常用语法

```
map(function, iterable, ...) #python3返回迭代器
l.count()
sorted([(),(),()])
dict1.items() #返回元组
split()直接就可以分割空格，不需要" "参数
deque的用法 appendleft() pop() popleft() append()
```

错题本

22067: 快速堆猪（heap用法）

http://cs101.openjudge.cn/dsapre/22067/

小明有很多猪，他喜欢玩叠猪游戏，就是将猪一头头叠起来。猪叠上去后，还可以把顶上的猪拿下来。小明知道每头猪的重量，而且他还随时想知道叠在那里的猪最轻的是多少斤。

输入

有三种输入

1. push n n是整数(0<=0 <=20000)，表示叠上一头重量是n斤的新猪
2. pop 表示将猪堆顶的猪赶走。如果猪堆没猪，就啥也不干
3. min 表示问现在猪堆里最轻的猪多重。如果猪堆没猪，就啥也不干

输入总数不超过100000条

输出

对每个min输入，输出答案。如果猪堆没猪，就啥也不干

样例输入

```
pop
min
push 5
push 2
push 3
min
push 4
min
```

样例输出

```
2
2
```

来源: Guo wei

用辅助栈：用一个单调栈维护最小值，再用另外一个栈维护其余的值。

每次push时，在辅助栈中加入当前最轻的猪的体重，pop时也同步pop，这样栈顶始终是当前猪堆中最轻的体重，查询时直接输出即可 字典标记，懒删除

```
import heapq
from collections import defaultdict

out = defaultdict(int)
pigs_heap = []
pigs_stack = []

while True:
    try:
        s = input()
    except EOFError:
        break

    if s == "pop":
        if pigs_stack:
            out[pigs_stack.pop()] += 1
    elif s == "min":
        if pigs_stack:
            while True:
                x = heapq.heappop(pigs_heap)
                if not out[x]:
                    heapq.heappush(pigs_heap, x)
                    print(x)
                    break
                out[x] -= 1
    else:
        y = int(s.split()[1])
        pigs_stack.append(y)
        heapq.heappush(pigs_heap, y)
```

04082: 树的镜面映射

http://cs101.openjudge.cn/practice/04082/

思路：这个题第一眼看实在是没什么想法，因为除了先用树的数据结构重构树，别的方法我也想不到了，但是重构树的话，我不知道怎么可以把二叉树转化为树，然后再把树镜面反射，所以我就看题解了。 print_tree用栈实现镜面反射效果的那块，确实想不到 这个题目有三点：

1. 把伪满二叉树变为二叉树

2. 把二叉树变为树
3. 把树镜面反射

把二叉树变成树之前只知道怎么在纸上做，没写过代码，步骤就是左儿子右兄弟

build_tree真的很难看懂，第一次看这种递归建树的代码

代码

```
from collections import deque

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

def create_node():
    return TreeNode('')

def build_tree(tempList, index):
    # 创建一个新的节点
    node = create_node()
    # 设置节点的值
    node.x = tempList[index][0]
    # 如果当前节点的第二个值为'0'，说明它有子节点
    if tempList[index][1] == '0':
        # 递归构建左子节点，并更新索引
        index += 1
        child, index = build_tree(tempList, index)
        # 将左子节点添加到当前节点的children列表中
        node.children.append(child)
        # 递归构建右子节点，并更新索引
        index += 1
        child, index = build_tree(tempList, index)
        # 将右子节点添加到当前节点的children列表中
        node.children.append(child)
    # 返回当前节点和索引
    return node, index


#这里没用重新建一个树，而是利用右儿子是兄弟的特性，直接找到一层的原树的兄弟，这个顺序是从root开始，从上往下的
def print_tree(p):
    Q = deque()
    s = deque()

    # 遍历右子节点并将非虚节点加入栈s
    while p is not None:
        if p.x != '$':
            s.append(p)
        p = p.children[1] if len(p.children) > 1 else None #如果节点p有超过一个子节点（也就是说，它有右子节点），那么p就被设置为它的右子节点（p.children[1]）。否则，如果节点p没有右子节点，那么p就被设置为None。
```

```
    # 将栈s中的节点逆序放入队列Q
    while s:
        Q.append(s.pop())

    # 宽度优先遍历队列Q并打印节点值
    while Q:
        p = Q.popleft()
        print(p.x, end=' ')

        # 如果节点有左子节点，将左子节点及其右子节点加入栈s
        if p.children:
            p = p.children[0]
            ############################这部分和上面一样
            while p is not None:
                if p.x != '$':
                    s.append(p)
                p = p.children[1] if len(p.children) > 1 else None
            ############################

            # 将栈s中的节点逆序放入队列Q
            while s:
                Q.append(s.pop())


n = int(input())
tempList = input().split()

# 构建多叉树
root, _ = build_tree(tempList, 0)

# 执行宽度优先遍历并打印镜像映射序列
print_tree(root)



######################################################################
def tree_height(root):  # 计算二叉树高度
    if not root:
        return -1
    else:
        return max(tree_height(root.left), tree_height(root.right)) + 1


print(f'{h_orig} => {h_bin}')


def prase_tree(root: TreeNode):
    if root.children:
        root.left = prase_tree(root.children.pop(0)) #从左边弹出一个节点
        cur = root.left
        while root.children:
            cur.right = prase_tree(root.children.pop(0))
            cur = cur.right
```

```
    return root
#我就是在这里卡住了，一直想不到怎么转换成二叉树，原来还是递归!
```

08581: 扩展二叉树

```
def postorder_traversal(root):
    if root is None:
        return ''
    return postorder_traversal(root.left) + postorder_traversal(root.right) +
root.x

def midorder_traversal(root):
    if root is None:
        return ''
    return midorder_traversal(root.left) + root.x + midorder_traversal(root.right)
```

```
while True:
    try:
        preorder, inorder = input().split()
        root = build_tree(preorder, inorder)
        print(postorder_traversal(root))
    except EOFError:
        break
```

28046: 词梯

bfs, http://cs101.openjudge.cn/practice/28046/

思路：讲义里面的，我自己加了点注释辅助理解，把建图部分改一下就好，不是读取txt文件了

代码

```
import sys
from collections import deque

class Graph:
    def __init__(self): #键是'FOOL',值是class Vertex
        self.vertices = {}  # 字典，用于存储图中的所有顶点，键是顶点的键，值是顶点对象
        self.num_vertices = 0  # 整数，用于存储图中的顶点数量

    def add_vertex(self, key):
        self.num_vertices = self.num_vertices + 1  # 顶点数量加1
        new_vertex = Vertex(key)  # 创建一个新的顶点
        self.vertices[key] = new_vertex  # 将新的顶点添加到字典中
        return new_vertex  # 返回新的顶点

    def get_vertex(self, n):
```

```
        if n in self.vertices:  # 如果n是字典的键
            return self.vertices[n]  # 返回键为n的顶点
        else:
            return None  # 否则，返回None

    def __len__(self):
        return self.num_vertices  # 返回图中的顶点数量

    def __contains__(self, n):
        return n in self.vertices  # 检查n是否是字典的键，即检查n是否是图中的一个顶点

    def add_edge(self, f, t, cost=0):
        if f not in self.vertices:  # 如果f不是图中的一个顶点
            nv = self.add_vertex(f)  # 添加一个新的顶点f
        if t not in self.vertices:  # 如果t不是图中的一个顶点
            nv = self.add_vertex(t)  # 添加一个新的顶点t
        self.vertices[f].add_neighbor(self.vertices[t], cost)  # 在顶点f和顶点t之间
添加一条边，边的权重是cost

    def get_vertices(self):
        return list(self.vertices.keys())

    def __iter__(self):
        return iter(self.vertices.values())


class Vertex:
    def __init__(self, num):
        self.key = num  # 顶点的键
        self.connectedTo = {}  # 字典，用于存储与这个顶点相连的其他顶点及其权重
        self.color = 'white'  # 顶点的颜色，用于图的搜索和遍历
        self.distance = sys.maxsize  # 顶点的距离，用于图的搜索和遍历
        self.previous = None  # 顶点的前一个顶点，用于图的搜索和遍历
        self.disc = 0  # 顶点的发现时间，用于图的搜索和遍历
        self.fin = 0  # 顶点的完成时间，用于图的搜索和遍历

    def add_neighbor(self, nbr, weight=0):
        self.connectedTo[nbr] = weight  # 添加一个邻居顶点，nbr是邻居顶点，weight是从
这个顶点到邻居顶点的边的权重

    # def __lt__(self,o):
    #     return self.id < o.id  # 比较两个顶点，这个方法被注释掉了，所以在当前的
Vertex类中并没有被使用

    # def setDiscovery(self, dtime):
    #     self.disc = dtime  # 设置顶点的发现时间，这个方法被注释掉了，所以在当前的
Vertex类中并没有被使用

    # def setFinish(self, ftime):
    #     self.fin = ftime  # 设置顶点的完成时间，这个方法被注释掉了，所以在当前的
Vertex类中并没有被使用
    #
    # def getFinish(self):
    #     return self.fin
```

```
    #
    # def getDiscovery(self):
    #     return self.disc

    def get_neighbors(self):
        return self.connectedTo.keys()

    # def getWeight(self, nbr):
    #     return self.connectedTo[nbr]

    # def __str__(self):
    #     return str(self.key) + ":color " + self.color + ":disc " +
str(self.disc) + ":fin " + str(
    #         self.fin) + ":dist " + str(self.distance) + ":pred \n\t[" +
str(self.previous) + "]\n"




def build_graph(namelist):
    buckets = {}
    the_graph = Graph()
    all_words = namelist
    # all_words = ["bane", "bank", "bunk", "cane", "dale", "dunk", "foil", "fool",
"kale",
    #              "lane", "male", "mane", "pale", "pole", "poll", "pool", "quip",
    #              "quit", "rain", "sage", "sale", "same", "tank", "vain", "wane"
    #              ]

    # create buckets of words that differ by 1 letter
    for line in all_words:
        word = line.strip()
        for i, _ in enumerate(word):
            bucket = f"{word[:i]}_{word[i + 1:]}"
            buckets.setdefault(bucket, set()).add(word)  # 在字典buckets中添加键值
对。如果bucket不是字典的键，就添加一个新的键bucket，值是一个空的集合。然后，将word添加到
键为bucket的集合中。

    # connect different words in the same bucket
    # 这个程序有个bug，他不能把孤立的单词加入到图中，所以我们需要手动添加——韩萱
    for similar_words in buckets.values():
        if len(similar_words) > 1:
            for word1 in similar_words:
                for word2 in similar_words - {word1}:
                    the_graph.add_edge(word1, word2)
        else:
            if list(similar_words)[0] not in the_graph.get_vertices():
                the_graph.add_vertex(list(similar_words)[0])

    return the_graph


n = int(input())
namelist = []
```

```
for i in range(n):
    namelist.append(input())
g = build_graph(namelist)


def bfs(start):
    start.distance = 0
    start.previous = None
    vert_queue = deque()
    vert_queue.append(start)
    while len(vert_queue) > 0:
        current = vert_queue.popleft()  # 取队首作为当前顶点
        for neighbor in current.get_neighbors():   # 遍历当前顶点的邻接顶点
            if neighbor.color == "white":
                neighbor.color = "gray"
                neighbor.distance = current.distance + 1
                neighbor.previous = current
                vert_queue.append(neighbor)
        current.color = "black" # 当前顶点已经处理完毕，设黑色

"""
BFS 算法主体是两个循环的嵌套：while-for
    while 循环对图中每个顶点访问一次，所以是 O(|V|);
    嵌套在 while 中的 for，由于每条边只有在其起始顶点u出队的时候才会被检查一次，
    而每个顶点最多出队1次，所以边最多被检查次，一共是 O(|E|);
    综合起来 BFS 的时间复杂度为 0(V+|E|)

词梯问题还包括两个部分算法
    建立 BFS 树之后，回溯顶点到起始顶点的过程，最多为 O(|V|)
    创建单词关系图也需要时间，时间是 O(|V|+|E|) 的，因为每个顶点和边都只被处理一次
"""



#bfs(g.getVertex("fool"))

# 以FOOL为起点，进行广度优先搜索，从FOOL到SAGE的最短路径，
# 并为每个顶点着色、赋距离和前驱。
start, end = input().split()
bfs(g.get_vertex(start)) #get_vertex是Graph类的方法，返回字典vertices中键为"FOOL"的
值，即Vertex类的对象


# 回溯路径
def traverse(starting_vertex):
    ans = []
    current = starting_vertex
    while (current.previous):
        ans.append(current.key)
        current = current.previous
    ans.append(current.key)

    return ans
```

```
# ans = traverse(g.get_vertex("sage"))
#接下来是输出答案的部分
if g.get_vertex(end).previous == None:  # 如果SAGE的前驱是None，说明SAGE不可达
    print("NO")
else:
    ans = traverse(g.get_vertex(end)) # 从SAGE开始回溯，逆向打印路径，直到FOOL
    print(*ans[::-1]) #[::-1]是Python的切片操作，::表示从头到尾，-1表示步长为-1，也就是逆序
```

04123: 马走日(图的面向对象)

dfs, http://cs101.openjudge.cn/practice/04123

```
# pylint: skip-file
import sys

class Graph:
    def __init__(self):
        self.vertices = {}
        self.num_vertices = 0

    def add_vertex(self, key):
        self.num_vertices = self.num_vertices + 1
        new_ertex = Vertex(key)
        self.vertices[key] = new_ertex
        return new_ertex

    def get_vertex(self, n):
        if n in self.vertices:
            return self.vertices[n]
        else:
            return None

    def __len__(self):
        return self.num_vertices

    def __contains__(self, n):
        return n in self.vertices

    def add_edge(self, f, t, cost=0):
        if f not in self.vertices:
            nv = self.add_vertex(f)
        if t not in self.vertices:
            nv = self.add_vertex(t)
        self.vertices[f].add_neighbor(self.vertices[t], cost)
        #self.vertices[t].add_neighbor(self.vertices[f], cost)

    def getVertices(self):
        return list(self.vertices.keys())
```

```
    def __iter__(self):
        return iter(self.vertices.values())


class Vertex:
    def __init__(self, num):
        self.key = num
        self.connectedTo = {}
        self.color = 'white'
        self.distance = sys.maxsize
        self.previous = None
        self.disc = 0
        self.fin = 0

    def __lt__(self,o):
        return self.key < o.key

    def add_neighbor(self, nbr, weight=0):
        self.connectedTo[nbr] = weight


    # def setDiscovery(self, dtime):
    #     self.disc = dtime
    #
    # def setFinish(self, ftime):
    #     self.fin = ftime
    #
    # def getFinish(self):
    #     return self.fin
    #
    # def getDiscovery(self):
    #     return self.disc

    def get_neighbors(self):
        return self.connectedTo.keys()

    # def getWeight(self, nbr):
    #     return self.connectedTo[nbr]

    def __str__(self):
        return str(self.key) + ":color " + self.color + ":disc " + str(self.disc)
+ ":fin " + str(
            self.fin) + ":dist " + str(self.distance) + ":pred \n\t[" +
str(self.previous) + "]\n"



def knight_graph(n,m):
    kt_graph = Graph()
    for row in range(n):           #遍历每一行
        for col in range(m):       #遍历行上的每一个格子
            node_id = pos_to_node_id(row, col, n, m) #把行、列号转为格子ID
            new_positions = gen_legal_moves(row, col, n, m) #按照 马走日，返回下一步
可能位置
```

```
            for row2, col2 in new_positions:
                other_node_id = pos_to_node_id(row2, col2, n, m) #下一步的格子ID
                kt_graph.add_edge(node_id, other_node_id) #在骑士周游图中为两个格子
加一条边
    return kt_graph

def pos_to_node_id(x, y, n, m):
    return x * m + y

def gen_legal_moves(row, col, n, m):
    new_moves = []
    move_offsets = [                        # 马走日的8种走法
        (-1, -2),  # left-down-down
        (-1, 2),  # left-up-up
        (-2, -1),  # left-left-down
        (-2, 1),  # left-left-up
        (1, -2),  # right-down-down
        (1, 2),  # right-up-up
        (2, -1),  # right-right-down
        (2, 1),  # right-right-up
    ]
    for r_off, c_off in move_offsets:
        if (                                # #检查，不能走出棋盘
            0 <= row + r_off < n
            and 0 <= col + c_off < m
        ):
            new_moves.append((row + r_off, col + c_off))
    return new_moves

# def legal_coord(row, col, board_size):
#     return 0 <= row < board_size and 0 <= col < board_size


def knight_tour(n, path, u, limit):
    global ans
    u.color = "gray"
    path.append(u)              #当前顶点涂色并加入路径
    if n < limit:
        neighbors = ordered_by_avail(u) #对所有的合法移动依次深入
        #neighbors = sorted(list(u.get_neighbors()))
        i = 0

        for nbr in neighbors:
            if nbr.color == "white" and \
                knight_tour(n + 1, path, nbr, limit):    #选择“白色”未经深入的点，层
次加一，递归深入
                nbr.color = "white" #之后把这个nbr重新标记回白色，因为我还想探索经过这
个nbr其他的可能性。dfs完nbr之后，不要回溯，不要回溯，不要回溯！不然第一次return True之
后，就回一直回溯到第一层函数了，这样就不对了
        else:                           #所有的“下一步”都试了走不通
            path.pop()                  #回溯，从路径中删除当前顶点
            u.color = "white"           #当前顶点改回白色
            return False
    else:
```

```
        u.color = "white"
        ans += 1
        return True


def ordered_by_avail(n):
    res_list = []
    for v in n.get_neighbors():
        if v.color == "white":
            c = 0
            for w in v.get_neighbors():
                if w.color == "white":
                    c += 1
            res_list.append((c,v))
    res_list.sort(key = lambda x: x[0])
    return [y[1] for y in res_list]

# class DFSGraph(Graph):
#     def __init__(self):
#         super().__init__()
#         self.time = 0                   #不是物理世界，而是算法执行步数
#
#     def dfs(self):
#         for vertex in self:
#             vertex.color = "white"      #颜色初始化
#             vertex.previous = -1
#         for vertex in self:             #从每个顶点开始遍历
#             if vertex.color == "white":
#                 self.dfs_visit(vertex)  #第一次运行后还有未包括的顶点
#                                         # 则建立森林
#
#     def dfs_visit(self, start_vertex):
#         start_vertex.color = "gray"
#         self.time = self.time + 1       #记录算法的步骤
#         start_vertex.discovery_time = self.time
#         for next_vertex in start_vertex.get_neighbors():
#             if next_vertex.color == "white":
#                 next_vertex.previous = start_vertex
#                 self.dfs_visit(next_vertex)     #深度优先递归访问
#         start_vertex.color = "black"
#         self.time = self.time + 1
#         start_vertex.closing_time = self.time


def main():
    global ans
    T = int(input())
    for i in range(T):
        ans = 0
        n, m, x, y = map(int, input().split())
        g = knight_graph(n, m)
        start_vertex = g.get_vertex(pos_to_node_id(x, y, n, m))
        path = []
        knight_tour(0, path, start_vertex, n * m - 1)
```

```
        print(ans)


if __name__ == "__main__":
    main()
```

04115: 鸣人和佐助（方格地图题）

bfs, http://cs101.openjudge.cn/practice/04115/

思路：稍复杂的bfs问题。visited需要维护经过时的最大查克拉数，只有大于T值时候才能通过，然后就是常见bfs。

```
# 夏天明 元培学院

from collections import deque

# 读取输入的行数M、列数N和初始能量T
M, N, T = map(int, input().split())
# 读取输入的图，将其存储为一个二维列表
graph = [list(input()) for i in range(M)]
# 定义四个方向：右、下、左、上
direc = [(0,1), (1,0), (-1,0), (0,-1)]
# 初始化起点和终点
start, end = None, None
# 找到起点
for i in range(M):
    for j in range(N):
        if graph[i][j] == '@':
            start = (i, j)

# 定义广度优先搜索函数
def bfs():
    # 初始化队列，将起点和初始能量T添加到队列中
    q = deque([start + (T, 0)])
    # 初始化访问列表，记录每个位置的能量值，初始值为-1
    visited = [[-1]*N for i in range(M)]
    # 将起点的能量值设置为T
    visited[start[0]][start[1]] = T
    # 当队列不为空时，进行循环
    while q:
        # 从队列中取出一个元素，包括其坐标、能量值和时间
        x, y, t, time = q.popleft()
        # 时间增加1
        time += 1
        # 遍历四个方向
        for dx, dy in direc:
            # 如果新的坐标在图内
            if 0<=x+dx<M and 0<=y+dy<N:
                # 如果新的位置是'*'，并且当前能量值大于该位置的能量值
```

```
                if (elem := graph[x+dx][y+dy]) == '*' and t > visited[x+dx][y+dy]:
                    # 更新该位置的能量值
                    visited[x+dx][y+dy] = t
                    # 将新的位置和能量值添加到队列中
                    q.append((x+dx, y+dy, t, time))
                # 如果新的位置是'#'，并且当前能量值大于0，并且当前能量值-1大于该位置的
能量值
                elif elem == '#' and t > 0 and t-1 > visited[x+dx][y+dy]:
                    # 更新该位置的能量值
                    visited[x+dx][y+dy] = t-1
                    # 将新的位置和能量值-1添加到队列中
                    q.append((x+dx, y+dy, t-1, time))
                # 如果新的位置是'+'，则返回时间
                elif elem == '+':
                    return time
    # 如果没有找到'+'，则返回-1
    return -1

# 打印广度优先搜索的结果
print(bfs())
```

dijkstra和图的oop

```
import heapq
import sys

class Vertex:
    def __init__(self, key):
        self.id = key
        self.connectedTo = {}
        self.distance = sys.maxsize
        self.pred = None

    def addNeighbor(self, nbr, weight=0):
        self.connectedTo[nbr] = weight

    def getConnections(self):
        return self.connectedTo.keys()

    def getWeight(self, nbr):
        return self.connectedTo[nbr]

    def __lt__(self, other):
        return self.distance < other.distance

class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self, key):
```

```
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        self.numVertices += 1
        return newVertex

    def getVertex(self, n):
        return self.vertList.get(n)

    def addEdge(self, f, t, cost=0):
        if f not in self.vertList:
            self.addVertex(f)
        if t not in self.vertList:
            self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], cost)

def dijkstra(graph, start):
    # 初始化优先队列
    pq = []
    # 将起点的距离设置为0
    start.distance = 0
    # 将起点添加到优先队列中
    heapq.heappush(pq, (0, start))
    # 初始化已访问的顶点集合
    visited = set()

    # 当优先队列不为空时，进行循环
    while pq:
        # 从优先队列中取出距离最小的顶点
        currentDist, currentVert = heapq.heappop(pq)
        # 如果这个顶点已经被访问过，就跳过这次循环
        if currentVert in visited:
            continue
        # 将这个顶点添加到已访问的顶点集合中
        visited.add(currentVert)

        # 遍历这个顶点的所有邻居
        for nextVert in currentVert.getConnections():
            # 计算从起点到邻居的距离
            newDist = currentDist + currentVert.getWeight(nextVert)
            # 如果这个距离小于邻居的当前距离，就更新邻居的距离和前驱顶点
            if newDist < nextVert.distance:
                nextVert.distance = newDist
                nextVert.pred = currentVert
                # 将邻居添加到优先队列中
                heapq.heappush(pq, (newDist, nextVert))
```

20106: 走山路（迪杰斯克拉）

```
# 23 蒋子轩
from heapq import heappop, heappush
```

```
# 定义广度优先搜索函数
def bfs(x1, y1):
    # 初始化优先队列，将起点和时间0添加到队列中
    q = [(0, x1, y1)]
    # 初始化已访问的位置集合
    visited = set()
    # 当队列不为空时，进行循环
    while q:
        # 从队列中取出一个元素，包括其时间和位置
        t, x, y = heappop(q)
        # 如果这个位置已经被访问过，就跳过这次循环
        if (x, y) in visited:
            continue
        # 将这个位置添加到已访问的位置集合中
        visited.add((x, y))
        # 如果这个位置是终点，就返回时间
        if x == x2 and y == y2:
            return t
        # 遍历四个方向
        for dx, dy in dir:
            # 计算新的位置
            nx, ny = x+dx, y+dy
            # 如果新的位置在地图内，且不是'#'，且没有被访问过
            if 0 <= nx < m and 0 <= ny < n and \
                    ma[nx][ny] != '#' and (nx, ny) not in visited:
                # 计算新的时间
                nt = t+abs(int(ma[nx][ny])-int(ma[x][y]))
                # 将新的位置和时间添加到队列中
                heappush(q, (nt, nx, ny))
    # 如果没有找到终点，就返回'NO'
    return 'NO'


# 读取输入的行数m、列数n和查询次数p
m, n, p = map(int, input().split())
# 读取输入的地图，将其存储为一个二维列表
ma = [list(input().split()) for _ in range(m)]
# 定义四个方向：下、上、右、左
dir = [(1, 0), (-1, 0), (0, 1), (0, -1)]
# 对每一次查询
for _ in range(p):
    # 读取输入的起点和终点
    x1, y1, x2, y2 = map(int, input().split())
    # 如果起点或终点是'#'，就打印'NO'，并跳过这次查询
    if ma[x1][y1] == '#' or ma[x2][y2] == '#':
        print('NO')
        continue
    # 打印广度优先搜索的结果
    print(bfs(x1, y1))
```

05442: 兔子与星空（最小生成树kruscal）

```python
# 蔡嘉华 物理学院
class DisjSet:
    def __init__(self, n):
        self.parent = [i for i in range(n)]
        self.rank = [0]*n

    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):
        xset, yset = self.find(x), self.find(y)
        if self.rank[xset] > self.rank[yset]:
            self.parent[yset] = xset
        else:
            self.parent[xset] = yset
            if self.rank[xset] == self.rank[yset]:
                self.rank[yset] += 1

def kruskal(n, edges):
    # 初始化一个并查集
    dset = DisjSet(n)
    # 将所有的边按照权重进行排序
    edges.sort(key = lambda x:x[2])
    # 初始化最小生成树的权重和为0
    sol = 0
    # 遍历所有的边
    for u, v, w in edges:
        # 将顶点的字符表示转换为数字表示
        u, v = ord(u)-65, ord(v)-65
        # 如果两个顶点不在同一个集合中
        if dset.find(u) != dset.find(v):
            # 将两个集合合并
            dset.union(u, v)
            # 将这条边的权重加到最小生成树的权重和中
            sol += w
    # 如果最后并查集中的集合数量大于1，说明图不连通，返回-1
    if len(set(dset.find(i) for i in range(n))) > 1:
        return -1
    # 返回最小生成树的权重和
    return sol

# 读取顶点的数量
n = int(input())
# 初始化边的列表
edges = []
# 读取每个顶点的边
for _ in range(n-1):
    arr = input().split()
    root, m = arr[0], int(arr[1])
    # 将每条边的起点、终点和权重添加到边的列表中
    for i in range(m):
```

```
        edges.append((root, arr[2+2*i], int(arr[3+2*i])))
# 打印最小生成树的权重和
print(kruskal(n, edges))
```

01258: Agri-Net（prim也是dp）

以顶点A作为起点，将A到其他所有顶点的距离都初始化为无穷大。检查A的相邻顶点后，可以更新从A到B和C的距离，因为实际的距离小于无穷大。更新距离之后，B和C被移到优先级队列的头部。并且，它们的前驱顶点被设置为A。注意，我们还没有把B和C添加到生成树中。只有在从优先级队列中移除时，顶点才会被添加到生成树中。

```
#王昊 光华管理学院
from heapq import heappop, heappush

# 无限循环，直到输入无效才退出
while True:
    try:
        # 读取顶点的数量
        n = int(input())
    except:
        # 如果输入无效，退出循环
        break
    # 初始化邻接矩阵和当前节点
    mat, cur = [], 0
    # 读取邻接矩阵
    for i in range(n):
        mat.append(list(map(int, input().split())))
    # 初始化距离数组、已访问节点集合、优先队列和最小生成树的权重和
    d, v, q, cnt = [100000 for i in range(n)], set(), [], 0
    # 将起点的距离设为0
    d[0] = 0
    # 将起点添加到优先队列中
    heappush(q, (d[0], 0))
    # 当优先队列不为空时，进行循环
    while q:
        # 从优先队列中取出一个节点，包括其距离和编号
        x, y = heappop(q)
        # 如果这个节点已经被访问过，就跳过这次循环
        if y in v:
            continue
        # 将这个节点添加到已访问节点集合中
        v.add(y)
        # 将这个节点的距离加到最小生成树的权重和中
        cnt += d[y]
        # 遍历所有的节点
        for i in range(n):
            # 如果这个节点的距离大于当前节点到这个节点的距离
            if d[i] > mat[y][i]:
                # 更新这个节点的距离
                d[i] = mat[y][i]
                # 将这个节点添加到优先队列中
```

```
                heappush(q, (d[i], i))
    # 打印最小生成树的权重和
    print(cnt)
```

28203:【模板】单调栈

```
n = int(input())  # 读取输入的整数n
a = list(map(int, input().split()))  # 读取输入的n个整数，并将它们存储在列表a中
stack = []  # 初始化一个空栈

# 遍历列表a中的每个元素
for i in range(n):
    # 当栈不为空，且栈顶元素小于当前元素时，进行循环
    while stack and a[stack[-1]] < a[i]:
        # 将栈顶元素出栈，并将其在列表a中的值更新为i + 1
        a[stack.pop()] = i + 1

    # 将当前元素的索引入栈
    stack.append(i)

# 当栈不为空时，进行循环
while stack:
    # 将栈顶元素出栈，并将其在列表a中的值更新为0
    a[stack[-1]] = 0
    stack.pop()

# 打印列表a中的所有元素
print(*a)
```

28190: 奶牛排队（单调栈）

利用单调栈， left_bound用于记录以当前点为最右端，满足条件的最左端的索引减1； right_bound用于记录以当前节点为最左端，满足条件的最右端的索引加1，最终答案就是两段拼起来之后的最长长度。

```
"""
https://www.luogu.com.cn/problem/solution/P6510
简化题意：求一个区间，使得区间左端点最矮，区间右端点最高，且区间内不存在与两端相等高度的奶牛，输出这个区间的长度。
我们设左端点为 A ,右端点为 B
因为 A 是区间内最矮的，所以 [A.B]中，都比 A 高。所以只要 A 右侧第一个 ≤A的奶牛位于 B 的右侧，则 A 合法
同理，因为B是区间内最高的，所以 [A.B]中，都比 B 矮。所以只要 B 左侧第一个 ≥B 的奶牛位于A的左侧，则 B合法
对于 “ 左/右侧第一个 ≥/≤ ” 我们可以使用单调栈维护。用单调栈预处理出 zz数组表示左，r 数组表示右。
然后枚举右端点 B寻找 A，更新 ans 即可。

这个算法的时间复杂度为 O(n)，其中 n 是奶牛的数量。
"""
```

```
N = int(input())
heights = [int(input()) for _ in range(N)]

left_bound = [-1] * N
right_bound = [N] * N

stack = []  # 单调栈，存储索引

# 求左侧第一个≥h[i]的奶牛位置
for i in range(N):
    while stack and heights[stack[-1]] < heights[i]:
        stack.pop()

    if stack:
        left_bound[i] = stack[-1]

    stack.append(i)

stack = []  # 清空栈以供寻找右边界使用

# 求右侧第一个≤h[i]的奶牛位
for i in range(N-1, -1, -1):
    while stack and heights[stack[-1]] > heights[i]:
        stack.pop()

    if stack:
        right_bound[i] = stack[-1]

    stack.append(i)

ans = 0

# for i in range(N-1, -1, -1):  # 从大到小枚举是个技巧
#     for j in range(left_bound[i] + 1, i):
#         if right_bound[j] > i:
#             ans = max(ans, i - j + 1)
#             break
#
#     if i <= ans:
#         break

for i in range(N):  # 枚举右端点 B寻找 A，更新 ans
    for j in range(left_bound[i] + 1, i):
        if right_bound[j] > i:
            ans = max(ans, i - j + 1)
            break
print(ans)
```