

Leetcode Solutions

SVR

August 23, 2025

Contents

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.

Examples

1. Example 1:

- Input: `nums = [2,7,11,15]`, `target = 9`
- Output: `[0,1]`
- Explanation: Because `nums[0] + nums[1] == 9`, we return `[0, 1]`.

2. Example 2:

- Input: `nums = [3,2,4]`, `target = 6`
- Output: `[1,2]`

3. Example 3:

- Input: `nums = [3,3]`, `target = 6`
- Output: `[0,1]`

Constraints:

- $2 \leq \text{nums.length} \leq 10^4$

- $-10^9 \leq \text{nums}[i] \leq 10^9$
- $-10^9 \leq \text{target} \leq 10^9$
- Only one valid answer exists.

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.

Examples

1. Example 1:

- Input: $l1 = [2,4,3]$, $l2 = [5,6,4]$
- Output: $[7,0,8]$
- Explanation: $342 + 465 = 807$.

2. Example 2:

- Input: $l1 = [0]$, $l2 = [0]$
- Output: $[0]$

3. Example 3:

- Input: $l1 = [9,9,9,9,9,9,9]$, $l2 = [9,9,9,9]$
- Output: $[8,9,9,9,0,0,0,1]$

Constraints:

- The number of nodes in each linked list is in the range $[1, 100]$.

- $0 \leq \text{Node.val} \leq 9$
- It is guaranteed that the list represents a number that does not have leading zeros.

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

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

1. Example 1:

- Input: $s = \text{"abcabcbb"}$
- Output: 3
- Explanation: The answer is "abc", with the length of 3.

2. Example 2:

- Input: $s = \text{"bbbbbb"}$
- Output: 1
- Explanation: The answer is "b", with the length of 1.

3. Example 3:

- Input: $s = \text{"pwwkew"}$
- Output: 3
- Explanation: The answer is "wke", with the length of 3. Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

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 }

```

4. Median Of Two Sorted Array

Given two sorted arrays `nums1` and `nums2` of size `m` and `n` respectively, return the median of the two sorted arrays. The overall run time complexity should be $O(\log(m+n))$.

Examples

1. Example 1:

- Input: `nums1 = [1,3]`, `nums2 = [2]`
- Output: 2.00000
- Explanation: merged array = `[1,2,3]` and median is 2.

2. Example 2:

- Input: `nums1 = [1,2]`, `nums2 = [3,4]`
- Output: 2.50000
- Explanation: merged array = `[1,2,3,4]` and median is $(2 + 3) / 2 = 2.5$.

Constraints:

1. `nums1.length == m`
2. `nums2.length == n`
3. $0 \leq m \leq 1000$

4. $0 \leq n \leq 1000$
5. $1 \leq m + n \leq 2000$
6. $-10^6 \leq \text{nums1}[i], \text{nums2}[i] \leq 10^6$

Code

```
1 int main(){
2
3     // input- configuration
4     vector<int> nums1 {1, 2};
5     vector<int> nums2 {3, 4};
6
7
8     // setup
9     vector<int>& first = nums1[0] <= nums2[0] ? nums1 : nums2;
10    vector<int>& second = nums1[0] > nums2[0] ? nums1 : nums2;
11    int left_first {0};
12    int right_first {static_cast<int>(first.size())-1};
13    int left_second {0};
14    int right_second {static_cast<int>(second.size())-1};
15    int left_value = first[left_first] < second[left_second] ? first[left_first] : second[left_second];
16    int right_value = first[right_first] > second[right_second] ? first[right_first] : second[right_second];
17    int numiterations {static_cast<int>((nums1.size() + nums2.size())/2)};
18
19
20    // running for a certain number of iterations
21    for(int i = 0; i<numiterations+1; ++i){
22
23        // updating left
24        if (first[left_first] < second[left_second]) {left_value = first[left_first]; ++left_first;}
25        else {left_value = second[left_second]; ++left_second;}
```

```
26     if (first[right_first] > second[right_second]) {right_value = first[right_first]; --right_first;}
27     else {right_value = second[right_second]; --right_second;}
28
29     // printing
30     cout << format("left-value = {}, right-value = {}\n", left_value, right_value);
31 }
32
33 cout << format("median = {}\n", static_cast<double>(left_value + right_value)/2.0);
34
35
36
37 // return
38 return(0);
39
40 }
```

6. Zigzag Conversion

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

P	-	A	-	H	-	N
A	P	L	S	I	I	G
Y	-	I	-	R	-	-

And then read line by line: "PAHNAPLSIIGYIR"

Examples

1. Example 1:

- Input: s = "PAYPALISHIRING", numRows = 3
- Output: "PAHNAPLSIIGYIR"

2. Example 2:

- Input: s = "PAYPALISHIRING", numRows = 4
- Output: "PINALSIGYAHRPI"

3. Example 3:

- Input: s = "A", numRows = 1
- Output: "A"

Constraints:

1. $1 \leq s.length \leq 1000$
2. s consists of English letters (lower-case and upper-case), ',' and '.'.
3. $1 \leq numRows \leq 1000$

Code

```
1 int main(){
2
3     // input- configuration
4     string s {"PAYPALISHIRING"};
5     int numRows {4};
6
7     // trivial case
8     if (numRows == 1) {cout << format("finaloutput = {}\n", s); return 0;}
9
10    // setup
11    int modlength {2*numRows-2};
12    int numblocks {(static_cast<int>(s.size()+ modlength-1)/modlength)};
13    int sourceindex {-1};
14    string finaloutput;
15
16    // going through the thing
17    for(int row = 0; row < numRows; ++row){
18        for(int i = 0; i<numblocks; ++i){
19
20            // first column of each block
21            sourceindex = row + modlength * i;
22            if (sourceindex<s.size()) {finaloutput += s[sourceindex];}
23
24        }
```

```
24     // continuing in case of boundary rows
25     if (row == 0 || row == numRows-1) {continue;}
26
27     // taking care of the case where non-boundary rows
28     sourceindex = modlength - row + modlength*i;
29     if (sourceindex < s.size())      {finaloutput += s[sourceindex];}
30 }
31 }
32
33 // printing the final output
34 cout << format("final-output = {}\n", finaloutput);
35
36
37 // return
38 return(0);
39
40 }
```

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 `i`th 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.

Examples

1. Example 1:

- Input: `height = [1,8,6,2,5,4,8,3,7]`
- Output: 49
- Explanation: The above vertical lines are represented by array `[1,8,6,2,5,4,8,3,7]`. In this case, the max area of water (blue section) the container can contain is 49.

2. Example 2:

- Input: `height = [1,1]`
- Output: 1

Constraints

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

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 }
```

12. Integer to Roman

Roman numerals are formed by appending the conversions of decimal place values from highest to lowest. Converting a decimal place value into a Roman numeral has the following rules:

- If the value does not start with 4 or 9, select the symbol of the maximal value that can be subtracted from the input, append that symbol to the result, subtract its value, and convert the remainder to a Roman numeral.
- If the value starts with 4 or 9 use the subtractive form representing one symbol subtracted from the following symbol, for example, 4 is 1 (I) less than 5 (V): IV and 9 is 1 (I) less than 10 (X): IX. Only the following subtractive forms are used: 4 (IV), 9 (IX), 40 (XL), 90 (XC), 400 (CD) and 900 (CM).
- Only powers of 10 (I, X, C, M) can be appended consecutively at most 3 times to represent multiples of 10. You cannot append 5 (V), 50 (L), or 500 (D) multiple times. If you need to append a symbol 4 times use the subtractive form.

Given an integer, convert it to a Roman numeral.

Examples

1. Example 1

- Input: num = 3749
- Output: "MMMDCCXLIX"
- Explanation:
 - 3000 = MMM as 1000 (M) + 1000 (M) + 1000 (M)
 - 700 = DCC as 500 (D) + 100 (C) + 100 (C)
 - 40 = XL as 10 (X) less of 50 (L)
 - 9 = IX as 1 (I) less of 10 (X)

- Note: 49 is not 1 (I) less of 50 (L) because the conversion is based on decimal places

2. Example 2:

- Input: num = 58
- Output: "LVIII"
- Explanation:
 - 50 = L
 - 8 = VIII

3. Example 3:

- Input: num = 1994
- Output: "MCMXCIV"
- Explanation:
 - 1000 = M
 - 900 = CM
 - 90 = XC
 - 4 = IV

Constraints

- $1 \leq \text{num} \leq 3999$

Code

```

1  int main(){
2
3      // input- configuration
4      int num    {1994};
5
6      // setup
7      vector<pair<int, string>> numToString {
8          {1, "I"},
9          {4, "IV"},
10         {5, "V"},
11         {9, "IX"},
12         {10, "X"},
13         {40, "XL"},
14         {50, "L"},
15         {90, "XC"},
16         {100, "C"},
17         {400, "CD"},
18         {500, "D"},
19         {900, "CM"},
20         {1000, "M"}
21     };
22     string finaloutput;
23     int    count;
24     auto mulstring = [](const int& count,
25                         const string& inputstring,
26                         string& finaloutput){
27         if (count == 0) {return;}
28         for(int i = 0; i<count; ++i){finaloutput += inputstring;}
29     };
30
31     // going through the hashmap from the end
32     for(int i = numToString.size()-1; i>=0; --i){
33
34         // number-string pairs
35         // variable to hold the final output
36         // variable that will hold the counts
37
38         // lambda-function for int * string

```

```
34     // calculating count
35     count  = num / numToString[i].first;
36     num    = num - numToString[i].first*count;
37
38     // adding to final output
39     mulstring(count, numToString[i].second, finaloutput);
40 }
41
42 // printing the final-output
43 cout << format("finaloutput = {}\n", finaloutput);
44
45 // return
46 return(0);
47
48 }
```

13. Roman To Integer

Roman numerals are represented by seven different symbols: I(1), V(5), X(10), L(50), C(100), D(500) and M(1000). For example, 2 is written as II in Roman numeral, just two ones added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II. Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

1. I can be placed before V (5) and X (10) to make 4 and 9.
2. X can be placed before L (50) and C (100) to make 40 and 90.
3. C can be placed before D (500) and M (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer.

Examples

1. Example 1

- Input: s = "III"
- Output: 3
- Explanation: III = 3.

2. Example 2

- Input: s = "LVIII"
- Output: 58

- Explanation: L = 50, V = 5, III = 3.

3. Example 3

- Input: s = "MCMXCIV"
- Output: 1994
- Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.

Constraints

1. $1 \leq s.length \leq 15$
2. s contains only the characters ('I', 'V', 'X', 'L', 'C', 'D', 'M').
3. It is guaranteed that s is a valid roman numeral in the range [1, 3999].

Code

```
1 int main(){
2
3     // input- configuration
4     string s {"MCMXCIV"};
5
6     // setup
7     int finaloutput {0};
8     unordered_map<char, int> charToInt {{'I', 1},
9                                           {'V', 5},
10                                          {'X', 10},
11                                          {'L', 50},
12                                          {'C', 100},
```

```
13         {'D', 500},
14         {'M', 1000}};
15
16 // going through the string
17 for(int i = 0; i<s.size(); ++i){
18     if ((i+1)<s.size() && charToInt[s[i]] < charToInt[s[i+1]]) {finaloutput -= charToInt[s[i]]; }
19     else {finaloutput += charToInt[s[i]]; }
20 }
21
22 // printing the final output
23 cout << format("finaloutput = {}\n", finaloutput);
24
25 // return
26 return(0);
27
28 }
```

14. Longest Common Prefix

Write a function to find the longest common prefix string amongst an array of strings. If there is no common prefix, return an empty string `""`.

Examples

1. Example 1:

- Input: `strs = ["flower", "flow", "flight"]`
- Output: `"fl"`

2. Example 2:

- Input: `strs = ["dog", "racecar", "car"]`
- Output: `""`
- Explanation: There is no common prefix among the input strings.

Constraints:

- $1 \leq \text{strs.length} \leq 200$
- $0 \leq \text{strs}[i].\text{length} \leq 200$
- `strs[i]` consists of only lowercase English letters if it is non-empty.

Code

```
1  int main(){
2
3      // input- configuration
4      vector<string> strs {
5          "flower",
6          "flow",
7          "flight"
8      };
9
10     // setup
11     int p          {0};                // index-pointer for boundary
12     int runcondition {true};           // breaking condition
13     string prefix;
14
15     // going through the vector
16     while(runcondition){
17
18         // breaking if it doesn't meet first words length
19         if (p >= str[0].size())        {++p; runcondition = false; break;}
20
21         // checking if this candidate
22         for(int i = 1; i<strs.size(); ++i){
23
24             // checking if valid
25             if (p >= str[i].size())    {runcondition = false; break;}
26
27             // checking if same
28             if (strs[i][p] != str[0][p]) {runcondition = false; break;}
29         }
30
31         // updating p
32         ++p;
33     }
```

```
34
35 // subsetting and printing the prefix
36 prefix = string(strs[0].begin(), strs[0].begin()+p-1);
37 cout << format("finaloutput = {}\n", prefix);
38
39 // return
40 return(0);
41
42 }
```

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`.

Examples

1. Example 1:

- Input: `nums = [1,1,2]`
- Output: 2, `nums = [1,2,_]`
- Explanation: Your function should return `k = 2`, with the first two elements of `nums` being 1 and 2 respectively. It does not matter what you leave beyond the returned `k` (hence they are underscores).

2. Example 2:

- Input: `nums = [0,0,1,1,1,2,2,3,3,4]`
- Output: 5, `nums = [0,1,2,3,4,_,_,_,_,_]`
- Explanation: Your function should return `k = 5`, with the first five elements of `nums` being 0, 1, 2, 3, and 4 respectively. It does not matter what you leave beyond the returned `k` (hence they are underscores).

Constraints:

- $1 \leq \text{nums.length} \leq 3 * 10^4$
- $-100 \leq \text{nums}[i] \leq 100$
- nums is sorted in non-decreasing order.

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`.

Examples

1. Example 1:

- Input: `nums = [3,2,2,3]`, `val = 3`
- Output: `2`, `nums = [2,2,_,_]_`
- Explanation: Your function should return `k = 2`, with the first two elements of `nums` being `2`. It does not matter what you leave beyond the returned `k` (hence they are underscores).

2. Example 2:

- Input: `nums = [0,1,2,2,3,0,4,2]`, `val = 2`
- Output: `5`, `nums = [0,1,4,0,3,_,_,_]_`
- Explanation: Your function should return `k = 5`, with the first five elements of `nums` containing `0`, `0`, `1`, `3`, and `4`. Note that the five elements can be returned in any order. It does not matter what you leave beyond the returned `k` (hence they are underscores).

Constraints

- $0 \leq \text{nums.length} \leq 100$
- $0 \leq \text{nums}[i] \leq 50$
- $0 \leq \text{val} \leq 100$

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
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 }
```

28. Find the Index of the First Occurrence in a String

Given two strings `needle` and `haystack`, return the index of the first occurrence of `needle` in `haystack`, or -1 if `needle` is not part of `haystack`.

Examples

1. Example 1:

- Input: `haystack = "sadbutsad"`, `needle = "sad"`
- Output: 0
- Explanation: "sad" occurs at index 0 and 6. The first occurrence is at index 0, so we return 0.

2. Example 2:

- Input: `haystack = "leetcode"`, `needle = "leeto"`
- Output: -1
- Explanation: "leeto" did not occur in "leetcode", so we return -1.

Constraints

- $1 \leq \text{haystack.length}, \text{needle.length} \leq 10^4$
- `haystack` and `needle` consist of only lowercase English characters.

Code

```
1 int main(){
2
3     // input- configuration
4     string haystack {"leetcode"};
5     string needle {"leeto"};
6
7
8     // setup
9     int finaloutput {-1};
10    auto beginsearch = [haystack, needle](int currindex){
11        // starting search
12        if(currindex + needle.size() > haystack.size()) {return false;}
13
14        // checking if they're a subset
15        for(int i = 0; i<needle.size(); ++i){
16            if (haystack[currindex + i] != needle[i]) {return false;}
17        }
18
19        return true;
20    };
21
22    // going through
23    for(int i = 0; i < haystack.size(); ++i){
24
25        // begin search at each index
26        auto curroutput = beginsearch(i);
27
28        // writing final output, if a mach
29        if (curroutput) {finaloutput = i; break;}
30    }
31
32    // printing final output
33
```

```
34     cout << format("final-output = {}\n", finaloutput);
35
36     // return
37     return(0);
38
39 }
```

36. Valid Sudoku

Determine if a 9 x 9 Sudoku board is valid. Only the filled cells need to be validated according to the following rules:

1. Each row must contain the digits 1-9 without repetition.
2. Each column must contain the digits 1-9 without repetition.
3. Each of the nine 3 x 3 sub-boxes of the grid must contain the digits 1-9 without repetition.

Examples

1. Example 1

- Input: board =
 - `[["5","3",".",".",".", "7",".",".",".", "."]`
 - `["6",".",".", "1","9","5",".",".", "."]`
 - `[[".", "9","8",".", "6",".", "3"]`
 - `["8",".", "6",".", "3"]`
 - `["4",".", "8",".", "3",".", "1"]`
 - `["7",".", "2",".", "6"]`
 - `[[".", "6",".", "2","8",".", "5"]`
 - `[[".", "4","1","9",".", "5"]`
 - `[[".", "8",".", "7","9"]]`

- Output: true

2. Example 2:

- Input: board =
 - `[["8","3",".",".","7",".",".",".","."],`
 - `["6",".",".","1","9","5",".",".","."],`
 - `[["9","8",".",".","","6","."],`
 - `["8",".",".","6",".",".","3"],`
 - `["4",".","8",".","3",".","1"],`
 - `["7",".","2",".","","6"],`
 - `[["6",".","","2","8","."],`
 - `[["","4","1","9","","5"],`
 - `[["","8","","7","9"]]`
- Output: false
- Explanation: Same as Example 1, except with the 5 in the top left corner being modified to 8. Since there are two 8's in the top left 3x3 sub-box, it is invalid.

Constraints

- `board.length == 9`
- `board[i].length == 9`
- `board[i][j]` is a digit 1-9 or '.'.

Note:

- A Sudoku board (partially filled) could be valid but is not necessarily solvable.
- Only the filled cells need to be validated according to the mentioned rules.

Code

```
1 // main-file
2 int main(){
3
4     // input-configuration
5     vector< vector<char> > board;
6     board.push_back({'5','3','.','.','7','.','.','.','.'});
7     board.push_back({'6','.','.','1','9','5','.','.','.'});
8     board.push_back({'.','9','8','.','.','.','.','6','.'});
9     board.push_back({'8','.','.','.','6','.','.','.','3'});
10    board.push_back({'4','.','.','8','.','3','.','.','1'});
11    board.push_back({'7','.','.','.','2','.','.','.','6'});
12    board.push_back({'.','6','.','.','.','.','2','8','.'});
13    board.push_back({'.','.'.','.','4','1','9','.','.','5'});
14    board.push_back({'.','.'.','.','.','8','.','.','7','9'});
15
16    // basic method
17    int xoffset, yoffset;
18    int row_local, col_local;
19
20    // lambda for converting char to inger
21    auto fConvert = [](char x) -> int {
22        if (x == '.') return -1;
23        else { return static_cast<int>(x - '0');}
24    };
25
26    // checking row and column entries
27    for(int i = 0; i < 9; ++i){
28
29        // register for each jumnn
30        vector<int> rowRegister(9, 0);
31        vector<int> colRegister(9, 0);
32        vector<int> blockRegister(9,0);
33
```

```

34 // going through each jumn
35 for(int j = 0; j<9; ++j){
36
37     // along the row
38     int var00 = fConvert(board[i][j]);
39     if (var00 != -1) {if (++rowRegister[var00-1] > 1) return false;}
40
41     // down the column
42     var00 = fConvert(board[j][i]);
43     if (var00 !=-1) {if (++colRegister[var00-1] > 1) return false;}
44
45     // checking block
46     row_local = j / 3;
47     col_local = j % 3;
48     xoffset = i / 3;
49     yoffset = i % 3;
50
51     // calculating registers
52     var00 = fConvert(board[3*xoffset + row_local][col_local+3*yoffset]);
53     if (var00!=-1) {if (++blockRegister[var00-1]>1) return false;}
54 }
55 }
56
57 // returning true
58 return true;
59 }

```

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

}

58. Length of Last Word

Given a string *s* consisting of words and spaces, return the length of the last word in the string. A word is a maximal substring consisting of non-space characters only.

Example

1. Example 1:

- Input: *s* = "Hello World"
- Output: 5
- Explanation: The last word is "World" with length 5.

2. Example 2:

- Input: *s* = " fly me to the moon "
- Output: 4
- Explanation: The last word is "moon" with length 4.

3. Example 3:

- Input: *s* = "luffy is still joyboy"
- Output: 6
- Explanation: The last word is "joyboy" with length 6.

Constraints

- $1 \leq \text{s.length} \leq 10^4$
- s consists of only English letters and spaces ' '.
- There will be at least one word in s.

Code

```
1 int main(){
2
3     // input- configuration
4     string s  {" fly me to the moon "};
5
6     // setup
7     int p1    {-1};
8     int finaloutput {-1};
9     string laststring;
10
11    // moving from the end
12    for(int i = s.size()-1; i>=0; --i){
13
14        // continuing until you find a non-space character
15        if (s[i] == ' ') {continue;}
16
17        // launch the start of first word
18        p1 = i;
19
20        // moving p1 until we find the first space or nonword thing
21        while(p1>=0 && s[p1]!=' '){--p1;}
22
23        // calculating the length
```

```
24     finaloutput = i - p1;
25     laststring = string(s.begin() + p1, s.begin() + i+1);
26
27     // breaking
28     break;
29 }
30
31 // printing
32 cout << format("length = {}, last-word = {}\n", finaloutput, laststring);
33
34 // return
35 return(0);
36
37 }
```

68. Text Justification

Given an array of strings `words` and a width `maxWidth`, format the text such that each line has exactly `maxWidth` characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly `maxWidth` characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line does not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left-justified, and no extra space is inserted between words.

Note:

1. A word is defined as a character sequence consisting of non-space characters only.
2. Each word's length is guaranteed to be greater than 0 and not exceed `maxWidth`.
3. The input array `words` contains at least one word.

Examples

1. Example 1:

- Input: `words = ["This", "is", "an", "example", "of", "text", "justification."], maxWidth = 16`
- Output:
 - [
 - "This is an",
 - "example of text",

- "justification. "
-]

2. Example 2:

- Input: words = ["What","must","be","acknowledgment","shall","be"], maxWidth = 16
- Output:
 - [
 - "What must be",
 - "acknowledgment ",
 - "shall be "
 -]
- Explanation: Note that the last line is "shall be " instead of "shall be", because the last line must be left-justified instead of fully-justified. Note that the second line is also left-justified because it contains only one word.

3. Example 3:

- Input: words = ["Science","is","what","we","understand","well","enough","to","explain","to","a","computer.","Art","is","everything"], maxWidth = 20
- Output:
 - [
 - "Science is what we",
 - "understand well",
 - "enough to explain to",
 - "a computer. Art is",
 - "everything else we",
 - "do "
 -]

Constraints

- $1 \leq \text{words.length} \leq 300$
- $1 \leq \text{words}[i].\text{length} \leq 20$
- $\text{words}[i]$ consists of only English letters and symbols.
- $1 \leq \text{maxWidth} \leq 100$
- $\text{words}[i].\text{length} \leq \text{maxWidth}$

Code

```
1 // function to calculate number of non-space characters
2 int fCalcLengthOfTempWithoutSpaces(std::vector<std::string> temp){
3     int num_nonspaces = 0;
4     for(auto x: temp) num_nonspaces += x.size();
5     return num_nonspaces;
6 }
7
8 // function to calculate words formed with temp
9 int fCalcLengthOfTempWithSpaces(std::vector<std::string> temp){
10     int num_nonspaces = 0;
11     for(auto x: temp) num_nonspaces += 1+ x.size();
12     return num_nonspaces-1;
13 }
14
15 // printing temp
16 void fPrintTemp(std::vector<std::string> temp){
17     // printing temp
18     std::cout << "temp = ";
19     for(auto x: temp) std::cout << x << ", ";
```

```

20     std::cout << std::endl;
21 }
22
23 // main-file =====
24 int main(){
25
26     // input- configuration
27     auto words    = vector<string>({"This", "is", "an", "example", "of", "text", "justification."});
28     auto maxWidth {16};
29
30     // setup
31     std::vector<std::string> finalOutput;
32
33     // going through strings
34     int acc = 0;
35     int numwords = 0;
36     int currwidth = 0;
37     std::vector<std::string> temp;
38
39     for(int i = 0; i<words.size(); ++i){
40
41         // updating temp
42         temp.push_back(words[i]);    // updating words in temp
43
44         // checking if width has been crossed
45         if (fCalcLengthOfTempWithSpaces(temp) >= maxWidth || i == words.size()-1){
46
47             // condition temp based on length
48             if(fCalcLengthOfTempWithSpaces(temp)>maxWidth){
49                 temp.pop_back();    // last words gotta go
50                 --i;                // making sure its taken care of in next iteration
51             }
52
53             // finding length of characters in temp
54             int num_nonspaces = fCalcLengthOfTempWithoutSpaces(temp);

```

```

55
56 // finding number of spaces to add
57 int numspacetofill = maxWidth - num_nonspaces;
58
59 // calculating numspots
60 int numspots = temp.size()-1;
61
62 // calculating ideal number of spaces
63 int idealnumspacesperspot;
64 int remainders;
65 if (numspots!=0){
66     idealnumspacesperspot = numspacetofill/numspots;
67     remainders             = numspacetofill%numspots;
68 }
69 else{
70     idealnumspacesperspot = numspacetofill/1;
71     remainders             = numspacetofill%1;
72 }
73
74 // constructing candidate string
75 std::string candidate;
76
77 // adding each word in temp to the candidate
78 for(int j = 0; j < temp.size(); ++j){
79
80     // fetching word
81     auto x = temp[j];
82
83     // adding word to candidate
84     candidate += x;
85
86     // adding spaces
87     if (j!=temp.size()-1)
88         for(int var00 = 0; var00 < idealnumspacesperspot; ++var00)
89             candidate += " ";

```



```

90
91     // checking if there is any remainder left
92     if (remainders > 0){
93         candidate += " ";
94         --remainders;
95     }
96 }
97
98 // checking if there are remaindeers
99 while (remainders > 0){
100     candidate += " ";
101     --remainders;
102 }
103
104 while (candidate.size() != maxWidth) {candidate += " ";}
105
106 // appending candidate to final output
107 finalOutput.push_back(candidate);
108
109 // getting rid of everything
110 if (i != words.size()-1) {temp.clear();}
111 }
112 }
113
114 // making function left justified
115 std::string lastline;
116 for(int i = 0; i < temp.size(); ++i){
117     lastline += temp[i];
118     if (i != temp.size()-1) {lastline += " ";}
119 }
120
121 // adding spaces until end
122 while(lastline.size() != maxWidth) {lastline += " ";}
123
124 // replacing last line

```

```
125     finalOutput[finalOutput.size()-1] = lastline;
126
127     // return
128     return(0);
129
130 }
```

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`.

Examples

1. Example 1:

- Input: `nums1 = [1,2,3,0,0,0]`, `m = 3`, `nums2 = [2,5,6]`, `n = 3`
- Output: `[1,2,2,3,5,6]`
- Explanation: The arrays we are merging are `[1,2,3]` and `[2,5,6]`. The result of the merge is `[1,2,2,3,5,6]` with the underlined elements coming from `nums1`.

2. Example 2:

- Input: `nums1 = [1]`, `m = 1`, `nums2 = []`, `n = 0`
- Output: `[1]`
- Explanation: The arrays we are merging are `[1]` and `[]`. The result of the merge is `[1]`.

3. Example 3:

- Input: `nums1 = [0]`, `m = 0`, `nums2 = [1]`, `n = 1`

- Output: [1]
- Explanation: The arrays we are merging are [] and [1]. The result of the merge is [1]. Note that because $m = 0$, there are no elements in nums1. The 0 is only there to ensure the merge result can fit in nums1.

Constraints:

1. $\text{nums1.length} == m + n$
2. $\text{nums2.length} == n$
3. $0 \leq m, n \leq 200$
4. $1 \leq m + n \leq 200$
5. $-10^9 \leq \text{nums1}[i], \text{nums2}[j] \leq 10^9$

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 *i*th 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.

Examples

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

Constraints:

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

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 }
```

125. Valid Palindrome

A phrase is a palindrome if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers. Given a string *s*, return true if it is a palindrome, or false otherwise.

Examples

1. Example 1:

- Input: *s* = "A man, a plan, a canal: Panama"
- Output: true
- Explanation: "amanaplanacanalpanama" is a palindrome.

2. Example 2:

- Input: *s* = "race a car"
- Output: false
- Explanation: "raceacar" is not a palindrome.

3. Example 3:

- Input: *s* = ""
- Output: true
- Explanation: *s* is an empty string "" after removing non-alphanumeric characters. Since an empty string reads the same forward and backward, it is a palindrome.

Constraints

- $1 \leq s.length \leq 2 * 10^5$
- s consists only of printable ASCII characters.

Code

```
1 int main(){
2
3     // input- configuration
4     auto s = string("A man, a plan, a canal: Panama");
5
6     // setup
7     auto pleft      {0};
8     auto pright     {static_cast<int>(s.size())-1};
9     auto finaloutput {true};
10
11     // lambda to check if alphanumeric
12     auto isalphanumeric = [](const int& x){
13         if (x-65 >= 0 && 90-x >= 0) {return true;} // upper-case check
14         if (x-97 >= 0 && 122-x >= 0) {return true;} // lower-case check
15         if (x-48 >= 0 && 57-x >= 0) {return true;} // numeric check
16         return false;
17     };
18
19     // running
20     while(pleft < s.size() && pright >= 0){
21
22         // moving pleft until we find the position
23         while(pleft < s.size() && isalphanumeric(s[pleft]) == false) {++pleft;}
24         while(pright >= 0 && isalphanumeric(s[pright]) == false) {--pright;}
25     }
```

```
26 // checking bounds
27 if (pleft>=pright) {break;}
28
29 // checking if they're the same
30 if (std::tolower(s[pleft]) != std::tolower(s[pright])) {finaloutput = false; break;}
31
32 // updating pointers
33 ++pleft; --pright;
34 }
35
36 // printing
37 cout << format("final-output = {}\n", finaloutput);
38
39 // return
40 return(0);
41
42 }
```

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 }
```

167. Two Sum II - Input Array Is Sorted

Given a 1-indexed array of integers `numbers` that is already sorted in non-decreasing order, find two numbers such that they add up to a specific target number. Let these two numbers be `numbers[index1]` and `numbers[index2]` where $1 \leq \text{index1} < \text{index2} \leq \text{numbers.length}$.

Return the indices of the two numbers, `index1` and `index2`, added by one as an integer array `[index1, index2]` of length 2.

The tests are generated such that there is exactly one solution. You may not use the same element twice.

Your solution must use only constant extra space.

Examples

1. Example 1:

- Input: `numbers = [2,7,11,15]`, `target = 9`
- Output: `[1,2]`
- Explanation: The sum of 2 and 7 is 9. Therefore, `index1 = 1`, `index2 = 2`. We return `[1, 2]`.

2. Example 2:

- Input: `numbers = [2,3,4]`, `target = 6`
- Output: `[1,3]`
- Explanation: The sum of 2 and 4 is 6. Therefore `index1 = 1`, `index2 = 3`. We return `[1, 3]`.

3. Example 3:

- Input: `numbers = [-1,0]`, `target = -1`

- Output: [1,2]
- Explanation: The sum of -1 and 0 is -1. Therefore $\text{index1} = 1$, $\text{index2} = 2$. We return [1, 2].

Constraints

- $2 \leq \text{numbers.length} \leq 3 * 10^4$
- $-1000 \leq \text{numbers}[i] \leq 1000$
- numbers is sorted in non-decreasing order.
- $-1000 \leq \text{target} \leq 1000$
- The tests are generated such that there is exactly one solution.

Code

```
1 int main(){
2
3     // input- configuration
4     auto numbers = vector<int>{2,7,11,15};
5     auto target  = 9;
6
7     // setup
8     std::vector<int> finalOutput;
9     auto left    {0};
10    auto right   {static_cast<int>(numbers.size()-1)};
11    auto currsum {0};
12
13    // usual left-right loop
14    while (left < right){
```

```
15
16 // checking sum of two values
17 currsum = numbers[left] + numbers[right];
18
19 // comparing against target
20 if (currsum > target)    {--right;}
21 else if (currsum < target) {++left;}
22 else {
23     finalOutput.push_back( left+1);
24     finalOutput.push_back( right+1);
25     break;
26 }
27 }
28
29 // printign the final output
30 cout << format("finaloutput = {}\n", finalOutput);
31
32 // return
33 return(0);
34
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.

Examples

- **Example 1**

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

- **Example 2**

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

Constraints:

- `n == nums.length`
- $1 \leq n \leq 5 * 10^4$
- $-10^9 \leq \text{nums}[i] \leq 10^9$

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 Rotate Array

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

Examples

- **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]`

Constraints

- $1 \leq \text{nums.length} \leq 10^5$
- $-2^31 \leq \text{nums}[i] \leq 2^31 - 1$
- $0 \leq k \leq 10^5$

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     }
```

```
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 }
```

209. Minimum Size Subarray Sum

Given an array of positive integers `nums` and a positive integer `target`, return the minimal length of a subarray whose sum is greater than or equal to `target`. If there is no such subarray, return 0 instead.

Examples

1. Example 1:

- Input: `target = 7`, `nums = [2,3,1,2,4,3]`
- Output: 2
- Explanation: The subarray `[4,3]` has the minimal length under the problem constraint.

2. Example 2:

- Input: `target = 4`, `nums = [1,4,4]`
- Output: 1

3. Example 3:

- Input: `target = 11`, `nums = [1,1,1,1,1,1,1,1]`
- Output: 0

Constraints

- $1 \leq \text{target} \leq 10^9$

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

Code

```
1 int main(){
2
3     // input- configuration
4     vector<int> nums {1,2,3,4,5};
5     auto target {15};
6
7     // setup
8     auto finaloutput {std::numeric_limits<int>::max()};
9     auto pleft {0};
10    auto sum {nums[pleft]};
11    auto i {1};
12
13    // in case the first element itself is greater
14    if (sum > target) {finaloutput = 1;}
15
16    // lambda to update finaloutput
17    auto updatefinaloutput = [&sum,
18                             &target,
19                             &finaloutput,
20                             &i,
21                             &pleft,
22                             &nums]() -> void{
23
24        auto numelements = i - pleft + 1;
25        finaloutput = numelements < finaloutput ? numelements : finaloutput;
26        sum -= nums[pleft++];
27    };
```

```
28
29 // going through the array
30 for(; i<nums.size(); ++i){
31
32     // adding to sum
33     sum += nums[i];
34
35     // updating
36     while (sum>=target) {updatefinaloutput();}
37 }
38
39 // updating
40 if(finaloutput == std::numeric_limits<int>::max()) {finaloutput = 0;}
41
42 // printing the finaloutput
43 cout << format("finaloutput = {}\n", finaloutput);
44
45 // return
46 return(0);
47
48 }
```

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 }
```

283. Move Zeros

Given an integer array `nums`, move all 0's to the end of it while maintaining the relative order of the non-zero elements. Note that you must do this in-place without making a copy of the array.

Examples

1. Example 1:

- Input: `nums = [0,1,0,3,12]`
- Output: `[1,3,12,0,0]`

2. Example 2:

- Input: `nums = [0]`
- Output: `[0]`

Constraints:

- $1 \leq \text{nums.length} \leq 10^4$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$

Code

```
1 int main(){
2
3     // input- configuration
4     vector<int> nums {0,1,0,3,12};
5
6     // setup
7     int explorer {0};
8     int anchor {0};
9
10    // going through the nums
11    while(explorer < nums.size()){
12
13        // moving explorer until we arrive at a non-zero value
14        while(explorer < nums.size() && nums[explorer] == 0) {explorer++;}
15
16        // copying value
17        if (explorer<nums.size() && anchor <nums.size())
18            nums[anchor++] = nums[explorer++];
19    }
20
21    // zeroing out the rest
22    while(anchor < nums.size()) {nums[anchor++] = 0;}
23
24    // printing the finaloutput
25    cout << format("finaloutput = "); fPrintVector(nums);
26
27    // return
28    return(0);
29
30 }
```

345. Reverse Vowels Of A String

Given a string *s*, reverse only all the vowels in the string and return it. The vowels are 'a', 'e', 'i', 'o', and 'u', and they can appear in both lower and upper cases, more than once.

Examples

1. Example 1:

- Input: *s* = "IceCreAm"
- Output: "AceCreIm"
- Explanation: The vowels in *s* are ['I', 'e', 'e', 'A']. On reversing the vowels, *s* becomes "AceCreIm".

2. Example 2:

- Input: *s* = "leetcode"
- Output: "leotcede"

Constraints

- $1 \leq s.length \leq 3 * 10^5$
- *s* consist of printable ASCII characters.

Code

```
1 int main(){
2
3     // input- configuration
4     string s {"leetcode"};
5
6     // going through the string
7     string vowels {"aeiouAEIOU"};
8     vector<int> vowel_indices;
9     string reversed_vowels;
10    string finaloutput = s;
11
12    // going through the string
13    for(int i = 0; i<s.size(); ++i){
14        if (vowels.find(s[i]) != string::npos){
15            reversed_vowels+=s[i];
16            vowel_indices.push_back(i);
17        }
18    }
19
20    // refilling the indices
21    for(int i = 0; i<vowel_indices.size(); ++i){
22        finaloutput[vowel_indices[i]] = reversed_vowels[reversed_vowels.size()-1-i];
23    }
24
25    // printing the final output
26    cout << format("finaloutput = {}\n", finaloutput);
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).

Examples

1. Example 1:

- Input: $s = \text{"abc"}, t = \text{"ahbgdc"}$
- Output: true

2. Example 2:

- Input: $s = \text{"axc"}, t = \text{"ahbgdc"}$
- Output: false

Constraints

- $0 \leq s.length \leq 100$
- $0 \leq t.length \leq 10^4$
- s and t consist only of lowercase English letters.

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 }
```

394. Decode String

Given an encoded string, return its decoded string.

The encoding rule is: $k[\text{encoded_string}]$, where the `encoded_string` inside the square brackets is being repeated exactly k times. Note that k is guaranteed to be a positive integer.

You may assume that the input string is always valid; there are no extra white spaces, square brackets are well-formed, etc. Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, k . For example, there will not be input like `3a` or `2[4]`.

The test cases are generated so that the length of the output will never exceed 10^5 .

Examples

1. Example 1:

- Input: `s = "3[a]2[bc]"`
- Output: `"aaabcbc"`

2. Example 2:

- Input: `s = "3[a2[c]]"`
- Output: `"accaccacc"`

3. Example 3:

- Input: `s = "2[abc]3[cd]ef"`
- Output: `"abcbcbccdcddcdef"`

Constraints:

- $1 \leq s.length \leq 30$
- s consists of lowercase English letters, digits, and square brackets '['].
- s is guaranteed to be a valid input.
- All the integers in s are in the range [1, 300].

Code

```
1 int main(){
2
3     // input- configuration
4     string s {"100[leetcode]"};
5
6     // running
7     std::stack<char> mystack;                // stack
8     string temp;                            // temporary string used for decoding
9     int repcount {1};                      // used for decoding
10
11     // going through the inputs
12     for(int i = 0; i<s.size(); ++i){
13
14         if(s[i] != '[') {mystack.push(s[i]);} // pushing characters to stack until we
15         // arrive at "]"
16         else{
17             temp = "";                        // initializing temporary string
18             while(mystack.top() != '[') {    // expanding mini-string until we arrive at
19                 temp = mystack.top() + temp;
20                 mystack.pop();
21             }
22         }
23     }
24 }
```



```

20     }
21
22     mystack.pop(); // removing "["
23
24     // calculating the repcount
25     string numberasstring = "";
26     while(mystack.size() != 0 &&
27           mystack.top() - '0' >= 0 && '9' - mystack.top() >= 0)
28     {
29         numberasstring = mystack.top() + numberasstring;
30         mystack.pop();
31     }
32     repcount = std::stoi(numberasstring);
33
34     // mini-decoding
35     int multitempsize = repcount * temp.size(); // calculating size after multiplication
36     for(int j = 0; j<multitempsize; ++j) { // filling up the stack with decoded content
37         mystack.push(temp[j%temp.size()]);
38     }
39 }
40 }
41
42 // creating the final output
43 string finaloutput;
44 while(mystack.size()){
45     finaloutput = mystack.top() + finaloutput;
46     mystack.pop();
47 }
48
49 // printing the final output
50 cout << format("finaloutput = {}\n", finaloutput);
51
52 // return
53 return(0);
54

```


443. String Compression

Given an array of characters `chars`, compress it using the following algorithm:

Begin with an empty string `s`. For each group of consecutive repeating characters in `chars`:

1. If the group's length is 1, append the character to `s`.
2. Otherwise, append the character followed by the group's length.

The compressed string `s` should not be returned separately, but instead, be stored in the input character array `chars`. Note that group lengths that are 10 or longer will be split into multiple characters in `chars`. After you are done modifying the input array, return the new length of the array. You must write an algorithm that uses only constant extra space.

Examples

1. Example 1:

- Input: `chars = ["a","a","b","b","c","c","c"]`
- Output: Return 6, and the first 6 characters of the input array should be: `["a","2","b","2","c","3"]`
- Explanation: The groups are "aa", "bb", and "ccc". This compresses to "a2b2c3".

2. Example 2:

- Input: `chars = ["a"]`
- Output: Return 1, and the first character of the input array should be: `["a"]`
- Explanation: The only group is "a", which remains uncompressed since it's a single character.

3. Example 3:

- Input: chars = ["a","b","b","b","b","b","b","b","b","b","b","b","b"]
- Output: Return 4, and the first 4 characters of the input array should be: ["a","b","1","2"].
- Explanation: The groups are "a" and "bbbbbbbbbbbb". This compresses to "ab12".

Constraints

- $1 \leq \text{chars.length} \leq 2000$
- chars[i] is a lowercase English letter, uppercase English letter, digit, or symbol.

Code

```
1 int main(){
2
3     // input- configuration
4     vector<char> chars {'a','b','b','b','b','b','b','b','b','b','b','b','b'};
5
6     // going through the character
7     int p1          {0};
8     char runningchar {};
9     char curr       {};
10    int count        {0};
11    string finaloutput;
12
13    // going through the inputs
14    while(p1<chars.size()){
15
16        // getting current tchar
```

```

17     curr = chars[p1];
18
19     // increasing count
20     if (count == 0)           {runningchar = chars[p1]; ++count;}
21     else if(curr == runningchar) {++count;}
22     else if(curr != runningchar) {
23         finaloutput += runningchar; // writing character to current pointer
24         if (count != 1)
25             finaloutput += std::to_string(count); // increasing write-pointer
26         runningchar = curr;
27         count = 1;
28     }
29
30     // increasing pointer
31     ++p1;
32 }
33
34 // flushing out
35 if (count != 0){
36     finaloutput += runningchar; // writing character to current pointer
37     if (count != 1)
38         finaloutput += std::to_string(count); // increasing write-pointer
39 }
40
41 // writing to input
42 for(int i = 0; i<finaloutput.size(); ++i){
43     chars[i] = finaloutput[i];
44 }
45
46 // printing the final output
47 cout << format("finaloutput = {}\n", finaloutput);
48 cout << "chars = "; fPrintVector(chars);
49 cout << format("return-value = {}\n", finaloutput.size());
50
51 // return

```

```
52     return(0);  
53  
54 }
```
