ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

Computational Thinking

PROF. EDUARDO GONDO

Recursão — Introdução

- técnica de programação aplicada a métodos ou funções
- o método chama, direta ou indiretamente, a ele próprio
- quais tipos de problemas podemos aplicar recursão?
- naqueles cuja entrada do problema podem ser resolvidos usando uma entrada menor (e teoricamente mais fácil) do mesmo problema
- exemplos: fatorial, mdc, fibonacci, potência, busca binária, ordenação, listas lineares, árvores, etc
- também muito utilizado em soluções baseadas em tentativa e erro (back tracking)

Recursão — Introdução

Considerando a *entrada* do problema, muitos algoritmos recursivos possuem a seguinte estrutura:

```
se a entrada do problema for pequena então:
   resolva-a diretamente
senão:
   reduza a entrada para uma entrada menor do mesmo
       problema
   aplique o método a entrada menor
   use a solução da entrada menor com o objetivo de
       resolver à entrada original
```

Todo algoritmo recursivo deve ter, como primeira instrução, sua condição de parada. Senão corremos sério risco que ele nunca termine!

Recursão — Torre de Hanoi

Torre de Hanoi é um jogo que consiste em passar um conjunto de discos (maior ou igual a 2) de diâmetros distintos da haste A para haste C com o menor número de movimentos possíveis obedecendo duas regras:

1. apenas um disco pode ser movido por vez para outra haste;
2. um disco com diâmetro maior nunca pode ficar sobre um disco de diâmetro menor.

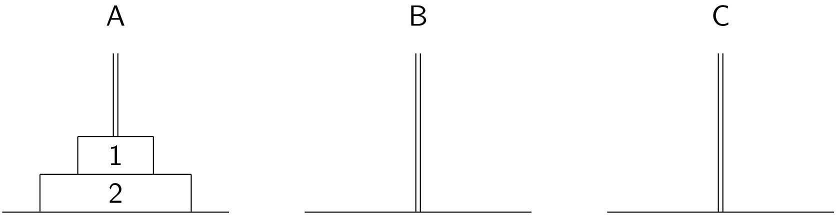
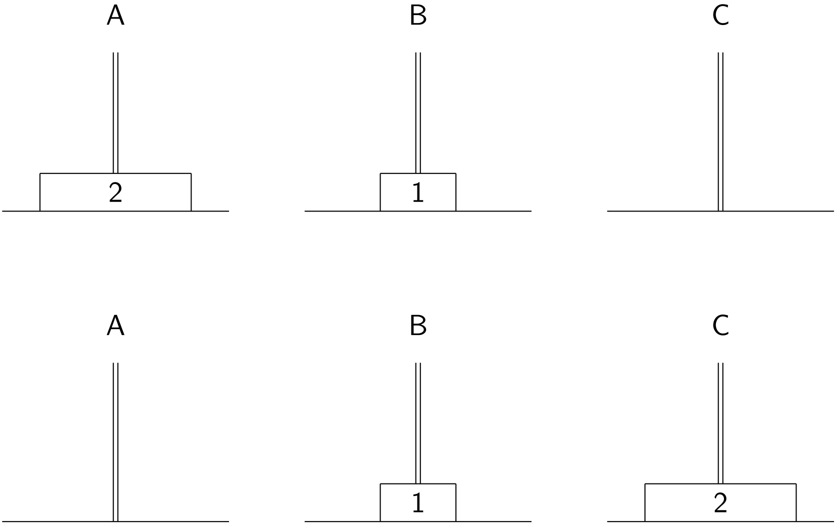


Figura: Torre de Hanoi com 2 discos

Recursão — Hanoi Solução



Recursão — Hanoi Solução

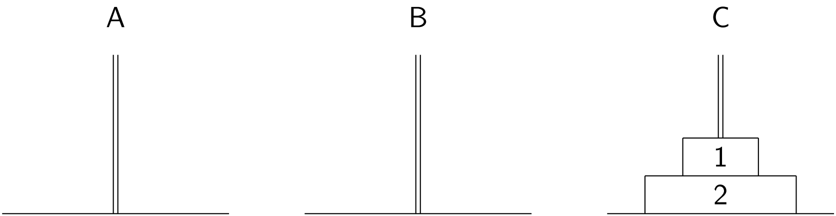


Figura: Solução do problema de Hanoi para 2 discos

- foram necessários 3 movimentos
- $1 \rightarrow$ B; $2 \rightarrow$ C; $1 \rightarrow$ C
- podemos dizer que é um problema fácil de resolver, certo?

Recursão — Hanoi Solução

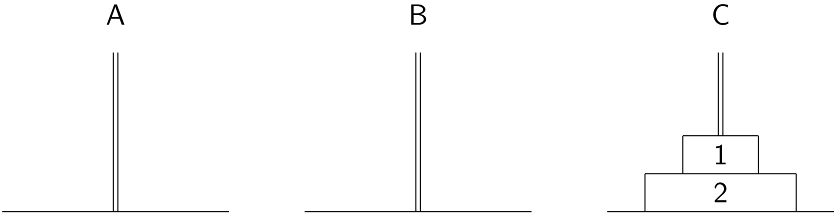


Figura: Solução do problema de Hanoi para 2 discos

- foram necessários 3 movimentos
- $1 \rightarrow$ B; $2 \rightarrow$ C; $1 \rightarrow$ C
- podemos dizer que é um problema fácil de resolver, certo?
- e se fossem 3 discos?

Torre de Hanoi 3.0

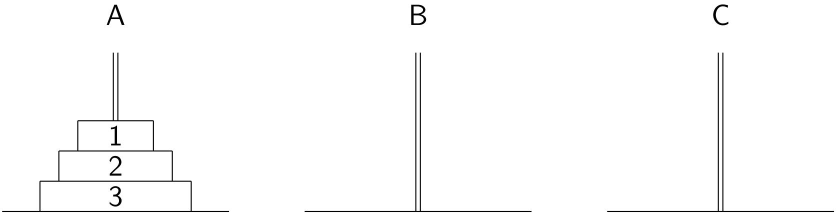
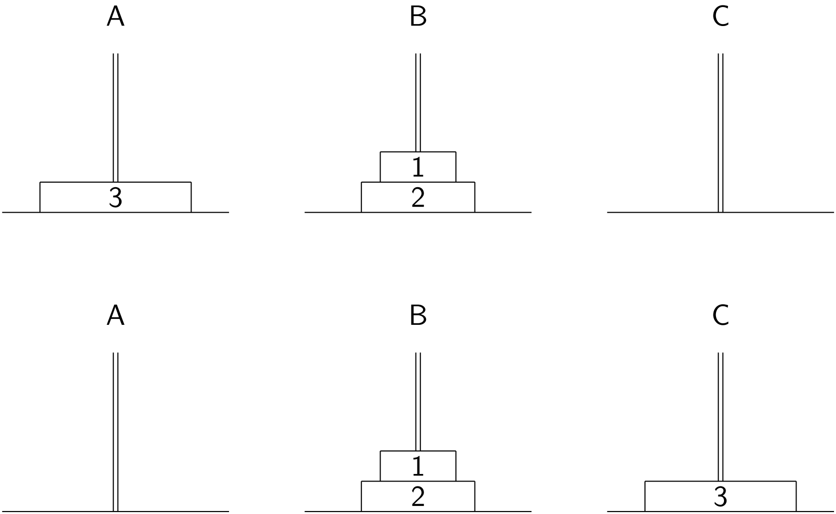


Figura: Torre de Hanoi 3.0 discos

- usamos a solução anterior movendo os discos 1 e 2 para a haste B
- movemos o disco 3 para a haste C
- novamente usamos a solução anterior para mover os discos 1 e 2 para a haste C

Torre de Hanoi 3.0



Considerações — Torre de Hanoi

- para solucionar o jogo de hanoi para 3 discos usamos a solução de 2 discos
- usamos exatamente a estrutura que muitos algoritmos recursivos possuem
- vamos relembrá-la:

```
se a entrada do problema for pequena então:
    resolva-a diretamente
senão:
    reduza a entrada para uma entrada menor do mesmo
        problema
    aplique o método a entrada menor
    use a solução da entrada menor com o objetivo de
        resolver à entrada original
```

- agora podemos descrever o algoritmo recursivo para o problema da torre de hanoi

Algoritmo — Torre de Hanoi

Considere que a primeira haste é a origem, a segunda é a haste auxiliar e a terceira é a haste de destino.

```
hanoi(discos, origem, aux, destino):
    se discos == 2 entao //condicao de parada
        movimente os 2 discos da haste origem para destino
    senao:
        hanoi(discos - 1, origem, destino, aux)
        mova o maior disco da haste origem para destino
        hanoi(discos - 1, aux, origem, destino)
```

Note a semelhança entre o algoritmo da torre de hanoi com a estrutura da maioria dos algoritmos recursivos.

Recursão — Fatorial

O fatorial de um número $n \geq 0$ é dado pela seguinte fórmula:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

Por exemplo, $6! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 = 720$. Também podemos defini-lo recursivamente:

$$n! = \begin{cases} 1 & \text{se } n = 0; \\ n \cdot (n-1)! & \text{se } n > 0 \end{cases}$$

Vamos tentar resolver $6!$ usando a definição recursiva:

Recursão — resolvendo 6!

Como $n = 6$, usamos a 2ª linha da definição:

$$6! = 6 \cdot 5!$$

, mas quanto vale 5!?
Novamente aplicamos a definição recursiva:

$$5! = 5 \cdot 4!$$

e novamente estamos com problemas pois devemos saber quanto vale 4!, e assim continuamos até chegar a 1! que, pela definição, sabemos que vale 1. Daí voltamos as expressões anteriores até encontrar o valor de 6!

Recursão — Exemplo Fatorial

PROBLEMA 17.1 Escreva um método recursivo que dado $n \geq 0$ inteiro, calcule n! (fatorial de n).

Recursão — Exemplo Fatorial

PROBLEMA 17.1 Escreva um método recursivo que dado $n \geq 0$ inteiro, calcule $n!$ (fatorial de n).

```
def fatorial(n):
    if n == 0:  #condição de parada
        return 1
    else:
        return n * fatorial(n-1)
```

Vamos fazer o teste de mesa desse método para $n = 5$, ou seja, 5!.

Observações

Veja a comparação entre o método e a definição recursiva:

$$n! = \begin{cases} 1 & \text{se } n = 0; \\ n \cdot (n-1)! & \text{se } n > 0 \end{cases}$$

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

O problema maior é encontrar a definição matemática recursiva pois, como podemos perceber, escrever o algoritmo recursivo usando ela é muito simples. Vejamos outros problemas que podem ser resolvidos da mesma maneira:

Recursão — Exemplo MDC

PROBLEMA 17.2 Escreva um algoritmo que dados dois números inteiros $a > 0$ e $b > 0$ calcula o $mdc(a, b)$.

Segue abaixo a definição do mdc recursiva e seu algoritmo:

$$mdc(a, b) = \begin{cases} b & \text{se } a \ \% \ b = 0; \\ mdc(b, a \ \% \ b) & \text{caso contrário} \end{cases}$$

Recursão — Exemplo MDC

PROBLEMA 17.2 Escreva um algoritmo que dados dois números inteiros $a > 0$ e $b > 0$ calcula o $mdc(a, b)$.

Segue abaixo a definição do mdc recursiva e seu algoritmo:

$$mdc(a,b) = \begin{cases} b & \text{se } a \ \% \ b = 0; \\ mdc(b, a \ \% \ b) & \text{caso contrário} \end{cases}$$

```
def mdc(a, b):
    if a % b == 0:    //condição de parada
        return b
    else:
        return mdc(b, a % b)
```

Recursão — Exemplo Fibonacci

PROBLEMA 17.3 Escreva um algoritmo que dado um inteiro $n > 0$, encontra o n-ésimo número da sequência de Fibonacci.

Segue abaixo a definição da sequência de Fibonacci recursiva e seu algoritmo:

$$F_n = \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2; \\ F_{n-1} + F_{n-2} & \text{se } n > 2 \end{cases}$$

Recursão — Exemplo Fibonacci

PROBLEMA 17.3 Escreva um algoritmo que dado um inteiro $n > 0$, encontra o n-ésimo número da sequência de Fibonacci.

Segue abaixo a definição da sequência de Fibonacci recursiva e seu algoritmo:

$$F_n = \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2; \\ F_{n-1} + F_{n-2} & \text{se } n > 2 \end{cases}$$

```
def fibonacci(n):
    if n == 1 or n == 2:     #condição de parada
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Recursão — Observações Gerais

- a repetição é feita de maneira implícita, ou seja, ela está presente nas chamadas recursivas como podemos ver na simulação dos algoritmos
- apesar da simplicidade do código, a solução recursiva pode ser muito ineficiente se comparada com a versão iterativa
- o computador precisa empilhar as chamadas recursivas dos métodos e controlar todas em memória (o que pode ocasionar um estouro de pilha)
- no algoritmo de fibonacci note que para cada chamada recursiva mais duas chamadas são executadas
- isso inviabiliza o uso de recursão para solução desses problemas
- vamos testar o algoritmo recursivo de Fibonacci para alguns valores de n e comparar com a versão iterativa

Recursão — Exemplo Busca binária

PROBLEMA 17.4 Escreva o algoritmo de busca binária usando recursão.

Para aplicar recursão no algoritmo de busca binária usaremos o conceito de **divisão e conquista** que foi introduzido no capítulo de repetições encaixadas. Vamos relembrá-la:

- divisão: dividir o problema em entradas menores
- conquista: resolver a entrada ou entradas e combiná-las para chegar na solução do problema geral

Na busca binária a divisão consiste em dividir o vetor ao meio e a conquista chamar novamente o algoritmo do busca binária para somente uma das metades do vetor. Contudo, devemos nos atentar para a condição de parada, ou seja, quando não mais existir elementos no vetor.

Recursão — Exemplo Busca binária

Considere que nosso método deverá passar o vetor, o elemento que estamos procurando e as extremidades que devemos considerar do vetor:

```
funcao buscaBinaria(lista, elem, ini, fim):
    se não há elementos na lista:
        retorna -1
    senao:
        calcule a posição central da lista: meio
        se lista[meio] < elem:
            chame e retorne o busca binaria para a metade
                da lista
        senao se lista[meio] > elem:
            chame e retorne o busca binaria para a outra
                metade do vetor
        senao:
            retorna meio
```

Exercícios

1) Dados um vetor de números reais, escreva um método recursivo que recebe o vetor e retorna o máximo valor contido nesse vetor.
2) Escreva um método recursivo que recebe um vetor e imprime o vetor de trás para frente.
3) A operação x^n onde x é um número real e n um natural pode ser definida da seguinte forma recursiva:

$$x^n = \begin{cases} 1 & \text{se } n = 0; \\ x \cdot x^{n-1} & \text{se } n > 0 \end{cases}$$

Exercícios

4) Usando a assinatura de método abaixo, implemente o algoritmo de busca binária recursivo.

```
def buscaBinaria(lista, elem, inicio, fim)
```

5) Dado um número inteiro $n > 0$, escreva um método recursivo que recebe n e retorna a somatória dos dígitos de n. Por exemplo: se $n = 1204$ seu método deverá retornar 7 que representa a soma dos dígitos $1 + 2 + 0 + 4$.
6) Dado um número inteiro $n > 0$, escreva um algoritmo recursivo que retorna o inverso de n, por exemplo, suponha que $n = 124$ seu método deverá retornar 421.

Exercícios

7) Dada uma lista contendo números reais, implemente um algoritmo recursivo que soma todos os elementos da lista.
8) Escreva um algoritmo recursivo que resolve o problema de Torres de Hanoi para *n* discos. Seu algoritmo deverá listar os movimentos necessários para movimentar os *n* da haste A para a haste C.

Referência Bibliográfica

- Puga e Rissetti - Lógica de Programação e Estrutura de Dados
- Ascêncio e Campos - Fundamentos da Programação de Computadores
- Forbelone e Eberspacher - Lógica de programação: a construção de algoritmos e estruturas de dados
- Documentação do Python - https://docs.python.org/3.8/
- Python Programming For Beginners: Learn The Basics Of Python Programming (Python Crash Course, Programming for Dummies) (English Edition). Kindle
- Python: 3 Manuscripts in 1 book: - Python Programming For Beginners - Python Programming For Intermediates - Python Programming for Advanced (English Edition). Kindle

Copyleft