



## Aula 1 - Programação Dinâmica

---

Módulo 3 - PrepTech Google

## Programação Dinâmica | Conceito

---

# Programação Dinâmica

## Conceito

A programação dinâmica é uma **técnica de otimização** de algoritmos que decompõe um problema complexo em subproblemas menores, sobrepostos e otimizados.

A solução para o problema original é então construída a partir das soluções dos subproblemas.



## Características principais:

**Subestrutura ótima:** A solução ótima de um problema pode ser construída a partir das soluções ótimas de seus subproblemas.

**Subproblemas sobrepostos:** Os mesmos subproblemas são resolvidos repetidamente.

**Memoização:** Os resultados dos subproblemas são armazenados para evitar recalcular os mesmos valores.

## Comparando com Divisão e Conquista:

Na **divisão e conquista**, resolvemos um problema maior decompondo ele em subproblemas menores que são independentes entre si, nunca se sobrepõem.

Já na programação dinâmica os subproblemas podem se sobrepor. Como resultado, podemos precisar recalcular o mesmo subproblema várias vezes. Para evitar isso, usamos **memoization** para salvar esses resultados já calculados.

## Programação Dinâmica | Primeiro Exemplo

---

# Programação Dinâmica

Prática no LeetCode - <https://leetcode.com/problems/best-time-to-buy-and-sell-stock>

## 121. Best Time to Buy and Sell Stock

Easy

Topics

Companies

You are given an array `prices` where `prices[i]` is the price of a given stock on the  $i^{\text{th}}$  day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return `0`.

### Example 1:

Input: `prices = [7,1,5,3,6,4]`

Output: `5`

Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit =  $6 - 1 = 5$ .

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

### Example 2:

Input: `prices = [7,6,4,3,1]`

Output: `0`

Explanation: In this case, no transactions are done and the max profit = `0`.

### Constraints:

- $1 \leq \text{prices.length} \leq 10^5$
- $0 \leq \text{prices}[i] \leq 10^4$



## Best Time to Buy and Sell Stock

Entendendo o problema

O que significa essa entrada?

7	1	5	3	6	4
---	---	---	---	---	---



## Best Time to Buy and Sell Stock

Entendendo o problema

O que significa essa entrada?

7	1	5	3	6	4
---	---	---	---	---	---

↑  
Preço no  
dia 1

↑  
Preço no  
dia 2

↑  
Preço no  
dia 3

↑  
Preço no  
dia 4

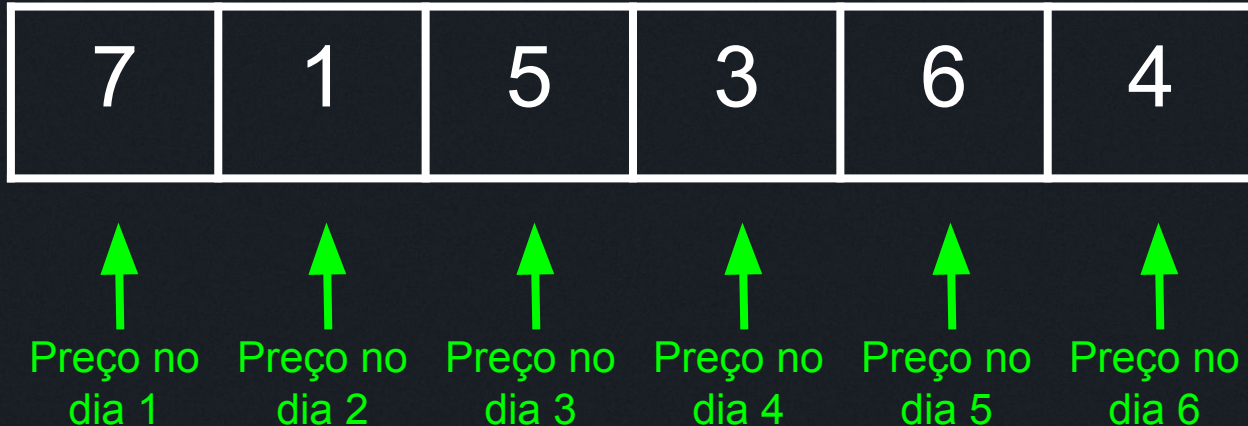
↑  
Preço no  
dia 5

↑  
Preço no  
dia 6

## Best Time to Buy and Sell Stock

Entendendo o problema

Qual seria o melhor dia para comprar de forma que eu ganhe o máximo possível vendendo em outro dia futuro?



## Best Time to Buy and Sell Stock

Entendendo o problema

Qual seria o melhor dia para comprar de forma que eu ganhe o máximo possível vendendo em outro dia futuro?

7	1	5	3	6	4
---	---	---	---	---	---



Dia da compra



Dia da venda

$$\text{Lucro obtido} = \$6 - \$1 = \$5$$

## Best Time to Buy and Sell Stock

Entendendo o problema

Pense em uma primeira solução

7	1	5	3	6	4
---	---	---	---	---	---

↑  
Preço no  
dia 1

↑  
Preço no  
dia 2

↑  
Preço no  
dia 3

↑  
Preço no  
dia 4

↑  
Preço no  
dia 5

↑  
Preço no  
dia 6



## Best Time to Buy and Sell Stock

Entendendo o problema

Pense em uma primeira solução

7	1	5	3	6	4
---	---	---	---	---	---

↑      ↑      ↑      ↑      ↑      ↑  
Preço no   Preço no   Preço no   Preço no   Preço no   Preço no  
dia 1      dia 2      dia 3      dia 4      dia 5      dia 6

Força bruta: podemos tentar todos os possíveis dias de compra e venda e ficar com o maior lucro

## Best Time to Buy and Sell Stock

Entendendo o problema

Implemente a força bruta!

7	1	5	3	6	4
---	---	---	---	---	---

- Inicie uma variável acumuladora **bestProfit** com zero.
- Use um laço que itera sobre todo o vetor **prices**, tentando todos os possíveis dias de compra.
- Use outro laço dentro desse primeiro que, para cada dia de compra, tenta todos os possíveis dias de venda no futuro.
- O lucro obtido é a diferença de preço entre o dia da venda e o dia da compra. Se esse lucro for maior que o **bestProfit**, atualiza para ele.
- No final, retorna o **bestProfit**, que será o maior lucro obtido.

## Best Time to Buy and Sell Stock

Entendendo o problema

Implemente a força bruta!

7	1	5	3	6	4
---	---	---	---	---	---

```
def maxProfit(self, prices):  
    bestProfit = 0  
    for buyingDay in range(len(prices)):  
        for sellingDay in range(buyingDay+1, len(prices)):  
            profit = prices[sellingDay]-prices[buyingDay]  
            bestProfit = max(bestProfit, profit)  
    return bestProfit
```

## Best Time to Buy and Sell Stock

Entendendo o problema

Implemente a força bruta!

7	1	5	3	6	4
---	---	---	---	---	---

- O que ocorre quando envia essa solução no LeetCode?



## Best Time to Buy and Sell Stock

Entendendo o problema

Implemente a força bruta!

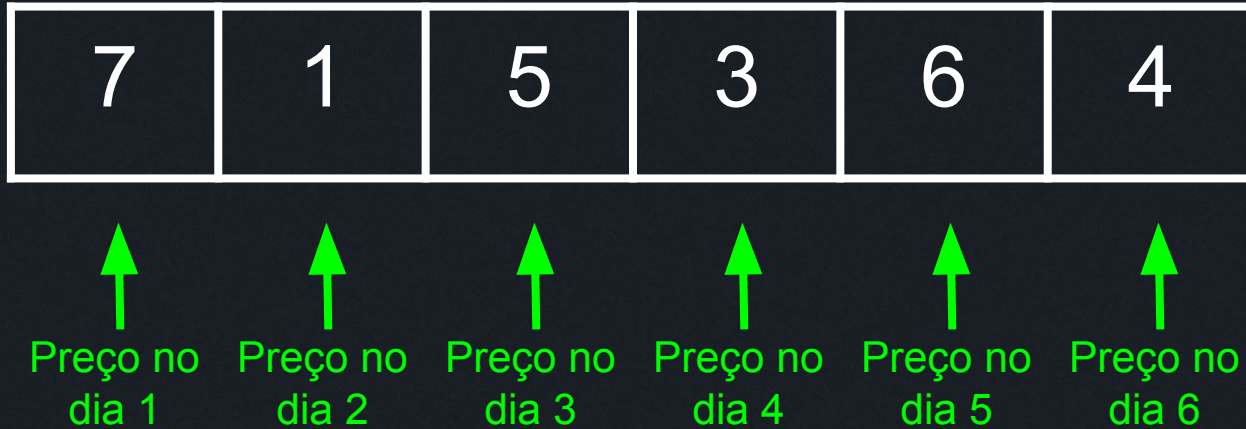
7	1	5	3	6	4
---	---	---	---	---	---

- O que ocorre quando envia essa solução no LeetCode?
- Por que deu TLE (Tempo Limite Excedido)?
  - A complexidade desse algoritmo é  $O(n^2)$ , onde  $n$  é o tamanho do vetor **prices**.
- Notaram o tamanho máximo do vetor prices no enunciado?
  - $1 \leq \text{prices.length} \leq 10^5$

# Best Time to Buy and Sell Stock

Entendendo o problema

Pense novamente

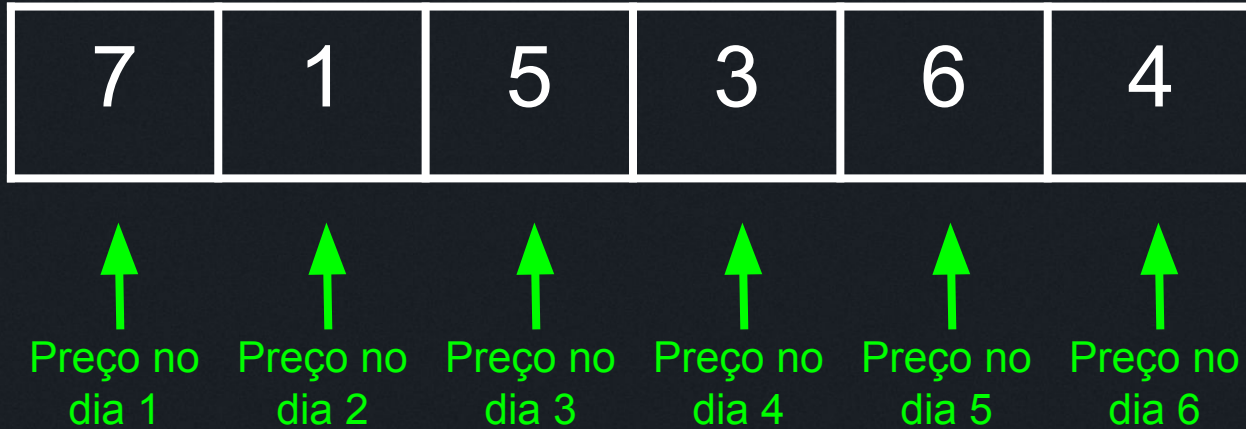


- Quando estou vendendo em um dia qualquer, qual é o melhor dia anterior para ter comprado?
- Qual é esse dia anterior que faz eu ter o maior lucro possível vendendo hoje?

# Best Time to Buy and Sell Stock

Entendendo o problema

Pense novamente



- Se tiver salvo o **menor valor** possível antes de hoje em uma variável, posso calcular o **maior lucro** de vender hoje sem precisar de um outro laço retornando todos os dias anteriores.

# Best Time to Buy and Sell Stock

Entendendo o problema

Pense novamente

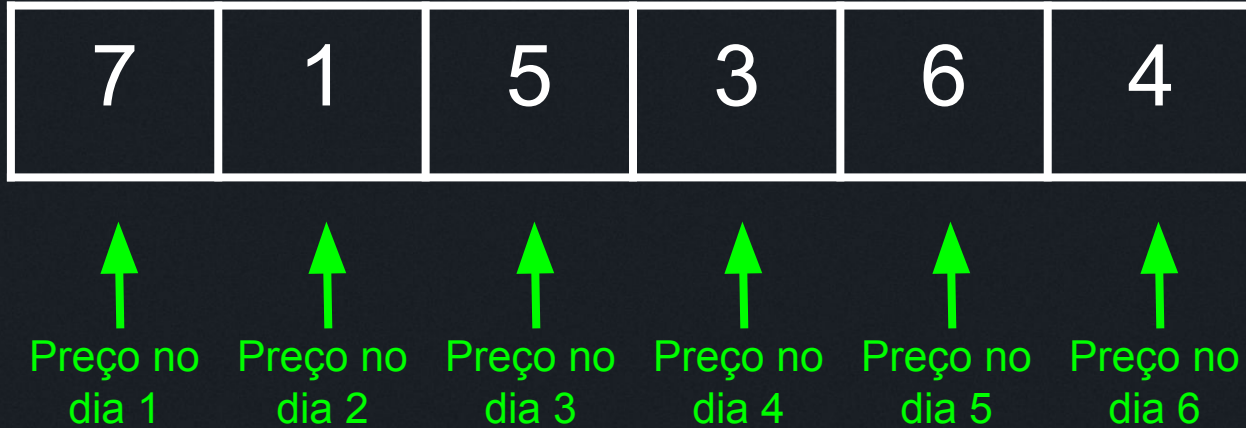
```
def maxProfit(self, prices: List[int]) -> int:
    bestProfit = 0
    minValue = prices[0]
    for i in range(1, len(prices)):
        profit = prices[i] - minValue
        bestProfit = max(bestProfit, profit)
        minValue = min(prices[i], minValue)
    return bestProfit
```



## Best Time to Buy and Sell Stock

Entendendo o problema

Pense novamente



- O que ocorreu agora com esse novo código no LeetCode?
- A complexidade agora é  $O(n)$  porque estamos usando **memoization**, salvando o menor valor em uma variável, ocupando espaço  $O(1)$ .



Programação Dinâmica | Os três lembretes

---

# Os Três Lembretes de Programação Dinâmica

---



## MEMOIZATION

Inicializar uma memória auxiliar para salvar resultados calculados



## CHECAR MEMOIZATION


Antes de calcular, consultar no memo se já o fez  
**(como um cache)**



## SALVAR NA MEMOIZATION

Ao final do cálculo, salvar o resultado no memo





Programação Dinâmica | Mais Exemplos

---



# Programação Dinâmica

Prática no LeetCode - <https://leetcode.com/problems/climbing-stairs>

## 70. Climbing Stairs

Easy

Topics

Companies

Hint

You are climbing a staircase. It takes  $n$  steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

### Example 1:

Input:  $n = 2$

Output: 2

Explanation: There are two ways to climb to the top.

1. 1 step + 1 step
2. 2 steps

### Example 2:

Input:  $n = 3$

Output: 3

Explanation: There are three ways to climb to the top.

1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step

### Constraints:

- $1 \leq n \leq 45$

### Escada com 1 degrau

Só tem uma forma de subir: dando um passo e assim indo para o primeiro e último degrau.

### Escada com 2 degraus

Tem duas formas de subir:

- Dando um passo, pisando no primeiro degrau, para depois dar outro passo e pisar no segundo degrau

- Dar um passo duplo e ir direto para o segundo degrau

# Escada com 3 degraus

Você pode iniciar com um **passo simples** ou um **passo duplo**:

Se der um **passo simples**, resta subir uma escada de 2 degraus (um a menos)

Aqui já sabemos que tem 2 formas diferentes de subir uma escada de 2 degraus

Se der um **passo duplo**, resta subir uma escada com 1 degrau (dois a menos)

Aqui já sabemos que tem 1 forma de subir uma escada com 1 degrau

Resultado final? Temos três formas (2+1) de subir uma escada com 3 degraus.

# Escada com 3 degraus

Você pode iniciar com um **passo simples** ou um **passo duplo**:

Se der um **passo simples**, resta subir uma escada de 2 degraus (um a menos)

Aqui já sabemos que tem 2 formas diferentes de subir uma escada de 2 degraus

Se der um **passo duplo**, resta subir uma escada com 1 degrau (dois a menos)

Aqui já sabemos que tem 1 forma de subir uma escada com 1 degrau

Resultado final? Temos três formas (2+1) de subir uma escada com 3 degraus.

**Isso não é uma indução? Conseguimos generalizar essa ideia!**



# Escada com n degraus

`climbingStairs(n)`

Se o **n for 1**,  
temos 1 forma de subir

Se o **n for 2**,  
temos 2 formas de subir

Se o **n for 3 ou mais**,  
temos `climbingStairs(n-1) + climbingStairs(n-2)`

Conseguem ver uma recursão com essa ideia?  
Tentem essa ideia recursiva no LeetCode. O n é limitado a 45, vai funcionar!

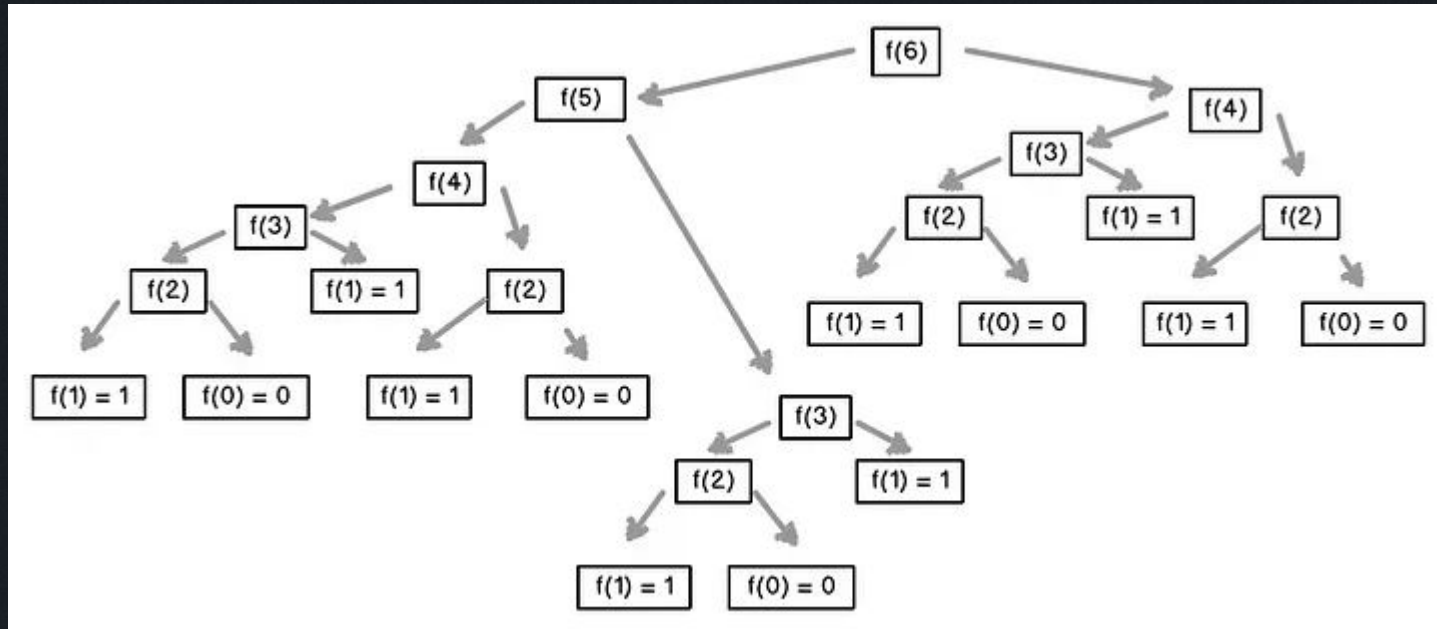
### Solução recursiva

```
def climbStairs(self, n: int) -> int:
    if n==1:
        return 1
    elif n==2:
        return 2
    else:
        return self.climbStairs(n-1)+self.climbStairs(n-2)
```

Pense nas chamadas dessa função para calcular a função para  $n=6$ .

Essa função parece muito com a Fibonacci.

## Chamadas recursivas para $n=6$



Veja quantas vezes recalculamos  $f(3)$  ou  $f(2)$ .

# Pensando recursão com PD

Não é verdade que a quantidade de formas diferentes de subir uma escada com um certo número de degraus nunca muda? Podemos salvar esse resultado então!

Relembrando os passos da técnica de PD, precisamos de uma memória auxiliar para guardar o resultado das chamadas. Uma vez que a gente calcule o resultado para um determinado valor de **n**, não o recalculamos novamente, apenas consultamos dessa memória (cache).

Um dicionário pode servir como memoization, com a chave sendo o número de degraus **n** da escada.



# Os Três Lembretes de Programação Dinâmica

---



## MEMOIZATION

Inicializar uma memória auxiliar para salvar resultados calculados



## CHECAR MEMOIZATION

Antes de calcular, consultar no memo se já o fez  
**(como um cache)**



## SALVAR NA MEMOIZATION

Ao final do cálculo, salvar o resultado no memo

## Solução recursiva com PD

```
def climbStairs(self, n: int, memo=dict()) -> int: #1 => iniciando memoization
    if n==1:
        return 1
    elif n==2:
        return 2
    elif n in memo: #2 => consultando memoization
        return memo[n]
    else:
        memo[n] = self.climbStairs(n-1, memo)+ self.climbStairs(n-2, memo)
        return memo[n] #3 => salvando no memo depois que calcula
```

# Outra solução: iterativa

Já não vimos que conseguimos usar um vetor para ir salvando o resultado da Série de Fibonacci e assim não precisarmos recalcular?

Nesse problema, o  $n$  vai apenas até 45, portanto podíamos usar um vetor com os índices de 0 a 45 para salvar os resultados calculados.

**IMPORTANTE:** é um bom argumento falar que dá para fazer o mesmo apenas com duas variáveis. Você poderia salvar apenas o anterior e o anterior do anterior para calcular o atual. É outro bom argumento o fato que 45 e 2 são constantes, portanto acaba sendo meio  $O(1)$  a complexidade de espaço usando esse vetor para um  $n$  tão pequeno.

## Solução iterativa

```
def climbStairs(self, n: int) -> int:
    memo = [-1]*46
    memo[1] = 1
    memo[2] = 2
    for i in range(3, 46):
        memo[i] = memo[i-1] + memo[i-2]
    return memo[n]
```



# Programação Dinâmica

Prática no LeetCode - <https://leetcode.com/problems/pascals-triangle/>

## 118. Pascal's Triangle

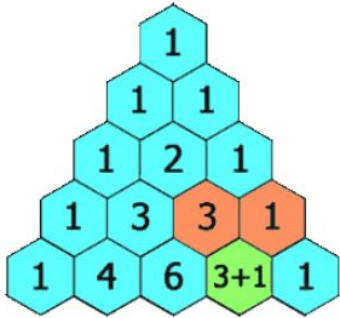
Easy

Topics

Companies

Given an integer `numRows`, return the first `numRows` of **Pascal's triangle**.

In **Pascal's triangle**, each number is the sum of the two numbers directly above it as shown:



Example 1:

Input: `numRows = 5`

Output: `[[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]`

Example 2:

Input: `numRows = 1`

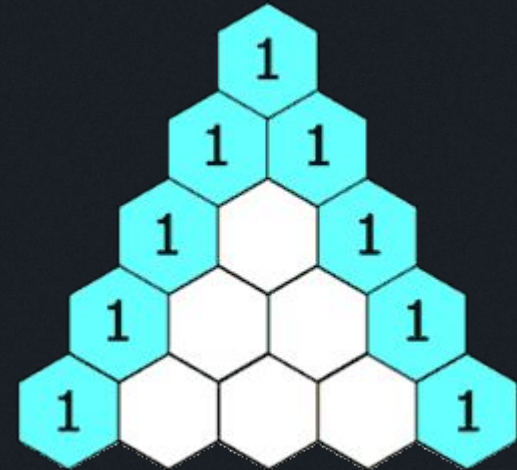
Output: `[[1]]`

Constraints:

- `1 <= numRows <= 30`

## Pensando uma solução

- A figura ajuda muito a pensar uma solução
  - Veja como a primeira linha tem um elemento com valor 1
  - Cada nova linha tem um valor a mais
  - O primeiro e último valor de cada linha sempre é 1
  - Apenas os elementos internos precisam ser calculados, somando dois da linha anterior
- Tente criar uma solução iterativa
- Depois crie uma segunda solução recursiva



## Solução iterativa

```
def generate(self, numRows: int) -> List[List[int]]:
    pascalTriangle = []
    for row in range(numRows):
        thisRow = [1]*(row+1) #initializes a row with 1's
        for i in range(1, row):
            thisRow[i] = pascalTriangle[row-1][i-1]+pascalTriangle[row-1][i]
        pascalTriangle.append(thisRow)
    return pascalTriangle
```

## Solução recursiva

```
def generate(self, numRows: int) -> List[List[int]]:
    if numRows==1:
        return [[1]]
    else:
        pascalTriangle = self.generate(numRows-1)
        row = [1]*(numRows) # this is row index "numRows-1"
        for i in range(1, numRows-1):
            row[i] = pascalTriangle[numRows-2][i-1]+pascalTriangle[numRows-2][i]
        pascalTriangle.append(row)
        return pascalTriangle
```



Obrigado