注意： 1、质数筛法：欧拉筛

更注意：对于 n 大于 10^7 的数据，只需算出根号 n + 1 以内的质数，然后用 n 去除，均无法整除即为质数

2、善用栈的后进后出模拟中间抽出去一些元素/暂存元素

3、差分数组：核心即，只需要改两端的值，省时

考虑数组 $a = [1, 3, 3, 5, 8]$，对其中的相邻元素两两作差（右边减左边），得到数组 $[2, 0, 2, 3]$。然后在开头补上 $a[0]$，得到差分数组

$$d = [1, 2, 0, 2, 3]$$

这有什么用呢？如果从左到右累加 d 中的元素，我们就「还原」回了 a 数组 $[1, 3, 3, 5, 8]$。这类似**求导与积分**。

这又有什么用呢？现在把连续子数组 $a[1], a[2], a[3]$ 都加上 10，得到 $a' = [1, 13, 13, 15, 8]$。再次两两作差，并在开头补上 $a'[0]$，得到差分数组

$$d' = [1, 12, 0, 2, -7]$$

对比 d 和 d'，可以发现只有 $d[1]$ 和 $d[4]$ 变化了，这意味着对 a 中连续子数组的操作，可以转变成对差分数组 d 中**两个数**的操作。

定义和性质

对于数组 a，定义其**差分数组**（difference array）为

$$d[i] = \begin{cases} a[0], & i = 0 \\ a[i] - a[i-1], & i \geq 1 \end{cases}$$

性质 1：从左到右累加 d 中的元素，可以得到数组 a。

性质 2：如下两个操作是等价的。

- 把 a 的子数组 $a[i], a[i+1], \ldots, a[j]$ 都加上 x。
- 把 $d[i]$ 增加 x，把 $d[j+1]$ 减少 x。

利用性质 2，我们只需要 $O(1)$ 的时间就可以完成对 a 的子数组的操作。最后利用性质 1 从差分数组复原出数组

4、贝尔曼松弛算法：增益环（好像不考？）/k 次中转的最小价格（几次松弛就是最多几段路径）

5、水淹七军，注意后放的水币先放的水的更高的情形：即，visited 列表要慎用

6、骑士周游，优先选下一个节点可去位置少的

7、边不太多的时候迪杰斯特拉也挺快的，不一定要用松弛/全节点最小距离做

8、很多节点找边的时候可以很多“桶”将相邻节点（具有相同特征）放在一起，只需遍历节点一次（如，单词梯那道题）

9、伪满二叉树：右节点和自己同级，左节点为下一级，注意观察变形之后的树和原来的树的关系

10、二叉树充分利用左右递归关系（如统计节点数目/前序/后序/中序等等）；以及，左右子树大小关系等等根据现有节点进行回溯：即，通过给出的特征，判断当前节点是上一节点的左还是右，不断回溯直到根节点

树的结构：

- 根节点 `(1,1)`
- 左子树：`(a + b, b)`
- 右子树：`(a, a + b)`

对于 **给定的** `(i, j)`，我们要找到它在二叉树中的位置，并计算 **左转和右转的次数**：

- **如果** `i > j`，意味着 `(i, j)` 来自 **左子树**，所以 **左转**。
- **如果** `i < j`，意味着 `(i, j)` 来自 **右子树**，所以 **右转**。
- **如果** `i == j`，那么 `i, j` 必须是 `(1,1)`，但由于题目保证 `(i,j)` 是合法的，我们不需要考虑这种情况。

因此，我们可以不断递归：

- 如果 `(i > j)`，它的 **父节点** 是 `(i - j, j)`，左转计数 `+1`。
- 如果 `(i < j)`，它的 **父节点** 是 `(i, j - i)`，右转计数 `+1`。

最终 `(1,1)` 会是终点。

11、千万注意后序遍历是先左后右再根节点

12、双链表合并

13、在寻找最大（最长路径的时候一定注意负数权值的边）

14、二叉搜索树构建，考虑递归（即，比当前根节点小则考虑左节点，为空则插入，不为空则递归下一层），二叉搜索树的“落叶”在“重构”等价于按落叶顺序的倒序直接建树

15、公共祖先（根节点）：dfs，二者路径中最后一个相同的节点

16、树结合前缀和（找特定的区间和特别好用），dfs，用一个字典记录，回溯时对应的 value 减一

17、二叉搜索树验证：中序遍历形成的列表是单调的

18、整理文件那题见最后，有代码

19、中序过程中的操作

```python
class Solution:
    def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
        def dfs(root):
            if not root: return
            dfs(root.left)
            if self.k == 0: return
            self.k -= 1
            if self.k == 0: self.res = root.val
            dfs(root.right)

        self.k = k
        dfs(root)
        return self.res
```

20、位运算

Python位运算

在Python中，使用位运算是一种高效的操作方式，特别是在需要处理整数时。位运算直接对整数的二进制表示进行操作，这使得它们在执行如掩码、切换位、计算位标志等任务时非常有用。Python提供了以下几种位运算符：

1. **按位与（AND）**：`&`
2. **按位或（OR）**：`|`
3. **按位异或（XOR）**：`^`
4. **按位取反（NOT）**：`~`
5. **左移**：`<<`
6. **右移**：`>>`

3. 按位异或（XOR）

Python Copy code

```python
a = 5  # 二进制表示为 0101
b = 3  # 二进制表示为 0011

result = a ^ b  # 结果为 6，二进制表示为 0110
print(result)  # 输出：6
```

4. 按位取反（NOT）

Python Copy code

```python
a = 5  # 二进制表示为 000...00101（取决于位数，这里以8位为例）

result = ~a  # 结果为 -6，二进制表示为 111...11010（同样以8位为例）
print(result)  # 输出：-6
```

注意：在Python中，负数是通过补码形式存储的。例如，8位的整数范围是-128到127，所以 `~5` 实际上是计算了 `-6` 的补码。

21、跳跃游戏，贪心，还是在相同数量的桥的情况下尽可能走得远

22、找到一个只含 01 的数，是 N 的倍数：

考虑到我们的结果至多是100位数，我们需要找到一个能被n整除的数字，并且全是0、1，实际上所有位上都是0、1的数字是可以构成一颗二叉树的，我们从1作为根节点，左右儿子分别进行*10和*10+1操作——并且我们不关心数字实际的大小，而关心这个数mod n的结果，由于mod运算的结合、分配律，所以我们可以进行vis去重，这大大降低了搜索的成本，于是我们直接用bfs就可以了（虽然题目不要求最短这个性质，但即便如此相比之下bfs也比dfs更高效）

对余数进行BFS（广度优先搜索）。思路是从最小的满足条件的数开始搜索，即1，然后通过不断添加0或1在数的末尾来生成新的数字，直到找到一个数字既是n的倍数又只包含数字0和1。

由于直接操作数字可能会很快超出整数范围，特别是当n很大时，我们可以在BFS过程中仅保存数字的模n值以及该数字的十进制表示。每次从队列中取出一个元素，计算加0或加1后的模n值，如果新模n值为0，则找到了解；否则，如果这个模n值是首次访问，则将其加入队列继续搜索。

```python
# 钟明衡 物理学院
def bfs(n):
    l = [0]
    s, e = 0, 1
    while s != e:
        for i in range(s, e):
            for j in (0, 1):
                x = l[i]*10+j
                if x:
                    if x % n:
                        l.append(x)
                    else:
                        return str(x)
        s, e = e, len(l)
    return ''


while (n := int(input())):
    c = 0
    while (n+1) % 2:
        n //= 2
        c += 1
    print(bfs(n)+'0'*c)
```

23、最大矩形：核心想法——只在“下山”的时候讨论上一个“山峰”的最大矩形，单调栈记录“上山”过程

```python
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        heights.append(-1)  # 最后大火收汁，用 -1 把栈清空（-1 可以改成 0）
        st = [-1]  # 在栈中只有一个数的时候，栈顶的「下面那个数」是 -1，对应 left[i] = -1 的情况
        ans = 0
        for right, h in enumerate(heights):
            while len(st) > 1 and h <= heights[st[-1]]:
                i = st.pop()  # 矩形的高（的下标）
                left = st[-1]  # 栈顶下面那个数就是 left
                ans = max(ans, heights[i] * (right - left - 1))
            st.append(right)
        return ans
```

24、最大子矩阵：多维压缩为一维，然后 kanada

```python
def kadane(arr):
    # max_ending_here 用于追踪到当前元素为止包含当前元素的最大子数组和。
    # max_so_far 用于存储迄今为止遇到的最大子数组和。
    max_end_here = max_so_far = arr[0]
    for x in arr[1:]:
            # 对于每个新元素，我们决定是开始一个新的子数组（仅包含当前元素 x），
        # 还是将当前元素添加到现有的子数组中。这一步是 Kadane 算法的核心。
        max_end_here = max(x, max_end_here + x)
        max_so_far = max(max_so_far, max_end_here)
    return max_so_far
```

25、dp 找最长公共子串的基本功，拓展到找某序列的最长回文子序列，等价于找该序列与反转序列的最长公共子序列

问题是需要找到将给定字符串转换为回文所需的最少插入字符数。回文是指从左到右和从右到左读都相同的字符串。可以通过动态规划的方法来解决这个问题，具体思路是计算原字符串与其反转字符串的最长公共子序列（LCS），然后用原字符串的长度减去LCS的长度，得到所需插入的最少字符数。

方法思路

1. **问题分析**：回文的结构具有对称性，利用动态规划来找到原字符串与其反转字符串的最长公共子序列。这个子序列的长度即为原字符串中最长回文子序列的长度。

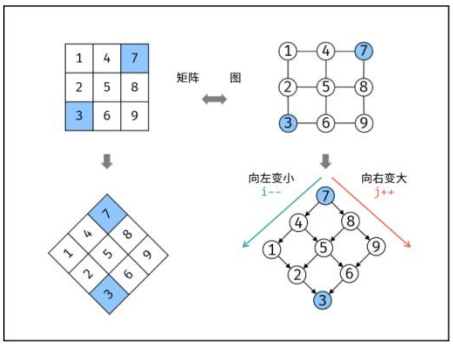
```
n = int(input())
s = input().strip()
t = s[::-1]

dp = [0] * (n + 1)

for i in range(n):
    current_char = s[i]
    prev_prev = 0
    for j in range(1, n + 1):
        current = dp[j]
        if current_char == t[j - 1]:
            dp[j] = prev_prev + 1
        else:
            if dp[j - 1] > dp[j]:
                dp[j] = dp[j - 1]
        prev_prev = current

print(n - dp[n])
```

26、直接找算法想不到的话，不妨按题目描述模拟一下，如，合法的出栈序列

27、

思路：输入时将各条序列sort，先只考虑两条序列，（0,0）一定最小，用heapq存储，下一步最小一定在（i+1, j）和（i, j+1）之间，以此类推找到最小的n个存为序列seq，再将seq与第三条序列重复操作，以此类推。注意m=1的情况。

如下图所示，我们将矩阵逆时针旋转 45° ，并将其转化为图形式，发现其类似于 **二叉搜索树** ，即对于每个元素，其左分支元素更小、右分支元素更大。因此，通过从 “根节点” 开始搜索，遇到比 `target` 大的元素就向左，反之向右，即可找到目标值 `target` 。



模板：

经典Prim算法

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


while True:
    try:
        n = int(input())
    except:
        break
    mat, cur = [], 0
    for i in range(n):
        mat.append(list(map(int, input().split())))
    d, v, q, cnt = [100000 for i in range(n)], set(), [], 0
    d[0] = 0
    heappush(q, (d[0], 0))
    while q:
        x, y = heappop(q)
        if y in v:
            continue
        v.add(y)
        cnt += d[y]
        for i in range(n):
            if d[i] > mat[y][i]:
                d[i] = mat[y][i]
                heappush(q, (d[i], i))
    print(cnt)
```

Dp 树

```
class Solution:
    def rob(self, root: Optional[TreeNode]) -> int:
        def dfs(node: Optional[TreeNode]) -> Tuple[int, int]:
            if node is None:  # 递归边界
                return 0, 0  # 没有节点，怎么选都是 0
            l_rob, l_not_rob = dfs(node.left)  # 递归左子树
            r_rob, r_not_rob = dfs(node.right)  # 递归右子树
            rob = l_not_rob + r_not_rob + node.val  # 选
            not_rob = max(l_rob, l_not_rob) + max(r_rob, r_not_rob)  # 不选
            return rob, not_rob
        return max(dfs(root))  # 根节点选或不选的最大值
```

二分查找：

```python
def max_cable_length(cables, K):
    # 转换为整数（厘米）
    cables_cm = [int(round(c * 100)) for c in cables]
    low, high = 1, max(cables_cm) + 1  # 长度至少为1cm

    result = 0
    while low < high:
        mid = (low + high) // 2
        count = sum(cable // mid for cable in cables_cm)

        if count >= K:
            result = mid  # 尝试更长
            low = mid + 1
        else:
            high = mid

    # 输出结果以米为单位，并保留两位小数
    return f"{result / 100:.2f}" if result > 0 else "0.00"

# 输入读取部分
def main():
    N, K = map(int, input().split())
    cables = [float(input()) for _ in range(N)]
    print(max_cable_length(cables, K))


main()
```

拓扑排序：

```python
class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List[int]:
        g = [[] for _ in range(numCourses)]
        indegree = [0] * numCourses
        for x, y in prerequisites:
            indegree[x] += 1
            g[y].append(x)
        q = deque([i for i, x in enumerate(indegree) if x == 0])
        ans = []
        while q:
            u = q.popleft()
            ans.append(u)
            for v in g[u]:
                indegree[v] -= 1
                if indegree[v] == 0:
                    q.append(v)
        return ans if len(ans) == numCourses else []
```

松弛（好像不考）

```
def main():
    # 读取第一行：N, M, S, V
    data = sys.stdin.read().strip().split()
    N, M = map(int, data[:2])
    S = int(data[2])        # 起始货币编号
    V = float(data[3])      # 起始金额

    # 解析后续每个兑换点的信息
    edges = []
    idx = 4
    for _ in range(M):
        A = int(data[idx]);     B = int(data[idx+1])
        R_ab = float(data[idx+2]);  C_ab = float(data[idx+3])
        R_ba = float(data[idx+4]);  C_ba = float(data[idx+5])
        idx += 6

        # 从 A->B 的边
        edges.append((A, B, R_ab, C_ab))
        # 从 B->A 的边
        edges.append((B, A, R_ba, C_ba))

    # best[i] 表示最终能在货币 i 上得到的最大金额
    best = [0.0] * (N + 1)
    best[S] = V

    # Bellman-Ford 核心：最多做 N 轮松弛
    for iteration in range(1, N + 1):
        updated = False
        for u, v, rate, fee in edges:
            if best[u] > fee:
                x = (best[u] - fee) * rate
                if x > best[v] + 1e-12:  # 加一点 eps 防止浮点误差
                    best[v] = x
                    updated = True
```

```
                    # 如果再次更新回了起始货币 S，且金额大于初始值，立刻输出 YES
                    if v == S and best[S] > V:
                        print("YES")
                        return

        # 如果第 N 轮仍有更新，说明存在正增益回路
        if iteration == N and updated:
            print("YES")
            return

        # 没有任何更新，可提前结束
        if not updated:
            break

    # 没发现任何增益方案
    print("NO")


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

并查集：

```
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))  # 获取不同集合的根节点
        ans = sorted(unique_parents)  # 输出有冰阔落的杯子编号
        print(len(ans))
        print(*ans)

    except EOFError:
        break
```

前缀树：

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

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

    def insert(self, number):
        node = self.root
        for digit in number:
            if digit not in node.children:
                node.children[digit] = TrieNode()
            node = node.children[digit]
            # 如果当前节点已经是某个电话号码的结尾，则说明存在前缀冲突
            if node.is_end_of_number:
                return False
        # 插入完成后，标记为完整电话号码
        node.is_end_of_number = True
        # 如果当前节点还有子节点，说明有其他号码以它为前缀
        return len(node.children) == 0

    def is_consistent(self, numbers):
        # 按长度从短到长排序，确保短号码先被检查
        numbers.sort(key=len)
        for number in numbers:
            if not self.insert(number):
                return False
        return True

def main():
    import sys
    input = sys.stdin.read
    data = input().splitlines()

    t = int(data[0])  # 测试样例数量
```

```
def main():
    import sys
    input = sys.stdin.read
    data = input().splitlines()

    t = int(data[0])  # 测试样例数量
    index = 1
    results = []

    for _ in range(t):
        n = int(data[index])  # 当前测试样例的电话号码数量
        index += 1
        numbers = data[index:index + n]
        index += n

        trie = Trie()
        if trie.is_consistent(numbers):
            results.append("YES")
        else:
            results.append("NO")

    print("\n".join(results))

# 调用主函数
if __name__ == "__main__":
    main()
```

哈夫曼编码（唯一）：

```
import heapq

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

def huffman_code(root , str , step) :
    if root.val == str and root.left == None and root.right == None :
        return step
    elif root.left == None and root.right == None :
        return ""
    else :
        return(huffman_code(root.left , str , step + "0") +
huffman_code(root.right , str , step + "1"))

n = int(input())
nodes = []
for _ in range(n) :
    val , weight = input().split()
    weight = int(weight)
    heapq.heappush(nodes , (weight , val , TreeNode(val , weight)))

while len(nodes) > 1 :
    weight1 , val1 , node1 = heapq.heappop(nodes)
    weight2 , val2 , node2 = heapq.heappop(nodes)
    node = TreeNode(min(val1 , val2) , weight1 + weight2)
    node.left = node1
    node.right = node2
    heapq.heappush(nodes , (weight1 + weight2 , node.val , node))

root = nodes[0][2]

#print(root.val)
#print(root.right.weight)
#print(root.right.left.val)

while True :
    try :
        a = input()
```

```
        if a[0] not in ["1" , "2" , "3" , "4" , "5" , "6" , "7" , "8" , "9"] :
            answer = ""
            for i in range(len(a)) :
                answer += huffman_code(root , a[i] , "")
            print(answer)
        else :
            ind = 0
            now = root
            answer = ""
            while ind < len(a) :
                if now.left == None and now.right == None :
                    answer += str(now.val)
                    now = root
                if a[ind] == "0" :
                    now = now.left
                else :
                    now = now.right
                ind += 1
            answer += str(now.val)
            print(answer)
    except :
        break
```

前缀和：

```python
class Solution:
    def pathSum(self, root: Optional[TreeNode], targetSum: int) -> int:
        ans = 0
        cnt = defaultdict(int)
        cnt[0] = 1

        def dfs(node: Optional[TreeNode], s: int) -> None:
            if node is None:
                return

            nonlocal ans
            s += node.val
            # 把 node 当作路径的终点，统计有多少个起点
            ans += cnt[s - targetSum]

            cnt[s] += 1
            dfs(node.left, s)
            dfs(node.right, s)
            cnt[s] -= 1  # 恢复现场

        dfs(root, 0)
        return ans
```

归并排序：

```
def hebing(lista, listb):
    answerlist = []
    answer = 0
    i = 0
    j = 0
    while True:
        if i == len(lista) and len(listb) == j:
            break
        elif j == len(listb):
            answerlist.append(lista[i])
            i += 1
        elif i == len(lista):
            answerlist.append(listb[j])
            j += 1
            answer += i
        elif lista[i] < listb[j]:
            answerlist.append(lista[i])
            i += 1
        else:
            answer += i
            answerlist.append(listb[j])
            j += 1
    return (answerlist, answer)


def bingchaji(list0):
    if len(list0) >= 2:
        a, ana = bingchaji(list0[:(len(list0) // 2)])
        b, anb = bingchaji(list0[(len(list0) // 2):len(list0)])
        new, ans = hebing(a, b)
        return new, ana + anb + ans
    else:
        return (list0, 0)



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

_, answer = bingchaji(nums)print(answer)
```

答案代码：

```
import sys

def merge_sort(arr, temp, left, right):
    if left >= right:
        return 0
    mid = (left + right) // 2
    inv_count = merge_sort(arr, temp, left, mid) + merge_sort(arr, temp,
mid + 1, right)

    # 归并过程，同时计算逆序数
    i, j, k = left, mid + 1, left
    while i <= mid and j <= right:
        if arr[i] >= arr[j]:  # 注意这里是 >=，保证稳定性
            temp[k] = arr[i]
            i += 1
        else:
            temp[k] = arr[j]
            inv_count += (mid - i + 1)  # 统计逆序对
            j += 1
        k += 1

    while i <= mid:
        temp[k] = arr[i]
        i += 1
        k += 1
    while j <= right:
        temp[k] = arr[j]
        j += 1
        k += 1

    # 拷贝回原数组
    for i in range(left, right + 1):
        arr[i] = temp[i]

    return inv_count

if __name__ == "__main__":
    n = int(sys.stdin.readline().strip())
    arr = [int(sys.stdin.readline().strip()) for _ in range(n)]
    temp = [0] * n
    result = merge_sort(arr, temp, 0, n - 1)
    print(result)
```

解数独：

```
class Solution:
    def solveSudoku(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        self.backtracking(board)

    def backtracking(self, board: List[List[str]]) -> bool:
        # 若有解，返回 True；若无解，返回 False
        for i in range(len(board)): # 遍历行
            for j in range(len(board[0])):  # 遍历列
                # 若空格内已有数字，跳过
                if board[i][j] != '.': continue
                for k in range(1, 10):
                    if self.is_valid(i, j, k, board):
                        board[i][j] = str(k)
                        if self.backtracking(board): return True
                        board[i][j] = '.'
                # 若数字 1-9 都不能成功填入空格，返回 False 无解
                return False
        return True # 有解

    def is_valid(self, row: int, col: int, val: int, board:
List[List[str]]) -> bool:
        # 判断同一行是否冲突
        for i in range(9):
            if board[row][i] == str(val):
                return False
        # 判断同一列是否冲突
        for j in range(9):
            if board[j][col] == str(val):
                return False
        # 判断同一九宫格是否有冲突
        start_row = (row // 3) * 3
        start_col = (col // 3) * 3
        for i in range(start_row, start_row + 3):
            for j in range(start_col, start_col + 3):
                if board[i][j] == str(val):
                    return False
        return True
```

练习 T20576: printExp（逆波兰表达式建树）
http://cs101.openjudge.cn/practice/20576/

输出中缀表达式（去除不必要的括号）

输入

一个字串

输出

一个字串

样例输入

```
( not ( True or False ) ) and ( False or True and True )
```

样例输出

```
not ( True or False ) and ( False or True and True )
```

这三个操作符：`not`：优先级最高，`and`：其次，`or`：优先级最低。

`printTree`函数是一个递归函数，接收一个`BinaryTree`对象作为参数，然后根据树的结构和节点的值生成一个字符串列表。

函数的工作方式如下：

1. 首先，检查树的根节点的值。根据值的不同，函数会执行不同的操作。
2. 如果根节点的值为"or"，函数会递归地调用自身来处理左子树和右子树，然后将结果合并，并在两个结果之间插入"or"。
3. 如果根节点的值为"not"，函数会递归地调用自身来处理左子树。如果左子树的根节点的值不是"True"或"False"，则会在左子树的结果周围添加括号。
4. 如果根节点的值为"and"，函数会递归地调用自身来处理左子树和右子树。如果左子树或右子树的根节点的值为"or"，则会在相应子树的结果周围添加括号。
5. 如果根节点的值为"True"或"False"，函数会直接返回一个包含该值的列表。
6. 最后，函数会将生成的字符串列表合并为一个字符串，并返回。

```
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

    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}

    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

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()
```

整理文件：

```
# 夏天明，元培学院
from sys import exit

class dir:
    def __init__(self, dname):
        self.name = dname
        self.dirs = []
        self.files = []
```

```
    def getGraph(self):
        g = [self.name]
        for d in self.dirs:
            subg = d.getGraph()
            g.extend(["|   " + s for s in subg])
        for f in sorted(self.files):
            g.append(f)
        return g

n = 0
while True:
    n += 1
    stack = [dir("ROOT")]
    while (s := input()) != "*":
        if s == "#": exit(0)
        if s[0] == 'f':
            stack[-1].files.append(s)
        elif s[0] == 'd':
            stack.append(dir(s))
            stack[-2].dirs.append(stack[-1])
        else:
            stack.pop()
    print(f"DATA SET {n}:")
    print(*stack[0].getGraph(), sep='\n')
    print()
```

2048：

```
# pylint: skip-file
def move_left(board):
    m, n = len(board), len(board[0])
    new_board = []
    for row in board:
        # 压缩：去除 0，保留非 0 数值
        filtered = [x for x in row if x != 0]
        merged = []
        skip = False
        i = 0
        while i < len(filtered):
            if i + 1 < len(filtered) and filtered[i] == filtered[i + 1]:
                # 合并，注意每行内只允许合并一次
                merged.append(filtered[i] * 2)
```

```
                    i += 2
                else:
                    merged.append(filtered[i])
                    i += 1
            # 补 0 到尾部
            merged += [0] * (n - len(merged))
            new_board.append(merged)
        return new_board


def reverse_board(board):
    # 将每一行反转（用于模拟向右移动）
    return [row[::-1] for row in board]


def transpose(board):
    return [list(x) for x in zip(*board)]


def move_right(board):
    # 向右移动：先反转->左移->再反转
    reversed_board = reverse_board(board)
    moved = move_left(reversed_board)
    return reverse_board(moved)


def move_up(board):
    # 向上移动：转置->左移->再转置
    trans = transpose(board)
    moved = move_left(trans)
    return transpose(moved)


def move_down(board):
    # 向下移动：转置->右移->再转置
    trans = transpose(board)
    moved = move_right(trans)
    return transpose(moved)


def get_max_tile(board):
    return max(max(row) for row in board)
```

```
def dfs(board, moves_left):
    global answer
    current_max = get_max_tile(board)
    answer = max(answer, current_max)
    if moves_left == 0:
        return

    # 对四个方向进行移动
    for move_func in [move_left, move_right, move_up, move_down]:
        new_board = move_func(board)
        # 若该操作没有产生变化，则无需再搜索
        if new_board == board:
            continue
        dfs(new_board, moves_left - 1)


if __name__ == '__main__':
    m, n, p = map(int, input().split())
    board = []
    for i in range(m):
        row = list(map(int, input().split()))
        board.append(row)

    answer = 0
    dfs(board, p)
    print(answer)
```

并查集

归并排序

4.2 强连通单元（SCCs）

Tarjan 算法

Kosaraju 算法

Bellman-Ford 算法

TO1860: Currency Exchange

算法思路

1. **节点**：货币种类编号 1...N。
2. **边**：每个兑换点 i（描述为 A,B,RAB,CAB,RBA,CBA）对应两条有向边：
   - 从 A 到 B，如果当前在 A 手上有 x 单位，可以兑换到

     $x\prime = (x - CAB) \times RAB$

     但仅当 x>CAB 时才可能兑换，不然兑换后金额为负。
   - 从 B 到 A，同理：

     $x\prime = (x - CBA) \times RBA$
3. **状态**：用数组 best[1..N] 记录在每个货币上能够达到的"最大金额"。初始化：

   $best[S] = V, best[i \neq S] = 0.$
4. **松弛操作**：对每条有向边 (u→v) 重复下面操作：

```
if best[u] > fee(u→v)  then
    best[v] = max(best[v], (best[u] - fee(u→v)) * rate(u→v))
```

5. **检测"增益回路"**：
   - 纯粹为了把"最终回到 S 的金额 > V"这一目标化为「检测可达正权环」：
     - 在做了 N−1 次松弛之后，若还能在第 N 次松弛中让任意节点的 best 值发生增大，就说明图中存在能够无限增大的"套利回路"；
     - 或者在任何一次松弛中，best[S] 超过了初始值 V，就可以立即判定为 "YES"。