

# Leetcode Solutions

SVR

October 19, 2025

<b>Contents</b>		<b>14. Longest Common Prefix</b>	<b>50</b>
<b>1. Two Sum</b>	<b>18</b>	<b>17. Letter Combinations of a Phone Number</b>	<b>53</b>
<b>2. Add Two Numbers</b>	<b>21</b>	<b>19. Remove Nth Node From End of List</b>	<b>56</b>
<b>3. Longest Substring Without Repeating Characters</b>	<b>24</b>	<b>20. Valid Parentheses</b>	<b>59</b>
<b>4. Median Of Two Sorted Array</b>	<b>27</b>	<b>21. Merge Two Sorted Lists</b>	<b>62</b>
<b>5. Longest Palindromic Substring</b>	<b>30</b>	<b>22. Generate Parentheses</b>	<b>66</b>
<b>6. Zigzag Conversion</b>	<b>33</b>	<b>23. Merge k Sorted Lists</b>	<b>69</b>
<b>7. Reverse Integer</b>	<b>36</b>	<b>24. Swap Nodes in Pairs</b>	<b>73</b>
<b>9. Palindrome Number</b>	<b>38</b>	<b>25. Reverse Nodes in k-Group</b>	<b>76</b>
<b>11. Container with most water</b>	<b>41</b>	<b>26. Remove Duplicates From Sorted Array</b>	<b>79</b>
<b>12. Integer to Roman</b>	<b>43</b>	<b>27. Remove Element</b>	<b>82</b>
<b>13. Roman To Integer</b>	<b>47</b>	<b>28. Find the Index of the First Occurrence in a String</b>	<b>85</b>

33. Search in Rotated Sorted Array	88	49. Group Anagrams	127
34. Find First and Last Position of Element in Sorted Array	91	50. Pow(x, n)	131
35. Search Insert Position	94	53. Maximum Subarray	134
36. Valid Sudoku	97	54. Spiral Matrix	137
39. Combination Sum	101	55. Jump Game	140
40. Combination Sum II	105	56. Merge Intervals	143
41. First Missing Positive	109	57. Insert Interval	146
42. Trapping Rain Water	111	58. Length of Last Word	149
45. Jump Game II	114	59. Spiral Matrix II	152
46. Permutations	117	60. Permutation Sequence	155
47. Permutations II	120	61. Rotate List	159
48. Rotate Image	124	62. Unique Paths	162

63. Unique Paths II	165	77. Combinations	204
64. Minimum Path Sum	168	78. Subsets	207
66. Plus One	171	80. Remove Duplicates from Sorted Array II	210
67. Add Binary	174	Remove Duplicates from Sorted List II	212
68. Text Justification	177	86. Partition List	215
69. Sqrt(x)	184	88. Merge Sorted Array	218
70. Climbing Stairs	186	92. Reverse Linked List II	221
71. Simplify Path	188	94. Binary Tree Inorder Traversal	224
73. Set Matrix Zeroes	192	97. Interleaving String	226
74. Search a 2D Matrix	195	98. Validate Binary Search Tree	229
75. Sort Colors	198	99. Recover Binary Search Tree	232
76. Minimum Window Substring	200	100. Same Tree	235

101. Symmetric Tree	238	114. Flatten Binary Tree to Linked List	272
102. Binary Tree Level Order Traversal	241	116. Populating Next Right Pointers in Each Node	275
103. Binary Tree Zigzag Level Order Traversal	243	117. Populating Next Right Pointers in Each Node II	279
104. Maximum Depth of Binary Tree	246	118. Pascal's Triangle	282
105. Construct Binary Tree from Preorder and Inorder Traversal	249	119. Pascal's Triangle II	285
107. Binary Tree Level Order Traversal II	252	120. Triangle	288
108. Convert Sorted Array to Binary Search Tree	255	121. Best Time To Buy And Sell Stock	291
109. Convert Sorted List to Binary Search Tree	257	122. Best Time To Buy And Sell Stock II	293
110. Balanced Binary Tree	260	124. Binary Tree Maximum Path Sum	296
111. Minimum Depth of Binary Tree	263	125. Valid Palindrome	299
112. Path Sum	266	127. Word Ladder	302
113. Path Sum II	269	128. Longest Consecutive Sequence	307

129. Sum Root to Leaf Numbers	310	153. Find Minimum in Rotated Sorted Array	340
130. Surrounded Regions	313	162. Find Peak Element	343
134. Gas Station	317	167. Two Sum II - Input Array Is Sorted	346
135. Candy	320	169 Majority Element	349
136. Single Number	323	172. Factorial Trailing Zeroes	351
137. Single Number II	325	189 Rotate Array	354
141. Linked List Cycle	327	190. Reverse Bits	357
144. Binary Tree Preorder Traversal	330	191. Number of 1 Bits	360
145. Binary Tree Postorder Traversal	332	198. House Robber	362
148. Sort List	334	199. Binary Tree Right Side View	365
150. Evaluate Reverse Polish Notation	336	200. Number of Islands	367
151. Reverse Words In A String	337	202. Happy Number	371

205. Isomorphic Strings	374	228. Summary Ranges	416
206. Reverse Linked List	378	230. Kth Smallest Element in a BST	419
207. Course Schedule	381	235. Lowest Common Ancestor of a Binary Search Tree	422
209. Minimum Size Subarray Sum	387	236. Lowest Common Ancestor of a Binary Tree	425
212. Word Search II	390	238. Product of Array Except Self	428
213. House Robber II	395	239. Sliding Window Maximum	431
215. Kth Largest Element in an Array	398	242. Valid Anagram	435
216. Combination Sum III	400	268. Missing Number	437
217. Contains Duplicate	405	274. H-Index	440
219. Contains Duplicate II	407	279. Perfect Squares	442
222. Count Complete Tree Nodes	410	283. Move Zeros	445
226. Invert Binary Tree	413	289. Game of Life	447

290. Word Pattern	452	417. Pacific Atlantic Water Flow	487
300. Longest Increasing Subsequence	456	424. Longest Repeating Character Replacement	493
322. Coin Change	458	429. N-ary Tree Level Order Traversal	496
345. Reverse Vowels Of A String	461	433. Minimum Genetic Mutation	499
347. Top K Frequent Elements	463	443. String Compression	504
349. Intersection of Two Arrays	466	448. Find All Numbers Disappeared in an Array	508
350. Intersection of Two Arrays II	468	452. Minimum Number of Arrows to Burst Balloons	510
365. Water and Jug Problem	471	463. Island Perimeter	514
383. Ransom Note	474	485. Max Consecutive Ones	520
392. Is Subsequence	477	530. Minimum Absolute Difference in BST	522
394. Decode String	479	559. Maximum Depth of N-ary Tree	524
414. Third Maximum Number	483	560. Subarray Sum Equals K	527



572. Subtree of Another Tree	529	883. Projection Area of 3D Shapes	570
593. Valid Square	532	892. Surface Area of 3D Shapes	573
605. Can Place Flowers	536	918. Maximum Sum Circular Subarray	577
637. Average of Levels in Binary Tree	538	965. Univalued Binary Tree	580
662. Maximum Width of Binary Tree	543	973. K Closest Points to Origin	583
695. Max Area of Island	547	988. Smallest String Starting From Leaf	587
724. Find Pivot Index	551	1004. Max Consecutives Ones III	591
739. Daily Temperatures	557	1037. Valid Boomerang	594
783. Minimum Distance Between BST Nodes	560	1091. Shortest Path in Binary Matrix	597
812. Largest Triangle Area	562	1143. Longest Common Subsequence	601
836. Rectangle Overlap	565	1207. Unique Number Of Occurences	604
867. Transpose Matrix	568	1232. Check If It Is a Straight Line	606

1266. Minimum Time Visiting All Points	609	2390. Removing Stars From A String	650
1379. Find a Corresponding Node of a Binary Tree in a Clone of That Tree	612	2481. Minimum Cuts to Divide a Circle	653
1431. Kids With Greatest Number Of Candies	615		
1493. Longest Subarray Of 1s After Deleting One Element	618		
1657. Determine If Two Strings Are Close	621		
1679. Max Number Of K-Sum Pairs	625		
1768. Merge Strings Alternately	628		
1971. Find if Path Exists in Graph	631		
2101. Detonate the Maximum Bombs	635		
2215. Find The Difference Of Two Arrays	640		
2249. Count Lattice Points Inside a Circle	643		
2352. Equal Row And Column Pairs	647		

## 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 // including header-files
2 #include <algorithm>
3 #include <unordered_set>
4 #include <bitset>
5 #include <climits>
6 #include <cstdint>
7 #include <iostream>
8 #include <limits>
9 #include <map>
10 #include <new>
11 #include <stdlib.h>
12 #include <unordered_map>
13 #include <vector>
14 #include <set>
15 #include <numeric>
16 #include <functional>
17 #include <deque>
18
19
20 // hash-deinfes
```

```

21 #define PRINTSPACE std::cout << "\n\n\n" << std::endl;
22 #define PRINTLINE std::cout << "===== " << std::endl;
23
24 // borrowing from namespace std
25 using std::cout;
26 using std::endl;
27 using std::vector;
28 using std::string;
29 using std::unordered_map;
30 using std::map;
31 using std::format;
32 using std::deque;
33 using std::pair;
34 using std::min;
35 using std::max;
36
37 // vector printing function
38 template<typename T>
39 void fPrintVector(vector<T> input){
40     for(auto x: input) cout << x << ",";
41     cout << endl;
42 }
43
44 template<typename T>
45 void fpv(vector<T> input){
46     for(auto x: input) cout << x << ",";
47     cout << endl;
48 }
49
50 template<typename T>
51 void fPrintMatrix(vector<T> input){
52     for(auto x: input){
53         for(auto y: x){
54             cout << y << ",";
55         }

```

```

56         cout << endl;
57     }
58 }
59
60 template<typename T, typename T1>
61 void fPrintHashMap(unordered_map<T, T1> input){
62     for(auto x: input){
63         cout << format("{}{} | ", x.first, x.second);
64     }
65     cout <<endl;
66 }
67
68 struct TreeNode {
69     int val;
70     TreeNode *left;
71     TreeNode *right;
72     TreeNode() : val(0), left(nullptr), right(nullptr) {}
73     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
74     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
75 };
76
77
78 struct ListNode {
79     int val;
80     ListNode *next;
81     ListNode() : val(0), next(nullptr) {}
82     ListNode(int x) : val(x), next(nullptr) {}
83     ListNode(int x, ListNode *next) : val(x), next(next) {}
84 };
85
86 void fPrintBinaryTree(TreeNode* root){
87     // sending it back
88     if (root == nullptr) return;
89
90     // printing

```

```

91     PRINTLINE
92     cout << "root->val = " << root->val << endl;
93
94     // calling the children
95     fPrintBinaryTree(root->left);
96     fPrintBinaryTree(root->right);
97
98     // returning
99     return;
100
101 }
102
103 void fPrintLinkedList(string prefix,
104                      ListNode* root){
105     if (root == nullptr) return;
106     cout << prefix;
107     std::function<void(ListNode*)> runlinkedlist = [&runlinkedlist](ListNode* root){
108         if (root == nullptr) return;
109         cout << root->val << " -> ";
110         runlinkedlist(root->next);
111     };
112     runlinkedlist(root);
113     cout << "|" << endl;
114     return;
115 }
116
117 template<typename T>
118 void fPrintContainer(T input){
119     for(auto x: input) cout << x << ", ";
120     cout << endl;
121     return;
122 }
123
124 struct Timer
125 {

```

```

126 std::chrono::time_point<std::chrono::high_resolution_clock> startpoint;
127 std::chrono::time_point<std::chrono::high_resolution_clock> endpoint;
128 std::chrono::duration<long long, std::nano> duration;
129
130 // constructor
131 Timer() {startpoint = std::chrono::high_resolution_clock::now();}
132 void start() {startpoint = std::chrono::high_resolution_clock::now();}
133 void stop() {endpoint = std::chrono::high_resolution_clock::now(); fetchtime();}
134
135 void fetchtime(){
136     duration = std::chrono::duration_cast<std::chrono::nanoseconds>(endpoint - startpoint);
137     cout << format("{} nanoseconds \n", duration.count());
138 }
139 void fetchtime(string stringarg){
140     duration = std::chrono::duration_cast<std::chrono::nanoseconds>(endpoint - startpoint);
141     cout << format("{} took {} nanoseconds \n", stringarg, duration.count());
142 }
143 void measure(){
144     auto temp = std::chrono::high_resolution_clock::now();
145     auto nsduration = std::chrono::duration_cast<std::chrono::nanoseconds>(temp - startpoint);
146     auto msduration = std::chrono::duration_cast<std::chrono::microseconds>(temp - startpoint);
147     auto sduration = std::chrono::duration_cast<std::chrono::seconds>(temp - startpoint);
148     cout << format("{} nanoseconds | {} microseconds | {} seconds \n",
149         nsduration.count(), msduration.count(), sduration.count());
150 }
151 ~Timer(){
152     measure();
153 }
154 };
155
156 // main-file =====
157 int main(){
158
159     // starting timer
160     Timer timer;

```



```
161
162     // input- configuration
163
164
165     // setup
166
167
168
169
170
171     // return
172     return(0);
173
174 }
```

---

## 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.

#### Examples

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.



## Constraints

- $0 \leq \text{s.length} \leq 5 * 10^4$
- s consists of English letters, digits, symbols and spaces.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto s      {string("abcabcbb")};
8
9     // trivial cases
10    if (s.size() <= 1) {cout << format("final-output = {}\n", s.size()); return 0;}
11
12    // setup
13    unordered_map<char, int> histogram;
14    char curr;
15    auto p1      {0};
16    auto finaloutput {-1};
17    auto temp_length {-1};
18
19    // going through the thing
20    for(int p2 = 0; p2<s.size(); ++p2){
21
22        // moving to another variable
23        curr = s[p2];
24
25        // checking if current character is in histogram
```

```

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

```

---

## 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 }
```

---

## 5. Longest Palindromic Substring

Given a string  $s$ , return the longest palindromic substring in  $s$ .

### Examples

#### 1. Example 1:

- Input:  $s = \text{"babad"}$
- Output:  $\text{"bab"}$
- Explanation:  $\text{"aba"}$  is also a valid answer.

#### 2. Example 2:

- Input:  $s = \text{"cbabd"}$
- Output:  $\text{"bb"}$

### Constraints

- $1 \leq s.length \leq 1000$
- $s$  consist of only digits and English letters.

### Code

---

```

1  int main(){
2
3      // starting timer
4      Timer timer;
5
6      // input- configuration
7      auto s {string("babad")};
8
9
10     // setup
11     int maxlength {-1};
12     int boundary {1};
13     string largestpalindrome = "";
14
15     // going through the elements
16     for(int i = 0; i<s.size(); ++i){
17
18         // checking odd-palindromes from here
19         auto oddlength {1};
20         auto boundary {1};
21         string oddstring {s[i]};
22
23         while(i - boundary >= 0 && i + boundary < s.size() && s[i-boundary] == s[i+boundary]){
24             oddlength += 2;           // updating length
25             oddstring = s.substr(i-boundary, 2*boundary+1); // subsetting
26             ++boundary;              // updating boundary
27         }
28
29         // checking even-palindromes from here
30         auto evenlength {0};
31         boundary = 1;
32         auto evenstring {string("")};
33
34         while(i+1-boundary >= 0 && i+boundary < s.size() && s[i+1-boundary] == s[i+boundary]){

```

```

35     evenlength += 2;                                // updating length
36     evenstring = s.substr(i+1-boundary, 2*boundary); // subsetting
37     ++boundary;                                     // updating boundary
38 }
39
40 // updating largest-string
41 largestpalindrome = oddlength > largestpalindrome.size() ? oddstring : largestpalindrome;
42 largestpalindrome = evenlength > largestpalindrome.size() ? evenstring : largestpalindrome;
43
44 }
45
46 // returning
47 cout << format("final-output = {}\n", largestpalindrome);
48
49 // return
50 return(0);
51
52 }

```

---



## 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 }
```

---

## 7. Reverse Integer

Given a signed 32-bit integer  $x$ , return  $x$  with its digits reversed. If reversing  $x$  causes the value to go outside the signed 32-bit integer range  $[-2^{31}, 2^{31} - 1]$ , then return 0.

Assume the environment does not allow you to store 64-bit integers (signed or unsigned).

### Examples

1. **Example 1:**

- Input:  $x = 123$
- Output: 321

2. **Example 2:**

- Input:  $x = -123$
- Output: -321

3. **Example 3:**

- Input:  $x = 120$
- Output: 21

### Constraints

- $-2^{31} \leq x \leq 2^{31} - 1$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto x {123};
8
9     // setup
10    auto finaloutput    {static_cast<long long>(0)};
11    auto leftboundary    {- std::pow(2, 31)};
12    auto rightboundary   {-1 + std::pow(2, 31)};
13
14    // building final output
15    while(x){
16
17        finaloutput = finaloutput*10 + x%10;
18        x           = x/10;
19        if (finaloutput < leftboundary || finaloutput > rightboundary) {return 0;} // checking the cases
20    }
21
22    // returning the final output
23    cout << format("final-output = {}\n", finaloutput);
24
25    // return
26    return(0);
27
28 }
```

---

## 9. Palindrome Number

Given an integer  $x$ , return true if  $x$  is a palindrome, and false otherwise.

### Examples

#### 1. Example 1:

- Input:  $x = 121$
- Output: true
- Explanation: 121 reads as 121 from left to right and from right to left.

#### 2. Example 2:

- Input:  $x = -121$
- Output: false
- Explanation: From left to right, it reads -121. From right to left, it becomes 121-. Therefore it is not a palindrome.

#### 3. Example 3:

- Input:  $x = 10$
- Output: false
- Explanation: Reads 01 from right to left. Therefore it is not a palindrome.

### Constraints

- $-2^{31} \leq x \leq -1$

## Code

---

```
1  int main(){
2
3      // starting timer
4      Timer timer;
5
6      // input- configuration
7      auto x {121};
8
9      // setup
10     auto finaloutput {false};
11
12     // retarded question
13     if (x<0) {cout << format("final-output = {}\n", finaloutput); return 0;}
14
15     // running
16     auto x_clone {x};
17     auto x_reverse {static_cast<long int>(0)};
18
19     // running through the value
20     while(x_clone){
21         x_reverse = x_reverse*10 + x_clone%10;
22         x_clone = x_clone/10;
23     }
24
25     // comparing the two
26     if (x == x_reverse) {finaloutput = true;}
27     else {finaloutput = false;}
28
29
30     // printing
31     cout << format("final-output = {}\n", finaloutput);
32
33 }
```

```
34     // return
35     return(0);
36
37 }
```

---



## 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 >= strs[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 >= strs[i].size())   {runcondition = false; break;}
26
27            // checking if same
28            if (strs[i][p] != strs[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 }
```

---

## 17. Letter Combinations of a Phone Number

Given a string containing digits from 2-9 inclusive, return all possible letter combinations that the number could represent. Return the answer in any order.

A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

### Examples

#### 1. Example 1:

- Input: digits = "23"
- Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]

#### 2. Example 2:

- Input: digits = ""
- Output: []

#### 3. Example 3:

- Input: digits = "2"
- Output: ["a", "b", "c"]

### Constraints

- $0 \leq \text{digits.length} \leq 4$
- `digits[i]` is a digit in the range ['2', '9'].

## Code

---

```
1 void foo(int          horizontalindex,
2         int          verticalindex,
3         string       digits,
4         string       runningstring,
5         vector<string>& finalOutput,
6         unordered_map<char, vector<string>>& charToLetter)
7 {
8
9     // adding current to running sum
10    runningstring += charToLetter[digits[verticalindex]][horizontalindex];
11
12    // sending it back
13    if (verticalindex == digits.size()-1) {finalOutput.push_back(runningstring); return;}
14
15    // running recursion on different sub-paths
16    for(int j = 0; j < charToLetter[digits[verticalindex+1]].size(); ++j)
17        foo(j, verticalindex+1,digits, runningstring, finalOutput, charToLetter);
18
19    // returning
20    return;
21 }
22
23 int main(){
24
25     // starting timer
26     Timer timer;
27
28     // input- configuration
29     auto digits {string("23")};
30
31     // trivial case
32     if (digits.size()==0) {
33         cout << format("final-output = {}\n", vector<string>({}));
```

```

34     return 0;
35 }
36
37 // setup
38 unordered_map<char, vector<string>> charToLetter;
39 charToLetter['2'] = vector<string>({"a", "b", "c"});
40 charToLetter['3'] = vector<string>({"d", "e", "f"});
41 charToLetter['4'] = vector<string>({"g", "h", "i"});
42 charToLetter['5'] = vector<string>({"j", "k", "l"});
43 charToLetter['6'] = vector<string>({"m", "n", "o"});
44 charToLetter['7'] = vector<string>({"p", "q", "r", "s"});
45 charToLetter['8'] = vector<string>({"t", "u", "v"});
46 charToLetter['9'] = vector<string>({"w", "x", "y", "z"});
47
48 // going through each character on top level
49 vector<string> finalOutput;
50 for(int i = 0; i<charToLetter[digits[0]].size(); ++i)
51     foo(i, 0, digits, "", finalOutput, charToLetter);
52
53 // returning
54 cout << format("final-output = {}\n", finalOutput);
55
56 // return
57 return(0);
58
59 }

```

---

## 19. Remove Nth Node From End of List

Given the head of a linked list, remove the nth node from the end of the list and return its head.

### Examples

1. **Example 1:**

- Input: head = [1,2,3,4,5], n = 2
- Output: [1,2,3,5]

2. **Example 2:**

- Input: head = [1], n = 1
- Output: []

3. **Example 3:**

- Input: head = [1,2], n = 1
- Output: [1]

### Constraints

- The number of nodes in the list is sz.
- $1 \leq sz \leq 30$
- $0 \leq \text{Node.val} \leq 100$
- $1 \leq n \leq sz$



## Code

---

```
1  int main(){
2
3      // starting timer
4      Timer timer;
5
6      // input- configuration
7      ListNode* head          = new ListNode(1);
8      head->next               = new ListNode(2);
9      head->next->next          = new ListNode(3);
10     head->next->next->next      = new ListNode(4);
11     head->next->next->next->next = new ListNode(5);
12     auto n {2};
13
14     // trivial case
15     if (!head) {cout << format("final-output = "); fPrintLinkedList(head); cout << endl;}
16
17     // setup
18     auto nodecounter {0};
19     auto prehead      {new ListNode};
20
21     prehead->next      = head;
22     ListNode* traveller = head;
23     ListNode* delayedTraveller = prehead;
24
25     // going through the list
26     while(traveller){
27         if (++nodecounter > n) {delayedTraveller = delayedTraveller->next;}
28         traveller = traveller->next;
29     }
30
31     // reconnecting
32     delayedTraveller->next = delayedTraveller->next->next;
33 }
```

```
34 // sending back
35 cout << format("final-output = ");
36 fPrintLinkedList(prehead->next); cout << endl;
37
38
39
40
41
42 // return
43 return(0);
44
45 }
```

---

## 20. Valid Parentheses

Given a string `s` containing just the characters '(', ')', '[', ']', '{' and '}', determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

### Examples

#### 1. Example 1:

- Input: `s = "()"`
- Output: `true`

#### 2. Example 2:

- Input: `s = "()[]"`
- Output: `true`

#### 3. Example 3:

- Input: `s = "(]"`
- Output: `false`

## Constraints

- $1 \leq s.length \leq 10^4$
- s consists of parentheses only '()[]'.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     string s  {"() [] {}"};
8
9     // seutp
10    deque<char> var00;
11
12    // going through the string
13    for(auto x: s){
14        // pushing or poppping
15        if (x == '(' || x == '[' || x == '{') {var00.push_back(x);}
16        else{
17            if (var00.size()==0) {cout << format("final-output = false\n"); return 0;}
18            if (x == ')') && var00.back() == '(') {var00.pop_back(); continue;}
19            if (x == ']') && var00.back() == '[') {var00.pop_back(); continue;}
20            if (x == '}') && var00.back() == '{') {var00.pop_back(); continue;}
21            cout << format("final-output = false\n"); return 0;
22        }
23    }
24
25    // checking if anything is left
```

```
26     if (var00.size()!=0) {cout << format("final-output = false\n"); return 0;}
27
28
29     // true-case
30     cout << format("final-output = true\n"); return 0;
31
32     // return
33     return(0);
34
35 }
```

---

## 21. Merge Two Sorted Lists

You are given the heads of two sorted linked lists list1 and list2.

Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists.

Return the head of the merged linked list.

### Examples

#### 1. Example 1:

- Input: list1 = [1,2,4], list2 = [1,3,4]
- Output: [1,1,2,3,4,4]

#### 2. Example 2:

- Input: list1 = [], list2 = []
- Output: []

#### 3. Example 3:

- Input: list1 = [], list2 = [0]
- Output: [0]

### Constraints

- The number of nodes in both lists is in the range [0, 50].

- $-100 \leq \text{Node.val} \leq 100$
- Both list1 and list2 are sorted in non-decreasing order.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     ListNode* list1 = new ListNode(1);
8     list1->next      = new ListNode(2);
9     list1->next->next = new ListNode(4);
10
11     ListNode* list2 = new ListNode(1);
12     list2->next      = new ListNode(3);
13     list2->next->next = new ListNode(4);
14
15     // setup
16     ListNode* finalOutput = new ListNode;
17     ListNode* traveller_final = finalOutput;
18     ListNode* traveller_1 = list1;
19     ListNode* traveller_2 = list2;
20
21     // going through the two lists
22     while (traveller_1 != nullptr && traveller_2 != nullptr) {
23
24         // comparing values
25         if (traveller_1->val < traveller_2->val) {
26             traveller_final->next = traveller_1;           // linking node to final
27             traveller_final        = traveller_final->next; // moving node to new link
```

```

28
29     if (traveller_1->next != nullptr)
30         traveller_1          = traveller_1->next;      // updating traveller 1
31     else{
32         traveller_1 = nullptr;
33         break;
34     }
35
36 }
37 else {
38     traveller_final->next = traveller_2;                // linking node to final
39     traveller_final      = traveller_final->next;      // moving node to new link
40     if (traveller_2->next != nullptr)
41         traveller_2      = traveller_2->next;          // updating travel 2
42     else{
43         traveller_2 = nullptr;
44         break;
45     }
46 }
47 }
48
49 // checking if anything is left
50 while (traveller_1 != nullptr){
51     traveller_final->next = traveller_1;
52     traveller_final      = traveller_final->next;
53     traveller_1          = traveller_1->next;
54 }
55
56 while (traveller_2 != nullptr){
57     traveller_final->next = traveller_2;
58     traveller_final      = traveller_final->next;
59     traveller_2          = traveller_2->next;
60 }
61
62 // printing the final-output

```



```
63     cout << format("finaloutput = ");
64     fPrintLinkedList(finalOutput); cout << endl;
65
66     // return
67     return(0);
68 }
```

---

## 22. Generate Parentheses

Given  $n$  pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

### Examples

#### 1. Example 1:

- Input:  $n = 3$
- Output: ["((()))", "(()())", "(())()", "()(())", "()()()"]

#### 2. Example 2:

- Input:  $n = 1$
- Output: ["()"]

### Constraints

- $1 \leq n \leq 8$

### Code

---

```
1 void foo(int          numopens,
2     int          numcloses,
3     const int&    n,
4     string        runningstring,
5     vector<string>& finalOutput){
```

```

6
7 // sending it back
8 if (numopens > n || numcloses > n) return;
9 else if (numopens == n && numcloses == n) {finalOutput.push_back(runningstring); return;}
10
11 // opening route
12 foo(numopens+1, numcloses, n, runningstring + "(", finalOutput);
13
14 // closing route
15 if (numopens>numcloses)
16     foo(numopens, numcloses+1, n, runningstring+")", finalOutput);
17
18 // returning
19 return;
20
21 }
22
23 int main(){
24
25     // starting timer
26     Timer timer;
27
28     // input- configuration
29     auto n {3};
30
31     // trivial case
32     if (n == 0) {cout << format("finalOutput = []\n"); return 0;}
33
34     // calling the function
35     vector<string> finalOutput;
36     foo(0, 0, n, "", finalOutput);
37
38     // returning
39     cout << format("finalOutput = {}\n", finalOutput);
40

```

```
41     // return
42     return(0);
43
44 }
```

---

## 23. Merge k Sorted Lists

You are given an array of k linked-lists lists, each linked-list is sorted in ascending order.

Merge all the linked-lists into one sorted linked-list and return it.

### Examples

#### 1. Example 1:

- Input: lists = [[1,4,5],[1,3,4],[2,6]]
- Output: [1,1,2,3,4,4,5,6]
- Explanation: The linked-lists are:  
1→4→5,  
1→3→4,  
2→6.
  - merging them into one sorted linked list: 1→1→2→3→4→4→5→6

#### 2. Example 2:

- Input: lists = []
- Output: []

#### 3. Example 3:

- Input: lists = [[]]
- Output: []

## Constraints

- $k == \text{lists.length}$
- $0 \leq k \leq 104$
- $0 \leq \text{lists}[i].\text{length} \leq 500$
- $-10^4 \leq \text{lists}[i][j] \leq 10^4$
- $\text{lists}[i]$  is sorted in ascending order.
- The sum of  $\text{lists}[i].\text{length}$  will not exceed  $10^4$ .

## Code

---

```
1 // main-file =====
2 int main(){
3
4     // starting timer
5     Timer timer;
6
7     // input- configuration
8     auto head0 {new ListNode(1)};
9     head0->next = new ListNode(4);
10    head0->next->next = new ListNode(5);
11
12    auto head1 {new ListNode(1)};
13    head1->next = new ListNode(3);
14    head1->next->next = new ListNode(4);
15
16    auto head2 {new ListNode(2)};
17    head2->next = new ListNode(6);
```

```

18
19 vector<ListNode*> lists {head0, head1, head2};
20
21
22 // setup
23 ListNode* prehead = new ListNode();
24 ListNode* traveller {prehead};
25 bool runcondition {true};
26 vector<ListNode*> travellerList;
27 int smallestindex {-1};
28 int smallestvalue {std::numeric_limits<int>::max()};
29
30 // filling up the traveller lists
31 for(int i = 0; i<lists.size(); ++i) {travellerList.push_back(lists[i]);}
32
33 // running the loop
34 int counter {0};
35 while(runcondition){
36
37     // going through the values
38     for(int i = 0; i < travellerList.size(); ++i){
39         if (travellerList[i] != nullptr &&
40             travellerList[i]->val < smallestvalue) {
41             smallestvalue = travellerList[i]->val;
42             smallestindex = i;
43         }
44     }
45
46     // now that we have the smallest value and smallest index, we add to the prehead
47     if (smallestindex == -1) {break;}
48     traveller->next = travellerList[smallestindex];
49     traveller = traveller->next;
50     travellerList[smallestindex] = travellerList[smallestindex]->next;
51
52     // resetting

```

```
53     smallestvalue = std::numeric_limits<int>::max();
54     smallestindex = -1;
55 }
56
57 // printing the final-linked list
58 fPrintLinkedList("final-output = ", prehead->next);
59
60 // return
61 return(0);
62
63 }
```

---



## 24. Swap Nodes in Pairs

Given a linked list, swap every two adjacent nodes and return its head. You must solve the problem without modifying the values in the list's nodes (i.e., only nodes themselves may be changed.)

### Examples

1. **Example 1:**

- Input: head = [1,2,3,4]
- Output: [2,1,4,3]

2. **Example 2:**

- Input: head = []
- Output: []

3. **Example 3:**

- Input: head = [1]
- Output: [1]

### Constraints

- The number of nodes in the list is in the range [0, 100].
- $0 \leq \text{Node.val} \leq 100$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto head          {new ListNode(1)};
8     head->next          = new ListNode(2);
9     head->next->next     = new ListNode(3);
10    head->next->next->next = new ListNode(4);
11
12    // setup
13    ListNode* prehead    = new ListNode;
14    prehead->next         = head;
15    ListNode* traveller  = prehead;
16
17    // going through it
18    while(traveller){
19
20        // jump-condition
21        if (traveller->next == nullptr || traveller->next->next == nullptr) break;
22
23        // swapping
24        if (traveller->next      != nullptr && traveller->next->next != nullptr){
25
26            ListNode* a    = traveller->next;
27            ListNode* b    = traveller->next->next;
28            ListNode* c    = traveller->next->next->next;
29
30            traveller->next = b;
31            b->next        = a;
32            a->next        = c;
33            traveller      = a;
```

```
34     }  
35 }  
36  
37 // printing the linked List  
38 fPrintLinkedList("final-output = ", prehead->next);  
39  
40 // return  
41 return(0);  
42  
43 }
```

---

## 25. Reverse Nodes in k-Group

Given the head of a linked list, reverse the nodes of the list k at a time, and return the modified list.

k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k then left-out nodes, in the end, should remain as it is.

You may not alter the values in the list's nodes, only nodes themselves may be changed.

### Examples

#### 1. Example 1:

- Input: head = [1,2,3,4,5], k = 2
- Output: [2,1,4,3,5]

#### 2. Example 2:

- Input: head = [1,2,3,4,5], k = 3
- Output: [3,2,1,4,5]

### Constraints

- The number of nodes in the list is n.
- $1 \leq k \leq n \leq 5000$
- $0 \leq \text{Node.val} \leq 1000$

## Code

---

```
1  int main(){
2
3      // starting timer
4      Timer timer;
5
6      // input- configuration
7      auto head          {new ListNode(1)};
8      head->next          = new ListNode(2);
9      head->next->next     = new ListNode(3);
10     head->next->next->next = new ListNode(4);
11     head->next->next->next->next = new ListNode(5);
12     auto k {2};
13
14     // setup
15     auto prehead        {new ListNode()};
16     prehead->next       = head;
17     auto traveller      {prehead};
18     vector<ListNode*> memberList(k, nullptr);
19     memberList.reserve(k);
20
21     // runnings
22     while(traveller){
23
24         // filling up the memberList
25         auto counter {0};
26         auto tempotraveller {traveller->next};
27
28         while(counter < k && tempotraveller != nullptr){
29             memberList[counter] = tempotraveller;
30             tempotraveller      = tempotraveller->next;
31             ++counter;
32         }
33     }
```

```

34 // checking breaking condition
35 if (counter!=k) break;
36
37 // reconnecting
38 auto beginningOfNextSegment {memberList[memberList.size()-1]->next};
39
40 // reconnecting
41 traveller->next = memberList[memberList.size()-1];
42 for(int i = memberList.size()-1; i>=1; --i) {memberList[i]->next = memberList[i-1];}
43 memberList[0]->next = beginningOfNextSegment;
44
45 // updating traveller
46 traveller = memberList[0];
47 }
48
49 // returning
50 fPrintLinkedList("final-output = ", prehead->next);
51
52 // return
53 return(0);
54
55 }

```

---

## 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 }
```

---

### 33. Search in Rotated Sorted Array

There is an integer array `nums` sorted in ascending order (with distinct values).

Prior to being passed to your function, `nums` is possibly left rotated at an unknown index  $k$  ( $1 \leq k < \text{nums.length}$ ) such that the resulting array is `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]` (0-indexed). For example, `[0,1,2,4,5,6,7]` might be left rotated by 3 indices and become `[4,5,6,7,0,1,2]`.

Given the array `nums` after the possible rotation and an integer `target`, return the index of `target` if it is in `nums`, or `-1` if it is not in `nums`.

You must write an algorithm with  $O(\log n)$  runtime complexity.

#### Examples

1. **Example 1:**

- Input: `nums = [4,5,6,7,0,1,2]`, `target = 0`
- Output: 4

2. **Example 2:**

- Input: `nums = [4,5,6,7,0,1,2]`, `target = 3`
- Output: -1

3. **Example 3:**

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



## Constraints

- $1 \leq \text{nums.length} \leq 5000$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- All values of nums are unique.
- nums is an ascending array that is possibly rotated.
- $-10^4 \leq \text{target} \leq 10^4$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums {vector<int>{4,5,6,7,0,1,2}};
8     auto target {0};
9
10    // setup
11    auto left    {0};
12    auto right   {static_cast<int>(nums.size())-1};
13    auto mid     {-1};
14    auto finaloutput {-1};
15
16
17    // going through the array
18    while(left <= right){
19        // fetching mid-value
```

```

20     mid = (left + right)/2;
21     if (nums[mid] == target) {finaloutput = mid; break;}
22
23     // checking if the left is sorted
24     if(nums[left] <= nums[mid]){
25         // checking if the target is out of bounds
26         if (target < nums[left] || target > nums[mid]) {left = mid +1;}
27         else {right = mid -1;}
28     }
29     else{
30         // checking if target is out of bounds
31         if (target < nums[mid] || target > nums[right]) {right = mid -1;}
32         else {left = mid + 1;}
33     }
34 }
35
36 // returning negative one in case none of these works
37 cout << format("final-output = {}\n", finaloutput);
38
39 // return
40 return(0);
41
42 }

```

---

## 34. Find First and Last Position of Element in Sorted Array

Given an array of integers `nums` sorted in non-decreasing order, find the starting and ending position of a given target value.

If target is not found in the array, return `[-1, -1]`.

You must write an algorithm with  $O(\log n)$  runtime complexity.

### Examples

#### 1. Example 1:

- Input: `nums = [5,7,7,8,8,10]`, `target = 8`
- Output: `[3,4]`

#### 2. Example 2:

- Input: `nums = [5,7,7,8,8,10]`, `target = 6`
- Output: `[-1,-1]`

#### 3. Example 3:

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

### Constraints

- $0 \leq \text{nums.length} \leq 10^5$

- $-10^9 \leq \text{nums}[i] \leq 10^9$
- `nums` is a non-decreasing array.
- $-10^9 \leq \text{target} \leq 10^9$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums {vector<int>{5,7,7,8,8,10}};
8     auto target {8};
9
10    // setup
11    auto left    {0};
12    auto right   {static_cast<int>(nums.size())-1};
13    auto mid     {-1};
14    auto leftedge {-1};
15    auto rightedge {-1};
16    auto finaloutput {vector<int>{-1, -1}};
17
18    // trivial case
19    if (nums.size()==0) {cout << format("final-output = {}\n", finaloutput); return 0;}
20
21    // finding left-edge
22    while(left <= right){
23        mid = (left+right)/2;
24        if (nums[mid] < target)    {left = mid +1;}
25        else if (target < nums[mid]) {right = mid-1;}
```

```

26         else                                {leftedge = mid; right = mid-1;}
27     }
28
29     // finding right-edge
30     left = 0; right = nums.size()-1;
31     while(left <= right){
32         mid = (left+right)/2;
33         if (nums[mid] < target)    {left = mid +1;}
34         else if (target < nums[mid]) {right = mid-1;}
35         else                      {rightedge = mid; left = mid+1;}
36     }
37
38     // building final output
39     finaloutput = vector<int>{leftedge, rightedge};
40
41     // printing the final output
42     cout << format("final-output = {}\n", finaloutput);
43
44     // return
45     return(0);
46
47 }

```

---

## 35. Search Insert Position

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with  $O(\log n)$  runtime complexity.

### Examples

1. **Example 1:**

- Input: `nums = [1,3,5,6]`, `target = 5`
- Output: 2

2. **Example 2:**

- Input: `nums = [1,3,5,6]`, `target = 2`
- Output: 1

3. **Example 3:**

- Input: `nums = [1,3,5,6]`, `target = 7`
- Output: 4

### Constraints

- $1 \leq \text{nums.length} \leq 10^4$

- $-10^4 \leq \text{nums}[i] \leq 10^4$
- nums contains distinct values sorted in ascending order.
- $-10^4 \leq \text{target} \leq 10^4$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums {vector<int>{1, 3, 5, 6}};
8     auto target {5};
9
10    // setup
11    auto left    {0};
12    auto right   {static_cast<int>(nums.size())-1};
13    auto mid     {-1};
14    auto finaloutput {-1};
15
16    // running the loop
17    while(left<=right){
18
19        // updating mid
20        mid = (left +right)/2;
21
22        // checking if mid is the value
23        if (nums[mid] == target) {finaloutput = mid; cout << format("final-output = {}\n", finaloutput); return 0;}
24        else if(nums[mid]<target) {left = mid+1;}
25        else                      {right = mid-1;}
```

```
26     }
27
28
29     // returning the midvalue
30     finaloutput = left;
31     cout << format("final-output = {}\n", finaloutput);
32
33     // return
34     return(0);
35
36 }
```

---



## 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","."],`  
• `["8",".",".","6",".",".","3"],`  
• `["4",".","8",".","3",".","1"],`  
• `["7",".",".","2",".",".","6"],`  
• `[["6",".",".","2","8","."],`  
• `[[".",".","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 }

```

---

## 39. Combination Sum

Given an array of distinct integers candidates and a target integer target, return a list of all unique combinations of candidates where the chosen numbers sum to target. You may return the combinations in any order.

The same number may be chosen from candidates an unlimited number of times. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

The test cases are generated such that the number of unique combinations that sum up to target is less than 150 combinations for the given input.

### Examples

#### 1. Example 1:

- Input: candidates = [2,3,6,7], target = 7
- Output: [[2,2,3],[7]]
- Explanation:
  - 2 and 3 are candidates, and  $2 + 2 + 3 = 7$ . Note that 2 can be used multiple times.
  - 7 is a candidate, and  $7 = 7$ .
  - These are the only two combinations.

#### 2. Example 2:

- Input: candidates = [2,3,5], target = 8
- Output: [[2,2,2,2],[2,3,3],[3,5]]

#### 3. Example 3:

- Input: candidates = [2], target = 1
- Output: []

## Constraints

- $1 \leq \text{candidates.length} \leq 30$
- $2 \leq \text{candidates}[i] \leq 40$
- All elements of candidates are distinct.
- $1 \leq \text{target} \leq 40$

## Code

---

```
1 void foo(const vector<int>& candidates,
2         vector<int> runningvector,
3         const int& target,
4         vector<vector<int>>& finalOutput)
5 {
6     // calculating running sum
7     int runningsum = std::accumulate(runningvector.begin(),
8                                     runningvector.end(),
9                                     0);
10
11     // sending it back
12     if (runningsum > target) {return;}
13     if (runningsum == target) {finalOutput.push_back(runningvector); return;}
14
15     // going through the different options
16     for(auto x: candidates){
```

```

17
18     // checking if this should be
19     if (runningvector.size() != 0 && x < runningvector[runningvector.size()-1]) {continue;}
20
21     auto temp {runningvector};
22     temp.push_back(x);
23
24     // recursive-call
25     foo(candidates, temp, target, finalOutput);
26 }
27
28 // returning
29 return;
30 }
31 int main(){
32
33     // starting timer
34     Timer timer;
35
36     // input- configuration
37     auto candidates {vector<int>{2,3,6,7}};
38     auto target      {7};
39
40     // setup
41     vector<vector<int>> finalOutput;
42     vector<int>         runningvector;
43
44     // calling the function
45     foo(candidates, runningvector, target, finalOutput);
46
47     // returning
48     cout << format("final-output = {}\n", finalOutput);
49
50     // return
51     return(0);

```

52

53

}



## 40. Combination Sum II

Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target.

Each number in candidates may only be used once in the combination.

Note: The solution set must not contain duplicate combinations.

### Examples

#### 1. Example 1:

- Input: candidates = [10,1,2,7,6,1,5], target = 8
- Output: [[1,1,6],[1,2,5],[1,7],[2,6]]

#### 2. Example 2:

- Input: candidates = [2,5,2,1,2], target = 5
- Output: [[1,2,2],[5]]

### Constraints

- $1 \leq \text{candidates.length} \leq 100$
- $1 \leq \text{candidates}[i] \leq 50$
- $1 \leq \text{target} \leq 30$

## Code

---

```
1      cout << format("{} nanoseconds | {} microseconds | {} milliseconds | {} seconds \n",
2          nsduration.count(), msduration.count(), milliduration.count(), sduration.count());
3  }
4  ~Timer(){
5      measure();
6  }
7  };
8
9  // main-file =====
10 void foo(const vector<int>&      candidates,
11         vector<int>           pathsofar,
12         vector<vector<int>>&    finaloutput,
13         const int              currindex,
14         const int&             target,
15         const vector<int>&      cumsum){
16
17     // adding to paths ofar
18     if (std::find(pathsofar.begin(),
19                 pathsofar.end(),
20                 currindex) == pathsofar.end()) {pathsofar.push_back(currindex);}
21     else {return;}
22
23     //
24     auto sumsofar = std::accumulate(pathsofar.begin(),
25                                     pathsofar.end(),
26                                     0,
27                                     [&](auto acc, auto argx){return acc + candidates[argx];});
28     auto complement {target - sumsofar};
29
30     // making decisions based on complement
31     if (complement == 0) {
32
33         vector<int> valuessofar(pathsofar.size());
```

```

34
35     std::transform(pathsofar.begin(),
36                    pathsofar.end(),
37                    valuessofar.begin(),
38                    [&](auto argx){return candidates[argx];});
39
40     // checking if top value is same
41     if (std::find(finaloutput.begin(),
42                  finaloutput.end(),
43                  valuessofar) == finaloutput.end()) {finaloutput.push_back(valuessofar);}
44
45     return;
46 }
47 else if(complement < 0) {return;}
48
49 // checking if it is possible to go from here
50 auto maxpotentialfromhere {cumsum[currindex] - candidates[currindex]};
51 if (complement > maxpotentialfromhere) { return;}
52
53 // going through the rest of the indices
54 for(int nextindex = currindex + 1; nextindex < candidates.size(); ++nextindex){
55     foo(candidates, pathsofar, finaloutput, nextindex, target, cumsum);
56 }
57
58 }
59
60 int main(){
61
62     // starting timer
63     Timer timer;
64
65     // input-configuration
66     auto candidates    {vector<int>{10,1,2,7,6,1,5}};
67     auto target        {8};
68

```

```

69 // setup
70 vector<vector<int>> finaloutput;
71 vector<int>          pathsofar;
72
73 // sorting number
74 std::sort(candidates.begin(), candidates.end(), std::greater<int>());
75
76 // creating cumulative sums
77 std::vector<int> cumsum(candidates.size(), 0);
78 std::partial_sum(candidates.rbegin(), candidates.rend(), cumsum.rbegin());
79
80 // calling the function
81 for(int i = 0; i < candidates.size(); ++i){
82     foo(candidates,
83         pathsofar,
84         finaloutput,
85         i,
86         target,
87         cumsum);
88 }
89
90 // printing finaloutput
91 cout << format("finaloutput = {}\n", finaloutput);
92
93 // return
94 return(0);
95
96 }

```

---

## 41. First Missing Positive

Given an unsorted integer array `nums`. Return the smallest positive integer that is not present in `nums`.

You must implement an algorithm that runs in  $O(n)$  time and uses  $O(1)$  auxiliary space.

### Examples

#### 1. Example 1:

- Input: `nums = [1,2,0]`
- Output: 3
- Explanation: The numbers in the range `[1,2]` are all in the array.

#### 2. Example 2:

- Input: `nums = [3,4,-1,1]`
- Output: 2
- Explanation: 1 is in the array but 2 is missing.

#### 3. Example 3:

- Input: `nums = [7,8,9,11,12]`
- Output: 1
- Explanation: The smallest positive integer 1 is missing.

## Constraints

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

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums {vector<int>{3,4,-1,1}};
8
9     // in the off-chance that minvalue is one, building ordered set of values
10    std::set<int> orderedvalues;
11    for(auto x: nums)        {if (x>0) orderedvalues.insert(x);}
12
13    // going through the ordered set
14    auto finaloutput {1};
15    auto flag        {0};
16    for(const auto& x: orderedvalues)
17        if (x - finaloutput++ != 0) {--finaloutput; break;}
18
19    // printing based on flag value
20    cout << format("final-output = {}\n", finaloutput);
21
22    // return
23    return(0);
24
25 }
```

---

## 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 }

```

---

## 46. Permutations

Given an array `nums` of distinct integers, return all the possible permutations. You can return the answer in any order.

### Examples

1. **Example 1:**

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

2. **Example 2:**

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

3. **Example 3:**

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

### Constraints

- $1 \leq \text{nums.length} \leq 6$
- $-10 \leq \text{nums}[i] \leq 10$
- All the integers of `nums` are unique.

## Code

---

```
1 // traversing function
2 void fRecursive(const std::vector<int>&      nums,      \
3               std::vector< std::vector<int> >& finalOutput, \
4               const std::vector<int>      path,      \
5               const std::vector<int>      numsleft){
6
7     // check if length is reached
8     if (path.size() == nums.size()){
9
10        // add to final output
11        if (std::find(finalOutput.begin(), finalOutput.end(), path) == finalOutput.end())
12            finalOutput.push_back(path);
13
14        // returning since the path is complete
15        return;
16    }
17
18    // choosing from numsleft
19    for(int i = 0; i<numsleft.size(); ++i){
20
21        // choosing a number,
22        auto num      {numsleft[i]};
23        auto pathtemp  {path};
24        pathtemp.push_back(num);
25
26        // making a copy of nums left, removing character
27        auto numsleft_temp  {numsleft};
28        numsleft_temp.erase(numsleft_temp.begin() + i);
29
30        // calling function
31        fRecursive(nums, finalOutput, pathtemp, numsleft_temp);
32    }
33
```

```

34     // returning results
35     return;
36 }
37
38 int main(){
39
40     // starting timer
41     Timer timer;
42
43     // input- configuration
44     auto nums {vector<int>{1,2,3}};
45
46     // setup
47     std::vector<std::vector<int>> finalOutput;
48     auto path {vector<int>{}};
49     auto numsleft {nums};
50
51     // calling the function
52     fRecursive(nums, finalOutput, path, numsleft);
53
54     // returning the output
55     cout << format("final-output = {}\n", finalOutput);
56
57     // return
58     return(0);
59 }

```

---

## 47. Permutations II

Given a collection of numbers, `nums`, that might contain duplicates, return all possible unique permutations in any order.

### Examples

#### 1. Example 1:

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

#### 2. Example 2:

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

### Constraints

- $1 \leq \text{nums.length} \leq 8$
- $-10 \leq \text{nums}[i] \leq 10$

### Code



---

```

1  auto foo(const vector<int>& inputvector,
2          const int&      elementtoadd){
3
4      // creating the different options that can stem from this
5      vector<vector<int>> finaloutput; // max size is n(n+1)
6
7      for(int i = 0; i<inputvector.size()+1; ++i){
8
9          auto insertindex {i};
10         auto temp        {vector<int>(inputvector.size()+1)};
11
12         for(int j = 0; j<insertindex; ++j)          {temp[j] = inputvector[j];}
13         temp[insertindex] = elementtoadd;
14         for(int j = insertindex; j<inputvector.size(); ++j)
15         for(int j = insertindex; j<inputvector.size(); ++j) {temp[j+1] = inputvector[j];}
16
17         // check if temp is already in the finaloutput
18         if (std::find(finaloutput.begin(),
19                     finaloutput.end(),
20                     temp) == finaloutput.end()) {finaloutput.push_back(std::move(temp));}
21     }
22
23     // returning
24     return finaloutput;
25
26
27 }
28
29 int main(){
30
31     // starting timer
32     Timer timer;
33
34     // input- configuration

```

```

35 auto nums {vector<int>{2,2,1,1}};
36
37 // setup
38 std::deque<vector<int>> pool;
39 pool.push_back(vector<int>{nums[0]});
40
41 // going through a loop
42 for(int i = 1; i<nums.size(); ++i){
43
44     auto drainpool {pool};
45     pool.clear();
46
47     while(drainpool.size() != 0){
48         auto frontvector = drainpool.front(); drainpool.pop_front(); // popping from the front
49         auto elementtoadd {nums[i]};
50         auto returntemp {foo(frontvector, elementtoadd)}; // calling the function
51         for(const auto& x: returntemp) {
52             if (std::find(pool.begin(),
53                 pool.end(),
54                 x) == pool.end()) {pool.push_back(x);}
55         } // pushing all to the back of the pool
56     }
57 }
58
59 // printing the pool in the end
60 cout << format("pool = {}\n", pool);
61
62
63
64
65
66
67
68
69

```

```
70     // return
71     return(0);
72
73 }
```

---

## 48. Rotate Image

You are given an  $n \times n$  2D matrix representing an image, rotate the image by 90 degrees (clockwise).

You have to rotate the image in-place, which means you have to modify the input 2D matrix directly. DO NOT allocate another 2D matrix and do the rotation.

### Examples

#### 1. Example 1:

- Input: matrix = `[[1,2,3],[4,5,6],[7,8,9]]`
- Output: `[[7,4,1],[8,5,2],[9,6,3]]`

#### 2. Example 2:

- Input: matrix = `[[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]`
- Output: `[[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]`

### Constraints

- $n == \text{matrix.length} == \text{matrix}[i].\text{length}$
- $1 \leq n \leq 20$
- $-1000 \leq \text{matrix}[i][j] \leq 1000$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     vector<vector<int>> matrix {
8         {1,2,3},
9         {4,5,6},
10        {7,8,9}
11    };
12
13    // setup
14    auto t_edge {0};
15    auto b_edge {static_cast<int>(matrix.size())-1};
16    auto l_edge {0};
17    auto r_edge {static_cast<int>(matrix[0].size())-1};
18    auto temp {-1};
19
20    // shifting layer by layer.
21    int numlayers = (matrix.size())/2;
22
23    // shifting layer by layer
24    for(int i = 0; i<numlayers; ++i){
25
26        // breaking if they're the same for some reason
27        if(t_edge == b_edge || l_edge == r_edge) {break;}
28
29        // calculatin width
30        auto currwidth {r_edge - l_edge + 1};
31
32        // shifting it around
33        for(int j = 0; j<currwidth-1; ++j){
```

```

34
35     // shifting the four elements for each
36     temp                = matrix[t_edge][l_edge+j];
37     matrix[ t_edge ][ l_edge+j ] = matrix[b_edge-j][l_edge];
38     matrix[ b_edge-j ][ l_edge ] = matrix[b_edge][r_edge-j];
39     matrix[ b_edge ][ r_edge-j ] = matrix[t_edge+j][r_edge];
40     matrix[ t_edge+j ][ r_edge ] = temp;
41 }
42
43 // updating edge-parameters based on the layer we're working with
44 t_edge += 1;
45 b_edge -= 1;
46 l_edge += 1;
47 r_edge -= 1;
48
49 }
50
51 // printing the matrix
52 cout << format("print-matrix\n");
53 fPrintMatrix(matrix);
54
55 // return
56 return(0);
57
58 }

```

---

## 49. Group Anagrams

Given an array of strings `strs`, group the anagrams together. You can return the answer in any order.

### Examples

#### 1. Example 1:

- Input: `strs = ["eat", "tea", "tan", "ate", "nat", "bat"]`
- Output: `[["bat"], ["nat", "tan"], ["ate", "eat", "tea"]]`
- Explanation:
  - There is no string in `strs` that can be rearranged to form "bat".
  - The strings "nat" and "tan" are anagrams as they can be rearranged to form each other.
  - The strings "ate", "eat", and "tea" are anagrams as they can be rearranged to form each other.

#### 2. Example 2:

- Input: `strs = [""]`
- Output: `[[""]]`

#### 3. Example 3:

- Input: `strs = ["a"]`
- Output: `[["a"]]`

## Constraints

- $1 \leq \text{strs.length} \leq 10^4$
- $0 \leq \text{strs}[i].\text{length} \leq 100$
- $\text{strs}[i]$  consists of lowercase English letters.

## Code

---

```
1 // build histogram
2 vector<int> fBuildHist(string input){
3     vector<int> myhist(26, 0);
4     for(auto x: input) {++myhist[static_cast<int>(x - 'a')];}
5     return myhist;
6 }
7
8 // Custom hash function for vector<int>
9 struct VectorHash {
10     std::size_t operator()(const std::vector<int>& v) const {
11         std::size_t hashValue = 0;
12         for (int num : v)
13             hashValue ^= std::hash<int>{}(num) + 0x9e3779b9 + (hashValue << 6) + (hashValue >> 2);
14
15         return hashValue;
16     }
17 };
18 // int main
19 int main(){
20
21     // starting timer
22     Timer timer;
23 }
```



```

24 // input- configuration
25 vector<string> strs{
26     "eat",
27     "tea",
28     "tan",
29     "ate",
30     "nat",
31     "bat"
32 };
33
34 // unordered map for hist to vector of vector of strings
35 unordered_map< vector<int>, vector<string>, VectorHash > histToGroup;
36
37 for(auto x: strs){
38
39     // building hist of current word
40     auto xhist = fBuildHist(x);
41
42     // checking if it exists
43     if (histToGroup.find(xhist) == histToGroup.end())
44         histToGroup[xhist] = vector<string>({x});
45     else
46         histToGroup[xhist].push_back(x);
47
48 }
49
50 // building final output
51 vector<vector<string>> finalOutput;
52 for(auto x: histToGroup) {finalOutput.push_back(x.second);}
53
54 // printing
55 cout << format("final-output = {}\n", finalOutput);
56
57 // return
58 return(0);

```



## 50. Pow(x, n)

Implement  $\text{pow}(x, n)$ , which calculates  $x$  raised to the power  $n$  (i.e.,  $x^n$ ).

### Examples

#### 1. Example 1:

- Input:  $x = 2.00000$ ,  $n = 10$
- Output:  $1024.00000$

#### 2. Example 2:

- Input:  $x = 2.10000$ ,  $n = 3$
- Output:  $9.26100$

#### 3. Example 3:

- Input:  $x = 2.00000$ ,  $n = -2$
- Output:  $0.25000$
- Explanation:  $2^{-2} = 1/2^2 = 1/4 = 0.25$

### Constraints

- $-100.0 < x < 100.0$
- $-2^{31} \leq n \leq 2^{31} - 1$

- $n$  is an integer.
- Either  $x$  is not zero or  $n \neq 0$ .
- $-10^4 \leq x^n \leq 10^4$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto x {2.00000};
8     auto n {10};
9
10    // setup
11    std::function<double(double, long int)> foo = [&foo](
12        double    x,
13        long int   n) -> double
14    {
15
16        // base-cases
17        if (x == 0)    {return 0;}
18        if (n == 0)    {return 1;}
19
20        // performing calculations
21        double result = foo(x, n/2);
22        result = result * result;
23
24        // in case there is some odd-wor
25        if (n%2 == 1) {result = result * x;}
```

```
26
27     // returning the results
28     return result;
29
30 };
31
32 // calling the function
33 auto finalresult = foo(x, std::abs(static_cast<long int>(n)));
34
35 // taking care of the negative case
36 if (n < 0) {finalresult = static_cast<double>(1)/finalresult;}
37
38 // returning the final result
39 cout << format("final-output = {}\n", finalresult);
40
41 // return
42 return(0);
43
44 }
```

---

## 53. Maximum Subarray

Given an integer array `nums`, find the subarray with the largest sum, and return its sum.

### Examples

#### 1. Example 1:

- Input: `nums = [-2,1,-3,4,-1,2,1,-5,4]`
- Output: 6
- Explanation: The subarray `[4,-1,2,1]` has the largest sum 6.

#### 2. Example 2:

- Input: `nums = [1]`
- Output: 1
- Explanation: The subarray `[1]` has the largest sum 1.

#### 3. Example 3:

- Input: `nums = [5,4,-1,7,8]`
- Output: 23
- Explanation: The subarray `[5,4,-1,7,8]` has the largest sum 23.

## Constraints

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

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums    {vector<int>({-2,1,-3,4,-1,2,1,-5,4})};
8
9     // setup
10    auto runningsum {0};
11    auto finaloutput {std::numeric_limits<int>::min()};
12
13    // going through the array
14    for(int i =0; i<nums.size(); ++i){
15        runningsum += nums[i];
16        finaloutput = std::max(runningsum, finaloutput);
17        if (runningsum <0) {runningsum = 0;}
18    }
19
20    // printing the final output
21    cout << format("finaloutput = {}\n", finaloutput);
22
23    // return
24    return(0);
25}
```





## 54. Spiral Matrix

Given an  $m \times n$  matrix, return all elements of the matrix in spiral order.

### Examples

#### 1. Example 1:

- Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
- Output: [1,2,3,6,9,8,7,4,5]

#### 2. Example 2:

- Input: matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
- Output: [1,2,3,4,8,12,11,10,9,5,6,7]

### Constraints

- $m == \text{matrix.length}$
- $n == \text{matrix}[i].\text{length}$
- $1 \leq m, n \leq 10$
- $-100 \leq \text{matrix}[i][j] \leq 100$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     vector<vector<int>> matrix{
8         {1,2,3,4},
9         {5,6,7,8},
10        {9,10,11,12}
11    };
12
13    // setup
14    int left {0};
15    int right {static_cast<int>(matrix[0].size()-1)};
16    int top {0};
17    int bottom {static_cast<int>(matrix.size()-1)};
18    vector<int> finalOutput;
19
20    // moving through this
21    while(left <= right && top <= bottom){
22
23        // moving on upside
24        for(int i = left; i <= right; ++i) {finalOutput.push_back(matrix[top][i]);}
25
26        // moving through the right side
27        for(int i = top+1; i<=bottom-1; ++i) {finalOutput.push_back(matrix[i][right]);}
28
29        // moving through the bottom
30        for(int i = right; top != bottom && i>= left; --i) {finalOutput.push_back(matrix[bottom][i]);}
31
32        // moving through the left
33        for(int i = bottom-1; left!=right && i>= top+1; --i) {finalOutput.push_back(matrix[i][left]);}
```

```
34
35     // updating boundaries
36     ++left; --right; ++top; --bottom;
37
38 }
39
40 // printing the finaloutput
41 cout << format("final-output = {}\n", finalOutput);
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

}

---

## 56. Merge Intervals

Given an array of intervals where  $\text{intervals}[i] = [\text{start}_i, \text{end}_i]$ , merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

### Examples

#### 1. Example 1:

- Input:  $\text{intervals} = [[1,3],[2,6],[8,10],[15,18]]$
- Output:  $[[1,6],[8,10],[15,18]]$
- Explanation: Since intervals  $[1,3]$  and  $[2,6]$  overlap, merge them into  $[1,6]$ .

#### 2. Example 2:

- Input:  $\text{intervals} = [[1,4],[4,5]]$
- Output:  $[[1,5]]$
- Explanation: Intervals  $[1,4]$  and  $[4,5]$  are considered overlapping.

### Constraints

- $1 \leq \text{intervals.length} \leq 104$
- $\text{intervals}[i].\text{length} == 2$
- $0 \leq \text{start}_i \leq \text{end}_i \leq 104$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     vector<vector<int>> intervals{
8         {1,3},
9         {2,6},
10        {8,10},
11        {15,18}
12    };
13
14    // setup
15    vector<vector<int>> finalOutput;
16
17    // sorting the input
18    std::sort(intervals.begin(), intervals.end(),
19        [](const vector<int>& a, const vector<int>& b){return a[0] < b[0];});
20
21    // going through the intervals
22    vector<int> runninginterval {intervals[0][0], intervals[0][1]};
23    for(int i = 1; i<intervals.size(); ++i){
24        if (runninginterval[1] < intervals[i][0]){
25            finalOutput.push_back(runninginterval);           // pushing the results
26            runninginterval = intervals[i];                   // getting new running interval
27        }
28        else {runninginterval[1] = max(runninginterval[1], intervals[i][1]);}
29    }
30    finalOutput.push_back(runninginterval);                   // pushing the results
31
32    // returning
33    cout << format("final-output = {}\n", finalOutput);
```



```
34
35 // return
36 return(0);
37 }
```

---

## 57. Insert Interval

You are given an array of non-overlapping intervals `intervals` where `intervals[i] = [starti, endi]` represent the start and the end of the *i*th interval and `intervals` is sorted in ascending order by `starti`. You are also given an interval `newInterval = [start, end]` that represents the start and end of another interval.

Insert `newInterval` into `intervals` such that `intervals` is still sorted in ascending order by `starti` and `intervals` still does not have any overlapping intervals (merge overlapping intervals if necessary).

Return `intervals` after the insertion.

Note that you don't need to modify `intervals` in-place. You can make a new array and return it.

### Examples

#### 1. Example 1:

- Input: `intervals = [[1,3],[6,9]]`, `newInterval = [2,5]`
- Output: `[[1,5],[6,9]]`

#### 2. Example 2:

- Input: `intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]]`, `newInterval = [4,8]`
- Output: `[[1,2],[3,10],[12,16]]`
- Explanation: Because the new interval `[4,8]` overlaps with `[3,5]`, `[6,7]`, `[8,10]`.

### Constraints

- $0 \leq \text{intervals.length} \leq 10^4$

- `intervals[i].length == 2`
- $0 \leq \text{start}_i \leq \text{end}_i \leq 10^5$
- `intervals` is sorted by `start_i` in ascending order.
- `newInterval.length == 2`
- $0 \leq \text{start} \leq \text{end} \leq 10^5$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     vector<vector<int>> intervals{
8         {1,2},
9         {3,5},
10        {6,7},
11        {8,10},
12        {12,16}
13    };
14    vector<int> newInterval{4,8};
15
16
17    // setup
18    vector<vector<int>> finalOutput;
19    intervals.push_back(newInterval);
20
21    // sorting the interval
```

```

22     std::sort(intervals.begin(),
23             intervals.end(),
24             [](const vector<int>& a,
25               const vector<int>& b){return a[0] < b[0];});
26     vector<int>      runninginterval {intervals[0]};
27
28     // going through the inputs
29     for(int i = 1; i<intervals.size(); ++i){
30         if (runninginterval[1] < intervals[i][0]){
31             finalOutput.push_back(runninginterval);
32             runninginterval = intervals[i];
33         }
34         else{
35             runninginterval[1] = max(runninginterval[1], intervals[i][1]);
36         }
37     }
38
39     // pushing it back
40     finalOutput.push_back(runninginterval);
41     cout << format("final-output = {}\n", finalOutput);
42
43     // return
44     return(0);
45
46 }

```

---

## 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 }
```

---

## 59. Spiral Matrix II

Given a positive integer  $n$ , generate an  $n \times n$  matrix filled with elements from 1 to  $n^2$  in spiral order.

### Examples

#### 1. Example 1:

- Input:  $n = 3$
- Output:  $[[1, 2, 3], [8, 9, 4], [7, 6, 5]]$

#### 2. Example 2:

- Input:  $n = 1$
- Output:  $[[1]]$

### Constraints

- $1 \leq n \leq 20$

### Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5 }
```



```

6 // input- configuration
7 auto n {5};
8
9 // setup
10 auto finaloutput {std::vector<std::vector<int>>>(
11     n,
12     std::vector<int>(n, 0)
13 )};
14
15 //
16 auto top_edge {0};
17 auto bottom_edge {n-1};
18 auto left_edge {0};
19 auto right_edge {n-1};
20 auto count {1};
21 auto curr_row {0};
22 auto curr_col {0};
23
24 // running the loop
25 while(count <= n*n){
26
27     // moving from top left to top right
28     for(auto col = left_edge; col <= right_edge && count <= n*n; ++col) {finaloutput[curr_row][col] = count++;}
29     curr_col = right_edge;
30
31     // moving from topright to bottom right
32     for(auto row = top_edge+1; row <= bottom_edge-1 && count <= n*n; ++row) {finaloutput[row][curr_col] = count++;}
33     curr_row = bottom_edge;
34
35     // moving from bottom-right to bottom-left
36     for(auto col = right_edge; col >= left_edge && count <= n*n; --col) {finaloutput[curr_row][col] = count++;}
37     curr_col = left_edge;
38
39     // moving from bottom-left to top-left
40     for(auto row = bottom_edge-1; row >= top_edge+1 && count <= n*n; --row) {finaloutput[row][curr_col] = count++;}

```

```
41     curr_row    =  top_edge+1;
42
43     // updating boundaries
44     ++top_edge;   --bottom_edge;
45     ++left_edge;  --right_edge;
46 }
47
48 fPrintMatrix(finaloutput);
49
50
51 // return
52 return(0);
53
54 }
```

---

## 60. Permutation Sequence

The set  $[1, 2, 3, \dots, n]$  contains a total of  $n!$  unique permutations.

By listing and labeling all of the permutations in order, we get the following sequence for  $n = 3$

- "123"
- "132"
- "213"
- "231"
- "312"
- "321"

Given  $n$  and  $k$ , return the  $k$ th permutation sequence.

### Examples

#### 1. Example 1

- Input:  $n = 3, k = 3$
- Output: "213"

#### 2. Example 2

- Input:  $n = 4, k = 9$

- Output: "2314"

### 3. Example 3

- Input:  $n = 3, k = 1$
- Output: "123"

## Constraints

- $1 \leq n \leq 9$
- $1 \leq k \leq n!$

## Code

---

```
1 string vectortostr(const vector<int>& pathsofar){
2     auto temp {string("")};
3     for(int i = 0; i<pathsofar.size(); ++i) {temp += std::to_string(pathsofar[i]);}
4     return temp;
5 }
6
7 void foo(vector<int>          pathsofar,
8         const int&          n,
9         int                curr,
10        int&                 leafcount,
11        const int&           k,
12        string&              finaloutput,
13        bool&                 found){
14
15     // sending it back
16     if (found == true) {return;}
```

```

17
18 // adding current number to pathsofar
19 if (std::find(pathsofar.begin(), pathsofar.end(), curr) != pathsofar.end()) {return;}
20
21 // adding to the path
22 pathsofar.push_back(curr);
23
24 // checking if we've reached a leaf
25 if (pathsofar.size() == n) {
26     ++leafcount;
27     if (leafcount == k) {
28         finaloutput = vectortostring(pathsofar);
29         found = true;
30     }
31     return;
32 }
33
34 // calling on the others
35 for(int i = 1; i<=n; ++i) {foo(pathsofar, n, i, leafcount, k, finaloutput, found);}
36 }
37
38 int main(){
39
40     // starting timer
41     Timer timer;
42
43     // input- configuration
44     auto n {8};
45     auto k {29805};
46
47     // setup
48     auto finaloutput {string("")};
49     auto found {false};
50
51     // building the directing block

```

```

52 auto directionalblocks {vector<int>(n-1, 1)};
53 for(int i = directionalblocks.size()-2; i >= 0; --i)
54     directionalblocks[i] = directionalblocks[i+1] * (directionalblocks.size()-i);
55
56 // printing
57 auto startingpoint {k/directionalblocks[0]};
58 startingpoint = std::max(startingpoint, 1);
59
60 // printing the leaf
61 auto leafcount    {directionalblocks[0] * (startingpoint-1)};
62
63 // running
64 for(int i = startingpoint; i<=n; ++i){
65     vector<int> pathsofar;
66     foo(pathsofar, n, i, leafcount, k, finaloutput, found);
67 }
68
69 // printing the final-output
70 cout << format("final-output = {}\n", finaloutput);
71
72 // return
73 return(0);
74 }

```

---

## 61. Rotate List

Given the head of a linked list, rotate the list to the right by  $k$  places.

### Examples

#### 1. Example 1:

- Input: head = [1,2,3,4,5],  $k = 2$
- Output: [4,5,1,2,3]

#### 2. Example 2:

- Input: head = [0,1,2],  $k = 4$
- Output: [2,0,1]

### Constraints

- The number of nodes in the list is in the range [0, 500].
- $-100 \leq \text{Node.val} \leq 100$
- $0 \leq k \leq 2 * 10^9$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto head {new ListNode(1)};
8     head->next = new ListNode(2);
9     head->next->next = new ListNode(3);
10    head->next->next->next = new ListNode(4);
11    head->next->next->next->next = new ListNode(5);
12    auto k {2};
13
14    // trivial cases
15    if (head == nullptr) {fPrintLinkedList("head = ", head); return 0;}
16
17    // storing the starting point
18    auto startingpoint {head};
19    auto listsize      {0};
20
21    // connecting the last-node with the first
22    auto traveller {head};
23    while(traveller->next) {traveller = traveller->next; ++listsize;}
24    ++listsize;
25    traveller->next = head;
26
27    // to get rid of cycles
28    k = k % listsize;
29    auto cutoffpoint {listsize - k};
30    auto count       {0};
31    auto finalresult  {static_cast<ListNode*>(nullptr)};
32
33    // finding the cutting off point
```



```

34     traveller = head;
35     while(traveller){
36         ++count;                                // appending count
37         if (count == cutoffpoint){
38             finalresult = traveller->next;        // setting up the new head
39             traveller->next = nullptr;           // setting up last pointer
40             break;                               // breaking up
41         }
42         traveller = traveller->next;              // moving to the next point
43     }
44
45     // printing the linked list
46     fPrintLinkedList("final-result = ", finalresult);
47
48     // return
49     return(0);
50
51 }

```

---

## 62. Unique Paths

There is a robot on an  $m \times n$  grid. The robot is initially located at the top-left corner (i.e., `grid[0][0]`). The robot tries to move to the bottom-right corner (i.e., `grid[m - 1][n - 1]`). The robot can only move either down or right at any point in time.

Given the two integers  $m$  and  $n$ , return the number of possible unique paths that the robot can take to reach the bottom-right corner.

The test cases are generated so that the answer will be less than or equal to  $2 * 10^9$ .

### Examples

#### 1. Example 1:

- Input:  $m = 3, n = 7$
- Output: 28

#### 2. Example 2:

- Input:  $m = 3, n = 2$
- Output: 3

### Constraints

- $1 \leq m, n \leq 100$

### Code

---

```

1  int main(){
2
3      // starting timer
4      Timer timer;
5
6      // input- configuration
7      auto m {3};
8      auto n {7};
9
10     // setup
11     auto row    = [&n](int x){return x/n;};
12     auto col    = [&n](int x){return x%n;};
13     auto tid    {m*n - 1};
14
15     // building dp-table
16     vector<vector<int>> dptable;
17     for(int i = 0; i<m; ++i) {dptable.push_back(vector<int>(n, 0));}
18     dptable[m-1][n-1] = 1;
19
20     // moving through the thing
21     while(tid>=0){
22
23         auto currcol    {col(tid)};
24         auto currow      {row(tid)};
25         auto nummethods  {dptable[currow][currcol]};
26
27         if (currcol < n-1) {nummethods += dptable[currow][currcol+1];}
28         if (currow < m-1) {nummethods += dptable[currow+1][currcol];}
29
30         dptable[currow][currcol] = nummethods;
31
32         --tid;
33     }
34

```

```
35 // returning
36 cout << format("final-output = {}\n", dptable[0][0]);
37
38 // return
39 return(0);
40
41 }
```

---

## 63. Unique Paths II

You are given an  $m \times n$  integer array `grid`. There is a robot initially located at the top-left corner (i.e., `grid[0][0]`). The robot tries to move to the bottom-right corner (i.e., `grid[m - 1][n - 1]`). The robot can only move either down or right at any point in time.

An obstacle and space are marked as 1 or 0 respectively in `grid`. A path that the robot takes cannot include any square that is an obstacle.

Return the number of possible unique paths that the robot can take to reach the bottom-right corner.

The testcases are generated so that the answer will be less than or equal to  $2 * 10^9$ .

### Examples

#### 1. Example 1:

- Input: `obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]`
- Output: 2
- Explanation: There is one obstacle in the middle of the 3x3 grid above.
  - There are two ways to reach the bottom-right corner:
    - 1. Right → Right → Down → Down
    - 2. Down → Down → Right → Right

#### 2. Example 2:

- Input: `obstacleGrid = [[0,1],[0,0]]`
- Output: 1

## Constraints

- $m == \text{obstacleGrid.length}$
- $n == \text{obstacleGrid}[i].\text{length}$
- $1 \leq m, n \leq 100$
- $\text{obstacleGrid}[i][j]$  is 0 or 1.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     vector<vector<int>> obstacleGrid {
8         {0,0,0},
9         {0,1,0},
10        {0,0,0}
11    };
12
13    // returning zero if the obstacleGrid has a 1 in the end
14    if (obstacleGrid[obstacleGrid.size()-1][obstacleGrid[0].size()-1] == 1) return 0;
15
16    // setup
17    vector<vector<long long>> dptable;
18    for(int i = 0; i<obstacleGrid.size()+1; ++i){
19        dptable.push_back(vector<long long>(obstacleGrid[0].size()+1, 0));
20    }
21}
```

```

22 // writing a one in the destination
23 dptable[obstacleGrid.size()-1][obstacleGrid[0].size()-1] = 1;
24
25 // setting up the copying
26 for(int i = obstacleGrid.size()-1; i>=0; --i){
27     for(int j = obstacleGrid[0].size()-1; j>=0; --j){
28
29         // leaving the paths from this as zero in the case where there is an obstacle.
30         if (obstacleGrid[i][j] == 1) continue;
31
32         // summing up from the right
33         long long currvalue {dptable[i][j] + dptable[i+1][j] + dptable[i][j+1]};
34
35         // writing the sum to the current-value
36         dptable[i][j] = currvalue;
37     }
38 }
39
40 // returning the starting point
41 cout << format("final-output = {}\n", static_cast<int>(dptable[0][0]));
42
43 // return
44 return(0);
45
46 }

```

---

## 64. Minimum Path Sum

Given a  $m \times n$  grid filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

### Examples

#### 1. Example 1:

- Input: grid = 

1	3	1
1	5	1
4	2	1
- Output: 7
- Explanation: Because the path  $1 \rightarrow 3 \rightarrow 1 \rightarrow 1 \rightarrow 1$  minimizes the sum.

#### 2. Example 2:

- Input: grid = 

1	2	3
4	5	6
- Output: 12

### Constraints

- $m == \text{grid.length}$



- $n == \text{grid}[i].\text{length}$
- $1 \leq m, n \leq 200$
- $0 \leq \text{grid}[i][j] \leq 200$

## Code

---

```

1  int main(){
2
3      // starting timer
4      Timer timer;
5
6      // input- configuration
7      vector<vector<int>> grid {
8          {1,3,1},
9          {1,5,1},
10         {4,2,1}
11     };
12
13     // setup
14     auto mingrid = grid;
15
16     // going through the values from the bottom right to left
17     for(int i = grid.size()*grid[0].size()-1; i>=0; --i){
18
19         // converting numbers to the row--index and colindex
20         auto rowindex {i/static_cast<int>(grid[0].size())};
21         auto colindex {i%static_cast<int>(grid[0].size())};
22
23         // finding sum of the children with present
24         auto smallestchild {std::numeric_limits<int>::max()};
25

```

```

26 // checking the bottom left
27 if (rowindex == grid.size()-1 && colindex == grid[0].size()-1) {smallestchild = 0; }
28 else if(rowindex == grid.size()-1) {smallestchild = mingrid[rowindex][colindex+1]; }
29 else if (colindex == grid[0].size()-1) {smallestchild = mingrid[rowindex+1][colindex];}
30 else {smallestchild = min(mingrid[rowindex+1][colindex],
31                             mingrid[rowindex][colindex+1]);}
32
33 // storing to the final results
34 mingrid[rowindex][colindex] = mingrid[rowindex][colindex] + smallestchild;
35
36 }
37
38 // returning the smallest value
39 cout << format("final-output = {}\n", mingrid[0][0]);
40
41 // return
42 return(0);
43
44 }

```

---

## 66. Plus One

You are given a large integer represented as an integer array `digits`, where each `digits[i]` is the *i*th digit of the integer. The digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's.

Increment the large integer by one and return the resulting array of digits.

### Examples

#### 1. Example 1:

- Input: `digits = [1,2,3]`
- Output: `[1,2,4]`
- Explanation: The array represents the integer 123.
  - Incrementing by one gives  $123 + 1 = 124$ .
  - Thus, the result should be `[1,2,4]`.

#### 2. Example 2:

- Input: `digits = [4,3,2,1]`
- Output: `[4,3,2,2]`
- Explanation: The array represents the integer 4321.
  - Incrementing by one gives  $4321 + 1 = 4322$ .
  - Thus, the result should be `[4,3,2,2]`.

#### 3. Example 3:

- Input: `digits = [9]`

- Output: [1,0]
- Explanation: The array represents the integer 9.
  - Incrementing by one gives  $9 + 1 = 10$ .
  - Thus, the result should be [1,0].

## Constraints

- $1 \leq \text{digits.length} \leq 100$
- $0 \leq \text{digits}[i] \leq 9$
- digits does not contain any leading 0's.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto digits {vector<int>{1,2,3}} ;
8
9     // setup
10    auto carry {1};
11    auto sum {-1};
12    vector<int> finalOutput(digits.size()+1, 0);
13
14    // goign through the numbers
15    for(int i = 0; i<digits.size(); ++i){
```

```
16
17     // calculating sum and carry
18     sum    = digits[digits.size()-1-i] + carry;
19     carry  = sum / 10;
20     sum    = sum - carry*10;
21
22     // storing sum and carry
23     finalOutput[finalOutput.size()-1-i] = sum;
24 }
25
26 // in case there is one carry left
27 if (carry != 0) {finalOutput[0] = 1;}
28 else {finalOutput = vector<int>(finalOutput.begin()+1, finalOutput.end());}
29
30 // sending back the finaloutput
31 cout << format("finalOutput = {}\n", finalOutput);
32
33 // return
34 return(0);
35
36 }
```

---

## 67. Add Binary

Given two binary strings *a* and *b*, return their sum as a binary string.

### Examples

1. **Example 1:**

- Input: *a* = "11", *b* = "1"
- Output: "100"

2. **Example 2:**

- Input: *a* = "1010", *b* = "1011"
- Output: "10101"

### Constraints

- $1 \leq a.length, b.length \leq 10^4$
- *a* and *b* consist only of '0' or '1' characters.
- Each string does not contain leading zeros except for the zero itself.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto a {string("1010")};
8     auto b {string("1011")};
9
10
11    // setup
12    int carry {0};
13    int a_top {0};
14    int b_top {0};
15    int c_top {0};
16    string finalOutput {};
17
18    // running until the things are empty
19    while(a.size()!=0 || b.size()!=0){
20
21        // copying the carry over tot he sum
22        c_top = carry;
23
24        // shaving off the top
25        if (a.size()){
26            a_top = static_cast<int>(a[a.size()-1] - '0'); // taking the top
27            a = a.substr(0, a.size()-1); // removing the top
28            c_top += a_top; // adding to the sum
29        }
30        if (b.size()){
31            b_top = static_cast<int>(b[b.size()-1] - '0'); // taking the top
32            b = b.substr(0, b.size()-1); // removing the top
33            c_top += b_top; // adding to the sum
```

```

34     }
35
36     // evaluating carry
37     if (c_top == 0)      {carry = 0; c_top = 0;}
38     else if (c_top == 1) {carry = 0; c_top = 1;}
39     else if (c_top == 2) {carry = 1; c_top = 0;}
40     else if (c_top == 3) {carry = 1; c_top = 1;}
41     else                 {cout << format("what the actual fuck") << endl;}
42
43     // adding c-TOp to the end of the final output
44     finalOutput = std::string(1,c_top+'0') + finalOutput;
45
46 }
47
48 // check if there is a carry out there
49 if (carry) {finalOutput = std::string(1,carry+'0') + finalOutput;}
50
51 // returnign the final output
52 cout << format("final-output = {}\n", finalOutput);
53
54 // return
55 return(0);
56
57 }

```

---



## 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", "else", "we", "do"], 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 }
```

---

## 69. Sqrt(x)

Given a non-negative integer  $x$ , return the square root of  $x$  rounded down to the nearest integer. The returned integer should be non-negative as well.

### Examples

#### 1. Example 1:

- Input:  $x = 4$
- Output: 2
- Explanation: The square root of 4 is 2, so we return 2.

#### 2. Example 2:

- Input:  $x = 8$
- Output: 2
- Explanation: The square root of 8 is 2.82842..., and since we round it down to the nearest integer, 2 is returned.

### Constraints

- $0 \leq x \leq 2^{31} - 1$



## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto x {8};
8
9     // setup
10    int finaloutput {-1};
11    auto square = [](int x){return x*x;};
12
13    // running
14    for(int i = 0; i<x; ++i){
15        cout << format("{}{},{}\n",square(i),x,square(i+1));
16        if (square(i) <= x && square(i+1)>x) {finaloutput = i; break;}
17    }
18
19    // printing the output
20    cout << format("final-output = {}\n", finaloutput);
21
22    // return
23    return(0);
24
25 }
```

---

## 70. Climbing Stairs

You are climbing a staircase. It takes  $n$  steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

### Examples

#### 1. Example 1:

- Input:  $n = 2$
- Output: 2
- Explanation: There are two ways to climb to the top.
  - 1. 1 step + 1 step
  - 2. 2 steps

#### 2. Example 2:

- Input:  $n = 3$
- Output: 3
- Explanation: There are three ways to climb to the top.
  - 1. 1 step + 1 step + 1 step
  - 2. 1 step + 2 steps
  - 3. 2 steps + 1 step

### Constraints

- $1 \leq n \leq 45$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto n {2};
8
9     // setup
10    std::vector<int> dpTable;
11
12    // number of steps from the last step to the last
13    dpTable.push_back(0);
14    dpTable.push_back(1);
15    dpTable.push_back(2);
16
17    // finding number of ways this can be done
18    for(int i = 2; i<n; ++i){
19        // adding up the last two values
20        dpTable.push_back(dpTable[dpTable.size() - 1] + dpTable[dpTable.size() - 2]);
21    }
22
23    // return the final output
24    cout << format("final-output = {}\n", dpTable[n]);
25
26    // return
27    return(0);
28
29 }
```

---

## 71. Simplify Path

You are given an absolute path for a Unix-style file system, which always begins with a slash '/'. Your task is to transform this absolute path into its simplified canonical path.

The rules of a Unix-style file system are as follows:

- A single period '.' represents the current directory.
- A double period '..' represents the previous/parent directory.
- Multiple consecutive slashes such as '/' and '//' are treated as a single slash '/'.
- Any sequence of periods that does not match the rules above should be treated as a valid directory or file name. For example, '...' and '....' are valid directory or file names.

The simplified canonical path should follow these rules:

- The path must start with a single slash '/'.
- Directories within the path must be separated by exactly one slash '/'.
- The path must not end with a slash '/', unless it is the root directory.
- The path must not have any single or double periods ('.' and '..') used to denote current or parent directories.

Return the simplified canonical path.

## Examples

### 1. Example 1:

- Input: path = `"/home/"`
- Output: `"/home"`
- Explanation: The trailing slash should be removed.

### 2. Example 2:

- Input: path = `"/home//foo/"`
- Output: `"/home/foo"`
- Explanation: Multiple consecutive slashes are replaced by a single one.

### 3. Example 3:

- Input: path = `"/home/user/Documents/../Pictures"`
- Output: `"/home/user/Pictures"`
- Explanation: A double period `".."` refers to the directory up a level (the parent directory).

## Constraints

- $1 \leq \text{path.length} \leq 3000$
- path consists of English letters, digits, period `'.'`, slash `'/'` or `'_'`.
- path is a valid absolute Unix path.

## Code

---

```
1  int main(){
2
3      // starting timer
4      Timer timer;
5
6      // input- configuration
7      string path    {"home/foo/"};
8
9      // setup
10     vector<string> stack;
11
12     // going through the array
13     int i = 0 ;
14     while (i < path.size()){
15
16         // moving until end of string or a /
17         string temp;
18         while(path[i]!='/' && i<path.size()){
19             temp += std::string(1,path[i++]);
20         }
21
22         // continuing if temp size is zero
23         if (temp.size()==0)  {++i; continue;}
24
25         // evaluating the current string
26         if (temp == ".")      {++i; continue;}
27         else if (temp == "..") {
28             if (stack.size()!=0) {stack.pop_back();}
29             ++i;}
30         else {stack.push_back(temp); ++i;}
31     }
32
33 }
```

```
34 // reconstructing final path
35 string finalOutput;
36
37 // reconstructing
38 for(auto x: stack) {finalOutput += "/" + x;}
39
40 // size thing
41 if (finalOutput.size() == 0) {finalOutput += "/";}
42
43 // returning
44 cout << format("finalOutput = {}\n", finalOutput);
45
46 // return
47 return(0);
48
49 }
```

---

## 73. Set Matrix Zeroes

Given an  $m \times n$  integer matrix `matrix`, if an element is 0, set its entire row and column to 0's. You must do it in place.

### Examples

#### 1. Example 1

- Input: `matrix = [[1,1,1],[1,0,1],[1,1,1]]`
- Output: `[[1,0,1],[0,0,0],[1,0,1]]`

#### 2. Example 2

- Input: `matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]`
- Output: `[[0,0,0,0],[0,4,5,0],[0,3,1,0]]`

### Constraints

- $m == \text{matrix.length}$
- $n == \text{matrix}[0].\text{length}$
- $1 \leq m, n \leq 200$
- $-231 \leq \text{matrix}[i][j] \leq 231 - 1$



## Code

---

```
1  int main(){
2
3      // starting timer
4      Timer timer;
5
6      // input- configuration
7      vector<vector<int>> matrix{
8          {0,1,2,0},
9          {3,4,5,2},
10         {1,3,1,5}
11     };
12
13     // setup
14     auto colregister    {vector<bool>(matrix[0].size(), false)};
15     auto rowregister    {vector<bool>(matrix.size(), false)};
16
17     // registering
18     #pragma omp parallel for
19     for(int i = 0; i<matrix.size(); ++i)
20         for(int j = 0; j<matrix[0].size(); ++j)
21             if (matrix[i][j] == 0) {colregister[j] = true; rowregister[i] = true;}
22
23     // rewriting
24     #pragma omp parallel for
25     for(int i = 0; i<matrix.size(); ++i)
26         for(int j = 0; j<matrix[0].size(); ++j)
27             if (colregister[j] == true || rowregister[i] == true) {matrix[i][j] = 0;}
28
29     // printing the final output
30     cout << format("final-output \n");
31     fPrintMatrix(matrix);
32
33     // return
```

```
34     return(0);
35
36 }
```

---

## 74. Search a 2D Matrix

You are given an  $m \times n$  integer matrix with the following two properties:

1. Each row is sorted in non-decreasing order.
2. The first integer of each row is greater than the last integer of the previous row.

Given an integer target, return true if target is in matrix or false otherwise.

You must write a solution in  $O(\log(m * n))$  time complexity.

### Examples

#### 1. Example 1

- Input: matrix = `[[1,3,5,7],[10,11,16,20],[23,30,34,60]]`, target = 3
- Output: true

#### 2. Example 2

- Input: matrix = `[[1,3,5,7],[10,11,16,20],[23,30,34,60]]`, target = 13
- Output: false

### Constraints

- `m == matrix.length`

- $n == \text{matrix}[i].\text{length}$
- $1 \leq m, n \leq 100$
- $-10^4 \leq \text{matrix}[i][j], \text{target} \leq 10^4$

## Code

---

```

1 struct Matrix{
2     vector<vector<int>> matrix;
3     int numcols;
4     int numrows;
5     Matrix(vector<vector<int>> matrix0): matrix(matrix0){
6         numcols = matrix0[0].size();
7         numrows = matrix0.size();
8     }
9     int operator()(int row, int col) {return matrix[row][col];}
10    int operator[] (int tid)          {return matrix[tid/numcols][tid%numcols];}
11    int numel()                       {return matrix.size() * matrix[0].size();}
12 };
13
14 int main(){
15
16     // starting timer
17     Timer timer;
18
19     // input- configuration
20     vector<vector<int>> matrix = {
21         {1,3,5,7},
22         {10,11,16,20},
23         {23,30,34,60}};
24     auto target {3};
25

```

```

26 // setup
27 Matrix m(matrix);
28 int left {0};
29 int right {static_cast<int>(matrix.size() * matrix[0].size()) - 1};
30 int mid {-1};
31
32 // binary search
33 while(left <= right){
34     mid = (left + right)/2;
35     if (m[mid] < target) {left = mid+1;}
36     else if (m[mid] > target) {right = mid-1;}
37     else {cout << format("final-output = true\n"); return 0;}
38 }
39
40 // returning false
41 cout << format("final-output = false\n");
42
43 // return
44 return(0);
45
46 }

```

---

## 75. Sort Colors

Given an array `nums` with `n` objects colored red, white, or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

### Examples

#### 1. Example 1:

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

#### 2. Example 2:

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

### Constraints

- `n == nums.length`
- $1 \leq n \leq 300$
- `nums[i]` is either 0, 1, or 2.

## Code

---

```
1  int main(){
2
3      // starting timer
4      Timer timer;
5
6      // input- configuration
7      auto nums {vector<int>{2,0,2,1,1,0}};
8
9      // setup
10     vector<int> histogram(3, 0); histogram.reserve(histogram.size());
11
12     // moving through inputs
13     for(const auto& x: nums) {++histogram[x];}
14
15     // rewriting
16     auto counter {0};
17     for(int i = 0; i<nums.size(); ++i){
18         while(counter < histogram.size() && histogram[counter] == 0) ++counter;
19         nums[i] = counter; histogram[counter]--;
20     }
21
22     // printing the final-output
23     cout << format("final-output = {}\n", nums);
24
25     // return
26     return(0);
27
28 }
```

---

## 76. Minimum Window Substring

Given two strings  $s$  and  $t$  of lengths  $m$  and  $n$  respectively, return the minimum window substring of  $s$  such that every character in  $t$  (including duplicates) is included in the window. If there is no such substring, return the empty string `""`.

The testcases will be generated such that the answer is unique.

### Examples

#### 1. Example 1:

- Input:  $s = \text{"ADOBECODEBANC"}, t = \text{"ABC"}$
- Output: `"BANC"`
- Explanation: The minimum window substring `"BANC"` includes 'A', 'B', and 'C' from string  $t$ .

#### 2. Example 2:

- Input:  $s = \text{"a"}, t = \text{"a"}$
- Output: `"a"`
- Explanation: The entire string  $s$  is the minimum window.

#### 3. Example 3:

- Input:  $s = \text{"a"}, t = \text{"aa"}$
- Output: `""`
- Explanation: Both 'a's from  $t$  must be included in the window.
  - Since the largest window of  $s$  only has one 'a', return empty string.



## Constraints

- $m == s.length$
- $n == t.length$
- $1 \leq m, n \leq 10^5$
- $s$  and  $t$  consist of uppercase and lowercase English letters.

## Code

---

```
1 // function to convert letter to index
2 int fCharToIndex(char x){
3     if (x >= 65 && x <=90)        {return x - 65;}
4     else if (x >= 97 && x <= 122) {return (x - 97 + (90-65+1));}
5     else                          {std::cerr<<"this shouldn't happen"; return -1;}
6 }
7
8 // compare histograms
9 template<typename T>
10 bool fCompareHist(const vector<T>& t_hist, const vector<T>& runninghist){
11
12     // going throguh the list
13     for(int i = 0; i<t_hist.size(); ++i)
14         if (t_hist[i] > runninghist[i])
15             return false;
16
17     // in case all conditions are met
18     return true;
19 }
20 int main(){
21
```

```

22 // starting timer
23 Timer timer;
24
25 // input- configuration
26 auto s {string("ADOBE CODEBANC")};
27 auto t {string("ABC")};
28
29 // setup
30 int p1 {0};
31 int p2 {0};
32 vector<int> finalOutputBoundaries(2, -1);
33 vector<int> t_hist(52, 0); t_hist.reserve(t_hist.size());
34 vector<int> runninghist(52, 0); runninghist.reserve(runninghist.size());
35 auto minlength {std::numeric_limits<int>::max()};
36
37 // setting up t-hist
38 for(const auto x: t) {++t_hist[fCharToIndex(x)];}
39
40 // moving through the string
41 while(p1 <= p2 && p2<s.size()){
42
43     // adding current character
44     if (p2<s.size()) {++runninghist[fCharToIndex(s[p2])];}
45
46     // checking if the two histograms are comparable
47     while(p1<p2 && fCompareHist(t_hist, runninghist)){
48
49         auto currstringsize {p2-p1+1};
50
51         if (currstringsize>0 && currstringsize < minlength){
52             finalOutputBoundaries = vector<int>({p1, p2});
53             minlength = currstringsize;
54         }
55
56         --runninghist[fCharToIndex(s[p1])];

```

```

57     ++p1;
58 }
59
60 // incrementing p2;
61 ++p2;
62
63 }
64
65 // printing the final output
66 string finalOutput {string(s.begin()+finalOutputBoundaries[0], s.begin() + finalOutputBoundaries[1]+1)};
67
68 // returning
69 if (finalOutputBoundaries[0] == -1 || finalOutputBoundaries[1] == -1) {finalOutput = "";}
70
71 // printing the final-output
72 cout << format("final-output = {}\n", finalOutput);
73
74 // return
75 return(0);
76
77 }

```

---

## 77. Combinations

Given two integers  $n$  and  $k$ , return all possible combinations of  $k$  numbers chosen from the range  $[1, n]$ .

You may return the answer in any order.

### Examples

#### 1. Example 1:

- Input:  $n = 4, k = 2$
- Output:  $[[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]$
- Explanation: There are 4 choose 2 = 6 total combinations.
  - Note that combinations are unordered, i.e.,  $[1,2]$  and  $[2,1]$  are considered to be the same combination.

#### 2. Example 2:

- Input:  $n = 1, k = 1$
- Output:  $[[1]]$
- Explanation: There is 1 choose 1 = 1 total combination.

### Constraints

- $1 \leq n \leq 20$
- $1 \leq k \leq n$

## Code

---

```
1 void foo(vector<int>          pathsofar,
2         int                  n,
3         int                  k,
4         vector<vector<int>>& finalOutput){
5
6     // checking legnth of pathsofar
7     if (pathsofar.size() > k) return;
8
9     // adding the current path to the final output if the length is k
10    if (pathsofar.size() == k) {finalOutput.push_back(pathsofar); return;}
11
12    // setting up starting point
13    int startingpoint;
14    if (pathsofar.size() == 0) startingpoint = 0+1;
15    else                        startingpoint = pathsofar[pathsofar.size()-1]+1;
16
17    // running through the options
18    for(int i = startingpoint; i<=n; ++i){
19
20        // calling the graph on those
21        auto pathsofar_temp = pathsofar;
22        pathsofar_temp.push_back(i);
23
24        // calling the functiona again
25        foo(pathsofar_temp, n, k, finalOutput);
26    }
27
28    // returning
29    return;
30 }
31
32 int main(){
33
```

```
34 // starting timer
35 Timer timer;
36
37 // input- configuration
38 auto n {4};
39 auto k {2};
40
41 // setup
42 vector< vector<int> > finalOutput;
43 vector<int>          pathsofar;
44
45 // calling the function
46 foo(pathsofar, n, k, finalOutput);
47
48 // printing the output
49 cout << format("finalOutput = {}\n", finalOutput);
50
51 // return
52 return(0);
53
54 }
```

---

## 78. Subsets

Given an integer array `nums` of unique elements, return all possible subsets (the power set).

The solution set must not contain duplicate subsets. Return the solution in any order.

### Examples

1. **Example 1:**

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

2. **Example 2:**

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

### Constraints

- $1 \leq \text{nums.length} \leq 10$
- $-10 \leq \text{nums}[i] \leq 10$
- All the numbers of `nums` are unique.

## Code

---

```
1 // recursion function
2 void foo(vector<vector<int>>& finalOutput,
3         const vector<int>&   nums,
4         int                 currindex,
5         vector<int>         runningvec,
6         const int&          maxnumelements)
7 {
8     // checking size
9     if (runningvec.size() == maxnumelements) {finalOutput.push_back(runningvec);return;}
10
11    // checking the next set of options
12    for(int i = currindex+1; i < nums.size(); ++i){
13
14        // creating arguments for next call
15        auto runningvec_temp {runningvec};
16        runningvec_temp.push_back(nums[i]);
17        foo(finalOutput, nums, i, runningvec_temp, maxnumelements);
18    }
19
20    // returning
21    return;
22 }
23
24 int main(){
25
26    // starting timer
27    Timer timer;
28
29    // input- configuration
30    auto nums {1,2,3};
31
32    // setup
```



```
34     vector<vector<int>> finalOutput {vector<int>({})};
35
36     // calling the function
37     #pragma omp parallel for
38     for(int i = 1; i<=nums.size(); ++i)
39         foo(finalOutput, nums, -1, vector<int>({}), i);
40
41     // printing the final-output
42     cout << format("finalOutput = {}\n", finalOutput);
43
44     // return
45     return(0);
46
47 }
```

---

## 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 }

```

---

## Remove Duplicates from Sorted List II

Given the head of a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list. Return the linked list sorted as well.

### Examples

1. **Example 1:**

- Input: head = [1,2,3,3,4,5]
- Output: [1,2,5]

2. **Example 2:**

- Input: head = [1,1,1,2,3]
- Output: [2,3]

### Constraints

- The number of nodes in the list is in the range [0, 300].
- $-100 \leq \text{Node.val} \leq 100$
- The list is guaranteed to be sorted in ascending order.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto head = new ListNode(1);
8     head->next = new ListNode(2);
9     head->next->next = new ListNode(3);
10    head->next->next->next = new ListNode(3);
11    head->next->next->next->next = new ListNode(4);
12    head->next->next->next->next->next = new ListNode(4);
13    head->next->next->next->next->next->next = new ListNode(5);
14
15    // setup
16    auto traveller {head};
17    auto prehead_finaloutput {new ListNode(std::numeric_limits<int>::min())};
18    auto traveller_finaloutput {prehead_finaloutput};
19
20    auto currvalue {-1};
21    auto nextvalue {-1};
22
23    // going through the list
24    while(traveller){
25
26        // checking if current-value is different from top
27        currvalue = traveller->val;
28        nextvalue = traveller->next == nullptr ?
29            std::numeric_limits<int>::min() : traveller->next->val;
30
31        // checking if they're the same
32        if (currvalue == nextvalue){
33            // moving until we find a new-value
```

```

34     while(traveller && traveller->val == currvalue) {traveller = traveller->next;}
35 }
36 else{
37     // pushing current-value
38     traveller_finaloutput->next = traveller;
39     traveller_finaloutput      = traveller_finaloutput->next;
40     traveller                  = traveller->next;
41 }
42 }
43
44 // printing the final-output
45 fPrintLinkedList("final-output = ", prehead_finaloutput->next);
46
47 // return
48 return(0);
49
50 }

```

---

## 86. Partition List

Given the head of a linked list and a value  $x$ , partition it such that all nodes less than  $x$  come before nodes greater than or equal to  $x$ . You should preserve the original relative order of the nodes in each of the two partitions.

### Examples

#### 1. Example 1:

- Input: head = [1,4,3,2,5,2],  $x = 3$
- Output: [1,2,2,4,3,5]

#### 2. Example 2:

- Input: head = [2,1],  $x = 2$
- Output: [1,2]

### Constraints

- The number of nodes in the list is in the range  $[0, 200]$ .
- $-100 \leq \text{Node.val} \leq 100$
- $-200 \leq x \leq 200$

## Code

---

```
1  int main(){
2
3      // starting timer
4      Timer timer;
5
6      // input- configuration
7      auto head {new ListNode(1)};
8      head->next = new ListNode(4);
9      head->next->next = new ListNode(3);
10     head->next->next->next = new ListNode(2);
11     head->next->next->next->next = new ListNode(5);
12     head->next->next->next->next->next = new ListNode(2);
13     auto x {3};
14
15     // setup
16     ListNode* traveller      = head;
17     ListNode* lefthead       = new ListNode(-1);
18     ListNode* lefttraveller  = lefthead;
19     ListNode* righthead      = new ListNode(-1);
20     ListNode* righttraveller = righthead;
21
22     // going through the list
23     while(traveller){
24         // checking next-value
25         if (traveller->val < x){
26             lefttraveller->next = new ListNode(traveller->val);
27             lefttraveller       = lefttraveller->next;
28         }
29         else {
30             righttraveller->next = new ListNode(traveller->val);
31             righttraveller       = righttraveller->next;
32         }
33     }
```



```
34     // moving traveller
35     traveller = traveller->next;
36 }
37
38 // connecting
39 lefttraveller->next = righthead->next;
40
41 // returning
42 fPrintLinkedList("Final-output = ", lefthead->next);
43
44 // return
45 return(0);
46
47 }
```

---

## 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 }
```

---

## 92. Reverse Linked List II

Given the head of a singly linked list and two integers left and right where left  $\leq$  right, reverse the nodes of the list from position left to position right, and return the reversed list.

### Examples

#### 1. Example 1:

- Input: head = [1,2,3,4,5], left = 2, right = 4
- Output: [1,4,3,2,5]

#### 2. Example 2:

- Input: head = [5], left = 1, right = 1
- Output: [5]

### Constraints

- The number of nodes in the list is n.
- $1 \leq n \leq 500$
- $-500 \leq \text{Node.val} \leq 500$
- $1 \leq \text{left} \leq \text{right} \leq n$

## Code

---

```
1  int main(){
2
3      // starting timer
4      Timer timer;
5
6      // input- configuration
7      ListNode* head          = new ListNode(1);
8      head->next               = new ListNode(2);
9      head->next->next          = new ListNode(3);
10     head->next->next->next      = new ListNode(4);
11     head->next->next->next->next = new ListNode(5);
12     auto left {2};
13     auto right {4};
14
15     // trivial case
16     if (left == right) {fPrintLinkedList(head); cout << endl; return 0;}
17
18     // setup
19     ListNode* traveller      = nullptr;
20     ListNode* entrypoint     = nullptr;
21     ListNode* exitpoint      = nullptr;
22     ListNode* revhead        = nullptr;
23     ListNode* dummy          = new ListNode(0);
24     dummy->next               = head;
25
26     // moving through list
27     int nodeposition = -1;
28     traveller = dummy;
29     while(traveller){
30
31         ++nodeposition;    // updating node position
32
33         // finding important points
```

```

34     if(nodeposition == left-1)      {entrypoint = traveller;}
35     else if(nodeposition == left)   {revhead  = traveller;}
36     else if(nodeposition == right+1) {exitpoint = traveller;}
37
38     // moving traveller to next point
39     traveller = traveller->next;
40 }
41
42 // reversing the head
43 ListNode* prev      = nullptr;
44 ListNode* actualright = nullptr;
45 traveller          = revhead;
46 nodeposition       = left;
47
48 while(traveller && nodeposition <= right){
49     ++nodeposition;
50     actualright  = traveller->next; // storing original right
51     traveller->next = prev;        // reconnection
52     prev          = traveller;    // storing prev for next iteration
53     traveller     = actualright;  // moving to original right
54 }
55
56 // tying things together
57 entrypoint->next = prev;
58 revhead->next   = exitpoint;
59
60 // returning dummy-es nstex
61 fPrintLinkedList(dummy->next); cout << endl;
62
63 // return
64 return(0);
65
66 }

```

---

## 94. Binary Tree Inorder Traversal

Given the root of a binary tree, return the inorder traversal of its nodes' values.

### Constraints

- The number of nodes in the tree is in the range [0, 100].
- $-100 \leq \text{Node.val} \leq 100$

### Code

---

```
1 void foo(TreeNode* root,
2     vector<int>& finalOutput)
3 {
4     // sending it back
5     if (root == nullptr) return;
6
7     // sending it left
8     foo(root->left, finalOutput);
9
10    // adding the current-value
11    finalOutput.push_back(root->val);
12
13    // sending it right
14    foo(root->right, finalOutput);
15
16    // sending it back
17    return;
18 }
19
```



```
20 int main(){
21
22     // starting timer
23     Timer timer;
24
25     // input- configuration
26     auto root      {new TreeNode(1)};
27     root->right     = new TreeNode(2);
28     root->right->left = new TreeNode(3);
29
30     // setup
31     vector<int> finalOutput;
32
33     // calling the function
34     foo(root, finalOutput);
35
36     // printing
37     cout << format("final-output = {}\n", finalOutput);
38
39     // return
40     return(0);
41
42 }
```

---

## 97. Interleaving String

Given strings  $s_1$ ,  $s_2$ , and  $s_3$ , find whether  $s_3$  is formed by an interleaving of  $s_1$  and  $s_2$ .

An interleaving of two strings  $s$  and  $t$  is a configuration where  $s$  and  $t$  are divided into  $n$  and  $m$  substrings respectively, such that:

- $s = s_1 + s_2 + \dots + s_n$
- $t = t_1 + t_2 + \dots + t_m$
- $|n - m| \leq 1$
- The interleaving is  $s_1 + t_1 + s_2 + t_2 + s_3 + t_3 + \dots$  or  $t_1 + s_1 + t_2 + s_2 + t_3 + s_3 + \dots$

Note:  $a + b$  is the concatenation of strings  $a$  and  $b$ .

### Examples

#### 1. Example 1:

- Input:  $s_1 = \text{"aabcc"}$ ,  $s_2 = \text{"dbbca"}$ ,  $s_3 = \text{"aadbbcbcac"}$
- Output: true
- Explanation: One way to obtain  $s_3$  is:
  - Split  $s_1$  into  $s_1 = \text{"aa"} + \text{"bc"} + \text{"c"}$ , and  $s_2$  into  $s_2 = \text{"dbbc"} + \text{"a"}$ . Interleaving the two splits, we get  $\text{"aa"} + \text{"dbbc"} +$   
Since  $s_3$  can be obtained by interleaving  $s_1$  and  $s_2$ , we return true.

#### 2. Example 2:

- Input:  $s_1 = \text{"aabcc"}$ ,  $s_2 = \text{"dbbca"}$ ,  $s_3 = \text{"aadbbbacc"}$

- Output: false
- Explanation: Notice how it is impossible to interleave s2 with any other string to obtain s3.

### 3. Example 3:

- Input: s1 = "", s2 = "", s3 = ""
- Output: true

## Constraints

- $0 \leq s1.length, s2.length \leq 100$
- $0 \leq s3.length \leq 200$
- s1, s2, and s3 consist of lowercase English letters.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto s1    {string("aabcc")};
8     auto s2    {string("dbbca")};
9     auto s3    {string("aadbcbcac")};
10
11     // trivial case
12     if (s1.size() + s2.size() != s3.size()) return static_cast<bool>(0);
```

```

13
14 // setup
15 vector<vector<bool>> dptable;
16
17 // creating dptable
18 for(int i = 0; i<s1.size()+1; ++i) {dptable.push_back(vector<bool>(s2.size()+1, false));}
19
20 // storing a true at the real end of the table
21 dptable[dptable.size()-1][dptable[0].size()-1] = true;
22
23 // moving back from the end
24 for(int i = dptable.size()-1; i>=0; --i){
25     for(int j = dptable[0].size()-1; j>=0; --j){
26
27         // fetching the character at this position
28         auto s2_char {s2[j]};
29         auto s1_char {s1[i]};
30         auto s3_char {s3[i + j]};
31
32         // checking if we can move right
33         if (s1_char == s3_char) {dptable[i][j] = dptable[i][j] + ((i+1) < dptable.size() ? dptable[i+1][j] : 0);}
34
35         // checking if we can move down
36         if (s2_char == s3_char) {dptable[i][j] = dptable[i][j] + ((j+1) < dptable[0].size() ? dptable[i][j+1] : 0);}
37
38     }
39 }
40
41 // returning the first value
42 cout << format("final-output = {}\n", dptable[0][0]);
43
44 // return
45 return(0);
46
47 }

```

## 98. Validate Binary Search Tree

Given the root of a binary tree, determine if it is a valid binary search tree (BST).

A valid BST is defined as follows:

- The left subtree of a node contains only nodes with keys strictly less than the node's key.
- The right subtree of a node contains only nodes with keys strictly greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

### Constraints

- The number of nodes in the tree is in the range  $[1, 104]$ .
- $-2^{31} \leq \text{Node.val} \leq 2^{31} - 1$

### Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto root    {new TreeNode(2)};
8     root->left    = new TreeNode(1);
9     root->right   = new TreeNode(3);
10
```

```

11 // setup
12 bool validBSTFlag = true;
13 vector<int> nodevalues;
14
15 std::function<void(const TreeNode*)> foo = [&foo, &nodevalues, &validBSTFlag](
16     const TreeNode* root){
17
18     // if false, stop everything
19     if (validBSTFlag == false) {return;}
20
21     // returning
22     if (root == nullptr) {return;}
23
24     // going left
25     if (root->left) {foo(root->left);}
26
27     // adding current
28     if (nodevalues.size() == 0) {nodevalues.push_back(root->val);}
29     else{
30         // check top
31         auto topvalue = nodevalues[nodevalues.size()-1];
32
33         // check if top value is less than or equal
34         if (topvalue >= root->val) {validBSTFlag = false;} // since this is not valid
35         else {nodevalues.push_back(root->val);}
36     }
37
38     // going right
39     if (root->right) {foo(root->right);}
40 };
41
42 // calling function
43 foo(root);
44
45 // returning

```

```
46     cout << format("final-output = {}\n", validBSTFlag);
47
48     // return
49     return(0);
50 }
```

---

## 99. Recover Binary Search Tree

You are given the root of a binary search tree (BST), where the values of exactly two nodes of the tree were swapped by mistake. Recover the tree without changing its structure.

### Examples

#### 1. Example 1:

- Input: root = [1,3,null,null,2]
- Output: [3,1,null,null,2]
- Explanation: 3 cannot be a left child of 1 because 3 > 1. Swapping 1 and 3 makes the BST valid.

#### 2. Example 2:

- Input: root = [3,1,4,null,null,2]
- Output: [2,1,4,null,null,3]
- Explanation: 2 cannot be in the right subtree of 3 because 2 < 3. Swapping 2 and 3 makes the BST valid.

### Constraints

- The number of nodes in the tree is in the range [2, 1000].
- $-2^{31} \leq \text{Node.val} \leq 2^{31} - 1$



## Code

---

```
1 // the function
2 void foo(TreeNode*          root,
3     vector<TreeNode*>&      nodeaddresses,
4     vector<int>&           nodevalues){
5
6     // sending it back
7     if (root == nullptr) return;
8
9     // going through the left
10    foo(root->left, nodeaddresses, nodevalues);
11
12    // adding the current
13    nodeaddresses.push_back( root);
14    nodevalues.push_back(    root->val);
15
16    // going right
17    foo(root->right, nodeaddresses, nodevalues);
18
19    // returning
20    return;
21 }
22
23 int main(){
24
25     // starting timer
26     Timer timer;
27
28     // input- configuration
29     auto root    {new TreeNode(1)};
30     root->left    = new TreeNode(3);
31     root->left->right = new TreeNode(2);
32
33     // setup
```

```
34     vector<TreeNode*> nodeaddresses;  
35     vector<int>         nodevalues;  
36  
37     // calling the function  
38     foo(root, nodeaddresses, nodevalues);  
39  
40     // sort the values and assign it  
41     std::sort(nodevalues.begin(), nodevalues.end());  
42  
43     // writing the values as is  
44     #pragma omp  
45     for(int i = 0; i<nodeaddresses.size(); ++i){  
46         nodeaddresses[i]->val = nodevalues[i];  
47     }  
48  
49     // return  
50     return(0);  
51  
52 }
```

---

## 100. Same Tree

Given the roots of two binary trees *p* and *q*, write a function to check if they are the same or not.

Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

### Examples

#### 1. Example 1:

- Input: *p* = [1,2,3], *q* = [1,2,3]
- Output: true

#### 2. Example 2:

- Input: *p* = [1,2], *q* = [1,null,2]
- Output: false

#### 3. Example 3:

- Input: *p* = [1,2,1], *q* = [1,1,2]
- Output: false

### Constraints

- The number of nodes in both trees is in the range [0, 100].
- $-10^4 \leq \text{Node.val} \leq 10^4$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto p {new TreeNode(1)};
8     p->left  = new TreeNode(2);
9     p->right  = new TreeNode(3);
10
11     auto q {new TreeNode(1)};
12     q->left  = new TreeNode(2);
13     q->right  = new TreeNode(3);
14
15     // setup
16     std::function<bool(TreeNode*, TreeNode*)> fCompareTrees = [&fCompareTrees](
17         TreeNode* p, TreeNode* q){
18
19         // basecase
20         if (p == nullptr && q == nullptr)    return true;
21         if (p == nullptr && q != nullptr)    return false;
22         if (p != nullptr && q == nullptr)    return false;
23         if (p->val != q->val)                  return false;
24
25         // going through the next set of branches
26         return (fCompareTrees(p->left, q->left) && fCompareTrees(p->right, q->right));
27     };
28
29     // going through the two nodes
30     bool finalOutput = fCompareTrees(p, q);
31
32     // returning final output
33     cout << format("final-output = {}\n", finalOutput);
```

```
34
35     // return
36     return(0);
37
38 }
```

---

## 101. Symmetric Tree

Given the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).

### Examples

#### 1. Example 1:

- Input: root = [1,2,2,3,4,4,3]
- Output: true

#### 2. Example 2:

- Input: root = [1,2,2,null,3,null,3]
- Output: false

### Constraints

- The number of nodes in the tree is in the range [1, 1000].
- $-100 \leq \text{Node.val} \leq 100$

### Code

---

```

1  int main(){
2
3      // starting timer
4      Timer timer;
5
6      // input- configuration
7      auto root      {new TreeNode(1)};
8      root->left      = new TreeNode(2);
9      root->right     = new TreeNode(2);
10     root->left->left  = new TreeNode(3);
11     root->left->right = new TreeNode(4);
12     root->right->left = new TreeNode(4);
13     root->right->right = new TreeNode(3);
14
15     // setup
16     std::function<bool(TreeNode*, TreeNode*)> dfs = [&dfs](
17         TreeNode* leftNode,
18         TreeNode* rightNode){
19
20         // printing the base-cases
21         if (leftNode == nullptr && rightNode == nullptr) return true;
22         if (leftNode == nullptr || rightNode == nullptr) return false;
23
24         // calling the children
25         bool finaloutput = (leftNode->val == rightNode->val)    && \
26                             dfs(leftNode->left, rightNode->right) && \
27                             dfs(leftNode->right, rightNode->left);
28
29         // returning the value
30         return finaloutput;
31     };
32
33     // running
34     cout << format("final-output = {}\n", dfs(root->left, root->right));

```

```
35
36 // return
37 return(0);
38 }
```

---



## 102. Binary Tree Level Order Traversal

Given the root of a binary tree, return the level order traversal of its nodes' values. (i.e., from left to right, level by level).

### Constraints

- The number of nodes in the tree is in the range [0, 2000].
- $-1000 \leq \text{Node.val} \leq 1000$

### Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto root      {new TreeNode(3)};
8     root->left      = new TreeNode(9);
9     root->right     = new TreeNode(20);
10    root->right->left = new TreeNode(15);
11    root->right->right = new TreeNode(7);
12
13    // setup
14    std::vector< std::vector<int> > finalOutput;
15    std::function<void(const TreeNode*, int)> f = [&f, &finalOutput](
16        const TreeNode* root,
17        int level){
18
19        // base-case
```

```

20     if (root == nullptr) return;
21
22     // finding current-level
23     level += 1;
24
25     // increasing size of the thing
26     while (finalOutput.size() < level+1)
27         finalOutput.push_back(std::vector<int>());
28
29     // appending to value
30     if (finalOutput.size() < level+1)
31         finalOutput[level] = std::vector<int>(root->val);
32     else
33         finalOutput[level].push_back(root->val);
34
35     // calling
36     f(root->left, level);
37     f(root->right, level);
38
39     // returning return;
40     return;
41
42 };
43
44 // going through the tree
45 f(root, -1);
46
47 // printing the final output
48 cout << format("finalOutput = {}\n", finalOutput);
49
50 // return
51 return(0);
52
53 }

```

---

## 103. Binary Tree Zigzag Level Order Traversal

Given the root of a binary tree, return the zigzag level order traversal of its nodes' values. (i.e., from left to right, then right to left for the next level and alternate between).

### Examples

#### 1. Example 1:

- Input: root = [3,9,20,null,null,15,7]
- Output: [[3],[20,9],[15,7]]

#### 2. Example 2:

- Input: root = [1]
- Output: [[1]]

#### 3. Example 3:

- Input: root = []
- Output: []

### Constraints

- The number of nodes in the tree is in the range [0, 2000].
- $-100 \leq \text{Node.val} \leq 100$

## Code

---

```
1 void f(TreeNode* root,
2     std::vector< std::vector<int> >& finalOutput,
3     int level)
4 {
5
6     // base-case
7     if (root == nullptr) return;
8
9     // finding current-level
10    level += 1;
11
12    // increasing size of the thing
13    while (finalOutput.size() < level+1) {finalOutput.push_back(std::vector<int>());}
14
15    // appending to value
16    if (finalOutput.size() < level+1 ) {finalOutput[level] = std::vector<int>(root->val);}
17    else{
18        // appending based on level
19        if (level%2 == 0)
20            finalOutput[level].push_back(root->val);
21        else
22        {
23            std::vector<int> temp {root->val};
24            for(auto x: finalOutput[level]) {temp.push_back(x);}
25            finalOutput[level] = temp;
26        }
27    }
28
29    // calling
30    f(root->left, finalOutput, level);
31    f(root->right, finalOutput, level);
32
33    // returning return;
```

```
34     return;
35 }
36 int main(){
37
38     // starting timer
39     Timer timer;
40
41     // input- configuration
42     auto root      {new TreeNode(3)};
43     root->left      = new TreeNode(9);
44     root->right      = new TreeNode(20);
45     root->right->left = new TreeNode(15);
46     root->right->right = new TreeNode(7);
47
48     // setup
49     std::vector< std::vector<int> > finalOutput;
50
51     // going through the tree
52     f(root, finalOutput, -1);
53
54     // sending back final output
55     cout << format("final-output = {}\n", finalOutput);
56
57     // return
58     return(0);
59 }
60
```

---

## 104. Maximum Depth of Binary Tree

Given the root of a binary tree, return its maximum depth.

A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

### Examples

#### 1. Example 1:

- Input: root = [3,9,20,null,null,15,7]
- Output: 3

#### 2. Example 2:

- Input: root = [1,null,2]
- Output: 2

### Constraints

- The number of nodes in the tree is in the range  $[0, 10^4]$ .
- $-100 \leq \text{Node.val} \leq 100$

### Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto root      {new TreeNode(3)};
8     root->left      = new TreeNode(9);
9     root->right      = new TreeNode(20);
10    root->right->left = new TreeNode(15);
11    root->right->right = new TreeNode(7);
12
13    // setup
14    auto maxcount {0};
15    auto count    {0};
16
17    // lambda - function
18    std::function<void(const TreeNode*, int)> fTraverse = [&fTraverse,
19                                                         &maxcount](
20        const TreeNode* root,
21        int count){
22
23        // checking if null
24        if (root == nullptr) return;
25
26        ++count;                                // updating length
27        maxcount = max(maxcount, count);        // checking maxlength
28
29        fTraverse(root->left, count);           // checking if we can go left
30        fTraverse(root->right, count);          // checking if we can go right
31
32        // returning
33        return;
34    };
```

```
35
36 // calling the function
37 fTraverse(root, count);
38
39 // returning
40 cout << format("final-output = {}\n", maxcount);
41
42 // return
43 return(0);
44
45 }
```

---



## 105. Construct Binary Tree from Preorder and Inorder Traversal

Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return the binary tree.

### Examples

#### 1. Example 1:

- Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
- Output: [3,9,20,null,null,15,7]

#### 2. Example 2:

- Input: preorder = [-1], inorder = [-1]
- Output: [-1]

### Constraints

- $1 \leq \text{preorder.length} \leq 3000$
- $\text{inorder.length} == \text{preorder.length}$
- $-3000 \leq \text{preorder}[i], \text{inorder}[i] \leq 3000$
- preorder and inorder consist of unique values.
- Each value of inorder also appears in preorder.

- preorder is guaranteed to be the preorder traversal of the tree.
- inorder is guaranteed to be the inorder traversal of the tree.

## Code

---

```
1  TreeNode* foo(vector<int> preorder, vector<int> inorder)
2  {
3
4      // returning for trivial case
5      if (preorder.size() == 0 || inorder.size() == 0) {return static_cast<TreeNode*>(nullptr);}
6
7      // setup
8      auto& rootvalue    {preorder[0]};
9      auto root          {new TreeNode(preorder[0])};
10
11     // sending it back for trivial case
12     if (preorder.size() == 1 && inorder.size() == 1) {return root;}
13
14     // finding root-position
15     auto rootposition = std::find(inorder.begin(),
16                                  inorder.end(),
17                                  rootvalue);
18
19     // creating left-subtree
20     auto leftinorder {vector<int>(inorder.begin(), rootposition)};
21     auto rightinorder {vector<int>(rootposition+1, inorder.end())};
22
23     // creating subset of preorder
24     auto leftpreorder {vector<int>(preorder.begin()+1,
25                                    preorder.begin() + leftinorder.size() + 1)};
26     auto rightpreorder {vector<int>(preorder.begin() + leftinorder.size()+1,
27                                     preorder.end())};
```

```
28
29 // calling the function on left and right
30 root->left    = foo(leftpreorder, leftinorder);
31 root->right   = foo(rightpreorder, rightinorder);
32
33 // returning the
34 return root;
35
36 }
37 int main(){
38
39 // starting timer
40 Timer timer;
41
42 // input- configuration
43 auto preorder {vector<int>{3,9,20,15,7}};
44 auto inorder  {vector<int>{9,3,15,20,7}};
45
46 // manually building the tree
47 auto root {foo(preorder, inorder)};
48
49 // return
50 return(0);
51
52 }
```

---

## 107. Binary Tree Level Order Traversal II

Given the root of a binary tree, return the bottom-up level order traversal of its nodes' values. (i.e., from left to right, level by level from leaf to root).

### Examples

#### 1. Example 1:

- Input: root = [3,9,20,null,null,15,7]
- Output: [[15,7],[9,20],[3]]

#### 2. Example 2:

- Input: root = [1]
- Output: [[1]]

#### 3. Example 3:

- Input: root = []
- Output: []

### Constraints

- The number of nodes in the tree is in the range [0, 2000].
- $-1000 \leq \text{Node.val} \leq 1000$

## Code

---

```
1 void foo(TreeNode* root,
2     map<int, vector<int>, std::greater<int>>& levelToMembers,
3     int currlevel){
4
5     // sending it back
6     if (root==nullptr) return;
7
8     // incrementing currlevel
9     ++currlevel;
10
11    // moving left
12    foo(root->left, levelToMembers, currlevel);
13
14    // checking if the current-level is present
15    if (levelToMembers.find(currlevel) == levelToMembers.end())
16        levelToMembers[currlevel] = vector<int>({root->val});
17    else
18        levelToMembers[currlevel].push_back(root->val);
19
20    // moving right
21    foo(root->right, levelToMembers, currlevel);
22
23    // returning
24    return;
25 }
26
27 int main(){
28
29    // starting timer
30    Timer timer;
31
32    // input- configuration
33    auto root {new TreeNode()};
```

```

34 root->left      = new TreeNode(9);
35 root->right     = new TreeNode(20);
36 root->right->left = new TreeNode(15);
37 root->right->right = new TreeNode(7);
38
39 // setup
40 map<int, vector<int>, std::greater<int>> levelToMembers;
41 vector<vector<int>> finalOutput;
42
43 // running the function
44 foo(root, levelToMembers, -1);
45
46 // printing the values
47 #pragma omp parallel for
48 for(auto x: levelToMembers) {finalOutput.push_back(x.second);}
49
50 // returning the final output
51 cout << format("final-output = {}\n", finalOutput);
52
53 // return
54 return(0);
55
56 }

```

---

## 108. Convert Sorted Array to Binary Search Tree

Given an integer array `nums` where the elements are sorted in ascending order, convert it to a height-balanced binary search tree.

### Constraints

- $1 \leq \text{nums.length} \leq 10^4$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- `nums` is sorted in a strictly increasing order.

### Code

---

```
1 void foo(TreeNode*      parent,
2     int                parentdirection,
3     vector<int>&        nums,
4     int                left,
5     int                right)
6 {
7     // sending it back if right is less than left
8     if (left > right) return;
9
10    // finding mid point
11    int mid = (left + right)/2;
12
13    // creating mid-node
14    TreeNode* midnode = new TreeNode(nums[mid]);
15
16    // connecting current to parent node
```

```

17     if (parentdirection == -1)        {parent->right = midnode;}
18     else if (parentdirection == 1)    {parent->left  = midnode;}
19
20     // going left
21     foo(midnode, 1, nums, left, mid-1);
22
23     // going right
24     foo(midnode, -1, nums, mid+1, right);
25
26     // returning
27     return;
28
29 }
30 int main(){
31
32     // starting timer
33     Timer timer;
34
35     // input- configuration
36     vector<int> nums  {-10,-3,0,5,9};
37
38     // setup
39     TreeNode* dummy = new TreeNode(-1);
40
41     // calling the function
42     foo(dummy, -1, nums, 0, nums.size()-1);
43
44     // return
45     return(0);
46
47 }

```

---



## 109. Convert Sorted List to Binary Search Tree

Given the head of a singly linked list where elements are sorted in ascending order, convert it to a height-balanced binary search tree.

### Constraints

- The number of nodes in head is in the range  $[0, 2 * 10^4]$ .
- $-105 \leq \text{Node.val} \leq 105$

### Code

---

```
1 void foo(TreeNode*      parent,
2     int                parentdirection,
3     vector<int>&        nums,
4     int                left,
5     int                right)
6 {
7     // sending it back if right is less than left
8     if (left > right) return;
9
10    // finding mid point
11    int mid = (left + right)/2;
12
13    // creating mid-node
14    TreeNode* midnode = new TreeNode(nums[mid]);
15
16    // connecting current to parent node
17    if (parentdirection == -1) parent->right = midnode;
18    else if (parentdirection == 1) parent->left = midnode;
19
```

```

20 // going left
21 foo(midnode, 1, nums, left, mid-1);
22
23 // going right
24 foo(midnode, -1, nums, mid+1, right);
25
26 // returning
27 return;
28
29 }
30
31 int main(){
32
33 // starting timer
34 Timer timer;
35
36 // input- configuration
37 auto head {new ListNode(-10)};
38 head->next = new ListNode(-3);
39 head->next->next = new ListNode(0);
40 head->next->next->next = new ListNode(5);
41 head->next->next->next->next = new ListNode(9);
42
43 // sending it back if the list is empty
44 if (head == nullptr) {cout << format("final-output = \n");}
45
46 // making a vector out of the whole list
47 auto traveller = head;
48 vector<int> nums;
49 while(traveller){
50     nums.push_back(traveller->val);
51     traveller = traveller->next;
52 }
53
54 // calling the previous function into this

```

```
55     TreeNode* dummy = new TreeNode(-1);
56
57     // calling the function
58     foo(dummy, -1, nums, 0, nums.size()-1);
59
60     // return
61     return(0);
62
63 }
```

---

## 110. Balanced Binary Tree

Given a binary tree, determine if it is height-balanced.

### Constraints

- The number of nodes in the tree is in the range [0, 5000].
- $-10^4 \leq \text{Node.val} \leq 10^4$

### Code

---

```
1 int foo(const TreeNode* root,
2         int currlevel,
3         bool& aightstop){
4
5     // sending it back if null
6     if (root == nullptr) {return 0;}
7
8     // checking if flag is up
9     if (aightstop == true) {return -1;}
10
11    // incrementing levle
12    ++currlevel;
13
14    // going left and right
15    auto leftdepth  {foo(root->left, currlevel, aightstop)};
16    auto rightdepth {foo(root->right, currlevel, aightstop)};
17
18    auto diff = std::abs(rightdepth - leftdepth);
19
```

```

20 // checking if depth is the same
21 if (diff>1)          {aightstop = true;}
22
23 // returning the difference
24 return (std::max(leftdepth, rightdepth)+1);
25
26 }
27
28 int main(){
29
30     // starting timer
31     Timer timer;
32
33     // input- configuration
34     auto root      {new TreeNode(3)};
35     root->left      = new TreeNode(9);
36     root->right     = new TreeNode(20);
37     root->right->left = new TreeNode(15);
38     root->right->right = new TreeNode(7);
39
40     // trivial case
41     if (root == nullptr) {cout << format("final-output = {}\n", true);}
42
43     // setup
44     int    currlevel {-1};
45     bool   aightstop {false};
46
47     // running the function
48     foo(root, currlevel, aightstop);
49
50     // returning the status
51     cout << format("final-output = {}\n", !aightstop);
52
53     // return
54     return(0);

```



## 111. Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Note: A leaf is a node with no children.

### Constraints

- The number of nodes in the tree is in the range  $[0, 10^5]$ .
- $-1000 \leq \text{Node.val} \leq 1000$

### Code

---

```
1 void foo(TreeNode* root,
2     int         currlevel,
3     int&        minlevel)
4 {
5     // sending it back
6     if (root == nullptr) {return;}
7     if (minlevel == 2)   {return;}
8
9     // incrementing level
10    ++currlevel;
11
12    // checking if child
13    if (root->left == nullptr && root->right == nullptr){
14        minlevel = std::min(minlevel, currlevel);
15        return;
```

```

16     }
17
18     // calling functions
19     foo(root->left,  currlevel, minlevel);
20     foo(root->right,  currlevel, minlevel);
21
22     // returning
23     return;
24 }
25
26 int main(){
27
28     // starting timer
29     Timer timer;
30
31     // input- configuration
32     auto root      {new TreeNode(3)};
33     root->left      = new TreeNode(9);
34     root->right     = new TreeNode(20);
35     root->right->left = new TreeNode(15);
36     root->right->right = new TreeNode(7);
37
38     // trivial case
39     if (root == nullptr) {cout << format("final-output = {}\n", 0);}
40
41     // setup
42     auto currlevel {0};
43     auto minlevel  {std::numeric_limits<int>::max()};
44
45     // calling the function
46     foo(root, currlevel, minlevel);
47
48     // returning the minlevel
49     cout << format("final-output = {}\n", minlevel);
50

```



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

---

## 112. Path Sum

Given the root of a binary tree and an integer targetSum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals targetSum.

### Constraints

- The number of nodes in the tree is in the range [0, 5000].
- $-1000 \leq \text{Node.val} \leq 1000$
- $-1000 \leq \text{targetSum} \leq 1000$

### Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto root      {new TreeNode(5)};
8
9     root->left      = new TreeNode(4);
10    root->right      = new TreeNode(8);
11
12    root->left->left   = new TreeNode(11);
13    root->right->left  = new TreeNode(13);
14    root->right->right = new TreeNode(4);
15
```

```

16 root->left->left->left    = new TreeNode(7);
17 root->left->left->right   = new TreeNode(2);
18 root->right->right->right = new TreeNode(1);
19
20 auto targetSum    {22};
21
22 // setup
23 auto runningsum   {0};
24 auto binaryflag   {false};
25 std::function<void(const TreeNode*, int)> fRunSum = [&fRunSum,
26                                     &targetSum,
27                                     &binaryflag](
28     const TreeNode* root,
29     int runningsum
30 ){
31
32     // sending it back
33     if (root == nullptr) return;
34     if (binaryflag == true) return;
35
36     // adding current val to running sum
37     runningsum += root->val;
38
39     // checking if there are children
40     if (root->left || root->right){
41         if (root->left) fRunSum(root->left, runningsum);
42         if (root->right) fRunSum(root->right, runningsum);
43     }
44     else{
45         // in this case, we check the running sum
46         if (runningsum == targetSum){ binaryflag = true;}
47     }
48
49     // returning
50     return;

```

```
51     };
52
53     // running
54     fRunSum(root, runningsum);
55
56     // printing
57     cout << format("final-output = {}\n", binaryflag);
58
59     // return
60     return(0);
61
62 }
```

---

## 113. Path Sum II

Given the root of a binary tree and an integer targetSum, return all root-to-leaf paths where the sum of the node values in the path equals targetSum. Each path should be returned as a list of the node values, not node references.

A root-to-leaf path is a path starting from the root and ending at any leaf node. A leaf is a node with no children.

### Constraints

- The number of nodes in the tree is in the range [0, 5000].
- $-1000 \leq \text{Node.val} \leq 1000$
- $-1000 \leq \text{targetSum} \leq 1000$

### Code

---

```
1 void foo(TreeNode*      root,
2     const int          targetSum,
3     int                runningsum,
4     vector<int>         runningvector,
5     vector<vector<int>>& finalOutput)
6 {
7     // sending it back
8     if (root==nullptr) return;
9
10    // adding current-value to running vector
11    runningvector.push_back(root->val);
12    runningsum += root->val;
13
```

```

14 // checking if current-value has reached target sum
15 if (runningsum == targetSum &&
16     root->left == nullptr &&
17     root->right == nullptr){
18     finalOutput.push_back(runningvector); return;
19 }
20
21 // going down left and right
22 foo(root->left, targetSum, runningsum, runningvector, finalOutput);
23 foo(root->right, targetSum, runningsum, runningvector, finalOutput);
24
25 // returning
26 return;
27 }
28
29 int main(){
30
31     // starting timer
32     Timer timer;
33
34     // input- configuration
35     auto root    {new TreeNode(1)};
36     root->left    = new TreeNode(2);
37     root->right   = new TreeNode(3);
38     auto targetSum {5};
39
40     // trivial case
41     if (root == nullptr) {
42         cout << format("final-output = {}\n", vector<vector<int>>());
43         return 0;
44     }
45
46     // setup
47     vector<vector<int>> finalOutput;
48     auto runningsum {0};

```

```
49     vector<int> runningvector;  
50  
51     // calling the function  
52     foo(root, targetSum, runningsum, runningvector, finalOutput);  
53  
54     // returning  
55     cout << format("final-output = {}\n", finalOutput);  
56  
57     // return  
58     return(0);  
59  
60 }
```

---

## 114. Flatten Binary Tree to Linked List

Given the root of a binary tree, flatten the tree into a “linked list”:

- The “linked list” should use the same `TreeNode` class where the right child pointer points to the next node in the list and the left child pointer is always null.
- The “linked list” should be in the same order as a pre-order traversal of the binary tree.

### Examples

#### 1. Example 1:

- Input: root = [1,2,5,3,4,null,6]
- Output: [1,null,2,null,3,null,4,null,5,null,6]

#### 2. Example 2:

- Input: root = []
- Output: []

#### 3. Example 3:

- Input: root = [0]
- Output: [0]



## Constraints

- The number of nodes in the tree is in the range [0, 2000].
- $-100 \leq \text{Node.val} \leq 100$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto root      {new TreeNode(1)};
8     root->left      = new TreeNode(2);
9     root->right     = new TreeNode(5);
10    root->left->left  = new TreeNode(3);
11    root->left->right = new TreeNode(4);
12    root->right->right = new TreeNode(6);
13
14    // recursion lambda
15    auto foo = [&foo](TreeNode*      root,
16                vector<TreeNode*>& nodeaddresses){
17
18        // sending it back
19        if (root == nullptr) return;
20
21        // adding address
22        nodeaddresses.push_back(root);
23
24        // going down left
25        if (root->left) foo(root->left, nodeaddresses);
```

```

26     if (root->right) foo(root->right, nodeaddresses);
27
28     // returning
29     return;
30 };
31
32 // trivial case
33 if (root == nullptr) return 0;
34
35 // setup
36 vector<TreeNode*> nodeaddresses;
37
38 // fill registers
39 foo(root, nodeaddresses);
40
41 // going through the addresses and reconnecting it
42 for(int i = 0; i<nodeaddresses.size()-1; ++i){
43     nodeaddresses[i]->left = nullptr;
44     nodeaddresses[i]->right = nodeaddresses[i+1];
45 }
46
47 // taking care of the last
48 nodeaddresses[nodeaddresses.size()-1]->left = nullptr;
49 nodeaddresses[nodeaddresses.size()-1]->right = nullptr;
50
51 // return
52 return(0);
53
54 }

```

---

## 116. Populating Next Right Pointers in Each Node

You are given a perfect binary tree where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

---

```
1 struct Node {  
2     int val;  
3     Node *left;  
4     Node *right;  
5     Node *next;  
6 }
```

---

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

### Constraints

- The number of nodes in the tree is in the range  $[0, 2^{12} - 1]$ .
- $-1000 \leq \text{Node.val} \leq 1000$

### Code

---

```
1 // Definition for a Node.  
2 class Node {  
3 public:  
4     int val;  
5     Node* left;
```

```

6     Node* right;
7     Node* next;
8
9     Node() : val(0), left(NULL), right(NULL), next(NULL) {}
10
11    Node(int _val) : val(_val), left(NULL), right(NULL), next(NULL) {}
12
13    Node(int _val, Node* _left, Node* _right, Node* _next)
14        : val(_val), left(_left), right(_right), next(_next) {}
15 };
16
17 void foo(deque<Node*> parents)
18 {
19     // sendint it back if empty
20     if (parents.size() < 1) {return;}
21
22     // adding a nullptr to the parents list as demarcater
23     parents.push_back(static_cast<Node*>(nullptr));
24
25     // adding children to parents list until we hit the nullptr
26     while(parents[1] != static_cast<Node*>(nullptr)){
27
28         // connecting the top
29         parents[0]->next = parents[1];
30
31         // adding the children to the pipeline
32         if(parents[0]->left) {parents.push_back(parents[0]->left);}
33         if(parents[0]->right) {parents.push_back(parents[0]->right);}
34
35         // popping the top
36         parents.pop_front();
37     }
38
39     // adding the children to the pipeline
40     if(parents[0]->left) {parents.push_back(parents[0]->left);}

```

```

41     if(parents[0]->right) {parents.push_back(parents[0]->right);}
42
43     // popping the top two parents
44     parents.pop_front();
45     parents.pop_front();
46
47     // calling the function again
48     foo(parents);
49
50     // returning
51     return;
52 }
53
54 int main(){
55
56     // starting timer
57     Timer timer;
58
59     // input- configuration
60     auto root          {new Node(1)};
61     root->left          = new Node(2);
62     root->right         = new Node(3);
63     root->left->left     = new Node(4);
64     root->left->right    = new Node(5);
65     root->right->left    = new Node(6);
66     root->right->right   = new Node(7);
67
68     // trivial case
69     if (root == nullptr) {return 0;}
70
71     // setup
72     deque<Node*> parents {root};
73
74     // calling the function
75     foo(parents);

```

```
76
77     // return
78     return(0);
79
80 }
```

---

## 117. Populating Next Right Pointers in Each Node II

Given a binary tree of type

---

```
1 struct Node {  
2     int val;  
3     Node *left;  
4     Node *right;  
5     Node *next;  
6 }
```

---

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL. Initially, all next pointers are set to NULL.

### Examples

#### 1. Example 1:

- Input: root = [1,2,3,4,5,null,7]
- Output: [1,#,2,3,#,4,5,7,#]
- Explanation: Given the above binary tree (Figure A), your function should populate each next pointer to point to its next right node, just like in Figure B. The serialized output is in level order as connected by the next pointers, with # signifying the end of each level.

#### 2. Example 2:

- Input: root = []
- Output: []

## Constraints

- The number of nodes in the tree is in the range [0, 6000].
- $-100 \leq \text{Node.val} \leq 100$

## Code

---

```
1 class Node {
2 public:
3     int val;
4     Node* left;
5     Node* right;
6     Node* next;
7
8     Node() : val(0), left(NULL), right(NULL), next(NULL) {}
9     Node(int _val) : val(_val), left(NULL), right(NULL), next(NULL) {}
10    Node(int _val, Node* _left, Node* _right, Node* _next)
11        : val(_val), left(_left), right(_right), next(_next) {}
12 };
13
14
15 int main(){
16
17     // starting timer
18     Timer timer;
19
20     // input- configuration
21     auto root    {new Node(1)};
22     root->left    = new Node(2);
23     root->right   = new Node(3);
24     root->left->left = new Node(4);
25     root->left->right = new Node(5);
```



```

26 root->right->right = new Node(7);
27
28 // running
29 if (root == nullptr) {cout << format("done!\n"); return 0;}
30
31 // calling the function
32 int count = -1;
33 vector< vector<Node*> > addressvectors;
34
35 // calling the efunction
36 foo(root, count, addressvectors);
37
38 // building the connections
39 for(auto x: addressvectors){
40     int i = 0;
41     for(; i<x.size()-1; ++i)
42         if (x[i] != nullptr) {x[i]->next = x[i+1];}           // look to the right
43
44     // ensuring that the last pointer points to the right most
45     x[i]->next = nullptr;
46 }
47
48 // completion
49 cout << format("done!\n");
50
51 // return
52 return(0);
53
54 }

```

---

## 118. Pascal's Triangle

Given an integer numRows, return the first numRows of Pascal's triangle.

In Pascal's triangle, each number is the sum of the two numbers directly above it as shown:

### Examples

#### 1. Example 1:

- Input: numRows = 5
- Output: `[[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]`

#### 2. Example 2:

- Input: numRows = 1
- Output: `[[1]]`

### Constraints

- $1 \leq \text{numRows} \leq 30$

### Code

---

```
1 void foo(vector<vector<int>>& finalOutput,  
2     int currlevel,  
3     const int terminatinglevel){
```

```

4 // sending it back if we overshot
5 ++currlevel;
6 if (currlevel > terminatinglevel) {return;}
7
8 // creating the final output
9 vector<int> tempoutput (currlevel, 0);
10 tempoutput[0] = 1;
11 tempoutput[tempoutput.size()-1] = 1;
12
13 // fetching the layer above
14 auto& layerabove = finalOutput[finalOutput.size()-1];
15
16 // producing the output for each level
17 for(int i = 1; i<currlevel-1; ++i) {tempoutput[i] = layerabove[i-1] + layerabove[i];}
18
19 // pushing to the final output
20 finalOutput.push_back(tempoutput);
21
22 // recursion
23 foo(finalOutput, currlevel, terminatinglevel);
24
25 // returning
26 return;
27
28 }
29
30 int main(){
31
32 // starting timer
33 Timer timer;
34
35 // input- configuration
36 auto numRows {5};
37
38 // setup

```

```
39     vector<vector<int>> finalOutput;  
40     finalOutput.push_back(vector<int>({1}));  
41     int currlevel {1};  
42  
43     // calling the function  
44     foo(finalOutput, currlevel, numRows);  
45  
46     // returning the final output  
47     cout << format("finalOutput = \n");  
48     fPrintMatrix(finalOutput);  
49  
50     // return  
51     return(0);  
52  
53 }
```

---

## 119. Pascal's Triangle II

Given an integer rowIndex, return the rowIndexth (0-indexed) row of the Pascal's triangle.

In Pascal's triangle, each number is the sum of the two numbers directly above it as shown:

### Examples

1. **Example 1:**

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

2. **Example 2:**

- Input: rowIndex = 0
- Output: [1]

3. **Example 3:**

- Input: rowIndex = 1
- Output: [1,1]

### Constraints

- $0 \leq \text{rowIndex} \leq 33$

## Code

---

```
1 void foo(vector<vector<int>>& finalOutput,  
2         int currlevel,  
3         const int terminatinglevel){  
4  
5     // sending it back if we overshoot  
6     ++currlevel;  
7     if (currlevel > terminatinglevel) {return;}  
8  
9     // creating the final output  
10    vector<int> tempoutput (currlevel, 0);  
11    tempoutput[0] = 1;  
12    tempoutput[tempoutput.size()-1] = 1;  
13  
14    // fetching the layer above  
15    auto& layerabove {finalOutput[finalOutput.size()-1]};  
16  
17    // producing the output for each level  
18    for(int i = 1; i<currlevel-1; ++i) {tempoutput[i] = layerabove[i-1] + layerabove[i];}  
19  
20    // pushing to the final output  
21    finalOutput.push_back(tempoutput);  
22  
23    // recursion  
24    foo(finalOutput, currlevel, terminatinglevel);  
25  
26    // returning  
27    return;  
28 }  
29  
30 int main(){  
31  
32     // starting timer  
33     Timer timer;
```

```
34
35 // input- configuration
36 auto rowIndex {3};
37
38 // setup
39 vector<vector<int>> finalOutput;
40 finalOutput.push_back(vector<int>({1}));
41 int currlevel {1};
42
43 // calling the function
44 foo(finalOutput, currlevel, rowIndex+1);
45
46 // returnign the final output
47 cout << format("final-output = {}\n", finalOutput[finalOutput.size()-1]);
48
49 // return
50 return(0);
51
52 }
```

---

## 120. Triangle

Given a triangle array, return the minimum path sum from top to bottom.

For each step, you may move to an adjacent number of the row below. More formally, if you are on index  $i$  on the current row, you may move to either index  $i$  or index  $i + 1$  on the next row.

### Examples

#### 1. Example 1:

- Input: `triangle = [[2],[3,4],[6,5,7],[4,1,8,3]]`
- Output: 11
- Explanation: The triangle looks like:
  - The minimum path sum from top to bottom is  $2 + 3 + 5 + 1 = 11$  (underlined above).

#### 2. Example 2:

- Input: `triangle = [[-10]]`
- Output: -10

### Constraints

- $1 \leq \text{triangle.length} \leq 200$
- `triangle[0].length == 1`
- `triangle[i].length == triangle[i - 1].length + 1`



- $-10^4 \leq \text{triangle}[i][j] \leq 10^4$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     vector<vector<int>> triangle = {
8         {2},
9         {3,4},
10        {6,5,7},
11        {4,1,8,3}
12    };
13
14    // setup - input
15    vector<int> row = triangle[triangle.size()-1];
16
17    // filling up the values
18    for(int i = triangle.size()-2; i>=0; --i){
19
20        // going through the rows
21        for(int j = 0; j<=i; ++j){
22
23            auto currentvalue {triangle[i][j]};           // fetching current value
24            auto smallerchild {std::min(row[j], row[j+1])}; // finding the smaller of the two children
25            row[j] = currentvalue + smallerchild;          // adding to the parent and storing to the new array
26
27        }
28    }
29}
```

```
30 // printign the final output
31 cout << format("final-output = {}\n", row[0]);
32
33 // return
34 return(0);
35
36 }
```

---

## 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 }
```

---

## 124. Binary Tree Maximum Path Sum

A path in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence at most once. Note that the path does not need to pass through the root.

The path sum of a path is the sum of the node's values in the path.

Given the root of a binary tree, return the maximum path sum of any non-empty path.

### Constraints

- The number of nodes in the tree is in the range  $[1, 3 * 10^4]$ .
- $-1000 \leq \text{Node.val} \leq 1000$

### Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto root    {new TreeNode(-10)};
8     root->left    = new TreeNode(9);
9     root->right   = new TreeNode(20);
10    root->right->left = new TreeNode(15);
11    root->right->right = new TreeNode(7);
12
13
14    // setup
```



```

15 auto finalresult {std::numeric_limits<int>::min()};
16
17 // creating lambda
18 std::function<int(const TreeNode*)> fFindMaxPathSum = [&fFindMaxPathSum,
19                                                     &finalresult](
20     const TreeNode* root){
21
22     // checking if leaf-node
23     if(root->left == nullptr && root->right == nullptr){
24
25         // checking the final-result with the leaf-node value
26         finalresult = finalresult > root->val ? finalresult : root->val;
27
28         // the potential from leaf-node
29         return root->val;
30     }
31     else{
32
33         // creating potentials-vector
34         vector<int> potentials;
35         if (root->left) {potentials.emplace_back(fFindMaxPathSum(root->left));}
36         if (root->right) {potentials.emplace_back(fFindMaxPathSum(root->right));}
37
38         // calculating sum of left and right
39         auto leftplusright {0};
40         for(int i = 0; i<potentials.size(); ++i) {leftplusright += potentials[i];}
41
42         // calculating sum of path curr with either paths
43         for(int i = 0; i<potentials.size(); ++i) {potentials[i] += root->val;}
44
45         // adding curr-value alone as a candidate
46         potentials.push_back(root->val);
47
48         // sending the maximum-value back
49         const auto maxelement = *(std::max_element(potentials.begin(), potentials.end()));

```

```

50
51 // path from left-curr-right
52 potentials.push_back(leftplusright + root->val);
53
54 // auto temp = *(std::max_element(potentials.begin(), potentials.end()));
55 const auto comparativemax = *(std::max_element(potentials.begin(), potentials.end()));
56
57 // finalresult = std::max(finalresult, maxelement);
58 finalresult = finalresult > comparativemax ? finalresult : comparativemax;
59
60 // returning the max-potential from here
61 return maxelement;
62 }
63 };
64
65
66 // calculating the final-result
67 fFindMaxPathSum(root);
68
69 // printing
70 cout << format("final-result = {}\n", finalresult);
71
72 // return
73 return(0);
74
75 }

```

---

## 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     }
```

```
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 }
```

---

## 127. Word Ladder

A transformation sequence from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord`  $\rightarrow$  `s1`  $\rightarrow$  `s2`  $\rightarrow$  ...  $\rightarrow$  `sk` such that:

- Every adjacent pair of words differs by a single letter.
- Every `si` for  $1 \leq i \leq k$  is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- `sk == endWord`

Given two words, `beginWord` and `endWord`, and a dictionary `wordList`, return the number of words in the shortest transformation sequence from `beginWord` to `endWord`, or 0 if no such sequence exists.

### Examples

#### 1. Example 1:

- Input: `beginWord` = "hit", `endWord` = "cog", `wordList` = ["hot","dot","dog","lot","log","cog"]
- Output: 5
- Explanation: One shortest transformation sequence is "hit"  $\rightarrow$  "hot"  $\rightarrow$  "dot"  $\rightarrow$  "dog"  $\rightarrow$  "cog", which is 5 words long.

#### 2. Example 2:

- Input: `beginWord` = "hit", `endWord` = "cog", `wordList` = ["hot","dot","dog","lot","log"]
- Output: 0
- Explanation: The `endWord` "cog" is not in `wordList`, therefore there is no valid transformation sequence.

## Constraints

- $1 \leq \text{beginWord.length} \leq 10$
- $\text{endWord.length} == \text{beginWord.length}$
- $1 \leq \text{wordList.length} \leq 5000$
- $\text{wordList}[i].\text{length} == \text{beginWord.length}$
- $\text{beginWord}$ ,  $\text{endWord}$ , and  $\text{wordList}[i]$  consist of lowercase English letters.
- $\text{beginWord} \neq \text{endWord}$
- All the words in  $\text{wordList}$  are unique.

## Code

---

```
1 // main-file =====
2 int main(){
3
4     // starting timer
5     Timer timer;
6
7     // input- configuration
8     string beginWord    {"hit"};
9     string endWord      {"cog"};
10    vector<string> wordList {
11        "hot","dot","dog","lot","log"
12    };
13
14    // returning error if endword is not in wordhlist
15    if (std::find(wordList.begin(), wordList.end(), endWord) == wordList.end()) {
```

```

16     cout << format("final-output = 0\n");
17     return 0;
18 }
19
20 // setup
21 unordered_map<string, std::set<string> > neighbours;
22
23 auto calculatedistances = [](const string a, const string b){
24     auto numdiffs {0};
25     for(int i = 0; i<a.size(); ++i) {if (a[i] != b[i]) {++numdiffs;}}
26     if (numdiffs == 0)           {return 0;}
27     else if (numdiffs == 1)      {return 1;}
28     else                         {return std::numeric_limits<int>::max();}
29 };
30
31 // starting again
32 vector<string> candidates {beginWord};
33 for(const auto x: wordList) {if (x!=endWord) candidates.push_back(x);}
34 vector<string> nextgencandidates {};
35
36 vector<string> recruiters      {{endWord}};
37 vector<string> nextgenrecruiters {};
38
39 // going through the loop
40 auto count {1};
41 auto runningcondition {true};
42 auto foundpath       {false};
43
44 while(runningcondition){
45     // increasing count
46     ++count;
47
48     // comparing distance between candidates and recruiters
49     for(const auto candidate: candidates){
50

```



```

51     auto shortestpath {std::numeric_limits<int>::max()};
52     for(const auto& recruiter: recruiters)
53         shortestpath = std::min(shortestpath, calculatedistances(candidate, recruiter));
54
55     // adding to next-generation of recruiters if diff == 1
56     if (shortestpath == 1){
57         nextgenrecruiters.push_back(candidate);    // adding to next gen
58         if (candidate == beginWord) {foundpath = true; runningcondition = false;}
59     }
60     else if (shortestpath == std::numeric_limits<int>::max()){
61         nextgencandidates.push_back(candidate);
62     }
63 }
64
65 // rewriting history
66 int prevnumcandidates = candidates.size();
67 int nexnumcandidates = nextgencandidates.size();
68 candidates = nextgencandidates; nextgencandidates.clear();
69 recruiters = nextgenrecruiters; nextgenrecruiters.clear();
70
71 // evaluating running condition
72 if (prevnumcandidates == nexnumcandidates) {runningcondition = false;}
73
74 }
75
76 // setting up final output
77 if (!foundpath) {count = 0;}
78
79 // printing final-output
80 cout << format("final-output = {}\n", count);
81
82
83 // return
84 return(0);
85

```



## 128. Longest Consecutive Sequence

Given an unsorted array of integers `nums`, return the length of the longest consecutive elements sequence.

You must write an algorithm that runs in  $O(n)$  time.

### Examples

#### 1. Example 1:

- Input: `nums = [100,4,200,1,3,2]`
- Output: 4
- Explanation: The longest consecutive elements sequence is `[1, 2, 3, 4]`. Therefore its length is 4.

#### 2. Example 2:

- Input: `nums = [0,3,7,2,5,8,4,6,0,1]`
- Output: 9

#### 3. Example 3:

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

### Constraints

- $0 \leq \text{nums.length} \leq 10^5$
- $-10^9 \leq \text{nums}[i] \leq 10^9$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums = vector<int>({0,3,7,2,5,8,4,6,0,1});
8
9     // setup
10    std::multiset<int> var00;
11
12    // trivial-cases
13    if (nums.size() <= 1) {cout << format("final-output = {}\n", nums.size()); return 0;}
14
15    // going through elements
16    for(auto x: nums) {var00.insert(x);}
17
18    // going through elements
19    int maxlength = std::numeric_limits<int>::min();
20    int temp = 1;
21    std::deque<int> var01;
22
23    for(auto x: var00){
24
25        if (var01.size()<1)          {var01.push_back(x);}
26        else{
27            // comparing previous element and current
28            if (x - var01[0] == 1)  {++temp; var01[0] = x; maxlength = max(maxlength, temp);}
29            else if(x - var01[0] == 0) {maxlength = max(maxlength, temp); continue;}
30            else                    {maxlength = max(maxlength, temp); temp = 1; var01[0] = x;}
31        }
32    }
33}
```

```
34 // returning the max-length
35 cout << format("final-output = {}\n", maxlength);
36
37 // return
38 return(0);
39
40 }
```

---

## 129. Sum Root to Leaf Numbers

You are given the root of a binary tree containing digits from 0 to 9 only.

Each root-to-leaf path in the tree represents a number. For example, the root-to-leaf path  $1 \rightarrow 2 \rightarrow 3$  represents the number 123. Return the total sum of all root-to-leaf numbers. Test cases are generated so that the answer will fit in a 32-bit integer.

### Constraints

- The number of nodes in the tree is in the range  $[1, 1000]$ .
- $0 \leq \text{Node.val} \leq 9$
- The depth of the tree will not exceed 10.

### Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto root    {new TreeNode(4)};
8     root->left    = new TreeNode(9);
9     root->right   = new TreeNode(0);
10    root->left->left = new TreeNode(5);
11    root->left->right = new TreeNode(1);
12
13    // trivial case
```

```

14  if (root == nullptr) return 0;
15
16  // setup
17  auto runningsum {0};
18  auto finalSum {0};
19  std::function<void(const TreeNode*, int)> foo = [&foo,
20                                              &finalSum](
21      const TreeNode*    root,
22      int                runningsum){
23
24      // sending it back
25      if (root == nullptr) return;
26
27      // adding current value to running sum
28      runningsum = runningsum*10 + root->val;
29
30      // going down left or right
31      if (root->left || root->right){
32          if (root->left)  foo(root->left,  runningsum);
33          if (root->right) foo(root->right, runningsum);
34      }
35      else{
36          // adding to final sum
37          finalSum += runningsum;
38      }
39
40      // return re
41      return;
42  };
43
44  // calling function
45  foo(root, runningsum);
46
47  // returning finalSum
48  cout << format("final-output = {}\n", finalSum);

```

```
49
50     // return
51     return(0);
52
53 }
```

---



## 130. Surrounded Regions

You are given an  $m \times n$  matrix board containing letters 'X' and 'O', capture regions that are surrounded:

- Connect: A cell is connected to adjacent cells horizontally or vertically.
- Region: To form a region connect every 'O' cell.
- Surround: The region is surrounded with 'X' cells if you can connect the region with 'X' cells and none of the region cells are on the edge of the board.

To capture a surrounded region, replace all 'O's with 'X's in-place within the original board. You do not need to return anything.

### Examples

#### 1. Example 1:

- Input: board = `[["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],["X","O","X","X"]]`
- Output: `[["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],["X","O","X","X"]]`
- Explanation: In the above diagram, the bottom region is not captured because it is on the edge of the board and cannot be surrounded.

#### 2. Example 2:

- Input: board = `[["X"]]`
- Output: `[["X"]]`

## Constraints

- `m == board.length`
- `n == board[i].length`
- $1 \leq m, n \leq 200$
- `board[i][j]` is 'X' or 'O'.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto board {std::vector<std::vector<char>>{
8         {'X','X','X','X'},
9         {'X','O','O','X'},
10        {'X','X','O','X'},
11        {'X','O','X','X'}
12    }};
13
14     // setup
15     std::deque<vector<int>> pipe;
16
17     // going from left to right
18     for(int col = 0; col < board[0].size(); ++col)
19         if(board[0][col] == 'O') {pipe.push_back({0, col});}
```

```

20
21 // going from top to bottom
22 for(int row = 0; row < board.size(); ++row)
23     if(board[row][board[0].size()-1] == '0')
24         pipe.push_back({row, static_cast<int>(board[0].size()-1)});
25
26 // going from right to left
27 for(int col = board[0].size()-1; col >= 0 && board.size() > 1; --col)
28     if(board[board.size()-1][col] == '0')
29         pipe.push_back({static_cast<int>(board.size()-1, col)});
30
31 // going from top to bottom
32 for(int row = board.size()-1; row >= 0; --row)
33     if(board[row][0] == '0') {pipe.push_back({row, 0});}
34
35 // writing the visited places
36 vector<vector<int>> registerVector;
37
38 //
39 while(pipe.size() != 0){
40
41     // fetching the front
42     auto front_value {pipe.front()}; pipe.pop_front();
43     const auto& row {front_value[0]};
44     const auto& col {front_value[1]};
45
46     // checking bounds
47     if (row < 0 || row >= board.size() || col < 0 || col >= board[0].size()) {continue;}
48
49     // checking if the token is already in register
50     if (std::find(registerVector.begin(),
51                 registerVector.end(),
52                 front_value) != registerVector.end()) {continue;}
53
54     // checking if current-point is an X

```

```

55     if(board[row][col] == 'X')                                {continue;}
56
57     // adding to register
58     registerVector.push_back(front_value);
59
60     // adding the neighbours
61     pipe.push_back({row, col+1});
62     pipe.push_back({row-1, col});
63     pipe.push_back({row, col-1});
64     pipe.push_back({row+1, col});
65
66 }
67
68 // creating a board of full ones
69 for(auto row = 0; row < board.size(); ++row)
70     std::for_each(board[row].begin(),
71                   board[row].end(),
72                   [](auto& argx){argx= 'X';});
73
74 // filling it with zeros
75 for(const auto& argx: registerVector)
76     board[argx[0]][argx[1]] = '0';
77
78 // printing the matrix
79 fPrintMatrix(board);
80
81 // return
82 return(0);
83
84 }

```

---

## 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 }

```

---

## 136. Single Number

Given a non-empty array of integers `nums`, every element appears twice except for one. Find that single one.

You must implement a solution with a linear runtime complexity and use only constant extra space.

### Examples

1. **Example 1:**

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

2. **Example 2:**

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

3. **Example 3:**

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

### Constraints

- $1 \leq \text{nums.length} \leq 3 * 10^4$
- $-3 * 10^4 \leq \text{nums}[i] \leq 3 * 10^4$
- Each element in the array appears twice except for one element which appears only once.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums {vector<int>{4,1,2,1,2}};
8
9     // going through the number
10    auto finalOutput {nums[0]};
11    for(int i = 1; i<nums.size(); ++i) {finalOutput = finalOutput^nums[i];}
12
13    // returning output
14    cout << format("final-output = {}\n", finalOutput);
15
16    // return
17    return(0);
18
19 }
```

---

## 137. Single Number II

Given an integer array `nums` where every element appears three times except for one, which appears exactly once. Find the single element and return it.

You must implement a solution with a linear runtime complexity and use only constant extra space.

### Examples

#### 1. Example 1:

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

#### 2. Example 2:

- Input: `nums = [0,1,0,1,0,1,99]`
- Output: `99`

### Constraints

- $1 \leq \text{nums.length} \leq 3 * 10^4$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$
- Each element in `nums` appears exactly three times except for one element which appears once.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums {vector<int>{0,1,99,0,1,0,1}};
8
9     // setup
10    std::unordered_map<int, int> histogram;
11    auto finaloutput {-1};
12
13    // going through the nums
14    for(const auto& x: nums){
15        if (histogram.find(x) == histogram.end()) {histogram[x] = 1;}
16        else {++histogram[x];}
17    }
18
19    // going through the histogram again to see who has count 1
20    for(const auto& [k,v]: histogram){
21        if (v == 1) {finaloutput = k;}
22    }
23
24    // printing the output
25    cout << format("final-output = {}\n", finaloutput);
26
27    // return
28    return(0);
29
30 }
```

---

## 141. Linked List Cycle

Given head, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter.

Return true if there is a cycle in the linked list. Otherwise, return false.

### Examples

#### 1. Example 1:

- Input: head = [3,2,0,-4], pos = 1
- Output: true
- Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

#### 2. Example 2:

- Input: head = [1,2], pos = 0
- Output: true
- Explanation: There is a cycle in the linked list, where the tail connects to the 0th node.

#### 3. Example 3:

- Input: head = [1], pos = -1
- Output: false
- Explanation: There is no cycle in the linked list.

## Constraints

- The number of the nodes in the list is in the range [0, 104].
- $-10^5 \leq \text{Node.val} \leq 10^5$
- pos is -1 or a valid index in the linked-list.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     ListNode* head      = new ListNode(3);
8     head->next           = new ListNode(2);
9     head->next->next      = new ListNode(0);
10    head->next->next->next  = new ListNode(-4);
11    head->next->next->next->next = head->next;
12
13    // trivial case
14    if (head == nullptr) {cout << format("final-output = false\n"); return 0;}
15
16    // setup
17    ListNode* dummy      = new ListNode(0);
18    dummy->next           = head;
19    ListNode* fastboi    = dummy;
20    ListNode* slowboi    = dummy;
21
22    while (fastboi != nullptr && fastboi->next != nullptr) {
23
```



```
24     // updating positions
25     fastboi = fastboi->next->next;
26     slowboi = slowboi->next;
27
28     // checking if they're the same
29     if (slowboi == fastboi) {cout << format("final-output = true\n"); return 0;}
30 }
31
32 // since exiting the list implies you left
33 cout << format("final-output = false\n"); return 0;
34
35 // return
36 return(0);
37 }
```

---

## 144. Binary Tree Preorder Traversal

Given the root of a binary tree, return the preorder traversal of its nodes' values.

### Constraints

- The number of nodes in the tree is in the range [0, 100].
- $-100 \leq \text{Node.val} \leq 100$

### Code

---

```
1 void foo(TreeNode* root, vector<int>& finaloutput){
2     // sending it back
3     if (root == nullptr) return;
4
5     // adding current-value to the final output
6     finaloutput.push_back(root->val);
7
8     // going left
9     foo(root->left, finaloutput);
10    foo(root->right, finaloutput);
11
12    // sending it back
13    return;
14 }
15
16 int main(){
17
18     // starting timer
19     Timer timer;
```

```
20
21 // input- configuration
22 auto root      {new TreeNode(1)};
23 root->right     = new TreeNode(2);
24 root->right->left = new TreeNode(3);
25
26 // setup
27 vector<int> finaloutput;
28
29 // calling the function
30 foo(root, finaloutput);
31
32 // returning the vector
33 cout << format("final-output = {}\n", finaloutput);
34
35 // return
36 return(0);
37
38 }
```

---

## 145. Binary Tree Postorder Traversal

Given the root of a binary tree, return the postorder traversal of its nodes' values.

### Constraints

- The number of the nodes in the tree is in the range  $[0, 100]$ .
- $-100$
- `Node.val`
- $100 \leq$

### Code

---

```
1 void foo(TreeNode* root, vector<int>& finaloutput){
2     // sending it back
3     if (root == nullptr) return;
4
5     // going left and right
6     foo(root->left,  finaloutput);
7     foo(root->right,  finaloutput);
8
9     // adding current
10    finaloutput.push_back(root->val);
11
12    // returning
13    return;
14 }
```

```
15  int main(){
16
17      // starting timer
18      Timer timer;
19
20      // input- configuration
21      auto root      {new TreeNode(1)};
22      root->right     = new TreeNode(2);
23      root->right->left = new TreeNode(3);
24
25      // setup
26      vector<int> finaloutput;
27
28      // calling the function
29      foo(root, finaloutput);
30
31      // returnign the final output
32      cout << format("final-output = {}\n", finaloutput);
33
34      // return
35      return(0);
36
37  }
```

---

## 148. Sort List

Given the head of a linked list, return the list after sorting it in ascending order.

### Constraints

- The number of nodes in the list is in the range  $[0, 5 * 10^4]$ .
- $-105 \leq \text{Node.val} \leq 105$

### Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto head      {new ListNode(4)};
8     head->next      = new ListNode(2);
9     head->next->next  = new ListNode(1);
10    head->next->next->next = new ListNode(3);
11
12    // setup
13    if (head == nullptr) {cout << format("empty-list\n"); return 0;}
14
15    // setup
16    ListNode* traveller = nullptr;
17    vector<pair<int, ListNode*>> nodeaddresses;
18
19    // goign through the list
```

```

20     traveller = head;
21     while(traveller){
22
23         // adding it ot the list
24         nodeaddresses.push_back(std::pair<int, ListNode*>({traveller->val, traveller}));
25
26         // moving on to the next one
27         traveller = traveller->next;
28     }
29
30     // sorting the whole thing
31     std::sort(nodeaddresses.begin(),
32             nodeaddresses.end(),
33             [](pair<int, ListNode*> a,
34               pair<int, ListNode*> b) {return a < b;});
35
36     // reconnecting
37     for(int i = 0; i<nodeaddresses.size()-1; ++i)
38         nodeaddresses[i].second->next = nodeaddresses[i+1].second;
39
40
41     // making the last one connect to nullptr
42     nodeaddresses[nodeaddresses.size()-1].second->next = nullptr;
43
44     // returning top of the list
45     fPrintLinkedList("finalOutput = ", nodeaddresses[0].second);
46
47     // return
48     return(0);
49 }

```

---

## 150. Evaluate Reverse Polish Notation

You are given an array of strings `tokens` that represents an arithmetic expression in a Reverse Polish Notation.

Evaluate the expression. Return an integer that represents the value of the expression.



## 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 }
```

---

## 153. Find Minimum in Rotated Sorted Array

Suppose an array of length  $n$  sorted in ascending order is rotated between 1 and  $n$  times. For example, the array `nums = [0,1,2,4,5,6,7]` might become

- `[4,5,6,7,0,1,2]` if it was rotated 4 times.
- `[0,1,2,4,5,6,7]` if it was rotated 7 times.

Notice that rotating an array `[a[0], a[1], a[2], ..., a[n-1]]` 1 time results in the array `[a[n-1], a[0], a[1], a[2], ..., a[n-2]]`.

Given the sorted rotated array `nums` of unique elements, return the minimum element of this array.

You must write an algorithm that runs in  $O(\log n)$  time.

### Examples

#### 1. Example 1:

- Input: `nums = [3,4,5,1,2]`
- Output: 1
- Explanation: The original array was `[1,2,3,4,5]` rotated 3 times.

#### 2. Example 2:

- Input: `nums = [4,5,6,7,0,1,2]`
- Output: 0
- Explanation: The original array was `[0,1,2,4,5,6,7]` and it was rotated 4 times.

### 3. Example 3:

- Input: nums = [11,13,15,17]
- Output: 11
- Explanation: The original array was [11,13,15,17] and it was rotated 4 times.

### Constraints

- $n == \text{nums.length}$
- $1 \leq n \leq 5000$
- $-5000 \leq \text{nums}[i] \leq 5000$
- All the integers of nums are unique.
- nums is sorted and rotated between 1 and n times.

### Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums {vector<int>{3,4,5,1,2}};
8
9     // setup
10    int left          = 0;
```

```
11     int right                = nums.size()-1;
12     int mid;
13
14     // looping through
15     while(left < right){
16
17         // calculating midpoint
18         mid = (left + right)/2;
19
20         // moving pointers
21         if(nums[mid] > nums[right]) {left = mid + 1;}
22         else                        {right = mid;}
23     }
24
25     // returning the mid-value
26     cout << format("final-output = {}\n", nums[left]);
27
28     // return
29     return(0);
30
31 }
```

---

## 162. Find Peak Element

A peak element is an element that is strictly greater than its neighbors.

Given a 0-indexed integer array `nums`, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks.

You may imagine that `nums[-1] = nums[n] =  $-\infty$` . In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

You must write an algorithm that runs in  $O(\log n)$  time.

### Examples

#### 1. Example 1:

- Input: `nums = [1,2,3,1]`
- Output: 2
- Explanation: 3 is a peak element and your function should return the index number 2.

#### 2. Example 2:

- Input: `nums = [1,2,1,3,5,6,4]`
- Output: 5
- Explanation: Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

## Constraints

- $1 \leq \text{nums.length} \leq 1000$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$
- $\text{nums}[i] \neq \text{nums}[i + 1]$  for all valid  $i$ .

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums = {vector<int>{1,2,3,1}};
8
9     // setup
10    int left = {0};
11    int right = {static_cast<int>(nums.size())-1};
12    int mid = {-1};
13
14    // running the lopo
15    while(left <= right){
16
17        // fetching middle index
18        mid = (left + right)/2;
19
20        // evaluating
21        if (mid > 0 && nums[mid] < nums[mid-1]) {right = mid-1;}
22        else if (mid < nums.size()-1 && nums[mid] < nums[mid+1]) {left = mid +1;}
23        else {break;}
```



```
24     }  
25  
26     // returning mid  
27     cout << format("final-output = {}\n", mid);  
28  
29     // return  
30     return(0);  
31  
32 }
```

---

## 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 index1 = 1, 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 }
```

---

## 172. Factorial Trailing Zeroes

Given an integer  $n$ , return the number of trailing zeroes in  $n!$ .

Note that  $n! = n * (n - 1) * (n - 2) * \dots * 3 * 2 * 1$ .

### Examples

#### 1. Example 1:

- Input:  $n = 3$
- Output: 0
- Explanation:  $3! = 6$ , no trailing zero.

#### 2. Example 2:

- Input:  $n = 5$
- Output: 1
- Explanation:  $5! = 120$ , one trailing zero.

#### 3. Example 3:

- Input:  $n = 0$
- Output: 0

### Constraints

- $0 \leq n \leq 10^4$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto n {5};
8
9     // setting up lambdas
10    auto counttwos    = [](int x){
11
12        // in case of x == 0 or x being odd
13        if(x%2 != 0 || x == 0) {return 0;}
14
15        // counting number of 2s we can pluck from this number
16        auto numtwos {0};
17        while(true){
18            if (x%2 == 0) {++numtwos; x /= 2; continue;}
19            else          {return numtwos;}
20        }
21
22        // returning
23        return numtwos;
24    };
25
26    auto countfives    = [](int x){
27
28        // in case of x == 0 or x not being a multiple of five
29        if(x%5 != 0 || x == 0)          {return 0;}
30
31        // counting number of 5s we can pluck from this number
32        auto numfives {0};
33        while(true){
```



```

34         if (x%5 == 0) {++numfives; x /= 5; continue;}
35         else          {return numfives;}
36     }
37
38     // returning numfives
39     return numfives;
40 };
41
42 // calculating numtwos and numfives
43 auto numtwos {0};
44 auto numfives {0};
45
46 for(int i = n; i>=0; --i){
47
48     numtwos    += counttwos(i);
49     numfives   += countfives(i);
50
51 }
52
53 // calculating numzeros
54 auto finaloutput {std::min(numtwos, numfives)};
55 cout << format("final-output = {}\n", finaloutput);
56
57 // return
58 return(0);
59
60 }

```

---

## 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     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 }
```

---

## 190. Reverse Bits

Reverse bits of a given 32 bits signed integer.

### Examples

#### 1. Example 1:

- Input: n = 43261596
- Output: 964176192
- Explanation:

Integer	Binary
43261596	00000010100101000001111010011100
964176192	00111001011110000010100101000000

#### 2. Example 2:

- Input: n = 2147483644
- Output: 1073741822
- Explanation:

Integer	Binary
2147483644	01111111111111111111111111111100
1073741822	00111111111111111111111111111110

## Constraints

- $0 \leq n \leq 2^{31} - 2$
- n is even.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     uint32_t n = 43261596;
8
9     // reversing
10    uint32_t n_reverse = 0b0;
11
12    // reversing loop
13    for (int i = 0; i<32; ++i) {
14        // making space for reverse number and adding current
15        n_reverse = n_reverse<<1 | (n & 1);
16
17        // cutting off tail.
18        n = n>>1;
19    }
20
21    // returning the final output
22    cout << format("final-output = {}\n", n_reverse);
23
24    // return
25    return(0);
```

26

27

}

## 191. Number of 1 Bits

Given a positive integer  $n$ , write a function that returns the number of set bits in its binary representation (also known as the Hamming weight).

### Examples

#### 1. Example 1:

- Input:  $n = 11$
- Output: 3
- Explanation: The input binary string 1011 has a total of three set bits.

#### 2. Example 2:

- Input:  $n = 128$
- Output: 1
- Explanation: The input binary string 10000000 has a total of one set bit.

#### 3. Example 3:

- Input:  $n = 2147483645$
- Output: 30
- Explanation: The input binary string 111111111111111111111111111101 has a total of thirty set bits.

### Constraints

- $1 \leq n \leq 2^{31} - 1$



## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto n {11};
8
9     // setup
10    int count = 0;;
11    while (n) {
12        if ((n & 1)==1) ++count;
13        n = n >> 1;
14    }
15
16    // printing the final-output
17    cout << format("final-output = {}\n", count);
18
19    // return
20    return(0);
21
22 }
```

---

## 198. House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given an integer array `nums` representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

### Examples

#### 1. Example 1:

- Input: `nums = [1,2,3,1]`
- Output: 4
- Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).  
Total amount you can rob =  $1 + 3 = 4$ .

#### 2. Example 2:

- Input: `nums = [2,7,9,3,1]`
- Output: 12
- Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).  
Total amount you can rob =  $2 + 9 + 1 = 12$ .

## Constraints

- $1 \leq \text{nums.length} \leq 100$
- $0 \leq \text{nums}[i] \leq 400$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums {vector<int>{1,2,3,1}};
8
9     // setup
10    std::vector<int> dp{nums[0]};
11    int maxCandidate;
12
13    // building the dp-table
14    for(int i = 1; i<nums.size(); ++i){
15
16        // checking which max value to use
17        if (i > 1) {maxCandidate = std::max(nums[i]+ dp[i-2], dp[i-1]);}
18        else      {maxCandidate = std::max(nums[i], dp[i-1]);}
19
20        // storing max-candidate
21        dp.push_back(maxCandidate);
22    }
23
24    // returning
25    cout << format("final-output = {}\n", dp[dp.size()-1]);
```

```
26
27 // return
28 return(0);
29
30 }
```

---

## 199. Binary Tree Right Side View

Given the root of a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

### Constraints

- The number of nodes in the tree is in the range  $[0, 100]$ .
- $-100 \leq \text{Node.val} \leq 100$

### Code

---

```
1 void fTraverse(TreeNode*      root,
2         std::vector<int>&    finalView,
3         int                  count){
4
5     // base-case
6     if (root == nullptr)      {return;}
7
8     // increasing size of finalView incase it is not big enough
9     while (finalView.size() < count+1) {finalView.push_back(-INT_MAX);}
10
11    // assigning values
12    finalView[count] = root->val;
13
14    // writing just the right value
15    if (root->left != nullptr)    {fTraverse(root->left, finalView, count +1);}
16
17    // going down this branch
```

```

18     if (root->right != nullptr)                {fTraverse(root->right, finalView, count+1);}
19
20     // returning the value
21     return;
22 }
23 int main(){
24
25     // starting timer
26     Timer timer;
27
28     // input- configuration
29     auto root      {new TreeNode(1)};
30     root->left      = new TreeNode(2);
31     root->right     = new TreeNode(3);
32     root->left->right = new TreeNode(5);
33     root->right->right = new TreeNode(4);
34
35     // checking something simple
36     if (root == nullptr)                        {cout << format("final-output = {}\n", std::vector<int>()); return 0;}
37
38     // method to print this out
39     std::vector<int> finalView;
40     auto count {0};
41
42     // calling the function that traverses through these things.
43     finalView.push_back(root->val);
44     fTraverse(root, finalView, 0);
45
46     // return
47     cout << format("final-output = {}\n", finalView);
48
49     // return
50     return(0);
51
52 }

```

---

## 200. Number of Islands

Given an  $m \times n$  2D binary grid `grid` which represents a map of '1's (land) and '0's (water), return the number of islands.

An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

### Examples

#### 1. Example 1:

- Input:

1	1	1	1	0
1	1	0	1	0
1	1	0	0	0
0	0	0	0	0

- Output: 1

#### 2. Example 2:

- Input:

1	1	0	0	0
1	1	0	0	0
0	0	1	0	0
0	0	0	1	1

- Output: 3

### Constraints

- $m == \text{grid.length}$

- $n == \text{grid}[i].\text{length}$
- $1 \leq m, n \leq 300$
- $\text{grid}[i][j]$  is '0' or '1'.

## Code

---

```

1  int main(){
2
3      // starting timer
4      Timer timer;
5
6      // input- configuration
7      vector<vector<string>> grid {
8          vector<string>({"1","1","1","1","0"}),
9          vector<string>({"1","1","0","1","0"}),
10         vector<string>({"1","1","0","0","0"}),
11         vector<string>({"0","0","0","0","0"})
12     };
13
14     // setup
15     std::vector< std::vector<bool> > didWeVisitThisPlace;
16     for(int i = 0; i<grid.size(); ++i)
17         didWeVisitThisPlace.push_back(std::vector<bool>(grid[0].size(), false));
18
19     // lambda to check validity
20     auto fCheckValidityOfCoordinate = [&grid](
21         std::vector<int> coords){
22
23         // spreading it out
24         auto row {coords[0]};
25         auto col {coords[1]};

```



```

26
27 // checking validity
28 if (row >= 0      &&
29     row < grid.size() &&
30     col >= 0      &&
31     col < grid[0].size())
32     if (grid[row][col] == "1") {return true;}
33
34 // in case the above condition is not met
35 return false;
36
37 };
38
39 // traversal function
40 std::function<void(int, int)> fMarkThemAll = [&fMarkThemAll,
41                                             &grid,
42                                             &didWeVisitThisPlace,
43                                             fCheckValidityOfCoordinate](
44     int row_index,
45     int col_index){
46
47     // setting up coordinates
48     std::vector<int> rightCoordinate({row_index, col_index+1});
49     std::vector<int> downCoordinate{row_index+1, col_index};
50
51     // marking the current coordinate as visited
52     didWeVisitThisPlace[row_index][col_index] = true;
53
54     // calling the function to the right
55     if (fCheckValidityOfCoordinate(rightCoordinate) == true && \
56         didWeVisitThisPlace[rightCoordinate[0]][rightCoordinate[1]] == false)
57         fMarkThemAll(rightCoordinate[0], rightCoordinate[1]);
58
59     // calling the function for the ones below
60     if (fCheckValidityOfCoordinate(downCoordinate) == true && \

```

```

61         didWeVisitThisPlace[downCoordinate[0]][downCoordinate[1]] == false)
62             fMarkThemAll(downCoordinate[0], downCoordinate[1]);
63     };
64
65
66     // going through the elements
67     int count = 0;
68     for(int row_index = 0; row_index < grid.size(); ++row_index){
69         for(int col_index = 0; col_index < grid[0].size(); ++col_index){
70             // starting an exploratory course if this point has not been visited
71             if (didWeVisitThisPlace[row_index][col_index] == false && \
72                 grid[row_index][col_index] == "1"){
73                 fMarkThemAll(row_index, col_index);
74                 ++count;
75             }
76         }
77     }
78
79     // return count
80     cout << format("final-output = {}\n", count);
81
82     // return
83     return(0);
84
85 }

```

---

## 202. Happy Number

Write an algorithm to determine if a number  $n$  is happy.

A happy number is a number defined by the following process:

1. Starting with any positive integer, replace the number by the sum of the squares of its digits.
2. Repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1.
3. Those numbers for which this process ends in 1 are happy.

Return true if  $n$  is a happy number, and false if not.

### Examples

#### 1. Example 1:

- Input:  $n = 19$
- Output: true
- Explanation:
  - $1^2 + 9^2 = 82$
  - $8^2 + 2^2 = 68$
  - $6^2 + 8^2 = 100$
  - $1^2 + 0^2 + 0^2 = 1$

#### 2. Example 2:

- Input:  $n = 2$
- Output: false

## Constraints

- $1 \leq n \leq 2^31 - 1$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto n {19};
8
9     // trivial case
10    if (n == 4 || n == 0) {cout << format("final-output = false\n"); return 0;}
11
12    // setting up lambda
13    auto fSumDigitSquares = [](int n) -> int
14    {
15        auto sum {0};
16        auto digit {-1};
17        while(n!=0){
18            digit = n%10;
19            sum += digit*digit;
20            n = n/10;
21        }
22        return sum;
23    };
24
25    // calling the function
26    while (n != 1){
27        // calculating digits sums
```

```
28     n = fSumDigitSquares(n); cout << format("n = {}\n", n);
29     if (n == 4 || n == 0)      {cout << format("final-output = false\n"); return 0;}
30 }
31
32 // printing and returning
33 cout << format("final-output = true \n"); return 0;
34
35 }
```

---

## 205. Isomorphic Strings

Given two strings *s* and *t*, determine if they are isomorphic.

Two strings *s* and *t* are isomorphic if the characters in *s* can be replaced to get *t*.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character, but a character may map to itself.

### Examples

#### 1. Example 1:

- Input: *s* = "egg", *t* = "add"
- Output: true
- Explanation: The strings *s* and *t* can be made identical by:
  - Mapping 'e' to 'a'.
  - Mapping 'g' to 'd'.

#### 2. Example 2:

- Input: *s* = "foo", *t* = "bar"
- Output: false
- Explanation: The strings *s* and *t* can not be made identical as 'o' needs to be mapped to both 'a' and 'r'.

#### 3. Example 3:

- Input: *s* = "paper", *t* = "title"
- Output: true

## Constraints:

- $1 \leq s.length \leq 5 * 10^4$
- $t.length == s.length$
- s and t consist of any valid ascii character.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     string s = {"egg"};
8     string t = {"add"};
9
10    // making mapping
11    unordered_map<char, char> mapping;
12    std::unordered_set<char> destinations;
13    string      r;
14
15    // going through string
16    for(int i = 0; i<s.size(); ++i){
17
18        // checking if two strings are different
19        if (s[i] != t[i]){
20            // checking if we already have a mapping for this
21            if (mapping.find(s[i]) == mapping.end()){
22
23                // before we add, we need to check if t[i] is already used as a destination
```

```

24     if (destinations.find(t[i]) != destinations.end()) {
25         cout << format("final-output = true \n");
26         return 0;
27     }
28
29     // adding new mapping
30     mapping[s[i]] = t[i];
31     destinations.insert(t[i]);
32 }
33 r.push_back(mapping[s[i]]);
34 }
35 else{
36     // checking if we already have a mapping for this
37     if (mapping.find(s[i]) == mapping.end()){
38
39         // before we add, we need to check if t[i] is already used as a destination
40         if (destinations.find(t[i]) != destinations.end()) {
41             cout << format("final-output = true \n");
42             return 0;
43         }
44
45         // adding new mapping
46         mapping[s[i]] = s[i];
47         destinations.insert(s[i]);
48     }
49     r.push_back(mapping[s[i]]);
50 }
51
52 // if what we have so far is different, we're sending it back
53 string s_subset(t.begin(), t.begin() + r.size());
54 if(s_subset != r) {cout << format("final-output = false \n"); return 0;}
55
56 }
57
58 // return true in the end

```



```
59     cout << format("final-output = true \n");
60
61     // return
62     return(0);
63 }
```

---

## 206. Reverse Linked List

Given the head of a singly linked list, reverse the list, and return the reversed list.

### Examples

1. **Example 1:**

- Input: head = [1,2,3,4,5]
- Output: [5,4,3,2,1]

2. **Example 2:**

- Input: head = [1,2]
- Output: [2,1]

3. **Example 3:**

- Input: head = []
- Output: []

### Constraints

- The number of nodes in the list is the range [0, 5000].
- $-5000 \leq \text{Node.val} \leq 5000$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto head          {new ListNode(1)};
8     head->next          = new ListNode(2);
9     head->next->next      = new ListNode(3);
10    head->next->next->next  = new ListNode(4);
11    head->next->next->next->next = new ListNode(5);
12
13    // outlier cases
14    if (head == nullptr) {fPrintLinkedList("final-output = ", nullptr);}
15
16    // setup
17    auto previous_node {static_cast<ListNode*>(nullptr)};
18    auto original_next {static_cast<ListNode*>(nullptr)};
19    auto traveller     {head};
20
21    // going through the list
22    traveller = head;
23    while(traveller!= nullptr){
24
25        original_next = traveller->next;    // storing original next
26        traveller->next = previous_node;    // making the current point to previous
27        previous_node = traveller;         // updating for next
28        traveller     = original_next;     // updating the node
29    }
30
31    // changing head position
32    head = previous_node;
33}
```

```
34     // returning head
35     fPrintLinkedList("final-output = ", head);
36
37     // return
38     return(0);
39
40 }
```

---

## 207. Course Schedule

There are a total of `numCourses` courses you have to take, labeled from 0 to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [ai, bi]` indicates that you must take course `bi` first if you want to take course `ai`.

For example, the pair `[0, 1]`, indicates that to take course 0 you have to first take course 1. Return `true` if you can finish all courses. Otherwise, return `false`.

### Examples

#### 1. Example 1:

- Input: `numCourses = 2, prerequisites = [[1,0]]`
- Output: `true`
- Explanation: There are a total of 2 courses to take.
  - To take course 1 you should have finished course 0. So it is possible.

#### 2. Example 2:

- Input: `numCourses = 2, prerequisites = [[1,0],[0,1]]`
- Output: `false`
- Explanation: There are a total of 2 courses to take.
  - To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

## Constraints

- $1 \leq \text{numCourses} \leq 2000$
- $0 \leq \text{prerequisites.length} \leq 5000$
- $\text{prerequisites}[i].\text{length} == 2$
- $0 \leq \text{ai}, \text{bi} < \text{numCourses}$
- All the pairs  $\text{prerequisites}[i]$  are unique.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto numCourses {3};
8     vector<vector<int>> prerequisites {
9         {1,0},
10        {1,2},
11        {0,1}
12    };
13
14    // setup
15    auto isStartingNode {std::vector<bool>(numCourses, true)};
16    auto finaloutput {false};
17
18    // building dependency tree
19    unordered_map<int, std::vector<int> > dmap;
```

```

20 for(const auto& x: prerequisites){
21     // creating source-to-destination connection
22     if (dmap.find(x[1]) == dmap.end()) {dmap[x[1]] = std::vector<int>({x[0]});}
23     else
24         {dmap[x[1]].push_back(x[0]);}
25
26     // updating
27     isStartingNode[x[0]] = false;
28 }
29
30 // printing
31 cout << format("isStartingNode = {}\n", isStartingNode);
32
33 // launching seach from all non-prerequisite courses
34 auto visited {std::vector<bool>(numCourses, false)};
35 auto pipe {std::deque<int>()};
36
37 // lambda to check if all are true
38 auto fCheckIfAllAreTrue = [&visited]() {
39     return std::all_of(visited.begin(),
40         visited.end(),
41         [](const auto& argx){return argx;});
42 };
43
44 cout<< format("fCheckIfAllAreTrue = {}\n", fCheckIfAllAreTrue());
45 cout << format("isStartingNode = {}\n", isStartingNode);
46
47 // going through each course
48 for(int i = 0; i < numCourses && fCheckIfAllAreTrue() == false; ++i){
49     // checking if this has prerequisites
50     if (isStartingNode[i] == true){
51
52         // creating pipe
53         pipe.clear();
54         pipe.push_back(i);

```

```

55 // running bfs
56 while(pipe.size() != 0){
57
58     // popping the front
59     const auto front_value {pipe.front()};
60     pipe.pop_front();
61
62     // printing
63     cout << format("front-value = {} | pipe = {}\n", front_value, pipe);
64
65     // checking if we've already marked this
66     if (visited[front_value]) {continue;}
67
68     // marking current node as visited
69     visited[front_value] = true;
70
71     // adding children to the pipe
72     for(const auto& x: dmap[front_value])
73         pipe.push_back(x);
74
75     }
76 }
77
78 // returning
79
80
81
82
83 // // setup
84 // vector<bool> visited(numCourses, false);
85 // vector<bool> hasPrerequisite(numCourses, false);
86 // unordered_map<int, vector<int>> hmap;
87
88 // // build prerequisite list
89 // for(const auto x: prerequisites){

```



```

90
91 // // pickign things out
92 // auto curr {x[0]};
93 // auto prereq {x[1]};
94
95 // // setting the flag for non
96 // hasPrerequisite[curr] = true;
97
98 // // adding to the hashmap
99 // if(hmap.find(prereq) == hmap.end()) {hmap[prereq] = vector<int>{curr};}
100 // else {hmap[prereq].push_back(curr);}
101
102 // }
103
104 // // find no-prerequisite course
105 // std::deque<int> indiceswithnoprerequisites;
106 // for(int i = 0; i<numCourses;++i)
107 //     if (hasPrerequisite[i]==false)
108 //         indiceswithnoprerequisites.push_back(i);
109
110 // // start from there and flag the
111 // std::deque<int> prev;
112 // while(indiceswithnoprerequisites.size()!=0 && prev != indiceswithnoprerequisites){
113
114 //     // storing for later comparison
115 //     prev = indiceswithnoprerequisites;
116
117 //     // popping the front
118 //     auto front {indiceswithnoprerequisites.front()};
119 //     indiceswithnoprerequisites.pop_front();
120
121 //     // launching search from "front"
122 //     if (hmap.find(front) == hmap.end()) {
123 //         visited[front] = true;
124 //         continue;

```

```
125 // }
126 // else {
127 //     visited[front] = true;
128 //     auto& bruh {hmap[front]};
129 //     for(const auto& x: bruh) {
130 //         if (visited[x] == false) {indiceswithnoprequisites.push_back(x);}
131 //         visited[x] = true;
132 //     }
133 // }
134 // }
135
136
137 // // finalizing output
138 // auto finaloutput {true};
139 // for(int i = 0; i<visited.size(); ++i) {finaloutput = finaloutput&&visited[i];}
140 // cout << format("final-output = {}\n", finaloutput);
141
142 // return
143 return(0);
144
145 }
```

---

## 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 }
```

---

## 212. Word Search II

Given an  $m \times n$  board of characters and a list of strings words, return all words on the board.

Each word must be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

### Examples

#### 1. Example 1:

- Input: board = `[["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]]`, words = `["oath","pea"]`
- Output: `["eat","oath"]`

#### 2. Example 2:

- Input: board = `[["a","b"],["c","d"]]`, words = `["abcb"]`
- Output: `[]`

### Constraints

- $m == \text{board.length}$
- $n == \text{board}[i].\text{length}$
- $1 \leq m, n \leq 12$
- `board[i][j]` is a lowercase English letter.

- $1 \leq \text{words.length} \leq 3 * 10^4$
- $1 \leq \text{words}[i].\text{length} \leq 10$
- `words[i]` consists of lowercase English letters.
- All the strings of words are unique.

## Code

---

```

1 void foo(const vector<vector<char>>& board,
2         const int
3         const int
4         const string&
5         bool&
6         vector<int>
7         std::function<int(const int&, const int&)> tid,
8         std::function<int(const int)>
9         std::function<int(const int)>
10
11 // if found is done
12 if (found == true) {return;}
13
14 // returning if tid is beyond the bounds
15 if (curr_tid >= (board.size() * board[0].size())) {return;}
16
17 // calculating curr-row and col
18 const auto rowcurr {row(curr_tid)}; if (rowcurr < 0 || rowcurr >= board.size()) {return;}
19 const auto colcurr {col(curr_tid)}; if (colcurr < 0 || colcurr >= board[0].size()) {return;}
20
21 // check: duplicate tid in path
22 if (std::find(pathsofar.begin(), pathsofar.end(), curr_tid) != pathsofar.end()) {return;}
23

```

```

24 // check: target-char vs curr-char
25 char curr_char          {board[rowcurr][colcurr]};
26 char charweresearchingfor {targetword[p]};
27 if (curr_char != charweresearchingfor)                                {return;}
28
29 // adding to path
30 pathsofar.push_back(curr_tid);
31
32 // checking if we've reached the end
33 if (pathsofar.size() == targetword.size()) {found = true; return;}
34
35 // moving into the neighbours
36 foo(board, p+1, tid(rowcurr, colcurr+1), targetword, found, pathsofar, tid, row, col); // moving right
37 foo(board, p+1, tid(rowcurr-1, colcurr), targetword, found, pathsofar, tid, row, col); // moving up
38 foo(board, p+1, tid(rowcurr, colcurr-1), targetword, found, pathsofar, tid, row, col); // moving left
39 foo(board, p+1, tid(rowcurr+1, colcurr), targetword, found, pathsofar, tid, row, col); // moving down
40
41 // returning
42 return;
43 }
44
45
46 int main(){
47
48 // starting timer
49 Timer timer;
50
51 // setup
52 vector<vector<char>> board {
53     {'a','b','c'},
54     {'a','e','d'},
55     {'a','f','g'}
56 };
57
58

```



```

59 vector<string> words {"abcdefg", "gfedcbaaa", "eaabcdgfa", "befa", "dgc", "ade"};
60
61 // setup
62 std::function<int(const int&, const int&)> tid = [&board](const int row, const int col) {
63     if (row < 0 || row >= board.size() || col < 0 || col >= board[0].size()) {return std::numeric_limits<int>::max();}
64     return row* static_cast<int>(board[0].size()) + col ;
65 };
66 std::function<int(const int)> row = [&board](const int tid){
67     if (tid < 0 || tid > board.size() * board[0].size()) {return std::numeric_limits<int>::max();}
68     return tid/static_cast<int>(board[0].size());
69 };
70 std::function<int(const int)> col = [&board](const int tid){
71     if (tid < 0 || tid > board.size() * board[0].size()) {return std::numeric_limits<int>::max();}
72     return tid%static_cast<int>(board[0].size());
73 };
74
75 // printing
76 vector<string> finaloutput;
77 for(const auto& targetword: words){
78
79     // flag for finding the word
80     bool found {false};
81
82     // launching from every coordinate
83     for(int i = 0; i<board.size() && found == false; ++i){
84         for(int j = 0; j<board[0].size() && found == false; ++j){
85             vector<int> pathsofar;
86             foo(board, 0, tid(i, j), targetword, found, pathsofar, tid, row, col);
87         }
88     }
89
90     // adding to final output
91     if (found) {finaloutput.push_back(targetword);}
92 }
93

```

```
94 // printing the final-output
95 PRINTSPACE
96 cout << format("finaloutput = {}\n", finaloutput);
97
98 // return
99 return(0);
100
101 }
```

---

## 213. House Robber II

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have a security system connected, and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given an integer array `nums` representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

### Examples

#### 1. Example 1:

- Input: `nums = [2,3,2]`
- Output: 3
- Explanation: You cannot rob house 1 (money = 2) and then rob house 3 (money = 2), because they are adjacent houses.

#### 2. Example 2:

- Input: `nums = [1,2,3,1]`
- Output: 4
- Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
  - Total amount you can rob =  $1 + 3 = 4$ .

#### 3. Example 3:

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

## Constraints

- $1 \leq \text{nums.length} \leq 100$
- $0 \leq \text{nums}[i] \leq 1000$

## Code

---

```
1 int fRob1(std::vector<int> nums){
2
3     // setup
4     std::vector<int> dp{nums[0]};
5     int maxCandidate;
6
7     // building the dp-table
8     for(int i = 1; i<nums.size(); ++i){
9
10        // checking which max value to use
11        if (i > 1)    {maxCandidate = std::max(nums[i]+ dp[i-2], dp[i-1]);}
12        else        {maxCandidate = std::max(nums[i], dp[i-1]);}
13
14        // storing max-candidate
15        dp.push_back(maxCandidate);
16
17    }
18
19    // returning the mx-value
20    return dp[dp.size()-1];
21 }
22
23 int main(){
24
25     // starting timer
```

```

26     Timer timer;
27
28     // input- configuration
29     auto nums {vector<int>{1,2,3,1}};
30
31     // sending things back
32     if(nums.size()== 0) {cout << format("final-output = 0\n"); return 0;}
33     if(nums.size()== 1) {cout << format("final-output = {}\n", nums[0]); return 0;}
34
35     // setup
36     auto firstCandidate {fRob1(std::vector<int>(nums.begin(), nums.end()-1))};
37     auto secondCandidate {fRob1(std::vector<int>(nums.begin()+1, nums.end()))};
38
39     // returning value
40     auto finaloutput {std::max(std::max(firstCandidate, secondCandidate), nums[0])};
41     cout << format("final-output = {}\n", finaloutput);
42
43     // return
44     return(0);
45
46 }

```

---

## 215. Kth Largest Element in an Array

Given an integer array `nums` and an integer `k`, return the `k`th largest element in the array.

Note that it is the `k`th largest element in the sorted order, not the `k`th distinct element.

Can you solve it without sorting?

### Examples

#### 1. Example 1:

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

#### 2. Example 2:

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

### Constraints

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

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums {vector<int>{3,2,3,1,2,4,5,5,6}};
8     auto k     {4};
9
10    // setup
11    std::nth_element(nums.begin(), nums.end()-k, nums.end());
12    auto finaloutput {nums[nums.size()-k]};
13
14    // printing
15    cout << format("final-output = {}\n", finaloutput);
16
17    // return
18    return(0);
19
20 }
```

---

## 216. Combination Sum III

Find all valid combinations of  $k$  numbers that sum up to  $n$  such that the following conditions are true:

- Only numbers 1 through 9 are used.
- Each number is used at most once.

Return a list of all possible valid combinations. The list must not contain the same combination twice, and the combinations may be returned in any order.

### Examples

#### 1. Example 1:

- Input:  $k = 3, n = 7$
- Output: `[[1,2,4]]`
- Explanation:
  - $1 + 2 + 4 = 7$
  - There are no other valid combinations.

#### 2. Example 2:

- Input:  $k = 3, n = 9$
- Output: `[[1,2,6],[1,3,5],[2,3,4]]`



- Explanation:
  - $1 + 2 + 6 = 9$
  - $1 + 3 + 5 = 9$
  - $2 + 3 + 4 = 9$
  - There are no other valid combinations.

### 3. Example 3:

- Input:  $k = 4, n = 1$
- Output: []
- Explanation: There are no valid combinations.
  - Using 4 different numbers in the range [1,9], the smallest sum we can get is  $1+2+3+4 = 10$  and since  $10 \neq 1$ , there are no valid combination.

## Constraints

- $2 \leq k \leq 9$
- $1 \leq n \leq 60$

## Code

---

```

1 void fTraverse(std::vector<int> numberPath, \
2               int target, \
3               int k, \
4               std::vector< std::vector<int> >& finalOutput){
5
6     // checking if length is too much
7     if (numberPath.size() > k) return;
```

```

8
9 // calculating sum so far
10 int sumsofar = std::accumulate(numberPath.begin(), numberPath.end(),0);
11
12 // number to fill
13 int sumtofill = target - sumsofar;
14
15 // if this is zero, we can add it to list and just sendn it back
16 if (sumtofill == 0){
17     if (numberPath.size() == k){
18         if (std::find(finalOutput.begin(),
19                     finalOutput.end(),
20                     numberPath) == finalOutput.end())
21             finalOutput.push_back(numberPath); // add if it doesn't already exist
22     }
23
24     return;
25 }
26
27 // valid candidates
28 std::vector<int> candidates;
29 int biggest_number = 1;
30 if (numberPath.size()!=0){
31     auto iter = std::max_element(numberPath.begin(), numberPath.end());
32     biggest_number = *iter;
33 }
34
35 sumtofill = std::min(sumtofill, 9);
36 for(int i = biggest_number; i<=sumtofill; ++i){
37     // add if no on th epath so far
38     if (std::find(numberPath.begin(),
39                 numberPath.end(),
40                 i) == numberPath.end()) {candidates.push_back(i);}
41 }
42

```

```

43 // if there are no candidates, we're going back
44 if (candidates.size() == 0) return;
45
46 // trying each candidate
47 std::vector<int> numberPath_local;
48 for(auto x: candidates){
49     // sending down each candidate route
50     numberPath_local = numberPath;
51     numberPath_local.push_back(x);
52     fTraverse(numberPath_local, target, k, finalOutput);
53 }
54
55 // returning
56 return;
57 }
58
59 int main(){
60
61     // starting timer
62     Timer timer;
63
64     // input- configuration
65     auto k {3};
66     auto n {7};
67
68     // setup
69     std::vector< std::vector<int> > finalOutput;
70     std::vector<int> numberPath;
71
72     // recursion
73     fTraverse(numberPath, n, k, finalOutput);
74
75     // printing
76     cout << format("final-output = {}\n", finalOutput);
77

```

```
78     // return
79     return(0);
80
81 }
```

---

## 217. Contains Duplicate

Given an integer array `nums`, return `true` if any value appears at least twice in the array, and return `false` if every element is distinct.

### Examples

1. **Example 1:**

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

2. **Example 2:**

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

3. **Example 3:**

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

### Constraints

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^9 \leq \text{nums}[i] \leq 10^9$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums {vector<int>{1,2,3,1}};
8
9     // setup
10    unordered_map<int, int> histogram;
11    auto finaloutput {false};
12
13    // building histogram
14    for(const auto& x: nums){
15        if (histogram.find(x) == histogram.end()) {histogram[x] = 1;}
16        else {finaloutput = true; break;}
17    }
18
19    // printing the final-output
20    cout << format("final-output = {}\n", finaloutput);
21
22    // return
23    return(0);
24
25 }
```

---

## 219. Contains Duplicate II

Given an integer array `nums` and an integer `k`, return `true` if there are two distinct indices `i` and `j` in the array such that `nums[i] == nums[j]` and `abs(i - j) ≤ k`.

### Examples

1. **Example 1:**

- Input: `nums = [1,2,3,1]`, `k = 3`
- Output: `true`

2. **Example 2:**

- Input: `nums = [1,0,1,1]`, `k = 1`
- Output: `true`

3. **Example 3:**

- Input: `nums = [1,2,3,1,2,3]`, `k = 2`
- Output: `false`

### Constraints

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^9 \leq \text{nums}[i] \leq 10^9$
- $0 \leq k \leq 10^5$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums = std::vector<int>({1,2,3,1,2,3});
8     auto k     {2};
9
10    // setup
11    std::unordered_map<int, int> valueToIndex;
12
13    // going through the array
14    for(int i = 0; i<nums.size(); ++i){
15
16        // adding current element to the hamp
17        if (valueToIndex.find(nums[i]) == valueToIndex.end()){
18            // if it doesn't exist, we add to it
19            valueToIndex[nums[i]] = i;
20        }
21        else{
22            // if it already exists, calculating distance from the first index
23            if (i - valueToIndex[nums[i]] <= k) {cout << format("final-output = true\n");}
24            else {valueToIndex[nums[i]] = i;}
25        }
26    }
27
28    // returning false in the final case
29    cout << format("final-output = false\n");
30
31    // return
32    return(0);
33}
```





## 222. Count Complete Tree Nodes

Given the root of a complete binary tree, return the number of the nodes in the tree.

According to Wikipedia, every level, except possibly the last, is completely filled in a complete binary tree, and all nodes in the last level are as far left as possible. It can have between  $1$  and  $2^h$  nodes inclusive at the last level  $h$ .

Design an algorithm that runs in less than  $O(n)$  time complexity.

### Examples

#### 1. Example 1:

- Input: root = [1,2,3,4,5,6]
- Output: 6

#### 2. Example 2:

- Input: root = []
- Output: 0

#### 3. Example 3:

- Input: root = [1]
- Output: 1

## Constraints

- The number of nodes in the tree is in the range  $[0, 5 * 10^4]$ .
- $0 \leq \text{Node.val} \leq 5 * 10^4$
- The tree is guaranteed to be complete.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto root      {new TreeNode(1)};
8     root->left      = new TreeNode(2);
9     root->right     = new TreeNode(3);
10    root->left->left  = new TreeNode(4);
11    root->left->right = new TreeNode(5);
12    root->right->left = new TreeNode(6);
13
14    // setup
15    auto finalresult {0};
16    std::function<void(const TreeNode*)> fCount = [&fCount, &finalresult](
17        const TreeNode* root){
18
19        // returning
20        if (root == nullptr) {return;}
21        ++finalresult;
22
23        // going down left and right
```

```
24         fCount(root->left);
25         fCount(root->right);
26
27         // returning
28         return;
29     };
30
31     // calling the function
32     fCount(root);
33
34     // printing the final output
35     cout << format("final-output = {}\n", finalresult);
36
37
38     // return
39     return(0);
40
41 }
```

---

## 226. Invert Binary Tree

Given the root of a binary tree, invert the tree, and return its root.

### Examples

1. **Example 1:**

- Input: root = [4,2,7,1,3,6,9]
- Output: [4,7,2,9,6,3,1]

2. **Example 2:**

- Input: root = [2,1,3]
- Output: [2,3,1]

3. **Example 3:**

- Input: root = []
- Output: []

### Constraints

- The number of nodes in the tree is in the range [0, 100].
- $-100 \leq \text{Node.val} \leq 100$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto root      {new TreeNode(4)};
8     root->left      = new TreeNode(2);
9     root->right      = new TreeNode(7);
10    root->left->left   = new TreeNode(1);
11    root->left->right  = new TreeNode(3);
12    root->right->left  = new TreeNode(6);
13    root->right->right = new TreeNode(9);
14
15    // setup
16    std::function<void(TreeNode*)> fTwist = [&fTwist](TreeNode* root){
17        // base-case
18        if (root == nullptr) return;
19
20        // switching left and right
21        fTwist(root->left);
22        fTwist(root->right);
23
24        // twisting current branches
25        auto temp = root->left;
26        root->left = root->right;
27        root->right = temp;
28
29        // returning
30        return;
31    };
32
33    // going flipping through everything
```

```
34     fTwist(root);  
35  
36     // return  
37     return(0);  
38  
39 }
```

---

## 228. Summary Ranges

You are given a sorted unique integer array `nums`.

A range `[a,b]` is the set of all integers from `a` to `b` (inclusive).

Return the smallest sorted list of ranges that cover all the numbers in the array exactly. That is, each element of `nums` is covered by exactly one of the ranges, and there is no integer `x` such that `x` is in one of the ranges but not in `nums`.

Each range `[a,b]` in the list should be output as:

"a→b" if `a != b`

"a" if `a == b`

### Examples

#### 1. Example 1:

- Input: `nums = [0,1,2,4,5,7]`
- Output: `["0→2","4→5","7"]`
- Explanation: The ranges are:
  - `[0,2] ⇒ "0→2"`
  - `[4,5] ⇒ "4→5"`
  - `[7,7] ⇒ "7"`

#### 2. Example 2:

- Input: `nums = [0,2,3,4,6,8,9]`
- Output: `["0","2→4","6","8→9"]`



- Explanation: The ranges are:
  - $[0,0] \implies "0"$
  - $[2,4] \implies "2 \rightarrow 4"$
  - $[6,6] \implies "6"$
  - $[8,9] \implies "8 \rightarrow 9"$

## Constraints

- $0 \leq \text{nums.length} \leq 20$
- $-2^31 \leq \text{nums}[i] \leq 2^31 - 1$
- All the values of nums are unique.
- nums is sorted in ascending order.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums = vector<int>({0,1,2,4,5,7});
8
9     // trivial-case
10    if (nums.size() == 0) {cout << format("final-output = {}\n", vector<string>()); return 0;}
11
12    // stup
```

```

13 auto arrow {"->"};
14 auto p1 {0};
15 auto p2 {0};
16 vector<string> finalOutput;
17
18 // while loop
19 while (p2<nums.size()){
20
21     // checking prev element
22     if (nums[p2] <= 1 + nums[p2-1]) {++p2;}
23     else{
24         if (p1!=p2-1) {finalOutput.push_back(std::to_string(nums[p1]) + arrow + std::to_string(nums[p2-1]));}
25         else {finalOutput.push_back(std::to_string(nums[p2-1]));}
26         p1 = p2++;
27     }
28 }
29
30 // checking if p1 and p2 areright ne
31 if (p1!=p2-1) {finalOutput.push_back(std::to_string(nums[p1]) + arrow + std::to_string(nums[p2-1]));}
32 else {finalOutput.push_back(std::to_string(nums[p2-1]));}
33
34 // returning restul
35 cout << format("final-output = {}\n", finalOutput);
36
37 // return
38 return(0);
39
40 }

```

---

## 230. Kth Smallest Element in a BST

Given the root of a binary search tree, and an integer k, return the kth smallest value (1-indexed) of all the values of the nodes in the tree.

### Examples

1. **Example 1:**

- Input: root = [3,1,4,null,2], k = 1
- Output: 1

2. **Example 2:**

- Input: root = [5,3,6,2,4,null,null,1], k = 3
- Output: 3

### Constraints

- The number of nodes in the tree is n.
- $1 \leq k \leq n \leq 10^4$
- $0 \leq \text{Node.val} \leq$

## Code

---

```
1 void foo(const TreeNode* root,
2         int& leftindex,
3         int& finalOutput,
4         bool& stopsearch,
5         const int k)
6 {
7     // sending it back
8     if (root == nullptr) return;
9
10    // checking left
11    foo(root->left, leftindex, finalOutput, stopsearch, k);
12
13    // appending count
14    ++leftindex;
15    if (leftindex == k){
16
17        finalOutput = root->val;
18        stopsearch = true;
19
20    }
21
22    // going right
23    foo(root->right, leftindex, finalOutput, stopsearch, k);
24
25    // goign back
26    return;
27 }
28
29 int main(){
30
31    // starting timer
32    Timer timer;
33
```

```

34 // input- configuration
35 auto root      {new TreeNode(3)};
36 root->left     = new TreeNode(1);
37 root->right    = new TreeNode(4);
38 root->left->right = new TreeNode(2);
39
40 auto k {1};
41
42 // setup
43 auto finalOutput {-1};
44 auto stopsearch  {false};
45 auto leftindex   {0};
46
47 // running the search
48 foo(root, leftindex, finalOutput, stopsearch, k);
49
50 // printing otuput
51 cout << format("final-output = {}\n", finalOutput);
52
53 // return
54 return(0);
55
56 }

```

---

## 235. Lowest Common Ancestor of a Binary Search Tree

Given a binary search tree (BST), find the lowest common ancestor (LCA) node of two given nodes in the BST.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).”

### Examples

#### 1. Example 1:

- Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
- Output: 6
- Explanation: The LCA of nodes 2 and 8 is 6.

#### 2. Example 2:

- Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
- Output: 2
- Explanation: The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

#### 3. Example 3:

- Input: root = [2,1], p = 2, q = 1
- Output: 2

## Constraints

- The number of nodes in the tree is in the range  $[2, 10^5]$ .
- $-10^9 \leq \text{Node.val} \leq 10^9$
- All  $\text{Node.val}$  are unique.
- $p \neq q$
- $p$  and  $q$  will exist in the BST.

## Code

---

```
1 void foo(const TreeNode*    root,
2         const int          smallervalue,
3         const int          largervalue,
4         TreeNode*&          finalanswer){
5
6     if (root== nullptr)      return;
7     if (finalanswer != nullptr) return;
8
9     // checking current value
10    if (smallervalue <= root->val && root->val <= largervalue) {finalanswer = root;}
11    else if (root->val < smallervalue) {foo(root->right, smallervalue, largervalue, finalanswer);}
12    else if (root->val > largervalue) {foo(root->left,  smallervalue, largervalue, finalanswer);}
13
14    // returnring
15    return;
16 }
17
18 int main(){
19
```

```

20 // starting timer
21 Timer timer;
22
23 // input- configuration
24 auto root {new TreeNode(6)};
25
26 root->left    = new TreeNode(2);
27 root->right   = new TreeNode(8);
28
29 root->left->left = new TreeNode(0);
30 root->left->right = new TreeNode(4);
31 root->right->left = new TreeNode(7);
32 root->right->right = new TreeNode(9);
33
34 root->left->right->left = new TreeNode(3);
35 root->left->right->right = new TreeNode(5);
36
37 auto p    {root->left};
38 auto q    {root->right};
39
40 // setup
41 const auto smallvalue {std::min(p->val, q->val)};
42 const auto largervalue {std::max(p->val, q->val)};
43 auto      finalanswer  {static_cast<TreeNode*>(nullptr)};
44
45 //calling the function
46 foo(root, smallvalue, largervalue, finalanswer);
47
48 // returning the final answer
49 cout << format("final-output = {}\n", finalanswer->val);
50
51 // return
52 return(0);
53
54 }

```



## 236. Lowest Common Ancestor of a Binary Tree

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).”

### Constraints

- The number of nodes in the tree is in the range [2, 105].
- $-10^9 \leq \text{Node.val} \leq 10^9$
- All Node.val are unique.
- $p \neq q$
- p and q will exist in the tree.

### Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto root {new TreeNode(3)};
8
9     root->left = new TreeNode(5);
```

```

10 root->right = new TreeNode(1);
11
12 root->left->left = new TreeNode(6);
13 root->left->right = new TreeNode(2);
14
15 root->right->left = new TreeNode(0);
16 root->right->right = new TreeNode(8);
17
18 root->left->right->left = new TreeNode(7);
19 root->left->right->right = new TreeNode(4);
20
21 auto p {root->left};
22 auto q {root->right};
23
24 // setup
25 TreeNode *finalanswer {static_cast<TreeNode*>(nullptr)};
26 std::function<int(TreeNode*)> foo = [&foo,
27                                     &finalanswer,
28                                     &p, &q](
29     TreeNode* root){
30
31     // sending it back
32     if (root== nullptr) {return 0;}
33     if (finalanswer != nullptr) {return 0;}
34
35     // searching left
36     auto leftsum = foo(root->left);
37     auto rightsum = foo(root->right);
38
39     // summing up values
40     auto sumvalues = leftsum + rightsum;
41     if (root == p || root == q) {sumvalues+=1;}
42
43     // updating final answer
44     if (sumvalues == 2 &&

```

```
45         finalanswer == nullptr)      {finalanswer = root;}
46
47         // returning values
48         return sumvalues;
49
50     };
51
52     // running the function
53     foo(root);
54
55     // returning the final answer
56     cout << format("Ancestor-value = {}, Ancestor-address = {}\n",
57         finalanswer->val, static_cast<void*>(finalanswer));
58
59
60     // return
61     return(0);
62
63 }
```

---

## 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     auto nums {vector<int>{1,2,3,4}};
5
6     // setup
7     auto left {vector<int>(nums.size(), 1)};
8     auto right {vector<int>(nums.size(), 1)};
9
10    // building the cumulative products
11    std::partial_sum(nums.begin(), nums.end()-1,
12                    left.begin()+1,
13                    [](auto arg0,
14                      auto arg1){
15                        return arg0 * arg1;
16                    });
17    std::partial_sum(nums.rbegin(), nums.rend()-1,
18                    right.rbegin()+1,
19                    [](auto arg0,
20                      auto arg1){
21                        return arg0 * arg1;
22                    });
23
24    // producing the final-output
25    auto finaloutput {vector<int>(nums.size(),1)};
26    std::transform(left.begin(), left.end(),
27                  right.begin(),
28                  finaloutput.begin(),
29                  std::multiplies<int>());
30
31    // printing the final-output
32    cout << format("final-output = {}\n", finaloutput);
33
```

```
34 // return
35 return(0);
36
37 }
```

---

## 239. Sliding Window Maximum

You are given an array of integers `nums`, there is a sliding window of size `k` which is moving from the very left of the array to the very right. You can only see the `k` numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

### Examples

#### 1. Example 1:

- Input: `nums = [1,3,-1,-3,5,3,6,7]`, `k = 3`
- Output: `[3,3,5,5,6,7]`
- Explanation:

Window position	Max
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

#### 2. Example 2:

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

## Constraints

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

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums = vector<int>{1,3,-1,-3,5,3,6,7};
8     auto k = {3};
9
10
11    // setup
12    auto finaloutput = std::vector<int>(nums.size() - k + 1);
13
14
15    // going through the arrays
16    auto count = 0;
17    for(int i = k; i <= nums.size(); ++i){
18
19        // storing the max to the output
20        finaloutput[i-k] = *(std::max_element(
21            nums.begin()+i-k,
22            nums.begin()+i
23        ));
```



```

24 }
25
26 // printing the final output
27 cout << format("final-output = {}\n", finaloutput);
28
29
30
31
32
33
34
35
36
37
38
39 // // setup
40 // auto finaloutput      {vector<int>{}};
41 // auto heap             {std::vector(nums.begin(), nums.begin()+k)};
42 // std::make_heap(heap.begin(), heap.end());
43 // auto numbertobedeleted {-1};
44 // auto numbertobeadded  {-1};
45 // auto topvalue         {-1};
46 // finaloutput.push_back(*(std::max_element(heap.begin(), heap.end())));
47
48 // // going through the nums
49 // for(int i = 1; i<=nums.size()-k; ++i){
50
51 //     // removing the element that just left the window
52 //     numbertobedeleted = nums[i-1];
53 //     auto deletion_it  = std::find(heap.begin(),
54 //                                   heap.end(),
55 //                                   numbertobedeleted);
56 //     heap.erase(deletion_it);
57 //     std::make_heap(heap.begin(),
58 //                   heap.end());

```

```

59
60 //      // adding new number to heap
61 //      numbertobeadded  = nums[i+k-1];
62 //      heap.push_back(numbertobeadded);
63 //      std::push_heap(heap.begin(),
64 //                      heap.end());
65
66 //      // picking biggest number
67 //      std::pop_heap(heap.begin(),
68 //                   heap.end());
69 //      topvalue       = heap.back();
70
71 //      // adding to final-output
72 //      std::push_heap(heap.begin(),
73 //                   heap.end());
74 //      finaloutput.push_back(topvalue);
75 // }
76
77 // // printing
78 // cout << format("finaloutput = {}\n", finaloutput);
79
80 // return
81 return(0);
82
83 }

```

---

## 242. Valid Anagram

Given two strings  $s$  and  $t$ , return true if  $t$  is an anagram of  $s$ , and false otherwise.

### Examples

#### 1. Example 1:

- Input:  $s = \text{"anagram"}, t = \text{"nagaram"}$
- Output: true

#### 2. Example 2:

- Input:  $s = \text{"rat"}, t = \text{"car"}$
- Output: false

### Constraints

- $1 \leq s.length, t.length \leq 5 * 10^4$
- $s$  and  $t$  consist of lowercase English letters.

### Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     string s {"anagram"};
8     string t {"nagaram"};
9
10    // setup
11    std::vector<int> s_count(26,0);
12    std::vector<int> t_count(26,0);
13
14    // building count-vector for both
15    for(auto x: s) ++s_count[static_cast<int>(x)-97];
16    for(auto x: t) ++t_count[static_cast<int>(x)-97];
17
18    // comparing the two
19    for (int i = 0; i<s_count.size(); ++i) {
20
21        // element-wise checking
22        if (s_count[i] != t_count[i]) {cout << format("final-output = false \n");}
23
24    }
25
26    // returning
27    cout << format("final-output = true \n");
28
29    // return
30    return(0);
31
32 }
```

---

## 268. Missing Number

Given an array `nums` containing  $n$  distinct numbers in the range  $[0, n]$ , return the only number in the range that is missing from the array.

### Examples

#### 1. Example 1:

- Input: `nums = [3,0,1]`
- Output: 2
- Explanation:  $n = 3$  since there are 3 numbers, so all numbers are in the range  $[0,3]$ . 2 is the missing number in the range since it does not appear in `nums`.

#### 2. Example 2:

- Input: `nums = [0,1]`
- Output: 2
- Explanation:  $n = 2$  since there are 2 numbers, so all numbers are in the range  $[0,2]$ . 2 is the missing number in the range since it does not appear in `nums`.

#### 3. Example 3:

- Input: `nums = [9,6,4,2,3,5,7,0,1]`
- Output: 8
- Explanation:  $n = 9$  since there are 9 numbers, so all numbers are in the range  $[0,9]$ . 8 is the missing number in the range since it does not appear in `nums`.

## Constraints

- $n == \text{nums.length}$
- $1 \leq n \leq 10^4$
- $0 \leq \text{nums}[i] \leq n$
- All the numbers of nums are unique.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums {vector<int>{9,6,4,2,3,5,7,0,1}};
8
9     // setup
10    auto n {nums.size()+1};
11    auto finaloutput {0};
12
13    // going through the numbers
14    for(int i = 0; i<nums.size(); ++i){
15        finaloutput += i+1 - nums[i];
16    }
17
18    // printing the finaloutput
19    cout << format("finaloutput = {}\n", finaloutput);
20
21    // return
```

```
22     return(0);
23
24 }
```

---

## 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 }
```

---

## 279. Perfect Squares

Given an integer  $n$ , return the least number of perfect square numbers that sum to  $n$ .

A perfect square is an integer that is the square of an integer; in other words, it is the product of some integer with itself. For example, 1, 4, 9, and 16 are perfect squares while 3 and 11 are not.

### Examples

#### 1. Example 1:

- Input:  $n = 12$
- Output: 3
- Explanation:  $12 = 4 + 4 + 4$ .

#### 2. Example 2:

- Input:  $n = 13$
- Output: 2
- Explanation:  $13 = 4 + 9$ .

### Constraints

- $1 \leq n \leq 10^4$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto n {static_cast<int>(13)};
8
9     // setup
10    auto length_of_square_numbers {static_cast<std::size_t>(1+std::ceil(std::sqrt(n)))};
11    auto square_numbers {std::vector<int>(length_of_square_numbers, 0)};
12
13    // filling up the vector
14    for(auto i = 0; i < square_numbers.size(); ++i)
15        square_numbers[i] = i*i;
16
17    // building the dptable
18    auto dptable {std::vector<int>(n+1)};
19    dptable[0] = 1;
20
21    // building the dp-table
22    for(auto i = 1; i <= n; ++i){
23
24        // consider previous answer
25        auto& curr_value {i};
26
27        // checking if the current-value can be achieved with just the coins
28        if(std::find(square_numbers.begin(),
29                    square_numbers.end(),
30                    curr_value) != square_numbers.end()) {dptable[i] = 1; continue;}
31
32        // considering all the previous guys
33        auto min_num_coins {std::numeric_limits<int>::max()};
```

```

34 for(int j = i-1; j > (i-1)/2; --j)
35 {
36     // getting current dp-entry
37     const auto var00 {j};
38     const auto var01 {curr_value - j};
39
40     // checking what coin to use to get here
41     const auto num_coins_required {dptable[var00] + dptable[var01]};
42
43     // calculating number of coins required
44     min_num_coins = std::min(num_coins_required, min_num_coins);
45 }
46
47 // writing to entry
48 dptable[i] = min_num_coins;
49 }
50
51 // printing final-output
52 cout << format("final-output = {}\n", dptable[dptable.size()-1]);
53
54 // return
55 return(0);
56
57 }

```

---

## 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 }
```

---

## 289. Game of Life

According to Wikipedia's article: "The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."

The board is made up of an  $m \times n$  grid of cells, where each cell has an initial state: live (represented by a 1) or dead (represented by a 0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

1. Any live cell with fewer than two live neighbors dies as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by over-population.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The next state of the board is determined by applying the above rules simultaneously to every cell in the current state of the  $m \times n$  grid board. In this process, births and deaths occur simultaneously.

Given the current state of the board, update the board to reflect its next state.

Note that you do not need to return anything.

### Examples

#### 1. Example 1

- Input: board = `[[0,1,0],[0,0,1],[1,1,1],[0,0,0]]`

- Output: `[[0,0,0],[1,0,1],[0,1,1],[0,1,0]]`

## 2. Example 2

- Input: `board = [[1,1],[1,0]]`
- Output: `[[1,1],[1,1]]`

## Constraints

- `m == board.length`
- `n == board[i].length`
- $1 \leq m, n \leq 25$
- `board[i][j]` is 0 or 1.

## Code

---

```
1 int fRules(vector< vector<int> >& board,
2           int row,
3           int col){
4
5     //setup
6     int row_curr;
7     int col_curr;
8     int num_live_neighbours = 0;
9     int selfstatus = board[row][col];
10
11     for(int i = -1; i <=1; ++i){
12         for(int j = -1; j<= 1; ++j){
```



```

13
14 // finding current row and column
15 row_curr = row +i;
16 col_curr = col +j;
17
18 // continuing if same element
19 if (row_curr == row && col_curr == col) continue;
20
21 // continueing for edge cases
22 if (row_curr <0 || \
23     row_curr >= board.size() || \
24     col_curr < 0 || \
25     col_curr >= board[0].size()) {continue;}
26
27 // checking if neighbour is live or not
28 if (board[row_curr][col_curr] == 1)
29     ++num_live_neighbours;
30 }
31 }
32
33 // the wild case where the current cell is dead and numneighbrs = 3
34 if (selfstatus == 0 && num_live_neighbours == 3) return 1;
35 if (selfstatus == 0) return 0;
36
37 // checking life-rule conditions
38 if (selfstatus == 1 && num_live_neighbours < 2) return 0;
39 else if (selfstatus == 1 && num_live_neighbours < 4) return 1;
40 else if (selfstatus == 1 && num_live_neighbours > 3) return 0;
41
42 // return default
43 return -1000;
44 }
45
46 int main(){
47

```

```

48 // starting timer
49 Timer timer;
50
51 // input- configuration
52 vector<vector<int>> board{
53     {0,1,0},
54     {0,0,1},
55     {1,1,1},
56     {0,0,0}
57 };
58
59 // setup
60 auto x = board;
61 int statustoadd;
62
63 // going through the elements
64 for(int i = 0; i<board.size(); ++i){
65     for(int j = 0; j<board[0].size(); ++j){
66
67         // updating cell based on rules
68         statustoadd = fRules(board, i, j);
69
70         // adding status to x
71         x[i][j] = statustoadd;
72     }
73 }
74
75 // copying results back
76 std::copy(x.begin(), x.end(), board.begin());
77
78 // printing the matrix
79 cout << format("board = \n"); fPrintMatrix(board);
80
81 // return
82 return(0);

```



## 290. Word Pattern

Given a pattern and a string *s*, find if *s* follows the same pattern.

Here follow means a full match, such that there is a bijection between a letter in pattern and a non-empty word in *s*. Specifically:

- Each letter in pattern maps to exactly one unique word in *s*.
- Each unique word in *s* maps to exactly one letter in pattern.
- No two letters map to the same word, and no two words map to the same letter.

### Examples

#### 1. Example 1:

- Input: pattern = "abba", s = "dog cat cat dog"
- Output: true
- Explanation: The bijection can be established as:
  - 'a' maps to "dog".
  - 'b' maps to "cat".

#### 2. Example 2:

- Input: pattern = "abba", s = "dog cat cat fish"
- Output: false

#### 3. Example 3:

- Input: pattern = "aaaa", s = "dog cat cat dog"
- Output: false

## Constraints

- $1 \leq \text{pattern.length} \leq 300$
- pattern contains only lower-case English letters.
- $1 \leq \text{s.length} \leq 3000$
- s contains only lowercase English letters and spaces ' '.
- s does not contain any leading or trailing spaces.
- All the words in s are separated by a single space.

## Code

---

```
1 bool fCheckHashValues(const unordered_map<char, string>& charToWord,
2                       const string& tocheck){
3
4     // going through it
5     for(auto x: charToWord) {if (x.second == tocheck) return true;}
6
7     // return false
8     return false;
9
10 }
11
12 int main(){
13
14     // starting timer
15     Timer timer;
16
17     // input- configuration
```

```

18 string pattern {"abba"};
19 string s      {"dog cat cat dog"};
20
21 // setup
22 unordered_map<char, string> charToWord;
23 auto wordi                {0};
24 auto metaletter {false};
25 auto numwords  {0};
26
27 // going through the two variables
28 for(int i = 0; i<pattern.size(); ++i){
29
30     // temp string
31     string temp;
32
33     // extracting first word
34     int prevwordi = wordi;
35     while (wordi < s.size()){
36
37         // checking if something is non-space
38         if (s[wordi] != ' ') {temp.push_back(s[wordi]); metaletter = true;}
39         else                  {if (metaletter == true) break;}
40
41         // appending wordi
42         ++wordi;
43     }
44
45
46     // resetting stats
47     metaletter = false;
48     if(prevwordi!=wordi) {++numwords;}
49
50     // check if current char is in hashmap
51     if (charToWord.find(pattern[i]) == charToWord.end()){
52

```

```

53     // check if destination is taken
54     if (fCheckHashValues(charToWord, temp)) {cout << format("final-output = false \n"); return 0;}
55     else                                     {charToWord[pattern[i]] = temp;}
56
57 }
58 else {if (charToWord[pattern[i]] != temp) {cout << format("final-output = false \n"); return 0;}} // checking if
      mapping is same as temp
59 }
60
61 // printing
62 cout << format("numwords = {}, pattern.size() = {} \n", numwords, pattern.size());
63
64 // checking number of words and characters
65 if (pattern.size()!=numwords) {cout << format("final-output = false \n"); return 0;}
66
67 // check if there were more to go
68 if (wordi!=s.size())          {cout << format("final-output = false \n"); return 0;}
69
70 // printing
71 cout << format("final-output = true \n"); return 0;
72 }

```

---

## 300. Longest Increasing Subsequence

Given an integer array `nums`, return the length of the longest strictly increasing subsequence.

### Examples

#### 1. Example 1:

- Input: `nums = [10,9,2,5,3,7,101,18]`
- Output: 4
- Explanation: The longest increasing subsequence is `[2,3,7,101]`, therefore the length is 4.

#### 2. Example 2:

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

#### 3. Example 3:

- Input: `nums = [7,7,7,7,7,7,7]`
- Output: 1

### Constraints

- $1 \leq \text{nums.length} \leq 2500$
- $-10^4 \leq \text{nums}[i] \leq 10^4$



## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums {vector<int>{10,9,2,5,3,7,101,18}};
8
9     // finding number of elements
10    auto n {static_cast<int>(nums.size())};
11    std::vector<int> dp(n, 1);    // table to store the dynamic variables
12
13    // going through the arrays
14    for(int i = 0; i<n; ++i)
15        for(int j = 0; j<i; ++j)
16            if (nums[i] > nums[j]) {dp[i] = std::max(dp[i], dp[j]+1);}
17
18    // printing max-value
19    auto maxiter = std::max_element(dp.begin(), dp.end());
20    cout << format("final-output = {}\n", *maxiter);
21
22    // return
23    return(0);
24
25 }
```

---

## 322. Coin Change

You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money.

Return the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

### Examples

#### 1. Example 1:

- Input: `coins = [1,2,5]`, `amount = 11`
- Output: 3
- Explanation:  $11 = 5 + 5 + 1$

#### 2. Example 2:

- Input: `coins = [2]`, `amount = 3`
- Output: -1

#### 3. Example 3:

- Input: `coins = [1]`, `amount = 0`
- Output: 0

## Constraints

- $1 \leq \text{coins.length} \leq 12$
- $1 \leq \text{coins}[i] \leq 2^31 - 1$
- $0 \leq \text{amount} \leq 104$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto coins    {std::vector<int>({1,2,5})};
8     auto amount   {11};
9
10    // trivial cases
11    if (coins.size() == 1 && (amount % coins[0] == 0))
12    {
13        return static_cast<int>(amount/coins[0]);
14    }
15    if (amount == 0) {return 0;}
16
17    // setup
18    vector<int> dptable(amount + 1, std::numeric_limits<int>::max());
19    dptable[0] = 0;
20
21    // going through each coin
22    for (int coin : coins) {
23        // calculating upward from that coin
```

```
24     for (int x = coin; x <= amount; ++x) {
25         if (dptable[x - coin] != std::numeric_limits<int>::max())
26             dptable[x] = std::min(dptable[x],
27                                   dptable[x - coin] + 1);
28     }
29 }
30
31 // returning final output
32 return dptable[amount] == std::numeric_limits<int>::max() ? -1 : dptable[amount];
33
34 // returning final output
35 cout << format("final-output = {}\n", finaloutput);
36
37
38
39
40
41 // return
42 return(0);
43
44 }
```

---

## 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 }
```

---

## 347. Top K Frequent Elements

Given an integer array `nums` and an integer `k`, return the `k` most frequent elements. You may return the answer in any order.

### Examples

1. **Example 1:**

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

2. **Example 2:**

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

3. **Example 3:**

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

### Constraints

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- `k` is in the range `[1, the number of unique elements in the array]`.
- It is guaranteed that the answer is unique.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums {vector<int>{4,1,-1,2,-1,2,3}};
8     auto k     {2};
9
10    // setup
11    unordered_map<int, int> hist;
12
13    // building histogram
14    for(const auto& num: nums){
15        if (hist.find(num) == hist.end()) {hist[num] = 1;}
16        else {++hist[num];}
17    }
18
19    // creating vector pair
20    vector<vector<int>> kv;
21    for(const auto& x: hist) {kv.emplace_back(vector<int>{x.first, x.second});}
22
23    // building the sorting lambda
24    auto sortinglambda = [](const auto& a, const auto& b) {return a[1] > b[1];};
25
26    // since we want just the first k-elements, use partial sort
27    std::partial_sort(kv.begin(), kv.begin() + k, kv.end(), sortinglambda);
28
29    // building the final output
30    auto finaloutput {vector<int>{}};
31    for(int i = 0; i<k; ++i) {finaloutput.push_back(kv[i][0]);}
32
33    // printing the final output
```



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

---

## 349. Intersection of Two Arrays

Given two integer arrays `nums1` and `nums2`, return an array of their intersection. Each element in the result must be unique and you may return the result in any order.

### Examples

#### 1. Example 1:

- Input: `nums1 = [1,2,2,1]`, `nums2 = [2,2]`
- Output: `[2]`

#### 2. Example 2:

- Input: `nums1 = [4,9,5]`, `nums2 = [9,4,9,8,4]`
- Output: `[9,4]`
- Explanation: `[4,9]` is also accepted.

### Constraints

- $1 \leq \text{nums1.length}, \text{nums2.length} \leq 1000$
- $0 \leq \text{nums1}[i], \text{nums2}[i] \leq 1000$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums1 {vector<int>{4,9,5}};
8     auto nums2 {vector<int>{9,4,9,8,4}};
9
10    // setup
11    std::set<int> nums1set(nums1.begin(), nums1.end());
12    std::set<int> nums2set(nums2.begin(), nums2.end());
13
14    // printing
15    cout << format("nums1set = {}\n", nums1set);
16    cout << format("nums2set = {}\n", nums2set);
17
18    // calculating the intersection
19    vector<int> finaloutput;
20    for(const auto& x: nums1set){
21
22        if (std::find(nums2set.begin(),
23                      nums2set.end(),
24                      x) != nums2set.end()) {finaloutput.push_back(x);}
25    }
26
27    // printing
28    cout << format("final-output = {}\n", finaloutput);
29
30    // return
31    return(0);
32
33 }
```

---

## 350. Intersection of Two Arrays II

Given two integer arrays `nums1` and `nums2`, return an array of their intersection. Each element in the result must appear as many times as it shows in both arrays and you may return the result in any order.

### Examples

#### 1. Example 1:

- Input: `nums1 = [1,2,2,1]`, `nums2 = [2,2]`
- Output: `[2,2]`

#### 2. Example 2:

- Input: `nums1 = [4,9,5]`, `nums2 = [9,4,9,8,4]`
- Output: `[4,9]`
- Explanation: `[9,4]` is also accepted.

### Constraints

- $1 \leq \text{nums1.length}, \text{nums2.length} \leq 1000$
- $0 \leq \text{nums1}[i], \text{nums2}[i] \leq 1000$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums1    {vector<int>{4,9,5}};
8     auto nums2    {vector<int>{9,4,9,8,4}};
9
10    // setup
11    unordered_map<int, int> histogram1;
12    unordered_map<int, int> histogram2;
13
14    // lambda to add to histogram
15    auto addtohistogram = [] (unordered_map<int, int>& histogram, int entryvalue){
16        if (histogram.find(entryvalue) == histogram.end()) {histogram[entryvalue] = 1;}
17        else {++histogram[entryvalue];}
18    };
19
20    // building the histogram
21    for(const auto x: nums1) {addtohistogram(histogram1, x);}
22    for(const auto x: nums2) {addtohistogram(histogram2, x);}
23
24    // going through the first histogram
25    vector<int> finaloutput;
26    for(const auto kvp: histogram1){
27
28        // splitting the pair
29        auto& keyvalue {kvp.first};
30        auto& value    {kvp.second};
31
32        // checking if it is there in the second histogram
33        if(histogram2.find(keyvalue) != histogram2.end()){
```

```
34     auto countforthis {std::min(value, histogram2[keyvalue])};
35     auto tempvector    {vector<int>(countforthis, keyvalue)};
36     for(int i = 0; i < countforthis; ++i) {finaloutput.push_back(keyvalue);}
37 }
38
39 }
40
41 // printing the output
42 cout << format("final-output = {}\n", finaloutput);
43
44 // return
45 return(0);
46
47 }
```

---

## 365. Water and Jug Problem

You are given two jugs with capacities  $x$  liters and  $y$  liters. You have an infinite water supply. Return whether the total amount of water in both jugs may reach target using the following operations:

1. Fill either jug completely with water.
2. Completely empty either jug.
3. Pour water from one jug into another until the receiving jug is full, or the transferring jug is empty.

### Examples

#### 1. Example 1:

- Input:  $x = 3$ ,  $y = 5$ , target = 4
- Output: true
- Explanation: Follow these steps to reach a total of 4 liters:
  - Fill the 5-liter jug (0, 5).
  - Pour from the 5-liter jug into the 3-liter jug, leaving 2 liters (3, 2).
  - Empty the 3-liter jug (0, 2).
  - Transfer the 2 liters from the 5-liter jug to the 3-liter jug (2, 0).
  - Fill the 5-liter jug again (2, 5).
  - Pour from the 5-liter jug into the 3-liter jug until the 3-liter jug is full. This leaves 4 liters in the 5-liter jug (3, 4).
  - Empty the 3-liter jug. Now, you have exactly 4 liters in the 5-liter jug (0, 4).

#### 2. Example 2:

- Input:  $x = 2, y = 6, \text{target} = 5$
- Output: false

### 3. Example 3:

- Input:  $x = 1, y = 2, \text{target} = 3$
- Output: true
- Explanation: Fill both jugs. The total amount of water in both jugs is equal to 3 now.

## Constraints

- $1 \leq x, y, \text{target} \leq 103$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto x      {34};
8     auto y      {5};
9     auto target {6};
10
11    // trivial case
12    if (target > (x + y)) {return false;}
13
14    // lambda to calculate gcd
15    std::function<int(int, int)>
```



```
16 calculateGCD = [&calculateGCD](int x,  
17                             int y){  
18     if (y == 0) {return x;};  
19     return calculateGCD(y, x%y);  
20 };  
21  
22 // calculating gcd  
23 auto gcd_xy      {calculateGCD(x, y)};  
24 auto finaloutput {target % calculateGCD(x, y) == 0};  
25  
26 // returning final output  
27 cout << format("finaloutput = {}\n", finaloutput);  
28  
29 // return  
30 return(0);  
31  
32 }
```

---

### 383. Ransom Note

Given two strings ransomNote and magazine, return true if ransomNote can be constructed by using the letters from magazine and false otherwise. Each letter in magazine can only be used once in ransomNote.

#### Examples

1. **Example 1:**

- Input: ransomNote = "a", magazine = "b"
- Output: false

2. **Example 2:**

- Input: ransomNote = "aa", magazine = "ab"
- Output: false

3. **Example 3:**

- Input: ransomNote = "aa", magazine = "aab"
- Output: true

#### Constraints

- $1 \leq \text{ransomNote.length}, \text{magazine.length} \leq 10^5$
- ransomNote and magazine consist of lowercase English letters.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     string ransomNote {"aa"};
8     string magazine   {"aab"};
9
10    // trivial case
11    if (ransomNote.size()>magazine.size()) {
12        cout << format("status = true\n");
13        return 0;
14    }
15
16    // setup
17    vector<int> hist(26,0);
18
19    // building histogram
20    for(auto x: magazine)    {++hist[static_cast<int>(x-'a')];}
21
22    // going through ransom note
23    for(auto x: ransomNote){
24        if(hist[static_cast<int>(x-'a')]-- == 0) {
25            cout << format("status = true\n"); return 0;
26        }
27    }
28
29    // return true
30    cout << format("status = true\n");
31
32    // return
33    return(0);
```

34

35

}

## 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[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: `"abccabccdcddcdef"`

## 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

```



## 414. Third Maximum Number

Given an integer array `nums`, return the third distinct maximum number in this array. If the third maximum does not exist, return the maximum number.

### Examples

#### 1. Example 1:

- Input: `nums = [3,2,1]`
- Output: 1
- Explanation:
  - The first distinct maximum is 3.
  - The second distinct maximum is 2.
  - The third distinct maximum is 1.

#### 2. Example 2:

- Input: `nums = [1,2]`
- Output: 2
- Explanation:
  - The first distinct maximum is 2.
  - The second distinct maximum is 1.
  - The third distinct maximum does not exist, so the maximum (2) is returned instead.

#### 3. Example 3:

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

- Output: 1
- Explanation:
  - The first distinct maximum is 3.
  - The second distinct maximum is 2 (both 2's are counted together since they have the same value).
  - The third distinct maximum is 1.

## 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     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto    nums    {vector<int>{1,2}};
8
9     // setup
10    auto    largestnumber {std::numeric_limits<int>::min()};
11    auto    numsset       {std::set<int>(nums.begin(), nums.end())};
12    auto    myheap        {std::vector<int>{}};
13    std::make_heap(myheap.begin(),
14                  myheap.end(),
15                  std::greater<int>());
16
```

```

17 // going through the histogram
18 for(const auto x: numsset){
19
20     // adding element to it
21     myheap.push_back(x);
22     largestnumber = std::max(largestnumber, x);
23
24     // making a heap out of it
25     std::push_heap(myheap.begin(),
26                   myheap.end(),
27                   std::greater<int>());
28
29     // popping if size is greater
30     if (myheap.size() > 3){
31         std::pop_heap(myheap.begin(),
32                      myheap.end(),
33                      std::greater<int>());
34         myheap.pop_back();
35     }
36 }
37
38 // taking the last element
39 int finaloutput = largestnumber;
40 if (myheap.size() == 3){
41     std::pop_heap(myheap.begin(),
42                  myheap.end(),
43                  std::greater<int>());
44     finaloutput = myheap.back();
45 }
46
47 // printing the finaloutput
48 cout << format("final-output = {}\n", finaloutput);
49
50 // return
51 return(0);

```

52

53

}

## 417. Pacific Atlantic Water Flow

There is an  $m \times n$  rectangular island that borders both the Pacific Ocean and Atlantic Ocean. The Pacific Ocean touches the island's left and top edges, and the Atlantic Ocean touches the island's right and bottom edges.

The island is partitioned into a grid of square cells. You are given an  $m \times n$  integer matrix `heights` where `heights[r][c]` represents the height above sea level of the cell at coordinate  $(r, c)$ .

The island receives a lot of rain, and the rain water can flow to neighboring cells directly north, south, east, and west if the neighboring cell's height is less than or equal to the current cell's height. Water can flow from any cell adjacent to an ocean into the ocean.

Return a 2D list of grid coordinates result where `result[i] = [ri, ci]` denotes that rain water can flow from cell  $(r_i, c_i)$  to both the Pacific and Atlantic oceans.

### Examples

#### 1. Example 1:

- Input: `heights = [[1,2,2,3,5],[3,2,3,4,4],[2,4,5,3,1],[6,7,1,4,5],[5,1,1,2,4]]`
- Output: `[[0,4],[1,3],[1,4],[2,2],[3,0],[3,1],[4,0]]`

#### 2. Example 2:

- Input: `heights = [[1]]`
- Output: `[[0,0]]`
- Explanation: The water can flow from the only cell to the Pacific and Atlantic oceans.

## Constraints

- $m == \text{heights.length}$
- $n == \text{heights}[r].\text{length}$
- $1 \leq m, n \leq 200$
- $0 \leq \text{heights}[r][c] \leq 10^5$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input-configuration
7     auto heights {std::vector<std::vector<int>>>{
8         {1,2,2,3,5},
9         {3,2,3,4,4},
10        {2,4,5,3,1},
11        {6,7,1,4,5},
12        {5,1,1,2,4}
13    }};
14
15    // setup
16    std::vector<int> tidList;
17    auto finaloutput {std::vector<std::vector<int>>>()};
18
19    // lambda for calculating tid
20    auto fetchtid = [&heights](const int row, const int col){
21        if (row < 0 || col < 0) throw std::runtime_error("row, col < 0");
```



```

22     return col + row * heights[0].size();
23 };
24
25 // building marking table
26 auto visitingTable {std::vector<std::vector<int>>>(
27     heights.size(),
28     std::vector<int>(heights[0].size(), 0)
29 )};
30
31 // beginning search from pacific
32 auto pacific_pipe {std::deque<std::vector<int>>>()};
33
34 // adding to pacific pipe before running bfs
35 for(auto row = 0; row < heights.size(); ++row){
36     pacific_pipe.push_back(std::vector<int>({row, 0}));
37 }
38 for(auto col = 1; col < heights[0].size(); ++col){
39     pacific_pipe.push_back(std::vector<int>({0, col} ));
40 }
41
42 auto checkIfUpstream = [&heights](
43     const auto& curr_row,
44     const auto& curr_col,
45     const auto& next_row,
46     const auto& next_col){
47
48     // checking if either coordinates are valid
49     if (curr_row < 0 || curr_row >= heights.size()) {return false;}
50     if (curr_col < 0 || curr_col >= heights[0].size()) {return false;}
51     if (next_row < 0 || next_row >= heights.size()) {return false;}
52     if (next_col < 0 || next_col >= heights[0].size()) {return false;}
53
54     // checking difference in value
55     return heights[next_row][next_col] >= heights[curr_row][curr_col];
56 };

```

```

57
58 // running bfs from
59 while(pacific_pipe.size()!=0)
60 {
61     // popping front
62     auto curr {pacific_pipe.front()}; pacific_pipe.pop_front();
63     auto& row {curr[0]};
64     auto& col {curr[1]};
65
66     // ERROR-CHECK: checking if it is already visited
67     const auto tid {fetchtid(row, col)};
68     if(std::find(tidList.begin(),
69                 tidList.end(),
70                 tid) != tidList.end()) {continue;}
71
72     // adding to visit-list
73     tidList.push_back(tid);
74
75     // marking to dptable
76     visitingTable[row][col] += 1;
77
78     // adding the neighbours to the list
79     if(checkIfUpstream(row, col, row, col+1)) {pacific_pipe.push_back(std::vector<int>({row, col+1}));}
80     if(checkIfUpstream(row, col, row-1, col)) {pacific_pipe.push_back(std::vector<int>({row-1, col}));}
81     if(checkIfUpstream(row, col, row, col-1)) {pacific_pipe.push_back(std::vector<int>({row, col-1}));}
82     if(checkIfUpstream(row, col, row+1, col)) {pacific_pipe.push_back(std::vector<int>({row+1, col}));}
83 }
84
85
86 // creating atlantic pipe
87 auto& atlantic_pipe {pacific_pipe}; pacific_pipe.clear();
88 tidList.clear();
89
90 // filling up atlantic pipe
91 for(auto row = 0; row < heights.size(); ++row)

```

```

92     atlantic_pipe.push_back({row, static_cast<int>(heights[0].size() - 1)});
93 for(auto col = 0; col < heights[0].size(); ++col)
94     atlantic_pipe.push_back({static_cast<int>(heights.size()-1), col});
95
96 // running bfs
97 while(atlantic_pipe.size() != 0)
98 {
99     // popping front
100    auto curr {atlantic_pipe.front()}; atlantic_pipe.pop_front();
101    auto& row {curr[0]};
102    auto& col {curr[1]};
103
104    // checking if visited
105    const auto tid {fetchtid(row, col)};
106    if (std::find(tidList.begin(),
107                 tidList.end(),
108                 tid) != tidList.end()) {continue;}
109
110    // adding to tid
111    tidList.push_back(tid);
112
113    // marking the table
114    visitingTable[row][col] += 2;
115
116    // adding to final output
117    if (visitingTable[row][col] == 3)
118        finaloutput.push_back(std::vector<int>({row, col}));
119
120    // adding the neighbours to the list
121    if(checkIfUpstream(row, col, row, col+1)) {atlantic_pipe.push_back(std::vector<int>({row, col+1}));}
122    if(checkIfUpstream(row, col, row-1, col)) {atlantic_pipe.push_back(std::vector<int>({row-1, col}));}
123    if(checkIfUpstream(row, col, row, col-1)) {atlantic_pipe.push_back(std::vector<int>({row, col-1}));}
124    if(checkIfUpstream(row, col, row+1, col)) {atlantic_pipe.push_back(std::vector<int>({row+1, col}));}
125
126 }

```

```
127
128 // printing the markes
129 cout << format("final-output = {}\n", finaloutput);
130
131 // return
132 return(0);
133
134 }
```

---

## 424. Longest Repeating Character Replacement

You are given a string  $s$  and an integer  $k$ . You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most  $k$  times.

Return the length of the longest substring containing the same letter you can get after performing the above operations.

### Examples

#### 1. Example 1:

- Input:  $s = \text{"ABAB"}, k = 2$
- Output: 4
- Explanation: Replace the two 'A's with two 'B's or vice versa.

#### 2. Example 2:

- Input:  $s = \text{"AABABBA"}, k = 1$
- Output: 4
- Explanation: Replace the one 'A' in the middle with 'B' and form "AABBBBA".
  - The substring "BBBB" has the longest repeating letters, which is 4.
  - There may exist other ways to achieve this answer too.

### Constraints

- $1 \leq s.length \leq 10^5$

- s consists of only uppercase English letters.
- $0 \leq k \leq s.length$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto s {string("ABAB")};
8     auto k {2};
9
10    // setup
11    auto left          {0};
12    auto right         {0};
13    auto num_characters_local {0};
14    auto finalOutput   {0};
15    auto maxelement    {-1};
16
17    std::vector<int> characterCount(26, 0); // histogram
18
19    // going through the elements
20    for(right = 0; right < s.size(); ++right){
21
22        // adding to count
23        characterCount[static_cast<int>(s[right])-65] += 1;
24
25        // setup
26        num_characters_local = right - left + 1;
27    }
```

```

28 // checking if number of repl
29 maxelement = *(std::max_element(characterCount.begin(), characterCount.end()));
30
31 if (num_characters_local - maxelement > k ) {
32
33     characterCount[static_cast<int>(s[left])-65] -= 1;           // removing the leftmost element
34     ++left;                                                     // moving left
35 }
36
37 // calculating longest
38 finalOutput = std::max(finalOutput, right-left+1);
39
40 }
41
42 // printing the final-output
43 cout << format("final-output = {}\n", finalOutput);
44
45 // return
46 return(0);
47
48 }

```

---

## 429. N-ary Tree Level Order Traversal

Given an n-ary tree, return the level order traversal of its nodes' values.

Nary-Tree input serialization is represented in their level order traversal, each group of children is separated by the null value.

### Constraints

- The height of the n-ary tree is less than or equal to 1000
- The total number of nodes is between  $[0, 10^4]$

### Code

---

```
1 // Definition for a Node.
2 class Node {
3 public:
4     int val;
5     vector<Node*> children;
6
7     Node() {}
8
9     Node(int _val) {
10         val = _val;
11     }
12
13     Node(int _val, vector<Node*> _children) {
14         val = _val;
15         children = _children;
16     }
17 };
```



```

18
19 struct NodeLevelPair
20 {
21     Node* ptr;
22     int level;
23     NodeLevelPair() = default;
24     NodeLevelPair(Node* ptr_, int level_): ptr(ptr_), level(level_) {}
25 };
26
27 int main(){
28
29     // starting timer
30     Timer timer;
31
32     // input- configuration
33     auto root {new Node(1)};
34     root->children.push_back(new Node(3));
35     root->children.push_back(new Node(2));
36     root->children.push_back(new Node(4));
37     root->children[0]->children.push_back(new Node(5));
38     root->children[0]->children.push_back(new Node(6));
39
40     // setup
41     std::vector<std::vector<int>> finaloutput;
42     auto pipe {std::deque<NodeLevelPair>()};
43     pipe.push_back(NodeLevelPair({root, 0}));
44
45
46     // bfs
47     while(pipe.size() != 0)
48     {
49         // popping front
50         auto curr {pipe.front()}; pipe.pop_front();
51         auto curr_level {curr.level};
52

```

```
53 // writing to final output
54 if (curr_level < finaloutput.size()) {finaloutput[curr_level].push_back(curr.ptr->val);}
55 else {
56     finaloutput.push_back(std::vector<int>({curr.ptr->val}));
57 }
58
59 // printing final output
60 cout << format("curr->val = {}, finaloutput = {}\n", curr.ptr->val, finaloutput);
61
62 // adding children
63 for(const auto& child: curr.ptr->children)
64     pipe.push_back({child, curr_level + 1});
65 }
66
67 // printing the final output
68 cout << format("finaloutput = {}\n", finaloutput);
69
70
71 // return
72 return(0);
73
74 }
```

---

## 433. Minimum Genetic Mutation

A gene string can be represented by an 8-character long string, with choices from 'A', 'C', 'G', and 'T'.

Suppose we need to investigate a mutation from a gene string `startGene` to a gene string `endGene` where one mutation is defined as one single character changed in the gene string.

For example, "AACCGGTT" → "AACCGGTA" is one mutation.

There is also a gene bank `bank` that records all the valid gene mutations. A gene must be in `bank` to make it a valid gene string.

Given the two gene strings `startGene` and `endGene` and the gene bank `bank`, return the minimum number of mutations needed to mutate from `startGene` to `endGene`. If there is no such a mutation, return -1.

Note that the starting point is assumed to be valid, so it might not be included in the bank.

### Examples

#### 1. Example 1:

- Input: `startGene` = "AACCGGTT", `endGene` = "AACCGGTA", `bank` = ["AACCGGTA"]
- Output: 1

#### 2. Example 2:

- Input: `startGene` = "AACCGGTT", `endGene` = "AAACGGTA", `bank` = ["AACCGGTA","AACCGCTA","AAACGGTA"]
- Output: 2

## Constraints

- $0 \leq \text{bank.length} \leq 10$
- $\text{startGene.length} == \text{endGene.length} == \text{bank}[i].\text{length} == 8$
- $\text{startGene}$ ,  $\text{endGene}$ , and  $\text{bank}[i]$  consist of only the characters ['A', 'C', 'G', 'T'].

## Code

---

```
1 void foo(unordered_map<string, vector<string>>& stringtoneighbours,
2         vector<string> pathsofar,
3         bool& foundpath,
4         string endGene,
5         int& finaloutput){
6
7     // checking if the top of the stack has valid neighbours
8     string top = pathsofar[pathsofar.size()-1];
9
10    // checking if the current one is the final output
11    if (top == endGene)
12        finaloutput = finaloutput < pathsofar.size() ? finaloutput : pathsofar.size();
13
14    // checking its possible next-states
15    auto nextnodes = stringtoneighbours[top];
16
17    // going depth-first
18    for(auto x: nextnodes){
19
20        // not considering if it is already in the path
21        if(std::find(pathsofar.begin(),
22                    pathsofar.end(),
23                    x) != pathsofar.end()) continue;
```

```

24
25     // updating path so far
26     auto pathsofar_temp = pathsofar;
27     pathsofar_temp.push_back(x);
28
29     // calling function do it
30     foo(stringtoneighbours, pathsofar_temp, foundpath, endGene, finaloutput);
31 }
32 }
33
34 // main-file =====
35 int main(){
36
37     // starting timer
38     Timer timer;
39
40     // input- configuration
41     string startGene    = "AACCGGTT";
42     string endGene      = "AACCGGTA";
43     vector<string> bank  = {"AACCGGTA"};
44
45     // setup
46     unordered_map<string, vector<string>> stringtoneighbours;
47
48     // checking if endgene is in the bank
49     if (std::find(bank.begin(),
50                 bank.end(),
51                 endGene) == bank.end()) {cout << format("finalOutput = -1\n");}
52
53     // going through the bank and building neighbours
54     bank.push_back(startGene);
55
56     for(auto x: bank){
57
58         // finding valid transactions with the other strings in the bank

```

```

59     for(auto y: bank){
60
61         // checking number of string differences between the two
62         auto count = 0;
63         for(int i=0; i<8; ++i) {if (x[i] != y[i]) ++count;}
64
65         // checking count and adding to valid transactions
66         if (count == 1)      {stringtoneighbours[x].push_back(y);}
67     }
68
69     // checking if this particular gene can jump to final gene
70     // checking number of string differences between the two
71     auto count = 0;
72     for(int i =0; i<8; ++i) {if (x[i] != endGene[i]) ++count;}
73
74     // checking count and adding to valid transactions
75     if (count == 1)      {stringtoneighbours[x].push_back(endGene);}
76 }
77
78 // recursion
79 vector<string> pathsofar {startGene};
80 bool          foundpath {false};
81 int           finalOutput {-1};
82
83 // calling the function
84 foo(stringtoneighbours, pathsofar, foundpath, endGene, finalOutput);
85
86 // returning the finaloutput
87 if (finalOutput > -1) --finalOutput;
88
89 // printing final output
90 cout << format("finalOutput = {}\n", finalOutput);
91
92 // return
93 return(0);

```

94

95

}

## 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 }
```

---

## 448. Find All Numbers Disappeared in an Array

Given an array `nums` of `n` integers where `nums[i]` is in the range `[1, n]`, return an array of all the integers in the range `[1, n]` that do not appear in `nums`.

### Examples

#### 1. Example 1:

- Input: `nums = [4,3,2,7,8,2,3,1]`
- Output: `[5,6]`

#### 2. Example 2:

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

### Constraints

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

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums {vector<int>{1,1}};
8
9     // setup
10    auto flags {vector<bool>(nums.size(), false)};
11
12    // going through the array
13    for(const auto& x: nums) {flags[x-1] = true;}
14
15    // building final-output
16    auto finaloutput {vector<int>{}};
17    for(int i = 0; i<flags.size(); ++i) {if (flags[i] == false) {finaloutput.push_back(i+1);}}
18
19    // printing
20    cout << format("final-output = {}\n", finaloutput);
21
22    // return
23    return(0);
24
25 }
```

---

## 452. Minimum Number of Arrows to Burst Balloons

There are some spherical balloons taped onto a flat wall that represents the XY-plane. The balloons are represented as a 2D integer array points where points[i] = [xstart, xend] denotes a balloon whose horizontal diameter stretches between xstart and xend. You do not know the exact y-coordinates of the balloons.

Arrows can be shot up directly vertically (in the positive y-direction) from different points along the x-axis. A balloon with xstart and xend is burst by an arrow shot at x if  $xstart \leq x \leq xend$ . There is no limit to the number of arrows that can be shot. A shot arrow keeps traveling up infinitely, bursting any balloons in its path.

Given the array points, return the minimum number of arrows that must be shot to burst all balloons.

### Examples

#### 1. Example 1:

- Input: points = [[10,16],[2,8],[1,6],[7,12]]
- Output: 2
- Explanation: The balloons can be burst by 2 arrows:
  - - Shoot an arrow at x = 6, bursting the balloons [2,8] and [1,6].
  - - Shoot an arrow at x = 11, bursting the balloons [10,16] and [7,12].

#### 2. Example 2:

- Input: points = [[1,2],[3,4],[5,6],[7,8]]
- Output: 4
- Explanation: One arrow needs to be shot for each balloon for a total of 4 arrows.

#### 3. Example 3:

- Input: points = [[1,2],[2,3],[3,4],[4,5]]
- Output: 2
- Explanation: The balloons can be burst by 2 arrows:
  - - Shoot an arrow at x = 2, bursting the balloons [1,2] and [2,3].
  - - Shoot an arrow at x = 4, bursting the balloons [3,4] and [4,5].

## Constraints

- $1 \leq \text{points.length} \leq 10^5$
- $\text{points}[i].\text{length} == 2$
- $-2^31 \leq \text{xstart} \leq \text{xend} \leq 2^31 - 1$

## Code

---

```

1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     vector<vector<int>> points{
8         {10,16},
9         {2,8},
10        {1,6},
11        {7,12}
12    };
13
14    // setup

```

```

15 vector< vector<int> > intersections;
16
17 // sorting intervals in case they aren't
18 std::sort(points.begin(), \
19           points.end(), \
20           [](const vector<int>& a, const vector<int>& b){
21               return a[0]<b[0];});
22
23 // pushing the first interval to the intersections list
24 intersections.push_back(points[0]);
25 int finaloutput = 0;
26
27 // going through arrays one by one
28 for(int i = 1; i<points.size(); ++i){
29
30     // fetching reference to running intersection
31     auto& runningInterval = intersections[intersections.size()-1];
32
33     // fetching incoming interval
34     auto incomingInterval = points[i];
35
36     // finding intersections
37     if (runningInterval[1] < incomingInterval[0]){
38         // no intersection,
39         ++finaloutput; intersections.push_back(incomingInterval);
40     }
41     else{
42         // calculating max of start-point
43         runningInterval[0] = max(runningInterval[0], incomingInterval[0]);
44
45         // calculating min of end-points
46         runningInterval[1] = min(runningInterval[1], incomingInterval[1]);
47     }
48 }
49

```



```
50 // returning results
51 cout << format("final-output = {}\n", intersections.size());
52
53 // return
54 return(0);
55
56 }
```

---

## 463. Island Perimeter

You are given row x col grid representing a map where  $\text{grid}[i][j] = 1$  represents land and  $\text{grid}[i][j] = 0$  represents water.

Grid cells are connected horizontally/vertically (not diagonally). The grid is completely surrounded by water, and there is exactly one island (i.e., one or more connected land cells).

The island doesn't have "lakes", meaning the water inside isn't connected to the water around the island. One cell is a square with side length 1. The grid is rectangular, width and height don't exceed 100. Determine the perimeter of the island.

### Examples

#### 1. Example 1:

- Input:  $\text{grid} = [[0,1,0,0],[1,1,1,0],[0,1,0,0],[1,1,0,0]]$
- Output: 16
- Explanation: The perimeter is the 16 yellow stripes in the image above.

#### 2. Example 2:

- Input:  $\text{grid} = [[1]]$
- Output: 4

#### 3. Example 3:

- Input:  $\text{grid} = [[1,0]]$
- Output: 4

## Constraints

- `row == grid.length`
- `col == grid[i].length`
- $1 \leq \text{row}, \text{col} \leq 100$
- `grid[i][j]` is 0 or 1.
- There is exactly one island in grid.

## Code

---

```
1 bool fCheckCellValidity(const vector<vector<int>>& nums,
2                       const vector<vector<bool>>& flags,
3                       vector<int> currposition,
4                       const std::deque<vector<int>>& pipe){
5
6     // splitting tid into row and column
7     auto& rowcurr {currposition[0]};
8     auto& colcurr {currposition[1]};
9
10    // checking validity
11    if (rowcurr < 0 || rowcurr >= nums.size()) {return false;}
12    if (colcurr < 0 || colcurr >= nums[0].size()) {return false;}
13
14    // checking visitors register
15    if (flags[rowcurr][colcurr] == true) {return false;}
16
17    // checking if this position is an island or water
18    if (nums[rowcurr][colcurr] == 0) {return false;}
19
```

```

20 // returning false if the currposition is already in the pipe
21 if (std::find(pipe.begin(),
22             pipe.end(),
23             currposition) != pipe.end()) {return false;}
24
25 // returning true
26 return true;
27
28 }
29 bool fCheckBounds(const vector<vector<int>>& nums,
30                 const int rowcurr,
31                 const int colcurr){
32
33 // checking validity
34 if (rowcurr < 0 || rowcurr >= nums.size()) {return false;}
35 if (colcurr < 0 || colcurr >= nums[0].size()) {return false;}
36
37 // returning true
38 return true;
39
40 }
41 void foo(const vector<vector<int>>& nums,
42         vector<vector<bool>>& flags,
43         vector<int> currposition,
44         int& runningperimeter,
45         std::deque<vector<int>> pipe){
46
47 // splitting tid into row and column
48 auto rowcurr {currposition[0]};
49 auto colcurr {currposition[1]};
50
51 // checking if current cell has already been visited
52 if (nums[rowcurr][colcurr] == 0 || flags[rowcurr][colcurr] == true) {return;}
53
54 // setting the current to true

```

```

55 flags[rowcurr][colcurr] = true;
56
57 // visiteds result in reduced boundaries
58 auto amounttosubtract {0};
59 if(fCheckBounds(nums, rowcurr, colcurr+1) && flags[rowcurr][colcurr+1] == true) {++amounttosubtract;}
60 if(fCheckBounds(nums, rowcurr-1, colcurr) && flags[rowcurr-1][colcurr] == true) {++amounttosubtract;}
61 if(fCheckBounds(nums, rowcurr, colcurr-1) && flags[rowcurr][colcurr-1] == true) {++amounttosubtract;}
62 if(fCheckBounds(nums, rowcurr+1, colcurr) && flags[rowcurr+1][colcurr] == true) {++amounttosubtract;}
63
64 // unvisited cells add to boundary
65 auto amounttoadd {0};
66 if(fCheckBounds(nums, rowcurr, colcurr+1) == false || (fCheckBounds(nums, rowcurr, colcurr+1) &&
67     flags[rowcurr][colcurr+1] == false) ) {++amounttoadd;}
68 if(fCheckBounds(nums, rowcurr-1, colcurr) == false || (fCheckBounds(nums, rowcurr-1, colcurr) &&
69     flags[rowcurr-1][colcurr] == false) ) {++amounttoadd;}
70 if(fCheckBounds(nums, rowcurr, colcurr-1) == false || (fCheckBounds(nums, rowcurr, colcurr-1) &&
71     flags[rowcurr][colcurr-1] == false) ) {++amounttoadd;}
72 if(fCheckBounds(nums, rowcurr+1, colcurr) == false || (fCheckBounds(nums, rowcurr+1, colcurr) &&
73     flags[rowcurr+1][colcurr] == false) ) {++amounttoadd;}
74
75 // updating the running perimeter
76 runningperimeter += amounttoadd - amounttosubtract;
77
78 // // if next is one, has not been visited, we add to the stack
79 if(fCheckCellValidity(nums, flags, {rowcurr, colcurr+1}, pipe)) {pipe.push_back({rowcurr, colcurr+1});}
80 if(fCheckCellValidity(nums, flags, {rowcurr-1, colcurr}, pipe)) {pipe.push_back({rowcurr-1, colcurr});}
81 if(fCheckCellValidity(nums, flags, {rowcurr, colcurr-1}, pipe)) {pipe.push_back({rowcurr, colcurr-1});}
82 if(fCheckCellValidity(nums, flags, {rowcurr+1, colcurr}, pipe)) {pipe.push_back({rowcurr+1, colcurr});}
83
84 // sending it back if thing is empty
85 if (pipe.size() == 0) {return;}
86
87 // popping one from the middle
88 auto frontentry = pipe.front(); pipe.pop_front();

```

```

86 // calling the function on the next
87 foo(nums,
88     flags,
89     frontentry,
90     runningperimeter,
91     pipe);
92
93 }
94
95 int main(){
96
97     // starting timer
98     Timer timer;
99
100    // input- configuration
101    vector<vector<int>> nums {
102        {0,1,0,0},
103        {1,1,1,0},
104        {0,1,0,0},
105        {1,1,0,0}
106    };
107
108    // setup
109    vector<vector<bool>> flags(nums.size(),
110                               vector<bool>(nums[0].size(),
111                                              false));
112    auto runningperimeter {0};
113    std::deque<vector<int>> pipe;
114
115    // setup
116    for(int i = 0; i<nums.size(); ++i){
117
118        // finding where the number one is
119        auto it {std::find(nums[i].begin(),
120                           nums[i].end(),

```

```
121         1});
122
123     // continuing if there is no ones here
124     if (it == nums[i].end()) {continue;}
125
126     // finding the starting point
127     int startingcol = std::distance(nums[i].begin(), it);
128
129     // launching a search
130     foo(nums,
131         flags,
132         {i,startingcol},
133         runningperimeter,
134         pipe);
135
136     // breaking
137     break;
138 }
139
140 // printing the final output
141 cout << format("final-output = {}\n", runningperimeter);
142
143 // return
144 return(0);
145
146 }
```

---

## 485. Max Consecutive Ones

Given a binary array `nums`, return the maximum number of consecutive 1's in the array.

### Examples

#### 1. Example 1:

- Input: `nums = [1,1,0,1,1,1]`
- Output: 3
- Explanation: The first two digits or the last three digits are consecutive 1s. The maximum number of consecutive 1s is 3.

#### 2. Example 2:

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

### Constraints

- $1 \leq \text{nums.length} \leq 10^5$
- `nums[i]` is either 0 or 1.

### Code



---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums {vector<int>{1,1,0,1,1,1}};
8
9     // setup
10    auto p1      {-1};
11    auto finaloutput {-1};
12
13    // going through the nums
14    for(int i = 0; i<nums.size(); ++i){
15        if (nums[i] == 1){
16            if (p1 == -1) {p1 = i;}           // updating p1
17            auto temp {i - p1 + 1};          // calculating size so far
18            finaloutput = std::max(temp, finaloutput); // updating finaloutput
19        }
20        else {p1 = -1;}
21    }
22
23    // printing the final output
24    cout << format("final-output = {}\n", finaloutput);
25
26    // return
27    return(0);
28
29 }
```

---

## 530. Minimum Absolute Difference in BST

Given the root of a Binary Search Tree (BST), return the minimum absolute difference between the values of any two different nodes in the tree.

### Constraints

- The number of nodes in the tree is in the range  $[2, 10^4]$ .
- $0 \leq \text{Node.val} \leq 105$

### Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto root      {new TreeNode(4)};
8     root->left      = new TreeNode(2);
9     root->right     = new TreeNode(6);
10    root->left->left  = new TreeNode(1);
11    root->left->right = new TreeNode(3);
12
13    // setup
14    vector<int> nodevalues;
15    std::function<void(const TreeNode*)> foo = [&foo, &nodevalues](
16        const TreeNode* root){
17        // returning
```

```

18     if (root == nullptr) return;
19
20     // going down left
21     if (root->left) {foo(root->left);}
22
23     // adding current-value
24     nodevalues.push_back(root->val);
25
26     // going down right
27     if (root->right) {foo(root->right);}
28
29     // returning
30     return;
31 };
32
33 // calling function
34 foo(root);
35
36 // moving through node values
37 auto minvalue {std::numeric_limits<int>::max()};
38 auto temp      {-1};
39 for(int i = 0; i<nodevalues.size()-1; ++i){
40     // checking difference
41     temp      = std::abs(nodevalues[i]- nodevalues[i+1]);
42     minvalue  = std::min(minvalue, temp);
43 }
44
45 // returning minvalue
46 cout << format("final-output = {}\n", minvalue);
47
48 // return
49 return(0);
50
51 }

```

---

## 559. Maximum Depth of N-ary Tree

Given a n-ary tree, find its maximum depth.

1. The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.
2. Nary-Tree input serialization is represented in their level order traversal, each group of children is separated by the null value (See examples).

### Constraints

- The total number of nodes is in the range  $[0, 10^4]$ .
- The depth of the n-ary tree is less than or equal to 1000.

### Code

---

```
1
2 // Definition for a Node.
3 class Node {
4 public:
5     int val;
6     vector<Node*> children;
7
8     Node() {}
9
10    Node(int _val) {
11        val = _val;
12    }
13
```

```

14     Node(int _val, vector<Node*> _children) {
15         val = _val;
16         children = _children;
17     }
18 };
19
20 int main(){
21
22     // starting timer
23     Timer timer;
24
25     // input- configuration
26     auto root {new Node(1)};
27     root->children.push_back(new Node(3));
28     root->children.push_back(new Node(2));
29     root->children.push_back(new Node(4));
30     root->children[0]->children.push_back(new Node(5));
31     root->children[0]->children.push_back(new Node(6));
32
33     // setup
34     int final_output {std::numeric_limits<int>::min()};
35
36     // defining
37     std::function<void(const Node*, int)>
38     fCalculate = [&fCalculate,
39                 &final_output](
40         const Node* curr_node,
41         int curr_level){
42
43         // adding height
44         ++curr_level;
45
46         // check if we're at the end
47         if (curr_node->children.size() == 0){
48             final_output = final_output > curr_level ? final_output : curr_level;

```

```
49     return;
50 }
51
52 // calling function on the rest of them
53 for(const auto& x: curr_node->children) {fCalculate(x,curr_level);}
54
55 // returning
56 return;
57
58 };
59
60 // calling the function
61 fCalculate(root, 0);
62
63 // printing the final-output
64 cout << format("final-output = {}\n", final_output);
65
66 // return
67 return(0);
68
69 }
```

---

## 560. Subarray Sum Equals K

Given an array of integers `nums` and an integer `k`, return the total number of subarrays whose sum equals to `k`.

A subarray is a contiguous non-empty sequence of elements within an array.

### Examples

1. **Example 1:**

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

2. **Example 2:**

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

### Constraints

- $1 \leq \text{nums.length} \leq 2 * 10^4$
- $-1000 \leq \text{nums}[i] \leq 1000$
- $-10^7 \leq k \leq 10^7$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input-configuration
7     auto nums {vector<int>{6,4,3,1}};
8     auto k     {10};
9
10    // setup
11    auto leftSum {vector<int>(nums.size())};
12    std::partial_sum(nums.begin(), nums.end(), leftSum.begin());
13
14    // calculating final-output
15    auto finaloutput {0};
16    finaloutput += std::count(leftSum.begin(), leftSum.end(), k);
17
18    // moving from left to right
19    for(int i = 0; i < leftSum.size(); ++i){
20        for(int j = i+1; j < leftSum.size(); ++j){
21            auto sumvalue {leftSum[j] - leftSum[i]};
22            if (sumvalue == k) {++finaloutput;}
23        }
24    }
25
26    // printing
27    cout << format("final-output = {}\n", finaloutput);
28
29    // return
30    return(0);
31
32 }
```

---



## 572. Subtree of Another Tree

Given the roots of two binary trees `root` and `subRoot`, return `true` if there is a subtree of `root` with the same structure and node values of `subRoot` and `false` otherwise.

A subtree of a binary tree `tree` is a tree that consists of a node in `tree` and all of this node's descendants. The `tree` could also be considered as a subtree of itself.

### Constraints

- The number of nodes in the root tree is in the range  $[1, 2000]$ .
- The number of nodes in the `subRoot` tree is in the range  $[1, 1000]$ .
- $-10^4 \leq \text{root.val} \leq 10^4$
- $-10^4 \leq \text{subRoot.val} \leq 10^4$

### Code

---

```
1 bool fAreWeTheSame(TreeNode* root, TreeNode* subroot){
2
3     // checking if root-note values are the same
4     if (root == nullptr && subroot == nullptr) return true;
5     if (root == nullptr || subroot == nullptr) return false;
6     if (root->val != subroot->val) return false;
7
8     // going through the branches
9     auto isLeftBranchEqual {fAreWeTheSame(root->left, subroot->left)};
10    auto isRightBranchEqual {fAreWeTheSame(root->right, subroot->right)};
```

```

11
12 // returning
13 return (isLeftBranchEqual && isRightBranchEqual);
14 }
15
16 void f(TreeNode* root, TreeNode* subroot, bool& logicAccum){
17
18 // base-case
19 if (root == nullptr || subroot == nullptr) return;
20
21 // checking if they have the same head-value
22 if (root->val == subroot->val)
23     logicAccum = logicAccum || fAreWeTheSame(root, subroot);
24
25 // sending it back if we found one already
26 if (logicAccum) return;
27
28 // moving onto the next
29 f(root->left, subroot, logicAccum);
30 f(root->right, subroot, logicAccum);
31
32 // return
33 return;
34
35 }
36
37 int main(){
38
39 // starting timer
40 Timer timer;
41
42 // input- configuration
43 auto root {new TreeNode(3)};
44 root->left = new TreeNode(4);
45 root->right = new TreeNode(5);

```

```
46     root->left->left  = new TreeNode(1);
47     root->left->right = new TreeNode(2);
48
49     auto subRoot      {new TreeNode(4)};
50     subRoot->left      = new TreeNode(1);
51     subRoot->right     = new TreeNode(2);
52
53     // setup
54     bool logicAccum = false;
55     f(root, subRoot, logicAccum);
56
57     // printing output
58     cout << format("final-output = {}\n", logicAccum);
59
60     // return
61     return(0);
62
63 }
```

---

## 593. Valid Square

Given the coordinates of four points in 2D space p1, p2, p3 and p4, return true if the four points construct a square.

The coordinate of a point  $p_i$  is represented as  $[x_i, y_i]$ . The input is not given in any order.

A valid square has four equal sides with positive length and four equal angles (90-degree angles).

### Examples

#### 1. Example 1:

- Input: p1 = [0,0], p2 = [1,1], p3 = [1,0], p4 = [0,1]
- Output: true

#### 2. Example 2:

- Input: p1 = [0,0], p2 = [1,1], p3 = [1,0], p4 = [0,12]
- Output: false

#### 3. Example 3:

- Input: p1 = [1,0], p2 = [-1,0], p3 = [0,1], p4 = [0,-1]
- Output: true

### Constraints

- $p1.length == p2.length == p3.length == p4.length == 2$
- $-10^4 \leq x - i, y_i \leq 10^4$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto p1    {vector<int>{0, 0}};
8     auto p2    {vector<int>{5, 0}};
9     auto p3    {vector<int>{5, 5}};
10    auto p4    {vector<int>{0, 5}};
11
12    // setup
13    auto finaloutput {false};
14    auto center      = [&p1](auto p){
15        return vector<int>{p[0] - p1[0], p[1] - p1[1]};
16    };
17    auto addvectors = [](auto p00, auto p01){
18        return vector<int>{p00[0] + p01[0], p00[1] + p01[1]};
19    };
20
21    // changing the coordinate
22    auto p1_centered {center(p1)};
23    auto p2_centered {center(p2)};
24    auto p3_centered {center(p3)};
25    auto p4_centered {center(p4)};
26
27    // checking if its parallelogram
28    auto parallelogram_check {false};
29    vector<int> first_arm, second_arm;
30    if (p4_centered == addvectors(p2_centered, p3_centered)) {
31        parallelogram_check = true;
32        first_arm           = p2_centered;
33        second_arm          = p3_centered;
```

```

34 }
35 else if (p3_centered == addvectors(p2_centered, p4_centered)) {
36     parallelogram_check = true;
37     first_arm           = p2_centered;
38     second_arm          = p4_centered;
39 }
40 else if (p2_centered == addvectors(p3_centered, p4_centered)) {
41     parallelogram_check = true;
42     first_arm           = p3_centered;
43     second_arm          = p4_centered;
44 }
45
46 // perpendicular check
47 auto perpendicular_check {false};
48 auto inner_product = std::inner_product(first_arm.begin(), first_arm.end(),
49                                         second_arm.begin(),
50                                         0);
51 perpendicular_check = inner_product == 0 ? true : false;
52
53 // length check
54 auto length_check      {false};
55 auto first_arm_length {std::accumulate(first_arm.begin(),
56                                         first_arm.end(),
57                                         0,
58                                         [](auto acc, auto argx){ return acc + argx * argx;})};
59 auto second_arm_length {std::accumulate(second_arm.begin(),
60                                         second_arm.end(),
61                                         0,
62                                         [](auto acc, auto argx){ return acc + argx * argx;})};
63 length_check = first_arm_length == second_arm_length ? true : false;
64 if (first_arm_length == 0) {length_check = false;}
65
66 // performing final output
67 if (parallelogram_check && perpendicular_check && length_check) {finaloutput = true;}
68

```

```
69
70 // printing
71 cout << format("parallelogram_check = {}\n", parallelogram_check);
72 cout << format("perpendicular_check = {}\n", perpendicular_check);
73 cout << format("length_check      = {}\n", length_check);
74 cout << format("final-output      = {}\n", finaloutput);
75
76 // return
77 return(0);
78
79 }
```

---

## 605. Can Place Flowers

You have a long flowerbed in which some of the plots are planted, and some are not. However, flowers cannot be planted in adjacent plots. Given an integer array `flowerbed` containing 0's and 1's, where 0 means empty and 1 means not empty, and an integer `n`, return `true` if `n` new flowers can be planted in the flowerbed without violating the no-adjacent-flowers rule and `false` otherwise.

### Examples

#### 1. Example 1:

- Input: `flowerbed = [1,0,0,0,1]`, `n = 1`
- Output: `true`

#### 2. Example 2:

- Input: `flowerbed = [1,0,0,0,1]`, `n = 2`
- Output: `false`

### Constraints

- $1 \leq \text{flowerbed.length} \leq 2 * 10^4$
- `flowerbed[i]` is 0 or 1.
- There are no two adjacent flowers in `flowerbed`.
- $0 \leq n \leq \text{flowerbed.length}$



## Code

---

```
1 int main(){
2
3     // input- configuration
4     vector<int> flowerbed {1, 0, 0, 0, 1};
5     int n {1};
6
7
8     // going through this
9     for(int i = 0; i<flowerbed.size(); ++i){
10         if (flowerbed[i] == 0                &&
11             (i == 0 || flowerbed[i-1] == 0)    &&
12             (i == flowerbed.size()-1) || flowerbed[i+1] == 0)
13         {
14             flowerbed[i] = 1;
15             --n;
16         }
17     }
18
19     // printing final output
20     cout << format("Possibility = {}\n", static_cast<bool>(n<=0));
21
22     // return
23     return(0);
24 }
25
```

---

## 637. Average of Levels in Binary Tree

Given the root of a binary tree, return the average value of the nodes on each level in the form of an array. Answers within 10<sup>-5</sup> of the actual answer will be accepted.

### Constraints

- The number of nodes in the tree is in the range [1, 104].
- $-2^{31} \leq \text{Node.val} \leq 2^{31} - 1$

### Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto root      {new TreeNode(3)};
8     root->left      = new TreeNode(9);
9     root->right     = new TreeNode(20);
10    root->right->left = new TreeNode(15);
11    root->right->right = new TreeNode(7);
12
13    // setup
14
15
16    // running
17    // setup
```

```

18 vector<double> levelaverages;
19 vector<int> levelcount;
20
21 std::function<void(const TreeNode*, int)> foo = [&foo,
22         &levelaverages,
23         &levelcount](
24     const TreeNode* root,
25     int currentlevel){
26
27     // checking if valid
28     if (root == nullptr) return;
29
30     // increasing level count
31     ++currentlevel;
32
33     // adding the sum to the current-value
34     while (levelaverages.size() < currentlevel+1)
35     {
36         // increasing size
37         levelaverages.push_back(0);
38         levelcount.push_back( 0);
39     }
40
41     // adding to sum
42     levelaverages[currentlevel] = \
43         levelaverages[currentlevel] * \
44         (static_cast<double>(levelcount[currentlevel])/static_cast<double>(levelcount[currentlevel]+1)) + \
45         (static_cast<double>(root->val) / (levelcount[currentlevel] + 1));
46
47     // increasing count value
48     ++levelcount[currentlevel];
49
50     // moving left
51     if (root->left) {foo(root->left, currentlevel);}
52

```

```
53     // moving right
54     if (root->right) {foo(root->right, currentlevel);}
55
56     // returning
57     return;
58
59 };
60
61
62 // going through elements
63 foo(root, -1);
64
65 // returning level averages
66 cout << format("level-averages = {}\n", levelaverages);
67
68
69 // return
70 return(0);
71
72 }
```

---

## 643. Maximum Average Subarray I

You are given an integer array `nums` consisting of `n` elements, and an integer `k`. Find a contiguous subarray whose length is equal to `k` that has the maximum average value and return this value. Any answer with a calculation error less than  $10^{-5}$  will be accepted.

### Examples

#### 1. Example 1:

- Input: `nums = [1,12,-5,-6,50,3]`, `k = 4`
- Output: 12.75000
- Explanation: Maximum average is  $(12 - 5 - 6 + 50) / 4 = 51 / 4 = 12.75$

#### 2. Example 2:

- Input: `nums = [5]`, `k = 1`
- Output: 5.00000

### Constraints:

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

## Code

---

```
1 int main(){
2
3     // input- configuration
4     vector<int> nums {1,12,-5,-6,50,3};
5     int k           {4};
6
7     // setup
8     double windowavg {std::numeric_limits<double>::min()};
9
10    // calculating the first window average
11    for(int i = 0; i<k; ++i) {windowavg += static_cast<double>(nums[i])/static_cast<double>(k);}
12    double runningavg {windowavg};
13
14    // going through the rest
15    for(int i = k; i < nums.size(); ++i){
16        // adding head
17        runningavg += static_cast<double>(nums[i])/static_cast<double>(k);
18
19        // subtracting tail
20        runningavg -= static_cast<double>(nums[i-k])/static_cast<double>(k);
21
22        // finding the bigger-value
23        windowavg = windowavg > runningavg ? windowavg : runningavg;
24    }
25
26    // printing
27    cout << format("largest-average = {}\n", windowavg);
28
29    // return
30    return(0);
31 }
```

---

## 662. Maximum Width of Binary Tree

Given the root of a binary tree, return the maximum width of the given tree.

1. The maximum width of a tree is the maximum width among all levels.
2. The width of one level is defined as the length between the end-nodes (the leftmost and rightmost non-null nodes), where the null nodes between the end-nodes that would be present in a complete binary tree extending down to that level are also counted into the length calculation.
3. It is guaranteed that the answer will in the range of a 32-bit signed integer.

### Examples

#### 1. Example 1:

- Input: root = [1,3,2,5,3,null,9]
- Output: 4
- Explanation: The maximum width exists in the third level with length 4 (5,3,null,9).

#### 2. Example 2:

- Input: root = [1,3,2,5,null,null,9,6,null,7]
- Output: 7
- Explanation: The maximum width exists in the fourth level with length 7 (6,null,null,null,null,null,7).

#### 3. Example 3:

- Input: root = [1,3,2,5]
- Output: 2
- Explanation: The maximum width exists in the second level with length 2 (3,2).

## Constraints

- The number of nodes in the tree is in the range [1, 3000].
- $-100 \leq \text{Node.val} \leq 100$

## Code

---

```

1 struct PipeContent
2 {
3     TreeNode* node;
4     unsigned long long    branch_path;
5     int                curr_level;
6 };
7
8 int main(){
9
10     // starting timer
11     Timer timer;
12
13     // input- configuration
14     auto root {new TreeNode(1)};
15     root->left = new TreeNode(3);
16     root->right = new TreeNode(2);
17     root->left->left = new TreeNode(5);
18     root->right->right = new TreeNode(9);

```



```

19 root->left->left->left = new TreeNode(6);
20 root->right->right->left = new TreeNode(7);
21
22
23 // setup
24 auto pipe {std::deque<PipeContent>()};
25 auto levels {std::vector<std::vector<unsigned long long>>()};
26
27 // filling pipe
28 pipe.push_back({root, 0, 0});
29
30 // running bfs
31 while(pipe.size() != 0)
32 {
33     // popping front
34     auto front_entry {pipe.front()}; pipe.pop_front();
35     auto& curr_node {front_entry.node};
36     auto& branch_path {front_entry.branch_path};
37     auto& curr_level {front_entry.curr_level};
38
39     // adding contents to the levels structure
40     if(curr_level == levels.size()) {levels.push_back(std::vector<unsigned long long>());}
41     levels[curr_level].push_back(branch_path);
42
43     // adding children to pipe
44     if (curr_node->left) {pipe.push_back({curr_node->left, branch_path*2 + 0, curr_level+1});}
45     if (curr_node->right) {pipe.push_back({curr_node->right, branch_path*2 + 1, curr_level + 1});}
46 }
47
48 // calculating largest size
49 auto finaloutput {std::numeric_limits<int>::min()};
50 for(const auto& x: levels){
51     // fetching numbers from branch paths
52     auto starting_point {x[0]};
53     auto ending_point {x[x.size()-1]};

```

```
54
55     // calculating final output
56     finaloutput = std::max(finaloutput, static_cast<int>(ending_point - starting_point + 1));
57 }
58
59 // printing the final output
60 cout << format("final-output = {}\n", finaloutput);
61
62 // return
63 return(0);
64
65 }
```

---

## 695. Max Area of Island

You are given an  $m \times n$  binary matrix grid. An island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

The area of an island is the number of cells with a value 1 in the island.

Return the maximum area of an island in grid. If there is no island, return 0.

### Examples

#### 1. Example 1:

- Input: grid

0	0	1	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1	1	1	0	0	0
0	1	1	0	1	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	0	1	0	1	0	0
0	1	0	0	1	1	0	0	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1	1	1	0	0	0
0	0	0	0	0	0	0	1	1	0	0	0	0

Table 1: Grid representation of the data

- Output: 6
- Explanation: The answer is not 11, because the island must be connected 4-directionally.

## 2. Example 2:

- Input: grid = [[0,0,0,0,0,0,0]]
- Output: 0

## Constraints

- $m == \text{grid.length}$
- $n == \text{grid}[i].\text{length}$
- $1 \leq m, n \leq 50$
- $\text{grid}[i][j]$  is either 0 or 1.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto grid {std::vector<std::vector<int>>({
8         {0,0,1,0,0,0,0,1,0,0,0,0,0},
9         {0,0,0,0,0,0,0,1,1,1,0,0,0},
10        {0,1,1,0,1,0,0,0,0,0,0,0,0},
11        {0,1,0,0,1,1,0,0,1,0,1,0,0},
12        {0,1,0,0,1,1,0,0,1,1,1,0,0},
13        {0,0,0,0,0,0,0,0,0,0,1,0,0},
14        {0,0,0,0,0,0,0,1,1,1,0,0,0},
```

```

15         {0,0,0,0,0,0,0,1,1,0,0,0,0}
16     }));
17
18
19     // setup
20     auto curr_island_id    {1};
21     auto global_count     {0};
22     auto pipe              {std::deque<std::vector<int>>()};
23     auto finaloutput      {std::numeric_limits<int>::min()};
24
25     // going through each values
26     for(auto row = 0; row < grid.size(); ++row){
27         for(auto col = 0; col < grid[0].size(); ++col){
28
29             // continuing if not 1
30             if (grid[row][col] != 1) {continue;}
31
32             // clearing pipes and what not
33             pipe.clear();
34             ++curr_island_id;
35             global_count = 0;
36             pipe.push_back({row, col});
37
38             // bfs
39             while(pipe.size() != 0){
40
41                 // popping top
42                 const auto front_entry {pipe.front()}; pipe.pop_front();
43                 const auto& front_row  {front_entry[0]};
44                 const auto& front_col  {front_entry[1]};
45
46                 // checks
47                 if (front_row < 0 || front_row >= grid.size()) {continue;}
48                 if (front_col < 0 || front_col >= grid[0].size()) {continue;}
49                 if (grid[front_row][front_col] != 1) {continue;}

```

```

50
51     // registering
52     grid[front_row][front_col] = curr_island_id;
53     ++global_count;
54
55     // adding neighbours to pipe
56     pipe.push_back({front_row, front_col+1});
57     pipe.push_back({front_row-1, front_col});
58     pipe.push_back({front_row, front_col-1});
59     pipe.push_back({front_row+1, front_col});
60
61 }
62
63 // updating final output
64 finaloutput = std::max(finaloutput, global_count);
65 }
66 }
67
68 // printing final output
69 cout << format("final-output = {}\n", finaloutput);
70
71 // return
72 return(0);
73
74 }

```

---

## 724. Find Pivot Index

Given an array of integers `nums`, calculate the pivot index of this array. The pivot index is the index where the sum of all the numbers strictly to the left of the index is equal to the sum of all the numbers strictly to the index's right. If the index is on the left edge of the array, then the left sum is 0 because there are no elements to the left. This also applies to the right edge of the array. Return the leftmost pivot index. If no such index exists, return -1.

### Examples

#### 1. Example 1:

- Input: `nums = [1,7,3,6,5,6]`
- Output: 3
- Explanation:
  - The pivot index is 3.
  - Left sum = `nums[0] + nums[1] + nums[2] = 1 + 7 + 3 = 11`
  - Right sum = `nums[4] + nums[5] = 5 + 6 = 11`

#### 2. Example 2:

- Input: `nums = [1,2,3]`
- Output: -1
- Explanation: There is no index that satisfies the conditions in the problem statement.

#### 3. Example 3:

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

- Output: 0
- Explanation: The pivot index is 0.
  - Left sum = 0 (no elements to the left of index 0)
  - Right sum =  $\text{nums}[1] + \text{nums}[2] = 1 + -1 = 0$

## Constraints:

- $1 \leq \text{nums.length} \leq 10^4$
- $-1000 \leq \text{nums}[i] \leq 1000$

## Code

---

```
1 int main(){
2
3     // input- configuration
4     vector<int> nums {1, 7, 3, 6, 5, 6};
5
6     // setup
7     vector<int> numsleft(nums.size(), 0);
8     vector<int> numsright(nums.size(), 0);
9     int         sumleft  {0};
10    int         sumright {0};
11    int         leftindex {0};
12    int         rightindex {0};
13
14    // runs
15    for(int i = 0; i<nums.size(); ++i){
16
17        // fetching indices
```



```

18     leftindex = i;
19     rightindex = nums.size()-1-i;
20
21     // accumulating left and right values
22     sumleft  += nums[leftindex];
23     sumright += nums[rightindex];
24
25     if (leftindex < nums.size()-1) {numsleft[leftindex+1] = sumleft;}
26     if (rightindex > 0)          {numsright[rightindex-1] = sumright;}
27 }
28
29 // finding the mid-element
30 int finaloutput {-1};
31 for(int i = 0; i<nums.size(); ++i){
32     if(numsleft[i] == numsright[i]){
33         finaloutput = i;
34         break;
35     }
36 }
37
38 // printing
39 cout << format("numsleft = "); fPrintVector(numsleft);
40 cout << format("numsright = "); fPrintVector(numsright);
41 cout << format("finaloutput = {}\n", finaloutput);
42
43 // return
44 return(0);
45
46 }

```

---

## 735. Asteroid Collision

We are given an array `asteroids` of integers representing asteroids in a row. The indices of the asteroid in the array represent their relative position in space. For each asteroid, the absolute value represents its size, and the sign represents its direction (positive meaning right, negative meaning left). Each asteroid moves at the same speed. Find out the state of the asteroids after all collisions. If two asteroids meet, the smaller one will explode. If both are the same size, both will explode. Two asteroids moving in the same direction will never meet.

### Examples

#### 1. Example 1:

- Input: `asteroids = [5,10,-5]`
- Output: `[5,10]`
- Explanation: The 10 and -5 collide resulting in 10. The 5 and 10 never collide.

#### 2. Example 2:

- Input: `asteroids = [8,-8]`
- Output: `[]`
- Explanation: The 8 and -8 collide exploding each other.

#### 3. Example 3:

- Input: `asteroids = [10,2,-5]`
- Output: `[10]`
- Explanation: The 2 and -5 collide resulting in -5. The 10 and -5 collide resulting in 10.

## Constraints

- $2 \leq \text{asteroids.length} \leq 104$
- $-1000 \leq \text{asteroids}[i] \leq 1000$
- $\text{asteroids}[i] \neq 0$

## Code

---

```
1 int main(){
2
3     // input- configuration
4     vector<int> asteroids {5, 10, -5};
5
6     // setup
7     vector<int> finaloutput;
8     int p;
9
10    // moving through the thing
11    while(p<asteroids.size())
12    {
13
14        // check current-value
15        if (finaloutput.size() == 0 || asteroids[p] > 0){
16            finaloutput.push_back(asteroids[p]); ++p;
17        }
18        else if (asteroids[p]<0){
19
20            auto topvalue = finaloutput.back();
21            if (topvalue > std::abs(asteroids[p]))    {;}
22            else if (topvalue == std::abs(asteroids[p])) {finaloutput.pop_back();}
23            else                                     {finaloutput.pop_back(); finaloutput.push_back(asteroids[p]);}
```

```

24     ++p;
25
26     // ensuring the top-value's
27     while(finaloutput.size() >= 2 && finaloutput.back() < 0){
28         auto negvalue = finaloutput.back(); finaloutput.pop_back();
29         topvalue      = finaloutput.back();
30         if (topvalue > std::abs(negvalue))    {;}
31         else if (topvalue == std::abs(negvalue)) {finaloutput.pop_back();}
32         else                                {finaloutput.pop_back(); finaloutput.push_back(negvalue);}
33     }
34 }
35
36
37 // printing the final output
38 cout << format("final-output = "); fPrintVector(finaloutput);
39
40
41 // return
42 return(0);
43 }

```

---

## 739. Daily Temperatures

Given an array of integers `temperatures` represents the daily temperatures, return an array `answer` such that `answer[i]` is the number of days you have to wait after the `i`th day to get a warmer temperature. If there is no future day for which this is possible, keep `answer[i] == 0` instead.

### Examples

1. **Example 1:**

- Input: `temperatures = [73,74,75,71,69,72,76,73]`
- Output: `[1,1,4,2,1,1,0,0]`

2. **Example 2:**

- Input: `temperatures = [30,40,50,60]`
- Output: `[1,1,1,0]`

3. **Example 3:**

- Input: `temperatures = [30,60,90]`
- Output: `[1,1,0]`

### Constraints

- $1 \leq \text{temperatures.length} \leq 10^5$
- $30 \leq \text{temperatures}[i] \leq 100$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input - configuration
7     auto temperatures {vector<int>{73,74,75,71,69,72,76,73}};
8
9     // setup
10    vector<vector<int>> tempindexstack;
11    auto finaloutput {vector<int>(temperatures.size(), 0)};
12
13    // going through the temperatures
14    for(int i = 0; i<temperatures.size(); ++i){
15
16        // fetching current-element
17        auto& curr = temperatures[i];
18
19        while (tempindexstack.size() != 0 && tempindexstack.back()[0] < curr){
20
21            auto& temp = tempindexstack.back()[0]; // fetching top-temperature
22            auto& index = tempindexstack.back()[1]; // fetching top-index
23            finaloutput[index] = (i - index); // calculating distance
24            tempindexstack.pop_back(); // popping back
25
26        }
27
28        // pushing element-index pair into the stack
29        tempindexstack.push_back(vector<int>{curr, i});
30    }
31
32    // printing the final-output
33    cout << format("final-output = {}\n", finaloutput);
```

```
34
35 // return
36 return(0);
37
38 }
```

---

## 783. Minimum Distance Between BST Nodes

Given the root of a Binary Search Tree (BST), return the minimum difference between the values of any two different nodes in the tree.

### Constraints

- The number of nodes in the tree is in the range [2, 100].
- $0 \leq \text{Node.val} \leq 10^5$

### Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto root      {new TreeNode(90)};
8     root->left      = new TreeNode(69);
9     root->left->left  = new TreeNode(49);
10    root->left->right  = new TreeNode(89);
11    root->left->left->right = new TreeNode(52);
12
13    // setup
14    auto finaloutput {std::numeric_limits<int>::max()};
15    auto values      {std::vector<int>()};
16
17    // setting up pipe
```



```

18 std::deque<TreeNode*> pipe;
19 pipe.push_back(root);
20 auto last_value {std::numeric_limits<int>::min()};
21
22 // recursive lambda
23 std::function<void(const TreeNode*, const int)> InOrderTraversal = [
24     &InOrderTraversal,
25     &values](const TreeNode* root,
26             const int parent_value)
27 {
28     const auto& curr_value {root->val};
29     if(root->left) {InOrderTraversal( root->left, curr_value);}
30     values.push_back(curr_value);
31     if(root->right) {InOrderTraversal( root->right, curr_value);}
32 };
33
34 // calling the function
35 InOrderTraversal(root, std::numeric_limits<int>::min());
36
37 // calculating final output
38 finaloutput = std::transform_reduce(
39     values.begin() + 1, values.end(),
40     values.begin(),
41     std::numeric_limits<int>::max(),
42     [](int a, int b){ return std::min(a, b); },
43     [](int x, int y){ return std::abs(x - y); }
44 );
45
46 // printing final output
47 cout << format("final-output = {}\n", finaloutput);
48
49 // return
50 return(0);
51
52 }

```

## 812. Largest Triangle Area

Given an array of points on the X-Y plane  $\text{points}$  where  $\text{points}[i] = [x_i, y_i]$ , return the area of the largest triangle that can be formed by any three different points.

### Examples

#### 1. Example 1:

- Input:  $\text{points} = [[0,0],[0,1],[1,0],[0,2],[2,0]]$
- Output: 2.00000
- Explanation: The five points are shown in the above figure. The red triangle is the largest.

#### 2. Example 2:

- Input:  $\text{points} = [[1,0],[0,0],[0,1]]$
- Output: 0.50000

### Constraints

- $3 \leq \text{points.length} \leq 50$
- $-50 \leq x_i, y_i \leq 50$
- All the given points are unique.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     vector<vector<int>> points {{0,0}, {0,1}, {1,0}, {0,2}, {2,0}};
8
9     // setup
10    double finaloutput {-1};
11    auto calculatearea = [](vector<int> points1, vector<int> points2, vector<int> points3){
12
13        // calculating the area
14        int firstcomponent    {points1[0] * (points2[1] - points3[1])};
15        int secondcomponent    {points2[0] * (points3[1] - points1[1])};
16        int thirdcomponent     {points3[0] * (points1[1] - points2[1])};
17        double finaloutput     {static_cast<double>(0.5 * std::abs(firstcomponent + secondcomponent + thirdcomponent))};
18
19        // returning
20        return finaloutput;
21    };
22
23    // going through the elements
24    for(int i = 0; i<points.size(); ++i){
25        for(int j = i+1; j<points.size(); ++j){
26            for(int k = j+1; k<points.size(); ++k){
27                finaloutput = std::max(temp, calculatearea(points[i], points[j], points[k]));
28            }
29        }
30    }
31
32    // printing area
33    cout << format("final-output = {}\n", finaloutput);
```

```
34
35 // return
36 return(0);
37
38 }
```

---

## 836. Rectangle Overlap

An axis-aligned rectangle is represented as a list  $[x1, y1, x2, y2]$ , where  $(x1, y1)$  is the coordinate of its bottom-left corner, and  $(x2, y2)$  is the coordinate of its top-right corner. Its top and bottom edges are parallel to the X-axis, and its left and right edges are parallel to the Y-axis.

Two rectangles overlap if the area of their intersection is positive. To be clear, two rectangles that only touch at the corner or edges do not overlap.

Given two axis-aligned rectangles `rec1` and `rec2`, return `true` if they overlap, otherwise return `false`.

### Examples

#### 1. Example 1:

- Input: `rec1 = [0,0,2,2]`, `rec2 = [1,1,3,3]`
- Output: `true`

#### 2. Example 2:

- Input: `rec1 = [0,0,1,1]`, `rec2 = [1,0,2,1]`
- Output: `false`

#### 3. Example 3:

- Input: `rec1 = [0,0,1,1]`, `rec2 = [2,2,3,3]`
- Output: `false`

## Constraints

- `rec1.length == 4`
- `rec2.length == 4`
- $-10^9 \leq \text{rec1}[i], \text{rec2}[i] \leq 10^9$
- `rec1` and `rec2` represent a valid rectangle with a non-zero area.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto rec1 {vector<int>{4,4,14,7}};
8     auto rec2 {vector<int>{4,3,8,8}};
9
10    // checking if all points in rec2 are to the right of xcoordinates
11    const auto& top1 {rec1[3]};
12    const auto& bottom1 {rec1[1]};
13    const auto& left1 {rec1[0]};
14    const auto& right1 {rec1[2]};
15
16    const auto& top2 {rec2[3]};
17    const auto& bottom2 {rec2[1]};
18    const auto& left2 {rec2[0]};
19    const auto& right2 {rec2[2]};
20
21    // comparing
```

```
22     bool finaloutput {true};
23     if (right1 < left2)      {finaloutput = false;}
24     else if(top1 < bottom2) {finaloutput = false;}
25     else if(bottom1 > top2) {finaloutput = false;}
26     else if(right2 < left1) {finaloutput = false;}
27
28     // printing
29     cout << format("final-output = {}\n", finaloutput);
30
31     // return
32     return(0);
33
34 }
```

---

## 867. Transpose Matrix

Given a 2D integer array matrix, return the transpose of matrix.

The transpose of a matrix is the matrix flipped over its main diagonal, switching the matrix's row and column indices.

### Examples

#### 1. Example 1:

- Input: matrix = `[[1,2,3],[4,5,6],[7,8,9]]`
- Output: `[[1,4,7],[2,5,8],[3,6,9]]`

#### 2. Example 2:

- Input: matrix = `[[1,2,3],[4,5,6]]`
- Output: `[[1,4],[2,5],[3,6]]`

### Constraints

- `m == matrix.length`
- `n == matrix[i].length`
- $1 \leq m, n \leq 1000$
- $1 \leq m * n \leq 10^5$
- $-10^9 \leq \text{matrix}[i][j] \leq 10^9$



## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto matrix {std::vector<std::vector<int>>({
8         {1,2,3},
9         {4,5,6},
10        {7,8,9},
11        {7,8,9}
12    })};
13
14    // setup canvas
15    auto canvas {std::vector<std::vector<int>>(
16        matrix[0].size(),
17        std::vector<int>(matrix.size(), 0)
18    )};
19
20    // filling the matrix
21    for(auto col = 0; col < matrix[0].size(); ++col){
22        for(auto row = 0; row < matrix.size(); ++row){
23            canvas[col][row] = matrix[row][col];
24        }
25    }
26
27    // return
28    return(0);
29
30 }
```

---

## 883. Projection Area of 3D Shapes

You are given an  $n \times n$  grid where we place some  $1 \times 1 \times 1$  cubes that are axis-aligned with the  $x$ ,  $y$ , and  $z$  axes.

Each value  $v = \text{grid}[i][j]$  represents a tower of  $v$  cubes placed on top of the cell  $(i, j)$ .

We view the projection of these cubes onto the  $xy$ ,  $yz$ , and  $zx$  planes.

A projection is like a shadow, that maps our 3-dimensional figure to a 2-dimensional plane. We are viewing the "shadow" when looking at the cubes from the top, the front, and the side.

Return the total area of all three projections.

### Examples

#### 1. Example 1:

- Input: `grid = [[1,2],[3,4]]`
- Output: 17
- Explanation: Here are the three projections ("shadows") of the shape made with each axis-aligned plane.

#### 2. Example 2:

- Input: `grid = [[2]]`
- Output: 5

#### 3. Example 3:

- Input: `grid = [[1,0],[0,2]]`
- Output: 8

## Constraints

- $n == \text{grid.length} == \text{grid}[i].\text{length}$
- $1 \leq n \leq 50$
- $0 \leq \text{grid}[i][j] \leq 50$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     vector<vector<int>> grid {
8         {1, 0},
9         {0, 2}
10    };
11
12    // setup
13    auto finaloutput {0};
14    auto xzvector {vector<int>(grid.size(), 0)};
15    auto yzvector {vector<int>(grid[0].size(), 0)};
16
17    auto maxheightvalues {vector<int>{}};
18    for(int row = 0; row<grid.size(); ++row){
19        for(int col = 0; col<grid[0].size(); ++col){
20
21            // calculating xy-counts
22            if (grid[row][col] != 0) {++finaloutput;}
23        }
```

```

24     // updating xzvector
25     xzvector[row] = std::max(xzvector[row], grid[row][col]);
26
27     // updating yzvector
28     yzvector[col] = std::max(yzvector[col], grid[row][col]);
29
30 }
31 }
32
33 // calculating sums
34 finaloutput = std::accumulate(xzvector.begin(), xzvector.end(), finaloutput);
35 finaloutput = std::accumulate(yzvector.begin(), yzvector.end(), finaloutput);
36
37 // printing the final-output
38 cout << format("final-output = {}\n", finaloutput);
39
40 // return
41 return(0);
42
43 }

```

---

## 892. Surface Area of 3D Shapes

You are given an  $n \times n$  grid where you have placed some  $1 \times 1 \times 1$  cubes. Each value  $v = \text{grid}[i][j]$  represents a tower of  $v$  cubes placed on top of cell  $(i, j)$ .

After placing these cubes, you have decided to glue any directly adjacent cubes to each other, forming several irregular 3D shapes.

Return the total surface area of the resulting shapes.

Note: The bottom face of each shape counts toward its surface area.

### Examples

#### 1. Example 1:

- Input:  $\text{grid} = [[1,2],[3,4]]$
- Output: 34

#### 2. Example 2:

- Input:  $\text{grid} = [[1,1,1],[1,0,1],[1,1,1]]$
- Output: 32

#### 3. Example 3:

- Input:  $\text{grid} = [[2,2,2],[2,1,2],[2,2,2]]$
- Output: 46

## Constraints

- $n == \text{grid.length} == \text{grid}[i].\text{length}$
- $1 \leq n \leq 50$
- $0 \leq \text{grid}[i][j] \leq 50$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     vector<vector<int>> grid {
8         {1,1,1},
9         {1,0,1},
10        {1,1,1}
11    };
12
13    // setup
14    auto finaloutput    {0};
15    auto neighbourheight {0};
16    auto currheight     {0};
17
18    // constructing lambdas
19    auto fCheckBounds = [&grid](const int& row,
20                               const int& col){
21        if (row < 0 || row >= grid.size()) {return false;}
22        if (col < 0 || col >= grid[0].size()) {return false;}
23        return true;
```

```

24 };
25 auto updatefinaloutput = [&finaloutput,
26                           &grid](const int& currheight,
27                                   const int& neighbourheight){
28     if(currheight > neighbourheight){
29         finaloutput += currheight - neighbourheight;
30     }
31 };
32 auto fCheckNextCell = [&grid,
33                        &finaloutput,
34                        &fCheckBounds,
35                        &updatefinaloutput](const int& currheight,
36                                            const int& nextrow,
37                                            const int& nextcol){
38     if (fCheckBounds(nextrow, nextcol)) {
39         auto neighbourheight = grid[nextrow][nextcol];
40         updatefinaloutput(currheight, neighbourheight);
41     }
42     else {finaloutput += currheight;}
43 };
44
45 // goign through the elements
46 for(int row = 0; row < grid.size(); ++row){
47     for(int col = 0; col < grid[0].size(); ++col){
48
49         // fetching current-height
50         currheight = grid[row][col];
51
52         // adding top surfaces
53         if (grid[row][col] != 0) {finaloutput += 2;}
54
55         // checking the four directions
56         fCheckNextCell(currheight, row, col+1);
57         fCheckNextCell(currheight, row-1, col);
58         fCheckNextCell(currheight, row, col-1);

```

```
59         fCheckNextCell(currheight, row+1, col);
60
61     }
62 }
63
64 // printing the finaloutput
65 cout << format("final-output = {}\n", finaloutput);
66
67 // return
68 return(0);
69
70 }
```

---



## 918. Maximum Sum Circular Subarray

Given a circular integer array `nums` of length `n`, return the maximum possible sum of a non-empty subarray of `nums`.

A circular array means the end of the array connects to the beginning of the array. Formally, the next element of `nums[i]` is `nums[(i + 1) % n]` and the previous element of `nums[i]` is `nums[(i - 1 + n) % n]`.

A subarray may only include each element of the fixed buffer `nums` at most once. Formally, for a subarray `nums[i]`, `nums[i + 1]`, ..., `nums[j]`, there does not exist  $i \leq k_1$ ,  $k_2 \leq j$  with  $k_1 \% n == k_2 \% n$ .

### Examples

#### 1. Example 1:

- Input: `nums = [1,-2,3,-2]`
- Output: 3
- Explanation: Subarray `[3]` has maximum sum 3.

#### 2. Example 2:

- Input: `nums = [5,-3,5]`
- Output: 10
- Explanation: Subarray `[5,5]` has maximum sum  $5 + 5 = 10$ .

#### 3. Example 3:

- Input: `nums = [-3,-2,-3]`
- Output: -2
- Explanation: Subarray `[-2]` has maximum sum -2.

## Constraints

- $n == \text{nums.length}$
- $1 \leq n \leq 3 * 10^4$
- $-3 * 10^4 \leq \text{nums}[i] \leq 3 * 10^4$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto nums = {vector<int>{1,-2,3,-2}};
8
9     // setup
10    int global_maximum = {nums[0]};
11    int global_minimum = {nums[0]};
12    int running_max = {0};
13    int running_min = {0};
14    int total = {0};
15
16    // going through the array
17    for(auto x: nums){
18
19        // calculating running maximum and minimum values
20        running_max = max(running_max + x, x);
21        running_min = min(running_min + x, x);
22
23        // running up the total value
```

```
24     total += x;
25
26     // updating global maximum and minumum values
27     global_maximum = max(global_maximum, running_max);
28     global_minimum = min(global_minimum, running_min);
29 }
30
31 // some condition
32 if (global_maximum > 0)
33     global_maximum = max(global_maximum, total - global_minimum);
34
35 // printing the maximum
36 cout << format("final-output = {}\n", global_maximum);
37
38 // return
39 return(0);
40
41 }
```

---

## 965. Univalued Binary Tree

A binary tree is uni-valued if every node in the tree has the same value. Given the root of a binary tree, return true if the given tree is uni-valued, or false otherwise.

### Examples

#### 1. Example 1:

- Input: root = [1,1,1,1,1,null,1]
- Output: true

#### 2. Example 2:

- Input: root = [2,2,2,5,2]
- Output: false

### Constraints

- The number of nodes in the tree is in the range [1, 100].
- $0 \leq \text{Node.val} < 100$

### Code

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto root {new TreeNode(2)};
8     root->left = new TreeNode(2);
9     root->right = new TreeNode(2);
10    root->left->left = new TreeNode(5);
11    root->left->right = new TreeNode(2);
12
13    // setup
14    auto pipe {std::deque<TreeNode*>()};
15    pipe.push_back(root);
16    const auto& root_value {root->val};
17    auto final_output {true};
18
19    // bfs
20    while (pipe.size() != 0)
21    {
22        // popping front-value
23        const auto& curr_root {pipe.front()};
24        pipe.pop_front();
25        const auto& curr_value {curr_root->val};
26
27        // checking if current-value is different from root-value
28        if (curr_value != root_value) {final_output = false; break;}
29
30        // adding children to pipe
31        if(curr_root->left) {pipe.push_back(curr_root->left);}
32        if(curr_root->right) {pipe.push_back(curr_root->right);}
33    }
34}
```

```
35 // return
36 return(0);
37 }
```

---

## 973. K Closest Points to Origin

Given an array of points where  $\text{points}[i] = [x_i, y_i]$  represents a point on the X-Y plane and an integer  $k$ , return the  $k$  closest points to the origin  $(0, 0)$ .

The distance between two points on the X-Y plane is the Euclidean distance.

You may return the answer in any order. The answer is guaranteed to be unique (except for the order that it is in).

### Examples

#### 1. Example 1:

- Input:  $\text{points} = [[1,3],[-2,2]]$ ,  $k = 1$
- Output:  $[-2,2]$
- Explanation:
  - The distance between  $(1, 3)$  and the origin is  $\sqrt{10}$ .
  - The distance between  $(-2, 2)$  and the origin is  $\sqrt{8}$ .
  - Since  $\sqrt{8} < \sqrt{10}$ ,  $(-2, 2)$  is closer to the origin.
  - We only want the closest  $k = 1$  points from the origin, so the answer is just  $[-2,2]$ .

#### 2. Example 2:

- Input:  $\text{points} = [[3,3],[5,-1],[-2,4]]$ ,  $k = 2$
- Output:  $[[3,3],[-2,4]]$
- Explanation: The answer  $[-2,4],[3,3]$  would also be accepted.

## Constraints

- $1 \leq k \leq \text{points.length} \leq 10^4$
- $-10^4 \leq x_i, y_i \leq 10^4$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     vector<vector<int>> points = {
8         {3,3},
9         {5,-1},
10        {-2,4}
11    };
12    auto k {2};
13
14    // setup
15    auto distance = [](auto argx){return std::sqrt(argx[0]*argx[0] + argx[1]*argx[1]);};
16
17    // building heap
18    vector<std::pair<int, double>> indexToDistance;
19
20    // Basis on which the heap is built
21    auto heap_comparator = [](const auto& pair_a, const auto& pair_b){
22        return pair_a.second < pair_b.second;
23    };
24
25    // making heap
```



```

26 std::make_heap(indexToDistance.begin(), indexToDistance.end(), heap_comparator);
27
28 // lambda-function for pushing to heap
29 auto pushHeap = [&indexToDistance, &heap_comparator, &k](auto curr_pair){
30
31     indexToDistance.push_back(curr_pair);
32     std::push_heap(indexToDistance.begin(), indexToDistance.end(), heap_comparator);
33
34     if (indexToDistance.size() > k){
35         std::pop_heap(indexToDistance.begin(), indexToDistance.end(), heap_comparator);
36         indexToDistance.pop_back();
37     }
38 };
39
40 // going through the elements
41 for(int i = 0; i<points.size(); ++i){
42
43     // making current-entry
44     auto curr {std::pair{i, distance(points[i])}};
45
46     // adding element to the heap
47     pushHeap(curr);
48 }
49
50 // building the final output
51 auto finaloutput {vector<vector<int>>{}};
52 for(int i = 0; i<indexToDistance.size(); ++i)
53     finaloutput.push_back(points[indexToDistance[i].first]);
54
55 // printing the final-output
56 cout << format("final-output = {}\n", finaloutput);
57
58 // return
59 return(0);
60

```



## 988. Smallest String Starting From Leaf

You are given the root of a binary tree where each node has a value in the range  $[0, 25]$  representing the letters 'a' to 'z'.

Return the lexicographically smallest string that starts at a leaf of this tree and ends at the root.

As a reminder, any shorter prefix of a string is lexicographically smaller.

For example, "ab" is lexicographically smaller than "aba". A leaf of a node is a node that has no children.

### Examples

#### 1. Example 1:

- Input: root = [0,1,2,3,4,3,4]
- Output: "dba"

#### 2. Example 2:

- Input: root = [25,1,3,1,3,0,2]
- Output: "adz"

#### 3. Example 3:

- Input: root = [2,2,1,null,1,0,null,0]
- Output: "abc"

## Constraints

- The number of nodes in the tree is in the range [1, 8500].
- $0 \leq \text{Node.val} \leq 25$

## Code

---

```
1 string fCompareStrings(const string& runningstring,
2                       const string& finaloutput)
3 {
4
5     // copying values
6     auto a {runningstring};
7     auto b {finaloutput};
8
9     // init
10    if (finaloutput.size() == 0) {return runningstring;}
11
12    // going through characters one after the other
13    auto numminchars = std::min(a.size(), b.size());
14
15    for(int i = 0; i<numminchars; ++i){
16        if (a[i] < b[i])        {return a;}
17        else if (a[i] > b[i])    {return b;}
18        else                    {continue;}
19    }
20
21    // if we're here, it means that the first minchars are the same
22    if (a.size() < b.size())    {return a;}
23    else                        {return b;}
24
25 }
```

```

26 void foo(TreeNode* root,
27         string  runningstring,
28         string&  finaloutput){
29
30     // sending it back
31     if (root == nullptr) return;
32
33     // adding current character to string
34     runningstring = std::string(1, static_cast<char>(root->val + 'a')) + runningstring;
35
36     // calling the function on the two children
37     if (root->left == nullptr && root->right == nullptr){
38         finaloutput = fCompareStrings(runningstring, finaloutput);
39     }
40     else{
41         // calling the children
42         foo(root->left,  runningstring, finaloutput);
43         foo(root->right, runningstring, finaloutput);
44     }
45
46     // returning
47     return;
48 }
49
50 int main(){
51
52     // starting timer
53     Timer timer;
54
55     // input- configuration
56     auto root      {new TreeNode(0)};
57     root->left      = new TreeNode(1);
58     root->right     = new TreeNode(2);
59
60     root->left->left = new TreeNode(3);

```

```
61     root->left->right = new TreeNode(4);
62
63     root->right->left = new TreeNode(3);
64     root->right->right = new TreeNode(4);
65
66
67     // setup
68     string finaloutput = "";
69
70     // calling the function
71     foo(root, "", finaloutput);
72
73     // returning finaloutput
74     cout << format("final-output = {}\n", finaloutput);
75
76     // return
77     return(0);
78
79 }
```

---

## 1004. Max Consecutive Ones III

Given a binary array `nums` and an integer `k`, return the maximum number of consecutive 1's in the array if you can flip at most `k` 0's.

### Examples

#### 1. Example 1:

- Input: `nums = [1,1,1,0,0,0,1,1,1,0]`, `k = 2`
- Output: 6
- Explanation: `[1,1,1,0,0,1,1,1,1,0]` Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.

#### 2. Example 2:

- Input: `nums = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,1,1,1,1]`, `k = 3`
- Output: 10
- Explanation: `[0,0,1,1,1,1,1,1,1,1,0,0,1,1,1,1]` Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.

### Constraints

- $1 \leq \text{nums.length} \leq 10^5$
- `nums[i]` is either 0 or 1.
- $0 \leq k \leq \text{nums.length}$

## Code

---

```
1 int main(){
2
3     // input-configuration
4     vector<int> nums {0, 0, 0, 1};
5     int k {4};
6
7     // seutp
8     int zerocounter {0};
9     int p1 {0};
10    int running {-1};
11    int maxlength {-1};
12
13    // running
14    for (int i = 0; i<nums.size(); ++i){
15        // incrementing zeros
16        if (nums[i] == 0) {++zerocounter;}
17
18        // printing
19        cout << format("substring = ");
20        fPrintVector(std::vector<int>(nums.begin() + p1, nums.begin() + i + 1));
21
22        // moving tail
23        if (zerocounter == k+1){
24            while(nums[p1] != 0) {++p1;}
25            ++p1;
26            --zerocounter;
27        }
28
29        // assessing lengths
30        running = i - p1+1;
31        maxlength = running > maxlength ? running : maxlength;
32    }
33}
```



```
34 // printing
35 cout << format("max-length = {}\n", maxlength);
36
37 // return
38 return(0);
39
40 }
```

---

## 1037. Valid Boomerang

Given an array points where  $\text{points}[i] = [x_i, y_i]$  represents a point on the X-Y plane, return true if these points are a boomerang.

A boomerang is a set of three points that are all distinct and not in a straight line.

### Examples

#### 1. Example 1:

- Input: points = `[[1,1],[2,3],[3,2]]`
- Output: true

#### 2. Example 2:

- Input: points = `[[1,1],[2,2],[3,3]]`
- Output: false

### Constraints

- `points.length == 3`
- `points[i].length == 2`
- $0 \leq x_i, y_i \leq 100$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     vector<vector<int>> points {
8         {82,23},
9         {98,88},
10        {94,72}
11    };
12
13    // setup
14    vector<int> vecAB {points[1][0] - points[0][0], points[1][1] - points[0][1]};
15    vector<int> vecAC {points[2][0] - points[0][0], points[2][1] - points[0][1]};
16
17    // lambda
18    auto fCalculateInnerProduct = [](const vector<int>& vecAB,
19                                     const vector<int>& vecAC){
20        auto modAB {std::sqrt(vecAB[0]*vecAB[0] + vecAB[1]*vecAB[1])};
21        auto modAC {std::sqrt(vecAC[0]*vecAC[0] + vecAC[1]*vecAC[1])};
22
23        if (modAB == 0 || modAC == 0) {return static_cast<double>(1);}
24
25        auto dotproduct {vecAB[0] * vecAC[0] + vecAB[1] * vecAC[1]};
26        return dotproduct/(modAB * modAC);
27    };
28
29    // calculating innerproduct
30    auto innerproduct = fCalculateInnerProduct(vecAB, vecAC);
31
32    // checking
33    auto finaloutput {true};
```

```
34     if (std::abs(1 - std::abs(innerproduct)) < 1e-7) {finaloutput = false;}
35
36     // printing the finaloutput
37     cout << format("final-output = {}\n", finaloutput);
38
39     // return
40     return(0);
41
42 }
```

---

## 1091. Shortest Path in Binary Matrix

Given an  $n \times n$  binary matrix grid, return the length of the shortest clear path in the matrix. If there is no clear path, return -1.

A clear path in a binary matrix is a path from the top-left cell (i.e., (0, 0)) to the bottom-right cell (i.e., (n - 1, n - 1)) such that:

- All the visited cells of the path are 0.
- All the adjacent cells of the path are 8-directionally connected (i.e., they are different and they share an edge or a corner).
- The length of a clear path is the number of visited cells of this path.

### Examples

#### 1. Example 1:

- Input: grid = `[[0,1],[1,0]]`
- Output: 2

#### 2. Example 2:

- Input: grid = `[[0,0,0],[1,1,0],[1,1,0]]`
- Output: 4

#### 3. Example 3:

- Input: grid = `[[1,0,0],[1,1,0],[1,1,0]]`
- Output: -1

## Constraints

- $n == \text{grid.length}$
- $n == \text{grid}[i].\text{length}$
- $1 \leq n \leq 100$
- $\text{grid}[i][j]$  is 0 or 1

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto grid {std::vector<std::vector<int>>>({
8         {0,0,0},
9         {1,1,0},
10        {1,1,0}
11    })} ;
12
13    // setting up shortest paths
14    auto spath {std::vector<std::vector<int>>>(
15        grid.size(),
16        std::vector<int>(grid[0].size(), std::numeric_limits<int>::max())
17    )};
18
19    // running bfs
20    auto pipe {std::deque<std::vector<int>>>()};
21    pipe.emplace_back(std::vector<int>({0,0,1}));
```

```

22
23
24 // lambda for checking validity
25 auto fCheckValidity = [&grid,&spath](auto row,auto col){
26     if (row < 0 || row >= grid.size()) {return false;}
27     if (col < 0 || col >= grid[0].size()) {return false;}
28     if (grid[row][col] == 1) {return false;}
29     return true;
30 };
31
32 // running bfs
33 while (pipe.size() != 0 && fCheckValidity(0,0))
34 {
35
36     // popping vector from the front
37     const auto front_entry {pipe.front()}; pipe.pop_front();
38     const auto& row {front_entry[0]};
39     const auto& col {front_entry[1]};
40     const auto& pathlength {front_entry[2]};
41
42     // checking the spath
43     if (pathlength >= spath[row][col]) {continue;}
44     else {spath[row][col] = pathlength;}
45
46     // adding the neighbours to the path
47     if(fCheckValidity(row, col+1)) {pipe.push_back(std::vector<int>({row, col+1, pathlength + 1}));}
48     if(fCheckValidity(row-1, col+1)) {pipe.push_back(std::vector<int>({row-1, col+1, pathlength + 1}));}
49     if(fCheckValidity(row-1, col)) {pipe.push_back(std::vector<int>({row-1, col, pathlength + 1}));}
50     if(fCheckValidity(row-1, col-1)) {pipe.push_back(std::vector<int>({row-1, col-1, pathlength + 1}));}
51
52     if(fCheckValidity(row, col-1)) {pipe.push_back(std::vector<int>({row, col-1, pathlength + 1}));}
53     if(fCheckValidity(row+1, col-1)) {pipe.push_back(std::vector<int>({row+1, col-1, pathlength + 1}));}
54     if(fCheckValidity(row+1, col)) {pipe.push_back(std::vector<int>({row+1, col, pathlength + 1}));}
55     if(fCheckValidity(row+1, col+1)) {pipe.push_back(std::vector<int>({row+1, col+1, pathlength + 1}));}
56 }

```

```
57
58 // printing finaloutput
59 auto finaloutput {spath[spath.size()-1][spath[0].size()-1]};
60 finaloutput = finaloutput == std::numeric_limits<int>::max() ? -1 : finaloutput;
61
62 // printing the final output
63 cout << format("final-output = {}\n", finaloutput);
64
65 // return
66 return(0);
67
68 }
```

---



## 1143. Longest Common Subsequence

Given two strings `text1` and `text2`, return the length of their longest common subsequence. If there is no common subsequence, return 0.

A subsequence of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

For example, "ace" is a subsequence of "abcde". A common subsequence of two strings is a subsequence that is common to both strings.

### Examples

#### 1. Example 1:

- Input: `text1 = "abcde"`, `text2 = "ace"`
- Output: 3
- Explanation: The longest common subsequence is "ace" and its length is 3.

#### 2. Example 2:

- Input: `text1 = "abc"`, `text2 = "abc"`
- Output: 3
- Explanation: The longest common subsequence is "abc" and its length is 3.

#### 3. Example 3:

- Input: `text1 = "abc"`, `text2 = "def"`
- Output: 0
- Explanation: There is no such common subsequence, so the result is 0.

## Constraints

- $1 \leq \text{text1.length}, \text{text2.length} \leq 1000$
- text1 and text2 consist of only lowercase English characters.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto text1      {string("bsbininm")};
8     auto text2      {string("jmjkbkjkv")};
9
10    // setup
11    auto entryvalue  {-1};
12    vector<vector<int>> dptable;
13
14    // creating dp-table
15    for(int i = 0; i<text1.size()+1; ++i)
16        dptable.push_back(vector<int>(text2.size()+1, 0));
17
18    // filling dptable
19    for(int row = text1.size()-1; row>=0; --row){
20        for(int col = text2.size()-1; col >= 0; --col){
21
22            // checking if current-values are the same
23            if (text1[row] == text2[col]) {entryvalue = 1 + dptable[row+1][col+1];}
24            else {entryvalue = std::max(dptable[row+1][col], dptable[row][col+1]);}
25        }
```

```
26         // storing to dptable
27         dptable[row][col] = entryvalue;
28     }
29 }
30
31 // printing the dptable
32 cout << format("final-output = {}\n", dptable[0][0]);
33
34 // printing matrix
35 fPrintMatrix(dptable);
36
37 // return
38 return(0);
39
40
41 }
```

---

## 1207. Unique Number Of Occurences

Given an array of integers arr, return true if the number of occurrences of each value in the array is unique or false otherwise.

### Examples

#### 1. Example 1:

- Input: arr = [1,2,2,1,1,3]
- Output: true
- Explanation: The value 1 has 3 occurrences, 2 has 2 and 3 has 1. No two values have the same number of occurrences.

#### 2. Example 2:

- Input: arr = [1,2]
- Output: false

#### 3. Example 3:

- Input: arr = [-3,0,1,-3,1,1,1,-3,10,0]
- Output: true

### Constraints

- $1 \leq \text{arr.length} \leq 1000$
- $-1000 \leq \text{arr}[i] \leq 1000$

## Code

---

```
1 int main(){
2
3     // input- configuration
4     vector<int> arr {-3,0,1,-3,1,1,1,-3,10,0};
5
6     // building histogram
7     unordered_map<int, int> histogram;
8     for(const auto& x: arr){
9         if (histogram.find(x) == histogram.end()) {histogram[x] = 1;}
10        else                                     {++histogram[x];}
11    }
12
13    // building a set out of the histogram
14    std::set<int> uniquecounts;
15    bool finaloutput {true};
16    int uniquecounts_prevsize = 0;
17    for(const auto& x: histogram){
18        uniquecounts.insert(x.second);
19        if (uniquecounts.size() == uniquecounts_prevsize) {finaloutput = false; break;}
20        uniquecounts_prevsize = uniquecounts.size();
21    }
22
23    // printing
24    cout << format("histogram.size() = {}\n", histogram.size());
25    cout << format("finaloutput = {}\n", finaloutput);
26    cout << format("uniquecounts = "); for(const auto& x: uniquecounts) {cout << x << ", "};
27
28
29    // return
30    return(0);
31
32 }
```

---

## 1232. Check If It Is a Straight Line

You are given an array `coordinates`, `coordinates[i] = [x, y]`, where `[x, y]` represents the coordinate of a point. Check if these points make a straight line in the XY plane.

### Examples

#### 1. Example 1:

- Input: `coordinates = [[1,2],[2,3],[3,4],[4,5],[5,6],[6,7]]`
- Output: `true`

#### 2. Example 2:

- Input: `coordinates = [[1,1],[2,2],[3,4],[4,5],[5,6],[7,7]]`
- Output: `false`

### Constraints

- $2 \leq \text{coordinates.length} \leq 1000$
- `coordinates[i].length == 2`
- $-10^4 \leq \text{coordinates}[i][0], \text{coordinates}[i][1] \leq 10^4$
- `coordinates` contains no duplicate point.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     vector<vector<int>> coordinates {
8         {1,2},
9         {2,3},
10        {3,4},
11        {4,5},
12        {5,6},
13        {6,7}
14    };
15
16    // setup
17    vector<vector<int>> geometric_vectors(coordinates.size()-1);
18    std::transform(coordinates.begin()+1,
19                    coordinates.end(),
20                    geometric_vectors.begin(),
21                    [&coordinates](auto argx){return vector<int>{argx[0] - coordinates[0][0], argx[1] - coordinates[0][1]};}
22                    );
23
24    // calculating slope
25    vector<double> geometric_slopes(geometric_vectors.size());
26    std::transform(geometric_vectors.begin(),
27                    geometric_vectors.end(),
28                    geometric_slopes.begin(),
29                    [](auto argx){
30                        if (argx[0] == 0) {return std::numeric_limits<double>::max();}
31                        else {return static_cast<double>(argx[1])/static_cast<double>(argx[0]);}
32                    });
33
```

```
34 // checking if all elements are same
35 auto areallsame = std::adjacent_find(geometric_slopes.begin(),
36                                     geometric_slopes.end(),
37                                     std::not_equal_to<>()) == geometric_slopes.end();
38
39 cout << format("coordinates = {}\n", coordinates);
40 cout << format("vectors = {}\n", geometric_vectors);
41 cout << format("geometric_slops = {}\n", geometric_slopes);
42 cout << format("final-output = {}\n", areallsame);
43
44 // return
45 return(0);
46
47 }
```

---



## 1266. Minimum Time Visiting All Points

On a 2D plane, there are  $n$  points with integer coordinates  $\text{points}[i] = [x_i, y_i]$ . Return the minimum time in seconds to visit all the points in the order given by points.

You can move according to these rules:

1. In 1 second, you can either:
  - move vertically by one unit,
  - move horizontally by one unit, or
  - move diagonally  $\sqrt{2}$  units (in other words, move one unit vertically then one unit horizontally in 1 second).
2. You have to visit the points in the same order as they appear in the array.
3. You are allowed to pass through points that appear later in the order, but these do not count as visits.

### Examples

#### 1. Example 1:

- Input:  $\text{points} = [[1,1],[3,4],[-1,0]]$
- Output: 7
- Explanation: One optimal path is  $[1,1] \rightarrow [2,2] \rightarrow [3,3] \rightarrow [3,4] \rightarrow [2,3] \rightarrow [1,2] \rightarrow [0,1] \rightarrow [-1,0]$ 
  - Time from  $[1,1]$  to  $[3,4] = 3$  seconds
  - Time from  $[3,4]$  to  $[-1,0] = 4$  seconds
  - Total time = 7 seconds

#### 2. Example 2:

- Input: points = [[3,2],[-2,2]]
- Output: 5

## Constraints

- points.length == n
- $1 \leq n \leq 100$
- points[i].length == 2
- $-1000 \leq \text{points}[i][0], \text{points}[i][1] \leq 1000$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     vector<vector<int>> points {
8         {3,2},
9         {-2,2}
10    };
11
12    // setup
13    auto fCalculateDistance = [](const vector<int>& coordinate_A,
14                                const vector<int>& coordinate_B){
15
16        auto xdif          {std::abs(coordinate_A[0] - coordinate_B[0])};
```

```

17     auto ydif          {std::abs(coordinate_A[1] - coordinate_B[1])};
18     auto common        {std::min(xdif, ydif)};
19     auto distleft       {std::max(xdif, ydif) - std::min(xdif, ydif)};
20     return distleft + common;
21 };
22
23 // calculating distances
24 auto finaloutput {std::inner_product(points.begin(),
25                                     points.end()-1,
26                                     points.begin()+1,
27                                     0,
28                                     std::plus<int>(),
29                                     fCalculateDistance)};
30
31 // printing
32 cout << format("final-output = {}\n", finaloutput);
33
34 // return
35 return(0);
36
37 }

```

---

## 1379. Find a Corresponding Node of a Binary Tree in a Clone of That Tree

Given two binary trees original and cloned and given a reference to a node target in the original tree. The cloned tree is a copy of the original tree. Return a reference to the same node in the cloned tree. Note that you are not allowed to change any of the two trees or the target node and the answer must be a reference to a node in the cloned tree.

### Examples

#### 1. Example 1:

- Input: tree = [7,4,3,null,null,6,19], target = 3
- Output: 3
- Explanation: In all examples the original and cloned trees are shown. The target node is a green node from the original tree. The answer is the yellow node from the cloned tree.

#### 2. Example 2:

- Input: tree = [7], target = 7
- Output: 7

#### 3. Example 3:

- Input: tree = [8,null,6,null,5,null,4,null,3,null,2,null,1], target = 4
- Output: 4

## Constraints

- The number of nodes in the tree is in the range  $[1, 10^4]$ .
- The values of the nodes of the tree are unique.
- target node is a node from the original tree and is not null.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto original      {new TreeNode(7)};
8     original->left      = new TreeNode(4);
9     original->right     = new TreeNode(3);
10    original->right->left = new TreeNode(6);
11    original->right->right = new TreeNode(19);
12
13    auto cloned          {new TreeNode(7)};
14    cloned->left          = new TreeNode(4);
15    cloned->right         = new TreeNode(3);
16    cloned->right->left    = new TreeNode(6);
17    cloned->right->right   = new TreeNode(19);
18
19    auto target         {original->right};
20
21    // setup
22    auto finaloutput     {static_cast<TreeNode*>(nullptr)};
23    auto found           {false};
```

```

24
25 // setting up lambda
26 std::function<void(const TreeNode*, TreeNode*)>
27 seachLambda = [&seachLambda,
28               &target,
29               &found,
30               &finaloutput](
31     const TreeNode* curr_root,
32     TreeNode* curr_root_parallel
33 )
34 {
35     // quitting if already found
36     if(found == true) {return;}
37
38     // checking if current-value is target
39     if(curr_root == target) {finaloutput = curr_root_parallel; found = true; return;}
40
41     // passing control to the children
42     if (curr_root->left != nullptr) {seachLambda(curr_root->left, curr_root_parallel->left);}
43     if (curr_root->right != nullptr) {seachLambda(curr_root->right, curr_root_parallel->right);}
44
45     // returning
46     return;
47 };
48
49 // call the function
50 seachLambda(original, cloned);
51
52 // return
53 return(0);
54
55 }

```

---

## 1431. Kids With Greatest Number Of Candies

There are  $n$  kids with candies. You are given an integer array `candies`, where each `candies[i]` represents the number of candies the  $i$ th kid has, and an integer `extraCandies`, denoting the number of extra candies that you have. Return a boolean array `result` of length  $n$ , where `result[i]` is `true` if, after giving the  $i$ th kid all the `extraCandies`, they will have the greatest number of candies among all the kids, or `false` otherwise. Note that multiple kids can have the greatest number of candies.

### Examples

#### 1. Example 1:

- Input: `candies = [2,3,5,1,3]`, `extraCandies = 3`
- Output: `[true,true,true,false,true]`
- Explanation: If you give all `extraCandies` to:
  - Kid 1, they will have  $2 + 3 = 5$  candies, which is the greatest among the kids.
  - Kid 2, they will have  $3 + 3 = 6$  candies, which is the greatest among the kids.
  - Kid 3, they will have  $5 + 3 = 8$  candies, which is the greatest among the kids.
  - Kid 4, they will have  $1 + 3 = 4$  candies, which is not the greatest among the kids.
  - Kid 5, they will have  $3 + 3 = 6$  candies, which is the greatest among the kids.

#### 2. Example 2:

- Input: `candies = [4,2,1,1,2]`, `extraCandies = 1`
- Output: `[true,false,false,false]`
- Explanation: There is only 1 extra candy.
  - Kid 1 will always have the greatest number of candies, even if a different kid is given the extra candy.

#### 3. Example 3:

- Input: candies = [12,1,12], extraCandies = 10
- Output: [true,false,true]

## Constraints

- $n == \text{candies.length}$
- $2 \leq n \leq 100$
- $1 \leq \text{candies}[i] \leq 100$
- $1 \leq \text{extraCandies} \leq 50$

## Code

---

```
1 int main(){
2
3     // input- configuration
4     vector<int> candies {2,3,5,1,3};
5     int extraCandies {3};
6
7     // setup
8     auto iter = std::max_element(candies.begin(), candies.end()); // finding max value
9     vector<bool> finaloutput;
10
11    // going through all values
12    for(int i = 0; i<candies.size(); ++i){
13        if ((candies[i]+extraCandies) >= *iter) {finaloutput.push_back(true);}
14        else {finaloutput.push_back(false);}
15    }
16
```



```
17 // printing
18 cout << format("finaloutput = "); fPrintVector(finaloutput);
19
20 // return
21 return(0);
22
23 }
```

---

## 1493. Longest Subarray Of 1s After Deleting One Element

Given a binary array `nums`, you should delete one element from it. Return the size of the longest non-empty subarray containing only 1's in the resulting array. Return 0 if there is no such subarray.

### Examples

#### 1. Example 1:

- Input: `nums = [1,1,0,1]`
- Output: 3
- Explanation: After deleting the number in position 2, `[1,1,1]` contains 3 numbers with value of 1's.

#### 2. Example 2:

- Input: `nums = [0,1,1,1,0,1,1,0,1]`
- Output: 5
- Explanation: After deleting the number in position 4, `[0,1,1,1,1,1,0,1]` longest subarray with value of 1's is `[1,1,1,1,1]`.

#### 3. Example 3:

- Input: `nums = [1,1,1]`
- Output: 2
- Explanation: You must delete one element.

## Constraints

- $1 \leq \text{nums.length} \leq 10^5$
- `nums[i]` is either 0 or 1.

## Code

---

```
1 int main(){
2
3     // input- configuration
4     vector<int> nums {0,1,1,1,0,1,1,0,1};
5
6     // setup
7     int zerocounter {0};
8     int p1 {0};
9     int running {0};
10    int maxlength {0};
11
12    // running the code
13    for(int i = 0; i<nums.size(); ++i){
14
15        // incrementing zero-count
16        if(nums[i] == 0) {++zerocounter;}
17
18        // updating length
19        if (zerocounter <= 1){
20            running = i-p1;
21            maxlength = running > maxlength ? running : maxlength;
22        }
23        else {
24            // moving tail
25            while(nums[p1] != 0) {++p1;}
```

```
26         ++p1; --zerocounter;
27
28         // updating length
29         running    = i - p1;
30         maxlength  = running > maxlength ? running : maxlength;
31     }
32 }
33
34 // printing the final output
35 cout << format("maxlength = {}\n", maxlength);
36
37
38 // return
39 return(0);
40 }
```

---

## 1657. Determine If Two Strings Are Close

Two strings are considered close if you can attain one from the other using the following operations:

- Operation 1: Swap any two existing characters.
  - For example, `abcde`  $\implies$  `aecdb`
- Operation 2: Transform every occurrence of one existing character into another existing character, and do the same with the other character.
  - For example, `aacabb`  $\implies$  `bbcbba` (all a's turn into b's, and all b's turn into a's)

You can use the operations on either string as many times as necessary. Given two strings, `word1` and `word2`, return `true` if `word1` and `word2` are close, and `false` otherwise.

### Examples

#### 1. Example 1:

- Input: `word1 = "abc"`, `word2 = "bca"`
- Output: `true`
- Explanation: You can attain `word2` from `word1` in 2 operations.
  - Apply Operation 1: `"abc"`  $\rightarrow$  `"acb"`
  - Apply Operation 1: `"acb"`  $\rightarrow$  `"bca"`

#### 2. Example 2:

- Input: `word1 = "a"`, `word2 = "aa"`

- Output: false
- Explanation: It is impossible to attain word2 from word1, or vice versa, in any number of operations.

### 3. Example 3:

- Input: word1 = "cabbba", word2 = "abbccc"
- Output: true
- Explanation: You can attain word2 from word1 in 3 operations.
  - Apply Operation 1: "cabbba" -> "caabbb"
  - Apply Operation 2: "caabbb" -> "baaccc"
  - Apply Operation 2: "baaccc" -> "abbccc"

## Constraints

- $1 \leq \text{word1.length}, \text{word2.length} \leq 105$
- word1 and word2 contain only lowercase English letters.

## Code

---

```
1 int main(){
2
3     // input- configuration
4     string word1  {"cabbba"};
5     string word2  {"abbccc"};
6
7     // setup
8     bool finaloutput {false};
9 }
```

```

10 // building histogram
11 unordered_map<char, int> histogram1, histogram2;
12 for(const auto x: word1){
13     if(histogram1.find(x) == histogram1.end()) {histogram1[x] = 1;}
14     else {++histogram1[x];}
15 }
16 for(const auto x: word2){
17     if(histogram2.find(x) == histogram2.end()) {histogram2[x] = 1;}
18     else {++histogram2[x];}
19 }
20
21 // checking if one can be obtained from the other just using shuffling
22 if (histogram1 == histogram2) {
23     finaloutput = true;
24     cout << format("(same histogram) finaloutput = {}\n", finaloutput);
25     return 0;
26 }
27
28 // if number of keys are different
29 if (histogram1.size() != histogram2.size()){
30     finaloutput = false;
31     cout << format("(different number of keys) finaloutput = {}\n", finaloutput);
32     return 0;
33 }
34
35
36 // checking if keys match
37 bool keysmatch {true};
38 {
39     auto it_histogram1 = histogram1.begin();
40     auto it_histogram2 = histogram2.begin();
41     vector<char> keys1, keys2;
42     while(it_histogram1 != histogram1.end()){
43         keys1.push_back(it_histogram1->first);
44         keys2.push_back(it_histogram2->first);

```

```

45         ++it_histogram1;
46         ++it_histogram2;
47     }
48     std::sort(keys1.begin(), keys1.end());
49     std::sort(keys2.begin(), keys2.end());
50     if (keys1 == keys2) {keysmatch = true;}
51     else                 {keysmatch = false;}
52 }
53
54 // checking if counts match
55 bool countsmatch {false};
56 {
57     vector<int> counts1, counts2;
58     for(const auto& x: histogram1) {counts1.push_back(x.second);}
59     for(const auto& x: histogram2) {counts2.push_back(x.second);}
60     sort(counts1.begin(), counts1.end());
61     sort(counts2.begin(), counts2.end());
62     if (counts1 == counts2) {countsmatch = true;}
63 }
64
65 // producing the final output
66 if (keysmatch && countsmatch) {finaloutput = true;}
67 else                         {finaloutput = false;}
68
69 // return
70 return(0);
71
72 }

```

---



## 1679. Max Number Of K-Sum Pairs

You are given an integer array `nums` and an integer `k`. In one operation, you can pick two numbers from the array whose sum equals `k` and remove them from the array. Return the maximum number of operations you can perform on the array.

### Examples

#### 1. Example 1:

- Input: `nums = [1,2,3,4]`, `k = 5`
- Output: 2
- Explanation: Starting with `nums = [1,2,3,4]`:
  - Remove numbers 1 and 4, then `nums = [2,3]`
  - Remove numbers 2 and 3, then `nums = []`
  - There are no more pairs that sum up to 5, hence a total of 2 operations.

#### 2. Example 2:

- Input: `nums = [3,1,3,4,3]`, `k = 6`
- Output: 1
- Explanation: Starting with `nums = [3,1,3,4,3]`:
  - Remove the first two 3's, then `nums = [1,4,3]`
  - There are no more pairs that sum up to 6, hence a total of 1 operation.

### Constraints

- $1 \leq \text{nums.length} \leq 10^5$

- $1 \leq \text{nums}[i] \leq 10^9$
- $1 \leq k \leq 10^9$

## Code

---

```
1 int main(){
2
3     // input- configuration
4     vector<int> nums
5         {13,16,49,4,56,64,83,35,20,73,53,67,73,73,17,28,20,16,55,16,20,66,13,46,9,14,52,70,46,66,40,21,5,88,48,21,21,44,27,56,75,
6         int k          {77};
7
8     // setup
9     unordered_map<int, int> histogram;
10    int count {0};
11
12    // go through list
13    for(int i = 0; i<nums.size(); ++i){
14
15        // fetching current value
16        int curr {nums[i]};
17        int comp {k - curr};
18
19        // updating histogram
20        if (histogram.find(comp) == histogram.end() || histogram[comp] == 0)
21        {
22            // in case where complement hasn't even been entered
23            if (histogram.find(curr) != histogram.end()) {++histogram[curr];}
24            else {histogram[curr] = 1;}
25        }
26        else if(histogram[comp] > 0)
27        {
```

```
27         --histogram[comp];
28         ++count;
29     }
30 }
31
32 // printing the count
33 cout << format("count = {} \n", count);
34
35 // return
36 return(0);
37
38 }
```

---

## 1768. Merge Strings Alternately

You are given two strings `word1` and `word2`. Merge the strings by adding letters in alternating order, starting with `word1`. If a string is longer than the other, append the additional letters onto the end of the merged string. Return the merged string.

### Examples

#### 1. Example 1:

- Input: `word1 = "abc"`, `word2 = "pqr"`
- Output: `"apbqcr"`
- Explanation: The merged string will be merged as so:
  - `word1: a b c`
  - `word2: p q r`
  - `merged: a p b q c r`

#### 2. Example 2:

- Input: `word1 = "ab"`, `word2 = "pqrs"`
- Output: `"apbqrs"`
- Explanation: Notice that as `word2` is longer, `"rs"` is appended to the end.
  - `word1: a b`
  - `word2: p q r s`
  - `merged: a p b q r s`

#### 3. Example 3:

- Input: `word1 = "abcd"`, `word2 = "pq"`

- Output: "apbqcd"
- Explanation: Notice that as word1 is longer, "cd" is appended to the end.
  - word1: a b c d
  - word2: p q
  - merged: a p b q c d

## Constraints

- $1 \leq \text{word1.length}, \text{word2.length} \leq 100$
- word1 and word2 consist of lowercase English letters.

## Code

---

```
1 int main(){
2
3     // input- configuration
4     string word1 {"ab"};
5     string word2 {"pqrs"};
6
7     // setup
8     int p1 {0};
9     int p2 {0};
10    string finaloutput;
11
12    // going through
13    while(p1<word1.size() || p2 < word2.size()){
14        // pushing to final output
15        if(p1<word1.size()) {finaloutput += word1[p1++];}
16        if (p2<word2.size()) {finaloutput += word2[p2++];}
```

```
17     }
18
19     // printing the final output
20     cout << format("final-output = {}\n", finaloutput);
21
22     // return
23     return(0);
24
25 }
```

---

## 1971. Find if Path Exists in Graph

There is a bi-directional graph with  $n$  vertices, where each vertex is labeled from 0 to  $n - 1$  (inclusive). The edges in the graph are represented as a 2D integer array `edges`, where each `edges[i] = [ui, vi]` denotes a bi-directional edge between vertex `ui` and vertex `vi`. Every vertex pair is connected by at most one edge, and no vertex has an edge to itself.

You want to determine if there is a valid path that exists from vertex `source` to vertex `destination`.

Given `edges` and the integers `n`, `source`, and `destination`, return `true` if there is a valid path from `source` to `destination`, or `false` otherwise.

### Examples

#### 1. Example 1:

- Input: `n = 3`, `edges = [[0,1],[1,2],[2,0]]`, `source = 0`, `destination = 2`
- Output: `true`
- Explanation: There are two paths from vertex 0 to vertex 2:
  - $0 \rightarrow 1 \rightarrow 2$
  - $0 \rightarrow 2$

#### 2. Example 2:

- Input: `n = 6`, `edges = [[0,1],[0,2],[3,5],[5,4],[4,3]]`, `source = 0`, `destination = 5`
- Output: `false`
- Explanation: There is no path from vertex 0 to vertex 5.

## Constraints

- $1 \leq n \leq 2 * 10^5$
- $0 \leq \text{edges.length} \leq 2 * 10^5$
- $\text{edges}[i].\text{length} == 2$
- $0 \leq u_i, v_i \leq n - 1$
- $u_i \neq v_i$
- $0 \leq \text{source}, \text{destination} \leq n - 1$
- There are no duplicate edges.
- There are no self edges.

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     const auto n {3};
8     // const auto edges {std::vector<std::vector<int>>{
9     const auto edges {std::vector<std::vector<int>>{
10         {0,1},
11         {1,2},
12         {2,0}
13     }};
```



```

14  const auto    source    {0};
15  const auto    destination {2};
16
17  // building hashmap
18  unordered_map<int, vector<int>> hmap;
19
20  // going through the inputs
21  for(const auto& x: edges){
22      // adding to hashmap
23      if (hmap.find(x[0]) == hmap.end()) hmap[x[0]] = std::vector<int>({x[1]});
24      else hmap[x[0]].push_back(x[1]);
25      if (hmap.find(x[1]) == hmap.end()) hmap[x[1]] = std::vector<int>({x[0]});
26      else hmap[x[1]].push_back(x[0]);
27  }
28
29  // setting up for bfs launch
30  auto    visited    {std::vector<bool>(n, false)};
31  auto    found      {false};
32  auto    pipe        {std::deque<int>()};
33  pipe.push_back(source);
34
35  // performing bfs launch
36  while(pipe.size() != 0 && found == false){
37
38      // popping from the front
39      auto    front_object    {pipe.front()}; pipe.pop_front();
40
41      // printing
42      cout << format("curr = {} | destination = {}\n", front_object, destination);
43
44      // checking if current point is destination and updating found
45      if (front_object == destination) {found = true; break;}
46
47      // checking if the current point has been visited already. if yes, continue
48      if (visited[front_object] == true) {continue;}

```

```
49         else                                {visited[front_object] = true;}
50
51         // else add the neighbours to the pipe
52         for(const auto& x: hmap[front_object])
53             pipe.push_back(x);
54     }
55
56     // printing hashmap
57     PRINTLINE
58     cout << format("final-output = {}\n", found);
59
60
61
62
63
64     // return
65     return(0);
66
67 }
```

---

## 2101. Detonate the Maximum Bombs

You are given a list of bombs. The range of a bomb is defined as the area where its effect can be felt. This area is in the shape of a circle with the center as the location of the bomb.

The bombs are represented by a 0-indexed 2D integer array `bombs` where `bombs[i] = [xi, yi, ri]`. `xi` and `yi` denote the X-coordinate and Y-coordinate of the location of the `i`th bomb, whereas `ri` denotes the radius of its range.

You may choose to detonate a single bomb. When a bomb is detonated, it will detonate all bombs that lie in its range. These bombs will further detonate the bombs that lie in their ranges.

Given the list of bombs, return the maximum number of bombs that can be detonated if you are allowed to detonate only one bomb.

### Examples

#### 1. Example 1:

- Input: `bombs = [[2,1,3],[6,1,4]]`
- Output: 2
- Explanation:
  - The above figure shows the positions and ranges of the 2 bombs.
  - If we detonate the left bomb, the right bomb will not be affected.
  - But if we detonate the right bomb, both bombs will be detonated.
  - So the maximum bombs that can be detonated is  $\max(1, 2) = 2$ .

#### 2. Example 2:

- Input: `bombs = [[1,1,5],[10,10,5]]`
- Output: 1

- Explanation: Detonating either bomb will not detonate the other bomb, so the maximum number of bombs that can be detonated is 1.

### 3. Example 3:

- Input: bombs = [[1,2,3],[2,3,1],[3,4,2],[4,5,3],[5,6,4]]
- Output: 5
- Explanation: The best bomb to detonate is bomb 0 because:
  - - Bomb 0 detonates bombs 1 and 2. The red circle denotes the range of bomb 0.
  - - Bomb 2 detonates bomb 3. The blue circle denotes the range of bomb 2.
  - - Bomb 3 detonates bomb 4. The green circle denotes the range of bomb 3.
  - Thus all 5 bombs are detonated.

## Constraints

- $1 \leq \text{bombs.length} \leq 100$
- $\text{bombs}[i].\text{length} == 3$
- $1 \leq x_i, y_i, r_i \leq 10^5$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
```

```

7  vector<vector<int>>> bombs {
8      {1,2,3},
9      {2,3,1},
10     {3,4,2},
11     {4,5,3},
12     {5,6,4}
13 };
14
15 // setup
16 unordered_map<int, vector<int>>> treemap;
17
18 auto checkOverlap = [&treemap](const vector<int>& source,
19                               const vector<int>& destination,
20                               const int& source_index,
21                               const int& destination_index){
22
23     auto relative_vector      {vector<double>{static_cast<double>(source[0] - destination[0]),
24                                              static_cast<double>(source[1] - destination[1])}};
25     auto dist_between_centers {std::sqrt(std::inner_product(relative_vector.begin(),
26                                                             relative_vector.end(),
27                                                             relative_vector.begin(),
28                                                             0.00))};
29
30     // dist_between_centers = std::sqrt(dist_between_centers);
31
32     // adding to hashmap
33     if (dist_between_centers <= source[2]){
34         if (treemap.find(source_index) == treemap.end()) {treemap[source_index] = vector<int>{destination_index};}
35         else {treemap[source_index].push_back(destination_index);}
36     }
37     if(dist_between_centers <= destination[2]){
38         if (treemap.find(destination_index) == treemap.end()) {treemap[destination_index] = vector<int>{source_index};}
39         else {treemap[destination_index].push_back(source_index);}
40     }
41
42     return;

```

```

42
43 };
44
45 // going through the combinations
46 for(int i = 0; i<bombs.size(); ++i){
47     for(int j = i+1; j<bombs.size(); ++j){
48         checkOverlap(bombs[i], bombs[j], i, j);
49     }
50 }
51
52 // building dfs
53 int finaloutput = -1;
54 std::function<void(vector<int>&, int, int&>
55 dfs = [&dfs,
56         &treemap](vector<int>& pathsofar,
57                 int currindex,
58                 int& finaloutput)
59 {
60
61
62     // sending it back if its already visited
63     if (std::find(pathsofar.begin(), pathsofar.end(), currindex) != pathsofar.end()) {return;}
64
65     // adding current-element to path
66     pathsofar.push_back(currindex);
67     finaloutput = finaloutput > (int)pathsofar.size() ? finaloutput : (int)pathsofar.size();
68
69     // movign to the rest
70     for(const auto& x: treemap[currindex]) {dfs(pathsofar, x, finaloutput);}
71
72     // returning
73     return;
74
75 };
76

```

```
77
78 // calling the function
79 for(int i = 0; i<bombs.size(); ++i) {
80     vector<int> temp;
81     dfs(temp, i, finaloutput);
82 }
83
84 // printing the final-output
85 cout << format("final-output = {}\n",finaloutput);
86
87 // return
88 return(0);
89
90 }
```

---

## 2215. Find The Difference Of Two Arrays

Given two 0-indexed integer arrays `nums1` and `nums2`, return a list `answer` of size 2 where:

- `answer[0]` is a list of all distinct integers in `nums1` which are not present in `nums2`.
- `answer[1]` is a list of all distinct integers in `nums2` which are not present in `nums1`.

Note that the integers in the lists may be returned in any order.

### Examples

#### 1. Example 1:

- Input: `nums1 = [1,2,3]`, `nums2 = [2,4,6]`
- Output: `[[1,3],[4,6]]`
- Explanation:
  - For `nums1`, `nums1[1] = 2` is present at index 0 of `nums2`, whereas `nums1[0] = 1` and `nums1[2] = 3` are not present in `nums2`. Therefore, `answer[0] = [1,3]`.
  - For `nums2`, `nums2[0] = 2` is present at index 1 of `nums1`, whereas `nums2[1] = 4` and `nums2[2] = 6` are not present in `nums1`. Therefore, `answer[1] = [4,6]`.

#### 2. Example 2:

- Input: `nums1 = [1,2,3,3]`, `nums2 = [1,1,2,2]`
- Output: `[[3],[[]]]`



- Explanation:
  - For `nums1`, `nums1[2]` and `nums1[3]` are not present in `nums2`. Since `nums1[2] == nums1[3]`, their value is only included once and `answer[0] = [3]`.
  - Every integer in `nums2` is present in `nums1`. Therefore, `answer[1] = []`.

## Constraints

- $1 \leq \text{nums1.length}, \text{nums2.length} \leq 1000$
- $-1000 \leq \text{nums1}[i], \text{nums2}[i] \leq 1000$

## Code

---

```
1 int main(){
2
3     // input- configuration
4     vector<int> nums1 {1,2,3};
5     vector<int> nums2 {2,4,6};
6
7     // setup
8     std::set<int>      set1(nums1.begin(), nums1.end());
9     std::set<int>      set2(nums2.begin(), nums2.end());
10    vector<int>         firstinput;
11    vector<int>         secondinput;
12    vector<vector<int>> finaloutput;
13
14    // filling first entry
15    for(const auto& x: set1){
16        if(set2.find(x) == set2.end()) {firstinput.push_back(x);}
17    }
18
```

```
19 // filling second entry
20 for(const auto& x: set2){
21     if (set1.find(x) == set1.end()) {secondinput.push_back(x);}
22 }
23
24 // pushing to the final output
25 finaloutput.push_back(firstinput);
26 finaloutput.push_back(secondinput);
27
28 // printing
29 cout << format("first input = "); fPrintVector(firstinput);
30 cout << format("second input = ");fPrintVector(secondinput);
31
32 // return
33 return(0);
34 }
```

---

## 2249. Count Lattice Points Inside a Circle

Given a 2D integer array `circles` where `circles[i] = [xi, yi, ri]` represents the center  $(x_i, y_i)$  and radius  $r_i$  of the  $i$ th circle drawn on a grid, return the number of lattice points (point with integer coordinates) that are present inside or on at least one circle.

### Examples

#### 1. Example 1:

- Input: `circles = [[2,2,1]]`
- Output: 5
- Explanation:
  - The figure above shows the given circle.
  - The lattice points present inside the circle are (1, 2), (2, 1), (2, 2), (2, 3), and (3, 2) and are shown in green.
  - Other points such as (1, 1) and (1, 3), which are shown in red, are not considered inside the circle.
  - Hence, the number of lattice points present inside at least one circle is 5.

#### 2. Example 2:

- Input: `circles = [[2,2,2],[3,4,1]]`
- Output: 16
- Explanation:
  - The figure above shows the given circles.
  - There are exactly 16 lattice points which are present inside at least one circle.
  - Some of them are (0, 2), (2, 0), (2, 4), (3, 2), and (4, 4).

## Constraints

- $1 \leq \text{circles.length} \leq 200$
- $\text{circles}[i].\text{length} == 3$
- $1 \leq x_i, y_i \leq 100$
- $1 \leq r_i \leq \min(x_i, y_i)$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     vector<vector<int>> circles {
8         {2,2,2},
9         {3,4,1}
10    };
11
12    // setup
13
14    auto checkIfWithin = [](auto x_center,
15                           auto y_center,
16                           auto radius,
17                           auto x,
18                           auto y){
19
20        auto relative_coordinate {vector<int>{x - x_center, y - y_center}};
21        auto distance            {std::sqrt(std::pow(relative_coordinate[0],2)+
```

```

22         std::pow(relative_coordinate[1],2));};
23     return distance <= radius;
24 };
25
26 // going through the points
27 auto xmin {std::numeric_limits<int>::max()};
28 auto xmax {std::numeric_limits<int>::min()};
29 auto ymin {std::numeric_limits<int>::max()};
30 auto ymax {std::numeric_limits<int>::min()};
31
32 // going through the circles
33 for(const auto& x: circles){
34     xmin = std::min(xmin, x[0]-x[2]);
35     xmax = std::max(xmax, x[0]+x[2]);
36     ymin = std::min(ymin, x[1]-x[2]);
37     ymax = std::max(ymax, x[1]+x[2]);
38 }
39
40 // searching through the grid
41 auto finaloutput {0};
42 for(int i = xmin; i<=xmax; ++i){
43     for(int j = ymin; j<=ymax; ++j){
44
45         // printing the points
46         for(const auto& circle: circles){
47             auto temp00 {checkIfWithin(circle[0],
48                                     circle[1],
49                                     circle[2],
50                                     i,
51                                     j)};
52             if (temp00) {++finaloutput; break;}
53         }
54     }
55 }
56 }

```

```
57
58 // printing the final output
59 cout << format("final-output = {}\n", finaloutput);
60
61 // return
62 return(0);
63
64 }
```

---

## 2352. Equal Row And Column Pairs

Given a 0-indexed  $n \times n$  integer matrix `grid`, return the number of pairs  $(ri, cj)$  such that row  $ri$  and column  $cj$  are equal. A row and column pair is considered equal if they contain the same elements in the same order (i.e., an equal array).

### Examples

#### 1. Example 1:

- Input: `grid = [[3,2,1],[1,7,6],[2,7,7]]`
- Output: 1
- Explanation: There is 1 equal row and column pair:
  - (Row 2, Column 1): `[2,7,7]`

#### 2. Example 2:

- Input: `grid = [[3,1,2,2],[1,4,4,5],[2,4,2,2],[2,4,2,2]]`
- Output: 3
- Explanation: There are 3 equal row and column pairs:
  - (Row 0, Column 0): `[3,1,2,2]`
  - (Row 2, Column 2): `[2,4,2,2]`
  - (Row 3, Column 2): `[2,4,2,2]`

### Constraints

- $n == \text{grid.length} == \text{grid}[i].\text{length}$

- $1 \leq n \leq 200$
- $1 \leq \text{grid}[i][j] \leq 10^5$

## Code

---

```
1 int main(){
2
3     // input- configuration
4     vector<vector<int>> grid;
5     grid.push_back(vector<int>({3, 2, 1}));
6     grid.push_back(vector<int>({1, 7, 6}));
7     grid.push_back(vector<int>({2, 7, 7}));
8
9     // setup
10    int nrows {static_cast<int>(grid.size())};
11    int ncols {static_cast<int>(grid[0].size())};
12    int nelems {nrows};
13    int counts {0};
14
15    // going through the values
16
17    for(int i = 0; i<nrows; ++i){
18        for(int j = 0; j<ncols; ++j){
19
20            // comparing row and column
21            bool same {true};
22            for(int k = 0; k<nelems; ++k){
23                if(grid[i][k] != grid[k][j]) {same = false;}
24            }
25
26            // increasing count
27            if (same) {++counts;}
```



```
28     }
29 }
30
31 // printing
32 cout << format("counts = {}\n", counts);
33
34 // return
35 return(0);
36
37 }
```

---

## 2390. Removing Stars From A String

You are given a string `s`, which contains stars `*`.

In one operation, you can:

- Choose a star in `s`.
- Remove the closest non-star character to its left, as well as remove the star itself.

Return the string after all stars have been removed.

Note:

- The input will be generated such that the operation is always possible.
- It can be shown that the resulting string will always be unique.

### Examples

#### 1. Example 1:

- Input: `s = "leet**cod*e"`
- Output: `"lecoe"`
- Explanation: Performing the removals from left to right:
  - The closest character to the 1st star is 't' in `"leet**cod*e"`. `s` becomes `"lee*cod*e"`.
  - The closest character to the 2nd star is 'e' in `"lee*cod*e"`. `s` becomes `"lecod*e"`.
  - The closest character to the 3rd star is 'd' in `"lecod*e"`. `s` becomes `"lecoe"`.
  - There are no more stars, so we return `"lecoe"`.

## 2. Example 2:

- Input: `s = "erase*****"`
- Output: `""`
- Explanation: The entire string is removed, so we return an empty string.

## Constraints

- $1 \leq s.length \leq 10^5$
- `s` consists of lowercase English letters and stars `*`.
- The operation above can be performed on `s`.

## Code

---

```
1 int main(){
2
3     // input- configuration
4     string s {"leet**cod*e"};
5
6     // going through the inputs
7     std::string finalOutput;
8     for(auto x: s){
9         if (x == '*') finalOutput.pop_back();
10        else finalOutput.push_back(x);
11    }
12
13    // returning output
14    cout << format("final-output = {}\n", finalOutput);
```

```
15
16 // return
17 return(0);
18
19 }
```

---

## 2481. Minimum Cuts to Divide a Circle

A valid cut in a circle can be:

1. A cut that is represented by a straight line that touches two points on the edge of the circle and passes through its center, or
2. A cut that is represented by a straight line that touches one point on the edge of the circle and its center.

Given the integer  $n$ , return the minimum number of cuts needed to divide a circle into  $n$  equal slices.

### Examples

#### 1. Example 1:

- Input:  $n = 4$
- Output: 2
- Explanation: The above figure shows how cutting the circle twice through the middle divides it into 4 equal slices.

#### 2. Example 2:

- Input:  $n = 3$
- Output: 3
- Explanation:
  - At least 3 cuts are needed to divide the circle into 3 equal slices.
  - It can be shown that less than 3 cuts cannot result in 3 slices of equal size and shape.
  - Also note that the first cut will not divide the circle into distinct parts.

## Constraints

- $1 \leq n \leq 100$

## Code

---

```
1 int main(){
2
3     // starting timer
4     Timer timer;
5
6     // input- configuration
7     auto n {4};
8
9     // setup
10    auto finaloutput    {0};
11    if (n == 1)          {finaloutput = 0;}
12    else if (n%2 == 0)   {finaloutput = n/2;}
13    else                 {finaloutput = n;}
14
15    // printing
16    cout << format("final-output = {}\n", finaloutput);
17
18    // return
19    return(0);
20
21 }
```

---