基础知识

多行读入 不确定何时结束：

```
while True:
    try:
        line = input()  #读取一行输入
        print(f"你输入的内容是：{line}")    #或者主要代码部分
    except EOFError:
        break
```

多元运算

```
print("Yes" if flag else "No")
```

浮点数的输出：

小数：f"{value:.nf}"　　#其中 n 是保留小数位数

科学计数法：　print(f"Scientific notation: {large_number:.2e}")　　# 输出：1.23e+06

无空格输出

```
print(''.join(map(str, x)))    #x 为列表或字符串
```

矩阵保护圈的写法

```
matrix = [[1] * (m + 2) for i in range(n + 2)]
for _ in range(1,n + 1):
    matrix[_][1:-1] = input()        #不带空格的输入
                      list(map(int, input().split()))   #带空格的输入
```

列表排序：

```
lst.sort(key=lambda x: x[0])
lst.sort(key=lambda x: (x[0], -x[1]))
```

枚举：　for i,x in enumerate(list),遍历 list 中的（下标，值）对

集合的使用

```
# 创建集合
my_set = {1, 2, 3, 4, 5}
another_set = set([3, 4, 5, 6, 7])
#注意：set() 用来创建集合时，它接受一个可迭代对象（如列表、元组、字符串等），因而这里set() 会自动从列表中提取元素并创建集合，而不能直接set(3, 4, 5, 6, 7)，因为set()括号里只可以有一个参数，而{}则不同。

# 添加元素
my_set.add(6)
# 删除元素（不存在元素可抛出错误）
my_set.remove(2)
# 删除不存在的元素，不会抛出错误
my_set.discard(10)
```

ASCII 码表的使用：

chr() 数字转字符 ord() 字符转数字

注意：48 – 57 对应 0 – 9；65 – 90 对应 A – Z；97 – 122 对应 a – z。

字典的使用

```
# 创建字典
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}

#通过键来搜索值
法一：print(my_dict['name'])  # 输出：Alice
法二：print(my_dict.get('name'))  # 输出：Alice
print(my_dict.get('address', 'Not Found'))  # 输出：Not Found

#通过值来搜索键
找到所有的键：
法一：keys = [key for key, value in my_dict.items() if value == search_value]
法二：keys = list(filter(lambda key: my_dict[key] == search_value, my_dict))
找到第一个符合条件的键：
key = next((key for key, value in my_dict.items() if value == search_value), None)

#添加或更新元素（键值对）：
my_dict['age'] = 26  # 更新
my_dict['country'] = 'USA'  # 添加

#向字典中某一个键下添加元素：
my_dict = {'key1': [1, 2, 3], 'key2': [4, 5]}
my_dict['key1'].append(4)
```

```
#删除键值对
法一：del my_dict['city']     # 删除 'city' 键值对
法二：age = my_dict.pop('age')
print(age)  # 输出：26

#遍历字典：
    # 遍历键
    for key in my_dict:

    # 遍历值
    for value in my_dict.values():
        print(value)

    # 遍历键值对
    for key, value in my_dict.items():
        print(f"{key}: {value}")


#字典推导式举例：
numbers = [1, 2, 3, 4, 5]
squared_dict = {n: n**2 for n in numbers}
print(squared_dict)

#字典排序：
sorted_dict = dict(sorted(my_dict.items(), key=lambda x: x[1], reverse=True))
```

排序中 Lambda 函数的使用（待补充）

```
基本模板：lambda arguments: expression  #参数：对参数进行的操作

在字典排序中：
sorted_dict = sorted(my_dict.items(), key=lambda x: x[1])
#按值升序排序，注意sorted得到的是一个列表!
#如果想要降序并转化为字典格式如下：
sorted_dict = dict(sorted(my_dict.items(), key=lambda x: x[1], reverse=True))

与map结合：
# 对列表中的每个元素进行平方操作
squared_numbers = list(map(lambda x: x ** 2, numbers))
```

```
count_dict = defaultdict(int)
count_dict['apple'] += 1
print(count_dict['apple'])  # 输出: 1
print(count_dict['banana'])  # 输出: 0，注意这里'banana'之前未定义

# 使用 list 作为工厂函数，适用于存储分组信息
group_dict = defaultdict(list)
group_dict['fruits'].append('apple')
group_dict['fruits'].append('banana')
print(group_dict['fruits'])  # 输出: ['apple', 'banana']
print(group_dict['vegetables'])  # 输出: []，空列表
```

排序：

冒泡排序：

```
for i in range(n):
    ok=True
    for j in range(0,n-i-1):
        if arr[j]>arr[j+1]:
            arr[j],arr[j+1]=arr[j+1],arr[j]
            ok=False
    if ok:
        break
```

快速随机排序：

```
def quicksort(arr, left, right):
    if left < right:
        mid = partition(arr, left, right)
        quicksort(arr, left, mid - 1)
        quicksort(arr, mid + 1, right)

def partition(arr, left, right):
    i = left
    j = right - 1
    pivot = arr[right]
    while i <= j:
```

```
        while i <= right and arr[i] < pivot:
            i += 1
        while j >= left and arr[j] >= pivot:
            j -= 1
        if i < j:
            arr[i], arr[j] = arr[j], arr[i]
    if arr[i] > pivot:
        arr[i], arr[right] = arr[right], arr[i]
    return i
```

分治排序

```
def mergeSort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        L = arr[:mid]   # Dividing the array elements
        R = arr[mid:] # Into 2 halves
        mergeSort(L) # Sorting the first half
        mergeSort(R) # Sorting the second half
        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
        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

二分搜索

取mid==(l+r)//2时，左端点取“能取到的下限”，右端点取“能取到的上限+1”
while l<r-1: 如果能行l=mid 不能行r=mid

过程：

> 估计答案范围（可以很粗略）；
> 判断有无单调性；
> 建立check函数；
> 复杂度一般为O（nlogn），10^5以上就可以考虑了。

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid  # 返回目标元素的索引
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1  # 如果未找到目标元素，返回 -1
```

差分数组

1. **差分数组定义**:
   - 对于原数组 `arr`，其差分数组 `diff` 满足:
     - `diff[0] = arr[0]`
     - `diff[i] = arr[i] - arr[i-1]` (i ≥ 1)
   - 通过差分数组可还原原数组: `arr[i] = diff[0] + diff[1] + ... + diff[i]`
2. **区间更新操作**:
   - 若需对 `arr` 的区间 `[l, r]` 统一增加 `val`，只需修改差分数组的两个端点:

     ```
     diff[l] += val
     if r + 1 < len(diff):  # 防止越界
         diff[r + 1] -= val
     ```

   - 通过前缀和还原更新后的数组。

```
def corpFlightBookings(bookings, n):
    diff = [0] * (n + 1)  # 差分数组（多开一位防越界）

    for first, last, seats in bookings:
        diff[first - 1] += seats  # 注意下标从0开始
        if last < n:  # 防止越界
            diff[last] -= seats

    # 前缀和还原数组
    res = [diff[0]]
    for i in range(1, n):
        res.append(res[-1] + diff[i])
    return res
```

栈

单调栈

有重复数字也是一样的操作（等于也弹出），但是最后要进行一遍右答案的修正（因为有可能记录的是相等的值）（从右往左修正）

```
#求左右两边严格小于自身的最近的数 并且有重复值 的模板
#遍历
```

```python
for i in range(n):
    while st and arr[st[-1]]>arr[i]:
        cur=st.pop()
        ans[cur][0]=st[-1] if st else -1
        ans[cur][1]=i
    st.append(i)
#清算
while st:
    cur=st.pop()
    ans[cur][0]=st[-1] if st else -1
    ans[cur][1]=-1
#修正
#n-1一定是-1，所以不需要修正
for i in range(n-2,-1,-1):
    if ans[i][1]!=-1 and arr[ans[i][1]]==arr[i]:
        ans[i][1]=ans[ans[i][1]][1]
```

重复一定要特判，子数组一题重复的就要作为ans才可以不重不漏。有些时候中间的相等值答案可能不对，只要后续的相等值进来能把答案修正对就可以了（回忆最大矩形一题，相等也弹出）

其他用法：维持答案的一种可能性，比如求数组中的坡，维持栈中是递减的，遇到大的弹出，然后再从右往左更新答案。
比如字典序最小的规定字符的字符串，先用counter记录能不能删某个字符，再用单调栈去维护字典序最小

2.1.1匹配括号

```python
def par_checker(symbol_string):
    s = [] # Stack()
    balanced = True
    index = 0
    while index < len(symbol_string) and balanced:
        symbol = symbol_string[index]
        if symbol in "([{":
            s.append(symbol) # push(symbol)
        else:
            top = s.pop()
            if not matches(top, symbol):
                balanced = False
        index += 1
        #if balanced and s.is_empty():
        if balanced and not s:
            return True
        else:
            return False
def matches(open, close):
    opens = "([{"
    closes = ")]}"
    return opens.index(open) == closes.index(close)
print(par_checker('{{}}[]]'))
```

中序转后序Shunting Yard：

基本步骤：

1. 初始化运算符栈和输出栈为空。
2. 从左到右遍历中缀表达式的每个符号。
    - 如果是操作数（数字），则将其添加到输出栈。
    - 如果是左括号，则将其推入运算符栈。
    - 如果是运算符：

        - 如果运算符的优先级大于运算符栈顶的运算符，或者运算符栈顶是左括号，则将当前运算符推入运算符栈。
        - 否则，将运算符栈顶的运算符弹出并添加到输出栈中，直到满足上述条件（或者运算符栈为空）。
        - 将当前运算符推入运算符栈。
    - 如果是右括号，则将运算符栈顶的运算符弹出并添加到输出栈中，直到遇到左括号。将左括号弹出但不添加到输出栈中。
3. 如果还有剩余的运算符在运算符栈中，将它们依次弹出并添加到输出栈中。
4. 输出栈中的元素就是转换后的后缀表达式。

```
def turn(s):
    fuhao={'+':1,'-':1,'*':2,'/':2}
    stack=[]
    ans=[]
    num=''
    for i in s:
        if i in '0123456789.':
            num+=i
        else:
            if num:
                ans.append(num)
                num=''
            if i in '+-*/':
                while stack and stack[-1] in '+-*/' and fuhao[i]
<=fuhao[stack[-1]]:
                    ans.append(stack.pop())
                stack.append(i)
            elif i=='(':
                stack.append(i)
            elif i==')':
                while stack and stack[-1]!='(':
                    ans.append(stack.pop())
                stack.pop()
    if num:
        ans.append(num)
    while stack:
        ans.append(stack.pop())
    return ' '.join(str(i) for i in ans)
case=int(input())
for _ in range(case):
    s=input()
    print(turn(s))
```

链表：

快慢指针

```
slow,fast=head,head
while fast.next and fast.next.next:
    slow=slow.next
    fast=fast.next.next
```

```
    #slow此时是中偏左位置
    slow=slow.next
```

单链表反转

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def reverse_linked_list(head: ListNode) -> ListNode:
    prev = None
    curr = head
    while curr is not None:
        next_node = curr.next  # 暂存当前节点的下一个节点
        curr.next = prev       # 将当前节点的下一个节点指向前一个节点
        prev = curr            # 前一个节点变为当前节点
        curr = next_node       # 当前节点变更为原先的下一个节点
    return prev
```

双链表反转

```
def fan(head):
    pre,nt=None,None
    while head!=None:
        nt=head.next
        head.next=pre
        head.last=nt#last表示上一个
        pre=head
        head=nt
    return pre
```

链表判断环：

```
def detectCycle(head):
    if head is None or head.next is None or head.next.next is None:
        return None
    slow=head.next
    fast=head.next.next
    while slow!=fast:
        if fast.next is None or fast.next.next is None:
            return None
        slow=slow.next
        fast=fast.next.next
    fast=head
    while slow!=fast:
        slow=slow.next
        fast=fast.next
    return slow
```

合并两个排序链表

```
def merge_sorted_lists(l1, l2):
    dummy = Node(0)             # 哨兵节点
    tail = dummy
    while l1 and l2:
        if l1.data < l2.data:
            tail.next = l1
            l1 = l1.next
        else:
            tail.next = l2
            l2 = l2.next
        tail = tail.next
    if l1:                      # 如果还有没处理完的部分
        tail.next = l1
    else:
        tail.next = l2
    return dummy.next
```

树

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

class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
```

建树

```python
#类似邻接表
#第一行是一个整数n，表示二叉树的结点个数。二叉树结点编号从1到n，根结点为1，n <= 10 接下来有n行，依次对应二叉树的n个节点。 每行有两个整数，分别表示该节点的左儿子和右儿子的节点编号。如果第一个（第二个）数为-1则表示没有左（右）儿子
def build_tree(nodes):
    if not nodes:
        return None

    tree_nodes = [None] * (len(nodes) + 1)
    for i in range(1, len(nodes) + 1):
        tree_nodes[i] = TreeNode(i)

    for i, (left, right) in enumerate(nodes, start=1):
        if left != -1:
            tree_nodes[i].left = tree_nodes[left]
        if right != -1:
            tree_nodes[i].right = tree_nodes[right]

    return tree_nodes[1]

def main():
    n = int(input())
    nodes = []
    index = 1
    for _ in range(n):
        left, right = map(int, input().split())
        nodes.append((left, right))
#完全二叉树建树
n = int(input())
a = list(map(int, input().split()))
node = []
for i in range(n):
    node.append(Node(a[i]))
for i in range(n):
```

```python
    if 0 <= 2 * i + 1 < n: node[i].left = node[2 * i + 1]
    if 0 <= 2 * i + 2 < n: node[i].right = node[2 * i + 2]
```

二叉树的遍历

```python
def preorder_traversal(root): #前序
    if root:
        print(root.val)  # 访问根节点
        preorder_traversal(root.left)  # 递归遍历左子树
        preorder_traversal(root.right)  # 递归遍历右子树
```

```python
def inorder_traversal(root): #中序
    if root:
        inorder_traversal(root.left)  # 递归遍历左子树
        print(root.val)  # 访问根节点
        inorder_traversal(root.right)  # 递归遍历右子树
```

```python
def postorder_traversal(root): #后序
    if root:
        postorder_traversal(root.left)  # 递归遍历左子树
        postorder_traversal(root.right)  # 递归遍历右子树
        print(root.val)  # 访问根节点
```

邻接表的使用

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

Huffman编码树

```
import heapq
class node:
    def __init__(self, char,freq):
        self.char = char
        self.left=None
        self.right=None
        self.freq=freq
    #用于比较
    def __lt__(self, other):
        if self.freq==other.freq:
            return self.char<other.char
        return self.freq<other.freq
def build_huffman(d):
    q=[]
    for char,freq in d.items():
        heapq.heappush(q,node(char,freq))
    heapq.heapify(q)
    while len(q)>1:
        left=heapq.heappop(q)
        right=heapq.heappop(q)
        if left.char<right.char:
            c=left.char
        else:
            c=right.char
        nn=node(c,left.freq+right.freq)
        nn.left=left
        nn.right=right
        heapq.heappush(q,nn)
    return heapq.heappop(q)
def build_code(root):
    stack=[(root,'')]
    di={}
    dic={}
    while stack:
        x,y=stack.pop()
        if x.left:
            stack.append((x.left,y+'0'))
        if x.right:
            stack.append((x.right,y+'1'))
        if not x.left and not x.right:
            di[x.char]=y
            dic[y]=x.char
    return di,dic
n=int(input())
```

```python
d={}
for i in range(n):
    char,freq=input().split()
    freq=int(freq)
    d[char]=freq
root=build_huffman(d)
d_str,d_num=build_code(root)
while True:
    try:
        s=input()
        if s[0]=='0' or s[0]=='1':
            a=''
            for i in s:
                a+=i
                if a in d_num:
                    print(d_num[a],end='')
                    a=''
        else:
            for i in s:
                print(d_str[i],end='')
        print()
    except EOFError:
        break
```

```
前序中序转后序
def hx(qx,zx):
    if not qx:
        return ''
    root = qx[0]
    left_zx,right_zx = zx[0:zx.index(root)],zx[zx.index(root)+1:]
    left_qx,right_qx = qx[1:1 + len(left_zx)],qx[1 + len(left_zx):]
    left_hx,right_hx = hx(left_qx, left_zx),hx(right_qx, right_zx)
    return left_hx + right_hx + root
后序中序转前序
def qx(hx, zx):
    if not hx:
        return ''
    root = hx[-1]
    left_zx, right_zx = zx[:zx.index(root)], zx[zx.index(root)+1:]
    left_hx, right_hx = hx[:zx.index(root)], hx[zx.index(root):-1]
    left_qx = hx_zx_to_qx(left_hx, left_zx)
    right_qx = hx_zx_to_qx(right_hx, right_zx)
    return root + left_qx + right_qx
```

10. 前缀树（字典树）Trie

```
class TrieNode:
    def __init__(self):
        self.child={}
class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, nums):
        curnode = self.root
        for x in nums:
            if x not in curnode.child:
                curnode.child[x] = TrieNode()
            curnode=curnode.child[x]

    def search(self, num):
        curnode = self.root
        for x in num:
            if x not in curnode.child:
                return 0
            curnode = curnode.child[x]
        return 1
```

图

仍然需注意邻接表的使用。

并查集（重要！）

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

def union(x, y):
    parent[find(x)] = find(y)   #这样也只能把x这条路上的点给重置了，别的点可能还没归到y去，再
使用时还需要重新find

n, m = map(int, input().split())
parent = list(range(n + 1))

for _ in range(m):
    a, b = map(int, input().split())
    union(a, b)
#通过树形结构来表示集合，并通过父指针列表来实现这种结构
```

寻找联通块（结合邻接表和并查集）

```
def dfs(node, visited, adjacency_list):
    visited[node] = True
    for neighbor in adjacency_list[node]:
        if not visited[neighbor]:
            dfs(neighbor, visited, adjacency_list)

n, m = map(int, input().split())
adjacency_list = [[] for _ in range(n)]
for _ in range(m):
    u, v = map(int, input().split())
```

```python
    adjacency_list[u].append(v)
    adjacency_list[v].append(u)

visited = [False] * n
connected_components = 0
for i in range(n):
    if not visited[i]:
        dfs(i, visited, adjacency_list)
        connected_components += 1

print(connected_components)
```

拓扑排序

```python
from collections import deque, defaultdict

def topological_sort(graph):
    indegree = defaultdict(int)
    result = []
    queue = deque()

    # 计算每个顶点的入度
    for u in graph:
        for v in graph[u]:
            indegree[v] += 1

    # 将入度为 0 的顶点加入队列
    for u in graph:
        if indegree[u] == 0:
            queue.append(u)

    # 执行拓扑排序
    while queue:
        u = queue.popleft()
        result.append(u)

        for v in graph[u]:
            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)

    # 检查是否存在环
    if len(result) == len(graph):
        return result
    else:
        return None
```

三色标记法（判断环）：

如果在递归过程中，发现下一个节点在递归栈中（正在访问中），则找到了环。

对于每个节点 x，都定义三种颜色值（状态值）：

0：节点 x 尚未被访问到。
1：节点 x 正在访问中，$dfs(x)$ 尚未结束。
2：节点 x 已经完全访问完毕，$dfs(x)$ 已返回。

```python
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        g = [[] for _ in range(numCourses)]
        for a, b in prerequisites:
            g[b].append(a)

        colors = [0] * numCourses
        # 返回 True 表示找到了环
        def dfs(x: int) -> bool:
            colors[x] = 1  # x 正在访问中
            for y in g[x]:
                if colors[y] == 1 or colors[y] == 0 and dfs(y):
                    return True  # 找到了环
            colors[x] = 2  # x 完全访问完毕
            return False  # 没有找到环

        for i, c in enumerate(colors):
            if c == 0 and dfs(i):
                return False  # 有环
        return True  # 没有环
```

无向图判环

```python
def has_cycle_undirected(graph):
    visited = set()

    def dfs(node, parent):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                if dfs(neighbor, node):
                    return True
            elif neighbor != parent:
                return True
        return False

    for node in graph:
        if node not in visited:
            if dfs(node, -1):
                return True
    return False
```

有向图判环

```python
def has_cycle_directed(graph): #dfs + recursion stack
    visited = set()
    rec_stack = set()
    def dfs(node):
        visited.add(node)
        rec_stack.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                if dfs(neighbor):
                    return True
```

```
            elif neighbor in rec_stack:
                return True
        rec_stack.remove(node)
        return False

    for node in graph:
        if node not in visited:
            if dfs(node):
                return True
    return False
```

```
from collections import deque, defaultdict

def has_cycle_topo_sort(graph): #拓扑 Kahn算法
    indegree = defaultdict(int)
    for u in graph:
        for v in graph[u]:
            indegree[v] += 1

    queue = deque([node for node in graph if indegree[node] == 0])
    visited_count = 0

    while queue:
        node = queue.popleft()
        visited_count += 1
        for neighbor in graph[node]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)

    return visited_count != len(graph)
```

DFS

由于DFS的特性，path可以不参与变量的传递，这样只用一个全局变量path修改就行了，找到了可能的答案就copy到ans（字符串可以用两个变量，copy的时候就不会出事）。

```
wansdroff 接下来访问的点的能访问数是最少的（回忆骑士周游的degree优化
def degree(x,y,board):
    global n,di
    cnt=0
    for dx,dy in di:
        nx,ny=x+dx,y+dy
        if 0<=nx<n and 0<=ny<n and board[nx][ny]==-1:
            cnt+=1
    return cnt
def dfs(x,y,cnt):
    global di,board,n
    if cnt==n*n:
        return True
    next_move=[]
    for dx,dy in di:
        nx,ny=x+dx,y+dy
```

```
        if 0<=nx<n and 0<=ny<n and board[nx][ny]==-1:
            next_move.append((degree(nx,ny,board),nx,ny))
    next_move.sort()
    for _,nx,ny in next_move:
        board[nx][ny]=cnt
        if dfs(nx,ny,cnt+1):
            return True
        board[nx][ny]=-1
    return False

最大连通域面积
s = 0
def dfs(a,b):
    directions = [[1,1],[1,0],[1,-1],[0,1],[0,-1],[-1,1],[-1,0],[-1,-1]]
    global s
    if a < 0 or b < 0 or a >= len(matrix) or b >= len(matrix[0]) or matrix[a][b]
== ".":
        return
    matrix[a][b] = "."
    s += 1
    for i in range(len(directions)):
        dfs(a + directions[i][0],b + directions[i][1])

马走日（回溯）
ans = 0
def dfs(x,y,cnt):
    directions = [[1, 2], [2, 1], [1, -2], [2, -1], [-1, 2], [-2, 1], [-1, -2],
[-2, -1]]
    global ans
    if cnt == n * m:
        ans += 1
        return
    for i in range(len(directions)):
        nx = x + directions[i][0]
        ny = y + directions[i][1]
        if 0 <= nx < n and 0 <= ny < m:
            if matrix[nx][ny] != 1:
                matrix[nx][ny] = 1
                dfs(nx, ny, cnt + 1)
                matrix[nx][ny] = 0
```

bfs模板

```
from collections import deque

MAX_DIRECTIONS = 4
dx = [0, 0, 1, -1]; dy = [1, -1, 0, 0]

def is_valid_move(x, y):
    return 0 <= x < n and 0 <= y < m and maze[x][y] == 0 and not in_queue[x][y]

def bfs(start_x, start_y):
    queue = deque()
```

```
    queue.append((start_x, start_y))
    in_queue[start_x][start_y] = True
    while queue:
        x, y = queue.popleft()
        if x == n - 1 and y == m - 1:
            return
        for i in range(MAX_DIRECTIONS):
            next_x = x + dx[i]; next_y = y + dy[i]
            if is_valid_move(next_x, next_y):
                prev[next_x][next_y] = (x, y)
                in_queue[next_x][next_y] = True
                queue.append((next_x, next_y))

def print_path(pos):
    prev_position = prev[pos[0]][pos[1]]
    if prev_position == (-1, -1):
        print(pos[0] + 1, pos[1] + 1)
        return
    print_path(prev_position)
    print(pos[0] + 1, pos[1] + 1)

n, m = map(int, input().split())
maze = [list(map(int, input().split())) for _ in range(n)]
in_queue = [[False] * m for _ in range(n)]
prev = [[(-1, -1)] * m for _ in range(n)]
bfs(0, 0)
print_path((n - 1, m - 1))

///////////////////////////////////
from collections import deque
def bfs(x,y):
    directions = [[0, 1], [1, 0], [-1, 0], [0, -1]]
    q = deque([(0,(x,y))])
    in_queue = {(x,y)}
    while q:
        step,(x,y) = q.popleft()
        if matrix[x][y] == 1:
            return step
        for i in range(len(directions)):
            nx = x + directions[i][0]
            ny = y + directions[i][1]
            if matrix[nx][ny] != 2 and (nx,ny) not in in_queue:
                in_queue.add((nx,ny))
                q.append((step + 1,(nx,ny)))
    return "NO"
m,n = map(int,input().split())
matrix = [[2] * (n + 2) for i in range(m + 2)]
for _ in range(1,m + 1):
    matrix[_][1:-1] = map(int,input().split())
print(bfs(1,1))
```

dijkstra

过程：

1.distance[i]表示从源点到i点的最短距离，visited[i]表示i节点是否从小根堆弹出过

2.准备好小根堆，小根堆存放记录：（源点到x的距离，x点），小根堆根据距离组织

3.令distance[源点]=0，(0，源点）放入小根堆

4.从小根堆弹出（源点到u的距离，u点）

   a.如果visited[u]==true，不做处理

   b.如果visited[u]==false. visited[u]=true

      然后考察u的每一条边，去往v，边权为w

      如果visited[v]==false and distance[u]+w<distance[v]，令distance[v]=distance[u]+w把(distance[u]+w,v)加入小根堆

```
distance=[float('inf')]*n
distance[s]=0
visied=[False]*n
q=[]
heapq.heappush((0,s))
while q:
    u=heappop(q)
    if visited[u]:
        continue
    visited[u]=True
    for v,w in e[u]:
        if visited[v] and distance[u]+w<distance[v]:
            diatance[v]=distance[u]+w
            heapq.heappush(q,(distance[u]+w,v))
```

```
import heapq

def dijkstra(N, G, start):
    INF = float('inf')
    dist = [INF] * (N + 1)  # 存储源点到各个节点的最短距离
    dist[start] = 0  # 源点到自身的距离为0
    pq = [(0, start)]  # 使用优先队列，存储节点的最短距离
    while pq:
        d, node = heapq.heappop(pq)  # 弹出当前最短距离的节点
        if d > dist[node]:  # 如果该节点已经被更新过了，则跳过
            continue
        for neighbor, weight in G[node]:  # 遍历当前节点的所有邻居节点
            new_dist = dist[node] + weight  # 计算经当前节点到达邻居节点的距离
            if new_dist < dist[neighbor]:  # 如果新距离小于已知最短距离，则更新最短距离
                dist[neighbor] = new_dist
                heapq.heappush(pq, (new_dist, neighbor))  # 将邻居节点加入优先队列
    return dist
```

上学期写过的模板题（孤岛最短距离）

```
孤岛最短距离
import heapq
def dijkstra(a,b):
    directions = [[0,1],[1,0],[-1,0],[0,-1]]
    q = []
```

```
    visited = [[False] * len(matrix[0]) for _ in range(n)]
    heapq.heappush(q,(0,a,b))
    while q:
        step,x,y = heapq.heappop(q)
        if visited[x][y]:
            continue
        visited[x][y] = True
        if matrix[x][y] == 1 and step > 0:
            return step
        for i in range(len(directions)):
            nx = x + directions[i][0]
            ny = y + directions[i][1]
            if 0 <= nx < n and 0 <= ny < len(matrix[0]) and not visited[nx][ny]:
                heapq.heappush(q,(step + 1 - matrix[nx][ny],nx,ny))
n = int(input())
matrix = [list(map(int,input())) for _ in range(n)]
for i in range(n):
    for j in range(len(matrix[0])):
        if matrix[i][j] == 1:
            a,b = i,j
print(dijkstra(a,b))
```

Bellman-Ford

松弛操作:

> 假设源点为A，从A到任意点F的最短距离为distance[F]
> 假设从点P出发某条边，去往点S，边权为W
> 如果发现，distance[P] + W<distance[S]，也就是通过该边可以让distance[S]变小

那么就说P出发的这条边对点S进行了松弛操作

Bellman-Ford过程:

> 1.每一轮考察每条边，每条边都尝试进行松弛操作，那么若干点的distance会变小
> 2.当某一轮发现不再有松弛操作出现时，停止

```
def bellman_ford(graph, V, source):
    # 初始化距离
    dist = [float('inf')] * V
    dist[source] = 0

    # 松弛 V-1 次
    for _ in range(V - 1):
        for u, v, w in graph:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    # 检测负权环
    for u, v, w in graph:
        if dist[u] != float('inf') and dist[u] + w < dist[v]:
            print("图中存在负权环")
            return None

    return dist
```

Floyd-Warshell

用**邻接矩阵**储存图，求任意两点之间最短距离

```
def floyd_warshall(graph):
    n = len(graph)
    dist = [[float('inf')] * n for _ in range(n)]

    for i in range(n):
        for j in range(n):
            if i == j:
                dist[i][j] = 0
            elif j in graph[i]:
                dist[i][j] = graph[i][j]

    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist
```

kruskal算法求解最小生成树（结合并查集）

greedy+disjointset

1.把所有的边按照权值sort，从权值小的开始考虑
2.如果当前边的两个节点不在一个集合：选择这个边
3.如果在一个集合：不选

```
class DisjointSet:
    def __init__(self, num_vertices):
        self.parent = list(range(num_vertices))
        self.rank = [0] * num_vertices

    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):
        root_x = self.find(x)
        root_y = self.find(y)

        if root_x != root_y:
            if self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            elif self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            else:
                self.parent[root_x] = root_y
                self.rank[root_y] += 1


def kruskal(graph):
```

```
    num_vertices = len(graph)
    edges = []

    # 构建边集
    for i in range(num_vertices):
        for j in range(i + 1, num_vertices):
            if graph[i][j] != 0:
                edges.append((i, j, graph[i][j]))

    # 按照权重排序
    edges.sort(key=lambda x: x[2])

    # 初始化并查集
    disjoint_set = DisjointSet(num_vertices)

    # 构建最小生成树的边集
    minimum_spanning_tree = []

    for edge in edges:
        u, v, weight = edge
        if disjoint_set.find(u) != disjoint_set.find(v):
            disjoint_set.union(u, v)
            minimum_spanning_tree.append((u, v, weight))

    return minimum_spanning_tree
```

Prim

1.解锁的点的集合叫set，解锁的边的集合叫heap。set&heap为空。
2.从任意点开始，开始点加到set，开始点的所有边加入到heap
3.从heap弹出权值最小的边e，查看边e去往的点x：
    如果x in set，e舍弃
    如果不在，e属于最小生成树，x加入set

```
import heapq

def prim(graph, n):
    visited = [False] * n
    min_heap = [(0, 0)]  # (weight, vertex)
    min_spanning_tree_cost = 0

    while min_heap:
        weight, vertex = heapq.heappop(min_heap)

        if visited[vertex]:
            continue

        visited[vertex] = True
        min_spanning_tree_cost += weight

        for neighbor, neighbor_weight in graph[vertex]:
            if not visited[neighbor]:
                heapq.heappush(min_heap, (neighbor_weight, neighbor))

    return min_spanning_tree_cost if all(visited) else -1
```