cheat sheet

图

无向图

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

1.判断是否连通

```
def is_connected(graph,n):
    visited = [False] * n
    stack = [0]
    visited[0] = True

    while stack:
        node = stack.pop()
        for neighbor in graph[node]:
            if not visited[neighbor]:
                stack.append(neighbor)
                visited[neighbor] = True

    return all[visited]
```

2.判断是否有回路

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

    visited = [False] * n
    for node in range(n):
        if not visited[node]:
            if dfs(node,visited,-1):
```

```
                return True
    return False
```

dfs

```
def dfsTravel(G,op): #G是邻接表
    def dfs(v):
        visited[v] = True
        op(v)
        for u in G[v]:
            if not visited[u]:
                dfs(u)
    n = len(G)  # 顶点数目
    visited = [False for i in range(n)]
    for i in range(n):  # 顶点编号0到n-1
        if not visited[i]:
            dfs(i)

n,m = map(int,input().split())
G = [[] for i in range(n)]
for i in range(m):
    s,e = map(int,input().split())
    G[s].append(e)
    G[e].append(s)
dfsTravel(G,lambda x:print(x,end = " "))
```

最小生成树

Prim

```
from heapq import heappop, heappush

while True:
    try:
        n = int(input())
    except:
        break
    matrix = []
    for i in range(n):
        matrix.append(list(map(int, input().split())))
    distance = [100000 for i in range(n)] #初始化距离
    v = set() #初始化已访问节点的集合v
    cnt = 0 #初始化总长度cnt
    q = [] #初始化优先队列q

    distance[0] = 0
    heappush(q, (distance[0], 0)) #将起始节点（距离为0，节点编号为0）推入优先队列
    while q:
        x, y = heappop(q) #从优先队列中弹出距离最小的节点y及其距离x
```

```
        if y in v: #如果节点y已访问，跳过本次循环
            continue
        v.add(y) #将节点y标记为已访问
        cnt += distance[y] #将节点y的距离加到总长度cnt上
        for i in range(n): #遍历节点y的所有邻接节点i
            if distance[i] > matrix[y][i]: #如果节点i的当前距离大于通过节点y到达
i的距离
                distance[i] = matrix[y][i] #则更新距离
                heappush(q, (distance[i], i)) #将更新后的距离和节点编号推入优先
队列
    print(cnt)
```

agri-net

```
import heapq

def prim(graph):
    n = len(graph)
    visited = [False] * n
    min_heap = [(0, 0)]  # (distance, node)
    min_fiber = 0

    while min_heap:
        dist, node = heapq.heappop(min_heap)
        if visited[node]:
            continue
        visited[node] = True
        min_fiber += dist
        for neighbor, weight in enumerate(graph[node]):
            if not visited[neighbor] and weight != 0:
                heapq.heappush(min_heap, (weight, neighbor))

    return min_fiber

def main():
    while True:
        try:
            n = int(input())
            graph = []
            for _ in range(n):
                graph.append(list(map(int, input().split())))
            min_fiber = prim(graph)
            print(min_fiber)
        except EOFError:
            break

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

Kruskal

以下是Kruskal算法的基本步骤：

1. 将图中的所有边按照权重从小到大进行排序。（队列）
2. 初始化一个空的边集，用于存储最小生成树的边。
3. 重复以下步骤，直到边集中的边数等于顶点数减一或者所有边都已经考虑完毕：（用并查集判断新加入的边是否合法）
    - 选择排序后的边集中权重最小的边。
    - 如果选择的边不会导致形成环路（即加入该边后，两个顶点不在同一个连通分量中），则将该边加入最小生成树的边集中。
4. 返回最小生成树的边集作为结果。

Kruskal算法的核心思想是通过不断选择权重最小的边，并判断是否会形成环路来构建最小生成树。算法开始时，每个顶点都是一个独立的连通分量，随着边的不断加入，不同的连通分量逐渐合并为一个连通分量，直到最终形成最小生成树。

实现Kruskal算法时，一种常用的数据结构是并查集（Disjoint Set）。并查集可以高效地判断两个顶点是否在同一个连通分量中，并将不同的连通分量合并。

```
##class DisjointSet:
''''''

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
```

有向图

1.拓扑结构（判断是否有环）

1.1、无向图 使用拓扑排序可以判断一个无向图中是否存在环，具体步骤如下：

求出图中所有结点的度。 将所有度 <= 1 的结点入队。（独立结点的度为 0） 当队列不空时，弹出队首元素，把与队首元素相邻节点的度减一。如果相邻节点的度变为一，则将相邻结点入队。 循环结束时判断已经访问的结点数是否等于 n。等于 n 说明全部结点都被访问过，无环；反之，则有环。 1.2、有向图 使用拓扑排序判断无向图和有向图中是否存在环的区别在于：

在判断无向图中是否存在环时，是将所有度 <= 1 的结点入队； 在判断有向图中是否存在环时，是将所有入度 = 0 的结点入队。

```
from collections import deque

def detect_cycle_in_directed_graph():
    T = int(input())
    for _ in range(T):
        N, M = map(int, input().split())
        graph = [[] for _ in range(N + 1)]
        in_degree = [0] * (N + 1)

        for _ in range(M):
            u, v = map(int, input().split())
            graph[u].append(v)
            in_degree[v] += 1

        def has_cycle(graph, in_degree, n):
            queue = deque()
            for i in range(1, n + 1):
                if in_degree[i] == 0:
                    queue.append(i)

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

            return visited_count != n

        if has_cycle(graph, in_degree, N):
            print('Yes')
        else:
            print('No')

detect_cycle_in_directed_graph()
```

最小奖金额

```
import collections
n,m = map(int,input().split())
G = [[] for i in range(n)]
award = [0 for i in range(n)]
inDegree = [0 for i in range(n)]

for i in range(m):
    a,b = map(int,input().split())
    G[b].append(a)
    inDegree[a] += 1
q = collections.deque()
for i in range(n):
    if inDegree[i] == 0:
        q.append(i)
        award[i] = 100
while len(q) > 0:
    u = q.popleft()
    for v in G[u]:
        inDegree[v] -= 1
        award[v] = max(award[v],award[u] + 1)
        if inDegree[v] == 0:
            q.append(v)
total = sum(award)
print(total)
```

2.sorting it all out 用图的邻接表、拓扑结构

```
from collections import deque

def topo_sort(graph):
    # 初始化入度字典
    in_degree = {u: 0 for u in graph}
    for u in graph:
        for v in graph[u]:
            in_degree[v] += 1

    # 将所有入度为0的节点入队
    q = deque([u for u in in_degree if in_degree[u] == 0])
    topo_order = []
    flag = True  # 用于检测拓扑排序是否唯一确定

    while q:
        if len(q) > 1:
            flag = False  # 如果同时有多个入度为0的节点，说明拓扑排序不唯一确定
        u = q.popleft()
        topo_order.append(u)
        for v in graph[u]:
            in_degree[v] -= 1
```

```
            if in_degree[v] == 0:
                q.append(v)

    # 如果拓扑排序没有包含所有节点，说明存在环
    if len(topo_order) != len(graph):
        return 0
    return topo_order if flag else None

while True:
    n, m = map(int, input().split())
    if n == 0:
        break

    graph = {chr(x + 65): [] for x in range(n)}
    edges = [tuple(input().split('<')) for _ in range(m)]

    for i in range(m):
        a, b = edges[i]
        graph[a].append(b)

        t = topo_sort(graph)
        if t:
            s = ''.join(t)
            print(f"Sorted sequence determined after {i + 1} relations:
{s}.")
            break
        elif t == 0:
            print(f"Inconsistency found after {i + 1} relations.")
            break
    else:
        print("Sorted sequence cannot be determined.")
```

二分查找

bisect

```
import bisect

# 示例列表
a = [1, 2, 4, 4, 5, 6]

#查找插入位置
x = 4
left = bisect.bisect_left(a, x)
right = bisect.bisect_right(a, x)

print(f"Left insertion point for {x} is at index: {left}")
print(f"Right insertion point for {x} is at index: {right}")

#插入操作
new_element = 3
```

```
bisect.insort_left(a, new_element)
##如果说 bisect.bisect_left() 是为了在序列a中查找元素x的插入点（左侧），那么
bisect.insort_left()就是在找到插入点的基础上，真正地将元素x插入序列a，从而改变序列a同
时保持元素顺序。
print(f"List after inserting {new_element} using insort_left: {a}")

new_element = 4
bisect.insort_right(a, new_element)
print(f"List after inserting {new_element} using insort_right: {a}")
```

```
import bisect

def insert_sorted(array, values):
    for value in values:
        bisect.insort(array, value)
    return array

# 示例列表
sorted_list = [1, 3, 4, 7]
new_values = [5, 2, 6]

# 插入新值并保持有序
result = insert_sorted(sorted_list, new_values)
print(f"Sorted list after inserting new values: {result}")
```

最长上升子序列

```
from bisect import bisect

n = int(input())
scores = list(map(int,input().split()))
cur_list = []

for i in range(n):
    cur = scores[i]

    if cur_list:
        if cur >= cur_list[-1]:
            cur_list[-1] = cur
        else:
            num = bisect(cur_list,cur)
            if num == 0:
                cur_list.insert(0,cur)
            else:
                cur_list[num - 1] = cur
    else:
        cur_list.append(cur)
```

```
print(len(cur_list))
```

```
from bisect import bisect_left

n = int(input())
nums = list(map(int, input().split()))
temp = []

for i in range(n):
    cur = nums[i]
    if not temp or cur > temp[-1]:
        temp.append(cur)
    else:
        idx = bisect_left(temp, cur)
        temp[idx] = cur

print(len(temp))
```

less or equal

http://cs101.openjudge.cn/2024sp_routine/27932/

```
def count_less_or_equal(nums,x):
    count = 0
    for i in nums:
        if i <= x:
            count += 1
    return count

def find(k,nums):
    a = sorted(nums)
    left = 1
    right = 10**9
    result = -1

    while left <= right:
        mid = (left+right)//2
        if count_less_or_equal(a,mid) == k:
            result = mid
            right = mid - 1
        elif count_less_or_equal(a,mid) < k:
            left = mid + 1
        else:
            right = mid - 1

    return result

n, k = map(int, input().split())
```

```
nums = list(map(int,input().split()))

result = find(k,nums)
print(result)
```

树

二叉树

1.每个节点有0或2个子节点。 2.叶子节点（空节点）的数量应当满足二叉树结构的规则。 -节点非空 = +1（+2-1） -节点空 = -1

建立二叉树

```
class TreeNode:
    def __init__(self,val = 0,left = None,right = None):
        self.val = val
        self.left = left
        self.right = right
```

二叉树深度

```
class TreeNode:
    def __init__(self,val = 0,left = None,right = None):
        self.val = val
        self.left = left
        self.right = right

def maxDepth(root):
    if not root:
        return 0
    return 1 + max(maxDepth(root.left), maxDepth(root.right))

def BuildTree(node_list):
    if not node_list:
        return None

    nodes = {i: TreeNode(i) for i in range(1, len(node_list) + 1)}
    for i,(left,right) in enumerate(node_list,1):
        if left != -1:
            nodes[i].left = nodes[left]
        if right != -1:
            nodes[i].right = nodes[right]
    return nodes[1]

if __name__ == "__main__":
    n = int(input())
    node_list = []
    for _ in range(n):
```

```
        left, right = map(int, input().split())
        node_list.append((left, right))

root = BuildTree(node_list)
depth = maxDepth(root)

print(depth)
```

求二叉树的高度和叶子数目

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def height(root):
    if root is None:
        return -1
    left = height(root.left)
    right = height(root.right)

    return max(left, right) + 1

def leaf_counts(root):
    if root is None:
        return 0
    if root.left is None and root.right is None:
        return 1
    return leaf_counts(root.left) + leaf_counts(root.right)

if __name__ == "__main__":
    n = int(input())
    nodes = [TreeNode() for _ in range(n)]
    has_parent = [False] * n  # 用来标记节点是否有父节点

    for i in range(n):
        left_index, right_index = map(int, input().split())
        if left_index != -1:
            nodes[i].left = nodes[left_index]
            has_parent[left_index] = True
        if right_index != -1:
            nodes[i].right = nodes[right_index]
            has_parent[right_index] = True

        # 寻找根节点，也就是没有父节点的节点
        root_index = has_parent.index(False)
        root = nodes[root_index]

    # 计算高度和叶子节点数
    height = height(root)
    leaves = leaf_counts(root)
```

```
    print(height,leaves)
```

二叉搜索树的遍历 前序->后序

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


def buildTree(preorder):
    if len(preorder) == 0:
        return None

    node = Node(preorder[0])

    idx = len(preorder)
    for i in range(1, len(preorder)):
        if preorder[i] > preorder[0]:
            idx = i
            break
    node.left = buildTree(preorder[1:idx])
    node.right = buildTree(preorder[idx:])

    return node


def postorder(node):
    if node is None:
        return []
    output = []
    output.extend(postorder(node.left))
    output.extend(postorder(node.right))
    output.append(str(node.val))

    return output


n = int(input())
preorder = list(map(int, input().split()))
print(' '.join(postorder(buildTree(preorder))))
```

前序、中序找后序

```
def buildTree(preorder,inorder):

    if not preorder:
```

```
        return ''

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

    left = buildTree(preorder[1:1+root_index],inorder[:root_index])
    right = buildTree(preorder[1+root_index:],inorder[root_index+1:])

    return left + right + root

while True:
    try:
        preorder,inorder = input().split()
        postorder = buildTree(preorder,inorder)
        print(postorder)
    except EOFError:
        break
```

中序、后序找前序

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

def buildTree(inorder,postorder):
    if not inorder or not postorder:
        return None

    #后序遍历的最后一个元素是当前的根节点
    root_val = postorder.pop()
    root = TreeNode(root_val)

    #在中序遍历中找到根节点的位置
    root_index = inorder.index(root_val)

    root.right = buildTree(inorder[root_index +1:],postorder)
    root.left = buildTree(inorder[:root_index],postorder)

    return root

def preorder(root):
    result = []
    if root:
        result.append(root.val)
        result.extend(preorder(root.left))
        result.extend(preorder(root.right))
    return result

inorder = input().strip()
postorder = input().strip()
```

```
root = buildTree(list(inorder), list(postorder))
print(''.join(preorder(root)))
```

后序—>建树->层次遍历

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

def build_tree(postfix):
    stack = []
    for char in postfix:
        node = TreeNode(char)
        if char.isupper():
            node.right = stack.pop()
            node.left = stack.pop()
        stack.append(node)
    return stack[0]

def level_order_traversal(root):
    queue = [root]
    traversal = []
    while queue:
        node = queue.pop(0)
        traversal.append(node.value)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
    return traversal

n = int(input().strip())
for _ in range(n):
    postfix = input().strip()
    root = build_tree(postfix)
    queue_expression = level_order_traversal(root)[::-1]
    print(''.join(queue_expression))
```

27637: 括号嵌套二叉树

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

```
def parse_tree(s):
    if s == '*':
        return None
    if '(' not in s:
        return TreeNode(s)

    # Find the root value and the subtrees
    root_value = s[0]
    subtrees = s[2:-1]  # Remove the root and the outer parentheses

    # Use a stack to find the comma that separates the left and right
subtrees
    stack = []
    comma_index = None
    for i, char in enumerate(subtrees):
        if char == '(':
            stack.append(char)
        elif char == ')':
            stack.pop()
        elif char == ',' and not stack:
            comma_index = i
            break

    left_subtree = subtrees[:comma_index] if comma_index is not None else
subtrees
    right_subtree = subtrees[comma_index + 1:] if comma_index is not None
else None

    # Parse the subtrees
    root = TreeNode(root_value)
    root.left = parse_tree(left_subtree)
    root.right = parse_tree(right_subtree) if right_subtree else None
    return root


# Define the traversal functions
def preorder_traversal(root):
    if root is None:
        return ""
    return root.value + preorder_traversal(root.left) +
preorder_traversal(root.right)


def inorder_traversal(root):
    if root is None:
        return ""
    return inorder_traversal(root.left) + root.value +
inorder_traversal(root.right)


# Input reading and processing
n = int(input().strip())
for _ in range(n):
    tree_string = input().strip()
```

```
    tree = parse_tree(tree_string)
    preorder = preorder_traversal(tree)
    inorder = inorder_traversal(tree)
    print(preorder)
    print(inorder)
```

FBI树

```
def build_FBI_tree(N, s):
    if len(s) == 1:
        if s == '0':
            return 'B'
        else:
            return 'I'
    else:
        mid = len(s) // 2
        left = s[:mid]
        right = s[mid:]
        root_type = 'B' if '1' not in s else 'F' if '0' in s else 'I'
        left_tree = build_FBI_tree(N - 1, left)
        right_tree = build_FBI_tree(N - 1, right)
        return left_tree + right_tree + root_type

N = int(input())
binary_string = input()
post_order_traversal = build_FBI_tree(N, binary_string)
print(post_order_traversal)
```

遍历树

描述 请你对输入的树做遍历。遍历的规则是：遍历到每个节点时，按照该节点和所有子节点的值从小到大进行遍历，例如：

```
    7
/   |   \
```

10 3 6 对于这个树，你应该先遍历值为3的子节点，然后是值为6的子节点，然后是父节点7，最后是值为10的子节点。

本题中每个节点的值为互不相同的正整数，最大不超过9999999。

输入 第一行：节点个数n (n<500) 接下来的n行：第一个数是此节点的值，之后的数分别表示它的所有子节点的值。每个数之间用空格隔开。如果没有子节点，该行便只有一个数。 **输出** 输出遍历结果，一行一个节点的值。

```
class TreeNode:
    def __init__(self, value):
```

```
        self.value = value
        self.children = []


def traverse_print(root, nodes):
    if root.children == []:
        print(root.value)
        return
    pac = {root.value: root}
    for child in root.children:
        pac[child] = nodes[child]
    for value in sorted(pac.keys()):
        if value in root.children:
            traverse_print(pac[value], nodes)
        else:
            print(root.value)


n = int(input())
nodes = {}
children_list = []
for i in range(n):
    info = list(map(int, input().split()))
    nodes[info[0]] = TreeNode(info[0])
    for child_value in info[1:]:
        nodes[info[0]].children.append(child_value)
        children_list.append(child_value)
root = nodes[[value for value in nodes.keys() if value not in
children_list][0]]
traverse_print(root, nodes)
```

扩展二叉树

前序->中序、后序

```
def build_tree(preorder):
    if not preorder or preorder[0] == '.':
        return None, preorder[1:]
    root = preorder[0]
    left, preorder = build_tree(preorder[1:])
    right, preorder = build_tree(preorder)
    return (root, left, right), preorder

def inorder(tree):
    if tree is None:
        return ''
    root, left, right = tree
    return inorder(left) + root + inorder(right)

def postorder(tree):
    if tree is None:
        return ''
```

```
    root, left, right = tree
    return postorder(left) + postorder(right) + root

preorder = input().strip()
tree, _ = build_tree(preorder)

print(inorder(tree))
print(postorder(tree))
```

二叉搜索树 BST

前序->后序

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

def bst(preorder):
    if not preorder:
        return None

    root_val = preorder[0]
    root = TreeNode(root_val)

    split_index = 1
    while split_index < len(preorder) and preorder[split_index] <=
root_val:
        split_index += 1

    root.left = bst(preorder[1:split_index])
    root.right = bst(preorder[split_index:])

    return root

def postorder(root):
    if root is None:
        return []
    return postorder(root.left) + postorder(root.right) + [root.val]

n = int(input())
preorder = list(map(int, input().split()))

root = bst(preorder)
result = postorder(root)

print(" ".join(map(str, result)))
```

二叉搜索树的层次遍历

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

def buildTree(nums):
    if not nums:
        return None

    root = TreeNode(nums[0])
    for num in nums[1:]:
        insert(root, num)

    return root

def insert(root, val):
    if val < root.val:
        if root.left is None:
            root.left = TreeNode(val)
        else:
            insert(root.left, val)
    elif val > root.val:
        if root.right is None:
            root.right = TreeNode(val)
        else:
            insert(root.right, val)

def levelOrderTraversal(root):
    if root is None:
        return []

    result = []
    queue = [root]

    while queue:
        node = queue.pop(0)
        result.append(node.val)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)

    return result

nums = list(map(int, input().split()))
root = buildTree(nums)
result = levelOrderTraversal(root)
print(' '.join(map(str, result)))
```

树的转换

左儿子，右兄弟

```
def tree_heights(s):
    old_height = 0
    max_old = 0
    new_height = 0
    max_new = 0
    stack = []
    for c in s:
        if c == 'd':
            old_height += 1
            max_old = max(max_old, old_height)

            new_height += 1
            stack.append(new_height)
            max_new = max(max_new, new_height)
        else:
            old_height -= 1

            new_height = stack[-1]
            stack.pop()
    return f"{max_old} => {max_new}"

s = input().strip()
print(tree_heights(s))
```

其他

欧拉筛

```
def euler_sieve(n):
    is_prime = [True] * (n + 1)
    primes = []

    for i in range(2, n + 1):
        if is_prime[i]:
            primes.append(i)
            for j in range(i * i, n + 1, i):
                is_prime[j] = False

    return primes
```

判断素数

```
import math

def is_prime(x):
    if x < 2:
```

```
        return False
    for i in range(2, int(math.sqrt(x)) + 1):
        if x % i == 0:
            return False
    return True
```

matrix

```
matrix = [['' for _ in range(cols)] for _ in range(rows)] #col列, row行
```

```
# 从矩阵中提取原始信息
original = ''
for col in range(cols):
    for row in range(rows):
        original += matrix[row][col]
```

约瑟夫斯问题

```
from collections import deque
n,k = map(int,input().split())
queue = deque(x for x in range(1,n+1))
res = []

while len(queue) >= 2:
    for _ in range(k-1):
        a = queue.popleft()
        queue.append(a)
    b = queue.popleft()
    res.append(b)

print(*res) #被淘汰的顺序
```

归并排序

```
def merge_sort(nums):
    if len(nums) <= 1:
        return nums,0

    middle = len(nums)//2
    left,inv_left = merge_sort(nums[:middle])
    right,inv_right = merge_sort(nums[middle:])

    merged,inv_merge = merge(left,right)

    return merged, inv_left + inv_right + inv_merge
```

```
def merge(left,right):
    merged = []
    inv_count = 0
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
            inv_count += len(left) - i

    merged += left[i:]
    merged += right[j:]

    return merged, inv_count

while True:
    n = int(input())
    if n == 0:
        break

    nums = []
    for _ in range(n):
        nums.append(int(input()))

    _, inversions = merge_sort(nums)
    print(inversions)
```

matrix/dfs

最大连通域面积

```
def max_connected_area(grid,n,m,i,j):
    if i < 0 or i >= n or j <0 or j >= m or grid[i][j] != 'W':
        return 0

    grid[i][j] == '.'

    result = 1
    for x in range(-1, 2):
        for y in range(-1, 2):
            result += max_connected_area(grid, n, m, i + x, j + y)

    return result

def largest_connected_area(T,data):
    for _ in range(T):
```

```
        n,m = data[_][0]
        grid = data[_][1]

        max_area = 0
        for i in range(n):
            for j in range(m):
                if grid[i][j] == 'W':
                    area = max_connected_area(grid, n, m, i, j)
                    max_area = max(max_area, area)
    print(max_area)

if __name__ == "__main__":
    T = int(input())
    data = []

    for _ in range(T):
        n, m = map(int, input().split())
        grid = [list(input()) for _ in range(n)]
        data.append(((n, m), grid))

    largest_connected_area(T, data)
```

dfs

适合搜索全部的解

骑士周游

描述 在一个国际象棋棋盘上，一个棋子“马”（骑士），按照“马走日”的规则，从一个格子出发，要走遍所有棋盘格恰好一次。把一个这样的走棋序列称为一次“周游”。在 8 × 8 的国际象棋棋盘上，合格的“周游”数量有1.305×10^35这么多，走棋过程中失败的周游就更多了。

采用图搜索算法，是解决骑士周游问题最容易理解和编程的方案之一，解决方案分为两步： 首先用图表示骑士在棋盘上的合理走法；采用图搜索算法搜寻一个长度为（行 × 列-1）的路径，路径上包含每个顶点恰一次。

输入 第一行是一个整数n，表示正方形棋盘边长，3 <= n <= 19。 第二行是空格分隔的两个整数sr, sc，表示骑士的起始位置坐标。棋盘左上角坐标是 0 0。0 <= sr <= n-1, 0 <= sc <= n-1。 **输出** 如果是合格的周游，输出 success，否则输出 fail。

```
def is_valid_move(board_size, visited, row, col):
    return 0 <= row < board_size and 0 <= col < board_size and not
visited[row][col]

def knight_tour(board_size, start_row, start_col):
    moves = [(-2, -1), (-2, 1), (-1, -2), (-1, 2),
             (1, -2), (1, 2), (2, -1), (2, 1)]

    visited = [[False] * board_size for _ in range(board_size)]
    visited[start_row][start_col] = True
```

```
    def get_neighbors(row, col):
        neighbors = []
        for dr, dc in moves:
            next_row, next_col = row + dr, col + dc
            if is_valid_move(board_size, visited, next_row, next_col):
                count = sum(1 for dr, dc in moves if
is_valid_move(board_size, visited, next_row + dr, next_col + dc))
                neighbors.append((count, next_row, next_col))
        return neighbors

    def dfs(row, col, count):
        if count == board_size ** 2 - 1:
            return True

        neighbors = get_neighbors(row, col)
        neighbors.sort()

        for _, next_row, next_col in neighbors:
            visited[next_row][next_col] = True
            if dfs(next_row, next_col, count + 1):
                return True
            visited[next_row][next_col] = False

        return False

    return dfs(start_row, start_col, 0)

board_size = int(input())
start_row, start_col = map(int, input().split())
if knight_tour(board_size, start_row, start_col):
    print("success")
else:
    print("fail")
```

马走日

```
def is_valid_move(board_size, visited, row, col):
    return 0 <= row < board_size[0] and 0 <= col < board_size[1] and not
visited[row][col]

def knight_tour(board_size, start_row, start_col):
    moves = [(2, 1), (2, -1), (-2, 1), (-2, -1),
             (1, 2), (1, -2), (-1, 2), (-1, -2)]

    visited = [[False] * board_size[1] for _ in range(board_size[0])]
    visited[start_row][start_col] = True
    count = [0]

    def dfs(row, col, visited, count):
        if all(all(row) for row in visited): #检查是否所有的方格都被访问过了。
            count[0] += 1
            return
```

```
        for dr, dc in moves:
            next_row, next_col = row + dr, col + dc
            if is_valid_move(board_size, visited, next_row, next_col):
                visited[next_row][next_col] = True
                dfs(next_row, next_col, visited, count)
                visited[next_row][next_col] = False

    dfs(start_row, start_col, visited, count)
    return count[0]


T = int(input())

for _ in range(T):
    n, m, x, y = map(int, input().split())
    print(knight_tour((n, m), x, y))
```

八皇后

```
def check(board, row, col):
    for i in range(row):
        if board[i] == col or abs(row - i) == abs(col - board[i]):
            return False
    return True

def solve(board, row, count, target):
    if row == 8:
        count[0] += 1
        if count[0] == target:
            print("".join(str(col + 1) for col in board))
            return True
    else:
        for col in range(8):
            if check(board, row, col):
                board[row] = col
                if solve(board, row + 1, count, target):
                    return True
    return False

def find_queen_sequence(b):
    board = [0] * 8
    count = [0]
    solve(board, 0, count, b)

n = int(input())
for _ in range(n):
    b = int(input())
    find_queen_sequence(b)
```

算鹰

```
def dfs(board,row,col,visited):
    if 0 > row or row >= 10 or 0 > col or col >= 10 or visited[row][col]
or board[row][col] == '-':
        return  #退出函数

    visited[row][col] == True
    dfs(board,row+1,col,visited)
    dfs(board,row-1,col,visited)
    dfs(board,row,col+1,visited)
    dfs(board,row,col-1,visited)

def eagles(board):
    visited = [[False]*10 for _ in range(10)]
    count = 0

    for i in range(10):
        for j in range(10):
            if not visited[i][j] and board[i][j] == '.':
                dfs(board,i,j,visited)
                count += 1
    return count

board = []
for _ in range(10):
    row = input().strip()
    board.append(row)
print(eagles(board))
```

bfs

找最短路径、最少步数、最少交换次数等。

鸣人和佐助

```
from collections import deque

moves = [(-1, 0), (0, -1), (1, 0), (0, 1)]
flag = 0
ans = 0


def bfs(x, y, t):
    visited = set()
    global ans, flag
    q = deque()
    q.append((t, x, y, 0))
    while q:
        t, x, y, ans = q.popleft()
```

```
        for dx, dy in moves:
            nx = x + dx
            ny = y + dy
            if 0 <= nx < m and 0 <= ny < n:
                if g[nx][ny] != "#":
                    nt = t
                else:
                    nt = t - 1
                if nt >= 0 and (nt, nx, ny) not in visited:

                    newans = ans + 1
                    if g[nx][ny]=="+":
                        flag = 1
                        return flag,newans
                    q.append((nt, nx, ny, newans))
                    visited.add((nt, nx, ny))
    return flag,ans


m, n, t = map(int, input().split())
g = []
for i in range(m):
    g.append(list(input()))
for i in range(m):
    for j in range(n):
        if g[i][j] == "@":
            x = i
            y = j
flag,newans=bfs(x, y, t)
if flag:
    print(newans)
else:
    print(-1)
```

走山路 dijkstra

```
def bfs(x, y):
    directions = [(0, -1), (0, 1), (1, 0), (-1, 0)]
    queue = [(x, y)]  #存储待处理的节点（坐标）
    distances = {(x, y): 0} #存储每个节点（坐标）到起点的最短距离

    while queue:
        current_x, current_y = queue.pop(0)

        for dx, dy in directions:
            new_x, new_y = current_x + dx, current_y + dy

            if 0 <= new_x < m and 0 <= new_y < n:
                if d[new_x][new_y] != '#':
                    new_distance = distances[(current_x, current_y)] +
abs(int(d[new_x][new_y]) - int(d[current_x][current_y]))
```

```
                    if (new_x, new_y) not in distances or new_distance <
distances[(new_x, new_y)]:
                        distances[(new_x, new_y)] = new_distance
                        queue.append((new_x, new_y))

    return distances

m, n, p = map(int, input().split())
d = []
for _ in range(m):
    row = input().split()
    d.append(row)

for _ in range(p):
    x1, y1, x2, y2 = map(int, input().split())

    if d[x1][y1] == '#' or d[x2][y2] == '#':
        print('NO')
        continue

    distances = bfs(x1, y1)

    if (x2, y2) in distances:
        print(distances[(x2, y2)])
    else:
        print('NO')
```

寻宝

```
from collections import deque

def find(m,n,treasures):
    start_x,start_y = 0,0
    queue = deque([(start_x, start_y,0)])
    directions = [(1,0),(-1,0),(0,1),(0,-1)]
    visited = [[False]*n for _ in range(m)]
    visited[start_x][start_y] = True

    while queue:
        x,y,count = queue.popleft()

        if treasures[x][y] == 1:
            return count

        for dx,dy in directions:
            new_x,new_y = x + dx,y + dy

            if 0 <= new_x < m and 0 <= new_y < n and not visited[new_x]
[new_y] and treasures[new_x][new_y] != 2:
                visited[new_x][new_y] = True
                queue.append((new_x,new_y,count+1))
```

```
    return 'NO'


m,n = map(int,input().split())
treasures = []
for _ in range(m):
    row = list(map(int,input().split()))
    treasures.append(row)

result = find(m,n,treasures)
print(result)
```

pots

```
def bfs(A,B,C):
    start = (0,0)
    visited = set()
    visited.add(start)
    queue = [(start,[])]

    while queue:
        (a,b),actions = queue.pop(0) #a,b是A瓶和B瓶的状态

        if a == C or b == C:
            return actions

        next_steps = [(A,b),(a,B),(0,b),(a,0),(min(a+b,A),max(0,a+b-A)),
(max(0,a+b-B),min(B,a+b))]
        #(A, b): 装满第一个水壶。
        #(a, B): 装满第二个水壶。
        #(0, b): 倒空第一个水壶。
        #(a, 0): 倒空第二个水壶。
        #(min(a + b, A), max(0, a + b - A)): 从第二个水壶倒水到第一个水壶，直到第
一个水壶满或第二个水壶空。
        #(max(0, a + b - B), min(a + b, B)): 从第一个水壶倒水到第二个水壶，直到第
二个水壶满或第一个水壶空。

        for i in next_steps:
            if i not in visited:
                visited.add(i)
                new_actions = actions + [get_action(a, b, i)]
                queue.append((i,new_actions))

    return ['impossible']

def get_action(a,b,next_steps):
    if next_steps == (A, b):
        return "FILL(1)"
    elif next_steps == (a, B):
        return "FILL(2)"
    elif next_steps == (0, b):
        return "DROP(1)"
```

```
    elif next_steps == (a, 0):
        return "DROP(2)"
    elif next_steps == (min(a + b, A), max(0, a + b - A)):
        return "POUR(2,1)"
    else:
        return "POUR(1,2)"

A, B, C = map(int, input().split())
solution = bfs(A, B, C)

if solution == ["impossible"]:
    print(solution[0])
else:
    print(len(solution))
    for i in solution:
        print(i)
```

shunting yard

中序表达式转后序表达式

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 precedence(op):
    if op == '+' or op == '-':
        return 1
    elif op == '*' or op == '/':
         return 2
    else:
        return 0

def infix_to_postfix(expression):
    stack = []
    postfix = []
    operation = set(['+','-','*','/'])
    numbers = ''
```

```
    for char in expression:
        if char.isdigit() or char == '.':
            numbers += char
        elif char in operation:
            if numbers:
                postfix.append(numbers)
                numbers = ''
            while stack and stack[-1] in operation and precedence(char) <=
precedence(stack[-1]):
                postfix.append(stack.pop())
            stack.append(char)
        elif char == '(':
            stack.append(char)
        elif char == ')':
            if numbers:
                postfix.append(numbers)
                numbers = ''
            while stack and stack[-1] != '(':
                postfix.append(stack.pop())
            stack.pop()  # 弹出左括号

    if numbers:
        postfix.append(numbers)

    while stack:
        postfix.append(stack.pop())

    return ' '.join(postfix)

n = int(input())
for _ in range(n):
    expression = input().strip()
    print(infix_to_postfix(expression))
```

deque

先进先出

```
from collections import deque
```

*如果print的时候不要有deque的格式记得要：print(' '.join(map(str,result)))

小组队列

```
from collections import deque

t = int(input())
groups = {}
```

```
member_to_group = {}

for _ in range(t):
    members = list(map(int, input().split()))
    group_id = members[0]
    groups[group_id] = deque()
    for member in members:
        member_to_group[member] = group_id

queue = deque()
queue_set = set()


while True:
    command = input().split()
    if command[0] == 'STOP':
        break
    elif command[0] == 'ENQUEUE':
        x = int(command[1])
        group = member_to_group.get(x, None)
        if group is None:
            group = x
            groups[group] = deque([x])
            member_to_group[x] = group
        else:
            groups[group].append(x)
        if group not in queue_set:
            queue.append(group)
            queue_set.add(group)
    elif command[0] == 'DEQUEUE':
        if queue:
            group = queue[0]
            x = groups[group].popleft()
            print(x)
            if not groups[group]:
                queue.popleft()
                queue_set.remove(group)
```

stack

先进后出

单调栈

单调递增/单调递减

描述 给出项数为 n 的整数数列 a1...an。

定义函数 f(i) 代表数列中第 i 个元素之后第一个大于 ai 的元素的下标。若不存在，则 f(i)=0。

试求出 f(1...n)。

输入 第一行一个正整数 n。 第二行 n 个正整数 a1...an。 **输出** 一行 n 个整数表示 f(1), f(2), ..., f(n) 的值。 **样例输入** 5 1 4 2 3 5 **样例输出** 2 5 4 5 0

```
n = int(input())
nums = list(map(int,input().split()))
stack = []

for i in range(n):
    while stack and nums[stack[-1]] < nums[i]: #单调递增
        nums[stack.pop()] = i + 1

    stack.append(i)

while stack:
    nums[stack[-1]] = 0
    stack.pop()

print(*nums)
```

奶牛排队

1. 使用单调递增栈来找到每个元素左边第一个比它大的元素。
2. 使用单调递减栈来找到每个元素右边第一个比它小的元素。
3. 然后在这些范围内找到符合条件的最多奶牛数。

```
def max_cows(n,hi):
    left = [-1] * n
    right = [n] * n

    stack = []
    for i in range(n):
        while stack and hi[stack[-1]] < hi[i]:
            stack.pop()
        if stack:
            left[i] = stack[-1]
        stack.append(i)

    stack = []
    for i in range(n-1,-1,-1):
        while stack and hi[stack[-1]] > hi[i]:
            stack.pop()
        if stack:
            right[i] = stack[-1]
        stack.append(i)

    max_cows = 0
    for i in range(n):
        for j in range(left[i]+1,i):
            if right[j] > i:
                max_cows = max(max_cows,i-j+1)
```

```
                break

    return max_cows

n = int(input())
hi = [int(input()) for _ in range(n)]

print(max_cows(n,hi))
```

其他stack题目

合法出栈序列

```
def find(s1,s2):
    stack = []
    if len(s1) != len(s2):
        return False
    else:
        l = len(s1)
        stack.append(s1[0])
        p1,p2 = 1,0
        while p1 < l:
            if len(stack) > 0 and stack[-1] == s2[p2]:
                stack.pop()
                p2 += 1
            else:
                stack.append(s1[p1])
                p1 += 1
        return ''.join(stack[::-1]) == s2[p2:]

s1 = input()
while True:
    try:
        s2 = input()
    except:
        break
    if find(s1,s2):
        print('YES')
    else:
        print('NO')
```

检测括号嵌套

```
import sys

s = input()
stack = []
dt = {"]":"[",")":"(","}":"{"}
maxDepth = 0
```

```
found = False
for c in s:
    if c in '[({':
        stack.append(c)
    elif c in '])}':
        if stack[-1] == dt[c]:
            if len(stack) < maxDepth:
                found = True
            else:
                maxDepth = len(stack)
            stack.pop(-1)
        else:
            print("ERROR")
            sys.exit()
if len(stack) > 1:
    print("ERROR")
else:
    if found:
        print("YES")
    else:
        print("NO")
```

整人的提词本

```

s = input()
stack = []

for char in s:
    if char == ')':
        sub = []
        while stack and stack[-1] != '(':
            sub.append(stack.pop())
        if stack:
            stack.pop()
        stack.extend(sub)
    else:
        stack.append(char)

print(''.join(stack))
```

并查集

并查集的基本操作

并查集的两个基本操作是Find和Union

find

```

```

```
def find(x):
    if parent[x] != x:
        parent[x] = find(parent, parent[x])  # 路径压缩
    return parent[x]
```

union

```
def union(x, y):
    rootX = find(x)
    rootY = find(y)
    if rootX != rootY:
        parent[rootY] = rootX
```

题目

我想完成数算作业：代码

描述 假设a和b作业雷同，b和c作业雷同，则a和c作业雷同。所有抄袭现象都会被发现，且雷同的作业只有一份独立完成的原版，请输出独立完成作业的人数

输入 第一行输入两个正整数表示班上的人数n与总比对数m，接下来m行每行均为两个1-n中的整数i和j，表明第i个同学与第j个同学的作业雷同。 **输出** 独立完成作业的人数

```
n,m = map(int,input().split())
parent = [i for i in range(n+10)]
#sum = [1 for i in range(n+10)] #如果要统计每个集合最终多少个元素，可以定义这个列表
def find(a):
    if parent[a] != a:
        parent[a] = find(parent[a])
    return parent[a]
def merge(a,b):
    pa = getRoot(a)
    pb = getRoot(b)
    if pa != pb:
        parent[pa] = parent[pb]
        #sum[pb] += sum[pa] #如果要统计最终每个集合的元素个数，可以开设sum[]列表，此处执行本语句
        #则最后
for i in range(m):
    a,b = map(int,input().split())
    merge(a,b)
total = 0
for i in range(1,n+1):
    if parent[i] == i:  #只有父结点是自身的结点，才是树根。注意，只有i为树根时，sum[i]才能表示集合的元素个数
        total += 1
print(total)
```

宗教信仰

描述 世界上有许多宗教，你感兴趣的是你学校里的同学信仰多少种宗教。 你的学校有n名学生（0 < n <= 50000），你不太可能询问每个人的宗教信仰，因为他们不太愿意透露。但是当你同时找到2名学生，他们却愿意告诉你他们是否信仰同一宗教，你可以通过很多这样的询问估算学校里的宗教数目的上限。你可以认为每名学生只会信仰最多一种宗教。

输入 输入包括多组数据。 每组数据的第一行包括n和m，0 <= m <= n(n-1)/2，其后m行每行包括两个数字i和j，表示学生i和学生j信仰同一宗教，学生被标号为1至n。输入以一行 n = m = 0 作为结束。 **输出** 对于每组数据，先输出它的编号（从1开始），接着输出学生信仰的不同宗教的数目上限。

```
def find_sets(n, pairs):
    parent = [-1] * (n + 1)

    def find(x):
        if parent[x] < 0:
            return x
        parent[x] = find(parent[x])
        return parent[x]

    def union(x, y):
        x_root = find(x)
        y_root = find(y)
        if x_root != y_root:
            parent[y_root] = x_root

    for pair in pairs:
        union(pair[0], pair[1])

    distinct_sets = set()
    for i in range(1, n + 1):
        distinct_sets.add(find(i))

    return len(distinct_sets)


def main():
    case = 1
    while True:
        n, m = map(int, input().split())
        if n == 0 and m == 0:
            break

        pairs = []
        for _ in range(m):
            i, j = map(int, input().split())
            pairs.append((i, j))

        max_religions = find_sets(n, pairs)
        print("Case {}: {}".format(case, max_religions))
        case += 1
```

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

排队

描述 更形式化地，初始时刻，操场上有 n 位同学，自成一列。每次操作，老师的指令是 "x y"，表示 x 所在的队列排到 y 所在的队列的后面，即 x 的队首排在 y 的队尾的后面。（如果 x 与 y 已经在同一队列，请忽略该指令） 最终的队列数量远远小于 n，老师很满意。请你输出最终时刻每位同学所在队列的队首（排头），老师想记录每位同学的排头，方便找人。

输入 第一行一个整数 T (T≤5)，表示测试数据组数。 接下来 T 组测试数据，对于每组数据，第一行两个整数 n 和 m (n,m≤30000)，紧跟着 m 行每行两个整数 x 和 y (1≤x,y≤n)。 **输出** 共 T 行。 每行 n 个整数，表示每位同学的排头。

```
parent = None
def getRoot(a):
    if parent[a] != a:
        parent[a] = getRoot(parent[a])
    return parent[a]
def merge(a,b):
    pa = getRoot(a)
    pb = getRoot(b)
    if pa != pb:
        parent[pa] = parent[pb]

t = int(input())
for i in range(t):
    n, m = map(int, input().split())
    parent = [i for i in range(n + 10)]
    for i in range(m):
        x,y = map(int,input().split())
        merge(x,y)
    for i in range(1,n+1):
        print(getRoot(i),end = " ")
        #注意，一定不能写成 print(parent[i],end= " ")
        #因为只有执行路径压缩getRoot(i)以后，parent[i]才会是i的树根
    print()
```

食物链

```
def find(x):    # 并查集查询
    if p[x] == x:
        return x
    else:
        p[x] = find(p[x])  # 父节点设为根节点。目的是路径压缩。
        return p[x]

n, k = map(int, input().split())
```

```
p = [0] * (3 * n + 1) #其长度为3*n+1，表示三种状态（同类、猎物、天敌）的n个动物。
for i in range(3 * n + 1):  #并查集初始化
    p[i] = i

ans = 0
for _ in range(k):
    a, x, y = map(int, input().split())
    if x > n or y > n:
        ans += 1
        continue

#每个动物x有三种状态：x：本身、x + n：猎物、x + 2 * n：天敌

    if a == 1:
        if find(x + n) == find(y) or find(y + n) == find(x):
            ans += 1
            continue

        # 合并
        p[find(x)] = find(y)
        p[find(x + n)] = find(y + n)
        p[find(x + 2 * n)] = find(y + 2 * n)
    else:
        if find(x) == find(y) or find(y + n) == find(x):
            ans += 1
            continue
        p[find(x + n)] = find(y)
        p[find(y + 2 * n)] = find(x)
        p[find(x + 2 * n)] = find(y + n)

print(ans)
```

冰阔洛

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

def union(x, y):
    root_x = find(x)
    root_y = find(y)
    if root_x != root_y:
        parent[root_y] = root_x

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

        for _ in range(m):
```

```
            a, b = map(int, input().split())
            if find(a) == find(b):
                print('Yes')
            else:
                print('No')
                union(a, b)

        unique_parents = set(find(x) for x in range(1, n + 1))
        ans = sorted(unique_parents)
        print(len(ans))
        print(*ans)

    except EOFError:
        break
```