# Grafos

[**Grafos 1**](#_6f0ii9x4wbhh)

[**Arvore\_Binaria 1**](#_9wf1bporlomn)

[Arvore\_Binaria\_Busca 2](#_7vejqgbft2ji)

[**Dijkstra 4**](#_lpyov1oso36m)

[Euleriano 7](#_vaxwtlffyis8)

[**HeapMax 9**](#_a0yq8a41xx08)

[HeapMin 12](#_immg0fac50kv)

[**Lista\_Adjacencias 15**](#_l6ji889nfpr5)

[Lista\_Adjacencias\_Valorado 16](#_4cumrxxq5l8p)

[**Lista\_Encadeada 17**](#_6vbp88lwc496)

[Matriz\_Adjacencias 18](#_dn5u9qzai643)

[**Busca em largura: 20**](#_htbtpgtotizb)

[Contagem de Grupos Conectados 21](#_t2ijh78iynxl)

[**Algoritmo de Kruskal 23**](#_1gxybgfhml8c)

[Jogos de nim iniciante 25](#_1pjw9sp7x98g)

[**Jogos de nim avançado 26**](#_2ry7zziz9foy)

[Caracteres em trechos monótonos 27](#_i3hpcdlf2cj9)

[Entrada 27](#_46yz44sle6aj)

[Saída 27](#_rgdgoh2osp7i)

[Estourando Balões 28](#_tleg5ruckxro)

[Entrada 28](#_b7vzmgrbilyj)

[Saída 28](#_vj9cybvtjpy2)

[**Jogando 23 29**](#_myylbgvfgjsh)

[Entrada 30](#_duyr6fs7wcp)

[Saída 30](#_9mq4qvjd9zec)

## Arvore\_Binaria

class No:

def \_\_init\_\_(self, valor):

self.valor = valor

self.esquerda = None

self.direita = None

def obtervalor(self):

return self.valor

def setesquerda(self, esquerda):

self.esquerda = esquerda

def setdireita(self, direita):

self.direita = direita

def obteresquerda(self):

return self.esquerda

def obterdireita(self):

return self.direita

no1 = No(4)

no2 = No(2)

no3 = No(5)

no1.setesquerda(no2)

no1.setdireita(no3)

print(no1.obtervalor())

print(no1.obterdireita())

print(no1.obterdireita().obtervalor())

print(no1.obteresquerda())

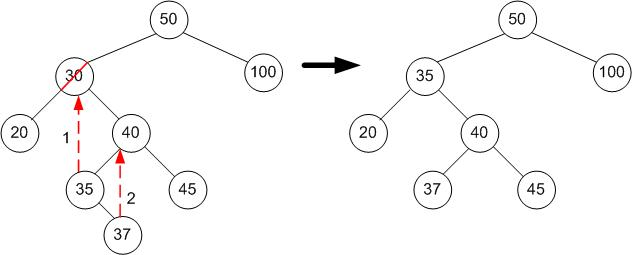
print(no1.obteresquerda().obtervalor())

## Arvore\_Binaria\_Busca

2

1 3 4 6 7 8 10 13 14 2

Uma árvore binária de busca é uma estrutura de dados de árvore binária baseada em nós, onde todos os nós da subárvore esquerda possuem um valor numérico inferior ao nó raiz e todos os nós da subárvore direita possuem um valor superior ao nó raiz.



class No:

def \_\_init\_\_(self, valor):

self.valor = valor

self.esquerda = None

self.direita = None

def obtervalor(self):

return self.valor

def setesquerda(self, esquerda):

self.esquerda = esquerda

def setdireita(self, direita):

self.direita = direita

def obteresquerda(self):

return self.esquerda

def obterdireita(self):

return self.direita

class ArvoreBinariaBusca:

def \_\_init\_\_(self):

self.raiz = None

def obterraiz(self):

return self.raiz

def insere(self, valor):

no = No(valor)

if self.raiz == None:

self.raiz = no

else:

no\_pai = None

no\_atual = self.raiz

while True:

if no\_atual != None:

no\_pai = no\_atual

if no.obtervalor() < no\_atual.obtervalor():

no\_atual = no\_atual.obteresquerda()

else:

no\_atual = no\_atual.obterdireita()

else:

if no.obtervalor() < no\_pai.obtervalor():

no\_pai.setesquerda(no)

else:

no\_pai.setdireita(no)

break

def mostraarvore(self, no\_atual): #percursso em ordem simétrica

if no\_atual != None:

self.mostraarvore(no\_atual.obteresquerda())

print(f'{no\_atual.obtervalor()}', end=' ')

self.mostraarvore(no\_atual.obterdireita())

t = ArvoreBinariaBusca()

t.insere(8)

t.insere(3)

t.insere(6)

t.insere(10)

t.insere(14)

t.insere(1)

t.insere(7)

t.insere(13)

t.insere(4)

t.mostraarvore(t.obterraiz())

## Dijkstra

A matriz de adjacências é:

[0, 5, 6, 10, 0, 0, 0]

[5, 0, 0, 0, 13, 0, 0]

[6, 0, 0, 3, 11, 6, 0]

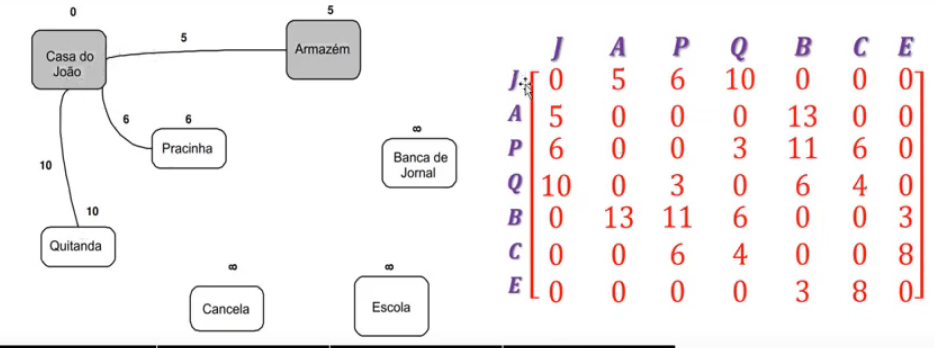
[10, 0, 3, 0, 6, 4, 0]

[0, 13, 11, 6, 0, 0, 3]

[0, 0, 6, 4, 0, 0, 8]

[0, 0, 0, 0, 3, 8, 0]

[[0, 1], [5, 1], [6, 1], [9, 3], [15, 4], [12, 3], [18, 5]]



import math

class HeapMin:

def \_\_init\_\_(self):

self.nos = 0

self.heap = []

def adiciona\_no(self, u, indice):

self.heap.append([u, indice])

self.nos += 1

f = self.nos

while True:

if f == 1:

break

p = f // 2

if self.heap[p-1][0] <= self.heap[f-1][0]:

break

else:

self.heap[p-1], self.heap[f-1] = self.heap[f-1], self.heap[p-1]

f = p

def mostra\_heap(self):

print('A estrutura heap é a seguinte:')

nivel = int(math.log(self.nos, 2))

a = 0

for i in range(nivel):

for j in range(2 \*\* i):

print(f'{self.heap[a]}', end=' ')

a += 1

print('')

for i in range(self.nos-a):

print(f'{self.heap[a]}', end=' ')

a += 1

print('')

def remove\_no(self):

x = self.heap[0]

self.heap[0] = self.heap[self.nos - 1]

self.heap.pop()

self.nos -= 1

p = 1

while True:

f = 2 \* p

if f > self.nos:

break

if f + 1 <= self.nos:

if self.heap[f][0] < self.heap[f-1][0]:

f += 1

if self.heap[p-1][0] <= self.heap[f-1][0]:

break

else:

self.heap[p-1], self.heap[f-1] = self.heap[f-1], self.heap[p-1]

p = f

return x

def tamanho(self):

return self.nos

def menor\_elemento(self):

if self.nos != 0:

return self.heap[0]

return 'A árvore está vazia'

def filho\_esquerda(self, u):

if self.nos >= 2\*u:

return self.heap[2\*u-1]

return 'Esse nó não tem filho'

def filho\_direita(self, u):

if self.nos >= 2\*u+1:

return self.heap[2\*u]

return 'Esse nó não tem filho da direita'

def pai(self, u):

return self.heap[u // 2]

class Grafo:

def \_\_init\_\_(self, vertices):

self.vertices = vertices

self.grafo = [[0] \* self.vertices for i in range(self.vertices)]

def adiciona\_aresta(self, u, v, peso):

self.grafo[u-1][v-1] = peso

self.grafo[v-1][u-1] = peso

def mostra\_matriz(self):

print('A matriz de adjacências é:')

for i in range(self.vertices):

print(self.grafo[i])

def dijkstra(self, origem):

custo\_vem = [[-1, 0] for i in range(self.vertices)]

custo\_vem[origem - 1] = [0, origem]

h = HeapMin()

h.adiciona\_no(0, origem)

while h.tamanho() > 0:

dist, v = h.remove\_no()

for i in range(self.vertices):

if self.grafo[v-1][i] != 0:

if custo\_vem[i][0] == -1 or custo\_vem[i][0] > dist + self.grafo[v-1][i]:

custo\_vem[i] = [dist + self.grafo[v-1][i], v]

h.adiciona\_no(dist + self.grafo[v-1][i], i+1)

return custo\_vem

g = Grafo(7)

g.adiciona\_aresta(1, 2, 5)

g.adiciona\_aresta(1, 3, 6)

g.adiciona\_aresta(1, 4, 10)

g.adiciona\_aresta(2, 5, 13)

g.adiciona\_aresta(3, 4, 3)

g.adiciona\_aresta(3, 5, 11)

g.adiciona\_aresta(3, 6, 6)

g.adiciona\_aresta(4, 5, 6)

g.adiciona\_aresta(4, 6, 4)

g.adiciona\_aresta(5, 7, 3)

g.adiciona\_aresta(6, 7, 8)

g.mostra\_matriz()

resultado\_dijkstra = g.dijkstra(1)

print(resultado\_dijkstra)

## Euleriano

O grafo não é euleriano e nem semieulariano!

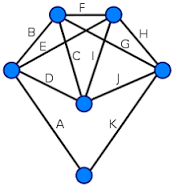
A matriz de adjacências é:

[0, 1, 0, 0]

[1, 0, 0, 0]

[0, 0, 0, 1]

[0, 0, 1, 0]



Um Caminho Euleriano é um caminho em um grafo que visita toda aresta exatamente uma vez. Com caso especial, um Circuito Euleriano é um caminho Euleriano que começa e termina no mesmo vértice.

class Grafo:

def \_\_init\_\_(self, vertices):

self.vertices = vertices

self.grafo = [[0]\*self.vertices for i in range(self.vertices)]

def adiciona\_aresta(self, u, v):

# estou pensando em grafos não direcionados

self.grafo[u-1][v-1] += 1

if u != v:

self.grafo[v-1][u-1] += 1

def mostra\_matriz(self):

print('A matriz de adjacências é:')

for i in range(self.vertices):

print(self.grafo[i])

def tem\_aresta(self, u, v):

if self.grafo[u-1][v-1] == 0:

print(f'Não tem aresta entre os vértices {u} e {v}')

else:

print(f'Existe {self.grafo[u-1][v-1]} de arestas entre os vértices {u} e {v}')

def eh\_euleriano(self):

contador = 0

for i in range(self.vertices):

grau = 0

for j in range(self.vertices):

if i == j:

grau = grau + 2 \* self.grafo[i][j]

else:

grau += self.grafo[i][j]

if grau % 2 != 0:

contador += 1

if contador == 0:

print('É um grafo euleriano!')

elif contador == 2:

print('É um grafo semieuleriano!')

else:

print('O grafo não é euleriano e nem semieulariano!')

g = Grafo(4)

g.adiciona\_aresta(1, 2)

g.adiciona\_aresta(3, 4)

#g.adiciona\_aresta(2, 3)

#g.adiciona\_aresta(1,4)

#g.tem\_aresta(2,4)

g.eh\_euleriano()

g.mostra\_matriz()

## HeapMax

O elemento máximo é: 100

A estrutura heap é a seguinte:

36

19 25

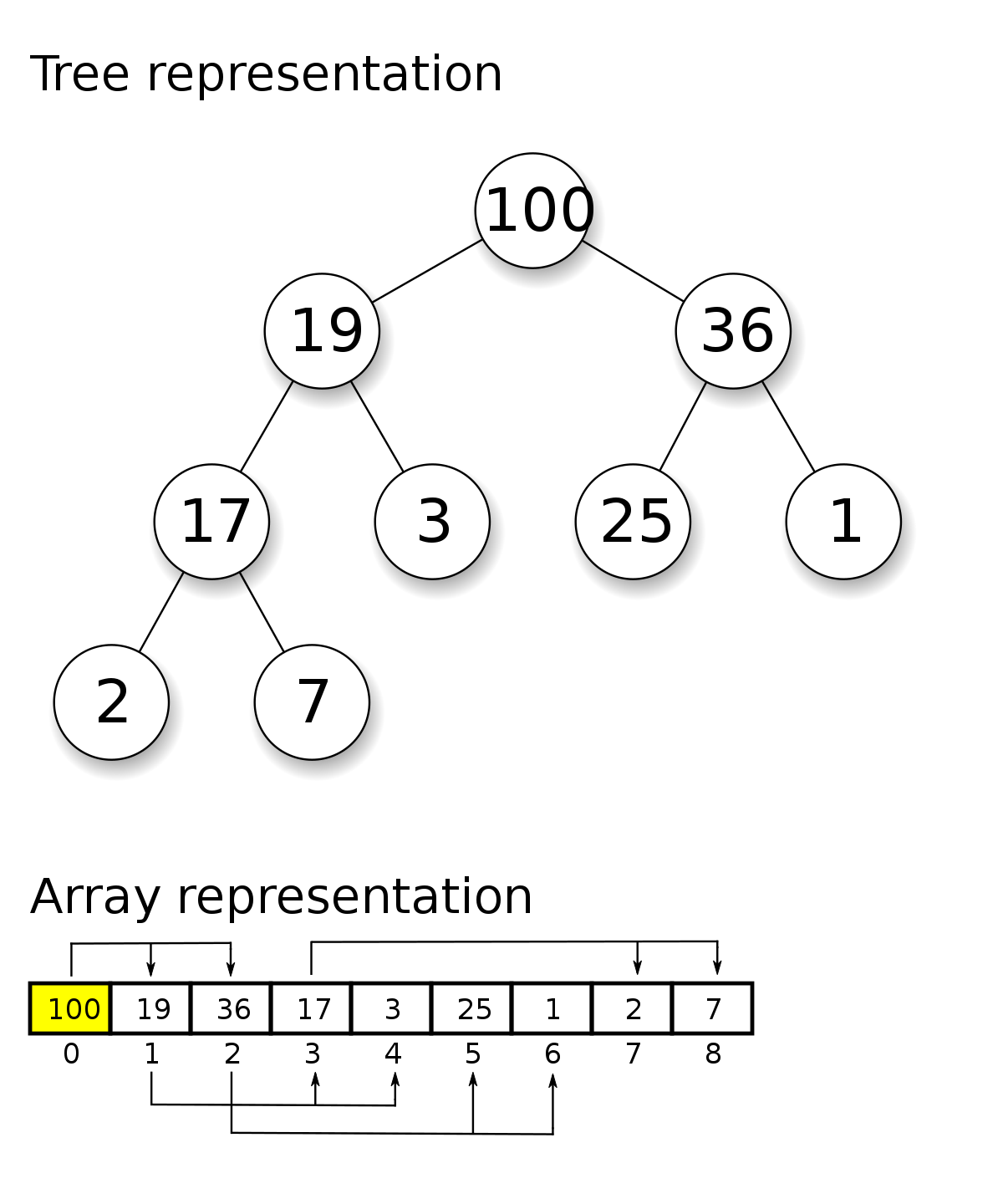
17 3 7 1

2

Tamanho: 8

Filho à esquerda de 17: 2

Filho à direita de 17: Esse nó não tem filho à direita!



import math

class HeapMax:

def \_\_init\_\_(self):

self.nos = 0

self.heap = []

def adiciona\_no(self, u):

self.heap.append(u)

self.nos += 1

f = self.nos

while True:

if f == 1:

break

p = f // 2

if self.heap[p-1] >= self.heap[f-1]: # <= HeapMin

break

else:

self.heap[p-1], self.heap[f-1] = self.heap[f-1], self.heap[p-1]

f = p

def mostra\_heap(self):

#print(self.heap)

print('A estrutura heap é a seguinte:')

nivel = int(math.log(self.nos, 2))

a = 0

for i in range(nivel):

for j in range(2 \*\* i):

print(f'{self.heap[a]}', end = ' ')

a += 1

print('')

for i in range(self.nos - a):

print(f'{self.heap[a]}', end = ' ')

a += 1

print('')

def remove\_no(self):

x = self.heap[0]

self.heap[0] = self.heap[self.nos - 1]

self.heap.pop()

self.nos -= 1

p = 1

while True:

f = 2 \* p

if f > self.nos:

break

if f+1 <= self.nos:

if self.heap[f] > self.heap[f-1]: # < se for HeapMin

f += 1

if self.heap[p-1] >= self.heap[f-1]: # <= se for HeapMin

break

else:

self.heap[f-1], self.heap[p-1] = self.heap[p-1], self.heap[f-1]

p = f

return x

def tamanho(self):

return self.nos

def maior\_elemento(self):

if self.nos != 0:

return self.heap[0]

return 'A árvore está vazia'

def filho\_esqueda(self, i):

if self.nos >= 2\*i:

return self.heap[2\*i - 1]

return 'Esse nó não tem filho!'

def filho\_direita(self, i):

if self.nos >= 2\*i+1:

return self.heap[2\*i]

return 'Esse nó não tem filho à direita!'

def pai(self, i):

return self.heap[i // 2]

h = HeapMax()

h.adiciona\_no(17)

h.adiciona\_no(36)

h.adiciona\_no(25)

h.adiciona\_no(7)

h.adiciona\_no(3)

h.adiciona\_no(100)

h.adiciona\_no(1)

h.adiciona\_no(2)

h.adiciona\_no(19)

elementomax = h.remove\_no()

print(f'O elemento máximo é: {elementomax}')

h.mostra\_heap()

print(f'Tamanho: {h.tamanho()}')

print(f'Filho à esquerda de 17: {h.filho\_esqueda(4)}')

print(f'Filho à direita de 17: {h.filho\_direita(4)}')

## HeapMin

O elemento minimo é: 1

A estrutura heap é a seguinte:

2

7 3

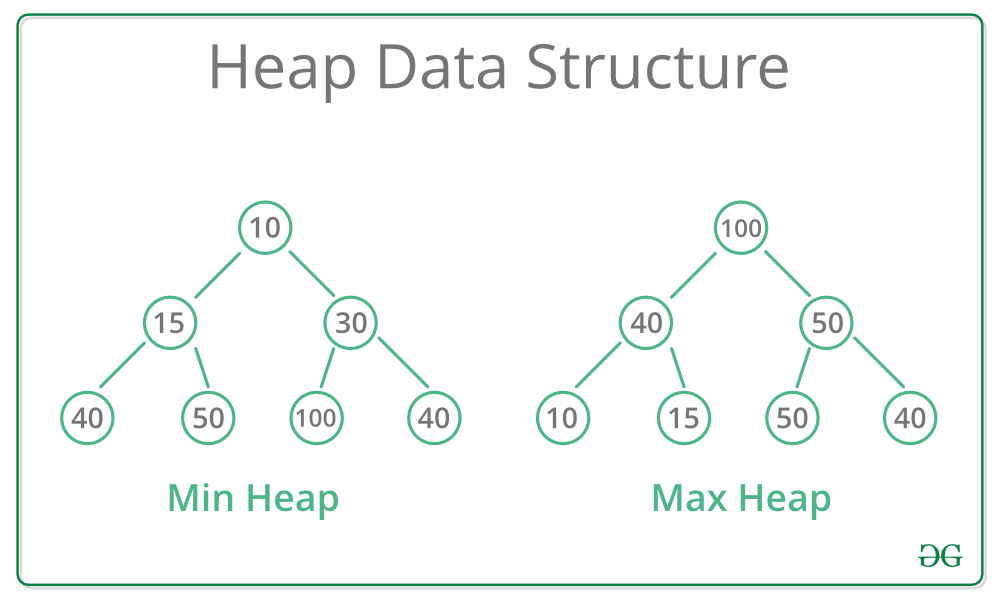
19 17 100 25

36

Tamanho: 8

Filho à esquerda de 17: Esse nó não tem filho!

Filho à direita de 17: Esse nó não tem filho à direita!



import math

class HeapMin:

def \_\_init\_\_(self):

self.nos = 0

self.heap = []

def adiciona\_no(self, u):

self.heap.append(u)

self.nos += 1

f = self.nos

while True:

if f == 1:

break

p = f // 2

if self.heap[p-1] <= self.heap[f-1]: # >= HeapMax

break

else:

self.heap[p-1], self.heap[f-1] = self.heap[f-1], self.heap[p-1]

f = p

def mostra\_heap(self):

#print(self.heap)

print('A estrutura heap é a seguinte:')

nivel = int(math.log(self.nos, 2))

a = 0

for i in range(nivel):

for j in range(2 \*\* i):

print(f'{self.heap[a]}', end = ' ')

a += 1

print('')

for i in range(self.nos - a):

print(f'{self.heap[a]}', end = ' ')

a += 1

print('')

def remove\_no(self):

x = self.heap[0]

self.heap[0] = self.heap[self.nos - 1]

self.heap.pop()

self.nos -= 1

p = 1

while True:

f = 2 \* p

if f > self.nos:

break

if f+1 <= self.nos:

if self.heap[f] < self.heap[f-1]: # > se for HeapMax

f += 1

if self.heap[p-1] <= self.heap[f-1]: # >= se for HeapMax

break

else:

self.heap[f-1], self.heap[p-1] = self.heap[p-1], self.heap[f-1]

p = f

return x

def tamanho(self):

return self.nos

def menor\_elemento(self):

if self.nos != 0:

return self.heap[0]

return 'A árvore está vazia'

def filho\_esqueda(self, i):

if self.nos >= 2\*i:

return self.heap[2\*i - 1]

return 'Esse nó não tem filho!'

def filho\_direita(self, i):

if self.nos >= 2\*i+1:

return self.heap[2\*i]

return 'Esse nó não tem filho à direita!'

def pai(self, i):

return self.heap[i // 2]

h = HeapMin()

h.adiciona\_no(17)

h.adiciona\_no(36)

h.adiciona\_no(25)

h.adiciona\_no(7)

h.adiciona\_no(3)

h.adiciona\_no(100)

h.adiciona\_no(1)

h.adiciona\_no(2)

h.adiciona\_no(19)

elementomin = h.remove\_no()

print(f'O elemento minimo é: {elementomin}')

h.mostra\_heap()

print(f'Tamanho: {h.tamanho()}')

print(f'Filho à esquerda de 17: {h.filho\_esqueda(5)}')

print(f'Filho à direita de 17: {h.filho\_direita(5)}')

## Lista\_Adjacencias

1: 2 -> 3 -> 4 ->

2: 3 ->

3:

4:

class Grafo:

def \_\_init\_\_(self, vertices):

self.vertices = vertices

self.grafo = [[] for i in range(self.vertices)]

def adiciona\_aresta(self, u, v):

# estamos pensando em grafo direcionado sem peso nas arestas

self.grafo[u-1].append(v)

# self.grafo[v-1].append(u) se o grafo não for direcionado

def mostra\_lista(self):

for i in range(self.vertices):

print(f'{i+1}:', end=' ')

for j in self.grafo[i]:

print(f'{j} ->', end=' ')

print('')

g = Grafo(4)

g.adiciona\_aresta(1,2)

g.adiciona\_aresta(1,3)

g.adiciona\_aresta(1,4)

g.adiciona\_aresta(2,3)

g.mostra\_lista()

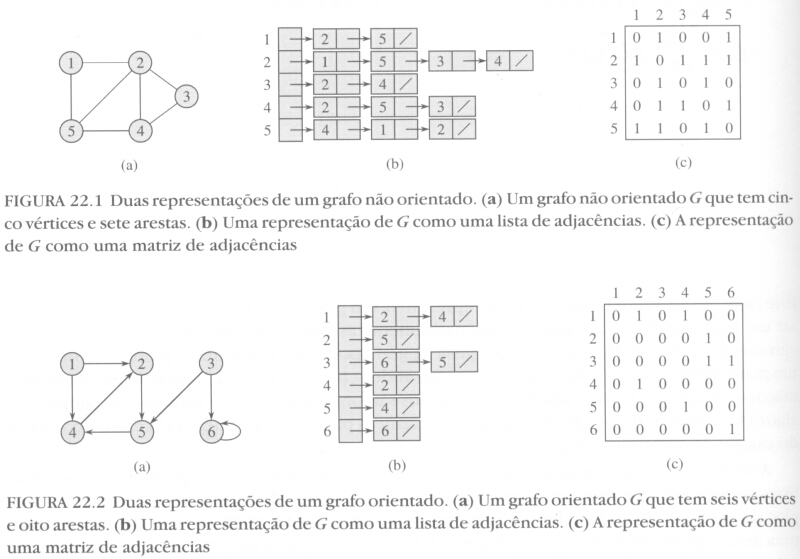
## Lista\_Adjacencias\_Valorado

1: [2, 5] -> [3, 7] -> [4, 6] ->

2: [3, 9] ->

3:

4:



class Grafo:

def \_\_init\_\_(self, vertices):

self.vertices = vertices

self.grafo = [[] for i in range(self.vertices)]

def adiciona\_aresta(self, u, v, peso):

# estamos pensando em grafo direcionado com peso nas arestas

self.grafo[u-1].append([v, peso])

# self.grafo[v-1].append([u,peso]) se o grafo não for direcionado

def mostra\_lista(self):

for i in range(self.vertices):

print(f'{i+1}:', end=' ')

for j in self.grafo[i]:

print(f'{j} ->', end=' ')

print('')

g = Grafo(4)

g.adiciona\_aresta(1, 2, 5)

g.adiciona\_aresta(1, 3, 7)

g.adiciona\_aresta(1, 4, 6)

g.adiciona\_aresta(2, 3, 9)

g.mostra\_lista()

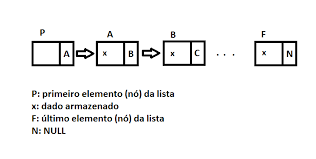
## Lista\_Encadeada

4

7

<\_\_main\_\_.No object at 0x7fc12763ac10>

7



class No:

def \_\_init\_\_(self, valor):

self.valor = valor

self.proximo = None

def obtervalor(self):

return self.valor

def setproximo(self, proximo):

self.proximo = proximo

def obterproximo(self):

return self.proximo

no1 = No(4)

no2 = No(7)

print(no1.obtervalor())

print(no2.obtervalor())

no1.setproximo(no2)

print(no1.obterproximo())

print(no1.obterproximo().obtervalor())

## Matriz\_Adjacencias

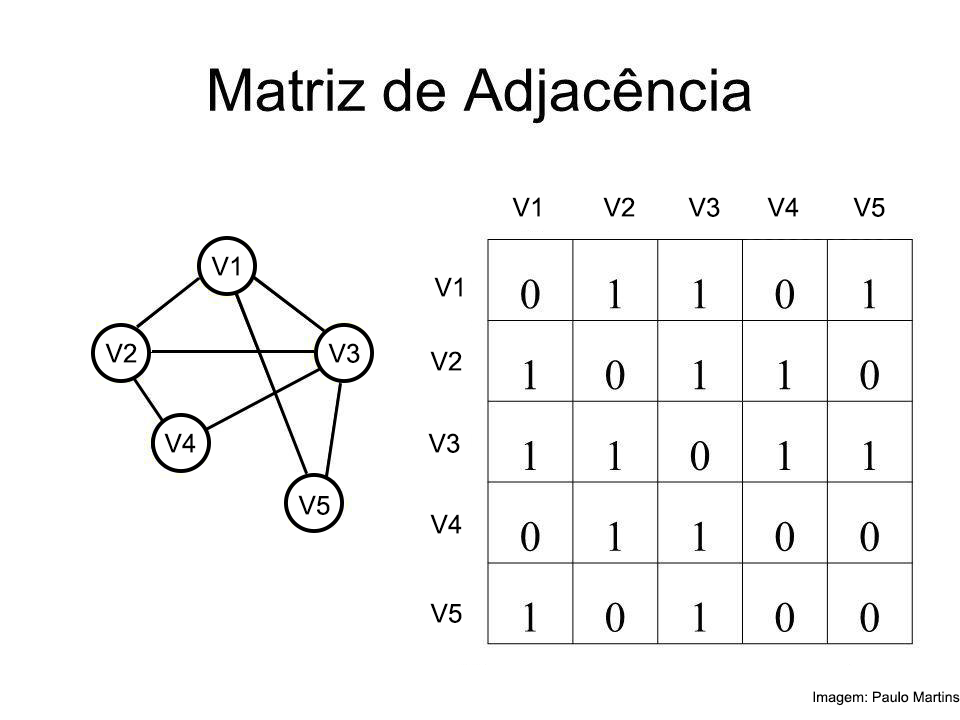
A matriz de adjacências é:

[0, 1, 0, 0]

[0, 0, 1, 0]

[0, 0, 0, 1]

[0, 0, 0, 0]



class Grafo:

def \_\_init\_\_(self, vertices):

self.vertices = vertices

self.grafo = [[0]\*self.vertices for i in range(self.vertices)]

def adiciona\_aresta(self, u, v):

# estou pensando em grafos direcionados simples

self.grafo[u-1][v-1] = 1 #trocar = por += ser for grafo múltiplo

# self.grafo[v-1][u-1] = 1 (caso o grafo não seja direcionado)

def mostra\_matriz(self):

print('A matriz de adjacências é:')

for i in range(self.vertices):

print(self.grafo[i])

g = Grafo(4)

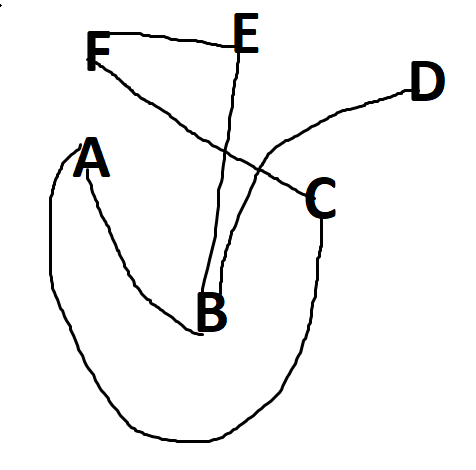
g.adiciona\_aresta(1, 2)

g.adiciona\_aresta(3, 4)

g.adiciona\_aresta(2, 3)

g.mostra\_matriz()

## Busca em largura:



Nesse exemplo, a função bfs implementa o algoritmo de Busca em Largura sem a importação de nenhuma biblioteca externa. Ela usa uma lista para representar a fila e um conjunto para controlar os vértices visitados. A lógica do algoritmo é a mesma, onde visitamos os vizinhos de um vértice, adicionamos na fila se ainda não foram visitados e continuamos a busca até encontrar o vértice alvo ou esvaziar a fila.

def bfs(graph, start, target):

visited = set()

queue = [(start, [])]

while queue:

current, path = queue.pop(0)

visited.add(current)

if current == target:

return path + [current]

for neighbor in graph[current]:

if neighbor not in visited:

queue.append((neighbor, path + [current]))

return None

# Definição do grafo como um dicionário de listas de adjacência

graph = {

'A': ['B', 'C'],

'B': ['A', 'D', 'E'],

'C': ['A', 'F'],

'D': ['B'],

'E': ['B', 'F'],

'F': ['C', 'E']

}

start\_vertex = 'A'

target\_vertex = 'F'

shortest\_path = bfs(graph, start\_vertex, target\_vertex)

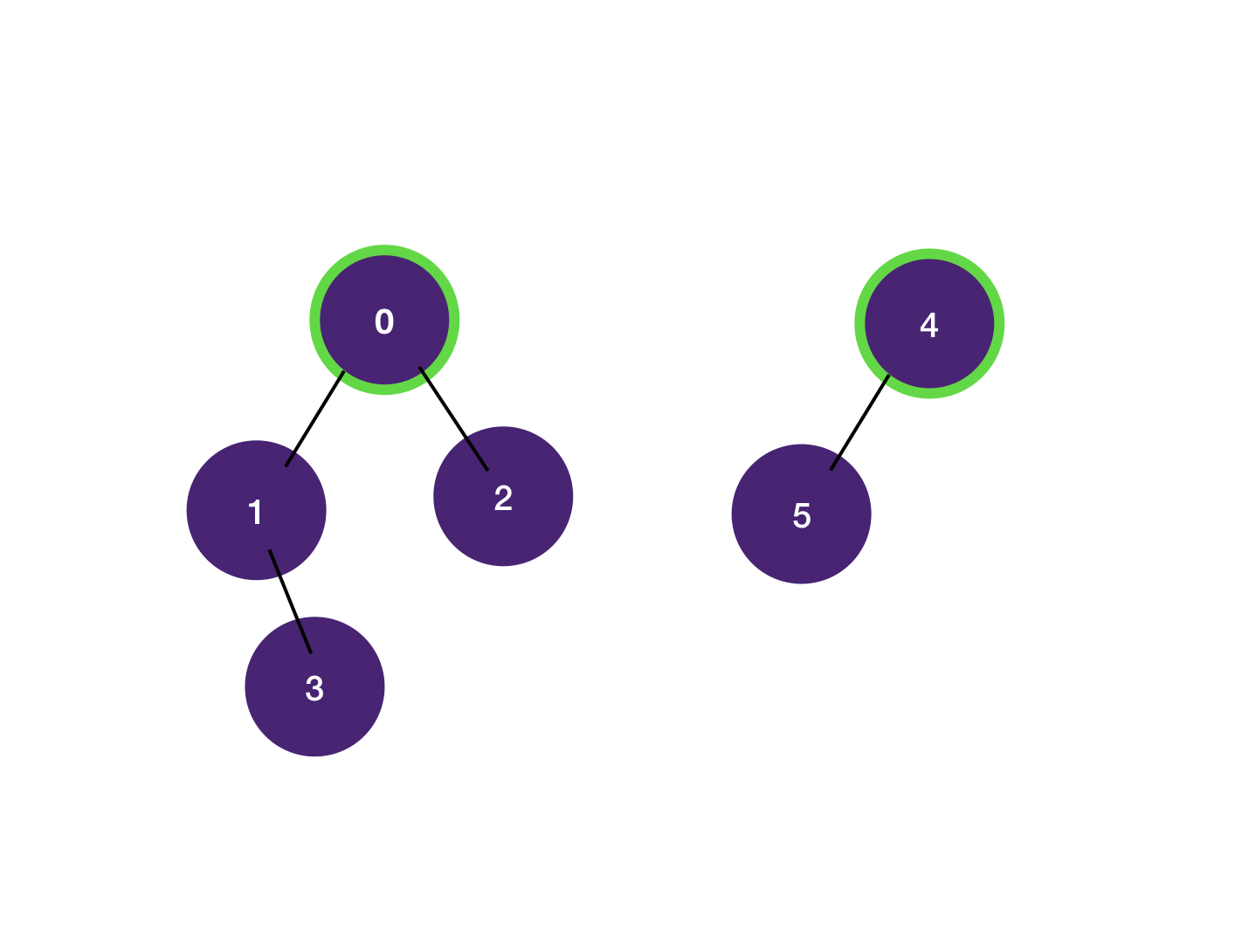
if shortest\_path:

print(f"Menor caminho de {start\_vertex} para {target\_vertex}: {' -> '.join(shortest\_path)}")

else:

print(f"Não existe caminho de {start\_vertex} para {target\_vertex}.")

## Contagem de Grupos Conectados



Isso indica que as pessoas 0 e 2 são amigas, as pessoas 1 e 3 são amigas, as pessoas 2 e 3 são amigas e as pessoas 4 e 5 são amigas. O objetivo é determinar quantos grupos de amigos existem no total.

class UnionFind:

def \_\_init\_\_(self, n):

self.parent = list(range(n))

self.rank = [0] \* n

def find(self, x):

if self.parent[x] != x:

self.parent[x] = self.find(self.parent[x])

return self.parent[x]

def union(self, x, y):

root\_x = self.find(x)

root\_y = self.find(y)

if root\_x != root\_y:

if self.rank[root\_x] > self.rank[root\_y]:

self.parent[root\_y] = root\_x

else:

self.parent[root\_x] = root\_y

if self.rank[root\_x] == self.rank[root\_y]:

self.rank[root\_y] += 1

def count\_groups\_of\_friends(n, friendships):

dsu = UnionFind(n)

for friendship in friendships:

dsu.union(friendship[0], friendship[1])

groups = set()

for i in range(n):

groups.add(dsu.find(i))

return len(groups)

# Exemplo de uso

friendships = [(0, 2), (1, 3), (2, 3), (4, 5)]

total\_groups = count\_groups\_of\_friends(6, friendships)

print("Número de grupos de amigos:", total\_groups) # Saída: 2

## Algoritmo de Kruskal

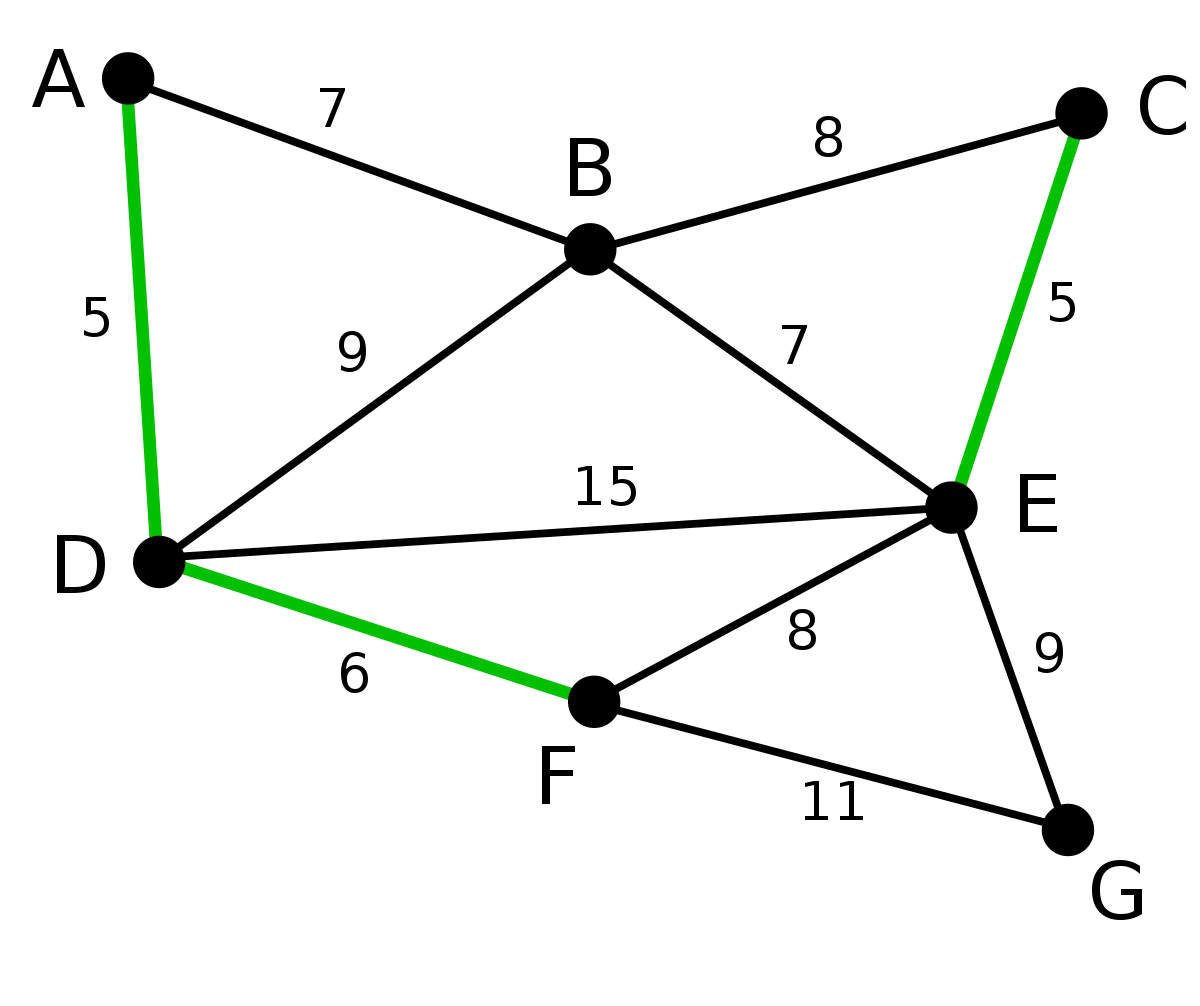
Árvore geradora mínima:

2 - 3: 4

0 - 3: 5

0 - 1: 10

O algoritmo de Kruskal é iterativo. Cada iteração começa com uma floresta geradora F de G . A cada iteração, o algoritmo acrescenta a F uma aresta de G . O critério de escolha da nova aresta é guloso: o algoritmo escolhe uma aresta de peso mínimo dentre as que são externas à floresta.



class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = []

def add\_edge(self, u, v, w):

self.graph.append((u, v, w))

def kruskal(self):

self.graph = sorted(self.graph, key=lambda item: item[2])

parent = [i for i in range(self.V)]

result = []

def find\_parent(node):

if parent[node] == node:

return node

return find\_parent(parent[node])

def union(u, v):

parent[find\_parent(u)] = find\_parent(v)

for u, v, w in self.graph:

if find\_parent(u) != find\_parent(v):

result.append((u, v, w))

union(u, v)

return result

# Exemplo de uso

g = Graph(4)

g.add\_edge(0, 1, 10)

g.add\_edge(0, 2, 6)

g.add\_edge(0, 3, 5)

g.add\_edge(1, 3, 15)

g.add\_edge(2, 3, 4)

minimum\_spanning\_tree = g.kruskal()

print("Árvore geradora mínima:")

for u, v, w in minimum\_spanning\_tree:

print(f"{u} - {v}: {w}")

## Jogos de nim iniciante

Você está jogando um jogo de Nim com um oponente. Existem 3 montes de pedras, e em cada monte, há um número diferente de pedras. Os jogadores alternam turnos, removendo pelo menos uma pedra de um dos montes em seu turno. O jogador que retirar a última pedra vence. Suponha que você comece o jogo. Escreva um programa que determine se você pode vencer o jogo, dado o número inicial de pedras em cada monte.



def can\_win\_nim\_game(piles):

total\_xor = 0

for pile in piles:

total\_xor ^= pile

return total\_xor != 0

# Exemplo de uso

piles = [3, 4, 5]

if can\_win\_nim\_game(piles):

print("Você pode vencer o jogo!")

else:

print("Você não pode vencer o jogo.")

## Jogos de nim avançado

Você está jogando um jogo de Nim mais avançado com um oponente. Existem N montes de pedras, numerados de 1 a N, e em cada monte, há um número diferente de pedras. Os jogadores alternam turnos, removendo pelo menos uma pedra de um dos montes em seu turno. O jogador que retirar a última pedra vence. Suponha que você comece o jogo. Escreva um programa que determine se você pode vencer o jogo, dado o número inicial de pedras em cada monte.

No entanto, há uma nova regra: em cada turno, você pode remover qualquer quantidade de pedras de um monte, mas também pode dividir um monte em dois montes menores, desde que cada novo monte tenha pelo menos uma pedra.

def can\_win\_advanced\_nim\_game(piles):

def calculate\_nimber(pile):

if pile == 0:

return 0

if nimbers[pile] != -1:

return nimbers[pile]

seen = set()

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

seen.add(calculate\_nimber(pile - i))

nim = 0

while nim in seen:

nim += 1

nimbers[pile] = nim

return nim

total\_xor = 0

nimbers = [-1] \* (max(piles) + 1)

for pile in piles:

nimber = calculate\_nimber(pile)

total\_xor ^= nimber

return total\_xor != 0

# Exemplo de uso

piles = [3, 4, 5, 6]

if can\_win\_advanced\_nim\_game(piles):

print("Você pode vencer o jogo!")

else:

print("Você não pode vencer o jogo.")

## Caracteres em trechos monótonos

Neste problemas iremos lidar com sequências de caracteres, muitas vezes chamadas de *strings*. Uma sequência é *não-trivial* se ela possui ao menos dois elementos.

Dada uma sequência **s**, dizemos que um trecho ***s****i*, ... ,***s****j* é monótono se todos seus caracteres são iguais, e dizemos que ela é maximal se este trecho não pode ser estendido à esquerda e nem à direita sem perder a monotonicidade.

Dada uma sequência composta apenas por caracteres “**a**” e “**b**”, determine quantos caracteres “**a**”  
ocorrem em trechos monótonos maximais não-triviais.

## **Entrada**

A entrada é composta por duas linhas. A primeira linha contém um único inteiro **N**, satisfazendo 1≤ **N** ≤105. A segunda linha contém uma string, com exatamente **N** caracteres, composta apenas pelos caracteres “**a**” e “**b**”.

## **Saída**

A saída é composta por uma única linha contendo um inteiro correspondente à quantidade total de vezes que o caractere “**a**” ocorre em trechos monótonos maximais não-triviais.

def contar\_caracteres\_a\_em\_trechos\_monotonos(s):

n = len(s)

count\_a = 0

i = 0

while i < n:

j = i

while j < n and s[j] == s[i]:

j += 1

if j - i > 1 and s[i] == 'a':

count\_a += j - i

i = j

return count\_a

# Leitura da entrada

n = int(input())

sequencia = input()

# Chamando a função para contar os caracteres 'a' em trechos monótonos maximais não-triviais

resultado = contar\_caracteres\_a\_em\_trechos\_monotonos(sequencia)

# Imprimindo o resultado

print(resultado)

## 

## Estourando Balões

Após a cerimônia de encerramento da Maratona um número enorme de balões soltos estão flutuando no espaço do salão. O dono do salão está bravo porque no dia seguinte ocorre outro evento importante e os balões precisam ser removidos. Felizmente este ano Carlinhos veio preparado com seu arco e flecha para estourar os balões.

Felizmente também, devido ao fluxo do ar condicionado, os balões estão todos em um mesmo plano vertical (isto é, um plano paralelo a uma das paredes), embora em alturas e posições distintas.

Carlinhos vai atirar a partir do lado esquerdo do salão, a uma altura de sua escolha, em direção ao lado direito do salão. Cada flecha se move da esquerda para a direita, na altura em que foi lançada, no mesmo plano vertical dos balões. Ao encontrar um balão, este estoura e a flecha continua seu movimento para a direita, com a altura diminuída de 1. Ou seja, se a flecha estava a uma altura **H**, após atingir um balão ela continua na altura **H**-1.

Carlinhos quer estourar todos os balões atirando o menor número possível de flechas. Você pode ajudá-lo?

## **Entrada**

A primeira linha da entrada contém um inteiro **N**, o número de balões (1≤**N**≤5\*105). Como todos os balões estão num mesmo plano vertical, vamos definir que a altura de um balão é dada em relação ao eixo **y** e a posição de um balão é dada em relação ao eixo **x** desse plano. Os balões são numerados de 1 a **N**. Os números dos balões indicam as suas posições, da esquerda (balão número 1) para a direita (balão número **N**), independentemente das suas alturas. A posição do balão número i é diferente da posição do balão número i+1, para todo i. A segunda linha contém **N** inteiros **H**i, onde **H**i indica a altura em que o balão número i está (1≤**H**i≤106 para 1≤i≤**N**).

## **Saída**

Seu programa deve produzir uma única linha, contendo um único inteiro, o menor número de flechas que Carlinhos precisa atirar para estourar todos os balões.

def menor\_numero\_flechas(N, alturas):

# Criar uma lista para manter o número mínimo de flechas necessárias para estourar os balões

flechas\_necessarias = [1] \* N

# Para cada balão, verifique se ele pode ser atingido por uma flecha anterior

for i in range(1, N):

for j in range(i):

# Se a altura do balão atual for maior ou igual à altura do balão anterior

# e o número de flechas necessárias para atingir o balão anterior for menor ou igual

# ao número de flechas necessárias para atingir o balão atual, então atualize o número

# mínimo de flechas necessárias para atingir o balão atual.

if alturas[i] >= alturas[j] and flechas\_necessarias[j] <= flechas\_necessarias[i]:

flechas\_necessarias[i] = flechas\_necessarias[j] + 1

# O resultado é o número máximo de flechas necessárias para atingir qualquer balão

resultado = max(flechas\_necessarias)

return resultado

# Leitura da entrada

N = int(input())

alturas = list(map(int, input().split()))

# Chamar a função e imprimir o resultado

resultado = menor\_numero\_flechas(N, alturas)

print(resultado)

# **Jogando 23**

Por Sociedade Brasileira de Computação (SBC), Maratona de Programação da SBC - ICPC - 2022 BR Brazil

**Timelimit: 1**

Vinte e três é um jogo de cartas simples, jogado por crianças. Como o nome sugere, ele é uma variação do jogo vinte e um (blackjack em inglês), que é um dos jogos mais jogados em cassinos e sites de jogos.

O jogo utiliza um baralho de 52 cartas, com quatro naipes, cada naipe com 13 cartas (ás, 2, 3, 4, 5, 6, 7, 8, 9, 10, valete, dama e rei). Os naipes das cartas não são relevantes. As cartas com figuras (valete, dama e rei) valem dez pontos, as cartas com números valem o seu número em pontos (por exemplo, a carta 4 vale quatro pontos) e o ás vale um ponto.

Ganha o jogo o jogador que tiver o maior número de pontos, desde que não exceda 23. Se um jogador tem um número de pontos maior do que 23 dizemos que o jogador estourou.

As regras do jogo são simples: a cada partida, inicialmente o baralho é embaralhado, as cartas são colocadas em um monte e cada jogador recebe duas cartas do monte. Todas as cartas são distribuídas com a face para cima (todos os jogadores vêem as cartas de todos os jogadores). O passo seguinte, chamado de rodada, é repetido enquanto houver jogadores ativos: uma carta é retirada do monte e colocada na mesa com a face para cima. Essa carta, denominada carta comum, vale para todos os jogadores. Se um jogador estourar, ele sai do jogo. Vence a partida o jogador que numa determinada rodada somar 23 (somando suas duas cartas iniciais mais as cartas comuns), ou se o jogador for o único jogador ativo ao final da rodada. Note que pode haver mais de um vencedor (cujas cartas somam 23) e que pode não haver vencedor em uma partida.

João e Maria estão jogando vinte e três. Os dois são os únicos jogadores, nenhum dos dois estourou e nenhum dos dois tem 23 pontos. Além disso, a pontuação dos jogadores é tal que a próxima carta comum pode fazer com que a partida termine.

Dadas as cartas iniciais de João e Maria e as cartas comuns, determine qual é o valor da carta de menor valor que deve ser retirada do monte na próxima rodada para que Maria vença a partida.

## **Entrada**

A primeira linha da entrada contém um inteiro **N** (1≤**N**≤8), o número de rodadas do jogo até o momento. Cada carta é descrita por um inteiro **I** (1≤**I**≤13). Note que as cartas com figuras (valete, dama e rei) são representadas na entrada pelos valores 11, 12 e 13 e não por quantos pontos elas valem. A segunda linha contém dois inteiros, descrevendo as duas cartas iniciais de João. A terceira linha contém dois inteiros, descrevendo as duas cartas iniciais de Maria. A quarta e última linha contém **N** inteiros, descrevendo as cartas comuns, na ordem em que são retiradas do monte.

## **Saída**

Seu programa deve produzir uma única linha, contendo um único inteiro, o valor da carta de menor valor que deve ser retirada do monte na próxima rodada para Maria vencer a partida, ou -1 se não for possível Maria vencer a partida nessa próxima rodada.

| Exemplos de Entrada | Exemplos de Saída |
| --- | --- |
| 1 10 5 9 10 1 | 3 |

def menor(a, b):

return a if a < b else b

def soma(lista):

return sum(map(lambda a: menor(a, 10), lista))

def cartas\_disponiveis(lista):

return sorted([i for i in range(1, 14) for j in range(1, 5) if i not in lista])

n = int(input())

j, m, comum = map(int, input().split())

joao = soma([j, comum])

maria = soma([m, comum])

cartas = cartas\_disponiveis([j, m, comum])

carta = -1

for c in reversed(cartas):

if (maria + menor(c, 10) == 23 or

(maria + menor(c, 10) < 23 and joao + menor(c, 10) > 23)):

carta = c

print(carta)