SEARCHING & SORTING - SET 3 SOLUTIONS

Note: All solutions are written in C++.

Question 1:

- **Problem link**: https://leetcode.com/problems/sort-array-by-parity-ii/
- **Difficulty level**: Easy

Explanation:

Use two pointers to search for misplaced odd and even elements, and swap them.

Solution:

```
vector<int> sortArrayByParityII(vector<int>& A) {
    for (int i = 0, j = 1; i < A.size(); i += 2, j += 2) {
        while (i < A.size() && A[i] % 2 == 0) i += 2;
        while (j < A.size() && A[j] % 2 == 1) j += 2;
        if (i < A.size()) swap(A[i], A[j]);
    }
    return A;
}
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 2:

- **Problem link**: https://leetcode.com/problems/sorting-the-sentence/
- **Difficulty level**: Easy

Explanation:

By traversing the sentence extract each word with appending one extra space at the end of the word. And store these words to the respective position of the last number of each word into an array string. Concatenate all words one by one and remove the last space of the sentence.

Solution:

```
class Solution {
public:
    string sortSentence(string s) {
        int n = s.length();
        vector<string> pos(10 , "");

        for(int i=0; i<n; i++){
            string temp;
            while(!isdigit(s[i])){
                temp.push_back(s[i++]);
            }
            temp.push_back(' ');
            pos[s[i]-'0']=temp;
            i++;
        }
        string res;
        for(int i=1; i<10; i++)
              if(pos[i].length()>0)
                  res+=pos[i];

        res.pop_back();
```

```
        return res;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(n)

Question 3:

- **Problem link**: https://leetcode.com/problems/find-all-numbers-disappeared-in-an-array/
- **Difficulty level**: Easy

Explanation:

Since the numbers of the array are in the range of the size of the array and all the numbers will be positive, we can change the sign of the indices of the occurring elements. After traversing the entire array, the indices which have positive elements are the missing numbers.

Solution:

```
class Solution {
public:
    vector<int> findDisappearedNumbers(vector<int>& nums) {
        int n=nums.size();
        for(int i=0;i<n;i++)
        {
            if(nums[abs(nums[i])-1]>0)
            {
                nums[abs(nums[i])-1]*=-1;
            }
```

```
        }
        vector<int>res;
        for(int i=0;i<n;i++)
        {
            if(nums[i]>0)
            {
                res.push_back(i+1);
            }
        }
        return res;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(1) as the output array does not count.

Question 4:

- **Problem link**: https://leetcode.com/problems/minimum-deletions-to-make-character-frequencies-unique/
- **Difficulty level**: Medium

Explanation:

Maintain an array 'count' in which the number of occurrences of each alphabet is stored. Sort the 'count' array. Traverse the 'count' array from the second last element to the 1st element. If for an index 'i', arr[i] >= arr[i+1], change the value of arr[i] to arr[i+1]-1 if arr[i+1] is non zero. Else make arr[i] = 0. Record the number of deletions in each case.

Solution:

```
class Solution {
public:
    int minDeletions(string s) {

        vector<int> count(26,0);
        int ALPHABET = 26;

        for(int i=0; i<s.length(); i++)
            count[s[i]-'a']++;

        sort(count.begin(), count.end());
        int deletions = 0;

        for(int i=ALPHABET-2; i>=0; i--)
        {
            if(count[i] == 0)
                break;

            if(count[i] >= count[i+1])
            {
                int temp = count[i];

                if(count[i+1] == 0)
                {
                    count[i] = 0;
                }
                else
                {
                    count[i]= count[i+1]-1;
                }

                deletions += (temp-count[i]);
            }
        }
```

```
        return deletions;
    }
};
```

Complexity:

- Time: O(n)
  Sorting a vector of length always takes constant time. So, the time complexity only involves the time to count the number of occurrences of each alphabet.
- Space: O(1)
  Vector of fixed size 26 is considered to be constant space only.

Question 5:

- **Problem link**: https://leetcode.com/problems/find-right-interval/
- **Difficulty level**: Medium

Explanation:

Maintain a vector of pairs 'p' in which the 'start' and index of each interval is stored as a pair. Sort this vector of pairs 'p'. Traverse through each interval present in the list 'intervals' and use binary search to find the index of the pair in 'p' whose 'start' is just greater than or equal to the 'end' of the current interval. If this index of the pair is in between 0 and size of intervals-1, then, the index stored as the second element of the pair is the answer. Else the answer is -1.

Solution:

```
class Solution {
public:
    vector<int> findRightInterval(vector<vector<int>>& intervals) {

        int n= intervals.size();
```

```
        vector<pair<int,int>> p;
        vector<int> output;

        for(int i=0; i<n; i++)
        {
            p.push_back(make_pair(intervals[i][0], i));
        }

        sort(p.begin(), p.end());

        for(int i=0; i<n; i++)
        {
            int low = 0;
            int high = n-1;

            while(low<=high)
            {
                int mid = low + (high-low)/2;

                if(p[mid].first == intervals[i][1])
                {
                    low = mid;
                    break;
                }
                else if(p[mid].first < intervals[i][1])
                {
                    low = mid+1;
                }
                else
                {
                    high = mid-1;
                }
            }

            if(low>=0 && low<n)
```

```
                output.push_back(p[low].second);
            else
                output.push_back(-1);
        }
        return output;
    }
};
```

Complexity:

- Time: O(n log n)
- Space: O(n)

Question 6:

- **Problem link**: https://leetcode.com/problems/search-a-2d-matrix-ii/
- **Difficulty level**: Medium

Explanation:

Search from the top right or bottom left element. If starting from top right, if the target is greater than the current element, search the next row else search the previous column.

Solution:

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int l=0,r=matrix[0].size()-1;
        while(l<matrix.size() and r>=0)
        {
            if(matrix[l][r]==target)
            {
```

```
                return true;
            }
            if(matrix[l][r]>target)
            {
                r--;
            }
            else
            {
                l++;
            }
        }
        return false;
    }
};
```

Complexity:

- Time: O(m+n) for a matrix with m rows and n columns.
- Space: O(1)

Question 7:

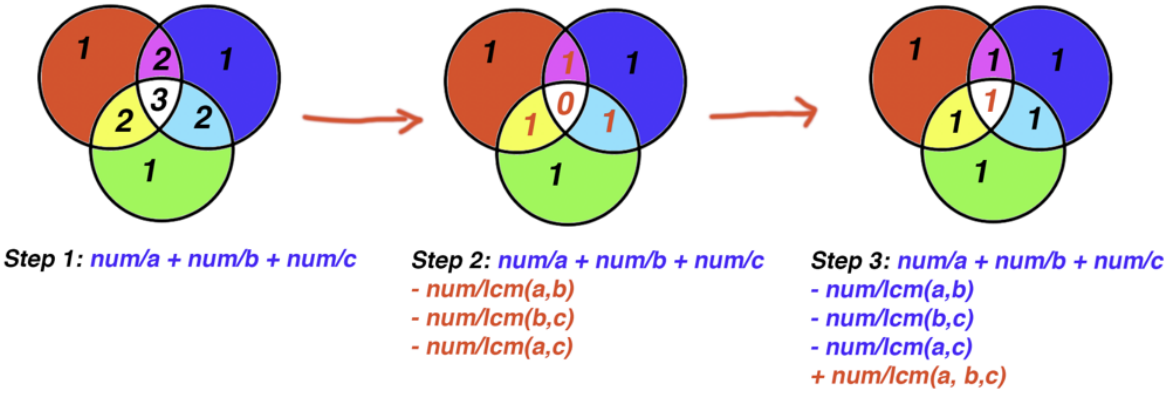
- **Problem link**: https://leetcode.com/problems/ugly-number-iii/
- **Difficulty level**: Medium

Explanation:

Calculate how many numbers from `1` to `num` are divisible by either `a`, `b` or `c` by using below formula:

`num/a + num/b + num/c – num/lcm(a, b) – num/lcm(b, c) – num/lcm(a, c) + num/lcm(a, b, c)`

Formula intuition :



The key idea is that the range of searching is monotonic, i.e., If F(a) == true, then for every b > a, F(b) = true. So our goal is to find the leftmost point at which F(a) == true, which can be solved by binary search.

Solution:

```
typedef long long ll;
#define MAX_ANS 2e9 // 2 * 10^9

class Solution {
public:
    int nthUglyNumber(int n, int a, int b, int c) {
        int left = 0, right = MAX_ANS, result = 0;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (count(mid, a, b, c) >= n) { // find mid as small as
possible that count == n
                result = mid;
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        return result;
    }
    int count(ll num, ll a, ll b, ll c) {
```

```
        return (int)(num / a + num / b + num / c
            - num / lcm(a, b)
            - num / lcm(b, c)
            - num / lcm(a, c)
            + num / (lcm(a, lcm(b, c))));
    }
    ll gcd(ll a, ll b) {
        if (a == 0) return b;
        return gcd(b % a, a);
    }
    ll lcm(ll a, ll b) {
        return a * b / gcd(a, b);
    }
};
```

Complexity:

- Time: O(log(max_ans)) [max_ans=2*10^9]
- Space: O(1)

Question 8:

- **Problem link**: https://leetcode.com/problems/split-array-largest-sum/
- **Difficulty level**: Hard

Explanation:

Intuition is the same as set2 8th question.

The final result is in the interval [left, right] (where left is the maximal number in the array, right is sum of all numbers).

So, what we need to do is to find out the first element in [left, right], which exactly we cannot split the array into m subarrays whose sum is no greater than that element. Then its previous one is the final result.

Solution:

```
class Solution {
public:
    int splitArray(vector<int>& nums, int m) {
        int left = 0, right = 0;
        for (int num : nums) {
            left = max(left, num);
            right += num;
        }
        while (left <= right) {
            int mid = left + (right-left)/2;
            if (canSplit(nums, m, mid)<=m)
                right = mid-1;
            else
                left = mid+1;
        }
        return left;
    }

 int canSplit(vector<int>& nums, int m, int sum) {
        int c = 1;
        int s = 0;
        for (int num : nums) {

   s += num;
            if (s > sum) {
                s = num;
                c++;
            }
        }
        return c;
    }

};
```

Complexity:

- Time: O(nlog(n))
- Space: O(1)