1. You are given a string s, and an array of pairs of indices in the string pairs where pairs[i] = [a, b] indicates 2 indices(0-indexed) of the string.You can swap the characters at any pair of indices in the given pairs any number of times. Return the lexicographically smallest string that s can be changed to after using the swaps.

```
from collections import defaultdict

def smallestStringWithSwaps(s, pairs):
    def find(x):
        if parent[x] != x:
            parent[x] = find(parent[x])
        return parent[x]

    def union(x, y):
        rootX = find(x)
        rootY = find(y)
        if rootX != rootY:
            parent[rootY] = rootX

    n = len(s)
    parent = list(range(n))

    for a, b in pairs:
        union(a, b)

    groups = defaultdict(list)
    for i in range(n):
        root = find(i)
        groups[root].append(i)

    result = list(s)
    for indices in groups.values():
        chars = sorted(result[i] for i in indices)
        for i, char in zip(sorted(indices), chars):
            result[i] = char

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

output

abcd

time complexity

O(n log n)

2. Given two strings: s1 and s2 with the same size, check if some permutation of string s1 can break some permutation of string s2 or vice-versa. In other words s2 can break s1 or vice-versa. A string x can break string y (both of size n) if x[i] >= y[i] (in alphabetical order) for all i between 0 and n-1.

```
def checkIfCanBreak(s1, s2):
```

```
s1 = sorted(s1)
s2 = sorted(s2)

def canBreak(x, y):
    return all(a >= b for a, b in zip(x, y))

return canBreak(s1, s2) or canBreak(s2, s1)

s1 = "abc"
s2 = "xya"
print(checkIfCanBreak(s1, s2))
```

output

True

Time complexity

O(n log n)

3. You are given a string s. s[i] is either a lowercase English letter or '?'. For a string t having length m containing only lowercase English letters, we define the function cost(i) for an index i as the number of characters equal to t[i] that appeared before it, i.e. in the range [0, i - 1]. The value of t is the sum of cost(i) for all indices i. For example, for the string t = "aab":

cost(0) = 0

cost(1) = 1

cost(2) = 0

Hence, the value of "aab" is 0 + 1 + 0 = 1. Your task is to replace all occurrences of '?' in s with any lowercase English letter so at the value of s is minimized.

```
def minimizeCost(s):

  from collections import defaultdict


  def calculate_value(t):

    counts = defaultdict(int)

    cost = 0

    for c in t:

      cost += counts[c]

      counts[c] += 1

    return cost


  def dfs(i):

    if i == len(s):
```

```
            return 0

        if s[i] != '?':
            return dfs(i + 1)

        min_cost = float('inf')
        for c in 'abcdefghijklmnopqrstuvwxyz':
            s[i] = c
            min_cost = min(min_cost, calculate_value(s[:i+1]) + dfs(i + 1))
            s[i] = '?'

        return min_cost

    s = list(s)
    return dfs(0)

s = "a?b?c"
print(minimizeCost(s))
```

output

0

Time complexity

O(n^k)

4. You are given a string s. Consider performing the following operation until s becomes empty: For every alphabet character from 'a' to 'z', remove the first occurrence of that character in s (if it exists). For example, let initially s = "aabcbbca". We do the following operations: Remove the underlined characters s = "aabcbbca". The resulting string is s = "abbca". Remove the underlined characters s = "abbca". The resulting string is s = "ba". Remove the underlined characters s = "ba". The resulting string is s = "". Return the value of the string s right before applying the last operation. In the example above, answer is "ba".

```
def findStringBeforeLastOperation(s):
    import collections
    s = list(s)
    while s:
        counter = collections.Counter(s)
        for char in 'abcdefghijklmnopqrstuvwxyz':
            if char in counter:
```

```
            s.remove(char)
        if len(set(s)) == len(s):
            break
    return ''.join(s)

s = "aabcbbca"
print(findStringBeforeLastOperation(s))
```

output

ba

time complexity

O(n^2)

5. Given an integer array nums, find the subarray with the largest sum, and return its sum.

Example 1:

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]

Output: 6

Explanation: The subarray [4,-1,2,1] has the largest sum 6.

```
def maxSubArray(nums):

    current_sum = max_sum = nums[0]

    for num in nums[1:]:

        current_sum = max(num, current_sum + num)

        max_sum = max(max_sum, current_sum)

    return max_sum


nums = [-2,1,-3,4,-1,2,1,-5,4]

print(maxSubArray(nums))
```

output

6

Time complexity

O(n)

6. You are given an integer array nums with no duplicates. A maximum binary tree can be built recursively from nums using the following algorithm: Create a root node whose value is the maximum value in nums. Recursively build the left subtree on the subarray prefix to the left of the maximum value. Recursively build the right subtree on the subarray suffix to the right of the maximum value. Return the maximum binary tree built from nums.

```
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_index = nums.index(max(nums))


    root = TreeNode(nums[max_index])


    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)

def preorderTraversal(root):
    if root:
        print(root.val, end=' ')
        preorderTraversal(root.left)
        preorderTraversal(root.right)

preorderTraversal(root)
```

output

6 3 2 1 5 0

Time complexity

O(n^2)

7. Given a circular integer array nums of length n, return the maximum possible sum of a non-empty subarray of nums.A circular array means the end of the array connects to the beginning of the array. Formally, the next element of nums[i] is nums[(i + 1) % n] and the previous element of nums[i] is nums[(i - 1 + n) % n].A subarray may only include each element of the fixed buffer nums at most once. Formally, for a subarray nums[i], nums[i + 1], ..., nums[j], there does not exist i <= k1, k2 <= j with k1 % n == k2 % n.

```
def maxSubarraySumCircular(nums):
    def kadane(nums):
        current_sum = max_sum = nums[0]
        for num in nums[1:]:
            current_sum = max(num, current_sum + num)
            max_sum = max(max_sum, current_sum)
        return max_sum
```

```
    total_sum = sum(nums)
    max_kadane = kadane(nums)

    nums = [-num for num in nums]
    max_wraparound = total_sum + kadane(nums)

    return max(max_kadane, max_wraparound) if max_wraparound != 0 else max_kadane


nums = [1,-2,3,-2]
print(maxSubarraySumCircular(nums))
```
output

3

Time complexity

O(n)

8. You are given an array nums consisting of integers. You are also given a 2D array queries, where queries[i] = [posi, xi].For query i, we first set nums[posi] equal to xi, then we calculate the answer to query i which is the maximum sum of a subsequence of nums where no two adjacent elements are selected. Return the sum of the answers to all queries. Since the final answer may be very large, return it modulo 109 + 7. A subsequence is an array that can be derived from another array by deleting some or no elements without changing the order of the remaining elements.

```
def maxSumAfterQueries(nums, queries):
    def maxNonAdjacentSum(nums):
        incl, excl = 0, 0
        for num in nums:
            new_excl = max(excl, incl)
            incl = excl + num
            excl = new_excl
        return max(incl, excl)

    total_sum = 0
    for pos, xi in queries:
        nums[pos] = xi
        total_sum += maxNonAdjacentSum(nums)
    return total_sum % (10**9 + 7)

nums = [1,2,3,4]
queries = [[0,2],[1,3],[2,4]]
print(maxSumAfterQueries(nums, queries))
```
output

20

time complexity

O(q.n)

9. Given an array of points where points[i] = [xi, yi] represents a point on the X-Y plane and an integer k, return the k closest points to the origin (0, 0).The distance between two points on the X-Y plane is the Euclidean distance (i.e., √(x1 - x2)2 + (y1 - y2)2). You may return the

answer in any order. The answer is guaranteed to be unique (except for the order that it is in).

```
import heapq

def kClosest(points, k):
   points.sort(key=lambda point: point[0]**2 + point[1]**2)
   return points[:k]

points = [[1,3],[-2,2]]
k = 1
print(kClosest(points, k))
```

output

[[-2, 2]]

Time complexity

O(log(min(m,n)))

10. Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

```
def findMedianSortedArrays(nums1, nums2):
   if len(nums1) > len(nums2):
      nums1, nums2 = nums2, nums1

   x, y = len(nums1), len(nums2)
   low, high = 0, x

   while low <= high:
      partitionX = (low + high) // 2
      partitionY = (x + y + 1) // 2 - partitionX

      maxX = float('-inf') if partitionX == 0 else nums1[partitionX - 1]
      minX = float('inf') if partitionX == x else nums1[partitionX]

      maxY = float('-inf') if partitionY == 0 else nums2[partitionY - 1]
      minY = float('inf') if partitionY == y else nums2[partitionY]

      if maxX <= minY and maxY <= minX:
         if (x + y) % 2 == 0:
            return (max(maxX, maxY) + min(minX, minY)) / 2
         else:
            return max(maxX, maxY)
      elif maxX > minY:
         high = partitionX - 1
      else:
         low = partitionX + 1

nums1 = [1, 3]
nums2 = [2]
print(findMedianSortedArrays(nums1, nums2))
```

output
2

Time complexity

O(nlogn)