1.class Graph:

def \_init\_(self, vertices):

self.V = vertices

self.graph = [[0 for \_ in range(vertices)] for \_ in range(vertices]

def is\_safe(self, v, colour, c):

for i in range(self.V):

if self.graph[v][i] == 1 and colour[i] == c:

return False

return True

def graph\_colouring\_util(self, m, colour, v):

if v == self.V:

return True

for c in range(1, m+1):

if self.is\_safe(v, colour, c):

colour[v] = c

if self.graph\_colouring\_util(m, colour, v+1):

return True

colour[v] = 0

def graph\_colouring(self, m):

colour = [0] \* self.V

if not self.graph\_colouring\_util(m, colour, 0):

return False

print("Solution exists. The assigned colours are:")

for c in colour:

print(c, end=" ")

return True

# Example Usage

g = Graph(4)

g.graph = [[0, 1, 1, 1],

[1, 0, 1, 0],

[1, 1, 0, 1],

[1, 0, 1, 0]]

m = 3

g.graph\_colouring(m)

2.l=[11,13,15,17,19,21,23,35,37]

l2=sorted(l)

print("max",l2[-1])

print("min",l2[0])

3.def rob(nums):

def rob\_linear(houses):

prev,curr=0,0

for money in houses:

prev,curr=curr,max(curr,prev+money)

return curr

if len(nums)==1:

return nums[0]

return max(rob\_linear(nums[1:]),rob\_linear(nums[:-1]))

print(rob([2,3,2]))

4.def selection(arr):

n=len(arr)

for i in range(n):

min=i

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

if arr[j]<arr[min]:

min=j

arr[i],arr[min]=arr[min],arr[i]

return arr

arr=[5,2,9,1,5,6]

print(selection(arr))

5.import heapq

def dijkstra(graph, start):

distances = {node: float('infinity') for node in graph}

distances[start] = 0

queue = [(0, start)]

while queue:

current\_distance, current\_node = heapq.heappop(queue)

if current\_distance > distances[current\_node]:

continue

for neighbor, weight in graph[current\_node].items():

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(queue, (distance, neighbor))

return distances

graph = {

'A': {'B': 1, 'C': 4},

'B': {'A': 1, 'C': 2, 'D': 5},

'C': {'A': 4, 'B': 2, 'D': 1},

'D': {'B': 5, 'C': 1}

}

start\_node = 'A'

result = dijkstra(graph, start\_node)

print(result)

6.def findKthPositive(arr,k):

missing=[]

num=1

while len(missing)<k:

if num not in arr:

missing.append(num)

num += 1

return missing[-1]

arr1=[2, 3, 4, 7, 11]

k=5

output1=findKthPositive(arr1,k)

print(output1)

7.def binary(arr,x,low,high):

while low<=high:

mid=low+(high-low)//2

if arr[mid]==x:

return mid

elif arr[mid]<x:

low=mid+1

else:

high=mid-1

return -1

arr=[10,20,30,40,50,60]

x=50

result=binary(arr,x,0,len(arr)-1)

print(result)

8.def com(nums,target):

dp=[[] for i in range(target+1)]

dp[0]=[[]]

for c in nums:

for i in range(c,target+1):

dp[i]+=[comb+[c] for comb in dp[i-c]]

return dp[target]

nums=[2,3,6,7]

target=7

print(com(nums,target))

9.def merge\_sort(arr):

if len(arr)>1:

mid=len(arr)//2

l=arr[:mid]

r=arr[mid:]

merge\_sort(l)

merge\_sort(r)

i=j=k=0

while i<len(l) and j<len(r):

if l[i]<r[j]:

arr[k]=l[i]

i+=1

else:

arr[k]=r[j]

j+=1

k+=1

while i<len(l):

arr[k]=l[i]

i+=1

k+=1

while j<len(r):

arr[k]=r[j]

j+=1

k+=1

return arr

arr=[31,23,35,27,21,15,28]

print(merge\_sort(arr))

10.import heapq

def kclosest(points,k):

max\_heap=[]

for x,y in points:

dist=-(x\*x+y\*y)

if len(max\_heap)<k:

heapq.heappush(max\_heap,(dist,x,y))

else:

heapq.heappushpop(max\_heap,(dist,x,y))

return [(x,y) for i ,x,y in max\_heap]

points=[[1,3],[-2,2],[2,-2]]

k=2

print(kclosest(points,k))