## 根据前中序表达式建树

```
def build_tree(preorder,inorder):
    if not preorder or not inorder:
        return ''

    root=preorder[0]
    root_index=inorder.index(root)

    left_preorder=preorder[1:root_index+1]
    right_preorder=preorder[root_index+1:]

    left_inorder=inorder[:root_index]
    right_inorder=inorder[root_index+1:]

    left_tree=build_tree(left_preorder,left_inorder)
    right_tree=build_tree(right_preorder,right_inorder)

    return left_tree+right_tree+root

preorder,inorder=input().split()
postorder=build_tree(preorder,inorder)
print(postorder)
```

## 根据中后序表达式建树

```
def build_preorder(inorder,postorder):
    if not inorder or not postorder:
        return ''

    root=postorder[-1]
    root_index=inorder.index(root)

    left_inorder=inorder[:root_index]
    right_inorder=inorder[root_index+1:]

    left_postorder=postorder[:len(left_inorder)]
    right_postorder=postorder[len(left_inorder):-1]

    return
root+build_preorder(left_inorder,left_postorder)+build_preorder(right_inorder,rig
ht_postorder)

inorder=input()
postorder=input()
preorder=build_preorder(inorder,postorder)
print(preorder)
```

## 判断图连通&回路

```
#无向图模板，邻接表存储
def is_connected():
    visited=[False]*n
    def dfs(node):
        visited[node]=True
        for neighbor in edges[node]:
            if not visited[neighbor]:
                dfs(neighbor)
    dfs(0)
    if all(visited):
        return 'connected:yes'
    else:
        return 'connected:no'

def has_loop():
    visited=[False]*n
    def dfs(node,parent,visited_set):
        visited[node]=True
        visited_set.add(node)
        for neighbor in edges[node]:
            if not visited[neighbor]:
                if dfs(neighbor,node,visited_set):
                    return True
            elif neighbor!=parent and neighbor in visited_set:
            #有向图改为elif neighbor in visited_set:
                return True
        visited_set.remove(node)
        return False
    for i in range(n):
        if not visited[i]:
            visited_set=set()
            if dfs(i,-1,visited_set):
                return 'loop:yes'
    return 'loop:no'

n,m=map(int,input().split())
edges=[[] for _ in range(n)]
for _ in range(m):
    u,v=map(int,input().split())
    edges[u].append(v)
    edges[v].append(u)
print(is_connected())
print(has_loop())
```

## Prim算法

```
#堆Prim
from heapq import heappop, heappush, heapify
def prim(graph, start_node):
    mst = set()
```

```
    visited = set([start_node])
    edges = [(cost, start_node, to) for to, cost in graph[start_node].items()]
    heapify(edges)
    while edges:
        cost, frm, to = heappop(edges)
        if to not in visited:
            visited.add(to)
            mst.add((frm, to, cost))
            for to_next, cost2 in graph[to].items():
                if to_next not in visited:
                    heappush(edges, (cost2, to, to_next))

    return mst
```

## Dijkstra算法

```
import heapq
def dijkstra():
    heap=[(0,0,1)]
    visited=set()
    while heap:
        length,cost,city=heapq.heappop(heap)
        if city==n and cost<=k:
            return length
        if city in visited or cost>k:
            continue
        visited.add(city)
        for nbr,nlenth,ncost in edges[city]:
            if nbr not in visited and cost+ncost<=k:
                heapq.heappush(heap,(length+nlenth,cost+ncost,nbr))
        visited.remove(city)
    return -1

k=int(input())
n=int(input())
edges=[[] for _ in range(n+1)]
for _ in range(int(input())):
    s,d,l,t=map(int,input().split())
    edges[s].append((d,l,t))
print(dijkstra())
```

## 并查集

```
#以题为例
#冰阔落
def find_parent(i):
    if parents[i]!=i:
        parents[i]=find_parent(parents[i])
    return parents[i]
```

```
def union(x,y):
    rootX=find_parent(x)
    rootY=find_parent(y)
    parents[rootY]=rootX

while True:
    try:
        n,m=map(int,input().split())
        parents=list(range(n+1))
        rank=[0]*(n+1)
        for _ in range(m):
            x,y=map(int,input().split())
            if find_parent(x)==find_parent(y):
                print('Yes')
            else:
                print('No')
                union(x,y)
        roots={find_parent(i) for i in range(1,n+1)}
        print(len(roots))
        print(*sorted(roots),sep=' ')
    except EOFError:
        break

#Agri-Net
def find_parent(i):
    if parent[i]==i:
        return i
    return find_parent(parent[i])

def union(x,y):
    rootX=find_parent(x)
    rootY=find_parent(y)
    parent[rootY]=rootX

def minmum_fibers(egdes):
    total_lenth=0
    for lenth,x,y in sorted(edges,key=lambda x:x[0]):
        rootX=find_parent(x)
        rootY=find_parent(y)
        if rootX!=rootY:
            total_lenth+=lenth
            union(rootX,rootY)
        if len(set(find_parent(i) for i in range(n)))==1:
            break
    return total_lenth

while True:
    try:
        n=int(input())
        edges=[]
        parent=list(range(n))
        for i in range(n):
            lenths=input().split()
            for j in range(n):
                lenth=int(lenths[j])
                edges.append((lenth,i,j))
```

```
        print(minmum_fibers(edges))
    except EOFError:
        break

#繁忙的厦门
class Edge:
    def __init__(self,u,v,c):
        self.u=u
        self.v=v
        self.c=c

class UnionFind:
    def __init__(self,size):
        self.parent=[i for i in range(size)]
        self.rank=[0]*size

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

    def union(self,x,y):
        rootX=self.find(x)
        rootY=self.find(y)
        if rootX!=rootY:
            if self.rank[rootX]>self.rank[rootY]:
                self.parent[rootY]=rootX
            elif self.rank[rootX]<self.rank[rootY]:
                self.parent[rootX]=rootY
            else:
                self.parent[rootY]=rootX
                self.rank[rootX]+=1

def kruskal(n,edges):
    edges.sort(key=lambda x:x.c)
    uf=UnionFind(n)
    mst=[]
    max_weight=0
    total_edges=0
    for edge in edges:
        if uf.find(edge.u)!=uf.find(edge.v):
            uf.union(edge.u,edge.v)
            mst.append(edge)
            total_edges+=1
            max_weight=max(max_weight,edge.c)

    return total_edges,max_weight

n,m=map(int,input().split())
edges=[]
for _ in range(m):
    u,v,c=map(int,input().split())
    edges.append(Edge(u-1,v-1,c))
total_edges,max_weight=kruskal(n,edges)
print(total_edges,max_weight)
```

## BFS 常见写法

```
from collections import deque
def bfs():
    queue=deque([startNode])
    visited={startNode}
    while queue:
        node=queue.popleft()
        for nb in node.nbs:
            if nb not in visited:
                queue.append(nb)
                visited.add(nb)

#词梯
from collections import defaultdict,deque
def categorize(word_list):
    categorization=defaultdict(list)
    for word in word_list:
        for i in range(4):
            pattern=word[:i]+'*'+word[i+1:]
            categorization[pattern].append(word)
    return categorization

def bfs(start_word,end_word):
    queue=deque([(start_word,[start_word])])
    visited=set([start_word])
    while queue:
        current_word,path=queue.popleft()
        if current_word==end_word:
            return path
        for i in range(4):
            pattern=current_word[:i]+'*'+current_word[i+1:]
            for neighbor in categorization[pattern]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append((neighbor,path+[neighbor]))
    return []

n=int(input())
word_list=[input() for _ in range(n)]
start_word,end_word=input().split()
categorization=categorize(word_list)
shortest_path=bfs(start_word,end_word)
if shortest_path:
    print(' '.join(shortest_path))
else:
    print('NO')

#Pots
def pour_water(A,B,C):
    visited=set()
    queue=[(0,0,[])]
    while queue:
```

```
        a,b,operations=queue.pop(0)
        if a==C or b==C:
            return operations
        if (a,b) in visited:
            continue
        visited.add((a,b))
        if a<A:
            queue.append((A,b,operations+['FILL(1)']))  # 填充A
        if b<B:
            queue.append((a,B,operations+['FILL(2)']))  # 填充B
        if a>0:
            queue.append((0,b,operations+['DROP(1)']))  # 清空A
        if b>0:
            queue.append((a,0,operations+['DROP(2)']))  # 清空B
        if a>0 and b<B:
            pour_amount=min(a,B-b)
            queue.append((a-pour_amount,b+pour_amount,operations+['POUR(1,2)']))
 # 从A向B倒水
        if b>0 and a<A:
            pour_amount=min(b,A-a)
            queue.append((a+pour_amount,b-pour_amount,operations+['POUR(2,1)']))
 # 从B向A倒水
    return 'impossible'

A,B,C=map(int,input().split())
result=pour_water(A,B,C)
print(result if result=='impossible' else
str(len(result))+'\n'+'\n'.join(result))

#鸣人和佐助
from collections import deque
def bfs():
    directions=[(0,1),(0,-1),(1,0),(-1,0)]
    while queue:
        x,y,chakra,time=queue.popleft()
        time+=1
        for dx,dy in directions:
            nx,ny=x+dx,y+dy
            if 0<=nx<m and 0<=ny<n:
                elem=grid[nx][ny]
                if elem=='*' and chakra>visited[nx][ny]:
                    visited[nx][ny]=chakra
                    queue.append((nx,ny,chakra,time))
                elif elem=='#' and chakra-1>visited[nx][ny]:
                    visited[nx][ny]=chakra-1
                    queue.append((nx,ny,chakra-1,time))
                elif elem=='+':
                    return time
    return -1

m,n,t=map(int,input().split())
grid=[input() for _ in range(m)]
start=None
visited=[[-1]*n for _ in range(m)]
for i in range(m):
    for j in range(n):
```

```
        if grid[i][j]=='@':
            start=(i,j)
            visited[i][j]=t
    if start:
        break
queue=deque([start+(t,0)])
print(bfs())

#走山路
import heapq
def bfs():
    if grid[x1][y1]=='#' or grid[x2][y2]=='#':
        return 'NO'
    direc=[(-1,0),(1,0),(0,1),(0,-1)]
    d=[[float('inf')]*n for _ in range(m)]
    d[x1][y1]=0
    points=[(0,x1,y1)]
    while points:
        stamina,x,y=heapq.heappop(points)
        h=grid[x][y]
        if (x,y)==(x2,y2):
            return stamina
        for dx,dy in direc:
            nx,ny=x+dx,y+dy
            if 0<=nx<m and 0<=ny<n and grid[nx][ny]!='#' and d[nx]
[ny]>stamina+abs(grid[nx][ny]-h):
                d[nx][ny]=stamina+abs(grid[nx][ny]-h)
                heapq.heappush(points,(d[nx][ny],nx,ny))
    return 'NO'

m,n,p=map(int,input().split())
grid=[[int(x) if x!='#' else '#' for x in input().split()] for _ in range(m)]
for _ in range(p):
    x1,y1,x2,y2=map(int,input().split())
    print(bfs())
```

## 升空的焰火，从侧面看

```
# bfs，存一下每层节点
from collections import deque,defaultdict
def bfs(lst):
    queue=deque([(1,0)])
    res=defaultdict(list)
    while queue:
        node,level=queue.popleft()
        res[level].append(node)
        left,right=lst[node-1]
        if left!=-1:
            queue.append((left,level+1))
        if right!=-1:
            queue.append((right,level+1))
    return res
```

```
n=int(input())
lst=[map(int,input().split()) for _ in range(n)]
res=[d[-1] for d in bfs(lst).values()]
print(*res)
```

## 【模板】单调栈

```
from collections import deque
n=int(input())
res=[0]*n
nums=list(map(int,input().split()))
stack=[]
for i,t in enumerate(nums):
    while stack and t>nums[stack[-1]]:
        res[stack.pop()]=i+1
    stack.append(i)
print(*res)
```

## 小组队列

```
def simulate_queue(groups,command):
    if command[0]=='ENQUEUE':
        person=command[1]
        l=len(queue)
        if l in [0,1]:
            queue.append(person)
        else:
            i=0
            while i<l-1:
                if group_map[person]==group_map[queue[i]] and
group_map[person]!=group_map[queue[i+1]]:
                    queue.insert(i+1,person)
                    break
                else:
                    i+=1
            if i==l-1:
                queue.append(person)
    else:
        print(queue.pop(0))

queue=[]
group_map={}
groups=[input().split() for _ in range(int(input()))]
for group_index,group in enumerate(groups):
    for person in group:
        group_map[person]=group_index
while True:
    command=input().split()
    if command[0]=='STOP':
        break
```

```
    else:
        simulate_queue(groups,command)
```

## 中序转后序

```
def infixtopostfix(a):
    prec={'*':3,'/':3,'+':2,'-':2,'(': 1}
    infix=a.split()
    stack=[]
    postfix=[]
    for i in infix:
        if i not in ['*','/','+','-','(',')']:
            postfix.append(i)
        elif i=='(':
            stack.append(i)
        elif i==')':
            top=stack.pop()
            while top!='(':
                postfix.append(top)
                top=stack.pop()
        else:
            while stack!=[] and prec[stack[-1]]>=prec[i]:
                postfix.append(stack.pop())
            stack.append(i)
    while len(stack)!=0:
        postfix.append(stack.pop())
    return ' '.join(postfix)

for _ in range(int(input())):
    a=input()
    b=''
    for i in a:
        if i in ['*','/','+','-','(',')']:
            b+=' '+i+' '
        else:
            b+=i
    print(infixtopostfix(b))
```

## 树的转换

```
def tree_height1(s):
    height=0
    max_height=0
    for char in s:
        if char=='d':
            height+=1
            max_height=max(max_height,height)
        else:
            height-=1
    return max_height
```

```
def tree_height2(s):
    height=0
    stack=[]
    max_height=0
    for char in s:
        if char=='d':
            height+=1
            stack.append(height)
            max_height=max(max_height,height)
        else:
            height=stack.pop()
    return max_height

s=input()
print(f'{tree_height1(s)} => {tree_height2(s)}')
```

## 遍历树

```
class TreeNode:
    def __init__(self,value):
        self.value=value
        self.children=[]

def sort_node(node):
    global res
    node.children.sort(key=lambda x:x.value)
    sgn=1
    for child in node.children:
        idx=res.index(node.value)
        if child.value<node.value:
            res.insert(idx,child.value)
        else:
            res.insert(idx+sgn,child.value)
            sgn+=1
    for child in node.children:
        sort_node(child)
    return res

nodes={}
for _ in range(int(input())):
    values=list(map(int,input().split()))
    node_val=values[0]
    children_vals=values[1:]
    if node_val not in nodes:
        nodes[node_val]=TreeNode(node_val)
    node=nodes[node_val]
    if children_vals:
        for child_val in children_vals:
            if child_val not in nodes:
                nodes[child_val]=TreeNode(child_val)
            child_node=nodes[child_val]
            node.children.append(child_node)
```

```
root=None
for node_val,node in nodes.items():
    is_root=True
    for _,other_node in nodes.items():
        if node_val in [child.value for child in other_node.children]:
            is_root=False
            break
    if is_root:
        root=node
        break

res=[root.value]
result=sort_node(root)
for val in result:
    print(val)
```

## 骑士周游

```
def is_valid(x,y):
    return 0<=x<n and 0<=y<n and not visited[x][y]

def sort_neighbors(x,y):
    directions=[(-2,-1),(-2,1),(-1,-2),(-1,2),(1,-2),(1,2),(2,-1),(2,1)]
    neighbors=[]
    for dx,dy in directions:
        nx,ny=x+dx,y+dy
        if is_valid(nx,ny):
            count=0
            for dx1,dy1 in directions:
                if is_valid(nx+dx1,ny+dy1):
                    count+=1
            neighbors.append((nx,ny,count))
    return sorted(neighbors,key=lambda x:x[2])

def knight_tour(x,y,count):
    visited[x][y]=True
    count+=1
    if count==n*n:
        return True
    for nx,ny,_ in sort_neighbors(visited,x,y):
        if knight_tour(nx,ny,count):
            return True
    visited[x][y]=False
    return False

n=int(input())
visited=[[False]*n for _ in range(n)]
x,y=map(int,input().split())
print('success' if knight_tour(x,y,0) else 'fail')
```

## 最大连通域面积

```
def dfs(x,y):
    if x<0 or x>=n or y<0 or y>=m or board[x][y]=='.' or visited[x][y]:
        return 0
    visited[x][y]=True
    area=1
    directions=[(x,y) for x in [-1,0,1] for y in [-1,0,1]]
    for dx,dy in directions:
        area+=dfs(x+dx,y+dy)
    return area

def find_largest_area(board):
    max_area=0
    for i in range(n):
        for j in range(m):
            if board[i][j]=='W' and not visited[i][j]:
                max_area=max(max_area,dfs(i,j))
    return max_area

for _ in range(int(input())):
    n,m=map(int,input().split())
    board=[input() for _ in range(n)]
    visited=[[False for _ in range(m)] for _ in range(n)]
    print(find_largest_area(board))
```

## 最大权值连通块

```
def dfs(v):
    visited[v]=True
    wgt=weights[v]
    for nbr in neighbor[v]:
        if not visited[nbr]:
            wgt+=dfs(nbr)
    return wgt

n,m=map(int,input().split())
weights=list(map(int,input().split()))
visited=[False]*n
neighbor=[[] for _ in range(n)]
for _ in range(m):
    u,v=map(int,input().split())
    neighbor[u].append(v)
    neighbor[v].append(u)
dp=[0]*n
for v in range(n):
    if not visited[v]:
        dp[v]=dfs(v)
print(max(dp))
```

## 舰队、海域出击!

思路：下面给了两种dfs写法

```
#
def has_loop():
    visited=[False]*n
    def dfs(node,parent,visited_set):
        visited[node]=True
        visited_set.add(node)
        for neighbor in edges[node]:
            if not visited[neighbor]:
                if dfs(neighbor,node,visited_set):
                    return True
            elif neighbor in visited_set:
                return True
        visited_set.remove(node)
        return False
    for i in range(n):
        if not visited[i]:
            visited_set=set()
            if dfs(i,-1,visited_set):
                return 'Yes'
    return 'No'
```

```
#
def has_loop():
    visited=[False]*(n+1)
    recursion_stack=[False]*(n+1)
    def dfs(node):
        visited[node]=True
        recursion_stack[node]=True
        for nbr in edges[node]:
            if not visited[nbr]:
                if dfs(nbr):
                    return True
            elif recursion_stack[nbr]:
                return True
        recursion_stack[node]=False
        return False

    for node in range(1,n+1):
        if not visited[node]:
            if dfs(node):
                return 'Yes'
    return 'No'

for _ in range(int(input())):
    n,m=map(int,input().split())
    edges=[[] for _ in range(n+1)]
    for _ in range(m):
        x,y=map(int,input().split())
        edges[x].append(y)
```

```
    print(has_loop())
```

## 月度开销

```
# binary_search，这个check的作用就是检查在当前“最大值”下的最差划分能否继续分割，即是否该划分不够m个fajo月，不够说明当前“最大值”大了，取right=mid继续二分，反之取left=mid+1即可
def check(mid,m):
    total,count=0,0
    for expense in cost:
        total+=expense
        if total>mid:
            total=expense
            count+=1
    return count<m

def min_max_fajo():
    total=sum(cost)
    left,right=max(cost),total
    while left<right:
        mid=(left+right)//2
        if check(mid,m):
            right=mid
        else:
            left=mid+1
    return left

n,m=map(int,input().split())
cost=[int(input()) for _ in range(n)]
print(min_max_fajo())
```

## 道路

```
# 很明显的dijkstra
import heapq
def dijkstra():
    heap=[(0,0,1)]
    visited=set()
    while heap:
        length,cost,city=heapq.heappop(heap)
        if city==n and cost<=k:
            return length
        if city in visited or cost>k:
            continue
        visited.add(city)
        for nbr,nlenth,ncost in edges[city]:
            if nbr not in visited and cost+ncost<=k:
                heapq.heappush(heap,(length+nlenth,cost+ncost,nbr))
        visited.remove(city)
    return -1
```

```
k=int(input())
n=int(input())
edges=[[] for _ in range(n+1)]
for _ in range(int(input())):
    s,d,l,t=map(int,input().split())
    edges[s].append((d,l,t))
print(dijkstra())
```

## 食物链

```
# 并查集
def find(x):
    if parent[x]!=x:
        parent[x]=find(parent[x])
    return parent[x]

n,k=map(int,input().split())
parent =list(range(3*n+1))
#分别用x+n和x+2n表示被x吃的和x吃的种类
f_count=0
for _ in range(k):
    d,x,y=map(int,input().split())
    if x>n or y>n:
        f_count+=1
        continue
    if d==1:
        if find(x+n)==find(y) or find(y+n)==find(x):
            f_count+=1
            continue
        #x和y为同类时合并x和y以及其吃与被吃的种类
        parent[find(x)]=find(y)
        parent[find(x+n)]=find(y+n)
        parent[find(x+2*n)]=find(y+2*n)
    else:
        if find(x)==find(y) or find(y+n)==find(x):
            f_count+=1
            continue
        parent[find(x)]=find(y+2*n)
        parent[find(x+n)]=find(y)
        parent[find(x+2*n)]=find(y+n)
print(f_count)
```

## 快速堆猪

```
class PigStack:
    def __init__(self):
        self.pig_stack=[]
        self.min_weights=[]

    def push(self,weight):
```

```
        self.pig_stack.append(weight)
        if not self.min_weights or weight<self.min_weights[-1]:
            self.min_weights.append(weight)
        else:
            self.min_weights.append(self.min_weights[-1])

    def pop(self):
        if self.pig_stack:
            self.pig_stack.pop()
            self.min_weights.pop()

    def get_min_weight(self):
        if not self.min_weights:
            return None
        return self.min_weights[-1]

pig_stack=PigStack()
while True:
    try:
        command=input().split()
        if command[0]=='push':
            pig_stack.push(int(command[1]))
        elif command[0]=='pop':
            pig_stack.pop()
        else:
            min_weight=pig_stack.get_min_weight()
            if min_weight is not None:
                print(min_weight)
    except EOFError:
        break
```

## 括号嵌套树

```
def parse_tree(s):
    pre=[]
    post=[]
    for char in s:
        post.append(char)
        if char.isupper():
            pre.append(char)
        elif char==')':
            post.pop()
            a=''
            while post[-1]!='(':
                if post[-1]!=',':
                    a+=post.pop()
                else:
                    post.pop()
            post.pop()
            a=post.pop()+a
            post+=a[::-1]
    return pre,post
```

```
s=input()
pre,post=parse_tree(s)
print(''.join(pre))
print(''.join(post))
```

## 动态中位数

```
# 维护两个堆，一个最大堆，一个最小堆
import heapq
def dynamic_median(nums):
    min_heap=[]
    max_heap=[]
    medians=[]
    for i,num in enumerate(nums):
        if len(max_heap)==0 or num<=-max_heap[0]:
            heapq.heappush(max_heap,-num)
        else:
            heapq.heappush(min_heap,num)
        if len(max_heap)>len(min_heap)+1:
            heapq.heappush(min_heap,-heapq.heappop(max_heap))
        elif len(min_heap)>len(max_heap):
            heapq.heappush(max_heap,-heapq.heappop(min_heap))
        if (i+1)%2==1:
            medians.append(-max_heap[0])
    return medians

for _ in range(int(input())):
    nums=list(map(int,input().split()))
    res=dynamic_median(nums)
    print(len(res))
    print(*res,sep=' ')
```

## 奶牛排队

```
# 单调栈
def sorted_cows(heights):
    left=[0]*n
    stack=[]
    for i in range(n):
        while stack and heights[stack[-1]]<heights[i]:
            stack.pop()
        left[i]=stack[-1]+1 if stack else 0
        stack.append(i)

    right=[n]*n
    stack=[]
    for i in range(n-1,-1,-1):
        while stack and heights[stack[-1]]>heights[i]:
            stack.pop()
```

```
        right[i]=stack[-1]+1 if stack else n
        stack.append(i)

    max_length=0
    for i in range(n-1,-1,-1):
        for j in range(left[i],i):
            if right[j]>i:
                max_length=max(max_length,i-j+1)
                break
        if i<=max_length:
            break
    return max_length

n=int(input())
heights=[int(input()) for _ in range(n)]
print(sorted_cows(heights))
```

## 兔子与樱花

```
# dijkstra
import heapq
def dijkstra(graph,start):
    distances={node: float('inf') for node in graph}
    distances[start]=0
    pq=[(0,start)]
    while pq:
        curr_dist,curr_node=heapq.heappop(pq)
        if curr_dist>distances[curr_node]:
            continue
        for neighbor,distance in graph[curr_node]:
            new_dist=curr_dist+distance
            if new_dist<distances[neighbor]:
                distances[neighbor]=new_dist
                heapq.heappush(pq,(new_dist,neighbor))
    return distances

n=int(input())
locations=[input() for _ in range(n)]
roads={}
for _ in range(int(input())):
    start,end,distance=input().split()
    distance=int(distance)
    if start in roads:
        roads[start].append((end,distance))
    else:
        roads[start]=[(end,distance)]
    if end in roads:
        roads[end].append((start,distance))
    else:
        roads[end]=[(start,distance)]
routes=[input().split() for _ in range(int(input()))]
graph={location:[] for location in locations}
for start,edges in roads.items():
```

```
    graph[start]=edges
for start,end in routes:
    distances=dijkstra(graph,start)
    shortest_distance=distances[end]
    path=[]
    curr_node=end
    while curr_node!=start:
        path.append(curr_node)
        neighbors=graph[curr_node]
        prev_node=None
        for neighbor,distance in neighbors:
            if distances[neighbor] + distance==distances[curr_node]:
                prev_node=neighbor
                break
        path.append(f'({distance})')
        curr_node=prev_node
    path.append(start)
    print('->'.join(reversed(path)))
```

## 二叉树的操作

```
class Node:
    def __init__(self,val):
        self.val=val
        self.left=None
        self.right=None

def swap_nodes(nodes,x,y):
    for node in nodes:
        if node.left and node.left.val in[x,y]:
            node.left=nodes[y] if node.left.val==x else nodes[x]
        if node.right and node.right.val in[x,y]:
            node.right=nodes[y] if node.right.val==x else nodes[x]

def find_leftmost(node):
    while node and node.left:
        node=node.left
    return node.val if node else -1

for _ in range(int(input())):
    n,m=map(int,input().split())
    nodes=list(map(Node,range(n)))
    for _ in range(n):
        x,y,z=map(int,input().split())
        if y!=-1:
            nodes[x].left=nodes[y]
        if z!=-1:
            nodes[x].right=nodes[z]
    for _ in range(m):
        operation=list(map(int,input().split()))
        if operation[0]==1:
            x,y=operation[1:]
            swap_nodes(nodes,x,y)
```

```
        else:
            print(find_leftmost(nodes[operation[1]]))
```