Kasitphoom Thowongs

```python
"""Increasing Subsequence"""
memorization = {}

"""Increasing recursive function to find increasing sequences"""
def increasing(k):

    if k in memorization:
        return memorization[k]

    if k == 0:
        return 1

    max_length = 0
    for i in range(k):
        if sample[i] ≤ sample[k]:
            val = increasing(i)
            if max_length < val:
                max_length = val
    memorization[k] = 1 + max_length
    return memorization[k]

def lis():
    max_length = 0
    for i in range(len(sample)):
        val = increasing(i)
        if val > max_length:
            max_length = val
    return max_length

sample = list(map(int, input().split()))
print(lis())
```

```python
"""Two Item finding"""
"""Function of finding sum of two numbers"""
def opt():
    target = int(input())
    arr = list(map(int, input().split()))

    arr = sorted(arr)
    low = 0
    high = len(arr) - 1

    while low ≠ high:
        calc = arr[low] + arr[high]
        if calc == target:
            return "Yes"
        elif calc < target:
            low += 1
        else:
            high -= 1

    return "No"

print(opt())
```

```python
"""Double, Triple, and Increment"""
memo = {}
def increment(x):
    if x in memo:
        return memo[x]
    if x == 1:
        memo[1] = 0
        return memo[1]

    if x % 3 == 0:
        memo[x] = min(increment(x / 3), increment(x - 1)) + 1
        return memo[x]
    if x % 2 == 0:
        memo[x] = min(increment(x / 2), increment(x - 1)) + 1
        return memo[x]
    if x - 1 > 0:
        memo[x] = increment(x - 1) + 1
        return memo[x]

print(increment(int(input())))
```

```python
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

    return arr
```

```python
"""Finding subset sum problem"""
mem = {}

"""Finding that is there a target in the set"""
def check(index, target):
    if (index, target) in mem:
        return mem[(index, target)]

    if index == 0:
        mem[(index, target)] = (target == 0)
        return (target == 0)

    if target < inputSet[index - 1]:
        mem[(index, target)] = check(index - 1, target)
        return mem[(index, target)]
    else:
        mem[(index, target)] = check(index - 1, target) or check(index - 1, target - inputSet[index - 1])
        return mem[(index, target)]

inputSet = list(map(int, input().split()))
target = int(input())
print(check(len(inputSet), target))
```

```python
def quick_sort(arr):
    if len(arr) ≤ 1:
        return arr
    else:
        pivot = arr[0]
        less = [x for x in arr[1:] if x ≤ pivot]
        greater = [x for x in arr[1:] if x > pivot]
        return quick_sort(less) + [pivot] + quick_sort(greater)
```

```python
def largest_x(y):

    left = 0
    right = y

    while left ≤ right:
        mid = (left + right) // 2
        if mid * mid + mid ≤ y:
            left = mid + 1
        else:
            right = mid - 1

    return right
```

```python
list_numbers = list(map(int, input().split()))

def picking_number(list_num):
    if len(list_num) == 0:
        return True
    else:
        return picking_numbers(list_num[1:], list_num[0]) or picking_numbers(list_num[:-1], list_num[-1])

def picking_numbers(list_num, picked_num):
    if len(list_num) == 0:
        return True
    else:
        f_h = False
        s_h = False
        if abs(list_num[0] - picked_num) ≤ 9:
            f_h = picking_numbers(list_num[1:], list_num[0])
        if abs(list_num[-1] - picked_num) ≤ 9:
            s_h = picking_numbers(list_num[:-1], list_num[-1])
        return f_h or s_h
```

```python
def binary_search_approximation(arr, l, r, x):
    if r ≥ l:
        mid = l + (r - l) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            return binary_search_approximation(arr, l, mid - 1, x)
        else:
            return binary_search_approximation(arr, mid + 1, r, x)
    else:
        return r
```

```python
def cutting(length, cuts):
    memo = {}

    def opt(l, r):
        if r - l == 1:
            return 0
        if (l, r) in memo:
            return memo[(l, r)]
        res = float("inf")
        for c in cuts:
            if l < c < r:
                res = min(res, r - l + opt(l, c) + opt(c, r))
        if res == float("inf"):
            res = 0
        memo[(l, r)] = res
        return res

    return opt(0, length)
```

Cutting Stick

```
opt_profit[1…N] = Array of length n
opt_cuts[1…N] = Array of length n
opt_profit[1] = P[1]
opt_cuts[1] = []

for n = 2 to N:
  max_profit = P[n]
  max_cuts = []
  for k from 1 to n-1
    if max_profit < P[k]+opt_profit[n-k]-C
       max_profit = P[k]+opt_profit[n-k]-C
       max_cuts = [k] + opt_cuts[n-k]
  opt_profit[n] = max_profit
  opt_cuts[n] = max_cuts

print(opt_profit[N], opt_cuts[N])
```


Max 1D Range Sum – Kadane's Algorithm

- There is an O(n) algorithm to solve the Max 1D Range Sum problem. The algorithm described below is attributed to Jay Kadane.

```
// Kadane's Algorithm
sum = 0
max_sum = 0

for i = 1 … n
  sum += x[i]
  max_sum = max(max_sum, sum)
  if(sum < 0) sum = 0

return max_sum
```

- Try running the above algorithm on the array x =

4, -5, 4, -3, 4, 4, -4, 4, -5

Closest Pair of Points (Improved)

CLOSEST-PAIR(Px: list of points sorted in x, Py: list of points sorted in y)

Find vertical line L such that half the points are on each side of the line.	$\Theta(1)$
PxLeft= Px with all the points to the right of L removed	$\Theta(n)$
PxRight = Px with all the points to the left of L removed	$\Theta(n)$
PyLeft = Py with all the points to the right of L removed	$\Theta(n)$
PyRight = Py with all the points to the left of L removed	$\Theta(n)$
δ1 ← CLOSEST-PAIR(PxLeft, PyLeft).	$C'(n/2)$
δ2 ← CLOSEST-PAIR(PxRight, PyRight).	$C'(n/2)$
δ ← min { δ1 , δ2 }.	$\Theta(1)$
Delete all points further than δ from line L.	$\Theta(n)$
~~Sort remaining points by y-coordinate.~~	
Scan points in y-order and compare distance between each point and next 7 neighbors. If any of these distances is less than δ, update δ.	$\Theta(n)$
RETURN δ.	$\Theta(1)$

Max 2D Range Sum

- Given an n x n table of integers, find a pair (a, b, c, d) of locations in the table which maximizes the partial sum, i.e.

sum(a, b, c, d) ≥ sum(x, y, x', y')

for all indices w, x, y, z where x ≤ x' and y ≤ y'

- A straightforward implementation:

```
a = b = c = d = 1
max = x[a,b]

for i1 = 1 … n
  for j1 = 1 … n
    for i2 = i1 … n
      for j2 = j1 … n
        if(sum(i1,j1,i2,j2) > max)
          max = sum(i1,j1,i2,j2)
          a = i1; b = j1
          c = i2; d = j2
return (a,b,c,d)
```


Celebrity Problem

- **Attempt 4:** An improved version of Attempt 2.

```
function celeb_dq(S)
    if |S| == 1 then return the (only) person in S
    else
        Pick two people P1 and P2 from S.
        if P1 knows P2 then P'=P1 else P'=P2
        S' = S - {P'}
        C' = celeb_dq(S')
        if C' ≠ None then
            if P' knows C' and C' does not know P
            then return C' else return None
        else
            return None
```

```python
def max_earning(n, jobs):
    # Sort jobs based on finish times
    jobs.sort(key=lambda x: x[1])

    # Initialize an array to store maximum total earnings for each job
    dp = [0] * n

    # Base case: the maximum total earning for the first job is its own earning
    dp[0] = jobs[0][2]

    # Iterate through the jobs
    for i in range(1, n):
        # Find the latest non-conflicting job
        latest_non_conflicting = -1
        for j in range(i - 1, -1, -1):
            if jobs[j][1] <= jobs[i][0]:
                latest_non_conflicting = j
                break

        # Update the maximum total earning for the current job
        dp[i] = max(dp[i - 1], jobs[i][2] + (dp[latest_non_conflicting] if latest_non_conflicting != -1 else 0))

    # The final result is the maximum total earning for the last job
    return dp[-1]

# Read the number of jobs
n = int(input())

# Read the jobs as lines of strings, then split each line
jobs = [list(map(int, input().split())) for _ in range(n)]

# Output the result
print(max_earning(n, jobs))
```

```python
"""Program to find least Articulation points in a graph"""

def art_points(graph, length):
    """Articulation points in a graph."""
    visited = [0 for _ in range(length)]
    parent = [-1 for _ in range(length)]
    low = [0 for _ in range(length)]
    disc = [0 for _ in range(length)]
    articulation_points = [0 for _ in range(length)]
    time = 0

    def dfs(node):
        """Depth First Search."""
        nonlocal time
        children = 0
        visited[node] = 1
        low[node] = disc[node] = time
        time += 1

        for i in range(length):
            if graph[node][i]:
                if not visited[i]:
                    children += 1
                    parent[i] = node
                    dfs(i)
                    low[node] = min(low[node], low[i])
                    if parent[node] == -1 and children > 1:
                        articulation_points[node] = 1
                    if parent[node] != -1 and low[i] >= disc[node]:
                        articulation_points[node] = 1
                elif i != parent[node]:
                    low[node] = min(low[node], disc[i])

    for i in range(length):
        if not visited[i]:
            dfs(i)

    return [i + 1 for i in range(length) if articulation_points[i]]
```

```python
"""Bipartite graph algorithms."""
def bipartite():
    """Bipartite graph algorithm."""
    graph_length = int(input())
    graph = [
        [0 for _ in range(graph_length)]
        for _ in range(graph_length)
    ]

    while True:
        edge = list(map(int, input().split()))
        if edge == [0, 0]:
            break
        graph[edge[0] - 1][edge[1] - 1] = 1

    queue = [0]
    visited = [0 for _ in range(graph_length)]
    color = [0 for _ in range(graph_length)]
    is_bipartite = True
      You, 1 hour ago • first commit
    while queue and is_bipartite:
        node = queue.pop(0)
        for i in range(graph_length):
            if graph[node][i] and not visited[i]:
                queue.append(i)
                visited[i] = 1
                color[i] = not color[node]
            elif graph[node][i] and color[i] == color[node]:
                is_bipartite = False
                break

    if is_bipartite:
        # list every 0 in the color list
        print(*[i + 1 for i in range(graph_length) if not color[i]])
        # list every 1 in the color list
        print(*[i + 1 for i in range(graph_length) if color[i]])
    else:
        print("Not possible")
```

```python
def findSCC(G):
  global state, counter, low, num, stack
  state = [UNDISCOVERED for i in range(G.numNodes)]
  low = [0 for i in range(G.numNodes)]
  num = [0 for i in range(G.numNodes)]
  counter = 1
  stack = []
  for s in range(G.numNodes):
    if state[s]==UNDISCOVERED:
      state[s] = DISCOVERED
      dfsTarjan(G,s)
```

```python
def max_jobs(jobs):
    # Sort jobs by finish time
    sorted_jobs = sorted(jobs, key=lambda x: x[1])

    count = 0
    last_finish = 0
    for start, finish in sorted_jobs:
        if start >= last_finish:
            count += 1
            last_finish = finish

    return count

if __name__ == "__main__":
    n = int(input())
    jobs = [tuple(map(int, input().split())) for _ in range(n)]
    print(max_jobs(jobs))
```

```python
def main():
    """Main function."""
    graph_length = int(input())
    graph = [
        [0 for _ in range(graph_length)]
        for _ in range(graph_length)
    ]

    while True:
        edge = list(map(int, input().split()))
        if edge == [0, 0]:
            break
        graph[edge[0] - 1][edge[1] - 1] = 1
        graph[edge[1] - 1][edge[0] - 1] = 1

    print(*art_points(graph, graph_length))
```

```python
"""BFS Shortest Path Algorithm non weighted graph"""
def bfs_shortest():
    """Graph Transversal function"""

    queue = [starting_node - 1]
    visited = [0 for _ in range(graph_length)]
    back = [None for _ in range(graph_length)]
    found = False

    visited[starting_node - 1] = 1
    while queue and not found:
        node = queue.pop(0)
        for i in range(graph_length):
            if graph[node][i] and i == destination_node - 1
                back[i] = node
                found = True
                break
            if graph[node][i] and not visited[i]:
                queue.append(i)
                visited[i] = 1
                back[i] = node

    if found:
        path = [destination_node]
        while path[-1] != starting_node:
            path.append(back[path[-1] - 1] + 1)

        print(*path[::-1])
    else:
        print("No path")
```

```python
"""min weight"""
def dijkstra():
    graph_length = int(input())
    graph = [[float('inf') for i in range(graph_length)] for j in range(graph_length)]
    distance = [float('inf') for i in range(graph_length)]
    visited = [0 for i in range(graph_length)]

    start = int(input()) - 1
    distance[start] = 0

    queue = [start]
    visited[start] = 1

    while queue:
        node = queue.pop(0)
        visited[node] = 1
        for i in range(graph_length):
            if graph[node][i]:
                distance[i] = min(distance[i], distance[node] + graph[node][i])
                if not visited[i]:
                    queue.append(i)
                    visited[i] = 1

    print(*distance)
```

```python
def bellmanFord():
    graph_length = int(input())
    edges = []

    while True:
        edge = list(map(int, input().split()))
        if edge[0] == 0 and edge[1] == 0:
            break
        edges.append([edge[0] - 1, edge[1] - 1, edge[2]])

    table = [[float("inf") for i in range(graph_length)] for j in range(graph_length)]

    table[0][0] = 0
    for n in range(1, graph_length):
        for t in range(graph_length):
            table[n][t] = table[n - 1][t]
        for edge in edges:
            v, t, w = edge
            if table[n - 1][v] + w < table[n][t]:
                table[n][t] = table[n - 1][v] + w

        if n == 1:
            table[n][0] = float("inf")

    return table[graph_length - 1]
```

```python
def floyd_warshall():
    for k in range(graph_length):
        for i in range(graph_length):
            for j in range(graph_length):
                graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j])
    print(*graph, sep='\n')
```

```python
def dfsTarjan(G,u):
  global state, counter, low, num, stack
  low[u] = num[u] = counter
  counter+=1
  stack.append(u)

  for v in G.neighbors(u):
    if state[v]==UNDISCOVERED:
      # Tree edge: DISCOVERED -> UNDISCOVERED
      state[v] = DISCOVERED
      dfsTarjan(G,v)
      low[u] = min(low[u],low[v])
    elif state[v]==DISCOVERED:
      # Back edge: DISCOVERED -> DISCOVERED
      low[u] = min(low[u],num[v])
    elif state[v]==EXPLORED:
      # Forward/cross edge: DISCOVERED -> EXPLORED
      low[u] = min(low[u],num[v])

  if low[u]==num[u]:
    scc = []
    while(True):
      v = stack.pop()
      state[v] = DELETED
      scc.append(v)
      if u==v:
        break
    print(scc)
  else:
    state[u] = EXPLORED
```

```python
def kruskal():
    graph_length = int(input())
    edges = []

    while True:
        edge = list(map(int, input().split()))
        if edge[0] == 0 and edge[1] == 0:
            break
        edges.append([edge[0], edge[1], edge[2]])

    edges.sort(key=lambda x: x[2])

    mst = []

    parent = [i for i in range(graph_length)]

    def find(x):
        if parent[x] != x:
            parent[x] = find(parent[x])
        return parent[x]

    def union(x, y):
        parent[find(x)] = find(y)

    for edge in edges:
        u, v, w = edge
        if find(u) != find(v):
            mst.append(edge)
            union(u, v)

    return mst
```

```python
def topSort(G):
  # Computing in-degree of each node
  indegree = [0 for i in range(G.numNodes)]
  for u in range(G.numNodes):
    for v in G.neighbors(u):
      indegree[v]+=1

  S = [u for u in range(G.numNodes) if indegree[u]==0]
  L = []
  while len(S)>0:
    u = S.pop()
    L.append(u)
    for v in G.neighbors(u):
      indegree[v]-=1
      if indegree[v]==0:
        S.append(v)

  return L
```

```python
def prim():
    graph_length = int(input())
    edges = []

    while True:
        edge = list(map(int, input().split()))
        if edge[0] == 0 and edge[1] == 0:
            break
        edges.append([edge[0], edge[1], edge[2]])

    X = [0]
    T = []

    while len(X) < graph_length:
        min_edge = [None, None, float("inf")]
        for edge in edges:
            if edge[0] in X and edge[1] not in X and edge[2] < min_edge[2]:
                min_edge = edge
            elif edge[1] in X and edge[0] not in X and edge[2] < min_edge[2]:
                min_edge = edge

        T.append(min_edge)
        X.append(min_edge[1])

    return T
```

```python
"""Best First Search Algorithm"""
import heapq

# Declaring constants
UNDISCOVERED = 0
DISCOVERED = 1

def bestfs(G,s):
  state = [UNDISCOVERED for i in range(G.numNodes)]
  H = []
  state[s] = DISCOVERED
  heapq.heappush(H,s)
  while len(H)>0:
    u = heapq.heappop(H)
    print("Exploring Node " + str(u))
    for v in G.neighbors(u):
      if state[v] == UNDISCOVERED:
        state[v] = DISCOVERED
        heapq.heappush(H,v)
```

```python
class WDGraphAdjLst(WDGraph):
  def __init__(self, N):
    self.adjLst = [deque() for j in range(N)]
    self.numNodes = N
    self.numEdges = 0

  def addEdge(self,u,v,d):
    if not self.isAdjacent(u,v):
      self.adjLst[u].append((v,d))
      self.numEdges+=1
```

```
[deque([(1, 2), (3, 5)]), deque([(2, 3), (3, 7)]), deque([(5, 4)])]
```

```python
from collections import deque

class DGraphAdjLst():
  def __init__(self, N):
    self.adjLst = [deque() for j in range(N)]
    self.numNodes = N
    self.numEdges = 0

  def addEdge(self,u,v):
    if not self.isAdjacent(u,v):
      self.adjLst[u].append(v)
      self.numEdges+=1

  def removeEdge(self,u,v):
    if self.isAdjacent(u,v):
      self.adjLst[u].remove(v)
      self.numEdges-=1

  def isAdjacent(self,u,v):
    for w in self.adjLst[u]:
      if w == v:
        return True
    return False

  def neighbors(self,u):
    return list(self.adjLst[u])

  def edges(self):
    L = []
    for u in range(self.numNodes):
      for v in self.adjLst[u]:
        L.append((u,v))
    return L
```

Both

```python
memo = {}
def test(flee, list_num):
    if (flee, tuple(list_num)) in memo:
        return memo[(flee, tuple(list_num))]
    elif len(list_num) == 0:
        return 0
    elif len(list_num) == 1:
        memo[(flee, tuple(list_num))] = list_num[0] - flee
        return memo[(flee, tuple(list_num))]
    else:
        profit_from_f = (list_num[-1] - flee) + test(flee, list_num[:-1])
        profit_from_s = (list_num[0] - flee) + test(flee, list_num[1:])
        memo[(flee, tuple(list_num))] = max(profit_from_f, profit_from_s)
        return memo[(flee, tuple(list_num))]
flee = 50
list_numbers = [54, 55, 40, 48, 51, 50, 56, 49, 54, 47]
list_num1 = [45, 50, 49]
test(flee, list_numbers)
# get max value from memo
result = 0
for key, value in memo.items():
    if value > result:
        result = value
print(result if result > 0 else "No")
```

```python
def findVisitable(G, u):
    visitable = [False]*G.numNodes
    queue = deque()
    queue.append(u)

    for v in G.neighbors(u):
        queue.append(v)
        visitable[v] = True
    while queue:
        u = queue.popleft()
        for v in G.neighbors(u):
            if not visitable[v]:
                queue.append(v)
                visitable[v] = True

    for i in range(G.numNodes):
        if visitable[i]:
            print(i+1, end=" ")


number = int(input())
G1 = DGraphAdjLst(number)
start = int(input())

while True:
    a = input().split()
    u = int(a[0])
    v = int(a[1])
    if u == 0 and v == 0:
        break
    G1.addEdge(u-1, v-1)

findVisitable(G1, start-1)
```