Cheat Shit

```
import statistics

```

1. 当然！Python 中的 `statistics` 模块提供了计算数值数据的数学统计函数。让我们来探讨一些关键的函数：
    1. `mean()`：计算数据集的算术平均值（平均值）。
    2. `median()`：查找数据集的中位数（中间值）。
    3. `mode()`：确定离散或名义数据的单一众数（最常见的值）。
    4. `stdev()`：计算数据的样本标准差。
    5. `variance()`：计算数据的样本方差。

宇宙安全声明

```
from functools import lru_cache
@lru_cache(maxsize = 128) +函数

import sys
sys.setrecursionlimit(1 << 30)

import sys
input == sys.stdin.readline
+加快读取速度

import heapq  # 堆
import itertools  # 非必要不要用这个，容易TLE
from collections import deque  # 双向队列 popleft()
import re  # 处理去吧
? *+{7}{2, }{2, 6}(ab)+|[abc]+###abc aabbcc\[^0-9]
\d\D\w\W\s\S\b\B.\.^$
from collections import defaultdict  # 一个默认有返回值的dict，有时很好用
defaultdict(int)  # values为int类
```

```
# 素数可以可用于3个因子，可以用于一些奇怪要求的题目
# 完全平方数的因子是奇数个，其他是偶数个
# 欧拉筛
import math
n = int(1e5)
ans = [False]*(n+1)
ans[1] = True
ans_list = []
for i in range(2,int(math.sqrt(n+1)+1)):
  if not ans[i]:
    for j in range(i**2,n+1,i):
      ans[j]= True
for i in range(2,n+1):
```

```
  if not ans[i]:
    ans_list.append(i)
print(ans_list)
```

```
print("%.2f" % (a/b))  #四舍五入算法
print(','.join(map(str, ans)))  #插入“, ”
print(str(5).zfill(10))  #补足0
print(f'{a:5d}')
```

```
def check(x):
    num, s = 1, 0
    for i in range(n):
        if s + expenditure[i] > x:
            s = expenditure[i]  # 装不了了
            num += 1  # 新开一个月
        else:
            s += expenditure[i]  # 向月里加天
    return [False, True][num > m]


def isvalid(former,row,col):
    for i in range(row):  # 肯定不共行，判断是否共列或共对角线
        if former[i] == col or abs(i-row) == abs(former[i]-col):
            return False
    return True


if 0 <= px <= w + 1 and 0 <= py <= h + 1 and (i, px, py) not in vis and matrix[py][px] != "X":
# 对于某个状态的时间，我们可以取模后作为 visited[x][y][time] 的第三个变量
```

```
###MergeSort
def MergeSort(lists):
    if len(lists) <= 1:
        return lists
    Mid = len(lists)//2
    Left_lists = MergeSort(lists[:Mid])
    Right_lists = MergeSort(lists[Mid:])
    return Merge(Left_lists,Right_lists)

def Merge(Left,Right):
    Sortedlist = []
    i,j = 0,0
    while i < len(Left) and j < len(Right):
        if Left[i]+Right[j] <= Right[j]+Left[i]:
            Sortedlist.append(Left[i])
            i += 1
        else:
            Sortedlist.append(Right[j])
            j += 1
```

```
    Sortedlist += Left[i:]
    Sortedlist += Right[j:]
    return Sortedlist
```

String Opt

推荐模版：24591:中序表达式转后序表达式

```
def infix_to_postfix(expression):
    stack = []
    postfix = []
    number = ""
    precedence = {"+": 1, "-": 1, "*": 2, "/": 2}

    for char in expression:
        if char.isnumeric() or char == ".":
            number += char
        else:
            if number:
                num = float(number)
                postfix.append(int(num) if num.is_integer() else num)
                number = ""
            if char in "+-*/":
                while stack and stack[-1] in "+-*/" and precedence[stack[-1]] >= precedence[char]:
                    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()
    if number:
        num = float(number)
        postfix.append(int(num) if num.is_integer() else num)

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

    return " ".join(str(x) for x in postfix)


n = int(input())
for _ in range(n):
    expression = input()
    print(infix_to_postfix(expression))
```

02694:波兰表达式

```
num = -1


def step():
    global num
    num += 1
    if opt[num] == "+":
        return step() + step()
    elif opt[num] == "-":
        return step() - step()
    elif opt[num] == "*":
        return step() * step()
    elif opt[num] == "/":
        return step() / step()
    else:
        return float(opt[num])


opt = list(map(str,input().split()))
print("%.6f"%step())
```

十六进制

```
def base_converter(dec_num, base):
    digits = "0123456789ABCDEF"

    rem_stack = [] # Stack()

    while dec_num > 0:
        rem = dec_num % base
        #rem_stack.push(rem)
        rem_stack.append(rem)
        dec_num = dec_num // base

    new_string = ""
    #while not rem_stack.is_empty():
    while rem_stack:
        new_string = new_string + digits[rem_stack.pop()]

    return new_string

print(base_converter(25, 2))
print(base_converter(2555, 16))

# 11001
# 9FB
```

DP

02757: 最长上升子序列

```python
N = int(input())
nums=list(map(int,input().split()))  # 输入一组序列
length=len(nums)
# print(n)

dp=[1]*(length+1)

for i in range(length):
    for j in range(0,i):
        if nums[i]>nums[j]:
                    # 状态：dp[i] 表示以 nums[i] 结尾的「上升子序列」的长度
                    # 当nums[i]前面存在小于nums[i]的nums[j],
                    # 则暂存在dp[j]+1就是当前nums[i]的最长增长子序列的长度
            dp[i]=max(dp[i],dp[j]+1)

print(max(dp))  # 用函数max直接找到dp数组的最大值，无需再遍历了
```

02806:公共子序列

```python
while True:
    try:
        X, Y = input().split()
        a = len(X)
        b = len(Y)
        dp = [[0 for j in range(b+1)] for i in range(a+1)]
        for i in range(1, a+1):  # 1-a的自然数列表
            for j in range(1, b+1):
                if X[i-1] == Y[j-1]:
                    dp[i][j] = dp[i-1][j-1] + 1  # 这是递进程序
                else:
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1])  # 这是回溯程序
        print(dp[a][b])
    except EOFError:
        break
```

搜索（遍历，dfs，bfs

re 统计单词数

```python
import re
re.sub(pattern, repl, string, count=0, flags=0)  # 替换
print("yes" if re.match(p, s) else "no")  #匹配

word = input().lower()
```

```
article = input().lower()

a = re.findall(r'\b'+word+r'\b', article)
cnt = len(a)
if cnt == 0:
    print(-1)
else:
    aa  = re.search(r'\b'+word+r'\b', article)
    print(cnt, aa.start())
```

最大连通域

```
temp = 0
def search(i,j):
    global temp
    temp += 1
    matrix[i][j] = "."
    for p in dfs:
        if matrix[i+p[0]][j+p[1]] == "W":
            search(i+p[0],j+p[1])

dfs = [(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1)]
T = int(input())
for _ in range(T):
    maxium = 0
    N,M = map(int,input().split())
    matrix = [["."]*(M+2)]
    for i in range(N):
        matrix.append(["."]+list(input())+["."])
    matrix.append(["."]*(M+2))
    #print(matrix)
    for i in range(1,N+1):
        for j in range(1,M+1):
            if matrix[i][j] == "W":
                temp = 0
                search(i,j)
                maxium = max(maxium,temp)
    print(maxium)
```

算n点

recursion

```
#gpt
'''
在这个优化的代码中，我们使用了递归和剪枝策略。首先按照题目的要求，输入的4个数字保持不变，
不进行排序。在每一次运算中，我们首先尝试加法和乘法，因为它们的运算结果更少受到数字大小的影响。
然后，我们根据数字的大小关系尝试减法和除法，只进行必要的组合运算，避免重复运算。

值得注意的是，这种优化策略可以减少冗余计算，但对于某些输入情况仍需要遍历所有可能的组合。
因此，在最坏情况下仍然可能需要较长的计算时间。
```

```
'''

def find(nums):
    if len(nums) == 1:
        return abs(nums[0] - 24) <= 0.000001  # <<<<<<<<<<<修改项
    for i in range(len(nums)):
        for j in range(i+1, len(nums)):
            a = nums[i]
            b = nums[j]
            remaining_nums = []
            for k in range(len(nums)):
                if k != i and k != j:
                    remaining_nums.append(nums[k])
            if find(remaining_nums + [a + b]) or find(remaining_nums + [a * b]):
                return True
            if a > b and find(remaining_nums + [a - b]):
                return True
            if b > a and find(remaining_nums + [b - a]):
                return True
            if b != 0 and find(remaining_nums + [a / b]):
                return True
            if a != 0 and find(remaining_nums + [b / a]):
                return True
    return False
n = int(input())
card = list(map(int,input().split()))
print("YES" if find(card) else "NO")
```

数据预处理

01328: Radar Installation

greedy

数据都建好了，但最后雷达的建立范围有点没搞清楚，逆序排列，这里每个pos都是局部最小值。从后往前推，如果最小值比前一个的最大值大的话，那就再建一个站，如果前一个的最大值大于最小值的话， 那就应该建在后一个的最小值到前一个最大值的范围内。又已知这里的最小值就是局部的最小值，那么把雷达建在最小值处就是最优解

```
            pos.append([float(a-(d**2-b**2)**0.5),float(a+(d**2-b**2)**0.5)])
    input()
    if len(pos) < n:
        print(f'Case {turn}: -1')
    else:
        pos.sort(reverse=True)    #<<<这里排序很重要
        number = len(pos)
        c = pos[0][0]
        for j in range(1,n):
            if c > pos[j][1]:
                c = pos[j][0]
```

```
            else:
                number -= 1
        print(f'Case {turn}: {number}')

```

只因线段覆盖

```
### 线段全覆盖 ###
N = int(input())
a = list(map(int,input().split()))
intervals = [(max(0,i-a[i]),min(N-1,i+a[i])) for i in range(N)]
intervals.sort()  <<< # 这里可能需要反转思考

ans = 0
right = 0
temp = -1
index = 0
while index < N and right < N:
    while index < N and intervals[index][0] <= right:
        temp = max(temp,intervals[index][1])
        index += 1
    right = temp + 1
    ans += 1

print(ans)
```

```
### 线段最大覆盖 ###
def generate_intervals(x, width, m):
    temp = []
    for start in range(max(0, x-width+1), min(m, x+1)):
        end = start+width
        if end <= m:
            temp.append((start, end))
    return temp

n, m = map(int, input().split())
plans = [tuple(map(int, input().split())) for _ in range(n)]
intervals = []
for x, width in plans:
    intervals.extend(generate_intervals(x, width, m))
intervals.sort(key=lambda x: (x[1], x[0]))
cnt = 0
last_end = 0
for start, end in intervals:
    if start >= last_end:
        last_end = end
        cnt += 1
print(cnt)
```

DFS

推荐模板：27310:积木

优美

```
N = int(input())
block = [set(input()) for _ in range(4)]


def dfs(word):
    if len(word) == 0:
        return True
    for i in range(4):
        if not v[i]:
            if word[0] in block[i]:
                v[i] = True

                if dfs(word[1:]):
                    return True
                # 回溯
                v[i] = False

    return False


for i in range(N):
    s = input()
    n = len(s)
    v = [False] * 4
    if dfs(s):
        print("YES")
    else:
        print("NO")
```

01084:正方形破坏者

```
# IDA搜索，全称为迭代加深A搜索（Iterative Deepening A*），是一种结合了深度优先搜索和A*搜索的算法。它通过设置一个阈值，对深度进行限制，然后进行深度优先搜索。如果在阈值内找到了目标，就直接返回结果；如果没有找到，就增加阈值，然后再次进行搜索。  IDA搜索的主要优点是它可以在有限的内存中处理大规模的问题，因为它只需要存储一条从根到叶子的路径，而不是像宽度优先搜索或A搜索那样需要存储整个搜索树。同时，它也能找到最优解，这是因为它结合了A*搜索的启发式搜索策略。  在你的代码中，estimate()函数就是IDA搜索中的估价函数，它用于估计从当前状态到目标状态的代价。在每次迭代中，dfs(t)函数会调用estimate()函数来检查当前的t（已经标记的节点数）加上estimate()的结果是否大于limit（限制）。如果大于limit，就返回，否则继续搜索。这就是IDA搜索的基本思想。
import copy
import sys
sys.setrecursionlimit(1 << 30)
found = False

```

```python
def check1(x, tmp):
    for y in graph[x]:
        if tmp[y]:
            return False
    return True

def check2(x):
    for y in graph[x]:
        if judge[y]:
            return False
    return True

def estimate(): # 估价函数，这个很好玩，给了一个估价函数，然后就可以用IDA*搜索了
    cnt = 0
    tmp = copy.deepcopy(judge)
    for x in range(1, total+1):
        if check1(x, tmp):
            cnt += 1
            for u in graph[x]:
                tmp[u] = True
    return cnt

def dfs(t):
    global found
    if t + estimate() > limit:
        return
    for x in range(1, total+1):
        if check2(x):
            for y in graph[x]:
                judge[y] = True
                dfs(t+1)
                judge[y] = False
                if found:
                    return
            return
    found = True

for _ in range(int(input())):
    n = int(input())
    lst = list(map(int, input().split()))
    d, m, nums, total = 2*n+1, lst[0], lst[1:], 0
    graph = {}
    for i in range(n):
        for j in range(n):
            for k in range(1, n+1):
                if i+k <= n and j+k <= n:
                    total += 1
                    graph[total] = []
                    for p in range(1, k+1):
                        graph[total] += [d*i+j+p, d*(i+p)+j-n, d*(i+p)+j-n+k, d*(i+k)+j+p]
    judge = [False for _ in range(2*n*(n+1)+1)]
```

```
    for num in nums:
        judge[num] = True
    limit = estimate()
    found = False
    while True:
        dfs(0)
        if found:
            print(limit)
            break
        limit += 1
```

骑士周游

```
from functools import lru_cache

# initializing
size = int(input())
matrix = [[False]*size for i in range(size)]
x, y = map(int, input().split())
dir = [(2, 1), (1, 2), (-1, 2), (-2, 1), (-2, -1), (-1, -2), (1, -2), (2, -1)]


def valid(x, y):
    return 0 <= x < size and 0 <= y < size and not matrix[x][y]


def get_degree(x, y):
    count = 0
    for dx, dy in dir:
        nx, ny = x + dx, y + dy
        if valid(nx, ny):
            count += 1
    return count


@lru_cache(maxsize = 1<<30)
def dfs(x, y, count):
    if count == size**2:
        return True

    matrix[x][y] = True

    next_moves = [(dx, dy) for dx, dy in dir if valid(x + dx, y + dy)]
    next_moves.sort(key=lambda move: get_degree(x + move[0], y + move[1]))

    for dx, dy in next_moves:
        if dfs(x + dx, y + dy, count + 1):
            return True

    matrix[x][y] = False
    return False
```

```
39
40 if dfs(x, y, 1):
41     print("success")
42 else:
43     print("fail")
```

并查集

```
1  class DisjSet:
2      def __init__(self, n):
3          # Constructor to create and
4          # initialize sets of n items
5          self.rank = [1] * n
6          self.parent = [i for i in range(n)]
7
8      def find(self, x):
9          # Find the root of the set in which element x belongs
10         if self.parent[x] != x:
11             # Path compression: Make the parent of x the root of its set
12             self.parent[x] = self.find(self.parent[x])
13         return self.parent[x]
14
15     def union(self, x, y):
16         # Perform union of two sets
17         x_root, y_root = self.find(x), self.find(y)
18
19         if x_root == y_root:
20             return
21         # Attach smaller rank tree under root of higher rank tree
22         if self.rank[x_root] < self.rank[y_root]:
23             self.parent[x_root] = y_root
24         else:
25             self.parent[y_root] = x_root
26             self.rank[x_root] += 1
27
28
29
30 # 示例用法
31 A = DisjSet(5)
32 B = DisjSet(5)
33
34 A.union(0, 1)
35 A.union(2, 3)
36
37 print(A.rank)    # 输出: [2, 1, 2, 1, 1]
38 print(A.parent)  # 输出: [0, 0, 2, 2, 4]
39 print(B.rank)    # 输出: [1, 1, 1, 1, 1]
40 print(B.parent)  # 输出: [0, 1, 2, 3, 4]
```

栈

```python
def queen_stack(n):
    stack = []  # 用于保存状态的栈
    solutions = [] # 存储所有解决方案的列表

    stack.append((0, []))  # 初始状态为第一行，所有列都未放置皇后,栈中的元素是 (row, queens) 的元组

    while stack:
        row, cols = stack.pop() # 从栈中取出当前处理的行数和已放置的皇后位置
        if row == n:    # 找到一个合法解决方案
            solutions.append(cols)
        else:
            for col in range(n):
                if is_valid(row, col, cols): # 检查当前位置是否合法
                    stack.append((row + 1, cols + [col]))

    return solutions

def is_valid(row, col, queens):
    for r in range(row):
        if queens[r] == col or abs(row - r) == abs(col - queens[r]):
            return False
    return True


# 获取第 b 个皇后串
def get_queen_string(b):
    solutions = queen_stack(8)
    if b > len(solutions):
        return None
    b = len(solutions) + 1 - b

    queen_string = ''.join(str(col + 1) for col in solutions[b - 1])
    return queen_string

test_cases = int(input())  # 输入的测试数据组数
for _ in range(test_cases):
    b = int(input())  # 输入的 b 值
    queen_string = get_queen_string(b)
    print(queen_string)
```

背包类

输入模块

```
Space,stuffNum = map(int,input().split())
worth, occupy = [0], [0]
for i in range(stuffNum):
    current_worth, current_occupy = map(int, input().split())
    worth.append(current_worth)
    occupy.append(current_occupy)
# print(worth)
```

采药一维版

```
dp = [0 for j in range(Space+1)]

for i in range(1, stuffNum+1):
    for j in range(Space,occupy[i]-1,-1):  # 反向
        if j >= occupy[i]:
            dp[j] = max(dp[j], dp[j-occupy[i]]+worth[i])
print(dp[Space])
```

采药二维版

```
dp = [[0 for j in range(T+1)] for i in range(M+1)]

for i in range(1, M+1):
    for j in range(1, T+1):
        if j < cost[i]:
            dp[i][j] = dp[i-1][j]
        else:
            dp[i][j] = max(dp[i-1][j], dp[i-1][j-cost[i]]+w[i])
print(dp[M][T])
```

完全背包

```
dp = [0 for j in range(Space+1)]

for i in range(1, stuffNum+1):
    for j in range(1,Space+1):
        if j % occupy[i] == 0:
            dp[j] = max(dp[j], dp[j-occupy[i]] + worth[i])
            print(dp)
print(dp[Space])
```

多重背包(NBA)

```
# 多重背包中的最优解问题
n = int(input())
if n % 50 != 0:
    print('Fail')
    exit()
```

```
n //= 50
nums = list(map(int, input().split()))
price = [1, 2, 5, 10, 20, 50, 100]
dp = [float('inf')] * (n + 1)
dp[0] = 0
for i in range(7):
#for i in range(6, -1, -1):
    cur_price = price[i]
    cur_num = nums[i]
    k = 1
    while cur_num > 0:  #二进制分组优化，时间缩短了将近两个数量级。
                                            #相同物品避免重复工作，「二进制分组」提高效率。
        use_num = min(cur_num, k)
        cur_num -= use_num
        for j in range(n, cur_price * use_num - 1, -1):
            dp[j] = min(dp[j], dp[j - cur_price * use_num] + use_num)
        k *= 2
if dp[-1] == float('inf'):
    print('Fail')
else:
    print(dp[-1])  # dp中包含了所有可能
```

GRAPH

dijkstra算法

```
import heapq

def dijkstra(graph, start):
    distances = {node: (float('infinity'), []) for node in graph}
    distances[start] = (0, [start])
    queue = [(0, start, [start])]
    while queue:
        current_distance, current_node, path = heapq.heappop(queue)
        if current_distance > distances[current_node][0]:
            continue
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor][0]:
                distances[neighbor] = (distance, path + [neighbor])
                heapq.heappush(queue, (distance, neighbor, path + [neighbor]))
    return distances


P = int(input())
places = {input(): i for i in range(P)}
graph = {i: {} for i in range(P)}

Q = int(input())
```

```python
for _ in range(Q):
    place1, place2, distance = input().split()
    distance = int(distance)
    graph[places[place1]][places[place2]] = distance
    graph[places[place2]][places[place1]] = distance

R = int(input())
for _ in range(R):
    start, end = input().split()
    distances = dijkstra(graph, places[start])
    path = distances[places[end]][1]
    result = ""
    for i in range(len(path) - 1):
        result += f"{list(places.keys())[list(places.values()).index(path[i])]}->({graph[path[i]][path[i+1]]})->"
    result += list(places.keys())[list(places.values()).index(path[-1])]
    print(result)
```

联通线

```python
import heapq

def prim(graph, start):
    mst = []
    used = set([start])  # 已经使用过的点
    edges = [
        (cost, start, to)
        for to, cost in graph[start].items()
    ]  # (cost, frm, to) 的列表
    heapq.heapify(edges)  # 转换成最小堆

    while edges:  # 当还有边可以选择时
        cost, frm, to = heapq.heappop(edges)    # 弹出最小边
        if to not in used:  # 如果这个点还没被使用过
            used.add(to)  # 标记为已使用
            mst.append((frm, to, cost))  # 加入到最小生成树中
            for to_next, cost2 in graph[to].items():  # 将与这个点相连的边加入到堆中
                if to_next not in used:  # 如果这个点还没被使用过
                    heapq.heappush(edges, (cost2, to, to_next))  # 加入到堆中

    return mst  # 返回最小生成树

n = int(input())
graph = {chr(i+65): {} for i in range(n)}
for i in range(n-1):
    data = input().split()
    node = data[0]
    for j in range(2, len(data), 2):
        graph[node][data[j]] = int(data[j+1])
        graph[data[j]][node] = int(data[j+1])

```

```
mst = prim(graph, 'A')  # 从A开始生成最小生成树
print(sum([cost for frm, to, cost in mst]))  # 输出最小生成树的总权值
```

词梯

```
from collections import defaultdict, deque


def visit_vertex(queue, visited, other_visited, graph):
    word, path = queue.popleft()
    for i in range(len(word)):
        pattern = word[:i] + '_' + word[i + 1:]
        for next_word in graph[pattern]:
            if next_word in other_visited:
                return path + other_visited[next_word][::-1]
            if next_word not in visited:
                visited[next_word] = path + [next_word]
                queue.append((next_word, path + [next_word]))


def word_ladder(words, start, end):
    graph = defaultdict(list)
    for word in words:
        for i in range(len(word)):
            pattern = word[:i] + '_' + word[i + 1:]
            graph[pattern].append(word)

    queue_start = deque([(start, [start])])
    queue_end = deque([(end, [end])])
    visited_start = {start: [start]}
    visited_end = {end: [end]}

    while queue_start and queue_end:
        result = visit_vertex(queue_start, visited_start, visited_end, graph)
        if result:
            return ' '.join(result)
        result = visit_vertex(queue_end, visited_end, visited_start, graph)
        if result:
            return ' '.join(result[::-1])

    return 'NO'


n = int(input())
words = [input() for i in range(n)]
start, end = input().split()
print(word_ladder(words, start, end))
```

拓扑排序：给定一个有向图，求拓扑排序序列。

输入：第一行是整数 n，表示图有 n 顶点 (1<=n<=100)，编号 1 到 n。接下来 n 行，第 i 行列了顶点 i 的所有邻点，以 0 结尾。没有邻点的顶点，对应行就是单独一个0。

输出：一个图的拓扑排序序列。如果图中有环，则输出“Loop”。

样例输入 (#及其右边的文字是说明，不是输入的一部分)：

```
5                    #5 个顶点
0                    #1 号顶点无邻点
4 5 1 0          #2 号顶点有邻点 4 5 1
1 0
5 3 0
3 0
```

样例输出

```
2 4 5 3 1
```

请对下面的解题程序进行填空

```
class Edge: # 表示邻接表中的图的边,v 是终点
    def __init__(self, v):
        self.v = v


def topoSort(G):    # G 是邻接表，顶点从 0 开始编号
    # G[i][j]是 Edge 对象，代表边 <i, G[i][j].v>
    n = len(G)
    import queue
    inDegree = [0] * n  # inDegree[i]是顶点 i 的入度
    q = queue.Queue()
    # q 是队列，q.put(x)可以将 x 加入队列，q.get()取走并返回对头元素
    # q.empty()返回队列是否为空

    for i in range(n):
        for e in G[i]:
            inDegree[e.v] += 1  # 【1 分】

    for i in range(n):
        if inDegree[i] == 0:
            q.put(i)    # 【1 分】

    seq = []
    while not q.empty():
        k = q.get()
        seq.append(k)   # 【1 分】
        for e in G[k]:
            inDegree[e.v] -= 1  # 【1 分】
            if inDegree[e.v] == 0:
```

```
                q.put(e.v)  # 【1 分】

    if len(seq) != n:   # 【1 分】
        return None
    else:
        return seq


n = int(input())
G = [[] for _ in range(n)]  # 邻接表
for i in range(n):
    lst = list(map(int, input().split()))
    print(lst)
    G[i] = [Edge(x - 1) for x in lst[:-1]]
    print(G[i])

result = topoSort(G)
if result is not None:
    for x in result:
        print(x + 1, end=" ")
else:
    print("Loop")

```

手搓

链表操作：读入一个从小到大排好序的整数序列到链表，然后在链表中删除重复的元素，使得重复的元素只保留 1 个，然后将整个链表内容输出。

输入样例：

```
1 2 2 2 3 3 4 4 6
```

输出样例:

```
1 2 3 4 6
```

请对程序填空:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

a = list(map(int, input().split()))
head = Node(a[0])
p = head

```

```
for x in a[1:]:
    p.next = Node(x)    # 【2 分】
    p = p.next

p = head
while p:
    while p.next and p.data == p.next.data: # 【2 分】
        p.next = p.next.next    #【1 分】
    p = p.next

p = head
while p:
    print(p.data, end=" ")
    p = p.next  # 【2 分】

```

无向图判定：给定一个无向图，判断是否连通，是否有回路。

输入：第一行两个整数 n,m，分别表示顶点数和边数。顶点编号从 0 到 n-1。 (1<=n<=110, 1<=m<= 10000) 接下来 m 行，每行两个整数 u 和 v，表示顶点 u 和 v 之间有边。

输出:
如果图是连通的，则在第一行输出“connected:yes",否则第一行输出“connected:no"。
如果图中有回路，则在第二行输出“loop:yes ",否则第二行输出“loop:no"。

样例输入

```
3 2
0 1
0 2
```

样例输出

```
connected:yes
loop:no
```

请进行程序填空：

```
def isConnected(G): # G 是邻接表,顶点编号从 0 开始，判断是否连通
    n = len(G)
    visited = [False for _ in range(n)]
    total = 0

    def dfs(v):
        nonlocal total
        visited[v] = True
        total += 1
        for u in G[v]:

```

```
            if not visited[u]:
                dfs(u)

    dfs(0)
    return total == n      # 【2 分】

def hasLoop(G): # G 是邻接表,顶点编号从 0 开始，判断有无回路
    n = len(G)
    visited = [False for _ in range(n)]

    def dfs(v, x): # 返回值表示本次 dfs 是否找到回路,x 是深度优先搜索树上 v 的父结点
        visited[v] = True
        for u in G[v]:
            if visited[u] == True:
                if u != x: # 【2 分】
                    return True
            else:
                if dfs(u, v):   # 【2 分】
                    return True
        return False

    for i in range(n):
        if not visited[i]:  # 【1 分】
            if dfs(i, -1):
                return True
    return False

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

if isConnected(G):
    print("connected:yes")
else:
    print("connected:no")

if hasLoop(G):
    print("loop:yes")
else:
    print("loop:no")

```

堆排序：输入若干个整数，下面的程序使用堆排序算法对这些整数从小到大排序，请填空。

程序中建立的堆是大顶堆（最大元素在堆顶）

输入样例：

```
1 3 43 8 7
```

输出样例:

```
1 3 7 8 43
```

请进行程序填空:

```python
def heap_sort(arr):
    heap_size = len(arr)

    def goDown(i):
        if i * 2 + 1 >= heap_size:  # a[i]没有儿子
            return
        L, R = i * 2 + 1, i * 2 + 2

        if R >= heap_size or arr[L] > arr[R]:   # 【1 分】
            s = L
        else:
            s = R

        if arr[s] > arr[i]:
            arr[s], arr[i] = arr[i], arr[s] # 【2 分】
            goDown(s)

    def heapify():  # 将列表 a 变成一个堆
        for k in range(len(arr) // 2 - 1, -1, -1): # 【1 分】
            goDown(k)

    heapify()
    for i in range(len(arr) - 1, -1, -1):
        arr[0], arr[i] = arr[i], arr[0] # 【1 分】
        heap_size -= 1
        goDown(0)   # 【1 分】


a = list(map(int, input().split()))
heap_sort(a)
for x in a:
    print(x, end=" ")

```

卷面写法怪异，正常写法应该是

```python
def heapify(arr, n, i):
    largest = i  # 将当前节点标记为最大值
```

```python
    left = 2 * i + 1  # 左子节点的索引
    right = 2 * i + 2  # 右子节点的索引

    # 如果左子节点存在且大于根节点，则更新最大值索引
    if left < n and arr[i] < arr[left]:
        largest = left

    # 如果右子节点存在且大于根节点或左子节点，则更新最大值索引
    if right < n and arr[largest] < arr[right]:
        largest = right

    # 如果最大值索引发生了变化，则交换根节点和最大值，并递归地堆化受影响的子树
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)


def buildMaxHeap(arr):
    n = len(arr)

    # 从最后一个非叶子节点开始进行堆化
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)


def heapSort(arr):
    n = len(arr)

    buildMaxHeap(arr)  # 构建大顶堆

    # 逐步取出堆顶元素（最大值），并进行堆化调整
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # 交换堆顶元素和当前最后一个元素
        heapify(arr, i, 0)  # 对剩余的元素进行堆化

    return arr

a = list(map(int, input().split()))
heapSort(a)
for x in a:
    print(x, end=" ")
```