9002 — Aula 22
Algoritmos e Programação de Computadores

Instituto de Engenharia – UFMT

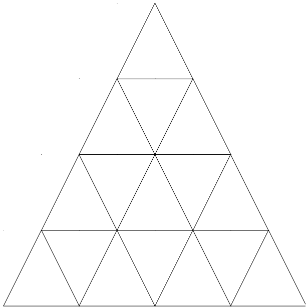
Segundo Semestre de 2014

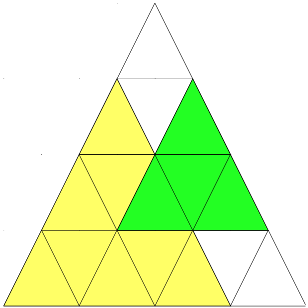
08 de dezembro de 2014

Roteiro

Introdução

Considere o problema a seguir.

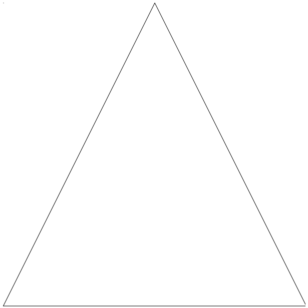




Problema

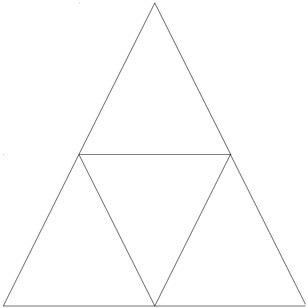
Quantos triângulos de pé (ver exemplos coloridos) podemos encontrar em uma grade de triângulos com altura n?

Triângulos



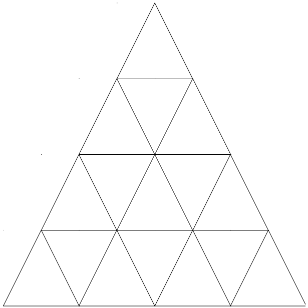
Para uma grade de altura $n = 1$, temos $t(1) = 1$ triângulo.

Triângulos



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

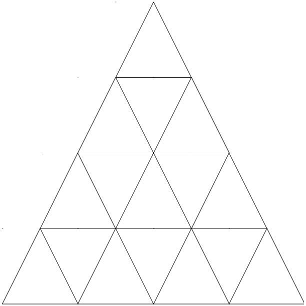
Triângulos



E para $n = 4$?

Podemos encontrar algum padrão?

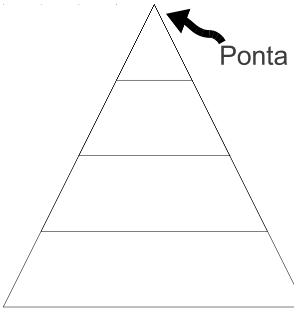
Triângulos



E para $n = 4$?

Podemos encontrar algum padrão?

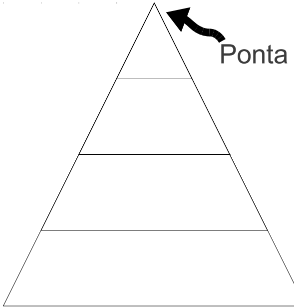
Triângulos



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

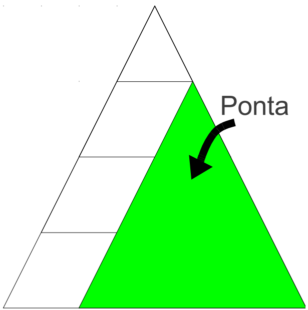
Além desses, quantos faltam?

Triângulos



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

Triângulos

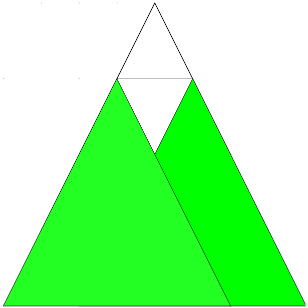


Faltam os triângulos do lado direito

e os triângulos do lado esquerdo.

Mas como calcular o número de triângulos de um certo lado?

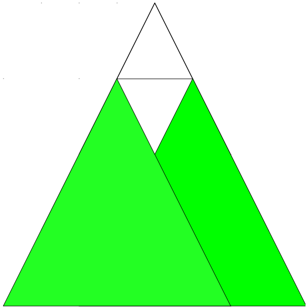
Triângulos



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

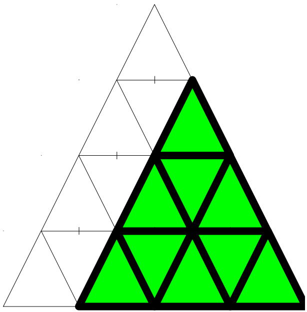
Mas como calcular o número de triângulos de um certo lado?

Triângulos



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

Triângulos

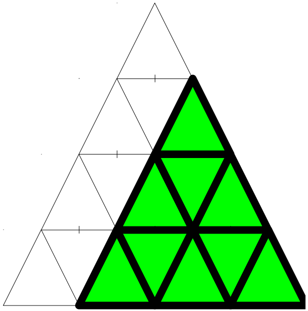


Caímos no mesmo problema anterior...

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

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

Triângulos

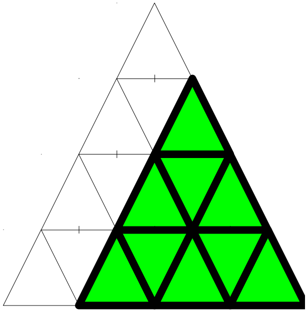


Caímos no mesmo problema anterior...

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

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

Triângulos

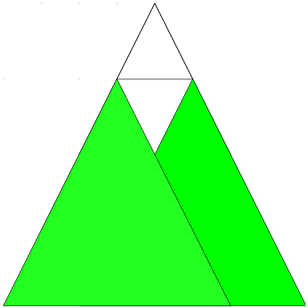


Caímos no mesmo problema anterior...

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

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

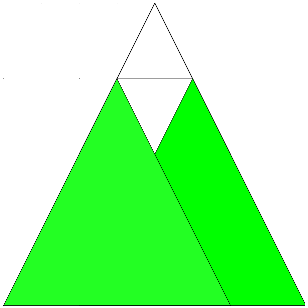
Triângulos



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

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

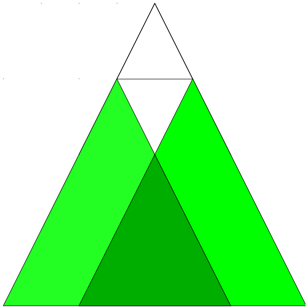
Triângulos



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

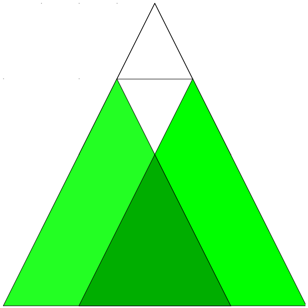


Suponha que já sabemos:
$t(1) = 1, t(2) = 4, t(3) = 10.$
Como podemos calcular $t(4)$?

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

$t(4) = 4 + t(3) + t(3) - t(2) = 20.$

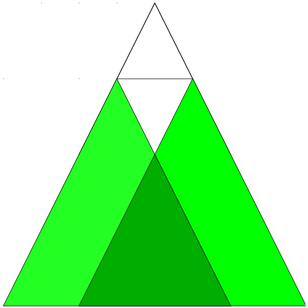
Triângulos



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.
$t(4) = 4 + t(3) + t(3) - t(2) = 20.$

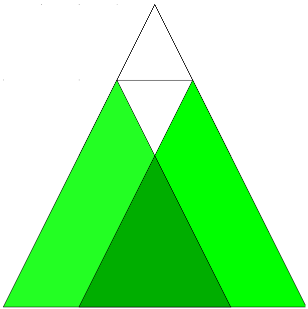
Triângulos



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

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

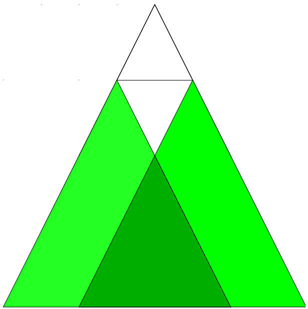
Triângulos



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, $t(n) = n + 2 \cdot t(n-1) - t(n-2)$.

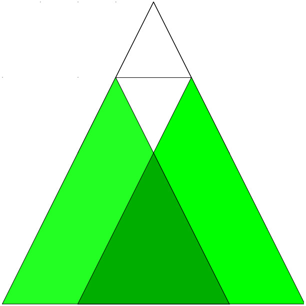
Triângulos



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



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.

```
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.

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.

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:

- 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.

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.

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.

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.

Exemplo - Fatorial

Certos problemas são naturalmente recursivos.

Problema

O fatorial de n é definido como:

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

Escreva um programa para calcular o valor do fatorial de um número.

Qual é a base da recursão?
Resposta: o caso $n = 0$.

A que instância menor nós reduzimos o problema?
Resposta: para a instância $(n-1)!$.

Como nós combinamos o resultado para resolver o problema original?
Resposta: multiplicando por n o resultado do problema menor.

Exemplo - Fatorial

Certos problemas são naturalmente recursivos.

Problema

O fatorial de n é definido como:

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

Escreva um programa para calcular o valor do fatorial de um número.

Qual é a base da recursão?

Resposta: o caso $n = 0$.

A que instância menor nós reduzimos o problema?

Resposta: para a instância $(n-1)!$.

Como nós combinamos o resultado para resolver o problema original?

Resposta: multiplicando por n o resultado do problema menor.

Exemplo - Fatorial

Certos problemas são naturalmente recursivos.

Problema

O fatorial de n é definido como:

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

Escreva um programa para calcular o valor do fatorial de um número.

Qual é a base da recursão?
Resposta: o caso $n = 0$.

A que instância menor nós reduzimos o problema?
Resposta: para a instância $(n-1)!$.

Como nós combinamos o resultado para resolver o problema original?
Resposta: multiplicando por n o resultado do problema menor.

Exemplo - Fatorial

Certos problemas são naturalmente recursivos.

Problema

O fatorial de n é definido como:

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

Escreva um programa para calcular o valor do fatorial de um número.

Qual é a base da recursão?
Resposta: o caso $n = 0$.

A que instância menor nós reduzimos o problema?
Resposta: para a instância $(n-1)!$.

Como nós combinamos o resultado para resolver o problema original?
Resposta: multiplicando por n o resultado do problema menor.

Exemplo - Fatorial

Certos problemas são naturalmente recursivos.

Problema

O fatorial de n é definido como:

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

Escreva um programa para calcular o valor do fatorial de um número.

Qual é a base da recursão?
Resposta: o caso $n = 0$.

A que instância menor nós reduzimos o problema?
Resposta: para a instância $(n-1)!$.

Como nós combinamos o resultado para resolver o problema original?
Resposta: multiplicando por n o resultado do problema menor.

Exemplo - Fatorial

Certos problemas são naturalmente recursivos.

Problema

O fatorial de n é definido como:

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

Escreva um programa para calcular o valor do fatorial de um número.

Qual é a base da recursão?
Resposta: o caso $n = 0$.

A que instância menor nós reduzimos o problema?
Resposta: para a instância $(n-1)!$.

Como nós combinamos o resultado para resolver o problema original?
Resposta: multiplicando por n o resultado do problema menor.

Exemplo - Fatorial

Certos problemas são naturalmente recursivos.

Problema

O fatorial de n é definido como:

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

Escreva um programa para calcular o valor do fatorial de um número.

Qual é a base da recursão?
Resposta: o caso $n = 0$.

A que instância menor nós reduzimos o problema?
Resposta: para a instância $(n-1)!$.

Como nós combinamos o resultado para resolver o problema original?
Resposta: multiplicando por n o resultado do problema menor.

Exemplo - Fatorial

```
int fatorial(int n) {
    int x, y;

    // Caso base
    if (n == 0) {
        return 1;

    // Caso geral
    } else {
        // reduzimos o problema para instância menor x
        x = n - 1;
        y = fatorial(x);

        // combinamos o resultado y da instância menor
        return n * y;
    }
}
```

Pilha de chamadas

Lidando com variáveis da função

- Toda função tem suas próprias variáveis locais
- **Então**, cada chamada da função `fatorial` cria as variáveis n, x e y.
- Mas chamamos a função fatorial várias vezes: `fatorial(n)`, `fatorial(n-1)`, ..., `fatorial(1)` e `fatorial(0)`!
- **Então**, em um dado instante, podem existir várias variáveis n, x e y, um trio para cada chamada.

Pilha de chamadas

Para manter várias cópias de variáveis locais de uma função, elas são colocadas em uma **pilha de chamada** (ou **pilha de execução**) toda vez que uma função é chamada e são retiradas da pilha quando a função termina.

Pilha de chamadas

Lidando com variáveis da função

- Toda função tem suas próprias variáveis locais
- **Então**, cada chamada da função `fatorial` cria as variáveis n, x e y.
- Mas chamamos a função fatorial várias vezes: `fatorial(n)`, `fatorial(n-1)`, ..., `fatorial(1)` e `fatorial(0)`!
- **Então**, em um dado instante, podem existir várias variáveis n, x e y, um trio para cada chamada.

Pilha de chamadas

Para manter várias cópias de variáveis locais de uma função, elas são colocadas em uma **pilha de chamada** (ou **pilha de execução**) toda vez que uma função é chamada e são retiradas da pilha quando a função termina.

Pilha de chamadas

Lidando com variáveis da função

- Toda função tem suas próprias variáveis locais
- **Então**, cada chamada da função `fatorial` cria as variáveis n, x e y.

- Mas chamamos a função fatorial várias vezes: `fatorial(n)`, `fatorial(n-1)`, ..., `fatorial(1)` e `fatorial(0)`!
- **Então**, em um dado instante, podem existir várias variáveis n, x e y, um trio para cada chamada.

Pilha de chamadas

Para manter várias cópias de variáveis locais de uma função, elas são colocadas em uma **pilha de chamada** (ou **pilha de execução**) toda vez que uma função é chamada e são retiradas da pilha quando a função termina.

Pilha de chamadas

Lidando com variáveis da função

- Toda função tem suas próprias variáveis locais
- **Então**, cada chamada da função `fatorial` cria as variáveis n, x e y.

- Mas chamamos a função fatorial várias vezes: `fatorial(n)`, `fatorial(n-1)`, ..., `fatorial(1)` e `fatorial(0)`!
- **Então**, em um dado instante, podem existir várias variáveis n, x e y, um trio para cada chamada.

Pilha de chamadas

Para manter várias cópias de variáveis locais de uma função, elas são colocadas em uma **pilha de chamada** (ou **pilha de execução**) toda vez que uma função é chamada e são retiradas da pilha quando a função termina.

Pilha de chamadas

Lidando com variáveis da função

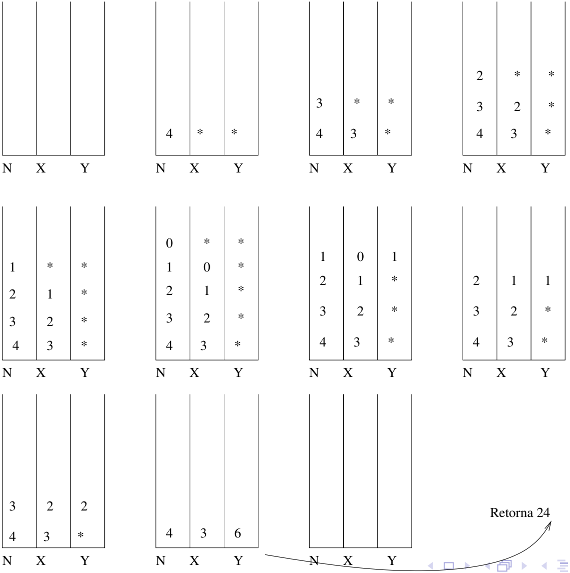
- Toda função tem suas próprias variáveis locais
- **Então**, cada chamada da função `fatorial` cria as variáveis n, x e y.
- Mas chamamos a função fatorial várias vezes: `fatorial(n)`, `fatorial(n-1)`, ..., `fatorial(1)` e `fatorial(0)`!
- **Então**, em um dado instante, podem existir várias variáveis n, x e y, um trio para cada chamada.

Pilha de chamadas

Para manter várias cópias de variáveis locais de uma função, elas são colocadas em uma **pilha de chamada** (ou **pilha de execução**) toda vez que uma função é chamada e são retiradas da pilha quando a função termina.

Pilha de chamadas - Fatorial

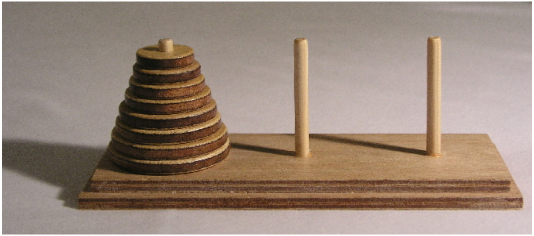
Estado da pilha de chamadas para *fatorial(4)*:



Exercício

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. Escreva uma função recursiva que calcule o *n*-ésimo valor da sequência de Fibonacci.

Exercício - 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.

Exercício - Torres de Hanói

1. Entre no endereço `http://www6.ufrgs.br/psicoeduc/hanoi/` e resolva a torre de Hanói para o número de discos $n = 3$, $n = 4$ e $n = 5$.
2. Tente escrever uma programa em C que leia um número n do teclado e instrua o usuário a resolver a torre de Hanói com n discos.
   Se precisar de dicas, veja a próxima página.
   1. É difícil resolver o problema quando temos apenas um ou dois discos? Esses casos são básicos?
   2. Se soubermos resolver o problema de mover os discos da estaca A para C, como podemos resolver o problema de mover os discos da estaca B para a C?
   3. Se tivermos dez discos na estaca A, mas os nove discos superiores estiverem colados, como podemos mover todos para a estaca C?

Exercício - Torres de Hanói

1. Entre no endereço `http://www6.ufrgs.br/psicoeduc/hanoi/` e resolva a torre de Hanói para o número de discos $n = 3$, $n = 4$ e $n = 5$.
2. Tente escrever uma programa em C que leia um número n do teclado e instrua o usuário a resolver a torre de Hanói com n discos.
   Responda às seguintes perguntas:
   1. É difícil resolver o problema quando temos apenas um ou dois discos? Esses casos são básicos?
   2. Se soubermos resolver o problema de mover os discos da estaca A para C, como podemos resolver o problema de mover os discos da estaca B para a C?
   3. Se tivermos dez discos na estaca A, mas os nove discos superiores estiverem colados, como podemos mover todos para a estaca C?