# Programação Dinâmica

[**Programação Dinâmica**](#_sirt0i6nooy)

**1**

[Mochila 0-1 - Dado um conjunto de itens, cada um com um peso e um valor, determine o valor máximo que você pode obter ao preencher uma mochila com capacidade limitada:](#_6e8qhgnz8v24) 1

[**Exercício verificar se um número está presente na frequência de fibonacci:**](#_hfjgmmz8w8ee) 2

[Encontrar o n-ésimo número na sequência de Fibonacci](#_6dbd3lkudm5s) 2

[**Maior Subsequência Crescente: Encontrar o comprimento da maior subsequência crescente em um array de números.**](#_v86g2rbh9sgy) 3

[Problema da Partição: Dividir um conjunto de números em dois subconjuntos de forma que a diferença entre as somas dos subconjuntos seja minimizada.](#_3i18j4seby2y) 4

[**Corte de Hastes: Dada uma haste de comprimento n e uma tabela de preços para diferentes comprimentos de corte, encontrar a maneira mais lucrativa de cortar a haste.**](#_6gmoscbf7cia) 5

[Problema da Máxima Subsequência Contígua: Encontrar a soma máxima de uma subsequência situado ao lado em um array de números.](#_7mj7efiv1ats) 5

[**Problema de Empilhamento de Caixas: Dado um conjunto de caixas com dimensões diferentes, encontrar a máxima altura possível ao empilhá-las umas sobre as outras.**](#_5tzyrptfk5mv) 6

## Mochila 0-1 - Dado um conjunto de itens, cada um com um peso e um valor, determine o valor máximo que você pode obter ao preencher uma mochila com capacidade limitada:

Valor máximo da mochila: 15

def knapsack\_max\_value(items, capacity):

n = len(items)

dp = [[0] \* (capacity + 1) for \_ in range(n + 1)]

for i in range(1, n + 1):

for w in range(1, capacity + 1):

weight = items[i - 1][0]

value = items[i - 1][1]

if weight <= w:

dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weight] + value)

else:

dp[i][w] = dp[i - 1][w]

return dp[n][capacity]

# Exemplo de uso

items = [(2, 3), (3, 4), (4, 5), (5, 8), (9, 10)]

capacity = 10

max\_value = knapsack\_max\_value(items, capacity)

print("Valor máximo da mochila:", max\_value)

## Exercício verificar se um número está presente na frequência de fibonacci:

Digite um número: 56

56 não está presente na sequência de Fibonacci.

def is\_perfect\_square(n):

return int(n \*\* 0.5) \*\* 2 == n

def is\_fibonacci\_number(number):

if number < 0:

return False

if number == 0 or number == 1:

return True

a, b = 0, 1

while b < number:

a, b = b, a + b

return b == number

# Exemplo de uso

num = int(input("Digite um número: "))

if is\_fibonacci\_number(num):

print(f"{num} está presente na sequência de Fibonacci.")

else:

print(f"{num} não está presente na sequência de Fibonacci.")

## Encontrar o n-ésimo número na sequência de Fibonacci

Digite o valor de n para encontrar o n-ésimo número de Fibonacci: 6

O 6-ésimo número de Fibonacci é: 8

def fibonacci(n):

if n <= 0:

return 0

elif n == 1:

return 1

else:

fib = [0, 1]

for i in range(2, n+1):

fib.append(fib[i-1] + fib[i-2])

return fib[n]

# Solicita o valor de n

n = int(input("Digite o valor de n para encontrar o n-ésimo número de Fibonacci: "))

# Calcula o n-ésimo número de Fibonacci

resultado = fibonacci(n)

# Exibe o resultado

print(f"O {n}-ésimo número de Fibonacci é:", resultado)

## Maior Subsequência Crescente: Encontrar o comprimento da maior subsequência crescente em um array de números.

Digite os números separados por espaço: 1 2 3 4 5 8 7 1 2 3

Comprimento da maior subsequência crescente: 6

def longest\_increasing\_subsequence(nums):

n = len(nums)

lis = [1] \* n

for i in range(1, n):

for j in range(0, i):

if nums[i] > nums[j] and lis[i] < lis[j] + 1:

lis[i] = lis[j] + 1

return max(lis)

# Solicita a entrada do usuário para o array de números

entrada = input("Digite os números separados por espaço: ")

numeros = list(map(int, entrada.split()))

# Chama a função para encontrar o comprimento da maior subsequência crescente

comprimento = longest\_increasing\_subsequence(numeros)

# Exibe o comprimento da maior subsequência crescente

print("Comprimento da maior subsequência crescente:", comprimento)

## Problema da Partição: Dividir um conjunto de números em dois subconjuntos de forma que a diferença entre as somas dos subconjuntos seja minimizada.

Digite os números separados por espaço: 52 8 9 6 5

Diferença mínima entre as somas dos subconjuntos: 24

def min\_partition\_difference(nums):

total\_sum = sum(nums)

n = len(nums)

dp = [[False] \* (total\_sum // 2 + 1) for \_ in range(n + 1)]

for i in range(n + 1):

dp[i][0] = True

for i in range(1, n + 1):

for j in range(1, total\_sum // 2 + 1):

dp[i][j] = dp[i - 1][j]

if nums[i - 1] <= j:

dp[i][j] |= dp[i - 1][j - nums[i - 1]]

for j in range(total\_sum // 2, -1, -1):

if dp[n][j]:

return total\_sum - 2 \* j

return float('inf')

# Solicita a entrada do usuário para o conjunto de números

entrada = input("Digite os números separados por espaço: ")

numeros = list(map(int, entrada.split()))

# Chama a função para calcular a diferença mínima entre as somas dos subconjuntos

diferenca\_minima = min\_partition\_difference(numeros)

# Exibe a diferença mínima entre as somas dos subconjuntos

print("Diferença mínima entre as somas dos subconjuntos:", diferenca\_minima)

## Corte de Hastes: Dada uma haste de comprimento n e uma tabela de preços para diferentes comprimentos de corte, encontrar a maneira mais lucrativa de cortar a haste.

Lucro máximo: 10

def cut\_rod(prices, n):

dp = [0] \* (n + 1)

for i in range(1, n + 1):

max\_value = -1

for j in range(1, i + 1):

max\_value = max(max\_value, prices[j] + dp[i - j])

dp[i] = max\_value

return dp[n]

# Solicita a entrada do usuário para o número de comprimentos e os preços correspondentes

num\_lengths = int(input("Digite o número de comprimentos possíveis: "))

prices = [0] # Preços correspondentes aos comprimentos, começando a partir do índice 1

print("Digite os preços para cada comprimento:")

for \_ in range(num\_lengths):

price = int(input())

prices.append(price)

length = int(input("Digite o comprimento da haste: "))

# Chama a função para calcular o valor máximo

max\_value = cut\_rod(prices, length)

# Exibe o valor máximo possível

print("Valor máximo possível:", max\_value)

## Problema da Máxima Subsequência Contígua: Encontrar a soma máxima de uma subsequência situado ao lado em um array de números.

Digite os números separados por espaço: 1 5 8 6 3 5

Soma máxima da subsequência contígua: 28

def max\_subarray\_sum(nums):

max\_sum = current\_sum = nums[0]

for num in nums[1:]:

current\_sum = max(num, current\_sum + num)

max\_sum = max(max\_sum, current\_sum)

return max\_sum

# Solicita a entrada do usuário para o array de números

entrada = input("Digite os números separados por espaço: ")

numeros = list(map(int, entrada.split()))

# Chama a função para encontrar a soma máxima da subsequência contígua

soma\_maxima = max\_subarray\_sum(numeros)

# Exibe a soma máxima da subsequência contígua

print("Soma máxima da subsequência contígua:", soma\_maxima)

## Problema de Empilhamento de Caixas: Dado um conjunto de caixas com dimensões diferentes, encontrar a máxima altura possível ao empilhá-las umas sobre as outras.

Digite as dimensões das caixas (comprimento largura altura) uma por linha:

12 54 2

2 6 9

65 8 7

Máxima altura possível de empilhamento: 9

class Box:

def \_\_init\_\_(self, length, width, height):

self.length = length

self.width = width

self.height = height

def can\_be\_stacked\_on(self, other\_box):

return (

self.length < other\_box.length and

self.width < other\_box.width and

self.height < other\_box.height

)

def max\_box\_stack\_height(boxes):

n = len(boxes)

dp = [0] \* n

dp[0] = boxes[0].height

for i in range(1, n):

max\_height = 0

for j in range(i):

if boxes[i].can\_be\_stacked\_on(boxes[j]):

max\_height = max(max\_height, dp[j])

dp[i] = max\_height + boxes[i].height

return max(dp)

# Solicita a entrada do usuário para o número de caixas e suas dimensões

num\_boxes = int(input("Digite o número de caixas: "))

boxes = []

print("Digite as dimensões das caixas (comprimento largura altura) uma por linha:")

for \_ in range(num\_boxes):

length, width, height = map(int, input().split())

boxes.append(Box(length, width, height))

# Chama a função para calcular a máxima altura possível de empilhamento

max\_height = max\_box\_stack\_height(boxes)

# Exibe a máxima altura possível de empilhamento

print("Máxima altura possível de empilhamento:", max\_height)