Blind 75

Bhanu Prasanna Koppolu

2024-02-09

Table of contents

Blind 75

I want to improve my problem solving skills. I am starting with Blind 75.

Language Used - C++

Two Sum - Easy

Notes

The optimal solution uses a Unordered Map which is internally built using Hash Table concepts.

Usage of Hash Table enables the least cost of operations like search, delete, and insert.

Program

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> s;

        for (int i = 0; i < nums.size(); i++) {
            int com = target - nums[i];
            if (s.find(com) != s.end()) {
                return {s[com], i};
            }
            s.insert({nums[i], i});
        }
        return {};
    }
};
```

Contains Duplicate - Easy

Notes

The optimal solution uses a Unordered Set which is internally built using Hash Table concepts.

Usage of Hash Table enables the least cost of operations like search, delete, and insert.

Program

```
class Solution {
public:
    bool containsDuplicate(vector<int>& nums) {
        unordered_set<int> us;

        for (int i : nums) {
            if (us.find(i) != us.end()) {
                return true;
            }
            us.insert(i);
        }

        return false;
    }
};
```

Valid Anagram - Easy

Notes

The optimal solution is to count the number of characters in each string for the 26 alphabets.

Program

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        if (s.size() != t.size()) {
            return false;
        }

        int c1[26] = {0};
        int c2[26] = {0};

        for (int i = 0; i < s.size(); i++) {
            c1[(s[i] - 97)]++;
            c2[(t[i] - 97)]++;
        }

        for (int i = 0; i < 26; i++) {
            if (c1[i] != c2[i]) {
                return false;
            }
        }

        return true;
    }
};
```

Group Anagrams - Medium

Notes

The optimal solution uses the same 26 alphabet count method but additionally uses a map to store the same type anagrams.

Program

```
class Solution {
private:
    string getKey (string str) {
        vector<int> v(26,0);

        for (int i = 0; i < str.size(); i++) {
            v[str[i] - 'a']++;
        }

        string s = "";

        for (int i = 0; i < 26; i++) {
            s += '#' + to_string(v[i]);
        }

        return s;
    }
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        unordered_map<string, vector<string>> um;

        for (int i = 0; i < strs.size(); i++) {
            string k = getKey(strs[i]);
            um[k].push_back(strs[i]);
        }

        vector<vector<string>> v;

        for (auto i = um.begin(); i != um.end(); i++) {
            v.push_back(i->second);
        }
```

```
        return v;
    }
};
```

Top K Frequent Elements - Medium

Notes

Usage of modified Bucket Sort approach which provides this optimal solution. This solution can also be done using Max Heap.

Program

```
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int, int> um;
        for (int i = 0; i < nums.size(); i++) {
            um[nums[i]]++;
        }

        vector<vector<int>> v(nums.size() + 1);
        for (auto it = um.begin(); it != um.end(); it++) {
            v[it->second].push_back(it->first);
        }

        vector<int> result;

        for (int i = nums.size(); i >= 0; i--) {
            if (result.size() >= k) {
                return result;
            }
            if (v[i].size() != 0) {
                result.insert(result.end(), v[i].begin(), v[i].end());
            }
        }

        return {};
    }
};
```

Product of Array Except Itself - Medium

Notes

The optimal solution is using a prefix and a postfix to parse through the array from both sides and multiplying.

Program

```
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        vector<int> result(nums.size(), 1);
        int pre = 1;
        int post = 1;

        for (int i = 0; i < nums.size(); i++) {
            result[i] = result[i] * pre;
            pre = pre * nums[i];
        }

        for (int i = nums.size() - 1; i >= 0; i--) {
            result[i] = result[i] * post;
            post = post * nums[i];
        }

        return result;
    }
};
```

Longest Consecutive Sequence - Medium

Notes

The optimal solution makes use of Unordered Set which provides an efficient way to search consecutive elements to find the Longest Consecutive Sequence.

Program

```
class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
        unordered_set<int> s;
        copy(nums.begin(), nums.end(), inserter(s, s.end()));

        int longest = 0;

        for (int i = 0; i < nums.size(); i++) {
            int length = 0;
            if (s.find(nums[i] - 1) == s.end()) {
                length++;
                while (s.find(nums[i]+length) != s.end()) {
                    length++;
                }
                longest = max(length, longest);
            }
        }

        return longest;
    }
};
```

Valid Palindrome - Easy

Notes

This problem can be approached in multiple ways. Mine is one of them.

Program

```
class Solution {
public:
    bool isPalindrome(string s) {
        string t = "";

        for (auto i:s) {
            if (iswalnum(i)) t += i;
        }

        transform(t.begin(), t.end(), t.begin(), ::tolower);

        s = t;
        std::reverse(s.begin(), s.end());

        cout << s << endl << t << endl;
        if (s == t) return true;
        return false;
    }
};
```

Two Sum II - Input Array Is Sorted - Medium

Notes

The optimal solution is to use the Two Pointer approach. Where one pointer from index 0 and the other pointer at the last index of the array.

Program

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        int i = 0;
        int j = nums.size() - 1;

        while (i < j) {
            if (nums[i] + nums[j] > target) {
                j--;
            } else if (nums[i] + nums[j] < target) {
                i++;
            } else if (nums[i] + nums[j] == target) {
                return {i+1, j+1};
            }
        }
        return {};
    }
};
```

Three Sum - Medium

Notes

The optimal solution uses the Two Sum II approach in this problem.

Program

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        vector<vector<int>> result;

        for (int i = 0; i < nums.size(); i++) {
            if (nums[i] > 0) {
                break;
            }
            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }

            int l = i + 1;
            int r = nums.size() - 1;

            while (l < r) {
                int threeSum = nums[i] + nums[l] + nums[r];

                if (threeSum > 0) {
                    r--;
                } else if (threeSum < 0) {
                    l++;
                } else {
                    result.push_back({nums[i], nums[l], nums[r]});

                    while (l < r && nums[l] == nums[l+1]) {
                        l++;
                    }
                    l++;
```

```
                    while (l < r && nums[r] == nums[r-1]) {
                        r--;
                    }
                    r--;
                }
            }
        }

        return result;
    }
};
```

Container With Most Water - Medium

Notes

The optimal solution is by using the Two Pointer approach and calculating the max area.

Program

```
class Solution {
public:
    int maxArea(vector<int>& height) {
        int i = 0;
        int j = height.size() - 1;
        int len = height.size() - 1;
        int maxWater = INT_MIN;

        while (i < j) {
            int ar = min(height[i], height[j]) * len;
            if (ar > maxWater) {
                maxWater = ar;
            }

            if (height[i] > height[j]) {
                j--;
                len--;
            } else if (height[i] < height[j]) {
                i++;
                len--;
            } else {
                i++;
                j--;
                len -= 2;
            }
        }

        return maxWater;
    }
};
```

Best Time to Buy and Sell Stock - Easy

Notes

The optimal solution uses the Two Pointer approach. The first pointer points to the first day stock price and the second pointer points to the next day's stock price.

Program

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int l = 0;
        int r = 1;
        int maxProfit = INT_MIN;

        while (r < prices.size()) {
            if (prices[r] - prices[l] > maxProfit) {
                maxProfit = prices[r] - prices[l];
            }

            if (prices[l] > prices[r]) {
                l = r;
                r = l + 1;
            } else {
                r = r + 1;
            }
        }

        if (maxProfit < 0) {
            return 0;
        }

        return maxProfit;
    }
};
```

Valid Parentheses - Easy

Notes

The optimal solution is by using a Stack.

Program

```
class Solution {
public:
    bool isValid(string s) {
        stack<char> st;
        unordered_map<char, char> um = {
            {')', '('},
            {']', '['},
            {'}', '{'}
        };

        for (int i = 0; i < s.length(); i++) {
            if (um.find(s[i]) != um.end()) {
                if (st.empty() || st.top() != um[s[i]]) {
                    return false;
                }

                st.pop();
            } else {
                st.push(s[i]);
            }
        }

        return st.empty();
    }
};
```