1. 语法模板

(1) 二分查找

- 要求操作对象是**有序**序列（此处记作lst）

```
import bisect
bisect.bisect_left(lst,x)
# 使用bisect_left查找插入点，若x∈lst，返回最左侧x的索引；否则返回最左侧的使x若插入后能位于其左侧的元素的当前索引。
bisect.bisect_right(lst,x)
# 使用bisect_right查找插入点，若x∈lst，返回最右侧x的索引；否则返回最右侧的使x若插入后能位于其右侧的元素的当前索引。
bisect.insort(lst,x)
# 使用insort插入元素，返回插入后的lst
```

(2) 排列组合

```
from itertools import permutations,combinations
l = [1,2,3]
print(list(permutations(l))) # 输出：[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
print(list(combinations(l,2))) # 输出：[(1, 2), (1, 3), (2, 3)]
```

(3) OOP:

方法名	用途说明
init	构造函数，创建对象时自动调用
_de1	析构函数，对象删除前调用
str_	控制 print(obj）时的输出
repr	控制对象在解释器中的表现
Ten_	支持Ten(obj)
getitem	支持obj[key]
_setitem	支持obj[key] = value
__iter	使对象可迭代 (如用于for循环)
next	支持迭代器的下一个元素

方法名	用途说明
_ca11	使对象可以像函数一样调用
enter/exit_	用于上下文管理器 (with 语句)
eq 1t -gt ne *ge*	比较运算

方法名	用途说明
_add _sub _mul 等等	支持算术运算符重载

(4) 其它

print(f'{ans},{res:.1f}')print 是可以带 sep 和 end 参数的

可以用 round 进行四舍六入五成双的操作 (5成双仅当小数部分严格为0.5)；`math.ceil()`：向上取整；math.gcd() 最大公约数

枚举： enumerate(list), 遍历 list 中的（下标，值）对

defaultdict替代： buckets.setdefault(bucket, set()).add(word)

集合运算：并| 交& 差- ；位运算：与& 或 | 异或 ^ 非~ 左移<< 右移>> (形如a>>n)

输入：addresses=sys.stdin.read().split("\n")

list.sort(key=lambda x:x[1])

2. 一般算法

2.1 单调栈

题目：找出相邻n个长方形柱子中拼出的最大矩形

两边补0，用栈维护单调增长的柱子的高度，当高度相对大小被破坏时记录面积并和res取max

```
from typing import List
 class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        stack = []
        heights = [0] + heights + [0]
        res = 0
        for i in range(len(heights)):
            while stack and heights[i] < heights[stack[-1]]:
                h = heights[stack.pop()]
                w = i - stack[-1] - 1
                res = max(res, h * w)
            stack.append(i)
        return res
 if __name__ == '__main__':
    s = Solution()
    print(s.largestRectangleArea([2,1,5,6,2,3]))
```

2.2 调度场（中缀转后缀）

以下是 Shunting Yard 算法的基本步骤：

1. 初始化运算符栈和输出栈为空。

2. 从左到右遍历中缀表达式的每个符号。 如果是操作数（数字），则将其添加到输出栈。 如果是左括号，则将其推入运算符栈。 如果是运算符： 如果运算符的优先级大于运算符栈顶的运算符，或者运算符栈顶是左括号，则将当前运算符推 入运算符栈。 否则，将运算符栈顶的运算符弹出并添加到输出栈中，直到满足上述条件（或者运算符栈为 空）。 将当前运算符推入运算符栈。 如果是右括号，则将运算符栈顶的运算符弹出并添加到输出栈中，直到遇到左括号。将左括号弹出 但不添加到输出栈中。
3. 如果还有剩余的运算符在运算符栈中，将它们依次弹出并添加到输出栈中。
4. 输出栈中的元素就是转换后的后缀表达式

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

2.3 归并排序

数组二分，分别排序；将排序后的两子数组排序（将左右两个数组中从前往后取较小值填入原始数组）

```
def mergeSort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        L = arr[:mid] # Dividing the array elements
        R = arr[mid:] # Into 2 halves
```

```
        mergeSort(L) # Sorting the first half
        mergeSort(R) # Sorting the second half
        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
            arr[k] = R[j]
                j += 1
                k += 1
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

2.4 二分查找

利用二分查找猜测性搜索答案，然后进行验证

```
import math
n,k=[int(_) for _ in input().split()]
length=[]
base=[0]*n
for i in range(n):
    length.append(int(input()))
ans=0
if k>sum(length):
    print(0)
else:
    left=1
    right=min(length)+10000
    while left<=right:
        mid=(left+right)//2
        temp=0
        for i in range(n):
            temp+=int(length[i]/mid)
        if temp>=k:
            ans=mid
            left=mid+1
        else:
            right=mid-1
    print(ans)
```

3. 字符串/链表

3.1 KMP

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

查找字符串各个前缀最大循环节数目：

```
def max_circles(text):
    n=len(text)
    lps=compute_lps(text)
    circles=[]
    checked=set()
    for i in range(1,n//2+1):
        if lps[i*2-1]==i:
            for j in range(2,n//i+1):
                if text[:i*j].endswith(text[:i]):
                    if i*j not in checked:
                        circles.append([i*j,j])
                        checked.add(i*j)
                else:
                    break
    return circles
```

通过将lps声明为global回收内存来避免MLE

3.2 回文链表（快慢指针）

快慢指针找中间 --> 反转慢指针未遍历的后半段用B记录 --> 对比前半段和反转后的后半段

```
class Solution:
    def isPalindrome(self, head: Optional[ListNode]) -> bool:
        slow=head
        fast=head
    while fast:
        if fast.next and fast.next.next:
                    fast=fast.next.next
        else:
            break
            slow=slow.next
    A=slow
    B=ListNode(A.val)
    while A.next:
        A=A.next
        B=ListNode(A.val,B)
    past=head
    now=B
    while now:
        if past.val!=now.val:
            return False
        now=now.next
        past=past.next
    return True
```

3.3 分割回文串（dp,backtracking)

首先用数组记录[i,j]是否为回文串，同时记录i是否为长度大于1的回文串的开头（用is_start记录），同时用pre记录当前路径，用ans记录所有路径，依次尝试将[i,j]加入pre中（若i不为更长回文串开头，则只加入i以剪 枝），搜索完后删除当前搜索的切片，继续搜索，直到完成。

```
class Solution:
    def partition(self, s: str) -> List[List[str]]:
        n=len(s)
        ans=[]
        pre=[]
        huiwen=[[True]*n for _ in range(n)] #不知道英文只能拼音了
        is_start=[False]*n
        for i in range(n+1,-1,-1):
            for j in range(i+1,n):
                huiwen[i][j]=(s[i]==s[j]) and huiwen[i+1][j-1]
                is_start[i]=True
        def backtracking(index=0):
        if index==n:
            ans.append(pre.copy())
        elif is_start[index]==False:
            pre.append(s[index]) =
            backtracking(index+1)
            pre.pop()
        else:
            for j in range(index,n):
```

```
                if huiwen[index][j]:
                    pre.append(s[index:j+1])
                    backtracking(j+1)
                    pre.pop()
            backtracking()
    return ans
```

4. 图算法

4.1 拓扑排序

计算入度、重复让入度为0入队、删除入度为0的节点并更新。

4.2 强连通单元

4.2.1Kosaraju

2次dfs：第一次dfs结束时入栈记录完成时间 --> 反转图 --> 对反转图按栈内依次出栈顺序dfs，每次dfs得到一个SCC

```
def dfs1(graph, node, visited, stack):
     visited[node] = True
     for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs1(graph, neighbor, visited, stack)
     stack.append(node)
def dfs2(graph, node, visited, component):
     visited[node] = True
     component.append(node)
     for neighbor in graph[node]:
         if not visited[neighbor]:
             dfs2(graph, neighbor, visited, component)
 def kosaraju(graph):
     # Step 1: Perform first DFS to get finishing times
     stack = []
     visited = [False] * len(graph)
     for node in range(len(graph)):
        if not visited[node]:
            dfs1(graph, node, visited, stack)
     # Step 2: Transpose the graph
     transposed_graph = [[] for _ in range(len(graph))]
     for node in range(len(graph)):
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)
     # Step 3: Perform second DFS on the transposed graph to find SCCs
     visited = [False] * len(graph)
     sccs = []
     while stack:
        node = stack.pop()
        if not visited[node]:
            scc = []
            dfs2(transposed_graph, node, visited, scc)
            sccs.append(scc)
```

```
    return sccs
```

4.3 最短路

4.3.1 Dijkstra（非负）

```
from pythonds3.graphs import PriorityQueue
def dijkstra(graph,start):
    pq = PriorityQueue()
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v) for v in graph])
    while pq:
        distance, current_v = pq.delete()
        for next_v in current_v.getneighbors():
            new_distance = current_v.distance + current_v.get_neighbor(next_v)
            if new_distance < next_v.distance:
                next_v.distance = new_distance
                next_v.previous = current_v
                pq.change_priority(next_v,new_distance)
```

4.3.2 Bellman_Ford

初始化到起点距离 --> 松弛V-1次（遍历每条边，尝试缩短指向节点到原点的距离） --> **再松弛一次检测负权环**

4.3.3 多源最短路径Floyd-Warshall算法

具体步骤如下：

1. 初始化一个二维数组dist，用于存储任意两个顶点之间的最短距离。初始时，dist[i][j]表示顶点i到顶点j的直接边的权重，如果i和j不直接相连，则权重为无穷大。
2. 对于每个顶点k，在更新dist数组时，考虑顶点k作为中间节点的情况。遍历所有的顶点对(i, j)，如果通 过顶点k可以使得从顶点i到顶点j的路径变短，则更新dist[i][j]为更小的值。

   ```
   dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
   ```

3. 重复进行上述步骤，对于每个顶点作为中间节点，进行迭代更新dist数组。最终，dist数组中存储的 就是所有顶点之间的最短路径。

```
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
```

4.4 最小生成树

4.4.1 Prim

选取起点入堆 --> 每次取出到起点最近的点，将若能更新相邻节点的距离则入堆 -->直到不再更新

```
def prim(graph,start):
    pq = PriorityQueue()
    for vertex in graph:
        vertex.distance = sys.maxsize
        vertex.previous = None
    start.distance = 0
    pq.buildHeap([(v.distance,v) for v in graph])
    while pq:
        distance, current_v = pq.delete()
        for next_v in current_v.get_eighbors():
          new_distance = current_v.get_neighbor(next_v)
          if next_v in pq and new_distance < next_v.distance:
              next_v.previous = current_v
              next_v.distance = new_distance
              pq.change_priority(next_v,new_distance)
```

4.4.2 Kruskal

所有边按长度排序入堆 --> 取出最小边，若不成环则添加入图 --> 用并查集记录成环情况

```
def kruskal(graph):
    num_vertices = len(graph)
    edges = []
    # 构建边集
    for i in range(num_vertices):
        for j in range(i + 1, num_vertices):
            if graph[i][j] != 0:
                edges.append((i, j, graph[i][j]))
    edges.sort(key=lambda x: x[2])
    disjoint_set = DisjointSet(num_vertices)
    # 构建最小生成树的边集
    minimum_spanning_tree = []
    for edge in edges:
        u, v, weight = edge
        if disjoint_set.find(u) != disjoint_set.find(v):
            disjoint_set.union(u, v)
            minimum_spanning_tree.append((u, v, weight))
    return minimum_spanning_tree
```

5. 树算法

5.1 并查集

数组记录父节点 --> 向上追溯直到父节点为本身 --> 压缩路径以加快搜索

```
 def find(i):
  if Parent[i] == i:
    return i
  else:
    result = find(Parent[i])
    Parent[i] = result
    return result
```

合并/添加 i,j 间父子关系：将parent[i],parent[j] 之一指向另一个

5.2 哈夫曼编码树

```
class Node:
    def __init__(self, weight, char=None):
        self.weight = weight
        self.char = char
        self.left = None
        self.right = None
    def __lt__(self, other):
        if self.weight == other.weight:
            return self.char < other.char
            return self.weight < other.weight
def huffman_encoding(char_freq):
    heap = [Node(freq, char) for char, freq in char_freq.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(left.weight + right.weight, min(left.char, right.char))
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
    return heap[0]
def external_path_length(node, depth=0):
    if node is None:
        return 0
    if node.left is None and node.right is None:
        return depth * node.weight
        return (external_path_length(node.left, depth + 1) +
external_path_length(node.right, depth + 1)
def main():
    char_freq = {'a': 3, 'b': 4, 'c': 5, 'd': 6, 'e': 8, 'f': 9, 'g': 11, 'h':
12}
    huffman_tree = huffman_encoding(char_freq)
    external_length = external_path_length(huffman_tree)
    print("The weighted external path length of the Huffman tree is:",
external_length)
```

5.3 前中序遍历建树

前序遍历的第一个节点是根节点，在中序遍历中找到， 其左侧为左子树，右侧为右子树，然后取切片递归地建树，然后后续遍历。

```
import sys
class Treenode():
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
 def buildtree(preorder,inorder):
    if len(preorder)==0:
        return None
    root_val=preorder[0]
    root=Treenode(root_val)
    root_index=inorder.index(root_val)
    root.left=buildtree(preorder[1:root_index+1],inorder[:root_index])
    root.right=buildtree(preorder[root_index+1:],inorder[root_index+1:])
return root
def postorder(root,is_root=0):
    if root==None:
        return
    postorder(root.left)
    postorder(root.right)
    print(root.val,end='') if is_root==0 else print(root.val)
batch=[]
strings=sys.stdin.readlines()
for line in strings:
    batch.append(line.strip())
    if len(batch)==2:
        root=buildtree(batch[0],batch[1])
        postorder(root,1)
        batch=[]
```

5.4 零数组变换（贪心+优先队列）

我们肯定要在 queries 中找到至少 nums[0] 个左端点为 0 的元素保留。贪心地考虑，是选右端点最大的那些。选完之后考虑 nums[1]。前一步操作选出的部分元素，可能会有不包含下标 1 的，我们需要将它去掉。这一部分也可以用差分数组 deltaArray 来完成。此时累积的操作数可能不能将 nums[1] 降为 0，从未被挑选的元素中，选出左端点 ≤1 的部分中，选出右端点最大的一部分，直到操作数满足 nums[1] 可以降到 0。这个计算过程，可以由一个优先队列 heap 来完成。不停地遍历 nums 的过程中，将对应下标为左端点的 queries 的右端点放入 heap 中，并且当操作数不够时，从 heap 中不停取出最大的右端点，直到操作数满足要求。遍历完成后，heap 的长度就是可以删除的 queries。

```
class Solution:
    def maxRemoval(self, nums: List[int], queries: List[List[int]]) -> int:
        queries.sort(key = lambda x : x[0])
        heap = []
        deltaArray = [0] * (len(nums) + 1)
        operations = 0
        j = 0
        for i, num in enumerate(nums):
```

```
            operations += deltaArray[i]
            while j < len(queries) and queries[j][0] == i:
                heappush(heap, -queries[j][1])
                j += 1
            while operations < num and heap and -heap[0] >= i:
                operations += 1
                deltaArray[-heappop(heap) + 1] -= 1
            if operations < num:
                return -1
        return len(heap)
```