数论&小知识点

前缀和&差分数组

```
sumof=[0]*(n+10)#前缀序列
diff=[0]*(n+10)#差分序列
nums=[0]+list(map(int,input().split()))
for i in range(1,n+1):
    sumof[i]=sumof[i-1] + nums[i]
    diff[i] = nums[i]-nums[i-1]
for i in range(m):
    #差分数组实现区间加法更新
    l, r = map(int, input().split())
    diff[l] += 1
    diff[r + 1] -= 1
#对差分数组求前缀和，得到修改后每个数字
for i in range(1, n + 1):
    s[i] = s[i - 1] + diff[i]
```

并查集

```
par=[i for i in range(n+1)]     #初始化
def find_father(x):             #并查集查找函数
    if par[x]==x:
        return x
    par[x]=find_father(par[x])#路径压缩
    return par[x]
def unite(x:int,y:int):      #并查集组合函数
    fx=find_father(x) ; fy=find_father(y)
    if fx==fy:return
    else:
        par[fx]=fy
for i in range(1,n+1):
    unite(i,int(input()))#根据条件添加关系，完成合并
```

快速幂

```
#快速模幂算法

def assignment(a,m):
    x = a; n = m; y = 1         #步骤一，赋值
    while n:                    # n == 0，时自动结束算法
        if n == 0:              # 指数为1，则结果为1，直接结束算法
            break
        else:
            if n %2 == 0:       # 指数为偶数，底数a平方，指数n为原来的一半，重复此步骤直至
n = 1
                x = x * x
                n = n / 2
            else:               # 指数为奇数时，-1变成偶数，重复偶数的情况
                y = x * y
                n = n - 1
    return y
```

```python
a = 2
m = 10
b = 13
res = assignment(a,m)
print("底数为%d，指数幂为%d"%(a,m))      #偶数幂
print("指数为偶数时，快速模幂运算结果为：",res)
res1 = assignment(a,b)
print()
print("底数为%d，指数幂为%d"%(a,b))      #奇数
print("指数为奇数时，快速模幂运算结果为：",res1)
```

唯一分解定理

```python
# 质因数分解模板,p为质因数，g为指数
def factor(n:int,p:list,g:list):
    pivot = -1
    for i in range(2,int(n**0.5+2)):
        if n%i==0:
            pivot += 1
        while n%i==0:
            p[pivot] = i
            g[pivot] += 1
            n = n//i
    if n>1:
        pivot += 1
        p[pivot] = n
        g[pivot] += 1
```

质数

```python
def isPrim(n: int) -> bool:
    """判断是否是质数  O(√n)"""
    if n < 2:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True


def countPrim(n: int) -> int:
    """[0, n) 内有多少个质数 厄拉多塞筛法 O(n*(?n))"""
    count = 0
    signs = [True] * n
    for i in range(2, n):
        if signs[i]:
            count += 1
            for j in range(i + i, n, i):
                signs[j] = False
    return count
```

二分

```python
def check(k):
#根据题目要求（难点）
    return True/False
#实数搜索
def bin_float_search(l,r):
    while r-l>0.001:#精度
        mid = (l + r) / 2
        if check(mid) :
            l = mid
        else:
            r = mid
    mid = (l + r) / 2
    return mid
#整数搜索
def bin_int_search(l,r):
    while l<r:
        mid=(l+r)>>1
        if check(mid):
            r=mid
        else:
            l=mid+1
    return mid
```

阶乘

```python
math.factorial(5)
```

线段树

```python
"""
为了方便建树，这里的话我们将从1开始作为我们的下标
"""
class SegmentTree(object):
    def __init__(self, date):
        self.date = [0] + date
        self.len_date = len(self.date)
        self.tree = [self.__Node() for _ in range(4 * self.len_date)]
        self.__build(1, 1, self.len_date - 1)

    def __build(self, i, l, r):

        self.tree[i].l = l
        self.tree[i].r = r
        if (l == r):
            self.tree[i].v = self.date[r]
            return
        mid = (l + r) // 2
        self.__build(2*i, l, mid)
        self.__build(2*i+1, mid + 1, r)
        self.tree[i].v = self.tree[i * 2].v + self.tree[i * 2 + 1].v

    def search(self, i, l, r):
        if (self.tree[i].l >= l and self.tree[i].r <= r):
            return self.tree[i].v
```

```python
        if (self.tree[i].lazy != 0):
            self.__putdown(i)
        t = 0
        if (self.tree[2 * i].r >= l):
            t += self.search(2 * i, l, r)
        if (self.tree[2 * i + 1].l <= r):
            t += self.search(2 * i + 1, l, r)
        return t

    def update(self, i, l, r, k):
        if (self.tree[i].l >= l and self.tree[i].r <= r):
            self.tree[i].v += k * (self.tree[i].r - self.tree[i].l + 1)
            self.tree[i].lazy = k
            return
        if (self.tree[i].lazy != 0):
            self.__putdown(i)
        if (self.tree[2 * i].r >= l):
            self.update(2 * i, l, r, k)
        if (self.tree[2 * i + 1].l <= r):
            self.update(2 * i + 1, l, r, k)
        self.tree[i].v = self.tree[2*i].v+self.tree[2*i+1].v

    def __putdown(self, i):

        self.tree[2 * i].lazy = self.tree[i].lazy
        self.tree[2 * i + 1].lazy = self.tree[i].lazy
        mid = (self.tree[i].l + self.tree[i].r) // 2
        self.tree[2 * i].v += self.tree[i].lazy * (mid - self.tree[i].l + 1)
        self.tree[2 * i + 1].v += self.tree[i].lazy * (self.tree[i].r - mid)
        self.tree[i].lazy = 0


    class __Node():
        l: int = 0
        r: int = 0
        v: int = 0
        lazy: int = 0

        def __str__(self):
            return "left:{},right{},value:{},lazy:{}".format(self.l, self.r,
self.v, self.lazy)

if __name__ == '__main__':
    a = [1,2,3,4,5]
    seg = SegmentTree(a)
    seg.update(1,5,5,5) #从根节点开始找，更新区间为[5,5]的元素+5，也就是第五个元素+5
    print(seg.search(1, 4, 5))#从根节点开始找，查找区间为[4,5]的区间和
```

搜索

DFS

```
def dfs(层数，参数):
    if :#终止条件
        return #回溯
    for :枚举下层状态
        if:#判断当前状态是否搜说过
            True#没有，则标记该状态
            dfs(层数+1,参数)
            False#回溯
    return
```

BFS

```
start = input()
possible = []
experience = {start}
queue= [[start,0]]
while queue:
    old = queue.pop(0)
    for i in possible:
    #搜索下一步
        if  :#判断该状态是否合法
            if new_state == end:#判断是否结束搜索

                sys.exit(0)
            if new_state not in experience:#判断当前状态是否已经搜索
                experience.add(new_state)
                queue.append([new_state,step])
```

图论

Floyd算法

```
dis = [[0 for _ in range(n)] for _ in range(n)]
for k in range(n):
    for i in range(n):
        for j in range(n):
            dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j])
```

Dij算法

```
import heapq
def Dijkstra(s):
    done = [0 for  i  in range(n+1)]
    hp = []
    dis[s] = 0
    heapq.heappush(hp,(0,s))
    while hp:
        u =heapq.heappop(hp)
        if done[u]:
            continue
        done[u] = 1
        for i in range(len(G[u])):
            v,w=G[u][i]
```

```
            if done[v]:continue
            if dis[v]>dis[u]+w:
                dis[v]=dis[u]+w
                heapq.heappush(hp,(dis[v],v))
```

Bellman-Ford算法

```
def Bellman_Ford():
    for k in range(1, n + 1):
        for a, b, c in e:
            if b == n:
            dist[b] = min(dist[b], dist[a] + c)
    print(dist[n])
```

SPFA算法

```
# -*- coding: utf-8 -*-
# @Author : BYW-yuwei
# @Software: python3.8.6
import heapq
def spfa(s) :
    dis[s] = 0
    hp = []
    heapq. heappush(hp,s)
    inq=[0]*(n+1)
    inq[s]=1
    while hp:
        u = heapq. heappop(hp)
        inq[u]=0
        if dis[u]==INF :
            continue
        for v,w in e[u]:
            if dis[v] > dis[u] + w :
                dis[v] = dis[u] + w
                if inq[v]==0:
                    heapq. heappush(hp,v)
                    inq[v]=1
```

Prim算法

```
def prim_algorithm(graph):
    num_vertices = len(graph)

    # 初始化集合
    selected = set()
    selected.add(list(graph.keys())[0])  # 从第一个顶点开始
    unselected = set(graph.keys()) - selected

    # 初始化最小生成树结果
    minimum_spanning_tree = []

    while unselected:
        min_weight = float('inf')
        start_vertex = None
```

```
        end_vertex = None

        # 遍历已选集合中的每个顶点
        for vertex in selected:
            # 遍历未选集合中的每个顶点
            for neighbor, weight in graph[vertex].items():
                if neighbor in unselected:
                    # 找到权值最小的边
                    if min_weight > weight:
                        min_weight = weight
                        start_vertex = vertex
                        end_vertex = neighbor

        # 将找到的最小权值边添加到最小生成树结果中
        minimum_spanning_tree.append((start_vertex, end_vertex, min_weight))
        selected.add(end_vertex)
        unselected.remove(end_vertex)

    return minimum_spanning_tree

graph = {
    'A': {'B': 2, 'C': 9},
    'B': {'A': 2, 'D': 4, 'E': 8},
    'C': {'A': 9, 'E': 10, 'F': 3},
    'D': {'B': 4, 'E': 1, 'G': 5},
    'E': {'B': 8, 'C': 10, 'D': 1, 'F': 11, 'G': 6, 'H': 12},
    'F': {'C': 3, 'E': 11, 'H': 17},
    'G': {'D': 5, 'E': 6},
    'H': {'E': 12, 'F': 17},
}

result = prim_algorithm(graph)
print(result)
```

> [('A', 'B', 2), ('B', 'D', 4), ('D', 'E', 1), ('D', 'G', 5), ('A', 'C', 9), ('C', 'F', 3), ('E', 'H', 12)]

Kruskal算法

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

def union(parent, rank, vertex1, vertex2):
    root1 = find(parent, vertex1)
    root2 = find(parent, vertex2)

    if root1 != root2:
        if rank[root1] > rank[root2]:
            parent[root2] = root1
        else:
            parent[root1] = root2
            if rank[root1] == rank[root2]:
                rank[root2] += 1
```

```
def kruskal_algorithm(graph):
    # 初始化结果
    minimum_spanning_tree = []

    # 初始化并查集
    parent = {vertex: vertex for vertex in graph.keys()}
    rank = {vertex: 0 for vertex in graph.keys()}

    # 获取所有的边
    edges = []
    for vertex, neighbors in graph.items():
        for neighbor, weight in neighbors.items():
            edges.append((vertex, neighbor, weight))

    # 按权值排序边
    edges.sort(key=lambda edge: edge[2])

    # 不断取出权值最小的边并判断是否形成环
    for edge in edges:
        vertex1, vertex2, weight = edge
        if find(parent, vertex1) != find(parent, vertex2):
            union(parent, rank, vertex1, vertex2)
            minimum_spanning_tree.append(edge)

        if len(minimum_spanning_tree) == len(graph) - 1:
            break

    return minimum_spanning_tree

graph = {
    'A': {'B': 2, 'C': 9},
    'B': {'A': 2, 'D': 4, 'E': 8},
    'C': {'A': 9, 'E': 10, 'F': 3},
    'D': {'B': 4, 'E': 1, 'G': 5},
    'E': {'B': 8, 'C': 10, 'D': 1, 'F': 11, 'G': 6, 'H': 12},
    'F': {'C': 3, 'E': 11, 'H': 17},
    'G': {'D': 5, 'E': 6},
    'H': {'E': 12, 'F': 17},
}

result = kruskal_algorithm(graph)
print(result)
```

> [('D', 'E', 1), ('A', 'B', 2), ('C', 'F', 3), ('B', 'D', 4), ('D', 'G', 5), ('A', 'C', 9), ('E', 'H', 12)]

dp

```
# -*- coding: utf-8 -*-
# @Author : BYW-yuwei
# @Software: python3.8.6
#不滚动
T,M=map(int,input().split())
htime=[0]*(M+1)
hvalue=[0]*(M+1)
for i in range(1,M+1):
```

```
    htime[i],hvalue[i]=map(int,input().split())
dp=[[0]*(T+1) for i in range(M+1)]
for i in range(1,M+1):
    for j in range(0,T+1):
        if htime[i]>j:
            dp[i][j]=dp[i-1][j]
        else:
            npick=dp[i-1][j]
            pick=dp[i-1][j-htime[i]]+hvalue[i]
            dp[i][j]=max(npick,pick)
print(dp[M][T])


#交替滚动
T,M=map(int,input().split())
htime=[0]*(M+1)
hvalue=[0]*(M+1)
for i in range(1,M+1):
    htime[i],hvalue[i]=map(int,input().split())
dp=[[0]*(T+1) for i in range(2)]
new = 0
old = 1
for i in range(1,M+1):
    new,old = old,new
    for j in range(0,T+1):
        if htime[i]>j:
            dp[new][j]=dp[old][j]
        else:
            npick=dp[old][j]
            pick=dp[old][j-htime[i]]+hvalue[i]
            dp[new][j]=max(npick,pick)
print(dp[new][T])

#自身滚动
T,M=map(int,input().split())
htime=[0]*(M+1)
hvalue=[0]*(M+1)
for i in range(1,M+1):
    htime[i],hvalue[i]=map(int,input().split())
dp=[0]*(T+1)
for i in range(1,M+1):
    for j in range(T,htime[i]-1,-1):
        npick=dp[j]
        pick=dp[j-htime[i]]+hvalue[i]
        dp[j]=max(npick,pick)
print(dp[T])
```

01背包

```
# 从键盘输入中得到物品的体积和价值
def qu(N):
    for i in range(N):
        x = [int(j) for j in input().split()]
        v.append(x[0])
        w.append(x[1])
```

```
    return v, w


# 获取最大的价值
def max_():
    for i in range(1, n+1):  # 有几个物品可供选择
        for j in range(1, m + 1):  # 模拟背包容量从m+1
            if j < v[i-1]:  # 如果此时背包容量小于当前物品重量
                f[i][j] = f[i - 1][j]  # 不拿这个物品
            else:
                # 此时有两种选择,拿或不拿
                f[i][j] = max(f[i - 1][j], f[i - 1][j - v[i - 1]] + w[i-1])
                # 选择最好的一种方式，也就是两种情况作比较，取价值的较大值


# 取得物品的个数和背包的总体积
a = [int(i) for i in input().split()]
# 物品的个数
n = a[0]
# 背包总体积
m = a[1]
# 各个物品的体积列表
v = []
# 对应物品的价值
w = []
# 将物品的体积和价值装入列表中
qu(n)
# 模拟背包
f = [[0] * (m + 1) for _ in range(n + 1)]
# 获取最大的价值
max_()
print(f[n][m])
```

最长子序列

```
# 动态规划求解，存储解及解的计算过程
def lcs(x,y): # 求解并存储箭头方向，x，y为字符串、列表等序列
    m = len(x) # x的长度
    n = len(y) # y的长度
    c = [[0 for i in range(n+1)] for _ in range(m+1)] # 二维数组，初始值为0，用于存储长度结果
    d = [[0 for i in range(n+1)] for _ in range(m+1)] # 二维数组，初始值为0，用于存储箭头方向,1表示左上，2表示上，3表示左
    for i in range(1,m+1): # 按行遍历二维数组
        for j in range(1,n+1): # 每行的各数值遍历， c0j和ci0相关的值都为0，所以均从1开始
            if x[i - 1] == y[j - 1]: # xi=yi的情况，二维数组中i，j=0时，都为0已经确定，但字符串x，y仍需从0开始遍历
                c[i][j] = c[i - 1][j - 1] + 1 # 递推式
                d[i][j] = 1 # 箭头方向左上方
            elif c[i][j - 1] > c[i - 1][j]: # 递推式，选择更大的
                c[i][j] = c[i][j - 1]
                d[i][j] = 3 # 箭头左边
            else: # c[i-1][j] >= c[i][j-1]
                c[i][j] = c[i - 1][j]
                d[i][j] = 2 # 箭头上方
```

```
    return c[m][n], d

c, d = lcs("ABCBDAB", "BDCABA")
for _ in d:
    print(_)
```

例题

离散化+逆序对

```
def re_lst(lst: list):
    """
    该方法将传入的数组进行离散化，把lst变成一个只包含0~n-1的新数组
    :param lst:
    :return:
    """
    p = [(i, lst[i]) for i in range(0, len(lst))]
    p.sort(key=lambda x: x[1])
    for i in range(len(p)):
        lst[p[i][0]] = i


def merge_sort(lst, left, right):
    """
    归并排序
    :param lst:
    :return:
    """
    global couple
    mid = (left + right) // 2
    if left == right:
        return
    merge_sort(lst, left, mid)
    merge_sort(lst, mid + 1, right)

    i = left
    j = mid + 1
    k = left
    while i <= mid and j <= right:
        if lst[i] <= lst[j]:
            tem[k] = lst[i]
            k += 1
            i += 1
        else:
            tem[k] = lst[j]
            k += 1
            j += 1
            couple = (couple + mid - i + 1) % MOD
    while i <= mid:
        tem[k] = lst[i]
        k += 1
        i += 1
    while j <= right:
        tem[k] = lst[j]
        k += 1
```

```python
            j += 1
        for ii in range(left, right + 1):
            lst[ii] = tem[ii]


n = int(input())
first = list(map(int, input().split()))
second = list(map(int, input().split()))
# 将数组的范围限制在0~n-1
re_lst(first)
re_lst(second)
# 重新编号
dict_first = {v: i for i, v in enumerate(first)}
for i in range(n):
    first[i] = dict_first[first[i]]
    second[i] = dict_first[second[i]]
# 求second的逆序对即可，利用归并排序
tem = [0 for i in range(n)]
merge_sort(second, 0, n - 1)
print(couple)
```

谦虚数字

```python
from math import inf

k, n = map(int, input().split())
arr = list(map(int, input().split()))
count = [0] * k
res = [1]
for _ in range(n):
    ans = inf
    j = -1
    for pivot, value in enumerate(arr):
        tem = res[count[pivot]] * value
        if tem < ans:
            ans = tem
    for pivot, value in enumerate(arr):
        tem = res[count[pivot]] * value
        if tem == ans:
            count[pivot]+=1
    res.append(ans)
print(res[-1])
```