

# Leetcode Solutions

SVR

August 16, 2025

## Introduction

Following are my solutions for some leetcode problems. The solutions and code are primarily in C++ owing to the fact that I'm already using Python in my research, and C++ for the engineering part. However, C++ is something I'm trying to go deeper owing to the fact that I'm improving my ability to build low latency systems, which primarily use C/C++.

# Template Script

## Description

The following script is forked each time I want to locally work on a leetcode problem. The subsequent solutions in the later sections also have the functions present in this particular script in their scope. So this script also serves to provide an idea as to the functions, and what not, that are available. Note that the standard practice is to have these functions written in another file and have it included in the main script. However, I often tinker with these functions based on the problem at hand. Thus, the not-so-standard approach.

## Template.cpp

---

```
1 using std::map;
2 using std::format;
3 using std::deque;
4 using std::pair;
5
6 // vector printing function
7 template<typename T>
8 void fPrintVector(vector<T> input){
9     for(auto x: input) cout << x << ", ";
10    cout << endl;
11 }
12
13 template<typename T>
14 void fPrintMatrix(vector<T> input){
15     for(auto x: input){
16         for(auto y: x){
17             cout << y << ", ";
18         }
19         cout << endl;
20     }
```

```

21 }
22
23 template<typename T, typename T1>
24 void fPrintHashmap(unordered_map<T, T1> input){
25     for(auto x: input){
26         cout << format("{}{} \n", x.first, x.second);
27     }
28     cout << endl;
29 }
30
31 struct TreeNode {
32     int val;
33     TreeNode *left;
34     TreeNode *right;
35     TreeNode() : val(0), left(nullptr), right(nullptr) {}
36     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
37     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
38 };
39
40
41 struct ListNode {
42     int val;
43     ListNode *next;
44     ListNode() : val(0), next(nullptr) {}
45     ListNode(int x) : val(x), next(nullptr) {}
46     ListNode(int x, ListNode *next) : val(x), next(next) {}
47 };
48
49 void fPrintBinaryTree(TreeNode* root){
50     // sending it back
51     if (root == nullptr) return;
52
53     // printing
54     PRINTLINE
55     cout << "root->val = " << root->val << endl;

```

```

56
57 // calling the children
58 fPrintBinaryTree(root->left);
59 fPrintBinaryTree(root->right);
60
61 // returning
62 return;
63
64 }
65
66 void fPrintLinkedList(ListNode* root){
67     if (root == nullptr) return;
68     cout << root->val << ", ";
69     fPrintLinkedList(root);
70     return;
71 }
72
73 template<typename T>
74 void fPrintContainer(T input){
75     for(auto x: input) cout << x << ", ";
76     cout << endl;
77     return;
78 }
79
80 struct Stopwatch
81 {
82     std::chrono::time_point<std::chrono::high_resolution_clock> startpoint;
83     std::chrono::time_point<std::chrono::high_resolution_clock> endpoint;
84     std::chrono::duration<long long, std::nano> duration;
85
86     // constructor
87     Stopwatch() {startpoint = std::chrono::high_resolution_clock::now();}
88     void start() {startpoint = std::chrono::high_resolution_clock::now();}
89     void stop() {endpoint = std::chrono::high_resolution_clock::now(); fetchtime();}
90

```

```

91 void fetchtime(){
92     duration = std::chrono::duration_cast<std::chrono::nanoseconds>(endpoint - startpoint);
93     cout << format("{} nanoseconds \n", duration.count());
94 }
95 void fetchtime(string stringarg){
96     duration = std::chrono::duration_cast<std::chrono::nanoseconds>(endpoint - startpoint);
97     cout << format("{} took {} nanoseconds \n", stringarg, duration.count());
98 }
99 };
100
101
102 // main-file =====
103 int main(){
104
105     // input- configuration
106
107
108
109
110     // return
111     return(0);
112 }
113

```

---

## 1. Two Sum

Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`. You may assume that each input would have exactly one solution, and you may not use the same element twice. You can return the answer in any order.

### Code

---

```
1 int main(){
2
3     // input- configuration
4     vector<int> nums {2, 7, 11, 15};
5     int target {9};
6
7     // setup
8     int complement {0};
9     unordered_map<int, int> number_to_index;
10    vector<int> finaloutput;
11
12    // filling the unordered_map
13    for(int i = 0; i < nums.size(); ++i){
14
15        // calculating complement
16        complement = target - nums[i];
17
18        // checking if complement is present in registry
19        if(number_to_index.find(complement) != number_to_index.end()) [[unlikely]]
20        {
21            finaloutput.push_back(number_to_index[complement]); // adding first index
22            finaloutput.push_back(i); // adding second index
23            break; // breaking out
24        }
```

```

25     else [[likely]]
26     {
27         // check if current element is present
28         if (number_to_index.find(nums[i]) == number_to_index.end()) [[likely]]
29         {
30             // adding the [number, index] pair to the hashmap
31             number_to_index[nums[i]] = i;
32         }
33         else [[unlikely]]
34         {
35             // we'll do nothing since the number and its index is already present
36             continue;
37         }
38     }
39 }
40
41 // printing the final output
42 for(const auto& x : finaloutput) {cout << x << ", ";} cout << endl;
43
44 // return
45 return(0);
46
47 }

```

---



## 2. Add Two Numbers

You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list. You may assume the two numbers do not contain any leading zero, except the number 0 itself.

### Code

---

```
1 int main(){
2
3     // input- configuration
4     ListNode* l1  = new ListNode(2);
5     l1->next      = new ListNode(4);
6     l1->next->next = new ListNode(3);
7
8     ListNode* l2  = new ListNode(5);
9     l2->next      = new ListNode(6);
10    l2->next->next = new ListNode(4);
11
12    // setup
13    ListNode* traveller_1 = l1;
14    ListNode* traveller_2 = l2;
15    ListNode* finalOutput = new ListNode(-1);
16    ListNode* traveller_fo = finalOutput;
17
18    int sum          {0};
19    int carry        {0};
20    int value_1      {0};
21    int value_2      {0};
22
23    // moving through the two nodes
24    while(traveller_1 != nullptr || traveller_2 != nullptr){
```

```

25
26 // adding the two numbers
27 value_1 = traveller_1 == nullptr ? 0 : traveller_1->val;
28 value_2 = traveller_2 == nullptr ? 0 : traveller_2->val;
29
30 // calculating sum
31 sum = value_1 + value_2 + carry;
32 if (sum >= 10) [[unlikely]] {sum -= 10; carry = 1;}
33 else [[likely]] {carry = 0;}
34
35 // creating node
36 traveller_fo->next = new ListNode(sum);
37 traveller_fo = traveller_fo->next;
38
39 // updating the two pointers
40 if(traveller_1 != nullptr) [[likely]] {traveller_1 = traveller_1->next;}
41 if(traveller_2 != nullptr) [[likely]] {traveller_2 = traveller_2->next;}
42 }
43
44 // creating a final node if carry is non-zero
45 if (carry == 1) [[unlikely]] {
46     traveller_fo->next = new ListNode(carry);
47 }
48
49 // printing the final output
50 traveller_fo = finalOutput->next;
51 cout << format("final-output = ");
52 while(traveller_fo != nullptr){
53     cout << traveller_fo->val << ", ";
54     traveller_fo = traveller_fo->next;
55 }
56 cout << "\n";
57
58 // return
59 return(0);

```

60

61

}

## 11. Container with most water

- You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]).
- Find two lines that together with the x-axis form a container, such that the container contains the most water.
- Return the maximum amount of water a container can store.
- Notice that you may not slant the container.

### Code

---

```
1 int main(){
2
3     // input- configuration
4     vector<int> height {1,8,6,2,5,4,8,3,7};
5
6     // setup
7     int left      {0};
8     int right     {static_cast<int>(height.size())-1};
9     int maxvolume {-1};
10    int currvolume {-1};
11
12    // two-pointer approach
13    while(left < right){
14
15        // calculating volumes
16        currvolume = (right - left) * std::min(height[left], height[right]);
17        maxvolume = maxvolume > currvolume ? maxvolume : currvolume;
18
19        // adjusting left and right based on volume
```

```
20     if (height[left] < height[right]) {++left;}
21     else                                {--right;}
22 }
23
24 // printing
25 cout << format("maxvolume = {}\n", maxvolume);
26
27 // return
28 return(0);
29
30 }
```

---

## 26. Remove Duplicates From Sorted Array

Given an integer array `nums` sorted in non-decreasing order, remove the duplicates in-place such that each unique element appears only once. The relative order of the elements should be kept the same. Then return the number of unique elements in `nums`. Consider the number of unique elements of `nums` to be `k`, to get accepted, you need to do the following things:

- Change the array `nums` such that the first `k` elements of `nums` contain the unique elements in the order they were present in `nums` initially. The remaining elements of `nums` are not important as well as the size of `nums`.
- Return `k`.

### Code

---

```
1 int main(){
2
3     // input- configuration
4     vector<int> nums    {1,1};
5
6     // setup
7     int p    {0};
8     int counter {0};
9
10    // going through the values
11    for(int i = 1; i<nums.size(); ++i){
12
13        // check values
14        if (nums[i] == nums[p]) {continue;}
15
16        // writing values
17        ++p;
18        nums[p] = nums[i];
```

```
19     ++counter;
20 }
21
22 // printing the final output
23 cout << format("final-output = {}\n", counter+1);
24 cout << format("nums = "); fpv(nums);
25
26 // return
27 return(0);
28
29 }
```

---

## 27. Remove Element

Given an integer array `nums` and an integer `val`, remove all occurrences of `val` in `nums` in-place. The order of the elements may be changed. Then return the number of elements in `nums` which are not equal to `val`.

Consider the number of elements in `nums` which are not equal to `val` be `k`, to get accepted, you need to do the following things:

Change the array `nums` such that the first `k` elements of `nums` contain the elements which are not equal to `val`. The remaining elements of `nums` are not important as well as the size of `nums`. Return `k`.

### Code

---

```
1 int main(){
2
3     // input- configuration
4     vector<int> nums {0,1,2,2,3,0,4,2};
5     int val         {2};
6
7     // setup
8     int src          {0};
9     int dest         {0};
10    int numwrites {0};
11
12    // going through the indices
13    while(src < nums.size()){
14
15        // moving the dest until we find a val-position
16        while(nums[dest] != val) {++dest;}
17
18        // moving source until we find a non-val position after dest
19        src = std::max(src, dest+1);
20        while(nums[src] == val) {++src;};
21    }
```



```
21
22     // writing
23     if (dest < nums.size() && src < nums.size()){
24         nums[dest] = nums[src];
25         ++dest;
26         ++src;
27         ++numwrites;
28     }
29
30 }
31
32 // printing the length
33 cout << format("updated nums = "); fPrintVector(nums);
34 cout << format("finaloutput = {} \n", nums.size()-numwrites-1);
35
36 // return
37 return(0);
38
39 }
```

---

## 80. Remove Duplicates from Sorted Array II

Given an integer array `nums` sorted in non-decreasing order, remove some duplicates in-place such that each unique element appears at most twice. The relative order of the elements should be kept the same.

Since it is impossible to change the length of the array in some languages, you must instead have the result be placed in the first part of the array `nums`. More formally, if there are `k` elements after removing the duplicates, then the first `k` elements of `nums` should hold the final result. It does not matter what you leave beyond the first `k` elements.

Return `k` after placing the final result in the first `k` slots of `nums`.

Do not allocate extra space for another array. You must do this by modifying the input array in-place with  $O(1)$  extra memory.

### 1. Example 1

- Input: `nums = [1,1,1,2,2,3]`
- Output: 5, `nums = [1,1,2,2,3,_]`

### 2. Example 2

- Input: `nums = [0,0,1,1,1,1,2,3,3]`
- Output: 7, `nums = [0,0,1,1,2,3,3,_,_]`

## Code

---

```
1 int main(){
2
3     // input- configuration
4     vector<int> nums {1,1,1,2,2,3};
5 }
```

```

6 // setup
7 int destination {1};
8 int prev {nums[0]};
9 int element_counter {1};
10 int numwrites {1};
11
12 // going through the values
13 for(int i = 1; i < nums.size(); ++i){
14
15     // updating counter
16     if (nums[i-1] == nums[i]) {++element_counter;}
17     else {element_counter = 1;}
18
19     // checking the element counters
20     if (element_counter <=2) {nums[destination++] = nums[i];}
21
22 }
23
24 // printing the final output
25 cout << format("nums = "); fpv(nums);
26 cout << format("return-value = {}\n", destination);
27
28 // return
29 return(0);
30
31 }

```

---

## 88. Merge Sorted Array

You are given two integer arrays `nums1` and `nums2`, sorted in non-decreasing order, and two integers `m` and `n`, representing the number of elements in `nums1` and `nums2` respectively.

Merge `nums1` and `nums2` into a single array sorted in non-decreasing order.

The final sorted array should not be returned by the function, but instead be stored inside the array `nums1`. To accommodate this, `nums1` has a length of `m + n`, where the first `m` elements denote the elements that should be merged, and the last `n` elements are set to 0 and should be ignored. `nums2` has a length of `n`.

### Code

---

```
1 int main(){
2
3     // input- configuration
4     vector<int> nums1 {1, 2, 3, 0, 0, 0};
5     vector<int> nums2 {2, 5, 6};
6     int m {3};
7     int n {3};
8
9     // setup
10    int p1 {m-1};
11    int p2 {n-1};
12    int p3 {m+n-1};
13
14    int curr1 {-1};
15    int curr2 {-1};
16
17    // going the other way
18    while(p1 >= 0 || p2 >= 0)
19    {
```

```
20 // printing the values
21 curr1 = p1 >= 0 ? nums1[p1] : std::numeric_limits<int>::min();
22 curr2 = p2 >= 0 ? nums2[p2] : std::numeric_limits<int>::min();
23
24 // assigning value
25 if (curr1 > curr2) {nums1[p3] = curr1; --p3; --p1;}
26 else {nums1[p3] = curr2; --p3; --p2;}
27
28 }
29
30 // printing the final output
31 cout << format("finaloutput = "); fPrintVector(nums1);
32
33 // return
34 return(0);
35 }
```

---

## 169 Majority Element

Given an array `nums` of size `n`, return the majority element. The majority element is the element that appears more than  $\lfloor n/2 \rfloor$  times. You may assume that the majority element always exists in the array.

- **Example 1**

- Input: `nums = [3,2,3]`
- Output: `3`

- **Example 2**

- Input: `nums = [2,2,1,1,1,2,2]`
- Output: `2`

### Code

---

```
1 int main(){
2
3     // input- configuration
4     vector<int> nums {2,2,1,1,1,2,2};
5
6     // setup
7     unordered_map<int, int> histogram;
8     int max_element {std::numeric_limits<int>::min()};
9     int max_count {std::numeric_limits<int>::min()};
10    int updated_count {0};
11
12    // going through the elements
13    for(int i = 0; i<nums.size(); ++i){
```

```
14
15 // adding to histogram
16 if (histogram.find(nums[i]) == histogram.end()) {histogram[nums[i]] = 1; updated_count = 0;}
17 else {++histogram[nums[i]]; updated_count = histogram[nums[i]];}
18
19 // keeping track of max-element
20 if (updated_count > max_count) {max_element = nums[i]; max_count = updated_count;}
21
22 }
23
24 // printing the final output
25 cout << format("nums = "); fpv(nums);
26 cout << format("max-count = {}\n", max_count);
27
28 // return
29 return(0);
30
31 }
```

---

## 392. Is Subsequence

Given two strings *s* and *t*, return true if *s* is a subsequence of *t*, or false otherwise.

A subsequence of a string is a new string that is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (i.e., "ace" is a subsequence of "abcde" while "aec" is not).

### Code

---

```
1 int main(){
2
3     // input- configuration
4     string s {"abc"};
5     string t {"ahbgdc"};
6
7     // setup
8     int i = 0;
9
10    // going through the elements
11    for(auto x: t) if (x == s[i]) ++i;
12
13    // returning
14    cout << format("final-output = {}\n", static_cast<bool>(i == s.size())) ;
15
16
17    // return
18    return(0);
19
20 }
```

---