MC-202 — Aula 5
Revisão de recursão e simulação de recursão

Lehilton Pedrosa

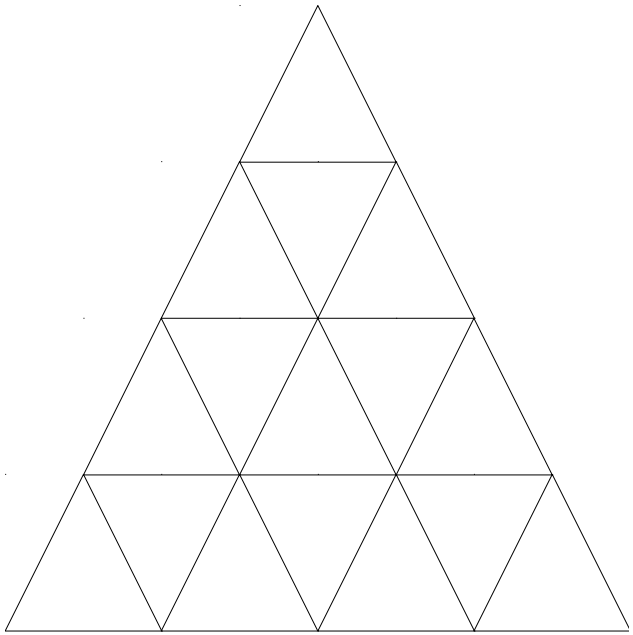
Instituto de Computação – Unicamp

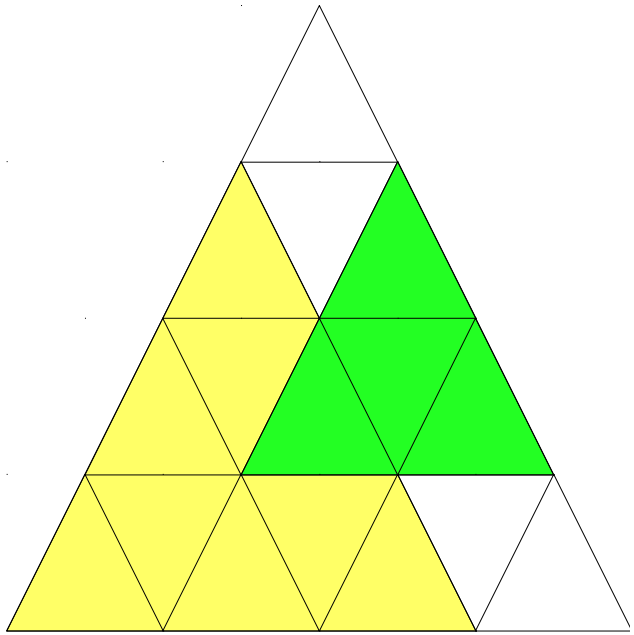
Segundo Semestre de 2015

Roteiro

Introdução

Considere o problema a seguir.

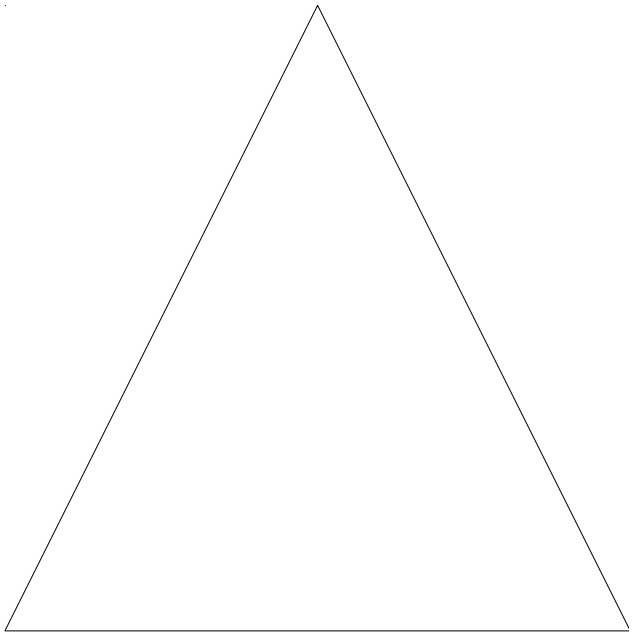




Problema

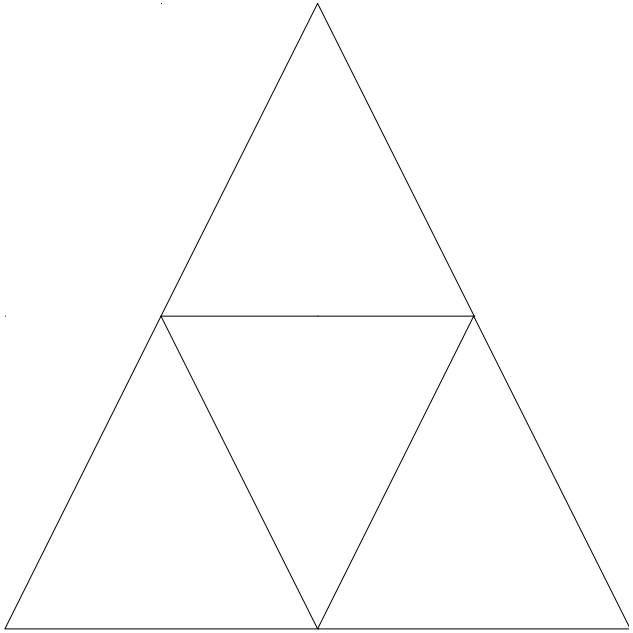
Quantos triângulos de pé (ver exemplos coloridos) podemos encontrar em uma grade de triângulos com altura n?

Triângulos



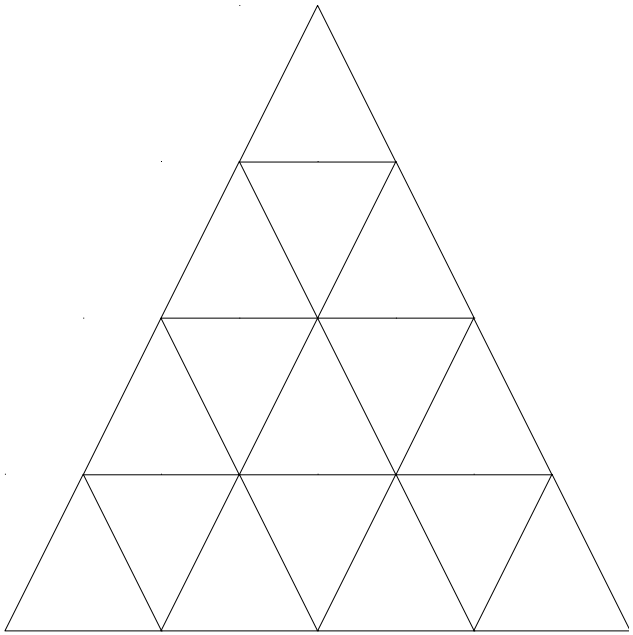
Para uma grade de altura $n = 1$, temos $t(1) = 1$ triângulo.

Triângulos



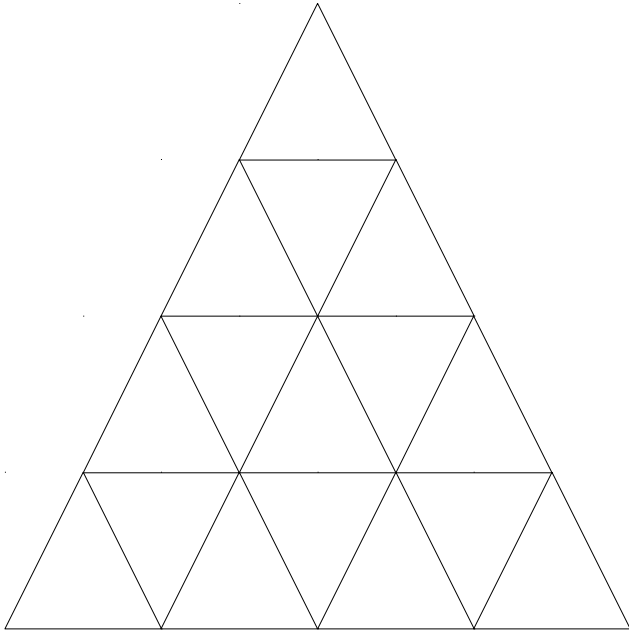
Para uma grade de altura $n = 2$, temos $t(2) = 4$ triângulos:
2 com a ponta superior e outros 2 novos.

Triângulos



E para $n = 4$?

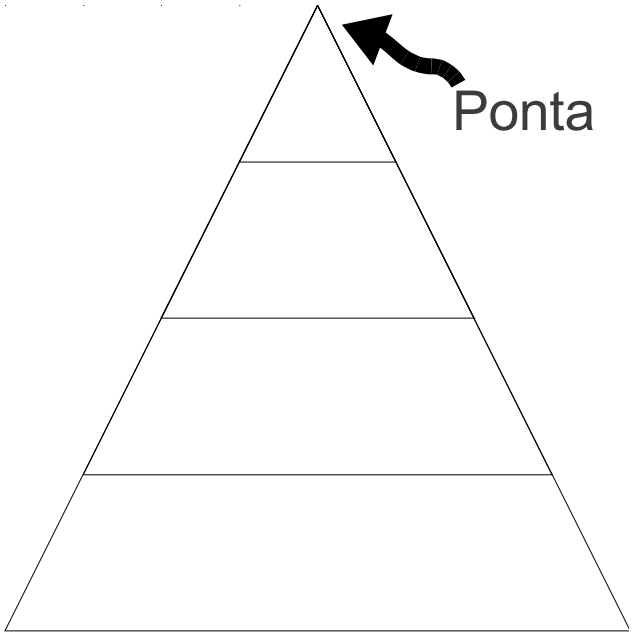
Triângulos



E para $n = 4$?

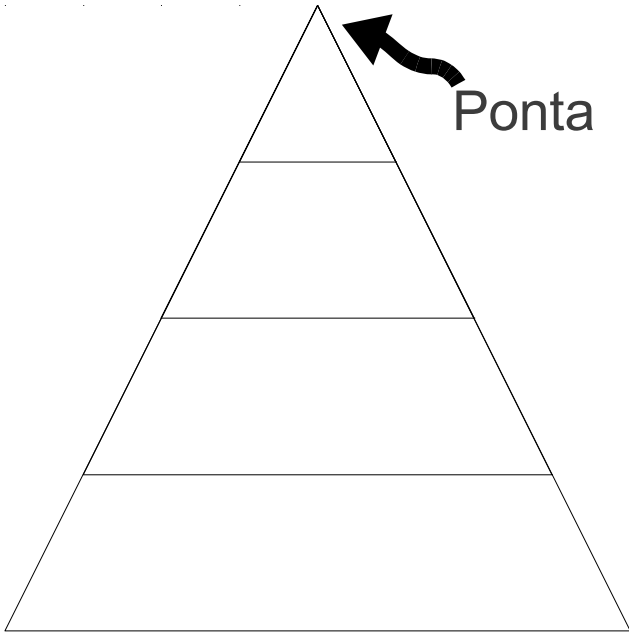
Podemos encontrar algum padrão?

Triângulos



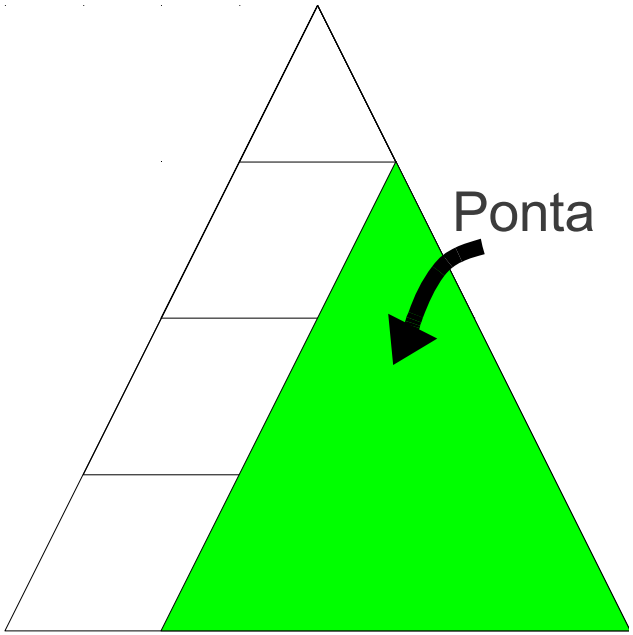
É fácil contar apenas os triângulos com a ponta no triângulo superior: 4.

Triângulos



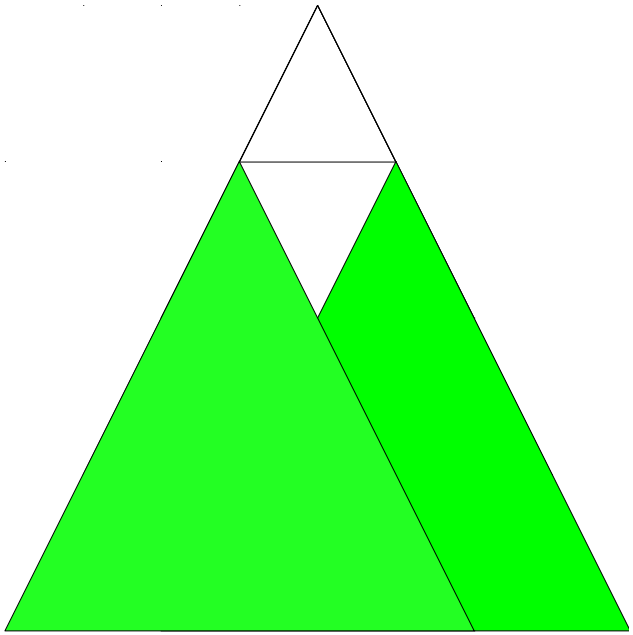
É fácil contar apenas os triângulos com a ponta no triângulo superior: 4. Além desses, quantos faltam?

Triângulos



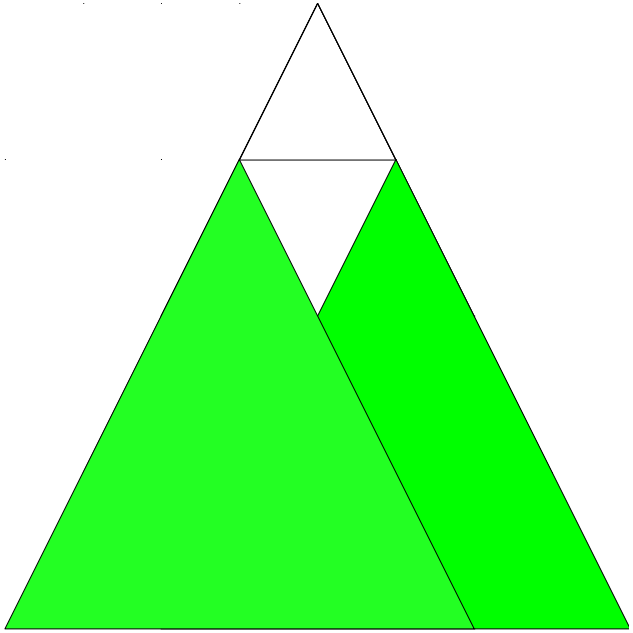
Faltam os triângulos do lado direito

Triângulos



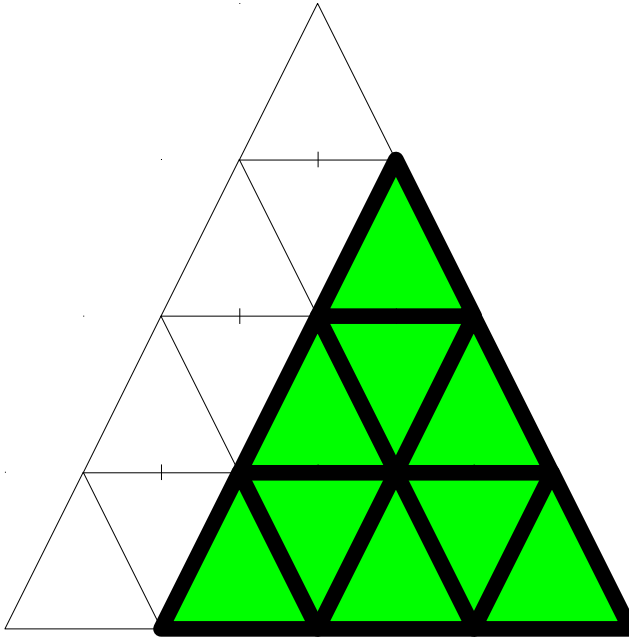
Faltam os triângulos do lado direito
e os triângulos do lado esquerdo.

Triângulos



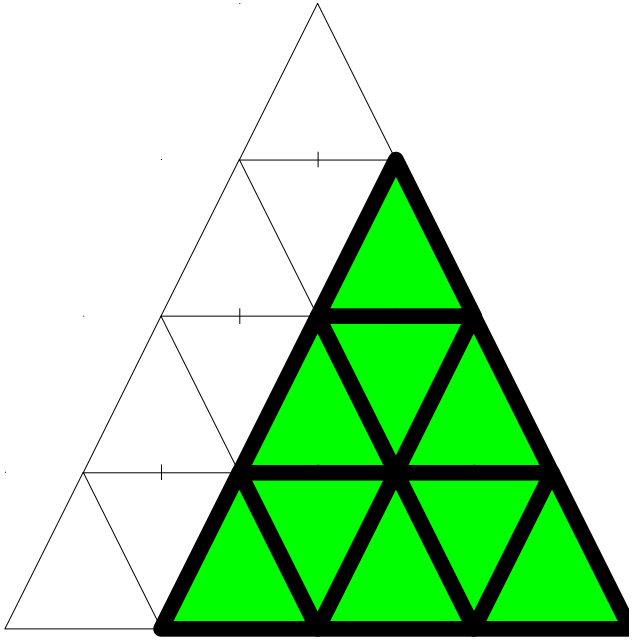
Faltam os triângulos do lado direito
e os triângulos do lado esquerdo.
Mas como calcular o número de triângulos de um certo lado?

Triângulos



Caímos no mesmo problema anterior...

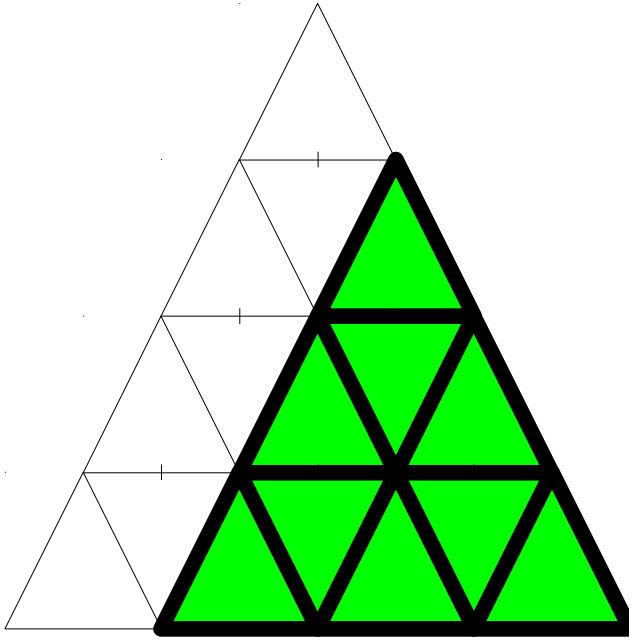
Triângulos



Caímos no mesmo problema anterior...

...mas agora para $n = 3$.

Triângulos

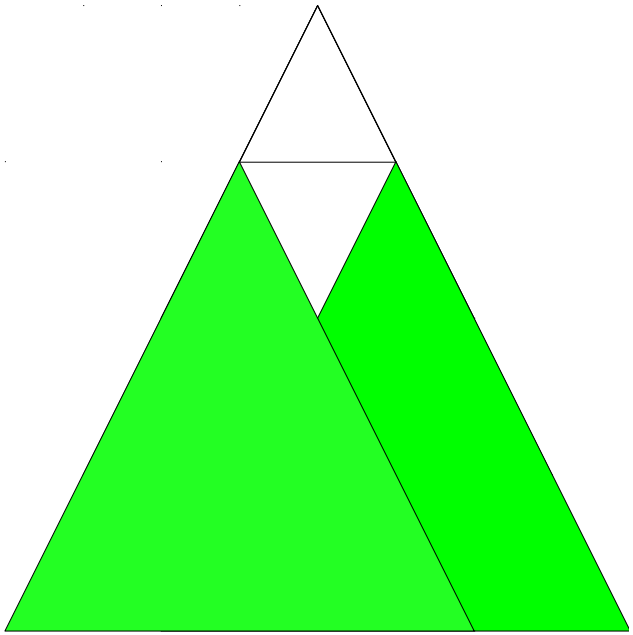


Caímos no mesmo problema anterior...

...mas agora para $n = 3$.

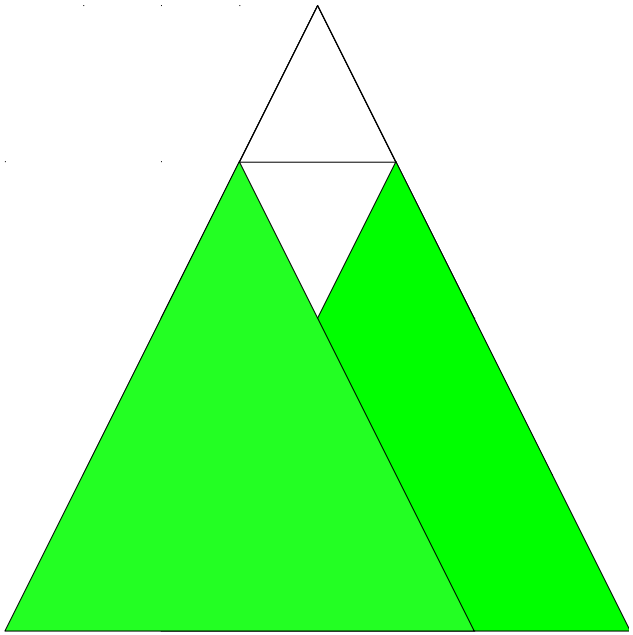
Podemos repetir o mesmo procedimento, para $n = 2$ e $n = 1$.

Triângulos



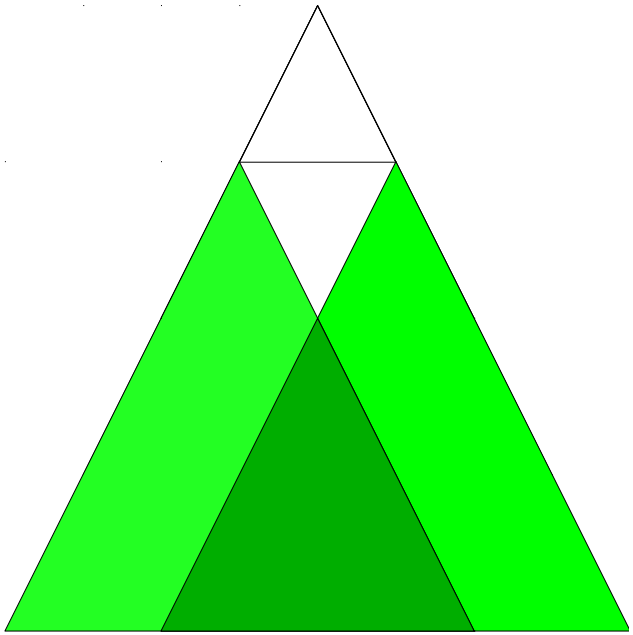
Suponha que já sabemos:
$t(1) = 1, t(2) = 4, t(3) = 10.$

Triângulos



Suponha que já sabemos:
$t(1) = 1, t(2) = 4, t(3) = 10.$
Como podemos calcular $t(4)$?

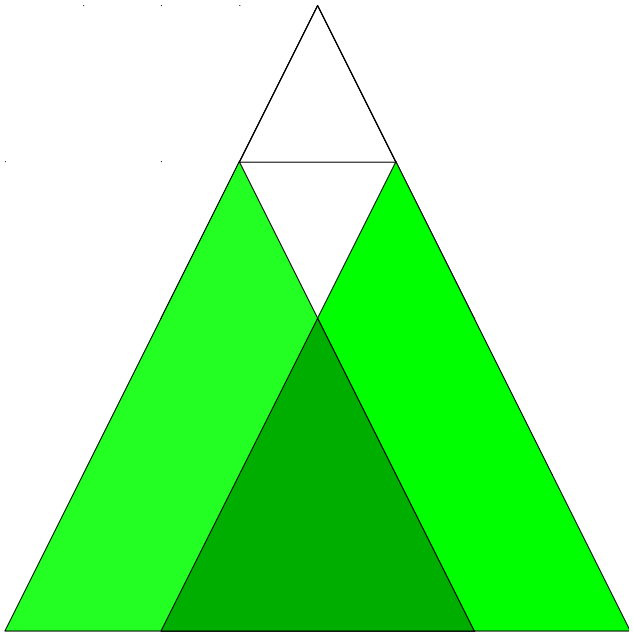
Triângulos



Suponha que já sabemos:
$t(1) = 1, t(2) = 4, t(3) = 10$.
Como podemos calcular $t(4)$?

Somamos os triângulos superiores
aos os triângulos da esquerda e da direita e subtraímos a interseção.

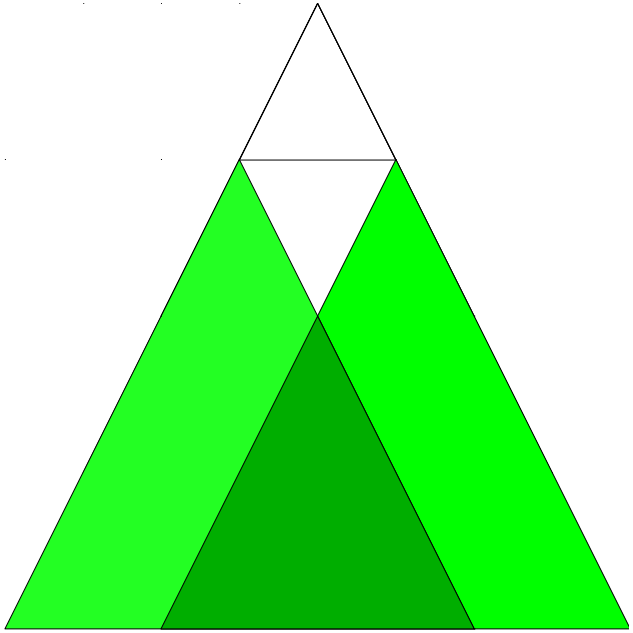
Triângulos



Suponha que já sabemos:
$t(1) = 1, t(2) = 4, t(3) = 10.$
Como podemos calcular $t(4)$?

Somamos os triângulos superiores
aos os triângulos da esquerda e da direita e subtraímos a interseção.
$t(4) = 4 + t(3) + t(3) - t(2) = 20.$

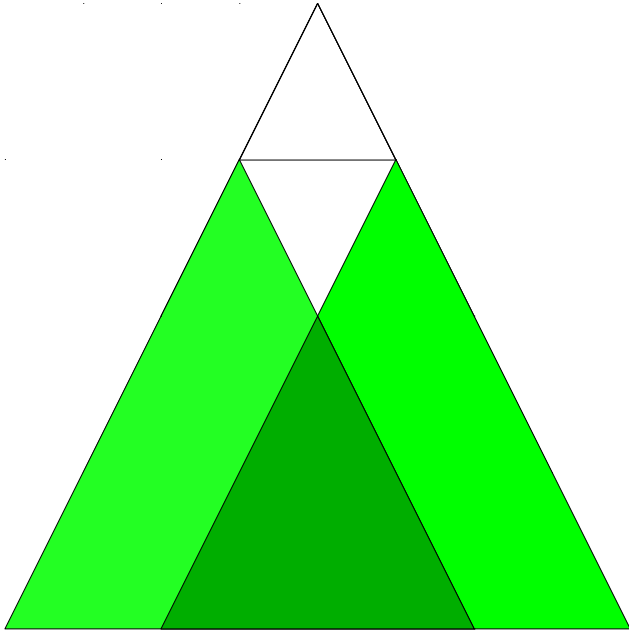
Triângulos



E para calcular o número de triângulos $t(n)$ para um n qualquer?

- Se $n = 0$, então $t(n) = 0$.

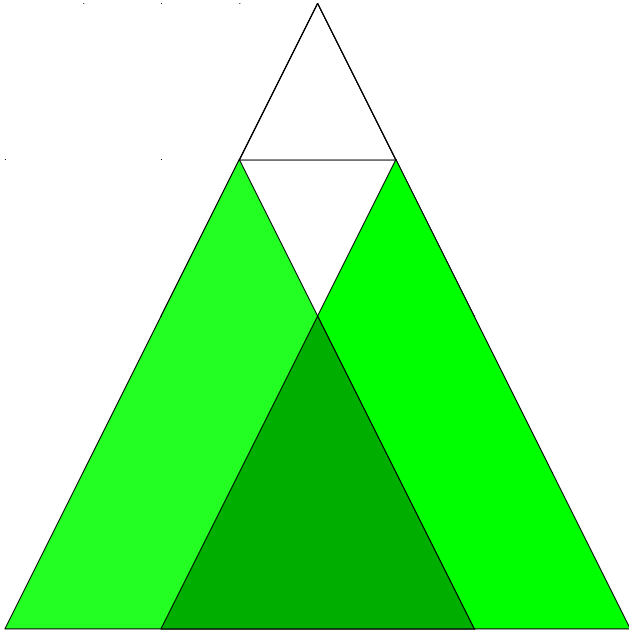
Triângulos



E para calcular o número de triângulos $t(n)$ para um n qualquer?

- Se $n = 0$, então $t(n) = 0$.
- Se $n = 1$, então $t(n) = 1$.

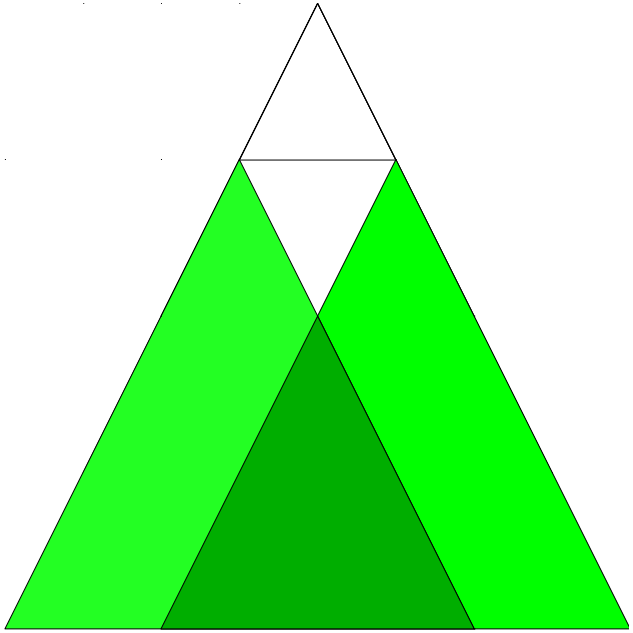
Triângulos



E para calcular o número de triângulos $t(n)$ para um n qualquer?

- Se $n = 0$, então $t(n) = 0$.
- Se $n = 1$, então $t(n) = 1$.
- Do contrário, $t(n) =$

Triângulos



E para calcular o número de triângulos $t(n)$ para um n qualquer?

- Se $n = 0$, então $t(n) = 0$.
- Se $n = 1$, então $t(n) = 1$.
- Do contrário, $t(n) = n + 2 \cdot t(n-1) - t(n-2)$.

Triângulos - programando

Escreva uma função que calcule o número de triângulos em pé de uma grade de tamanho n.

Triângulos - programando

Escreva uma função que calcule o número de triângulos em pé de uma grade de tamanho n.

```
int triangulos(int n) {
  if (n == 0)
      return 0;
  else if (n == 1)
      return 1;
  else
      return n + 2*triangulos(n-1) - triangulos(n-2);
}
```

Triângulos - programando

Escreva uma função que calcule o número de triângulos em pé de uma grade de tamanho n.

```
int triangulos(int n) {
  if (n == 0)
      return 0;
  else if (n == 1)
      return 1;
  else
      return n + 2*triangulos(n-1) - triangulos(n-2);
}
```

Observe que a função `triangulos` chama a própria função `triangulos`. **Isso é chamado de recursão**.

Recursão

Recursão

A ideia é que um problema pode ser resolvido da seguinte maneira:

Recursão

Recursão

A ideia é que um problema pode ser resolvido da seguinte maneira:

- Primeiramente, definimos as soluções para casos básicos.

Recursão

Recursão

A ideia é que um problema pode ser resolvido da seguinte maneira:

- Primeiramente, definimos as soluções para casos básicos.
- Em seguida, tentamos reduzir o problema para instâncias menores.

Recursão

Recursão

A ideia é que um problema pode ser resolvido da seguinte maneira:

- Primeiramente, definimos as soluções para casos básicos.
- Em seguida, tentamos reduzir o problema para instâncias menores.
- Finalmente, combinamos o resultado das instâncias menores para obter um resultado do problema original.

Genericamente

Recursão: considerando 2 casos

1. **Caso base:**
   - resolve instâncias pequenas diretamente

Genericamente

Recursão: considerando 2 casos

1. **Caso base:**
   - resolve instâncias pequenas diretamente
2. **Caso geral:**
   - reduz o problema para instâncias menores do mesmo problema
   - chama a função recursivamente

Genericamente

Recursão: considerando 2 casos

1. **Caso base:**
   - resolve instâncias pequenas diretamente
2. **Caso geral:**
   - reduz o problema para instâncias menores do mesmo problema
   - chama a função recursivamente

Exemplo: fatorial

```
1.  int fat(int n) {
2.      int ret;
3.      if (n == 0) { // caso base
4.          return 1;
5.      } else { // caso geral
6.          ret = fat(n-1); // instância menor
7.          return n * ret;
8.      }
9.  }
```

Comparando recursão e algoritmos iterativos

Características da recursão

Normalmente algoritmos recursivos são:

- claros
- mais simples de entender
- menores e mais fáceis de programar

Comparando recursão e algoritmos iterativos

Características da recursão

Normalmente algoritmos recursivos são:

- claros
- mais simples de entender
- menores e mais fáceis de programar

Mas algumas vezes podem ser

- muito ineficientes

Comparando recursão e algoritmos iterativos

Características da recursão

Normalmente algoritmos recursivos são:

- claros
- mais simples de entender
- menores e mais fáceis de programar

Mas algumas vezes podem ser

- muito ineficientes
  (quando comparados a algoritmos iterativos para o mesmo problema)

Comparando recursão e algoritmos iterativos

Características da recursão

Normalmente algoritmos recursivos são:

- claros
- mais simples de entender
- menores e mais fáceis de programar

Mas algumas vezes podem ser

- muito ineficientes
  (quando comparados a algoritmos iterativos para o mesmo problema)

Estratégia ideal

1. encontrar algoritmo recursivo para o problema
2. reescrevê-lo como um algoritmo iterativo

Responda: Isso sempre é possível? Quando for possível, sempre melhora a eficiência do algoritmo?

Fibonacci: recursivo *vs.* iterativo

Sequência de Fibonacci

1, 1, 2, 3, 5, 8, 13, ...

Fibonacci: recursivo *vs.* iterativo

Sequência de Fibonacci

1, 1, 2, 3, 5, 8, 13, ...

Recursivo

```
int fib(int n) {
```

Fibonacci: recursivo *vs.* iterativo

Sequência de Fibonacci

1, 1, 2, 3, 5, 8, 13, ...

Recursivo

```
int fib(int n) {
    if (n == 1)
        return 1;
```

Fibonacci: recursivo *vs.* iterativo

Sequência de Fibonacci

1, 1, 2, 3, 5, 8, 13, ...

Recursivo

```
int fib(int n) {
    if (n == 1)
        return 1;
    else if (n == 2)
        return 1;
```

Fibonacci: recursivo *vs.* iterativo

Sequência de Fibonacci

1, 1, 2, 3, 5, 8, 13, ...

Recursivo

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

Fibonacci: recursivo *vs.* iterativo

Sequência de Fibonacci

1, 1, 2, 3, 5, 8, 13, ...

Recursivo

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

Iterativo

```
int fib_iterativo(int n) {
```

Fibonacci: recursivo *vs.* iterativo

Sequência de Fibonacci

1, 1, 2, 3, 5, 8, 13, ...

Recursivo

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

Iterativo

```
int fib_iterativo(int n) {
    int a, b, c, i;
    a = b = 1;
```

Fibonacci: recursivo *vs.* iterativo

Sequência de Fibonacci

1, 1, 2, 3, 5, 8, 13, ...

Recursivo

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

Iterativo

```
int fib_iterativo(int n) {
    int a, b, c, i;
    a = b = 1;
    for (i = 3; i < n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
```

Fibonacci: recursivo *vs.* iterativo

Sequência de Fibonacci

1, 1, 2, 3, 5, 8, 13, ...

Recursivo

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

Iterativo

```
int fib_iterativo(int n) {
    int a, b, c, i;
    a = b = 1;
    for (i = 3; i < n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

Fibonacci: recursivo *vs.* iterativo

Sequência de Fibonacci

1, 1, 2, 3, 5, 8, 13, ...

Recursivo

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

Iterativo

```
int fib_iterativo(int n) {
    int a, b, c, i;
    a = b = 1;
    for (i = 3; i < n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

Eficiência:

- recursivo: $O(1.6^n)$
- iterativo: $O(n)$

Pilhas e recursão

Pergunta: Afinal, qual a relação entre *pilhas* e *recursão*?

Pilhas e recursão

Pergunta: Afinal, qual a relação entre *pilhas* e *recursão*?

Exemplo: fatorial

```
1.  int fat(int n) {
2.      int ret;
3.      if (n == 0) { // caso base
4.          return 1;
5.      } else { // caso geral
6.          ret = fat(n-1); // instância menor
7.          return n * ret;
8.      }
9.  }
```

Pilhas e recursão

Pergunta: Afinal, qual a relação entre *pilhas* e *recursão*?

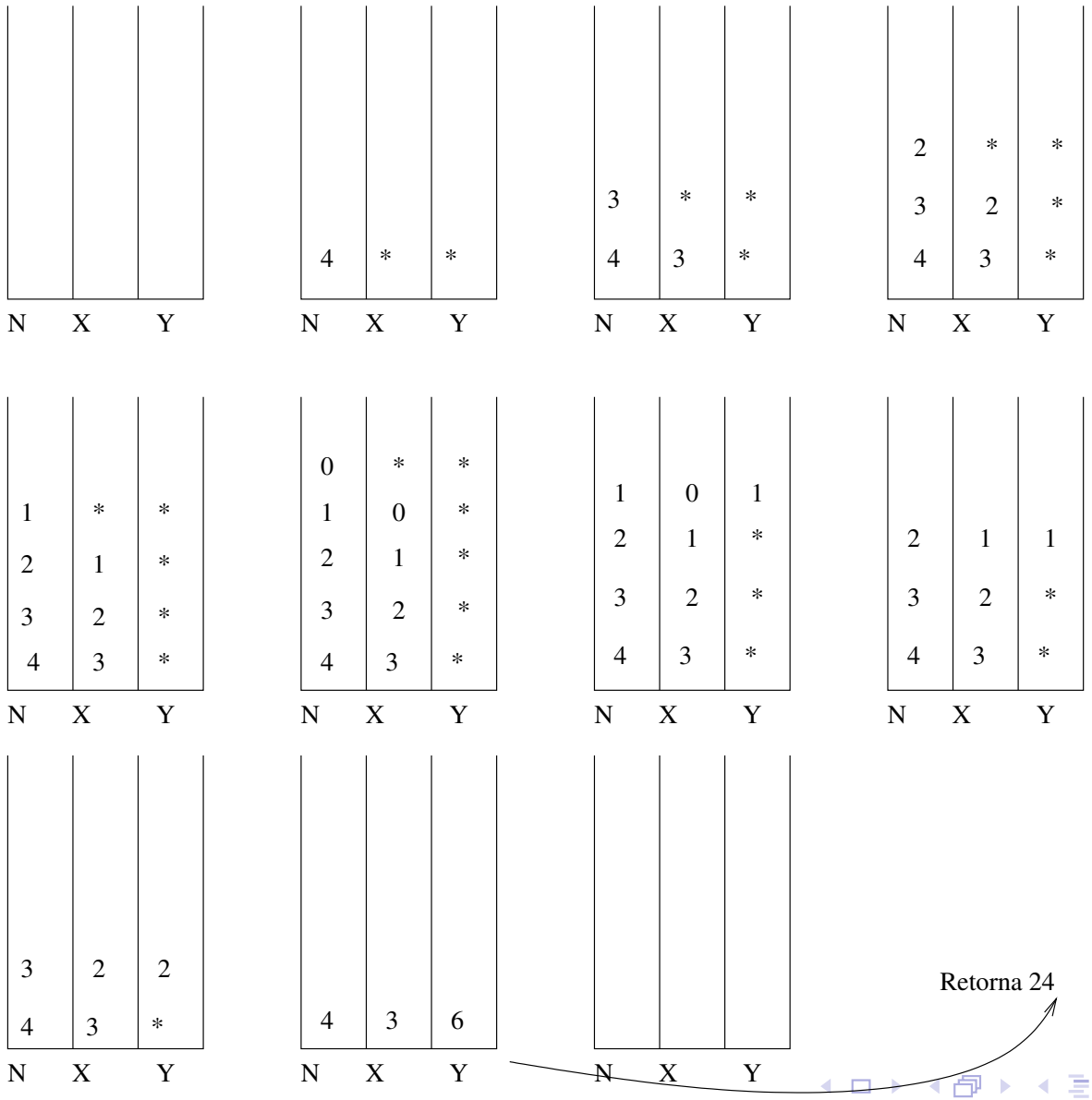
Exemplo: fatorial

```
1.  int fat(int n) {
2.      int ret;
3.      if (n == 0) { // caso base
4.          return 1;
5.      } else { // caso geral
6.          ret = fat(n-1); // instância menor
7.          return n * ret;
8.      }
9.  }
```

Vamos tentar descobrir simulando uma chamada: `fat(3)`

Chamadas - Fatorial

Estado da "pilha" de chamadas para `fatorial(4)`:



Pilhas e recursão (continuando)

O que fizemos:

Empilhamos Quando chamamos `fat(3)`, alocamos espaço para o par de variáveis locais `n` e `ret`. E também alocamos espaço para chamada `fat(2)`, `fat(1)` e `fat(0)`.

Pilhas e recursão (continuando)

O que fizemos:

Empilhamos Quando chamamos `fat(3)`, alocamos espaço para o par de variáveis locais `n` e `ret`. E também alocamos espaço para chamada `fat(2)`, `fat(1)` e `fat(0)`.

Desempilhamos Quando a chamada `fat(0)` retorna, apagamos o espaço do par de variáveis. E também apagamos para chamada `fat(1)`, `fat(2)` e `fat(3)`.

Pilhas e recursão (continuando)

O que fizemos:

Empilhamos Quando chamamos `fat(3)`, alocamos espaço para o par de variáveis locais `n` e `ret`. E também alocamos espaço para chamada `fat(2)`, `fat(1)` e `fat(0)`.

Desempilhamos Quando a chamada `fat(0)` retorna, apagamos o espaço do par de variáveis. E também apagamos para chamada `fat(1)`, `fat(2)` e `fat(3)`.

O conjunto de pares de variáveis formaram uma pilha.

Pilhas e recursão (continuando)

O que fizemos:

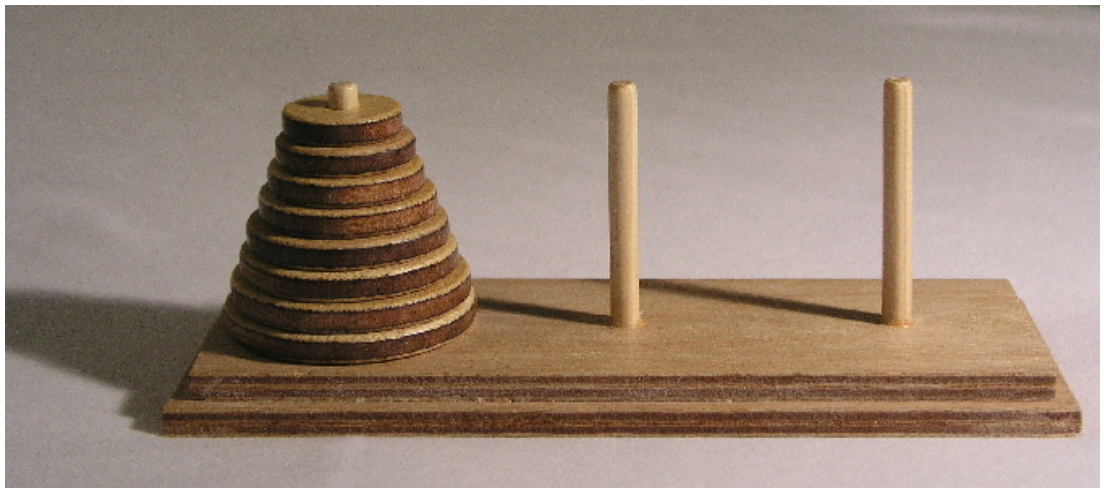
Empilhamos Quando chamamos `fat(3)`, alocamos espaço para o par de variáveis locais `n` e `ret`. E também alocamos espaço para chamada `fat(2)`, `fat(1)` e `fat(0)`.

Desempilhamos Quando a chamada `fat(0)` retorna, apagamos o espaço do par de variáveis. E também apagamos para chamada `fat(1)`, `fat(2)` e `fat(3)`.

O conjunto de pares de variáveis formaram uma pilha.

Resposta da pergunta: recursão pode ser simulada usando uma pilha de suas variáveis locais.

Um exemplo mais complexo: Torres de Hanói

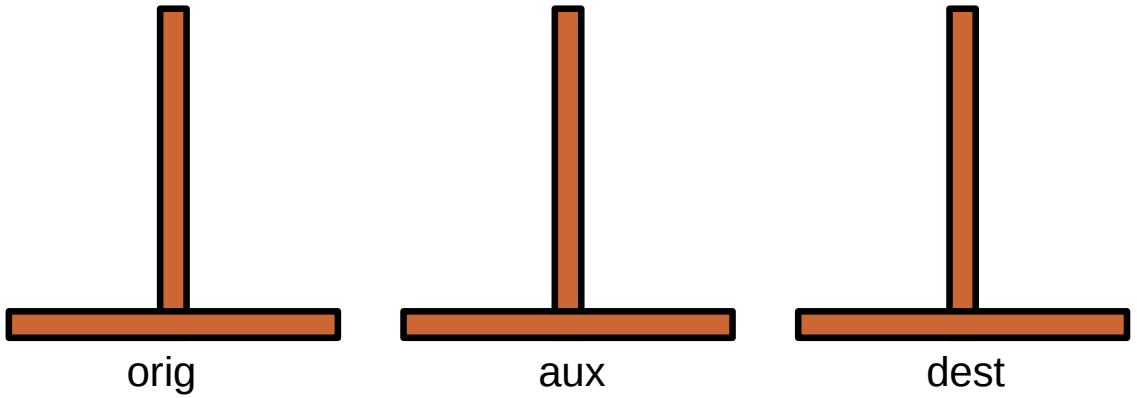


Problema

A torre de Hanói é um brinquedo com três estacas A, B e C e discos de tamanhos diferentes. O objetivo é mover todos os discos da estaca A para a estaca C respeitando as seguintes regras:

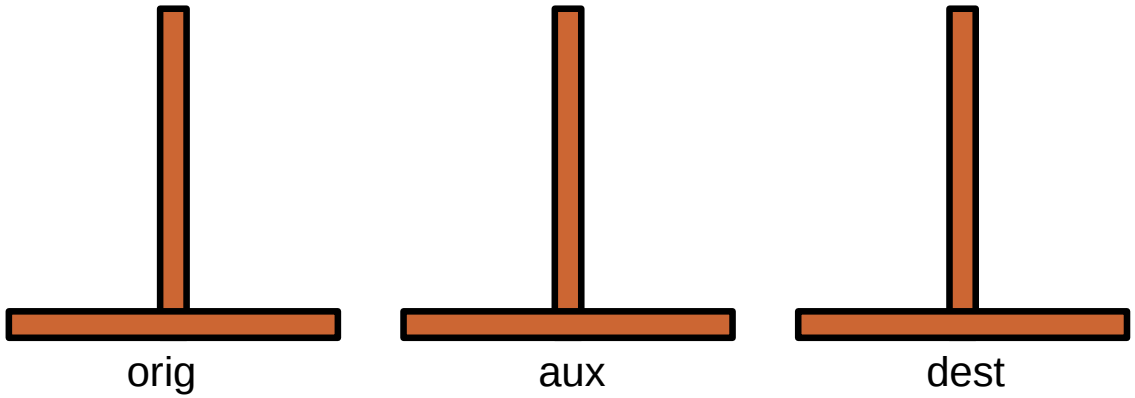
- Apenas um disco pode ser movido de cada vez.
- Um disco só pode ser colocado sobre um disco maior.

Torres de Hanói recursivo



Exemplo:

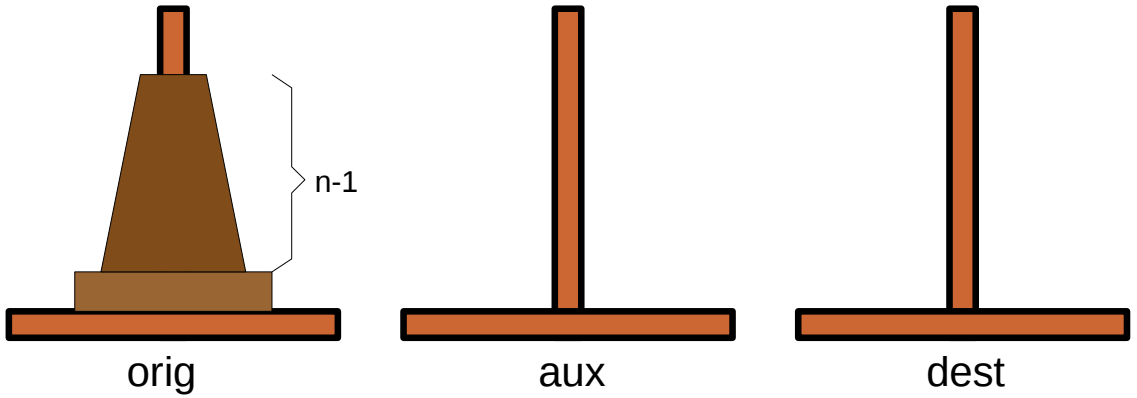
Torres de Hanói recursivo



Exemplo:

```
1.   void hanoi(int n, char orig, char dest, char aux) {
2.         if (n == 0) { /* caso base: não faz  nada */ }
```

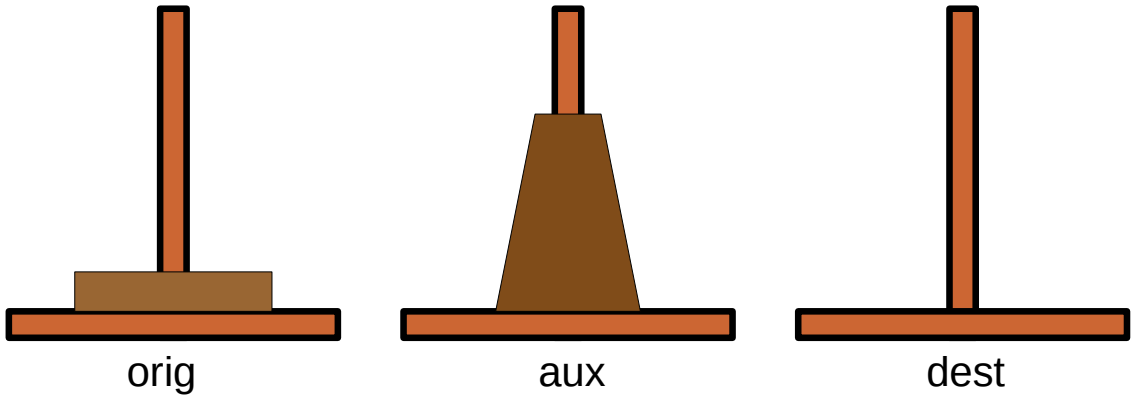
Torres de Hanói recursivo



Exemplo:

```
1.   void hanoi(int n, char orig, char dest, char aux) {
2.         if (n == 0) { /* caso base: não faz  nada */ }
3.         else {  /* caso geral */
```

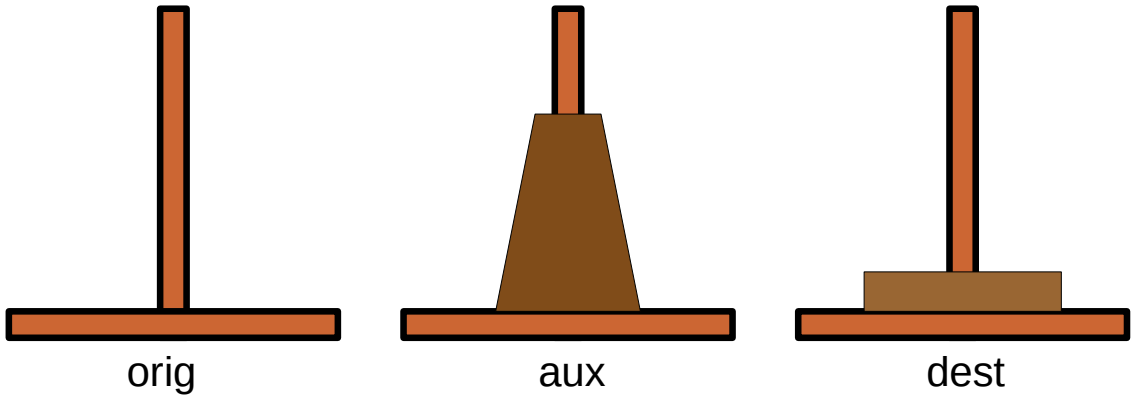
Torres de Hanói recursivo



Exemplo:

```
1.   void hanoi(int n, char orig, char dest, char aux) {
2.         if (n == 0) { /* caso base: não faz  nada */ }
3.         else {  /* caso geral */
4.                hanoi(n-1, orig, aux, dest);
```

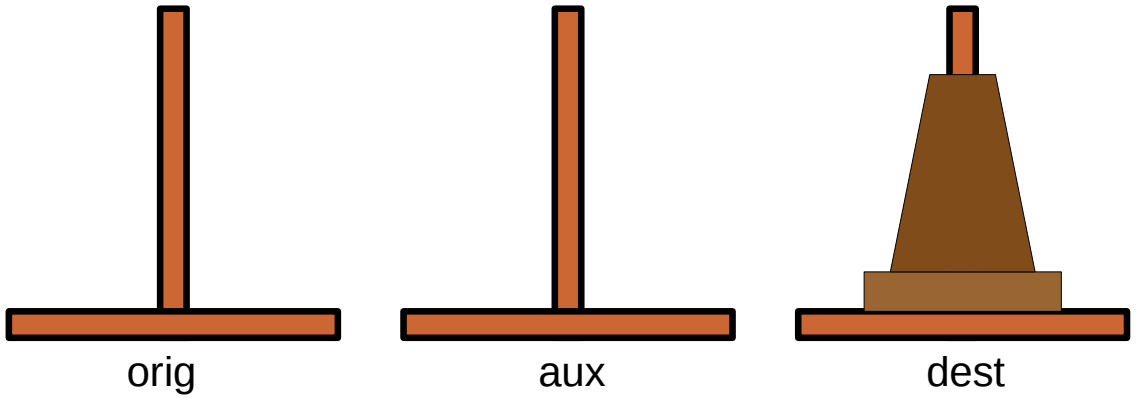
Torres de Hanói recursivo



Exemplo:

```
1.   void hanoi(int n, char orig, char dest, char aux) {
2.         if (n == 0) { /* caso base: não faz  nada */ }
3.         else {  /* caso geral */
4.                 hanoi(n-1, orig, aux, dest);
5.                 printf("move de %c para %c\n", orig, dest);
```

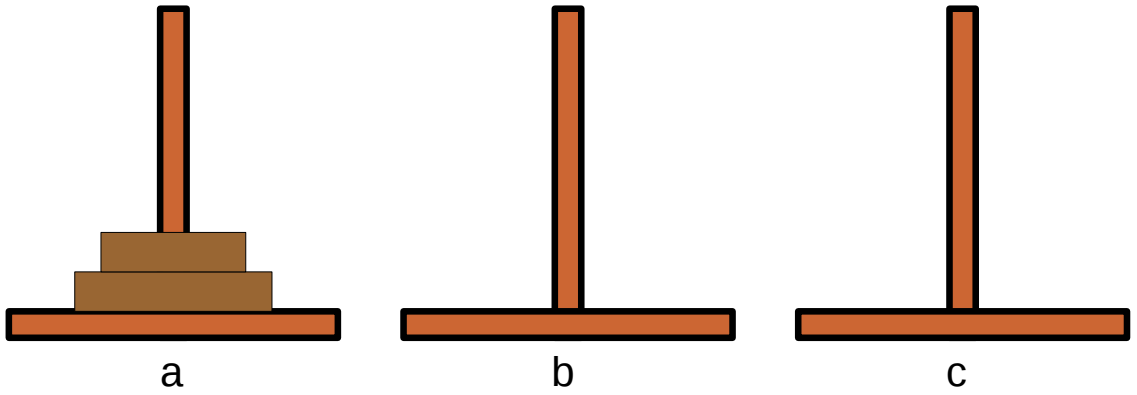
Torres de Hanói recursivo



Exemplo:

```
1.   void hanoi(int n, char orig, char dest, char aux) {
2.         if (n == 0) { /* caso base: não faz  nada */ }
3.         else {  /* caso geral */
4.                 hanoi(n-1, orig, aux, dest);
5.                 printf("move de %c para %c\n", orig, dest);
6.                 hanoi(n-1, aux, dest, orig);
7.         }
8.   }
```

Torres de Hanói recursivo

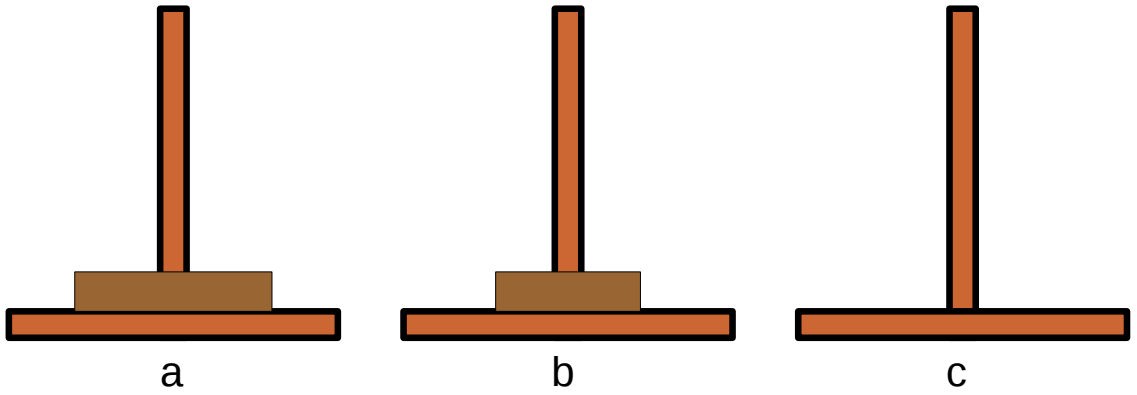


Exemplo:

```
1.   void hanoi(int n, char orig, char dest, char aux) {
2.         if (n == 0) { /* caso base: não faz  nada */ }
3.         else {  /* caso geral */
4.                 hanoi(n-1, orig, aux, dest);
5.                 printf("move de %c para %c\n", orig, dest);
6.                 hanoi(n-1, aux, dest, orig);
7.         }
8.   }

100. hanoi(2, 'a', 'c', 'b');   // vamos simular
```

Torres de Hanói recursivo

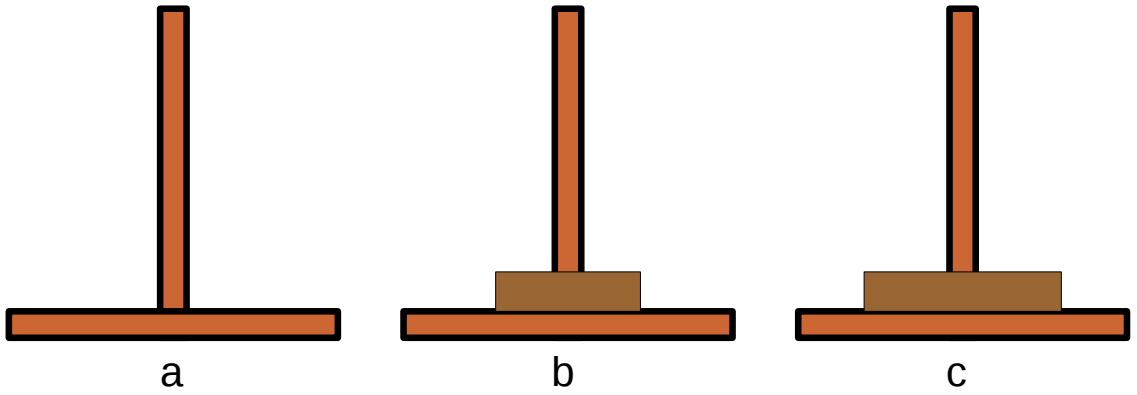


Exemplo:

```
1.   void hanoi(int n, char orig, char dest, char aux) {
2.         if (n == 0) { /* caso base: não faz  nada */ }
3.         else {  /* caso geral */
4.                 hanoi(n-1, orig, aux, dest);
5.                 printf("move de %c para %c\n", orig, dest);
6.                 hanoi(n-1, aux, dest, orig);
7.         }
8.   }

100. hanoi(2, 'a', 'c', 'b');   // vamos simular
```

Torres de Hanói recursivo

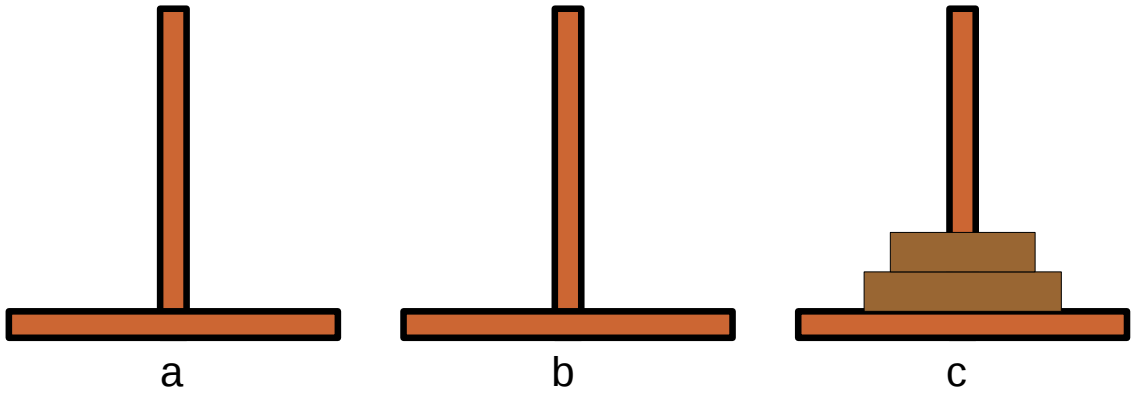


Exemplo:

```
1.   void hanoi(int n, char orig, char dest, char aux) {
2.         if (n == 0) { /* caso base: não faz  nada */ }
3.         else {  /* caso geral */
4.                 hanoi(n-1, orig, aux, dest);
5.                 printf("move de %c para %c\n", orig, dest);
6.                 hanoi(n-1, aux, dest, orig);
7.         }
8.   }

100. hanoi(2, 'a', 'c', 'b');   // vamos simular
```

Torres de Hanói recursivo



Exemplo:

```
1.   void hanoi(int n, char orig, char dest, char aux) {
2.         if (n == 0) { /* caso base: não faz  nada */ }
3.         else {  /* caso geral */
4.                 hanoi(n-1, orig, aux, dest);
5.                 printf("move de %c para %c\n", orig, dest);
6.                 hanoi(n-1, aux, dest, orig);
7.         }
8.   }

100. hanoi(2, 'a', 'c', 'b');   // vamos simular
```

Pilhas e recursão (novamente)

Agora também salvamos o endereço de retorno.

Registro de ativação:

Registro de ativação de uma função é o conjunto formado por:

1. Variáveis locais
2. Endereço de retorno após a chamada

Pilhas e recursão (novamente)

Agora também salvamos o endereço de retorno.

Registro de ativação:

Registro de ativação de uma função é o conjunto formado por:

1. Variáveis locais
2. Endereço de retorno após a chamada

Pilha de execução

Pilha de execução ou **pilha de chamadas** é a pilha dos registros de ativação das várias chamadas em execução em um programa.

Exercício 1

1. Escreva uma função que calcula o número de triângulos virados de ponta-a-cabeça (os triângulos com uma ponta em baixo e duas em cima) em uma grade de triângulos de altura n.
2. Crie um algoritmo recursivo que calcule a n-ésima potência de um número. O seu algoritmo não pode fazer mais do que $2 \log_2 n$ multiplicações. Demonstre isso.
3. Reescreva o algoritmo do item acima de forma iterativa.
4. Quem faz mais multiplicações: a versão iterativa ou a versão recursiva do fatorial? Os dois usam a mesma quantidade de memória?

Exercício 2 - Simulando recursão

Simulando o algoritmo hanoi recusivo com pilha

Vimos que na verdade a recursão nada mais é do uma sequência de chamadas de funções cujas as variáveis locais e o endereço de retorno são salvas na pilha. Com isso em mente:

1. quais são as variáveis que são salvas na pilha na função `hanoi`
2. uma chamada para uma função tem vários pontos de entrada: (a) quando ela é chamada inicialmente; (b) quando alguma função que ela tenha chamado retorna. Liste todos os pontos de entrada da função `hanoi` (isso é, quais são "os números" de linha a que voltamos sempre que uma chamada da pilha termina ou começa?).
3. descreva uma estrutura de dados que contenha o registro de ativação da pilha de chamadas do hanoi
4. (**desafio**) utilizando uma pilha e o resgistro de ativação descrito anteriormente, implemente uma versão iterativa da função `hanoi` que não usa recursão (explicitamente).