第十次作业

M04089:电话号码

状态: Accepted

基本信息

#:
题目:
提交人:
内存:
时间:
语言:
提交时间:

源代码

```
def is_consistent(numbers):
    numbers.sort()
    for i in range(len(numbers) - 1):
        if numbers[i+1].startswith(numbers[i]):
            return False
    return True

t = int(input())
for _ in range(t):
    n = int(input())
    numbers = [input().strip() for _ in range(n)]
    print("YES" if is_consistent(numbers) else "NO")
```


Startswith()语法

T28046:词梯

状态: Accepted

源代码

```
from collections import deque, defaultdict

def find_word_ladder():
    n = int(input())
    words = [input().strip() for _ in range(n)]
    start, end = input().split()

    if start == end:
        print(start)
        return

    # Preprocess: build a map from wildcard to list of words
    wildcard_map = defaultdict(list)
    for word in words:
        for i in range(4):
            wildcard = word[:i] + '_' + word[i+1:]
            wildcard_map[wildcard].append(word)

    # Build adjacency list
    graph = defaultdict(list)
    for word in words:
        for i in range(4):
            wildcard = word[:i] + '_' + word[i+1:]
            for neighbor in wildcard_map[wildcard]:
                if neighbor != word:
                    graph[word].append(neighbor)

    # BFS setup
    queue = deque()
    queue.append(start)
    visited = {start: None}  # to keep track of the path (stores parent

    found = False
    while queue:
        current = queue.popleft()
        if current == end:
            found = True
            break
        for neighbor in graph.get(current, []):
            if neighbor not in visited:
                visited[neighbor] = current
                queue.append(neighbor)

    if not found:
        print("NO")
    else:
        # Reconstruct the path
        path = []
        node = end
        while node is not None:
            path.append(node)
            node = visited[node]
        path.reverse()
        print(' '.join(path))

find_word_ladder()
```

defaultdict

```python
d = {}
d['a'].append(1)    # ❌ 오류 발생! ('a' 키가 없기 때문)
```

`defaultdict`는 키가 없을 때도, **자동으로 기본값을 만들어줍니다**.

```python
from collections import defaultdict

d = defaultdict(list)
d['a'].append(1)    # ✅ 'a'가 없으면 자동으로 빈 리스트 생성 후 append
print(d['a'])       # 출력: [1]
```

✅ 예제 1: 단어 등장 횟수 세기

python 复制

```python
from collections import defaultdict

sentence = "i love python and i love coding"
counter = defaultdict(int)

for word in sentence.split():
    counter[word] += 1

print(counter)
# 출력: {'i': 2, 'love': 2, 'python': 1, 'and': 1, 'coding': 1}
```

python

```
[1, 2, 3, None, 4]
# 트리 구조:
#     1
#    / \
#   2   3
#    \
#     4
```

✅ 리스트 → 트리 변환 코드 (Python)

python

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

def list_to_tree(arr):
    if not arr:
        return None

    nodes = [TreeNode(val) if val is not None else None for val in arr]

    for i in range(len(arr)):
        if nodes[i] is not None:
            left_index = 2 * i + 1
            right_index = 2 * i + 2
            if left_index < len(arr):
                nodes[i].left = nodes[left_index]
            if right_index < len(arr):
                nodes[i].right = nodes[right_index]

    return nodes[0]  # 루트 노드 반환
```

python 复制

```python
from collections import defaultdict

def reachable_node_count(n, edges, restricted):
    # 构建邻接表
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    restricted_set = set(restricted)
    visited = set()
    count = 0

    def dfs(node):
        nonlocal count
        visited.add(node)
        count += 1
        for neighbor in graph[node]:
            if neighbor not in visited and neighbor not in restricted_set:
                dfs(neighbor)

    dfs(0)
    return count

# ------------ 输入处理部分 ------------
n = int(input())
edges = [tuple(map(int, input().split())) for _ in range(n - 1)]
try:
    restricted = list(map(int, input().split()))
except:
    restricted = []

# 输出结果
print(reachable_node_count(n, edges, restricted))
```

🎁 보너스: 반대로 트리를 리스트로 바꾸기

python

```python
from collections import deque

def tree_to_list(root):
    if not root:
        return []
    result = []
    queue = deque([root])
    while queue:
        node = queue.popleft()
        if node:
            result.append(node.val)
            queue.append(node.left)
            queue.append(node.right)
        else:
            result.append(None)

    # 뒤에 불필요한 None 제거
    while result and result[-1] is None:
        result.pop()
    return result
```

✅ 5. 레벨 순회 (BFS, queue 이용)

python

```python
from collections import deque

def level_order(root):
    if not root:
        return
    queue = deque([root])
    while queue:
        node = queue.popleft()
        print(node.val, end=' ')
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
```

🔶 전위 순회 (preorder): root → left → right

python

```python
def preorder(node):
    if node:
        print(node.val, end=' ')
        preorder(node.left)
        preorder(node.right)

preorder(root)  # 출력: 1 2 3
```

🔶 중위 순회 (inorder): left → root → right

python

```python
def inorder(node):
    if node:
        inorder(node.left)
        print(node.val, end=' ')
        inorder(node.right)
```

🔶 후위 순회 (postorder): left → right → roo

python

```python
def postorder(node):
    if node:
        postorder(node.left)
        postorder(node.right)
        print(node.val, end=' ')
```

다익스트라

python

```
def insert(node, val):
    if node is None:
        return TreeNode(val)  # 새로운 노드를 만들어 리턴

    if val < node.val:
        node.left = insert(node.left, val)  # 왼쪽에 삽입
    else:
        node.right = insert(node.right, val)  # 오른쪽에 삽입

    return node  # 루트 노드를 계속 반환
```

> TreeNode 클래스는 생략되어 있지만, 보통 이렇게 생겼어요:

python

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

2. 트리 만들기 예시

python

```
root = None
for v in [5, 3, 7, 2, 4, 6, 8]:
    root = insert(root, v)
```

- 순서대로 BST에 삽입:

```
        5
      /   \
     3     7
    / \   / \
   2  4  6   8
```

이진탐색트리

单调栈

python

```
arr = [2, 1, 5, 6, 2, 3]
stack = []
res = [-1] * len(arr)

for i in range(len(arr)):
    while stack and arr[i] < arr[stack[-1]]:
        res[stack.pop()] = i
    stack.append(i)
```

python

```python
import heapq

def dijkstra(n, graph, start):
    # 거리 저장 배열: 초기값은 무한대
    dist = [float('inf')] * n
    dist[start] = 0

    # 최소힙 (거리, 노드)
    heap = [(0, start)]

    while heap:
        cost, u = heapq.heappop(heap)

        # 이미 더 짧은 경로가 있다면 패스
        if cost > dist[u]:
            continue

        for v, weight in graph[u]:
            if dist[v] > cost + weight:
                dist[v] = cost + weight
                heapq.heappush(heap, (dist[v], v))

    return dist
```

✅ 4. 입력 예제

python

```python
# 정점 수 n = 5
# 인접 리스트 방식 (0-based index)
graph = [
    [(1, 2), (2, 3)],       # 0번 노드 → 1(2), 2(3)
    [(0, 2), (3, 4)],       # 1번 노드
    [(0, 3), (3, 1), (4, 5)],  # 2번 노드
    [(1, 4), (2, 1), (4, 1)],  # 3번 노드
    [(2, 5), (3, 1)]        # 4번 노드
]

dist = dijkstra(5, graph, start=0)
print(dist)  # [0, 2, 3, 4, 5]
```

✅ 1. 최소 신장 트리 (MST)란?

🔷 정의

- **모든 정점을 연결**하되, **간선 가중치의 합이 최소**인 트리
- 총 간선 수는 항상 V−1 (V는 정점 수)
- **사이클 없음, 모든 노드 연결됨**

✅ 2. MST 알고리즘 2가지

알고리즘	핵심 아이디어	시간복잡도	구현 방식
Kruskal	가중치 작은 간선부터 선택 (Greedy)	O(E log E)	간선 정렬 + **유니온 파인드**
Prim	정점에서 출발하여 연결 확장 (Greedy)	O(E log V)	**우선순위 큐** 이용

✅ 4. Prim 알고리즘 (프림)

🔷 동작 방식

1. 임의의 정점에서 시작
2. **우선순위 큐(heap)**에 연결 가능한 간선 push
3. **가장 비용 낮은 간선**을 선택하면서 **트리 확장**

🔷 예제 코드

python

```python
import heapq

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

visited = [False] * (n + 1)
heap = [(0, 1)]  # (가중치, 시작 정점)
mst_cost = 0

while heap:
    w, u = heapq.heappop(heap)
    if visited[u]:
        continue
    visited[u] = True
    mst_cost += w
    for next_w, v in graph[u]:
        if not visited[v]:
            heapq.heappush(heap, (next_w, v))

print(mst_cost)
```

🔷 동작 방식

1. 모든 간선을 **가중치 오름차순 정렬**
2. **사이클이 생기지 않는다면** 간선 선택
3. 사이클 여부는 **유니온 파인드**로 판단

🔷 예제 코드

python 复制

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

def union(x, y):
    x, y = find(x), find(y)
    if x != y:
        parent[y] = x
        return True
    return False

# 입력
n, m = map(int, input().split())  # 정점, 간선 수
edges = []
for _ in range(m):
    u, v, w = map(int, input().split())
    edges.append((w, u, v))  # 가중치 먼저 저장

# 초기화
parent = list(range(n + 1))
edges.sort()

mst_cost = 0
count = 0
for w, u, v in edges:
    if union(u, v):
        mst_cost += w
        count += 1
        if count == n - 1:
            break

print(mst_cost)
```

✅ 5. 예제 입력

```
7 9
1 2 29
1 5 75
2 3 35
2 6 34
3 4 7
4 6 23
4 7 13
5 6 53
6 7 25
```

✅ MST 결과 (총 비용 출력)

```
159
```

二叉搜索树的层次遍历

#48985538提交状态

状态: Accepted

基本信

源代码

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

def insert_into_bst(root, value):
    if root is None:
        return TreeNode(value)
    if value < root.value:
        root.left = insert_into_bst(root.left, value)
    elif value > root.value:
        root.right = insert_into_bst(root.right, value)
    # 忽略重复值
    return root

def level_order_traversal(root):
    if not root:
        return []
    queue = [root]
    result = []
    while queue:
        current = queue.pop(0)
        result.append(str(current.value))
        if current.left:
            queue.append(current.left)
        if current.right:
            queue.append(current.right)
    return result

# 读取输入
numbers = list(map(int, input().split()))
# 构建BST
root = None
for num in numbers:
    root = insert_into_bst(root, num)
# 层次遍历
traversal = level_order_traversal(root)
print(' '.join(traversal))
```


Import heapq

✅ heapq 주요 기능 요약

함수	설명
heapq.heappush(heap, item)	힙에 값 추가
heapq.heappop(heap)	가장 작은 값 제거 및 반환
heapq.heapify(list)	일반 리스트를 힙으로 변환
heapq.heappushpop(heap, item)	push 후 pop (더 빠름)
heapq.nlargest(k, iterable)	가장 큰 k개
heapq.nsmallest(k, iterable)	가장 작은 k개

Counter는 요소들의 개수를 자동으로 세어주는 특수한 딕셔너리(dict) 클래스입니다.

python

```
from collections import Counter
```

✅ 예제 사용법

🔷 1. 문자열에서 문자 개수 세기

python

```
from collections import Counter

s = "banana"
counter = Counter(s)
print(counter)
# 출력: Counter({'a': 3, 'n': 2, 'b': 1})
```

🔷 2. 리스트에서 등장 횟수 세기

python

```
nums = [1, 2, 2, 3, 3, 3]
c = Counter(nums)
print(c)
# 출력: Counter({3: 3, 2: 2, 1: 1})
```

```
# 把字符转换为数字 (a -> 0, b -> 1, ..., z -> 25)
def char_to_index(c):
    return ord(c) - ord('a')
```

Kosaraju算法的关键在于第二次DFS的顺序，它保证了在DFS的过程中，我们能够优先访问到整个图中的强连通分量。因此，Kosaraju算法的时间复杂度为O(V + E)，其中V是顶点数，E是边数。

以下是Kosaraju算法的Python实现，使用stack模拟按照结束时间的递减顺序访问顶点。

```
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)
```

python 复制

```python
from collections import defaultdict

def reachable_node_count(n, edges, restricted):
    # 构建邻接表
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    restricted_set = set(restricted)
    visited = set()
    count = 0

    def dfs(node):
        nonlocal count
        visited.add(node)
        count += 1
        for neighbor in graph[node]:
            if neighbor not in visited and neighbor not in restricted_set:
                dfs(neighbor)

    dfs(0)
    return count

# ------------ 输入处理部分 ------------
n = int(input())
edges = [tuple(map(int, input().split())) for _ in range(n - 1)]
try:
    restricted = list(map(int, input().split()))
except:
    restricted = []

# 输出结果
print(reachable_node_count(n, edges, restricted))
```

레벨 리스트 루트부터 끝까지 경로

python 复制

```python
from collections import defaultdict

def reachable_node_count(n, edges, restricted):
    # 构建邻接表
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    restricted_set = set(restricted)
    visited = set()
    count = 0

    def dfs(node):
        nonlocal count
        visited.add(node)
        count += 1
        for neighbor in graph[node]:
            if neighbor not in visited and neighbor not in restricted_set:
                dfs(neighbor)

    dfs(0)
    return count

# ------------ 输入处理部分 ------------
n = int(input())
edges = [tuple(map(int, input().split())) for _ in range(n - 1)]
try:
    restricted = list(map(int, input().split()))
except:
    restricted = []

# 输出结果
print(reachable_node_count(n, edges, restricted))
```

源代码

```python
def infix_to_postfix(expression):
    # 定义运算符优先级
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2}

    # 初始化栈和输出列表
    stack = []
    output = []

    i = 0
    while i < len(expression):
        char = expression[i]

        # 如果是数字或小数点，读取完整的数字
        if char.isdigit() or char == '.':
            num = ''
            while i < len(expression) and (expression[i].isdigit() or e
                num += expression[i]
                i += 1
            output.append(num)
            continue

        # 如果是左括号，直接入栈
        elif char == '(':
            stack.append(char)

        # 如果是右括号，弹出栈中的元素直到遇到左括号
        elif char == ')':
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            stack.pop()  # 弹出左括号

        # 如果是运算符
        elif char in precedence:
            # 弹出栈中优先级大于等于当前运算符的运算符
            while stack and stack[-1] in precedence and precedence[stac
                output.append(stack.pop())
            stack.append(char)

        i += 1

    # 将栈中剩余的运算符弹出
    while stack:
        output.append(stack.pop())

    # 返回后序表达式，用空格分隔
    return ' '.join(output)

# 读取输入
n = int(input())
for _ in range(n):
    expression = input().strip()
    # 转换并输出后序表达式
    print(infix_to_postfix(expression))
```

#: 48646842
题目: 24591
提交人: 24n2300093(
内存: 3712kB
时间: 39ms
语言: Python3
提交时间: 2025-03-20

:森林的带度数层次序列存储

状态: Accepted

基本信息

#:	48816847
题目:	07161
提交人:	24n2300093007
内存:	3704kB
时间:	22ms
语言:	Python3
提交时间:	2025-04-04 14:29:45

源代码

```python
from collections import deque

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

def build_tree(sequence):
    if not sequence:
        return None

    nodes = []
    # 将序列分成节点和度数的对
    for i in range(0, len(sequence), 2):
        char = sequence[i]
        degree = int(sequence[i+1])
        nodes.append((char, degree))

    if not nodes:
        return None

    root = Node(nodes[0][0])
    queue = deque()
    queue.append((root, nodes[0][1]))

    index = 1
    while queue and index < len(nodes):
        current_node, degree = queue.popleft()
        # 接下来的degree个节点是当前节点的子节点
        children_nodes = nodes[index: index + degree]
        index += degree
        for char, child_degree in children_nodes:
            child_node = Node(char)
            current_node.children.append(child_node)
            queue.append((child_node, child_degree))
    return root

def postorder_traversal(root):
    result = []
    def traverse(node):
        if node:
            for child in node.children:
                traverse(child)
            result.append(node.value)
    traverse(root)
    return result

n = int(input())
trees = []
for _ in range(n):
    parts = input().split()
    tree = build_tree(parts)
    trees.append(tree)

postorder = []
for tree in trees:
    postorder.extend(postorder_traversal(tree))

print(' '.join(postorder))
```

```python
from typing import Optional, List

# 定义 TreeNode 类
class TreeNode:
    def __init__(self, val=0, left: Optional['TreeNode'] = None, right: Optional['TreeNode'] = None):
        self.val = val
        self.left = left
        self.right = right

# 二叉树前序遍历实现
class Solution:
    # 递归实现
    def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        res = []
        def preorder(node):
            if not node:
                return
            res.append(node.val)
            preorder(node.left)
            preorder(node.right)
        preorder(root)
        return res

    # 栈实现（迭代）
    def preorderTraversal2(self, root: Optional[TreeNode]) -> List[int]:
        if not root:
            return []
        res = []
        stack = [root]
        while stack:
            node = stack.pop()
            res.append(node.val)
            if node.right:
                stack.append(node.right)
            if node.left:
                stack.append(node.left)
        return res
```

```python
#通过递归实现中序遍历
def inorderTraversal(self,root:TreeNode)->list[int]:
    res=[]
    def inorder(root):
        if not root:
            return
        inorder(root.left)
        res.append(root.val)
        inorder(root.right)

    inorder(root)
    return res


#通过递归实现后序遍历
def postorderTraversal(self,root: TreeNode)->list[int]:
    res=[]
    def postorder(root):
        if not root:
            return
        postorder(root.left)
        postorder(root.right)
        res.append(root.val)
    postorder(root)
    return res

#二叉树的层次遍历：
def levelOrder(self,root:TreeNode)->list[list[int]]:
    if not root:
        return []
    queue = [root]
    order = []
    while queue:
        level=[]
        size=len(queue)
        for _ in range(size):
            curr=queue.pop(0)
            level.append(curr.val)
            if curr.left:
                queue.append(curr.left)
            if curr.right:
                queue.append(curr.right)
        if level:
            order.append(level)
    return order
```