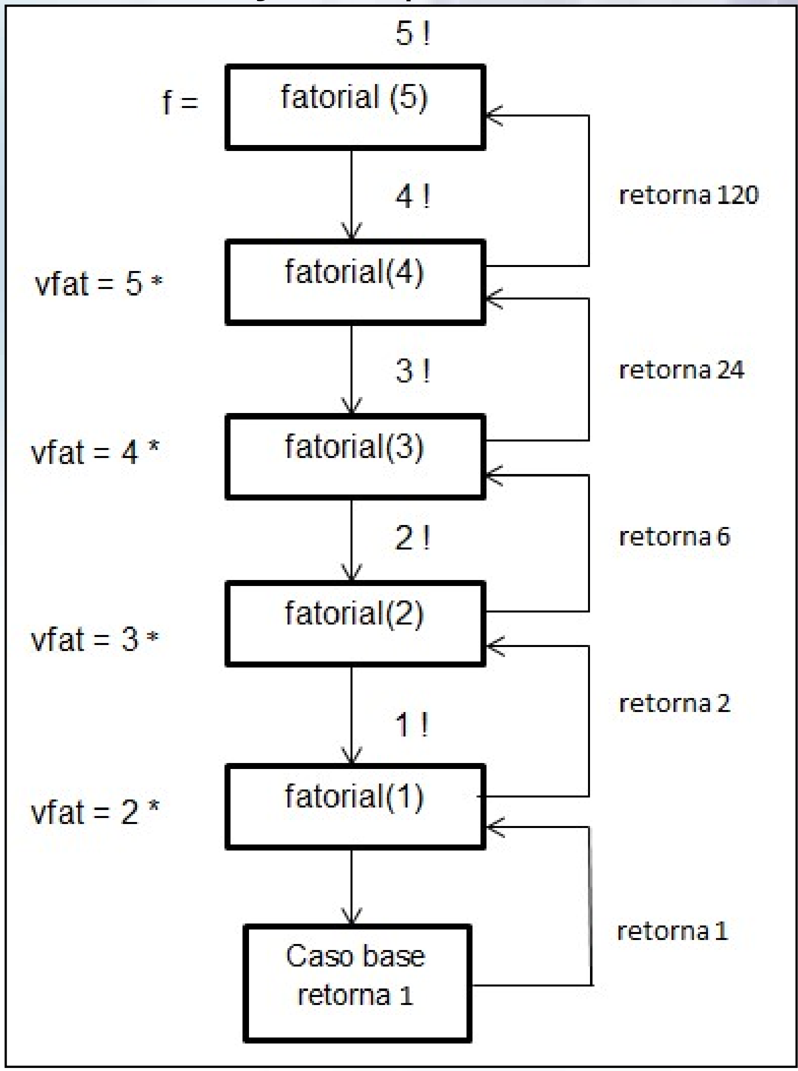
- **Problema:**
  - Um exemplo clássico de algoritmo recursivo é o cálculo de fatorial
  - 0! = 1! = 1
  - $n! = n \times (n - 1)!$
- **Exemplo:**
  - **5!** → 5 . 4 . 3 . 2 . 1 = **120**
    - **5! → 5 x 4! / 4! → 4 x 3! / 3! → 3 x 2! / 2! → 2 x 1! (1)**
- **Resolução Recursiva:**

```
long fatorial(int n)
{
    if(n<=1)
        return 1;
    return n * fatorial(n-1);
}
```

Problema 01: Fatorial (2)

- **Como é a execução passo a passo para fatorial(5) ?**
  - Simulação da recursividade, condição de parada e retorno da recursividade!

```
long fatorial(int n)
{
    if(n<=1)
        return 1;
    return n * fatorial(n-1);
}
```



Problema 01: Fatorial (3)

```
long fatorial(int n)
{
    if(n<=1)
        return 1;
    return n * fatorial(n-1);
}
```

- **Análise de Complexidade do Fatorial Recursivo:**
  - Seja **T(n)** a **complexidade de tempo** do fatorial recursivo:
    - **Caso Base**: T(1) = a
    - **Passo Recursivo**: T(n) = T(n-1) + b, para n > 1

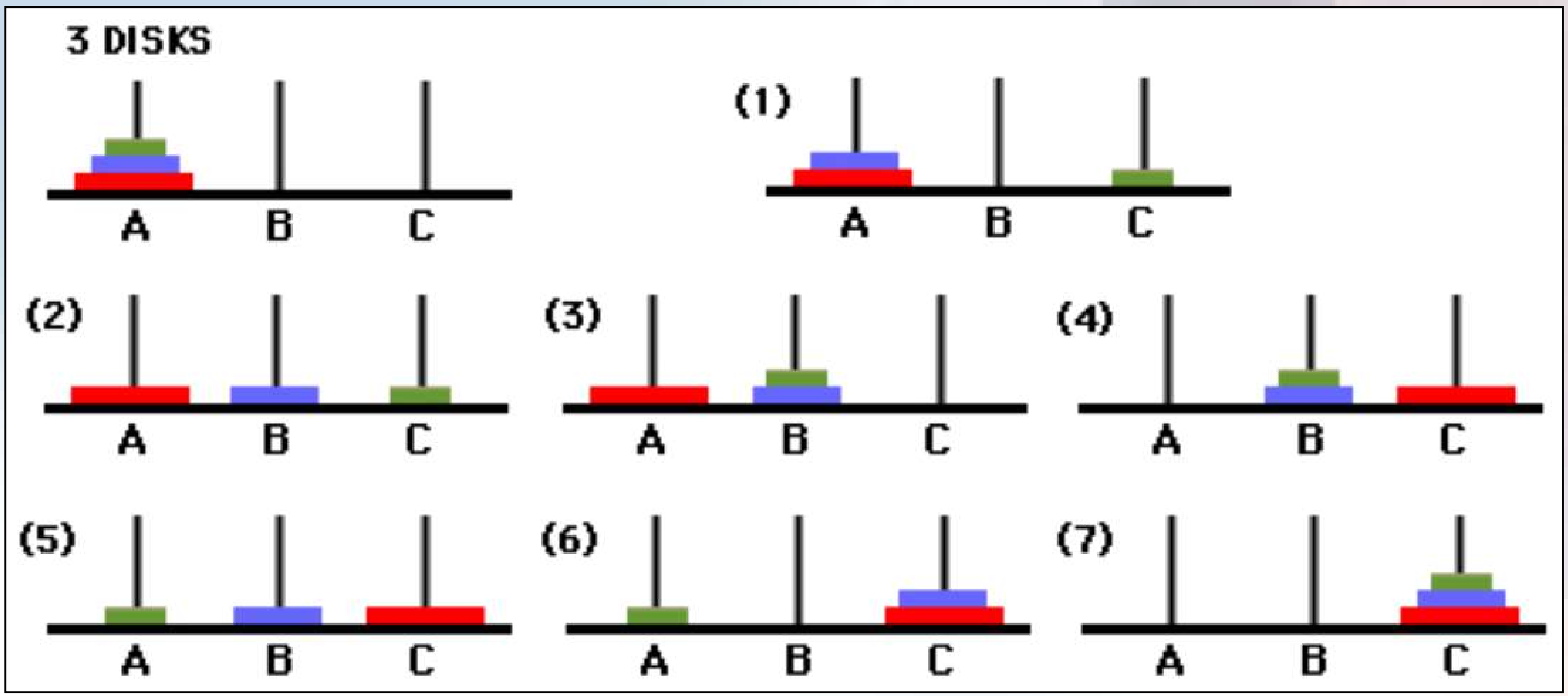
- **Resolvendo a Recorrência**:
  - **T(1) = a**
  - **T(2) =** T(1) + b = **a + b**
  - **T(3) =** T(2) + b = **a + 2b**
  - **T(4) =** T(3) + b = **a + 3b**
  - Generalizando, a **forma fechada** para a **Recorrência** é **T(n) = a + (n − 1)b**
  - Logo, sendo a e b constantes → **T (n)** ∈ **O(n)**

Problema 02: Torre de Hanói (1)

- **Problema:**
  - Em 1883, o matemático francês Edouard Lucas criou um jogo chamado Torre de Hanoi
  - Na versão clássica, é necessário **mover n discos** de **diâmetros diferentes**, **um de cada vez**, de uma **torre de origem** para **a torre de destino**, usando ainda uma **torre auxiliar**.
  - **Regra: Não é permitido posicionar um disco maior sobre outro menor**

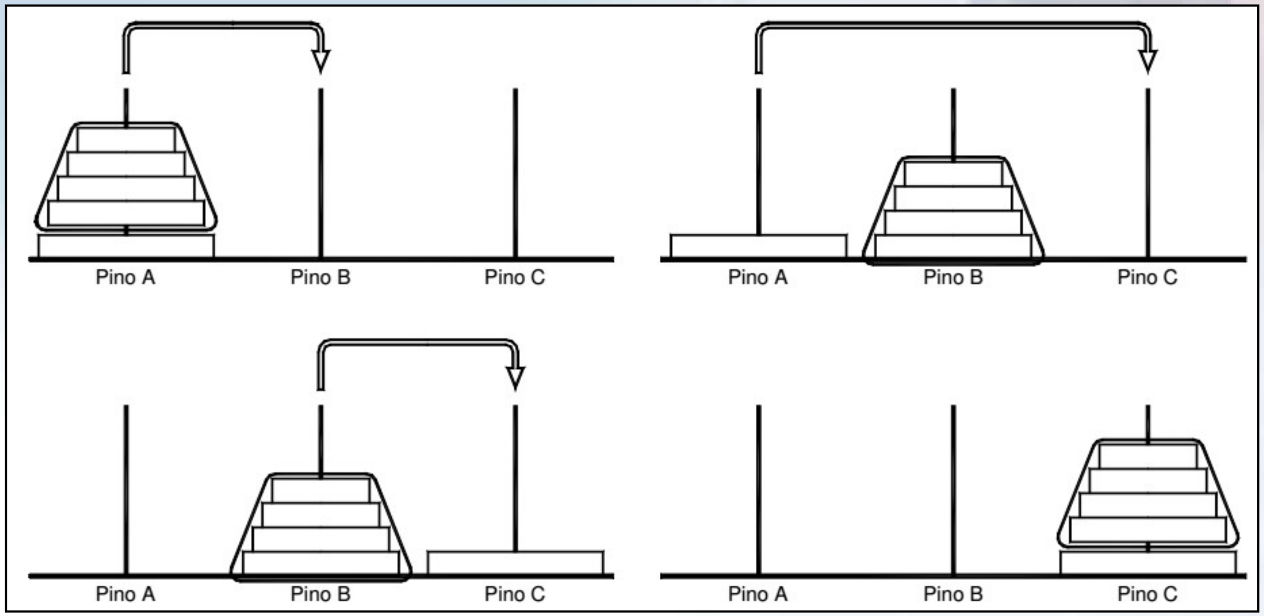
Problema 02: Torre de Hanói (2)

- **Exemplo:**
  - **Para n = 3** (quantidade de discos)
  - **Torre de Origem:** A
  - **Torre de Destino:** C



Problema 02: Torre de Hanói (3)

- **Método de Solução Generalizado:**
  - Para n discos, temos:
    - **Mova os n-1 discos** do topo da **origem** para **auxiliar**
    - **Mova o maior disco** da **origem** para o **destino**
    - **Mova os n-1 discos** de **auxiliar** para o **destino**



- **Insight:**
  - **Divida o problema original sucessivamente em subproblemas de tamanho n-1 e resolva recursivamente para tamanhos de n crescentes**

Problema 02: Torre de Hanói (4)

- **Resolução Recursiva:**

```c
void hanoi(int n, int origem, int destino, int auxiliar) {

    if (n > 0) {
        hanoi(n-1, origem, auxiliar, destino);
        mover(origem, destino);
        hanoi(n-1, auxiliar, destino, origem);
    }

}
```

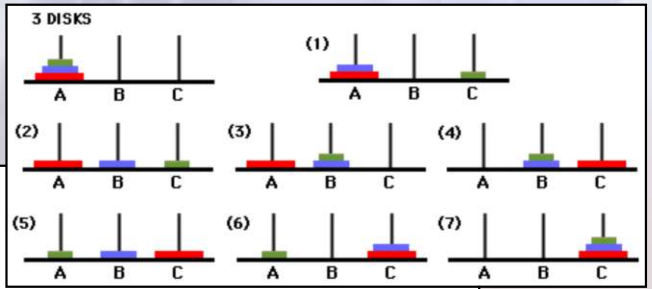
C:\Users\blromano\Desktop\Aula06-ProblemaHanoi\bin\D...

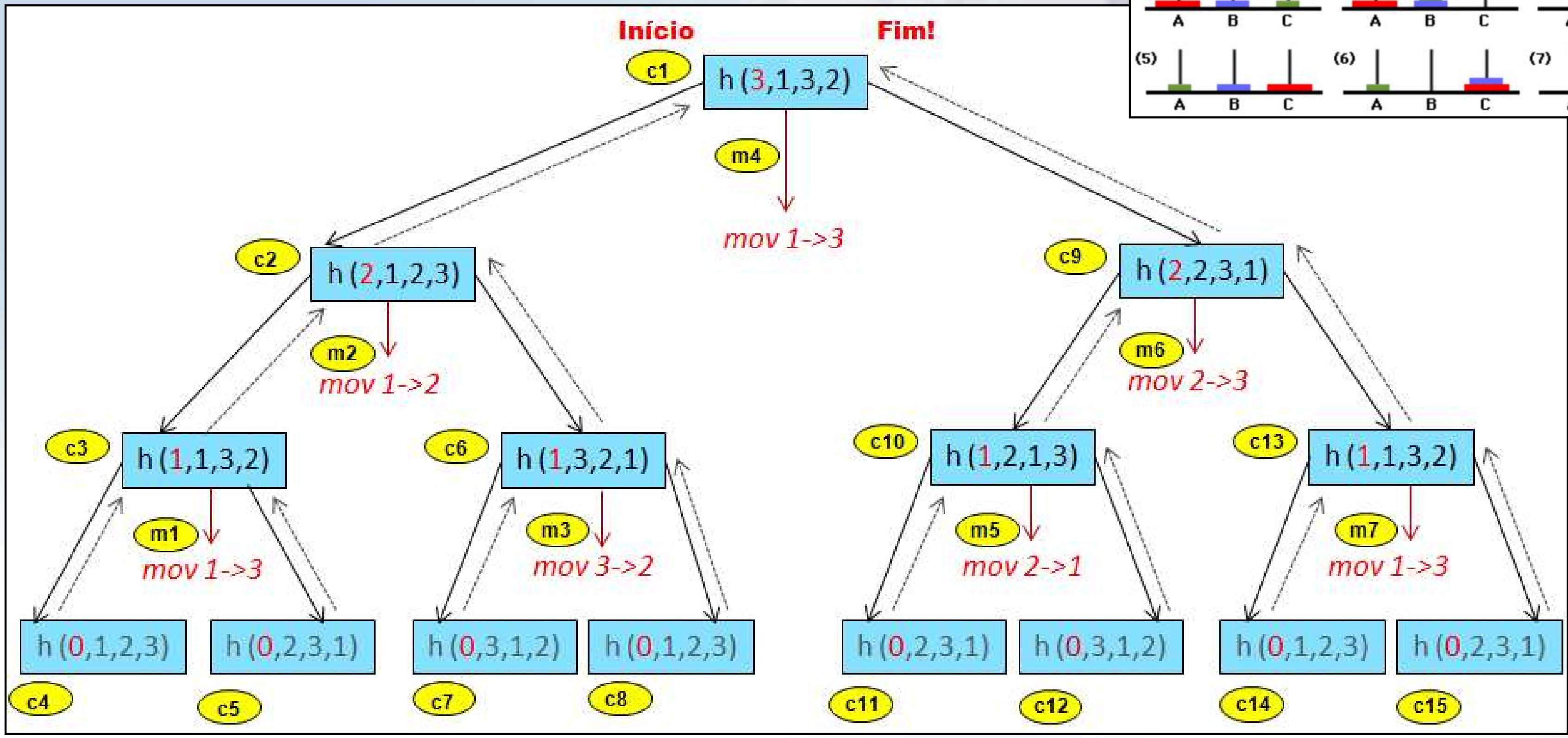
```
Entrar com o n·mero de discos: 3

Mover disco da Torre 1 para a Torre 3
Mover disco da Torre 1 para a Torre 2
Mover disco da Torre 3 para a Torre 2
Mover disco da Torre 1 para a Torre 3
Mover disco da Torre 2 para a Torre 1
Mover disco da Torre 2 para a Torre 3
Mover disco da Torre 1 para a Torre 3
Process returned 0 (0x0)   execution time : 1.471 s
Press any key to continue.
```

Problema 02: Torre de Hanói (5)

- **Como é a execução passo a passo para hanoi(3,1,3,2) ?**
  - Simulação da recursividade, condição de parada e retorno da recursividade!
  - **Sugestão: Utilizar uma Árvore de Recursão**
  - n = 3 discos, Torre Origem = 1, Torre Destino = 3 e Torre Auxiliar = 2

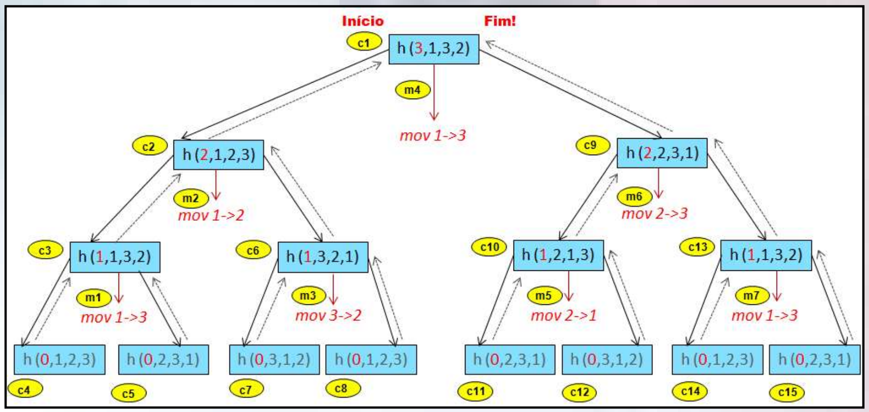
```
void hanoi(int n, int origem, int destino, int auxiliar) {

    if (n > 0) {
        hanoi(n-1, origem, auxiliar, destino);
        mover(origem, destino);
        hanoi(n-1, auxiliar, destino, origem);
    }

}
```





Problema 02: Torre de Hanói (6)

- **Complexidade de Tempo – T(n):**
  - T(n) corresponde ao **número mínimo de movimentos necessários** para **mover todos os n discos** para a **torre de destino** de acordo com o enunciado do problema
  - Por inspeção, temos que:
    - T(0) = 0
    - T(1) = 1
    - T(2) = 3
    - T(3) = 7
    - .....
  - Observe a árvore do slide anterior?



- **Fórmula de Recorrência:**
  - **Caso Base:** T(1) = 1
  - **Passo Recursivo:** T(n) = 2T(n-1) + 1, n>1

Problema 02: Torre de Hanói (7)

- **Resolvendo a Fórmula de Recorrência $T(n) = 2T(n-1) + 1, n \geq 1$**
  - **$T(1) = 1$**
  - **$T(2) =$** $2T(1) + 1 = 2 + 1$ **$= 3$**
  - **$T(3) =$** $2T(2) + 1 = 6 + 1$ **$= 7$**
  - **$T(4) =$** $2T(3) + 1 = 14 + 1$ **$= 15$**
  - **$T(5) =$** $2T(4) + 1 = 30 + 1$ **$= 31$**
  - **.....**
- **Generalizando, temos a Forma Fechada como sendo:**
  - **$T(n) = 2^n - 1$**
  - **$T(n) \in O(2^n)$**

Problema 02: Torre de Hanói (8)

- **Provando a forma fechada por Indução Matemática**
  - **Fórmula: $T(n) = 2^n – 1$**
  - **Caso base:**
    - Temos que $T(0) = 2^0 - 1 = 0$ e $T(1) = 2^1 - 1 = 1$, conforme descrito na recorrência
  - **Indução:**
    - Faremos a indução em n. Supomos que a forma fechada seja válida para todos os valores até $n - 1$, ou seja, $T(n - 1) = 2^{n-1} - 1$
    - Provaremos que a forma fechada também é válida para T(n):
      - $T(n) = 2T(n - 1) + 1$
      - $T(n) = 2(2^{n-1} - 1) + 1$
      - $T(n) = 2^n - 2 + 1$
      - $T(n) = 2^n - 1$
    - Portanto, a forma fechada também é válida pra n

Problema 03: Busca Binária (1)

- **Problema:**
  - O usuário deve fornecer um array ordenado de tamanho N e um valor x a ser procurado no array
    - Deve-se partir do pressuposto que o array de números inteiros encontra-se ordenado
- **Lógica para Resolução:**
  - A cada iteração deve-se chamar a função recursivamente reduzindo o espaço de busca dividindo o array pela metade, através das variáveis de início e fim do sub-array
  - Necessita-se na Função:
    - Array
    - Limite Inferior do Array após a Divisão
    - Limite Superior do Array após a Divisão
    - Valor Procurado (Alvo)

Problema 03: Busca Binária (2)

- **Exemplo:**
  - Simular para o Array A[] = {-8, -5, 1, 4, 14, 21, 23, 54, 67, 90}
  - Valor Procurado: 4

	0	1	2	3	4	5	6	7	8	9
V	-8	-5	1	4	14	21	23	54	67	90

elem | 4 | Elemento procurado

	0	1	2	3	4	5	6	7	8	9	
meio=4	-8	-5	1	4	**14**	21	23	54	67	90	Valor é menor: buscar no início
meio=1	-8	**-5**	1	4	14	21	23	54	67	90	Valor é maior: buscar no final
meio=2	-8	-5	**1**	4	14	21	23	54	67	90	Valor é maior: buscar no final
meio=3	-8	-5	1	**4**	14	21	23	54	67	90	Valor é igual: terminar a busca

Problema 03: Busca Binária (3)

- **Resolução Recursiva:**

```
int BuscaBinaria (int x, int Vet[], int inicio, int fim)
{
    int meio = (inicio + fim)/2;
    if (Vet[meio] == x)
        return meio;
    if (inicio >= fim)
        return -1; // não encontrado
    else
        if (Vet[meio] < x)
            return BuscaBinaria(x, Vet, meio+1, fim);
        else
            return BuscaBinaria(x, Vet, inicio, meio-1);
}
```

Problema 03: Busca Binária (4)

```
int BuscaBinaria (int x, int Vet[], int inicio, int fim)
{
    int meio = (inicio + fim)/2;
    if (Vet[meio] == x)
        return meio;
    if (inicio >= fim)
        return -1; // não encontrado
    else
        if (Vet[meio] < x)
            return BuscaBinaria(x, Vet, meio+1, fim);
        else
            return BuscaBinaria(x, Vet, inicio, meio-1);
}
```

- **Análise de Complexidade – T(n):**
  - Seja **T(n)** a **complexidade de tempo** da busca binária:
    - **Caso Base**: T(1) = a
    - **Passo Recursivo**: T(n) = T(n/2) + b, para n > 1

- **Resolvendo a Recorrência**:
  - **Intuitivamente:** Quantas vezes se consegue dividir n por 2 até chegar em 1?
    - T(1) = a (Caso Base)
    - T(2) = T(1) + b → $T(2^1) = T(2^0) + b$
    - T(4) = T(2) + b → $T(2^2) = T(2^1) + b$
    - T(8) = T(4) + b → $T(2^3) = T(2^2) + b$
    - T(16) = T(8) + b → $T(2^4) = T(2^3) + b$
    - ...
    - $\mathbf{T(2^k) = T(2^{k-1}) + b}$

Qual a profundidade? Ou seja, qual a altura da árvore?

Resposta: k

Complexidade: T(n) = k.b + a

Obs: k custos b + a (custo do caso base)

Problema 03: Busca Binária (5)

- **Análise de Complexidade – T(n):**
  - Para encontrar o valor da Complexidade **T(n) = k.b + a**
  - **Precisamos Resolver a Recorrência – Continuação...:**
    - Temos que: **$T(n) = T(n/2) + b$**
    - E também: **$T(2^k) = T(2^{k-1}) + b$**
    - **Assim, assumimos que: $n = 2^k$**
    - **Logo:**
      - Aplicando lg em ambos os lados, temos:
      - **$k = \lg n$**
  - **Substituindo, temos:**
    - **$T(n) = b. \lg n + a$**
    - **$T(n) \in O(\lg n)$ → Logarítmica**

Recorrências (1)

- **Encontrando formas fechadas:**
  - Pode-se encontrar a forma fechada para o valor de T(n) normalmente em três etapas:
    1. Analisar os pequenos casos de T(n), o que pode nos fornecer *insights*
    2. Encontrar e provar uma recorrência para o valor de T(n)
    3. Encontrar e provar uma forma fechada para a recorrência

Recorrências (2)

- **Definição:**
  - **Uma recorrência é uma equação ou desigualdade que descreve uma função em termos de seu próprio valor em entradas menores**
- **Aplicação e Resolução:**
  - A **complexidade de algoritmos recursivos** pode ser frequentemente descrita através de **recorrências**
  - Geralmente, recorremos ao **Teorema Mestre** para **resolver** estas **recorrências**
  - Em **casos** em que o **Teorema Mestre não se aplica**, a **recorrência** deve ser **resolvida** de **outras maneiras**
  - **Resolver** uma **recorrência significa eliminar** as **referências** que **ela faz a si mesma**
  - **Três dos métodos mais comuns** para resolução de recorrências são o método de **substituição**, o método de **árvore de recursão** e o **teorema mestre**
  - **Próxima aula...... Resolução de Recorrências**

Algoritmos Recursivos – Observações (1)

- A maioria dos **algoritmos recursivos** possuem uma **versão iterativa equivalente**, bastando utilizar uma **pilha explícita**
- 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
- **Erros de implementação fatalmente geram loop infinito** ou **estouro** de **pilha**

Algoritmos Recursivos – Observações (2)

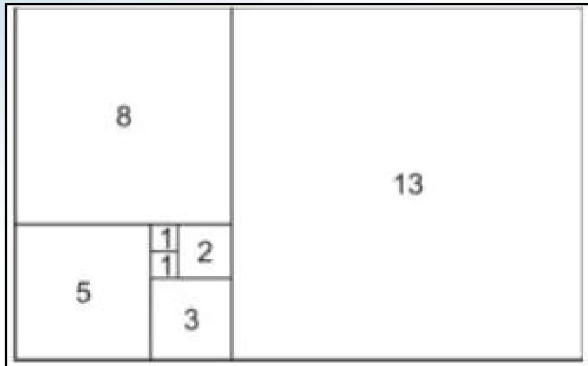
- **Estouro de Pilha:**
  - Normalmente, a **pilha de execução** possui uma **região limitada de memória**, geralmente no **início do próprio programa**
  - Quando um **há uso de mais memória do que o suportado**, ocorre o **estouro da pilha**, que implica na **suspensão da execução** do programa
  - 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

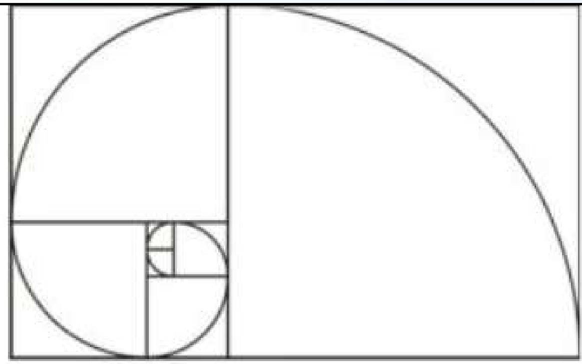
Problema 04: Fibonacci (1)

- **Problema:**
  - A **sequência [0, 1, 1, 2, 3, 5, 8, 13, 21, ...]** é conhecida como **Série de Fibonacci** e tem aplicações teóricas e práticas, na medida em que alguns padrões na natureza parecem seguí-la. Pode ser obtida através da definição recursiva:

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

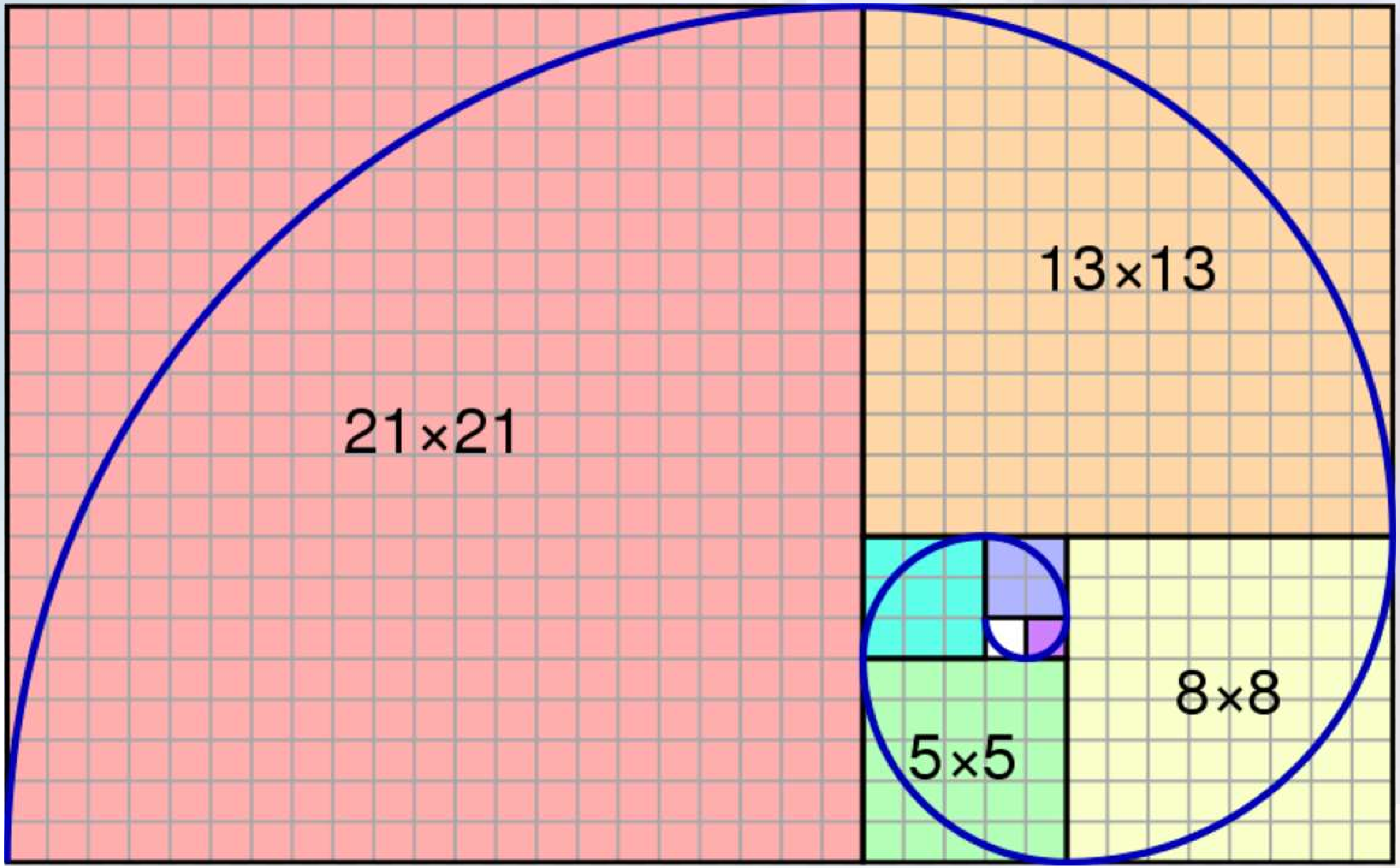
  - **Aplicação Prática:**





Problema 04: Fibonacci (2)

- **Aplicação Prática:**

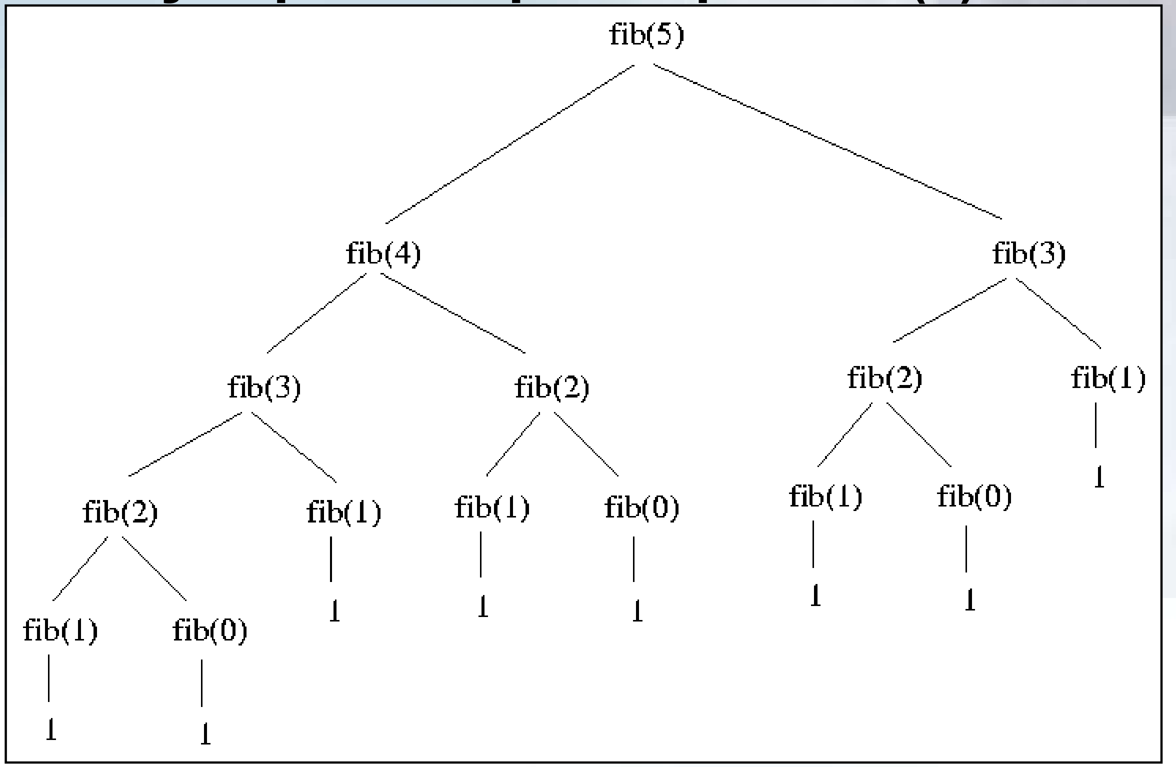


Problema 04: Fibonacci (3)

- **Resolução Recursiva:**

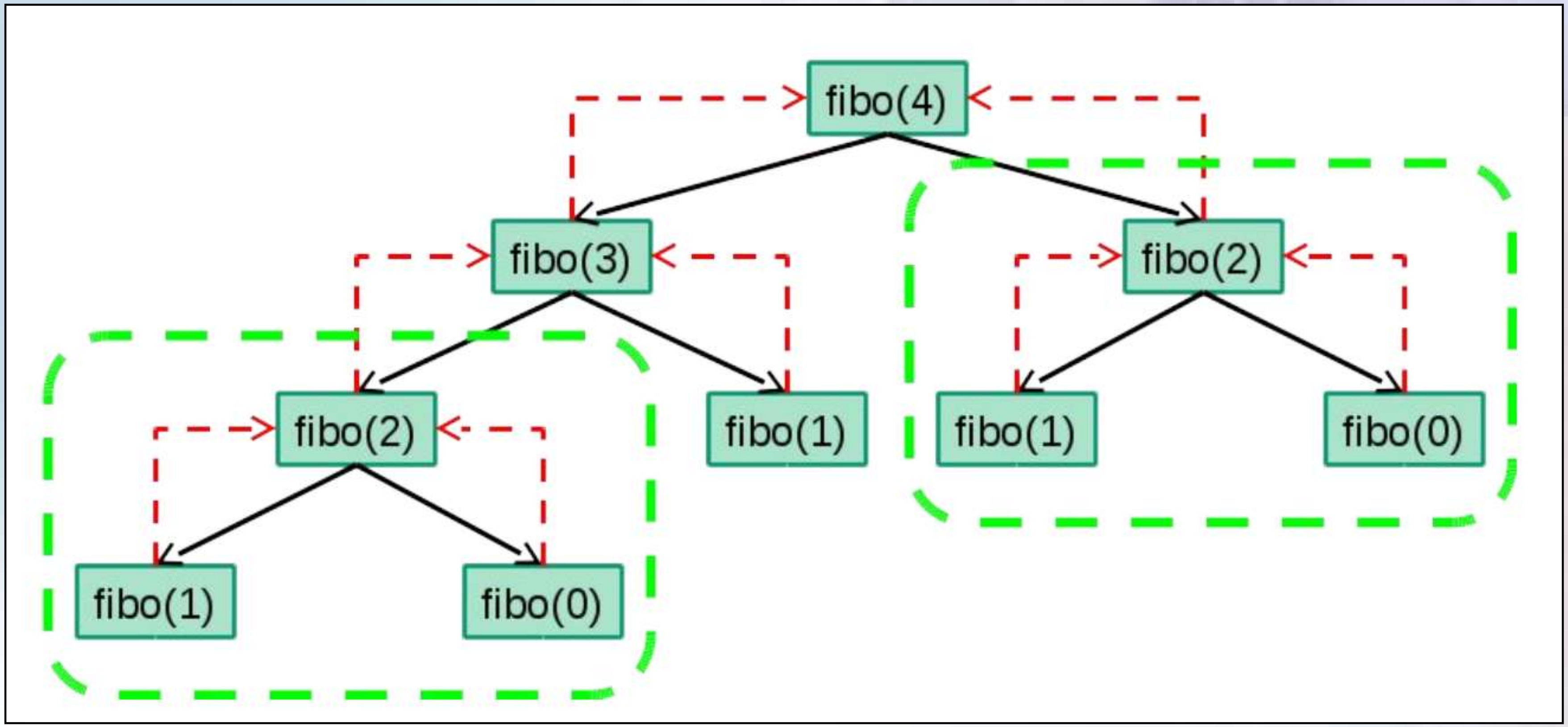
```
int fib(int n)
{
        if(n < 2)
                return 1;
        else
                return fib(n-1)+fib(n-2);
}
```

- **Como é a execução passo a passo para fib(5) = 8 ?**



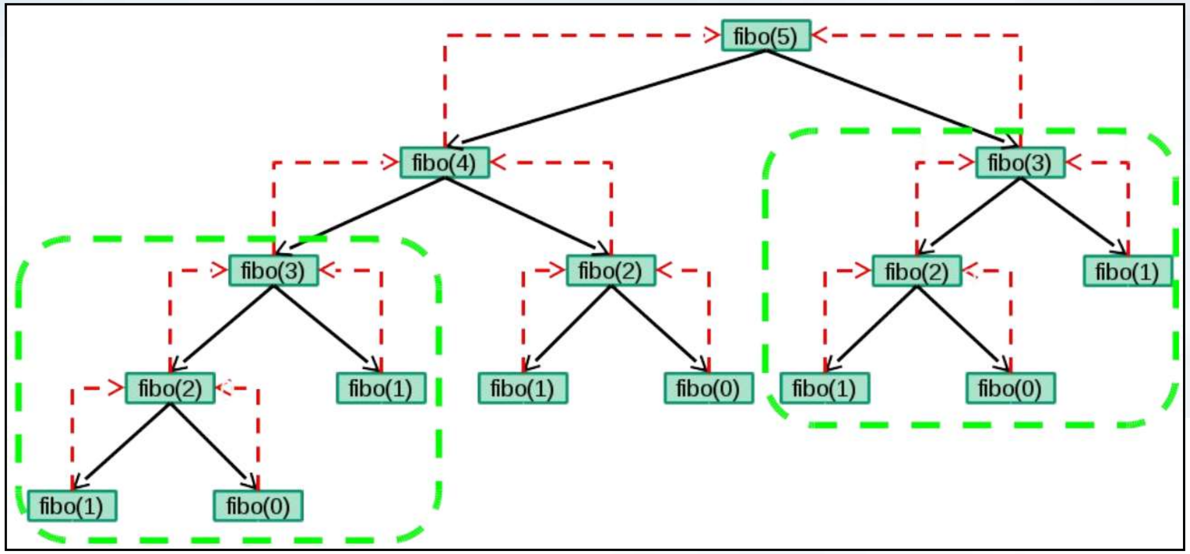
Problema 04: Fibonacci (4)

- **Qual o problema deste algoritmo?**



Problema 04: Fibonacci (5)

- **Qual o problema deste algoritmo?**
  - Os termos F(n-1) e F(n-2) são computados independentemente e repetidas vezes
  - Número de Chamadas Recursivas é igual ao número de Fibonacci sendo calculado
  - Não se pode utilizar as Fórmulas de Recorrência anteriores → O Comportamento de F(n-1) e F(n-2) são independentes
  - Árvore Computacional de tamanho aproximado 2^n



Problema 04: Fibonacci (6)

- **Complexidade de Tempo do Algoritmo Recursivo – T(n):**
  - Por inspeção, temos que:
    - T(n) = T(n-1) + T(n-2) + 1
    - T(n) = T(n-2) + T(n-3) + T(n-3) + T(n-4) + 1
    - T(n) = T(n-3) + T(n-4) + T(n-4) + T(n-5) + T(n-4) + T(n-5) + T(n-5) + T(n-6) + 1
    - .... E assim vai!
- **Fórmula de Recorrência Fechada:**
  - **Conhecida como Razão Áurea**
  - **Custo Exponencial**

$$f_n = \frac{1}{\sqrt{5}}[\phi^n - (-\phi)^{-n}]$$

$$\phi = \frac{\sqrt{5}+1}{2} \approx 1,618$$

Problema 04: Fibonacci (7)

- **Como resolver Fibonacci de Maneira Iterativa?**
  - Utilizando apenas uma Repetição

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

- **Complexidade de Tempo do Algoritmo Iterativo – T(n):**
  - **Pior Caso**: O(n) → Linear

Problema 04: Fibonacci (8)

- **Conclusões:**
  - O **algoritmo recursivo** é extremamente **ineficiente**, pois **recalcula repetidas vezes** o mesmo valor
    - A **complexidade** de **tempo** para calcular, considerando as operações de adição, é $T(n) = O(\varphi^n)$
  - A versão iterativa deste algoritmo possui complexidade de tempo $T(n) = O(n)$
- **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

- **Lição:**
  - **Recursividade é elegante mas nem sempre é eficiente**
    - Queremos quebrar em subproblemas
    - Fibonacci recursivo não faz isso

Outros Algoritmos Recursivos "Problemáticos" (1)

- **Função de Ackermann:**
  - Nomeada por Wilhelm Ackermann, é um dos mais simples e recém-descobertos exemplos de uma função computável **que não são funções recursivas primitivas**

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

  - Ver animação: https://gfredericks.com/things/arith/ackermann
- **Análise:**
  - Esta função **cresce assustadoramente rápido**:
    - A(4, 3) = A(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

Outros Algoritmos Recursivos "Problemáticos" (2)

- **Função não Bem Definida:**
  - Seja a função G : $Z^+ \rightarrow Z$. Para todos os inteiros n ≥ 1:

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

- **Análise:**
  - **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(10))) = 3 + **G(5)** → **Crash!!**

Conclusões

- **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**

PANC: Projeto e Análise de Algoritmos

Aula 06: Algoritmos Recursivos: Problemas e Análise da Complexidade

Breno Lisi Romano

Dúvidas???

http://sites.google.com/site/blromano

Instituto Federal de São Paulo – IFSP São João da Boa Vista
Bacharelado em Ciência da Computação – 3º Semestre

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista