== 数据结构与算法笔记整理 ==

一、排序算法&时间复杂度

1. 排序算法总结

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

① 冒泡排序：一轮一轮从左往右比较相邻元素（从0号开始，和右边比），那么每一轮都会找到一个最小/ 最大的从最右边冒出来。O(n^2)

② 插入排序：从左向右一个个将数圈起来，将最右侧的数据插到左侧的合适位置（其中是按照冒泡的相邻比 较）。因此循环套循环，O(n^2)

③ 希尔排序：插入排序的优化。按照某个步长进行插入排序，每一趟减小步长。

```
#shell_sort
def shellsort(arr, n):
    gap = n // 2

    while gap > 0:
        j = gap
        while j < n:
            i = j - gap

            while i >= 0:
                if arr[i+gap] > arr[i]:
                    break
                else:
                    arr[i+gap], arr[i] = arr[i], arr[i+gap]
                i -= gap
            j += 1
        gap = gap // 2
```

④ 选择排序：一趟趟遍历，找到最小和第i个交换，O(n^2)

⑤ 堆排序：建堆 + 反复heappop。PercDown至多下移树的高度次即Logn，又要调整n个数，为

O(nlogn)。PS：大根堆可以把所有数变成负数再放进小根堆。

⑥ 快速排序：冒泡排序的优化。选择pivot，一趟排序让数组分为大小两部分，再对两部分递归。O(nlogn)

```
#quick_sort(pivot取左)
def quicksort(arr, left, right):
    if left < right:
        key = partition(arr, left, right)
        quicksort(arr, left, key-1)
        quicksort(arr, key+1, right)

def partition(arr, left, right):
    i, j, pivot = left, right, arr[left]
    while i <= j:
        while i <= right and arr[i] <= pivot:
            i += 1
        while j >= left and arr[j] > pivot:
            j -= 1
        if i < j:
            arr[i], arr[j] = arr[j], arr[i]
    if arr[j] < pivot:
        arr[j], arr[left] = arr[left], arr[j]
    return j
#arr = [2,1,8,6,5,7,3,4]
#quicksort(arr,0,len(arr)-1)

#pivot取右就arr[j] >= pivot; 取中间就两个都取等
```

⑦ 归并排序：分治算法。将左右分别排序得到有序子列，再有序归并得到整个有序子列。递归进行，

O(nlogn)。值得注意的是，归并排序也是求逆序数/交换次数的算法（下面的ans）

```
#merge_sort
def mergeSort(ls):
    if len(ls) <= 1:
        return ls, 0
    mid = int(len(ls)/2)
    left, ans_left = mergeSort(ls[0:mid])
    right, ans_right = mergeSort(ls[mid:])
    mergedlist, ans_merge = merge(left, right)
    return mergedlist, ans_left + ans_right + ans_merge

def merge(left: list, right: list):
    ans = 0
    r, l = 0, 0
    result = []
    while l < len(left) and r < len(right):
        if left[l] <= right[r]:
            result.append(left[l])
            l += 1
```

```
        else:
            result.append(right[r])
            r += 1
            ans += len(left) + r - len(result)
    result += left[l:]
    result += right[r:] #如果r >= len(right)会返回空，不会报错
    return result, ans
```

二、基本数据结构——栈、队列、链表

1. 笔试知识：

2. 编程题目：

   ①栈：括号匹配——左括号进栈，遇到右括号弹栈，但是stack[-1]不匹配就错误。

   前中后序表达式：

(1)计算：后序计算只需要一个stack，从左往右遍历，遇到数字入栈、运算符弹两次栈计算，结果入栈，rt.

```
#s = '7 8 + 3 2 + /'
stack = []
for token in s:
    if token in "+-*/":
        num2, num1 = stack.pop(), stack.pop()    #num2后进先出，要注意num2就是运算符后面的那一个
        stack.append(doMath(token, num1, num2)) #doMath是计算 num1 token num2
    else:
        stack.append(int(token))
print(stack[-1])
```

前序计算也只要一个stack，从右往左遍历，遇到数字入栈，遇到操作符弹栈两次，结果入栈——注意先出的是num1而不是num2了

```
#s = '+ + 2 * 3 5 6'
stack = []
for i in range(len(s)-1, -1, -1):
    token = s[i]
    if token in '+-*/':
        num1, num2 = stack.pop(), stack.pop()
        stack.append(doMath(token, num1, num2))
    else:
        stack.append(int(token))
print(stack[-1])
```

(2)前中后序转化

Shunting Yard(调度场)算法——中序转后序：+、-：1、*、/：2，或者将括号标注（：0，）：2。之后用一个栈过渡、一个栈（列表）接收。从左到右遍历：

操作数：加到输出栈。
左括号进运算符栈。
运算符：——如果优先级高于运算符栈栈顶的，则将运算符推入运算符栈；
　　否则将栈顶运算符弹出、并添加到输出栈，直到优先级条件允许，再将该运算符入栈。
右括号：弹出栈顶运算符&添加到输出栈，直到遇到左括号（左括号不入栈）
剩余部分：把stack全部加到postfix里面。
Trick: number buffer——用number ->str 存储num和点，直到遇到运算符将number转化为float，针对有小数点的情况(如下，注意可能遍历完number != ''，要先加到postfix再处理stack里面的字符)

```
#
prec = {'+': 1, '-': 1, '*': 2, '/': 2}
def infix_to_postfix(s: str):
    stack, postfix = [], []
    number = '' #number buffer

    for char in s:
        if char.isnumeric() or char == '.':
            number += char
        else:
            if number:
                num = float(number)
                if num.is_integer():
                    postfix.append(int(num))
                else:
                    postfix.append(num)
            if char in '+-*/':  #上面的if和下面的判断不是一起的
                while stack and stack[-1] in '+-*/' and prec[char] <=
prec[stack[-1]]:
                    postfix.append(stack.pop())
                stack.append(char)
            elif char == '(':
                stack.append(char)
            elif char == ')':
                while stack and stack[-1] != '(':
                    postfix.append(stack.pop())
                stack.pop()
#处理剩余部分
    if number:
        num = float(number)
        if num.is_integer():
            postfix.append(int(num))
        else:
            postfix.append(num)

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

    return ' '.join(str(x) for x in postfix)
```

　　单调栈：用于找左/右第一个大/小数。有单调增/减栈。

```python
#单调递增栈（从栈顶向栈底递增）
nums = input()   #输入数组
stack = []
for i in range(len(nums)):   #例：正向遍历
    num = nums[i]
    while stack and stack[-1] < num:     #或者<=
        stack.pop()
    stack.append(num)
#其中可以在pop、多加一个index栈做文章

#单调递减栈只需要改成>或>=,此外还可以逆向遍历
```

1. 递增栈：

①找右侧第一个更大：正向遍历，当前即将插入的元素 是 pop的元素 的更大
②找左侧第一个更大：正向遍历，当前栈顶的元素（pop循环完之后） 是 当前即将插入的元素 的更大
PS：如果①没有pop或②插入时栈为空，说明没有更大元素。

2. 递减栈：

①找左侧第一个更小：正向遍历，当前栈顶元素 是 即将插入的元素的更大
②找右侧第一个更小：正向遍历，当前即将插入的元素 是 pop的元素 的更大
PS同上。

简记为：都可以正向遍历，找更大/更小用递增/递减栈，找左侧就看插入时栈顶、找右侧就看pop时即将插入的元素。
PS：找更大/更小(>,<)那么写成>=和<=，如果是≥/≤就写成<和>

变形题：接雨水
给定 `n` 个非负整数表示每个宽度为 `1` 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。
输入：height: list[int]，输出：int

思路：用递增栈，相当于找碗的左半边，当需要pop时说明找到右半边。注意计算雨水的时候可以**分层计算**

```python
#
height = list(map(int, input().split()))
stack, index = [], []   #相当于分别存x, y坐标

sol = 0
for i in range(len(height)):
    h = height[i]
    while stack and stack[-1] < h:   #≤和<都ok,可以从下面的+=中推导出来结果是一样的
        h0, i0 = stack.pop(), index.pop()
        if stack:
            sol += (min(stack[-1], h) - h0) * (i - i0 - 1)
    stack.append(h)
    index.append(i)
print(sol)
```

利用栈的后进先出，可以实现很多***括号分割、字符串逆序***的问题。
例：整人的题词本http://cs101.openjudge.cn/practice/20743/

```python
#
s = input()
stack = []
for token in s:
    if token == ')':
        temp = []
        while stack and stack[-1] != '(':
            temp.append(stack.pop())
        stack.pop()
        stack += temp
    else:
        stack.append(token)
print(''.join(stack))
```

②队列：from collections import deque是双端队列，此外还有循环队列——例：约瑟夫问题http://cs101.openjudge.cn/2024sp_routine/03253/

给n,p,m，其中n为数目(编号1-n)，m为步长，p为起始编号

```python
#
n, p, m = map(int, input().split())
queue = [i for i in range(n)]
sol = []
k = p - 2    #编号从1开始故减1，此外p小孩自己要报数所以再减1
while len(queue) > 0:
    k += m
    k = k % len(queue)
    ans.append(str(1 + queue.pop(k)))
    k -= 1  #k小孩自己又要报数所以减1
print(','.join(sol))
```

在bfs中有很丰富的利用。

③链表：和（双端队列、）循环队列一样，更多在笔试中考到，就先不复习了。

三、树

树的类实现：

```python
#二叉树
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
#多叉树
self.children = []
```

此外还可以用邻接表的方式建立树——{node: []}

编程题目:

①数据接收的方法（解析树）：

括号嵌套树：括号分割——利用栈http://cs101.openjudge.cn/practice/24729/

```
#input() = A(B(E),C(F,G),D(H(I)))
class TreeNode:...
def prase_tree(s):
    stack = []  #栈中存储()外即上一级node
    node = None
    for char in s:
        if char.isalpha():
            node = TreeNode(char)
            if stack:
                stack[-1].children.append(node)
        elif char == '(':
            if node:
                stack.append(node)
                node = None
        elif char == ')':
            if stack:
                node = stack.pop()
    return node
```

表达式树：http://cs101.openjudge.cn/practice/25140/

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

def prase_tree(s):
    stack = []
    for token in s:
        node = TreeNode(token)
        if token.islower():
            stack.append(node)
        else:
            b, a = stack.pop(), stack.pop()
            node.left, node.right = a, b
            stack.append(node)
    return stack[0]     #root
```

前+中/中+后建树：关键在用pre/post找到根节点，然后in中根节点分开左右子树，递归即可。

节点编号有序，给子节点编号：http://cs101.openjudge.cn/practice/27638/

```
#
def prase_tree():
    n = input()
```

```
    nodes = [TreeNode(i) for i in range(n)]
    has_parent = [False] * n
    for i in range(n):
        l, r = map(int,input().split())
        if l != -1:
            nodes[i].left = nodes[l]
            has_parent[l] = True
        if r != -1:
            nodes[i].right = nodes[r]
            has_parent[r] = True
    root_index = has_parent.index(False)
    root = nodes[root_index]
    return root
```

另外，无序的话把nodes改为字典 {val: node(val)}，找root也可以用字典。

两个奇怪的题：树的转换http://cs101.openjudge.cn/dsapre/04081/
    多叉树-->左儿子右兄弟树

```
class TreeNode:
    def __init__(self):
        self.children = []
        self.left = None
        self.right = None

root = TreeNode()
nodes = [root]
steps = list(input())
for step in steps:
    if step == 'd':
        node = TreeNode()
        nodes[-1].children.append(node)
        nodes.append(node)
    else:
        nodes.pop()

def prase_tree(root: TreeNode): #可以适当记忆一下
    if root.children:
        root.left = prase_tree(root.children.pop(0))
        cur = root.left
        while root.children:
            cur.right = prase_tree(root.children.pop(0))
            cur = cur.right
    return root
```

树的镜面映射：http://cs101.openjudge.cn/practice/04082/

```python
def prase_tree(arr, index): #左儿子右兄弟的左子树全部是儿子，递归处理完之后，index+=1，指向的是儿子的兄弟节点——也就是现在的节点的儿子。
    val, state = arr[index][0], int(arr[index][1])
    node = TreeNode(val)
    if state == 0 and val != '$':
        index += 1
        child, index = prase_tree(arr, index)
        node.children.append(child)
        index += 1
        child, index = prase_tree(arr, index)
        node.children.append(child)
    return node, index
```

各种算法：

① huffman编码：

从下往上建立树，权值越高的节点越靠近树的根部。因此利用heapq一直合并权值最小的两个节点。

②bfs、dfs算法：

分别利用queue和递归实现。

③并查集：

```python
#union1,union2几乎可以混用
class DisjSet:
    def __init__(self, n):
        self.parent = [i for i in range(n)]
        self.rank = [0] * n

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

    def union1(self, x, y):
        xset, yset = self.find(x), self.find(y)
        if xset == yset:
            return
        if self.rank[xset] > self.rank[yset]:
            self.parent[yset] = xset
        else:
            self.parent[xset] = yset
            if self.rank[xset] == self.rank[yset]:
                self.rank[yset] += 1

    def union2(self, x, y):
        xset, yset = self.find(x), self.find(y)
        self.parent[yset] = xset
```

编程题目：

食物链：http://cs101.openjudge.cn/practice/01182

```python
#开三倍数组，分别存ABC三类动物，
```

```python
class DisjSet:
    def __init__(self, n):
        self.parent = [i for i in range(n+1)]
        self.rank = [0]*(n+1)

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

    def union(self, x, y):
        xset, yset = self.find(x), self.find(y)
        self.parent[xset] = self.parent[yset]

n, k = map(int, input().split())
disjset = DisjSet(3 * n)
sol = 0
dire = [(0, n), (n, 2*n), (2*n, 0)]

for _ in range(k):
    info, x, y = map(int, input().split())
    if x > n or y > n:
        sol += 1
    elif info == 1:
        state = True
        for i,j in dire:
            if disjset.find(x+i) == disjset.find(y+j) or disjset.find(x+j) ==
disjset.find(y+i):
                state = False
                break
        if not state:
            sol += 1
        else:
            disjset.union(x, y)
            disjset.union(n + x, n + y)
            disjset.union(2*n + x, 2*n + y)

    elif info == 2:
        state = True
        for i,j in dire:
            if disjset.find(y+i) == disjset.find(x+j):
                state = False
                break
        if x == y:
            sol += 1
        elif not state:
            sol += 1
        elif disjset.find(x) == disjset.find(y) and disjset.find(n + x) ==
disjset.find(n + y) and disjset.find(2*n + x) == disjset.find(2*n + y):
            sol += 1
        else:
            for i, j in dire:
                disjset.union(x+i, y+j)
print(sol)
```

发现它抓住它：http://cs101.openjudge.cn/2024sp_routine/01703/

```
#开两倍并查集，分别代表两个团伙
t = int(input())
for _ in range(t):
    n, m = map(int, input().split())
    disjset = DisjSet(2*n)
    for i in range(m):
        info, a, b = input().split()
        a, b = int(a), int(b)
        if info == 'D':
            disjset.union(a, n+b)
            disjset.union(b, n+a)
        else:
            a_set, a_set1, b_set, b_set1 = disjset.find(a), disjset.find(n+a), 
disjset.find(b), disjset.find(n+b)
            if a_set == b_set or a_set1 == b_set1:
                print('In the same gang.')
            elif a_set == b_set1 or a_set1 == b_set:
                print('In different gangs.')
            else:
                print('Not sure yet.')
```

Trie前缀树

```
#
class TrieNode:
    def __init__(self):
        self.child = {}

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

    def insert(self, arr):
        cur = self.root
        for num in arr:
            if num not in cur.child:
                cur.child[num] = TrieNode()
            cur = cur.child[num]

    def search(self, arr):
        cur = self.root
        for num in arr:
            if num not in cur.child:
                return 0
            cur = cur.child[num]
        return 1
```

四、图

图的实现：

```
#类实现
import sys
class Vertex:
    def __init__(self, key):
        self.key = key
        self.con = {}
        self.color = 'white'    #用于标记是否遍历，还有gray,black
        self.distance = sys.maxsize #用于dijkstra,prim,下同
        self.pred = None

    def __lt__(self, other):
        return self.key < other.key

    def add_nbr(self, nbr, weight = 0): #nbr: Vertex
        self.con[nbr] = weight

    def get_nbr(self, nbr):
        return self.con.get(nbr)

    def get_con(self):
        return self.con.keys()

class Graph:
    def __init__(self):
        self.vt = {}
        self.vtnum = 0

    def add_vt(self, key):
        new_Vertex = Vertex(key)
        self.vtnum += 1
        self.vt[key] = new_Vertex
        return new_Vertex

    def get_vt(self, key):
        return self.vt.get(key)

    def add_edge(self, f, t, cost=0):
        if f not in self.vt:
            new_Vertex = self.add_vt(f)
        if t not in self.vt:
            new_Vertex = self.add_vt(t)
        self.vt[f].add_nbr(self.vt[t], cost)
        self.vt[t].add_nbr(self.vt[f], cost)    #无向图补这句
```

此外还可以用矩阵图、邻接表图。棋盘类用矩阵图比较好。

图算法：

罗佬的总结太好了，让我的理解得到了完完全全的升华，下面照搬了不少请罗佬见谅

DFS：递归写法

```
def dfs(node):
    """
    前序遍历以node为根的搜索树，但不访问已访问过的节点；
    没有 base case，靠搜完来终止.
    """
    # step1: 访问根
    # 访问
    visited.add(node)

    # step2: 遍历子树
    for nb in node.nbs:
        if nb not in visited:
            dfs(nb)


visited = set()
dfs(startNode)
```

DFS、BFS、其他算法的非递归写法

```
def search(startNode):
    container = Container([startNode])
    # stack -> dfs, queue -> bfs, heap -> ucs.
    expanded = set()

    while container:
        node = container.pop()
        # 在这里访问节点可以重复访问
        if node not in expanded: # 1. 找到下一个要扩展的节点
            # 在这里访问节点就只访问一次
            # 2. 扩展
            for nb in node.nbs:
                # if nb not in expanded:
                container.push(nb)

            # 3. 扩展完做标记
            expanded.add(node)
```

注意一个问题：**BFS标注visited的时机**

BFS 常见写法

```
def bfs():
    queue = deque([startNode])
    visited = {startNode} # 改个名比较好，因为其含义是是否已入过队

    while queue:
        node = queue.popleft()

        for nb in node.nbs:
            if nb not in visited:
                queue.append(nb)
                visited.add(nb)
```

如上所述, 这个写法对 BFS 来说是正确的, 但仅将队列换成栈后不是 DFS, 也不能套用到 UCS.

错误写法

```
def wrongSearch(startNode):
    container = Container([startNode])
    visited = set()

    while container:
        node = container.pop()
        visited.add(node)

        for nb in node.nbs:
            if nb not in expanded:
                container.push(nb)
```

这个写法的问题是, 一个节点进入容器但还没弹出 (及标记) 时, 它可以重复入队, 然后这些重复的节点每次弹出后都会做扩展, 出现重复扩展的问题.

要记住先把根节点标记visited，之后访问nbr的时候当场标记visited，而不是在弹出的时候标记。

狭义CSPs问题

这里的狭义CSPs指那些终止条件为完全遍历一遍的图问题，例如走遍棋盘每一个节点。运用的方法是DFS，但是实质上也是一种回溯——

　　在子树遍历结束之后要***回溯***到根节点。因此需要考虑回溯时是否要把改变的变量恢复到原来状态的问题。例如探索有多少路径能走完棋盘每一个点时，就要先标记新节点visited，再dfs新节点，dfs结束之后标记non-visited。

一些题目

```
#三维接雨水--ucs
import heapq, sys
class Solution:
    def trapRainWater(self, heightMap: List[List[int]]) -> int:
        m, n = len(heightMap), len(heightMap[0])
        dire = [(-1, 0), (1, 0), (0,-1), (0, 1)]
        pq, visited = [], set()
        for i in range(n):
```

```
            heapq.heappush(pq, (heightMap[0][i], (0, i)))
            visited.add((0, i))
            heapq.heappush(pq, (heightMap[m-1][i], (m-1, i)))
            visited.add((m-1, i))
        for i in range(m):
            heapq.heappush(pq, (heightMap[i][0], (i, 0)))
            visited.add((i, 0))
            heapq.heappush(pq, (heightMap[i][n-1], (i, n-1)))
            visited.add((i, n-1))

        sol = 0
        while pq:
            h0, pos = heapq.heappop(pq)
            x, y = pos

            if (x, y) not in visited:
                for dx, dy in dire:
                    nx, ny = x+dx, y+dy
                    if 0<nx<m-1 and 0<ny<n-1 and (nx, ny) not in visited:
                        h = max(h0, heightMap[nx][ny])
                        sol += max(0, h - heightMap[nx][ny])
                        heapq.heappush(pq, (h, (nx, ny)))
                visited.add((x,y))

        return sol
```

拓扑排序与环的检测

拓扑排序针对有向、无环图，利用的是A是B的条件的这种性质进行排序。无环则拓扑排序存在，二者是等价的。

拓扑排序的方法

拓扑排序的方法主要是对入度为0的节点node访问其nbr，将nbr入度减一之后删去node。删除过程中，如果剩下的所有节点入度都大于0，说明有环；如果节点删完了说明无环。
`由此引入Kahn算法`

Kahn算法

上述方法可以看出是BFS方法，直接上代码。

```
from collections import deque, defaultdict

def topological_sort(graph):
    indegree = defaultdict(int)
    result = []
    queue = deque()

# 计算每个顶点的入度
for u in graph:
    for v in graph[u]:
        indegree[v] += 1

# 将入度为 0 的顶点加入队列
```

```
for u in graph:
    if indegree[u] == 0:
        queue.append(u)

# 执行拓扑排序
while queue:
    u = queue.popleft()
    result.append(u)

    for v in graph[u]:
        indegree[v] -= 1
        if indegree[v] == 0:
            queue.append(v)

# 检查是否存在环
if len(result) == len(graph):
    return result
else:
    return None
```

无向图检测环

先理解无向图找环：无向图没有方向要求，dfs就是一路往下走，那么如果走回到visited节点就说明有环

但是无向图会有nbr的neighbor回到node的情况，所以要多parent参数。同时起始节点应该视为没有parent，那么就给一个参数 -1。

两个判断条件的理解：

①if not visited and dfs: 就是从nbr重新开始深搜，parent = node

②elif parent != nbr（即nbr已经visited）：说明从nbr开始的深搜路过node回到nbr；同时parent != nbr排除自己回到自己的情况

PS：不输入parent参数会有两种别的方法——其一是不要elif，但是会导致dfs回到nbr的情况被排除（实际上dfs不会得到True的结果）；其二是用0,1,2类似有向图的方法，但是会导致回到parent又回来（二者状态都是1，直接return True）

上代码

```
#无向图找环
#方法一：并查集（特别好理解）——
    #读取边的两个端点先判断如果find(x)=find(y),那么有环；如果无环就union(x,y)
#方法二：dfs（和有向图类似，但是有不同）

#graph为 列表套列表式 邻接表
visited = [False]*n
def dfs(node: int, parent: int):
    visited[node] = True
    for nbr in graph[node]:     #nbr=>int
        if not visited[nbr]:
            if dfs(nbr, node):
                return True
        elif parent != nbr:
            return True
    return False
for node in range(n):
    if not visited[node]:
        if dfs(node, -1):
```

```
            print('Yes')
            break
else:
    print('No')
```

有向图检测环

其实正常用拓扑排序（先决条件——边的头是尾的先决条件，如果自己是自己的先决条件就无法排序，此时正好图是有环的）的方法：不断移除入度为0的节点，全部移除就无环、未完全移除但是此时找不到这样的节点就有环。

另外的方法：将节点分为未访问、正在访问、访问结束三个状态（用visited 字典 存储0,1,2）表示；
　　两个判断条件的理解：
　　　　①if visited[v] == 0 and dfs(v)：就是从v重新开始深搜。
　　　　②elif visited[v] == 1：说明u是 从v开始的深搜 经历的点，而且此时回到了v。

　　上代码

```
def dfs(u):
    st[u] = 1 # 访问中

    for v in adj[u]:
        if st[v] == 0: # 未访问
            # prev[v] = u  # 无向图要记录父节点
            if dfs(v):
                return True
        elif st[v] == 1: # 子节点v在访问中，说明搜索树中可以从v到v，有环
            # if prev[u] != v:  # 无向图中要排除的情况
            return True

    st[u] = 2 # 访问完
    return False
```

Kosaraju算法：双DFS

最短路径与 Dijkstra 算法

Dijkstra找最短路径的方法主要是利用heapq实现的优先队列。一般情况下pq里面存储`(dist, state, (pos))`，其中state有时会有，是某种限制条件（比如最多走的步数等等用来记录步数）。基本步骤如下：

　　①设置dist数组，全部记录为sys.maxsize，其作用为剪枝。
　　②初始节点入队
　　③while循环：node出队，判断是否要扩展/判断是否为终点*（区别会在后面介绍）*；再扩展到nbr。注意nbr修改dist的条件是new_dist < dist[nbr]。
　　上代码

```
#Graph类的dijkstra
import heapq, sys
class Vertex:
    def __init__(self, key):
        self.dist = sys.maxsize
        self.pred = None
```

```python
def dijkstra(start):
    pq = []
    start.dist = 0
    heapq.heappush(pq, (0, start))
    visited = set()

    while pq:
        cur_dist, cur = heapq.heappop(pq)
        if cur in visited:
            continue
        visited.add(cur)

        for nbr in cur.con:
            new_dist = cur_dist + cur.get(nbr)
            if nbr not in visited and new_dist < nbr.dist:  #not in visited也可以不用，反正有new_dist的判断条件    #似乎不能不用？？？
                nbr.dist = new_dist
                nbr.pred = cur
                heapq.heappush(pq, (new_dist, nbr))

#邻接表或矩阵的dijkstra. graph: {a: {b: weight}}
import heapq,sys
dist = [sys.maxsize] * n
visited = set()
def dijkstra(start: int):
    pq = []
    dist[start] = 0
    heapq.heappush(pq, (0, start))

    while pq:
        cur_dist, cur = heapq.heappop(pq)
        if cur in visited:
            continue
        visited.add(cur)

        for nbr in graph[cur]:
            new_dist = cur_dist + graph[cur][nbr]
            if new_dist < dist[nbr] and nbr not in visited:
                dist[nbr] = new_dist
                heapq.heappush(pq, (new_dist, nbr))
                #可以用list记录前驱顺序
#多源最短路——记得重置dist和visited噢~
```

visited标记问题

一般情况下，dijkstra是出队标记。但是有的时候可以不标记：

给定终点，cur==end的时候return。

剪枝的有关问题

当遇到有state的限制条件的时候，并不能像一般的dijkstra一样一定能找到**答案的**最短路(对一般的dijkstra，答案的最短路实际上就是真正的最短路)，因为限制条件可能会让实际上会走入死胡同的真正最短路进入并且标记在dist里面，就妨碍了答案的最短路中的路径入队。

`一点说明：来自题目K站中转` https://leetcode.cn/problems/cheapest-flights-within-k-stops/solutions/874532/dijkstraji-bai-100yong-hu-jie-jue-guan-f-hpmn

因此我们需要在new_dist < dist[nbr]外加入新的剪枝——用数组记录到当前站点的最小中转次数，如果new_dist≥dist[nbr]，就让k<visited[nbr]的也入队——因为如果最短路走到死胡同，那么答案最短路的部分路径(指到达当前nbr节点的路径)一定中转次数要比最短路少！

最小生成树与 $Prim$ 算法

与最短路径不同，$Prim$算法构建最小生成树——遍历所有的节点使得权值和最小。但是相同的是二者也都是用优先队列实现的。

> *上代码*

```python
#和dijkstra的差异就在存储路径上面
import heapq, sys
def prim(start):
    pq, start.dist, visited = [], 0, set()
    heapq.heappush(pq, (0, start))

    while pq:
        cur_dist, cur = heapq.heappop(pq)
        if cur in visited:
            continue
        visited.add(cur)

        for nbr in cur.con:
            weight = cur.get_nbr(nbr)
            if nbr not in visited and weight < nbr.dist:
                nbr.dist = weight
                nbr.pred = cur
                heapq.heappush(pq, (weight, nbr))
#计算最小生成树总权值
sol = 0
cur = end   #cur指向end
while cur.pred:
    sol += cur.dist
    cur = cur.pred
sol += cur.dist
```

上面的计算总权值方法需要 Vertex 的 pred属性，但是如果用矩阵图或者邻接表怎么办呢？

> 可以在出队的时候sol += dist，同时加上全部遍历一遍的判断。

```python
#注意全部遍历的判断应该放在出栈标记(和sol+=)之后，否则若所有元素只入队一次，那么最后不会返回sol(而且sol的值少一部分)
sol = 0
while pq:
    cur_dist, cur = heapq.heappop(pq)
    if cur in visited:
        continue
    visited.add(cur)
    sol += dist[cur]

    if len(visited) == n:
        return sol
```

最小生成树与 $Kruskal$ 算法

$Kruskal$ 算法利用的是并查集。其基本步骤为：

> ①将图中所有边按权重从大到小排序
> ②初始化新的优先队列存储边 `[(f, t, weight)]`，用于存储最小生成树的边
> ③重复以下步骤，直到边集中的边数等于顶点数减一或者所有边都已经考虑完毕：

- 选择排序后的边集中权重最小的边。
- 如果选择的边不会导致形成环路（即加入该边后，两个顶点不在同一个连通分量中），则将该边的两端点合并，并且把边加入最小生成树的边集中。

> *上代码*

```python
#
p = [x for x in range(n)]

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

def union(a, b):
    p[find(a)] = find(b)

#edges = [(weight, a, b)]
def kruskal(edges: list[tuple[int]]):
    edges.sort()
    totalweight = 0
    for w, a, b in edges:
        if find(a) != find(b):
            union(a, b)
            totalweight += w
            # 此处可统计添加的边数，如果添加的总边数小于 n - 1 就说明图不连通
    return totalweight
```