ADS_2025fall_Cheatsheet_LocallyCH

认真读题

留意变量范围，考虑算法的时间复杂度是否够用

0 少量语法

0.1 输入输出

不定行输入的几种读法：

```
#try...except捕获异常
while True:
    try:
        #正常读取
    except EOFError:
        break

#利用sys逐行读取
import sys
for line in sys.stdin:
    line = line.strip()
    #处理每一行

#同样适用于一般的输入读取特别是大量数据输入流优化
data = sys.stdin.read()
lines = data.splitlines()
for line in lines:
    #处理每一行
```

注意：当然，最后一种方法中也可以使用 $data.split()$ 后用 $data[idx]$ 来得到输入中的各个项。

0.2 可变与不可变对象

常用不可变对象：整数，浮点数，字符串，元组 等。

常用可变对象：列表，字典，集合。

重要：可变对象作为参数传入函数，若在函数内被修改，会影响外部变量。

0.3 OOP

可以参考1.2部分的并查集代码。

1 若干值得一抄的常用技术/方法

1.1 二分查找：一类二分查找问题的通法

不失一般性，我们假设：$key(arr[idx])$ 关于 idx 单调增加。否则，若单调减少，取负的函数值即可。

1.1.1 寻找arr中从左往右第一个使得key函数值>x（或>=x）的元素

直接参考bisect库源码即可。注意“循环不变量”的思想：要保证目标下标始终位于闭区间 $[lo, hi]$ 中，这样更新边界的时候就不会搞错“加不加1”“减不减1”这类问题。

```
#寻找第一个使key值>x的元素，返回a中下标
#若无满足条件的下标，返回len(a)
def bisect_strict_first(a, key):
    lo, hi = 0, len(a)
    while lo < hi:
        mid = (lo + hi) // 2
        if key(a[mid]) > x:
            hi = mid
        else:
            lo = mid + 1
    return lo

#寻找第一个使key值>=x的元素，返回a中下标
#若无满足条件的下标，返回len(a)
def bisect_nonstrict_first(a, key):
    lo, hi = 0, len(a)
    while lo < hi:
        mid = (lo + hi) // 2
        if key(a[mid]) < x:
            lo = mid + 1
        else:
            hi = mid
    return lo
```

注意： 该源码实际上可以用于寻找合适的插入x的位置，即使数组中元素全部<=x（或全部<x），此代码仍然有效。初始化 hi 为 $len(a)$ 保证了这一点。

稍加修改，我们就能得到另一类镜像问题的解法：

1.1.2 寻找arr中从左往右最后一个使得key函数值<x（或<=x）的元素

```
#寻找最后一个使key值<x的元素，返回a中下标
#若无满足条件的下标，返回-1
def bisect_strict_last(a, key):
    lo, hi = -1, len(a) - 1
    while lo < hi:
        mid = (lo + hi + 1) // 2
        if key(a[mid]) < x:
            lo = mid
        else:
            hi = mid - 1
    return hi

#寻找最后一个使key值《=x的元素，返回a中下标
#若无满足条件的下标，返回-1
def bisect_nonstrict_last(a, key):
    lo, hi = -1, len(a) - 1
    while lo < hi:
        mid = (lo + hi + 1) // 2
        if key(a[mid]) > x:
            hi = mid - 1
        else:
            lo = mid
    return hi
```

1.2 并查集

基本用法可参考M02524: 宗教信仰 一题的代码：

```
class UnionFind:
    def __init__(self,n):
        self.parent=list(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 union(self,x,y):#按秩合并
        rx,ry = self.find(x),self.find(y)
        if rx == ry:
            return False
        if self.rank[rx] < self.rank[ry]:
            self.parent[rx] = ry
        elif self.rank[rx] > self.rank[ry]:
            self.parent[ry] = rx
        else:
```

```
            self.parent[ry] = rx
            self.rank[rx]+=1
        return True
case = 1
while True:
    n,m = map(int,input().split())
    if n == 0 and m == 0:
        break
    else:
        uf = UnionFind(n+1)
        for _ in range(m):
            p,q = map(int,input().split())
            uf.union(p,q)
        #注意！计数需要先使用find，确保同一类的元素的parent为同一代表元
        myset = {uf.find(x) for x in range(1,n+1)}
        print(f"Case {case}: {len(myset)}")
        case+=1
```

1.3 双指针

1.3.1 一种双指针的简单用法

双指针可以减少很多无效的暴力枚举。有时可以用for循环来起到快指针的作用，例如（题目为Leetcode 283: 移动零）：

```
class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        size = len(nums)
        p1 = 0
        for p2 in range(size):
            if nums[p2] != 0:
                nums[p1], nums[p2] = nums[p2], nums[p1]
                p1+=1
```

1.3.2 滑动窗口

一个较为清晰的写法如下（Leetcode 209: 长度最小的子数组）：

```
class Solution:
    def minSubArrayLen(self, target: int, nums: List[int]) -> int:
        s = 0
        size = len(nums)
        minlen = size + 1
        left = 0
```

```
        for right in range(size):#每次将右边界扩展一位
            s += nums[right]
            #尽可能缩小窗口
            while s >= target:
                minlen = min(minlen, right - left + 1)
                s-=nums[left]
                left+=1
        return minlen if minlen < size + 1 else 0
```

2 递归

2.1 回溯

回溯算法的一般步骤如下面伪代码所示。需要注意以下几点：

- 回溯函数返回值以及参数
- 回溯函数终止条件
- 回溯搜索的遍历过程（横向和纵向）

```
def backtracking(parameters):
    if ... : #终止条件
        #存放结果
        return
    for ... : #选择本层集合中的元素
        #处理节点
        backtracking(...) #参数一般为：路径，以及做过的选择的信息
        #撤销处理结果
```

记得考虑能否通过剪枝（去掉不可能的选择）来提高效率。

3. DP

4. 简单图论

4.1 深度优先搜索

可以参考递归、回溯部分内容。

4.2 广度优先搜索

4.3 Dijkstra算法

稀疏图宜用如下堆优化版Dijkstra算法并配合邻接表：

```python
import heapq
from collections import defaultdict
m, n, p = map(int, input().split())
mat = []
for i in range(m):
    row = input().split()
    for idx, item in enumerate(row):
        if item != '#':
            row[idx] = int(item)
    mat.append(row)

for i in range(p):
    sx, sy, ex, ey = map(int, input().split())
    if mat[sx][sy] == '#' or mat[ex][ey] == '#':
        print("NO")
        continue
    costs = [[float('inf')] * n for _ in range(m)]
    costs[sx][sy] = 0
    graph = defaultdict(list)#邻接表存图
    for r in range(m):
        for c in range(n):
            if mat[r][c] == '#':
                continue
            else:
                direls = [(1, 0), (0,1), (-1,0), (0, -1)]
                for dire in direls:
                    tx, ty = r + dire[0], c + dire[1]
                    if 0 <= tx < m and 0 <= ty < n and mat[tx][ty] != '#':
                        edge = (tx, ty, abs(mat[r][c] - mat[tx][ty]))
                        graph[(r,c)].append(edge)

    pq = []
    heapq.heappush(pq, (0, sx, sy))
    costs[sx][sy] = 0
    while pq:
        cur_dist, cx, cy = heapq.heappop(pq)
        for info in graph[(cx, cy)]:
            nx, ny, dist = info
            if cur_dist + dist < costs[nx][ny]:
                costs[nx][ny] = cur_dist + dist
                heapq.heappush(pq, (costs[nx][ny], nx, ny))
    if costs[ex][ey] == float('inf'):
        print("NO")
```

```
    else:
        print(costs[ex][ey])
```