

Leetcode Solutions

SVR

August 19, 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

}

3. Longest Substring Without Repeating Characters

Question

Given a string s, find the length of the longest substring without duplicate characters.

Solution

Code

```
1 int main(){
2
3     // input- configuration
4     string s {"tmmzuxt"};
5
6     // setup
7     unordered_map<char, int> histogram;
8     int p1 {0};
9     char curr;
10    int finaloutput {-1};
11    int temp_length {-1};
12
13    // going through the thing
14    for(int p2 = 0; p2<s.size(); ++p2){
15
16        // moving to another variable
17        curr = s[p2];
18
19        // checking if current character is in histogram
20        if (histogram.find(curr) == histogram.end()) [[unlikely]]
21        {
```

```

22     histogram[curr] = 1;
23 }
24 else [[likely]]
25 {
26     // checking if count is zero
27     if (histogram[curr] == 0)
28     {
29         histogram[curr] = 1;
30     }
31     else
32     {
33         // moving p1 until it arrives at first instance of curr
34         while(s[p1] != curr)
35         {
36             --histogram[s[p1]];
37             ++p1;
38         }
39         ++p1;
40         histogram[curr] = 1;
41     }
42 }
43
44 // calculating longest length
45 finaloutput = finaloutput > (p2-p1+1) ? finaloutput : (p2-p1+1);
46 }
47
48 // printing
49 cout << format("longest length = {} \n", finaloutput);
50
51 // return
52 return(0);
53 }

```

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 }
```

42. Trapping Rain Water

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

Examples

1. Example 1

- Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]
- Output: 6
- Explanation: The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

2. Example 2

- Input: height = [4,2,0,3,2,5]
- Output: 9

Constraints

1. $n == \text{height.length}$
2. $1 \leq n \leq 2 * 10^4$
3. $0 \leq \text{height}[i] \leq 10^5$

Code

```
1 int main(){
2
3     // input- configuration
4     vector<int> height {0,1,0,2,1,0,1,3,2,1,2,1};
5
6     // setup
7     vector<int> leftmaxes(height.size(), 0);
8     vector<int> rightmaxes(height.size(), 0);
9     int forwardindex {0};
10    int backwardindex {0};
11    int maxleft {-1};
12    int maxright {-1};
13    int finaloutput {0};
14
15    // building left-max
16    for(int i = 1; i<height.size(); ++i){
17
18        // calculating indices
19        forwardindex = i;
20        backwardindex = height.size()-1-i;
21
22        // calculating maxleft
23        maxleft = height[forwardindex-1] > maxleft ?
24                  height[forwardindex-1] : maxleft;
25        leftmaxes[forwardindex] = maxleft;
26
27        // calculating max right
28        maxright = height[backwardindex+1] > maxright ?
29                   height[backwardindex+1] : maxright;
30        rightmaxes[backwardindex] = maxright;
31    }
32
33    // going through the array to calculate maxvolume held by each column
```

```
// vector holding biggest-height to left
// vector holding biggest-height to the right
// for maintaining forward-index
// for maintaining backward-index
// keeping record of biggest left
// keeping record of biggest right
// storing final output

// forward-index
// backward-index

// running max-left
// storing to vector

// running max-right
// storing to vector
```

```

34 for(int i = 0; i < height.size(); ++i){
35
36     // finding max-height of the current column
37     auto minheight    = std::min({leftmaxes[i], rightmaxes[i]}); // finding max-height of borders
38     auto columnheight = minheight - height[i];                 // subtracting to find space
39     columnheight      = columnheight > 0 ? columnheight : 0;    // in case curr-height > max-height
40     finaloutput       += columnheight;                          // accumulating to water content
41 }
42
43 // printing the final output
44 cout << format("finaloutput = {}\n", finaloutput);
45
46 // return
47 return(0);
48
49 }

```

45 Jump Game II

You are given a 0-indexed array of integers `nums` of length `n`. You are initially positioned at index 0. Each element `nums[i]` represents the maximum length of a forward jump from index `i`. In other words, if you are at index `i`, you can jump to any index `(i + j)` where:

- $0 \leq j \leq \text{nums}[i]$
- $i + j \leq n$

Return the minimum number of jumps to reach index `n - 1`. The test cases are generated such that you can reach index `n - 1`.

Examples

1. Example 1

- Input: `nums = [2,3,1,1,4]`
- Output: 2
- Explanation: The minimum number of jumps to reach the last index is 2. Jump 1 step from index 0 to 1, then 3 steps to the last index.

2. Example 2

- Input: `nums = [2,3,0,1,4]`
- Output: 2

Constraints

- $1 \leq \text{nums.length} \leq 10^4$
- $0 \leq \text{nums}[i] \leq 1000$
- It's guaranteed that you can reach $\text{nums}[\text{n} - 1]$.

Code

```
1 int main(){
2
3     // input- configuration
4     vector<int> nums {2,3,0,1,4};
5
6     // setup
7     Timer timer;
8     vector<int> minjumps(nums.size(),0);
9     int leftboundary {-1};
10    int rightboundary {-1};
11
12    // moving from the back
13    for(int i = nums.size()-2; i>=0; --i){
14
15        // continuign if nums[i] = 0
16        if (nums[i] == 0) {
17            minjumps[i] = std::numeric_limits<int>::max();
18            continue;
19        }
20
21        // range of values it can go from here
22        leftboundary = i+1;
23        rightboundary = i+nums[i];
```

```
// setting a timer
// the dp table
// variable to hold the left-boundary
// variable to hold the right-boundary
```

```
// to prevent this from being chosen
// moving to next index
```

```
// the starting point of range
// the end point of range
```



```

24     rightboundary = rightboundary < nums.size()-1 ?
25         rightboundary : nums.size()-1;                                // ensuring within vector range
26
27     // calculating smallest element in range
28     auto it = std::min_element(minjumps.begin()+leftboundary,
29                               minjumps.begin()+rightboundary+1);    // finding the minimum value in the range
30
31     // adding min-element to the array
32     if (*it == std::numeric_limits<int>::max())
33         minjumps[i] = std::numeric_limits<int>::max();                // ensuring infity logic
34     else
35         minjumps[i] = (1 + *it);                                       // for regular values
36
37 }
38
39 // printing
40 cout << format("finaloutput = {}\n", minjumps[0]);
41 timer.measure();
42
43 // return
44 return(0);
45
46 }

```

55. Jump Game

You are given an integer array `nums`. You are initially positioned at the array's first index, and each element in the array represents your maximum jump length at that position. Return `true` if you can reach the last index, or `false` otherwise.

Examples

1. Example 1

- Input: `nums = [2,3,1,1,4]`
- Output: `true`
- Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.

2. Example 2

- Input: `nums = [3,2,1,0,4]`
- Output: `false`
- Explanation: You will always arrive at index 3 no matter what. Its maximum jump length is 0, which makes it impossible to reach the last index.

Constraints

- $1 \leq \text{nums.length} \leq 10^4$
- $0 \leq \text{nums}[i] \leq 10^5$

Code

```
1 int main(){
2
3     // input- configuration
4     vector<int> nums {3,2,1,0,4};
5
6     // setup
7     Timer timer;
8     int maxjumpdistance {0};
9     int currjumpdistance {0};
10    int finaloutput {0};
11
12    // going through the nums
13    for(int i = 0; i<=maxjumpdistance && i<nums.size(); ++i){
14
15        // calculating max-distance we can go from here
16        currjumpdistance = i + nums[i];
17
18        // updating max-jumpdistance
19        maxjumpdistance = currjumpdistance > maxjumpdistance ? \
20            currjumpdistance : maxjumpdistance;
21
22    }
23
24    // updating the final output
25    finaloutput = maxjumpdistance >= nums.size()-1 ? true : false;
26
27    // printing the thing
28    cout << format("final-output = {}\n", finaloutput);
29    timer.measure();
30
31
32    // return
33    return(0);
```

34

35

}

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 }
```

121. Best Time To Buy And Sell Stock

You are given an array prices where prices[i] is the price of a given stock on the ith day. You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock. Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

1. Example 1

- Input: prices = [7,1,5,3,6,4]
- Output: 5

2. Example 2

- Input: prices = [7,6,4,3,1]
- Output: 0

Code

```
1 int main(){
2
3     // input- configuration
4     vector<int> prices {7,6,4,3,1};
5
6     // setup
7     Stopwatch timer;                                // timer-object
8     int p0      {0};                                // first index-pointer
9     int p1      {1};                                // second index-pointer
10    int maxprofit {0};                                // variable to hold max-profit
11    int curr     {-1};                                // variable to hold current-profit
12}
```

```
13 // going through array
14 while(p1<prices.size()){
15     curr      = prices[p1] - prices[p0];           // calculating current profit
16     maxprofit = curr > maxprofit ? curr : maxprofit; // updating max-profit
17     if (curr < 0) {p0 = p1;}                       // updating p0 if we find lower point
18     ++p1;
19 }
20
21 // printing the final output
22 cout << format("maxprofit = {}\n", maxprofit);
23 timer.stop();
24
25 // return
26 return(0);
27
28 }
```

122. Best Time To Buy And Sell Stock II

You are given an integer array `prices` where `prices[i]` is the price of a given stock on the i th day. On each day, you may decide to buy and/or sell the stock. You can only hold at most one share of the stock at any time. However, you can buy it then immediately sell it on the same day. Find and return the maximum profit you can achieve.

Examples

1. Example 1

- Input: `prices = [7,1,5,3,6,4]`
- Output: 7
- Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = $5 - 1 = 4$. Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = $6 - 3 = 3$. Total profit is $4 + 3 = 7$.

2. Example 2

- Input: `prices = [1,2,3,4,5]`
- Output: 4
- Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = $5 - 1 = 4$. Total profit is 4.

3. Example 3

- Input: `prices = [7,6,4,3,1]`
- Output: 0
- Explanation: There is no way to make a positive profit, so we never buy the stock to achieve the maximum profit of 0.

Constraints

- $1 \leq \text{prices.length} \leq 3 * 10^4$
- $0 \leq \text{prices}[i] \leq 10^4$

Code

```
1 int main(){
2
3     // input- configuration
4     vector<int> prices {7,1,5,3,6,4};
5
6     // setup
7     int p1      {0};           // index-pointer to buying
8     int p2      {0};           // index-pointer to selling
9     int accprofit {0};          // variable to accumulate profit
10    int currprofit {std::numeric_limits<int>::min()}; // variable to hold curr-profit
11
12    // going through this
13    while(p2 < prices.size()){
14
15        currprofit = prices[p2] - prices[p1];           // calculating current profit
16
17        if (currprofit > 0){
18            accprofit += currprofit;                     // accumulating the profit
19            p1         = p2++;                             // moving the starting point
20            continue;                                     // moving into the next iteration
21        }
22        else if (currprofit < 0){
23            p1         = p2++;                             // moving the starting point
24            continue;
25        }
26    }
```

```
26         ++p2;
27         // updating p2
28     }
29
30     // printing the max-value
31     cout << format("accprofit = {}\n", accprofit);
32
33     // return
34     return(0);
35
36 }
```

134. Gas Station

There are n gas stations along a circular route, where the amount of gas at the i th station is $gas[i]$. You have a car with an unlimited gas tank and it costs $cost[i]$ of gas to travel from the i th station to its next $(i + 1)$ th station. You begin the journey with an empty tank at one of the gas stations. Given two integer arrays gas and $cost$, return the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -1 . If there exists a solution, it is guaranteed to be unique.

Examples

1. Example 1:

- Input: $gas = [1,2,3,4,5]$, $cost = [3,4,5,1,2]$
- Output: 3

2. Example 2:

- Input: $gas = [2,3,4]$, $cost = [3,4,3]$
- Output: -1

Constraints:

- $n == gas.length == cost.length$
- $1 \leq n \leq 10^5$
- $0 \leq gas[i], cost[i] \leq 10^4$
- The input is generated such that the answer is unique.

Code

```
1 int main(){
2
3     // input- configuration
4     vector<int> gas    {1,2,3,4,5};
5     vector<int> cost   {3,4,5,1,2};
6
7     // setup
8     auto acc    {0};
9     vector<int> diffvec;
10    auto temp    {0};
11    int finaloutput {-1};
12
13    // running through it
14    for(int i = 0; i<cost.size(); ++i){
15        temp    = gas[i] - cost[i];
16        acc      += temp;
17        diffvec.push_back(temp);
18    }
19    if (acc<0) {finaloutput = -1; return 0;}
20
21    // going through the diff-vec
22    acc = 0;
23    for(int i = 0; i<diffvec.size(); ++i){
24        acc += diffvec[i];
25        if (acc<0) {acc = 0; finaloutput = i+1;}
26    }
27
28
29    // printing the acc
30    cout << format("acc = {}\n", finaloutput);
31
32    // return
33    return(0);
```

34

35

}

135. Candy

There are n children standing in a line. Each child is assigned a rating value given in the integer array `ratings`. You are giving candies to these children subjected to the following requirements:

1. Each child must have at least one candy.
2. Children with a higher rating get more candies than their neighbors.
3. Return the minimum number of candies you need to have to distribute the candies to the children.

Examples

- **Example 1**

- Input: `ratings = [1,0,2]`
- Output: 5
- Explanation: You can allocate to the first, second and third child with 2, 1, 2 candies respectively.

- **Example 2**

- Input: `ratings = [1,2,2]`
- Output: 4
- Explanation: You can allocate to the first, second and third child with 1, 2, 1 candies respectively. The third child gets 1 candy because it satisfies the above two conditions.

Constraints

1. $n == \text{ratings.length}$
2. $1 \leq n \leq 2 * 10^4$
3. $0 \leq \text{ratings}[i] \leq 2 * 10^4$

Code

```
1 // main-file =====
2 int main(){
3
4     // input- configuration
5     vector<int> ratings {1,0,2};
6
7     // setup
8     auto candies      {std::vector<int>(ratings.size(),1)};
9     auto finaloutput  {static_cast<int>(candies.size())};
10    int leftrating, currrating, rightrating;
11
12    // left-pass
13    for(int i = 1; i<candies.size(); ++i){
14
15        // fetching the rating
16        leftrating = ratings[i-1];
17        currrating = ratings[i];
18
19        // fetching references to candy counts
20        int& leftcount = candies[i-1];
21        int& currcount = candies[i];
22
23        // updating based on left
```

```

24     if (currrating > leftrating){
25         currcount = leftcount+1;
26     }
27 }
28
29 // right pass
30 for(int i = ratings.size()-2; i>=0; --i){
31
32     // fetching ratings
33     currrating = ratings[i];
34     rightrating = ratings[i+1];
35
36     // fetching references to candies
37     int& curr candies = candies[i];
38     int& rightcandies = candies[i+1];
39
40     // updating based on right
41     if (currrating > rightrating){
42         curr candies = std::max(curr candies,
43                                 rightcandies + 1);
44     }
45 }
46
47 // summing up candies
48 finaloutput = std::accumulate(candies.begin(), candies.end(), 0);
49 cout << format("finaloutput = {}\n", finaloutput);
50
51 // return
52 return(0);
53
54 }

```

151. Reverse Words In A String

Given an input string *s*, reverse the order of the words. A word is defined as a sequence of non-space characters. The words in *s* will be separated by at least one space. Return a string of the words in reverse order concatenated by a single space. Note that *s* may contain leading or trailing spaces or multiple spaces between two words. The returned string should only have a single space separating the words. Do not include any extra spaces.

Examples

1. Example 1

- Input: *s* = "the sky is blue"
- Output: "blue is sky the"

2. Example 2

- Input: *s* = " hello world "
- Output: "world hello"
- Explanation: Your reversed string should not contain leading or trailing spaces.

3. Example 3

- Input: *s* = "a good example"
- Output: "example good a"
- Explanation: You need to reduce multiple spaces between two words to a single space in the reversed string.

Constraints

1. $1 \leq s.length \leq 10^4$
2. s contains English letters (upper-case and lower-case), digits, and spaces ' '.
3. There is at least one word in s.

Code

```
1 int main(){
2
3     // input- configuration
4     string s {"a good example"};
5
6     // setup
7     vector<string> listofwords;
8
9     // creating a list of words
10    int p1 {0};
11    string acc;
12    while(p1 < s.size()){
13
14        // checking if the current character is a non-space
15        if (s[p1] != ' '){acc += s[p1];}
16        else{
17            // if acc is non-empty, flush
18            if (acc.size() != 0) {listofwords.push_back(acc); acc = "";}
19            else {;}
20        }
21
22        // moving the index-pointer forward
23        p1++;
```

```
24 }
25
26 // check if acc is unflushed
27 if (acc.size() != 0) {listofwords.push_back(acc); acc = "";}
28
29 // building the finaloutput
30 string finaloutput;
31 for(int i = listofwords.size()-1; i>=0; --i){
32     finaloutput += listofwords[i];
33     if (i!=0) [[unlikely]] {finaloutput += " ";}
34 }
35
36 // printing the finaloutput
37 cout << format("finaloutput = {}\n", finaloutput);
38
39
40 // return
41 return(0);
42
43 }
```

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 }
```

189 RotateArray

Given an integer array nums, rotate the array to the right by k steps, where k is non-negative.

- **Example 1**

- Input: nums = [1,2,3,4,5,6,7], k = 3
- Output: [5,6,7,1,2,3,4]

- **Example 1**

- Input: nums = [-1,-100,3,99], k = 2
- Output: [3,99,-1,-100]

Code

```
1 int main(){
2
3     // input- configuration
4     vector<int> nums {-1,-100,3,99};
5     int k {2};
6
7     // setup
8     Stopwatch timer;                                // setting up the timer
9     k = k %static_cast<int>(nums.size());           // to ensure that the value is within range
10
11     int source      {0};
12     int temp_source {nums[source]};
13     int temp        {0};
14     int destination {0};
```

```

15
16 vector<bool> sourcelist(nums.size(), false);
17
18 // going through nums
19 for(int i = 0; i < nums.size(); ++i){
20
21     // check if curent-source has been taken care of
22     if (sourcelist[source] == true){
23         source      = (source+1) % nums.size();
24         temp_source = nums[source];
25     }
26
27     source          = source % nums.size(); // code to ensure range
28     destination     = (source + k)%nums.size(); // calculating the index we'll be writing to
29     sourcelist[source] = true; // updating source-list
30
31     temp            = nums[destination]; // safe-keeping the destination value
32     nums[destination] = temp_source; // storing new value at destination-index
33
34     source          = destination; // updating source-index
35     temp_source     = temp; // updating source-value
36 }
37
38 // printing the output
39 cout << format("nums = "); fpv(nums); // printing the updated array, "nums"
40 timer.stop(); // printing the time taken
41
42 // return
43 return(0);
44 }

```

238. Product of Array Except Self

Given an integer array `nums`, return an array `answer` such that `answer[i]` is equal to the product of all the elements of `nums` except `nums[i]`. The product of any prefix or suffix of `nums` is guaranteed to fit in a 32-bit integer. You must write an algorithm that runs in $O(n)$ time and without using the division operation.

Examples

1. Example 1

- Input: `nums = [1,2,3,4]`
- Output: `[24,12,8,6]`

2. Example 2

- Input: `nums = [-1,1,0,-3,3]`
- Output: `[0,0,9,0,0]`

Constraints

1. $2 \leq \text{nums.length} \leq 10^5$
2. $-30 \leq \text{nums}[i] \leq 30$
3. The input is generated such that `answer[i]` is guaranteed to fit in a 32-bit integer

Code

```
1 int main(){
2
3     // input- configuration
4     vector<int> nums {1,2,3,4};
5
6     // setup
7     vector<int> nums_left(nums.size(), 1);
8     vector<int> nums_right(nums.size(), 1);
9     int acc_left {1};
10    int acc_right {1};
11
12    // runs
13    for(int i = 0; i<nums.size(); ++i){
14
15        // source-indices
16        int source_left {i-1};
17        int source_right {static_cast<int>(nums.size())-i};
18
19        // printing values
20        acc_left  *= source_left == -1      ? 1 : nums[source_left];
21        acc_right *= source_right == nums.size() ? 1 : nums[source_right];
22
23        // writing to the two values
24        nums_left[i] = acc_left;
25        nums_right[nums.size()-i-1] = acc_right;
26    }
27
28    // building the accumulated value
29    vector<int> finaloutput(nums.size(),1);
30    for(int i = 0; i< finaloutput.size(); ++i){
31        finaloutput[i] = nums_left[i] * nums_right[i];
32    }
33}
```

```
34 // printing
35 cout << format("finaloutput = "); fPrintVector(finaloutput);
36
37 // return
38 return(0);
39
40 }
```

274. H-Index

Given an array of integers citations where citations[i] is the number of citations a researcher received for their ith paper, return the researcher's h-index. According to the definition of h-index on Wikipedia: The h-index is defined as the maximum value of h such that the given researcher has published at least h papers that have each been cited at least h times.

Examples

1. Example 1

- Input: citations = [3,0,6,1,5]
- Output: 3
- Explanation: [3,0,6,1,5] means the researcher has 5 papers in total and each of them had received 3, 0, 6, 1, 5 citations respectively. Since the researcher has 3 papers with at least 3 citations each and the remaining two with no more than 3 citations each, their h-index is 3.

2. Example 2

- Input: citations = [1,3,1]
- Output: 1

Constraints

1. $n == \text{citations.length}$
2. $1 \leq n \leq 5000$
3. $0 \leq \text{citations}[i] \leq 1000$

Code

```
1 int main(){
2
3     // input- configuration
4     vector<int> citations {3,0,6,1,5};
5
6     // sorting the citations first
7     std::sort(citations.begin(), citations.end(),
8               [](const int& a, const int& b) {return a>b;});
9
10    // running accumulations
11    auto hvalue = {0};
12    for(int i = 0; i<citations.size(); ++i){
13        if (citations[i] >= (i+1))    {hvalue = i+1;}
14    }
15
16    // printing citations
17    cout << format("hvalue = {}\n", hvalue);
18
19    // return
20    return(0);
21
22 }
```

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 }
```
