STRINGS - SET 3 SOLUTIONS

Note: All solutions are written in C++.

Question 1:

- **Problem link**: https://practice.geeksforgeeks.org/problems/roman-number-to-integer3201/1
- **Difficulty level**: Easy

Explanation:

Create a function **value()** that is used to return the values of ROMAN numerical symbols i.e; I, V,X,L,C, D & M. Now traverse the array character by character. If the current value of the symbol is greater than or equal to the value of the next symbol, then add this value to the running total else subtract this value by adding the value of the next symbol to the running total

Solution:

```
int value(char r)
{
      if (r == 'I')
      return 1;
      if (r == 'V')
      return 5;
      if (r == 'X')
      return 10;
      if (r == 'L')
      return 50;
      if (r == 'C')
      return 100;
```

```
      if (r == 'D')
      return 500;
      if (r == 'M')
      return 1000;

      return -1;
}
int romanToDecimal(string &str) {

   int n=str.length();
   int res=0;
   for (int i = 0; i < str.length(); i++)
      {
      int s1 = value(str[i]);
      if (i + 1 < str.length())
      {
            int s2 = value(str[i + 1]);

            if (s1 >= s2)
            {

                  res = res + s1;
            }
            else
            {
                  res = res + s2 - s1;
                  i++;
            }
      }
      else {
            res = res + s1;
      }
      }
      return res;
}
```

Complexity:

- Time: O(|S|)
- Space: O(1)

Question 2:

- **Problem link**: https://practice.geeksforgeeks.org/problems/twice-counter4236/1
- **Difficulty level**: Easy

Explanation:

The given problem can be solved using the hash table concept.

1. Traverse the given array, store counts of words in a hash table
2. Traverse hash table and count all words with count 2.

Solution:

```
class Solution
{
   public:
      int countWords(string list[], int n) {
         unordered_map<string, int> m;
         for (int i = 0; i < n; i++)
             m[list[i]] += 1;
         int res = 0;
         for (auto it = m.begin(); it != m.end(); it++)
            if ((it->second == 2))
               res++;
         return res;
      }
};
```

Complexity:

- Time: O(N)
- Extra space: O(N)

Question 3:

- **Problem link**: https://practice.geeksforgeeks.org/problems/largest-number-with-given-sum-1587115620/1
- **Difficulty level**: Easy

Explanation:

A **Simple Solution** is to consider all N digit numbers and keep track of maximum number with digit sum as S. A close upper bound on time complexity of this solution is $O(10^N)$.

There is a **Greedy approach** to solve the problem. The idea is to one by one fill all digits from leftmost to rightmost (or from most significant digit to least significant). We compare the remaining sum with 9 if the remaining sum is more than 9, we put 9 at the current position, else we put the remaining sum. Since we fill digits from left to right, we put the highest digits on the left side, hence get the largest number.

Solution:

```
class Solution
{
    public:
    string largestNumber(int n, int sum) {

        if (sum > 9*n) {
            return "-1";
        }
```

```
        int res[n];

        for (int i = 0; i < n; i++) {
            if (sum >= 9) {
                res[i] = 9;
                sum -= 9;
            }

            else {
                res[i] = sum;
                sum = 0;
            }
        }
        std::string str;
        for (int i: res) {
            str.push_back(i + '0');
        }
        return str;
    }
};
```

Complexity:

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

Question 4:

- **Problem link**: https://leetcode.com/problems/replace-all-digits-with-characters
- **Difficulty level**: Easy

Explanation: the i th index of the string s is checked and if it is a digit, it is made to append to the previous character which is at the (i-1)th index and then we subtract the character '0' from it. We subtract '0' from the character because 'charecter' - '0' = integer value of the character. Eg. 'g' - '0' = 7 (integer value). Thus we start from index 1 then append it to the previous character and then subtract '0' . basically the formula is string[i] = string[i-1] + string[i] -'0'

Solution:

```
class Solution
{
      public:
          string replace(string s)
      {
            for(int i=1;i<s.length();i++)
            {
                  if(isdigit(s[i]))
                        s[i]=s[i-1]+s[i]-'0';
            }
            return s;
            }
};
```

Complexity:

- Time: O(N)
- Space: O(N)

Question 5:

- **Problem link**: https://practice.geeksforgeeks.org/problems/edit-distance3702/1
- **Difficulty level**: Medium

Explanation:

Edit distance is a standard Dynamic Programming problem. E[i][j] represents the Edit distance between str1[0….i] and str2[0...j].When i is 0 i.e str1 is empty, the cost needed is to add all elements of str2[0...j] i.e. E[i][j]=j.The similar case happens when i is 0. At some point if both characters str1[i] and str2[j] are same, then E[i][j] would be E[i-1][j-1] as no operation needs to be performed when both are same.But incase they are unequal, minimum of incase of deletion, insertion and replacement is taken as the value of E[i][j]. Return E[n1][n2] to get the edit distance for the strings str1[0….n1] and str2[0…..n2].

Solution:

```
class Solution {

  public:
     int editDistance(string str1, string str2) {
     int m=str1.length();
     int n=str2.length();
     int E[m + 1][n + 1];
     for (int i = 0; i <= m; i++) {
           for (int j = 0; j <= n; j++) {
                 if (i == 0)
                 E[i][j] = j;

                 else if (j == 0)
                 E[i][j] = i;

                 else if (str1[i - 1] == str2[j - 1])
                 E[i][j] = E[i - 1][j - 1];

                 else
                 E[i][j]= 1+ min(E[i][j - 1],min(E[i - 1][j], E[i -
1][j - 1])); // Replace
           }
     }
```

```
        return E[m][n];
        }
};
```

Complexity:

- Time: O(|s|*|t|)
- Space: O(|s|*|t|)

Question 6:

- **Problem link**: https://practice.geeksforgeeks.org/problems/smallest-distant-window/0
- **Difficulty level**: Medium

Explanation:

Here we have used Sliding Window technique to arrive at the solution. This technique shows how a nested for loop in few problems can be converted to a single for loop and hence reducing the time complexity.

Basically a window of characters is maintained by using two pointers namely **start** and **end**. These **start** and **end** pointers can be used to shrink and increase the size of the window respectively. Whenever the window contains all characters of a given string, the window is shrunk from the left side to remove extra characters and then its length is compared with the smallest window found so far.

If in the present window, no more characters can be deleted then we start increasing the size of the window using the **end** until all the distinct characters present in the string are also there in the window. Finally, find the minimum size of each window.

Algorithm:

1. Maintain an array **(visited)** of maximum possible characters (256 characters) and as soon as we find any in the string, mark that index in the array **(this is to count all distinct characters in the string).**
2. Take two pointers **start** and **end** which will mark the start and end of the window.
3. Take a **counter=0** which will be used to count distinct characters in the window.
4. Now start reading the characters of the given string and if we come across a character which has not been visited yet increment the counter by **1**.
5. If the **counter** is equal to the total number of distinct characters, Try to shrink the window.
6. **For shrinking the window -:**
    1. If the **frequency** of character at start pointer is **greater than 1** increment the pointer as it is redundant.
    2. Now compare the length of the present window with the minimum window length.

Solution:

```
class Solution{
    public:
    string findSubString(string str) {
        int n = str.length();
        int MAX_CHARS = 256;
        int dist_count = 0;
        bool visited[MAX_CHARS] = { false };
        for (int i = 0; i < n; i++) {
            if (visited[str[i]] == false) {
                visited[str[i]] = true;
                dist_count++;
            }
        }

        int start = 0, start_index = -1, min_len = INT_MAX;
```

```
        int count = 0;
        int curr_count[MAX_CHARS] = { 0 };
        for (int j = 0; j < n; j++) {
            curr_count[str[j]]++;

            if (curr_count[str[j]] == 1)
                count++;

            if (count == dist_count) {
                while (curr_count[str[start]] > 1) {
                    if (curr_count[str[start]] > 1)
                        curr_count[str[start]]--;
                    start++;
                }
                int len_window = j - start + 1;
                if (min_len > len_window) {
                    min_len = len_window;
                    start_index = start;
                }
            }
        }
        return str.substr(start_index, min_len);
    }
};
```

Complexity:

- Time: O(256.N)
- Space: O(256)

Question 7:

- **Problem link**: https://leetcode.com/problems/bulb-switcher-iv/

- **Difficulty level**: Medium

Explanation:

We first start of with an initial configuration of the string as "000...0". We iterate through each index starting from 0, and we first subtract '0' from that character at that position in the string. Then we check if it is true(i.e if the result of the subtraction of '0' does not give f (f is a bool variable which can If at any instant value of f=false it means after the previous transformations on initial string "000..00", the value at that position will be 0. Similarly if f=true, value at that index will be 1. Thus finally we increment the count of the toggle we are doing. This will be the minimum number of flip operations required.

For eg. let target configuration be "101",

Initially string is "000" and f = false

Now at index 0 we have '1', '1' - '0' = 1. Thus result holds 'true' since result is 'true' the if condition is executed, and now f = 'true'. Hence count becomes 1 now. Now in the next iteration, result = 'false' (because '0' - '0' = false) thus we go to next iteration and since now result becomes 'true' again, hence f = 'false' now and counter is incremented again and hence becomes 2. Thus now we reached end of string and we have minimum number of flips required as 2.

Solution:

take values 'true' and 'false'). If it does not give f, we toggle the value of f because

```
class Solution {
public:
    int minimum(string config)
   {
        bool f=false,result;
        int count=0,i;
        for(i=0;i<config.length();i++)
        {
            bool result=config[i]-'0';
            if(result!=f)
             f=!f,count++;
```

```
        }
        return count;
    }
};
```

Complexity:

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

Question 8:

- **Problem link**: https://leetcode.com/problems/maximum-product-of-word-lengths/
- **Difficulty level**: Hard

Explanation:

Here we shall assign a numerical value to each word and store it in a vector mask.This numerical value will help us determine if two strings have a character or not. This value can be obtained by ORing 1<<(c-'a') for each character in a given string. If two such strings have the same value then it means they both have a common character. Find the max of product of such all possibilities and return the answer.

Solution:

```
int maxProduct(vector<string>& words) {
    vector<int> mask(words.size());
    int result = 0;
    for (int i=0; i<words.size(); ++i) {
        for (char c : words[i])
            mask[i] |= 1 << (c - 'a');
        for (int j=0; j<i; ++j)
            if (!(mask[i] & mask[j]))
```

```
                result = max(result, int(words[i].size() *
words[j].size()));
    }
    return result;
}
```

Complexity:

- Time: O(N^2)
- Space: O(N)

Question 9:

- **Problem link**: https://leetcode.com/problems/special-binary-string/
- **Difficulty level**: Hard

Explanation:

A brief description on how to solve this problem is by using the concept of divide and conquer(don't worry if you are unaware of this term, in short, it means we break a bigger problem into smaller subproblems and then solve the subproblems individually and then merge it back together). In this problem we break down the input string into a set of special binary strings, then we use divide and conquer and recursively solve each of the smaller binary strings. Then we sort those substrings in an order such that the substrings with more preceding 1's comes first and so on. Then we merge those substrings to get our final answer.

Solution:

```
    bool compare(const string& s1, const string& s2)
      { return s1 + s2 > s2 + s1; }

class Solution {
 public:
  string makeLargestSpecial(string s) {
    if (s.size() <= 2) return s;
    int balance = 0, start = 0;
    vector<string> subs;
    for (int i = 0; i < s.size(); i++) {
      balance += (s[i] - '0' == 0) ? -1 : 1;
      if (balance == 0) {
        subs.emplace_back(s.substr(start, i - start + 1));
        start = i + 1;
      }
    }
    for (int i = 0; i < subs.size(); i++) {
      subs[i] = '1' + makeLargestSpecial(subs[i].substr(1,
subs[i].size() - 2)) + '0';
    }
    sort(subs.begin(), subs.end(), compare);
    string ret = subs[0];
    for (int i = 1; i < subs.size(); i++) {
      ret += subs[i];
    }
    return ret;
  }
};
```

Complexity:

- Time: O(n^2)
- Space: O(n)