质数生成，输出小数位数

```
def sieve_of_eratosthenes(n):
    primes = [True] * (n + 1)
    primes[0] = False
    primes[1] = False

    for i in range(2, int(n ** 0.5) + 1):
        if primes[i]:
            for j in range(i * i, n + 1, i):
                primes[j] = False

    return set([i for i in range(2, n + 1) if primes[i]])
    # primes里True为质数


# 输出确定小数位数
num = 3.14159
print(f"{num:.3f}")  # 输出3.142
print(round(num, 3))
```

排序

```
# 以二叉树根节点值排序
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

trees = [Node(10), Node(5), Node(20), Node(15), Node(30)]

# 使用二叉树的根节点值作为排序键
sorted_trees = sorted(trees, key=lambda tree: tree.val)


# 以字典的值排序
dict1 = {"a": 3, "b": 2, "c": 1}

# 使用 sorted() 函数对字典进行排序
sorted_list = sorted(dict1.items(), key=lambda x: x[1])

# 将排序后的列表转换为字典
sorted_dict = dict(sorted_list)
```

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。

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。

以字典和列表建树的典型输入

```
def out(r):
    if tree[r] == []:
        print(r)
    else:
        l = tree[r] + [r]
        l.sort()
        for i in l:
            if i != r:
                out(i)
            else:
                print(r)


n = int(input())
tree = dict()
node_not_root = set()
for _ in range(n):
    l = list(map(int, input().split()))
    tree[l[0]] = l[1:]
    for i in l[1:]:
        node_not_root.add(i)
for i in tree:
    if i not in node_not_root:
        root = i
        break

out(root)
```

前序遍历：先访问根节点，然后依次访问左子树和右子树。

中序遍历：先访问左子树，然后访问根节点，最后访问右子树。

后序遍历：先访问左子树和右子树，最后访问根节点。

宽度优先遍历：按层级从左往右遍历。

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

def preorder_traversal(root): # 前序
    if root is None:
        return ""
```

```
    return root.val + preorder_traversal(root.left) +
preorder_traversal(root.right)

def inorder_traversal(root): #中序
    if root is None:
        return ""
    return inorder_traversal(root.left) + root.val +
inorder_traversal(root.right)

def postorder_traversal(root): #后序
    if root is None:
        return ""
    return postorder_traversal(root.left) + postorder_traversal(root.right) +
root.val

def level_order_traversal(root): #按层级从左往右遍历
    if root is None:
        return []
    queue = []  # 队列用于存储要遍历的节点
    result = []  # 列表用于存储遍历结果
    queue.append(root)
    while queue:
        level = []  # 临时列表用于存储当前层的节点值
        for _ in range(len(queue)):
            node = queue.pop(0)
            level.append(node.val)
            if node.left is not None:
                queue.append(node.left)
            if node.right is not None:
                queue.append(node.right)
        result.append(level)
    return result
```

二叉搜索树定义：

- 每个节点的值都大于其左子树中任何节点的值。
- 每个节点的值都小于其右子树中任何节点的值。

```
class searchtree: # 从一串列表输入建立二叉搜索树
    def __init__(self, v=None, l=None, r=None):
        self.root = v
        self.leftchild = l
        self.rightchild = r

    def put(self, n):
        if n == self.root:
            return
        elif n > self.root and not self.rightchild:
            self.rightchild = searchtree(v=n)
        elif n > self.root and self.rightchild:
            self.rightchild.put(n)
        elif n < self.root and not self.leftchild:
```

```
            self.leftchild = searchtree(v=n)
        else:
            self.leftchild.put(n)
```

由于先序、中序和后序序列中的任一个都不能唯一确定一棵二叉树，所以对二叉树做如下处理，将二叉树的空结点用·补齐。我们把这样处理后的二叉树称为原二叉树的扩展二叉树，扩展二叉树的先序和后序序列能唯一确定其二叉树。 现给出扩展二叉树的先序序列，要求输出其中序和后序序列。

```
class binarytree:
    def __init__(self, k=None, f=None):
        self.key = k
        self.left = None
        self.right = None
        self.father = f

    def is_full(self):
        return self.left and self.right

    def inorder(self):
        if self.key == ".":
            return ""
        return self.left.inorder() + self.key + self.right.inorder()

    def postorder(self):
        if self.key == ".":
            return ""
        return self.left.postorder() + self.right.postorder() + self.key


s = input()
t = binarytree()
current_t = t
for c in s:
    if c == ".":
        while current_t.is_full():
            current_t = current_t.father
        if not current_t.left:
            current_t.left = binarytree(k=c, f=current_t)
        else:
            current_t.right = binarytree(k=c, f=current_t)
    else:
        if not current_t.key:
            current_t.key = c
        else:
            while current_t.is_full():
                current_t = current_t.father
            if not current_t.left:
                current_t.left = binarytree(k=c, f=current_t)
                current_t = current_t.left
            else:
                current_t.right = binarytree(k=c, f=current_t)
```

```
                current_t = current_t.right


print(t.inorder())
print(t.postorder())
```

并查集

```
# 宗教信仰
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]

i = 0
while True:
    i += 1
    n, m = map(int, input().split())
    if n == 0 and m == 0:
        break
    p = [i for i in range(0, n+1)]
    ans = n
    for _ in range(m):
        a, b = map(int, input().split())
        find(a) # 先压缩
        find(b)
        if p[a] != p[b]:
            ans -= 1
            p[p[a]] = p[b] # 再合并
    print(f"Case {i}: {ans}")


# 冰可乐
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]

while True:
    try:
        n, m = map(int, input().split())
    except EOFError:
        break
    p = {i: i for i in range(1, n+1)} # 字典实现
    ans = n
    for _ in range(m):
        x, y = map(int, input().split())
        find(x)
```

```
        find(y)
        if p[x] == p[y]:
            print("Yes")
        else:
            print("No")
            ans -= 1
            p[p[y]] = p[x] # 将y导入x，注意合并方法
    print(ans)
    for i in range(1, n+1):
        find(i)
    print(" ".join(map(str, sorted(list(set(p.values())))))))


# 食物链
'''
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
'''

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 = [i for i in range(3*n+1)]

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)
```

最小生成树模板

```
# 各处联通
import heapq
while True:
    try:
        n = int(input())
    except EOFError:
        break
    matrix = []
    for _ in range(n):
        matrix.append(list(map(int, input().split())))

    mst = {i: False for i in range(n)}
    heap = []
    mst[0] = True
    total_weight = 0
    for i in range(1, n):
        heapq.heappush(heap, (matrix[0][i], 0, i))
    while heap:
        weight, a, b = heapq.heappop(heap)
        if mst[a] and mst[b]:
            continue
        mst[a] = True
        mst[b] = True
        total_weight += weight
        for i in range(n):
            if not mst[i]:
                heapq.heappush(heap, (matrix[a][i], a, i))
                heapq.heappush(heap, (matrix[b][i], b, i))

    print(total_weight)


#非各处联通，理论相同，表达复杂
import heapq
n = int(input())
edges = [[] for _ in range(n)]
for _ in range(n-1):
    l = input().split()
    for i in range(1, int(l[1])+1):
        edges[ord(l[0])-65].append((int(l[2*i+1]), l[0], l[2*i]))
        edges[ord(l[2*i])-65].append((int(l[2*i+1]), l[0], l[2*i]))


mst = {chr(i+65): False for i in range(n)}
heap = []
for edge in edges[0]:
    heapq.heappush(heap, edge)
mst["A"] = True
total_weight = 0
while heap:
```

```
    weight, star1, star2 = heapq.heappop(heap)
    if mst[star1] and mst[star2]:
        continue
    mst[star1] = mst[star2] = True
    total_weight += weight
    for edge in edges[ord(star1)-65] + edges[ord(star2)-65]:
        if not mst[edge[1]] or not mst[edge[2]]:
            heapq.heappush(heap, edge)
print(total_weight)
```

无向图最大权值联通块，要定义global变量，不要一条路往下走

```
# pylint: skip-file
def dfs(i):
    global ans
    visited.add(i)
    ans += weights[i]
    for j in g[i]:
        if j not in visited:
            dfs(j)


n, m = map(int, input().split())
weights = list(map(int, input().split()))
g = {i: [] for i in range(n)}
for _ in range(m):
    a, b = map(int, input().split())
    g[a].append(b)
    g[b].append(a)

ans_final = 0
visited = set()
for i in range(n):
    if i not in visited:
        global ans
        ans = 0
        dfs(i)
        ans_final = max(ans_final, ans)

print(ans_final)
```

马遍历棋盘问题，dfs的回溯法

```
# pylint: skip-file
def dfs(x, y, l, step):
    walk = [(2, 1), (2, -1), (-2, 1), (-2, -1), (1, -2), (1, 2), (-1, -2), (-1,
2)]
    if step == n * m:
```

```
        global ans
        ans += 1
    else:
        for i in range(8):
            x1 = x + walk[i][0]
            y1 = y + walk[i][1]
            if 0 <= x1 < n and 0 <= y1 < m and l[x1][y1] == 0:
                l[x1][y1] = 1
                dfs(x1, y1, l, step+1)
                l[x1][y1] = 0  # revert the change after dfs

t = int(input())
for _ in range(t):
    n, m, x, y = map(int, input().split())
    l = [[0 for _ in range(m)] for _ in range(n)]
    l[x][y] = 1
    ans = 0
    dfs(x, y, l, 1)
    print(ans)
```

八皇后，用dfs完成任务

```
def dfs(i, queen_list):
    if i == 8:
        ans.append(int("".join(map(str, queen_list))))
        return
    for k in range(8):
        if k not in queen_list:
            flag = True
            for j in range(i):
                if i - j == abs(queen_list[j] - k):
                    flag = False
                    break
            if flag:
                dfs(i+1, queen_list+[k])


graph = [[0 for _ in range(8)] for _ in range(8)]
ans = []
dfs(0, [])
ans.sort()

for _ in range(int(input())):
    print(11111111 + ans[int(input())-1])
```

鸣人和佐助，标准bfs写法，带上了体力值

```
import heapq
```

```
m, n, t = map(int, input().split())
matrix = []
for i in range(m):
    s = input()
    l1 = []
    for c in s:
        l1.append(c)
    if "@" in l1:
        y1 = l1.index("@")
        x1 = i
    if "+" in l1:
        y2 = l1.index("+")
        x2 = i
    matrix.append(list(l1))

heap = [[0, x1, y1, t]]
visited = set()
visited.add((x1, y1, t))
heapq.heapify(heap)
dire = [[0, 1], [0, -1], [1, 0], [-1, 0]]
while heap:
    step, x, y, t_n = heapq.heappop(heap)
    for i in range(4):
        x3 = x + dire[i][0]
        y3 = y + dire[i][1]
        if 0 <= x3 < m and 0 <= y3 < n:
            if x3 == x2 and y3 == y2:
                print(step+1)
                exit()
            elif matrix[x3][y3] == "*" and (x3, y3, t_n) not in visited:
                heapq.heappush(heap, [step+1, x3, y3, t_n])
                visited.add((x3, y3, t_n))
            elif t_n >= 1 and (x3, y3, t_n-1) not in visited:
                heapq.heappush(heap, [step+1, x3, y3, t_n-1])
                visited.add((x3, y3, t_n-1))

print(-1)
```

走山路，标准bfs，带权重时visited应该在取出节点时添加

```
import heapq
m, n, p = map(int, input().split())
graph = []
for _ in range(m):
    graph.append(input().split())
for _ in range(p):
    x1, y1, x2, y2 = map(int, input().split())
    if graph[x1][y1] == "#" or graph[x2][y2] == "#":
        print("NO")
        continue

    dire = [[0, -1], [0, 1], [-1, 0], [1, 0]]
```

```
    heap = [[0, x1, y1]]
    visited = set()
    ans = "NO"
    while heap:
        power, x, y = heapq.heappop(heap)
        visited.add((x, y))
        if x == x2 and y == y2:
            ans = power
            break
        for i in range(4):
            x3 = x + dire[i][0]
            y3 = y + dire[i][1]
            if 0 <= x3 < m and 0 <= y3 < n and (x3, y3) not in visited and
graph[x3][y3] != "#":
                heapq.heappush(heap, [power+abs(int(graph[x3][y3])-int(graph[x]
[y])), x3, y3])

    print(ans)
```

带权重无向图最短路径，非矩阵连接建图方式

```
import heapq
p = int(input())
g = {}
for _ in range(p):
    g[input()] = []
q = int(input())

for _ in range(q):
    a, b, d = input().split()
    g[a].append([b, int(d)])
    g[b].append([a, int(d)])
r = int(input())

for _ in range(r):
    a, b = input().split()
    heap = [(0, a, [a])]
    visited = set()
    while heap:
        distance, position, path = heapq.heappop(heap)
        if position == b:
            print("->".join(path))
            break
        if position not in visited:
            visited.add(position)
            for p, d in g[position]:
                heapq.heappush(heap, (distance+d, p, path+[f"({d})"]+[p]))
```

无向图是否联通和有环的判断方法

```
# pylint: skip-file
import heapq
n, m = map(int, input().split())
graph = {i: [] for i in range(n)}
for _ in range(m):
    x, y = map(int, input().split())
    graph[x].append(y)
    graph[y].append(x)

visited = set()
visited.add(0)
heap = [[0, 0]]
while heap:
    step, p = heapq.heappop(heap)
    for p_next in graph[p]:
        if p_next not in visited:
            visited.add(p_next)
            heapq.heappush(heap, [step+1, p_next])

if len(visited) == n:
    print("connected:yes")
else:
    print("connected:no")


def dfs(i):
    global ans
    for p in graph[i]:
        if p in visited:
            ans = "yes"
            return
        graph[i].remove(p)
        graph[p].remove(i)
        visited.add(p)
        dfs(p)


global ans
ans = "no"
for i in range(n):
    visited = set()
    visited.add(i)
    dfs(i)
print(f"loop:{ans}")
```

有向图是否有环的判断方法，利用入度

```
def dfs(p):
    vis[p] = True
```

```
    for q in graph[p]:
        in_degree[q] -= 1
        if in_degree[q] == 0:
            dfs(q)

for _ in range(int(input())):
    n, m = map(int, input().split())
    graph = {i: [] for i in range(1, n+1)}
    in_degree = [0] * (n+1)
    vis = [False] * (n+1)
    for _ in range(m):
        x, y = map(int, input().split())
        graph[x].append(y)
        in_degree[y] += 1

    for k in range(1, n+1):
        if in_degree[k] == 0 and not vis[k]:
            dfs(k)

    if vis[1:] == [True] * n:
        print("No")
    else:
        print("Yes")
```

有向图强连通分量的判断方法

```
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]

"""
```

拓扑排序，任务管理

```
def topological_sort(graph):
    in_degree = {u: 0 for u in graph}  # 初始化所有顶点的入度为0
    for u in graph:  # 计算每个顶点的入度
        for v in graph[u]:
            in_degree[v] += 1

    queue = []  # 用一个队列来存储所有入度为0的顶点
    for u in in_degree:  # 将所有入度为0的顶点添加到队列中
        if in_degree[u] == 0:
            queue.append(u)

    result = []  # 存储拓扑排序的结果
    while queue:  # 当队列非空时
        u = queue.pop(0)  # 从队列中取出一个顶点
        result.append(u)  # 将这个顶点添加到结果中
        for v in graph[u]:  # 减少所有从这个顶点出发的边的目标顶点的入度
            in_degree[v] -= 1
            if in_degree[v] == 0:  # 如果这个顶点的入度变为0，就将它添加到队列中
                queue.append(v)

    if len(result) == len(graph):  # 如果结果中的顶点数等于图中的顶点数，说明图是一个DAG
        result.reverse()
```

```
        return result  # 返回拓扑排序的结果
    else:  # 图中存在环
        return None

# 项目管理任务
tasks = {
    '设计蓝图': [],
    '购买材料': ['设计蓝图'],
    '建造地基': ['购买材料'],
    '建造墙壁': ['建造地基'],
    '安装电线': ['建造墙壁'],
    '安装管道': ['建造墙壁'],
    '建造屋顶': ['安装电线', '安装管道'],
    '粉刷墙壁': ['建造屋顶'],
    '安装门窗': ['粉刷墙壁'],
    '最后的清理工作': ['安装门窗'],
}

order = topological_sort(tasks)
if order is None:
    print("任务存在循环依赖，无法完成")
else:
    print("任务的执行顺序为：")
    for task in order:
        print(task)
```

单调栈

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)。

```
n = int(input())
l = list(map(int, input().split()))
ans = [-1 for _ in range(n)]
stack = []

for i in range(n):
    while stack and l[stack[-1]] < l[i]:
        ans[stack.pop()] = i + 1
    stack.append(i)

while stack:
    ans[stack.pop()] = 0

print(*ans) # 更好的输出方法
```

单调栈应用

```python
# 接雨水，计算雨水思路
def trap(heights):
    stack = []
    water = 0

    for i, height in enumerate(heights):
        while stack and heights[stack[-1]] < height:
            top = stack.pop()
            if not stack:
                break
            distance = i - stack[-1] - 1
            bounded_height = min(height, heights[stack[-1]]) - heights[top]
            water += distance * bounded_height
        stack.append(i)

    return water

n = int(input())
heights = list(map(int, input().split()))
print(trap(heights))


# 给定一个十进制正整数n(0 < n < 1000000000)，每个数位上数字均不为0。n的位数为m。现在从m位中删除k位(0<k < m)，求生成的新整数最小为多少？思路问题
def remove_k_digits(n, k):
    num = list(str(n))
    stack = []
    for digit in num:
        while k > 0 and stack and stack[-1] > digit:
            stack.pop()
            k -= 1
        stack.append(digit)
    while k > 0:
        stack.pop()
        k -= 1
    return int(''.join(stack))

t = int(input())
for _ in range(t):
    n, k = map(int, input().split())
    print(remove_k_digits(n, k))
```

走山路，Dijkstra算法，用于在图中找到最短路径的算法

```python
import heapq
m, n, p = map(int, input().split())
matrix = []
for _ in range(m):
    l = []
    for a in input().split():
        if a != "#":
            l.append(int(a))
```

```
        else:
            l.append(a)
    matrix.append(list(l))

for _ in range(p):
    x1, y1, x2, y2 = map(int, input().split())
    ans = "NO"
    if matrix[x1][y1] == "#" or matrix[x2][y2] == "#":
        print(ans)
    else:
        dist = [[float('inf')] * n for _ in range(m)]
        dist[x1][y1] = 0
        heap = [(0, x1, y1)]
        dire = [[0, 1], [0, -1], [1, 0], [-1, 0]]
        while heap:
            power, x, y = heapq.heappop(heap)
            if x == x2 and y == y2:
                ans = power
                break
            for i in range(4):
                x3 = x + dire[i][0]
                y3 = y + dire[i][1]
                if 0 <= x3 < m and 0 <= y3 < n and matrix[x3][y3] != "#" \
                and dist[x3][y3] > power + abs(matrix[x3][y3] - matrix[x][y]):
                    dist[x3][y3] = power + abs(matrix[x3][y3] - matrix[x][y])
                    heapq.heappush(heap, (dist[x3][y3], x3, y3))

        print(ans)
```

```
# Trie数据结构，使用它来检查一组输入的字符串（在这个例子中是电话号码）是否有任何一个字符串是另一个字符串的前缀。
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_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.is_word = True

    def same(self, prefix):
        node = self.root
```

```
        flag = True
        for char in prefix:
            if char in node.children:
                node = node.children[char]
            elif not node.children and node.is_word:
                break
            else:
                flag = False
        return flag


m = int(input())
for _ in range(m):
    n = int(input())
    t = Trie()
    numbers = []
    ans = "YES"
    for _ in range(n):
        numbers.append(input())
    for number in numbers:
        if t.same(number):
            ans = "NO"
            break
        else:
            t.insert(number)
    print(ans)
```

```
# 归并排序，递归
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0

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

    result.extend(left[i:])
```

```
    result.extend(right[j:])
    return result
```

```
# 波兰表达式是一种把运算符前置的算术表达式，例如普通的表达式2 + 3的波兰表示法为+ 2 3。波兰表达式的优点是运算符之间不必有优先级关系，也不必用括号改变运算次序，例如(2 + 3) * 4的波兰表示法为* + 2 3 4。本题求解波兰表达式的值，其中运算符包括+ - * /四个。
s = input().split()
ex = ["+", "-", "*", "/"]
while len(s) > 1:
    for i in range(len(s)-1, 1, -1):
        if s[i-2] in ex:
            a = eval(f"{s[i-1]}{s[i-2]}{s[i]}")
            s.pop(i-2)
            s.pop(i-2)
            s.pop(i-2)
            s.insert(i-2, a)
            break

print(f"{s[0]:.6f}")


# 根据后序表达式建立表达式树
e_postorder = input()
s = []
for c in e_postorder:
    if not c.isupper(): # 不是算符
        s.append(binarytree(c))
    else:
        t = binarytree(c)
        t.right = s.pop()
        t.left = s.pop()
        s.append(t)


# 波兰表达式与逆波兰表达式可以直接反转
```

中序表达式转后序表达式

```
def compare(a, b):
    if a in ["+", "-"]:
        return True
    elif a in ["*", "/"] and b in ["*", "/"]:
        return True
    else:
        return False
```

```python
n = int(input())
for _ in range(n):
    ex = input()
    ex_with_spaces = ""
    for char in ex:
        if char in ['+', '-', '*', '/', '(', ')']:
            ex_with_spaces += f" {char} "
        else:
            ex_with_spaces += char
    ex_in = ex_with_spaces.split()
    s = []
    ex_out = []
    c = ["+", "-", "*", "/"]

    for e in ex_in:
        if e == "(":
            s.append(e)
        elif e == ")":
            while s[-1] != "(":
                ex_out.append(s.pop())
            s.pop()
        elif e in c:
            while s != [] and s[-1] != "("  and compare(e, s[-1]):
                ex_out.append(s.pop())
            s.append(e)
        else:
            ex_out.append(e)

    while s != []:
        ex_out.append(s.pop())

    str_f = " ".join(ex_out)
    print(str_f)
```