Leetcode Solutions

SVR

August 20, 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

```cpp
using std::map;
using std::format;
using std::deque;
using std::pair;

// vector printing function
template<typename T>
void fPrintVector(vector<T> input){
    for(auto x: input) cout << x << ",";
    cout << endl;
}

template<typename T>
void fPrintMatrix(vector<T> input){
    for(auto x: input){
        for(auto y: x){
            cout << y << ",";
        }
        cout << endl;
    }
```

```
}

template<typename T, typename T1>
void fPrintHashmap(unordered_map<T, T1> input){
    for(auto x: input){
        cout << format("[{},{}] \n", x.first, x.second);
    }
    cout <<endl;
}

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};


struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};

void fPrintBinaryTree(TreeNode* root){
    // sending it back
    if (root == nullptr) return;

    // printing
    PRINTLINE
    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
```

```
    void fetchtime(){
        duration = std::chrono::duration_cast<std::chrono::nanoseconds>(endpoint - startpoint);
        cout << format("{} nanoseconds \n", duration.count());
    }
    void fetchtime(string stringarg){
        duration = std::chrono::duration_cast<std::chrono::nanoseconds>(endpoint - startpoint);
        cout << format("{} took {} nanoseconds \n", stringarg, duration.count());
    }
};


// main-file =========================================================================
int main(){

    // input- configuration




    // return
    return(0);

}
```

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

```cpp
int main(){

    // input- configuration
    vector<int> nums  {2, 7, 11, 15};
    int        target {9};

    // setup
    int                   complement    {0};
    unordered_map<int, int> number_to_index;
    vector<int>           finaloutput;

    // filling the unordered_map
    for(int i = 0; i < nums.size(); ++i){

        // calculating complement
        complement = target - nums[i];

        // checking if complement is present in registry
        if(number_to_index.find(complement) != number_to_index.end()) [[unlikely]]
        {
            finaloutput.push_back(number_to_index[complement]); // adding first index
            finaloutput.push_back(i);                           // adding second index
            break;                                              // breaking out
        }
```

```
        else [[likely]]
        {
            // check if current element is present
            if (number_to_index.find(nums[i]) == number_to_index.end()) [[likely]]
            {
                // adding the [number, index] pair to the hashmap
                number_to_index[nums[i]] = i;
            }
            else [[unlikely]]
            {
                // we'll do nothing since the number and its index is already present
                continue;
            }
        }
    }

    // printing the final output
    for(const auto& x : finaloutput) {cout << x << ", ";} cout << endl;

    // return
    return(0);

}
```

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

```
int main(){

    // input- configuration
    ListNode* l1   = new ListNode(2);
    l1->next       = new ListNode(4);
    l1->next->next = new ListNode(3);

    ListNode* l2   = new ListNode(5);
    l2->next       = new ListNode(6);
    l2->next->next = new ListNode(4);

    // setup
    ListNode* traveller_1 = l1;
    ListNode* traveller_2 = l2;
    ListNode* finalOutput = new ListNode(-1);
    ListNode* traveller_fo = finalOutput;

    int sum               {0};
    int carry             {0};
    int value_1           {0};
    int value_2           {0};

    // moving through the two nodes
    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

```
int main(){

    // input- configuration
    string s   {"tmmzuxt"};

    // setup
    unordered_map<char, int> histogram;
    int p1 {0};
    char curr;
    int finaloutput {-1};
    int temp_length {-1};

    // going through the thing
    for(int p2 = 0; p2<s.size(); ++p2){

        // moving to another variable
        curr = s[p2];

        // checking if current character is in histogram
        if (histogram.find(curr) == histogram.end()) [[unlikely]]
        {
```

```
            histogram[curr] = 1;
        }
        else [[likely]]
        {
            // checking if count is zero
            if (histogram[curr] == 0)
            {
                histogram[curr] = 1;
            }
            else
            {
                // moving p1 until it arrives at first instance of curr
                while(s[p1] != curr)
                {
                    --histogram[s[p1]];
                    ++p1;
                }
                ++p1;
                histogram[curr] = 1;
            }
        }

        // calculating longest length
        finaloutput = finaloutput > (p2-p1+1) ? finaloutput : (p2-p1+1);
    }

    // printing
    cout << format("longest length = {} \n", finaloutput);

    // return
    return(0);
}
```

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

```
int main(){

    // input- configuration
    vector<int> height {1,8,6,2,5,4,8,3,7};

    // setup
    int left      {0};
    int right     {static_cast<int>(height.size())-1};
    int maxvolume {-1};
    int currvolume {-1};

    // two-pointer approach
    while(left < right){

        // calculating volumes
        currvolume = (right - left) * std::min(height[left], height[right]);
        maxvolume  = maxvolume > currvolume ? maxvolume : currvolume;

        // adjusting left and right based on volume
```

```
        if (height[left] < height[right]) {++left;}
        else                              {--right;}
    }

    // printing
    cout << format("maxvolume = {}\n", maxvolume);

    // return
    return(0);

}
```

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

```cpp
int main(){

    // input- configuration
    int num    {1994};

    // setup
    vector<pair<int, string>> numToString {
        {1, "I"},
        {4, "IV"},
        {5, "V"},
        {9, "IX"},
        {10, "X"},
        {40, "XL"},
        {50, "L"},
        {90, "XC"},
        {100, "C"},
        {400, "CD"},
        {500, "D"},
        {900, "CM"},
        {1000, "M"}
    };                                                          // number-string pairs
    string finaloutput;                                         // variable to hold the final output
    int    count;                                               // variable that will hold the counts
    auto mulstring = [](const int& count,
                        const string& inputstring,
                        string& finaloutput){
        if (count == 0) {return;}
        for(int i = 0; i<count; ++i){finaloutput += inputstring;}
    };                                                          // lambda-function for int * string
         multiplications (python style)

    // going through the hashmap from the end
    for(int i = numToString.size()-1; i>=0; --i){

```

```
        // calculating count
        count   = num / numToString[i].first;
        num     = num - numToString[i].first*count;

        // adding to final output
        mulstring(count, numToString[i].second, finaloutput);
    }

    // printing the final-output
    cout << format("finaloutput = {}\n", finaloutput);

    // return
    return(0);

}
```

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 ≤ 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

```
int main(){

    // input- configuration
    string s   {"MCMXCIV"};

    // setup
    int finaloutput {0};
    unordered_map<char, int> charToInt {{'I', 1},
                                        {'V', 5},
                                        {'X', 10},
                                        {'L', 50},
                                        {'C', 100},
```

```
                                    {'D', 500},
                                    {'M', 1000}};

    // going through the string
    for(int i = 0; i<s.size(); ++i){
        if ((i+1)<s.size() && charToInt[s[i]] < charToInt[s[i+1]]) {finaloutput -= charToInt[s[i]];}
        else                                                       {finaloutput += charToInt[s[i]];}
    }

    // printing the final output
    cout << format("finaloutput = {}\n", finaloutput);

    // return
    return(0);

}
```

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

```
int main(){

    // input- configuration
    vector<int> nums      {1,1};

    // setup
    int p      {0};
    int counter {0};

    // going through the values
    for(int i = 1; i<nums.size(); ++i){

        // check values
        if (nums[i] == nums[p]) {continue;}

        // writing values
        ++p;
        nums[p] = nums[i];
```

```
        ++counter;
    }

    // printing the final output
    cout << format("final-output = {}\n", counter+1);
    cout << format("nums = "); fpv(nums);

    // return
    return(0);

}
```

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

```
int main(){

    // input- configuration
    vector<int> nums  {0,1,2,2,3,0,4,2};
    int val           {2};

    // setup
    int src       {0};
    int dest      {0};
    int numwrites {0};

    // going through the indices
    while(src < nums.size()){

        // moving the dest until we find a val-position
        while(nums[dest] != val) {++dest;}

        // moving source until we find a non-val position after dest
        src = std::max(src, dest+1);
        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  }
```

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

Code

```
int main(){

    // input- configuration
    vector<int> height {0,1,0,2,1,0,1,3,2,1,2,1};

    // setup
    vector<int> leftmaxes(height.size(), 0);                    // vector holding biggest-height to left
    vector<int> rightmaxes(height.size(), 0);                   // vector holding biggest-height to the right
    int forwardindex  {0};                                      // for maintaining forward-index
    int backwardindex {0};                                      // for maintaining backward-index
    int maxleft       {-1};                                     // keeping record of biggest left
    int maxright      {-1};                                     // keeping record of biggest right
    int finaloutput {0};                                        // storing final output

    // building left-max
    for(int i = 1; i<height.size(); ++i){

        // calculating indices
        forwardindex   = i;                                     // forward-index
        backwardindex  = height.size()-1-i;                     // backward-index

        // calculating maxleft
        maxleft    = height[forwardindex-1] > maxleft ?
                     height[forwardindex-1] : maxleft;          // running max-left
        leftmaxes[forwardindex] = maxleft;                      // storing to vector

        // calculating max right
        maxright   = height[backwardindex+1] > maxright ?
                     height[backwardindex+1] : maxright;        // running max-right
        rightmaxes[backwardindex] = maxright;                   // storing to vector
    }

    // going through the array to calculate maxvolume held by each column
```

```
    for(int i = 0; i < height.size(); ++i){

        // finding max-height of the current column
        auto minheight    = std::min({leftmaxes[i], rightmaxes[i]});     // finding max-height of borders
        auto columnheight = minheight - height[i];                       // subtracting to find space
        columnheight      = columnheight > 0 ? columnheight : 0;         // in case curr-height > max-height
        finaloutput       += columnheight;                               // accumlating to water content
    }

    // printing the final output
    cout << format("finaloutput = {}\n", finaloutput);

    // return
    return(0);

}
```

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 $\leq$ 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 nums[n - 1].

Code

```
int main(){

    // input- configuration
    vector<int> nums  {2,3,0,1,4};

    // setup
    Timer timer;                                                    // setting a timer
    vector<int> minjumps(nums.size(),0);                            // the dp table
    int leftboundary  {-1};                                         // variable to hold the left-boundary
    int rightboundary {-1};                                         // variable to hold the right-boundary

    // moving from the back
    for(int i = nums.size()-2; i>=0; --i){

        // continuign if nums[i] = 0
        if (nums[i] == 0) {
            minjumps[i] = std::numeric_limits<int>::max();          // to prevent this from being chosen
            continue;                                               // moving to next index
        }

        // range of values it can go from here
        leftboundary  = i+1;                                        // the starting point of range
        rightboundary = i+nums[i];                                  // the end point of range
```

```
        rightboundary  = rightboundary < nums.size()-1 ?
                         rightboundary : nums.size()-1;                    // ensuring within vector range

        // calculating smallest element in range
        auto it = std::min_element(minjumps.begin()+leftboundary,
                                   minjumps.begin()+rightboundary+1);      // finding the minimum value in the range

        // addding min-element to the array
        if (*it == std::numeric_limits<int>::max())
            minjumps[i] = std::numeric_limits<int>::max();                 // ensuring infty logic
        else
            minjumps[i] = (1 + *it);                                       // for regular values

    }

    // printing
    cout << format("finaloutput = {}\n", minjumps[0]);
    timer.measure();

    // return
    return(0);

}
```

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

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

Code

```cpp
int main(){

    // input- configuration
    vector<int> nums  {3,2,1,0,4};

    // setup
    Timer timer;                                                    // starting a timer
    int maxjumpdistance   {0};                                      // variable holding max-jump-distance
    int currjumpdistance  {0};                                      // variable holding max-jump-distance from here
    int finaloutput       {0};                                      // variable holding final verdict

    // going through the nums
    for(int i = 0; i<=maxjumpdistance && i<nums.size(); ++i){

        // calculating max-distance we can go from here
        currjumpdistance  = i + nums[i];

        // updating max-jumpdistance
        maxjumpdistance = currjumpdistance > maxjumpdistance ? \
                        currjumpdistance : maxjumpdistance;

    }

    // updating the final output
    finaloutput = maxjumpdistance >= nums.size()-1 ? true : false;

    // printing the thing
    cout << format("final-output = {}\n", finaloutput);
    timer.measure();


    // return
    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

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

Code

```
int main(){

    // input- configuration
    string s   {"  fly me  to  the moon "};

    // setup
    int p1        {-1};
    int finaloutput {-1};
    string laststring;

    // moving from the end
    for(int i = s.size()-1; i>=0; --i){

        // continuing until you find a non-space character
        if (s[i] == ' ') {continue;}

        // launch the start of first word
        p1 = i;

        // moving p1 until we find the first space or nonword thing
        while(p1>=0 && s[p1]!=' ') {--p1;}

        // calculating the length
```

```
        finaloutput = i - p1;
        laststring = string(s.begin() + p1, s.begin() + i+1);

        // breaking
        break;
    }

    // printing
    cout << format("length = {}, last-word = {}\n", finaloutput, laststring);

    // return
    return(0);

}
```

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

```cpp
int main(){

    // input- configuration
    vector<int> nums  {1,1,1,2,2,3};

```

```
    // setup
    int destination   {1};
    int prev          {nums[0]};
    int element_counter {1};
    int numwrites     {1};

    // going through the values
    for(int i = 1; i < nums.size(); ++i){

        // updating counter
        if (nums[i-1] == nums[i]) {++element_counter;}
        else                      {element_counter = 1;}

        // checking the element counters
        if (element_counter <=2) {nums[destination++] = nums[i];}

    }

    // printing the final output
    cout << format("nums = "); fpv(nums);
    cout << format("return-value = {}\n", destination);

    // return
    return(0);

}
```

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

```
int main(){

    // input- configuration
    vector<int> nums1 {1, 2, 3, 0, 0, 0};
    vector<int> nums2 {2, 5, 6};
    int m  {3};
    int n  {3};

    // setup
    int p1     {m-1};
    int p2     {n-1};
    int p3     {m+n-1};

    int curr1  {-1};
    int curr2  {-1};

    // going the other way
    while(p1 >= 0 || p2 >= 0)
    {
```

```
        // printing the values
        curr1 = p1 >= 0 ? nums1[p1] : std::numeric_limits<int>::min();
        curr2 = p2 >= 0 ? nums2[p2] : std::numeric_limits<int>::min();

        // assigning value
        if (curr1 > curr2) {nums1[p3] = curr1; --p3; --p1;}
        else               {nums1[p3] = curr2; --p3; --p2;}

    }

    // printing the final output
    cout << format("finaloutput = "); fPrintVector(nums1);

    // return
    return(0);
}
```

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

```
int main(){

    // input- configuration
    vector<int> prices {7,6,4,3,1};

    // setup
    StopWatch timer;                                    // timer-object
    int p0        {0};                                  // first index-pointer
    int p1        {1};                                  // second index-pointer
    int maxprofit {0};                                  // variable to hold max-profit
    int curr      {-1};                                 // variable to hold current-profit

```

```
    // going through array
    while(p1<prices.size()){
        curr       = prices[p1] - prices[p0];              // calculating current profit
        maxprofit  = curr > maxprofit ? curr : maxprofit;  // updating max-profit
        if (curr < 0) {p0 = p1;}                           // updating p0 if we find lower point
        ++p1;
    }

    // printing the final output
    cout << format("maxprofit = {}\n", maxprofit);
    timer.stop();

    // return
    return(0);

}
```

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 ith 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

```cpp
int main(){

    // input- configuration
    vector<int> prices {7,1,5,3,6,4};

    // setup
    int p1        {0};                                  // index-pointer to buying
    int p2        {0};                                  // index-pointer to selling
    int accprofit {0};                                  // variable to accumulate profit
    int currprofit {std::numeric_limits<int>::min()}; // variable to hold curr-profit

    // going through this
    while(p2 < prices.size()){

        currprofit = prices[p2] - prices[p1];          // calculating current profit

        if (currprofit > 0){
            accprofit  += currprofit;                  // accumulating the profit
            p1         = p2++;                         // moving the starting point
            continue;                                  // moving into the next iteration
        }
        else if (currprofit < 0){
            p1         = p2++;                         // moving the starting point
            continue;
        }
```

```
26
27          ++p2;                                       // 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 ith station is gas[i]. You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from the ith 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

```cpp
int main(){

    // input- configuration
    vector<int> gas   {1,2,3,4,5};
    vector<int> cost  {3,4,5,1,2};

    // setup
    auto acc   {0};                                 // variable to accumulate values
    vector<int> diffvec;                            // to store differences
    auto temp  {0};                                 // to maintain values in cache
    int finaloutput {-1};                           // the final output sotring var

    // running through it
    for(int i = 0; i<cost.size(); ++i){
        temp   = gas[i] - cost[i];                  // calculating flux
        acc    += temp;                             // appending to integral flux
        diffvec.push_back(temp);                    // storing instantial-flux to vector
    }
    if (acc<0) {finaloutput = -1; return 0;}        // if total flux is zero, the task cannot be completed

    // going through the diff-vec
    acc = 0;
    for(int i = 0; i<diffvec.size(); ++i){
        acc += diffvec[i];                          // accumulating flux
        if (acc<0) {acc = 0; finaloutput = i+1;}    // updating start-point (since we cannot go below zero)
    }


    // printing the acc
    cout << format("acc = {}\n", finaloutput);

    // return
    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 == ratings.length
2. $1 \leq n \leq 2 * 10^4$
3. $0 \leq$ ratings[i] $\leq 2 * 10^4$

Code

```
// main-file ========================================================================
int main(){

    // input- configuration
    vector<int> ratings {1,0,2};

    // setup
    auto candies      {std::vector<int>(ratings.size(),1)};
    auto finaloutput  {static_cast<int>(candies.size())};
    int leftrating, currrating, rightrating;

    // left-pass
    for(int i = 1; i<candies.size(); ++i){

        // fetching the rating
        leftrating = ratings[i-1];
        currrating = ratings[i];

        // fetching references to candy counts
        int& leftcount = candies[i-1];
        int& currcount = candies[i];

        // updating based on left
```

```
        if (currrating > leftrating){
            currcount = leftcount+1;
        }
    }

    // right pass
    for(int i = ratings.size()-2; i>=0; --i){

        // fetching ratings
        currrating = ratings[i];
        rightrating = ratings[i+1];

        // fetching references to candies
        int& currcandies  = candies[i];
        int& rightcandies = candies[i+1];

        // updating based on right
        if (currrating > rightrating){
            currcandies = std::max(currcandies,
                                   rightcandies + 1);
        }
    }

    // summing up candies
    finaloutput = std::accumulate(candies.begin(), candies.end(), 0);
    cout << format("finaloutput = {}\n", finaloutput);

    // return
    return(0);

}
```

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

Code

```
int main(){

    // input- configuration
    string s   {"a good  example"};

    // setup
    vector<string> listofwords;

    // creating a list of words
    int p1 {0};
    string acc;
    while(p1 < s.size()){

        // checking if the current character is a non-space
        if (s[p1] != ' ') {acc += s[p1];}
        else{
            // if acc is non-empty, flush
            if (acc.size() != 0)  {listofwords.push_back(acc); acc = "";}
            else                  {;}
        }

        // moving the index-pointer forward
        p1++;
```

```
    }

    // check if acc is unflushed
    if (acc.size() != 0)  {listofwords.push_back(acc); acc = "";}

    // building the finaloutput
    string finaloutput;
    for(int i = listofwords.size()-1; i>=0; --i){
        finaloutput += listofwords[i];
        if (i!=0) [[unlikely]] {finaloutput += " ";}
    }

    // printing the finaloutput
    cout << format("finaloutput = {}\n", finaloutput);


    // return
    return(0);

}
```

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

```
int main(){

    // input- configuration
    vector<int> nums  {2,2,1,1,1,2,2};

    // setup
    unordered_map<int, int> histogram;
    int max_element   {std::numeric_limits<int>::min()};
    int max_count     {std::numeric_limits<int>::min()};
    int updated_count {0};

    // going through the elements
    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

```
int main(){

    // input- configuration
    vector<int> nums  {-1,-100,3,99};
    int k  {2};

    // setup
    StopWatch timer;                                    // setting up the timer
    k = k %static_cast<int>(nums.size());               // to ensure that the value is within range

    int source        {0};
    int temp_source   {nums[source]};
    int temp          {0};
    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. $2 \leq$ nums.length $\leq 10^5$
2. -30 $\leq$ nums[i] $\leq$ 30
3. The input is generated such that answer[i] is guaranteed to fit in a 32-bit integer

Code

```
int main(){

    // input- configuration
    vector<int> nums  {1,2,3,4};

    // setup
    vector<int> nums_left(nums.size(), 1);
    vector<int> nums_right(nums.size(), 1);
    int acc_left          {1};
    int acc_right         {1};

    // runs
    for(int i = 0; i<nums.size(); ++i){

        // source-indces
        int source_left {i-1};
        int source_right {static_cast<int>(nums.size())-i};

        // printing values
        acc_left   *= source_left == -1        ? 1 : nums[source_left];
        acc_right  *= source_right == nums.size() ? 1 : nums[source_right];

        // writing to the two values
        nums_left[i]                 = acc_left;
        nums_right[nums.size()-i-1]  = acc_right;
    }

    // building the accumlated value
    vector<int> finaloutput(nums.size(),1);
    for(int i = 0; i< finaloutput.size(); ++i){
        finaloutput[i] = nums_left[i] * nums_right[i];
    }

```

```
    // printing
    cout << format("finaloutput = "); fPrintVector(finaloutput);

    // return
    return(0);

}
```

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

Code

```
int main(){

    // input- configuration
    vector<int> citations {3,0,6,1,5};

    // sorting the citations first
    std::sort(citations.begin(), citations.end(),
        [](const int& a, const int& b) {return a>b;});

    // running accumulations
    auto hvalue    {0};
    for(int i = 0; i<citations.size(); ++i){
        if (citations[i] >= (i+1))   {hvalue = i+1;}
    }

    // printing citations
    cout << format("hvalue = {}\n", hvalue);

    // return
    return(0);

}
```

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

```
int main(){

    // input- configuration
    string s   {"abc"};
    string t   {"ahbgdc"};

    // setup
    int i = 0;

    // going through the elements
    for(auto x: t) if (x == s[i]) ++i;

    // returning
    cout << format("final-output = {}\n", static_cast<bool>(i == s.size())) ;


    // return
    return(0);

}
```