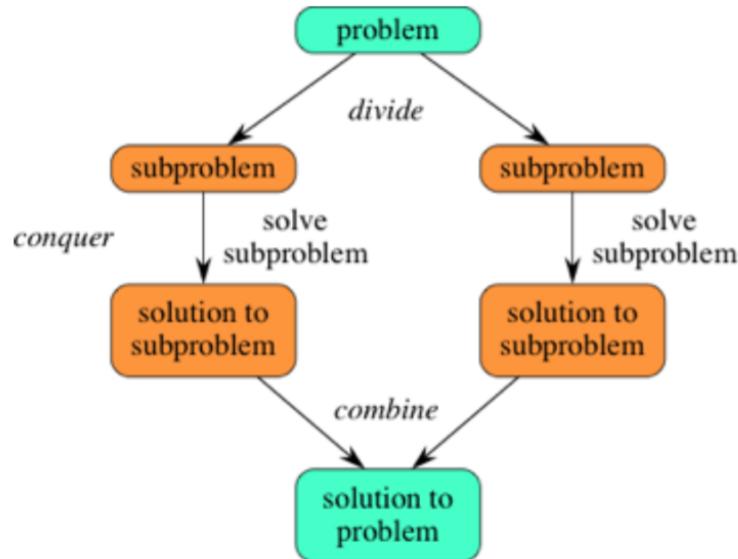


Divisão e Conquista



Algoritmo Genérico

```
DivisãoeConquista(x)
if x é pequeno ou simples then
    return resolver(x)
else
    decompor x em conjuntos menores x0, x1, ... xn
    for i ← 0 to n do
        yi ← DivisãoeConquista(xi)
    combinar todos os yi
    return y
```

Divisão: Dividir o problema original, em subproblemas menores

Conquista: Resolver cada subproblema recursivamente

Combinação: Combinar as soluções encontradas, compondo uma solução para o problema original

MergeSort (Algoritmo de Ordenação)

```

● ● ●
1 def merge(array, left, middle, right):
2     leftArray = array[left:middle + 1]
3     rightArray = array[middle + 1:right + 1]
4     leftArray.append(float('inf'))
5     rightArray.append(float('inf'))
6     i = 0
7     j = 0
8     for k in range(left, right + 1):
9         if leftArray[i] <= rightArray[j]:
10             array[k] = leftArray[i]
11             i += 1
12         else:
13             array[k] = rightArray[j]
14             j += 1
    
```

```

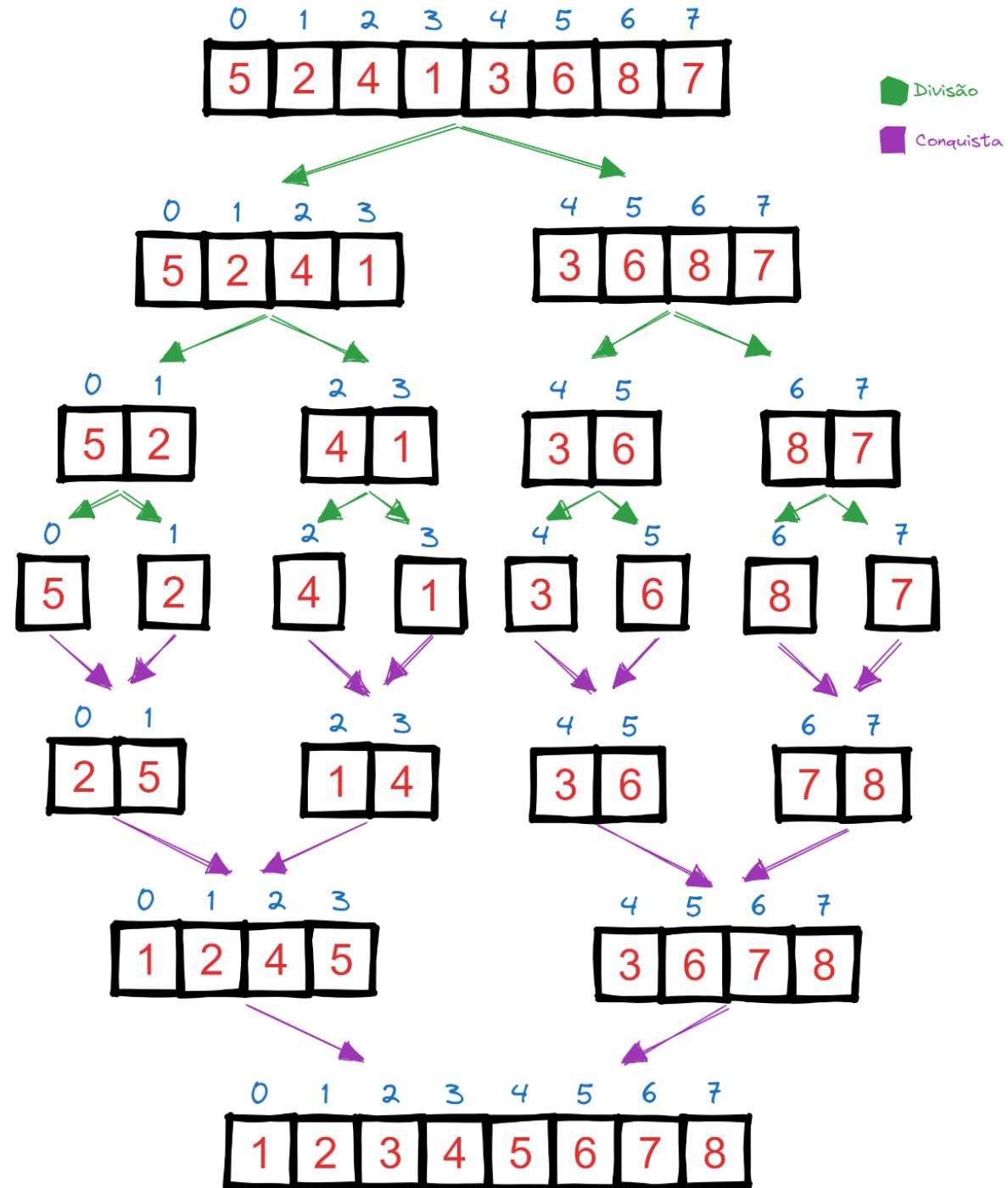
● ● ●
1 def mergeSort(array, left, right):
2     if left < right:
3         middle = (left + right) // 2
4         mergeSort(array, left, middle)
5         mergeSort(array, middle + 1, right)
6         merge(array, left, middle, right)
    
```

Relação de Recorrência

$$t(n) \leftarrow \begin{cases} t(1) = \Theta(1) \\ t(n) = 2t(n/2) + \Theta(n) \end{cases}$$

Custo

$$t(n) = \Theta(n * \log_2(n))$$



MergeSort (Algoritmo de Ordenação) - 2º Implementação

```
● ● ●  
1 def merge(array, leftArray, rightArray):  
2     leftArray.append(float('inf'))  
3     rightArray.append(float('inf'))  
4     i = 0  
5     j = 0  
6     for k in range(len(array)):  
7         if leftArray[i] <= rightArray[j]:  
8             array[k] = leftArray[i]  
9             i += 1  
10        else:  
11            array[k] = rightArray[j]  
12            j += 1  
13  
14 def mergeSort(array):  
15     if len(array) > 1:  
16         middle = len(array) // 2  
17         leftArray = array[:middle]  
18         rightArray = array[middle:]  
19         mergeSort(leftArray)  
20         mergeSort(rightArray)  
21         merge(array, leftArray, rightArray)
```

QuickSort (Algoritmo de Ordenação)

```

● ● ●
1 def partition(array, left, right):
2     pivot = array[right]
3     i = left - 1
4     for j in range(left, right):
5         if array[j] <= pivot:
6             i += 1
7             array[i], array[j] = array[j], array[i]
8     array[i + 1], array[right] = array[right], array[i + 1]
9     return i + 1

```

```

● ● ●
1 def quickSort(array, left, right):
2     if left < right:
3         pivot = partition(array, left, right)
4         quickSort(array, left, pivot - 1)
5         quickSort(array, pivot + 1, right)

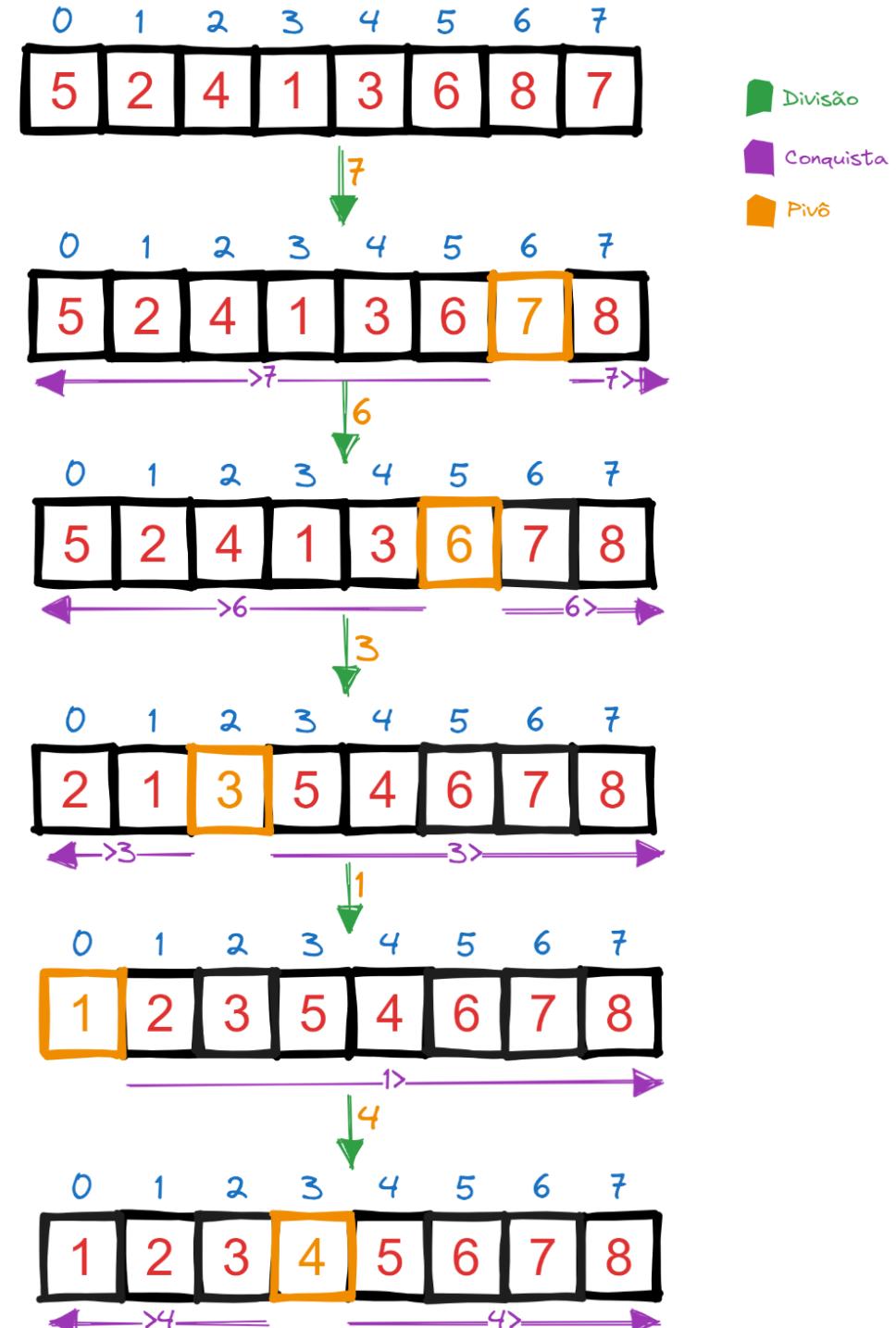
```

Relação de Recorrência

$$t(n) \leftarrow t(1) \\ t(n) = t(k) + t(n-k-1) + O(n)$$

k: é o número de elementos menores ou iguais ao pivô escolhido na função "partition".

Custo
médio/melhor caso
 $t(n) = \Theta(n * \log_2(n))$
pior caso
 $t(n) = \Theta(n^2)$



CountInversions

```

● ● ●
1 def countSplitInversions(array, leftArray, rightArray):
2     leftArray.append(float('inf'))
3     rightArray.append(float('inf'))
4     i = 0
5     j = 0
6     count = 0
7     for k in range(len(array)):
8         if leftArray[i] <= rightArray[j]:
9             array[k] = leftArray[i]
10            i += 1
11        else:
12            array[k] = rightArray[j]
13            j += 1
14            count += len(leftArray) - i - 1
15    return count

```

```

● ● ●
1 def countInversions(array):
2     if len(array) == 1:
3         return 0
4     else:
5         middle = len(array) // 2
6         leftArray = array[:middle]
7         rightArray = array[middle:]
8         leftCount = countInversions(leftArray)
9         rightCount = countInversions(rightArray)
10        splitCount = countSplitInversions(array, leftArray, rightArray)
11    return leftCount + rightCount + splitCount

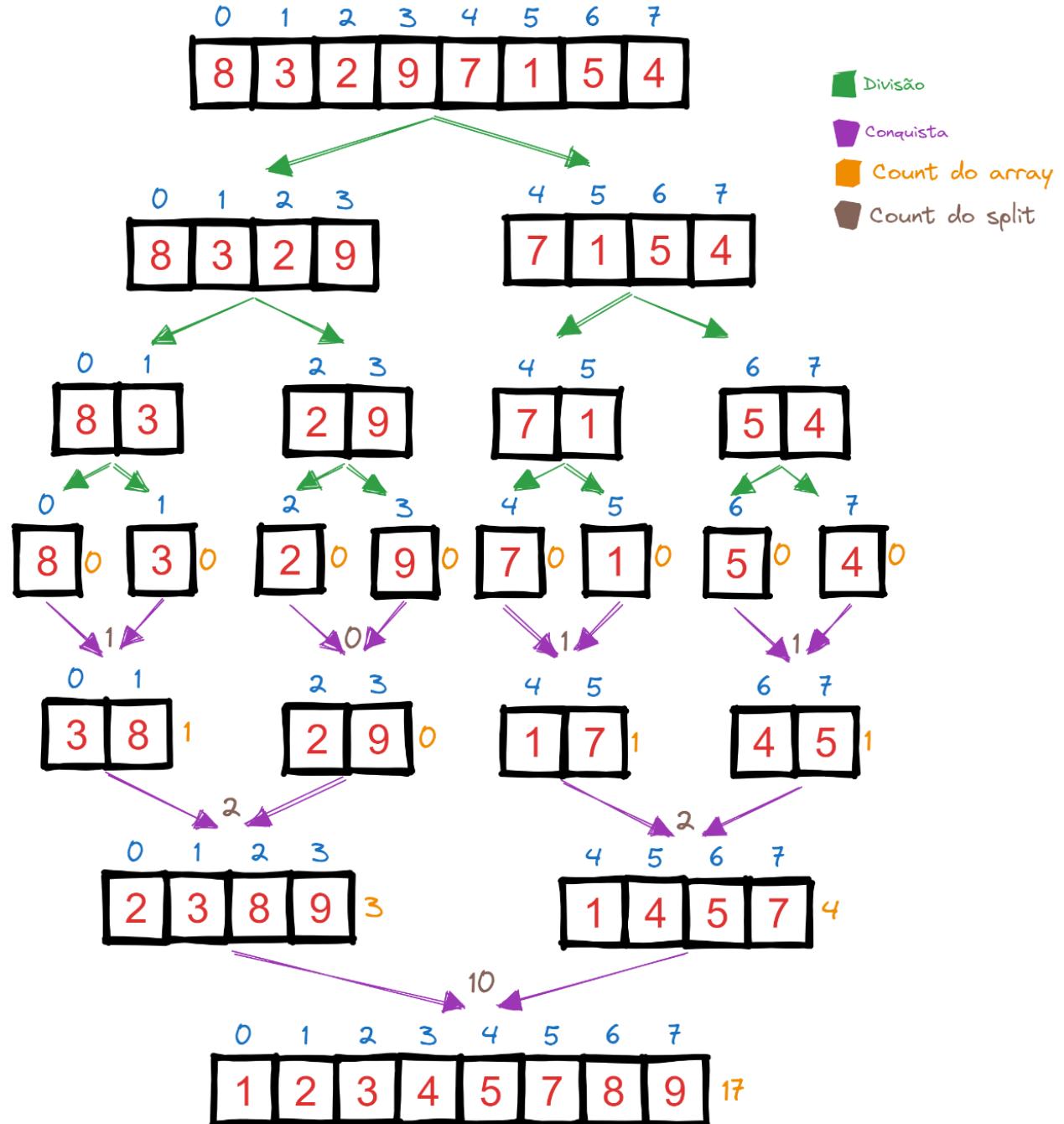
```

Relação de Recorrência

$$t(n) \leftarrow \begin{cases} t(1) = \Theta(1) \\ t(n) = 2t(n/2) + \Theta(n) \end{cases}$$

Custo

$$t(n) = \Theta(n * \log_2(n))$$

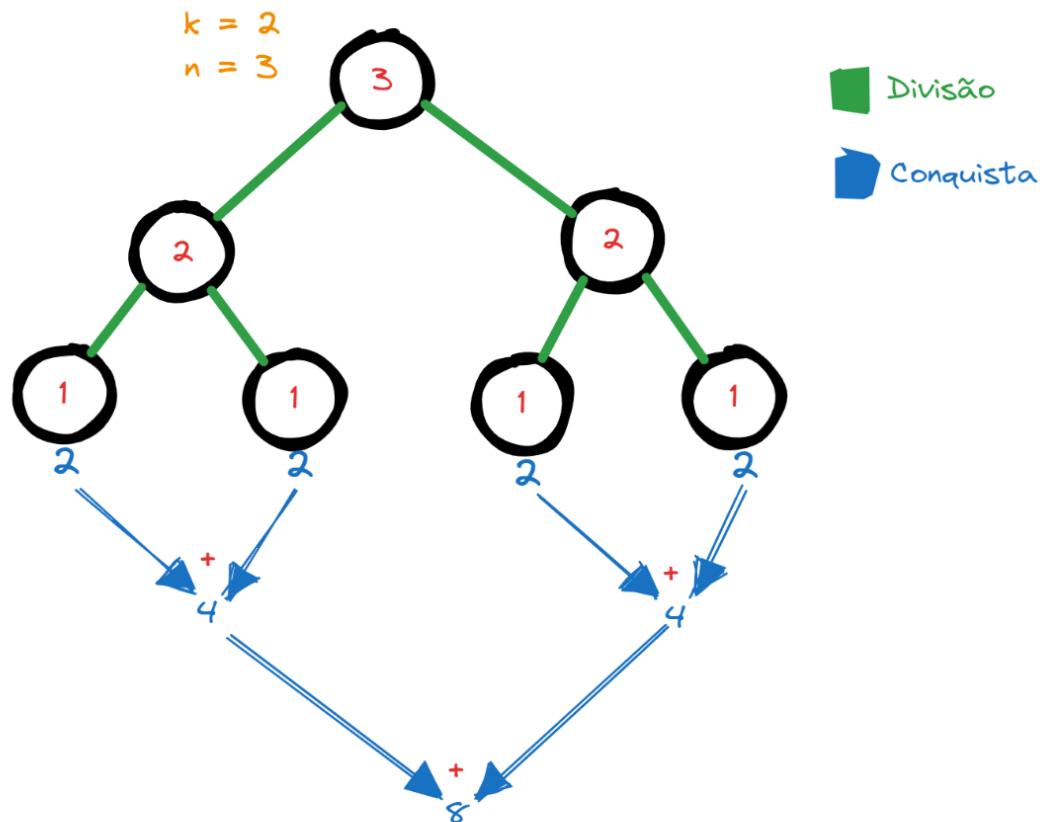


Potency (Calcular Potência)

```

● ● ●
1 def potency(k, n):
2     if n == 0:
3         return 1
4     elif n == 1:
5         return k
6     else:
7         left = potency(k, n - 1)
8         right = potency(k, n - 1)
9         return left + right

```



Relação de Recorrência

$$t(n) \begin{cases} t(1) = \Theta(1) \\ t(n) = 2t(n-1) + 1 \end{cases}$$

Custo

$$t(n) = \Theta(2^n)$$

Exemplo: cálculo de x^n

- ♦ Resolução iterativa com n multiplicações: $T(n) = O(n)$
- ♦ Resolução mais eficiente, com divisão e conquista:

$$x^n = \begin{cases} 1, & \text{se } n=0 \\ x, & \text{se } n=1 \\ x^{\frac{n}{2}} \times x^{\frac{n}{2}}, & \text{se } n \text{ par} > 1 \\ x \times x^{\frac{n-1}{2}} \times x^{\frac{n-1}{2}}, & \text{se } n \text{ ímpar} > 1 \end{cases}$$

```

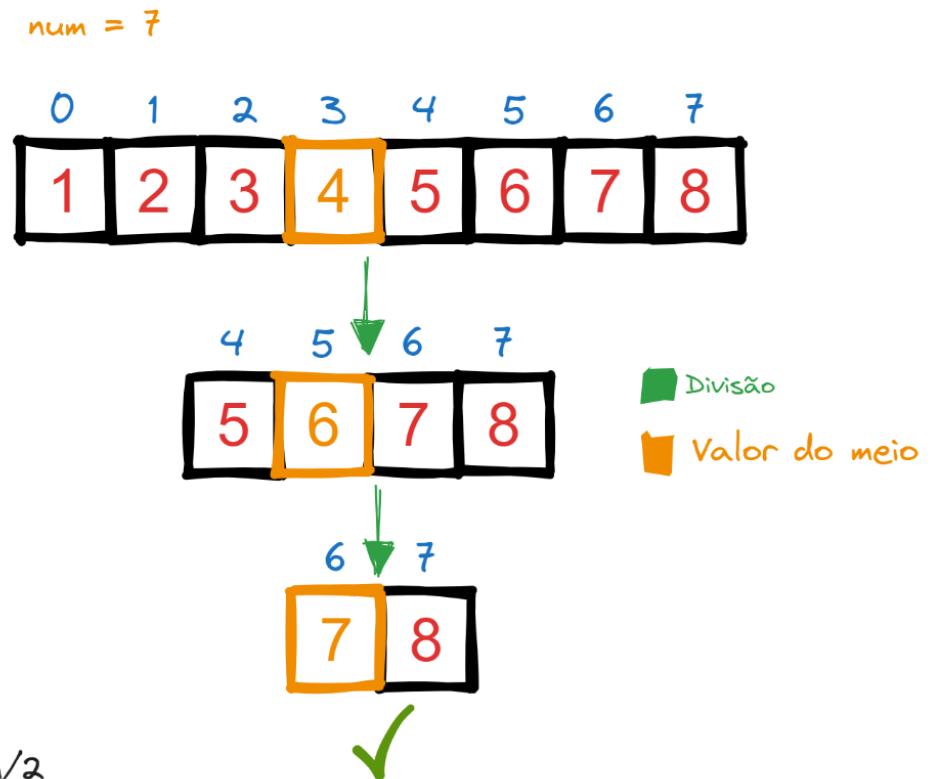
double power(double x, int n) {
    if (n == 0) return 1;
    if (n == 1) return x;
    double p = power(x, n / 2);
    if (n % 2 == 0) return p * p;
    else return x * p * p;
}

```

- ♦ Divisão em subproblemas iguais, junção em tempo $O(1)$
- ♦ N° de multiplicações reduzido para aprox. $\log_2 n$
- ♦ $T(n) = O(\log n)$ mas $S(n) = O(\log n)$ (espaço)

BinarySearch (Busca index de elemento)

```
●●●  
1 def binarySearch(array, left, right, num):  
2     if left > right:  
3         return -1  
4     else:  
5         middle = (left + right) // 2  
6         if array[middle] == num:  
7             return middle  
8         elif array[middle] > num:  
9             return binarySearch(array, left, middle - 1, num)  
10        else:  
11            return binarySearch(array, middle + 1, right, num)
```



- * Seja S uma sequência ordenada de n elementos, e sx um elemento que se pretende procurar dentro de S.
- * Se sx é o elemento médio, então retorna-se sx!
- * Senão:
 - * **Dividir:** divide-se S em duas sequências, S1 e S2, com $n/2$ elementos; se sx < que o elemento médio, escolhe-se S1 para continuar; se sx > que o elemento médio, escolhe-se S2 para continuar.
 - * **Conquistar:** tenta-se resolver a subsequência para determinar se sx está presente.
 - * Obtém-se a solução para a sequência a partir da solução do problema para as subsequências!

Relação de Recorrência

$$t(n) \begin{cases} t(1) = \Theta(1) \\ t(n) = t(n/2) + 1 \end{cases}$$

Custo
 $t(n) = \Theta(\log_2(n))$

Alguns Algoritmos Extras

encontrar o maior elemento de um array

```
● ● ●  
1 def maxim(array):  
2     if len(array) = 1:  
3         return array[0]  
4     else:  
5         middle = len(array) // 2  
6         left = maxim(array[:middle])  
7         right = maxim(array[middle:])  
8         return max(left, right)
```

$$t(n) = 2t(n/2) + 1$$

soma dos elementos de um array

```
● ● ●  
1 def sumArray(array):  
2     if len(array) = 1:  
3         return array[0]  
4     else:  
5         middle = len(array) // 2  
6         left = sumArray(array[:middle])  
7         right = sumArray(array[middle:])  
8         return left + right
```

$$t(n) = 2t(n/2) + 1$$

multiplicação de grandes inteiros

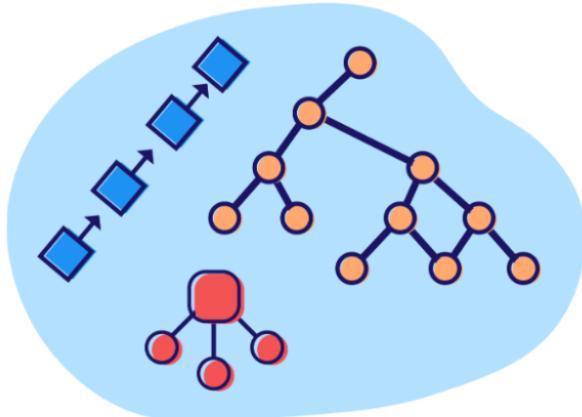
```
● ● ●  
1 def multiply(x, y):  
2     # Verificar se os números são de um único dígito  
3     if x < 10 or y < 10:  
4         return x * y  
5  
6     # Determinar o tamanho dos números  
7     size = max(len(str(x)), len(str(y)))  
8     half_size = size // 2  
9  
10    # Dividir os números em partes menores  
11    high1, low1 = divmod(x, 10**half_size)  
12    high2, low2 = divmod(y, 10**half_size)  
13  
14    # Calcular as multiplicações recursivamente  
15    z0 = multiply(low1, low2)  
16    z1 = multiply((low1 + high1), (low2 + high2))  
17    z2 = multiply(high1, high2)  
18  
19    # Calcular o resultado final usando as fórmulas da divisão e conquista  
20    return (z2 * 10**(2*half_size)) + ((z1 - z2 - z0) * 10**half_size) + z0
```

$$t(n) = 3t(n/2) + 1$$

Programação Dinâmica

Memorização para armazenar resultados de subproblemas

Aplicada quando recursão produz repetição dos mesmos subproblemas



Desenvolvendo uma solução

1. Caracterizar a estrutura de uma solução ótima
2. Definir recursivamente o valor de uma solução ótima através de uma relação de recorrência
3. Calcular o valor de uma solução ótima em um processo bottom-up
4. Construir uma solução ótima a partir de informações calculadas

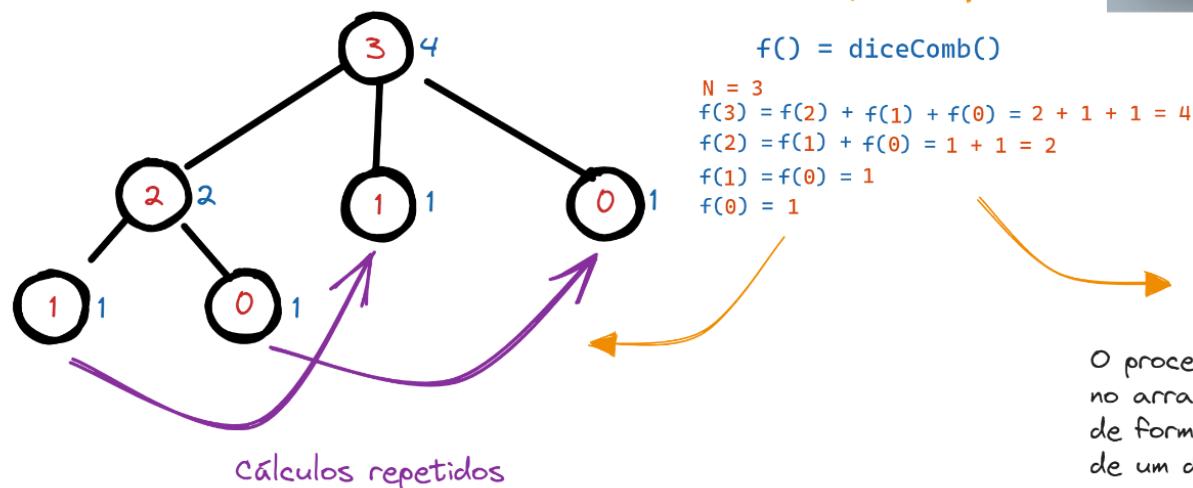
A - Dice Combinations

backtracking

```

● ● ●
1 def diceComb(n, mod):
2     if n == 1 or n == 0:
3         return 1
4     else:
5         count = 0
6         for i in range(1, n + 1):
7             if i <= 6:
8                 count += diceComb(n - i, mod) % mod
9     return count.
10
11 n = int(input())
12 mod = 10**9 + 7
13 print(diceComb(n, mod))

```

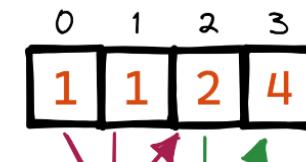


Programação Dinâmica

```

● ● ●
1 def diceComb(n, mod):
2     array = [0] * (n + 1)
3     array[0], array[1] = 1, 1
4
5     for i in range(2, n + 1):
6         for j in range(1, 7):
7             if i >= j:
8                 array[i] = (array[i] + array[i - j]) % mod
9             else:
10                break
11
12     return array[n]
13
14 n = int(input())
15 mod = 10**9 + 7
16 print(diceComb(n, mod))

```



O processo consiste em ir preenchendo cada posição no array onde a posição (i) representa o número de formas diferentes de representar o valor com n jogadas de um dado de 6 faces

Preenchendo do inicio até o valor (n) garantimos a utilização dos cálculos já realizados, por exemplo, ao calcular para 3 precisamos somar o do 0,1 e 2 que já foram calculados

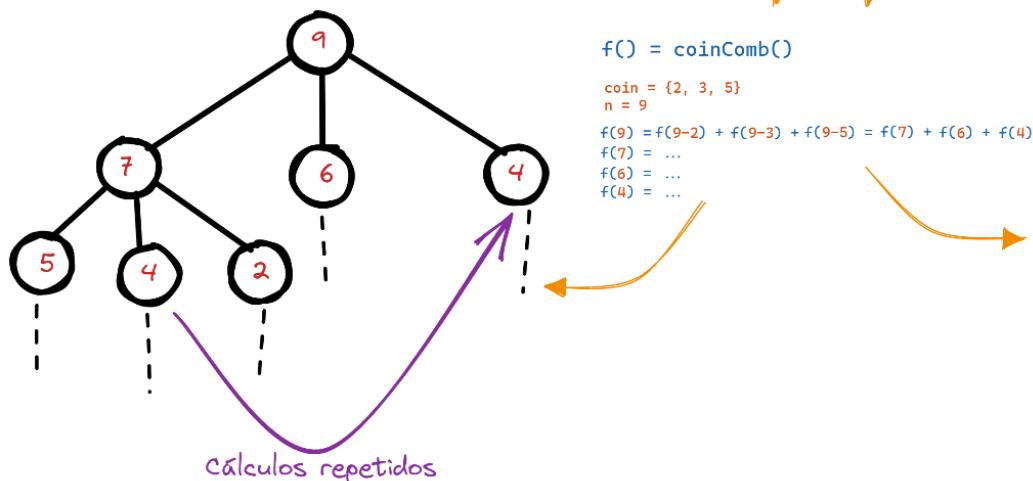
B – Coin Combinations

backtracking

```

1 def coinComb(coins, value):
2     if value == 0:
3         return 1
4     else:
5         count = 0
6         for coin in coins:
7             newValue = value - coin
8             if newValue >= 0:
9                 count += coinComb(coins, value - coin)
10    return count

```



Programação Dinâmica

```

1 def coinComb(coins, n, mod):
2     memo = [0] * (n + 1)
3     memo[0] = 1
4
5     for value in range(1, n + 1):
6         for coin in coins:
7             newValue = value - coin
8             if newValue >= 0:
9                 memo[value] = (memo[value] + memo[newValue]) % mod
10
11    return memo[n]

```

0	1	2	3	4	5	6	7	8	9
1	0	1	1	1	3	2	5	6	8

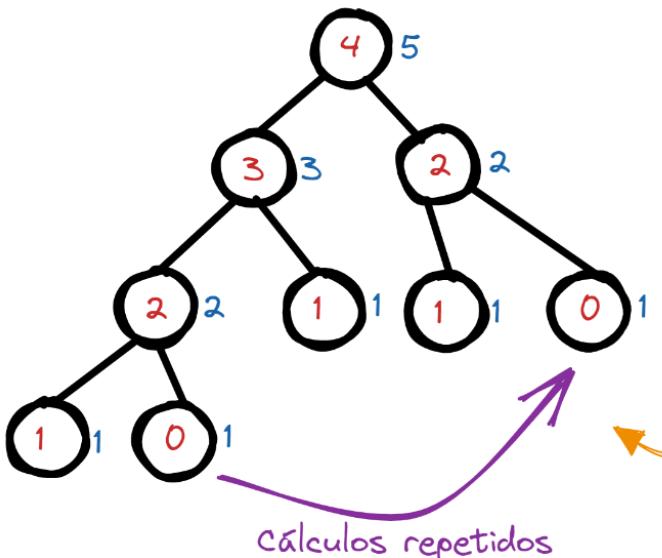
O processo consiste em ir preenchendo cada posição no array onde a posição (i) representa a contagem de formas diferentes de representar o valor com as moedas disponíveis

Preenchendo do início até o valor (9) garantimos a utilização dos cálculos já realizados, por exemplo, ao calcular para 5 precisamos somar o do 3, 2 e 0 que já foram calculados

Fibonacci

Divisão e Conquista

```
def fibonacci(n):
    if n == 0 or n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```



Programação Dinâmica

```
def fibonacciPD(n):
    memo = [0] * (n + 1)
    memo[0], memo[1] = 1, 1
    for i in range(2, n + 1):
        memo[i] = memo[i - 1] + memo[i - 2]
    return memo[n]
```

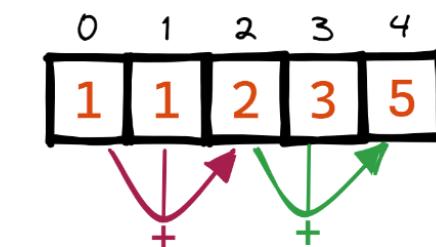
Relação de Recorrência

$$t(n) \leftarrow \begin{aligned} t(0) &= 0(1) \\ t(1) &= 0(1) \\ t(n) &= t(n-1) + t(n-2) + 0(1) \end{aligned}$$

$f() = \text{fibonacci}()$

$N = 4$

$$\begin{aligned} f(4) &= f(4-1) + f(4-2) = f(3) + f(2) = 3 + 2 = 5 \\ f(3) &= f(3-1) + f(3-2) = f(2) + f(1) = 2 + 1 = 3 \\ f(2) &= f(2-1) + f(2-2) = f(1) + f(0) = 1 + 1 = 2 \\ f(1) &= 1 \\ f(0) &= 1 \end{aligned}$$



Coeficiente Binomial

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

$$\binom{n}{k} = \begin{cases} 1 & \text{se } k=0 \text{ ou } k=n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 0 & \text{gg outra situação} \end{cases}$$

f() = binomial()

N = 4
 K = 3
 $f(4, 3) = f(3, 2) + f(3, 3)$
 $f(3, 2) = \dots$
 $f(3, 3) = \dots$

Dica:

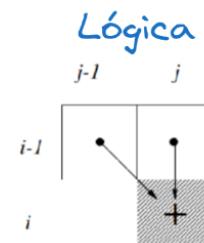
```
matrix = [[default_value] * colunas for _ in range(linhas)]
```

Usando divisão e conquista

```
Binomial(n,k)
if k == 0 or k == n
    return 1
    escolher o item
    não escolher o item
return Binomial(n-1,k-1) + Binomial(n-1,k)
```

```
● ● ●
1 def binomial(n ,k):
2     memo = [[0] * (k + 1) for _ in range(n + 1)]
3
4     for i in range(n + 1):
5         for j in range(k + 1):
6             if j == 0 or i == j:
7                 memo[i][j] = 1
8             elif 0 < j < i:
9                 memo[i][j] = memo[i - 1][j - 1] + memo[i - 1][j]
10
11     return memo[n][k]
```

		K:j			
		0	1	2	3
N:i	0	1	0	0	0
	1	1	1	0	0
2	1	2	1	0	
3	1	3	3	1	
4	1	4	6	4	



Problema do Troco

$$C(i, j) = \begin{cases} 0, & \text{se } i = 0 \text{ ou } j = 0 \\ j, & \text{se } di = 1 \\ C(i-1, j), & \text{se } di > j \\ \min(C(i-1, j), C(i, j-di) + 1), & \text{caso contrário} \end{cases}$$

não inclui moeda i

inclui moeda i

```
f() = minCoins()
N = 8
COINS = {1,4,6}
f({1,4,6}, 8) = min(f({1,4}, 8), f({1,4,6}, 2) + 1)
f({1,4}, 8) = ...
f({1,4,6}, 2) = ...
```

Armazenando as moedas no lugar da quantidade
Saída: [4,4]

```
def minCoins(coins, n):
    memo = [[[] for _ in range(n+1)] for _ in range(len(coins)+1)]
    for i in range(len(coins)+1):
        for j in range(n+1):
            coin = coins[i-1]
            if i == 0 or j == 0:
                memo[i][j] = []
            elif coin == 1:
                memo[i][j] = memo[i][j - coin] + [coin]
            elif coin > j:
                memo[i][j] = memo[i-1][j]
            else:
                if len(memo[i-1][j]) <= len(memo[i][j - coin] + [coin]):
                    memo[i][j] = memo[i-1][j]
                else:
                    memo[i][j] = memo[i][j - coin] + [coin]
    return memo[-1][-1]
```

```
def minCoins(coins, n):
    memo = [[0] * (n+1) for _ in range(len(coins)+1)]
    for i in range(len(coins)+1):
        for j in range(n+1):
            coin = coins[i-1]
            if i == 0 or j == 0:
                memo[i][j] = 0
            elif coin == 1:
                memo[i][j] = j
            elif coin > j:
                memo[i][j] = memo[i-1][j]
            else:
                memo[i][j] = min(memo[i-1][j], memo[i][j - coin] + 1)
    return memo[-1][-1]
```

	$N:j$								
	0	1	2	3	4	5	6	7	8
[]	0	0	0	0	1	0	0	0	0
[1]	0	1	2	3	4	5	6	7	8
[1,4]	0	1	2	3	1	2	3	4	2
[1,4,6]	0	1	2	3	1	2	1	2	2

Nesta versão modificada, a matriz memo é inicializada com listas vazias em vez de zeros. Em cada posição da matriz, em vez de armazenar um número inteiro, armazenamos uma lista que representa a combinação de moedas para alcançar aquele valor de troco.

Durante o preenchimento da matriz, sempre escolhemos a combinação com o menor número de moedas. Se houver uma combinação com o mesmo número de moedas, selecionamos a que contém menos moedas de maior valor.

Problema da Mochila

$$f(i, j) = \begin{cases} 0 & \Rightarrow i = 0 \text{ or } j = 0 \\ f(i-1, j) & \Rightarrow w_i > j \\ \max(f(i-1, j), f(i-1, j-w_i) + v_i) & \Rightarrow w \leq j \end{cases}$$

não leva o item diminui peso acrescenta valor
 leva o item

Obtendo os itens adicionados na mochila
Saída: [(4, 9), (1, 6)]

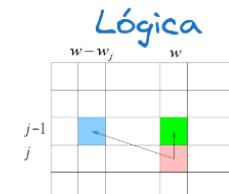
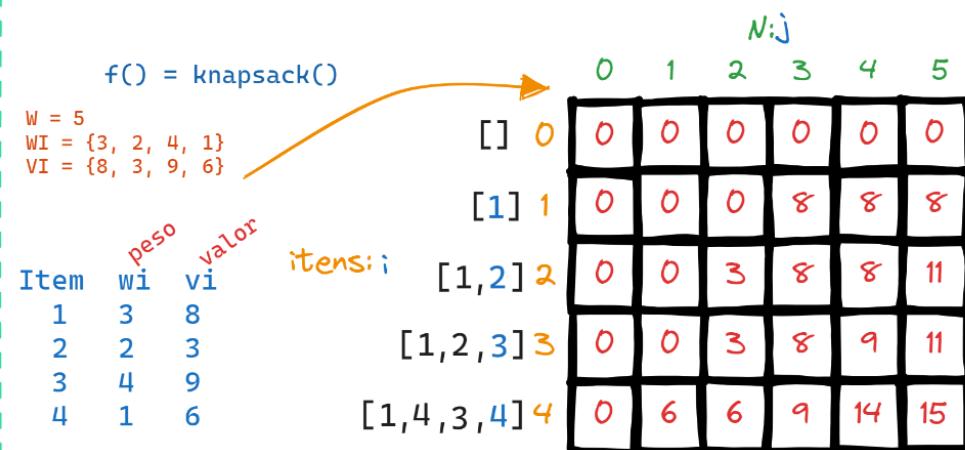
```

● ● ●
1 def getItems(wi, vi, memo):
2     items_added = []
3     n = len(wi)
4     i = n
5     j = len(memo[0]) - 1
6
7     while i > 0 and j > 0:
8         if memo[i][j] != memo[i - 1][j]:
9             items_added.append(i - 1)
10            j -= wi[i - 1]
11            i -= 1
12
13     # Inverte a lista para manter a ordem original dos itens
14     items_added.reverse()
15     items = [(wi[item], vi[item]) for item in items_added]
16
17     return items
  
```

Para obter os itens adicionados à mochila usando a matriz resultante do algoritmo, você pode percorrer a matriz de trás para frente, começando do último elemento $\text{memo}[-1][-1]$. Em cada passo, você verifica se o valor atual é diferente do valor acima dele na matriz. Se for diferente, significa que o item correspondente a essa posição foi adicionado à mochila.

```

● ● ●
1 def knapsack(w, wi, vi):
2     n = len(wi)
3     memo = [[0] * (w + 1) for _ in range(n + 1)]
4
5     for i in range(1, n + 1):
6         for j in range(1, w + 1):
7             if i == 0 or j == 0:
8                 memo[i][j] = 0
9             elif wi[i - 1] > j:
10                memo[i][j] = memo[i - 1][j]
11            else:
12                memo[i][j] = max(memo[i - 1][j], memo[i - 1][j - wi[i - 1]] + vi[i - 1])
13
14     return memo[-1][-1]
  
```



$$K(w, j) = \max \{K(w, j-1), K(w - w_i, j-1) + v_i\}$$

Alguns Algoritmos Extras

Máximo Subarray

Dada um vetor de n inteiros diferentes de zero, determine o intervalo de maior soma e retornar seu comprimento.

Exemplo:

entrada: [-2, 1, -3, 4, -1, 2, 1, -5, 4]

saída: 6

Maior subarray: [-2, 1, -3, 4, -1, 2, 1, -5, 4]

Solução usando o algoritmo de Kadane

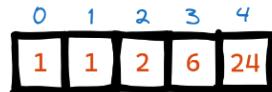
```
● ● ●  
1 def maximumSubarray(array):  
2     maxCurrent = maxGlobal = array[0]  
3  
4     for i in range(1, len(array)):  
5         maxCurrent = max(array[i], maxCurrent + array[i])  
6         maxGlobal = max(maxGlobal, maxCurrent)  
7  
8     return maxGlobal
```

Fatorial

Dada uma valor n , determinar o fatorial de n , sabendo que $\text{fatorial}(n) = n * \text{fatorial}(n - 1)$ e que fatorial de 1 ou 0 é sempre 1

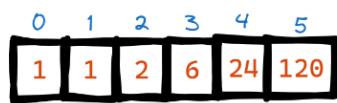
Exemplo:

entrada: 4



saída: 24

entrada: 5



saída: 120

```
● ● ●  
1 def fatorial(n):  
2     memo = [0] * (n + 1)  
3     memo[:2] = 1, 1  
4  
5     for i in range(2, n + 1):  
6         memo[i] = i * memo[i - 1]  
7  
8     return memo[-1]
```

Maior Subsequência Crescente (LIS)

Dada um vetor de n inteiros, é necessário determinar o comprimento da maior subsequência crescente. Note que a subsequência não necessariamente contígua.

Exemplo:

entrada: [-7, 10, 9, 2, 3, 8, 8, 1]

saída: 4

Maior subsequência: [-7, 10, 9, 2, 3, 8, 8, 1]

↓
[-7, 2, 3, 8]

```
● ● ●
1 def longestIncreasingSubsequence(array):
2     n = len(array)
3     memo = [1] * n
4
5     for i in range(1, n):
6         for j in range(i):
7             if array[i] > array[j] and memo[j] + 1 > memo[i]:
8                 memo[i] = memo[j] + 1
9
10    return max(memo)
```

Maior Subsequência Comum (LCS)

Dada uma sequência de dois vetores X e Y com comprimentos m e n , respectivamente, o objetivo é determinar o comprimento da maior subsequência comum.

Uma subsequência comum é uma subsequência que pode ser obtida a partir de ambas as sequências originais removendo-se zero ou mais elementos, mantendo-se a ordem relativa dos elementos restantes. É importante destacar que a subsequência não precisa ser contígua.

Exemplo:

entrada: [1, 2, 3, 2] e [2, 4, 3, 1, 2]

saída: 3

Maior subsequência: [1, 2, 3, 2] ↔ [2, 4, 3, 1, 2]

[2,3,2]

```
● ● ●
1 def longestCommonSubsequence(array1, array2):
2     m = len(array1)
3     n = len(array2)
4     memo = [[0] * (n + 1) for _ in range(m + 1)]
5
6     for i in range(1, m + 1):
7         for j in range(1, n + 1):
8             if array1[i - 1] == array2[j - 1]:
9                 memo[i][j] = memo[i-1][j-1] + 1
10            else:
11                memo[i][j] = max(memo[i-1][j], memo[i][j-1])
12
13    return memo[-1][-1]
```