2025 spring DSA cheat sheet

```
#pylint: skip-file 解决CE问题
```

Grammar 语法

OrderedDict 有序字典

```
from collections import OrderedDict # 本质用双向链表维护
od = OrderedDict()
od[key]=value
od.popitem(last=False) # 移除最早插入的元素
od.move_to_end(key, last=True) # 移到末尾，last=True可省略
del od[key]
```

换行打印——文件结构图

```
def print_dir(prev, node):
    for child in node.children:
        print(f'{prev}|     {child.name}')
        print_dir(prev + '|     ', child)
    node.files.sort()
    for file in node.files:
        print(f'{prev}{file}')
```

寻找前缀和的最大值

```
pre1 = list(accumulate(nums, max))
pre2 = list(accumulate(nums[::-1], min))[::-1]
```

lambda创建无名函数

```
lambda x, y:int(x/y) #:前是参数，:后是函数唯一的表达式
```

类的比较方法

运算符	方法名
`<`	`__lt__`
`<=`	`__le__`
`>`	`__gt__`
`>=`	`__ge__`
`==`	`__eq__`
`!=`	`__ne__`

Algorithm 算法

Binary Search 二分查找

```
#二分查找最大
def binary_search():
    low = 0
    high = (stall[-1] - stall[0])//(c-1)
    while low <= high:
        mid = (low + high) // 2
        if can_reach(mid):
            low = mid + 1
        else:
            high = mid - 1
    return high
#二分查找最小
def binary_search():
    low = 0
    high = 1000000000
    while low <= high:
        mid = (low+high)//2
        if check(mid):
            high = mid - 1
        else:
            low = mid + 1

    return low
```

KMP: 优化字符串匹配

```
def compute_lps(pattern):
    m = len(pattern)
    lps = [0] * m
    for i in range(1, m):
        length = lps[i-1]
        while length > 0 and pattern[length] != pattern[i]:
            length = lps[length-1]
        if pattern[i] == pattern[length]:
            length += 1
        lps[i] = length
    return lps

def kmp_search(text, pattern):
    lps = compute_lps(pattern)
    j = 0
    m = len(pattern)
    n = len(text)
    match = []
    for i in range(n):
        while j > 0 and pattern[j] != text[i]:
            j = lps[j-1]
        if text[i] == pattern[j]:
            j += 1
        if j == m:
            match.append(i-j+1)
            j = lps[j-1]
    return match

def max_t(s):
    lps = compute_lps(s)
    for i in range(m):
        if lps[i] != 0 and (i+1)%(i+1-lps[i]) == 0:
            k = (i+1)//(i+1-lps[i])
            print(i+1, k)
```

Merge Sort 归并排序

```
while True:
    n = int(input())
    if n == 0:
        break
    arr = [int(input()) for _ in range(n)]
    minimum = 0

    def mergesort(arr):
        global minimum

        if len(arr) > 1:
            mid = len(arr)//2
            left = arr[:mid]
            right = arr[mid:]

            left = mergesort(left)
            right = mergesort(right)

            Lptr = 0
            Rptr = 0
            ptr = 0
            while Lptr<len(left) and Rptr<len(right):
                if left[Lptr] <= right[Rptr]:
                    arr[ptr] = left[Lptr]
                    ptr += 1
                    Lptr += 1
                else:
                    arr[ptr] = right[Rptr]
                    ptr += 1
                    Rptr += 1
                    minimum += len(left)-Lptr
            while Lptr < len(left):
                arr[ptr] = left[Lptr]
                ptr += 1
                Lptr += 1
            while Rptr < len(right):
                arr[ptr] = right[Rptr]
                ptr += 1
                Rptr += 1
        return arr
    mergesort(arr)
    print(minimum)
```

滑动窗口

使用for循环讨论一个边界，再用while循环讨论另一个边界。

```
class Solution:
    def countCompleteSubarrays(self, nums: List[int]) -> int:
        ans = 0
        k = len(set(nums))
        left = 0
        freq = dict()
        for right in range(len(nums)):
            freq[nums[right]] = freq.get(nums[right], 0) + 1
            while len(freq) == k:
                ans += len(nums)-right
                freq[nums[left]] -= 1
                if freq[nums[left]] == 0:
                    del freq[nums[left]]
                left += 1
        return ans
```

Stack 栈

Encoding strings 字符串解码

```
s = input()
num = ''
temp = ''
stack = []
for i in range(len(s)):
    if s[i].isalpha() or s[i] == '[':
        if num != '':
            stack.append(int(num))
            num = ''
        stack.append(s[i])
    elif s[i].isnumeric():
        num += s[i]
    elif s[i] == ']':
        temp = ''
        while not isinstance(stack[-1], int) and stack[-1] != '[':
            temp = stack.pop() + temp
        if stack[-1] != '[':
            temp = stack.pop() * temp
        stack.pop()
        stack.append(temp)
print(*stack, sep='')
```

Poland Expression 波兰表达式

```python
from collections import deque
s = deque(input().split())
def f():
    op = s.popleft()
    if op in '+-*/':
        return eval(str(f())+op+str(f()))
    return op
print(f'{f():.6f}')
```

出栈序列

```python
x = input()
while True:
    try:
        s = input()
    except EOFError:
        break
    i = 0
    j = 0
    stack = []
    while i < len(x):
        stack.append(x[i])
        i += 1
        while stack and j < len(s) and stack[-1] == s[j]:
            stack.pop()
            j += 1
    if j == len(x):
        print('YES')
    else:
        print('NO')
```

所有出栈序列的数目（dp/Catalan Number)

```python
## dp
n = int(input())
dp = [0]*(n+1)
dp[0] = 1
for i in range(1, n+1):
    for j in range(i):
        dp[i] += dp[j]*dp[i-1-j]
print(dp[n])
## Catalan
import math
n = int(input())
print(math.comb(2 * n, n) // (n + 1))
```

Monotonic Stack 单调栈

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        n = len(heights)
        left, right = [0] * n, [n] * n

        mono_stack = list()
        for i in range(n):
            while mono_stack and heights[mono_stack[-1]] >= heights[i]:
                right[mono_stack[-1]] = i
                mono_stack.pop()
            left[i] = mono_stack[-1] if mono_stack else -1
            mono_stack.append(i)

        ans = max((right[i] - left[i] - 1) * heights[i] for i in range(n)) if n > 
0 else 0
        return ans
```

Shunting Yard 调度场：中序转后序

```
n = int(input())
for _ in range(n):
    s = input()
    stack = [] # 存储符号，符号栈
    buffer = [] # 存储后序表达式；注意，中序与后序的区别仅仅在于符号的顺序，操作数（数字）的顺
序是不变的。
    num= ''

    for char in s:
        if char.isnumeric() or char == '.':
            num += char
        else:
            if num:
                buffer.append(num)
                num= ''
            if char in '*/': # 知道后面是什么才能知道前面的后序表达式应该是什么
                while stack and stack[-1] in '*/':
                    buffer.append(stack.pop())
                stack.append(char)
            elif char in '+-':
                while stack and stack[-1] in '+-*/':
                    buffer.append(stack.pop())
                stack.append(char)
            elif char in '(':
                stack.append(char)
            elif char in ')':
                while stack[-1] in '+-*/':
                    buffer.append(stack.pop())
                stack.pop()
    if num:
        buffer.append(num)
    while stack:
        buffer.append(stack.pop())
    print(*buffer, sep = ' ')
```

Linked List 链表

快慢指针——回文链表

```
class Solution:
    def isPalindrome(self, head: Optional[ListNode]) -> bool:
        if not head or not head.next:
            return True

        slow, fast = head, head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        prev = None
        while slow:
            next_node = slow.next
            slow.next = prev
            prev = slow
            slow = next_node

        left, right = head, prev
        while right:
            if left.val != right.val:
                return False
            left = left.next
            right = right.next

        return True
```

反转链表

```
## prev 指向当前反转段后面的节点（即 fast）
prev = fast
curr = slow.next
for _ in range(k):
    temp = curr.next
    curr.next = prev
    prev = curr
    curr = temp
## 此时，prev 为反转后段的头，slow.next 为反转前段的头（现为尾）
temp = slow.next
slow.next = prev
slow = temp
```

Tree 树

notation 树的基本性质

```
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.children = [] # 多叉树
```

高度、深度、节点数、直径（左右） -> dfs

parse 建树

扩展二叉树，用'.'占位符补足空节点

```
def generate_tree(i):
    if s[i].isalpha():
        node = Node(s[i])
        node.left, i = generate_tree(i+1)
        node.right, i = generate_tree(i+1) # 在左子树建完后继承上面的编号继续建树
        return node, i
    elif s[i] == '.':
        return None, i
```

括号嵌套树，用（）标记树的层级

```
def generate_tree(s):
    stack = []
    for char in s:
        if char.isalpha():
            node = Node(char)
            if stack:
                stack[-1].children.append(node)
        elif char == '(':
            stack.append(node)
        elif char == ')':
            node = stack.pop()
    return node
```

根据层级level 建树

```
def parse_tree():
    stack = []
    while True:
        s = input()
        if s == '0':
            break
        if not stack:
            stack.append((len(s)-1, Node(s)))
        else:
            level = len(s)-1
            node = Node(s[-1])
            while level <= stack[-1][0]:
                stack.pop()
            parent_node = stack[-1][1]
            if parent_node.left:
                parent_node.right = node
                stack.append((level, node))
            else:
                parent_node.left = node
                stack.append((level, node))
    return stack[0][1]
```

Disjoint Set / Union Find 并查集

```
def find_father(x):
    if father[x] != x:
        father[x] = find_father(father[x])
    return father[x]
def union(a, b):
    fa_a = find_father(a)
    fa_b = find_father(b)
    if fa_a != fa_b:
        father[fa_a] = fa_b
        return True
    return False
```

traverse 树的遍历：preorder前序，inorder中序，postorder后序，level order层序的相互转化

```
def parse_tree(preorder, inorder):
    if not inorder or not preorder:
        return None
    node = Node(preorder[0])
    index = inorder.index(node.val)
    node.left = parse_tree(preorder[1:index+1], inorder[:index])
    node.right = parse_tree(preorder[index+1:], inorder[index+1:])
    return node

root = parse_tree(preorder, inorder)

def postorder(node):
    if not node:
        return ''
    return postorder(node.left)+postorder(node.right)+node.val

def traversal(root):
    stack = deque([root])
    ans = []
    while stack:
        node = stack.popleft()
        if node:
            ans.append(node.val)
            stack.append(node.left)
            stack.append(node.right)
    return ans
```

二叉树的条件求和

tree+prefix：计算路径和

```
class Solution:
    def __init__(self):
        self.ans = 0
    def pathSum(self, root: Optional[TreeNode], targetSum: int) -> int:
        prefix = defaultdict(int)
        prefix[0] = 1
        def dfs(node, temp):
            if node:
                temp += node.val
                self.ans += prefix[temp-targetSum]
                prefix[temp] += 1
                dfs(node.left, temp)
                dfs(node.right, temp)
                prefix[temp] -= 1
                temp -= node.val
        dfs(root, 0)
        return self.ans
```

tree+dp：小偷问题

```
class Solution:
    def rob(self, root: Optional[TreeNode]) -> int:
        def dfs(node):
            if node is None:
                return (0, 0)
            left_rob, left_not_rob = dfs(node.left)
            right_rob, right_not_rob = dfs(node.right)
            node_rob = node.val + left_not_rob + right_not_rob
            node_not_rob = max(left_rob, left_not_rob) + max(right_rob,
right_not_rob)
            return (node_rob, node_not_rob)
        return(max(dfs(root)))
```

二叉树的最近祖先

```
class Solution:
    """
    如果当前节点是 null 或者是目标节点之一（p 或 q），直接返回当前节点。
    递归左右子树：
    左子树返回值为 l，右子树返回值为 r。
    根据左右子树的返回值判断：
    如果左子树返回 null，说明 p 和 q 都在右子树中，返回右子树的结果。
    如果右子树返回 null，说明 p 和 q 都在左子树中，返回左子树的结果。
    如果左右子树都不为 null，说明当前节点就是最近公共祖先，返回当前节点。
    """
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q:
'TreeNode') -> 'TreeNode':
        node = root
        if root is None or root == p or root == q:
            return root
        node.left = self.lowestCommonAncestor(node.left, p, q)
        node.right = self.lowestCommonAncestor(node.right, p ,q)
        if node.left == None:
            return node.right
        if node.right == None:
            return node.left
        return node
```

operate 树的操作

交换节点（包括节点子树）

```
    node_x, node_y = nodes[x], nodes[y]
    fa_x, fa_y = node_x.father, node_y.father
    node_x.father, node_y.father = fa_y, fa_x
    for attr1 in ['left', 'right']:
        for attr2 in ['left', 'right']:
            if getattr(fa_x, attr1) == node_x and getattr(fa_y, attr2) == node_y:
                setattr(fa_x, attr1, node_y)
                setattr(fa_y, attr2, node_x)
```

BST 二叉搜索树

性质：比node小的**全部**在左侧，比node大的**全部**在右侧。

```
## 给定任意表达式建树
def parse_tree(node, val):
    if node is None:
        node = Node(val)
    elif node.val > val:
        node.left = parse_tree(node.left, val)
    elif node.val < val:
        node.right = parse_tree(node.right, val)
    return node
## 给定前序建树
def parse_tree(s):
    if not s:
        return None
    node = Node(s[0])
    idx = len(s)
    for i in range(1, len(s)):
        if s[i] > s[0]:
            idx = i
            break
    node.left = parse_tree(s[1:idx])
    node.right = parse_tree(s[idx:])
    return node
## 搜索第k小的元素
class Solution:
    def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
        stack = []
        while root or stack:
            while root:
                stack.append(root)
                root = root.left
            root = stack.pop()
            k -= 1
            if k == 0:
                return root.val
            root = root.right
```

AST 抽象语法树

```
precedence = {'not': 3, 'and': 2, 'or': 1}
class Node:
    def __init__(self, val):
        self.val= val
        self.left = None
        self.right = None
    def __le__(self, other):
        return precedence[self.val] <= precedence[other.val]
    def __lt__(self, other):
        return precedence[self.val] < precedence[other.val]
s = list(input().split())
def parse_tree(s):
    operands = {'False', 'True'}
    operators_stack = []
    operands_stack = []
    for i in range(len(s)):
        if s[i] in operands:
            operands_stack.append(Node(s[i]))
        elif s[i] in precedence:
            node = Node(s[i])
            if operators_stack and operators_stack[-1] != '(' and node <=
operators_stack[-1]:
                if operators_stack[-1].val == 'not':
                    operators_stack[-1].right = operands_stack.pop()
                    operands_stack.append(operators_stack.pop())
                else:
                    operators_stack[-1].right = operands_stack.pop()
                    operators_stack[-1].left = operands_stack.pop()
                    operands_stack.append(operators_stack.pop())
            operators_stack.append(node)
        elif s[i] == '(':
            operators_stack.append(s[i])
        else:
            while operators_stack[-1] != '(':
                if operators_stack[-1].val == 'not':
                    operators_stack[-1].right = operands_stack.pop()
                    operands_stack.append(operators_stack.pop())
                else:
                    operators_stack[-1].right = operands_stack.pop()
                    operators_stack[-1].left = operands_stack.pop()
                    operands_stack.append(operators_stack.pop())
            operators_stack.pop()
    while operators_stack:
        if operators_stack[-1].val == 'not':
            operators_stack[-1].right = operands_stack.pop()
            operands_stack.append(operators_stack.pop())
        else:
                operators_stack[-1].right = operands_stack.pop()
                operators_stack[-1].left = operands_stack.pop()
                operands_stack.append(operators_stack.pop())
    return operands_stack.pop()

def inorder(node):
    res = []
    if not node:
        return res
    if node.left and node.left.val in precedence and node.left < node:
        res.extend(['(']+inorder(node.left)+[')']+[node.val])
    else:
        res.extend(inorder(node.left)+[node.val])
    if node.right and node.right.val in precedence and node.right <= node:
        res.extend(['(']+inorder(node.right)+[')'])
    else:
        res.extend(inorder(node.right))
    return res
root = parse_tree(s)
print(*inorder(root))
```

Hufmann Tree 哈夫曼树

```
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):
        if self.freq == other.freq:
            return self.char < other.char
        return  self.freq < other.freq

n = int(input())
s = []
for _ in range(n):
    char, freq = input().split()
    heapq.heappush(s, Node(char, int(freq)))
while len(s) > 1:
     node_1 = heapq.heappop(s)
     node_2 = heapq.heappop(s)
     node = Node(min(node_1.char, node_2.char), node_1.freq+node_2.freq)
     node.left = node_1
     node.right = node_2
     heapq.heappush(s, node)
root = s.pop()
encoding = dict()
decoding = dict()
def dfs(node, code):
    if not node.left and not node.right:
        encoding[node.char] = code
        decoding[code] = node.char
        return
    dfs(node.left, code+'0')
    dfs(node.right, code+'1')
dfs(root, '')
while True:
    try:
        s = input()
    except EOFError:
        break
    if s.isalpha():
        ans = ''
        for i in range(len(s)):
            ans += encoding[s[i]]
        print(ans)
    elif s.isdigit():
        ans = ''
        temp = ''
        for i in range(len(s)):
            temp += s[i]
            if temp in decoding:
                ans += decoding[temp]
                temp = ''
        print(ans)
```

Trie 字典树

```
class Node:
    def __init__(self):
        self.children = dict()
class Trie:
    def __init__(self):
        self.root = Node()
    def insert_num(self, num):
        current_node = self.root # 注意回到原点，不要改动root指针
        for x in num:
            if x not in current_node.children:
                current_node.children[x] = Node()
            current_node = current_node.children[x]
            if 'end' in current_node.children:
                return False # 排除相同的前缀
        current_node.children['end']= None
        return True
```

Graph 图

最短路径

Dijkstra（没有负权环，单点最短路）

注意松弛的方式，不是返工，而是隐形分层！

```
class Solution:
    def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:
        edges = defaultdict(dict)
        for u, v, w in times:
            edges[u][v] = w
        distances = [0]+[float('inf')]*n
        distances[k] = 0
        q = [(0, k)]
        while q:
            distance, node = heapq.heappop(q)
            if distance > distances[node]:
                continue # 感觉dijkstra的变形都在这里？
            for neighbor in edges[node]:
                new_dis = distance+edges[node][neighbor]
                if new_dis < distances[neighbor]:
                    heapq.heappush(q, (new_dis, neighbor))
                    distances[neighbor] = new_dis
        ans = max(distances[1:])
        if ans == float('inf'):
            return -1
        return ans
```

Bellman Ford（检查负权环，多点最短）

```
## 套利问题
money = [0]*n
money[s] = v
for _ in range(n-1):
    for a, b, rab, cab, rba, cba in bank:
        if money[b] < (money[a]-cab)*rab:
            money[b] = (money[a]-cab)*rab
        if money[a] < (money[b]-cba)*rba:
            money[a] = (money[b] - cba) * rba

for a, b, rab, cab, rba, cba in bank:
    if money[a] - cab >= 0 and money[b] < (money[a] - cab) * rab:
        print("YES")
        break
    if money[b] - cba >= 0 and money[a] < (money[b] - cba) * rba:
        print("YES")
        break
else:
    print("NO")
```

Floyd-Warshall：多源，稠密图

```
def floyd_warshall(graph):
    n = len(graph)
    dist = [[float('inf')] * n for _ in range(n)]

    for i in range(n):
        for j in range(n):
            if i == j:
                dist[i][j] = 0
            elif j in graph[i]:
                dist[i][j] = graph[i][j]

    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist
```

最小生成树

Prim：heapq+visited, 稠密图加点

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

    edges = []
    visited = {0}
    ans = 0

    for i, edge in enumerate(matrix[0]):
        heapq.heappush(edges, (edge, i))
    while len(visited) < n:
        edge, node = heapq.heappop(edges)
        if node in visited:
            continue
        ans += edge
        visited.add(node)
        for i, edge in enumerate(matrix[node]):
            heapq.heappush(edges, (edge, i))
    print(ans)
```

Kruskal: sort+disjoint set，稀疏图加边

```
import heapq

class Solution:
    def minCostConnectPoints(self, points: List[List[int]]) -> int:
        n = len(points)
        edges = []
        for i in range(n):
            for j in range(i+1, n):
                heapq.heappush(edges, (abs(points[i][0]-points[j]
[0])+abs(points[i][1]-points[j][1]), i, j))
        father = [i for i in range(n)]
        def find_father(x):
            if father[x] != x:
                father[x] = find_father(father[x])
            return father[x]
        def union(a, b):
            fa_a = find_father(a)
            fa_b = find_father(b)
            if fa_a != fa_b:
                father[fa_a] = fa_b
                return True
            return False
        ans = 0
        for _ in range(len(edges)):
            edge, i, j = heapq.heappop(edges)
            if union(i, j):
                ans += edge
        return ans
```

Wandroff：启发式dfs优化

```
import sys

sys.setrecursionlimit(1 << 30)
n = int(input())
a, b = map(int, input().split())
visited = [[False]*n for _ in range(n)]
visited[a][b] = True

def get_degree(x, y):
    cnt = 0
    moves = [(1, 2), (1, -2), (-1, 2), (-1, -2), (2, 1), (2, -1), (-2, 1), (-2,
-1)]
    for dx, dy in moves:
        nx = x + dx
        ny = y + dy
        if 0 <= nx < n and 0 <= ny < n and not visited[nx][ny]:
            cnt += 1
    return cnt

def dfs(x, y, temp):
    if temp == n*n:
        return True
    moves = [(1, 2), (1, -2), (-1, 2), (-1, -2),(2, 1), (2, -1), (-2, 1), (-2,
-1)]
    n_step = []
    for dx, dy in moves:
        nx = x + dx
        ny = y + dy
        if 0 <= nx < n and 0 <= ny < n and not visited[nx][ny]:
            n_step.append((get_degree(nx, ny), nx, ny))
    n_step.sort()
    for _, nx, ny in n_step:
        visited[nx][ny] = True
        temp += 1
        if dfs(nx, ny, temp):
            return True
        temp -= 1
        visited[nx][ny] = False

if dfs(a, b, 1):
    print('success')
else:
    print('fail')
```

成环判断

有向图——三色dfs

```python
from collections import defaultdict
edges = defaultdict(set)
n, m = map(int, input().split())
for _ in range(m):
    u, v = map(int, input().split())
    edges[u].add(v)

visited = [-1]*n
def dfs(i):
    visited[i] = 0
    for vertex in edges[i]:
        if visited[vertex] == -1:
            if dfs(vertex):
                return True
        elif visited[vertex] == 0:
            return True
    visited[i] = 1
    return False

for i in range(n):
    if visited[i] == -1:
        if dfs(i):
            print('Yes')
            break
else:
    print('No')
```

有向图——topological order

```python
import sys
from collections import deque

input = sys.stdin.readline

t = int(input())
for _ in range(t):
    n, m = map(int, input().split())
    indeg = [0] * n
    adj = [[] for _ in range(n)]
    for _ in range(m):
        x, y = map(int, input().split())
        x -= 1; y -= 1
        adj[x].append(y)
        indeg[y] += 1

    q = deque(i for i in range(n) if indeg[i] == 0)
    visited = 0
    while q:
        u = q.popleft()
        visited += 1
        for v in adj[u]:
            indeg[v] -= 1
            if indeg[v] == 0:
                q.append(v)

    print('No' if visited == n else 'Yes')
```

无向图——disjoint set

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

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

n, m = map(int, input().split())
parent = list(range(n))
has_cycle = False

for _ in range(m):
    u, v = map(int, input().split())
    if not union(u, v):
        has_cycle = True
        break

print("Yes" if has_cycle else "No")
```

无向图——dfs

```
def dfs(u, parent):
    visited[u] = True
    for v in graph[u]:
        if not visited[v]:
            if dfs(v, u):
                return True
        elif v != parent:
            return True
    return False

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

visited = [False] * n
for i in range(n):
    if not visited[i]:
        if dfs(i, -1):
            print("Yes")
            break
else:
    print("No")
```

SCC 强连通分量： Kosaraju / 2 DFS

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

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


def kosaraju(graph):
    stack = []
    dfs1(0, set(), stack, graph)

    t_graph = [[] for _ in range(len(graph))]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            t_graph[neighbor].append(node)

    sccs = []
    visited = set()
    while stack:
        node = stack.pop()
        if node not in visited:
            scc = []
            dfs2(node, visited, scc, t_graph)
            sccs.append(scc[:])
    return sccs

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)
```