# Graphs

```
# -- BFS Traversal ---
from collections import defaultdict, deque
start = 'a'
q = deque()
q.append(start)
path = list()
path.append(start)
visited = defaultdict(bool)
visited[start] = True
while len(q) != 0:
    x = q.pop()
    for neg in graph[x]:
        if not visited[neg]:
            visited[neg] = True
            q.append(neg)
            path.append(neg)
print(path)
```

```
# ---DFS Traversal---
def DFS(graph, visited, start, path):
    path.append(start)
    visited[start] = True
    for nebar in graph[start]:
        if not visited[nebar]:
            DFS(graph, visited, nebar, path)
    return path
start = 'A'
path = list()
visited = defaultdict(bool)
dfs = DFS(graph, visited, start, path)
print(dfs)
```

```
#---Detect Cycle Directed Graph ---
from collections import defaultdict
def DFS_Util(graph, node, visited, recStack):
    visited[node] = True
    recStack[node] = True
    for nebr in graph[node]:
        if not visited[nebr]:
            if DFS_Util(graph, nebr, visited, recStack):
                return True
        elif recStack[nebr]:
            return True
    recStack[node] = False
    return False
def isCyclic(graph, nodes):
    visited = [False] * (nodes + 1)
    recStack = [False] * (nodes + 1)
    for node in range(nodes):
        if not visited[node]:
            if DFS_Util(graph, node, visited, recStack):
                return True
    return False
n, e = map(int, input().split(' '))
G = defaultdict(list)
for i in range(e):
    u, v = map(int, input().split(' '))
    G[u].append(v)
print(f'is cycle-> {isCyclic(G, n)}')
```

```
#--Detect cycle in Bidirectional graph ---
from collections import defaultdict
def DFS_Util(u, par, adj, vis):
    vis[u] = True
    for v in adj[u]:
        if v == par:
            continue
        elif vis[v]:
            return True
        else:
            if DFS_Util(v, u, adj, vis):
                return True
    return False
def isCycle(n, adj):
    vis = [False for i in range(n)]
    for i in range(n):
        if not vis[i]:
            if DFS_Util(i, -1, adj, vis):
                return True
    return False
n, e = map(int, input().split(' '))
g = defaultdict(list)
for _ in range(e):
    u, v = map(int, input().strip().split(' '))
    g[u].append(v)
    g[v].append(u)
print(isCycle(n, g))
```

```
# --BELLMAN FORD --
def bellman_ford(graph, nodes, src):
    dist = [float("Inf")] * nodes
    dist[src] = 0
    for _ in range(nodes - 1):
        for a, b, c in graph:
            if dist[a] != float("Inf") and dist[a] + c < dist[b]:
                dist[b] = dist[a] + c
    for a, b, c in graph:
        if dist[a] != float("Inf") and dist[a] + c < dist[b]:
            print("Graph contains negative weight cycle")
    return dist
g = {
    (0, 1, 2),
    (0, 2, 4),
    (1, 3, 2),
    (2, 4, 3),
    (2, 3, 4),
    (4, 3, -5),
}
x = bellman_ford(g, 5, 0)
print(x)
```

```
#--DIJKESTRA--
import sys
def dijkstra(G, nodes, src, d):
    dist = [sys.maxsize] * nodes
    dist[src] = 0
    SP_Set = [False] * nodes
    if src == dist:
        return 0
    if (src not in range(nodes + 1)) or (d not in range(nodes + 1)):
        return -1
    else:
        for cout in range(nodes):
            min = sys.maxsize
            u = 0
            for v in range(nodes):
                if dist[v] < min and SP_Set[v] == False:
                    min = dist[v]
                    u = v
            SP_Set[u] = True
            for v in range(nodes):
                if G[u][v] > 0 and SP_Set[v] == False and dist[v] > dist[u] + G[u][v]:
                    dist[v] = dist[u] + G[u][v]
    return dist[d]
x = dijkstra(adj, nodes, src,dist)
print(x)
```

```
# -- PRIME --
import sys
def printMST(g, nodes, parent):
    print("Edge \tWeight")
    for i in range(1, nodes):
        print(parent[i], "-", i, "\t", g[i][parent[i]])
def minKey(g, nodes, key, mstSet):
    min = sys.maxsize
    min_index = -1
    for v in range(nodes):
        if key[v] < min and mstSet[v] == False:
            min = key[v]
            min_index = v
    return min_index
def primMST(g, nodes):
    key = [sys.maxsize] * nodes
    parent = [None] * nodes
    key[0] = 0
    mstSet = [False] * nodes
    parent[0] = -1
    for cout in range(nodes):
        u = minKey(g, nodes, key, mstSet)
        mstSet[u] = True
        for v in range(nodes):
            if g[u][v] > 0 and mstSet[v] == False and key[v] > g[u][v]:
                key[v] = g[u][v]
                parent[v] = u
    printMST(g, nodes, parent)
primMST(adj, nodes)
```

```
# --Kruskal --
def search(parent, i):
    if parent[i] == i:
        return i
    return search(parent, parent[i])
def apply_union(parent, rank, x, y):
    xroot = search(parent, x)
    yroot = search(parent, y)
    if rank[xroot] < rank[yroot]:
        parent[xroot] = yroot
    elif rank[xroot] > rank[yroot]:
        parent[yroot] = xroot
    else:
        parent[yroot] = xroot
        rank[xroot] += 1
def kruskal(g, nodes):
    result = []
    i, e = 0, 0
    g = sorted(g, key=lambda item: item[2])
    parent = []
    rank = []
    for node in range(nodes):
        parent.append(node)
        rank.append(0)
    while e < nodes - 1:
        u, v, w = g[i]
        i = i + 1
        x = search(parent, u)
        y = search(parent, v)
        if x != y:
            e = e + 1
            result.append([u, v, w])
            apply_union(parent, rank, x, y)
    for u, v, weight in result:
        print("Edge:", u, v, end=" ")
        print("-", weight)
```

```
# 2 input of Kruskal
g = {
    (0, 1, 8),
    (0, 2, 5),
    (1, 2, 9),
    (1, 3, 11),
    (2, 3, 15),
    (2, 4, 10),
    (3, 4, 7),
}
kruskal(g, nodes)
```

```
# --Seles man --
from sys import maxsize
from itertools import permutations
def travellingSalesmanProblem(graph, node, start):
    vertex = [x for x in range(node) if node != start]
    min_path = maxsize
    next_permutation = permutations(vertex)
    for i in next_permutation:
        current_pathweight = 0
        k = start
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][start]
        min_path = min(min_path, current_pathweight)
    return min_path
print(travellingSalesmanProblem(adj, nodes,start))
```

```
# -- Floyd warshall ---
import sys
class Floyd:
    reslut: list = []
    def int(self, p):
        self.p = p
    def floyed1(self, n, w, x, y):
        d = [[0 for i in range(n)] for j in range(n)]
        self.p = [[0 for i in range(n)] for j in range(n)]
        for i in range(n):
            for j in range(n):
                d[i][j] = w[i][j]
                self.p[i][j] = -1
        for k in range(n):
            for i in range(n):
                for j in range(n):
                    if d[i][k] + d[k][j] < d[i][j]:
                        d[i][j] = d[i][k] + d[k][j]
                        self.p[i][j] = k
    #  print(d[x][y])
    def rec(self, u, v):
        self.reslut.append(u + 1)
        self.pathutil(u, v)
        self.reslut.append(v + 1)
    def pathutil(self, u, v):
        if (self.p[u][v]) == -1:
            return
        else:
            self.pathutil(u, self.p[u][v])
            self.reslut.append(self.p[u][v] + 1)
            self.pathutil(self.p[u][v], v)
```

```
# 2 main of Floyd warshall
n, e = map(int, input().split(' '))
arr = [[0] * n for rr in range(n)]
for i in range(len(arr)):
    for j in range(len(arr[0])):
        if i != j:
            arr[i][j] = sys.maxsize
for _ in range(e):
    u, v, w = map(int, input().split(' '))
    arr[u - 1][v - 1] = w
    arr[v - 1][u - 1] = w
g = Floyd()
g.floyed1(n, arr, 1 - 1, n - 1)
g.rec(1 - 1, n - 1)
rs = g.reslut
for i in range(1, n + 1):
    if i in rs:
        print('all')
    else:
        print('none')
```

```
# 1 --chinses Postman or rout inspection --
def sum_edges(graph):
    w_sum = 0
    l = len(graph)
    for i in range(l):
        for j in range(i, l):
            w_sum += graph[i][j]
    return w_sum


# 3 ---
def get_odd(graph):
    degrees = [0 for i in range(len(graph))]
    for i in range(len(graph)):
        for j in range(len(graph)):
            if (graph[i][j] != 0):
                degrees[i] += 1
    # print(degrees)
    odds = [i for i in range(len(degrees)) if degrees[i] % 2 != 0]
    # print('odds are:',odds)
    return odds
def gen_pairs(odds):
    pairs = []
    for i in range(len(odds) - 1):
        pairs.append([])
        for j in range(i + 1, len(odds)):
            pairs[i].append([odds[i], odds[j]])
    return pairs

# 5 -----
n, e = map(int, input().split())
adj = [[0] * n for _ in range(n)]
for i in range(e):
    u, v, w = map(int, input().split())
    adj[u - 1][v - 1] = w
    adj[v - 1][u - 1] = w
print(Chinese_Postman(adj))
```

```
# 2 ------
def dijktra(graph, source, dest):
    shortest = [0 for i in range(len(graph))]
    selected = [source]
    l = len(graph)
    inf = 10000000
    min_sel = inf
    ind = 0
    for i in range(l):
        if i == source:
            shortest[source] = 0  # graph[source][source]
        else:
            if graph[source][i] == 0:
                shortest[i] = inf
            else:
                shortest[i] = graph[source][i]
                if shortest[i] < min_sel:
                    min_sel = shortest[i]
                    ind = i
    if source == dest:
        return 0
    selected.append(ind)
    while ind != dest:
        # print('ind',ind)
        for i in range(l):
            if i not in selected:
                if graph[ind][i] != 0:
                    # Check if distance needs to be updated
                    if (graph[ind][i] + min_sel) < shortest[i]:
                        shortest[i] = graph[ind][i] + min_sel
        temp_min = 1000000
        for j in range(l):
            if j not in selected:
                if shortest[j] < temp_min:
                    temp_min = shortest[j]
                    ind = j
        min_sel = temp_min
        selected.append(ind)
    return shortest[dest]
```

```
# 4 ---
def Chinese_Postman(graph):
    odds = get_odd(graph)
    if (len(odds) == 0):
        return sum_edges(graph)
    pairs = gen_pairs(odds)
    l = (len(pairs) + 1) // 2
    pairings_sum = []
    def get_pairs(pairs, done=[], final=[]):
        if pairs[0][0][0] not in done:
            done.append(pairs[0][0][0])
            for i in pairs[0]:
                f = final[:]
                val = done[:]
                if i[1] not in val:
                    f.append(i)
                else:
                    continue
                if len(f) == l:
                    pairings_sum.append(f)
                    return
                else:
                    val.append(i[1])
                    get_pairs(pairs[1:], val, f)
        else:
            get_pairs(pairs[1:], done, final)
    get_pairs(pairs)
    min_sums = []
    for i in pairings_sum:
        s = 0
        for j in range(len(i)):
            s += dijktra(graph, i[j][0], i[j][1])
        min_sums.append(s)
    added_dis = min(min_sums)
    chinese_dis = added_dis + sum_edges(graph)
    return chinese_dis
```

**# -- by part or no ---**

```
class Graph():
    def __init__(self, V):
        self.V = V
        self.graph = [[0]*V for row in range(V)]
    def isBipartite(self, src):
        colorArr = [-1] * self.V
        colorArr[src] = 1
        queue = []
        queue.append(src)
        while queue:
            u = queue.pop()
            if self.graph[u][u] == 1:
                return False
            for v in range(self.V):
                if self.graph[u][v] == 1 and colorArr[v] == -1:
                    colorArr[v] = 1 - colorArr[u]
                    queue.append(v)
                elif self.graph[u][v] == 1 and colorArr[v] == colorArr[u]:
                    return False
        return True
g = Graph(4)
g.graph = [[0, 1, 0, 1],
           [1, 0, 1, 0],
           [0, 1, 0, 1],
           [1, 0, 1, 0]]
print("Yes" if g.isBipartite(0) else "No")
```

**# -- number of Tringle in directed or undirected graph--**

```
def countTriangle(g, isDirected):
    nodes = len(g)
    count_Triangle = 0
    for i in range(nodes):
        for j in range(nodes):
            for k in range(nodes):
                if i != j and i != k and j != k and g[i][j] and g[j][k] and g[k][i]:
                    count_Triangle += 1
    if isDirected:
        return count_Triangle // 3
    else:
        return count_Triangle // 6
print(countTriangle(adj, False))
```

**# -- Journey to moon**

```
N, I = map(int, input().strip().split())
assert 1 <= N <= 10 ** 5
assert 1 <= I <= 10 ** 6
lis_of_sets = []
for i in range(I):
    a, b = map(int, input().strip().split())
    assert 0 <= a < N and 0 <= b < N
    indices = []
    new_set = set()
    set_len = len(lis_of_sets)
    s = 0
    while s < set_len:
        if a in lis_of_sets[s] or b in lis_of_sets[s]:
            indices.append(s)
            new_set = new_set.union(lis_of_sets[s])
            del lis_of_sets[s]
            set_len -= 1
        else:
            s += 1
    new_set = new_set.union([a, b])
    lis_of_sets.append(new_set)
answer = N * (N - 1) / 2
for i in lis_of_sets:
    answer -= len(i) * (len(i) - 1) / 2
print(int(answer))
```

**#--vertex Cover problem---**

```
from collections import defaultdict
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(list)
    def addEdge(self, u, v):
        self.graph[u].append(v)
    def printVertexCover(self):
        visited = [False] * (self.V)
        for u in range(self.V):
            if not visited[u]:
                for v in self.graph[u]:
                    if not visited[v]:
                        visited[v] = True
                        visited[u] = True
                        break
        for j in range(self.V):
            if visited[j]:
                print(j, end=' ')
        print()
n,e = map(int,input().split())
g = Graph(n)
for i in range(e):
    s,d = map(int,input().split())
    g.addEdge(s,d)
g.printVertexCover()
```

```
# --Tow Clique Problem --
# -- sub tree is a sub tree --
from queue import Queue
def isBipartiteUtil(G, nodes, src, colorArr):
    colorArr[src] = 1
    q = Queue()
    q.put(src)
    while not q.empty():
        u = q.get()
        for v in range(nodes):
            if G[u][v] and colorArr[v] == -1:
                colorArr[v] = 1 - colorArr[u]
                q.put(v)
            elif G[u][v] and colorArr[v] == colorArr[u]:
                return False
    return True
def isBipartite(G, nodes):
    colorArr = [-1] * nodes
    for i in range(nodes):
        if colorArr[i] == -1:
            if not isBipartiteUtil(G, nodes, i, colorArr):
                return False
    return True
def canBeDividedinTwoCliques(G, nodes):
    GC = [[None] * nodes for i in range(nodes)]
    for i in range(nodes):
        for j in range(nodes):
            GC[i][j] = not G[i][j] if i != j else 0
    return isBipartite(GC, nodes)
adj = [[0, 1, 1, 1, 0],
       [1, 0, 1, 0, 0],
       [1, 1, 0, 0, 0],
       [0, 1, 0, 0, 1],
       [0, 0, 0, 1, 0]]
if canBeDividedinTwoCliques(adj, 5):
    print("Yes")
else:
    print("No")
```

```
# -- Frind chash flow --
def getMin(arr, n):
    minInd = 0
    for i in range(1, n):
        if arr[i] < arr[minInd]:
            minInd = i
    return minInd
def getMax(arr, n):
    maxInd = 0
    for i in range(1, n):
        if arr[i] > arr[maxInd]:
            maxInd = i
    return maxInd
def minOf2(x, y):
    return x if x < y else y
def minCashFlowRec(amount, n):
    mxCredit = getMax(amount, n)
    mxDebit = getMin(amount, n)
    if amount[mxCredit] == 0 and amount[mxDebit] == 0:
        return 0
    min = minOf2(-amount[mxDebit], amount[mxCredit])
    amount[mxCredit] -= min
    amount[mxDebit] += min
    print("Person ", mxDebit, " pays ", min, " to ", "Person ", mxCredit)
    minCashFlowRec(amount, n)
def minCashFlow(graph, n):
    amount = [0 for i in range(n)]
    for p in range(n):
        for i in range(n):
            amount[p] += (graph[i][p] - graph[p][i])
    minCashFlowRec(amount, n)
graph = [[0, 1000, 2000],
         [0, 0, 5000],
         [0, 0, 0]]
n = 3
minCashFlow(graph, n)
```

**# 1 ---minimum reverse make path source to distention --**

```
def addEdge(u, v, w):
    global adj
    adj[u].append((v, w))
def shortestPath(src):
    setds = {}
    dist = [10 ** 18 for i in range(n)]
    global adj
    setds[(0, src)] = 1
    dist[src] = 0
    while (len(setds) > 0):
        tmp = list(setds.keys())[0]
        del setds[tmp]
        u = tmp[1]
        for i in adj[u]:
            v = i[0];
            weight = i[1]
            if (dist[v] > dist[u] + weight):
                if (dist[v] != 10 ** 18):
                    del setds[(dist[v], v)]
                dist[v] = dist[u] + weight
                setds[(dist[v], v)] = 1
    return dist
def modelGraphWithEdgeWeight(edge, E, V):
    global adj
    for i in range(E):
        addEdge(edge[i][0], edge[i][1], 0)
        addEdge(edge[i][1], edge[i][0], 1)
def getMinEdgeReversal(edge, E, V, src, dest):
    modelGraphWithEdgeWeight(edge, E, V)
    dist = shortestPath(src)
    if (dist[dest] == 10 ** 18):
        return -1
    else:
        return dist[dest]
```

**# 2 ---------**

```
n, e = map(int,input().split())
edge = []
for i in range(e):
    sur , des = map(int,input().split())
    edge.append([sur,des])
adj =[[] for i in range(n+1)]
minEdgeToReverse = getMinEdgeReversal(edge, e, n,
0, 6)
if (minEdgeToReverse != -1):
    print(minEdgeToReverse)
else:
    print("Not possible")
```

**# -- is a path more then k from source to distention  length --**

```
def pathMoreThanK(graph, nodes, src, k):
    path = [False] * nodes
    path[src] = True
    return pathMoreThanKUtil(graph, src, k, path)
def pathMoreThanKUtil(adj, src, k, path):
    if k <= 0:
        return True
    i = 0
    while i != len(adj[src]):
        v = adj[src][i][0]
        w = adj[src][i][1]
        i += 1
        if path[v]:
            continue
        if w >= k:
            return True
        path[v] = True
        if pathMoreThanKUtil(adj, v, k - w, path):
            return True
        path[v] = False
    return False
n, e = map(int, input().split())
adj = [[] * n for _ in range(n)]
for _ in range(e):
    u, v, w = map(int, input().split())
    adj[u].append([v, w])
    adj[v].append([u, w])
sr, k = map(int, input().split())
x = pathMoreThanK(adj, n, sr, k)
if x:
    print(True)
else:
    print(False)
```

**#---water in juga problems---**

```
def BFS(a, b, target):
    m = {}
    isSolvable = False
    path = []
    result = list()
    q = deque()
    q.append((0, 0))
    while len(q) > 0:
        u = q.popleft()
        if (u[0], u[1]) in m:
            continue
        if ((u[0] > a or u[1] > b or
             u[0] < 0 or u[1] < 0)):
            continue
        path.append([u[0], u[1]])
        m[(u[0], u[1])] = 1
        if u[0] == target or u[1] == target:
            isSolvable = True
            if u[0] == target:
                if u[1] != 0:
                    path.append([u[0], 0])
            else:
                if u[0] != 0:
                    path.append([0, u[1]])
            sz = len(path)
            for i in range(sz):
                result.append((path[i][0], path[i][1]))
            break
```

**# 2 into while scope--**

```
q.append([u[0], b])
        q.append([a, u[1]])
        for ap in range(max(a, b) + 1):
            c = u[0] + ap
            d = u[1] - ap
            if c == a or (d == 0 and d >= 0):
                q.append([c, d])
            c = u[0] - ap
            d = u[1] + ap
            if (c == 0 and c >= 0) or d == b:
                q.append([c, d])
        q.append([a, 0])
        q.append([0, b])
    if not isSolvable:
        print("No solution")
    else:
        return result
Jug1, Jug2, target = 4, 3, 2
x = BFS(Jug1, Jug2, target)
print(len(x))
```

**# -- 2 Ford Flacons main --**

```
n = int(input())
s, d, e = map(int, input().strip().split())
gr = [[0] * n for i in range(n)]
for i in range(e):
    u, v, w = map(int, input().strip().split())
    gr[u - 1][v - 1] = w
g = Graph(gr)
source = s-1
sink = d-1
print("Max Flow: %d " % g.ford_fulkerson(source, sink))
```

**# -- Ford-Fulkerson algorithm ---max falow**

```
class Graph:
    def __init__(self, graph):
        self.graph = graph
        self.ROW = len(graph)
    def searching_algo_BFS(self, s, t, parent):
        visited = [False] * (self.ROW)
        queue = []
        queue.append(s)
        visited[s] = True
        while queue:
            u = queue.pop(0)
            for ind, val in enumerate(self.graph[u]):
                if visited[ind] == False and val > 0:
                    queue.append(ind)
                    visited[ind] = True
                    parent[ind] = u
        return True if visited[t] else False
    # Applying fordfulkerson algorithm
    def ford_fulkerson(self, source, sink):
        parent = [-1] * (self.ROW)
        max_flow = 0
        while self.searching_algo_BFS(source, sink, parent):
            path_flow = float("Inf")
            s = sink
            while (s != source):
                path_flow = min(path_flow, self.graph[parent[s]][s])
                s = parent[s]
            max_flow += path_flow
            v = sink
            while (v != source):
                u = parent[v]
                self.graph[u][v] -= path_flow
                self.graph[v][u] += path_flow
                v = parent[v]
        return max_flow
```

```
# -- the longest path for source to other -
# -- nodes cycle directed weighted graph -
def topologicalSortUtil(v):
    global Stack, visited, adj
    visited[v] = True
    for i in adj[v]:
        if not visited[i[0]]:
            topologicalSortUtil(i[0])
    Stack.append(v)
def longestPath(s):
    global Stack, visited, adj, V
    dist = [-10 ** 9 for i in range(V)]
    for i in range(V):
        if not visited[i]:
            topologicalSortUtil(i)
    dist[s] = 0
    while len(Stack) > 0:
        u = Stack[-1]
        del Stack[-1]
        # print(u)
        if dist[u] != 10 ** 9:
            for i in adj[u]:
                # print(u, i)
                if dist[i[0]] < dist[u] + i[1]:
                    dist[i[0]] = dist[u] + i[1]
    for i in range(V):
        print("INF ", end="") if (dist[i] == -10 ** 9) else print(dist[i], end=" ")
```

```
# 2 main ----------
V = 6
Stack = []
visited = [False for i in range(V + 1)]
adj = [[] for iti in range(V + 1)]
adj[0].append([1, 5])
adj[0].append([2, 3])
adj[1].append([3, 6])
adj[1].append([2, 2])
adj[2].append([4, 4])
adj[2].append([5, 2])
adj[2].append([3, 7])
adj[3].append([5, 1])
adj[3].append([4, -1])
adj[4].append([5, -2])
s = 1
print("Following are longest distances from source vertex ", s)
longestPath(s)
```

```
# - Detect Negative Cycle by bellman ford ---
class Edge:
    def __init__(self):
        self.src = 0
        self.dest = 0
        self.weight = 0
class Graph:
    def __init__(self):
        self.V = 0
        self.E = 0
        self.edge = None
def createGraph(V, E):
    graph = Graph()
    graph.edge = [Edge() for i in range(graph.E)]
    return graph
def isNegCycleBellmanFord(graph, node, edges, src):
    V, E = node, edges
    dist = [1000000 for i in range(V)]
    dist[src] = 0
    for i in range(1, V):
        for j in range(E):
            u = graph.edge[j].src
            v = graph.edge[j].dest
            weight = graph.edge[j].weight
            if dist[u] != 1000000 and dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight
    for i in range(E):
        u = graph.edge[i].src
        v = graph.edge[i].dest
        weight = graph.edge[i].weight
        if dist[u] != 1000000 and dist[u] + weight < dist[v]:
            return True
    return False
```

```
# 2 main ---
V, E = map(int, input().split())
graph = createGraph(V, E)
for i in range(E):
    s, d, w = map(int, input().split(' '))
    graph.edge[i].src = s
    graph.edge[i].dest = d
    graph.edge[i].weight = w

if isNegCycleBellmanFord(graph, V, E, 0):
    print("Yes")
else:
    print("No")
```

```
# -- Strongly Connected Component ---
from collections import defaultdict
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(list)
    def addEdge(self, u, v):
        self.graph[u].append(v)
    def DFSUtil(self, v, visited):
        visited[v] = True
        if v is not None:
            print(v)
        for i in self.graph[v]:
            if not visited[i]:
                self.DFSUtil(i, visited)
    def fillOrder(self, v, visited, stack):
        visited[v] = True
        for i in self.graph[v]:
            if not visited[i]:
                self.fillOrder(i, visited, stack)
        stack = stack.append(v)
    def getTranspose(self):
        g = Graph(self.V)
        for i in self.graph:
            for j in self.graph[i]:
                g.addEdge(j, i)
        return g
```

```
# 2 ----
def printSCCs(self):
        global strong
        stack = []
        visited = [False] * self.V
        for i in range(self.V):
            if not visited[i]:
                self.fillOrder(i, visited, stack)
        gr = self.getTranspose()
        visited = [False] * self.V
        while stack:
            i = stack.pop()
            if not visited[i]:
                gr.DFSUtil(i, visited)
                print()
n, e = map(int, input().split(' '))
g = Graph(n)
for i in range(e):
    u, v = map(int, input().split())
    g.addEdge(u, v)
print(g.printSCCs())
```

```
# --- Find Bridge in a Graph ----
brige = []
class Graph:
    global brige
    def __init__(self, graph, nodes):
        self.nodes = nodes
        self.graph = graph
        self.Time = 0
    def bridgeUtil(self, u, visited, parent, low, disc):
        visited[u] = True
        disc[u] = self.Time
        low[u] = self.Time
        self.Time += 1
        for v in self.graph[u]:
            if not visited[v]:
                parent[v] = u
                self.bridgeUtil(v, visited, parent, low, disc)
                low[u] = min(low[u], low[v])
                if low[v] > disc[u]:
                    brige.append((u, v))
            elif v != parent[u]:
                low[u] = min(low[u], disc[v])
    def bridge(self):
        visited = [False] * self.nodes
        disc = [sys.maxsize] * self.nodes
        low = [sys.maxsize] * self.nodes
        parent = [-1] * self.nodes
        brig = list()
        for i in range(self.nodes):
            if not visited[i]:
                self.bridgeUtil(i, visited, parent, low, disc)
```

```
# 2 main ---
t = int(input())
while t > 0:
    n, e = map(int, input().split())
    G = defaultdict(list)
    for _ in range(e):
        u, v = map(int, input().split())
        G[u].append(v)
        G[v].append(u)
    g = Graph(G, n)
    g.bridge()
    print(brige)
    brige = []
    t -= 1
```

```
# -- Snake and ladder --
for t in range(int(input())):
    L = int(input())
    ladders = []
    for i in range(L):
        a, b = map(int, input().strip().split())
        ladders.append([a, b])
    S = int(input())
    snakes = []
    for i in range(S):
        a, b = map(int, input().strip().split())
        snakes.append([a, b])
    ladders.extend(snakes)
    D = {}
    for a, b in ladders:
        D[a] = b
    V = set()  # visited squares
    S = set()
    S.add(1)
    moves = 0
    while 100 not in S:
        moves += 1
        S2 = set()
        for a in S:
            for d in range(1, 6 + 1):
                n = a + d
                if n in D:
                    n = D[n]
                if n in V:
                    continue
                V.add(n)
                S2.add(n)
        S = S2
    print(moves)
```

```
# ---Total number of MST in a graph ----
def findDeterminant(Matrix):
    det = 0
    if len(Matrix) == 1:
        return Matrix[0][0]
    elif len(Matrix) == 2:
        det = (Matrix[0][0] * Matrix[1][1] - Matrix[0][1] * Matrix[1][0])
        return det
    else:
        for p in range(len(Matrix[0])):
            tempMatrix = []
            for i in range(1, len(Matrix)):
                row = []
                for j in range(0, len(Matrix[i])):
                    if j != p:
                        row.append(Matrix[i][j])
                if len(row) > 0:
                    tempMatrix.append(row)
            det = det + Matrix[0][p] * pow(-1, p) * findDeterminant(tempMatrix)
        return det
```

```
# -- 2 mst----
def Total_spanningTrees(adjMatrix, n):
    degree = [0 for i in range(n)]
    for i in range(n):
        for j in range(n):
            if adjMatrix[i][j] == 1:
                degree[i] += 1
    for i in range(n):
        adjMatrix[i][i] = degree[i]
    for i in range(n):
        for j in range(n):
            if i != j and adjMatrix[i][j] == 1:
                adjMatrix[i][j] = -1
    sub_matrix = [[0 for i in range(n - 1)] for j in range(n - 1)]
    for i in range(n):
        for j in range(n):
            sub_matrix[i - 1][j - 1] = adjMatrix[i][j]
    return findDeterminant(sub_matrix)
t = int(input())
while t > 0:
    n, e = map(int, input().split())
    adj = [[0] * n for i in range(n)]
    for i in range(e):
        u, v, w = map(str, input().split())
        u = int(ord(u) - ord('a'))
        v = int(ord(v) - ord('a'))
        adj[u][v] = 1
        adj[v][u] = 1
    print(Total_spanningTrees(adj, n))
    t -= 1
```

```
# -- flood fill algorithm --
def floodFill(image, sr, sc, newColor):
    R, C = len(image), len(image[0])
    color = image[sr][sc]
    if color == newColor:
        return image
    def dfs(r, c):
        if image[r][c] == color:
            image[r][c] = newColor
            if r >= 1:
                dfs(r - 1, c)
            if r + 1 < R:
                dfs(r + 1, c)
            if c >= 1:
                dfs(r, c - 1)
            if c + 1 < C:
                dfs(r, c + 1)
    dfs(sr, sc)
    return image
```

```
# --Making  wired connection ---
# --how may connection fix to make network
def DFS(adj, node, visited):
    if visited[node]:
        return
    visited[node] = True
    if node in adj:
        for x in adj[node]:
            if not visited[x]:
                DFS(adj, x, visited)
def make_con(node, G_adj, edge):
    visited = [False] * node
    adj = {}
    edges = 0
    for i in range(edge):
        if G_adj[i][0] in adj:
            adj[G_adj[i][0]].append(G_adj[i][1])
        else:
            adj[G_adj[i][0]] = []
        if G_adj[i][1] in adj:
            adj[G_adj[i][1]].append(G_adj[i][0])
        else:
            adj[G_adj[i][1]] = []
        edges += 1
    components = 0
    for i in range(node):
        if not visited[i]:
            components += 1
            DFS(adj, i, visited)
    if edges < node - 1:
        return -1
    redundant = edges - ((node - 1) - (components - 1))
    if redundant >= (components - 1):
        return components - 1
    return -1
```

```
# -- Generate all path and value---
def allPath(G, src, dest, path=[]):
    path = path + [src]
    Paths = []
    if src == dest:
        return [path]
    if src not in G:
        return []
    for neb in G[src]:
        if neb not in path:
            newPath = allPath(G, neb, dest, path)
            for u in newPath:
                Paths.append(u)
    return Paths
def costOFpath(paths, val):
    costs = list()
    for i in range(len(paths)):
        cost = []
        for j in range(len(paths[i]) - 1):
            x = (paths[i][j], paths[i][j + 1])
            cost.append(x)
        c = 0
        for y in cost:
            for p in val:
                if y[0] == p[0] and y[1] == p[1]:
                    c += p[2]
                    break
        costs.append(c)
    return costs
```

```
#--Rat in a Maze--
def setup():
    global v
    v = [[0] * 100 for _ in range(100)]
    global ans
    ans = []
def path(arr, x, y, pth, n):
    if x == n - 1 and y == n - 1:
        global ans
        ans.append(pth)
        return
    global v
    if arr[x][y] == 0 or v[x][y] == 1:
        return
    v[x][y] = 1
    if x > 0:
        path(arr, x - 1, y, pth + 'U', n)
    if y > 0:
        path(arr, x, y - 1, pth + 'L', n)
    if x < n - 1:
        path(arr, x + 1, y, pth + 'D', n)
    if y < n - 1:
        path(arr, x, y + 1, pth + 'R', n)
    v[x][y] = 0
def findPath(matrix, size):
    global ans
    ans = []
    if matrix[0][0] == 0 or matrix[size - 1][size - 1] == 0:
        return ans
    setup()
    path(matrix, 0, 0, "", size)
    ans.sort()
    return ans
```

```
# 2 rate in maz
n = int(input().strip())
maz = list()
for i in range(n):
    row = list(map(int, input().strip().split(' ')))
    maz.append(row)
print(*findPath(maz, n))
```

```
# --Find Maximum step knight --
def isValid(x, y, N):
    return N > x >= 0 and N > y >= 0
def minStepToReachTarget(KnightPos, TargetPos, N):
    dxy = [[2, 1], [2, -1], [-2, 1], [-2, -1], [1, 2], [1, -2], [-1, 2], [-1, -2]]
    KnightPos[0] -= 1
    KnightPos[1] -= 1
    TargetPos[0] -= 1
    TargetPos[1] -= 1
    vis = [[False] * N for _ in range(N)]
    q = deque()
    q.append([KnightPos[0], KnightPos[1], 0])
    vis[KnightPos[0]][KnightPos[1]] = True
    while len(q) != 0:
        cur = q.popleft()
        x = cur[0]
        y = cur[1]
        steps = cur[2]
        if x == TargetPos[0] and y == TargetPos[1]:
            return steps
        for i in range(8):
            n_x = x + dxy[i][0]
            n_y = y + dxy[i][1]

            if isValid(n_x, n_y, N) and not vis[n_x][n_y]:
                q.append([n_x, n_y, steps + 1])
                vis[n_x][n_y] = True
    return -1
x = minStepToReachTarget(knig,Target,n)
```

```
#--Word ladder alien dictionary alg ----
from collections import deque
def shortestChainLen(src, dist, words):
    if src == dist:
        return 0
    if dist not in words:
        return 0
    level, wordLength = 0, len(src)
    Q = deque()
    Q.append(src)
    while len(Q) > 0:
        level += 1
        sizeofQ = len(Q)
        for i in range(sizeofQ):
            word = [j for j in Q.popleft()]
            for pos in range(wordLength):
                orig_char = word[pos]
                for c in range(ord('a'), ord('z') + 1):
                    word[pos] = chr(c)
                    if "".join(word) == dist:
                        return level + 1
                    if "".join(word) not in words:
                        continue
                    del words["".join(word)]
                    Q.append("".join(word))
                word[pos] = orig_char
    return 0
words = {"poon": 1, "plee": 1, "same": 1, "poie": 1, "plie": 1, "poin": 1, "plea": 1, }
print(shortestChainLen('toon', 'plea', words))
```

```
#---topological sort--
def DFSUtil(graph, node, visited, stack):
    visited[node] = True
    for i in graph[node]:
        if not visited[i]:
            DFSUtil(graph, i, visited, stack)
    stack.insert(0, node)

def topoSort(nodes, graph):
    visited = [False] * nodes
    stack = []
    for i in range(nodes):
        if not visited[i]:
            DFSUtil(graph, i, visited, stack)
    return stack
```

**#---witch job finish get another job --**

```
def printOrder(graph, n):
    indegree = [0] * (n + 1)
    for i in graph:
        for j in graph[i]:
            indegree[j] += 1
    job = [0] * (n + 1)
    q = []
    for i in range(1, n + 1):
        if indegree[i] == 0:
            q.append(i)
            job[i] = 1
    while q:
        cur = q.pop(0)
        for adj in graph[cur]:
            indegree[adj] -= 1
            if indegree[adj] == 0:
                job[adj] = 1 + job[cur]
                q.append(adj)
    for i in range(1, n + 1):
        print(job[i], end=" ")
```

**#--possible to finished all task--**

```
def dfs_cycle(graph, node, onpath, visited):
    if visited[node]:
        return False
    onpath[node] = visited[node] = True
    for neigh in graph[node]:
        if onpath[neigh] or dfs_cycle(graph, neigh, onpath, visited):
            return True
    return False
def canFinish(Tasks, prerequisites):
    graph = []
    for i in range(Tasks):
        graph.append([])
    for first, second in prerequisites:
        graph[second].append(first)
    onpath = [False] * Tasks
    visited = [False] * Tasks
    for i in range(Tasks):
        if not visited[i] and dfs_cycle(graph, i, onpath, visited):
            return False
    return True
task, per = map(int, input().strip().split(' '))
pre_task = list()
for i in range(per):
    f, s = map(int, input().strip().split(' '))
    pre_task.append((f, s))
print(f"Possible ->{canFinish(task, pre_task)}")
```

**#--Number of Island---**

```
import sys
sys.setrecursionlimit(10 ** 8)
class Solution:
    def numIslands(self, arr):
        # code here
        visited = [[0] * len(arr[0]) for i in range(len(arr))]
        def isValid(x, y):
            if 0 <= x < n and 0 <= y < m:
                return True
            return False
        def dfs(grid, x, y):
            visited[x][y] = 1
            for i in [[-1, -1], [1, 1], [1, 0], [0, 1], [1, -1], [-1, 1], [-1, 0], [0, -1]]:
                if isValid(x + i[0], y + i[1]) and visited[x + i[0]][y + i[1]] == 0:
                    if grid[x + i[0]][y + i[1]] == 1:
                        dfs(grid, x + i[0], y + i[1])
        count = 0
        for i in range(len(arr)):
            for j in range(len(arr[0])):
                if visited[i][j] == 0 and arr[i][j] == 1:
                    dfs(arr, i, j)
                    count += 1
        return count
n, m = map(int, input().strip().split(' '))
arr = []
for i in range(n):
    arr.append(list(map(int, input().strip().split(' '))))
s = Solution()
x = s.numIslands(arr)
print(x)
```

# Dynamic

```
# -- 0_1_Knapsak  --
val = [5, 4, 7, 7]
wt = [5, 6, 8, 4]
n1 = 4
n2 = 13
W = 13
n = len(val)
K = [[0 for x in range(W + 1)] for x in range(n + 1)]
for i in range(n + 1):
   for w in range(W + 1):
      if i == 0 or w == 0:
         K[i][w] = 0
      elif wt[i - 1] <= w:
         K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w])
      else:
         K[i][w] = K[i - 1][w]

print(K[n1][n2])
j = n2
for i in range(n1, 0, -1):
   if K[i][j] != K[i - 1][j]:
      print(i, end=" ")
      j -= wt[i - 1]
print()
```

```
#--- Making anagram ---
a = "bcadeh"
b = "hea"
buffer = [0] * 26
for char in a:
   buffer[ord(char) - ord('a')] += 1
for char in b:
   buffer[ord(char) - ord('a')] -= 1
print(sum(map(abs, buffer)))
```

```
# - Cutting road ---
"""
1 2 3 4 5 6 7 8
3 5 8 9 10 17 17 20
8
"""
ind = list(int(i) for i in input().split(' '))
val = list(int(i) for i in input().split(' '))
n = int(input())
result = []
for i in range(len(ind)):
   sub = []
   sub.append(ind[i])
   sub.append(val[i])
   sub.append(val[i]/ind[i])
   result.append(sub)
result.sort(key=lambda x:x[2],reverse=True)
print(result)
sum1 = 0
for i in range(len(val)):
   while result[i][0] <= n:
      sum1 +=result[i][1]
      n = n - result[i][0]
print(sum1)
```

```
#-- word brake --
def wordBreak(words, word, out=''):
   if not word:
      print(out)
      return
   for i in range(1, len(word) + 1):
      prefix = word[:i]
      if prefix in words:
         wordBreak(words, word[i:], out + ' ' + prefix)
words = [
      'self', 'th', 'is', 'famous', 'Word', 'break', 'b', 'r',
      'e', 'a', 'k', 'br', 'bre', 'brea', 'ak', 'problem'
   ]
word = 'Wordbreakproblem'
wordBreak(words, word)
```

```
#-- space capitalize --
import re
st = "BruceWayneIsBatman"
words = re.findall('[A-Z][a-z]*', st)
for i in words:
   st = i
   st = st.lower()
   print(st,end=" ")
print()
```

```
#-- job sequencing--
arr = [['1', 2, 100],  # Job Array
      ['2', 1, 19],
      ['3', 2, 27],
      ['4', 1, 25],
      ['5', 3, 15]]
n = len(arr)
arr.sort(key=lambda x:x[2],reverse=True)
t = 2
result = [False]*t
job = [-1]*t
job1 = [0]*t
for i in range(n):
      for j in range(min(t-1,arr[i][1]-1),-1,-1):
            if result[j] is False:
                  result[j]=True
                  job[j]=arr[i][0]
                  print(job)
                  job1[j]=arr[i][2]
                  break
print(job)
print(job1)
print(sum(job1))
```

```
#--- edit distance ----
def dist(X, m, Y, n):
   if m == 0:
      return n
   if n == 0:
      return m
   cost = 0 if (X[m - 1] == Y[n - 1]) else 1
   return min(dist(X, m - 1, Y, n) + 1,
           dist(X, m, Y, n - 1) + 1,
           dist(X, m - 1, Y, n - 1) + cost)
X = 'kitten'
Y = 'sitting'
print('The Levenshtein distance is', dist(X, len(X), Y, len(Y)))
```

```
# -- LCS -
def lcs(X, Y, m, n):
   L = [[0 for i in range(n + 1)] for j in range(m + 1)]
   for i in range(m + 1):
      for j in range(n + 1):
         if i == 0 or j == 0:
            L[i][j] = 0
         elif X[i - 1] == Y[j - 1]:
            L[i][j] = L[i - 1][j - 1] + 1
         else:
            L[i][j] = max(L[i - 1][j], L[i][j - 1])
   for i in L:
      print(*i)
   print(X)
   print(Y)
   lcs = ""
   i = m
   j = n
   while i > 0 and j > 0:
      if X[i - 1] == Y[j - 1]:
         lcs += X[i - 1]
         i -= 1
         j -= 1
      elif L[i - 1][j] > L[i][j - 1]:
         i -= 1
      else:
         j -= 1
   lcs = lcs[::-1]
   print("LCS of " + X + " and " + Y + " is " + lcs)
X = "AGGTAB"
Y = "GXTXAYB"
m = len(X)
n = len(Y)
lcs(X, Y, m, n)
```

```
#---Matrix chain multiplication--
import sys
def MatrixChainOrder(p, i, j):
   if i == j:
      return 0
   _min = sys.maxsize
   for k in range(i, j):
      count = (MatrixChainOrder(p, i, k) + MatrixChainOrder(p, k + 1, j) + p[i - 1] * p[k] * p[j])
      if count < _min:
         _min = count
   return _min
arr = [1, 2, 3, 4, 3]
n = len(arr)
print("Minimum number of multiplications is ", MatrixChainOrder(arr, 1, n - 1))
```

**#--- 1 Counter customer ----**

```
import java.util.Scanner;
public class BnkQ {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print(process(input));
        input.close();
    }
    private static int[] split(String line) {
        return new int[]{ //
            Integer.parseInt(line.substring(0, line.indexOf(" ")).trim()),
            nteger.parseInt(line.substring(line.indexOf(" ") + 1).trim()) //
        };
    }

 # into BanKQ class scope
 private static void debug(Queue last) {
        // System.out.println((" ".repeat(last.delay)) + ("#".repeat(last.weight)));
    }
}
```

**# 2 ---**

```
public static String process(Scanner input) {
    // test case count
    int tsCount = Integer.parseInt(input.nextLine().trim());
    String result = "";
    while (tsCount-- > 0) {
        // counter and customer count
        int[] ccCount = split(input.nextLine());
        Queue[] queues = new Queue[ccCount[0]];
        int count = ccCount[1];
        Queue lastAppendQueue = null;
        outer:
        while (count-- > 0) {
            // delay weight
            int[] dw = split(input.nextLine());
            Queue shortestQueue = null;
            for (int i = 0; i < queues.length; i++) {
                Queue queue = queues[i];
                if (queue == null) {
                    queues[i] = new Queue();
                    queue = queues[i];
                    queue.delay = dw[0] + (lastAppendQueue == null ? 0 : lastAppendQueue.delay);
                    queue.weight = dw[1];
                    queue.updateFinish();
                    lastAppendQueue = queue;
                    debug(lastAppendQueue);
                    continue outer;
                } else if (shortestQueue == null) {
                    shortestQueue = queue;
                } else if (queue.finish < shortestQueue.finish) {
                    shortestQueue = queue;
                }
            }
```

**# 3 ---**

```
int arrivedAt = lastAppendQueue.delay + dw[0];
            if (arrivedAt < shortestQueue.finish) {
                arrivedAt = shortestQueue.finish;
            }
            shortestQueue.delay = arrivedAt;
            shortestQueue.weight = dw[1];
            shortestQueue.updateFinish();
            debug(shortestQueue);
            lastAppendQueue = shortestQueue;
        }
        int lastCustomerFinish = -1;
        for (Queue queue : queues) {
            lastCustomerFinish = Math.max(queue.finish, lastCustomerFinish);
        }
        result += lastCustomerFinish + (tsCount > 0 ? "\n" : "");
    }
    return result;
}
```

```
class Queue {
    int delay;
    int weight;
    int finish;
    public void updateFinish() {
        this.finish = this.delay + this.weight;
    }
}
```

**# -- Maximum sum of rectangular of 2D matrix--**

```
# Given a 2D matrix M of dimensions RxC. Find the maximum sum submatrix in it.
def kadanes(arr, n):
    s, maxi = arr[0], arr[0]
    for i in range(1, n):
        s += arr[i]
        s = max(s, arr[i])
        maxi = max(s, maxi)
    return maxi
def maximumSumRectangle(R, C, M):
    res = M[0][0]
    for starti in range(R):
        subMatSum = [0 for _ in range(C)]
        for i in range(starti, R):
            for j in range(C):
                subMatSum[j] += M[i][j]
            res = max(res, kadanes(subMatSum, C))
    return res
R = 4
C = 5
M = [[1, 2, -1, -4, -20],
     [-8, -3, 4, 2, 1],
     [3, 8, 10, 1, 3],
     [-4, -1, 1, 7, -6]]
print(maximumSumRectangle(R, C, M))
```

**# --interleaved String --**

```
def isInterleave(a, b, c):
    m = len(a)
    n = len(b)
    dp = [[False] * (n + 1) for i in range(m + 1)]
    if m + n != len(c):
        return False
    for i in range(0, m + 1):
        for j in range(0, n + 1):
            if i == 0 and j == 0:
                dp[i][j] = True
            elif i == 0:
                if b[j - 1] == c[j - 1]:
                    dp[i][j] = dp[i][j - 1]
            elif j == 0:
                if a[i - 1] == c[i - 1]:
                    dp[i][j] = dp[i - 1][j]
            elif a[i - 1] == c[i + j - 1] and b[j - 1] != c[i + j - 1]:
                dp[i][j] = dp[i - 1][j]
            elif a[i - 1] != c[i + j - 1] and b[j - 1] == c[i + j - 1]:
                dp[i][j] = dp[i][j - 1]
            elif a[i - 1] == c[i + j - 1] and b[j - 1] == c[i + j - 1]:
                dp[i][j] = (dp[i - 1][j] or dp[i][j - 1])
    return dp[m][n]
a = 'XY'
b = 'X'
c = 'XXY'
print(isInterleave(a, b, c))
```

**# ---maximum profit buying and selling-----**

```
# a person buys a stock and sells it on some future date
def maxProfit(K, N, A):
    profit = [[0 for i in range(N + 2)] for i in range(K + 2)]
    for i in range(K + 1):
        profit[i][0] = 0
    for j in range(N + 1):
        profit[0][j] = 0
    INT_MIN = -1 * (1 << 32)
    for i in range(1, K + 1):
        prevDiff = INT_MIN
        for j in range(1, N):
            prevDiff = max(prevDiff, profit[i - 1][j - 1] - A[j - 1])
            profit[i][j] = max(profit[i][j - 1], A[j] + prevDiff)
    return profit[K][N - 1]
K = 3
N = 4
A = [20, 580, 420, 900]
print(maxProfit(K, N, A))
```

**# --- longest Pair chain------**

```
def findLongestChain(pairs):
    pairs.sort()
    dp = [1] * len(pairs)
    for j in range(len(pairs)):
        for i in range(j):
            if pairs[i][1] < pairs[j][0]:
                dp[j] = max(dp[j], dp[i] + 1)
    return max(dp)
pairs = [[1, 2], [2, 3], [3, 4]]
print(findLongestChain(pairs))
```

**# -- Largest area rectangular sub-matrix 0,1 matrix ---**

```
# Given a binary matrix. The problem is to find the largest area rectangular
# sub-matrix with equal number of 1's and 0's. Examples:
def maximalRectangleArea(A):
    n = len(A)
    m = len(A[0])
    for i in range(m):
        s = 0
        for j in range(n):
            if A[j][i] == 1:
                s += 1
                A[j][i] = s
            else:
                s = 0
    resultArea = 0
    for i in range(n):
        firstMinLeft = []
        firstMinRight = []
        st = []
        for j in range(m):
            while st and st[-1][0] >= A[i][j]:
                st.pop()
            if st:
                firstMinLeft.append(st[-1][1])
            else:
                firstMinLeft.append(-1)
            st.append((A[i][j], j))
        st = []
        for j in range(m - 1, -1, -1):
            while st and st[-1][0] >= A[i][j]:
                st.pop()
            if st:
                firstMinRight.append(st[-1][1])
            else:
                firstMinRight.append(m)
            st.append((A[i][j], j))
```

**# -- 2 into maxmalRec func scop ---**

```
firstMinRight = firstMinRight[::-1]
        for j in range(m):
            area = (firstMinRight[j] - firstMinLeft[j] - 1) * A[i][j]
            resultArea = max(area, resultArea)
    return resultArea
A = [[1, 1, 1],
     [0, 1, 1],
     [1, 0, 0]]
print(maximalRectangleArea(A))
```

**# --- Largest rectangular sub-matrix whose sum is 0 ---**

```
# Given a matrix mat[][] of size N x M.
# The task is to find the largest rectangular sub-matrix by area whose sum is 0.
def largest_rectanglar_sum_is_0(A):
    if not A:
        return 0
    C = len(A[0])
    col_expanded = []
    for row in A:
        expanded = []
        for i in range(C):
            s = 0
            for j in range(i, C):
                s += row[j]
                expanded.append(s)
        col_expanded.append(expanded)
    total = 0
    for i in range(len(col_expanded[0])):
        for j in range(len(col_expanded)):
            s = 0
            for k in range(j, len(col_expanded)):
                s += col_expanded[k][i]
                if s == 0:
                    total += 1
    return total
mat = [[9, 7, 16, 5],
       [1, -6, -7, 3],
       [1, 8, 7, 9],
       [7, -2, 0, 10]]
print(largest_rectanglar_sum_is_0(mat))
```

```
# --- how many way to Boolean Presentation ---
# Count the number of ways we can parenthesize
# the expression so that the value of expression evaluates to true.
def countWays(N, S):
    mod = 1003
    dp = []
    for i in range(201):
        temp = []
        for j in range(201):
            x = [-1] * 2
            temp.append(x)
        dp.append(temp)
    def solve(S, i, j, isTrue):
        if i > j:
            dp[i][j][isTrue] = 0
            return 0
        if i == j:
            if isTrue:
                if S[i] == 'T':
                    dp[i][j][isTrue] = 1
                else:
                    dp[i][j][isTrue] = 0
            else:
                if S[i] == 'F':
                    dp[i][j][isTrue] = 1
                else:
                    dp[i][j][isTrue] = 0
            return dp[i][j][isTrue]
        if dp[i][j][isTrue] != -1:
            return dp[i][j][isTrue]
        ans = 0
        # 2 into count way def scop --
        for k in range(i + 1, j, 2):
            if S[k] == '|':
                if isTrue:
                    ans += ((solve(S, i, k - 1, 1) * solve(S, k + 1, j, 1))
                            + (solve(S, i, k - 1, 1) * solve(S, k + 1, j, 0))
                            + (solve(S, i, k - 1, 0) * solve(S, k + 1, j, 1))) % mod
                else:
                    ans += (solve(S, i, k - 1, 0) * solve(S, k + 1, j, 0)) % mod
            elif S[k] == '&':
                if isTrue:
                    ans += (solve(S, i, k - 1, 1) * solve(S, k + 1, j, 1)) % mod
                else:
                    ans += ((solve(S, i, k - 1, 0) * solve(S, k + 1, j, 0))
                            + (solve(S, i, k - 1, 0) * solve(S, k + 1, j, 1))
                            + (solve(S, i, k - 1, 1) * solve(S, k + 1, j, 0))) % mod
            else:
                if isTrue:
                    ans += ((solve(S, i, k - 1, 0) * solve(S, k + 1, j, 1))
                            + (solve(S, i, k - 1, 1) * solve(S, k + 1, j, 0))) % mod
                else:
                    ans += ((solve(S, i, k - 1, 1) * solve(S, k + 1, j, 1))
                            + (solve(S, i, k - 1, 0) * solve(S, k + 1, j, 0))) % mod
        dp[i][j][isTrue] = ans % mod
        return dp[i][j][isTrue]
    return solve(S, 0, N - 1, 1)
S = 'T^F|F'
print(countWays(len(S), S))
```

```
# --- Mobile numeric keypad ----
# Given the mobile numeric keypad.You can only press buttons
# that are up, left, right, or down to the current button or
# the current button itself (like 00,11, etc.). You are not
# allowed to press the bottom row corner buttons (i.e. * and # ).
# Given a number N, the task is to find out the number of
# possible numbers of the given length
def solve(i, j, n, keypad, dp):
    if n == 1:
        return 1
    if dp[keypad[i][j]][n] != -1:
        return dp[keypad[i][j]][n]
    a = solve(i, j, n - 1, keypad, dp)
    if j - 1 >= 0 and keypad[i][j - 1] != -1:
        a += solve(i, j - 1, n - 1, keypad, dp)
    if j + 1 < 3 and keypad[i][j + 1] != -1:
        a += solve(i, j + 1, n - 1, keypad, dp)
    if i - 1 >= 0 and keypad[i - 1][j] != -1:
        a += solve(i - 1, j, n - 1, keypad, dp)
    if i + 1 < 4 and keypad[i + 1][j] != -1:
        a += solve(i + 1, j, n - 1, keypad, dp)
    dp[keypad[i][j]][n] = a
    return a
def getCount(n):
    ans = 0
    keypad = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [-1, 0, -1]]
    dp = [[-1 for _ in range(n + 1)] for _ in range(10)]
    for i in range(4):
        for j in range(3):
            if keypad[i][j] != -1:
                ans += solve(i, j, n, keypad, dp)
    return ans
print(getCount(1))  # --> 10
print(getCount(2))  # --> 36
```

**# -- word wrap problem ---**

```
# Given an array nums[] of size n,
# where nums[i] denotes the number of characters in one word.
# Let K be the limit on the number of characters that can be
# put in one line (line width).
# Put line breaks in the given sequence such that the lines are
# printed neatly.
import sys
def solveWordWrap(nums, k):
    n = len(nums)
    dp = [0] * n
    ans = [0] * n
    dp[n - 1] = 0
    ans[n - 1] = n - 1
    for i in range(n - 2, -1, -1):
        currlen = -1
        dp[i] = sys.maxsize
        for j in range(i, n):
            currlen += (nums[j] + 1)
            if currlen > k:
                break
            if j == n - 1:
                cost = 0
            else:
                cost = ((k - currlen) * (k - currlen) + dp[j + 1])
            if cost < dp[i]:
                dp[i] = cost
                ans[i] = j
```

**# 2 into def solve word fun scope ---**

```
l = 0
    res = 0
    while i < n:
        pos = 0
        for j in range(i, ans[i] + 1):
            pos = pos + nums[j]
        x = ans[i] - i
        if ans[i] + 1 != n:
            res = res + (k - x - pos) ** 2
        i = ans[i] + 1
    return res
words = [3, 2, 2, 5]
k = 6  # number of character can be one line
print(solveWordWrap(words, k))  # -> 10 line
```

**# -- Palindromic Partitioning ---**

```
# Given a string str,a partitioning of the string is a palindrome
# partitioning if every sub-string of the partition is a palindrome.
# Determine the fewest cuts needed for palindrome partitioning of the
# given string
# Input: str = "ababbbabbababa"
# Output: 3
# After 3 partitioning substrings
# are "a", "babbbab", "b", "ababa".
def palindromicPartition(s):
    n = len(s)
    C = [0] * n
    P = [[False for i in range(n)] for i in range(n)]
    for i in range(n):
        P[i][i] = True
    for L in range(2, n + 1):
        for i in range(n - L + 1):
            j = i + L - 1
            if L == 2:
                P[i][j] = (s[i] == s[j])
            else:
                P[i][j] = ((s[i] == s[j]) & P[i + 1][j - 1])
    for i in range(n):
        if P[0][i]:
            C[i] = 0
        else:
            C[i] = (1 << 32)
            for j in range(i):
                if P[j + 1][i] == True and C[j] + 1 < C[i]:
                    C[i] = C[j] + 1
    return C[n - 1]
print(palindromicPartition('ababbbabbababa'))
```

**# --- optimal binary search tree problem ---**

```
def optCost(freq, i, j):
    if j < i:
        return 0
    if j == i:
        return freq[i]
    fsum = Sum(freq, i, j)
    Min = 999999999999
    for r in range(i, j + 1):
        cost = (optCost(freq, i, r - 1) +
                optCost(freq, r + 1, j))
        if cost < Min:
            Min = cost
    return Min + fsum
def optimalSearchTree(freq, n):
    return optCost(freq, 0, n - 1)
def Sum(freq, i, j):
    s = 0
    for k in range(i, j + 1):
        s += freq[k]
    return s
keys = [10, 12, 20]
freq = [34, 8, 50]
n = len(keys)
print("Cost of Optimal BST is", optimalSearchTree(freq, n))
```

```
# -- Optimal Strategy For A Game ------
# You are given an array A of size N.
# The array contains integers and is of even length.
# The elements of the array represent N coin of values V1, V2, ....Vn.
# You play against an opponent in an alternating way
def optimalStrategyOfGame(arr, n):
    table = [[0 for i in range(n)] for i in range(n)]
    for gap in range(n):
        for j in range(gap, n):
            i = j - gap
            x = 0
            if (i + 2) <= j:
                x = table[i + 2][j]
            y = 0
            if (i + 1) <= (j - 1):
                y = table[i + 1][j - 1]
            z = 0
            if i <= (j - 2):
                z = table[i][j - 2]
            table[i][j] = max(arr[i] + min(x, y), arr[j] + min(y, z))
    return table[0][n - 1]
A = [5, 3, 7, 10]
print(optimalStrategyOfGame(A, len(A)))  # --> 15
```

```
# --- Count Derangement's ----
# (Permutation such that no element appears in its original position)
def countDer(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return (n - 1) * (countDer(n - 1) + countDer(n - 2))
n = 4
print("Count of Derangement's is ", countDer(n))
```

```
# ---- Maximum profit by buying and selling a share at most twice ------
# In daily share trading, a buyer buys shares
# in the morning and sells them on the same day.
# If the trader is allowed to make at most 2 transactions in a day,
# whereas the second transaction can only start after the first one
# is complete (Buy->sell->Buy->sell). Given stock prices throughout the day,
# find out the maximum profit that a share trader could have made.
# Returns maximum profit with
# two transactions on a given
# list of stock prices price[0..n-1]
def maxProfit(price, n):
    profit = [0] * n
    max_price = price[n - 1]
    for i in range(n - 2, 0, -1):
        if price[i] > max_price:
            max_price = price[i]
        profit[i] = max(profit[i + 1], max_price - price[i])
    min_price = price[0]
    for i in range(1, n):
        if price[i] < min_price:
            min_price = price[i]
        profit[i] = max(profit[i - 1], profit[i] + (price[i] - min_price))
    result = profit[n - 1]
    return result
price = [2, 30, 15, 10, 8, 25, 80]
print(maxProfit(price, len(price)))  # --> 100
```

```
# ---- Coin game winner where every player has three choices -----
# A and B are playing a game. At the beginning there are n coins.
# Given two more numbers x and y. In each move a player can pick x or y or 1 coins.
# A always starts the game. The player who picks the last coin wins the
# game or the person who is not able to pick any coin loses the game.
# For a given value of n, find whether A will win the game or not if both are playing optimally.
# Python3 program to find winner of game
# if player can pick 1, x, y coins
def findWinner(x, y, n):
    dp = [0 for i in range(n + 1)]
    dp[0] = False
    dp[1] = True
    for i in range(2, n + 1):
        if i - 1 >= 0 and not dp[i - 1]:
            dp[i] = True
        elif i - x >= 0 and not dp[i - x]:
            dp[i] = True
        elif i - y >= 0 and not dp[i - y]:
            dp[i] = True
        else:
            dp[i] = False
    return dp[n]
x = 3
y = 4
n = 5
if findWinner(x, y, n):
    print('A')
else:
    print('B')
```

```
# -- longest alternating sub sequence --
# A sequence {x1, x2, .. xn} is alternating sequence if its
# elements satisfy one of the following relations :
# x1 < x2 > x3 < x4 > x5..... or  x1 >x2 < x3 > x4 < x5.....
# Your task is to find the longest such sequence.
def AlternatingaMaxLength(arr):
    inc = 1
    dec = 1
    n = len(arr)
    for i in range(1, n):
        if arr[i] > arr[i - 1]:
            inc = dec + 1
        elif arr[i] < arr[i - 1]:
            dec = inc + 1
    return max(inc, dec)
nums = [1, 5, 4]
print(AlternatingaMaxLength(nums)) # --> 3
```

```
# -- weighted job Scheduling max profit
from functools import cmp_to_key
class Job:
    def __init__(self, start, finish, profit):
        self.start = start
        self.finish = finish
        self.profit = profit
def jobComparator(s1, s2):
    return s1.finish < s2.finish
def latestNonConflict(arr, i):
    for j in range(i - 1, -1, -1):
        if arr[j].finish <= arr[i - 1].start:
            return j
    return -1
def findMaxProfitRec(arr, n):
    if n == 1:
        return arr[n - 1].profit
    inclProf = arr[n - 1].profit
    i = latestNonConflict(arr, n)
    if i != -1:
        inclProf += findMaxProfitRec(arr, i + 1)
    exclProf = findMaxProfitRec(arr, n - 1)
    return max(inclProf, exclProf)
def findMaxProfit(arr, n):
    arr = sorted(arr, key=cmp_to_key(jobComparator))
    return findMaxProfitRec(arr, n)
values = [(3, 10, 20),
          (1, 2, 50),
          (6, 19, 100),
          (2, 100, 200)]
arr = []
for i in values:
    arr.append(Job(i[0], i[1], i[2]))
n = len(arr)
print("The optimal profit is", findMaxProfit(arr, n))
```

```
# -- longest palindrome  --
# Given a string s, return the longest palindromic substring in s.
def longestPalindrome(s):
    if len(s) <= 1:
        return s
    start = end = 0
    length = len(s)
    for i in range(length):
        max_len_1 = get_max_len(s, i, i + 1)
        max_len_2 = get_max_len(s, i, i)
        max_len = max(max_len_1, max_len_2)
        if max_len > end - start:
            start = i - (max_len - 1) // 2
            end = i + max_len // 2
    return s[start: end + 1]
def get_max_len(s, left, right):
    length = len(s)
    i = 1
    while left >= 0 and right < length and s[left] == s[right]:
        left -= 1
        right += 1
    return right - left - 1
s = "babad"
print(longestPalindrome(s))
```

```
# -- Count Palindromic Subsequences ----
# Given a string str of length N,
# you have to find number of palindromic subsequence
# (need not necessarily be distinct) present in the string str.
# Note: You have to return the answer module 109+7;
def countPalindromicSebcuences(s):
    t = [[-1 for i in range(1001)] for i in range(1001)]
    mod = 10 ** 9 + 7
    def solve(s, i, j, t):
        if i == j:
            return 1
        if i > j:
            return 0
        if t[i][j] != -1:
            return t[i][j]
        elif s[i] == s[j]:
            t[i][j] = 1 + solve(s, i + 1, j, t) % mod + solve(s, i, j - 1, t) % mod
            t[i][j] %= mod
            return t[i][j]
        else:
            t[i][j] = solve(s, i + 1, j, t) % mod + solve(s, i, j - 1, t) % mod - solve(s, i + 1, j - 1, t) % mod
            t[i][j] %= mod
            return t[i][j]
    return solve(s, 0, len(s) - 1, t)
s = "aab"
print(countPalindromicSebcuences(s))  # --> 4
```

```
# -- longest palindromic substring ---
def lps(seq, i, j):
    if i == j:
        return 1
    if seq[i] == seq[j] and i + 1 == j:
        return 2
    if seq[i] == seq[j]:
        return lps(seq, i + 1, j - 1) + 2
    return max(lps(seq, i, j - 1), lps(seq, i + 1, j))
seq = "GEEKSFORGEEKS" # --> 5
n = len(seq)
print("The length of the LPS is", lps(seq, 0, n - 1))
```

```
# -- partition equal sub set sum --
# Given an array arr[] of size N,
# check if it can be partitioned into two parts
# such that the sum of elements in both parts is the same.
def equalPartition(n, arr):
    if sum(arr) & 1:
        return 0
    sumo = sum(arr) // 2 + 1
    dp = [[-1 for i in range(sumo)] for j in range(n + 1)]
    def solve(i, s):
        if i == 0:
            return 0
        if dp[i][s] != -1:
            return dp[i][s]
        if s == arr[i - 1]:
            dp[i][s] = 1
            return 1
        if s > arr[i - 1]:
            dp[i][s] = solve(i - 1, s - arr[i - 1]) | solve(i - 1, s)
        else:
            dp[i][s] = solve(i - 1, s)
        return dp[i][s]
    return solve(n, sumo - 1)
arr = [1, 5, 11, 5]
print(equalPartition(len(arr), arr))
```

```
#---- 0-1 Knapsack with Duplicate Items ----
def knapSack(N, W, val, wt):
    dp = [0 for i in range(W + 1)]
    for i in range(W + 1):
        for j in range(N):
            if wt[j] <= i:
                dp[i] = max(dp[i], dp[i - wt[j]] + val[j])
    return dp[W]
N = 4
W = 8
val = [1, 4, 5, 7]
wt = [1, 3, 4, 5]
print(knapSack(N, W, val, wt))
```

```
# ----- smallest sum ----
# contiguous (connect ot us) subarray
maxsize = int(1e9 + 7)
def smallestSumSubarr(arr, n):
    min_ending_here = maxsize
    min_so_far = maxsize
    for i in range(n):
        if (min_ending_here > 0):
            min_ending_here = arr[i]
        else:
            min_ending_here += arr[i]
        min_so_far = min(min_so_far, min_ending_here)
    return min_so_far
arr = [3, -4, 2, -3, -1, 7, -5]
n = len(arr)
print("Smallest sum:", smallestSumSubarr(arr, n))
```

```
# --- Kasane's Algorithm ---
# Given an array Arr[] of N integers.
# Find the contiguous sub-array(containing at least one number)
# which has the maximum sum and return its sum.
def maxSubArraySum(a, size):
    max_so_far = -9999999 - 1
    max_ending_here = 0
    for i in range(0, size):
        max_ending_here = max_ending_here + a[i]
        if max_so_far < max_ending_here:
            max_so_far = max_ending_here
        if max_ending_here < 0:
            max_ending_here = 0
    return max_so_far
Arr = [1, 2, 3, -2, 5]
print(maxSubArraySum(Arr, len(Arr)))
```

```
# --- Balanced Binary Tree counter ------
# Given a height h, count the maximum number of balanced
# binary trees possible with height h. Print the result modulo 109 + 7.
# Note : A balanced binary tree is one in which for every node,
# the difference between heights of left and right subtree is
# not more than 1.
# Python3 program to count number of balanced
# binary trees of height h.
def countBT(h):
    MOD = 1000000007
    dp = [0 for i in range(h + 1)]
    dp[0] = 1
    dp[1] = 1
    for i in range(2, h + 1):
        dp[i] = (dp[i - 1] * ((2 * dp[i - 2]) % MOD + dp[i - 1]) % MOD) % MOD
    return dp[h]
h = 3
print("No. of balanced binary trees of height is: " + str(countBT(h)))  # --> 15
```

```
    # -- coin change ---
    def count(coins, n, sum):
        if sum == 0:
            return 1
        if sum < 0:
            return 0
        if n <= 0:
            return 0
        return count(coins, n - 1, sum) + count(coins, n, sum - coins[n - 1])
    k, n = map(int, input().split(' '))
    coins = [int(i) for i in input().split(' ')]
    print(count(coins, n, k))
```

```
# -- Reach a given score ---
# Consider a game where a player can score 3 or 5 or 10 points in a move.
# Given a total score n,
# find number of distinct combinations to reach the given score.
# input : 8 , 20 , 13 output: 1, 4, 2
# Explanation
# For 1st example when n = 8 { 3, 5 } and {5, 3}
# are the two possible permutations but these represent the same
# combination. Hence, output is 1
def count(n):
    ways = [3, 5, 10]
    return solve(ways, 3, n)
def solve(ways, n, target):
    if target == 0:
        return 1
    if n == 0:
        return 0
    if ways[n - 1] <= target:
        return solve(ways, n, target - ways[n - 1]) + solve(ways, n - 1, target)
    else:
        return solve(ways, n - 1, target)
print(count(20))
```

```
# --Gold mine problem give matrix and traverse for good way achieve gold—
# --  Right down right or right up --
def collectGold(gold, x, y, n, m):
    if (x < 0) or (x == n) or (y == m):
        return 0
    rightUpperDiagonal = collectGold(gold, x - 1, y + 1, n, m)
    right = collectGold(gold, x, y + 1, n, m)
    rightLowerDiagonal = collectGold(gold, x + 1, y + 1, n, m)
    return gold[x][y] + max(max(rightUpperDiagonal, rightLowerDiagonal), right)
def getMaxGold(gold, n, m):
    maxGold = 0
    for i in range(n):
        goldCollected = collectGold(gold, i, 0, n, m)
        maxGold = max(maxGold, goldCollected)
    return maxGold
gold = [[1, 3, 1, 5],
        [2, 2, 4, 1],
        [5, 0, 2, 3],
        [0, 6, 1, 2]]
m, n = 4, 4
print(getMaxGold(gold, n, m))
```

```
# ----count subarray that mult less than k --
def productSubSeqCount(arr, k):
    n = len(arr)
    dp = [[0 for i in range(n + 1)]
          for j in range(k + 1)]
    for i in range(1, k + 1):
        for j in range(1, n + 1):
            dp[i][j] = dp[i][j - 1]
            if 0 < arr[j - 1] <= i:
                dp[i][j] += dp[i // arr[j - 1]][j - 1] + 1
    return dp[k][n]
A = [1, 2, 3, 4]
k = 10
print(productSubSeqCount(A, k))
```

```
# -- painting the fence problem –
# -- how may way the to paint the wall
def countWays(n, k):
    dp = [0] * (n + 1)
    mod = 1000000007
    dp[1] = k
    dp[2] = k * k
    for i in range(3, n + 1):
        dp[i] = ((k - 1) * (dp[i - 1] + dp[i - 2])) % mod
    return dp[n]
n = 3
k = 2
print(countWays(n, k))
```

```
# -- longest Sub seq With Diff One --
def longestSubseqWithDiffOne(arr, n):
    dp = [1 for i in range(n)]
    for i in range(n):
        for j in range(i):
            if (arr[i] == arr[j] + 1) or (arr[i] == arr[j] - 1):
                dp[i] = max(dp[i], dp[j] + 1)
    result = 1
    print(dp)
    for i in range(n):
        if result < dp[i]:
            result = dp[i]
    return result
arr = [1, 2, 3, 4, 5, 3, 2]
n = len(arr)
print(longestSubseqWithDiffOne(arr, n))
```

```
def equal_1(arr):
    max_arr = max(arr)
    min_arr = min(arr)
    diff = max_arr - min_arr
    count = 0
    while diff != 0:
        if diff >= 5:
            increment = 5
        elif 5 > diff >= 2:
            increment = 2
        else:
            increment = 1
        for i in range(0, len(arr)):
            if max_arr != arr[i]:
                arr[i] = arr[i] + increment
        count += 1
        max_arr = max(arr)
        min_arr = min(arr)
        diff = max_arr - min_arr
    return count
t = int(input())
while t > 0:
    n = int(input())
    arr = [int(i) for i in input().split(' ')]
    print(equal_1(arr))
    t -= 1
```

```
# -- Maximum Difference zeros ones binary string --
MAX = 100
def allones(s, n):
    co = 0
    for i in s:
        co += 1 if i == '1' else 0
    return co == n
def findlength(arr, s, n, ind, st, dp):
    if ind >= n:
        return 0
    if dp[ind][st] != -1:
        return dp[ind][st]
    if not st:
        dp[ind][st] = max(arr[ind] +findlength(arr, s, n, ind + 1, 1, dp),(findlength(arr, s, n, ind + 1, 0, dp)))
    else:
        dp[ind][st] = max(arr[ind] + findlength(arr, s, n, ind + 1, 1, dp), 0)
    return dp[ind][st]
def maxLen(s, n):
    if allones(s, n):
        return -1
    arr = [0] * MAX
    for i in range(n):
        arr[i] = 1 if s[i] == '0' else -1
    dp = [[-1] * 3 for _ in range(MAX)]
    return findlength(arr, s, n, 0, 0, dp)
s = "11000010001"
n = 11
print(maxLen(s, n))
```

```
# -- Maximum sum increasing sub sequence ---
def maxSumIS(arr, n):
    max = 0
    msis = [0 for x in range(n)]
    for i in range(n):
        msis[i] = arr[i]
    for i in range(1, n):
        for j in range(i):
            if arr[i] > arr[j] and msis[i] < msis[j] + arr[i]:
                msis[i] = msis[j] + arr[i]
    for i in range(n):
        if max < msis[i]:
            max = msis[i]
    return max
arr = [1, 101, 2, 3, 100, 4, 5]
n = len(arr)
print("Sum of maximum sum increasing " + "subsequence is " + str(maxSumIS(arr, n)))
```

```
# - max sub square --
R = 6
C = 5
def printMaxSubSquare(M):
    global R, C
    Max = 0
    S = [[0 for col in range(C)] for row in range(2)]
    for i in range(R):
        for j in range(C):
            Entrie = M[i][j]
            if Entrie:
                if j:
                    Entrie = 1 + min(S[1][j - 1], min(S[0][j - 1], S[1][j]))
            S[0][j] = S[1][j]
            S[1][j] = Entrie
            Max = max(Max, Entrie)
    print("Maximum size sub-matrix is: ")
    for i in range(Max):
        for j in range(Max):
            print("1", end=" ")
        print()
M = [[0, 1, 1, 0, 1],
     [1, 1, 0, 1, 0],
     [0, 1, 1, 1, 0],
     [1, 1, 1, 1, 0],
     [1, 1, 1, 1, 1],
     [0, 0, 0, 0, 0]]
printMaxSubSquare(M)
```

```
# -- Maximum Segments ---
def maximumSegments(n, a, b, c):
    dp = [-1] * (n + 10)
    dp[0] = 0
    for i in range(0, n):
        if dp[i] != -1:
            if i + a <= n:
                dp[i + a] = max(dp[i] + 1, dp[i + a])
            if i + b <= n:
                dp[i + b] = max(dp[i] + 1, dp[i + b])
            if i + c <= n:
                dp[i + c] = max(dp[i] + 1, dp[i + c])
    return dp[n]
n = 7
a = 5
b = 2
c = 5
print(maximumSegments(n, a, b, c))
```

```
# ----- find max path ---
def findMaxPath(mat):
    for i in range(1, N):
        res = -1
        for j in range(M):
            if j > 0 and j < (M - 1):
                mat[i][j] += max(mat[i - 1][j], max(mat[i - 1][j - 1], mat[i - 1][j + 1]))
            elif j > 0:
                mat[i][j] += max(mat[i - 1][j], mat[i - 1][j - 1])
            elif j < M - 1:
                mat[i][j] += max(mat[i - 1][j], mat[i - 1][j + 1])
            res = max(mat[i][j], res)
    return res
N = 4
M = 6
mat = ([[10, 10, 2, 0, 20, 4],
        [1, 0, 0, 30, 2, 5],
        [0, 10, 4, 0, 2, 0],
        [1, 0, 2, 20, 0, 4]])
print(findMaxPath(mat))
```

```
def maxSumPairWithDifferenceLessThanK(arr, N, K):
    arr.sort()
    dp = [0] * N
    dp[0] = 0
    for i in range(1, N):
        dp[i] = dp[i - 1]
        if arr[i] - arr[i - 1] < K:
            if i >= 2:
                dp[i] = max(dp[i], dp[i - 2] + arr[i] + arr[i - 1])
            else:
                dp[i] = max(dp[i], arr[i] + arr[i - 1])
    return dp[N - 1]
arr = [3, 5, 10, 15, 17, 12, 9]
N = len(arr)
K = 4
print(maxSumPairWithDifferenceLessThanK(arr, N, K))
```

```
def maxSumWO3Consec(arr, n):
    sum = [0 for k in range(n)]
    if n >= 1:
        sum[0] = arr[0]
    if n >= 2:
        sum[1] = arr[0] + arr[1]
    if n > 2:
        sum[2] = max(sum[1], max(arr[1] + arr[2], arr[0] + arr[2]))
    for i in range(3, n):
        sum[i] = max(max(sum[i - 1], sum[i - 2] + arr[i]), arr[i] + arr[i - 1] + sum[i - 3])
    return sum[n - 1]
arr = [100, 1000, 100, 1000, 1]
n = len(arr)
print(maxSumWO3Consec(arr, n))
```

```
# --- ncr---
def nCr(n, r):
    return fact(n) / (fact(r) * fact(n - r))
def fact(n):
    if n == 0:
        return 1
    res = 1
    for i in range(2, n + 1):
        res = res * i
    return res
n, r = 5, 3
print(int(nCr(n, r)))
```

```
class val:
    def __init__(self, first, second):
        self.first = first
        self.second = second
def findMaxChainLen(p, n, prev, pos):
    global m
    if val(pos, prev) in m:
        return m[val(pos, prev)]
    if pos >= n:
        return 0
    if p[pos].first <= prev:
        return findMaxChainLen(p, n, prev, pos + 1)
    else:
        ans = max(findMaxChainLen(p, n, p[pos].second, 0) + 1, findMaxChainLen(p, n, prev, pos + 1))
        m[val(pos, prev)] = ans
        return ans
def maxChainLen(p, n):
    global m
    m.clear()
    ans = findMaxChainLen(p, n, 0, 0)
    return ans
n = 5
p = [0] * n
p[0] = val(5, 24)
p[1] = val(39, 60)
p[2] = val(15, 28)
p[3] = val(27, 40)
p[4] = val(50, 90)
m = {}
print(maxChainLen(p, n))
```

```
# --- min jumps ------
def minJumps(arr, l, h):
    if h == l:
        return 0
    if arr[l] == 0:
        return float('inf')
    min = float('inf')
    for i in range(l + 1, h + 1):
        if i < l + arr[l] + 1:
            jumps = minJumps(arr, i, h)
            if (jumps != float('inf') and
                    jumps + 1 < min):
                min = jumps + 1
    return min
arr = [1, 3, 6, 3, 2, 3, 6, 8, 9, 5]
n = len(arr)
print('Minimum number of jumps to reach', 'end is', minJumps(arr, 0, n - 1))
```

```
# -- Minimum Cost to fill given weighting the bag ---
INF = 1000000
def MinimumCost(cost, n, W):
    val = list()
    wt = list()
    size = 0
    for i in range(n):
        if cost[i] != -1:
            val.append(cost[i])
            wt.append(i + 1)
            size += 1
    n = size
    min_cost = [[0 for i in range(W + 1)] for j in range(n + 1)]
    for i in range(W + 1):
        min_cost[0][i] = INF
    for i in range(1, n + 1):
        min_cost[i][0] = 0
    for i in range(1, n + 1):
        for j in range(1, W + 1):
            if wt[i - 1] > j:
                min_cost[i][j] = min_cost[i - 1][j]
            else:
                min_cost[i][j] = min(min_cost[i - 1][j],min_cost[i][j - wt[i - 1]] + val[i - 1])
    if min_cost[n][W] == INF:
        return -1
    else:
        return min_cost[n][W]
cost = [1, 2, 3, 4, 5]
W = 5
n = len(cost)
print(MinimumCost(cost, n, W))
```

**# -- minimum removals from array to make max min equal k ---**

```
MAX = 100
dp = [[0 for i in range(MAX)]
     for i in range(MAX)]
for i in range(0, MAX):
   for j in range(0, MAX):
      dp[i][j] = -1
def countRemovals(a, i, j, k):
   global dp
   if i >= j:
      return 0
   elif (a[j] - a[i]) <= k:
      return 0
   elif dp[i][j] != -1:
      return dp[i][j]
   elif (a[j] - a[i]) > k:
      dp[i][j] = 1 + min(countRemovals(a, i + 1, j, k), countRemovals(a, i, j - 1, k))
   return dp[i][j]
def removals(a, n, k):
   a.sort()
   if n == 1:
      return 0
   else:
      return countRemovals(a, 0, n - 1, k)
a = [1, 3, 4, 9, 10,
    11, 12, 17, 20]
n = len(a)
k = 4
print(removals(a, n, k))
```

**# -- mobile number key pad ---**

```
ar = [
   '2', '22', '222',
   '3', '33', '333',
   '4', '44', '444', '4444',
   '5', '55', '555', '555',
   '6', '66', '666',
   '7', '77', '777', '7777',
   '8', '88', '888',
   '9', '99', '999', '9999'
]
st = input()
for i in st:
   index = ord(i) - 65
   print(ar[index], end='')
print()
```

**# -- partition equal sub set ---**

```
def isSubsetSum(arr, n, sum):
   if sum == 0:
      return True
   if n == 0 and sum != 0:
      return False
   if arr[n - 1] > sum:
      return isSubsetSum(arr, n - 1, sum)
   return isSubsetSum(arr, n - 1, sum) or isSubsetSum(arr, n - 1, sum - arr[n - 1])
def findPartion(arr, n):
   sum = 0
   for i in range(0, n):
      sum += arr[i]
   if sum % 2 != 0:
      return False
   return isSubsetSum(arr, n, sum // 2)
arr = [3, 1, 5, 9, 12]
n = len(arr)
if findPartion(arr, n) == True:
   print("Can be divided into two subsets of equal sum")
else:
   print("Can not be divided into two subsets of equal sum")
```

**# --- sherlock and cost --**

```
def cost(B):
   As = [[1, B[j]] for j in range(len(B))]
   print(As)
   score = [0, 0]
   new_score = []
   for i in range(1, len(B)):
      new_score = [0, 0]
      for p in [0, 1]:
         for q in [0, 1]:
            new_score[q] = max(new_score[q], score[p] + abs(As[i - 1][p] - As[i][q]))
      score[0] = new_score[0]
      score[1] = new_score[1]
   print(new_score)
   return max(score)
arr = [1, 2, 3]
print(cost(arr))
```

**# -- sub set sum ---**

```
def isSubsetSum(set, n, sum):
    if sum == 0:
        return True
    if n == 0:
        return False
    if set[n - 1] > sum:
        return isSubsetSum(set, n - 1, sum)
    return isSubsetSum(set, n - 1, sum) or isSubsetSum(set, n - 1, sum - set[n - 1])
set = [3, 34, 4, 12, 5, 2]
sum = 9
n = len(set)
if isSubsetSum(set, n, sum):
    print("Found a subset with given sum")
else:
    print("No subset with given sum")
```

**# - -minimum number of swap bracket balanced --**

```
def swapCount(s):
    swap = 0
    imbalance = 0
    for i in s:
        if i == '[':
            imbalance -= 1
        else:
            imbalance += 1
            if imbalance > 0:
                swap += imbalance
    return swap
s = "[]][]["
print(swapCount(s))
s = "[[][]]"
print(swapCount(s))
```

**# -- number of flips to make binary string alternate ---**

```
def flip(ch):
    return '1' if ch == '0' else '0'
def getFlipWithStartingCharcter(str, expected):
    flipCount = 0
    for i in range(len(str)):
        if str[i] != expected:
            flipCount += 1
        expected = flip(expected)
    return flipCount
def minFlipToMakeStringAlternate(str):
    return min(getFlipWithStartingCharcter(str, '0'),getFlipWithStartingCharcter(str, '1'))
str = "0001010111"
print(minFlipToMakeStringAlternate(str))
```

**# --- balanced prentices---**

```
st = input()
li = list()
if len(st) % 2 != 0:
    print("NO")
else:
    b = True
    for i in st:
        if i == "{" or i == "[" or i == "(":
            li.append(i)
        elif i == "}" or i == "]" or i == ")":
            if i == ")" and li.pop() != "(":
                b=False
                break
            elif i == "]" and li.pop() != "[":
                b=False
                break
            elif i == "}" and li.pop() != "{":
                b=False
                break
    if b:
        print("YES")
    else:
        print("NO")
```

**# -- boyar mores algorithm for pattern searching --**

```
NO_OF_CHARS = 256
def badCharHeuristic(string, size):
    badChar = [-1] * NO_OF_CHARS
    for i in range(size):
        badChar[ord(string[i])] = i
    return badChar
def search(txt, pat):
    m = len(pat)
    n = len(txt)
    badChar = badCharHeuristic(pat, m)
    s = 0
    while s <= n - m:
        j = m - 1
        while j >= 0 and pat[j] == txt[s + j]:
            j -= 1
        if j < 0:
            print("Pattern occur at shift = {}".format(s))
            s += (m - badChar[ord(txt[s + m])] if s + m < n else 1)
        else:
            s += max(1, j - badChar[ord(txt[s + j])])
txt = "ABAAABCD"
pat = "ABC"
search(txt, pat)
```

**# --- Rabin carap algorithm ---**

```
s = input()
s1 = input()
for i in range(len(s)):
    if s1 == s[i:len(s1)+i]:
        print(i)
```

```
# -- Decimal to roman numeral -----
ls = [1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1]
dict = {
   1: "I",
   4: "IV",
   5: "V",
   9: "IX",
   10: "X",
   40: "XL",
   50: "L",
   90: "XC",
   100: "C",
   400: "CD",
   500: "D",
   900: "CM",
   1000: "M"
}
ls2 = []
def func(no, res):
   rem = 0
   for i in range(0, len(ls)):
      if no in ls:
         res = dict[no]
         rem = 0
         break
      if ls[i] < no:
         quo = no // ls[i]
         rem = no % ls[i]
         res = res + dict[ls[i]] * quo
         break
   ls2.append(res)
   if rem == 0:
      pass
   else:
      func(rem, "")
func(3549, "")
print("".join(ls2))
```

```
# -- Roman to Decimal --
def value(r):
   if r == 'I':
      return 1
   if r == 'V':
      return 5
   if r == 'X':
      return 10
   if r == 'L':
      return 50
   if r == 'C':
      return 100
   if r == 'D':
      return 500
   if r == 'M':
      return 1000
   return -1
def romanToDecimal(str):
   res = 0
   i = 0
   while i < len(str):
      s1 = value(str[i])
      if i + 1 < len(str):
         s2 = value(str[i + 1])
         if s1 >= s2:
            res = res + s1
            i = i + 1
         else:
            res = res + s2 - s1
            i = i + 2
      else:
         res = res + s1
         i = i + 1
   return res
print(romanToDecimal("MCMIV"))
```

```
# -- longest common prefix --
def longestCommonPrefix(a):
   size = len(a)
   if size == 0:
      return ""
   if size == 1:
      return a[0]
   a.sort()
   end = min(len(a[0]), len(a[size - 1]))
   i = 0
   while (i < end and
         a[0][i] == a[size - 1][i]):
      i += 1
   pre = a[0][0: i]
   return pre
input = ["geeksforgeeks", "geeks", "geek", "geezer"]
print("The longest Common Prefix is :", longestCommonPrefix(input))
```

```
# -- longest suffix and prefix ---
s = input()
n = len(s)
b = False
for res in range(n // 2, 0, -1):
   prefix = s[0: res]
   suffix = s[n - res: n]
   if prefix == suffix:
      b=True
      print(res)
if not(b):
   print(0)
```

```
# -- next permutation ----
class Solution(object):
    def nextPermutation(self, nums):
        found = False
        i = len(nums) - 2
        while i >= 0:
            if nums[i] < nums[i + 1]:
                found = True
                break
            i -= 1
        if not found:
            nums.sort()
        else:
            m = self.findMaxIndex(i + 1, nums, nums[i])
            nums[i], nums[m] = nums[m], nums[i]
            nums[i + 1:] = nums[i + 1:][::-1]
        return nums
    def findMaxIndex(self, index, a, curr):
        ans = -1
        index = 0
        for i in range(index, len(a)):
            if a[i] > curr:
                if ans == -1:
                    ans = curr
                    index = i
                else:
                    ans = min(ans, a[i])
                    index = i
        return index
ob1 = Solution()
print(ob1.nextPermutation([1, 2, 3, 4, 5]))
```

```
# -- print all sub sequence ---
arr = list()
def allseq(s, out):
    if len(s) == 0:
        arr.append(out)
        return
    allseq(s[1:], out + s[0])
    allseq(s[1:], out)
s = input()
start, end = map(int, input().split(' '))
start -= 1
end -= 1
s = s[start:end]
out = ""
allseq(s, out)
print(arr)
```

```
# -- smallest distant window ---
import sys
s = list(input())
s1 = list(set(s))
print(s)
s1.sort()
print(s1)
min = sys.maxsize
for i in range(len(s)+1):
    for j in range(i+1,len(s)+1):
        sub = s[i:j]
        sub.sort()
        if len(sub) < min and s1 in s:
            min = len(sub)
print(min)
```

```
# -- Recursively print all sentences
# -- that can be formed from list of world list ---

R, C = 3, 3
def printUtil(arr, m, n, output):
    output[m] = arr[m][n]
    if m == R - 1:
        for i in range(R):
            print(output[i], end=" ")
        print()
        return
    for i in range(C):
        if arr[m + 1][i] != "":
            printUtil(arr, m + 1, i, output)
def printf(arr):
    output = [""] * R
    for i in range(C):
        if arr[0][i] != "":
            printUtil(arr, 0, i, output)
arr = [["you", "we", ""],
       ["have", "are", ""],
       ["sleep", "eat", "drink"]]
printf(arr)
```

```
# -- string rotation of another string --
s = input()
s1 = input()

if len(s) != len(s1):
    print("NO")
else:
    temp = s + s
    if temp.count(s1) > 0:
        print("YES")
    else:
        print("NO")
```

```
# -- generate all possible IP address from given string --
def is_valid(ip):
    ip = ip.split(".")
    for i in ip:
        if (len(i) > 3 or int(i) < 0 or
                int(i) > 255):
            return False
        if len(i) > 1 and int(i) == 0:
            return False
        if (len(i) > 1 and int(i) != 0 and
                i[0] == '0'):
            return False
    return True
def convert(s):
    sz = len(s)
    if sz > 12:
        return []
    snew = s
    l = []
    for i in range(1, sz - 2):
        for j in range(i + 1, sz - 1):
            for k in range(j + 1, sz):
                snew = snew[:k] + "." + snew[k:]
                snew = snew[:j] + "." + snew[j:]
                snew = snew[:i] + "." + snew[i:]
                if is_valid(snew):
                    l.append(snew)
                snew = s
    return l
A = "25525511135"
B = "25505011535"
print(convert(A))
print(convert(B))
```

```
# -- word wrap problem ---
n, s = map(int, input().split())
ar = list(map(int, input().split(',')))
b = False
for i in range(n + 1):
    for j in range(i + 1, n + 1):
        if sum(ar[i:j]) == s:
            print(i + 1, ' ', j)
            b = True
            break
    if b:
        break
if not (b):
    print(-1)
```

```
# -- sub string with equal string 0-1 --
s = input()
li = []
for i in range(len(s) + 1):
    for j in range(i + 1, len(s) + 1):
        sub = s[i:j]
        if len(sub) % 2 == 0:
            s1 = sub[:len(sub) // 2]
            s2 = sub[len(sub) // 2:]
            countz = s1.count('0')
            countone = s2.count('1')
            if len(sub) % 2 == 0 and countz == len(s1) and countone == len(s2):
                li.append(sub)
mx = 0
for i in li:
    if len(i) > mx:
        mx = len(i)
print(mx)
```

```
# -- wild card pattern matching ---
def strrmatch(strr, pattern, n, m):
    if m == 0:
        return n == 0
    lookup = [[False for i in range(m + 1)] for j in range(n + 1)]
    lookup[0][0] = True
    for j in range(1, m + 1):
        if pattern[j - 1] == '*':
            lookup[0][j] = lookup[0][j - 1]
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if pattern[j - 1] == '*':
                lookup[i][j] = lookup[i][j - 1] or lookup[i - 1][j]
            elif pattern[j - 1] == '?' or strr[i - 1] == pattern[j - 1]:
                lookup[i][j] = lookup[i - 1][j - 1]
            else:
                lookup[i][j] = False
    return lookup[n][m]
strr = "baaabab"
pattern = "*****ba*****ab"
if strrmatch(strr, pattern, len(strr), len(pattern)):
    print("Yes")
else:
    print("No")
```

```
# -- word break problem ---
def wordBreak(words, word, out=''):
    if not word:
        print(out)
        return
    for i in range(1, len(word) + 1):
        prefix = word[:i]
        if prefix in words:
            wordBreak(words, word[i:], out + '' + prefix)
ar = list(map(str, input().split(',')))
word = input()
wordBreak(ar, word)
```

```
# -- Transform one string to another
# using minimum number of given operation ---
def minOps(A, B):
    m, n = len(A), len(B)
    if n != m:
        return -1
    count = [0] * 256
    for i in range(n):
        count[ord(B[i])] += 1
    for i in range(n):
        count[ord(A[i])] -= 1
    for i in range(256):
        if count[i]:
            return -1
    res = 0
    i = n - 1
    j = n - 1
    while i >= 0:
        while i >= 0 and A[i] != B[j]:
            i -= 1
            res += 1
        if i >= 0:
            i -= 1
            j -= 1
    return res
A = "ABCD"
B = "EFGH"
print("Minimum number of operations required is " + str(minOps(A, B)))
```

```
global N #NQueen
N = 4
def printSolution(board):
    for i in range(N):
        for j in range(N):
            print(board[i][j], end=" ")
        print()
def isSafe(board, row, col):
    for i in range(col):
        if board[row][i] == 1:
            return False
    for i, j in zip(range(row, -1, -1),
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    for i, j in zip(range(row, N, 1),
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True
def solveNQUtil(board, col):
    if col >= N:
        return True
    for i in range(N):

        if isSafe(board, i, col):
            board[i][col] = 1
            if solveNQUtil(board, col + 1) == True:
                return True
            board[i][col] = 0
    return False
def solveNQ():
    board = [[0]*N for i in range(N)]
    if solveNQUtil(board, 0) == False:
        print("Solution does not exist")
        return False
    printSolution(board)
    return True
solveNQ()
```

```
# --- Game of thrones --------
st = input()
st1 = list(set(st))
res = []
for i in st1:
    res.append(st.count(i))
odd, even = 0, 0
for i in range(len(res)):
    if res[i] % 2 == 0:
        even += 1
    else:
        odd += 1
if odd <= 1:
    print("YES")
else:
    print("NO")
```

```
# -- alternating character ----
t = int(input())
while t > 0:
    st = list(input())
    for i in range(len(st) - 1):
        if st[i] == 'A' and st[i + 1] != 'B' or st[i] == 'B' and st[i + 1] != 'A':
            st[i] = ' '
    print(st.count(' '))
    t -= 1
```

```
# -- Highest value palindrome ---
n, k = map(int, input().split(' '))
s = input()
s = list(s)
n = len(s)
mark = [0] * n
l = 0
r = n - 1
while l <= r:
    if s[l] != s[r]:
        if s[l] > s[r]:
            s[r] = s[l]
            mark[r] = 1
        else:
            s[l] = s[r]
            mark[l] = 1
        k -= 1
    l += 1
    r -= 1
if k < 0:
    print(-1)
else:
    l = 0
    r = n - 1
    while l <= r:
        if l == r and k >= 1:
            s[l] = "9"
            break
        if s[l] < "9":
            if mark[l] == 0 and mark[r] == 0 and k >= 2:
                s[l] = s[r] = "9"
                k -= 2
            if (mark[l] == 1 or mark[r] == 1) and k >= 1:
                s[l] = s[r] = "9"
                k -= 1
        l += 1
        r -= 1
    print("".join(s))
```

```
# ---- Magic Squre----------
from itertools import permutations
X = []
X.extend(list(map(int,input().split())))
X.extend(list(map(int,input().split())))
X.extend(list(map(int,input().split())))
Ans = 81
for P in permutations(range(1,10)):
    if sum(P[0:3]) == 15 and sum(P[3:6]) == 15 and sum(P[0::3]) == 15 and
      sum(P[1::3]) == 15 and P[0] + P[4] + P[8] == 15 and (P[2] + P[4] + P[6] == 15):
        Ans = min(Ans, sum(abs(P[i] - X[i]) for i in range(0,9)))
print(Ans)
```

**# -- Maximum palindrome --**

```
from collections import defaultdict
def sherlockAndAnagrams(s):
    n = len(s)
    out = 0
    hash_map = defaultdict(lambda: 0)
    for i in range(n):
        for j in range(i + 1, n + 1):
            c = "".join(sorted(s[i:j]))
            out += hash_map[c]
            hash_map[c] += 1
    return out
s = input()
print(sherlockAndAnagrams(s))
```

**# --- Palindrome build ---**

```
def printSubStr(str, low, high):
    for i in range(low, high + 1):
        print(str[i], end="")
def longestPalSubstr(str):
    n = len(str)
    maxLength = 1
    start = 0
    for i in range(n):
        for j in range(i, n):
            flag = 1
            for k in range(0, ((j - i) // 2) + 1):
                if str[i + k] != str[j - k]:
                    flag = 0
            if flag != 0 and (j - i + 1) > maxLength:
                start = i
                maxLength = j - i + 1
    printSubStr(str, start, start + maxLength - 1)
    return maxLength
str = "bacbac"
print("Length is: ", longestPalSubstr(str))
```

**# -- palindrome index --**

```
def palindromeIndex(s):
    s = list(s)
    l, r = 0, len(s) - 1
    for i in range(len(s) // 2):
        if s[l] != s[r]:
            s.pop(l)
            if s == s[::-1]:
                return l
            else:
                return r
        l += 1
        r -= 1
    return -1
t = int(input())
res = []
while t > 0:
    st = list(input())
    print(palindromeIndex(st))
    t -= 1
```

**# -- separate the numbers ---**

```
q = int(input())
for __ in range(q):
    s = input().strip()
    if s[0] == '0':
        print("NO")
        continue
    found = False
    for pref in range(1, len(s) + 1):
        t = ""
        curnum = int(s[:pref])
        added = 0
        while len(t) < len(s):
            t += str(curnum)
            curnum += 1
            added += 1
        if added > 1 and t == s:
            print("YES", s[:pref])
            found = True
            break
    if not found:
        print("NO")
```

```
# --- sherlock and the valid string --
from collections import Counter
st = input()
dic = Counter(st)
most = Counter(dic.values())
ma = 0
key = ''
for i, j in most.items():
    if j > ma:
        ma = j
        key = i
count = 0
for i in dic.values():
    if i != key and i == 1:
        count += 1
    elif i != key:
        count += abs(int(i) - key)
if count <= 1:
    print("YES")
else:
    print("NO")
```

```
# -- sub string --
n, q = map(int, input().split(' '))
st = input()
st = list(st)
while q > 0:
    s, e = map(int, input().split(' '))
    res = list()
    for i in range(s, e + 1):
        for j in range(i + 1, e + 2):
            sub = st[i:j]
            if sub not in res:
                res.append(sub)
    print(len(res))
    q -= 1
```

```
# --- super reduce string ----
st = input()
while True:
    for i in range(len(st) - 1):
        if st[i] == st[i + 1]:
            st = st[0:i] + st[i + 2:]
            break
    b = False
    for i in range(len(st) - 1):
        if st[i] == st[i + 1]:
            b = True
    if not (b):
        break
if len(st) == 0:
    print("Empty String")
else:
    print(st)
```

```
# --- the love letter my story ---
t = int(input())
while t > 0:
    st = input()
    sum = 0
    for i in range(len(st) // 2):
        if ord(st[i]) != ord(st[-(i + 1)]):
            if ord(st[i]) > ord(st[-(i + 1)]):
                sum += ord(st[i]) - ord(st[-(i + 1)])
            elif ord(st[i]) < ord(st[-(i + 1)]):
                sum += ord(st[-(i + 1)]) - ord(st[i])
    print(sum)
    t -= 1
```

```
# -- tow string -------
n = int(input())
s = input()
assert s.isalpha()
ans = 0
for i in range(0, 26):
    for j in range(0, 26):
        if i == j:
            continue
        p1 = i
        p2 = j
        flag = 1
        l = 0
        for c in s:
            if ord(c) - ord('a') != p1 and ord(c) - ord('a') != p2:
                continue
            if ord(c) - ord('a') == p1:
                l = l + 1
                p1, p2 = p2, p1
            else:
                flag = 0
        if flag == 1 and l > 1:
            ans = max(ans, l)
print(ans)
```

```
# -- weighted uniform string -----
s = input()
t = int(input())
w = 0
dic = {}
for i in range(0, len(s)):
    if i == 0 or s[i] != s[i - 1]:
        w = ord(s[i]) - ord('a') + 1
    else:
        w = w + ord(s[i]) - ord('a') + 1
    dic[w] = 1
for i in range(t):
    x = int(input())
    print('Yes' if x in dic else 'No')
```

```
# --- tow character --
n = int(input())
st = input()
st = list(st)
while True:
    element = ""
    for i in range(len(st) - 1):
        if st[i] == st[i + 1]:
            element = st[i]
    i = 0
    while i < len(st):
        if st[i] == element:
            st.remove(st[i])
        i += 1
    b = False
    for i in range(len(st) - 1):
        if st[i] == st[i + 1]:
            b = True
    if not (b):
        break
print(st)
```

**# -- flip horizontal 2D matrix --**

```
def filp_horzontal(a):
    return [[a[j][i] for i in range(len(a[0]))][::-1] for j in range(len(a))]
```

**# -- flip horizontal and vertical 2D matrix --**

```
def filp_horzontal_and_Virtical(a):
    return [[a[j][i] for i in range(len(a[0]))][::-1] for j in range(len(a))][::-1]
```

**# -- transpose of matrix --**

```
def transpose(a):
    return [[a[i][j] for i in range(len(a))] for j in range(len(a[0]))]
```

**# -- rotate 90 degree clock ways --**

```
def rotate90degree(a):
    r = len(a)
    c = len(a[0])
    result = []
    for i in range(c):
        row = []
        for j in range(r):
            row.append(a[j][i])
        result.append(row[::-1])
    return result
```

**# -- find Target equal to k ------**

```
def findTarget(A, K):
    st = set()
    for i in range(len(A)):
        complement = K - A[i]
        if complement in st:
            return [complement, A[i]]
        else:
            st.add(A[i])
A = [5, 100, 50, 10, 30, 5, 7, 85, 90, 100]
k = 17
x = findTarget(A, k)
print(x)
```

**# -- Fractional knapsack greedy --**

```
W = int(input())
v = [int(x) for x in input().split(',')]
w = [int(x) for x in input().split(',')]
items = [(v[i], w[i], float(v[i] / w[i])) for i in range(len(v))]
items.sort(key=lambda x: x[2], reverse=True)
profit = 0
for i in items:
    if i[1] < W:
        W -= i[1]
        profit += i[0]
    else:
        profit += (W / i[1]) * i[0]
        W = 0
        break
        # capacity = int(capacity - (curWt * fraction))
print(profit)
```

**# -- Activity Selector --**

```
def printMaxActivities(s, f):
    n = len(f)
    print("The following activities are selected")
    i = 0
    print(i, end=' ')
    for j in range(1, n):
        if s[j] >= f[i]:
            print(j, end=' ')
            i = j
s = [1, 3, 0, 5, 8, 5]
f = [2, 4, 6, 7, 9, 9]
printMaxActivities(s, f)
```

**# -- Secret message keyboard wrong typing --**

```
#include <stdio.h>
char *k = "QWERTYUIOPASDFGHJKLZXCVBNM";
main(){
  int i,c;
  while (EOF != (c = getchar())) {
    for (i=0;k[i] && k[i]!=c;i++);
    if (k[i]) putchar(k[i+1]); else putchar(c);
  }
}
```

```
# -- Shortest super string --
public class ShortestSuperString {
   private static int findOverlappingPair(String s1, String s2, StringBuilder sb) {
      int max = Integer.MIN_VALUE;
      int n = Integer.min(s1.length(), s2.length());
      for (int i = 1; i <= n; i++) {
         if (s1.substring(s1.length() - i).equals(s2.substring(0, i))) {
            if (max < i) {
               max = i;
               sb.setLength(0);
               sb.append(s1).append(s2.substring(i));}}}
      for (int i = 1; i <= n; i++) {
         if (s1.substring(0, i).equals(s2.substring(s2.length() - i))) {
            if (max < i) {
               max = i;
               sb.setLength(0);
               sb.append(s2).append(s1.substring(i));
            }}}
      return max;}
```

```
# -- 2 of
public static String findShortestSuperstring(String[] words) {
      int n = words.length;
      while (n != 1) {
         int max = Integer.MIN_VALUE;
         int p = -1, q = -1;
         String res_str = "";
         for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
               StringBuilder sb = new StringBuilder();
               int r = findOverlappingPair(words[i], words[j], sb);
               if (max < r) {
                  max = r;
                  res_str = sb.toString();
                  p = i;
                  q = j;}}}
         n--;
         if (max == Integer.MIN_VALUE) {
            words[0] = words[0] + words[n];
         } else {
            words[p] = res_str;
            words[q] = words[n];}}
      return words[0];}
   public static void main(String[] args) {
      String[] words = {"banana", "ananas"};
      System.out.println("The shortest superstring is " + findShortestSuperstring(words));}}
```