Algoritmos e Estruturas de Dados II

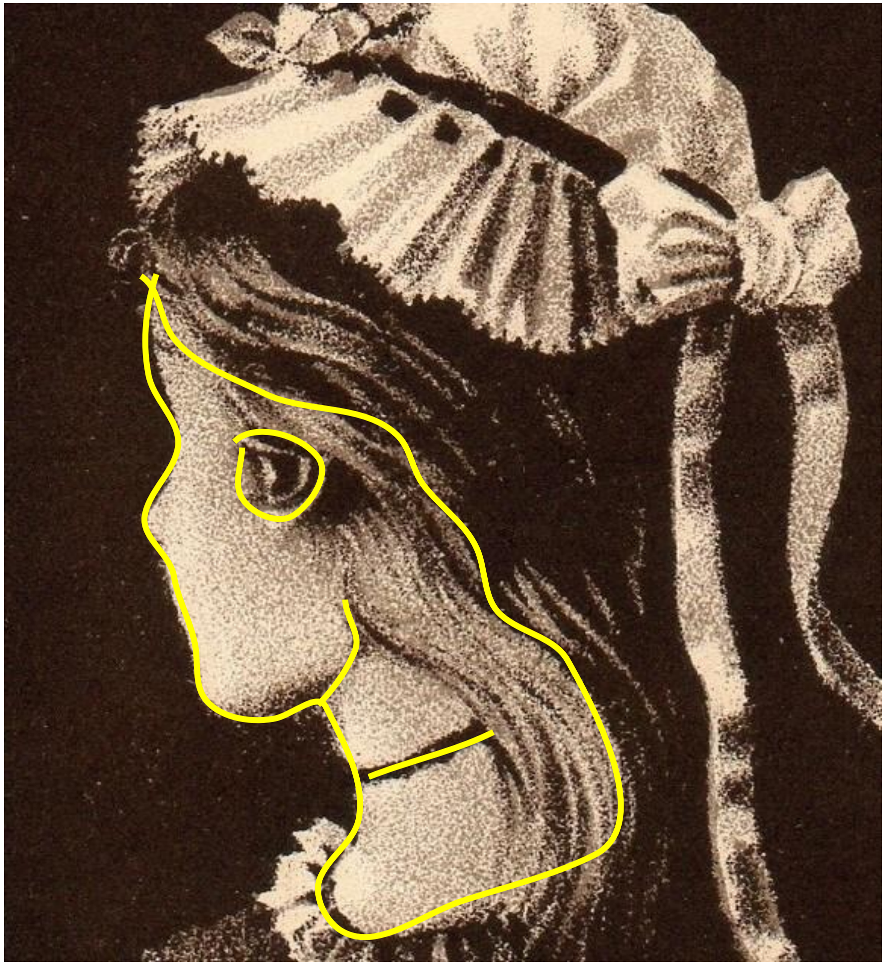
Recursão

versão 5.1

Fabiano Oliveira
fabiano.oliveira@ime.uerj.br

O que você está vendo?

O que você está vendo?



Recursão

```
função ObterPosicaoDoMaximo(B[], n: Inteiro): Inteiro
    //retorna a posição de B[1..n] com o elemento máximo
    ...

procedimento Ordenar(ref A[], n: Inteiro)
    var i, pmax: Inteiro

    para i ← n até 2 passo -1 faça
        pmax ← ObterPosicaoDoMaximo(A, i)
        A[pmax], A[i] ← A[i], A[pmax]
```

Recursão

- MUITO IMPORTANTE notar que:

    - A ***correção*** do ***algoritmo*** de `Ordenar()` pode ser analisada ***independentemente*** da correção do ***algoritmo*** de `ObterPosicaoDoMaximo()`

    - A ***depuração*** do ***algoritmo*** de `Ordenar()` pode ser feita ***independentemente*** da depuração do ***algoritmo*** de `ObterPosicaoDoMaximo()`

Recursão

- Uma função ***recursiva*** consiste de uma chamada a uma função especial: a própria função

- Embora ela tenha uma preocupação **a mais** por conta disso (condição de parada), ela não tem propriedades **a menos** (i.e., ambas as propriedades do slide anterior valem)

- O emprego da técnica de recursão é também chamada de ***divisão e conquista***

Recursão

- Uma função ***recursiva*** consiste de uma chamada a uma função especial: a própria função

- Embora ela tenha uma preocupação **a m**
disso (condição de parada), ela não tem
a menos (i.e., ambas as propriedades d
r
valem)

- O emprego da técnica de recursão é também chamada de ***divisão e conquista***

Recursão

- Ocorre quando a solução de um problema é descrita em função das soluções de instâncias menores do mesmo problema (subproblemas). Para tanto, são necessárias duas condições:
  - devem haver instâncias de problemas que sejam resolvidas diretamente (sem necessidade de se resolver subproblemas) **(casos base)**
  - as soluções de subproblemas sendo usadas devem ser de problemas "menores" e todos eles devem recair eventualmente nos casos bases **(caso geral)**

Recursão

- Algoritmo Geral:

```
procedimento/função funçãoRecursiva(...)
   se <expressão-detecção-caso-base> então
      ...
   senão
      ...
      funçãoRecursiva(...)
      ...
```

Recursão

```
função f(n: Inteiro): Inteiro
    //retorna n!
    se n = 0 então
        retornar 1
    senão
        retornar n*f(n-1)
```

Árvore de Recursão

```
f(4)=4*f(3)
```

Recursão

```
função f(n: Inteiro): Inteiro
    //retorna n!
    se n = 0 então
        retornar 1
    senão
        retornar n*f(n-1)
```

Árvore de Recursão

```
f(4)=4*f(3)
       |--- f(3)=3*f(2)
```

Recursão

```
função f(n: Inteiro): Inteiro
    //retorna n!
    se n = 0 então
        retornar 1
    senão
        retornar n*f(n-1)
```

Árvore de Recursão

```
f(4)=4*f(3)
       |--- f(3)=3*f(2)
                  |--- f(2)=2*f(1)
```

Recursão

```
função f(n: Inteiro): Inteiro
    //retorna n!
    se n = 0 então
        retornar 1
    senão
        retornar n*f(n-1)
```

Árvore de Recursão

```
f(4)=4*f(3)
       |--- f(3)=3*f(2)
                  |--- f(2)=2*f(1)
                             |--- f(1)=1*f(0)
```

Recursão

```
função f(n: Inteiro): Inteiro
    //retorna n!
    se n = 0 então
        retornar 1
    senão
        retornar n*f(n-1)
```

Árvore de Recursão

```
f(4)=4*f(3)
       |--- f(3)=3*f(2)
                  |--- f(2)=2*f(1)
                             |--- f(1)=1*f(0)
                                        |--- f(0)=1
```

Recursão

```
função f(n: Inteiro): Inteiro
    //retorna n!
    se n = 0 então
        retornar 1
    senão
        retornar n*f(n-1)
```

Árvore de Recursão

```
f(4)=4*f(3)
       |--- f(3)=3*f(2)
                  |--- f(2)=2*f(1)
                             |--- f(1)=1*f(0)=1
```

Recursão

```
função f(n: Inteiro): Inteiro
    //retorna n!
    se n = 0 então
        retornar 1
    senão
        retornar n*f(n-1)
```

Árvore de Recursão

```
f(4)=4*f(3)
       |--- f(3)=3*f(2)
                  |--- f(2)=2*f(1)=2
```

Recursão

```
função f(n: Inteiro): Inteiro
    //retorna n!
    se n = 0 então
        retornar 1
    senão
        retornar n*f(n-1)
```

Árvore de Recursão

```
f(4)=4*f(3)
          |--- f(3)=3*f(2)=6
```

Recursão

```
função f(n: Inteiro): Inteiro
    //retorna n!
    se n = 0 então
        retornar 1
    senão
        retornar n*f(n-1)
```

Árvore de Recursão

f(4)=4*f(3)=24

Recursão

```
função f(n: Inteiro): Inteiro
    //retorna n!
    se n = 0 então
        retornar 1
    senão
        retornar n*f(n-1)
```

Complexidade de Tempo: soma do número de passos de cada nó recursivo

Árvore de Recursão

```
                 Θ(1)
f(n)=n*f(n-1)
                                   Θ(1)
          |--- f(n-1)=(n-1)*f(n-2)
                              |--- ...........              Θ(1)
                                        |--- f(1)=1*f(0)
                                                     |--- f(0)
```

$$n \times \Theta(1) = \Theta(n)$$

Recursão

- Complexidade de Tempo:
  Se o método de contabilizar a soma dos passos dos nós da árvore de recursão se tornar difícil, tente também outros métodos apresentados no material

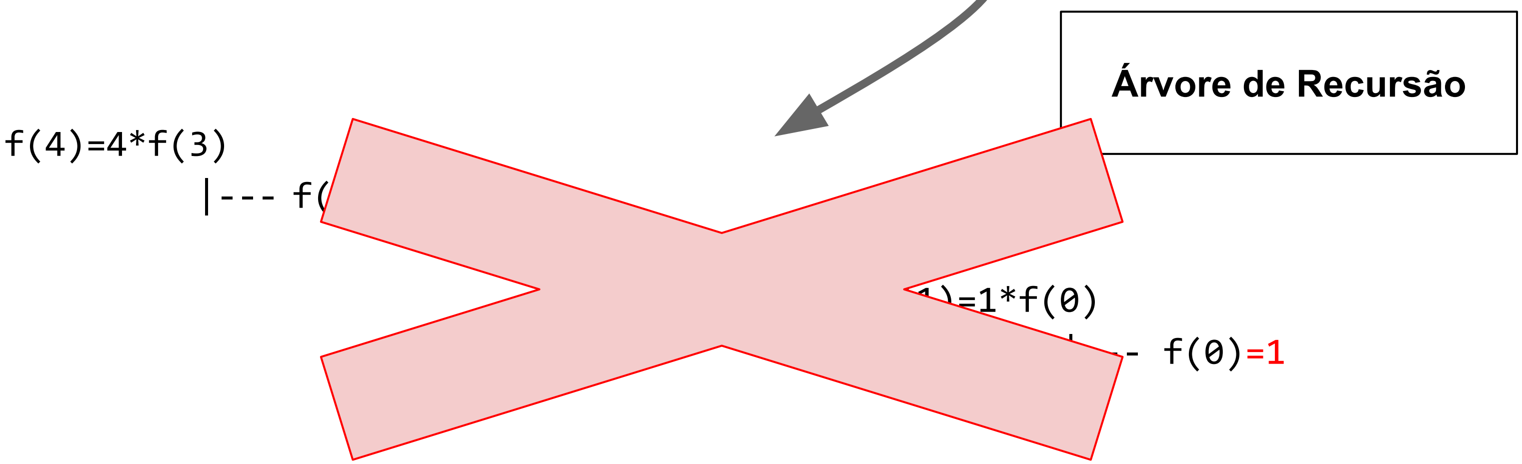
  [Análise de Complexidade de Algoritmos]

  na seção "Análise de Complexidade de Tempo em Algoritmos Recursivos"

Recursão

```
função f(n: Inteiro): Inteiro
    //retorna n!
    se n = 0 então
        retornar 1
    senão
        retornar n*f(n-1)
```

Não pense na árvore de recursão ao projetar um algoritmo recursivo!



Recursão

Projeto da recursão:

0. **[dividir para conquistar]:** decisão de usar recursão
1. **[caso geral]:** perguntar-se: "como o resultado de problemas menores (divisão) pode ajudar a resolver o problema original (conquista)?"
2. **[generalização]:** é necessário parametrizar mais o problema original?
3. **[caso base]:** determinar os casos para os quais não seja possível "dividir"; tais casos devem ser resolvidos à parte
4. **[memorizar]:** se os subproblemas repetem sistematicamente a mesma entrada, então guardar as saídas associadas às entradas
5. **[*reduzir a iteração*]:** se a árvore de recursão possuir profundidade elevada, então transformar em algoritmo iterativo equivalente

Recursão

- Exemplo: **Cálculo de Fatorial**

1. **[caso geral]:** como `fat(n-1)`, que resultará em (n-1)!, pode ajudar a resolver `fat(n)?` Com n! = n(n-1)!, então
   `fat(n) = n*fat(n-1)`
2. **[generalização]:** não foi necessário
3. **[caso base]:** se `n=0`, então `fat(n-1)` é indefinido. Para todo `n>0`, eventualmente se recai neste caso base.
4. **[memorizar]:** desnecessário
5. **[reduzir a iteração]:** desnecessário

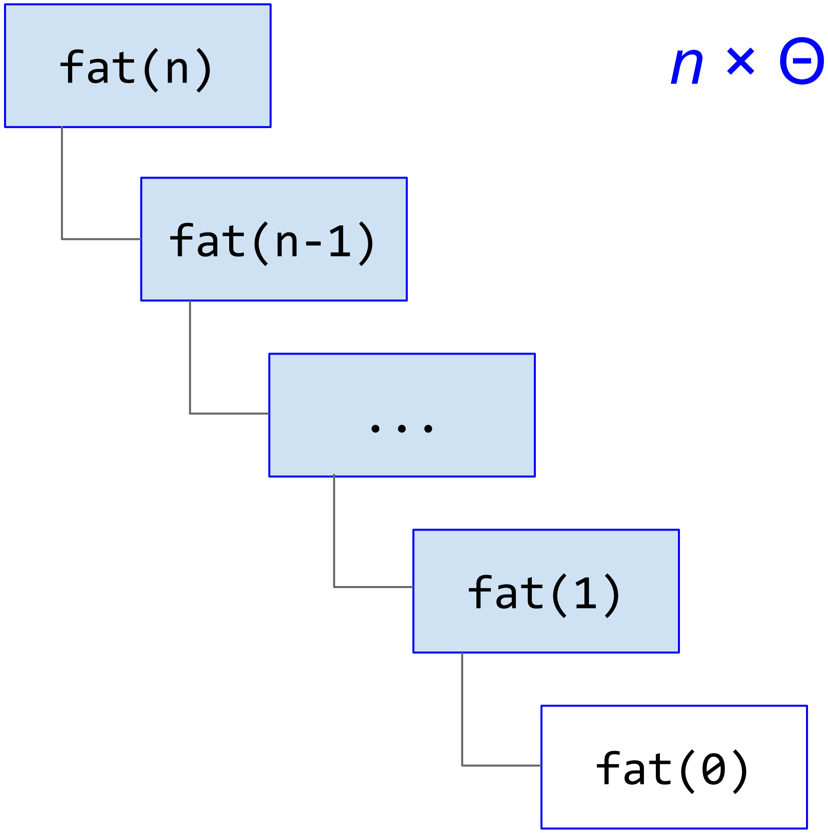
Recursão

- Exemplo: **Cálculo de Fatorial**

```
função fat(n: Inteiro): Inteiro
   se n = 0 então
      retornar 1
   senão
      retornar n * fat(n-1)
```

Recursão

- Análise de Complexidade:



$n \times \Theta(1) = \Theta(n)$

Recursão

- Exemplo: **Números de Fibonacci**
  - A série 1, 1, 2, 3, 5, 8, ... é conhecida como série de números de Fibonacci; nesta série,
    $F(1) = 1$, $F(2) = 1$, e
    $F(n) = F(n-1) + F(n-2)$, para todo $n \geq 3$
  - Os números de Fibonacci estão permeados na Natureza

Recursão

- Exemplo: **Números de Fibonacci**

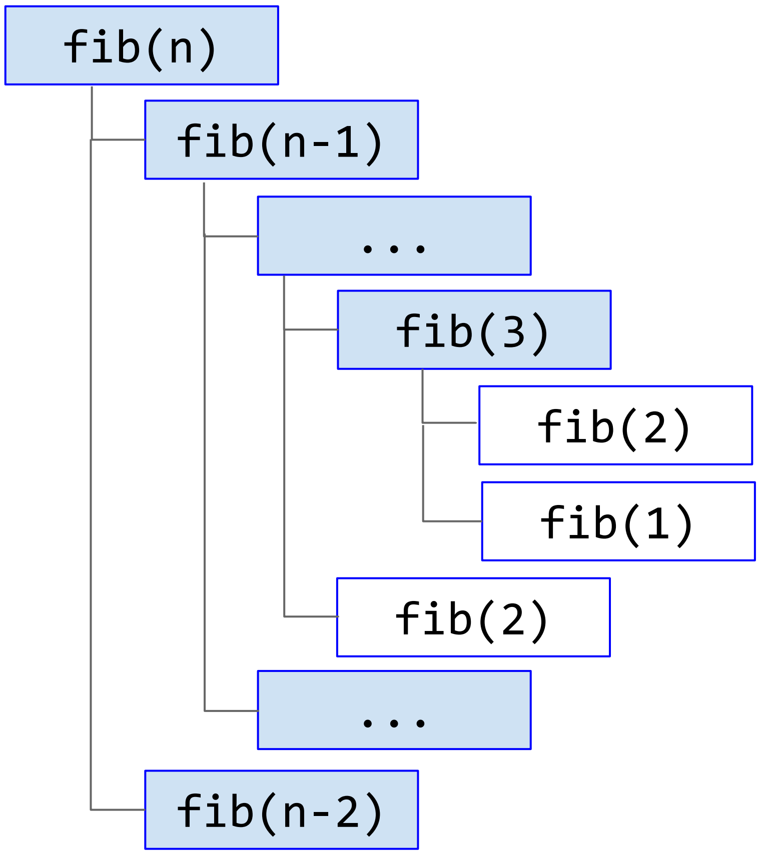
1. **[caso geral]:** como `fib(n')`, que resultará no `fib(n')` para todo `n'<n`, pode ajudar a resolver `fib(n)`?
   `fib(n) = fib(n-1)+fib(n-2)`
2. **[generalização]:** não foi necessário
3. **[caso base]:** se `n=0` ou `n=1`, então `fat(n-2)` é indefinido. Para `n>1`, eventualmente os problemas recaem nestes casos bases
4. **[memorização]:** necessário
5. **[reduzir a iteração]:** desnecessário

Recursão

```
função fib(n: Inteiro): Inteiro
   se n=1 ou n=2 então
      retornar 1
   senão
      retornar fib(n-1) + fib(n-2)
```

Recursão

- Análise de Complexidade:



$? \times \Theta(1) = ?$

Recursão

- Análise de Complexidade

Seja $T(n)$ a complexidade de tempo de `fib(n)`:

$T(n) = 1$, se $n \leq 2$. Caso contrário,

$T(n) = T(n-1) + T(n-2) + 1$ (como $T(n-1) > T(n-2)$)

$< T(n-1) + T(n-1) + 1 = 2T(n-1) + 1$

$< 2(2T(n-2) + 1) + 1 = 2^2T(n-2) + 2 + 1$

$< 2^2(2T(n-3) + 1) + 2 + 1 = 2^3T(n-3) + 2^2 + 2^1 + 1$

$< \ldots < 2^iT(n-i) + 2^{i-1} + \ldots + 2 + 1$ (para $n-i = 1$)

$= 2^{n-1}T(1) + 2^{n-2} + \ldots + 2 + 1 =$

$= 2^{n-1} + \ldots + 2 + 1 = 2^n - 1$

$= O(2^n)$

Recursão

- Análise de Complexidade

Por outro lado:

$T(n) = 1$, se $n \leq 2$. Caso contrário,

$T(n) = T(n-1) + T(n-2) + 1$ (como $T(n-1) > T(n-2)$)

$> T(n-2) + T(n-2) + 1$

$> 2T(n-2) > 2^2 T(n-4) > 2^3 T(n-6) > ... > 2^i T(n-2i)$

Fazendo-se $n - 2i = 1$, temos que $i = (n-1)/2$. Logo:

$T(n) > 2^{(n-1)/2} T(1) = 2^{(n-1)/2}$

$= \Omega(\sqrt{(2)}^n)$

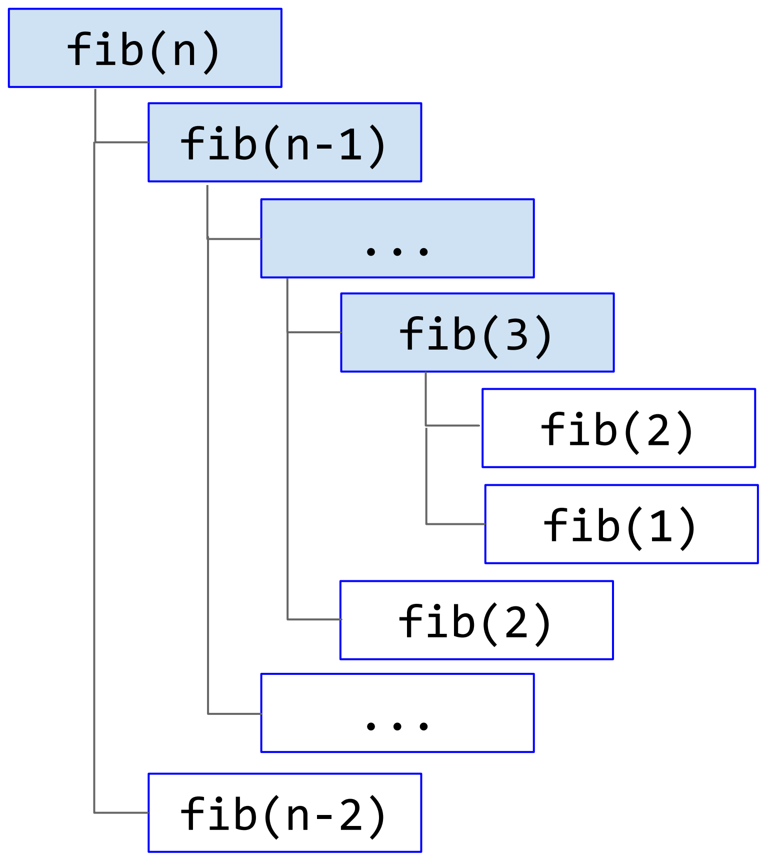
Recursão

- Versão com Memorização:

```
var T[1..n]: Inteiro ← -1
função fib(n: Inteiro): Inteiro
   se T[n]=-1 então
      se n=1 ou n=2 então
         T[n] ← 1
      senão
         T[n] ← fib(n-1) + fib(n-2)
   retornar T[n]
```

Recursão

- Análise de Complexidade:



$(n\text{-}2) \times \Theta(1) = \Theta(n)$

Recursão

Lições aprendidas:

- Em geral, a recursão é o meio natural de se computar funções definidas de forma recursiva (como exemplos, Fatorial e Fibonacci)

- Nem sempre conduzem diretamente a algoritmos eficientes ─ no caso de repetição de subproblemas, é necessário memorizar!

Recursão

Veremos que, não raramente, podemos resolver problemas de forma eficiente usando recursão mesmo em problemas que não são definidos recursivamente!

Com exercícios, podemos "enxergar" a forma recursiva escondida na descrição destes problemas

Recursão

- **Exercícios**:
  - Média Aritmética:
    $MA(N) = (a_1 + \ldots + a_N)/N$
    - Defina $MA(N)$ em função de $MA(N\text{-}1)$, a_N e N

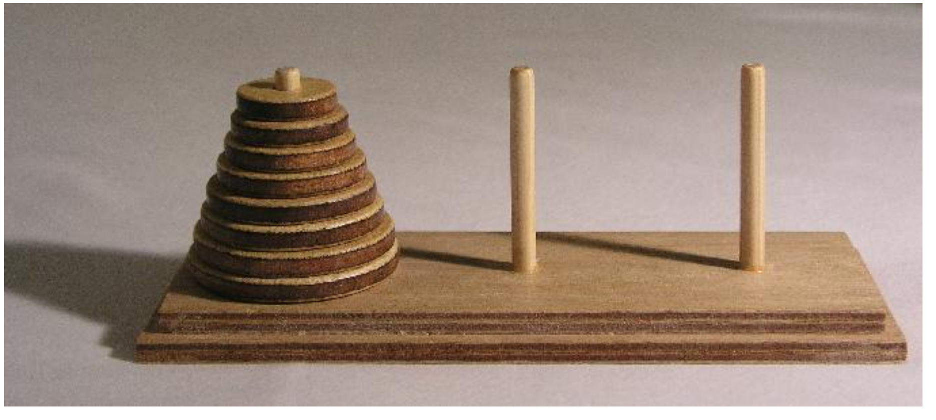
  - Média Ponderada:
    $MP(N) = (p_1 a_1 + \ldots + p_N a_N)/(p_1 + \ldots + p_N)$
    - Defina $MP(N)$ em função de $MP(N\text{-}1)$, a_N , p_N e N (**só?!**)

Recursão

- **Exercícios**:
  - $C(n, p)$ é o número de combinações de n elementos, tomados p a p.
    $C(n, p) = n! / ((n - p)!\, p!)$
    - Defina $C(n, p)$ em função de $C(n, p-1)$ e $C(n, 0)$
    - Defina $C(n, p)$ em função de $C(n-1, p)$ e $C(n, n)$
    - Defina $C(n, p)$ em função de $C(n-1, p-1)$ e $C(n, 0)$

Recursão

- Exemplo: **Torre de Hanoi**
    - Objetivo: transferir uma série de discos dispostos numa haste para uma outra haste, usando-se uma terceira haste auxiliar, com o seguinte procedimento:
        - Iterativamente, escolhe-se um disco de qualquer haste que esteja por cima e o coloca-se numa outra haste a escolha, desde que tal disco seja o menor disco daquela haste escolhida



Recursão

Como resolver?

```
procedimento hanoi(n: Inteiro)
//Supõe: n ≥ 0
//Garante:
       escrita de tuplas (Torre Origem -> Torre Destino)
       com a solução das Torres de Hanoi com torres
       'A', 'B', 'C' e n discos na torre 'A' com destino
       a torre 'B' usando a torre 'C' como auxílio
   ???

ler(n)
hanoi(n)
```

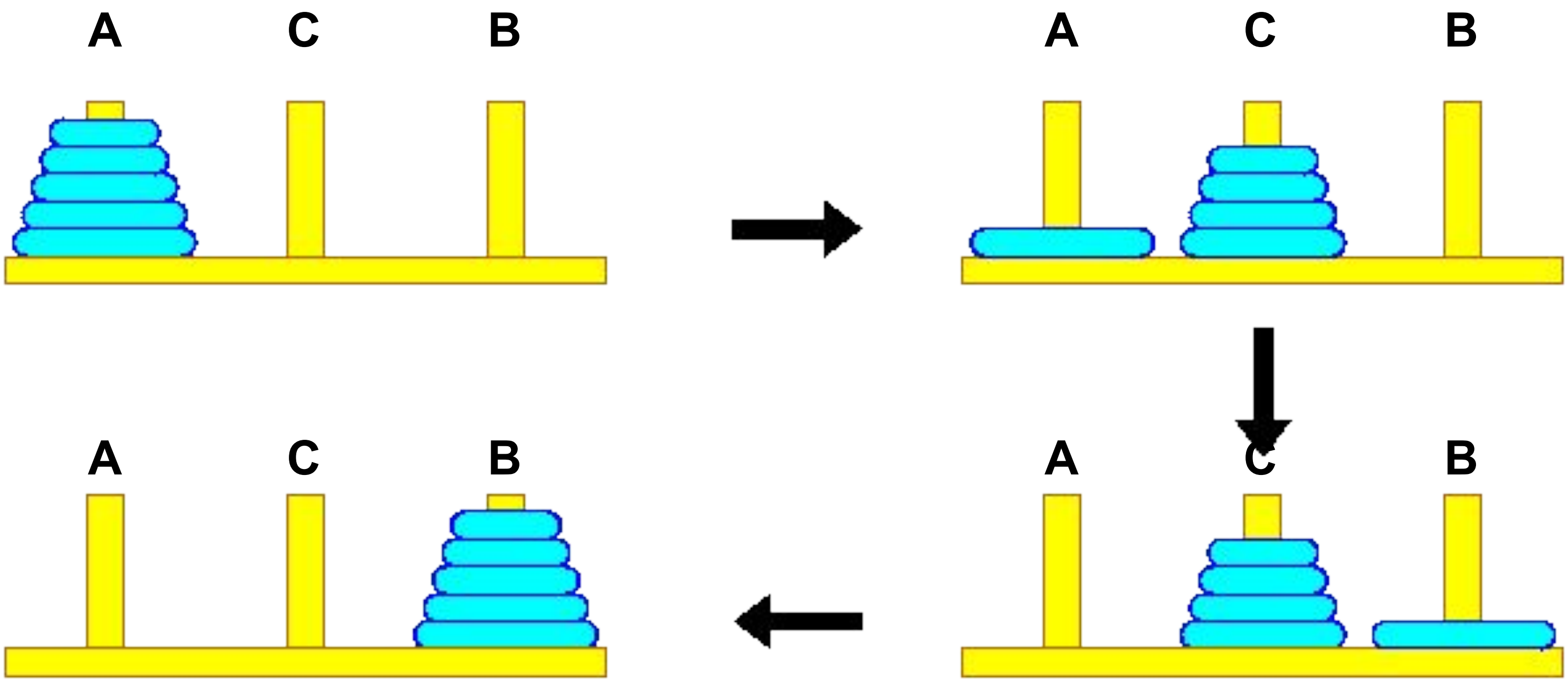
Recursão

Pergunta-chave que deve ser feita para se resolver problemas mais complexos usando recursão **[caso geral]**:

"Se eu já soubesse resolver este mesmo problema para entradas menores, como isso me ajudaria a resolver o problema com a entrada que me foi apresentada?"

Recursão

Uma ideia **[caso geral]**:



Recursão

- Note que apesar da solução apresentar um comportamento recursivo, ela não pode efetivamente usar chamadas recursivas pois a haste onde estão os discos ou aquela para onde eles irão é outra
- **Solução**: generalizar a definição do problema! **[generalização]**

Recursão

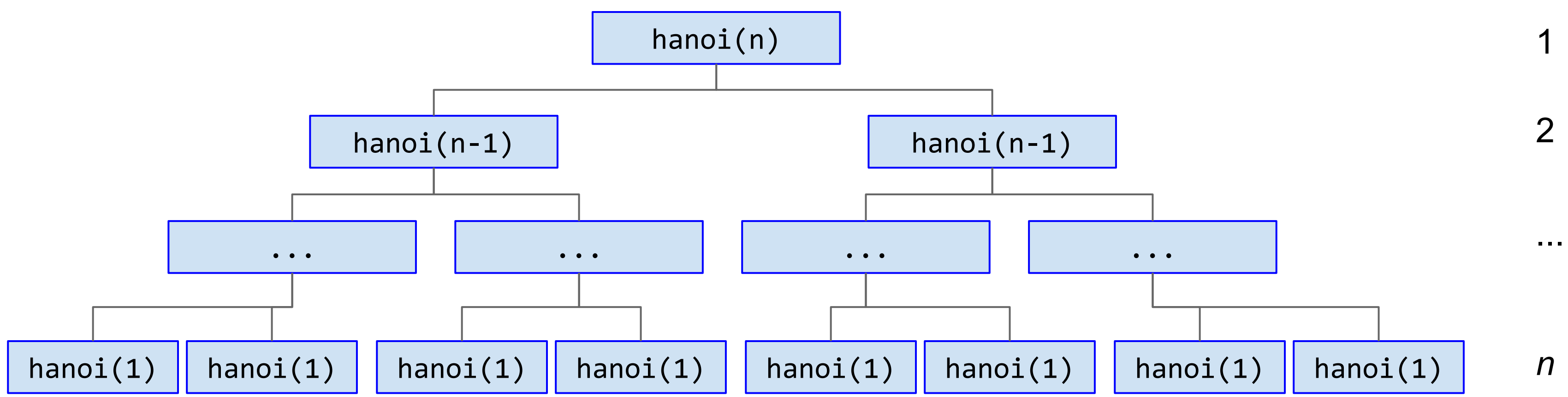
```
procedimento hanoi(n: Inteiro, TOrig,TDest,TAux: Caractere)
    //Supõe: n ≥ 0
    //Garante:
           escrita de tuplas (Torre Origem -> Torre Destino)
           com a solução das Torres de Hanoi com torres
           TOrig, TDest, TAux e n discos da torre
           TOrig com destino a TDest usando a TAux
           como auxílio
       ???

ler(n)
hanoi(n, "A", "B", "C")
```

Recursão

```
procedimento hanoi(n: Inteiro,
                   TOrig,TDest,TAux: Caractere)
  //Supõe: n ≥ 0
  //Garante: escrita de tuplas...
    se n > 0 então
      hanoi(n-1, TOrig, TAux, TDest)
      escrever(TOrig, " -> ", TDest)
      hanoi(n-1, TAux, TDest, TOrig)
```

Recursão

- Análise de Complexidade:



$$1+2+2^2+\ldots+2^{n-1}$$
$$= 2^n - 1$$

$$\therefore (2^n - 1) \times \Theta(1) = \Theta(2^n)$$

Recursão

- Análise de Complexidade
  - Seja T(n) a complexidade de tempo de `hanoi(n)`

T(n) = 1, se n = 0. Caso contrário,

$T(n) = 2\,T(n-1) + 1 = 2\,(2\,T(n-2) + 1\,) + 1 =$

$= 2^2\,T(n-2) + 2 + 1 =$

$= 2^3\,T(n-3) + 2^2 + 2^1 + 2 =$

$= 2^i\,T(n-i) + 2^{i-1} + ... + 2 + 1 =$

Fazendo-se $n - i = 0$, temos que $i = n$. Logo:

$T(n) = 2^n\,T(0) + 2^{n-1} + ... + 2 + 1 =$

$= 2^n + 2^{n-1} + ... + 2 + 1 = 2^{n+1}-1$

$= \Theta(2^n)$

Recursão

- Análise de Complexidade
  - Pode-se mostrar (não demonstraremos) que o número de movimentos para ***<u>qualquer</u>*** solução é no mínimo 2^n - 1.
  - Logo, o algoritmo recursivo é ótimo

Recursão

Lições aprendidas:

- Existe uma pergunta-chave que devemos nos fazer para encontrar a recursão: como usar a resolução de instâncias menores do problema para resolver o problema geral?

- É muito comum que a recursão envolva o passo de generalizar a assinatura do procedimento/função análogo não-recursivo

- Sem se desprender do processo de mentalmente “fazer o chinês” do algoritmo recursivo, é praticamente impossível elaborar recursões mais elaboradas

Recursão

- Exemplo: **Busca em Vetor**

Considere a busca de um elemento num vetor:

```
função busca(Lista[], N, x: Inteiro): Inteiro
//Supõe: |Lista| ≥ N, x ∈ Lista[1..N]
//Garante: Lista[retorno] = x
```

"Se eu já soubesse resolver este mesmo problema para entradas menores, como isso me ajudaria a resolver o problema com a entrada que me foi apresentada?"

```
ler (N, Lista[1..N], x)
escrever (busca(Lista, N, x))
```

Recursão

- Vejamos algumas estratégias **[caso geral]**:

1. Ou o elemento procurado é o último elemento, ou está entre os N-1 primeiros elementos.

2. Ou o elemento procurado é o primeiro elemento, ou está entre os N-1 últimos elementos (requer generalizar a assinatura da função!)

3. Ou o elemento procurado está na posição m (para algum $1 \leq m \leq N$, ou está entre os m-1 primeiros elementos, ou está na porção m+1..N do vetor (requer generalizar a assinatura da função!)

Recursão

```
função busca_1(Lista[], N, x: Inteiro): Inteiro
   se Lista[N] = x então
      retornar N
   senão
      retornar busca_1(Lista, N-1, x)

ler (N, Lista[1..N], x)
escrever (busca_1(Lista, N, x))
```

Recursão

```
função busca_2(Lista[], inicio, fim, x: Inteiro): Inteiro
    se Lista[inicio] = x então
        retornar inicio
    senão
        retornar busca_2(Lista, inicio+1, fim, x)


ler (N, Lista[1..N], x)
escrever (busca_2(Lista, 1, N, x))
```

Recursão

```
função busca_3(Lista[], inicio, fim, x: Inteiro): Inteiro
    var pos: Inteiro
    se inicio > fim então
        retornar -1
    senão
        meio ← (inicio + fim) div 2
        se Lista[meio] = x então
            retornar meio
        senão
            pos ← busca_3(Lista, inicio, meio-1, x)
            se pos = -1 então
                retornar busca_3(Lista, meio+1, fim, x)
            senão
                retornar pos
```

Recursão

- **Exercício**:
  Mostre que a complexidade de tempo de pior caso das três versões de busca anteriores é $\theta(N)$.

Recursão

- Mas se o vetor de entrada estiver ordenado?

```
função buscaOrd(Lista[], N, x: Inteiro): Inteiro
//Supõe: |Lista| ≥ N, x ∈ Lista[1..N],
          Lista[1..N] ordenado
//Garante:  Lista[retorno] = x
   ???
```

É possível tirar proveito desta condição?

Recursão

- Busca Binária

  É uma melhoria em relação à terceira estratégia da busca linear apresentada em slides anteriores: com o vetor ordenado, não é necessário buscar em ambas as metades do vetor.

2	4	...	13	28	30	39	...	72

N/2

onde está o 54? e o 8?

Recursão

```
função BuscaBinaria(Lista[], inicio, fim, x: Inteiro): Inteiro
//Supõe:   inicio ≤ fim ≤ |Lista|, x ∈ Lista[inicio..fim]
           Lista[inicio..fim] ordenado
//Garante: Lista[retorno] = x
    var meio: Inteiro
    meio ← (inicio + fim) div 2
    se Lista[meio] = x então
        retornar meio
    senão se Lista[meio] > x então
        retornar BuscaBinaria(Lista, inicio, meio-1, x)
    senão // Lista[meio] < elem
        retornar BuscaBinaria(Lista, meio+1, fim, x)

ler (N, Lista[1..N], x)
escrever (BuscaBinaria(Lista, 1, N, x))
```

Recursão

- Análise de Complexidade
  - Versão recursiva:
    - Seja $T(N)$: complexidade de tempo de `BuscaBinaria` em uma porção de N elementos

      $T(N) = 1$, se $N = 1$

      $T(N) \leq T(N/2) + 1$

      $\leq T(N/2^2) + 2$

      $\leq T(N/2^3) + 3 \leq \ \ldots \leq T(N/2^i) + i$

      Fazendo-se $N/2^i = 1$, temos que $i = \lg N$. Logo:

      $\leq T(1) + \lg N$

      $= O(\lg N)$

Recursão

- Análise de Complexidade:
  - **Busca Linear**: no pior caso, *N* comparações
  - **Busca Binária**: no pior caso, lg *N* comparações

N	**Tempo** (10^6 passos/s)	**lg *N***	**Tempo** (10^6 passos/s)
128	< 0,1 s	7	< 0,1 s
1.024	~ 0,1 s	10	< 0,1 s
1.048.576	~ 1 s	20	< 0,1 s
1.073.741.824	~ 17 min	30	< 0,1 s
1.099.511.627.776	~ 12 dias	40	< 0,1 s

Recursão

Lições aprendidas:

- Quando uma Recursão depende da solução de mais de um subproblema recursivo, ela será melhor se houver um balanceamento no tamanho destes subproblemas

Recursão

- Exemplo: **Ordenação de Vetores**

Usando a pergunta-chave:

Como a ordenação de
vetores com menos que N elementos
pode ajudar a ordenar um
vetor com N elementos?

Recursão

- Consulte no material de "Algoritmos de Ordenação":

1. InsertionSort
2. SelectionSort
3. BubbleSort
4. MergeSort
5. QuickSort

Recursão

Lições aprendidas:

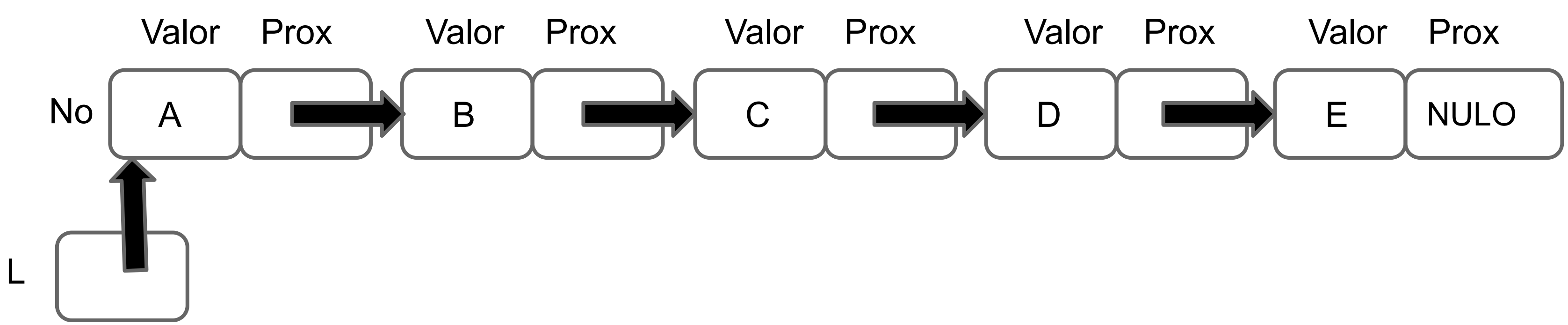
- Nota-se que Ordenação de vetores é um exemplo que reforça as seguintes lições anteriores:
  - Quanto maior o balanceamento de tamanhos dos subproblemas, mais eficiente o algoritmo recursivo

  - A pergunta-chave "como usar a resolução de instâncias menores do problema para resolver o problema geral?" praticamente resultou diretamente na construção dos algoritmos de ordenação apresentados

  - Sem se desprender do processo de mentalmente “fazer o chinês” do algoritmo recursivo, é praticamente impossível elaborar recursões mais elaboradas

Recursão

- Exemplo: **Operações em Lista Encadeada**



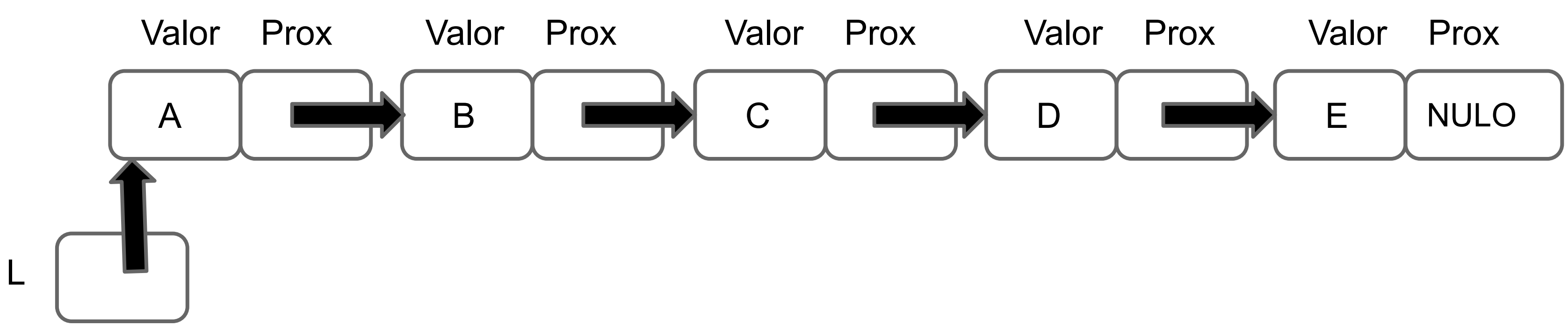
Recursão

- Como seria a implementação iterativa de, digamos, anexar um novo valor (ao final)?

```
procedimento Anexar(ref L: ^No, v: Inteiro)
    se L = NULO então
        alocar(L)
        L^.Valor, L^.Prox ← v, NULO
    senão
        var p: ^No ← L
        enquanto p^.Prox ≠ NULO faça
            p ← p^.Prox
        Alocar(p^.Prox)
        p^.Prox^.Valor, p^.Prox^.Prox ← v, NULO
```

Recursão

- Definição recursiva de listas encadeadas:
  Uma lista encadeada é:
  - um ponteiro nulo
  - um ponteiro p para uma estrutura tal que:
    - p^.Valor é um dado armazenado
    - p^.Prox é uma lista encadeada



Recursão

- Outra solução, portanto:

```
procedimento Anexar(ref L: ^No, v: Inteiro)
   se L = NULO então
      Alocar(L)
      L^.Valor, L^.Prox ← v, NULO
   senão
      Anexar(L^.Prox, v)
```

Recursão

Lições aprendidas:

- Os algoritmos recursivos muitas vezes são mais simples que os equivalentes iterativos

- Algoritmos recursivos são mais naturais para se resolver problemas que já possuem estrutura/definição recursiva

Recursão

- **Reduzir a Iteração**: O limite de empilhamento de funções pode ser relativamente baixo para atender a solução de determinado problema. Contraste a execução nas versões recursiva vs. iterativa de:
  - Busca Linear em Vetores (problema com recursão!)
  - Fibonacci (problema com recursão!)
  - Fatorial (problema com recursão desprezível)
  - Busca Binária (problema com recursão desprezível)

Recursão

- **Solução**: Podemos sempre transformar um algoritmo recursivo em outro equivalente iterativo com o uso de recursão de cauda!

- Uma ***recursão de cauda*** é uma recursão em que o resultado da função consiste do resultado direto da chamada recursiva, sem transformações. As recursões de cauda podem ser transformadas em comandos iterativos diretamente, eliminando a recursão:

Recursão

- Transformação:

função $f(p_1,\ldots,p_k)$
    se $b(p_1,\ldots,p_k)$ **então**
        retornar $h(p_1,\ldots,p_k)$
    senão
        ...
        retornar $f(q_1,\ldots,q_k)$

função $f(p_1,\ldots,p_k)$
    enquanto não $b(p_1,\ldots,p_k)$ **faça**
        ...
        $p_1,\ldots,p_k \leftarrow q_1,\ldots,q_k$
    retornar $h(p_1,\ldots,p_k)$

Recursão

- Exemplo:

```
função fat(n: Inteiro): Inteiro
   se n = 0 então
      retornar 1
   senão
      retornar n * fat(n-1) (não é de cauda!)
```

Recursão

- Exemplo:

```
função fat(n: Inteiro): Inteiro
   retornar fat-c(n,1)

função fat-c(n: Inteiro, a: Inteiro): Inteiro
   //retorna a*n!
   se n = 0 então
      retornar a
   senão
      retornar fat-c(n-1,a*n) (de cauda!)
```

Recursão

- Exemplo:

```
função fat-c(n: Inteiro, a: Inteiro): Inteiro
    //retorna a*n!
    enquanto n > 0 faça
        n,a ← n-1,a*n
    retornar a
```

https://startuplab.io/post/recursion

Exercícios

Recursão

1. Analise os problemas de estouro de pilha e de subproblemas repetidos nas diversas recursões dadas neste material.

2. Modifique um algoritmo recursivo de modo a eliminar seu caso base, deixando apenas o tratamento do caso geral, que sempre é executado. Implemente em alguma linguagem e o coloque em execução. O resultado foi o esperado?

3. Faça um algoritmo recursivo que verifique se uma cadeia, dada por um vetor C[1..N]: Caractere, é um palíndromo (uma cadeia que pode ser lida da esquerda para a direita ou vice-versa visitando a mesma sequência se símbolos em ambas as leituras).

4. Considere o algoritmo-modelo abaixo e determine a complexidade de tempo para os preenchimentos de `<A>` e `<B>` conforme os vários itens:

```
função f(N: Inteiro): Inteiro
    se N ≤ 1 então
        retornar 0
    senão
        <A>
        retornar 1 + <B>
```

Recursão

4.

a. <A> = ∅ ; <B> = f(N-1)

b. <A> = ∅ ; <B> = f(N-4)

c. <A> = ∅ ; <B> = f(⌊N/4⌋)

d. <A> =

```
para i ← 1 até N faça
    escrever (i)
```

<B> = f(N-1)

e. <A> =

```
para i ← 1 até N faça
    escrever (i)
```

<B> = f(N/2)

f. <A> =

```
para i ← 1 até N faça
    escrever (i)
```

<B> = f(N/2) + f(N/2) // não troque por 2*f(N/2)

Recursão

5. Considere que uma lista encadeada consiste de ponteiro para uma estrutura No que contém 2 campos: Valor: Inteiro, Próximo: ^No. Faça algoritmos recursivos que, dado uma lista L: ^No, determine:
    a. o número de valores
    b. o produto dos valores
    c. o maior valor
    d. o último valor
    e. o k-ésimo valor (k dado como entrada; k=1 corresponde ao primeiro, k=2 ao segundo, k=3 ao terceiro, etc.)
    f. o k-ésimo último valor (k dado como entrada; k=1 corresponde ao último, k=2 ao penúltimo, k=3 ao antepenúltimo, etc.)
    g. busque a posição de um valor dado como entrada
    h. remova um valor dado como entrada
    i. Verdadeiro ou Falso, indicando se os valores estão em ordem ascendente
    j. a média dos valores
    k. uma outra lista ligada com os valores da lista original sem repetições
    l. uma outra lista ligada, com o primeiro valor da lista original somado com o segundo, o segundo com o terceiro, etc.
    m. uma outra lista ligada com todos os valores da lista original somados dois a dois
    n. uma outra lista, com os elementos da lista original em ordem invertida

Recursão

6. Elabore algoritmos recursivos que:
    a. compute a^b para dois números inteiros a,b dados de entrada em uma linguagem que não possui a operação de potenciação, mas possui a operação de multiplicação/divisão)
    b. compute $a \cdot b$ para dois números inteiros a,b dados de entrada em uma linguagem que não possui a operação de multiplicação, mas possui a operação de soma/subtração
    c. compute a + b para dois números inteiros a,b dados de entrada em uma linguagem que não possui a operação de soma, mas possui a operação de incrementar/decrementar o valor de uma variável
    d. dados naturais N e p, compute o valor aproximado de $\sqrt{N}$ com erro máximo $\varepsilon = 10^{-p}$ (i.e., x é uma resposta válida se $|x - \sqrt{N}| \leq \varepsilon$) em uma linguagem que não possui a operação de radiciação, mas possui as 4 operações aritméticas básicas. O algoritmo deve ter tempo $O(p + \lg N)$.

Recursão

7. Faça algoritmos recursivos que, numa árvore binária onde cada nó da árvore consiste de uma estrutura NoArvore que contém 3 campos: Valor: Inteiro, NoEsquerdo, NoDireito: ^NoArvore, computem:
   a. o número de valores
   b. o produto dos valores
   c. a soma dos valores em folhas da árvore
   d. o maior valor
   e. uma lista ligada com os nós que não são folhas
   f. V ou F, indicando se a árvore é um max-heap (uma árvore é um max-heap se, para qualquer subárvore T desta árvore, o valor da raiz de T for maior ou igual que os valores dos filhos esquerdo e direito da raiz de T)
   g. o menor valor dentre aqueles associados a nós que não possuem subárvores vazias
   h. a maior soma de valores dos nós de uma subárvore (uma subárvore de T é a árvore que se obtém tomando-se um nó X de T e eliminando-se de T todos os nós que não sejam X ou não descendam de X)

Recursão

8. Faça um algoritmo recursivo que compute o valor de uma expressão aritmética dada como cadeia de entrada E[1..N]: Caractere. A cadeia pode conter apenas números, parênteses, +, -, *, e /. Exemplos:
    - Entrada: 3+4*2 ; Saída: 11
    - Entrada: (3+4)*2 ; Saída: 14
    - Entrada: (3+4)*2+4/2 ; Saída: 16
    - Entrada: ((3+4)*2+4)/2 ; Saída: 9
    - Entrada: (3+4*2+4))/2 ; Saída: ERRO DE SINTAXE

    Para isso, leve em conta que uma expressão é definida recursivamente como:
    Expressão = Fator+Expressão OU Fator-Expressão OU Fator
    Fator = Termo*Fator OU Termo/Fator OU Termo
    Termo = <inteiro> OU (Expressão)

Recursão

9. Escreva um algoritmo para as seguintes variantes do problema da Torre de Hanói:
   a. Além das regras originais do problema, acrescenta-se aquela que impede mover discos entre as torres A e B diretamente; ou seja, todos os movimentos de discos são a partir da torre C ou para a torre C.
   b. Além das regras originais do problema, acrescenta-se aquela que permite movimentos de discos somente da torre A para a torre C, da torre C para a torre B, e da torre B para a torre A.

10. Dado um vetor de naturais A[1..N], encontre a maior sequência não-decrescente de seus elementos. Uma sequência não-decrescente dos elementos de A é o maior k tal que existam índices $i_1 < i_2 < \ldots < i_k$ com $A[i_1] \leq A[i_2] \leq \cdots \leq A[i_k]$. O algoritmo deve ter complexidade de tempo $O(N^2)$. Exemplo: Entrada: N=9, A=[1,2,3,2,3,4,3,4,5]. Saída: 1,2,3,4,5 (portanto, k=5 e $i_1=1$, $i_2=2$, $i_3=3$, $i_4=6$, $i_5=9$)

11. Dados um vetor de inteiros A com N elementos, determine em tempo O(N lg N) a porção de A cuja soma dos elementos é maximizada, isto é, determinar $1 \leq i \leq j \leq N$ tais que $A[i]+A[i+1]+\cdots+A[j-1]+A[j]$ é máximo. (Note que o algoritmo força-bruta é $O(N^2)$.)

Recursão

12. Dados vetores A com M elementos e B com N elementos, ambos A e B ordenados, encontre o k-ésimo menor valor dentre os elementos de A[1..M] e B[1..N] com um algoritmo de tempo:
    a. O(máx{M, N})
    b. O(lg máx{M, N})

13. Dado um natural N, determine em tempo O(lg N) o número de algarismos "2" que ocorrem na lista de números de 1 a N.

14. Dados um natural T e um vetor M[1..N], determine o número de maneiras distintas de dar um troco de valor T usando-se moedas de valores M[1], M[2], …, M[N] cuja quantidade que pode ser usada de cada valor é ilimitada. Exemplo: Entrada: N = 4; M = [1, 5, 10, 25]; T = 11; Saída: 4 (referente aos trocos (11 moedas de 1), (6 moedas de 1, 1 moeda de 5), (1 moeda de 1, 2 moedas de 5), (1 moeda de 1, 1 moeda de 10).

Recursão

15. No problema da Torre de Hannoi, escreva um algoritmo que, ao invés de "pino de origem ⇒ pino de destino", imprima "A ⇒ B", onde A é o diâmetro do disco a mover e B é o diâmetro do disco que apoiará o disco sendo movido ou, na ausência de tal disco, o pino de destino. Suponha que os diâmetros dos discos são de 1 a N, onde N é o número de discos. A descoberta do diâmetro do disco que será movido deve gastar tempo constante. Exemplo: **Entrada**: N=3, **Saída**: 1 ⇒ B, 2 ⇒ C, 1 ⇒ 2, 3 ⇒ B, 1 ⇒ A, 2 ⇒ 3, 1 ⇒ 2.

16. Um circo está planejando um novo número onde acrobatas sobem uns nos ombros dos outros formando uma torre humana. Um acrobata X pode subir no ombro de outro Y se X for mais leve e mais baixo que Y. Dados um vetor de naturais H[1..N] indicando a altura de cada um dos N acrobatas e um vetor de naturais P[1..N] indicando os respectivos pesos, determinar a torre com o maior número de acrobatas possível. Exemplo:
    Entrada: N = 6, H = (165, 170, 156, 175, 160, 168) e P = (50, 44, 45, 95, 48, 55)
    Saída: 5 (referente à torre (cima) 3 → 5 → 1 → 6 → 4 (baixo))

Recursão

17. Dado um natural N, determinar o número de parentizações bem-formadas distintas nas quais ocorrem exatamente N abertura de parênteses. Exemplo:
    Entrada: N = 3
    Saída: 5 (referente a "()(())", "()()()", "(())()", "((()))", "(()())")

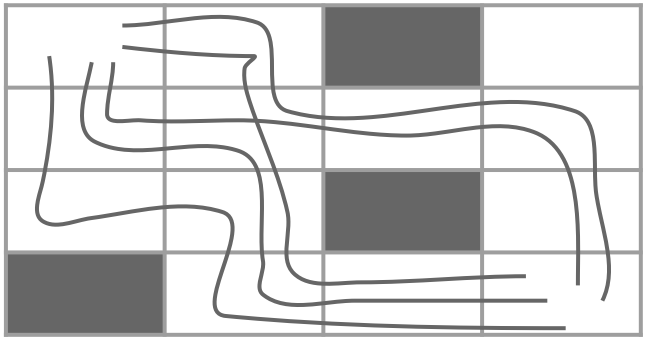
18. Elabore uma função recursiva que compute em tempo θ(N) a valiação de um polinômio de grau N. Mais especificamente, dados um inteiro N, um vetor A = [a_N, a_{N-1}, …, a_1, a_0] e um valor x, esta função deve computar

$$a_N x^N + a_{N-1} x^{N-1} + \ldots + a_1 x + a_{0.}$$

    usando-se apenas as operações de multiplicação e soma.

Recursão

18. Dados um natural N e vetores de naturais X[1..M] e Y[1..M], determinar quantos caminhos distintos um robô pode percorrer em uma matriz NxN até chegar no canto inferior direito (célula (N, N)), sabendo-se que o robô começa na célula do canto superior esquerdo (célula (1, 1)), que a cada passo o robô anda para a célula da direita ou para a célula de baixo da posição corrente, que ele não pode ultrapassar os limites da matriz e nem entrar em certas células, chamadas de obstáculos. Há M obstáculos, o obstáculo i localizado na célula (X[i], Y[i]) (X[i] células a partir do canto esquerdo da matriz, Y[i] células a partir do canto superior da matriz). Exemplo: Entrada: M = 3; N = 4; X = [1, 3, 3], Y = [4, 1, 3]; Saída: 5 (representados abaixo)



Recursão

19. O seguinte algoritmo ordena os N primeiros elementos do vetor B para alguns valores específicos de k. Dentre tais valores, qual o menor? Qual a complexidade do algoritmo para tal menor valor?

```
procedimento Ordena(B[]: Inteiro, inicio, fim, k: Inteiro)
     se fim-inicio = 1 então
          se B[inicio] > B[fim] então
               B[inicio], B[fim] ← B[fim], B[inicio]
     senão se fim-inicio > 1 então
          var t: Inteiro ← (fim-inicio+1) div 3
          para j ← 1 até k faça
               se j mod 2 = 1 então
                    Ordena(B, inicio, fim-t, k)
               senão
                    Ordena(B, inicio+t, fim, k)

var N, k: Inteiro
ler(N, k)
var B[1..N]: Inteiro
ler(B[1..N])
Ordena(B, 1, N, k)
```

Recursão

20. Dado um vetor B[1..N] de inteiros, pede-se determinar a ordem de contrações a serem feitas tal que o resultado final seja um número P dado. Cada contração substitui dois elementos x,y sucessivos na sequência pela diferença x-y. [Maratona ACM, 1998, América do Sul]

21. Considere 3 vasos v_1, v_2 e v_3 contendo água. Cada vaso v_i, para $1 \le i \le 3$, possui capacidade total para c_i ml, encontrando-se preenchido inicialmente com s_i ml. O objetivo é determinar qual o número mínimo de operações de transferência de água entre vasos para que o vaso v_i possua o_i ml ao final. Todos os valores c_i, s_i, o_i são dados como entrada, com $1 \le i \le 3$. Cada transferência consiste em transportar água de uma vaso de origem para um de destino até que ou enche-se o vaso de destino ou esvazia-se o de origem (supõe-se que não há perda de água no processo). Deve-se prever a possibilidade de não haver solução. (Maratona ACM, 2000, América do Sul).

Recursão

22. Elabore os algoritmos que cumpram suas respectivas especificações de forma recursiva. Determine a complexidade de tais algoritmos.
    a. **função** `Ímpar(n: Inteiro): Inteiro`
       // assume $n \geq 1$
       // retorna o n-ésimo número natural que é ímpar
    b. **função** `Pot(b,n: Inteiro): Inteiro`
       // assume $n \geq 0$
       // retorna b^n
    c. **função** `Bin(n, k: Inteiro): Inteiro`
       // assume $n \geq k \geq 0$
       // retorna o binomial (n,k), isto é, $n! / ((n-k)!\, k!)$
    d. **função** `RaizQuad(n: Inteiro, i,s:Inteiro): Inteiro`
       // assume $n \geq 0$ e $i \leq \lfloor\sqrt{(n)}\rfloor < s$
       // retorna $\lfloor\sqrt{(n)}\rfloor$
    e. **função** `Divisores(n: Inteiro, k: Inteiro): Inteiro`
       // assume $0 \leq k \leq n$
       // retorna o número de divisores de n que são menores ou iguais a k