Módulo Programação Dinâmica

Curso Estruturas de Dados e Algoritmos Expert  
Prof. Dr. Nelio Alves  
<https://devsuperior.com.br>

**Lista de exercícios**Soluções:

<https://github.com/devsuperior/curso-eda/tree/main/programacao-dinamica>

**Problema "climbing\_stairs"** (Adaptado de Leetcode 1971)

*Empresas: Amazon, Google, Facebook, Apple, Microsoft*

Você está subindo uma escada. São necessários n degraus para chegar ao topo.

A cada vez, você pode subir 1 ou 2 degraus. De quantas maneiras distintas você pode subir até o topo?

**Restrições**

* 1 <= n <= 45

Exemplo 1:

| **Entrada 1** | **Saída 1** |
| --- | --- |
| {  "n": 2  } | 2 |

Explicação: Há duas formas de subir ao topo.

1. 1 degrau + 1 degrau
2. 2 degraus

Exemplo 2:

| **Entrada 2** | **Saída 2** |
| --- | --- |
| {  "n": 3  } | 3 |

Explicação: Há três formas de subir ao topo.

1. 1 degrau + 1 degrau + 1 degrau
2. 1 degrau + 2 degraus
3. 2 degraus + 1 degrau

Exemplo 3:

| **Entrada 3** | **Saída 3** |
| --- | --- |
| {  "n": 45  } | 1836311903 |

Assinaturas:

Javascript:

var climbStairs = function(n)

Java:

public int climbStairs(int n)

C#:

public int ClimbStairs(int n)

Python:

def climbStairs(n):

**Problema "mincost\_climbing\_stairs"** (Adaptado de Leetcode 746)

*Empresas: Google, Amazon, Microsoft*

É dada uma matriz inteira cost em que cost[i] é o custo do i-ésimo degrau em uma escada. Depois de pagar o custo, você pode subir um ou dois degraus.

Você pode começar no degrau com índice 0 ou no degrau com índice 1.

Retorne o custo mínimo para chegar ao topo do andar.

**Restrições**

* 2 <= cost.length <= 1000
* 0 <= cost[i] <= 999

Exemplo 1:

| **Entrada 1** | **Saída 1** |
| --- | --- |
| {  "cost": [10, 15, 20]  } | 15 |

Explicação: Comece no índice 1. Pague 15 e suba dois degraus para chegar o topo. O custo total é 15.

Exemplo 2:

| **Entrada 2** | **Saída 2** |
| --- | --- |
| {  "cost": [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]  } | 6 |

Explicação: Comece no índice 0.

* Pague 1 e suba dois degraus até o índice 2.
* Pague 1 e suba dois degraus até o índice 4.
* Pague 1 e suba dois degraus até o índice 6.
* Pague 1 e suba um degrau até o índice 7.
* Pague 1 e suba dois degraus até o índice 9.
* Pague 1 e suba um degrau até chegar o topo.

O custo total é 6.

Assinaturas:

Javascript:

var minCostClimbingStairs = function(cost)

Java:

public int minCostClimbingStairs(int[] cost)

C#:

public int MinCostClimbingStairs(int[] cost)

Python:

def minCostClimbingStairs(cost)

**Problema "frog\_jumps"**

Suponha que temos uma sequência de pedras numeradas de 1 até N, onde cada pedra possui uma altura dada por height[i]. Um sapo está inicialmente na pedra 1 e deseja alcançar a pedra N. Para isso, ele pode pular para a próxima pedra (i+1) ou para a seguinte (i+2). No entanto, cada salto tem um custo |height[i] - height[j]| (valor absoluto), onde j é a pedra onde ele pousará.

Determine o custo mínimo total para que o sapo alcance a pedra N.

**Restrições**

* 2 <= n <= 100000
* 1 <= height[i] <= 10000

Exemplo 1:

| **Entrada 1** | **Saída 1** |
| --- | --- |
| {  "values": [10, 30, 40, 20]  } | 30 |

Explicação: Se seguirmos o caminho 1 → 2 → 4, o custo total é |10 - 30| + |30 - 20| = 30.

Exemplo 2:

| **Entrada 2** | **Saída 2** |
| --- | --- |
| {  "values": [10, 10]  } | 0 |

Explicação: Se seguirmos o caminho 1 → 2, o custo total é |10 - 10| = 0.

Exemplo 3:

| **Entrada 3** | **Saída 3** |
| --- | --- |
| {  "values": [30, 10, 60, 10, 60, 50]  } | 40 |

Explicação: Se seguirmos o caminho 1 → 3 → 5 → 6, o custo total é |30 - 60| + |60 - 60| + |60 - 50| = 40.

Assinaturas:

Javascript:

var maxProfit = minJumpsValue(values)

Java:

public int minJumpsValue(int[] values)

C#:

public int minJumpsValue(int[] values)

Python:

def minJumpsValue(values):

**Problema "coins"** (Adaptado de Leetcode 322)

*Empresas: Amazon, TikTok, Google, Apple, Microsoft, Facebook*

Dado um array de inteiros coins representando moedas de diferentes denominações e um inteiro amount representando um valor total de dinheiro, retorne o menor número possível de moedas necessário para formar esse valor. Se não for possível formar o valor com nenhuma combinação das moedas fornecidas, retorne -1.

Você pode assumir que há um número infinito de cada tipo de moeda disponível.

**Restrições**

* 1 <= coins.length <= 12
* 1 <= coins[i] <= 2³¹ - 1
* 0 <= amount <= 10⁴

Exemplo 1:

| **Entrada 1** | **Saída 1** |
| --- | --- |
| {  "coins": [1, 2, 5]  "amount": 11  } | 3 |

Explicação: 11 = 5 + 5 + 1

Exemplo 2:

| **Entrada 2** | **Saída 2** |
| --- | --- |
| {  "coins": [2]  "amount": 3  } | -1 |

Exemplo 3:

| **Entrada 3** | **Saída 3** |
| --- | --- |
| {  "coins": [1, 2, 3, 7, 11]  "amount": 10000  } | 910 |

Assinaturas:

Javascript:

var coinChange = function(coins, amount)

Java:

public int coinChange(int[] coins, int amount)

C#:

public int coinChange(int[] coins, int amount)

Python:

def coinChange(coins, amount):

**Problema "minimum\_path\_sum"** (Adaptado de Leetcode 64)

*Empresas: Goldman Sachs, Google, Facebook, Amazon*

Dado um grid m x n preenchido com números não negativos, encontre um caminho da parte superior esquerda para a parte inferior direita que minimize a soma de todos os números ao longo desse caminho.

**Observação:** Você só pode se mover para baixo ou para a direita.

**Restrições**

* m == grid.length
* n = grid[i].length
* 1 <= m, n <= 200
* 0 <= grid[i][j] <= 200

Exemplo 1:

| **Entrada 1** | **Saída 1** |
| --- | --- |
| {  "grid": [[1,3,1],[1,5,1],[4,2,1]]  } | 7 |

Explicação: O caminho 1 → 3 → 1 → 1 → 1 minimiza a soma.

Exemplo 2:

| **Entrada 2** | **Saída 2** |
| --- | --- |
| {  "grid": [[1,2,3],[4,5,6]]  } | 12 |

Assinaturas:

Javascript:

var minPathSum = function(grid)

Java:

public int minPathSum(int[][] grid)

C#:

public int MinPathSum(int[][] grid)

Python:

def minPathSum(grid):

**Problema "precious\_stones"**

Imagine que você é um lapidário e possui um bloco bruto de uma pedra preciosa de N gramas. Para lapidar a pedra em gemas, você pode quebrar esse bloco em tamanhos inteiros de gramas. Cada gema tem um valor associado que depende apenas da sua massa, e sabemos de antemão qual o valor values[i] de uma gema de massa 1, 2, …, n.

Por exemplo, suponha que o bloco bruto tenha 4 gramas e values = [2, 5, 7, 9], isso significa que:

* uma gema de 1 grama tem valor 2
* uma gema de 2 gramas tem valor 5
* uma gema de 3 gramas tem valor 7
* uma gema de 4 gramas tem valor 9

Nessa situação, para maximizar o lucro, a melhor decisão é lapidar duas gemas de 2 gramas, gerando valor final de 5 + 5 = 10.

Sabendo disso, dado um valor N de gramas do bloco bruto e o valor das gemas de acordo com a quantidade de gramas, determine o lucro máximo que pode ser obtido, dividindo o bloco bruto de forma ótima.

**Restrições**

* 1 <= n <= 20000
* 1 <= vi <= 100

Exemplo 1:

| **Entrada 1** | **Saída 1** |
| --- | --- |
| {  "n": 4,  "values": [2, 5, 7, 9]  } | 10 |

Explicação: O valor máximo é 10, obtido cortando a pedra em duas gemas de 2 gramas cada. 5 + 5 = 10

Exemplo 2:

| **Entrada 2** | **Saída 2** |
| --- | --- |
| {  "n": 8,  "values": [1, 5, 8, 9, 10, 17, 17, 20]  } | 22 |

Explicação: O valor máximo é 22, obtido cortando a pedra em duas gemas, uma de 2 gramas e outra de 6 gramas. 5 + 17 = 22

Exemplo 3:

| **Entrada 3** | **Saída 3** |
| --- | --- |
| {  "n": 8,  "values": [3, 5, 8, 9, 10, 17, 17, 20]  } | 24 |

Explicação: O valor máximo é 24, obtido cortando a pedra em oito gemas de 1 gramas cada. 8 \* 3 = 24

Assinaturas:

Javascript:

var maxProfit = function(n, values)

Java:

public int maxProfit(int n, int[] values)

C#:

public int MaxProfit(int n, int[] values)

Python:

def maxProfit(n, values):

**Problema "jump\_game"** (Adaptado de Leetcode 55)

*Empresas: Amazon, Apple, Google, Bloomberg, Microsoft, Facebook*

Você recebe um array de inteiros chamado nums. Inicialmente, você está posicionado no primeiro índice do array, e cada elemento no array representa o comprimento máximo de salto possível naquela posição.

Retorne true se você consegue chegar no último índice do array, ou false caso contrário.

**Restrições**

* 1 <= nums.length <= 10⁴
* 0 <= nums[i] <= 10⁵

Exemplo 1:

| **Entrada 1** | **Saída 1** |
| --- | --- |
| {  "nums": [2, 3, 1, 1, 4]  } | true |

Explicação: Pule 1 passo do índice 0 ao 1, depois 3 passos até o último índice..

Exemplo 2:

| **Entrada 2** | **Saída 2** |
| --- | --- |
| {  "nums": [3, 2, 1, 0, 4]  } | false |

Explicação: Você sempre chegará ao índice 3. Como seu pulo máximo é 0, não é possível chegar ao fim.

Assinaturas:

Javascript:

var canJump = function(grid)

Java:

public boolean canJump(int[] nums)

C#:

public bool CanJump(int[] nums)

Python:

def canJump(nums):

**Problema "min\_falling\_path\_sum"** (Adaptado de Leetcode 931)

*Empresas: Amazon, Apple, Google, Bloomberg, Microsoft, Facebook*

Dado uma matriz n x n de inteiros chamada matrix, retorne a soma mínima de qualquer caminho descendente através da matriz.

Um caminho descendente começa em qualquer elemento da primeira linha e escolhe o elemento na próxima linha que está diretamente abaixo ou diagonalmente à esquerda/direita. Especificamente, o próximo elemento a partir da posição (linha, coluna) será (linha + 1, coluna - 1), (linha + 1, coluna) ou (linha + 1, coluna + 1).

**Restrições**

* n == matrix.length == matrix[i].length
* 1 <= n <= 100
* -100 <= matrix[i][j] <= 100

Exemplo 1:

| **Entrada 1** | **Saída 1** |
| --- | --- |
| {  "matrix": [[2,1,3],[6,5,4],[7,8,9]]  } | 13 |

Explicação: Existem dois caminhos descendentes de soma mínima: 1 → 5 → 7 e 1 → 4 → 8.

Exemplo 2:

| **Entrada 2** | **Saída 2** |
| --- | --- |
| {  "matrix": [[-19,57],[-40,-5]]  } | -59 |

Explicação: O caminho descendente mínimo é -19 → -40.

Assinaturas:

Javascript:

var minFallingPathSum = function(matrix)

Java:

public int minFallingPathSum(int[][] matrix)

C#:

public int MinFallingPathSum(int[][] matrix)

Python:

def minFallingPathSum(matrix):

**Problema "house\_robber"** (Adaptado de Leetcode 198)

*Empresas: Amazon, Apple, Microsoft, Google, Facebook*

Você é um ladrão profissional planejando roubar casas ao longo de uma rua. Cada casa tem uma certa quantia de dinheiro guardada. A única restrição que impede você de roubar cada uma delas é que casas adjacentes possuem sistemas de segurança conectados, que **automaticamente alertarão a polícia se duas casas adjacentes forem invadidas na mesma noite**.

Dado um array de inteiros nums representando a quantidade de dinheiro em cada casa, retorne o valor máximo de dinheiro que você pode roubar esta noite **sem alertar a polícia**.

**Restrições**

* 1 <= nums.length <= 100
* 0 <= nums[i] <= 400

Exemplo 1:

| **Entrada 1** | **Saída 1** |
| --- | --- |
| {  "nums": [1,2,3,1]  } | 4 |

Explicação: Roube a casa 1 e a casa 3. Quantidade total roubada = 1 + 3 = 4.

Exemplo 2:

| **Entrada 2** | **Saída 2** |
| --- | --- |
| {  "nums": [2,7,9,3,1]  } | 12 |

Explicação: Roube a casa 1, roube a casa 3 e roube a casa 5. Quantidade total roubada = 2 + 9 + 1 = 12.

Assinaturas:

Javascript:

var rob = function(nums)

Java:

public int rob(int[] nums)

C#:

public int Rob(int[] nums)

Python:

def rob(nums):