Estruturas de Dados
Aula 14: Recursão

Fontes Bibliográficas

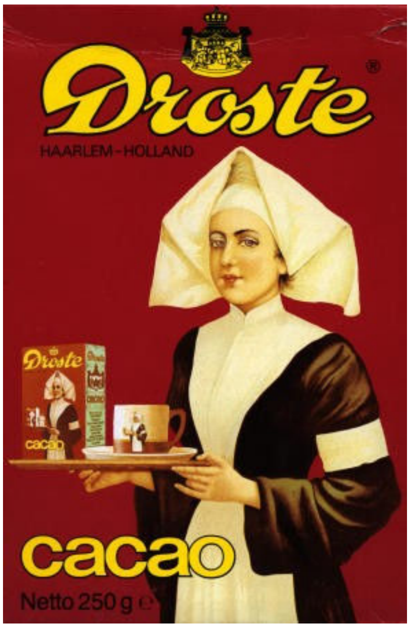
- Livros:
  - Projeto de Algoritmos (Nivio Ziviani): Capítulo 2;
  - Estruturas de Dados e seus Algoritmos (Szwarefiter, et. al): Capítulo 1;
  - Algorithms in C (Sedgewick): Capítulo 5;
- Slides baseados nas aulas de Sedgewick (http://www.cs.princeton.edu/~rs/)

Introdução

- O que é recursão?
  - É um método de programação no qual uma função pode chamar a si mesma
  - O termo é usado de maneira mais geral para descrever o processo de repetição de um objeto de um jeito similar ao que já fora mostrado
- Por que precisamos aprender recursão?
  - Paradigma de programação poderoso
  - Nova maneira de pensar
- Muitas estruturas têm natureza recursiva:
  - Estruturas encadeadas
  - Fatorial, máximo divisor comum
  - Uma pasta que contém outras pastas e arquivos

Introdução (cont.)

Uma forma visual de recursão conhecida como *efeito Droste*



Introdução (cont.)

Máximo Divisor Comum

- mdc (p, q): encontre o maior divisor comum entre p e q;
- Ex.: mdc (4032, 1272) = 24
  - $4032 = 2^6 \times 3^2 \times 7^1$
  - $1272 = 2^3 \times 3^1 \times 53^1$
- Uso de mdc:
  - Simplificação de frações: 1272/4032 = 53/168
  - Importante em mecanismos de criptografia

Máximo Divisor Comum (2)

- Algoritmo de Euclides
- mdc (p, q) = $\begin{cases} p & \text{se } q = 0 \\ \text{mdc}(q, p\%q) & \text{caso contrario} \end{cases}$

- caso base

- passo de redução, converge para o caso base

- mdc (4032, 1272) = mdc (1272, 216)
  mdc (216, 192)
  mdc (192, 24)
  mdc (24, 0)
  24

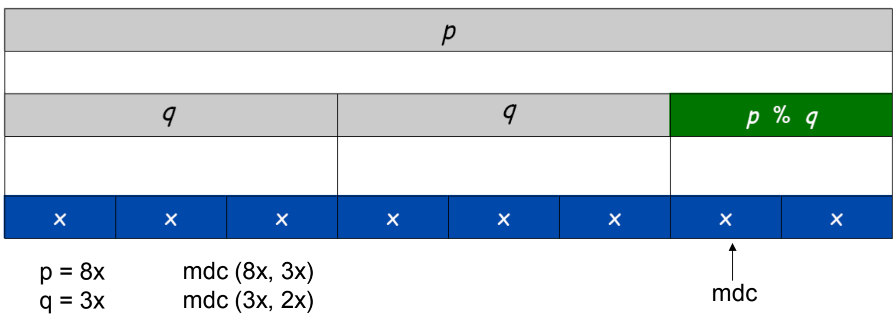
- 4032 / 1272 = 3 x 1272 + 216

Máximo Divisor Comum (3)

- mdc (p, q) = $\begin{cases} p & \text{se } q = 0 \\ \text{mdc}(q, p\%q) & \text{caso contrario} \end{cases}$

- caso base

- passo de redução, converge para o caso base



p = 8x
q = 3x

mdc (8x, 3x)
mdc (3x, 2x)
mdc (2x, x)
mdc (x,0)
mdc (p, q) = x

Máximo Divisor Comum (4)

- mdc (p, q) = $\begin{cases} p & \text{se } q = 0 \\ \text{mdc}(q, p\%q) & \text{caso contrario} \end{cases}$

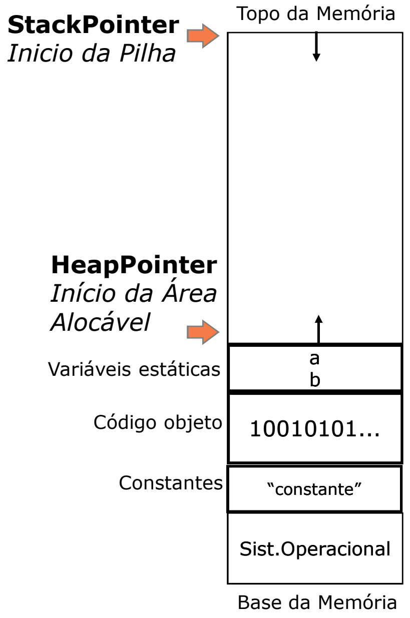
  - caso base

  - passo de redução, converge para o caso base

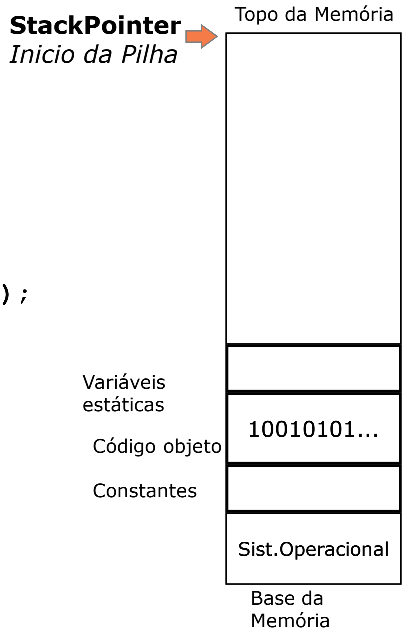
- Implementação em C

```
int mdc (int p, int q)
{
   if (q == 0) return p; //caso base
   else return mdc(q, p % q); //passo de redução
}
```

Memória

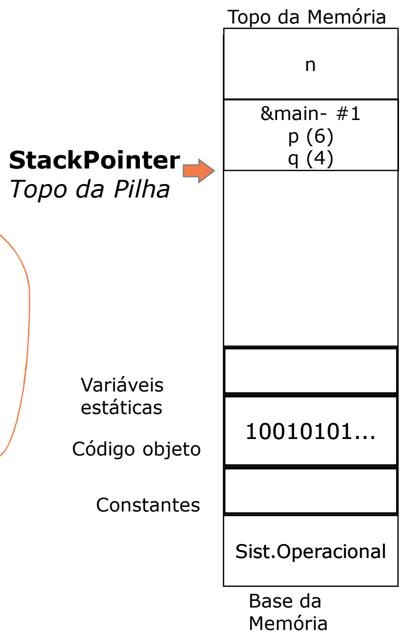


- Programa:




```
int mdc (int p, int q)
{
    if (q == 0) return p;
      //caso base
    else return mdc(q, p % q);
   //passo de redução
}
main ()
{
   int n = mdc(6, 4);
}
```

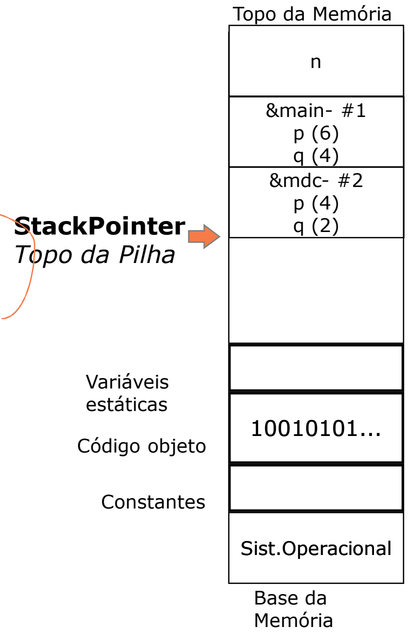
- Programa:




```
int mdc (int p, int q)
{
if (q == 0) return p;
else return mdc(q, p % q);
}
main ()
{
  int n = mdc(6, 4);
}
```

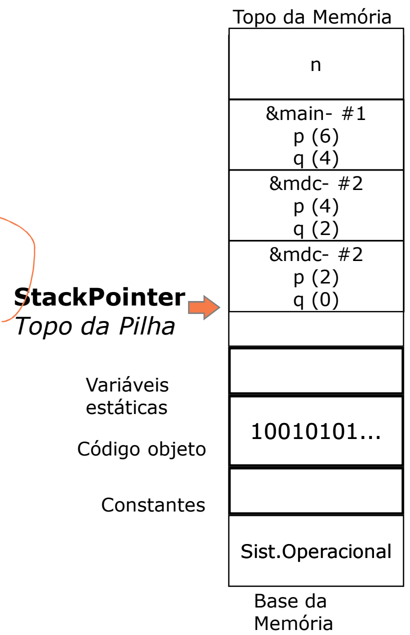
- Programa:

```
int mdc (int p, int q)
{
if (q == 0) return p;
else return mdc(q, p % q);
}
main ()
{
   int n = mdc(6, 4);
}
```



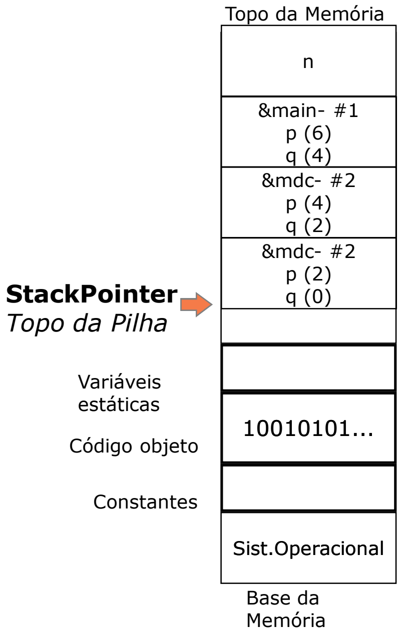
- Programa:

```
int mdc (int p, int q)
{
if (q == 0) return p;
else return mdc(q, p % q);
}
main ()
{
   int n = mdc(6, 4);
}
```



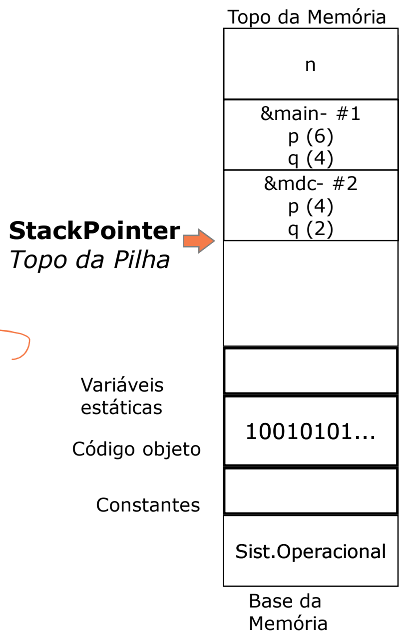
- Programa:

```
int mdc (int p, int q)
{
if (q == 0) return p;
else return mdc(q, p % q);
}
main ()
{
   int n = mdc(6, 4);
}
```



- Programa:

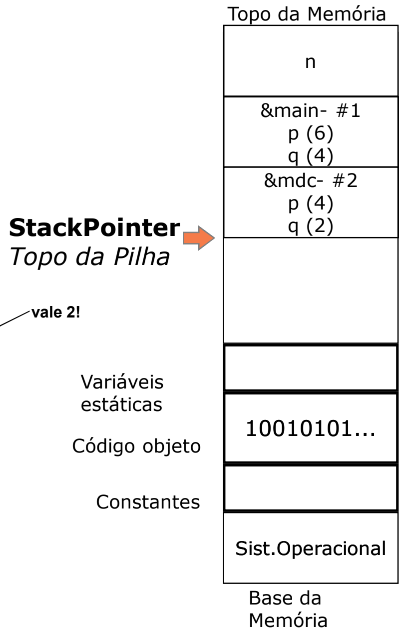
```
int mdc (int p, int q)
{
if (q == 0) return p;
else return mdc(q, p % q);
}
main ()
{
   int n = mdc(6, 4);
}
```



- Programa:

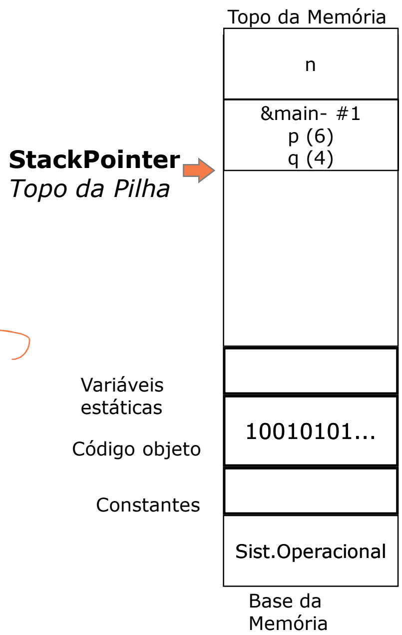
```
int mdc (int p, int q)
{
if (q == 0) return p;
else return mdc(q, p % q);
}
main ()
{
   int n = mdc(6, 4);
}
```

vale 2!



- Programa:

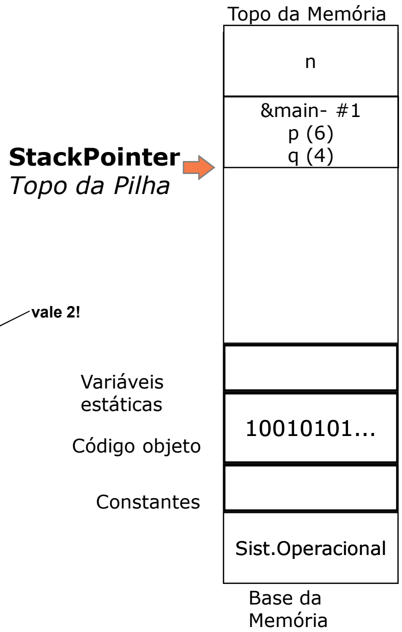
```
int mdc (int p, int q)
{
if (q == 0) return p;
else return mdc(q, p % q);
}
main ()
{
   int n = mdc(6, 4);
}
```



- Programa:

```
int mdc (int p, int q)
{
if (q == 0) return p;
else return mdc(q, p % q);
}
main ()
{
   int n = mdc(6, 4);
}
```

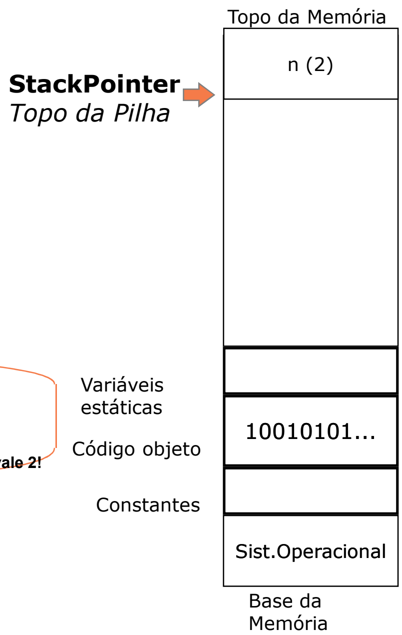
vale 2!



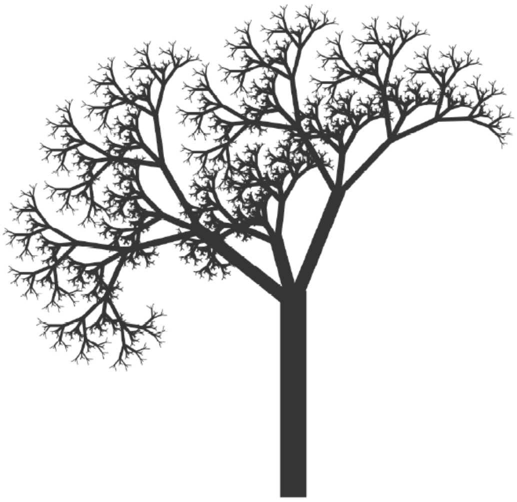
- Programa:

```
int mdc (int p, int q)
{
if (q == 0) return p;
else return mdc(q, p % q);
}
main ()
{
   int n = mdc(6, 4);
}
```

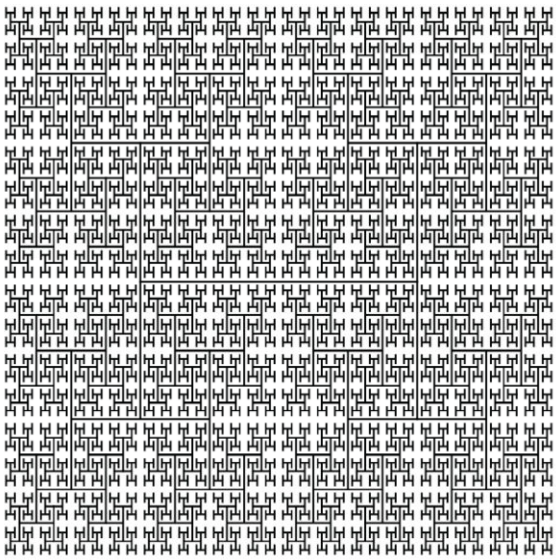
vale 2!



Gráficos Recursivos

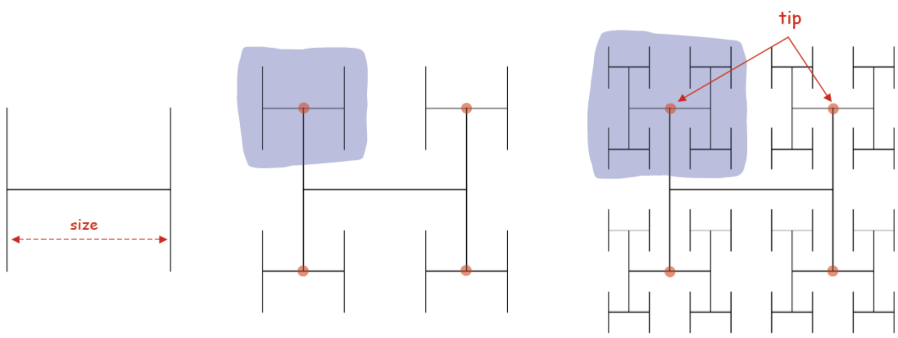


Árvore H



Árvore H

- Árvore-H de ordem *n*
  - Desenha uma letra H
  - Recursivamente desenha 4 árvores-H da ordem de n-1 (e metade do tamanho), cada árvore conectada em um “topo” (*tip*).



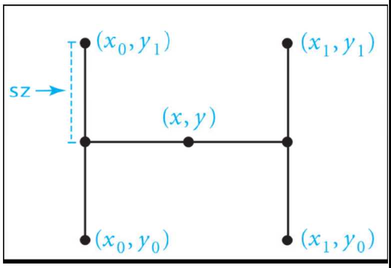
ordem 1 ordem 2 ordem 3

Implementação Recursiva da Árvore H (em C)

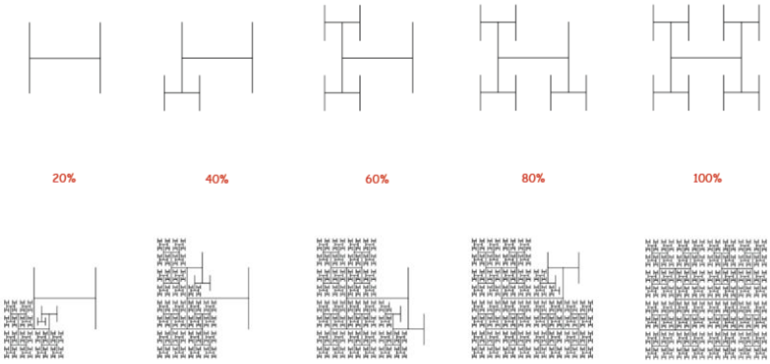
```
void draw(int n, double tam, double x, double y) {
   if (n == 0) return; //condicao de parada
   double x0 = x - tam/2; double x1 = x + tam/2;
   double y0 = y - tam/2; double y1 = y + tam/2;
   DesenhaLinha(x0, y, x1, y);
   DesenhaLinha(x0, y0, x0, y1);
   DesenhaLinha(x1, y0, x1, y1);
   draw(n-1, tam/2, x0, y0);
   draw(n-1, tam/2, x0, y1);
   draw(n-1, tam/2, x1, y0);
   draw(n-1, tam/2, x1, y1);
}
```

desenha o H centralizado em (x, y)

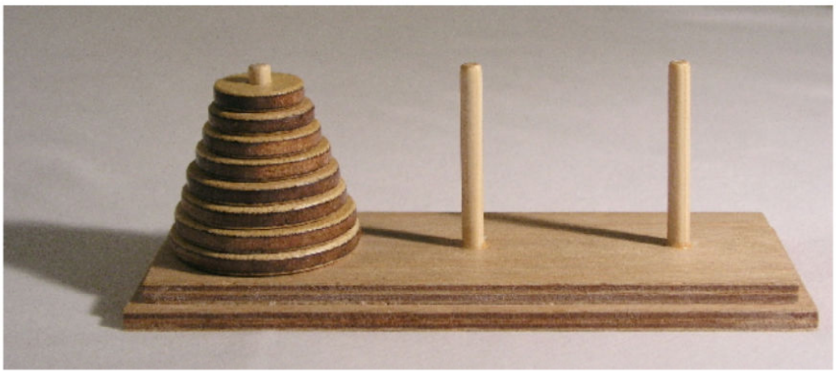
recursivamente desenha 4 Hs com a metade do tamanho



Animação Árvore H



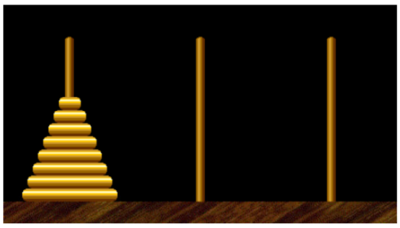
Torres de Hanoi



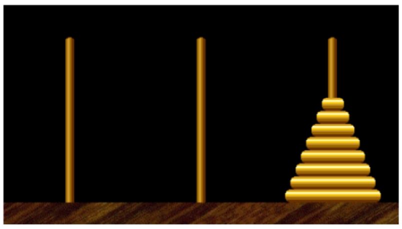
http://en.wikipedia.org/wiki/Image:Hanoiklein.jpg

Objetivo

- Mover os discos do pino mais a esquerda para o pino da direita
  - Somente um disco por vez pode ser movido;
  - Um disco pode ser colocado num pino vazio ou sobre um disco de tamanho maior;



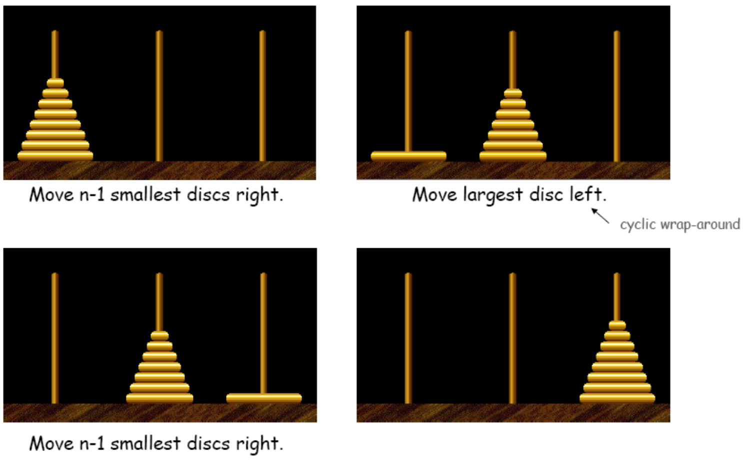
Início



Final

- Torres de Hanoi: animação

Torres de Hanoi: Solução Recursiva



Lenda das Torres de Hanoi

- Mundo vai acabar quando um grupo de monges conseguirem mover 64 discos de ouro em 3 pinos de diamante.
- Algoritmos de computação irão ajudar a resolver o problema?

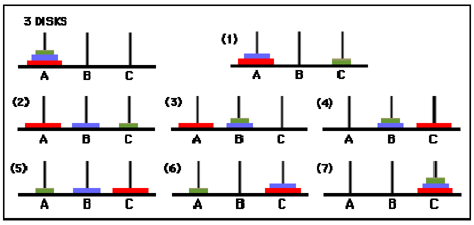
Torres de Hanoi: Implementação Recursiva

```
void moves (int N, int left)
{
  if (N == 0) return; // se não houver discos, retorna
  moves(N-1, !left);
  if (left)
      printf("%d left", N);
  else
      printf("%d right", N);
  moves (N-1, !left);
}
```

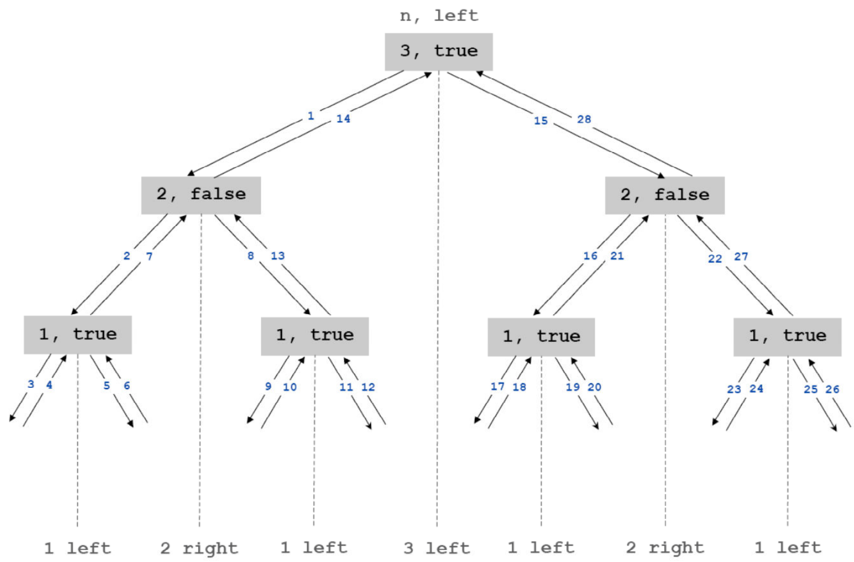
Torres de Hanoi: Implementação Recursiva

(para 3 discos)

moves (3, left)
    moves (2, right)
        moves (1, left)
            "1 left"
        "2 right"
        moves (1, left)
            "1 left"
    "3 left"
    moves (2, right)
        moves (1, left)
            "1 left"
        "2 right"
        moves (1, left)
            "1 left"



Torres de Hanoi: árvore de recursão



Torres de Hanoi: Propriedades da solução

- Leva $2^n - 1$ "moves" para resolver o problema com n discos;
- O algoritmo revela um fato:
  - São necessários 585 milhões de anos para n=64 (considerando que cada movimento de disco leve 1 segundo, os monges não cometam erros e que os monges saibam exatamente para onde movimentar o disco, sem pestanejar)
- Outro fato: qualquer solução possível para as torres de Hanoi levará no mínimo esse tempo!

Dividir para Conquistar

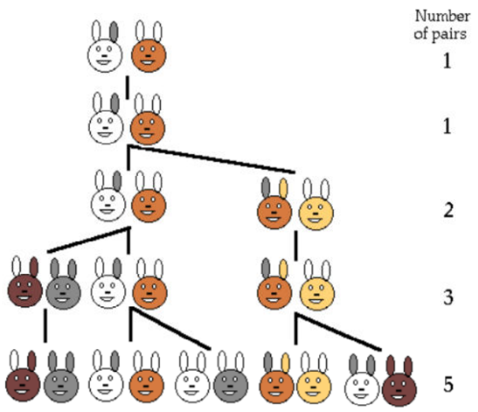
- Consiste em dividir o problema em problemas menores
- Problemas menores são resolvidos recursivamente usando o mesmo método
- Resultados são combinados para resolver problema original
- Vários algoritmos são resolvidos com essa técnica (e.x., quicksort, mergesort)

Pontos Negativos da Recursão

- Considere a sequência de Fibonacci: 0,1,1,2,3,5,8,13,21,34...

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

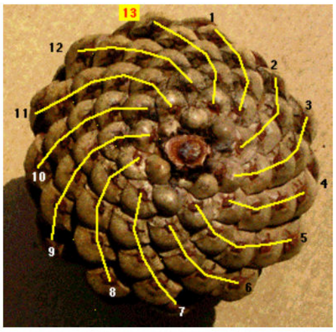
Sequência de Fibonacci



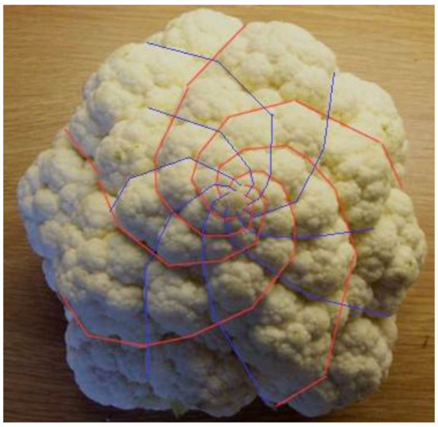
Fibonacci rabbits

L. P. Fibonacci
(1170 - 1250)

Sequência de Fibonacci e a Natureza

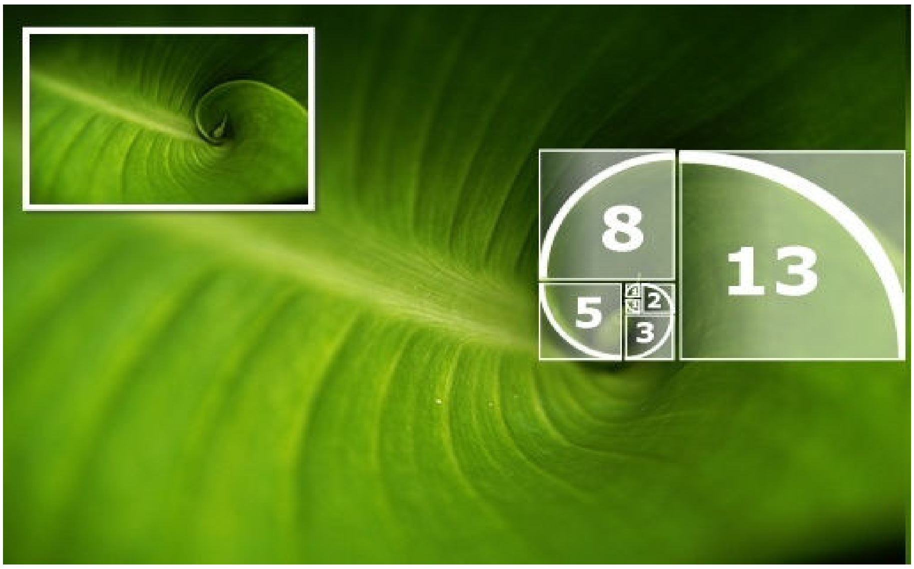


pinecone



cauliflower

Sequência de Fibonacci e a Natureza



Solução Recursiva?

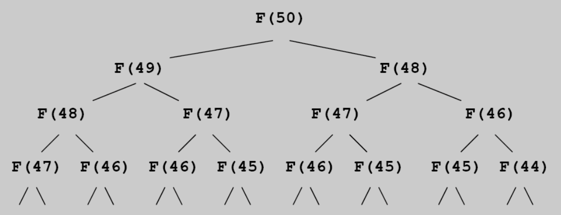
$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

```
long F(int n) {
   if (n == 0) return 0;
   if (n == 1) return 1;
   return F(n-1) + F(n-2);
}
```

-> Código muito ineficiente!

-> Leva muito tempo para computar F(50)!

Problema com Recursão



recursion tree for naïve Fibonacci function

F(50) é chamado uma vez

F(49) é chamado uma vez

F(48) é chamado 2 vezes

F(47) é chamado 3 vezes

F(46) é chamado 5 vezes

F(45) é chamado 8 vezes

...

F(1) é chamado 12,586,269,025 vezes

- **Pode facilmente levar a soluções incrivelmente ineficientes!**

Binet's formula.

$$F(n) = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}} = \left\lfloor \phi^n / \sqrt{5} \right\rceil$$

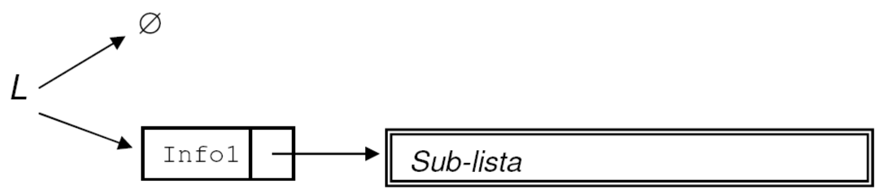
ϕ = golden ratio $\approx$ 1.618

Resumindo

- Como escrever programas recursivos simples?
  - Condição de parada, passo da recursão
- Dividir para conquistar
  - Técnica elegante de resolver problemas (não somente recursivos)

Implementação Recursiva de Listas

- Considere a lista sem sentinela e sem cabeçalho
- Definição recursiva:
  - Uma lista é:
    - Uma lista vazia; ou
    - Um elemento seguido de uma (sub)-lista



Implementação Recursiva de Listas

- Exemplo – função imprime
  - Se a lista for vazia, não imprime nada
  - Caso contrário:
    - Imprime o conteúdo da primeira célula (l->Item ou l->Item.*campo*)
    - Imprime a sub-lista dada por l->Prox, chamando a função recursivamente

Implementação Recursiva de Listas

```
/* Função imprime recursiva */
void lst_imprime_rec (TipoLista* l)
{
  if ( !lst_vazia(l)) {
     /* imprime primeiro elemento: lista de
  inteiros */
     printf("Item: %d\n",l->Item);
     /* imprime sub-lista */
     lst_imprime_rec(l->Prox);
  }
}
```

Implementação Recursiva de Listas

- Exemplo – função retira
  - retire o elemento, se ele for o primeiro da lista (ou da sub-lista)
  - caso contrário, chame a função recursivamente para retirar o elemento da sub-lista

Implementação Recursiva de Listas

```
/* Função retira recursiva */
TipoLista* lst_retira_rec (TipoLista* l, int v){
   if (!lst_vazia(l)) {
        /* verifica se elemento a ser retirado é o primeiro */
        if (l->Item == v) {
                TipoLista* t = l; /* temporário para liberar */
                l = l->Prox;
                free(t);
        }
        else {
                /* retira de sub-lista */
                l->Prox = lst_retira_rec(l->Prox,v);
        }
   }return l;
}
```

é necessário re-atribuir o valor de l->prox na chamada recursiva, já que a função pode alterar o valor da sub-lista

Implementação Recursiva de Listas

- Exemplo – função que testa igualdade entre duas listas

```
int lst_igual (TipoLista* l1, TipoLista* l2)
```

  - se as duas listas dadas são vazias, são iguais
  - se não forem ambas vazias, mas uma delas é vazia, são diferentes
  - se ambas não forem vazias, teste:
    - se informações associadas aos primeiros nós são iguais
    - se as sub-listas são iguais

Implementação Recursiva de Listas

```
int lst_igual (TipoLista* l1, TipoLista*
  l2){
  if (l1 == NULL && l2 == NULL)
     return 1;
  else if (l1 == NULL || l2 == NULL)
     return 0;
  else
     return l1->Item == l2->Item &&
     lst_igual(l1->Prox, l2->Prox);
}
```