AED - Algoritmos e Estruturas de Dados

Aula 6 - Recursividade

Prof. Rodrigo Mafort

Tópicos da Disciplina

- ✓Programação Orientada a Objetos
- ❑Recursividade
- ❑Análise da Complexidade Computacional de Algoritmos
- ❑Estruturas de Dados de Alocação Estática
  - ❑Listas, Filas e Pilhas
  - ❑Algoritmos de Ordenação
  - ❑Busca Binária
- ❑Estruturas de Dados de Alocação Dinâmica
  - ❑Listas Simplesmente e Duplamente Encadeadas
  - ❑Listas Circulares
  - ❑Árvores, Árvores Binárias e Árvores Binárias de Busca
  - ❑Grafos

Ideia da Recursividade

- Preciso ir até o final da sala.
- Quantos passos preciso dar para chegar até lá?
- Como eu posso percorrer essa distância?
  - Algoritmo: Andar até o final da sala
  - Estou no final da sala?
    - Não:
      - Dar um passo para a frente
      - Executar o mesmo algoritmo para o restante da distância menos um passo.
    - Sim:
      - Cheguei até o outro lado! Fim.

Ideia da Recursividade

- Seja P um problema qualquer
- Seja i uma instância do problema P
- Não sabemos resolver P(i)...

- Mas sabemos resolver um P(j) para uma instância menor j ($j < i$)

- Ideia da Recursividade:
  Resolver P(j) e usar a solução para resolver P(i).

Ideia da Recursividade

- Para implementar uma função recursiva precisamos identificar:
  - Qual é o caso base da recursão?
    - Identificar o(s) caso(s) que pode(m) ser resolvido diretamente.
  - Como decompor o problema em partes menores?
    - Identificar como um caso mais complexo pode ser resolvido através de casos menores e mais simples.
  - Como juntar as soluções para problemas pequenos e uma solução para um caso maior?
    - Após identificar como dividir o problema, é necessário verificar como usar essas pequenas soluções na solução do problema original.

Exemplo Prático: Fatorial

- Considere o fatorial de um número n
- $Fat(n) = n * n - 1 * n - 2 * \cdots * 1$

- Não sei calcular diretamente o fatorial de n
- Mas sei que:
  - $Fat(n) = n * n - 1 * n - 2 * \cdots * 1$
  - $n - 1 * n - 2 * \cdots * 1 = Fat(n-1)$
  - Logo: $Fat(n) = n * Fat(n-1)$

- Quanto é $Fat(n-1)$? $Fat(n-1) = n - 1 * Fat(n-2)$

Exemplo Prático: Fatorial

- $Fat(n) = n \ * Fat(n-1)$
- $Fat(n-1) = n-1 \ * Fat(n-2)$
- $Fat(n-2) = n-2 \ * Fat(n-3)$
- Mas até quando?

- Qual é fatorial que podemos responder sem fazer nenhum cálculo?
- $Fat(0) = 1$

Exemplo Prático: Fatorial

- Descobrimos que é possível usar a solução de um problema menor para resolver um problema maior.

- $Fat(n) = \begin{cases} n * Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Essa forma de apresentar um problema é chamada de **Fórmula da Recorrência**.
- Ela indica como e até onde dividir o problema, além de apresentar como usar as soluções dos problemas menores na solução das instâncias maiores, inclusive a instância original.

Exemplo Prático: Fatorial

- $Fat(n) = \begin{cases} n \ * Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Como calcular o fatorial de 5?

Exemplo Prático: Fatorial

- $Fat(n) = \begin{cases} n * Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Como calcular o fatorial de 5?

$Fat(5) = 5 * Fat(4)$

Exemplo Prático: Fatorial

- $Fat(n) = \begin{cases} n * Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Como calcular o fatorial de 4?

$Fat(4) = 4 * Fat(3)$
$Fat(5) = 5 * Fat(4)$

Exemplo Prático: Fatorial

- $Fat(n) = \begin{cases} n * Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Como calcular o fatorial de 3?

$Fat(4) = 4 * Fat(3)$
$Fat(5) = 5 * Fat(4)$

Exemplo Prático: Fatorial

- $Fat(n) = \begin{cases} n * Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Como calcular o fatorial de 3?

$$Fat(3) = 3 * Fat(2)$$

$$Fat(4) = 4 * Fat(3)$$

$$Fat(5) = 5 * Fat(4)$$

Exemplo Prático: Fatorial

- $Fat(n) = \begin{cases} n * Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Como calcular o fatorial de 2?

$Fat(3) = 3 * Fat(2)$
$Fat(4) = 4 * Fat(3)$
$Fat(5) = 5 * Fat(4)$

Exemplo Prático: Fatorial

- $Fat(n) = \begin{cases} n * Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Como calcular o fatorial de 2?

$Fat(2) = 2 * Fat(1)$
$Fat(3) = 3 * Fat(2)$
$Fat(4) = 4 * Fat(3)$
$Fat(5) = 5 * Fat(4)$

Exemplo Prático: Fatorial

- $Fat(n) = \begin{cases} n * Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Como calcular o fatorial de 1?

$Fat(2) = 2 * Fat(1)$
$Fat(3) = 3 * Fat(2)$
$Fat(4) = 4 * Fat(3)$
$Fat(5) = 5 * Fat(4)$

Exemplo Prático: Fatorial

- $Fat(n) = \begin{cases} n * Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Como calcular o fatorial de 1?

$Fat(1) = 1 * Fat(0)$
$Fat(2) = 2 * Fat(1)$
$Fat(3) = 3 * Fat(2)$
$Fat(4) = 4 * Fat(3)$
$Fat(5) = 5 * Fat(4)$

Exemplo Prático: Fatorial

- $Fat(n) = \begin{cases} n * Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Como calcular o fatorial de 0?

$Fat(1) = 1 * Fat(0)$
$Fat(2) = 2 * Fat(1)$
$Fat(3) = 3 * Fat(2)$
$Fat(4) = 4 * Fat(3)$
$Fat(5) = 5 * Fat(4)$

Exemplo Prático: Fatorial

- $Fat(n) = \begin{cases} n \, * \, Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Como calcular o fatorial de 0?

Esse caso sabemos resolver!!!

$\mathbf{Fat(0)} \ = \mathbf{1}$
$Fat(1) \ = 1 \, * \, Fat(0)$
$Fat(2) \ = 2 \, * \, Fat(1)$
$Fat(3) \ = 3 \, * \, Fat(2)$
$Fat(4) \ = \ 4 \, * \, Fat(3)$
$Fat(5) \ = \ 5 \, * \, Fat(4)$

Exemplo Prático: Fatorial

- $Fat(n) = \begin{cases} n * Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Agora podemos retornar aos valores que ainda não sabemos
- Mas usando os resultados que já conhecemos

$\mathbf{Fat(0) = 1}$
$Fat(1) = 1 * Fat(0)$
$Fat(2) = 2 * Fat(1)$
$Fat(3) = 3 * Fat(2)$
$Fat(4) = 4 * Fat(3)$
$Fat(5) = 5 * Fat(4)$

Exemplo Prático: Fatorial

- $Fat(n) = \begin{cases} n * Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Como calcular o fatorial de 1?

$Fat(1) = 1 * Fat(0) = 1 * 1 = 1$
$Fat(2) = 2 * Fat(1)$
$Fat(3) = 3 * Fat(2)$
$Fat(4) = 4 * Fat(3)$
$Fat(5) = 5 * Fat(4)$

Exemplo Prático: Fatorial

- $Fat(n) = \begin{cases} n * Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Como calcular o fatorial de 2?

$Fat(2) = 2 * Fat(1) = 2 * 1 = 2$
$Fat(3) = 3 * Fat(2)$
$Fat(4) = 4 * Fat(3)$
$Fat(5) = 5 * Fat(4)$

Exemplo Prático: Fatorial

- $Fat(n) = \begin{cases} n * Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Como calcular o fatorial de 3?

$$Fat(3) = 3 * Fat(2) = 3 * 2 = 6$$

$Fat(4) = 4 * Fat(3)$
$Fat(5) = 5 * Fat(4)$

Exemplo Prático: Fatorial

- $Fat(n) = \begin{cases} n * Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Como calcular o fatorial de 4?

$Fat(4) = 4 * Fat(3) = 4 * 6 = 24$

$Fat(5) = 5 * Fat(4)$

Exemplo Prático: Fatorial

- $Fat(n) = \begin{cases} n * Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Como calcular o fatorial de 5?

$$Fat(5) = 5 * Fat(4) = 5 * 24 = 120$$

Exemplo Prático: Fatorial

- $Fat(n) = \begin{cases} n \ * Fat(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$

- Como calcular o fatorial de 5?

- $Fat(5) \ = \ 120$

Recursão - Regras

- Para um problema admitir uma solução por recursão ele deve satisfazer três regras:
  - Deve existir um caso para o qual se conheça a solução do problema.
    - Este caso não pode depender de outros casos.
    - É o ponto de parada da recursão.
    - Chamado de base da recursão.
  - A solução para uma instância do problema deve ser composta das soluções de casos menores (Vide exemplo do fatorial)
    - Deve ser possível usar as soluções de casos menores para resolver casos maiores.
  - O algoritmo deve executar (ou chamar) a si mesmo

Recursão - Regras

- Sem um caso base, a recursão se torna infinita.

- Esse problema também pode ocorrer caso o caso base não seja alcançado.
  - Exemplo - erro de programação: $Fat(n) = n * Fat(n)$
    - Nesse caso, o algoritmo nunca atingirá o caso base
    - Logo a execução nunca terminará (teoricamente – considerando memória infinita)

- Caso não seja possível recompor a solução de casos menores em uma solução para um caso maior, a recursão não é adequada ao problema.

Algoritmo recursivo vs iterativo

```
int FR(int i)
{
    if (i > 0)
        return i * FR(i-1);
    else
        return 1;
}
```

```
int FI(int i)
{
    int res = 1;
    while ( i > 1)
    {
        res = res * i;
        i = i - 1;
    }
}
```

Algoritmo recursivo vs iterativo

```
int FR(int i)
{
    if (i > 0)
        return i * FR(i-1);
    else
        return 1;
}
```

Observe que a função chama a si mesma

```
int FI(int i)
{
    int res = 1;
    while ( i > 1)
    {
        res = res * i;
        i = i - 1;
    }
}
```

Algoritmo recursivo vs iterativo

```
int FR(int i)
{
    if (i > 0)
        return i * FR(i-1);
    else
        return 1;
}
```

Chamada recursiva

Base da recursão

```
int FI(int i)
{
    int res = 1;
    while ( i > 1)
    {
        res = res * i;
        i = i - 1;
    }
}
```

Recursão - Desvantagem

- A recursão apresenta uma desvantagem quando comparada aos métodos iterativos.
- Na forma com que foi apresentada até o momento, os resultados intermediários **não** são armazenados na memória.
- Exemplo: Sequência de Fibonacci:
  - Os dois primeiros termos são 0 e 1
  - Os termos subsequentes correspondem a soma dos dois anteriores
  - 0 1 1 2 3 5 8 13 21 ...

Recursão - Desvantagem

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

- Onde k é o k-ésimo ($k \geq 1$) termo da sequência

Recursão - Desvantagem

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

- $Fib(6) = Fib(5) + Fib(4)$
  - Nota-se duas chamadas recursivas
  - Primeiro o resultado de $Fib(5)$ é computado
  - Depois se calcula o resultado de $Fib(4)$.

- $Fib(5) = Fib(4) + Fib(3)$

Recursão - Desvantagem

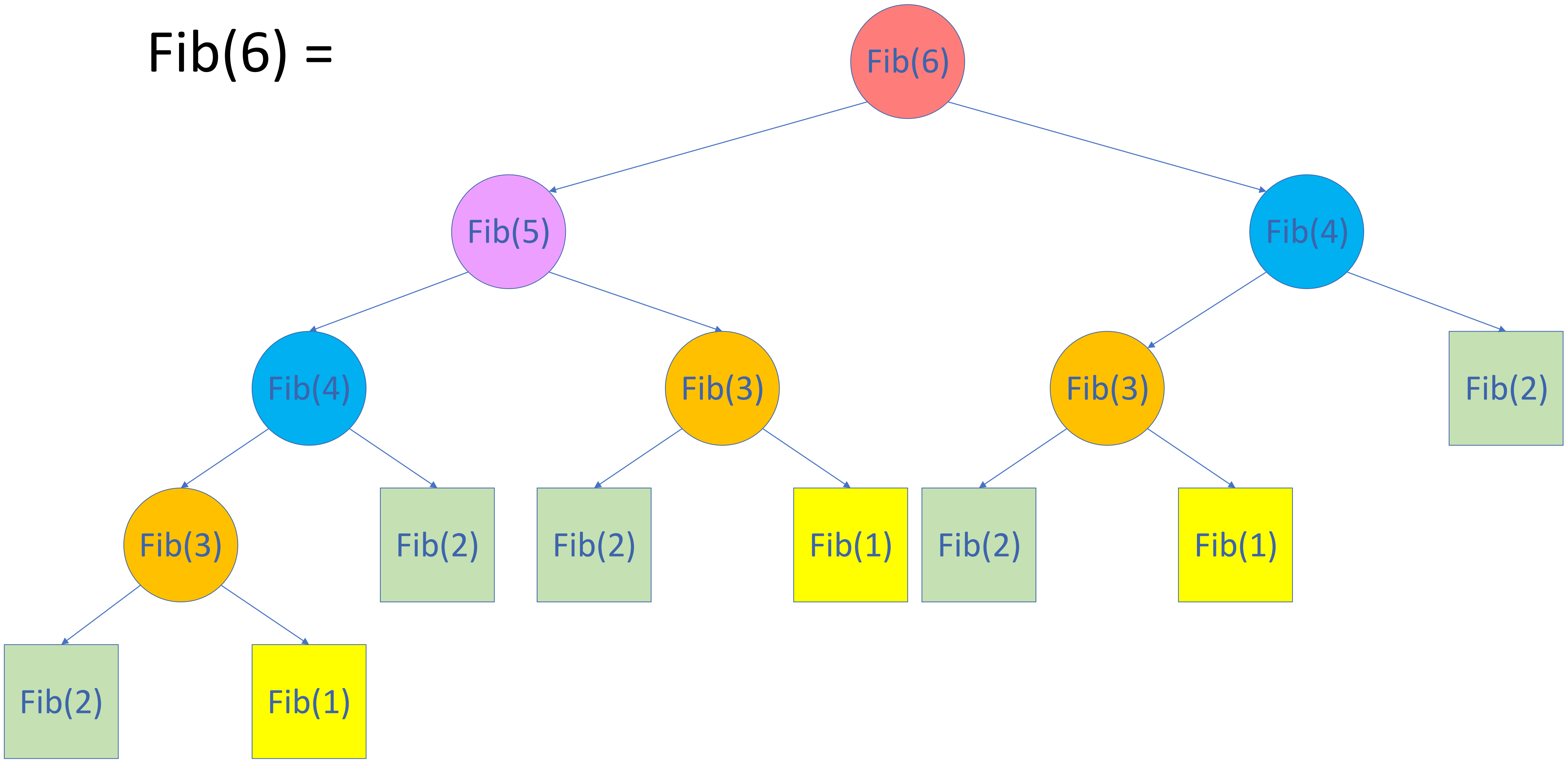
- $Fib(k) = \begin{cases} Fib(k-1) + Fib(k-2), & k > 2 \\ 0, & k = 1 \\ 1, & k = 2 \end{cases}$

- $Fib(6) = Fib(5) + Fib(4)$
- $Fib(5) = Fib(4) + Fib(3)$

- Como os resultados intermediários não são armazenados, algumas chamadas são realizadas várias vezes.

Recursão - Desvantagem

Fib(6) =



Fibonacci recursivo vs iterativo

```
int Fib_Rec(int k)
{
    if (k > 2)
        return  Fib_Rec(k -1) + Fib_Rec(k - 2);
    else
        if (k == 0)
            return 0;
        else
            return 1;
}
```

Fibonacci recursivo vs iterativo

```
int Fib_Ite(int k)
{
    int ult = 1, pult = 0, novo;
    for (int i = 2; i <= k; i++)
    {
        novo = ult + pult;
        pult = ult;
        ult = novo;
    }
    return novo;
}
```

Recursão - Desvantagens

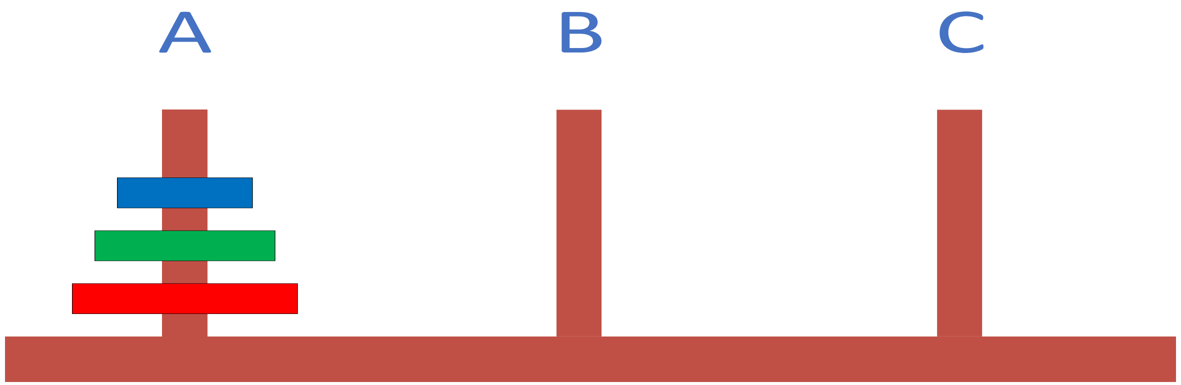
- As múltiplas chamadas a uma mesma função podem piorar o desempenho do algoritmo quando comparado a versão iterativa.

- Existe uma técnica de programação avançada (Programação Dinâmica) que busca atenuar esse problema ao salvar as etapas intermediárias do processamento.

- Ambos os tópicos são assunto de outra disciplina e fogem ao escopo desta aula.

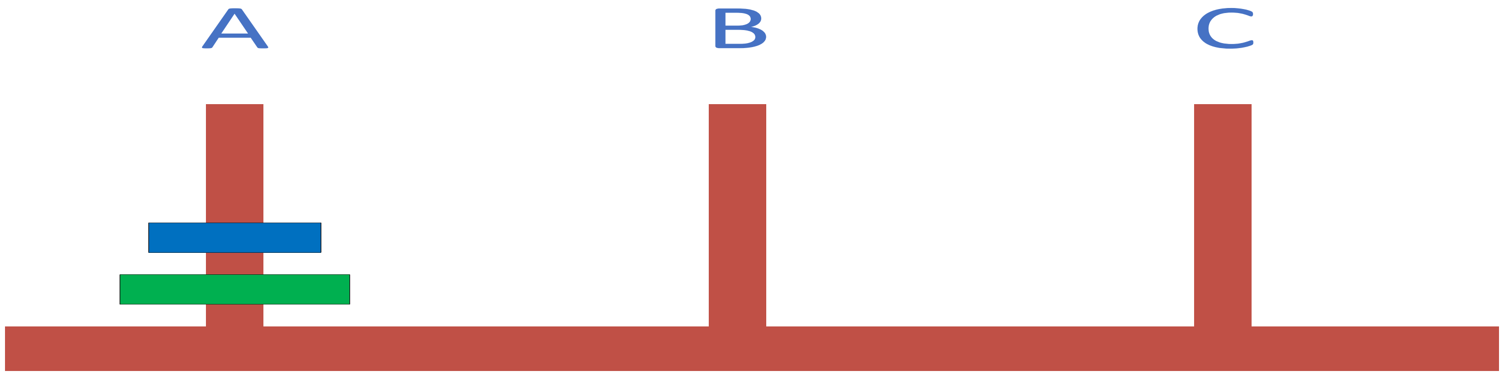
Recursividade - Vantagens

- Existem problemas cuja solução recursiva é mais intuitiva e natural. A solução iterativa pode ser de difícil implementação.
- Exemplo: Torres de Hanói
  - Objetivo: Mover os discos da torre A para a torre C
  - Regras:
    - Mover um disco por vez
    - Um disco maior não pode ser colocado sobre um menor.



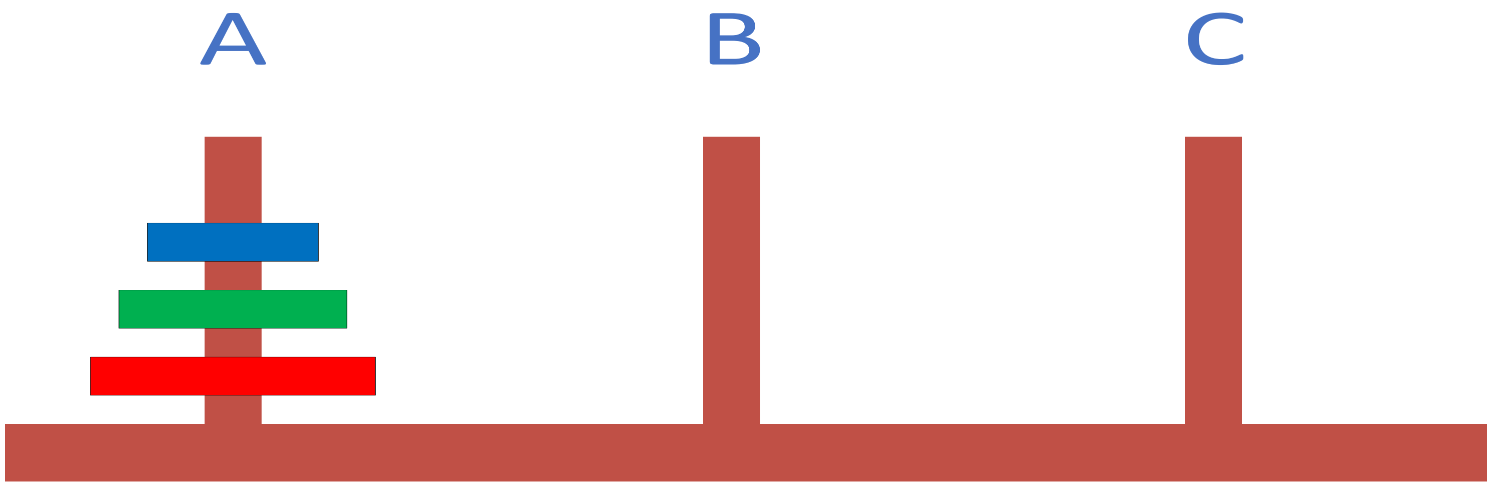
Recursividade - Vantagens

- Como resolver o problema?
  - Difícil visualizar a solução do problema.
  - Esse o problema fosse resolvido por partes?
  - Resolver o problema com 2 disco é mais fácil do que com 3.



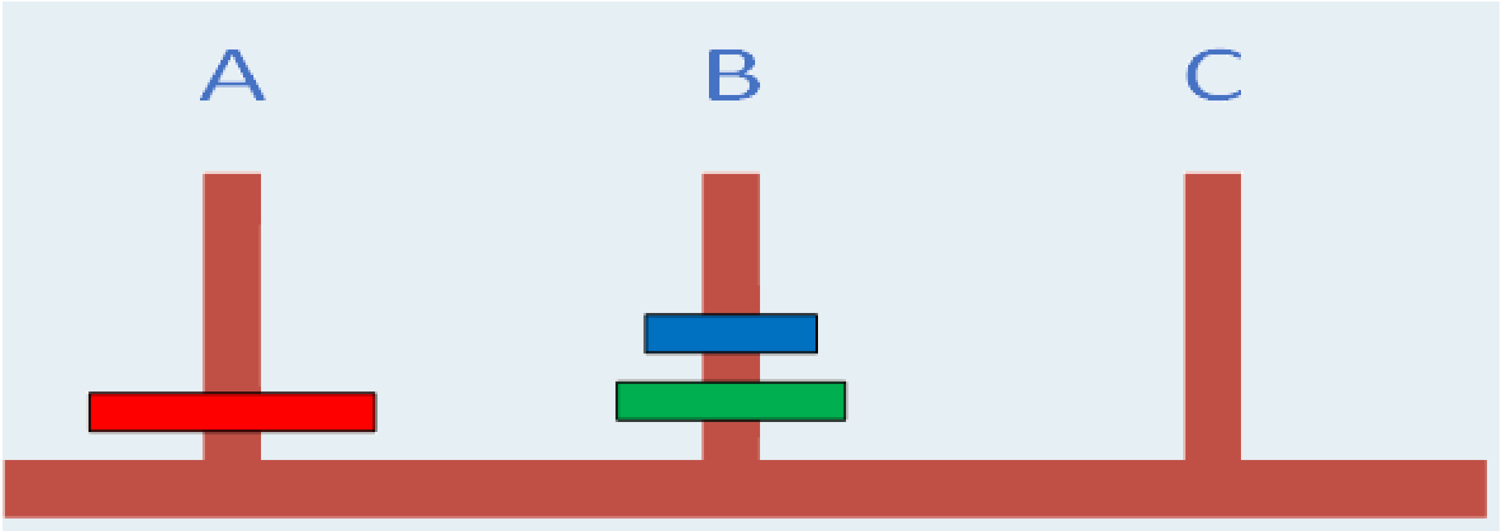
Recursividade - Vantagens

- Hanói com três discos: A → C
  - Hanói com dois discos: A → B
  - Mover terceiro disco: A → C
  - Hanói com dois discos: B → C



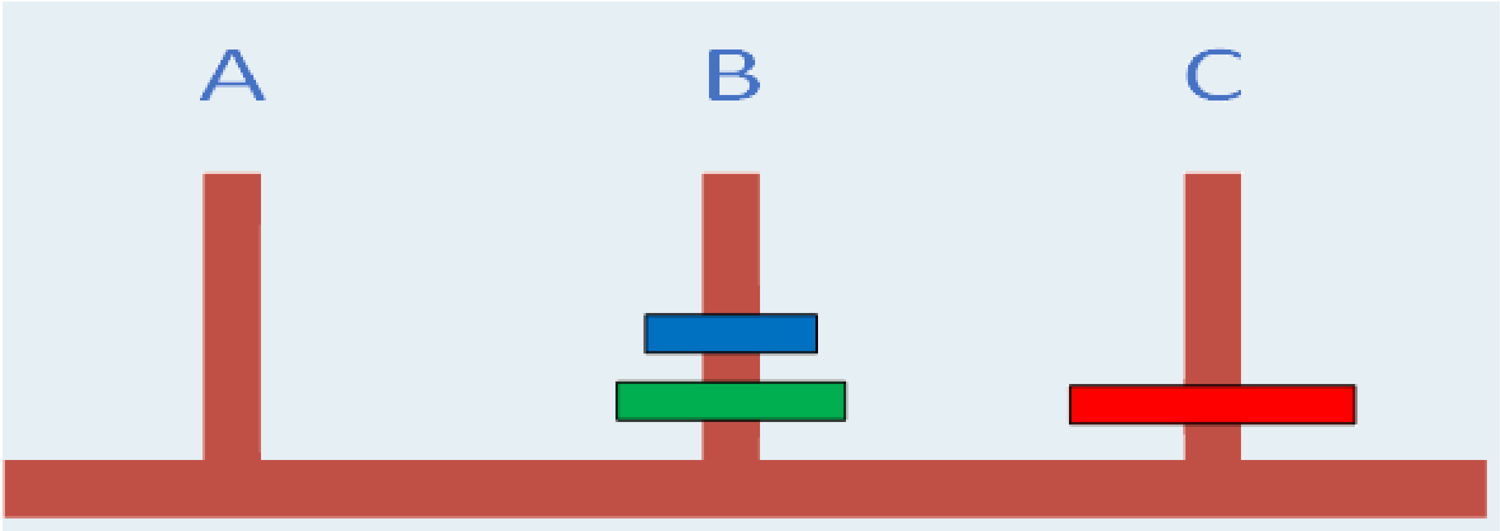
Recursividade - Vantagens

- Hanói com três discos: A → C
  - Hanói com dois discos: A → B
  - Mover terceiro disco: A → C
  - Hanói com dois discos: B → C



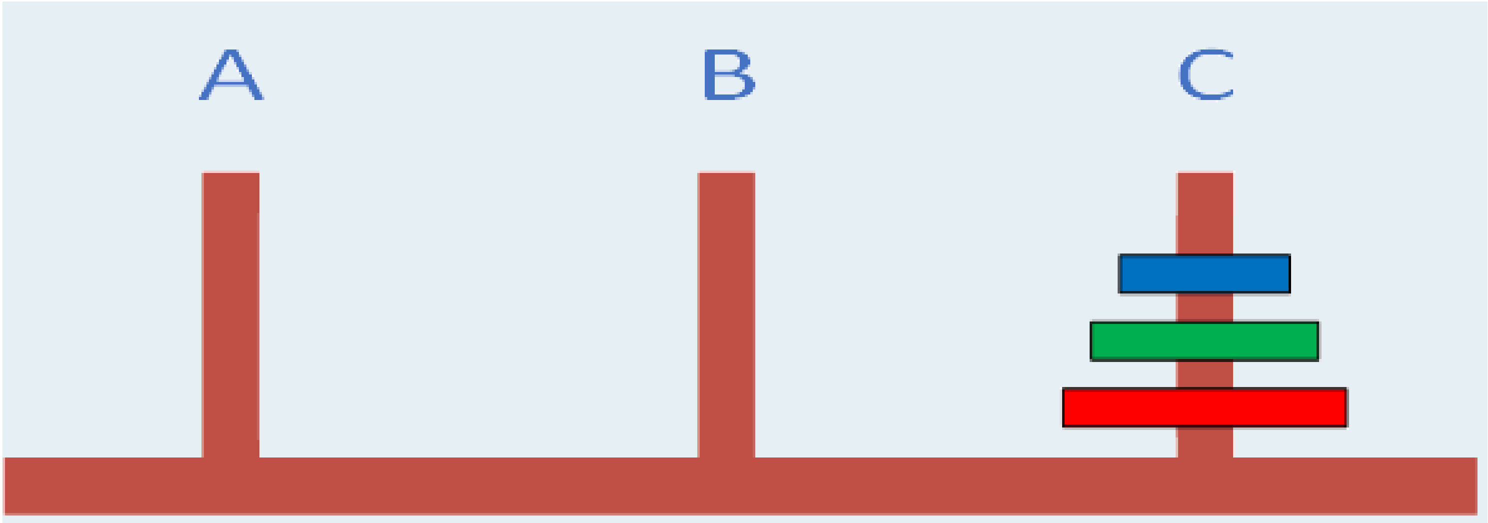
Recursividade - Vantagens

- Hanói com três discos: A → C
  - Hanói com dois discos: A → B
  - Mover terceiro disco: A → C
  - Hanói com dois discos: B → C



Recursividade - Vantagens

- Hanói com três discos: A → C
  - Hanói com dois discos: A → B
  - Mover terceiro disco: A → C
  - Hanói com dois discos: B → C



Recursividade - Vantagens

- Hanói com três discos: A → C
  - Hanói com dois discos: A → B
  - Mover terceiro disco: A → C
  - Hanói com dois discos: B → C

- Esse algoritmo pode ser modificado para qualquer número de discos.
- E um algoritmo iterativo para Hanói??

Pilha de Execução

- Toda chamada recursiva implica:
  - O processo que fez a chamada é pausado
  - A nova chamada é processada
  - Quando a nova chamada for finalizada, o processo anterior é retomado

- Para gerir a execução de chamadas recursivas, o sistema usa uma **pilha de chamadas**
  - O topo corresponde ao processo atual (que está sendo processado)
  - A cada chamada, o processo atual é pausado e um novo processo é empilhado e processado
  - Ao retornar, o processo do topo da pilha é descartado e o anterior (o que chamou) é retomado

Pilha de Execução

- Os dados de cada chamada (variáveis e parâmetros) são salvos na pilha
  - Uso de memória para controle das chamadas recursivas
  - Número finito de chamadas recursivas
- Pilha de execução para o exemplo do fatorial de 5:

$Fib(1) = 1$
$Fib(2) = 2 * Fib(1)$
$Fib(3) = 3 * Fib(2)$
$Fib(4) = 4 * Fib(3)$
$Fib(5) = 5 * Fib(4)$

Topo da Pilha:
Chamada em processamento

Exemplo:

- Implemente uma função recursiva que retorne a divisão de dois números inteiros positivos A e B (suponha A > B) utilizando apenas operadores de soma e subtração.

Exemplo: Resposta

```
int Dividir(int A, int B)
{
    if (A >= B)
    {
        return 1 + Dividir(A-B,B);
    }
    else
    {
        return 0;
    }
}
```

Exemplo:

- Implemente uma função recursiva que retorne a divisão de dois números inteiros positivos A e B (suponha A > B) utilizando apenas operadores de soma e subtração.

- Retorne por referência o resto da divisão (como parâmetro A).

Exemplo: Resposta

```
int Dividir(int *A, int B)
{
     if (*A >= B)
     {
          *A = *A - B;
          return 1 + Dividir(A,B);
     }
     else
     {
          return 0;
     }
}
```

Uso da Recursividade em ED

- Algoritmos de Ordenação:
  - Mergesort:
  - Quicksort
  - E outros
  - Ideia: Não sei ordenar um vetor desse tamanho, mas ordenar um menor é mais fácil. Essa ideia é aplicada até atingir um tamanho que sabemos ordenar – a base da recursão (em geral, um único elemento).

- Algoritmos de Busca:
  - Busca binária
  - Busca em árvores binárias

- Algoritmos de Busca/Percurso em Grafos:
  - Busca em Largura
  - Busca em Profundidade

Como identificar uma função recursiva

- Toda função recursiva chama a si mesma:

```
int Fatorial(int i)
{
  if (i > 0)
      return i * Fatorial(i-1);
  else
      return 1;
}
```

- Vale observar que as funções recursivas muitas vezes aparentam repetir um trecho do código sem usar nenhum laço de repetição.

Funções Recursivas

- Quando implementar uma função recursiva:
  1. Identifique se o problema tem as seguintes características:
     - A solução para um problema grande pode ser obtida pela “união” das soluções de problemas menores
     - É mais fácil resolver problemas menores
     - Existe um caso cuja resposta é trivial ou muito fácil de se obter
  2. Identifique o caso base
  3. Identifique como dividir o problema maior e como usar essas respostas para resolver o problema inicial

Cuidados - Recursão

- Sem um caso base, a recursão se torna infinita.
- Verifique se suas chamadas convergem para a base da recursão.
- Por exemplo: se implementarmos Fat(n) = Fat(n+1) / n, nunca chegaremos ao caso base da recursão Fat(0) = 1.

- Importante:
  - Esses casos se comportam como um loop infinito. Entretanto, cada chamada recursiva consome um pouco mais memória (para a pilha de chamadas).
  - Se sua pilha ficar muito longa, seu programa pode ser interrompido por falta de memória.

Função de Ackermann

- Observe a seguinte função recursiva:

```
long Ackermann(long m, long n)
{
    if (m == 0)
        return n + 1;
    else
        if (n == 0)
            return Ackermann(m - 1, 1);
        else
            return Ackermann(m - 1, Ackermann(m, n - 1));
}
```

- Ela converge para o caso base?
- Tente calcular Ackermann(3,2). Em seguida, tente Ackermann(4,2).

Exercícios

- Escreva uma função recursiva que some todos os elementos de um vetor.

- Escreva uma função recursiva que inverta um vetor.

- Escreva uma função recursiva que determine quantas vezes um dígito d ocorre em um número inteiro n (assuma $n > 0$).

- Escreva uma função recursiva que transforme um número natural n na base decimal para a base binária.