

RECURSÃO ... NOVAMENTE!?

MC102 - Algoritmos e
Programação de
Computadores

Santiago Valdés Ravelo
[https://ic.unicamp.br/~santiago/
ravelo@unicamp.br](https://ic.unicamp.br/~santiago/ravelo@unicamp.br)

06/25

25



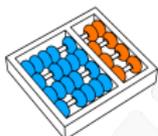
UNICAMP



Recursão, aqui vamos de novo!



DÚVIDAS DA AULA ANTERIOR

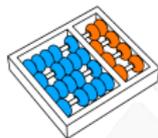


Dúvidas selecionadas

- ▶ Ao chamar um método sort, ele está executando qual dos algoritmos vistos?
- ▶ Existe uma maneira optimal de escolher o pivô no quicksort?
- ▶ Como o quicksort consegue, em média, ser mais rápido que o merge sort?
- ▶ Qual a vantagem da recursão na ordenação? Pq pra mim parece só mais complexo, demorado e consome mais memória.
- ▶ A ordenação por recursão é mais útil apenas para listas grandes?
- ▶ Com tantos metodos de ordenação, existe alguma forma de escolher qual o melhor para um caso?
- ▶ Não entendi o QuickSort.
- ▶ Não seria mais eficiente implementar também o caso base para listas com tamanho 2 para o merge sort?
- ▶ O mergesort é mais eficiente para listas mais ordenadas?
- ▶ A maioria dos softwares atuais utilizam o quicksort, ou é preferível utilizar uma combinação de vários tipos de formas de ordenação?
- ▶ Pq usamos uma lista auxiliar no mergesort em vez de ordenar diretamente no original?
- ▶ Estou achando muito interessantes as relações matemáticas associadas à complexidade de algoritmos. É possível que algum programa que revisa código calcule diretamente a complexidade operacional do programa?
- ▶ Agora que as dúvidas serão extras, como vai ficar os pesos da média?



UMA SOLUÇÃO RECURSIVA



Torres de Hanoi

Dadas três torres e n discos, onde:

- ▶ Cada disco tem tamanho diferente.
- ▶ Todos os discos estão empilhados na primeira torre, de forma tal que um disco maior não está em cima de um menor.

O objetivo:

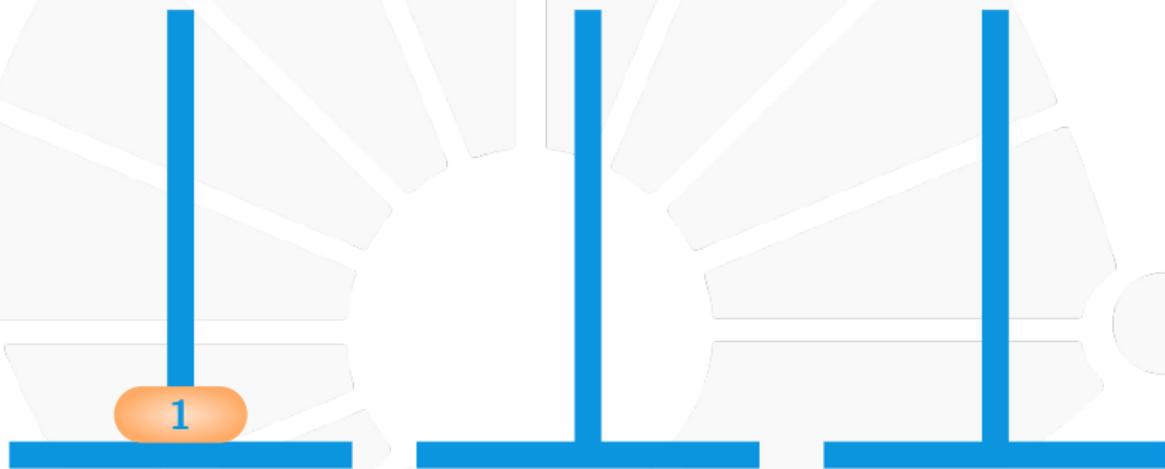
- ▶ Mover os discos da primeira torre até a terceira.

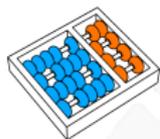
Restrições:

- ▶ Apenas é possível mover um disco por vez.
- ▶ Um disco maior nunca deve ser colocado em cima de um menor.

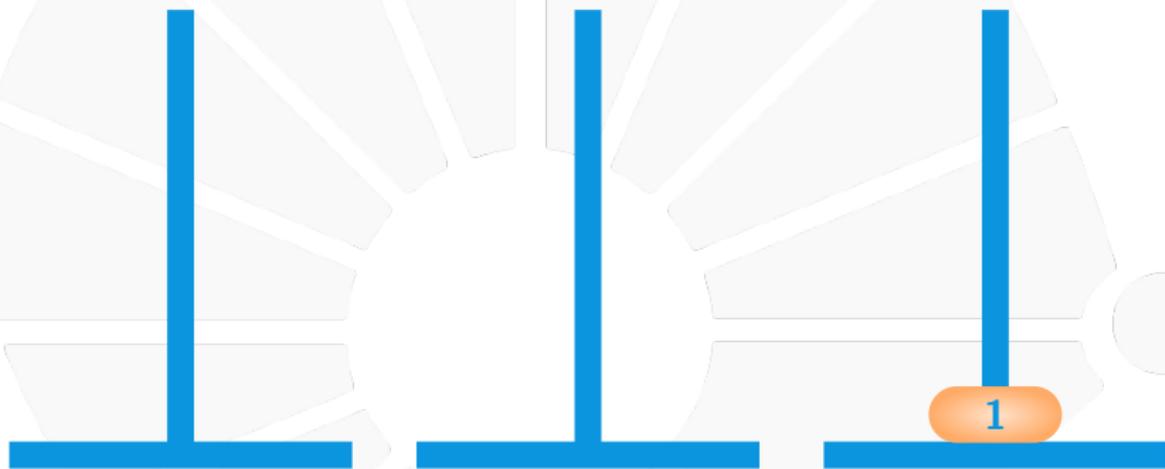


Torres de Hanoi – 1 discos

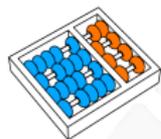




Torres de Hanoi – 1 discos



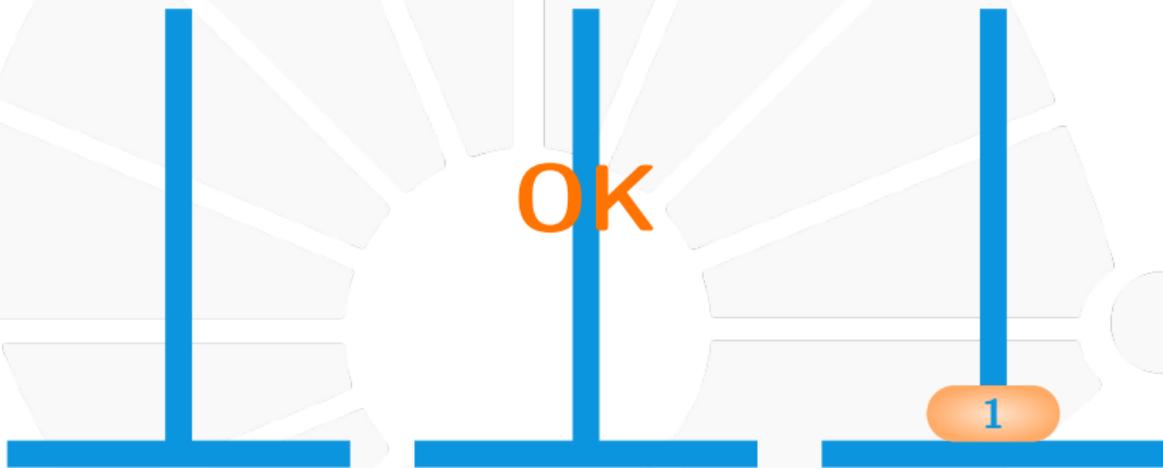
Mover o disco da torre 1 para a 3.

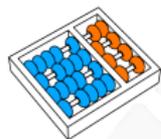


Torres de Hanoi – 1 discos

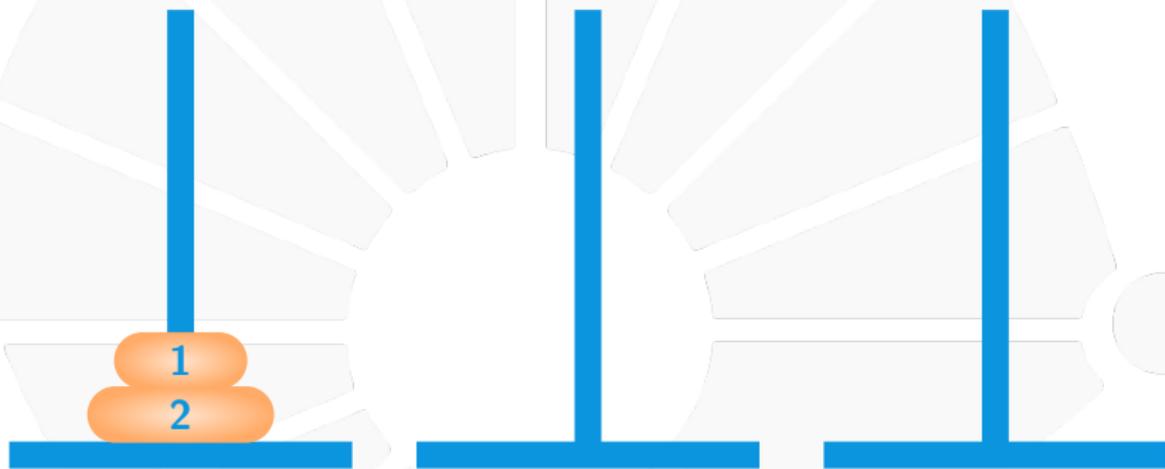
OK

1



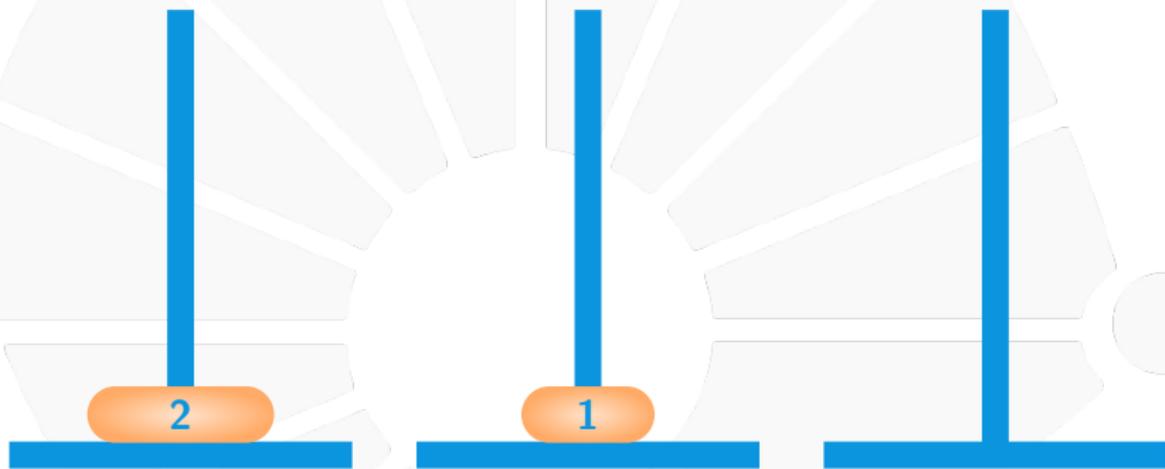


Torres de Hanoi – 2 discos

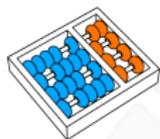




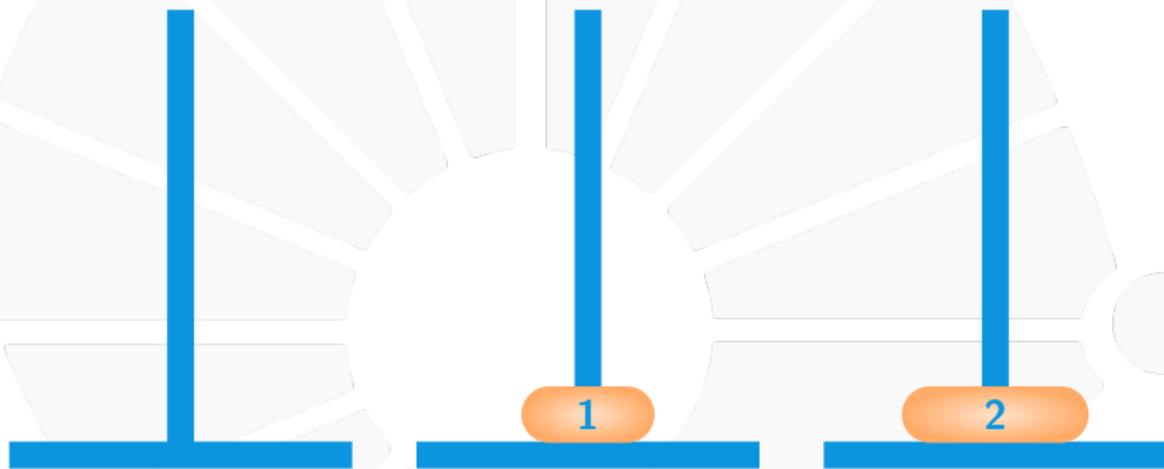
Torres de Hanoi – 2 discos



Mover o disco da torre 1 para a 2.



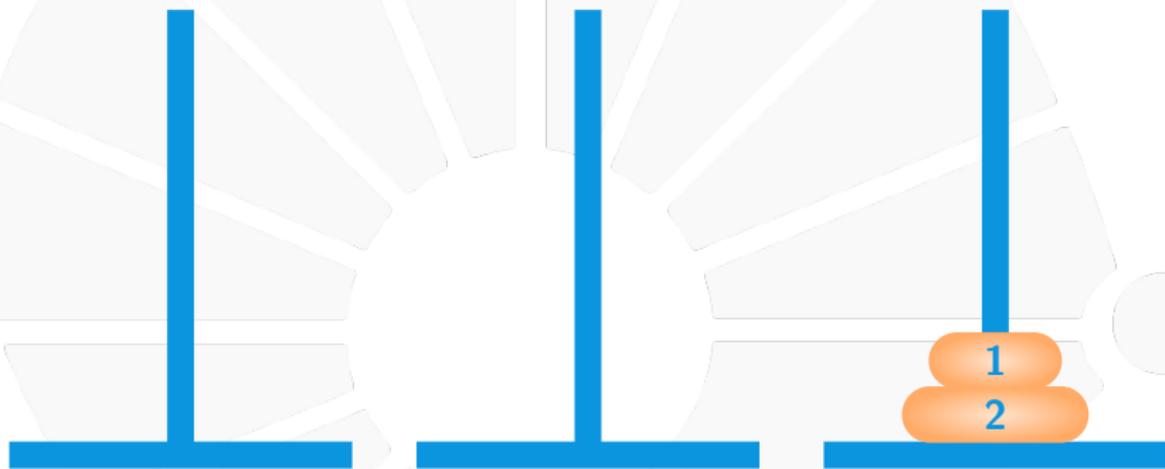
Torres de Hanoi – 2 discos



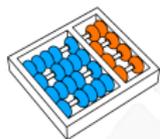
Mover o disco da torre 1 para a 3.



Torres de Hanoi – 2 discos

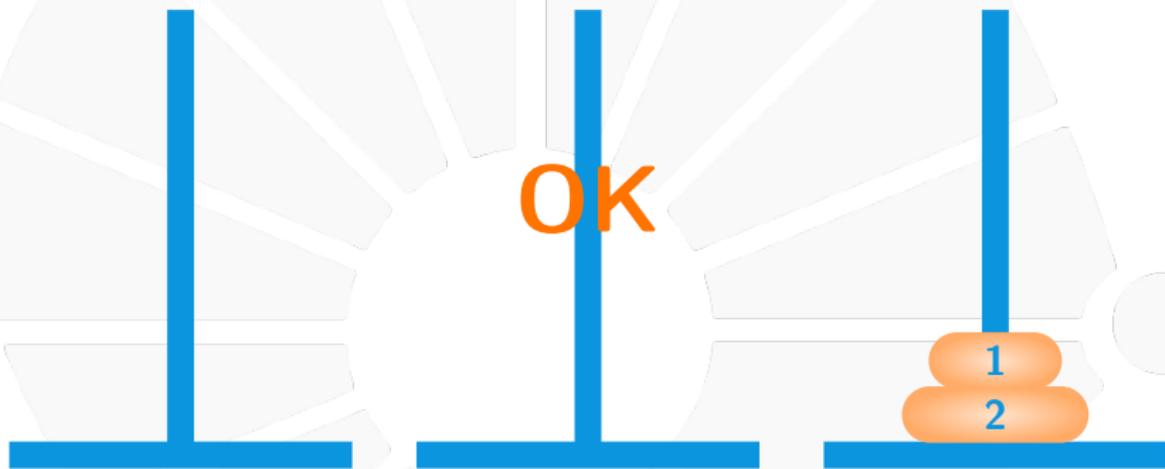


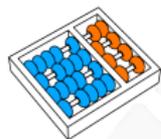
Mover o disco da torre 2 para a 3.



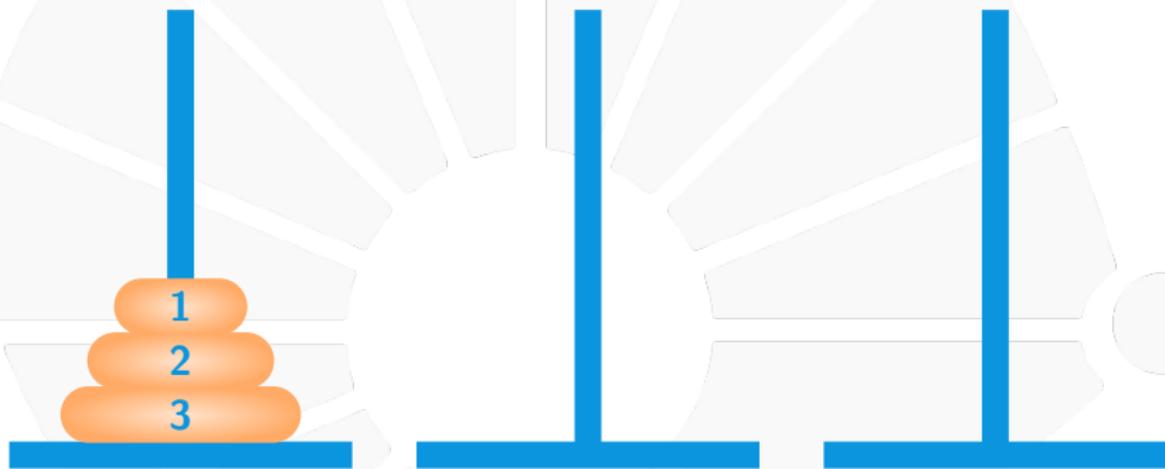
Torres de Hanoi – 2 discos

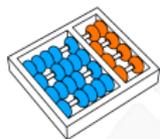
OK



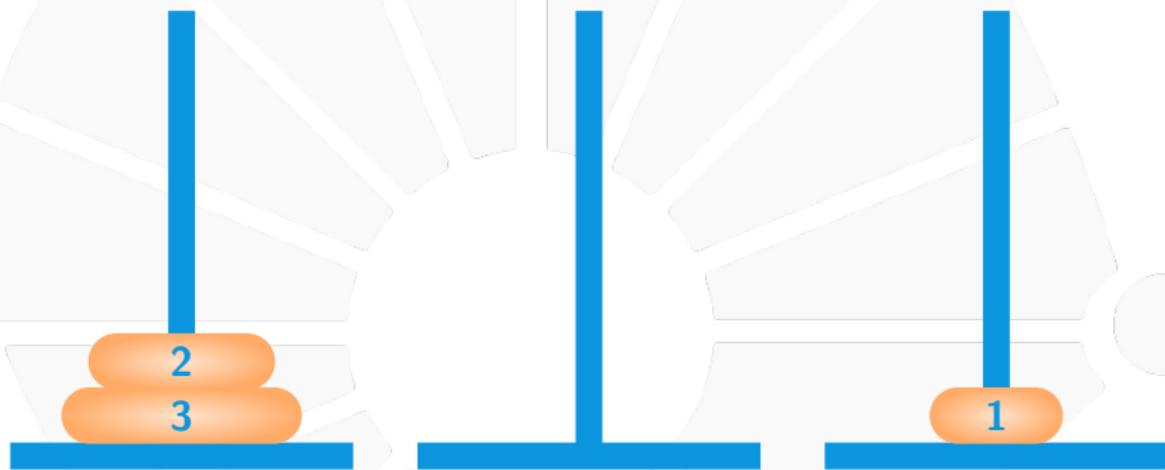


Torres de Hanoi – 3 discos

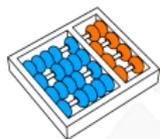




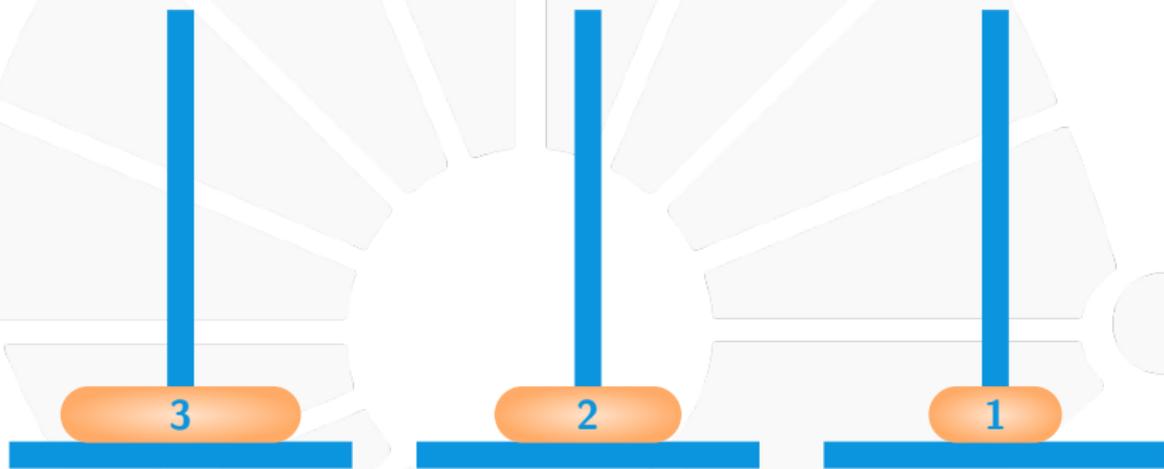
Torres de Hanoi – 3 discos



Mover o disco da torre 1 para a 3.



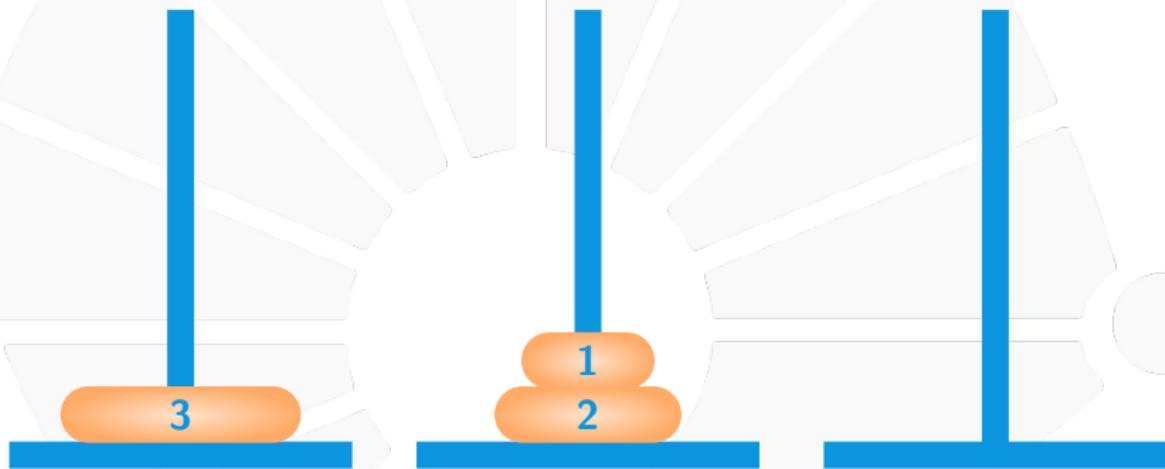
Torres de Hanoi – 3 discos



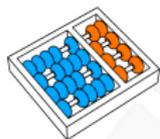
Mover o disco da torre 1 para a 2.



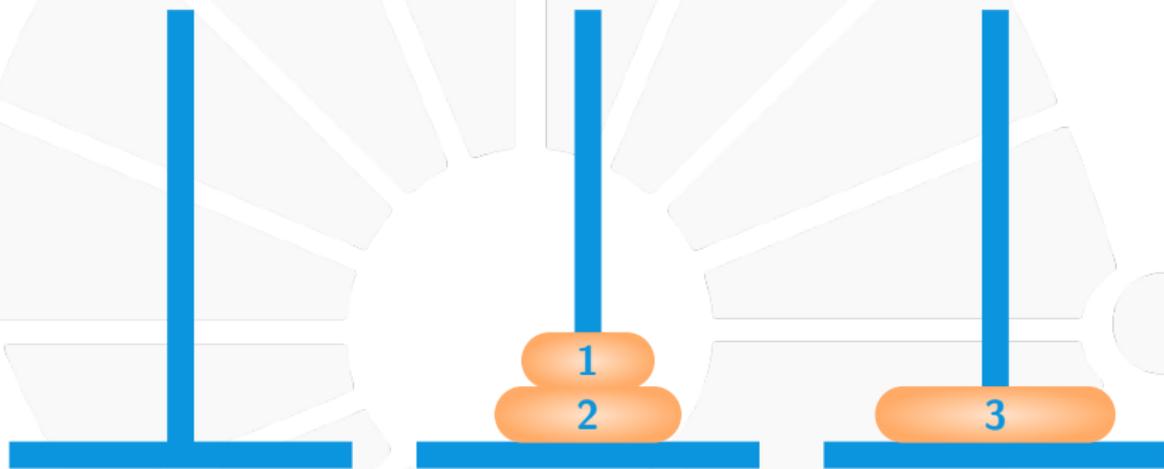
Torres de Hanoi – 3 discos



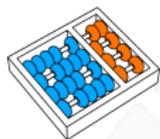
Mover o disco da torre 3 para a 2.



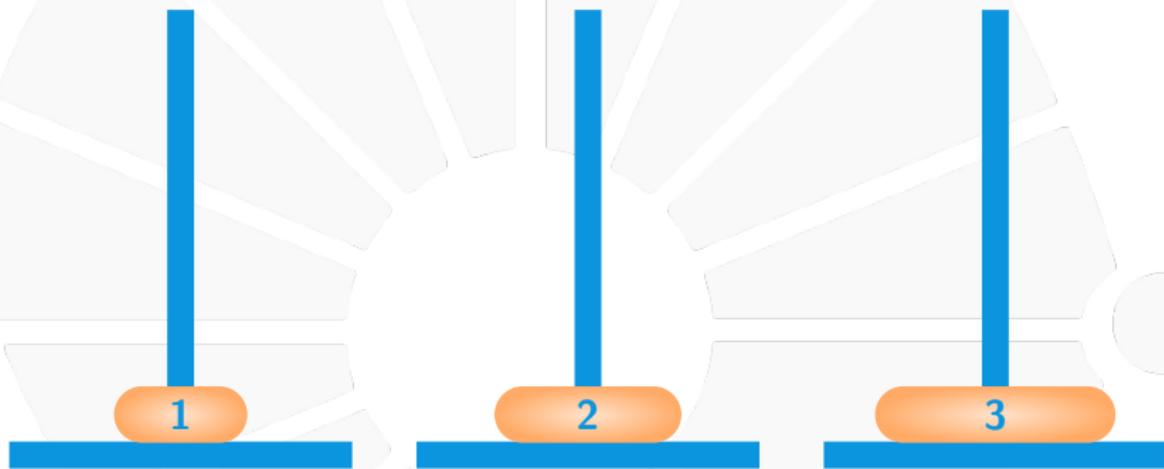
Torres de Hanoi – 3 discos



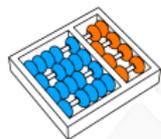
Mover o disco da torre 1 para a 3.



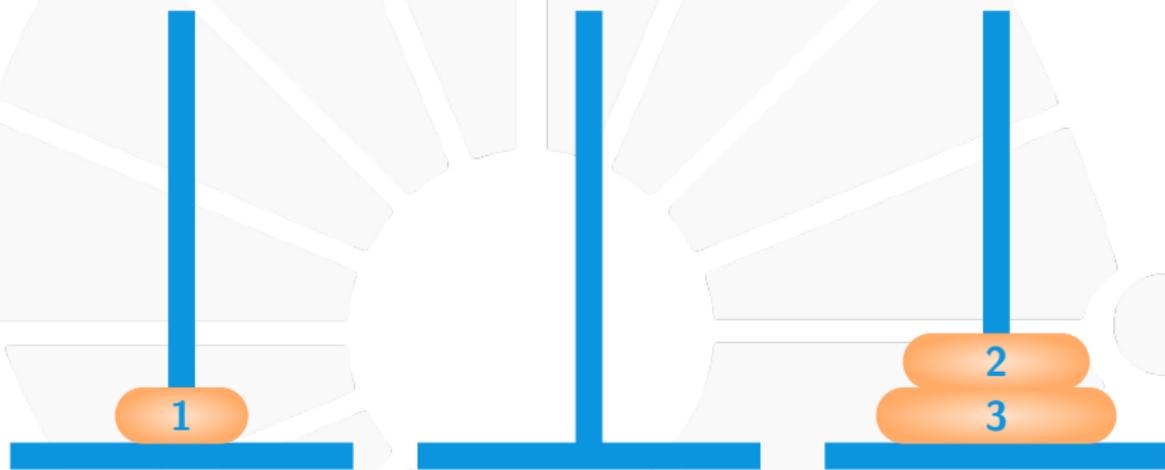
Torres de Hanoi – 3 discos



Mover o disco da torre 2 para a 1.



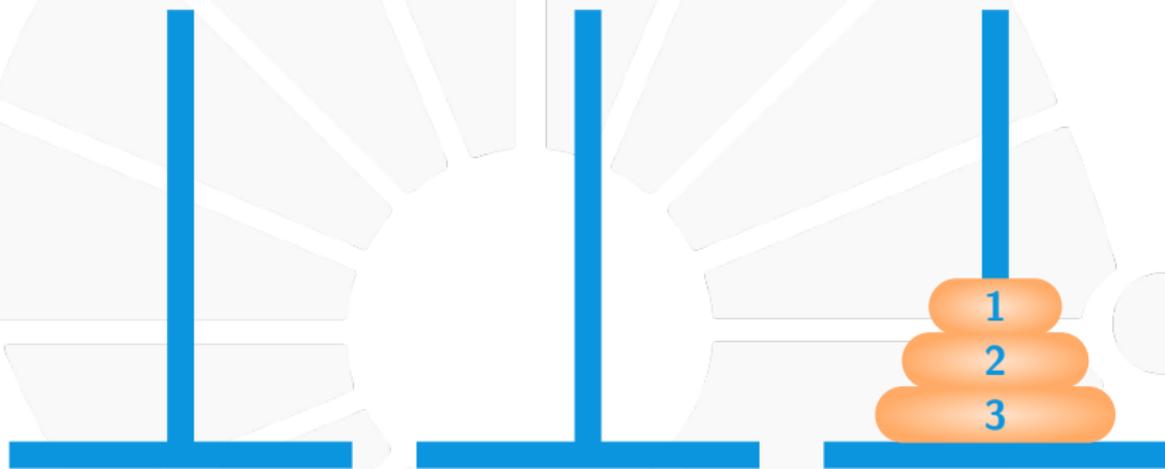
Torres de Hanoi – 3 discos



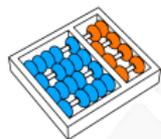
Mover o disco da torre 2 para a 3.



Torres de Hanoi – 3 discos

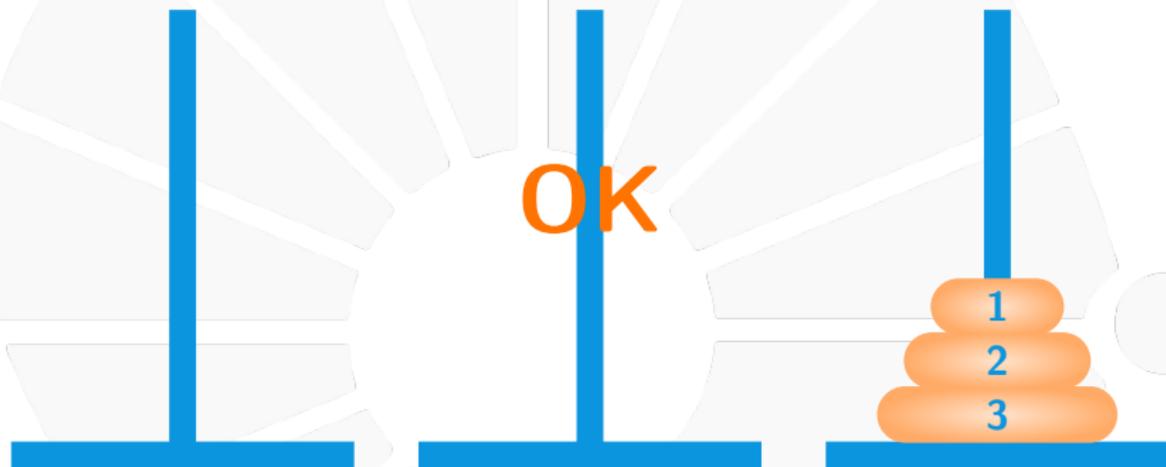


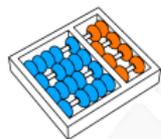
Mover o disco da torre 1 para a 3.



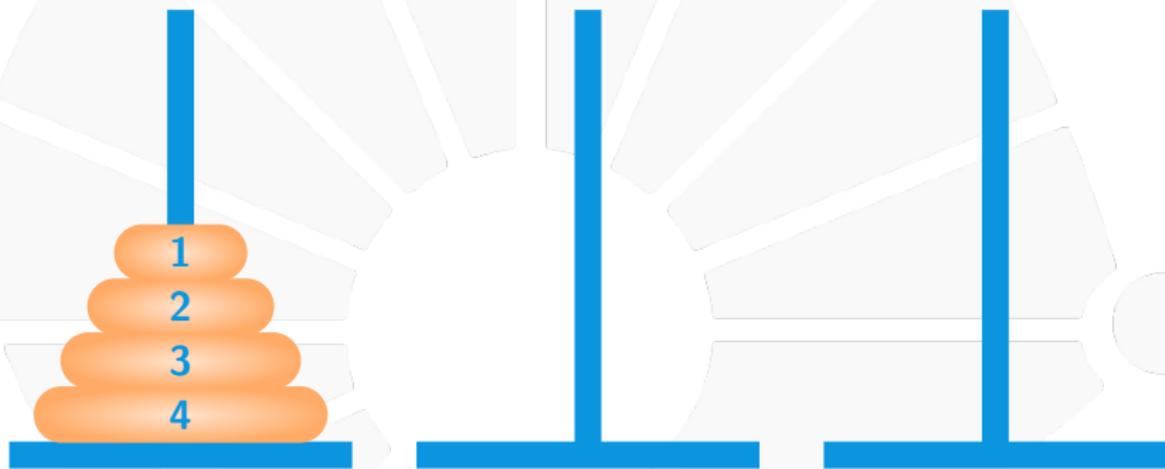
Torres de Hanoi – 3 discos

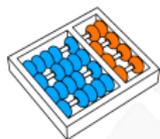
OK



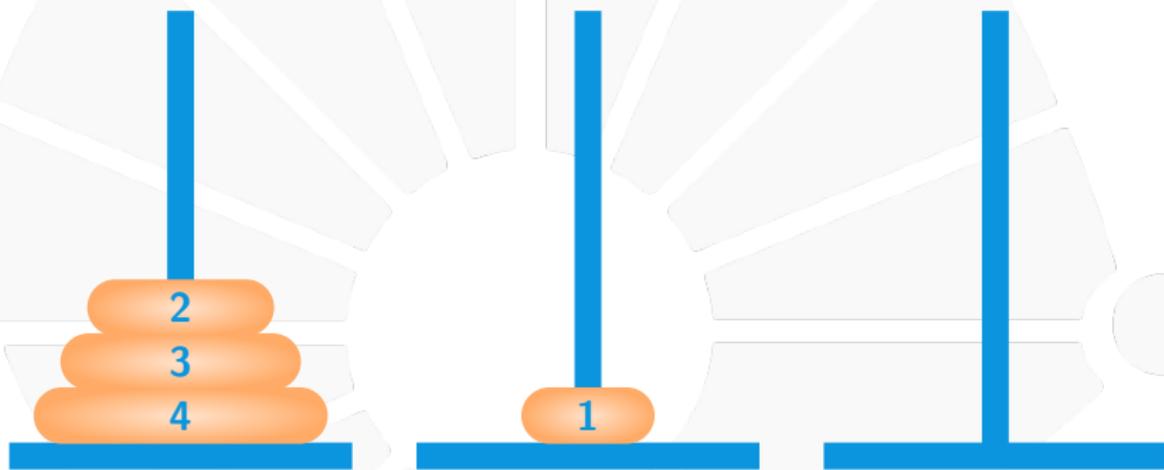


Torres de Hanoi – 4 discos

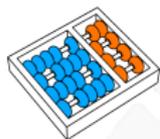




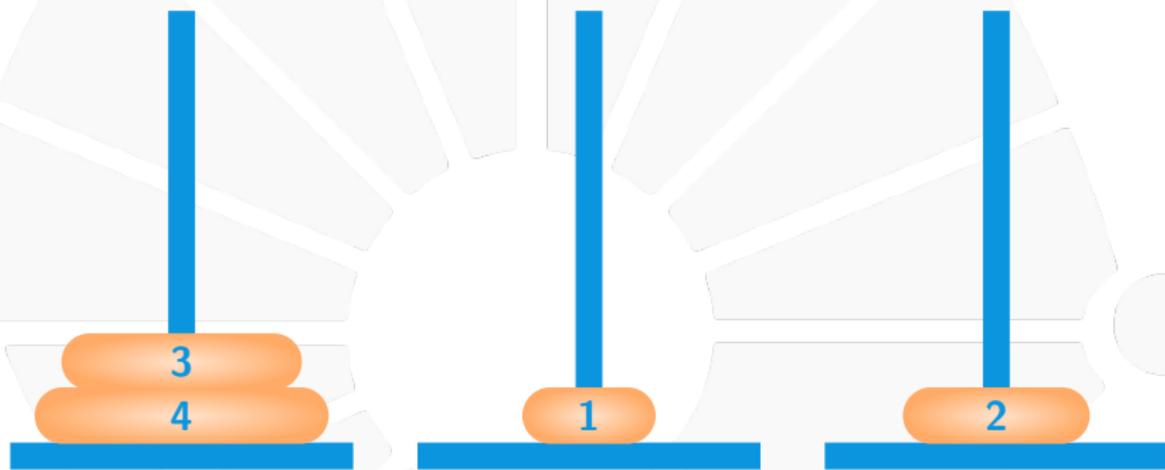
Torres de Hanoi – 4 discos



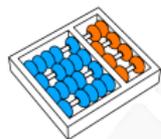
Mover o disco da torre 1 para a 2.



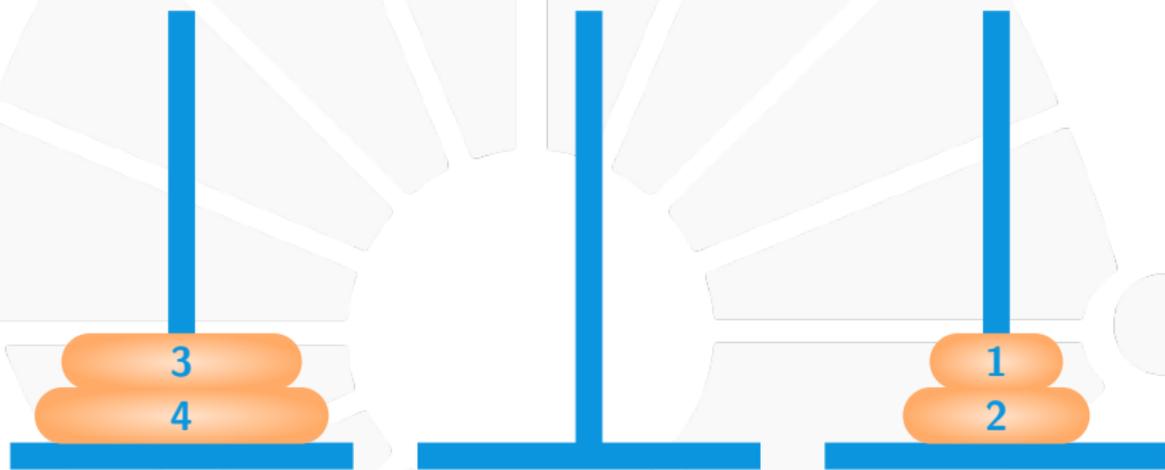
Torres de Hanoi – 4 discos



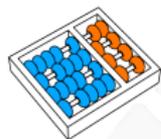
Mover o disco da torre 1 para a 3.



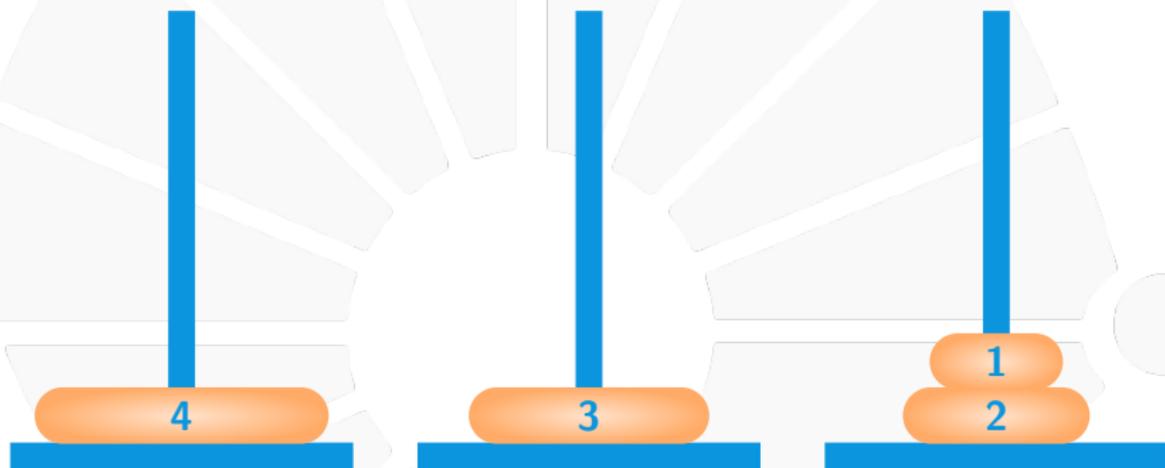
Torres de Hanoi – 4 discos



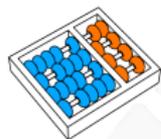
Mover o disco da torre 2 para a 3.



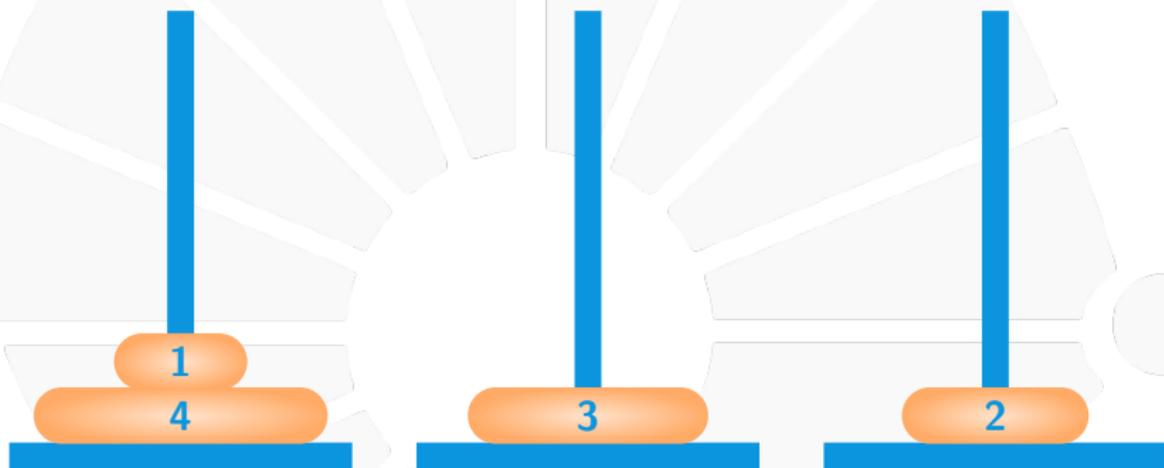
Torres de Hanoi – 4 discos



Mover o disco da torre 1 para a 2.



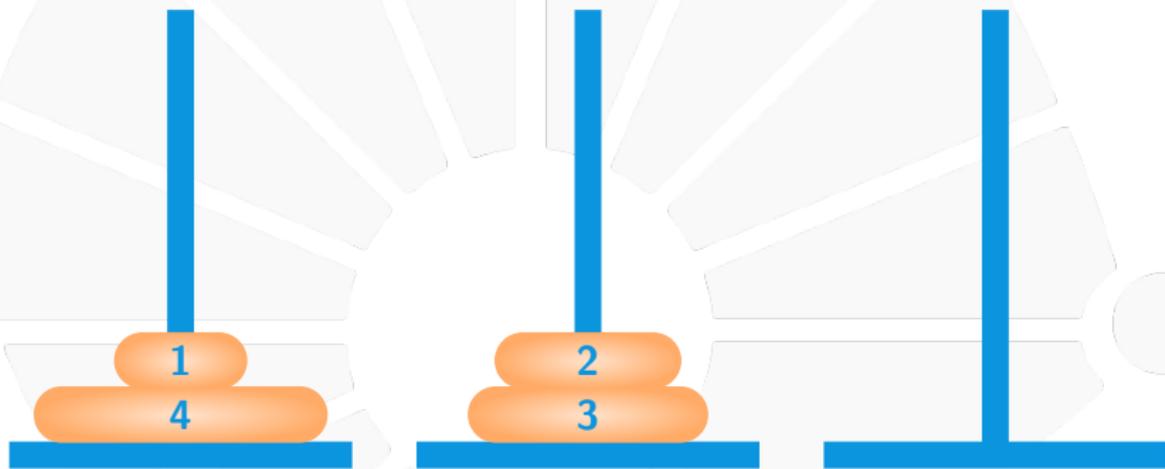
Torres de Hanoi – 4 discos



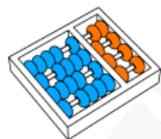
Mover o disco da torre 3 para a 1.



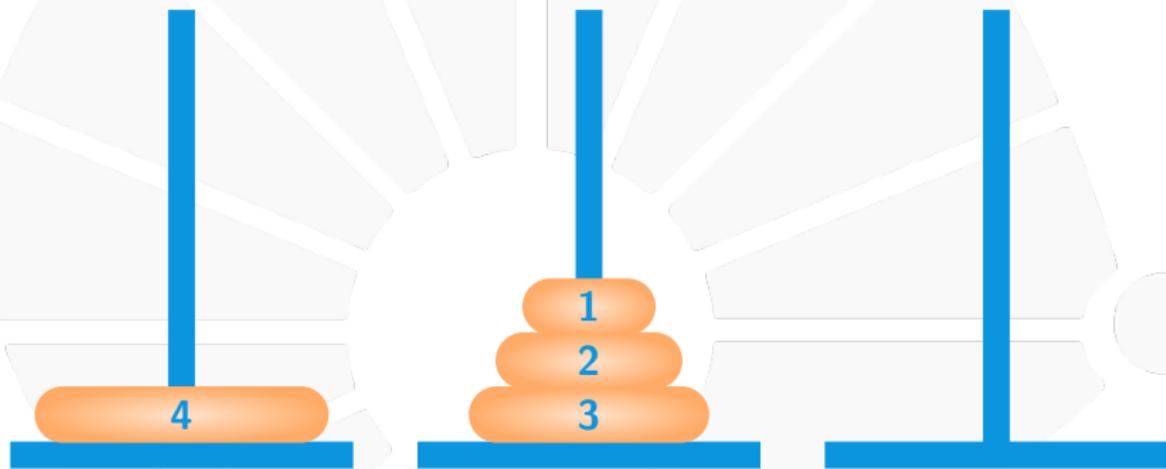
Torres de Hanoi – 4 discos



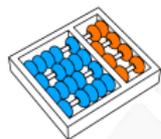
Mover o disco da torre 3 para a 2.



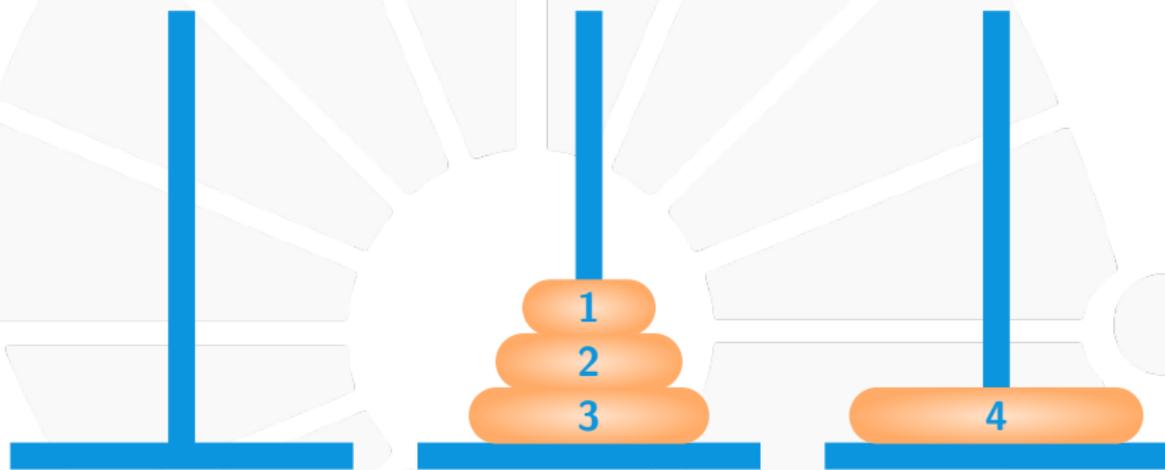
Torres de Hanoi – 4 discos



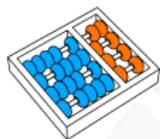
Mover o disco da torre 1 para a 2.



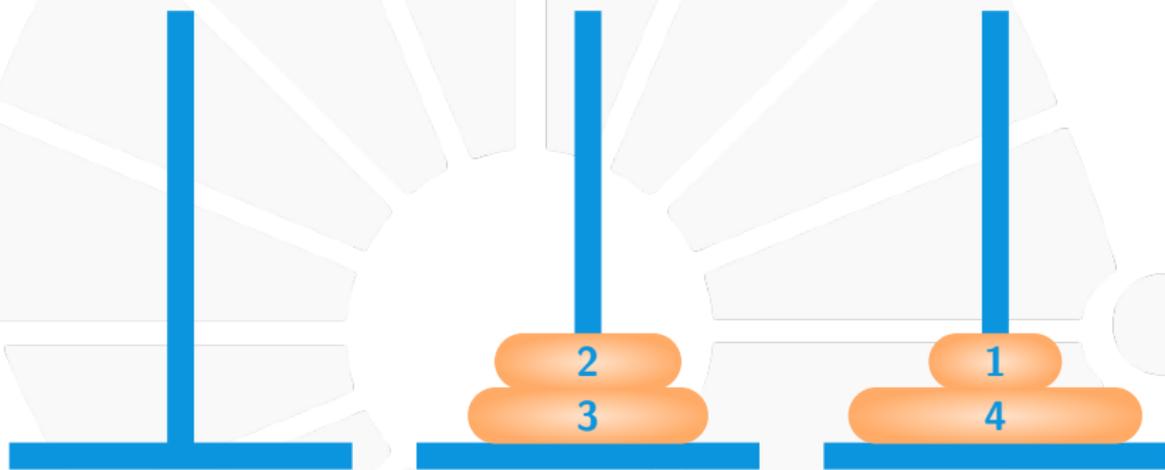
Torres de Hanoi – 4 discos



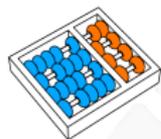
Mover o disco da torre 1 para a 3.



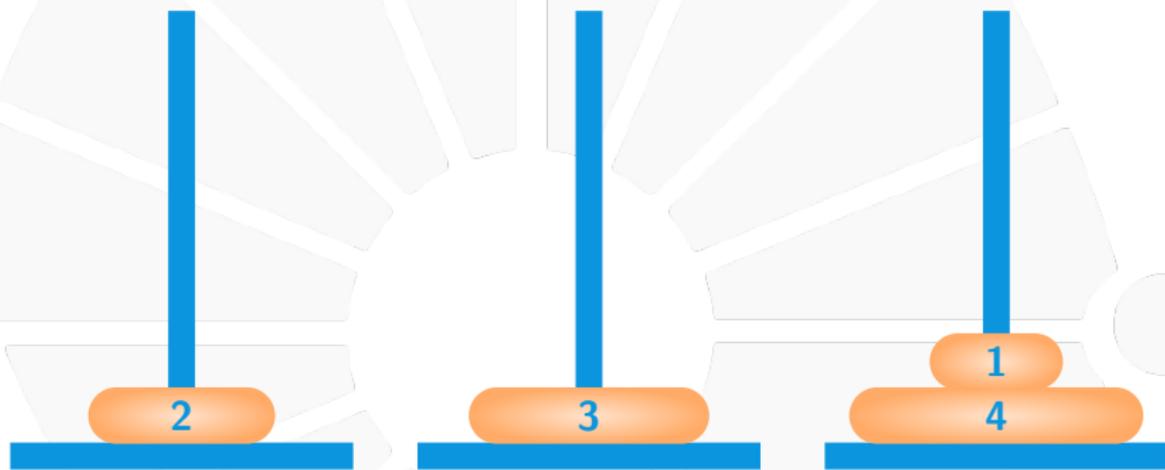
Torres de Hanoi – 4 discos



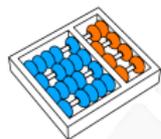
Mover o disco da torre 2 para a 3.



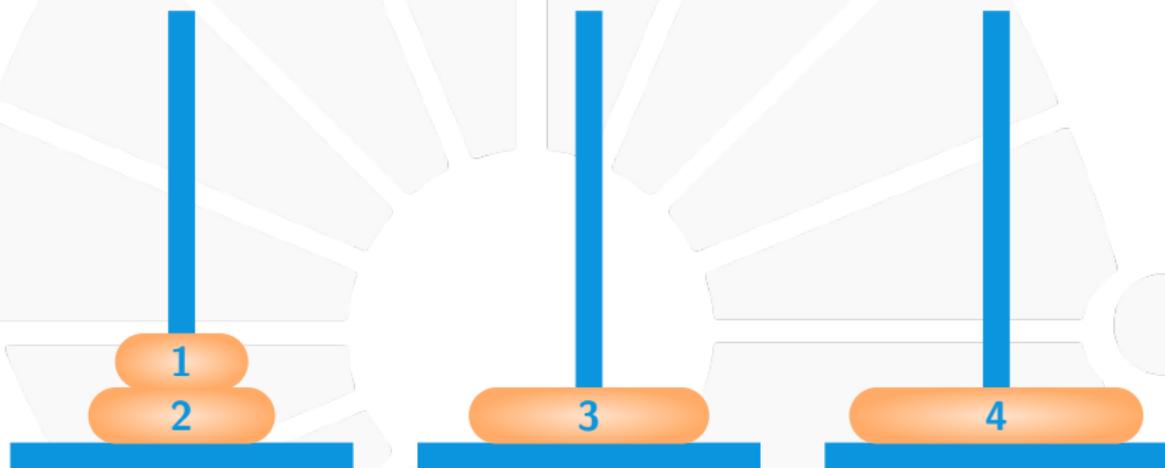
Torres de Hanoi – 4 discos



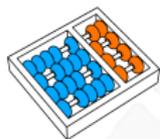
Mover o disco da torre 2 para a 1.



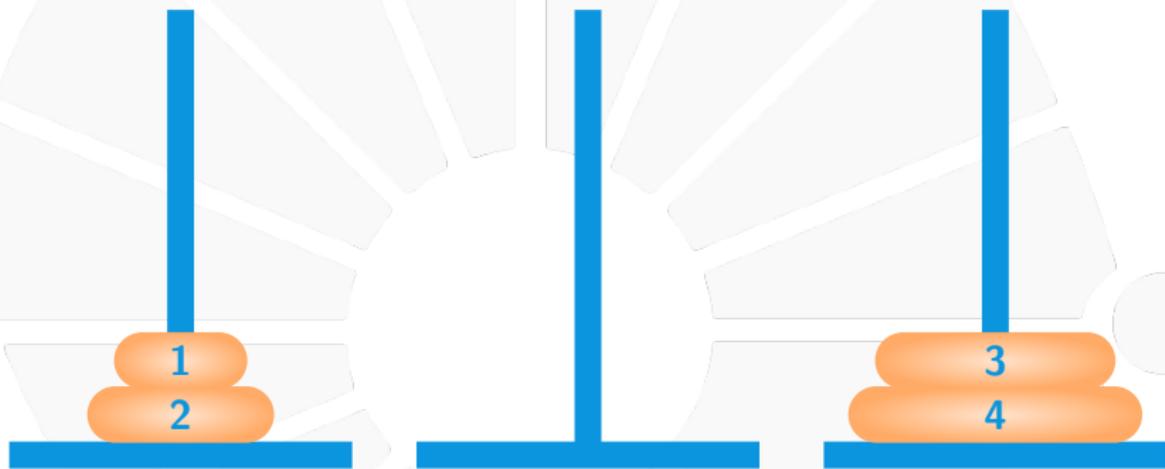
Torres de Hanoi – 4 discos



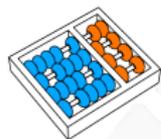
Mover o disco da torre 3 para a 1.



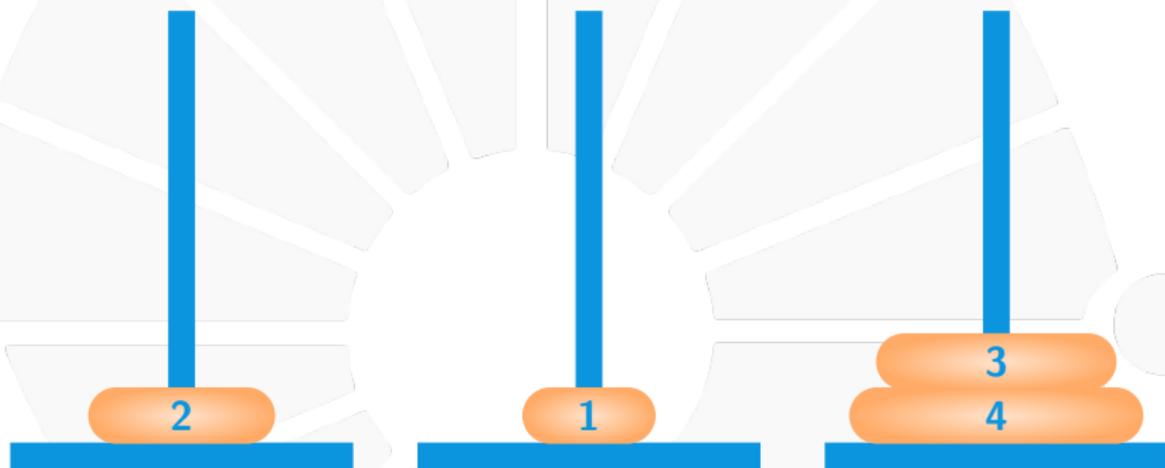
Torres de Hanoi – 4 discos



Mover o disco da torre 2 para a 3.



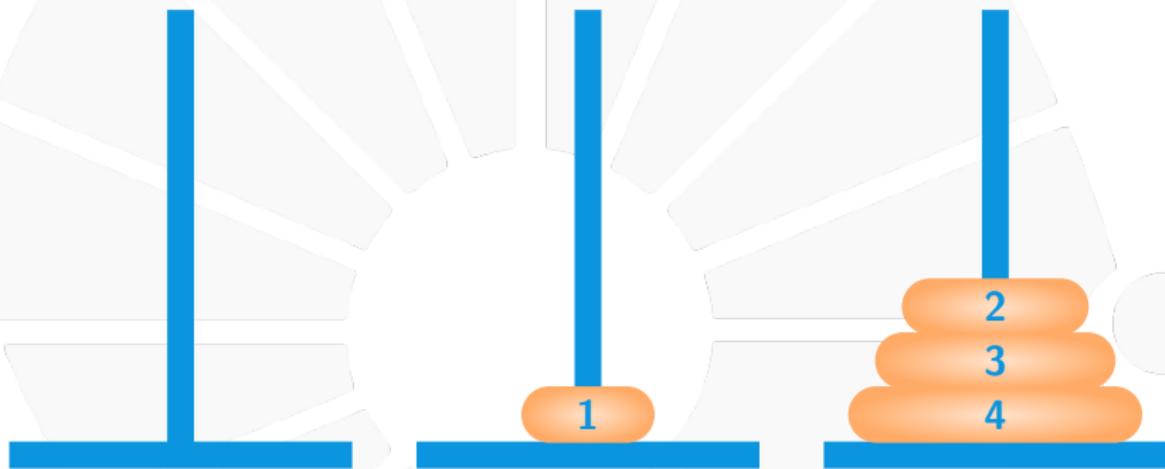
Torres de Hanoi – 4 discos



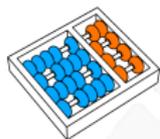
Mover o disco da torre 1 para a 2.



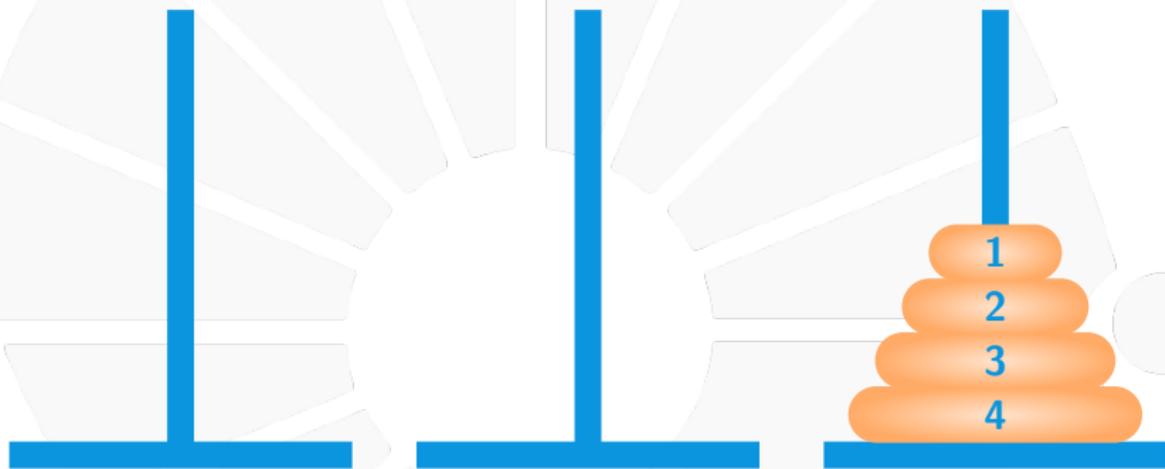
Torres de Hanoi – 4 discos



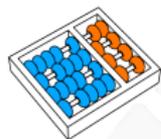
Mover o disco da torre 1 para a 3.



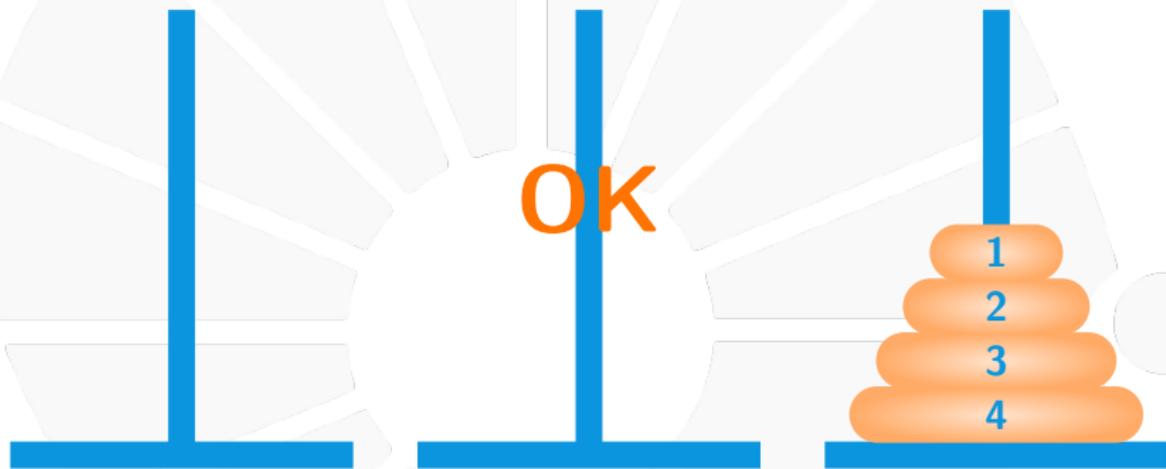
Torres de Hanoi – 4 discos

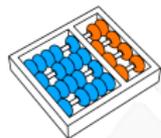


Mover o disco da torre 2 para a 3.

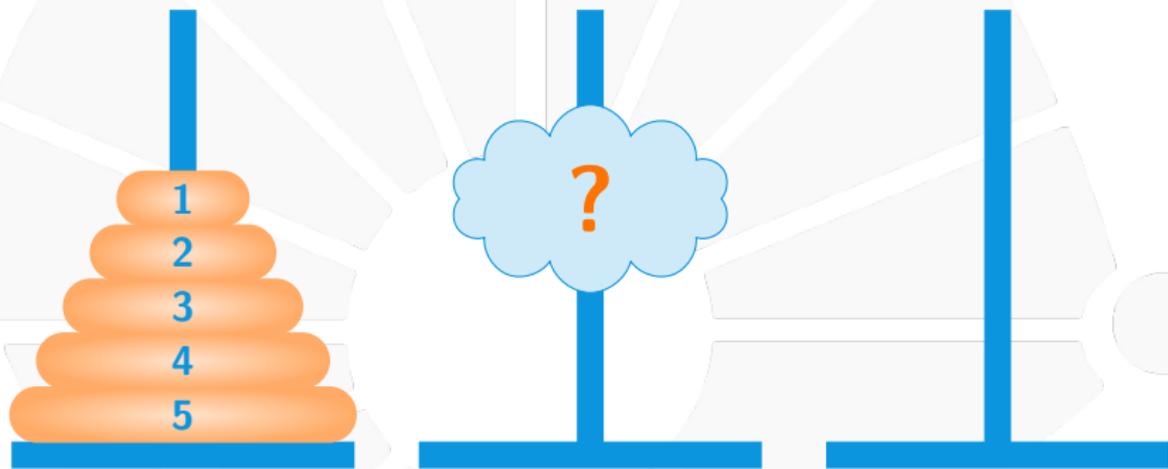


Torres de Hanoi – 4 discos





Torres de Hanoi – 5 discos





Torres de Hanoi

Suponha que desejamo levar n discos da torre i até a j , usando a torre k como auxiliar:

- ▶ **Base:** Solucionar o problema sem discos é trivial não precisamos fazer nada.
- ▶ **Geral:** Se houver mais $n \geq 1$ discos:
 - ▶ O disco n deve ser colocado na torre j antes dos outros.
 - ▶ Para isso, primeiro os $n-1$ discos menores devem ser movimentados da torre i até a k .
 - ▶ Depois o disco n pode ser movido de i para j .
 - ▶ Finalmente, os $n-1$ discos menores devem ser ir da torre k até a j .

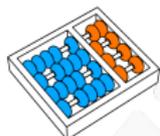


Torres de Hanoi

```
1 def hanoi(n, i, j, k):
2     if n >= 1:
3         hanoi(n - 1, i, k, j)
4         print('Mover o disco', n,
5             'da torre', i, 'para a', j)
6         hanoi(n - 1, k, j, i)
```



ENUMERANDO



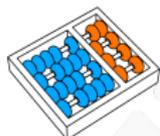
Enumerando

Recursão pode ser útil também para enumerar:

- ▶ Gerar todas as possíveis senhas de acordo com uma regra.
- ▶ Gerar todas as permutações de elementos.
- ▶ Gerar todos os subconjuntos de um conjunto.
- ▶ Etc...

Para tanto, precisamos construir a informação passo a passo:

- ▶ Armazenando ela em alguma estrutura de dados.



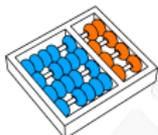
Exemplo: Sequências

Como imprimir todas as sequências de tamanho k de números entre 1 e n ?

Exemplo para $k=3$ e $n=4$:

111	131	211	231	311	331	411	431
112	132	212	232	312	332	412	432
113	133	213	233	313	333	413	433
114	134	214	234	314	334	414	434
121	141	221	241	321	341	421	441
122	142	222	242	322	342	422	442
123	143	223	243	323	343	423	443
124	144	224	244	324	344	424	444

Toda sequência que começa com i é seguida de uma sequência de tamanho $k-1$ de números entre 1 e n .



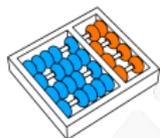
Exemplo: Sequências

- ▶ Armazenamos o prefixo da sequência que estamos construindo.
- ▶ Completamos com todos os possíveis sufixos recursivamente.

Recursão:

- ▶ **Base:** Se o tamanho do prefixo for k , imprime a sequência.
- ▶ **Geral:** Para cada $i \in \{1, \dots, n\}$, gere todas as sequências de $n - 1$ dígitos colocando i na frente.

```
1 def sequencias_rec(mem, k, n):
2     if len(mem) == k: # temos k números
3         print(' '.join(mem))
4     else:
5         for i in range(1, n + 1):
6             mem.append(str(i)) # coloca o número no final do prefixo
7             sequencias_rec(mem, k, n)
8             mem.pop() # retira o número do final do prefixo
9
10 def sequencias(k, n):
11     mem = []
12     sequencias_rec(mem, k, n)
```



Exemplo: Sequências sem repetições

E se quisermos imprimir todas as sequências de tamanho k de números entre 1 e n sem repetições?

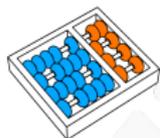
Primeiro algoritmo:

- ▶ Já temos um algoritmo que gera todas as sequências com repetições.
- ▶ Testar se uma sequência tem repetição é fácil.
- ▶ Basta imprimir as sequências que passarem no teste!



Exemplo: Sequências sem repetições

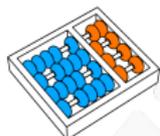
```
1 def tem_repeticao(mem):
2     for i in rang(len(mem)):
3         for j in range(i + 1, len(mem)):
4             if mem[i] == mem[j]: # verifica se mem[i] aparece depois
5                 return True
6     return False
7
8 def sequencias_rec(mem, k, n):
9     if len(mem) == k: # temos k números
10        if not tem_repeticao(mem): # imprime se não tiver repetições
11            print(' '.join(mem))
12        else:
13            for i in range(1, n + 1):
14                mem.append(str(i)) # coloca o número no final do prefixo
15                sequencias_rec(mem, k, n)
16                mem.pop() # retira o número do final do prefixo
17
18 def sequencias(k, n):
19     mem = []
20     sequencias_rec(mem, k, n)
```



Exemplo: Sequências sem repetições

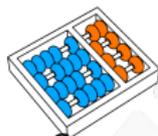
Outra solução:

- ▶ Ter uma lista de n valores indicando se o número i foi já usado.
- ▶ Somente adicionar o número (e fazer o chamado recursivo) se ele não estiver usado.



Exemplo: Sequências sem repetições

```
1 def sequencias_rec(mem, k, n, usados):
2     if len(mem) == k: # temos k números
3         print(' '.join(mem))
4     else:
5         for i in range(1, n + 1):
6             if not usados[i]: # adiciona se o número não foi usado
7                 mem.append(str(i)) # coloca o número no final
8                 usados[i] = True # indica que agora está usado
9                 sequencias_rec(mem, k, n, usados)
10                mem.pop() # retira o número do final do prefixo
11                usados[i] = False # o número não está mais na memória
12
13 def sequencias(k, n):
14     mem = []
15     usados = [False] * n # inicialmente nenhum número é usado
16     sequencias_rec(mem, k, n, usados)
```



Exemplo: Senhas numéricas

Como gerar todas as senhas numéricas de n dígitos?

- ▶ **Base:** Se $n = 0$, não temos nada a fazer.
- ▶ **Geral:** Para cada $i \in \{0, \dots, 9\}$, gere todas as senhas de $n - 1$ dígitos colocando i na frente.

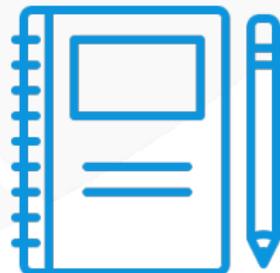
```
1 def senhas_rec(mem, n):
2     if len(mem) == n: # temos n digitos
3         print(''.join(mem), end=' ')
4     else:
5         for i in range(10):
6             mem.append(str(i)) # coloca o dígito na memória
7             senhas_rec(mem, n)
8             mem.pop() # retira o dígito da memória
9
10 def senhas(n):
11     mem = []
12     senhas_rec(mem, n)
```

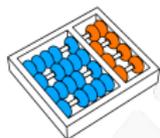


Sobre enumerar



Vamos fazer alguns exercícios?



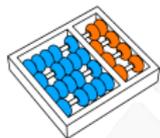


Exercícios

1. Faça uma função recursiva que, dado um inteiro positivo n , imprime todas as permutações de elementos de 1 a n .
2. Faça uma função recursiva que dado um inteiro n , imprime todas as combinações com n pares de parênteses balanceados.
3. Faça uma função recursiva que, dada uma lista l e um inteiro positivo r , imprime todas as combinações de r elementos de l .



BACKTRACKING



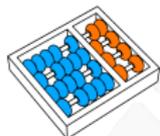
Backtracking - Retrocesso

Resolver um problema de forma recursiva, podendo tomar decisões erradas:

- ▶ Nesse caso, escolhemos outra decisão.

Construímos soluções passo-a-passo, **retrocedendo** se a solução parcial atual não é válida:

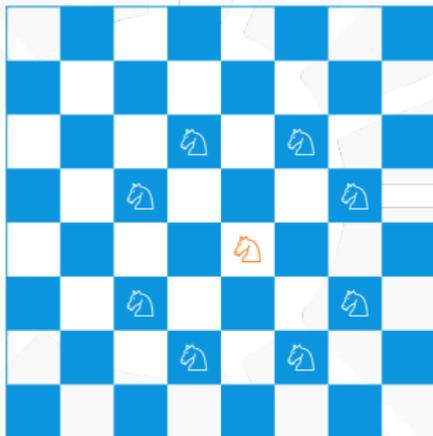
- ▶ Começamos com uma solução parcial vazia.
- ▶ Enquanto for possível, adicionamos um elemento à solução parcial.
- ▶ Se encontrarmos uma solução completa, terminamos.
- ▶ Se não é possível adicionar mais nenhum elemento à solução parcial, **retrocedemos**:
 - ▶ Removemos um ou mais elementos da solução parcial.
 - ▶ Tomamos decisões diferentes das que foram tomadas.



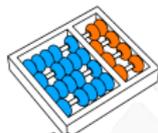
Exemplo: Passeio do cavalo

Movimento do cavalo no xadrez - formato de L:

- ▶ Dois quadrados horizontalmente e um verticalmente, ou
- ▶ dois quadrados verticalmente e um horizontalmente.



Dado um tabuleiro de xadrez $n \times n$ e uma posição (x, y) do tabuleiro queremos encontrar um passeio de um cavalo que visite cada casa exatamente uma vez.



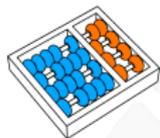
Exemplo: Passeio do cavalo

Uma matriz m armazenará os movimentos do cavalo:

- ▶ $m[l][c]=0$ se a posição (l,c) não foi visitada ainda.
- ▶ $m[l][c]=i>0$ se a posição (l,c) foi visitada no passo i .

52	47	56	45	54	5	22	13
57	44	53	4	23	14	25	6
48	51	46	55	26	21	12	15
43	58	3	50	41	24	7	20
36	49	42	27	62	11	16	29
59	2	37	40	33	28	19	8
38	35	32	61	10	63	30	17
1	60	39	34	31	18	9	64

- ▶ **Base:** Última posição visitada recebeu $n \times n$.
- ▶ **Geral:** Para cada possível movimento, realiza o movimento e verifica se soluciona (recursivamente), em caso contrário, retrocede.

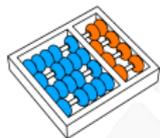


Exemplo: Passeio do cavalo

```

1  MOVS = ((2,1),(1,2),(-1,2),(-2,1),(-2,-1),(-1,-2),(1,-2),(2,-1))
2
3  def passeio_cavalo_rec(m, l, c):
4      if m[l][c] == len(m) * len(m):
5          return True
6      for mov in MOVS:
7          i = l + mov[0]
8          j = c + mov[1]
9          if 0 <= i < len(m) and 0 <= j < len(m) and m[i][j] == 0:
10             m[i][j] = 1 + m[l][c] # faz o movimento
11             if passeio_cavalo_rec(m, i, j): # tenta resolver
12                 return True
13             m[i][j] = 0 # se não consegue resolver, retrocede
14     return False
15
16 def passeio_cavalo(n, x, y):
17     m = [[0] * n for _ in range(n)] # cria matriz de zero
18     m[x][y] = 1 # coloca 1 na posição que o cavalo começa
19     passeio_cavalo_rec(m, x, y)

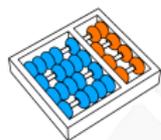
```



Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial.

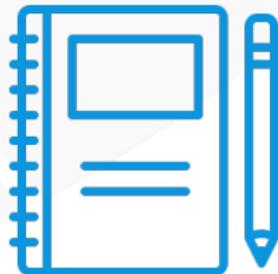
- ▶ Problemas de satisfação de restrições:
 - ▶ Encontrar uma solução que satisfaça as restrições.
 - ▶ Como o Sudoku, por exemplo.
- ▶ Problemas de Otimização Combinatória:
 - ▶ Conseguimos enumerar as soluções do problema.
 - ▶ Queremos encontrar uma de valor mínimo/máximo.

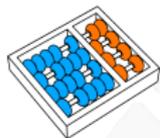


Sobre backtracking



Vamos fazer alguns exercícios?





Exercícios

1. Dado um jogo de Sudoku, encontre uma solução se ela existir. Considere que as casas não preenchidas tem o valor **0**.
2. Dado um inteiro **n**, determine se é possível colocar **n** rainhas (do xadrez) em um tabuleiro **$n \times n$** , sem que elas se ameacem.

RECURSÃO ... NOVAMENTE!?

MC102 - Algoritmos e
Programação de
Computadores

Santiago Valdés Ravelo
[https://ic.unicamp.br/~santiago/
ravelo@unicamp.br](https://ic.unicamp.br/~santiago/ravelo@unicamp.br)

06/25

25



UNICAMP

