DAY 03

1.Smallest string after swaps

```
 class UnionFind:
    def _init_(self, n):
        self.parent = list(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):
        rootX = self.find(x)
        rootY = self.find(y)
        if rootX != rootY:
            self.parent[rootY] = rootX
def smallestStringWithSwaps(s, pairs):
    n = len(s)
    uf = UnionFind(n)
    for a, b in pairs:
        uf.union(a, b)
    from collections import defaultdict
    components = defaultdict(list)
    res = list(s)
    for comp in components.values():
        indices = sorted(comp)
        chars = sorted(res[i] for i in indices)
        for i, char in zip(indices, chars):
            res[i] = char
```

```
    return ''.join(res)
s = "dcab"
pairs = [[0, 3], [1, 2]]
print(smallestStringWithSwaps(s, pairs))
```

2.Check if one permutation can break

```
def checkIfCanBreak(s1,s2):
    s1_sorted=sorted(s1)
    s2_sorted=sorted(s2)
    def can_break(x, y):
        return all(x[i] >= y[i] for i in range(len(x)))
    return can_break(s1_sorted,s2_sorted) or can_break(s2_sorted,s1_sorted)
s1 = "abc"
s2 = "xya"
print(checkIfCanBreak(s1,s2))
```

3.Minimize value of string with ‘?’

```
def minimizeCost(s):
    res=[]
    last_seen={}
    alphabet = 'abcdefghijklmnopqrstuvwxyz'
    for char in s:
        if char=='?':
            min_cost_char=None
            min_cost=float('inf')
            for letter in alphabet:
```

```
            cost=last_seen.get(letter, 0)
            if cost < min_cost:
                min_cost=cost
                min_cost_char=letter
        res.append(min_cost_char)
        last_seen[min_cost_char]=last_seen.get(min_cost_char,0)+1
    else:
        res.append(char)
        last_seen[char]=last_seen.get(char,0)+1
  return ''.join(res)
s = "a?b?c?"
print(minimizeCost(s))
```

4.Last string value before emptying

```
def lastStringBeforeEmptying(s):
  while True:
    new_s = list(s)
    for c in 'abcdefghijklmnopqrstuvwxyz':
      if c in new_s:
        new_s.remove(c)
    new_s = ''.join(new_s)
    if new_s == s:
      return s
    s = new_s
s = "aabcbbca"
print(lastStringBeforeEmptying(s))
```

5.Maximum subarray

```
def maxSubArray(nums):
    max_current = max_global = nums[0]
    for num in nums[1:]:
        max_current = max(num, max_current + num)
        if max_current>max_global:
            max_global=max_current
    return max_global
nums=[-2,1,-3,4,-1,2,1,-5,4]
print(maxSubArray(nums))
```

6.Maximum binary tree

```
class TreeNode:
    def _init_(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def constructMaximumBinaryTree(nums):
    if not nums:
        return None
    max_val=max(nums)
    max_index = nums.index(max_val)
    root = TreeNode(max_val)
    root.left=constructMaximumBinaryTree(nums[:max_index])
    root.right=constructMaximumBinaryTree(nums[max_index+1:])
    return root
```

```
nums=[3,2,1,6,0,5]
root=constructMaximumBinaryTree(nums)
```

7.Mamimumsum of circular subarray

```
def maxSubArray(nums):
    max_current=max_global=nums[0]
    for num in nums[1:]:
        max_current=max(num,max_current+num)
        if max_current>max_global:
            max_global=max_current
    return max_global
def maxSubarraySumCircular(nums):
    total_sum=sum(nums)
    max_kadane=maxSubArray(nums)
    min_kadane=-maxSubArray([-num for num in nums])
    if min_kadane==total_sum:
        return max_kadane
    return max(max_kadane,total_sum+min_kadane)
nums=[1, -2, 3, -2]
print(maxSubarraySumCircular(nums))
```

8.Max sum of non-adjacent subsequence after queries

```
def maxNonAdjacentSum(nums):
    include,exclude=0,0
    for num in nums:
        new_exclude=max(include,exclude)
        include=exclude+num
```

```
        exclude=new_exclude
    return max(include,exclude)
def processQueries(nums,queries):
    MOD=10**9 + 7
    total_sum=0
    for pos, val in queries:
        nums[pos]=val
        total_sum=(total_sum+ maxNonAdjacentSum(nums))%MOD
    return total_sum
nums=[1, 2, 3, 4]
queries=[[1, 3],[2, 4]]
print(processQueries(nums,queries))
```

9.K closest points to origin

```
import heapq
def kClosest(points, k):
    max_heap=[]
    for x, y in points:
        dist=-(x * x + y * y)
        if len(max_heap)<k:
            heapq.heappush(max_heap,(dist,x,y))
        else:
            heapq.heappushpop(max_heap,(dist,x,y))
    return [(x, y) for _, x, y in max_heap]
points=[[1, 3], [-2, 2], [2, -2]]
k=2
```

```
print(kClosest(points,k))
```

10.Median of two sorted arrays

```
def findMedianSortedArrays(nums1, nums2):
    if len(nums1) > len(nums2):
        nums1,nums2=nums2,nums1
    m, n=len(nums1),len(nums2)
    imin, imax, half_len=0, m, (m + n + 1) // 2
    while imin <= imax:
        i=(imin + imax)// 2
        j=half_len-i
        if i < m and nums1[i]<nums2[j-1]:
            imin = i + 1
        elif i > 0 and nums1[i-1]>nums2[j]:
            imax = i - 1
        else:
            if i==0: max_of_left=nums2[j-1]
            elif j==0:max_of_left=nums1[i-1]
            else: max_of_left=max(nums1[i-1],nums2[j-1])
            if (m + n) % 2==1:
                return max_of_left
            if i==m: min_of_right=nums2[j]
            elif j==n: min_of_right=nums1[i]
            else: min_of_right = min(nums1[i],nums2[j])
            return (max_of_left + min_of_right)/2.0
nums1 = [1, 3]
nums2 = [2]
```

```
print(findMedianSortedArrays(nums1,nums2))
```