2024春数算机考笔记

一、python语法

1. 欧拉筛

①输出列表的

```
def euler_sieve(n):
    is_prime, primes = [True] * (n + 1), []
    for i in range(2, n + 1):
        if is_prime[i]: primes.append(i); (for j in range(i * i, n + 1, i):
is_prime[j] = False)
    return primes
```

②输出真值表的

```
def euler_sieve(n):
    is_prime = [False, False] + [True] * (n - 1)
    for i in range(2, n + 1):
        if is_prime[i]: (for j in range(i * i, n + 1, i): is_prime[j] = False)

    return is_prime
```

2. OOP

```
class 'name of the class of certain objects':
    def __init__(self, 'attributes of the object, ...'): self.'notation' =
'attribute'
    def 'function 1'(self):
        ...
        return 'result'
    ...#when using functions in class, use the code"name.func()"
```

3. python特有的

①dic

```
dic = {'a':1, 'b':2, 'c':3}
print(dic.keys())
print(dic.values())
print(dic.items())
#输出
#dict_keys(['a', 'b', 'c'])
#dict_values([1, 2, 3])
#dict_items([('a', 1), ('b', 2), ('c', 3)])
```

②deque

```
from collections import deque
# 创建一个双端队列
d = deque([1, 2, 3, 4, 5])
# 向右循环移动 2 个位置
d.rotate(2)
print(d)  # 输出: deque([4, 5, 1, 2, 3])
# 向左循环移动 3 个位置
d.rotate(-3)
print(d)  # 输出: deque([2, 3, 4, 5, 1])
```

4. 其他

`return`在函数内跳出，`break`跳出循环，`continue`跳出这次循环，继续。

`while`在满足条件后继续，和用于可迭代对象的`for`各有千秋。

`while/for ... else ...`多重判断，不好理解所以尽量别写。

可迭代对象的索引，正数从零开始，倒数从-1开始递减。字符串也是可迭代对象。

`*list`的写法很好用也很好理解，星星符号枚举可迭代对象里面的东西。

`list = sorted(list, key = lambda x: ……)`对应的lambda函数可以有多个值，排序的时候会依先后顺序排序，就比如高考分数同分的时候先比数学，这是我们比较熟悉的例子，在python里大概可以这么写：`排名 = sorted(排名, key = lambda x: (x.总分, x.数学成绩, ……))`

二、数据结构

1. 栈stack

```
class stack:
    def __init__(self): self.items = []#用列表实现类
    def is_empty(self): return self.items == []#判断是否为空
    def push(self, item): self.items.append(item)#添加数据
    def pop(self): return self.items.pop()#弹出数据
    def peek(self): return self.items[len(self.items)-1]#查看数据
    def size(self): return len(self.items)#栈长度
```

2. 队列queue

```
class queue:
    def __init__(self): self.items = []#用列表实现类
    def is_empty(self): return self.items == []#判断是否为空
    def enqueue(self, item): self.items.insert(0, item)#添加数据
    def dequeue(self): return self.items.pop()#弹出数据
    def size(self): return len(self.items)#队列长度
```

3. 双端队列deque

```
class deque:
    def __init__(self): self.items = []#用列表实现类
    def is_empty(self): return self.items == []#判断是否为空
    def addFront(self, item): self.items.append(item)#添加数据
    def addRear(self, item): self.items.insert(0, item)#添加数据
    def removeFront(self): return self.items.pop()#弹出数据
    def removeRear(self): return self.items.pop(0)#弹出数据
    def size(self): return len(self.items)#双端队列长度
```

4. 链表linked_list

让GPT写的，prompt如下：

> 写四段代码，用oop的方式，分别是单向、双向、单环、双环链表。要求：每个类都要有添加数据、在某个数据前插入数据、在某个数据后插入数据、删除数据、查找数据的函数。每个链表都要有首，双向链表要求有尾。单向链表要求有reverse函数，将链表改为反向的链表。都要有输出链表长度的函数。所有的数据类型都要有display函数，打印出所有链节中的数据。

单向链表SinglyLinkedList

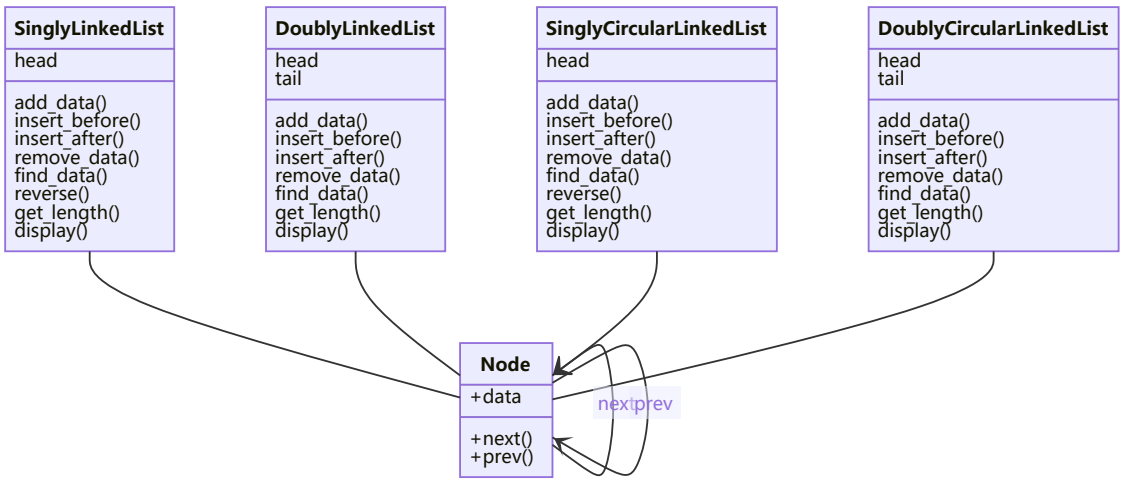
```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None
class SinglyLinkedList:
    def __init__(self):
        self.head = None
    def reverse(self):
        previous = None
        current = self.head
        while current:
            next_node = current.next
            current.next = previous
            previous = current
            current = next_node
        self.head = previous
```

双向循环链表DoubleCircularLinkedList

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None
        self.prev = None
class DoublyCircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
    def add_data(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            self.tail = new_node
```

```
            self.head.next = self.head
            self.head.prev = self.head
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            new_node.next = self.head
            self.head.prev = new_node
            self.tail = new_node
    def insert_before(self, existing_data, new_data):
        new_node = Node(new_data)
        if not self.head: return False
        current = self.head
        while current != self.tail.next:
            if current.data == existing_data:
                new_node.next = current
                new_node.prev = current.prev
                current.prev.next = new_node
                current.prev = new_node
                if current == self.head: self.head = new_node
                return True
            current = current.next
        return False
    def insert_after(self, existing_data, new_data):
        new_node = Node(new_data)
        if not self.head: return False
        current = self.head
        while current != self.tail.next:
            if current.data == existing_data:
                new_node.next = current.next
                new_node.prev = current
                current.next.prev = new_node
                current.next = new_node
                if current == self.tail: self.tail = new_node
                return True
            current = current.next
        return False
    def remove_data(self, data):
        if not self.head: return False
        current = self.head
        while current != self.tail.next:
            if current.data == data:
                if current == self.head:
                    self.head = current.next
                    self.head.prev = self.tail
                    self.tail.next = self.head
                elif current == self.tail:
                    self.tail = current.prev
                    self.tail.next = self.head
                    self.head.prev = self.tail
                else:
                    current.prev.next = current.next
                    current.next.prev = current.prev
                return True
            current = current.next
        return False
    def find_data(self, data):
        if not self.head: return False
        current = self.head
```

```
        while current != self.tail.next:
            if current.data == data: return True
            current = current.next
        return False
    def get_length(self):
        if not self.head: return 0
        count = 0
        current = self.head
        while current != self.tail: count += 1; current = current.next
        return count + 1
    def display(self):
        if not self.head: return
        current = self.head
        while current != self.tail.next: print(current.data); current =
current.next
```



5. 树tree

多叉树

```
class node():
    def __init__(self, name):
        self.name = name
        self.children = []

    def depth(self):
        if not self: return 0
        elif not (self.left or self.right): return 0
        else: return max(bnode.depth(self.left), bnode.depth(self.right)) + 1
    def leaf(self):
        if not self: return 0
        elif not (self.left or self.right): return 1
        else: return bnode.leaf(self.left) + bnode.right(self.right)

    def depth(self):
        if not children: return 0
        else: return max(node.depth(child) for child in self.children) + 1
    def leaf(self):
        if not children: return 1
```

```
        else: return sum(node.leaf(child) for child in self.children)

    def bfs(self):
        source, answer = [self], []
        while source: node = source.pop(0); answer += node.name,; source +=
node.children
        return answer
    def dfs(self):
        source, answer = [self], []
        while source: node = source.pop(); answer += node.name,; source +=
node.children[::-1]
        return answer
```

二叉树binaryTree

```
class binaryTree:
    def __init__(self, root):
        self.key = root
        self.left = None
        self.right = None

    def insertleft(self, new_node):
        if self.left = None: self.left = binaryTree(new_node)
        else: curr = binaryTree(new_node); curr.left = self.left; self.left =
curr
    def insertright(self, new_node):
        if self.right = None: self.right = binaryTree(new_node)
        else: curr = binaryTree(new_node); curr.right = self.right; self.right =
curr

    def get_left(self): return self.left
    def get_right(self): return self.right

    def set_root(self, data): self.key = data
    def get_root(self): return self.key

    def bfs(self):
        source, answer = [self], []
        while source: node = source.pop(0); answer += node.name,
            if node.left: source += node.left,
            if node.right: source += node.right,
        return answer
    def dfs(self):
        source, answer = [self], []
        while source: node = source.pop(); answer += node.name
            if node.right: source += node.right,
            if node.left: source += node.left,
        return answer

    def pre_order(self):
        if not self: return []
        return self.name + bnode.pre_order(self.left) +
bnode.pre_order(self.right)
    def in_order(self):
        if not self: return []
        return bnode.in_order(self.left) + self.name +
bnode.in_order(self.right)
```

```
    def post_order(self):
        if not self: return []
        return bnode.post_order(self.left) + bnode.post_order(self.right) +
self.name
```

二叉树特殊算法——根据前中建树、中后建树

```
class Node():
    def __init__(self, name):
        self.name = name
        self.left = []
        self.right = []

def build_tree_prein(pre_order, in_order):
    if not pre_order or not in_order: return None

    valu = pre_order[0]
    root, divi = Node(valu), in_order.index(valu)

    pre_left, pre_right = pre_order[1:divi+1], pre_order[divi+1:]
    in_left,in_right = in_order[:divi], in_order[divi+1:]
    root.left, root.right = build_tree_prein(pre_left, in_left),
build_tree_prein(pre_right, in_right)

    return root

def build_tree_inpost(in_order, post_order):
    if not in_order or not post_order: return None

    valu = post_order[-1]
    root, divi = Node(valu), in_order.index(valu)

    in_left,in_right = in_order[:divi], in_order[divi+1:]
    post_left, post_right = post_order[:divi], post_order[divi:-1]
    root.left, root.right = build_tree_inpost(in_left, post_left),
build_tree_inpost(in_right, post_right)

    return root
```

多叉树<->二叉树

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

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

def to_b_tree(t_node):
    if t_node is None: return None
    b_node = B_node(t_node.value)
    if len(t_node.children) > 0: b_node.left = to_b_tree(t_node.children[0])
```

```
    current_node = b_node.left
    for child in t_node.children[1:]: current_node.right = to_b_tree(child);
current_node = current_node.right
    return b_node

def to_tree(b_node):
    if b_node is None: return None
    t_node, child = T_node(b_node.value), b_node.left
    while child is not None: t_node.children += to_tree(child); child =
child.right
    return t_node
```

并查集disjointSet

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

    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)
        if xset == yset: return

        if self.rank[xset] < self.rank[yset]: self.parent[xset] = yset
        elif self.rank[xset] > self.rank[yset]: self.parent[yset] = xset
        else: self.parent[yset] = xset; self.rank[xset] = self.rank[xset] + 1
```

6. 图graph

内附dfs、bfs、dijkstra、prim、kurstal

```
from collections import deque
import heapq
class Vertex:
    def __init__(self, key):
        self.key = key
        self.connectedTo = {}
        self.color = "white"  # 顶点的颜色
        self.distance = float('inf')  # 顶点到起始顶点的距离，默认为无穷大
        self.previous = None  # 顶点在遍历中的前驱顶点
        self.disc = 0  # 顶点的发现时间
        self.fin = 0  # 顶点的完成时间
    def addNeighbor(self, nbr, weight=0): self.connectedTo[nbr] = weight
    def getNeighbor(self): return self.connectedTo.keys()
class Graph:
    def __init__(self):
        self.vertices = {}
        self.numVertices = 0
        self.numEdges = 0
    def add_vertex(self, key):
        self.numVertices += 1
        new_vertex = Vertex(key)
```

```
        self.vertices[key] = new_vertex
        return new_vertex
    def get_vertex(self, key): return self.vertices.get(key)
    def __len__(self): return self.numVertices
    def __contains__(self, key): return key in self.vertices
    def add_edge(self, f, t, weight=0):
        if f not in self.vertices: self.add_vertex(f)
        if t not in self.vertices: self.add_vertex(t)
        self.vertices[f].addNeighbor(self.vertices[t], weight)
        self.numEdges += 1
    def get_vertices(self): return self.vertices.keys()
    def __iter__(self): return iter(self.vertices.values())

    def dfs(self, start_vertex):
        start_vertex.color = "gray"
        for neighbor in start_vertex.getNeighbor():
            if neighbor.color == "white": self.dfs(neighbor)
        start_vertex.color = "black"
    def bfs(self, start_vertex):
        queue = deque()
        start_vertex.color = "gray"
        queue.append(start_vertex)
        while queue:
            current_vertex = queue.popleft()
            for neighbor in current_vertex.getNeighbor():
                if neighbor.color == "white": neighbor.color = "gray";
queue.append(neighbor)
            current_vertex.color = "black"

    def dijkstra(self, start_vertex):
        # 初始化距离和前驱顶点
        for vertex in self: vertex.distance = float('inf'); vertex.previous =
None
        start_vertex.distance = 0
        priority_queue = [(0, start_vertex)]
        while priority_queue:
            current_distance, current_vertex = heapq.heappop(priority_queue)
            if current_distance > current_vertex.distance: continue
            for neighbor in current_vertex.getNeighbor():
                weight = current_vertex.connectedTo[neighbor]
                distance = current_vertex.distance + weight
                if distance < neighbor.distance: neighbor.distance = distance;
neighbor.previous = current_vertex; heapq.heappush(priority_queue, (distance,
neighbor))

    def prim(self, start_vertex):
        # 初始化顶点的距离和前驱顶点
        for vertex in self: vertex.distance = float('inf'); vertex.previous =
None

        start_vertex.distance = 0
        priority_queue = [(0, start_vertex)]
        visited = set()

        while priority_queue:
            current_distance, current_vertex = heapq.heappop(priority_queue)
            if current_vertex in visited: continue
            visited.add(current_vertex)
```

```
            for neighbor in current_vertex.getNeighbor():
                weight = current_vertex.connectedTo[neighbor]
                if neighbor not in visited and weight < neighbor.distance:
neighbor.distance = weight; neighbor.previous = current_vertex;
heapq.heappush(priority_queue, (weight, neighbor))

    def kruskal(self):
        parent, rank = {}, {}
        for vertex in self: parent[vertex] = vertex; rank[vertex] = 0
        edges = []
        for vertex in self:
            for neighbor in vertex.getNeighbor(): weight =
vertex.connectedTo[neighbor]; edges.append((weight, vertex, neighbor))
        edges.sort()
        minimum_spanning_tree = Graph()
        for edge in edges:
            weight, vertex1, vertex2 = edge
            if self.find(parent, vertex1) != self.find(parent, vertex2):
minimum_spanning_tree.add_edge(vertex1.key, vertex2.key, weight);
self.union(parent, rank, vertex1, vertex2)
        return minimum_spanning_tree

    def find(self, parent, vertex):
        if parent[vertex] != vertex: parent[vertex] = self.find(parent,
parent[vertex])
        return parent[vertex]
    def union(self, parent, rank, vertex1, vertex2):
        root1, root2 = self.find(parent, vertex1); self.find(parent, vertex2)

        if rank[root1] < rank[root2]: parent[root1] = root2
        elif rank[root1] > rank[root2]: parent[root2] = root1
        else: parent[root2] = root1;rank[root1] += 1
```

有向图的拓扑排序

```
class DirectedGraph:
    def __init__(self):
        self.vertices = {}

    def add_vertex(self, vertex):
        self.vertices[vertex] = {"in_degree": 0, "out_degree": 0}

    def add_edge(self, start_vertex, end_vertex):
        if start_vertex in self.vertices and end_vertex in self.vertices:
            self.vertices[start_vertex]["out_degree"] += 1
            self.vertices[end_vertex]["in_degree"] += 1

    def remove_edge(self, start_vertex, end_vertex):
        if start_vertex in self.vertices and end_vertex in self.vertices:
            self.vertices[start_vertex]["out_degree"] -= 1
            self.vertices[end_vertex]["in_degree"] -= 1

    def get_adjacent_vertices(self, vertex):
        if vertex in self.vertices:
            adjacent_vertices = []
            for v in self.vertices:
                if self.has_edge(vertex, v):
```

```
                    adjacent_vertices.append(v)
            return adjacent_vertices

    def has_edge(self, start_vertex, end_vertex):
        if start_vertex in self.vertices and end_vertex in self.vertices:
            return self.vertices[start_vertex]["out_degree"] > 0 and
self.vertices[end_vertex]["in_degree"] > 0

    def topological_sort(self):
        in_degree_map = {v: self.vertices[v]["in_degree"] for v in
self.vertices}
        queue = [v for v, in_degree in in_degree_map.items() if in_degree == 0]
        result = []

        while queue:
            vertex = queue.pop(0)
            result.append(vertex)

            for adjacent_vertex in self.get_adjacent_vertices(vertex):
                in_degree_map[adjacent_vertex] -= 1
                if in_degree_map[adjacent_vertex] == 0:
                    queue.append(adjacent_vertex)

        if len(result) != len(self.vertices):
            # 图中存在环路，无法进行拓扑排序
            return []

        return result
```

三、算法

1. 排序

归并排序

```
def merge_count(arr, l, r):
    if l >= r:
        return 0

    mid = (l + r) // 2
    count = merge_count(arr, l, mid) + merge_count(arr, mid + 1, r)

    temp = []
    i, j = l, mid + 1
    while i <= mid and j <= r:
        if arr[i] <= arr [j]: temp.append(arr[i]); i += 1
        else: temp.append(arr[j]); j += 1; count += (mid - i + 1)

    while i <= mid: temp.append(arr[i]); i += 1
    while j <= r: temp.append(arr[j]); j += 1
    for i in range(len(temp)): arr[l + i] = temp[i]

    return count
```

2. 单调栈、单调队列

单调栈

```python
def find_left_greater(nums):
    stack = []
    result = [-1] * len(nums)
    for i in range(len(nums)):
        while stack and nums[stack[-1]] < nums[i]: result[stack.pop()] = nums[i]
        stack.append(i)
    return result

# 使用示例
nums = [2, 1, 4, 3, 5]
print(find_left_greater(nums))  # 输出: [-1, -1, 2, 2, 4]
```

单调队列

```python
from collections import deque

def max_in_window(nums, k):
    queue = deque()
    result = []
    for i in range(len(nums)):
        # 将队列中比当前元素小的都弹出
        while queue and nums[queue[-1]] < nums[i]: queue.pop()
        queue.append(i)
        # 如果窗口左边界已经不在队列中了,则将其从队列中移除
        if queue[0] == i - k: queue.popleft()
        # 如果窗口大小达到k,则将队列头部元素(即窗口内最大值)加入结果
        if i >= k - 1: result.append(nums[queue[0]])
    return result

# 使用示例
nums = [1, 3, -1, -3, 5, 3, 6, 7]
print(max_in_window(nums, 3))  # 输出: [3, 3, 5, 5, 6, 7]
```

《荀子·儒效》：“……与时迁徙，与世偃仰，千举万变，其道一也……”

20169:排队

总时间限制: 1000ms　内存限制: 65536kB

描述

操场上有好多好多同学在玩耍，体育老师冲了过来，要求他们排队。同学们纪律实在太散漫了，老师不得不来手动整队：
"A，你站在B的后面。"
"C，你站在D的后面。"
"B，你站在D的后面。哦，去D队伍的最后面。"

更形式化地，初始时刻，操场上有 n 位同学，自成一列。每次操作，老师的指令是 "x y"，表示 x 所在的队列排到 y 所在的队列的后面，即 x 的队首排在 y 的队尾的后面。（如果 x 与 y 已经在同一队列，请忽略该指令） 最终的队列数量远远小于 n，老师很满意。请你输出最终时刻每位同学所在队列的队首（排头），老师想记录每位同学的排头，方便找人。

输入

第一行一个整数 T (T≤5)，表示测试数据组数。 接下来 T 组测试数据，对于每组数据，第一行两个整数 n 和 m (n,m≤30000)，紧跟着 m 行每行两个整数
x 和 y (1≤x,y≤n)。

输出

共 T 行。 每行 n 个整数，表示每位同学的排头。

```
parent = None
def getRoot(a):
	if parent[a] != a:
		parent[a] = getRoot(parent[a])
	return parent[a]
def merge(a,b):
	pa = getRoot(a)
	pb = getRoot(b)
	if pa != pb:
		parent[pa] = parent[pb]
t = int(input())
for i in range(t):
	n, m = map(int, input().split())
	parent = [i	for i in range(n + 10)]
	for i in range(m):
		x,y = map(int,input().split())
		merge(x,y)
	for i in range(1,n+1):
		print(getRoot(i),end = " ")
		#注意，一定不能写成 print(parent[i],end= " ")
		#因为只有执行路径压缩 getRoot(i)以后，parent[i]才会是 i 的树根
	print()
```

22508:最小奖金方案

总时间限制: 1000ms　内存限制: 65536kB

描述

现在有n个队伍参加了比赛，他们进行了m次PK。现在赛事方需要给他们颁奖（奖金为整数），已知参加比赛就可获得100元，由于比赛双方会比较自己的奖金，所以获胜方的奖金一定要比败方奖金高。请问赛事方要准备的最小奖金为多少？奖金数额一定是整数。

输入

一组数据，第一行是两个整数n(1≤n≤1000)和m(0≤m≤2000)，分别代表n个队伍和m次pk，队伍编号从0到n-1。接下来m行是pk信息，具体信息a，b，代表编号为a的队伍打败了编号为b的队伍。
输入保证队伍之间的pk战胜关系不会形成有向环

输出

给出最小奖金w

```
import collections
n,m = map(int,input().split())
G = [[] for i in range(n)]
award = [0 for i in range(n)]
inDegree = [0 for i in range(n)]

for i in range(m):
        a,b = map(int,input().split())
        G[b].append(a)
        inDegree[a] += 1
q = collections.deque()
for i in range(n):
        if inDegree[i] == 0:
                q.append(i)
                award[i] = 100
while len(q) > 0:
        u = q.popleft()
        for v in G[u]:
                inDegree[v] -= 1
                award[v] = max(award[v],award[u] + 1)
                if inDegree[v] == 0:
                        q.append(v)
total = sum(award)
print(total)
```

001: PKU 版爱消除

你有一个字符串 S，大小写区分，一旦里面出现连续的 PKU 三个字符，就会消除。问最终稳定下来以后，这个字符串是什么样的？

输入
一行,一个字符串 S，表示消除前的字符串。
字符串 S 的长度不超过 100000，且只包含大小写字母。

输出
一行,一个字符串 T，表示消除后的稳定字符串

样例输入
TopSchoolPPKPPKUKUUPKUku
样例输出
TopSchoolPku

```
s = input()
stack = []
for c in s:
   if c == "U":
      if len(stack) >= 2 and stack[-1] == "K" and stack[-2] == "P":
         stack.pop()
         stack.pop()
      else:
         stack.append(c)
   else:
      stack.append(c)
print("".join(stack))
```

002: 检测括号嵌套

字符串中可能有 3 种成对的括号，"( )"、"[ ]"、"{}"。请判断字符串的括号是否都正确配对以及有无括号嵌套。无括号也算正确配对。括号交叉算不正确配对，例如"1234[78)ab]"就不算正确配对。一对括号被包含在另一对括号里面，例如"12(ab[8])"就算括号嵌套。括号嵌套不影响配对的正确性。 给定一个字符串: 如果括号没有正确配对，则输出 "ERROR" 如果正确配对了，且有括号嵌套现象，则输出"YES" 如果正确配对了，但是没有括号嵌套现象，则输出"NO"

输入
一个字符串，长度不超过 5000,仅由 ( ) [ ] { } 和小写英文字母以及数字构成

输出
根据实际情况输出 ERROR, YES 或 NO

样例输入

样例 1:
[](){}
样例 2:
[(a)]bv[]
样例 3:
[[()]]{}

样例输出
样例 1:
NO
样例 2:
YES
样例 3:
ERROR

```
def check_brackets(s):
    stack = []
    nested = False
    pairs = {')': '(', ']': '[', '}': '{'}
    for ch in s:
        if ch in pairs.values():
            stack.append(ch)
        elif ch in pairs.keys():
            if not stack or stack.pop() != pairs[ch]:
                return "ERROR"
            if stack:
                nested = True
    if stack:
        return "ERROR"
    return "YES" if nested else "NO"

s = input()
print(check_brackets(s))
```

一、几个模版

最小生成树

```
#最小生成树prim,输入用mat存编号为0~n-1的邻接表，表内元素为(x,d)x为下一个点d为权值，输出最小总权值
import heapq
def solve(mat):
    h=[(0,0)]
    n=len(mat)
    vis=[1 for i in range(n)]
    ans=0
    while h:
        (d,x)=heapq.heappop(h)
        if vis[x]:
            vis[x]=0
            ans+=d
            for (y,t) in mat[x]:
                if vis[y]:
                    heapq.heappush(h,(t,y))
    return ans
```

```
#最小生成树kruskal,输入同上
import heapq
p=[i for i in range(n)]
def find(x):
    if p[x]==x:
        return x
    p[x]=find(p[x])
    return p[x]
def union(x,y):
    u,v=find(x),find(y)
    p[u]=v
def solve(mat):
    h=[]
    ans=0
    for i in range(n):
        for (j,d) in mat[i]:
            heapq.heappush(h,(d,i,j))
    while h:
        (d,i,j)=heapq.heappop(h)
        if find(i)!=find(j):
            union(i,j)
            ans+=d
    return ans
```

003: 我想完成数算作业：代码

当卷王小 D 睡前意识到室友们每天熬夜吐槽的是自己也选了的课时，他距离早八随堂交的 ddl 只剩下了不到 4 小时。已经 debug 一晚上无果的小 D 有心要分无力做题，于是决定直接抄一份室友的作业完事。万万没想到，他们作业里完全一致的错误，引发了一场全面的作业查重……

假设 a 和 b 作业雷同，b 和 c 作业雷同，则 a 和 c 作业雷同。所有抄袭现象都会被发现，且雷同的作业只有一份独立完成的原版，请输出独立完成作业的人数

输入
第一行输入两个正整数表示班上的人数 n 与总比对数 m，接下来 m 行每行均为两个 1-n 中的整数 i 和 j，表明第 i 个同学与第 j 个同学的作业雷同。

输出
独立完成作业的人数

样例输入
3 2
1 2
1 3
样例 2:
4 2
2 4
1 3

样例输出
样例 1:
1
样例 2:
2

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

def union(parent, x, y):
    xroot = find(parent, x)
    yroot = find(parent, y)
    if xroot != yroot:
        parent[xroot] = yroot

n, m = map(int, input().split())
parent = list(range(n + 1))
for _ in range(m):
    i, j = map(int, input().split())
    union(parent, i, j)

count = sum(i == parent[i] for i in range(1, n + 1))
print(count)
```

最短路径

```
#dijkstra，输入同上，求出from和to之间的最短路径，需要保证输入无负权值，输出-1表示无法到达to
#若输入to则处理点对点问题，否则返回所有点的最短距离
import heapq
def solve(mat,f,to=-1):
    h=[(0,f)]
    n=len(mat)
    vis=[-1 for i in range(n)]
    while h:
        (d,x)=heapq.heappop(h)
        if x==to:
            return d
        if vis[x]==-1:
            vis[x]=d
            for (y,s) in mat[x]:
                if vis[y]==-1:
                    heapq.heappush(h,(d+s,y))
    return vis
```

拓扑排序

```
#输入为mat存邻接表，mat[i]为i指向的点的列表，输出-1表示有环，顺便做了输出字典序最小排序方式
import heapq
def solve(mat):
    n=len(mat)
    h=[]
    ru=[len(mat[i]) for i in range(n)]
    ans=[]
    for i in range(n):
        if ru[i]==0:
            heapq.heappush(h,i)
    while h:
        x=heapq.heappop(h)
        ans.append(x)
        for y in mat[x]:
            ru[y]-=1
            if ru[y]==0:
                heapq.heappush(h,y)
    if len(ans)<n:
        return -1
    else:
        return ans
```

008: 最小奖金方案

现在有 n 个队伍参加了比赛，他们进行了 m 次 PK。现在赛事方需要给他们颁奖（奖金为整数），已知参加比赛就可获得 100 元，由于比赛双方会比较自己的奖金，所以获胜方的奖金一定要比败方奖金高。请问赛事方要准备的最小奖金为多少？奖金数额一定是整数。

输入
一组数据，第一行是两个整数 n(1≤n≤1000)和 m(0≤m≤2000)，分别代表 n 个队伍和 m 次 pk，队伍编号从 0 到 n-1。接下来 m 行是 pk 信息，具体信息 a，b，代表编号为 a 的队伍打败了编号为 b 的队伍。
输入保证队伍之间的 pk 战胜关系不会形成有向环

输出

给出最小奖金 w

样例输入
5 6
1 0
2 0
3 0
4 1
4 2
4 3

样例输出
505

```
import collections
n,m = map(int,input().split())
G = [[] for i in range(n)]
award = [0 for i in range(n)]
inDegree = [0 for i in range(n)]

for i in range(m):
        a,b = map(int,input().split())
        G[b].append(a)
        inDegree[a] += 1
q = collections.deque()
for i in range(n):
        if inDegree[i] == 0:
                q.append(i)
                award[i] = 100
while len(q) > 0:
        u = q.popleft()
        for v in G[u]:
                inDegree[v] -= 1
                award[v] = max(award[v],award[u] + 1)
                if inDegree[v] == 0:
                        q.append(v)
total = sum(award)
print(total)
```

02945:拦截导弹

总时间限制: 1000ms　内存限制: 65536kB

描述

某国为了防御敌国的导弹袭击，开发出一种导弹拦截系统。但是这种导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都不能高于前一发的高度。某天，雷达捕捉到敌国的导弹来袭，并观测到导弹依次飞来的高度，请计算这套系统最多能拦截多少导弹。拦截来袭导弹时，必须按来袭导弹袭击的时间顺序，不允许先拦截后面的导弹，再拦截前面的导弹。

输入

输入有两行，
第一行，输入雷达捕捉到的敌国导弹的数量k（k<=25），
第二行，输入k个正整数，表示k枚导弹的高度，按来袭导弹的袭击时间顺序给出，以空格分隔。

输出

输出只有一行，包含一个整数，表示最多能拦截多少枚导弹。

```
k=int(input())
l=list(map(int,input().split()))
dp=[0]*k
for i in range(k-1,-1,-1):
```

```
    maxn=1
    for j in range(k-1,i,-1):
        if l[i]>=l[j] and dp[j]+1>maxn:
            maxn=dp[j]+1
    dp[i]=maxn
print(max(dp))
```

04147:汉诺塔问题

```
def moveOne(numDisk : int, init : str, desti : str):
    print("{}:{}->{}".format(numDisk, init, desti))

def move(numDisks : int, init : str, temp : str, desti : str):
    if numDisks == 1:
        moveOne(1, init, desti)
    else:
        move(numDisks-1, init, desti, temp)
        moveOne(numDisks, init, desti)
        move(numDisks-1, temp, init, desti)

n, a, b, c = input().split()
move(int(n), a, b, c)
```

27706:逐词倒放

总时间限制: 1000ms　内存限制: 65536kB

描述

给定一个句子，这个句子由若干个可能包含字母、数字和标点符号的单词组成，每两个单词之间有一个空格。请以单词为单位，倒序输出这个句子。

句中单词数不超过50，每个单词长度不超过10。

输入

一行，一句由若干个单词组成的句子

输出

一行，以单词为单位反转后得到的字符串

```
sentence = input().split()
reversed_sentence = ' '.join(sentence[::-1])
print(reversed_sentence)
```

27951:机器翻译

总时间限制: 1000ms　内存限制: 65536kB

描述

小晨的电脑上安装了一个机器翻译软件，他经常用这个软件来翻译英语文章。

这个翻译软件的原理很简单，它只是从头到尾，依次将每个英文单词用对应的中文含义来替换。对于每个英文单词，软件会先在内存中查找这个单词的中文含义，如果内存中有，软件就会用它进行翻译；如果内存中没有，软件就会在外存中的词典内查找，查出单词的中文含义然后翻译，并将这个单词和译义放入内存，以备后续的查找和翻译。

假设内存中有M个单元，每单元能存放一个单词和译义。每当软件将一个新单词存入内存前，如果当前内存中已存入的单词数不超过M−1，软件会将新单词存入一个未使用的内存单元；若内存中已存入M个单词，软件会清空最早进入内存的那个单词，腾出单元来，存放新单词。

假设一篇英语文章的长度为N个单词。给定这篇待译文章，翻译软件需要去外存查找多少次词典？假设在翻译开始前，内存中没有任何单词。

输入

共2行。每行中两个数之间用一个空格隔开。

第一行为两个正整数M，N，代表内存容量和文章的长度。M<=100,N<=1000

第二行为N个非负整数，按照文章的顺序，每个数（大小不超过1000）代表一个英文单词。文章中两个单词是同一个单词，当且仅当它们对应的非负整数相同。

输出

一个整数，为软件需要查词典的次数。

```
M, N = map(int, input().split())
words = list(map(int, input().split()))

memory = []

lookup_count = 0

for word in words:
    if word in memory:
        continue
    else:
        if len(memory) < M:
            memory.append(word)
        else:
            if memory:
                memory.pop(0)
            memory.append(word)
        lookup_count += 1

print(lookup_count)
```

27932:Less or Equal

总时间限制: 1000ms　内存限制: 65536kB

描述

给定一个长度为n和整数k的整数序列。请你打印出[1,10^9]范围内的**最小整数**x(即1≤x≤10^9)，使得给定序列中恰好有k个元素**小于或等于x**。

注意，序列可以包含相等的元素。
如果没有这样的x，打印"-1"(不带引号)。

输入

输入的第一行包含整数 n 和 k (1≤n≤2·10^5, 0≤k≤n)。
输入的第二行包含n个整数 a_1,a_2，…，a_n (1≤a_i≤10^9) ——序列本身。

输出

输出最小整数 x (1≤x≤10^9)，使得给定序列中恰好有k个元素小于或等于x。
如果没有这样的x，打印"-1"(不带引号)。

```
def count_elements_less_equal(sequence, x):
    count = 0
    for num in sequence:
        if num <= x:
            count += 1
    return count

n, k = map(int, input().split())
sequence = list(map(int, input().split()))

left, right = 1, 10**9

result = -1

while left <= right:
    mid = (left + right) // 2
    if count_elements_less_equal(sequence, mid) == k:
        result = mid
        right = mid - 1
    elif count_elements_less_equal(sequence, mid) < k:
        left = mid + 1
    else:
```

```
        right = mid - 1

print(result)
```

02808:校门外的树

总时间限制: 1000ms 内存限制: 65536kB

描述

某校大门外长度为L的马路上有一排树，每两棵相邻的树之间的间隔都是1米。我们可以把马路看成一个数轴，马路的一端在数轴0的位置，另一端在L的位置；数轴上的每个整数点，即0，1，2，……，L，都种有一棵树。
马路上有一些区域要用来建地铁，这些区域用它们在数轴上的起始点和终止点表示。已知任一区域的起始点和终止点的坐标都是整数，区域之间可能有重合的部分。现在要把这些区域中的树（包括区域端点处的两棵树）移走。你的任务是计算将这些树都移走后，马路上还有多少棵树。

输入

输入的第一行有两个整数L（1 <= L <= 10000）和 M（1 <= M <= 100），L代表马路的长度，M代表区域的数目，L和M之间用一个空格隔开。接下来的M行每行包含两个不同的整数，用一个空格隔开，表示一个区域的起始点和终止点的坐标。

输出

输出包括一行，这一行只包含一个整数，表示马路上剩余的树的数目。

```
l,m=list(map(int,input().split()))
lis=[1 for i in range(l+1)]
for i in range(m):
   t1,t2=list(map(int,input().split()))
   for j in range(t1,t2+1):lis[j]=0
print(sum(lis))
```

20449:是否被5整除

总时间限制: 10000ms 内存限制: 65536kB

描述

给定由0 和 1 组成的字串 A，我们定义 N_i：从 A[0] 到 A[i] 的第 i 个子数组被解释为一个二进制数

返回0和 1 组成的字串 answer，只有当 N_i 可以被 5 整除时，答案 answer[i] 为 1，否则为 0

具体请看例子

输入

一个0和1组成的字串

输出

一行长度等同于输入的0和1组成的字串

```
def binary_divisible_by_five(binary_string):
   result = ''
   num = 0
   for bit in binary_string:
      num = (num * 2 + int(bit)) % 5
      if num == 0:
         result += '1'
      else:
         result += '0'
   return result

binary_string = input().strip()
print(binary_divisible_by_five(binary_string))
```