swap the characters at any pair of indices in the given pairs any number of times.

In [1]:

```
import collections
def smallestStringWithSwaps(s, pairs):
    def find(x):
        if x != parent[x]:
            parent[x] = find(parent[x])
        return parent[x]
    def union(x, y):
        root_x, root_y = find(x), find(y)
        if rank[root_x] > rank[root_y]:
            parent[root_y] = root_x
        else:
            parent[root_x] = root_y
            if rank[root_x] == rank[root_y]:
                rank[root_y] += 1
    parent = list(range(len(s)))
    rank = [0] * len(s)
    for a, b in pairs:
        union(a, b)
    groups = collections.defaultdict(list)
    for i in range(len(s)):
        groups[find(i)].append(s[i])
    for group in groups:
        groups[group].sort(reverse=True)
    res = []
    for i in range(len(s)):
        res.append(groups[find(i)].pop())
    return ''.join(res)
s = "dcab"
pairs = [[0, 3], [1, 2]]
output = smallestStringWithSwaps(s, pairs)
print("Output:", output)
```

```
Output: bacd
```

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.

In [2]:

```
def check_if_can_break(s1, s2):
    s1_sorted = sorted(s1)
    s2_sorted = sorted(s2)
    if all(s1_char >= s2_char for s1_char, s2_char in zip(s1_sorted, s2_sor
        return True
    else:
        return False
s1 = "abc"
s2 = "xya"
result = check_if_can_break(s1, s2)
print(result)
```

```
True
```

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

In [3]:

```
def cost(i, t):
    return sum(1 for j in range(i) if t[j] == t[i])
t = "aab"
value_of_t = sum(cost(i, t) for i in range(len(t)))
print(value_of_t)
```

```
1
```

4.remove the first occurence

In [5]:

```
def remove_letters(s):
    for char in sorted(set(s)):
        s = s.replace(char, '', 1)
    return s
s = "abbca"
result = remove_letters(s)
print(result)
```

```
ba
```

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

In [6]:

```
def max_subarray_sum(nums):
    max_sum = float('-inf')
    current_sum = 0
    for num in nums:
        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(max_subarray_sum(nums))
```

```
6
```

In [19]:

```
 #Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def maxPathSum(self, root: TreeNode) -> int:
        # Helper function to perform depth-first search
        def dfs(node: TreeNode) -> int:
            # Base case: if the current node is None, return 0
            if not node:
                return 0

            # Recursively calculate the maximum path sum on the left subtree
            # If negative, we take 0 to avoid decreasing the overall path su
            left_max = max(0, dfs(node.left))

            # Similarly, do the same for the right subtree
            right_max = max(0, dfs(node.right))

            # Update the overall maximum path sum
            # This includes the node value and the maximum paths from both s
            nonlocal max_path_sum
            max_path_sum = max(max_path_sum, node.val + left_max + right_max

            # Return the maximum path sum without splitting
            # The current node's value plus the greater of its left or right
            return node.val + max(left_max, right_max)

        # Initialize the overall maximum path sum to negative infinity
        # To account for potentially all negative-valued trees
        max_path_sum = float('-inf')

        # Start DFS with the root of the tree
        dfs(root)

        # After DFS is done, max_path_sum holds the maximum path sum for the
        return max_path_sum
```

7. Given a circular integer array nums of length n, return the maximum possible sum of a non-empty subarray of nums

In [9]:

```
def maxSubarraySumCircular(nums):
    total_sum = max_sum = min_sum = max_temp = min_temp = nums[0]
    for num in nums[1:]:
        max_temp = max(num, max_temp + num)
        max_sum = max(max_sum, max_temp)
        min_temp = min(num, min_temp + num)
        min_sum = min(min_sum, min_temp)
        total_sum += num
    return max(max_sum, total_sum - min_sum) if max_sum > 0 else max_sum
nums = [1, -2, 3, -2]
output = maxSubarraySumCircular(nums)
print("Output:", output)
```

```
Output: 3
```

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

In [10]:

```
def maxSumAfterQuery(nums, queries):
    MOD = 10**9 + 7
    res = 0
    for x, (i, v) in enumerate(queries):
        nums[i] = v
        dp = [0] * len(nums)
        for j, num in enumerate(nums):
            dp[j] = num
            for k in range(j - 2, -1, -2):
                dp[j] = max(dp[j], dp[k] + num)
        res = (res + max(dp)) % MOD
    return res
nums = [1, 2, 3, 4, 5]
queries = [[1, 2], [4, 6]]
print(maxSumAfterQuery(nums, queries))
```

```
19
```

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).

In [11]:

```
import heapq
def kClosest(points, k):
    return heapq.nsmallest(k, points, key=lambda x: x[0]*2 + x[1]*2)
points = [[1,3],[-2,2]]
k = 1
output = kClosest(points, k)
print("Output:", output)
```

```
Output: [[-2, 2]]
```

10. Given two sorted arrays nums1 and nums2 of size m and n respectively,

In [13]:

```
def findMedianSortedArrays(nums1, nums2):
    nums = sorted(nums1 + nums2)
    n = len(nums)
    if n % 2 == 0:
        return (nums[n // 2 - 1] + nums[n // 2]) / 2
    else:
        return nums[n // 2]
nums1 = [1, 3]
nums2 = [2]
output = findMedianSortedArrays(nums1, nums2)
print("ans:", output)
```

```
ans: 2
```

In [26]:

```
initial_list = ["sachin", "dhoni"]
print ("The original list before sorting is: ",initial_list)
initial_list.sort()
print ("The original list after sorting is: ",initial_list)
```

```
The original list before sorting is:  ['sachin', 'dhoni']
The original list after sorting is:  ['dhoni', 'sachin']
```

In [29]:

```
fruit1 = input('Enter the 1st name :\n')
fruit2 = input('Enter the second name:\n')
if fruit1 < fruit2:
    print("fruit2 is greater")
elif fruit1 > fruit2:
    print("fruit1 is greater")
else:
    print("same")
```

```
Enter the 1st name :
orange
Enter the second name:
grape
fruit1 is greater
```

```
# time complexity
```

In [ ]:

```
1.o(nlogn)
2.o(nlogn)
3.o(n)
4.o(n)
5.o(n)
6.o(nlogn)
7.o(n)
8.o(n^2)
9.o(nlogk)
10.o(log min(m,n))
```