有关输入、数据预处理

首先数据处理当然是单一格式的最好，预处理还需要观察和分析题目需要什么样的数据，以简化计算。

Buffer（中序转前序）

```python
s = input().spilt()
word = ""
precedence={'+':1,'-':1,'*':2,'/':2}
for i in range(len(s)):
    if s[i].isalpha():
        word += s[i]

for char in expression:
        if char.isnumeric() or char == ".":
            number += char
        else:
            if number:
                num = float(number)
                postfix.append(int(num) if num.is_integer() else num)
                number = ""
            if char in "+-*/":
                while stack and stack[-1] in "+-*/" and precedence[stack[-1]] >= precedence[char]:
                    postfix.append(stack.pop())
                stack.append(char)
            elif char == "(":
                stack.append(char)
            elif char == ")":
                while stack and stack[-1] != "(":
                    postfix.append(stack.pop())
                stack.pop()
    if number:
        num = float(number)
        postfix.append(int(num) if num.is_integer() else num)

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

```python
def postfix_to_infix(expression):  # 后序转前序
    stack = []
    for token in expression:
        if token.isalnum(): # Operand
            stack.append(token)
        elif token in ["*", "+"]:
            if token == "*":
                if "+" in stack[-2]:
                    stack[-2] = "(" + stack[-2] + ")"
                if "+" in stack[-1]:
                    stack[-1] = "(" + stack[-1] + ")"
            operand2 = stack.pop()
            operand1 = stack.pop()
            stack.append(operand1 + token + operand2)
    return stack.pop()
```

Stack类可以处理的问题

逆波兰表达式求值

```
stack=[]
for t in s:
    if t in '+-*/':
        b,a=stack.pop(),stack.pop()
        stack.append(str(eval(a+t+b)))
    else:
        stack.append(t)
print(f'{float(stack[0]):.6f}')
```

22068: 合法出栈序列

思路：这里开了stack和bank；bank用来存储顺序元素，stack就是栈。元素顺序输出进栈，栈判断能否弹出顶元素，如果不能就存储，否则弹出。直至bank清空，判断stack是否不符合条件（空或顶元素==char）则判断为负

```
def is_valid_pop_sequence(origin, output):
    if len(origin) != len(output):
        return False
    stack = []
    bank = list(origin)

    for char in output:
        while (not stack or stack[-1] != char) and bank:
            stack.append(bank.pop(0))

        if not stack or stack[-1] != char:
            return False

        stack.pop()
    return True
```

🌲

树的接收

```
class TreeNode:
    def __init__(self, value):
        # 二叉树 (binary tree)
        self.value = value
        self.left = None
        self.right = None

        # 左儿子右兄弟树 (First child / Next sibling representation)
        self.value = value
        self.firstChild = None
        self.nextSibling = None
        # 这玩意像个链表，有时候会很好用

n = int(input())
# 一般而言会有一个存Nodes的dict或是list
nodes = [TreeNode() for i in range(n)]
# 甚至会让你找root，这也可以用于记录森林的树量
has_parents = [False] * n

for i in range(n):
    opt = map(int, input().spilt())
    if opt[0] != -1:
        nodes[i].left = nodes[opt[0]]
        has_parent[opt[0]] = True
    if opt[1] != -1:
        nodes[i].right = nodes[opt[1]]
        has_parent[opt[1]] = True
# 这里完成了树的建立

root = has_parent.index(False) # 对于一棵树而言，root可以被方便的确定
```

```
# 伪满二叉树（左孩子右兄弟树）
def build_tree(tempList, index):  # 构建多叉树 index为当前节点在tempList中的索引
    node = create_node()  # 创建节点
    node.x = tempList[index][0]  # 节点值

    if tempList[index][1] == '0' and node.x != '$':  # 如果节点值不为'$'且有子节点
        index += 1
        child, index = build_tree(tempList, index)  # 递归构建子节点
        node.children.append(child)  # 添加子节点
        index += 1
        child, index = build_tree(tempList, index)  # 递归构建子节点
        node.children.append(child)  # 添加子节点

    return node, index  # 返回当前节点及下一个节点的索引
```

```
index = 0  # 同样是二叉树，用"."填充
def tree_build(pre_order):
    global index
    if index >= len(pre_order) or pre_order[index] == ".":
        index += 1
        return None

    root = TreeNode(pre_order[index])
    index += 1
    root.l = tree_build(pre_order)
    root.r = tree_build(pre_order)
    return root

def validate_preorder(array):  # 判断二叉树的前序遍历是否合法，用"#"填充。
    stack = []
    for i in array:
        while stack and i == "#" and stack[-1] == "#":
```

```python
            stack.pop()  # pop the left child
            if not stack:
                return "F"
            stack.pop()  # pop the parent
        stack.append(i)
    return "T" if stack == ["#"] else "F"

while True:
    N = int(input())
    if N == 0:
        break
    array = list(map(str, input().split()))
    print(validate_preorder(array))
```

树的转换

```python
class TreeNode:          # 括号嵌套树 -> 正常的多叉树
    def __init__(self, value):
        self.value = value
        self.children = []

def build_Tree(string):
    node = None
    stack = [] # 及时处理
    for chr in string:
        if chr.isalpha(): # 这个是一个判断函数，多见于buffer
            node = TreeNode(chr)
            if stack:
                stack[-1].children.append(node)
        elif chr == "(":
            stack.append(node)
            node = None # 及时更新
        elif chr == ")":
            node = stack.pop() # 最后返回树根
        else:
            continue
    return node
# stack在这里的运用非常符合栈的定义和特征
```

$$\text{括号嵌套树} \Leftarrow \text{正常的多叉树} \tag{1}$$

```python
def convert_to_bracket_tree(node):
    # 两个终止条件
    if not node:
        return ""
    if not node.children:
        return node.val

    result = node.val + "("
    for i, child in enumerate(node.children):
        result += convert_to_bracket_tree(child)
        if i != len(node.children) - 1:
            result += "," # 核心是“，”的加入，这里选择在一层结束前加入
    result += ")"

    return result
```

文件转化树 (2)

```
def print_disktree(node, indent=0):
    print("|     " * indent + node.name)
    for child in node.children:
        print_disktree(child, indent+1)
    for file in sorted(node.files):
        print("|     " * indent + file)

class DictTree:
    def __init__(self, name):
        self.name = name
        self.files = []
        self.children = []

n = 1
while True:
    stack = [DictTree("ROOT")]
    while True:
        cur = input()
        if cur == "#":
            exit()
        if cur[0] == "f":
            stack[-1].files.append(cur)
        elif cur[0] == "d":
            new_node = DictTree(cur)
            stack[-1].children.append(new_node)
            stack.append(new_node)
        elif cur == "]":
            stack.pop()
        else:
            break
    print(f"DATA SET {n}:")
    n += 1
    print_disktree(stack[0])
    print()
```

建立起表达式树，按层次遍历表达式树的结果前后颠倒就得到队列表达式 (3)

```
多叉树->二叉树
def convert_to_binary(root):  # 传入多叉树的树根
    if not root:
        return None

    binary_node = Node(root.data)
    if root.children:
        binary_node.left = convert_to_binary(root.children[0])
        right_sibling = binary_node.left
        for child in root.children[1:]:
            right_sibling.right = convert_to_binary(child)
            right_sibling = right_sibling.right

    return binary_node  # 输出二叉树的树根
```

```
def build(s):  # s = xyPzwIM
    # 这是后序建树，stack存节点
    stack = []
    for i in s:
        if i.islower():
            node = TreeNode(i)
            stack.append(node)
        else:
            node = TreeNode(i)
            node.right = stack.pop()
            node.left = stack.pop()
            stack.append(node)
    return stack[0]
```

```
def rebuild(pre, mid): #前中序建树
    if not pre:
        return None
    node = TreeNode(pre[0])
    k = mid.index(pre[0])
    root.left = build(pre[1:k+1], mid[:k])  # root.left = build(post[0:k], mid[:k])
    root.right = build(pre[k+1:], mid[k+1:])  # root.right = build(post[k:-1], mid[k+1:])
    return node
```

树的输出

前序（Pre Order)

```
def preorder(root):
    output = [root]
    if root.left:
        output.extend(preorder(root.left))
    if root.right:
        output.extend(preorder(root.right))
    return "".join(output)
    # 对多叉树而言

    for i in root.children:  # 这里的输出不一样，因为孩子不止一个
        output.extend(preorder(i))
    return "".join(output)
```

层级遍历（Level Order Traversal）# 利用BFS（deque)

```
# 层级遍历通常存在于多叉树的问题
from collections import deque
def level_Order(root):
    queue = deque()
    queue.append(root)
    output = []
    while (len(queue) != 0): # 注意这里是一个特殊的BFS,以层为单位
        n = len(queue)
        while (n > 0): #一层层的输出结果
            node = queue.popleft()
            output.append(node.value)
            queue.extend(node.children)  # if node.left: queue.append(node.left)...
            n -= 1
    return output

def print_tree(p):  # 宽度优先遍历并打印镜像映射序列
    Q = deque()  # 队列Q
    s = deque()  # 栈s

    # 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  # 右子节点

    # 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]
            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())
```

解析树（中序改后序，布尔运算）

```
class BinaryTree:
    def __init__(self, root, left=None, right=None):
        self.root = root
        self.leftChild = left
        self.rightChild = right

    def getrightchild(self):
        return self.rightChild
```

```python

    def getleftchild(self):
        return self.leftChild

    def getroot(self):
        return self.root

def postorder(string):    #中缀改后缀 Shunting yard algorightm
    opStack = []
    postList = []
    inList = string.split()
    prec = { '(': 0, 'or': 1,'and': 2,'not': 3}  # 表达式树的优先级，这里是布尔运算的

    for word in inList:
        if word == '(':
            opStack.append(word)
        elif word == ')':
            topWord = opStack.pop()
            while topWord != '(':
                postList.append(topWord)
                topWord = opStack.pop()
        elif word == 'True' or word == 'False':
            postList.append(word)
        else:
            while opStack and prec[word] <= prec[opStack[-1]]:
                postList.append(opStack.pop())
            opStack.append(word)
    while opStack:
        postList.append(opStack.pop())
    return postList

def buildParseTree(infix):       #以后缀表达式为基础建树
    postList = postorder(infix)
    stack = []
    for word in postList:
        if word == 'not':
            newTree = BinaryTree(word)
            newTree.leftChild = stack.pop()
            stack.append(newTree)
        elif word == 'True' or word == 'False':
            stack.append(BinaryTree(word))
        else:
            right = stack.pop()
            left = stack.pop()
            newTree = BinaryTree(word)
            newTree.leftChild = left
            newTree.rightChild = right
            stack.append(newTree)
    currentTree = stack[-1]
    return currentTree

# 表达可以不看
def printTree(parsetree: BinaryTree):
    if parsetree.getroot() == 'or':
        return printTree(parsetree.getleftchild()) + ['or'] + printTree(parsetree.getrightchild())
    elif parsetree.getroot() == 'not':
        return ['not'] + (['('] + printTree(parsetree.getleftchild()) + [')'] if parsetree.leftChild.getroot() not in ['True', 'False'] else printTree(parsetree.getleftchild()))
    elif parsetree.getroot() == 'and':
        leftpart = ['('] + printTree(parsetree.getleftchild()) + [')'] if parsetree.leftChild.getroot() == 'or' else printTree(parsetree.getleftchild())
        rightpart = ['('] + printTree(parsetree.getrightchild()) + [')'] if parsetree.rightChild.getroot() == 'or' else printTree(parsetree.getrightchild())
        return leftpart + ['and'] + rightpart
    else:
        return [str(parsetree.getroot())]

def main():
    infix = input()
    Tree = buildParseTree(infix)
    print(' '.join(printTree(Tree)))

main()
```

Huffman(词频和深度问题)

```python
import heapq
class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq  # heap法

def huffman_encoding(char_freq):
    heap = [Node(char, freq) for char, freq in char_freq.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq) # note: 合并之后 char 字典是空
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]

# 同样的 以depth作为递归深度的线
def external_path_length(node, depth=0):
    if node is None:
        return 0
    if node.left is None and node.right is None:
        return depth * node.freq
    return (external_path_length(node.left, depth + 1) +
            external_path_length(node.right, depth + 1))

def main():
    char_freq = {'a': 3, 'b': 4, 'c': 5, 'd': 6, 'e': 8, 'f': 9, 'g': 11, 'h': 12}
    huffman_tree = huffman_encoding(char_freq)
    external_length = external_path_length(huffman_tree)

#以下把char 和密码对应上了
def encode_huffman_tree(root):
    codes = {}

    def traverse(node, code):
        #if node.char:
        if node.left is None and node.right is None:
            codes[node.char] = code
        else:
            traverse(node.left, code + '0')
            traverse(node.right, code + '1')

    traverse(root, '')
    return codes

def huffman_encoding(codes, string):
    encoded = ''
    for char in string:
        encoded += codes[char]
    return encoded

# 找到第一个字母为止
def huffman_decoding(root, encoded_string):
    decoded = ''
    node = root
    for bit in encoded_string:
        if bit == '0':
            node = node.left
        else:
            node = node.right

        #if node.char:
        if node.left is None and node.right is None:
            decoded += node.char
            node = root
    return decoded
```

BST(二叉搜索树，大的放右边，小的放左边)

```python
def insert(root, value):
    if root is None:
        return TreeNode(value)
    if value > root.val:
        root.right = insert(root.right, value)  # 这里是一个递归计算，核心区
    else:
        root.left = insert(root.left, value)
    return root
```

Trie🌲

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.end_of_word = True

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.end_of_word

    def is_prefix(self, word):  # 判断是否是前缀
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            elif node.children[char].end_of_word:
                return True
            node = node.children[char]
        return False

# 电话号码
t = int(input())
for _ in range(t):
    n = int(input())
    phone_numbers = [input() for _ in range(n)]
    phone_numbers.sort()
    trie = Trie()
    consistent = True

    for phone in phone_numbers:
        if trie.is_prefix(phone):
            consistent = False
            break
        trie.insert(phone)
    if consistent:
        print("YES")
    else:
        print("NO")
```

27928: 遍历树

在建树的时候将value与树建立关系，很清晰的逻辑

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

def add_node(nodes, parent, child):
    if parent not in nodes:
        nodes[parent] = Node(parent)
    if child not in nodes:
        nodes[child] = Node(child)
    nodes[parent].children.append(nodes[child])

def traverse(node):
    values = [node.value] + [child.value for child in node.children]
    values.sort()
    for value in values:
        if value == node.value:
            print(value)
        else:
            traverse(nodes[value])

# Parse the input
n = int(input())
nodes = {}
root = None
leaves = set()
for _ in range(n):
    line = list(map(int, input().split()))
    leaves |= set(line[1:])
    parent = line[0]
    if root is None:
        root = parent
    for child in line[1:]:
        add_node(nodes, parent, child)

for i in nodes.values():
    if i.value not in leaves:
        root = i.value
        break

# Traverse the tree
traverse(nodes[root])

```

并查集(DisjoinSet)

适用场景：划分，计算人群分类数量，连通性问题。这里的rank只是选用，最后还得“归一化”

模板

```
class DisjSet:
    def __init__(self, n):
        # Constructor to create and
        # initialize sets of n items
        self.rank = [1] * n
        self.parent = [i for i in range(n)]

    def find(self, x):
        # Find the root of the set in which element x belongs
        if self.parent[x] != x:
            # Path compression: Make the parent of x the root of its set
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        # Perform union of two sets
        x_root, y_root = self.find(x), self.find(y)

        if x_root == y_root:
            return
        # Attach smaller rank tree under root of higher rank tree
        if self.rank[x_root] < self.rank[y_root]:
            self.parent[x_root] = y_root
        else:
            self.parent[y_root] = x_root
            self.rank[x_root] += 1
# 示例用法
A = DisjSet(5)
B = DisjSet(5)

A.union(0, 1)
A.union(2, 3)

print(A.rank)  # 输出: [2, 1, 2, 1, 1]
print(A.parent)  # 输出: [0, 0, 2, 2, 4]
print(B.rank)  # 输出: [1, 1, 1, 1, 1]
print(B.parent)  # 输出: [0, 1, 2, 3, 4]
```

宗教信仰

```python
# __disjoinset__模板
case = 0
while True:
    n, m = map(int, input().split())
    if n == 0 and m == 0:
        break
    parent = list(range(n+1))
    for _ in range(m):
        i, j = map(int, input().split())
        union(i, j)
    religions = len(set(find(i) for i in range(1, n+1)))  # 归一化的结果
    case += 1
    print("Case %d: %d" % (case, religions))
```

食物链

```python
# 并查集，https://zhuanlan.zhihu.com/p/93647900/
'''
我们设[0,n)区间表示同类，[n,2*n)区间表示x吃的动物，[2*n,3*n)表示吃x的动物。

如果是关系1：
    将y和x合并。将y吃的与x吃的合并。将吃y的和吃x的合并。
如果是关系2：
    将y和x吃的合并。将吃y的与x合并。将y吃的与吃x的合并。
原文链接：https://blog.csdn.net/qq_34594236/article/details/72587829
'''
# p = [0]*150001

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)
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

    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)
```

01703:发现它，抓住它

```
class DisjSet:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n
        self.dist = [0] * n

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

    def union(self, x, y):
        px, py = self.find(x), self.find(y)

        if self.rank[px] < self.rank[py]:
            self.parent[px] = py
            self.dist[px] = self.dist[x] ^ self.dist[y] ^ 1
        else:
            self.parent[py] = px
            self.dist[py] = self.dist[x] ^ self.dist[y] ^ 1
            if self.rank[px] == self.rank[py]:
                self.rank[px] += 1

T = int(input())
for _ in range(T):
    N, M = map(int, input().split())
    ds = DisjSet(N+1)
    for _ in range(M):
        op, a, b = input().split()
        a, b = int(a), int(b)
        if op == 'D':
            ds.union(a, b)
        else:
            if ds.find(a) != ds.find(b):
                print("Not sure yet.")
            else:
                print("In the same gang." if ds.dist[a] == ds.dist[b] else "In different gangs.")
```

🐰

棋盘问题（回溯法）

```python
def dfs(row, k):
    if k == 0:
        return 1
    if row == n:
        return 0
    count = 0
    for col in range(n):
        if board[row][col] == '#' and not col_occupied[col]:
            col_occupied[col] = True
            count += dfs(row + 1, k - 1)
            col_occupied[col] = False
    count += dfs(row + 1, k)
    return count
col_occupied = [False] * n
print(dfs(0, k))
```

BFS体现层数

```python
def bfs(broad, end, start):
    ans = []
    x1, y1 = end
    x0, y0 = start
    vis = set()
    queue = [(1, (x0, y0), f"({x0},{y0})")]
    heapq.heapify(queue)
    max_length = float('inf')
    while queue:
        length, (cur_x, cur_y), path = heapq.heappop(queue)
        if (cur_x, cur_y) in vis:
            continue

        if (cur_x, cur_y) == (x1, y1):
            if not ans:
                max_length = length
            if max_length == length:
                ans.append(path)
            continue
        vis.add((cur_x, cur_y))

        length += 1
        if length <= max_length:
            for i in range(8):
                dx, dy = dic[i]
                pos_x, pos_y = dx//2, dy//2
                new_x, new_y = cur_x + dx, cur_y + dy
                if is_valid(new_x, new_y, vis) and (cur_x + pos_x, cur_y + pos_y) not in block:
                    new_path = path + f"-({cur_x+dic[i][0]},{cur_y+dic[i][1]})"
                    heapq.heappush(queue, (length, (cur_x+dic[i][0], cur_y+dic[i][1]), new_path))
        else:
            return ans
    return ans
```

长得像BFS的Dijkstra

```
# 1.使用vis集合
def dijkstra(start,end):
    heap=[(0,start,[start])]
    vis=set()
    while heap:
        (cost,u,path)=heappop(heap)
        if u in vis: continue
        vis.add(u)
        if u==end: return (cost,path)
        for v in graph[u]:
            if v not in vis:
                heappush(heap,(cost+graph[u][v],v,path+[v]))
# 2.使用dist数组
import heapq
def dijkstra(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    priority_queue = [(0, start)]
    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)
        if current_distance > distances[current_node]:
            continue
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))
    return distances
```

ROAD，有金币限制

```
from heapq import heappop, heappush
from collections import defaultdict

K, N, R = int(input()), int(input()), int(input())
graph = defaultdict(list)
for i in range(R):
    S, D, L, T = map(int, input().split())
    graph[S].append((D, L, T))

def Dijkstra(graph):
    global K, N, R
    q, ans = [], []
    heappush(q, (0, 0, 1, 0))  # (length,cost,cur,step)
    while q:
        l, cost, cur, step = heappop(q)
        if cur == N:
            return l

        for next, nl, nc in graph[cur]:
            if cost + nc <= K and step + 1 < N:
                heappush(q, (l + nl, cost + nc, next, step + 1))
    return -1

print(Dijkstra(graph))
```

兔子与樱花

```
import heapq


def dijkstra(graph, start):
    distances = {node: (float('infinity'), []) for node in graph}
    distances[start] = (0, [start])
    queue = [(0, start, [start])]
    visited = set()
    while queue:
        current_distance, current_node, path = heapq.heappop(queue)
        # 一般的限制条件在这里加
        if current_node in visited:  # 湮灭点
            continue
        visited.add(current_node)

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor][0]: # 湮灭点，可以是限制条件点
                distances[neighbor] = (distance, path + [neighbor])
                heapq.heappush(queue, (distance, neighbor, path + [neighbor]))
    return distances


P = int(input())
places = {input(): i for i in range(P)}
graph = {i: {} for i in range(P)}

Q = int(input()) # Graph的建立，邻接表
for _ in range(Q):
    place1, place2, distance = input().split()
    distance = int(distance)
    graph[places[place1]][places[place2]] = distance
    graph[places[place2]][places[place1]] = distance

R = int(input())
for _ in range(R):
    start, end = input().split()
    distances = dijkstra(graph, places[start])
    path = distances[places[end]][1]
    result = ""
    for i in range(len(path) - 1):
        result += f"{list(places.keys())[list(places.values()).index(path[i])]}->({graph[path[i]][path[i + 1]]})->"
    result += list(places.keys())[list(places.values()).index(path[-1])]
    print(result)
```

Prim 最小生成树算法：

```
import heapq  # truck_history
def calculate_distance(a, b):
    return sum(a[i] != b[i] for i in range(7))

def prim(truck_types, n):
    if n <= 1:
        return 0

    min_heap = [(0, 0)]
    dist = [float("inf")] * n  # 最小，这里也是剪枝
    dist[0] = 0
    in_mst = [False] * n
    total_cost = 0
    edge_count = 0

    while min_heap and edge_count < n:
        cost, u = heapq.heappop(min_heap)
        if in_mst[u]:  # 不访问见过的点
            continue
        in_mst[u] = True  # 及时标记
        total_cost += cost
        edge_count += 1  # 这是边缘条件，最大限度

        for v in range(n):
            if v != u:
                if not in_mst[v]:  # 及时运算，减少运算量，存dp可能会更快
                    distance = calculate_distance(truck_types[u], truck_types[v])
                    if distance < dist[v]:  # 最小，这里也是剪枝
                        heapq.heappush(min_heap, (distance, v))
                        dist[v] = distance

    return total_cost

while True:
    n = int(input())
    if n == 0:
        exit()
    trucks = []
    for i in range(n):
        trucks.append(list(input()))
    ans = prim(trucks, n)
    print(f"The highest possible quality is 1/{ans}.")
```

Prim的算法和Kruskal的算法都用于查找连接的加权图的最小生成树（MST）。

兔子与星空

```python
import heapq

def prim(graph, start):
    mst = []
    used = set([start])  # 已经使用过的点
    edges = [
        (cost, start, to)
        for to, cost in graph[start].items()
    ]  # (cost, frm, to) 的列表
    heapq.heapify(edges)  # 转换成最小堆

    while edges:  # 当还有边可以选择时
        cost, frm, to = heapq.heappop(edges)    # 弹出最小边
        if to not in used:  # 如果这个点还没被使用过
            used.add(to)  # 标记为已使用
            mst.append((frm, to, cost))  # 加入到最小生成树中
            for to_next, cost2 in graph[to].items():  # 将与这个点相连的边加入到堆中
                if to_next not in used:  # 如果这个点还没被使用过
                    heapq.heappush(edges, (cost2, to, to_next))  # 加入到堆中

    return mst  # 返回最小生成树

n = int(input())
graph = {chr(i+65): {} for i in range(n)}  # 邻接表
for i in range(n-1):
    data = input().split()
    node = data[0]
    for j in range(2, len(data), 2):
        graph[node][data[j]] = int(data[j+1])
        graph[data[j]][node] = int(data[j+1])

mst = prim(graph, 'A')  # 从A开始生成最小生成树
print(sum([cost for frm, to, cost in mst]))  # 输出最小生成树的总权值
```

Kruskal：（适用于稀疏图）

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(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):
        px, py = self.find(x), self.find(y)
        if self.rank[px] > self.rank[py]:
            self.parent[py] = px
        else:
            self.parent[px] = py
            if self.rank[px] == self.rank[py]:
                self.rank[py] += 1

def kruskal(n, edges):
    uf = UnionFind(n)
    edges.sort(key=lambda x: x[2])
    res = 0
    for u, v, w in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            res += w
    if len(set(uf.find(i) for i in range(n))) > 1:
        return -1
    return res

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

拓扑排序算法：

- DFS：用于对有向无环图（DAG）进行拓扑排序。
- Karn算法 / BFS ： 用于对有向无环图进行拓扑排序。

```
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，那环内的元素都出不去
    if len(result) == len(graph):
        return result
    else:
        return None

# 示例调用代码，建立邻接表
graph = {
    'A': ['B', 'C'],
    'B': ['C', 'D'],
    'C': ['E'],
    'D': ['F'],
    'E': ['F'],
    'F': []
}

sorted_vertices = topological_sort(graph)
if sorted_vertices:
    print("Topological sort order:", sorted_vertices)
else:
    print("The graph contains a cycle.")

# Output:
# Topological sort order: ['A', 'B', 'C', 'D', 'E', 'F']
```

Kosaraju算法(强连通图)：

```
def dfs1(graph, node, visited, stack):
    visited[node] = True
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs1(graph, neighbor, visited, stack)
    stack.append(node)

def dfs2(graph, node, visited, component):
    visited[node] = True
    component.append(node)
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs2(graph, neighbor, visited, component)

def kosaraju(graph):
    # Step 1: Perform first DFS to get finishing times
    stack = []
    visited = [False] * len(graph)
    for node in range(len(graph)):
        if not visited[node]:
            dfs1(graph, node, visited, stack)

    # Step 2: Transpose the graph
    transposed_graph = [[] for _ in range(len(graph))]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)

    # Step 3: Perform second DFS on the transposed graph to find SCCs
    visited = [False] * len(graph)
    sccs = []
    while stack:
        node = stack.pop()
        if not visited[node]:
            scc = []
            dfs2(transposed_graph, node, visited, scc)
            sccs.append(scc)
    return sccs

# Example
graph = [[1], [2, 4], [3, 5], [0, 6], [5], [4], [7], [5, 6]]
sccs = kosaraju(graph)
print("Strongly Connected Components:")
for scc in sccs:
    print(scc)

"""
Strongly Connected Components:
[0, 3, 2, 1]
[6, 7]
[5, 4]

"""
```

很怪的小组队列

```
from collections import deque

t = int(input())
teams = {i: deque(map(int, input().split())) for i in range(t)}
team_member = {person: i for i, team in teams.items() for person in team}
queue = deque()
group_queue = {i: deque() for i in range(t)}


while True:
    command = input().split()
    if command[0] == 'STOP':
        break
    elif command[0] == 'ENQUEUE':
        person = int(command[1])
        if person in team_member:
            i = team_member[person]
            group_queue[i].append(person)
            if i not in queue:
                queue.append(i)
        else:
            t += 1
            group_queue[t] = deque([person])
            queue.append(t)
    elif command[0] == 'DEQUEUE':
        group = queue[0]
        print(group_queue[group].popleft())
        if not group_queue[group]:
            queue.popleft()
```

词梯

```
from collections import defaultdict, deque

def visit_vertex(queue, visited, other_visited, graph):
    word, path = queue.popleft()
    for i in range(len(word)):
        pattern = word[:i] + '_' + word[i + 1:]
        for next_word in graph[pattern]:
            if next_word in other_visited:
                return path + other_visited[next_word][::-1]
            if next_word not in visited:
                visited[next_word] = path + [next_word]
                queue.append((next_word, path + [next_word]))

def word_ladder(words, start, end):
    graph = defaultdict(list)
    for word in words:
        for i in range(len(word)):
            pattern = word[:i] + '_' + word[i + 1:]
            graph[pattern].append(word)

    queue_start = deque([(start, [start])])
    queue_end = deque([(end, [end])])
    visited_start = {start: [start]}
    visited_end = {end: [end]}

    while queue_start and queue_end:
        result = visit_vertex(queue_start, visited_start, visited_end, graph)
        if result:
            return ' '.join(result)
        result = visit_vertex(queue_end, visited_end, visited_start, graph)
        if result:
            return ' '.join(result[::-1])

    return 'NO'


n = int(input())
words = [input() for i in range(n)]
start, end = input().split()
print(word_ladder(words, start, end))
```

骑士周游 · 启发式算法

```
from functools import lru_cache
```

```python

# initializing
size = int(input())
matrix = [[False]*size for i in range(size)]
x, y = map(int, input().split())
dir = [(2, 1), (1, 2), (-1, 2), (-2, 1), (-2, -1), (-1, -2), (1, -2), (2, -1)]

def valid(x, y):
    return 0 <= x < size and 0 <= y < size and not matrix[x][y]

def get_degree(x, y):
    count = 0
    for dx, dy in dir:
        nx, ny = x + dx, y + dy
        if valid(nx, ny):
            count += 1
    return count

@lru_cache(maxsize = 1<<30)
def dfs(x, y, count):
    if count == size**2:
        return True

    matrix[x][y] = True

    next_moves = [(dx, dy) for dx, dy in dir if valid(x + dx, y + dy)]
    next_moves.sort(key=lambda move: get_degree(x + move[0], y + move[1]))

    for dx, dy in next_moves:
        if dfs(x + dx, y + dy, count + 1):
            return True

    matrix[x][y] = False
    return False

if dfs(x, y, 1):
    print("success")
else:
    print("fail")
```

鸣人和佐助

```python
# 真就拯救行动了
import heapq
def find(matrix, N, M):
    for i in range(N):
        for j in range(M):
            if matrix[i][j] == '@':  # @代表鸣人
                return i, j
    return -2, -2

def bfs(matrix, N, M, T, i, j):
    dir = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    queue = [(0, i, j, T)]
    heapq.heapify(queue)
    visited = [[-1] * M for _ in range(N)]
    visited[i][j] = T
    while queue:
        step, i, j, cha = heapq.heappop(queue)
        for dx, dy in dir:
            x, y = i + dx, j + dy
            if 0 <= x < N and 0 <= y < M and matrix[x][y] == '+':  # +代表佐助

                return step + 1

            if 0 <= x < N and 0 <= y < M:
                if matrix[x][y] == '#' and cha > 0:  # #代表大蛇丸的手下
                    if visited[x][y] >= cha - 1:
                        continue
                    else:
                        heapq.heappush(queue, (step + 1, x, y, cha - 1))
                        visited[x][y] = cha - 1
                elif matrix[x][y] == '*':  # *代表通路
                    if visited[x][y] >= cha:
                        continue
                    else:
                        heapq.heappush(queue, (step + 1, x, y, cha))
                        visited[x][y] = cha

    return -1

while True:
    try:
        N, M, T = map(int, input().split())
        matrix = [list(map(str, input())) for _ in range(N)]
        i, j = find(matrix, N, M)
        res = bfs(matrix, N, M, T, i, j)
        print(res)
    except EOFError:
        break
```

模板

单调栈

右侧第一个大于的单调栈

```
n = int(input())
array = list(map(int, input().split()))
ans = [0] * n
stack = []
for i in range(n-1, -1, -1):
    while stack and array[stack[-1]] <= array[i]:
        stack.pop()

    if stack:
        ans[i] = stack[-1] + 1

    stack.append(i)

print(*ans)
# 奶牛排队
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)
```

快速堆猪（类最小堆）

```
stack = []
m_list = []
while True:
    try:
        opt = input().split()
        if opt[0] == "pop":
           if stack:
                out_ = stack.pop()
                if m_list[-1] == out_:
                    m_list.pop()
                # print(out)

        elif opt[0] == "min":
            if stack:
                print(m_list[-1])

        else:
            in_ = int(opt[1])
            stack.append(in_)
            if m_list:
                if in_ <= m_list[-1]:
                    m_list.append(in_)
            else:
                m_list.append(in_)

    except EOFError:
        break
```

N皇后

```
def isvalid(former,row,col):
    for i in range(row):  # 肯定不共行，判断是否共列或共对角线
        if former[i] == col or abs(i-row) == abs(former[i]-col):
            return False
    return True

def queen(former=[],row=0):
    if row == n:  # 结果储存
        result.append(former[:])
        return
    for col in range(n):
        if isvalid(former,row,col):
          former.append(col)  # 压入
          queen(former,row+1)  # 状态转移方程
          former.pop()  # 回溯

n = int(input())
result = []
queen()
if result:
    for _ in result:
        print(*_)
else:
    print("NO ANSWER")
```

动态中位数

```
import heapq


def find_median(numbers):
    min_heap = []
    max_heap = []
    for i, number in enumerate(numbers):
        heapq.heappush(max_heap, -heapq.heappushpop(min_heap, number))
        if len(max_heap) > len(min_heap):
            heapq.heappush(min_heap, -heapq.heappop(max_heap))

        if i % 2 == 0:
            ans.append(min_heap[0])


T = int(input())
for i in range(T):
    ans = []
    arr = list(map(int, input().split()))
    find_median(arr)
    print(len(ans))
    print(*ans)
```

质数筛

```python
import math
n = int(1e6)
ans = [False]*(n+1)
ans[1] = True
ans_list = set()
for i in range(2,int(math.sqrt(n+1)+1)):
    if not ans[i]:
        for j in range(i**2,n+1,i):
            ans[j]= True
for i in range(2,n+1):
    if not ans[i]:
        ans_list.add(i)
```

最长上升子序列

```python
N = int(input())
nums = list(map(int, input().split()))  # 输入一组序列
length = len(nums)
# print(n)
dp = [1] * (length + 1)

for i in range(length):
    for j in range(0, i):
        if nums[i] >= nums[j]:
            # 状态：dp[i] 表示以 nums[i] 结尾的「上升子序列」的长度
            # 当nums[i]前面存在小于nums[i]的nums[j]，
            # 则暂存在dp[j]+1就是当前nums[i]的最长增长子序列的长度
            dp[i] = max(dp[i], dp[j] + 1)
print(max(dp))  # 用函数max直接找到dp数组的最大值，无需再遍历了
```

Mergesort

```python
k = 0
def MergeSort(lists):
    if len(lists) <= 1:
        return lists
    Mid = len(lists)//2
    Left_lists = MergeSort(lists[:Mid])
    Right_lists = MergeSort(lists[Mid:])
    return Merge(Left_lists,Right_lists)

def Merge(Left,Right):
    global k
    Sortedlist = []
    i, j = 0, 0
    while i < len(Left) and j < len(Right):
        # print(i,j)
        if Left[i] <= Right[j]:
            Sortedlist.append(Left[i])
            i += 1
        else:
            Sortedlist.append(Right[j])
            k += len(Left) - i
            j += 1
        # print((Left,Right),k)
    Sortedlist += Left[i:]
    Sortedlist += Right[j:]
    # print(Sortedlist,k)
    return Sortedlist
```

汉诺塔

```
numDisks, a, b, c = input().split()
numDisks = int(numDisks)
def moveOne(x, init, desti):
    print(f"{x}:{init}->{desti}")
    return

def move(numDisks, init, temp, desti):
    if numDisks == 1:
        moveOne(1, init, desti)
    else:
        move(numDisks - 1, init, desti, temp)
        moveOne(numDisks, init, desti)
        move(numDisks - 1, temp, init, desti)

move(numDisks, a, b, c)
```

二分查找算法

月度开销

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

def check(x):
    num, s = 1, 0
    for i in range(n):
        if s + expenditure[i] > x:
            s = expenditure[i]  # 装不了了
            num += 1  # 新开一个月
        else:
            s += expenditure[i]  # 向月里加天

    return [False, True][num > m]

lo = max(expenditure)
hi = sum(expenditure) + 1  # 绝对大值
ans = 1
while lo < hi:
    mid = (lo + hi) // 2
    if check(mid):  # 返回True，是因为num>m，是确定不合适
        lo = mid + 1  # 所以lo可以置为 mid + 1。
```

```python
    else:
        ans = mid  # 如果num==m, mid就是答案
        hi = mid

# print(lo)
print(ans)
```

跳高（二分维护桶）

```python
from bisect import *
cur_temps = []
N = int(input())
scores = list(map(int, input().split()))
for idx in range(N):
    cur = scores[idx]
    if cur_temps:
        if cur >= cur_temps[-1]:
            cur_temps[-1] = cur
        else:
            ind = bisect(cur_temps, cur)
            if ind == 0:
                cur_temps.insert(0, cur)
            else:
                cur_temps[ind - 1] = cur
    else:
        cur_temps.append(cur)
print(len(cur_temps))
```

Pots：一种很新的模拟

```python
def bfs(A, B, C):
    queue = deque([((0, 0), [])])
    visited = set([(0, 0)])
    while queue:
        (a, b), path = queue.popleft()
        if a == C or b == C:
            return path
        states = [((A, b), path + ['FILL(1)']),
                  ((a, B), path + ['FILL(2)']),
                  ((0, b), path + ['DROP(1)']),
                  ((a, 0), path + ['DROP(2)']),
                  ((a-min(a, B-b), b+min(a, B-b)), path + ['POUR(1,2)']),
                  ((a+min(b, A-a), b-min(b, A-a)), path + ['POUR(2,1)'])]
        for state, new_path in states:
            if state not in visited:
                queue.append((state, new_path))
                visited.add(state)
    return None

```

工具们

MLE的应对法

1. 余数法（判断取余条件）
2. 剪枝
3. 加强条件

TLE的应对法：

1. 检查是否有冗余或低效的操作。（list 内存 比 string小，在set里找比算完判断快）
2. 改算法（快跑🏃💨

```python
input().replace('ud', 'x')
```

permutations：全排列

```python
from itertools import permutations
# 创建一个可迭代对象的排列
perm = permutations([1, 2, 3])
# 打印所有排列
for p in perm:
    print(p)
# 输出: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)
```

combinations：组合

```python
from itertools import combinations
# 创建一个可迭代对象的组合
comb = combinations([1, 2, 3], 2)
# 打印所有组合
for c in comb:
    print(c)
# 输出: (1, 2), (1, 3), (2, 3)
```

reduce：累次运算

```python
from functools import reduce
# 使用reduce计算列表元素的乘积
product = reduce(lambda x, y: x * y, [1, 2, 3, 4])
print(product)  # 输出: 24
```

product：笛卡尔积

```python
from itertools import product
# 创建两个可迭代对象的笛卡尔积
prod = product([1, 2], ['a', 'b'])
# 打印所有笛卡尔积对
for p in prod:
    print(p)
# 输出: (1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')
```

lrucache

```python
from functools import lru_cache
@lru_cache(maxsize=None)
```

递归次数

```python
import sys
sys.setrecursionlimit(1 << 30)
```