```
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
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)



arr=[2,4,6,8,10,12,14,18]
print("max is",arr[-1])
print("min is",arr[0])

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]))



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)

def selection(arr):
    for i in range(len(arr)):
        min=i
        for j in range(i+1,len(arr)):
            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))
```

```
def kthpositive(arr,k):
    missing=[]
    num=1
    while len(missing)<k:
        if num not in arr:
            missing.append(num)
        num+=1
    return missing[-1]
arr=[2,3,4,7,11]
k=5
print(kthpositive(arr,k))
```

```
def binary(arr,key):
    low=0
    high=len(arr)+1
    while low<=high:
        mid=(low+high)//2
        if arr[mid]<key:
            low=mid+1
        elif arr[mid]>key:
            high=mid-1
        else:
            return mid
arr=[10,20,30,40,50,60]
key=50
print(binary(arr,key))
```

```
def combinationSum(candidates, target):
   dp = [[] for _ in range(target + 1)]
   dp[0] = [[]]
   for c in candidates:
      for i in range(c, target + 1):
         dp[i] += [comb + [c] for comb in dp[i - c]]
   return dp[target]
candidates = [2, 3, 6, 7]
target = 7
print(combinationSum(candidates,target))
```

```
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=[31,23,35,27,11,21,15,28]
print(merge_sort(arr))


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 _,x,y in max_heap]
points=[[1,3],[-2,2],[5,8],[0,1]]
k=2
print(kclosest(points,k))
```