Leetcode Solutions

SVR

October 19, 2025

Contents

Introduction

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

Template Script

Description

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

Template.cpp

```cpp
// including header-files
#include <algorithm>
#include <unordered_set>
#include <bitset>
#include <climits>
#include <cstddef>
#include <iostream>
#include <limits>
#include <map>
#include <new>
#include <stdlib.h>
#include <unordered_map>
#include <vector>
#include <set>
#include <numeric>
#include <functional>
#include <deque>


// hash-deinfes
```

```cpp
#define PRINTSPACE std::cout << "\n\n\n\n" << std::endl;
#define PRINTLINE std::cout << "==============================================" << std::endl;

// borrowing from namespace std
using std::cout;
using std::endl;
using std::vector;
using std::string;
using std::unordered_map;
using std::map;
using std::format;
using std::deque;
using std::pair;
using std::min;
using std::max;

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

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

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

```
        cout << endl;
    }
}

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

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


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

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

    // printing
```

```
    PRINTLINE
    cout << "root->val = " << root->val << endl;

    // calling the children
    fPrintBinaryTree(root->left);
    fPrintBinaryTree(root->right);

    // returning
    return;

}

void fPrintLinkedList(string prefix,
                      ListNode* root){
    if (root == nullptr) return;
    cout << prefix;
    std::function<void(ListNode*)> runlinkedlist = [&runlinkedlist](ListNode* root){
        if (root == nullptr) return;
        cout << root->val << " -> ";
        runlinkedlist(root->next);
    };
    runlinkedlist(root);
    cout << "|" << endl;
    return;
}

template<typename T>
void fPrintContainer(T input){
    for(auto x: input) cout << x << ", ";
    cout << endl;
    return;
}

struct Timer
{
```

```cpp
    std::chrono::time_point<std::chrono::high_resolution_clock> startpoint;
    std::chrono::time_point<std::chrono::high_resolution_clock> endpoint;
    std::chrono::duration<long long, std::nano>           duration;

    // constructor
    Timer()       {startpoint = std::chrono::high_resolution_clock::now();}
    void start()  {startpoint = std::chrono::high_resolution_clock::now();}
    void stop()   {endpoint  = std::chrono::high_resolution_clock::now(); fetchtime();}

    void fetchtime(){
        duration = std::chrono::duration_cast<std::chrono::nanoseconds>(endpoint - startpoint);
        cout << format("{} nanoseconds \n", duration.count());
    }
    void fetchtime(string stringarg){
        duration = std::chrono::duration_cast<std::chrono::nanoseconds>(endpoint - startpoint);
        cout << format("{} took {} nanoseconds \n", stringarg, duration.count());
    }
    void measure(){
        auto temp = std::chrono::high_resolution_clock::now();
        auto nsduration = std::chrono::duration_cast<std::chrono::nanoseconds>(temp - startpoint);
        auto msduration = std::chrono::duration_cast<std::chrono::microseconds>(temp - startpoint);
        auto sduration  = std::chrono::duration_cast<std::chrono::seconds>(temp - startpoint);
        cout << format("{} nanoseconds | {} microseconds | {} seconds \n",
            nsduration.count(), msduration.count(), sduration.count());
    }
    ~Timer(){
        measure();
    }
};

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

    // starting timer
    Timer timer;
```

```
    // input- configuration

    // setup




    // return
    return(0);

}
```

1. Two Sum

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

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

```
int main(){

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

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

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

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

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

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

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

    // return
    return(0);

}
```

2. Add Two Numbers

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

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

```
int main(){

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

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

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

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

    // moving through the two nodes
    while(traveller_1 != nullptr || traveller_2 != nullptr){

        // adding the two numbers
        value_1 = traveller_1 == nullptr ? 0 : traveller_1->val;
```

```
        value_2 = traveller_2 == nullptr ? 0 : traveller_2->val;

        // calculating sum
        sum    = value_1 + value_2 + carry;
        if (sum >= 10) [[unlikely]]  {sum -= 10; carry = 1;}
        else          [[likely]]     {carry = 0;}

        // creating node
        traveller_fo->next = new ListNode(sum);
        traveller_fo       = traveller_fo->next;

        // updating the two pointers
        if(traveller_1 != nullptr) [[likely]] {traveller_1 = traveller_1->next;}
        if(traveller_2 != nullptr) [[likely]] {traveller_2 = traveller_2->next;}
    }

    // creating a final node if carry is non-zero
    if (carry == 1) [[unlikely]] {
        traveller_fo->next = new ListNode(carry);
    }

    // printing the final output
    traveller_fo = finalOutput->next;
    cout << format("final-output = ");
    while(traveller_fo != nullptr){
        cout << traveller_fo->val << ", ";
        traveller_fo = traveller_fo->next;
    }
    cout << "\n";

    // return
    return(0);

}
```

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 = "abcabcbb"`
    - Output: 3
    - Explanation: The answer is `"abc"`, with the length of 3.
2. **Example 2:**
    - Input: `s = "bbbbb"`
    - Output: 1
    - Explanation: The answer is `"b"`, with the length of 1.
3. **Example 3:**
    - Input: `s = "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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto s      {string("abcabcbb")};

    // trivial cases
    if (s.size() <= 1) {cout << format("final-output = {}\n", s.size()); return 0;}

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

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

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

        // checking if current character is in histogram
```

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

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

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

    // return
    return(0);

}
```

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. $0 \leq \text{n} \leq 1000$

5. $1 \leq \text{m} + \text{n} \leq 2000$

6. $-10^6 \leq \text{nums1[i], nums2[i]} \leq 10^6$

Code

```
int main(){

    // input- configuration
    vector<int> nums1 {1, 2};
    vector<int> nums2 {3, 4};


    // setup
    vector<int>& first = nums1[0] <= nums2[0] ? nums1 : nums2;
    vector<int>& second = nums1[0] > nums2[0] ? nums1 : nums2;
    int left_first {0};
    int right_first {static_cast<int>(first.size())-1};
    int left_second {0};
    int right_second  {static_cast<int>(second.size())-1};
    int left_value = first[left_first] < second[left_second] ? first[left_first] : second[left_second];
    int right_value = first[right_first] > second[right_second] ? first[right_first] : second[right_second];
    int numiterations {static_cast<int>((nums1.size() + nums2.size())/2)};


    // running for a certain number of iterations
    for(int i = 0; i<numiterations+1; ++i){

        // updating left
        if (first[left_first] < second[left_second]) {left_value = first[left_first]; ++left_first;}
        else                                         {left_value = second[left_second]; ++left_second;}
```

```
        if (first[right_first] > second[right_second]) {right_value = first[right_first]; --right_first;}
        else                                           {right_value = second[right_second]; --right_second;}

        // printing
        cout << format("left-value = {}, right-value = {}\n", left_value, right_value);
    }

    cout << format("median = {}\n", static_cast<double>(left_value + right_value)/2.0);



    // return
    return(0);

}
```

5. Longest Palindromic Substring

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

Examples

1. **Example 1:**
   - Input: s = ”babad”
   - Output: ”bab”
   - Explanation: ”aba” is also a valid answer.
2. **Example 2:**
   - Input: s = ”cbbd”
   - Output: ”bb”

Constraints

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

Code

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto s {string("babad")};


    // setup
    int maxlength {-1};
    int boundary  {1};
    string largestpalindrome = "";

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

        // checking odd-palindromes from here
        auto oddlength    {1};
        auto boundary     {1};
        string oddstring  {s[i]};

        while(i - boundary >= 0 && i + boundary < s.size() && s[i-boundary] == s[i+boundary]){
            oddlength += 2;                                     // updating length
            oddstring = s.substr(i-boundary, 2*boundary+1);    // subsetting
            ++boundary;                                         // updating boundary
        }

        // checking even-palindromes from here
        auto evenlength   {0};
        boundary          = 1;
        auto evenstring   {string("")};

        while(i+1-boundary >= 0 && i+boundary < s.size() && s[i+1-boundary] == s[i+boundary]){
```

```
            evenlength += 2;                                 // updating length
            evenstring = s.substr(i+1-boundary, 2*boundary);  // subsetting
            ++boundary;                                      // updating boundary
        }

        // updating largest-string
        largestpalindrome = oddlength > largestpalindrome.size() ? oddstring : largestpalindrome;
        largestpalindrome = evenlength > largestpalindrome.size() ? evenstring : largestpalindrome;

    }

    // returning
    cout << format("final-output = {}\n", largestpalindrome);

    // return
    return(0);

}
```

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

Code

```cpp
int main(){

    // input- configuration
    string s   {"PAYPALISHIRING"};
    int numRows {4};

    // trivial case
    if (numRows == 1) {cout << format("finaloutput = {}\n", s); return 0;}

    // setup
    int modlength {2*numRows-2};
    int numblocks {(static_cast<int>(s.size())+ modlength-1)/modlength};
    int sourceindex {-1};
    string finaloutput;

    // going through the thing
    for(int row = 0; row < numRows; ++row){
        for(int i = 0; i<numblocks; ++i){

            // first column of each block
            sourceindex = row + modlength * i;
            if (sourceindex<s.size())        {finaloutput += s[sourceindex];}

```

```
            // continuing in case of boundary rows
            if (row == 0 || row == numRows-1) {continue;}

            // taking care of the case where non-boundary rows
            sourceindex = modlength - row + modlength*i;
            if (sourceindex < s.size())      {finaloutput += s[sourceindex];}
        }
    }

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


    // return
    return(0);

}
```

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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto x {123};

    // setup
    auto finaloutput      {static_cast<long long>(0)};
    auto leftboundary     {- std::pow(2, 31)};
    auto rightboundary    {-1 + std::pow(2, 31)};

    // building final output
    while(x){

        finaloutput = finaloutput*10 + x%10;
        x           = x/10;
        if (finaloutput < leftboundary || finaloutput > rightboundary) {return 0;} // checking the cases
    }

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

    // return
    return(0);

}
```

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

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

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto x {121};

    // setup
    auto finaloutput  {false};

    // retarded question
    if (x<0)   {cout << format("final-output = {}\n", finaloutput); return 0;}

    // running
    auto x_clone  {x};
    auto x_reverse {static_cast<long int>(0)};

    // running through the value
    while(x_clone){
        x_reverse  = x_reverse*10 + x_clone%10;
        x_clone    = x_clone/10;
    }

    // comparing the two
    if (x == x_reverse)   {finaloutput = true;}
    else                  {finaloutput = false;}


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


```

```
    // return
    return(0);

}
```

11. Container with most water

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

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
- $2 \leq n \leq 10^5$
- $0 \leq height[i] \leq 10^4$

Code

```cpp
int main(){

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

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

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

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

        // adjusting left and right based on volume
        if (height[left] < height[right])   {++left;}
        else                                {--right;}
    }

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

    // return
    return(0);

}
```

12. Integer to Roman

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

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

Given an integer, convert it to a Roman numeral.

Examples

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

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

2. **Example 2:**
   - Input: num = 58
   - Output: "LVIII"
   - Explanation:
     - 50 = L
     - 8 = VIII

3. **Example 3:**
   - Input: num = 1994
   - Output: "MCMXCIV"
   - Explanation:
     - 1000 = M
     - 900 = CM
     - 90 = XC
     - 4 = IV

Constraints

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

Code

```cpp
int main(){

    // input- configuration
    int num    {1994};

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

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

```

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

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

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

    // return
    return(0);

}
```

13. Roman To Integer

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

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

Given a roman numeral, convert it to an integer.

Examples

1. **Example 1**
   - Input: s = ”III”
   - Output: 3
   - Explanation: III = 3.
2. **Example 2**
   - Input: s = ”LVIII”
   - Output: 58

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

3. **Example 3**

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

Constraints

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

Code

```
int main(){

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

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

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

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

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

    // return
    return(0);

}
```

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:

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

Code

```
int main(){

    // input- configuration
    vector<string> strs {
        "flower",
        "flow",
        "flight"
    };

    // setup
    int p              {0};                                                  // index-pointer for boundary
    int runcondition  {true};                                                // breaking condition
    string prefix;

    // going through the vector
    while(runcondition){

        // breaking if it doesn't meet first words length
        if (p >= strs[0].size())         {++p; runcondition = false; break;}

        // checking if this candidate
        for(int i = 1; i<strs.size(); ++i){

            // checking if valid
            if (p >= strs[i].size())     {runcondition = false; break;}

            // checking if same
            if (strs[i][p] != strs[0][p]) {runcondition = false; break;}
        }

        // updating p
        ++p;
    }
```

```
    // subsetting and printing the prefix
    prefix = string(strs[0].begin(), strs[0].begin()+p-1);
    cout << format("finaloutput = {}\n", prefix);

    // return
    return(0);

}
```

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

- $0 \leq$ digits.length ≤ 4
- digits[i] is a digit in the range [’2’, ’9’].

Code

```
void foo(int                horizontalindex,
         int                verticalindex,
         string             digits,
         string             runningstring,
         vector<string>&  finalOutput,
         unordered_map<char, vector<string>>& charToLetter)
{

    // adding current to running sum
    runningstring += charToLetter[digits[verticalindex]][horizontalindex];

    // sending it back
    if (verticalindex == digits.size()-1) {finalOutput.push_back(runningstring); return;}

    // running recursion on different sub-paths
    for(int j = 0; j < charToLetter[digits[verticalindex+1]].size(); ++j)
        foo(j, verticalindex+1,digits, runningstring, finalOutput, charToLetter);

    // returning
    return;
}

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto digits {string("23")};

    // trivial case
    if (digits.size()==0) {
        cout << format("final-output = {}\n", vector<string>({}));
```

```
        return 0;
    }

    // setup
    unordered_map<char, vector<string>> charToLetter;
    charToLetter['2'] = vector<string>({"a", "b", "c"});
    charToLetter['3'] = vector<string>({"d", "e", "f"});
    charToLetter['4'] = vector<string>({"g", "h", "i"});
    charToLetter['5'] = vector<string>({"j", "k", "l"});
    charToLetter['6'] = vector<string>({"m", "n", "o"});
    charToLetter['7'] = vector<string>({"p", "q", "r", "s"});
    charToLetter['8'] = vector<string>({"t", "u", "v"});
    charToLetter['9'] = vector<string>({"w", "x", "y", "z"});

    // going throguh each character on top level
    vector<string> finalOutput;
    for(int i = 0; i<charToLetter[digits[0]].size(); ++i)
        foo(i, 0, digits, "", finalOutput, charToLetter);

    // returning
    cout << format("final-output = {}\n", finalOutput);

    // return
    return(0);

}
```

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.
- $1 \leq sz \leq 30$
- $0 \leq Node.val \leq 100$
- $1 \leq n \leq sz$

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    ListNode* head                   = new ListNode(1);
    head->next                       = new ListNode(2);
    head->next->next                 = new ListNode(3);
    head->next->next->next           = new ListNode(4);
    head->next->next->next->next = new ListNode(5);
    auto n {2};

    // trivial case
    if (!head) {cout << format("final-output = "); fPrintLinkedList(head); cout << endl;}

    // setup
    auto nodecounter  {0};
    auto prehead      {new ListNode};

    prehead->next              = head;
    ListNode* traveller        = head;
    ListNode* delayedTraveller = prehead;

    // going through the list
    while(traveller){
        if (++nodecounter > n) {delayedTraveller = delayedTraveller->next;}
        traveller = traveller->next;
    }

    // reconnecting
    delayedTraveller->next = delayedTraveller->next->next;

```

```
    // sending back
    cout << format("final-output = ");
    fPrintLinkedList(prehead->next); cout << endl;





    // return
    return(0);

}
```

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

- $1 \leq \text{s.length} \leq 10^4$
- s consists of parentheses only ’()[]’.

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    string s   {"()[]{}"};

    // seutp
    deque<char> var00;

    // going through the string
    for(auto x: s){
        // pushing or poppping
        if (x == '(' || x == '[' || x == '{') {var00.push_back(x);}
        else{
            if (var00.size()==0)              {cout << format("final-output = false\n"); return 0;}
            if (x == ')' && var00.back() == '(') {var00.pop_back(); continue;}
            if (x == ']' && var00.back() == '[') {var00.pop_back(); continue;}
            if (x == '}' && var00.back() == '{') {var00.pop_back(); continue;}
            cout << format("final-output = false\n"); return 0;
        }
    }

    // checking if anything is left
```

```
    if (var00.size()!=0) {cout << format("final-output = false\n"); return 0;}


    // true-case
    cout << format("final-output = true\n"); return 0;

    // return
    return(0);

}
```

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

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    ListNode* list1   = new ListNode(1);
    list1->next       = new ListNode(2);
    list1->next->next = new ListNode(4);

    ListNode* list2   = new ListNode(1);
    list2->next       = new ListNode(3);
    list2->next->next = new ListNode(4);

    // setup
    ListNode* finalOutput = new ListNode;
    ListNode* traveller_final = finalOutput;
    ListNode* traveller_1 = list1;
    ListNode* traveller_2 = list2;

    // going through the two lists
    while (traveller_1 != nullptr && traveller_2 != nullptr) {

        // comparing values
        if (traveller_1->val < traveller_2->val) {
            traveller_final->next = traveller_1;            // linking node to final
            traveller_final      = traveller_final->next; // moving node to new link
```

```
            if (traveller_1->next != nullptr)
                traveller_1           = traveller_1->next;     // updating traveller 1
            else{
                traveller_1 = nullptr;
                break;
            }

        }
        else {
            traveller_final->next = traveller_2;            // linking node to final
            traveller_final       = traveller_final->next; // moving node to new link
            if (traveller_2->next != nullptr)
                traveller_2           = traveller_2->next;     // updating travel 2
            else{
                traveller_2 = nullptr;
                break;
            }
        }
    }

    // checking if anything is left
    while (traveller_1 != nullptr){
        traveller_final->next = traveller_1;
        traveller_final       = traveller_final->next;
        traveller_1           = traveller_1->next;
    }

    while (traveller_2 != nullptr){
        traveller_final->next = traveller_2;
        traveller_final       = traveller_final->next;
        traveller_2           = traveller_2->next;
    }

    // printing the final-output
```

```
    cout << format("finaloutput = ");
    fPrintLinkedList(finalOutput); cout << endl;

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

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

```cpp
void foo(int                numopens,
        int                 numcloses,
        const int&          n,
        string              runningstring,
        vector<string>&     finalOutput){
```

```

    // sending it back
    if (numopens > n || numcloses > n) return;
    else if (numopens == n && numcloses == n) {finalOutput.push_back(runningstring); return;}

    // opening route
    foo(numopens+1, numcloses, n, runningstring + "(", finalOutput);

    // closing route
    if (numopens>numcloses)
        foo(numopens, numcloses+1, n, runningstring+")", finalOutput);

    // returning
    return;

}

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto n {3};

    // trivial case
    if (n == 0) {cout << format("finalOutput = []\n"); return 0;}

    // calling the function
    vector<string> finalOutput;
    foo(0, 0, n, "", finalOutput);

    // returning
    cout << format("finalOutput = {}\n", finalOutput);

```

```
    // return
    return(0);

}
```

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

Code

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

    // starting timer
    Timer timer;

    // input- configuration
    auto head0 {new ListNode(1)};
    head0->next = new ListNode(4);
    head0->next->next = new ListNode(5);

    auto head1 {new ListNode(1)};
    head1->next = new ListNode(3);
    head1->next->next = new ListNode(4);

    auto head2 {new ListNode(2)};
    head2->next = new ListNode(6);
```

```cpp

    vector<ListNode*> lists {head0, head1, head2};


    // setup
    ListNode*         prehead = new ListNode();
    ListNode*         traveller     {prehead};
    bool              runcondition  {true};
    vector<ListNode*> travellerList;
    int               smallestindex {-1};
    int               smallestvalue {std::numeric_limits<int>::max()};

    // filling up the traveller lists
    for(int i = 0; i<lists.size(); ++i) {travellerList.push_back(lists[i]);}

    // running the loop
    int counter {0};
    while(runcondition){

        // going through the values
        for(int i = 0; i < travellerList.size(); ++i){
            if (travellerList[i]     != nullptr &&
                travellerList[i]->val <  smallestvalue) {
                smallestvalue = travellerList[i]->val;
                smallestindex = i;
            }
        }

        // now that we have the smallest value and smallest index, we add to the prehead
        if (smallestindex == -1) {break;}
        traveller->next              = travellerList[smallestindex];
        traveller                    = traveller->next;
        travellerList[smallestindex] = travellerList[smallestindex]->next;

        // resetting
```

```
        smallestvalue = std::numeric_limits<int>::max();
        smallestindex = -1;
    }

    // printing the final-linked list
    fPrintLinkedList("final-output = ", prehead->next);

    // return
    return(0);

}
```

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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto head                 {new ListNode(1)};
    head->next                = new ListNode(2);
    head->next->next          = new ListNode(3);
    head->next->next->next    = new ListNode(4);

    // setup
    ListNode* prehead     = new ListNode;
    prehead->next         = head;
    ListNode* traveller   = prehead;

    // going through it
    while(traveller){

        // jump-condition
        if (traveller->next == nullptr || traveller->next->next == nullptr) break;

        // swapping
        if (traveller->next      != nullptr && traveller->next->next != nullptr){

            ListNode* a    = traveller->next;
            ListNode* b    = traveller->next->next;
            ListNode* c    = traveller->next->next->next;

            traveller->next = b;
            b->next        = a;
            a->next        = c;
            traveller      = a;
```

```
        }
    }

    // printing the linked LIst
    fPrintLinkedList("final-output = ", prehead->next);

    // return
    return(0);

}
```

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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto head                    {new ListNode(1)};
    head->next                   = new ListNode(2);
    head->next->next             = new ListNode(3);
    head->next->next->next       = new ListNode(4);
    head->next->next->next->next = new ListNode(5);
    auto k {2};

    // setup
    auto prehead      {new ListNode()};
    prehead->next     = head;
    auto traveller    {prehead};
    vector<ListNode*> memberList(k, nullptr);
    memberList.reserve(k);

    // runnings
    while(traveller){

        // filling up the memberList
        auto counter   {0};
        auto tempotraveller {traveller->next};

        while(counter < k && tempotraveller != nullptr){
            memberList[counter]   = tempotraveller;
            tempotraveller        = tempotraveller->next;
            ++counter;
        }

```

```
        // checking breaking condition
        if (counter!=k) break;

        // reconnecting
        auto beginningOfNextSegment {memberList[memberList.size()-1]->next};

        // reconnecting
        traveller->next   = memberList[memberList.size()-1];
        for(int i = memberList.size()-1; i>=1; --i) {memberList[i]->next = memberList[i-1];}
        memberList[0]->next = beginningOfNextSegment;

        // updating traveller
        traveller = memberList[0];
    }

    // returning
    fPrintLinkedList("final-output = ", prehead->next);

    // return
    return(0);

}
```

26. Remove Duplicates From Sorted Array

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

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

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 $\leq$ nums[i] $\leq$ 100
- nums is sorted in non-decreasing order.

Code

```cpp
int main(){

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

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

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

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

        // writing values
        ++p;
        nums[p] = nums[i];
        ++counter;
    }

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

```
    cout << format("nums = "); fpv(nums);

    // return
    return(0);

}
```

27. Remove Element

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

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

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

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

```cpp
int main(){

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

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

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

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

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

        // writing
        if (dest < nums.size() && src < nums.size()){
```

```
            nums[dest] = nums[src];
            ++dest;
            ++src;
            ++numwrites;
        }

    }

    // printing the length
    cout << format("updated nums = "); fPrintVector(nums);
    cout << format("finaloutput = {} \n", nums.size()-numwrites-1);

    // return
    return(0);

}
```

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

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

Code

```
int main(){

    // input- configuration
    string haystack {"leetcode"};
    string needle {"leeto"};


    // setup
    int finaloutput {-1};
    auto beginsearch = [haystack, needle](int currindex){
        // starting search
        if(currindex + needle.size() > haystack.size()) {return false;}

        // checking if they're a subset
        for(int i = 0; i<needle.size(); ++i){
            if (haystack[currindex + i] != needle[i]) {return false;}
        }

        return true;
    };

    // going through
    for(int i = 0; i < haystack.size(); ++i){

        // begin search at each index
        auto curroutput = beginsearch(i);

        // writing final output, if a mach
        if (curroutput) {finaloutput = i; break;}

    }

    // printing final output
```

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

    // return
    return(0);

}
```

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 ($1 \leq k < nums.length$) 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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto nums  {vector<int>{4,5,6,7,0,1,2}};
    auto target {0};

    // setup
    auto left         {0};
    auto right        {static_cast<int>(nums.size())-1};
    auto mid          {-1};
    auto finaloutput  {-1};


    // going through the array
    while(left <= right){
        // fetching mid-value
```

```cpp
        mid = (left + right)/2;
        if (nums[mid] == target) {finaloutput = mid; break;}

        // checking if the left is sorted
        if(nums[left] <= nums[mid]){
            // checking if the target is out of bounds
            if (target < nums[left] || target> nums[mid])  {left = mid +1;}
            else                                           {right = mid -1;}
        }
        else{
            // checking if target is out of bounds
            if (target<nums[mid] || target > nums[right])  {right = mid -1;}
            else                                           {left = mid + 1;}
        }
    }

    // returning negative one in case none of thise works
    cout << format("final-output = {}\n", finaloutput);

    // return
    return(0);

}
```

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

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto nums  {vector<int>{5,7,7,8,8,10}};
    auto target {8};

    // setup
    auto left         {0};
    auto right        {static_cast<int>(nums.size())-1};
    auto mid          {-1};
    auto leftedge     {-1};
    auto rightedge    {-1};
    auto finaloutput  {vector<int>{-1, -1}};

    // trivial case
    if (nums.size()==0) {cout << format("final-output = {}\n", finaloutput); return 0;}

    // finding left-edge
    while(left <= right){
        mid = (left+right)/2;
        if (nums[mid] < target)      {left = mid +1;}
        else if (target < nums[mid]) {right = mid-1;}
```

```
        else                          {leftedge = mid; right = mid-1;}
    }

    // finding right-edge
    left = 0; right = nums.size()-1;
    while(left <= right){
        mid = (left+right)/2;
        if (nums[mid] < target)       {left = mid +1;}
        else if (target < nums[mid]) {right = mid-1;}
        else                          {rightedge = mid; left = mid+1;}
    }

    // building final output
    finaloutput = vector<int>{leftedge, rightedge};

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

    // return
    return(0);

}
```

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

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto nums  {vector<int>{1, 3, 5, 6}};
    auto target {5};

    // setup
    auto left         {0};
    auto right        {static_cast<int>(nums.size())-1};
    auto mid          {-1};
    auto finaloutput  {-1};

    // running the loop
    while(left<=right){

        // updating mid
        mid = (left +right)/2;

        // checking if mid is the value
        if (nums[mid] == target) {finaloutput = mid; cout << format("final-output = {}\n", finaloutput); return 0;}
        else if(nums[mid]<target) {left = mid+1;}
        else                      {right = mid-1;}
```

```
    }

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

    // return
    return(0);

}
```

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

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

    // input-configuration
    vector< vector<char> > board;
    board.push_back({'5','3','.','.','7','.','.','.','.'});
    board.push_back({'6','.','.','1','9','5','.','.','.'});
    board.push_back({'.','9','8','.','.','.','.','6','.'});
    board.push_back({'8','.','.','.','6','.','.','.','3'});
    board.push_back({'4','.','.','8','.','3','.','.','1'});
    board.push_back({'7','.','.','.','2','.','.','.','6'});
    board.push_back({'.','6','.','.','.','.','2','8','.'});
    board.push_back({'.','.','.','4','1','9','.','.','5'});
    board.push_back({'.','.','.','.','8','.','.','7','9'});

    // basic method
    int xoffset,  yoffset;
    int row_local, col_local;

    // lambda for converting char to inger
    auto fConvert = [](char x) -> int {
        if (x == '.') return -1;
        else { return static_cast<int>(x - '0');}
    };

    // checking row and column entries
    for(int i = 0; i < 9; ++i){

        // register for each jumn
        vector<int> rowRegister(9, 0);
        vector<int> colRegister(9, 0);
        vector<int> blockRegister(9,0);

```

```
        // going through each jumn
        for(int j = 0; j<9; ++j){

            // along the row
            int var00  = fConvert(board[i][j]);
            if (var00  != -1) {if (++rowRegister[var00-1] > 1) return false;}

            // down the column
            var00      = fConvert(board[j][i]);
            if (var00  !=-1)  {if (++colRegister[var00-1] > 1) return false;}

            // checking block
            row_local  = j / 3;
            col_local  = j % 3;
            xoffset    = i / 3;
            yoffset    = i % 3;

            // calculating registers
            var00 = fConvert(board[3*xoffset + row_local][col_local+3*yoffset]);
            if (var00!=-1)    {if (++blockRegister[var00-1]>1) return false;}
        }
    }

    // returning true
    return true;
}
```

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

```cpp
void foo(const vector<int>&  candidates,
         vector<int>          runningvector,
         const int&           target,
         vector<vector<int>>& finalOutput)
{
    // calculating running sum
    int runningsum = std::accumulate(runningvector.begin(),
                                     runningvector.end(),
                                     0);

    // sending it back
    if (runningsum > target)     {return;}
    if (runningsum == target)    {finalOutput.push_back(runningvector); return;}

    // going through the different options
    for(auto x: candidates){
```

```
        // checking if this should be
        if (runningvector.size() != 0 && x < runningvector[runningvector.size()-1]) {continue;}

        auto temp {runningvector};
        temp.push_back(x);

        // recursive-call
        foo(candidates, temp, target, finalOutput);
    }

    // returning
    return;
}
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto candidates   {vector<int>{2,3,6,7}};
    auto target       {7};

    // setup
    vector<vector<int>> finalOutput;
    vector<int>       runningvector;

    // calling the function
    foo(candidates, runningvector, target, finalOutput);

    // returning
    cout << format("final-output = {}\n", finalOutput);

    // return
    return(0);
```

```

}
```

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

- $1 \leq$ candidates.length ≤ 100
- $1 \leq$ candidates[i] ≤ 50
- $1 \leq$ target ≤ 30

```
        cout << format("{} nanoseconds | {} microseconds | {} milliseconds | {} seconds \n",
            nsduration.count(), msduration.count(), milliduration.count(), sduration.count());
    }
    ~Timer(){
        measure();
    }
};

// main-file ========================================================================
void foo(const vector<int>&     candidates,
         vector<int>                pathsofar,
         vector<vector<int>>&    finaloutput,
         const int                  currindex,
         const int&                 target,
         const vector<int>&         cumsum){

    // adding to paths ofar
    if (std::find(pathsofar.begin(),
                  pathsofar.end(),
                  currindex) == pathsofar.end()) {pathsofar.push_back(currindex);}
    else   {return;}

    //
    auto sumsofar     = std::accumulate(pathsofar.begin(),
                                         pathsofar.end(),
                                         0,
                                         [&](auto acc, auto argx){return acc + candidates[argx];});
    auto complement   {target - sumsofar};

    // making decisions based on complement
    if (complement == 0)  {

        vector<int> valuessofar(pathsofar.size());
```

```
        std::transform(pathsofar.begin(),
                       pathsofar.end(),
                       valuessofar.begin(),
                       [&](auto argx){return candidates[argx];});

        // checking if top value is same
        if (std::find(finaloutput.begin(),
                      finaloutput.end(),
                      valuessofar) == finaloutput.end()) {finaloutput.push_back(valuessofar);}

        return;
    }
    else if(complement < 0) {return;}

    // checking if it is possible to go from here
    auto maxpotentialfromhere {cumsum[currindex] - candidates[currindex]};
    if (complement > maxpotentialfromhere) { return;}

    // going through the rest of the indices
    for(int nextindex = currindex + 1; nextindex < candidates.size(); ++nextindex){
        foo(candidates, pathsofar, finaloutput, nextindex, target, cumsum);
    }

}

int main(){

    // starting timer
    Timer timer;

    // input-configuration
    auto candidates   {vector<int>{10,1,2,7,6,1,5}};
    auto target       {8};

```

```cpp
    // setup
    vector<vector<int>> finaloutput;
    vector<int>             pathsofar;

    // sorting number
    std::sort(candidates.begin(), candidates.end(), std::greater<int>());

    // creating cumulative sums
    std::vector<int> cumsum(candidates.size(), 0);
    std::partial_sum(candidates.rbegin(), candidates.rend(), cumsum.rbegin());

    // calling the function
    for(int i = 0; i < candidates.size(); ++i){
        foo(candidates,
            pathsofar,
            finaloutput,
            i,
            target,
            cumsum);
    }

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

    // return
    return(0);

}
```

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

```cpp
int main(){

    // starting timer
    Timer timer;

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

    // in the off-chance that minvalue is one, building ordered set of values
    std::set<int> orderedvalues;
    for(auto x: nums)            {if (x>0) orderedvalues.insert(x);}

    // going through the ordered set
    auto finaloutput  {1};
    auto flag         {0};
    for(const auto& x: orderedvalues)
        if (x - finaloutput++ != 0) {--finaloutput; break;}

    // printing based on flag value
    cout << format("final-output = {}\n", finaloutput);

    // return
    return(0);

}
```

42. Trapping Rain Water

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

Examples

1. **Example 1**
   - Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]
   - Output: 6
   - Explanation: The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.
2. **Example 2**
   - Input: height = [4,2,0,3,2,5]
   - Output: 9

Constraints

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

Code

```cpp
int main(){

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

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

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

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

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

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

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

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

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

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

    // return
    return(0);

}
```

45 Jump Game II

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

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

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

Examples

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

Constraints

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

Code

```
int main(){

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

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

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

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

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

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

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

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

    }

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

    // return
    return(0);

}
```

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

- $1 \leq$ nums.length ≤ 6
- $-10 \leq$ nums[i] ≤ 10
- All the integers of nums are unique.

Code

```
// traversing function
void fRecursive(const std::vector<int>&              nums,           \
                std::vector< std::vector<int> >&    finalOutput,    \
                const std::vector<int>              path,           \
                const std::vector<int>              numsleft){

    // check if length is reached
    if (path.size() == nums.size()){

        // add to final output
        if (std::find(finalOutput.begin(), finalOutput.end(), path) == finalOutput.end())
            finalOutput.push_back(path);

        // returning since the path is complete
        return;
    }

    // choosing from numsleft
    for(int i = 0; i<numsleft.size(); ++i){

        // choosing a number,
        auto num                {numsleft[i]};
        auto pathtemp           {path};
        pathtemp.push_back(num);

        // making a copy of nums left, removing character
        auto numsleft_temp      {numsleft};
        numsleft_temp.erase(numsleft_temp.begin() + i);

        // calling function
        fRecursive(nums, finalOutput, pathtemp, numsleft_temp);
    }

```

```
    // returning results
    return;
}

int main(){

    // starting timer
    Timer timer;

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

    // setup
    std::vector<std::vector<int>> finalOutput;
    auto path                    {vector<int>{}};
    auto numsleft                {nums};

    // calling the function
    fRecursive(nums, finalOutput, path, numsleft);

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

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

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

- $1 \leq$ nums.length ≤ 8
- $-10 \leq$ nums[i] ≤ 10

Code

```
auto foo(const vector<int>& inputvector,
         const int&       elementtoadd){

    // creating the different options that can stem from this
    vector<vector<int>> finaloutput; // max size is n(n+1)

    for(int i = 0; i<inputvector.size()+1; ++i){

        auto insertindex  {i};
        auto temp         {vector<int>(inputvector.size()+1)};

        for(int j = 0; j<insertindex; ++j)                  {temp[j] = inputvector[j];}
        temp[insertindex] = elementtoadd;
        for(int j = insertindex; j<inputvector.size(); ++j)
        for(int j = insertindex; j<inputvector.size(); ++j) {temp[j+1] = inputvector[j];}

        // check if temp is already in the finaloutput
        if (std::find(finaloutput.begin(),
                      finaloutput.end(),
                      temp) == finaloutput.end())  {finaloutput.push_back(std::move(temp));}
    }

    // returning
    return finaloutput;


}

int main(){

    // starting timer
    Timer timer;

    // input- configuration
```

```
    auto nums  {vector<int>{2,2,1,1}};

    // setup
    std::deque<vector<int>> pool;
    pool.push_back(vector<int>{nums[0]});

    // going throuhg a loop
    for(int i = 1; i<nums.size(); ++i){

        auto drainpool {pool};
        pool.clear();

        while(drainpool.size() != 0){
            auto frontvector  = drainpool.front(); drainpool.pop_front(); // popping from the front
            auto elementtoadd {nums[i]};
            auto returntemp   {foo(frontvector, elementtoadd)};           // calling the function
            for(const auto& x: returntemp) {
                if (std::find(pool.begin(),
                              pool.end(),
                              x) == pool.end())   {pool.push_back(x);}
            }                // pushing all to the back of the pool
        }

    }

    // printing the pool in the end
    cout << format("pool = {}\n", pool);
```

```
    // return
    return(0);

}
```

48. Rotate Image

You are given an n x 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 == matrix.length == matrix[i].length
- $1 \leq n \leq 20$
- $-1000 \leq matrix[i][j] \leq 1000$

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    vector<vector<int>> matrix {
        {1,2,3},
        {4,5,6},
        {7,8,9}
    };

    // setup
    auto t_edge    {0};
    auto b_edge    {static_cast<int>(matrix.size())-1};
    auto l_edge    {0};
    auto r_edge    {static_cast<int>(matrix[0].size())-1};
    auto temp      {-1};

    // shifting layer by layer.
    int numlayers = (matrix.size())/2;

    // shifting layer by layer
    for(int i = 0; i<numlayers; ++i){

        // breaking if they're the same for some reason
        if(t_edge == b_edge || l_edge == r_edge) {break;}

        // calculatin width
        auto currwidth {r_edge - l_edge + 1};

        // shifting it around
        for(int j = 0; j<currwidth-1; ++j){
```

```
            // shifting the four elements for each
            temp                        = matrix[t_edge][l_edge+j];
            matrix[ t_edge ][ l_edge+j ] = matrix[b_edge-j][l_edge];
            matrix[ b_edge-j ][ l_edge ] = matrix[b_edge][r_edge-j];
            matrix[ b_edge ][ r_edge-j ] = matrix[t_edge+j][r_edge];
            matrix[ t_edge+j ][ r_edge ] = temp;
        }

        // updating edge-parameters based on the layer we're working with
        t_edge += 1;
        b_edge -= 1;
        l_edge += 1;
        r_edge -= 1;

    }

    // printing the matrix
    cout << format("print-matrix\n");
    fPrintMatrix(matrix);

    // return
    return(0);

}
```

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

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

Code

```
// build histogram
vector<int> fBuildHist(string input){
    vector<int> myhist(26, 0);
    for(auto x: input) {++myhist[static_cast<int>(x - 'a')];}
    return myhist;
}

// Custom hash function for vector<int>
struct VectorHash {
    std::size_t operator()(const std::vector<int>& v) const {
        std::size_t hashValue = 0;
        for (int num : v)
            hashValue ^= std::hash<int>{}(num) + 0x9e3779b9 + (hashValue << 6) + (hashValue >> 2);

        return hashValue;
    }
};
// int main
int main(){

    // starting timer
    Timer timer;

```

```
    // input- configuration
    vector<string> strs{
        "eat",
        "tea",
        "tan",
        "ate",
        "nat",
        "bat"
    };

    // unordered map for hist to vector of vector of strings
    unordered_map< vector<int>, vector<string>, VectorHash > histToGroup;

    for(auto x: strs){

        // building hist of current word
        auto xhist = fBuildHist(x);

        // checking if it exists
        if (histToGroup.find(xhist) == histToGroup.end())
            histToGroup[xhist] = vector<string>({x});
        else
            histToGroup[xhist].push_back(x);

    }

    // building final output
    vector<vector<string>> finalOutput;
    for(auto x: histToGroup) {finalOutput.push_back(x.second);}

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

    // return
    return(0);
```

```
}
```

50. Pow(x, n)

Implement pow(x, n), which calculates x raised to the power n (i.e., xn).

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 = 1/22 = 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 ¿ 0.
- $-10^4 \leq x^n \leq 10^4$

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto x {2.00000};
    auto n {10};

    // setup
    std::function<double(double, long int)> foo = [&foo](
        double      x,
        long int    n) -> double
    {

        // base-cases
        if (x == 0)    {return 0;}
        if (n == 0)    {return 1;}

        // performing calculations
        double result = foo(x, n/2);
        result = result * result;

        // in case there is some odd-wor
        if (n%2 == 1) {result = result * x;}
```

```
        // returning the results
        return result;

    };

    // calling the function
    auto finalresult = foo(x, std::abs(static_cast<long int>(n)));

    // taking care of the negative case
    if (n < 0) {finalresult = static_cast<double>(1)/finalresult;}

    // returning the final result
    cout << format("final-output = {}\n", finalresult);

    // return
    return(0);

}
```

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

```cpp
int main(){

    // starting timer
    Timer timer;

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

    // setup
    auto runningsum {0};
    auto finaloutput {std::numeric_limits<int>::min()};

    // going through the array
    for(int i =0; i<nums.size(); ++i){
        runningsum += nums[i];
        finaloutput = std::max(runningsum, finaloutput);
        if (runningsum <0) {runningsum = 0;}
    }

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

    // return
    return(0);

```

```
}
```

54. Spiral Matrix

Given an m x 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 == matrix.length
- n == matrix[i].length
- $1 \leq m, n \leq 10$
- $-100 \leq matrix[i][j] \leq 100$

Code

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    vector<vector<int>> matrix{
        {1,2,3,4},
        {5,6,7,8},
        {9,10,11,12}
    };

    // setup
    int         left   {0};
    int         right  {static_cast<int>(matrix[0].size()-1)};
    int         top    {0};
    int         bottom {static_cast<int>(matrix.size()-1)};
    vector<int> finalOutput;

    // moving through this
    while(left <= right && top <= bottom){

        // moving on upside
        for(int i = left; i <= right; ++i)                      {finalOutput.push_back(matrix[top][i]);}

        // moving through the right side
        for(int i = top+1; i<=bottom-1; ++i)                    {finalOutput.push_back(matrix[i][right]);}

        // moving through the bottom
        for(int i = right; top != bottom && i>= left; --i)     {finalOutput.push_back(matrix[bottom][i]);}

        // moving through the left
        for(int i = bottom-1; left!=right && i>= top+1; --i)  {finalOutput.push_back(matrix[i][left]);}
```

```

        // updating boundaries
        ++left; --right; ++top; --bottom;

    }

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

    // return
    return(0);

}
```

55. Jump Game

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

Examples

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

Constraints

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

Code

```
int main(){

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

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

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

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

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

    }

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

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


    // return
    return(0);
```

```
}
```

56. Merge Intervals

Given an array of intervals where intervals[i] = [starti, endi], 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: 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: intervals = [[1,4],[4,5]]
    - Output: [[1,5]]
    - Explanation: Intervals [1,4] and [4,5] are considered overlapping.

Constraints

- $1 \leq \text{intervals.length} \leq 104$
- $\text{intervals[i].length} == 2$
- $0 \leq \text{starti} \leq \text{endi} \leq 104$

Code

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    vector<vector<int>> intervals{
        {1,3},
        {2,6},
        {8,10},
        {15,18}
    };

    // setup
    vector<vector<int>> finalOutput;

    // sorting the input
    std::sort(intervals.begin(), intervals.end(),
        [](const vector<int>& a, const vector<int>& b){return a[0] < b[0];});

    // going through the intervals
    vector<int> runninginterval {intervals[0][0], intervals[0][1]};
    for(int i = 1; i<intervals.size(); ++i){
        if (runninginterval[1] < intervals[i][0]){
            finalOutput.push_back(runninginterval);                    // pushing the results
            runninginterval = intervals[i];                            // getting new running interfval
        }
        else   {runninginterval[1] = max(runninginterval[1], intervals[i][1]);}
    }
    finalOutput.push_back(runninginterval);                            // pushing the results

    // returning
    cout << format("final-output = {}\n", finalOutput);
```

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

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 ith 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
- $0 \leq$ starti $\leq$ endi $\leq 10^5$
- intervals is sorted by starti in ascending order.
- newInterval.length == 2
- $0 \leq$ start$\leq$ end $\leq 10^5$

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    vector<vector<int>> intervals{
        {1,2},
        {3,5},
        {6,7},
        {8,10},
        {12,16}
    };
    vector<int> newInterval{4,8};


    // setup
    vector<vector<int>> finalOutput;
    intervals.push_back(newInterval);

    // sorting the interval
```

```
    std::sort(intervals.begin(),
              intervals.end(),
              [](const vector<int>& a,
                 const vector<int>& b){return a[0] < b[0];});
    vector<int>       runninginterval {intervals[0]};

    // going through the inputs
    for(int i = 1; i<intervals.size(); ++i){
        if (runninginterval[1] < intervals[i][0]){
            finalOutput.push_back(runninginterval);
            runninginterval = intervals[i];
        }
        else{
            runninginterval[1] = max(runninginterval[1], intervals[i][1]);
        }
    }

    // pushing it back
    finalOutput.push_back(runninginterval);
    cout << format("final-output = {}\n", finalOutput);

    // return
    return(0);

}
```

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

```cpp
int main(){

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

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

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

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

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

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

        // calculating the length
```

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

        // breaking
        break;
    }

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

    // return
    return(0);

}
```

59. Spiral Matrix II

Given a positive integer n, generate an n x 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

```
int main(){

    // starting timer
    Timer timer;

```

```cpp
    // input- configuration
    auto n {5};

    // setup
    auto   finaloutput   {std::vector<std::vector<int>>(
        n,
        std::vector<int>(n, 0)
    )};

    //
    auto   top_edge      {0};
    auto   bottom_edge   {n-1};
    auto   left_edge     {0};
    auto   right_edge    {n-1};
    auto   count         {1};
    auto   curr_row      {0};
    auto   curr_col      {0};

    // running the loop
    while(count <= n*n){

        // moving from top left to top right
        for(auto col = left_edge; col <= right_edge && count <= n*n; ++col)     {finaloutput[curr_row][col] = count++;}
        curr_col = right_edge;

        // moving from topright to bottom right
        for(auto row = top_edge+1; row <= bottom_edge-1 && count <= n*n; ++row) {finaloutput[row][curr_col] = count++;}
        curr_row   =   bottom_edge;

        // moving from bottom-right to bottom-left
        for(auto col = right_edge; col >= left_edge && count <= n*n; --col)     {finaloutput[curr_row][col] = count++;}
        curr_col   =   left_edge;

        // moving from bottom-left to top-left
        for(auto row = bottom_edge-1; row >= top_edge+1 && count <= n*n; --row) {finaloutput[row][curr_col] = count++;}
```

```
        curr_row   =   top_edge+1;

        // updating boundaries
        ++top_edge;    --bottom_edge;
        ++left_edge;   --right_edge;
    }

    fPrintMatrix(finaloutput);


    // return
    return(0);

}
```

60. Permutation Sequence

The set [1, 2, 3, ..., 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 kth 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

- $1 \leq \text{n} \leq 9$
- $1 \leq \text{k} \leq n!$

Code

```cpp
string vectortostring(const vector<int>& pathsofar){
    auto temp  {string("")};
    for(int i = 0; i<pathsofar.size(); ++i) {temp += std::to_string(pathsofar[i]);}
    return temp;
}

void foo(vector<int>          pathsofar,
         const int&           n,
         int                  curr,
         int&                 leafcount,
         const int&           k,
         string&              finaloutput,
         bool&                found){

    // sending it back
    if (found == true) {return;}
```

```
    // adding current number to pathsofar
    if (std::find(pathsofar.begin(), pathsofar.end(), curr) != pathsofar.end()) {return;}

    // adding to the path
    pathsofar.push_back(curr);

    // checking if we've reached a leaf
    if (pathsofar.size() == n) {
        ++leafcount;
        if (leafcount == k) {
            finaloutput = vectortostring(pathsofar);
            found = true;
        }
        return;
    }

    // calling on the others
    for(int i = 1; i<=n; ++i) {foo(pathsofar, n, i, leafcount, k, finaloutput, found);}
}

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto n {8};
    auto k {29805};

    // setup
    auto finaloutput  {string("")};
    auto found        {false};

    // building the directing block
```

```
    auto directionalblocks {vector<int>(n-1, 1)};
    for(int i = directionalblocks.size()-2; i >= 0; --i)
        directionalblocks[i] = directionalblocks[i+1] * (directionalblocks.size()-i);

    // printing
    auto startingpoint {k/directionalblocks[0]};
    startingpoint = std::max(startingpoint, 1);

    // printing the leaf
    auto leafcount    {directionalblocks[0] * (startingpoint-1)};

    // running
    for(int i = startingpoint; i<=n; ++i){
        vector<int> pathsofar;
        foo(pathsofar, n, i, leafcount, k, finaloutput, found);
    }

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

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

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 $\leq$ Node.val $\leq$ 100
- $0 \leq k \leq 2 * 10^9$

Code

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto head  {new ListNode(1)};
    head->next = new ListNode(2);
    head->next->next = new ListNode(3);
    head->next->next->next = new ListNode(4);
    head->next->next->next->next = new ListNode(5);
    auto k {2};

    // trivial cases
    if (head == nullptr) {fPrintLinkedList("head = ", head); return 0;}

    // storing the starting point
    auto startingpoint {head};
    auto listsize    {0};

    // connecting the last-node with the first
    auto traveller {head};
    while(traveller->next) {traveller = traveller->next; ++listsize;}
    ++listsize;
    traveller->next = head;

    // to get rid of cycles
    k = k % listsize;
    auto cutoffpoint  {listsize - k};
    auto count        {0};
    auto finalresult  {static_cast<ListNode*>(nullptr)};

    // finding the cutting off point
```

```
    traveller = head;
    while(traveller){
        ++count;                                                        // appending count
        if (count == cutoffpoint){
            finalresult = traveller->next;                              // setting up the new head
            traveller->next = nullptr;                                  // setting up last pointer
            break;                                                      // breaking up
        }
        traveller = traveller->next;                                    // moving to the next point
    }

    // printing the linked list
    fPrintLinkedList("final-result = ", finalresult);

    // return
    return(0);

}
```

62. Unique Paths

There is a robot on an m x 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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto m {3};
    auto n {7};

    // setup
    auto row   = [&n](int x){return x/n;};
    auto col   = [&n](int x){return x%n;};
    auto tid   {m*n - 1};

    // building dp-table
    vector<vector<int>> dptable;
    for(int i = 0; i<m; ++i) {dptable.push_back(vector<int>(n, 0));}
    dptable[m-1][n-1] = 1;

    // moving through the thing
    while(tid>=0){

        auto currcol      {col(tid)};
        auto currrow      {row(tid)};
        auto nummethods   {dptable[currrow][currcol]};

        if (currcol < n-1) {nummethods += dptable[currrow][currcol+1];}
        if (currrow < m-1) {nummethods += dptable[currrow+1][currcol];}

        dptable[currrow][currcol] = nummethods;

        --tid;
    }

```

```
    // returning
    cout << format("final-output = {}\n", dptable[0][0]);

    // return
    return(0);

}
```

63. Unique Paths II

You are given an m x 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 == obstacleGrid.length
- n == obstacleGrid[i].length
- $1 \leq$ m, n ≤ 100
- obstacleGrid[i][j] is 0 or 1.

Code

```cpp
int main(){

    // starting timer
    Timer timer;

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

    // returning zero if the obstacleGrid has a 1 in the end
    if (obstacleGrid[obstacleGrid.size()-1][obstacleGrid[0].size()-1] == 1) return 0;

    // setup
    vector<vector<long long>> dptable;
    for(int i = 0; i<obstacleGrid.size()+1; ++i){
        dptable.push_back(vector<long long>(obstacleGrid[0].size()+1, 0));
    }

```

```cpp
    // writing a one in the destination
    dptable[obstacleGrid.size()-1][obstacleGrid[0].size()-1] = 1;

    // setting up the copying
    for(int i = obstacleGrid.size()-1; i>=0; --i){
        for(int j = obstacleGrid[0].size()-1; j>=0; --j){

            // leaving the paths from this as zero in the case where there is an obstacle.
            if (obstacleGrid[i][j] == 1) continue;

            // summing up from the right
            long long currvalue {dptable[i][j] + dptable[i+1][j] + dptable[i][j+1]};

            // writing the sum to the current-value
            dptable[i][j] = currvalue;
        }
    }

    // returning the starting point
    cout << format("final-output = {}\n", static_cast<int>(dptable[0][0]));

    // return
    return(0);

}
```

64. Minimum Path Sum

Given a m x 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 = $\begin{array}{|c|c|c|}\hline 1 & 3 & 1 \\ \hline 1 & 5 & 1 \\ \hline 4 & 2 & 1 \\ \hline \end{array}$
   - Output: 7
   - Explanation: Because the path $1 \rightarrow 3 \rightarrow 1 \rightarrow 1 \rightarrow 1$ minimizes the sum.

2. **Example 2:**

   - Input: grid = $\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline \end{array}$
   - Output: 12

Constraints

- m == grid.length

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

Code

```cpp
int main(){

    // starting timer
    Timer timer;

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

    // setup
    auto mingrid = grid;

    // going through the values from the bottom right to left
    for(int i = grid.size()*grid[0].size()-1; i>=0; --i){

        // converting numbers to the row--index and colindex
        auto rowindex  {i/static_cast<int>(grid[0].size())};
        auto colindex  {i%static_cast<int>(grid[0].size())};

        // finding sum of the children with present
        auto smallestchild {std::numeric_limits<int>::max()};

```

```
        // checking the bottom left
        if (rowindex == grid.size()-1 && colindex == grid[0].size()-1)   {smallestchild = 0; }
        else if(rowindex == grid.size()-1)                               {smallestchild = mingrid[rowindex][colindex+1]; }
        else if (colindex == grid[0].size()-1)                           {smallestchild = mingrid[rowindex+1][colindex];}
        else                                                             {smallestchild = min(mingrid[rowindex+1][colindex],
                                                                                  mingrid[rowindex][colindex+1]);}

        // storing to the final results
        mingrid[rowindex][colindex] = mingrid[rowindex][colindex] + smallestchild;

    }

    // returning the smallest value
    cout << format("final-output = {}\n", mingrid[0][0]);

    // return
    return(0);

}
```

66. Plus One

You are given a large integer represented as an integer array digits, where each digits[i] is the ith 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

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

Code

```cpp
int main(){

    // starting timer
    Timer timer;

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

    // setup
    auto carry  {1};
    auto sum   {-1};
    vector<int> finalOutput(digits.size()+1, 0);

    // goign through the numbers
    for(int i = 0; i<digits.size(); ++i){
```

```
        // calculating sum and carry
        sum    = digits[digits.size()-1-i] + carry;
        carry  = sum / 10;
        sum    = sum - carry*10;

        // storing sum and carry
        finalOutput[finalOutput.size()-1-i] = sum;
    }

    // in case there is one carry left
    if (carry != 0)   {finalOutput[0] = 1;}
    else              {finalOutput = vector<int>(finalOutput.begin()+1, finalOutput.end());}

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

    // return
    return(0);

}
```

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 ≤ 104
- a and b consist only of ’0’ or ’1’ characters.
- Each string does not contain leading zeros except for the zero itself.

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto a {string("1010")};
    auto b {string("1011")};


    // setup
    int carry {0};
    int a_top {0};
    int b_top {0};
    int c_top {0};
    string finalOutput {};

    // running until the things are empty
    while(a.size()!=0 || b.size()!=0){

        // copying the carry over tot he sum
        c_top = carry;

        // shaving off the top
        if (a.size()){
            a_top   = static_cast<int>(a[a.size()-1] - '0');   // taking the top
            a       = a.substr(0, a.size()-1);                 // removing the top
            c_top   += a_top;                                  // adding to the sum
        }
        if (b.size()){
            b_top   = static_cast<int>(b[b.size()-1] - '0');   // taking the top
            b       = b.substr(0, b.size()-1);                 // removing the top
            c_top   += b_top;                                  // adding to the sum
```

```
        }

        // evaluating carry
        if (c_top == 0)          {carry = 0; c_top = 0;}
        else if (c_top == 1)     {carry = 0; c_top = 1;}
        else if (c_top == 2)     {carry = 1; c_top = 0;}
        else if (c_top == 3)     {carry = 1; c_top = 1;}
        else                     {cout << format("what the actual fuck") << endl;}

        // adding c-TOp to the end of the final output
        finalOutput = std::string(1,c_top+'0') + finalOutput;

    }

    // check if there is a carry out there
    if (carry)     {finalOutput = std::string(1,carry+'0') + finalOutput;}

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

    // return
    return(0);

}
```

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

Constraints

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

Code

```
// function to calculate number of non-space characters
int fCalcLengthOfTempWithoutSpaces(std::vector<std::string> temp){
    int num_nonspaces = 0;
    for(auto x: temp) num_nonspaces += x.size();
    return num_nonspaces;
}

// function to calculate words formed with temp
int fCalcLengthOfTempWithSpaces(std::vector<std::string> temp){
    int num_nonspaces = 0;
    for(auto x: temp) num_nonspaces += 1+ x.size();
    return num_nonspaces-1;
}

// printing temp
void fPrintTemp(std::vector<std::string> temp){
    // printing temp
    std::cout << "temp = ";
    for(auto x: temp) std::cout << x << ",";
```

```cpp
    std::cout << std::endl;
}

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

    // input- configuration
    auto words    = vector<string>({"This", "is", "an", "example", "of", "text", "justification."});
    auto maxWidth {16};

        // setup
    std::vector<std::string> finalOutput;

    // going through strings
    int acc = 0;
    int numwords = 0;
    int currwidth = 0;
    std::vector<std::string> temp;

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

        // updating temp
        temp.push_back(words[i]);    // updating words in temp

        // checking if width has been crossed
        if (fCalcLengthOfTempWithSpaces(temp) >= maxWidth || i == words.size()-1){

            // condition temp based on length
            if(fCalcLengthOfTempWithSpaces(temp)>maxWidth){
                temp.pop_back();      // last words gotta go
                --i;                  // making sure its taken care of in next iteration
            }

            // finding length of characters in temp
            int num_nonspaces = fCalcLengthOfTempWithoutSpaces(temp);
```

```cpp
        // finding number of spaces to add
        int numspacestofill = maxWidth - num_nonspaces;

        // calculating numspots
        int numspots = temp.size()-1;

        // calculating ideal number of spaces
        int idealnumspacesperspot;
        int remainders;
        if (numspots!=0){
            idealnumspacesperspot = numspacestofill/numspots;
            remainders            = numspacestofill%numspots;
        }
        else{
            idealnumspacesperspot = numspacestofill/1;
            remainders            = numspacestofill%1;
        }

        // constructing candidate string
        std::string candidate;

        // adding each word in temp to the candidate
        for(int j = 0; j < temp.size(); ++j){

            // fetching word
            auto x = temp[j];

            // adding word to candidate
            candidate += x;

            // adding spaces
            if (j!=temp.size()-1)
                for(int var00 = 0; var00 < idealnumspacesperspot; ++var00)
                    candidate += " ";
```

```
                // checking if there is any remainder left
                if (remainders > 0){
                    candidate += " ";                                       // adding another space
                    --remainders;                                           // subtracting
                }
            }

            // checking if there are remaindeers
            while (remainders > 0){
                candidate += " ";                                           // adding another space
                --remainders;                                               // subtracting
            }

            while (candidate.size()!=maxWidth) {candidate += " ";}          // adding another space

            // appending candidate to final output
            finalOutput.push_back(candidate);

            // getting rid of everything
            if (i != words.size()-1) {temp.clear();}
        }
    }

    // making function left justified
    std::string lastline;
    for(int i = 0; i<temp.size(); ++i){
        lastline += temp[i];
        if (i!=temp.size()-1) {lastline += " ";}
    }

    // adding spaces until end
    while(lastline.size()!=maxWidth) {lastline += " ";}

    // replacing last line
```

```
    finalOutput[finalOutput.size()-1] = lastline;

    // return
    return(0);

}
```

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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto x {8};

    // setup
    int finaloutput {-1};
    auto square = [](int x){return x*x;};

    // running
    for(int i = 0; i<x; ++i){
        cout << format("[{},{},{}]\n",square(i),x,square(i+1));
        if (square(i) <= x && square(i+1)>x) {finaloutput = i; break;}
    }

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

    // return
    return(0);

}
```

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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto n {2};

    // setup
    std::vector<int> dpTable;

    // number of steps from the last step to the last
    dpTable.push_back(0);
    dpTable.push_back(1);
    dpTable.push_back(2);

    // finding number of ways this can be done
    for(int i = 2; i<n; ++i){
        // adding up the last two values
        dpTable.push_back(dpTable[dpTable.size() - 1] + dpTable[dpTable.size() - 2]);
    }

    // return the final output
    cout << format("final-output = {}\n", dpTable[n]);

    // return
    return(0);

}
```

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

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

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    string path    {"/home//foo/"};

    // setup
    vector<string> stack;

    // going through the array
    int i = 0 ;
    while (i < path.size()){

        // moving until end of string or a /
        string temp;
        while(path[i]!='/' && i<path.size()){
            temp += std::string(1,path[i++]);
        }

        // continuing if temp size is zero
        if (temp.size()==0)   {++i; continue;}

        // evaluating the current string
        if (temp == ".")      {++i; continue;}
        else if (temp == "..") {
            if (stack.size()!=0) {stack.pop_back();}
            ++i;}
        else   {stack.push_back(temp); ++i;}
    }


```

```
    // reconstructing final path
    string finalOutput;

    // reconstructing
    for(auto x: stack) {finalOutput += "/" + x;}

    // size thing
    if (finalOutput.size() == 0) {finalOutput += "/";}

    // returning
    cout << format("finalOutput = {}\n", finalOutput);

    // return
    return(0);

}
```

73. Set Matrix Zeroes

Given an m x 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 == matrix.length
- n == matrix[0].length
- $1 \leq m, n \leq 200$
- $-231 \leq matrix[i][j] \leq 231 - 1$

Code

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    vector<vector<int>> matrix{
        {0,1,2,0},
        {3,4,5,2},
        {1,3,1,5}
    };

    // setup
    auto    colregister    {vector<bool>(matrix[0].size(), false)};
    auto    rowregister    {vector<bool>(matrix.size(), false)};

    // registering
    #pragma omp parallel for
    for(int i = 0; i<matrix.size(); ++i)
        for(int j = 0; j<matrix[0].size(); ++j)
            if (matrix[i][j] == 0) {colregister[j] = true; rowregister[i] = true;}

    // rewriting
    #pragma omp parallel for
    for(int i = 0; i<matrix.size(); ++i)
        for(int j = 0; j<matrix[0].size(); ++j)
            if (colregister[j] == true || rowregister[i] == true) {matrix[i][j] = 0;}

    // printing the final output
    cout << format("final-output \n");
    fPrintMatrix(matrix);

    // return
```

```
    return(0);

}
```

74. Search a 2D Matrix

You are given an m x n integer matrix 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 == matrix[i].length
- $1 \leq$ m, n ≤ 100
- $-10^4 \leq$ matrix[i][j], target $\leq 10^4$

Code

```
struct Matrix{
    vector<vector<int>> matrix;
    int                 numcols;
    int                 numrows;
    Matrix(vector<vector<int>> matrix0): matrix(matrix0){
        numcols = matrix0[0].size();
        numrows = matrix0.size();
    }
    int operator()(int row, int col) {return matrix[row][col];}
    int operator[](int tid)         {return matrix[tid/numcols][tid%numcols];}
    int numel()                     {return matrix.size() * matrix[0].size();}
};

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    vector<vector<int>> matrix = {
        {1,3,5,7},
        {10,11,16,20},
        {23,30,34,60}};
    auto target {3};

```

```
    // setup
    Matrix m(matrix);
    int left   {0};
    int right  {static_cast<int>(matrix.size() * matrix[0].size()) - 1};
    int mid    {-1};

    // binary search
    while(left <= right){
        mid = (left + right)/2;
        if (m[mid] < target)      {left = mid+1;}
        else if (m[mid] > target) {right = mid-1;}
        else                      {cout << format("final-output = true\n"); return 0;}
    }

    // returning false
    cout << format("final-output = false\n");

    // return
    return(0);

}
```

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

```cpp
int main(){

    // starting timer
    Timer timer;

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

    // setup
    vector<int> histogram(3, 0); histogram.reserve(histogram.size());

    // moving through inputs
    for(const auto& x: nums) {++histogram[x];}

    // rewriting
    auto counter   {0};
    for(int i = 0; i<nums.size(); ++i){
        while(counter < histogram.size() && histogram[counter] == 0) ++counter;
        nums[i] = counter; histogram[counter]--;
    }

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

    // return
    return(0);

}
```

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 = "ADOBECODEBANC", t = "ABC"`
   - Output: `"BANC"`
   - Explanation: The minimum window substring "BANC" includes 'A', 'B', and 'C' from string t.
2. **Example 2:**
   - Input: `s = "a", t = "a"`
   - Output: `"a"`
   - Explanation: The entire string s is the minimum window.
3. **Example 3:**
   - Input: `s = "a", t = "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

```
// function to convert letter to index
int fCharToIndex(char x){
    if (x >= 65 && x <=90)       {return x - 65;}
    else if (x >= 97 && x <= 122) {return (x - 97 + (90-65+1));}
    else                          {std::cerr<<"this shouldn't happen"; return -1;}
}

// compare histograms
template<typename T>
bool fCompareHist(const vector<T>& t_hist, const vector<T>& runninghist){

    // going throguh the list
    for(int i = 0; i<t_hist.size(); ++i)
        if (t_hist[i] > runninghist[i])
            return false;

    // in case all conditions are met
    return true;
}
int main(){

```

```cpp
    // starting timer
    Timer timer;

    // input- configuration
    auto s {string("ADOBECODEBANC")};
    auto t {string("ABC")};

    // setup
    int p1 {0};
    int p2 {0};
    vector<int> finalOutputBoundaries(2, -1);
    vector<int> t_hist(52, 0); t_hist.reserve(t_hist.size());
    vector<int> runninghist(52, 0); runninghist.reserve(runninghist.size());
    auto minlength {std::numeric_limits<int>::max()};

    // setting up t-hist
    for(const auto x: t)  {++t_hist[fCharToIndex(x)];}

    // moving through the string
    while(p1 <= p2 && p2<s.size()){

        // adding current character
        if (p2<s.size())  {++runninghist[fCharToIndex(s[p2])];}

        // checking if the two histograms are comparable
        while(p1<=p2 && fCompareHist(t_hist, runninghist)){

            auto currstringsize   {p2-p1+1};

            if (currstringsize>0 && currstringsize < minlength){
                finalOutputBoundaries = vector<int>({p1, p2});
                minlength             = currstringsize;
            }

            --runninghist[fCharToIndex(s[p1])];
```

```
            ++p1;
        }

        // incrementing p2;
        ++p2;

    }

    // printing the final output
    string finalOutput {string(s.begin()+finalOutputBoundaries[0], s.begin() + finalOutputBoundaries[1]+1)};

    // returning
    if (finalOutputBoundaries[0] == -1 || finalOutputBoundaries[1] == -1) {finalOutput = "";}

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

    // return
    return(0);

}
```

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

```
void foo(vector<int>          pathsofar,
         int                  n,
         int                  k,
         vector<vector<int>>& finalOutput){

    // checking legnth of pathsofar
    if (pathsofar.size() > k) return;

    // adding the current path to the final output if the length is k
    if (pathsofar.size() == k) {finalOutput.push_back(pathsofar); return;}

    // setting up starting point
    int startingpoint;
    if (pathsofar.size() == 0) startingpoint = 0+1;
    else                       startingpoint = pathsofar[pathsofar.size()-1]+1;

    // running through the options
    for(int i = startingpoint; i<=n; ++i){

        // calling the graph on those
        auto pathsofar_temp = pathsofar;
        pathsofar_temp.push_back(i);

        // calling the functiona again
        foo(pathsofar_temp, n, k, finalOutput);
    }

    // returning
    return;
}

int main(){

```

```
    // starting timer
    Timer timer;

    // input- configuration
    auto n {4};
    auto k {2};

    // setup
    vector< vector<int> > finalOutput;
    vector<int>           pathsofar;

    // calling the function
    foo(pathsofar, n, k, finalOutput);

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

    // return
    return(0);

}
```

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

- $1 \leq$ nums.length ≤ 10
- $-10 \leq$ nums[i] ≤ 10
- All the numbers of nums are unique.

Code

```cpp
// recursion function
void foo(vector<vector<int>>& finalOutput,
         const vector<int>&   nums,
         int                  currindex,
         vector<int>          runningvec,
         const int&           maxnumelements)
{
    // checking size
    if (runningvec.size() == maxnumelements) {finalOutput.push_back(runningvec);return;}

    // checking the next set of options
    for(int i = currindex+1; i < nums.size(); ++i){

        // creating arguments for next call
        auto runningvec_temp  {runningvec};
        runningvec_temp.push_back(nums[i]);
        foo(finalOutput, nums, i, runningvec_temp, maxnumelements);

    }

    // returning
    return;
}

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto nums  {1,2,3};

    // setup
```

```
    vector<vector<int>> finalOutput {vector<int>({})};

    // calling the function
    #pragma omp parallel for
    for(int i = 1; i<=nums.size(); ++i)
        foo(finalOutput, nums, -1, vector<int>({}), i);

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

    // return
    return(0);

}
```

80. Remove Duplicates from Sorted Array II

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

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

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

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

1. **Example 1**

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

2. **Example 2**

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

Code

```
int main(){

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

```

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

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

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

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

    }

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

    // return
    return(0);

}
```

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 = [1,2,3,3,4,4,5]
    - 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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto head                                   {new ListNode(1)};
    head->next                                  = new ListNode(2);
    head->next->next                            = new ListNode(3);
    head->next->next->next                      = new ListNode(3);
    head->next->next->next->next                = new ListNode(4);
    head->next->next->next->next->next          = new ListNode(4);
    head->next->next->next->next->next->next    = new ListNode(5);

    // setup
    auto traveller           {head};
    auto prehead_finaloutput {new ListNode(std::numeric_limits<int>::min())};
    auto traveller_finaloutput {prehead_finaloutput};

    auto currvalue           {-1};
    auto nextvalue           {-1};

    // going through the list
    while(traveller){

        // checking if current-value is different from top
        currvalue = traveller->val;
        nextvalue = traveller->next == nullptr ?
                    std::numeric_limits<int>::min() : traveller->next->val;

        // checking if they're the same
        if (currvalue == nextvalue){
            // moving until we find a new-value
```

```
            while(traveller && traveller->val == currvalue) {traveller = traveller->next;}
        }
        else{
            // pushing current-value
            traveller_finaloutput->next = traveller;
            traveller_finaloutput     = traveller_finaloutput->next;
            traveller                 = traveller->next;
        }
    }

    // printing the final-output
    fPrintLinkedList("final-output = ", prehead_finaloutput->next);

    // return
    return(0);

}
```

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

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto head  {new ListNode(1)};
    head->next = new ListNode(4);
    head->next->next = new ListNode(3);
    head->next->next->next = new ListNode(2);
    head->next->next->next->next = new ListNode(5);
    head->next->next->next->next->next = new ListNode(2);
    auto x {3};

    // setup
    ListNode* traveller      = head;
    ListNode* lefthead       = new ListNode(-1);
    ListNode* lefttraveller  = lefthead;
    ListNode* righthead      = new ListNode(-1);
    ListNode* righttraveller = righthead;

    // going through the list
    while(traveller){
        // checking next-value
        if (traveller->val < x){
            lefttraveller->next   = new ListNode(traveller->val);
            lefttraveller         = lefttraveller->next;
        }
        else {
            righttraveller->next  = new ListNode(traveller->val);
            righttraveller        = righttraveller->next;
        }

```

```
        // moving traveller
        traveller = traveller->next;
    }

    // connecting
    lefttraveller->next = righthead->next;

    // returning
    fPrintLinkedList("Final-output = ", lefthead->next);

    // return
    return(0);

}
```

88. Merge Sorted Array

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

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

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

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

Code

```
int main(){

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

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

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

    // going the other way
    while(p1 >= 0 || p2 >= 0)
    {
        // printing the values
        curr1 = p1 >= 0 ? nums1[p1] : std::numeric_limits<int>::min();
        curr2 = p2 >= 0 ? nums2[p2] : std::numeric_limits<int>::min();

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

    }

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

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

92. Reverse Linked List II

Given the head of a singly linked list and two integers left and right where left ¡= 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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    ListNode* head                    = new ListNode(1);
    head->next                        = new ListNode(2);
    head->next->next                  = new ListNode(3);
    head->next->next->next            = new ListNode(4);
    head->next->next->next->next = new ListNode(5);
    auto left  {2};
    auto right {4};

    // trivial case
    if (left == right) {fPrintLinkedList(head); cout << endl; return 0;}

    // setup
    ListNode* traveller   = nullptr;
    ListNode* entrypoint  = nullptr;
    ListNode* exitpoint   = nullptr;
    ListNode* revhead     = nullptr;
    ListNode* dummy       = new ListNode(0);
    dummy->next           = head;

    // moving through list
    int nodeposition = -1;
    traveller = dummy;
    while(traveller){

        ++nodeposition;   // updating node position

        // finding important points
```

```
        if(nodeposition == left-1)      {entrypoint = traveller;}
        else if(nodeposition == left)   {revhead   = traveller;}
        else if(nodeposition == right+1) {exitpoint = traveller;}

        // moving traveller to next point
        traveller = traveller->next;
    }

    // reversing the head
    ListNode* prev        = nullptr;
    ListNode* actualright = nullptr;
    traveller             = revhead;
    nodeposition          = left;

    while(traveller && nodeposition <= right){
        ++nodeposition;
        actualright    = traveller->next; // storing original right
        traveller->next = prev;          // reconnection
        prev           = traveller;      // storing prev for next iteration
        traveller      = actualright;    // moving to original right
    }

    // tying things together
    entrypoint->next  = prev;
    revhead->next     = exitpoint;

    // returning dummy-es nstex
    fPrintLinkedList(dummy->next); cout << endl;

    // return
    return(0);

}
```

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

```cpp
void foo(TreeNode*    root,
        vector<int>&  finalOutput)
{
    // sending it back
    if (root == nullptr) return;

    // sending it left
    foo(root->left, finalOutput);

    // adding the current-value
    finalOutput.push_back(root->val);

    // sending it right
    foo(root->right, finalOutput);

    // sending it back
    return;
}
```

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root         {new TreeNode(1)};
    root->right       = new TreeNode(2);
    root->right->left = new TreeNode(3);

    // setup
    vector<int> finalOutput;

    // calling the function
    foo(root, finalOutput);

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

    // return
    return(0);

}
```

97. Interleaving String

Given strings s1, s2, and s3, find whether s3 is formed by an interleaving of s1 and s2.

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:

- $s = s1 + s2 + ... + s_n$
- $t = t1 + t2 + ... + t_m$
- $|n - m| \leq 1$
- The interleaving is $s1 + t1 + s2 + t2 + s3 + t3 + ...$ or $t1 + s1 + t2 + s2 + t3 + s3 + ...$

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

Examples

1. **Example 1:**
    - Input: `s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac"`
    - Output: true
    - Explanation: One way to obtain s3 is:
        - Split s1 into `s1 = "aa" + "bc" + "c"`, and s2 into `s2 = "dbbc" + "a"`. Interleaving the two splits, we get `"aa" + "dbbc" +`

      Since s3 can be obtained by interleaving s1 and s2, we return true.
2. **Example 2:**
    - Input: `s1 = "aabcc", s2 = "dbbca", s3 = "aadbbbaccc"`

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

Code

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto s1    {string("aabcc")};
    auto s2    {string("dbbca")};
    auto s3    {string("aadbbcbcac")};

    // trivial case
    if (s1.size() + s2.size() != s3.size()) return static_cast<bool>(0);
```

```cpp

    // setup
    vector<vector<bool>> dptable;

    // creating dptable
    for(int i = 0; i<s1.size()+1; ++i)  {dptable.push_back(vector<bool>(s2.size()+1, false));}

    // storing a true at the real end of the table
    dptable[dptable.size()-1][dptable[0].size()-1] = true;

    // moving back from the end
    for(int i = dptable.size()-1; i>=0; --i){
        for(int j = dptable[0].size()-1; j>=0; --j){

            // fetching the character at this position
            auto s2_char  {s2[j]};
            auto s1_char  {s1[i]};
            auto s3_char  {s3[i + j]};

            // checking if we can move right
            if (s1_char == s3_char)  {dptable[i][j] = dptable[i][j] + ((i+1) < dptable.size() ? dptable[i+1][j] : 0);}

            // checking if we can move down
            if (s2_char == s3_char)  {dptable[i][j] = dptable[i][j] + ((j+1) < dptable[0].size() ? dptable[i][j+1] : 0);}

        }
    }

    // returning the first value
    cout << format("final-output = {}\n", dptable[0][0]);

    // return
    return(0);

}
```

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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root      {new TreeNode(2)};
    root->left     = new TreeNode(1);
    root->right    = new TreeNode(3);

```

```
// setup
bool validBSTFlag = true;
vector<int> nodevalues;

std::function<void(const TreeNode*)> foo = [&foo, &nodevalues, &validBSTFlag](
    const TreeNode* root){

    // if false, stop everything
    if (validBSTFlag == false)   {return;}

    // returning
    if (root == nullptr)         {return;}

    // going left
    if (root->left)              {foo(root->left);}

    // adding current
    if (nodevalues.size() == 0)  {nodevalues.push_back(root->val);}
    else{
        // check top
        auto topvalue = nodevalues[nodevalues.size()-1];

        // check if top value is less than or equal
        if (topvalue >= root->val) {validBSTFlag = false;}            // since this is not valid
        else                     {nodevalues.push_back(root->val);}
    }

    // going right
    if (root->right)             {foo(root->right);}
};

// calling function
foo(root);

// returning
```

```
    cout << format("final-output = {}\n", validBSTFlag);

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

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

```
// the function
void foo(TreeNode*                root,
         vector<TreeNode*>&       nodeaddresses,
         vector<int>&             nodevalues){

    // sending it back
    if (root == nullptr) return;

    // going through the left
    foo(root->left, nodeaddresses, nodevalues);

    // adding the current
    nodeaddresses.push_back( root);
    nodevalues.push_back(    root->val);

    // going right
    foo(root->right, nodeaddresses, nodevalues);

    // returning
    return;
}

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root         {new TreeNode(1)};
    root->left        = new TreeNode(3);
    root->left->right = new TreeNode(2);

    // setup
```

```
    vector<TreeNode*> nodeaddresses;
    vector<int>       nodevalues;

    // calling the function
    foo(root, nodeaddresses, nodevalues);

    // sort the values and assign it
    std::sort(nodevalues.begin(), nodevalues.end());

    // writing the values as is
    #pragma omp
    for(int i = 0; i<nodeaddresses.size(); ++i){
        nodeaddresses[i]->val = nodevalues[i];
    }

    // return
    return(0);

}
```

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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto p {new TreeNode(1)};
    p->left    = new TreeNode(2);
    p->right   = new TreeNode(3);

    auto q {new TreeNode(1)};
    q->left    = new TreeNode(2);
    q->right   = new TreeNode(3);

    // setup
    std::function<bool(TreeNode*, TreeNode*)> fCompareTrees = [&fCompareTrees](
        TreeNode* p, TreeNode* q){

        // basecase
        if (p == nullptr && q == nullptr)   return true;
        if (p == nullptr && q != nullptr)   return false;
        if (p != nullptr && q == nullptr)   return false;
        if (p->val != q->val)               return false;

        // going through the next set of branches
        return (fCompareTrees(p->left, q->left) && fCompareTrees(p->right, q->right));
    };

    // going through the two nodes
    bool finalOutput = fCompareTrees(p, q);

    // returning final output
    cout << format("final-output = {}\n", finalOutput);
```

```
    // return
    return(0);

}
```

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 $\leq$ Node.val $\leq$ 100

Code

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root         {new TreeNode(1)};
    root->left         = new TreeNode(2);
    root->right        = new TreeNode(2);
    root->left->left   = new TreeNode(3);
    root->left->right  = new TreeNode(4);
    root->right->left  = new TreeNode(4);
    root->right->right = new TreeNode(3);

    // setup
    std::function<bool(TreeNode*, TreeNode*)> dfs = [&dfs](
        TreeNode* leftNode,
        TreeNode* rightNode){

        // printing the base-cases
        if (leftNode == nullptr && rightNode == nullptr) return true;
        if (leftNode == nullptr || rightNode == nullptr) return false;

        // calling the children
        bool finaloutput = (leftNode->val == rightNode->val)      && \
                           dfs(leftNode->left, rightNode->right)   && \
                           dfs(leftNode->right, rightNode->left);

        // returning the value
        return finaloutput;
    };

    // running
    cout << format("final-output = {}\n", dfs(root->left, root->right));
```

```

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

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 $\leq$ Node.val $\leq$ 1000

Code

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root               {new TreeNode(3)};
    root->left              = new TreeNode(9);
    root->right             = new TreeNode(20);
    root->right->left       = new TreeNode(15);
    root->right->right      = new TreeNode(7);

    // setup
    std::vector< std::vector<int> > finalOutput;
    std::function<void(const TreeNode*, int)> f = [&f, &finalOutput](
        const TreeNode* root,
        int             level){

        // base-case
```

```cpp
        if (root == nullptr) return;

        // finding current-level
        level += 1;

        // increasing size of the thing
        while (finalOutput.size()<level+1)
            finalOutput.push_back(std::vector<int>());

        // appending to value
        if (finalOutput.size() < level+1)
            finalOutput[level] = std::vector<int>(root->val);
        else
            finalOutput[level].push_back(root->val);

        // calling
        f(root->left, level);
        f(root->right, level);

        // returning return;
        return;

    };

    // going through the tree
    f(root, -1);

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

    // return
    return(0);

}
```

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 $\leq$ Node.val $\leq$ 100

Code

```cpp
void f(TreeNode*                          root,
       std::vector< std::vector<int> >&  finalOutput,
       int                                level)
{

    // base-case
    if (root == nullptr) return;

    // finding current-level
    level += 1;

    // increasing size of the thing
    while (finalOutput.size()<level+1) {finalOutput.push_back(std::vector<int>());}

    // appending to value
    if (finalOutput.size() < level+1 ) {finalOutput[level] = std::vector<int>(root->val);}
    else{
        // appending based on level
        if (level%2 == 0)
            finalOutput[level].push_back(root->val);
        else
        {
            std::vector<int> temp {root->val};
            for(auto x: finalOutput[level]) {temp.push_back(x);}
            finalOutput[level] = temp;
        }
    }

    // calling
    f(root->left, finalOutput, level);
    f(root->right, finalOutput, level);

    // returning return;
```

```cpp
    return;
}
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root               {new TreeNode(3)};
    root->left              = new TreeNode(9);
    root->right             = new TreeNode(20);
    root->right->left       = new TreeNode(15);
    root->right->right      = new TreeNode(7);

    // setup
    std::vector< std::vector<int> > finalOutput;

    // going through the tree
    f(root, finalOutput, -1);

    // sending back final output
    cout << format("final-output = {}\n", finalOutput);

    // return
    return(0);

}
```

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 $\leq$ Node.val $\leq$ 100

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root         {new TreeNode(3)};
    root->left         = new TreeNode(9);
    root->right        = new TreeNode(20);
    root->right->left = new TreeNode(15);
    root->right->right = new TreeNode(7);

    // setup
    auto maxcount  {0};
    auto count     {0};

    // lambda - function
    std::function<void(const TreeNode*, int)> fTraverse = [&fTraverse,
                                                  &maxcount](
        const TreeNode* root,
        int             count){

        // checking if null
        if (root == nullptr) return;

        ++count;                                     // updating length
        maxcount = max(maxcount, count);             // checking maxlength

        fTraverse(root->left, count);                // checking if we can go left
        fTraverse(root->right, count);               // checking if we can go right

        // returning
        return;
    };
```

```

    // calling the function
    fTraverse(root, count);

    // returning
    cout << format("final-output = {}\n", maxcount);

    // return
    return(0);

}
```

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

- $1 \leq$ preorder.length ≤ 3000
- inorder.length == preorder.length
- $-3000 \leq$ preorder[i], inorder[i] ≤ 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

```
TreeNode* foo(vector<int> preorder, vector<int> inorder)
{

    // returning for trivial case
    if (preorder.size() == 0 || inorder.size() == 0) {return static_cast<TreeNode*>(nullptr);}

    // setup
    auto& rootvalue   {preorder[0]};
    auto root         {new TreeNode(preorder[0])};

    // sending it back for trivial case
    if (preorder.size() == 1 && inorder.size() == 1) {return root;}

    // finding root-position
    auto rootposition = std::find(inorder.begin(),
                                  inorder.end(),
                                  rootvalue);

    // creating left-subtree
    auto leftinorder {vector<int>(inorder.begin(), rootposition)};
    auto rightinorder {vector<int>(rootposition+1, inorder.end())};

    // creating subset of preorder
    auto leftpreorder {vector<int>(preorder.begin()+1,
                                   preorder.begin() + leftinorder.size() + 1)};
    auto rightpreorder {vector<int>(preorder.begin() + leftinorder.size()+1,
                                   preorder.end())};
```

```
    // calling the function on left and right
    root->left     = foo(leftpreorder,   leftinorder);
    root->right    = foo(rightpreorder,  rightinorder);

    // returning the
    return root;

}
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto preorder  {vector<int>{3,9,20,15,7}};
    auto inorder   {vector<int>{9,3,15,20,7}};

    // manually building the tree
    auto root  {foo(preorder, inorder)};

    // return
    return(0);

}
```

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 $\leq$ Node.val $\leq$ 1000

Code

```
void foo(TreeNode*                                root,
         map<int, vector<int>, std::greater<int>>& levelToMembers,
         int                                      currlevel){

    // sending it back
    if (root==nullptr) return;

    // incrementing currlevel
    ++currlevel;

    // moving left
    foo(root->left, levelToMembers, currlevel);

    // checking if the current-level is present
    if (levelToMembers.find(currlevel) == levelToMembers.end())
        levelToMembers[currlevel] = vector<int>({root->val});
    else
        levelToMembers[currlevel].push_back(root->val);

    // moving right
    foo(root->right, levelToMembers, currlevel);

    // returning
    return;
}

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root         {new TreeNode()};
```

```cpp
    root->left        = new TreeNode(9);
    root->right       = new TreeNode(20);
    root->right->left = new TreeNode(15);
    root->right->right = new TreeNode(7);

    // setup
    map<int, vector<int>, std::greater<int>> levelToMembers;
    vector<vector<int>>                  finalOutput;

    // running the function
    foo(root, levelToMembers, -1);

    // printing the values
    #pragma omp parallel for
    for(auto x: levelToMembers)  {finalOutput.push_back(x.second);}

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

    // return
    return(0);

}
```

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

```
void foo(TreeNode*       parent,
         int             parentdirection,
         vector<int>&    nums,
         int             left,
         int             right)
{
    // sending it back if right is less than left
    if (left > right) return;

    // finding mid point
    int mid = (left + right)/2;

    // creating mid-node
    TreeNode* midnode = new TreeNode(nums[mid]);

    // connecting current to parent node
```

```
    if (parentdirection == -1)       {parent->right = midnode;}
    else if (parentdirection == 1)   {parent->left  = midnode;}

    // going left
    foo(midnode, 1, nums, left, mid-1);

    // going right
    foo(midnode, -1, nums, mid+1, right);

    // returning
    return;

}
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    vector<int> nums  {-10,-3,0,5,9};

    // setup
    TreeNode* dummy = new TreeNode(-1);

    // calling the function
    foo(dummy, -1, nums, 0, nums.size()-1);

    // return
    return(0);

}
```

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 $\leq$ Node.val $\leq$ 105

Code

```
void foo(TreeNode*       parent,
        int               parentdirection,
        vector<int>&      nums,
        int               left,
        int               right)
{
    // sending it back if right is less than left
    if (left > right) return;

    // finding mid point
    int mid = (left + right)/2;

    // creating mid-node
    TreeNode* midnode = new TreeNode(nums[mid]);

    // connecting current to parent node
    if (parentdirection == -1)   parent->right = midnode;
    else if (parentdirection == 1) parent->left = midnode;

```

```
    // going left
    foo(midnode, 1, nums, left, mid-1);

    // going right
    foo(midnode, -1, nums, mid+1, right);

    // returning
    return;

}

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto head                       {new ListNode(-10)};
    head->next                      = new ListNode(-3);
    head->next->next                = new ListNode(0);
    head->next->next->next          = new ListNode(5);
    head->next->next->next->next    = new ListNode(9);

    // sending it back if the list is empty
    if (head == nullptr)     {cout << format("final-output = \n");}

    // making a vector out of the whole list
    auto traveller = head;
    vector<int> nums;
    while(traveller){
        nums.push_back(traveller->val);
        traveller = traveller->next;
    }

    // calling the previous function ito this
```

```
    TreeNode* dummy = new TreeNode(-1);

    // calling the function
    foo(dummy, -1, nums, 0, nums.size()-1);

    // return
    return(0);

}
```

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

```
int foo(const TreeNode*  root,
        int              currlevel,
        bool&            aightstop){

    // sending it back if null
    if (root == nullptr)  {return 0;}

    // checking if flag is up
    if (aightstop == true) {return -1;}

    // incrementing levle
    ++currlevel;

    // going left and right
    auto leftdepth    {foo(root->left, currlevel, aightstop)};
    auto rightdepth   {foo(root->right, currlevel, aightstop)};

    auto diff = std::abs(rightdepth - leftdepth);

```

```
    // checking if depth is the same
    if (diff>1)           {aightstop = true;}

    // returning the difference
    return (std::max(leftdepth, rightdepth)+1);

}

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root          {new TreeNode(3)};
    root->left         = new TreeNode(9);
    root->right        = new TreeNode(20);
    root->right->left  = new TreeNode(15);
    root->right->right = new TreeNode(7);

    // trivial case
    if (root == nullptr) {cout << format("final-output = {}\n", true);}

    // setup
    int     currlevel  {-1};
    bool    aightstop  {false};

    // running the function
    foo(root, currlevel, aightstop);

    // returning the status
    cout << format("final-output = {}\n", !aightstop);

    // return
    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 $\leq$ Node.val $\leq$ 1000

Code

```
void foo(TreeNode* root,
         int       currlevel,
         int&      minlevel)
{
    // sending it back
    if (root == nullptr)  {return;}
    if (minlevel == 2)    {return;}

    // incrementing level
    ++currlevel;

    // checking if child
    if (root->left == nullptr && root->right == nullptr){
        minlevel = std::min(minlevel, currlevel);
        return;
```

```cpp
    }

    // calling functions
    foo(root->left,   currlevel, minlevel);
    foo(root->right,  currlevel, minlevel);

    // returning
    return;
}

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root         {new TreeNode(3)};
    root->left         = new TreeNode(9);
    root->right        = new TreeNode(20);
    root->right->left = new TreeNode(15);
    root->right->right = new TreeNode(7);

    // trivial case
    if (root == nullptr)  {cout << format("final-output = {}\n", 0);}

    // setup
    auto currlevel {0};
    auto minlevel  {std::numeric_limits<int>::max()};

    // calling the function
    foo(root, currlevel, minlevel);

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

```

```
    // return
    return(0);

}
```

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 $\leq$ Node.val $\leq$ 1000
- -1000 $\leq$ targetSum $\leq$ 1000

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root                {new TreeNode(5)};

    root->left               = new TreeNode(4);
    root->right              = new TreeNode(8);

    root->left->left         = new TreeNode(11);
    root->right->left        = new TreeNode(13);
    root->right->right       = new TreeNode(4);

```

```
    root->left->left->left   = new TreeNode(7);
    root->left->left->right  = new TreeNode(2);
    root->right->right->right = new TreeNode(1);

    auto targetSum    {22};

    // setup
    auto runningsum   {0};
    auto binaryflag   {false};
    std::function<void(const TreeNode*, int)> fRunSum = [&fRunSum,
                  &targetSum,
                  &binaryflag](
        const TreeNode* root,
        int         runningsum
        ){

        // sending it back
        if (root       == nullptr)    return;
        if (binaryflag == true)       return;

        // adding current val to running sum
        runningsum += root->val;

        // checking if there are children
        if (root->left || root->right){
            if (root->left)   fRunSum(root->left,   runningsum);
            if (root->right)  fRunSum(root->right,  runningsum);
        }
        else{
            // in this case, we check the running sum
            if (runningsum == targetSum){ binaryflag = true;}
        }

        // returning
        return;
```

```
    };

    // running
    fRunSum(root, runningsum);

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

    // return
    return(0);

}
```

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 $\leq$ Node.val $\leq$ 1000
- -1000 $\leq$ targetSum $\leq$ 1000

Code

```cpp
void foo(TreeNode*             root,
         const int             targetSum,
         int                   runningsum,
         vector<int>           runningvector,
         vector<vector<int>>& finalOutput)
{
    // sending it back
    if (root==nullptr) return;

    // adding current-value to running vector
    runningvector.push_back(root->val);
    runningsum += root->val;

```

```cpp
    // checking if current-value has reached target sum
    if (runningsum == targetSum &&
        root->left == nullptr &&
        root->right == nullptr){
        finalOutput.push_back(runningvector); return;
    }

    // going down left and right
    foo(root->left, targetSum, runningsum, runningvector, finalOutput);
    foo(root->right, targetSum, runningsum, runningvector, finalOutput);

    // returning
    return;
}

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root       {new TreeNode(1)};
    root->left      = new TreeNode(2);
    root->right     = new TreeNode(3);
    auto targetSum {5};

    // trivial case
    if (root == nullptr)  {
        cout << format("final-output = {}\n", vector<vector<int>>());
        return 0;
    }

    // setup
    vector<vector<int>> finalOutput;
    auto runningsum {0};
```

```cpp
    vector<int> runningvector;

    // calling the function
    foo(root, targetSum, runningsum, runningvector, finalOutput);

    // returning
    cout << format("final-output = {}\n", finalOutput);

    // return
    return(0);

}
```

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

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root         {new TreeNode(1)};
    root->left         = new TreeNode(2);
    root->right        = new TreeNode(5);
    root->left->left   = new TreeNode(3);
    root->left->right  = new TreeNode(4);
    root->right->right = new TreeNode(6);

    // recursion lambda
    auto foo = [&foo](TreeNode*         root,
                      vector<TreeNode*>&  nodeaddresses){

        // sending it back
        if (root == nullptr) return;

        // adding address
        nodeaddresses.push_back(root);

        // going down left
        if (root->left) foo(root->left, nodeaddresses);
```

```
        if (root->right) foo(root->right, nodeaddresses);

        // returning
        return;
    };

    // trivial case
    if (root == nullptr) return 0;

    // setup
    vector<TreeNode*> nodeaddresses;

    // fill registers
    foo(root, nodeaddresses);

    // going through the addresses and reconnecting it
    for(int i = 0; i<nodeaddresses.size()-1; ++i){
        nodeaddresses[i]->left = nullptr;
        nodeaddresses[i]->right = nodeaddresses[i+1];
    }

    // taking care of the last
    nodeaddresses[nodeaddresses.size()-1]->left = nullptr;
    nodeaddresses[nodeaddresses.size()-1]->right = nullptr;

    // return
    return(0);

}
```

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:

```
struct Node {
    int val;
    Node *left;
    Node *right;
    Node *next;
}
```

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, 212 - 1].
- -1000 $\leq$ Node.val $\leq$ 1000

Code

```
// Definition for a Node.
class Node {
public:
    int val;
    Node* left;
```

```
    Node* right;
    Node* next;

    Node() : val(0), left(NULL), right(NULL), next(NULL) {}

    Node(int _val) : val(_val), left(NULL), right(NULL), next(NULL) {}

    Node(int _val, Node* _left, Node* _right, Node* _next)
        : val(_val), left(_left), right(_right), next(_next) {}
};

void foo(deque<Node*> parents)
{
    // sendint it back if empty
    if (parents.size() < 1)  {return;}

    // adding a nullptr to the parents list as demarcater
    parents.push_back(static_cast<Node*>(nullptr));

    // adding children to parents list until we hit the nullptr
    while(parents[1] != static_cast<Node*>(nullptr)){

        // connecting the top
        parents[0]->next = parents[1];

        // adding the children to the pipeline
        if(parents[0]->left)  {parents.push_back(parents[0]->left);}
        if(parents[0]->right) {parents.push_back(parents[0]->right);}

        // popping the top
        parents.pop_front();
    }

    // adding the children to the pipeline
    if(parents[0]->left)  {parents.push_back(parents[0]->left);}
```

```
    if(parents[0]->right) {parents.push_back(parents[0]->right);}

    // popping the top two parents
    parents.pop_front();
    parents.pop_front();

    // calling the function again
    foo(parents);

    // returning
    return;
}

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root                {new Node(1)};
    root->left               = new Node(2);
    root->right              = new Node(3);
    root->left->left         = new Node(4);
    root->left->right        = new Node(5);
    root->right->left        = new Node(6);
    root->right->right       = new Node(7);

    // trivial case
    if (root == nullptr)  {return 0;}

    // setup
    deque<Node*> parents {root};

    // calling the function
    foo(parents);
```

```

    // return
    return(0);

}
```

117. Populating Next Right Pointers in Each Node II

Given a binary tree of type

```
struct Node {
    int val;
    Node *left;
    Node *right;
    Node *next;
}
```

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

```cpp
class Node {
public:
    int val;
    Node* left;
    Node* right;
    Node* next;

    Node() : val(0), left(NULL), right(NULL), next(NULL) {}
    Node(int _val) : val(_val), left(NULL), right(NULL), next(NULL) {}
    Node(int _val, Node* _left, Node* _right, Node* _next)
        : val(_val), left(_left), right(_right), next(_next) {}
};


int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root          {new Node(1)};
    root->left         = new Node(2);
    root->right        = new Node(3);
    root->left->left   = new Node(4);
    root->left->right = new Node(5);
```

```
    root->right->right = new Node(7);

    // running
    if (root == nullptr) {cout << format("done!\n"); return 0;}

    // calling the function
    int count = -1;
    vector< vector<Node*> > addressvectors;

    // calling th efunction
    foo(root, count, addressvectors);

    // building the connections
    for(auto x: addressvectors){
        int i = 0;
        for(; i<x.size()-1; ++i)
            if (x[i] != nullptr)  {x[i]->next = x[i+1];}                  // look to the right

        // ensuring that the last pointer points to thr right most
        x[i]->next = nullptr;
    }

    // completion
    cout << format("done!\n");

    // return
    return(0);

}
```

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

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

```
    // sending it back if we overshot
    ++currlevel;
    if (currlevel > terminatinglevel) {return;}

    // creating the final output
    vector<int> tempoutput (currlevel, 0);
    tempoutput[0]                 = 1;
    tempoutput[tempoutput.size()-1] = 1;

    // fetching the layer above
    auto& layerabove = finalOutput[finalOutput.size()-1];

    // producing the output for each level
    for(int i = 1; i<currlevel-1; ++i) {tempoutput[i] = layerabove[i-1] + layerabove[i];}

    // pushing to the final output
    finalOutput.push_back(tempoutput);

    // recursion
    foo(finalOutput, currlevel, terminatinglevel);

    // returning
    return;

}

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto numRows   {5};

    // setup
```

```
    vector<vector<int>> finalOutput;
    finalOutput.push_back(vector<int>({1}));
    int currlevel {1};

    // calling the function
    foo(finalOutput, currlevel, numRows);

    // returning the final output
    cout << format("finalOutput = \n");
    fPrintMatrix(finalOutput);

    // return
    return(0);

}
```

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

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

    // sending it back if we overshot
    ++currlevel;
    if (currlevel > terminatinglevel) {return;}

    // creating the final output
    vector<int> tempoutput (currlevel, 0);
    tempoutput[0]                  = 1;
    tempoutput[tempoutput.size()-1] = 1;

    // fetching the layer above
    auto& layerabove  {finalOutput[finalOutput.size()-1]};

    // producing the output for each level
    for(int i = 1; i<currlevel-1; ++i) {tempoutput[i] = layerabove[i-1] + layerabove[i];}

    // pushing to the final output
    finalOutput.push_back(tempoutput);

    // recursion
    foo(finalOutput, currlevel, terminatinglevel);

    // returning
    return;
}

int main(){

    // starting timer
    Timer timer;
```

```
    // input- configuration
    auto rowIndex {3};

    // setup
    vector<vector<int>> finalOutput;
    finalOutput.push_back(vector<int>({1}));
    int currlevel {1};

    // calling the function
    foo(finalOutput, currlevel, rowIndex+1);

    // returnign the final output
    cout << format("final-output = {}\n", finalOutput[finalOutput.size()-1]);

    // return
    return(0);

}
```

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

- $1 \leq$ triangle.length ≤ 200
- triangle[0].length == 1
- triangle[i].length == triangle[i - 1].length + 1

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

Code

```cpp
int main(){

    // starting timer
    Timer timer;

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

    // setup - input
    vector<int> row = triangle[triangle.size()-1];

    // filling up the values
    for(int i = triangle.size()-2; i>=0; --i){

        // going through the rows
        for(int j = 0; j<=i; ++j){

            auto currentvalue {triangle[i][j]};                       // fetching current value
            auto smallerchild {std::min(row[j], row[j+1])};           // finding the smaller of the two children
            row[j] = currentvalue + smallerchild;                     // adding to the parent and storing to the new array

        }
    }

```

```
    // printign the final output
    cout << format("final-output = {}\n", row[0]);

    // return
    return(0);

}
```

121. Best Time To Buy And Sell Stock

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

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

```
int main(){

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

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

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

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

    // return
    return(0);

}
```

122. Best Time To Buy And Sell Stock II

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

Examples

1. **Example 1**
   - Input: prices = [7,1,5,3,6,4]
   - Output: 7
   - Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4. Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3. Total profit is 4 + 3 = 7.
2. **Example 2**
   - Input: prices = [1,2,3,4,5]
   - Output: 4
   - Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4. Total profit is 4.
3. **Example 3**
   - Input: prices = [7,6,4,3,1]
   - Output: 0
   - Explanation: There is no way to make a positive profit, so we never buy the stock to achieve the maximum profit of 0.

Constraints

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

Code

```cpp
int main(){

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

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

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

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

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

```
        ++p2;                                           // updating p2
    }

    // printing the max-value
    cout << format("accprofit = {}\n", accprofit);

    // return
    return(0);

}
```

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 $\leq$ Node.val $\leq$ 1000

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root          {new TreeNode(-10)};
    root->left          = new TreeNode(9);
    root->right         = new TreeNode(20);
    root->right->left = new TreeNode(15);
    root->right->right = new TreeNode(7);


    // setup
```

```
    auto finalresult  {std::numeric_limits<int>::min()};

    // creating lambda
    std::function<int(const TreeNode*)> fFindMaxPathSum = [&fFindMaxPathSum,
                                                           &finalresult](
        const TreeNode* root){

        // checking if leaf-node
        if(root->left  == nullptr && root->right == nullptr){

            // checking the final-result with the leaf-node value
            finalresult = finalresult > root->val ? finalresult : root->val;

            // the potential from leaf-node
            return root->val;
        }
        else{

            // creating potentials-vector
            vector<int> potentials;
            if (root->left)                         {potentials.emplace_back(fFindMaxPathSum(root->left));}
            if (root->right)                        {potentials.emplace_back(fFindMaxPathSum(root->right));}

            // calculating sum of left and right
            auto leftplusright {0};
            for(int i = 0; i<potentials.size(); ++i) {leftplusright += potentials[i];}

            // calculating sum of path curr with either paths
            for(int i = 0; i<potentials.size(); ++i) {potentials[i] += root->val;}

            // adding curr-value alone as a candidate
            potentials.push_back(root->val);

            // sending the maximum-value back
            const auto maxelement = *(std::max_element(potentials.begin(), potentials.end()));
```

```cpp

            // path from left-curr-right
            potentials.push_back(leftplusright + root->val);

            // auto temp = *(std::max_element(potentials.begin(), potentials.end()));
            const auto comparitivemax = *(std::max_element(potentials.begin(), potentials.end()));

            // finalresult = std::max(finalresult, maxelement);
            finalresult = finalresult > comparitivemax ? finalresult : comparitivemax;

            // returning the max-potential from here
            return maxelement;
        }
    };


    // calculating the final-result
    fFindMaxPathSum(root);

    // printing
    cout << format("final-result = {}\n", finalresult);

    // return
    return(0);

}
```

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

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

Code

```
int main(){

    // input- configuration
    auto s = string("A man, a plan, a canal: Panama");

    // setup
    auto pleft        {0};
    auto pright       {static_cast<int>(s.size())-1};
    auto finaloutput  {true};

    // lambda to check if alphanumeric
    auto isalphanumeric = [](const int& x){
        if (x-65 >= 0 && 90-x >= 0)    {return true;}    // upper-case check
        if (x-97 >= 0 && 122-x >= 0)   {return true;}    // lower-case check
        if (x-48 >= 0 && 57-x >= 0)    {return true;}    // numeric check
        return false;
    };

    // running
    while(pleft < s.size() && pright >= 0){

        // moving pleft until we find the position
        while(pleft < s.size() && isalphanumeric(s[pleft]) == false) {++pleft;}
        while(pright >= 0     && isalphanumeric(s[pright]) == false) {--pright;}

```

```
        // checking bounds
        if (pleft>=pright) {break;}

        // checking if they're the same
        if (std::tolower(s[pleft]) != std::tolower(s[pright])) {finaloutput = false; break;}

        // updating pointers
        ++pleft; --pright;
    }

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

    // return
    return(0);

}
```

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

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

Code

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

    // starting timer
    Timer timer;

    // input- configuration
    string beginWord         {"hit"};
    string endWord           {"cog"};
    vector<string> wordList  {
        "hot","dot","dog","lot","log"
    };

    // returning error if endword is not in wordhlist
    if (std::find(wordList.begin(), wordList.end(), endWord) == wordList.end()) {
```

```
        cout << format("final-output = 0\n");
        return 0;
    }

    // setup
    unordered_map<string, std::set<string> > neighbours;

    auto calculatedistances = [](const string a, const string b){
        auto numdiffs  {0};
        for(int i = 0; i<a.size(); ++i) {if (a[i] != b[i]) {++numdiffs;}}
        if (numdiffs == 0)           {return 0;}
        else if (numdiffs == 1)      {return 1;}
        else                         {return std::numeric_limits<int>::max();}
    };

    // starting again
    vector<string> candidates {beginWord};
    for(const auto x: wordList) {if (x!=endWord) candidates.push_back(x);}
    vector<string> nextgencandidates {};

    vector<string> recruiters        {{endWord}};
    vector<string> nextgenrecruiters {};

    // going through the loop
    auto count  {1};
    auto runningcondition {true};
    auto foundpath        {false};

    while(runningcondition){
        // increasing count
        ++count;

        // comparing distance between candidates and recruiters
        for(const auto candidate: candidates){
```

```
            auto shortestpath {std::numeric_limits<int>::max()};
            for(const auto& recruiter: recruiters)
                shortestpath  = std::min(shortestpath, calculatedistances(candidate, recruiter));

            // adding to next-generation of recruiters if diff == 1
            if (shortestpath == 1){
                nextgenrecruiters.push_back(candidate);     // adding to next gen
                if (candidate == beginWord) {foundpath = true; runningcondition = false;}
            }
            else if (shortestpath == std::numeric_limits<int>::max()){
                nextgencandidates.push_back(candidate);
            }
        }

        // rewriting history
        int prevnumcandidates = candidates.size();
        int nexnumcandidates  = nextgencandidates.size();
        candidates = nextgencandidates; nextgencandidates.clear();
        recruiters = nextgenrecruiters; nextgenrecruiters.clear();

        // evaluating running condition
        if (prevnumcandidates == nexnumcandidates) {runningcondition = false;}

    }

    // setting up final output
    if (!foundpath) {count = 0;}

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


    // return
    return(0);

```

```
}
```

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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto nums = vector<int>({0,3,7,2,5,8,4,6,0,1});

    // setup
    std::multiset<int> var00;

    // trivial-cases
    if (nums.size() <= 1) {cout << format("final-output = {}\n", nums.size()); return 0;}

    // going through elements
    for(auto x: nums) {var00.insert(x);}

    // going through elements
    int maxlength = std::numeric_limits<int>::min();
    int temp = 1;
    std::deque<int> var01;

    for(auto x: var00){

        if (var01.size()<1)          {var01.push_back(x);}
        else{
            // comparing previous element and current
            if (x - var01[0] == 1)   {++temp; var01[0] = x; maxlength = max(maxlength, temp);}
            else if(x - var01[0] == 0) {maxlength = max(maxlength, temp); continue;}
            else                     {maxlength = max(maxlength, temp); temp = 1; var01[0] = x;}
        }
    }

```

```
    // returning the max-length
    cout << format("final-output = {}\n", maxlength);

    // return
    return(0);

}
```

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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root         {new TreeNode(4)};
    root->left         = new TreeNode(9);
    root->right        = new TreeNode(0);
    root->left->left   = new TreeNode(5);
    root->left->right  = new TreeNode(1);

    // trivial case
```

```
    if (root == nullptr) return 0;

    // setup
    auto runningsum {0};
    auto finalSum  {0};
    std::function<void(const TreeNode*, int)> foo = [&foo,
                                     &finalSum](
        const TreeNode*   root,
        int               runningsum){

        // sending it back
        if (root == nullptr) return;

        // adding current value to running sum
        runningsum = runningsum*10 + root->val;

        // going down left or right
        if (root->left || root->right){
            if (root->left)   foo(root->left,   runningsum);
            if (root->right)  foo(root->right,  runningsum);
        }
        else{
            // adding to final sum
            finalSum += runningsum;
        }

        // return re
        return;
    };

    // calling function
    foo(root, runningsum);

    // retunring finalSum
    cout << format("final-output = {}\n", finalSum);
```

```
    // return
    return(0);

}
```

130. Surrounded Regions

You are given an m x 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 ≤ 200
- board[i][j] is 'X' or 'O'.

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto board {std::vector<std::vector<char>>{
        {'X','X','X','X'},
        {'X','O','O','X'},
        {'X','X','O','X'},
        {'X','O','X','X'}
    }};

    // setup
    std::deque<vector<int>> pipe;

    // going from left to right
    for(int col = 0; col < board[0].size(); ++col)
        if(board[0][col] == 'O') {pipe.push_back({0, col});}
```

```
    // going from top to bottom
    for(int row = 0; row < board.size(); ++row)
        if(board[row][board[0].size()-1] == 'O')
            pipe.push_back({row, static_cast<int>(board[0].size())-1});

    // going from right to left
    for(int col = board[0].size()-1; col >= 0 && board.size() > 1; --col)
        if(board[board.size()-1][col] == 'O')
            pipe.push_back({static_cast<int>(board.size())-1, col});

    // going from top to bottom
    for(int row = board.size()-1; row >= 0; --row)
        if(board[row][0] == 'O') {pipe.push_back({row, 0});}

    // writing the visited places
    vector<vector<int>> registerVector;

    //
    while(pipe.size() != 0){

        // fetching the front
        auto    front_value    {pipe.front()}; pipe.pop_front();
        const   auto&   row    {front_value[0]};
        const   auto&   col    {front_value[1]};

        // checking bounds
        if (row < 0 || row >= board.size() || col < 0 || col >= board[0].size()) {continue;}

        // checking if the token is already in register
        if (std::find(registerVector.begin(),
                      registerVector.end(),
                      front_value) != registerVector.end()) {continue;}

        // checking if current-point is an X
```

```cpp
        if(board[row][col] == 'X')                          {continue;}

        // adding to register
        registerVector.push_back(front_value);

        // adding the neighbours
        pipe.push_back({row, col+1});
        pipe.push_back({row-1, col});
        pipe.push_back({row, col-1});
        pipe.push_back({row+1, col});

    }

    // creating a board of full ones
    for(auto row = 0; row < board.size(); ++row)
        std::for_each(board[row].begin(),
                      board[row].end(),
                      [](auto& argx){argx= 'X';});

    // filling it with zeros
    for(const auto& argx: registerVector)
        board[argx[0]][argx[1]]  = 'O';

    // printing the matrix
    fPrintMatrix(board);

    // return
    return(0);

}
```

134. Gas Station

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

Examples

1. **Example 1:**
    - Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
    - Output: 3
2. **Example 2:**
    - Input: gas = [2,3,4], cost = [3,4,3]
    - Output: -1

Constraints:

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

Code

```cpp
int main(){

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

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

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

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


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

    // return
    return(0);
```

```
}
```

135. Candy

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

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

Examples

- **Example 1**
  - Input: ratings = [1,0,2]
  - Output: 5
  - Explanation: You can allocate to the first, second and third child with 2, 1, 2 candies respectively.
- **Example 2**
  - Input: ratings = [1,2,2]
  - Output: 4
  - Explanation: You can allocate to the first, second and third child with 1, 2, 1 candies respectively. The third child gets 1 candy because it satisfies the above two conditions.

Constraints

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

Code

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

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

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

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

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

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

        // updating based on left
```

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

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

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

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

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

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

    // return
    return(0);

}
```

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

```
int main(){

    // starting timer
    Timer timer;

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

    // going through the number
    auto finalOutput  {nums[0]};
    for(int i = 1; i<nums.size(); ++i) {finalOutput = finalOutput^nums[i];}

    // returning output
    cout << format("final-output = {}\n", finalOutput);

    // return
    return(0);

}
```

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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto nums  {vector<int>{0,1,99,0,1,0,1}};

    // setup
    std::unordered_map<int, int> histogram;
    auto finaloutput  {-1};

    // going through the nums
    for(const auto& x: nums){
        if (histogram.find(x) == histogram.end())  {histogram[x] = 1;}
        else                                       {++histogram[x];}
    }

    // going through the histogram again to see who has count 1
    for(const auto& [k,v]: histogram){
        if (v == 1) {finaloutput = k;}
    }

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

    // return
    return(0);

}
```

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

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    ListNode* head               = new ListNode(3);
    head->next                   = new ListNode(2);
    head->next->next             = new ListNode(0);
    head->next->next->next       = new ListNode(-4);
    head->next->next->next->next = head->next;

    // trivial case
    if (head == nullptr) {cout << format("final-output = false\n"); return 0;}

    // setup
    ListNode* dummy   = new ListNode(0);
    dummy->next       = head;
    ListNode* fastboi = dummy;
    ListNode* slowboi = dummy;

    while (fastboi != nullptr && fastboi->next != nullptr) {

```

```
        // updating positions
        fastboi = fastboi->next->next;
        slowboi = slowboi->next;

        // checking if they're the same
        if (slowboi == fastboi) {cout << format("final-output = true\n"); return 0;}
    }

    // since exiting the list implies you left
    cout << format("final-output = false\n"); return 0;

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

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 $\leq$ Node.val $\leq$ 100

Code

```
void foo(TreeNode* root, vector<int>& finaloutput){
    // sending it back
    if (root == nullptr) return;

    // adding current-value to the final output
    finaloutput.push_back(root->val);

    // going left
    foo(root->left, finaloutput);
    foo(root->right, finaloutput);

    // sending it back
    return;
}

int main(){

    // starting timer
    Timer timer;
```

```

    // input- configuration
    auto root          {new TreeNode(1)};
    root->right        = new TreeNode(2);
    root->right->left = new TreeNode(3);

    // setup
    vector<int> finaloutput;

    // calling the function
    foo(root, finaloutput);

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

    // return
    return(0);

}
```

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

```
void foo(TreeNode* root, vector<int>& finaloutput){
    // sending it back
    if (root == nullptr) return;

    // going left and right
    foo(root->left,   finaloutput);
    foo(root->right,  finaloutput);

    // adding current
    finaloutput.push_back(root->val);

    // returning
    return;
}
```

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root         {new TreeNode(1)};
    root->right       = new TreeNode(2);
    root->right->left = new TreeNode(3);

    // setup
    vector<int> finaloutput;

    // calling the function
    foo(root, finaloutput);

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

    // return
    return(0);

}
```

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 $\leq$ Node.val $\leq$ 105

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto head             {new ListNode(4)};
    head->next            = new ListNode(2);
    head->next->next      = new ListNode(1);
    head->next->next->next = new ListNode(3);

    // setup
    if (head == nullptr)  {cout << format("empty-list\n"); return 0;}

    // setup
    ListNode* traveller = nullptr;
    vector<pair<int, ListNode*>> nodeaddresses;

    // goign through the list
```

```cpp
    traveller = head;
    while(traveller){

        // adding it ot the list
        nodeaddresses.push_back(std::pair<int, ListNode*>({traveller->val, traveller}));

        // moving on to the next one
        traveller = traveller->next;
    }

    // sorting the whole thing
    std::sort(nodeaddresses.begin(),
              nodeaddresses.end(),
              [](pair<int, ListNode*> a,
                 pair<int, ListNode*> b) {return a < b;});

    // reconnecting
    for(int i = 0; i<nodeaddresses.size()-1; ++i)
        nodeaddresses[i].second->next = nodeaddresses[i+1].second;


    // making the last one connect to nullptr
    nodeaddresses[nodeaddresses.size()-1].second->next = nullptr;

    // returning top of the list
    fPrintLinkedList("finalOutput = ", nodeaddresses[0].second);

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

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

Code

```cpp
int main(){

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

    // setup
    vector<string> listofwords;

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

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

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

```
    }

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

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

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


    // return
    return(0);

}
```

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

Code

```
int main(){

    // starting timer
    Timer timer;

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

    // setup
    int left                     = 0;
```

```cpp
    int right                    = nums.size()-1;
    int mid;

    // looping through
    while(left < right){

        // calculating midpoint
        mid = (left + right)/2;

        // moving pointers
        if(nums[mid] > nums[right])  {left = mid + 1;}
        else                         {right = mid;}
    }

    // returning the mid-value
    cout << format("final-output = {}\n", nums[left]);

    // return
    return(0);

}
```

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

```cpp
int main(){

    // starting timer
    Timer timer;

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

    // setup
    int left   {0};
    int right  {static_cast<int>(nums.size())-1};
    int mid    {-1};

    // running the lopo
    while(left <= right){

        // fetching middle index
        mid = (left + right)/2;

        // evaluating
        if (mid > 0 && nums[mid] < nums[mid-1])                 {right = mid-1;}
        else if(mid < nums.size()-1 && nums[mid] < nums[mid+1])  {left = mid +1;}
        else                                                     {break;}
```

```
    }

    // returning mid
    cout << format("final-output = {}\n", mid);

    // return
    return(0);

}
```

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 $1 \leq \text{index1} < \text{index2} \leq$ 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 $\leq$ numbers[i] $\leq$ 1000
- numbers is sorted in non-decreasing order.
- -1000 $\leq$ target $\leq$ 1000
- The tests are generated such that there is exactly one solution.

Code

```
int main(){

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

    // setup
    std::vector<int> finalOutput;
    auto left     {0};
    auto right    {static_cast<int>(numbers.size()-1)};
    auto currsum  {0};

    // usual left-right loop
    while (left < right){
```

```cpp

        // checking sum of two values
        currsum = numbers[left] + numbers[right];

        // comparing against target
        if (currsum > target)    {--right;}
        else if (currsum < target) {++left;}
        else {
            finalOutput.push_back( left+1);
            finalOutput.push_back( right+1);
            break;
        }
    }

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

    // return
    return(0);

}
```

169 Majority Element

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

Examples

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

Constraints:

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

Code

```cpp
int main(){

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

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

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

        // adding to histogram
        if (histogram.find(nums[i]) == histogram.end()) {histogram[nums[i]] = 1; updated_count = 0;}
        else                                          {++histogram[nums[i]]; updated_count = histogram[nums[i]];}

        // keeping track of max-element
        if (updated_count > max_count) {max_element = nums[i]; max_count = updated_count;}

    }

    // printing the final output
    cout << format("nums = "); fpv(nums);
    cout << format("max-count = {}\n", max_count);

    // return
    return(0);

}
```

172. Factorial Trailing Zeroes

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

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

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

- $0 \le \text{n} \le 10^4$

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto n {5};

    // setting up lambdas
    auto counttwos    = [](int x){

        // in case of x == 0 or x being odd
        if(x%2 != 0 || x == 0) {return 0;}

        // counting number of 2s we can pluck from this number
        auto numtwos {0};
        while(true){
            if (x%2 == 0)  {++numtwos; x /= 2; continue;}
            else           {return numtwos;}
        }

        // returning
        return numtwos;
    };

    auto countfives   = [](int x){

        // in case of x == 0 or x not being a multiple of five
        if(x%5 != 0 || x == 0)       {return 0;}

        // counting number of 5s we can pluck from this number
        auto numfives {0};
        while(true){
```

```
            if (x%5 == 0)  {++numfives; x /= 5; continue;}
            else           {return numfives;}
        }

        // returning numfives
        return numfives;
    };

    // calculating numtwos and numfives
    auto numtwos   {0};
    auto numfives  {0};

    for(int i = n; i>=0; --i){

        numtwos    += counttwos(i);
        numfives   += countfives(i);

    }

    // calculating numzeros
    auto finaloutput  {std::min(numtwos, numfives)};
    cout << format("final-output = {}\n", finaloutput);

    // return
    return(0);

}
```

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

```
int main(){

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

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

    int source        {0};
    int temp_source   {nums[source]};
    int temp          {0};
    int destination   {0};

    vector<bool> sourcelist(nums.size(), false);

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

        // check if curent-source has been taken care of
        if (sourcelist[source] == true){
            source     = (source+1) % nums.size();
            temp_source = nums[source];
        }

        source             = source % nums.size();      // code to ensure range
        destination        = (source + k)%nums.size();  // calculating the index we'll be writing to
        sourcelist[source] = true;                      // updating source-list

        temp               = nums[destination];         // safe-keeping the destination value
        nums[destination] = temp_source;                // storing new value at destination-index

```

```
        source              = destination;                  // updating source-index
        temp_source         = temp;                         // updating source-value
    }

    // printing the output
    cout << format("nums = "); fpv(nums);                   // printing the updated array, "nums"
    timer.stop();                                           // printing the time taken

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

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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    uint32_t n = 43261596;

    // reversing
    uint32_t n_reverse = 0b0;

    // reversing loop
    for (int i = 0; i<32; ++i) {
        // making space for reverse number and adding current
        n_reverse = n_reverse<<1 | (n & 1);

        // cutting off tail.
        n = n>>1;
    }

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

    // return
    return(0);
```

```
}
```

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 1111111111111111111111111111101 has a total of thirty set bits.

Constraints

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

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto n {11};

    // setup
    int count = 0;;
    while (n) {
        if ((n & 1)==1) ++count;
        n = n >> 1;
    }

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

    // return
    return(0);

}
```

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

```cpp
int main(){

    // starting timer
    Timer timer;

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

    // setup
    std::vector<int> dp{nums[0]};
    int maxCandidate;

    // building the dp-table
    for(int i = 1; i<nums.size(); ++i){

        // checking which max value to use
        if (i > 1) {maxCandidate = std::max(nums[i]+ dp[i-2], dp[i-1]);}
        else       {maxCandidate = std::max(nums[i], dp[i-1]);}

        // storing max-candidate
        dp.push_back(maxCandidate);
    }

    // returning
    cout << format("final-output = {}\n", dp[dp.size()-1]);
```

```
    // return
    return(0);

}
```

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

```
void fTraverse(TreeNode*         root,
               std::vector<int>&  finalView,
               int                count){

    // base-case
    if (root == nullptr)                    {return;}

    // increasing size of finalView incase it is not big enough
    while (finalView.size() < count+1)      {finalView.push_back(-INT_MAX);}

    // assigning values
    finalView[count] = root->val;

    // writing just the right value
    if (root->left != nullptr)              {fTraverse(root->left, finalView, count +1);}

    // going down this branch
```

```
    if (root->right != nullptr)             {fTraverse(root->right, finalView, count+1);}

    // returning the value
    return;
}
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root         {new TreeNode(1)};
    root->left         = new TreeNode(2);
    root->right        = new TreeNode(3);
    root->left->right = new TreeNode(5);
    root->right->right = new TreeNode(4);

    // checking something simple
    if (root == nullptr)                    {cout << format("final-output = {}\n", std::vector<int>()); return 0;}

    // method to print this out
    std::vector<int> finalView;
    auto count {0};

    // calling the function that traverses through these things.
    finalView.push_back(root->val);
    fTraverse(root, finalView, 0);

    // return
    cout << format("final-output = {}\n", finalView);

    // return
    return(0);

}
```

200. Number of Islands

Given an m x 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 == grid.length

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

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    vector<vector<string>> grid {
        vector<string>({"1","1","1","1","0"}),
        vector<string>({"1","1","0","1","0"}),
        vector<string>({"1","1","0","0","0"}),
        vector<string>({"0","0","0","0","0"})
    };

    // setup
    std::vector< std::vector<bool> > didWeVisitThisPlace;
    for(int i = 0; i<grid.size(); ++i)
        didWeVisitThisPlace.push_back(std::vector<bool>(grid[0].size(), false));

    // lambda to check validity
    auto fCheckValidityOfCoordinate = [&grid](
        std::vector<int> coords){

        // spreading it out
        auto row  {coords[0]};
        auto col  {coords[1]};
```

```
        // checking validity
        if (row >= 0          &&
            row < grid.size() &&
            col >= 0          &&
            col <grid[0].size())
            if (grid[row][col] == "1") {return true;}

        // in case the above condition is not met
        return false;

    };

    // traversal function
    std::function<void(int, int)> fMarkThemAll = [&fMarkThemAll,
                                                  &grid,
                                                  &didWeVisitThisPlace,
                                                  fCheckValidityOfCoordinate](
        int row_index,
        int col_index){

        // setting up coordinates
        std::vector<int> rightCoordinate({row_index, col_index+1});
        std::vector<int> downCoordinate{row_index+1, col_index};

        // marking the current coordinate as visited
        didWeVisitThisPlace[row_index][col_index] = true;

        // calling the function to the right
        if (fCheckValidityOfCoordinate(rightCoordinate) == true &&\
            didWeVisitThisPlace[rightCoordinate[0]][rightCoordinate[1]] == false)
                fMarkThemAll(rightCoordinate[0], rightCoordinate[1]);

        // calling the function for the ones below
        if (fCheckValidityOfCoordinate(downCoordinate) == true && \
```

```
            didWeVisitThisPlace[downCoordinate[0]][downCoordinate[1]] == false)
                fMarkThemAll(downCoordinate[0], downCoordinate[1]);
    };


    // going through the elements
    int count = 0;
    for(int row_index = 0; row_index < grid.size(); ++row_index){
        for(int col_index = 0; col_index < grid[0].size(); ++col_index){
            // starting an exploratory course if this point has not been visited
            if (didWeVisitThisPlace[row_index][col_index] == false && \
                grid[row_index][col_index]            == "1"){
                fMarkThemAll(row_index, col_index);
                ++count;
            }
        }
    }

    // return count
    cout << format("final-output = {}\n", count);

    // return
    return(0);

}
```

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:
        - 12 + 92 = 82
        - 82 + 22 = 68
        - 62 + 82 = 100
        - 12 + 02 + 02 = 1
2. **Example 2:**
    - Input: n = 2
    - Output: false

Constraints

- $1 \leq \mathrm{n} \leq 2^{3}1 - 1$

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto n {19};

    // trivial case
    if (n == 4 || n == 0) {cout << format("final-output = false\n"); return 0;}

    // setting up lambda
    auto fSumDigitSquares = [](int n) -> int
    {
        auto sum   {0};
        auto digit {-1};
        while(n!=0){
            digit  = n%10;
            sum    += digit*digit;
            n      = n/10;
        }
        return sum;
    };

    // calling the function
    while (n != 1){
        // calculating digits sums
```

```
        n = fSumDigitSquares(n); cout << format("n = {}\n", n);
        if (n == 4 || n == 0)    {cout << format("final-output = false\n"); return 0;}
    }

    // printing and returning
    cout << format("final-output = true \n"); return 0;

}
```

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

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    string s = {"egg"};
    string t = {"add"};

    // making mapping
    unordered_map<char, char> mapping;
    std::unordered_set<char> destinations;
    string                   r;

    // going through string
    for(int i = 0; i<s.size(); ++i){

        // checking if two strings are different
        if (s[i] != t[i]){
            // checking if we already have a mapping for this
            if (mapping.find(s[i]) == mapping.end()){

                // before we addd, we need to check if t[i] is already used as a destination
```

```
                if (destinations.find(t[i]) != destinations.end()) {
                    cout << format("final-output = true \n");
                    return 0;
                }

                // adding new mapping
                mapping[s[i]] = t[i];
                destinations.insert(t[i]);
            }
            r.push_back(mapping[s[i]]);
        }
        else{
            // checking if we already have a mapping for this
            if (mapping.find(s[i]) == mapping.end()){

                // before we addd, we need to check if t[i] is already used as a destination
                if (destinations.find(t[i]) != destinations.end()) {
                    cout << format("final-output = true \n");
                    return 0;
                }

                // adding new mapping
                mapping[s[i]] = s[i];
                destinations.insert(s[i]);
            }
            r.push_back(mapping[s[i]]);
        }

        // if what we have so far is different, we're sending it back
        string s_subset(t.begin(), t.begin() + r.size());
        if(s_subset != r) {cout << format("final-output = false \n"); return 0;}

    }

    // return true in the end
```

```
    cout << format("final-output = true \n");

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

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

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto head                    {new ListNode(1)};
    head->next                   = new ListNode(2);
    head->next->next             = new ListNode(3);
    head->next->next->next       = new ListNode(4);
    head->next->next->next->next = new ListNode(5);

    // outlier cases
    if (head == nullptr)  {fPrintLinkedList("final-output = ", nullptr);}

    // setup
    auto previous_node {static_cast<ListNode*>(nullptr)};
    auto original_next {static_cast<ListNode*>(nullptr)};
    auto traveller    {head};

    // going through the list
    traveller = head;
    while(traveller!= nullptr){

        original_next  = traveller->next;       // storing original next
        traveller->next = previous_node;        // making the current point to previous
        previous_node  = traveller;             // updating for next
        traveller     = original_next;          // updating the node
    }

    // changing head position
    head = previous_node;

```

```
    // returning head
    fPrintLinkedList("final-output = ", head);

    // return
    return(0);

}
```

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] = $[a_i, b_i]$ indicates that you must take course bi first if you want to take course a_i.

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

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

Code

```cpp
int main(){

    // starting timer
    Timer timer;

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

    // setup
    auto   isStartingNode    {std::vector<bool>(numCourses, true)};
    auto   finaloutput       {false};

    // building dependency tree
    unordered_map<int, std::vector<int> > dmap;
```

```
    for(const  auto&   x: prerequisites){
        // creating source-to-destination connection
        if (dmap.find(x[1]) == dmap.end()) {dmap[x[1]] = std::vector<int>({x[0]});}
        else                               {dmap[x[1]].push_back(x[0]);}

        // updating
        isStartingNode[x[0]]  =   false;
    }

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

    // launching seach from all non-prerequisite courses
    auto   visited    {std::vector<bool>(numCourses, false)};
    auto   pipe       {std::deque<int>()};

    // lambda to check if all are true
    auto   fCheckIfAllAreTrue = [&visited](){
        return std::all_of(visited.begin(),
                           visited.end(),
                           [](const auto& argx){return argx;});
    };

    cout<< format("fCheckIfAllAreTrue = {}\n", fCheckIfAllAreTrue());
    cout << format("isStartingNode = {}\n", isStartingNode);

    // going through each course
    for(int i = 0; i < numCourses && fCheckIfAllAreTrue() == false; ++i){
        // checking if this has prerequisites
        if (isStartingNode[i] == true){

            // creating pipe
            pipe.clear();
            pipe.push_back(i);

```

```
            // running bfs
            while(pipe.size() != 0){

                // popping the front
                const  auto   front_value   {pipe.front()};
                pipe.pop_front();

                // printing
                cout << format("front-value = {} | pipe = {}\n", front_value, pipe);

                // checking if we've already marked this
                if (visited[front_value])    {continue;}

                // marking current node as visited
                visited[front_value]  =   true;

                // adding children to the pipe
                for(const  auto&   x: dmap[front_value])
                    pipe.push_back(x);

            }
        }
    }

    // returning



    // // setup
    // vector<bool> visited(numCourses, false);
    // vector<bool> hasPrerequisite(numCourses, false);
    // unordered_map<int, vector<int>> hmap;

    // // build prerequisite list
    // for(const auto x: prerequisites){
```

```
//     // pickign things out
//     auto curr  {x[0]};
//     auto prereq {x[1]};

//     // setting the flag for non
//     hasPrerequisite[curr] = true;

//     // adding to the hashmap
//     if(hmap.find(prereq) == hmap.end()) {hmap[prereq] = vector<int>{curr};}
//     else                                {hmap[prereq].push_back(curr);}

// }

// // find no-prerequisite course
// std::deque<int> indiceswithnoprerequisites;
// for(int i = 0; i<numCourses;++i)
//     if (hasPrerequisite[i]==false)
//         indiceswithnoprerequisites.push_back(i);

// // start from there and flag the
// std::deque<int> prev;
// while(indiceswithnoprerequisites.size()!=0 && prev != indiceswithnoprerequisites){

//     // storing for later comparison
//     prev = indiceswithnoprerequisites;

//     // popping the front
//     auto front {indiceswithnoprerequisites.front()};
//     indiceswithnoprerequisites.pop_front();

//     // launching search from "front"
//     if (hmap.find(front) == hmap.end()) {
//         visited[front] = true;
//         continue;
```

```
    //     }
    //     else {
    //         visited[front] = true;
    //         auto& bruh {hmap[front]};
    //         for(const auto& x: bruh) {
    //             if (visited[x] == false) {indiceswithnoprerequisites.push_back(x);}
    //             visited[x] = true;
    //         }
    //     }
    // }


    // // finalizing output
    // auto finaloutput  {true};
    // for(int i = 0; i<visited.size(); ++i) {finaloutput = finaloutput&&visited[i];}
    // cout << format("final-output = {}\n", finaloutput);

    // return
    return(0);

}
```

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

```cpp
int main(){

    // input- configuration
    vector<int> nums  {1,2,3,4,5};
    auto target    {15};

    // setup
    auto finaloutput  {std::numeric_limits<int>::max()};
    auto pleft        {0};
    auto sum          {nums[pleft]};
    auto i            {1};

    // in case the first element itself is greater
    if (sum > target) {finaloutput = 1;}

    // lambda to update finaloutput
    auto updatefinaloutput = [&sum,
                             &target,
                             &finaloutput,
                             &i,
                             &pleft,
                             &nums]() -> void{

        auto numelements = i - pleft + 1;
        finaloutput = numelements < finaloutput ? numelements : finaloutput;
        sum -= nums[pleft++];
    };
```

```
    // going through the array
    for(; i<nums.size(); ++i){

        // adding to sum
        sum += nums[i];

        // updating
        while (sum>=target) {updatefinaloutput();}
    }

    // updating
    if(finaloutput == std::numeric_limits<int>::max()) {finaloutput = 0;}

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

    // return
    return(0);

}
```

212. Word Search II

Given an m x 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 == board.length
- n == board[i].length
- $1 \leq m, n \leq 12$
- board[i][j] is a lowercase English letter.

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

Code

```
void foo(const vector<vector<char>>&             board,
        const int                                 p,
        const int                                 curr_tid,
        const string&                             targetword,
        bool&                                     found,
        vector<int>                               pathsofar,
        std::function<int(const int&, const int&)> tid,
        std::function<int(const int)>             row,
        std::function<int(const int)>             col){

    // if found is done
    if (found == true)                                                {return;}

    // returning if tid is beyond the bounds
    if (curr_tid >= (board.size() * board[0].size()) )                  {return;}

    // calculating curr-row and col
    const auto rowcurr  {row(curr_tid)}; if (rowcurr < 0 || rowcurr >= board.size()) {return;}
    const auto colcurr  {col(curr_tid)}; if (colcurr < 0 || colcurr >= board[0].size()) {return;}

    // check: duplicate tid in path
    if (std::find(pathsofar.begin(), pathsofar.end(), curr_tid) != pathsofar.end()) {return;}

```

```
    // check: target-char vs curr-char
    char curr_char            {board[rowcurr][colcurr]};
    char charweresearchingfor {targetword[p]};
    if (curr_char != charweresearchingfor)                                         {return;}

    // adding to path
    pathsofar.push_back(curr_tid);

    // checking if we've reached the end
    if (pathsofar.size() == targetword.size()) {found = true; return;}

    // moving into the neighbours
    foo(board, p+1, tid(rowcurr, colcurr+1), targetword, found, pathsofar, tid, row, col); // moving right
    foo(board, p+1, tid(rowcurr-1, colcurr), targetword, found, pathsofar, tid, row, col); // moving up
    foo(board, p+1, tid(rowcurr, colcurr-1), targetword, found, pathsofar, tid, row, col); // moving left
    foo(board, p+1, tid(rowcurr+1, colcurr), targetword, found, pathsofar, tid, row, col); // moving down

    // returning
    return;
}


int main(){

    // starting timer
    Timer timer;

    // setup
    vector<vector<char>> board {
        {'a','b','c'},
        {'a','e','d'},
        {'a','f','g'}
    };


```

```
    vector<string> words {"abcdefg","gfedcbaaa","eaabcdgfa","befa","dgc","ade"};

    // setup
    std::function<int(const int&, const int&)> tid = [&board](const int row, const int col) {
        if (row < 0 || row >= board.size() || col < 0 || col >= board[0].size()) {return std::numeric_limits<int>::max();}
        return row* static_cast<int>(board[0].size()) + col ;
    };
    std::function<int(const int)>           row = [&board](const int tid){
        if (tid < 0 || tid > board.size() * board[0].size())                {return std::numeric_limits<int>::max();}
        return tid/static_cast<int>(board[0].size());
    };
    std::function<int(const int)>           col = [&board](const int tid){
        if (tid < 0 || tid > board.size() * board[0].size())                 {return std::numeric_limits<int>::max();}
        return tid%static_cast<int>(board[0].size());
    };

    // printing
    vector<string> finaloutput;
    for(const auto& targetword: words){

        // flag for finding the word
        bool found {false};

        // launching from every coordinate
        for(int i = 0; i<board.size() && found == false; ++i){
            for(int j = 0; j<board[0].size() && found == false; ++j){
                vector<int> pathsofar;
                foo(board, 0, tid(i, j), targetword, found, pathsofar, tid, row, col);
            }
        }

        // adding to final output
        if (found) {finaloutput.push_back(targetword);}
    }

```

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

    // return
    return(0);

}
```

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

```cpp
int fRob1(std::vector<int> nums){

    // setup
    std::vector<int> dp{nums[0]};
    int maxCandidate;

    // building the dp-table
    for(int i = 1; i<nums.size(); ++i){

        // checking which max value to use
        if (i > 1)    {maxCandidate = std::max(nums[i]+ dp[i-2], dp[i-1]);}
        else          {maxCandidate = std::max(nums[i], dp[i-1]);}

        // storing max-candidate
        dp.push_back(maxCandidate);

    }

    // returning the mx-value
    return dp[dp.size()-1];
}

int main(){

    // starting timer
```

```
    Timer timer;

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

    // sending things back
    if(nums.size()== 0)   {cout << format("final-output = 0\n"); return 0;}
    if(nums.size()== 1)   {cout << format("final-output = {}\n", nums[0]); return 0;}

    // setup
    auto firstCandidate   {fRob1(std::vector<int>(nums.begin(), nums.end()-1))};
    auto secondCandidate  {fRob1(std::vector<int>(nums.begin()+1, nums.end()))};

    // returning value
    auto finaloutput      {std::max(std::max(firstCandidate, secondCandidate), nums[0])};
    cout << format("final-output = {}\n", finaloutput);

    // return
    return(0);

}
```

215. Kth Largest Element in an Array

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

Note that it is the kth largest element in the sorted order, not the kth 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

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

Code

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto nums  {vector<int>{3,2,3,1,2,4,5,5,6}};
    auto k     {4};

    // setup
    std::nth_element(nums.begin(), nums.end()-k, nums.end());
    auto finaloutput  {nums[nums.size()-k]};

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

    // return
    return(0);

}
```

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 ¿ 1, there are no valid combination.

Constraints

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

Code

```
void fTraverse(std::vector<int> numberPath, \
               int target, \
               int k, \
               std::vector< std::vector<int> >& finalOutput){

    // checking if length is too much
    if (numberPath.size() > k) return;
```

```
    // calculating sum so far
    int sumsofar = std::accumulate(numberPath.begin(), numberPath.end(),0);

    // number to fill
    int sumtofill = target - sumsofar;

    // if this is zero, we can add it to list and just sendn it back
    if (sumtofill == 0){
        if (numberPath.size() == k){
            if (std::find(finalOutput.begin(),
                          finalOutput.end(),
                          numberPath) == finalOutput.end())
                finalOutput.push_back(numberPath);                          // add if it doesn't already exist
        }

        return;
    }

    // valid candidates
    std::vector<int> candidates;
    int biggest_number = 1;
    if (numberPath.size()!=0){
        auto iter = std::max_element(numberPath.begin(), numberPath.end());
        biggest_number = *iter;
    }

    sumtofill = std::min(sumtofill, 9);
    for(int i = biggest_number; i<=sumtofill; ++i){
        // add if no on th epath so far
        if (std::find(numberPath.begin(),
                      numberPath.end(),
                      i) == numberPath.end()) {candidates.push_back(i);}
    }

```

```
    // if there are no candidates, we're going back
    if (candidates.size() == 0) return;

    // trying each candidate
    std::vector<int> numberPath_local;
    for(auto x: candidates){
        // sending down each candidate route
        numberPath_local = numberPath;
        numberPath_local.push_back(x);
        fTraverse(numberPath_local, target, k, finalOutput);
    }

    // returning
    return;
}

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto k {3};
    auto n {7};

    // setup
    std::vector< std::vector<int> > finalOutput;
    std::vector<int> numberPath;

    // recursion
    fTraverse(numberPath, n, k, finalOutput);

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

```

```
    // return
    return(0);

}
```

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

Constraints

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

Code

```cpp
int main(){

    // starting timer
    Timer timer;

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

    // setup
    unordered_map<int, int> histogram;
    auto finaloutput  {false};

    // building histogram
    for(const auto& x: nums){
        if (histogram.find(x) == histogram.end())  {histogram[x] = 1;}
        else                                        {finaloutput = true; break;}
    }

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

    // return
    return(0);

}
```

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) $\leq$ 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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto nums = std::vector<int>({1,2,3,1,2,3});
    auto k     {2};

    // setup
    std::unordered_map<int, int> valueToIndex;

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

        // adding current element to the hamp
        if (valueToIndex.find(nums[i]) == valueToIndex.end()){
            // if it doesn't exist, we add to it
            valueToIndex[nums[i]] = i;
        }
        else{
            // if it already exists, calculating distance from the first index
            if (i - valueToIndex[nums[i]] <= k) {cout << format("final-output = true\n");}
            else                                {valueToIndex[nums[i]] = i;}
        }
    }

    // returning false in the final case
    cout << format("final-output = false\n");

    // return
    return(0);

```

```
}
```

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 2h 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

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root          {new TreeNode(1)};
    root->left          = new TreeNode(2);
    root->right         = new TreeNode(3);
    root->left->left    = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->right->left = new TreeNode(6);

    // setup
    auto finalresult  {0};
    std::function<void(const TreeNode*)> fCount = [&fCount, &finalresult](
        const TreeNode* root){

        // returning
        if (root == nullptr)  {return;}
        ++finalresult;

        // going down left and right
```

```
        fCount(root->left);
        fCount(root->right);

        // returning
        return;
    };

    // calling the function
    fCount(root);

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


    // return
    return(0);

}
```

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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root                {new TreeNode(4)};
    root->left               = new TreeNode(2);
    root->right              = new TreeNode(7);
    root->left->left         = new TreeNode(1);
    root->left->right        = new TreeNode(3);
    root->right->left        = new TreeNode(6);
    root->right->right       = new TreeNode(9);

    // setup
    std::function<void(TreeNode*)> fTwist = [&fTwist](TreeNode* root){
        // base-case
        if (root == nullptr) return;

        // switching left and right
        fTwist(root->left);
        fTwist(root->right);

        // twisting current branches
        auto temp  = root->left;
        root->left = root->right;
        root->right = temp;

        // returning
        return;
    };

    // going flipping through eveyrthing
```

```
    fTwist(root);

    // return
    return(0);

}
```

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] $\Longrightarrow$ "0→2"
     - [4,5] $\Longrightarrow$ "4→5"
     - [7,7] $\Longrightarrow$ "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

- $0 \leq \text{nums.length} \leq 20$
- $-2^{3}1 \leq \text{nums[i]} \leq 2^{3}1 - 1$
- All the values of nums are unique.
- nums is sorted in ascending order.

Code

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto nums  = vector<int>({0,1,2,4,5,7});

    // trivial-case
    if (nums.size() == 0) {cout << format("final-output = {}\n", vector<string>()); return 0;}

    // stup
```

```
    auto arrow {"->"};
    auto p1    {0};
    auto p2    {0};
    vector<string> finalOutput;

    // while loop
    while (p2<nums.size()){

        // checking prev element
        if (nums[p2] <= 1 + nums[p2-1]) {++p2;}
        else{
            if (p1!=p2-1) {finalOutput.push_back(std::to_string(nums[p1]) + arrow + std::to_string(nums[p2-1]));}
            else          {finalOutput.push_back(std::to_string(nums[p2-1]));}
            p1 = p2++;
        }
    }

    // checking if p1 and p2 areright ne
    if (p1!=p2-1) {finalOutput.push_back(std::to_string(nums[p1]) + arrow + std::to_string(nums[p2-1]));}
    else          {finalOutput.push_back(std::to_string(nums[p2-1]));}

    // returning restul
    cout << format("final-output = {}\n", finalOutput);

    // return
    return(0);

}
```

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

```
void foo(const TreeNode* root,
         int&            leftindex,
         int&            finalOutput,
         bool&           stopsearch,
         const int       k)
{
    // sending it back
    if (root == nullptr)     return;

    // checking left
    foo(root->left, leftindex, finalOutput, stopsearch, k);

    // appending count
    ++leftindex;
    if (leftindex == k){

        finalOutput    = root->val;
        stopsearch     = true;

    }

    // going right
    foo(root->right, leftindex, finalOutput, stopsearch, k);

    // goign back
    return;
}

int main(){

    // starting timer
    Timer timer;

```

```cpp
    // input- configuration
    auto root          {new TreeNode(3)};
    root->left         = new TreeNode(1);
    root->right        = new TreeNode(4);
    root->left->right = new TreeNode(2);

    auto k {1};

    // setup
    auto finalOutput  {-1};
    auto stopsearch   {false};
    auto leftindex    {0};

    // running the search
    foo(root, leftindex, finalOutput, stopsearch, k);

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

    // return
    return(0);

}
```

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 Node.val are unique.
- $\text{p} \neq \text{q}$
- p and q will exist in the BST.

Code

```cpp
void foo(const TreeNode*     root,
         const int           smallervalue,
         const int           largervalue,
         TreeNode*&          finalanswer){

    if (root== nullptr)              return;
    if (finalanswer != nullptr)      return;

    // checking current value
    if (smallervalue <= root->val && root->val <= largervalue) {finalanswer = root;}
    else if (root->val < smallervalue)                          {foo(root->right, smallervalue, largervalue, finalanswer);}
    else if (root->val > largervalue)                           {foo(root->left,  smallervalue, largervalue, finalanswer);}

    // retunring
    return;
}

int main(){

```

```cpp
    // starting timer
    Timer timer;

    // input- configuration
    auto root  {new TreeNode(6)};

    root->left     = new TreeNode(2);
    root->right    = new TreeNode(8);

    root->left->left  = new TreeNode(0);
    root->left->right = new TreeNode(4);
    root->right->left = new TreeNode(7);
    root->right->right = new TreeNode(9);

    root->left->right->left  = new TreeNode(3);
    root->left->right->right = new TreeNode(5);

    auto p     {root->left};
    auto q     {root->right};

    // setup
    const auto smallervalue      {std::min(p->val, q->val)};
    const auto largervalue       {std::max(p->val, q->val)};
    auto       finalanswer       {static_cast<TreeNode*>(nullptr)};

    //calling the function
    foo(root, smallervalue, largervalue, finalanswer);

    // returning the final answer
    cout << format("final-output = {}\n", finalanswer->val);

    // return
    return(0);

}
```

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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root  {new TreeNode(3)};

    root->left = new TreeNode(5);
```

```
    root->right = new TreeNode(1);

    root->left->left = new TreeNode(6);
    root->left->right = new TreeNode(2);

    root->right->left = new TreeNode(0);
    root->right->right = new TreeNode(8);

    root->left->right->left = new TreeNode(7);
    root->left->right->right = new TreeNode(4);

    auto p {root->left};
    auto q {root->right};

    // setup
    TreeNode *finalanswer {static_cast<TreeNode*>(nullptr)};
    std::function<int(TreeNode*)> foo = [&foo,
                                        &finalanswer,
                                        &p, &q](
        TreeNode* root){

        // sending it back
        if (root== nullptr)                {return 0;}
        if (finalanswer != nullptr)        {return 0;}

        // searching left
        auto leftsum    = foo(root->left);
        auto rightsum   = foo(root->right);

        // summing up values
        auto sumvalues = leftsum + rightsum;
        if (root == p || root == q)      {sumvalues+=1;}

        // updating final answer
        if (sumvalues == 2 &&
```

```
            finalanswer == nullptr)      {finalanswer = root;}

        // returning values
        return sumvalues;

    };

    // running the function
    foo(root);

    // returning the final answer
    cout << format("Ancestor-value = {}, Ancestor-address = {}\n",
        finalanswer->val, static_cast<void*>(finalanswer));


    // return
    return(0);

}
```

238. Product of Array Except Self

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

Examples

1. **Example 1**
   - Input: nums = [1,2,3,4]
   - Output: [24,12,8,6]
2. **Example 2**
   - Input: nums = [-1,1,0,-3,3]
   - Output: [0,0,9,0,0]

Constraints

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

Code

```
int main(){

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

    // setup
    auto left  {vector<int>(nums.size(), 1)};
    auto right {vector<int>(nums.size(), 1)};

    // building the cumulative products
    std::partial_sum(nums.begin(), nums.end()-1,
                     left.begin()+1,
                     [](auto arg0,
                        auto arg1){
                            return arg0 * arg1;
                        });
    std::partial_sum(nums.rbegin(), nums.rend()-1,
                     right.rbegin()+1,
                     [](auto arg0,
                        auto arg1){
                            return arg0 * arg1;
                        });

    // producing the final-output
    auto finaloutput  {vector<int>(nums.size(),1)};
    std::transform(left.begin(), left.end(),
                   right.begin(),
                   finaloutput.begin(),
                   std::multiplies<int>());

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

```

```
    // return
    return(0);

}
```

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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto nums  {vector<int>{1,3,-1,-3,5,3,6,7}};
    auto k     {3};


    // setup
    auto   finaloutput {std::vector<int>(nums.size() - k + 1)};


    // going through the arrays
    auto   count = 0;
    for(int i = k; i <= nums.size(); ++i){

        // storing the max to the output
        finaloutput[i-k] = *(std::max_element(
            nums.begin()+i-k,
            nums.begin()+i
        ));
```

```
    }

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

    // // setup
    // auto finaloutput      {vector<int>{}};
    // auto heap             {std::vector(nums.begin(), nums.begin()+k)};
    // std::make_heap(heap.begin(), heap.end());
    // auto numbertobedeleted {-1};
    // auto numbertobeadded  {-1};
    // auto topvalue         {-1};
    // finaloutput.push_back(*(std::max_element(heap.begin(), heap.end())));

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

    //     // removing the element that just left the window
    //     numbertobedeleted = nums[i-1];
    //     auto deletion_it  = std::find(heap.begin(),
    //                                   heap.end(),
    //                                   numbertobedeleted);
    //     heap.erase(deletion_it);
    //     std::make_heap(heap.begin(),
    //                    heap.end());
```

```
    //     // adding new number to heap
    //     numbertobeadded   = nums[i+k-1];
    //     heap.push_back(numbertobeadded);
    //     std::push_heap(heap.begin(),
    //                    heap.end());

    //     // picking biggest number
    //     std::pop_heap(heap.begin(),
    //                   heap.end());
    //     topvalue          = heap.back();

    //     // adding to final-output
    //     std::push_heap(heap.begin(),
    //                    heap.end());
    //     finaloutput.push_back(topvalue);
    // }

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

    // return
    return(0);

}
```

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 = ”anagram”, t = ”nagaram”
    - Output: true
2. **Example 2:**
    - Input: s = ”rat”, t = ”car”
    - Output: false

Constraints

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

Code

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    string s   {"anagram"};
    string t   {"nagaram"};

    // setup
    std::vector<int> s_count(26,0);
    std::vector<int> t_count(26,0);

    // building count-vector for both
    for(auto x: s) ++s_count[static_cast<int>(x)-97];
    for(auto x: t) ++t_count[static_cast<int>(x)-97];

    // comparing the two
    for (int i = 0; i<s_count.size(); ++i) {

        // element-wise checking
        if (s_count[i] != t_count[i]) {cout << format("final-output = false \n");}

    }

    // returning
    cout << format("final-output = true \n");

    // return
    return(0);

}
```

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

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto nums  {vector<int>{9,6,4,2,3,5,7,0,1}};

    // setup
    auto n {nums.size()+1};
    auto finaloutput  {0};

    // going through the numbers
    for(int i = 0; i<nums.size(); ++i){
        finaloutput += i+1 - nums[i];
    }

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

    // return
```

```
    return(0);

}
```

274. H-Index

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

Examples

1. **Example 1**

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

2. **Example 2**

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

Constraints

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

Code

```
int main(){

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

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

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

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

    // return
    return(0);

}
```

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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto   n   {static_cast<int>(13)};

    // setup
    auto   length_of_square_numbers {static_cast<std::size_t>(1+std::ceil(std::sqrt(n)))};
    auto   square_numbers           {std::vector<int>(length_of_square_numbers, 0)};

    // filling up the vector
    for(auto i = 0; i < square_numbers.size(); ++i)
        square_numbers[i] =   i*i;

    // building the dptable
    auto   dptable    {std::vector<int>(n+1)};
    dptable[0] = 1;

    // building the dp-table
    for(auto i = 1; i <= n; ++i){

        // consider previous answer
        auto&  curr_value    {i};

        // checking if the current-value can be achieved with just the coins
        if(std::find(square_numbers.begin(),
                     square_numbers.end(),
                     curr_value)   != square_numbers.end()) {dptable[i] = 1; continue;}

        // considering all the previous guys
        auto   min_num_coins  {std::numeric_limits<int>::max()};
```

```
        for(int j = i-1; j > (i-1)/2; --j)
        {
            // getting current dp-entry
            const  auto  var00  {j};
            const  auto   var01  {curr_value - j};

            // checking what coin to use to get here
            const  auto   num_coins_required    {dptable[var00] + dptable[var01]};

            // calculating number of coins required
            min_num_coins =   std::min(num_coins_required, min_num_coins);
        }

        // writing to entry
        dptable[i] = min_num_coins;
    }

    // printing final-output
    cout << format("final-output = {}\n", dptable[dptable.size()-1]);

    // return
    return(0);

}
```

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

```cpp
int main(){

    // input- configuration
    vector<int> nums  {0,1,0,3,12};

    // setup
    int explorer   {0};
    int anchor     {0};

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

        // moving explorer until we arrive at a non-zero value
        while(explorer < nums.size() && nums[explorer] == 0) {explorer++;}

        // copying value
        if (explorer<nums.size() && anchor <nums.size())
            nums[anchor++]   = nums[explorer++];
    }

    // zeroing out the rest
    while(anchor < nums.size()) {nums[anchor++] = 0;}

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

    // return
    return(0);

}
```

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 x 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 x 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

```
int fRules(vector< vector<int> >& board,
           int                    row,
           int                    col){

    //setup
    int row_curr;
    int col_curr;
    int num_live_neighbours = 0;
    int selfstatus = board[row][col];

    for(int i = -1; i <=1; ++i){
        for(int j = -1; j<= 1; ++j){
```

```

            // finding current row and column
            row_curr = row +i;
            col_curr = col +j;

            // continuing if same element
            if (row_curr == row && col_curr == col) continue;

            // continueing for edge cases
            if (row_curr <0                  ||  \
                row_curr >= board.size()  ||  \
                col_curr < 0              ||  \
                col_curr >= board[0].size()) {continue;}

            // checking if neighbour is live or not
            if (board[row_curr][col_curr] == 1)
                ++num_live_neighbours;
        }
    }

    // the wild case where the current cell is dead and numneighbrs = 3
    if (selfstatus == 0 && num_live_neighbours == 3)   return 1;
    if (selfstatus == 0) return 0;

    // checking life-rule conditions
    if (selfstatus == 1 && num_live_neighbours < 2)    return 0;
    else if (selfstatus == 1 && num_live_neighbours < 4) return 1;
    else if (selfstatus == 1 && num_live_neighbours > 3) return 0;

    // return default
    return -1000;
    }

int main(){

```

```
    // starting timer
    Timer timer;

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

    // setup
    auto x = board;
    int statustoadd;

    // going through the elements
    for(int i = 0; i<board.size(); ++i){
        for(int j = 0; j<board[0].size(); ++j){

            // updating cell based on rules
            statustoadd = fRules(board, i, j);

            // adding status to x
            x[i][j] = statustoadd;
        }
    }

    // copying results back
    std::copy(x.begin(), x.end(), board.begin());

    // printing the matrix
    cout << format("board = \n"); fPrintMatrix(board);

    // return
    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

- $1 \leq$ pattern.length ≤ 300
- pattern contains only lower-case English letters.
- $1 \leq$ s.length ≤ 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

```
bool fCheckHashValues(const unordered_map<char, string>& charToWord,
                      const string&                      tocheck){

    // going through it
    for(auto x: charToWord) {if (x.second == tocheck) return true;}

    // return false
    return false;

}

int main(){

    // starting timer
    Timer timer;

    // input- configuration
```

```
    string pattern {"abba"};
    string s       {"dog cat cat dog"};

    // setup
    unordered_map<char, string> charToWord;
    auto wordi                {0};
    auto metaletter {false};
    auto numwords  {0};

    // going through the two variables
    for(int i = 0; i<pattern.size(); ++i){

        // temp string
        string temp;

        // extracting first word
        int prevwordi = wordi;
        while (wordi < s.size()){

            // checking if something is non-space
            if (s[wordi] != ' ')  {temp.push_back(s[wordi]); metaletter = true;}
            else                  {if (metaletter == true) break;}

            // appending wordi
            ++wordi;
        }


        // resetting stats
        metaletter = false;
        if(prevwordi!=wordi)  {++numwords;}

        // check if current char is in hashmap
        if (charToWord.find(pattern[i]) == charToWord.end()){

```

```
            // check if destination is taken
            if (fCheckHashValues(charToWord, temp)) {cout << format("final-output = false \n"); return 0;}
            else                                {charToWord[pattern[i]] = temp;}

        }
        else   {if (charToWord[pattern[i]] != temp) {cout << format("final-output = false \n"); return 0;}} // checking if mapping is same as temp
    }

    // printing
    cout << format("numwords = {}, pattern.size() = {} \n", numwords, pattern.size());

    // checking number of words and characters
    if (pattern.size()!=numwords) {cout << format("final-output = false \n"); return 0;}

    // check if there were more to go
    if (wordi!=s.size())         {cout << format("final-output = false \n"); return 0;}

    // printing
    cout << format("final-output = true \n"); return 0;
}
```

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

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto nums  {vector<int>{10,9,2,5,3,7,101,18}};

    // finding number of elements
    auto n {static_cast<int>(nums.size())};
    std::vector<int> dp(n, 1);   // table to store the dynamic variables

    // going through the arrays
    for(int i = 0; i<n; ++i)
        for(int j = 0; j<i; ++j)
            if (nums[i] > nums[j]) {dp[i] = std::max(dp[i], dp[j]+1);}

    // printing max-value
    auto maxiter = std::max_element(dp.begin(), dp.end());
    cout << format("final-output = {}\n", *maxiter);

    // return
    return(0);

}
```

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

- $1 \leq \text{coins.length} \leq 12$
- $1 \leq \text{coins[i]} \leq 2^{3}1 - 1$
- $0 \leq \text{amount} \leq 104$

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto    coins      {std::vector<int>({1,2,5})};
    auto    amount     {11};

    // trivial cases
    if (coins.size() == 1 && (amount % coins[0] == 0))
    {
        return static_cast<int>(amount/coins[0]);
    }
    if (amount == 0)  {return 0;}

    // setup
    vector<int> dptable(amount + 1, std::numeric_limits<int>::max());
    dptable[0] = 0;

    // going through each coin
    for (int coin : coins) {
        // calculating upward from that coin
```

```
        for (int x = coin; x <= amount; ++x) {
            if (dptable[x - coin] != std::numeric_limits<int>::max())
                dptable[x] = std::min(dptable[x],
                                      dptable[x - coin] + 1);
        }
    }

    // returning final output
    return dptable[amount] == std::numeric_limits<int>::max() ? -1 : dptable[amount];

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




    // return
    return(0);

}
```

329. Longest Increasing Path in a Matrix

Given an m x n integers matrix, return the length of the longest increasing path in matrix.

From each cell, you can either move in four directions: left, right, up, or down. You may not move diagonally or move outside the boundary (i.e., wrap-around is not allowed).

Examples

1. **Example 1:**
   - Input: matrix = [[9,9,4],[6,6,8],[2,1,1]]
   - Output: 4
   - Explanation: The longest increasing path is [1, 2, 6, 9].
2. **Example 2:**
   - Input: matrix = [[3,4,5],[3,2,6],[2,2,1]]
   - Output: 4
   - Explanation: The longest increasing path is [3, 4, 5, 6]. Moving diagonally is not allowed.
3. **Example 3:**
   - Input: matrix = [[1]]
   - Output: 1

Constraints

- m == matrix.length
- n == matrix[i].length
- $1 \leq m, n \leq 200$
- $0 \leq matrix[i][j] \leq 2^{31} - 1$

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto   matrix    {std::vector<std::vector<int>>({
        {0,1,2,3,4,5,6,7,8,9},
        {19,18,17,16,15,14,13,12,11,10},
        {20,21,22,23,24,25,26,27,28,29},
        {39,38,37,36,35,34,33,32,31,30},
        {40,41,42,43,44,45,46,47,48,49},
        {59,58,57,56,55,54,53,52,51,50},
        {60,61,62,63,64,65,66,67,68,69},
        {79,78,77,76,75,74,73,72,71,70},
        {80,81,82,83,84,85,86,87,88,89},
        {99,98,97,96,95,94,93,92,91,90},
        {100,101,102,103,104,105,106,107,108,109},
        {119,118,117,116,115,114,113,112,111,110},
        {120,121,122,123,124,125,126,127,128,129},
        {139,138,137,136,135,134,133,132,131,130},
```

```
        {0,0,0,0,0,0,0,0,0,0,0}
    })};



    int rows = matrix.size();
    int cols = matrix[0].size();
    std::vector<std::vector<int>> dp(rows, std::vector<int>(cols, 0)); // memoization
    int ans = 0;

    // directions: right, up, left, down
    int dr[] = {0, -1, 0, 1};
    int dc[] = {1, 0, -1, 0};

    std::function<int(int,int)> dfs = [&](int r, int c) {
        if(dp[r][c] != 0) return dp[r][c]; // already computed
        int maxLen = 1; // at least this cell

        for(int d=0; d<4; ++d){
            int nr = r + dr[d], nc = c + dc[d];
            if(nr >= 0 && nr < rows && nc >=0 && nc < cols && matrix[nr][nc] > matrix[r][c]){
                maxLen = max(maxLen, 1 + dfs(nr, nc));
            }
        }
        dp[r][c] = maxLen;
        return maxLen;
    };

    for(int r=0; r<rows; ++r){
        for(int c=0; c<cols; ++c){
            ans = max(ans, dfs(r,c));
        }
    }

    // printing the finaloutput
```

```
    cout << format("final-output = {}\n", ans);



    // return
    return(0);

}
```

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

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

Code

```cpp
int main(){

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

    // going through the string
    string     vowels        {"aeiouAEIOU"};
    vector<int> vowel_indices;
    string     reversed_vowels;
    string     finaloutput = s;

    // going through the string
    for(int i = 0; i<s.size(); ++i){
        if (vowels.find(s[i]) != string::npos){
            reversed_vowels+=s[i];
            vowel_indices.push_back(i);
        }
    }

    // refilling the indices
    for(int i = 0; i<vowel_indices.size(); ++i){
        finaloutput[vowel_indices[i]] = reversed_vowels[reversed_vowels.size()-1-i];
    }

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

    // return
    return(0);

}
```

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

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^4 \leq \text{nums[i]} \leq$
- k is in the range [1, the number of unique elements in the array].
- It is guaranteed that the answer is unique.

Code

```cpp
int main(){

    // starting timer
    Timer timer;

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

    // setup
    unordered_map<int, int> hist;

    // building histogram
    for(const auto& num: nums){
        if (hist.find(num) == hist.end()) {hist[num] = 1;}
        else                              {++hist[num];}
    }

    // creating vector pair
    vector<vector<int>> kv;
    for(const auto& x: hist)     {kv.emplace_back(vector<int>{x.first, x.second});}

    // building the sorting lambda
    auto sortinglambda = [](const auto& a, const auto& b) {return a[1] > b[1];};

    // since we want just the first k-elements, use partial sort
    std::partial_sort(kv.begin(), kv.begin() + k, kv.end(), sortinglambda);

    // building the final output
    auto finaloutput  {vector<int>{}};
    for(int i = 0; i<k; ++i) {finaloutput.push_back(kv[i][0]);}

    // printing the final output
```

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

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

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

- $1 \leq$ nums1.length, nums2.length ≤ 1000
- $0 \leq$ nums1[i], nums2[i] ≤ 1000

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto nums1 {vector<int>{4,9,5}};
    auto nums2 {vector<int>{9,4,9,8,4}};

    // setup
    std::set<int> nums1set(nums1.begin(), nums1.end());
    std::set<int> nums2set(nums2.begin(), nums2.end());

    // printing
    cout << format("nums1set = {}\n", nums1set);
    cout << format("nums2set = {}\n", nums2set);

    // calculating the intersection
    vector<int> finaloutput;
    for(const auto& x: nums1set){

        if (std::find(nums2set.begin(),
                      nums2set.end(),
                      x) != nums2set.end()) {finaloutput.push_back(x);}
    }

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

    // return
    return(0);

}
```

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

- $1 \leq$ nums1.length, nums2.length ≤ 1000
- $0 \leq$ nums1[i], nums2[i] ≤ 1000

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto nums1    {vector<int>{4,9,5}};
    auto nums2    {vector<int>{9,4,9,8,4}};

    // setup
    unordered_map<int, int> histogram1;
    unordered_map<int, int> histogram2;

    // lambda to add to histogram
    auto addtohistogram = [](unordered_map<int, int>& histogram, int entryvalue){
        if (histogram.find(entryvalue) == histogram.end()) {histogram[entryvalue] = 1;}
        else                                               {++histogram[entryvalue];}
    };

    // building the histogram
    for(const auto x: nums1) {addtohistogram(histogram1, x);}
    for(const auto x: nums2) {addtohistogram(histogram2, x);}

    // going through the first histogram
    vector<int> finaloutput;
    for(const auto kvp: histogram1){

        // splitting the pair
        auto& keyvalue {kvp.first};
        auto& value    {kvp.second};

        // checking if it is there in the second histogram
        if(histogram2.find(keyvalue) != histogram2.end()){
```

```
            auto countforthis {std::min(value, histogram2[keyvalue])};
            auto tempvector   {vector<int>(countforthis, keyvalue)};
            for(int i = 0; i < countforthis; ++i) {finaloutput.push_back(keyvalue);}
        }

    }

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

    // return
    return(0);

}
```

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, target = 5
    - Output: false

3. **Example 3:**
    - Input: x = 1, y = 2, 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, target ≤ 103

Code

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto    x           {34};
    auto    y           {5};
    auto    target      {6};

    // trivial case
    if (target > (x + y)) {return false;}

    // lambda to calculate gcd
    std::function<int(int, int)>
```

```
    calculateGCD = [&calculateGCD](int x,
                                   int y){
        if (y == 0) {return x;};
        return calculateGCD(y, x%y);
    };

    // calculating gcd
    auto    gcd_xy          {calculateGCD(x, y)};
    auto    finaloutput     {target % calculateGCD(x, y) == 0};

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

    // return
    return(0);

}
```

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

- $1 \leq$ ransomNote.length, magazine.length $\leq 10^5$
- ransomNote and magazine consist of lowercase English letters.

Code

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    string ransomNote {"aa"};
    string magazine   {"aab"};

    // trivial case
    if (ransomNote.size()>magazine.size()) {
        cout << format("status = true\n");
        return 0;
    }

    // setup
    vector<int> hist(26,0);

    // building histogram
    for(auto x: magazine)    {++hist[static_cast<int>(x-'a')];}

    // going through ransom note
    for(auto x: ransomNote){
        if(hist[static_cast<int>(x-'a')]-- == 0) {
            cout << format("status = true\n"); return 0;
        }
    }

    // return true
    cout << format("status = true\n");

    // return
    return(0);
```

```
}
```

392. Is Subsequence

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

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

Examples

1. **Example 1:**
   - Input: s = ”abc”, t = ”ahbgdc”
   - Output: true
2. **Example 2:**
   - Input: s = ”axc”, t = ”ahbgdc”
   - Output: false

Constraints

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

Code

```cpp
int main(){

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

    // setup
    int i = 0;

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

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


    // return
    return(0);

}
```

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: ”abcabccdcdcdef”

Constraints:

- $1 \leq$ s.length ≤ 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

```
int main(){

    // input- configuration
    string s   {"100[leetcode]"};

    // running
    std::stack<char>  mystack;                                          // stack
    string            temp;                                             // temporary string used for decoding
    int               repcount   {1};                                   // used for decoding

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

        if(s[i] != ']')   {mystack.push(s[i]);}                          // pushing characters to stack until we arrive at "]"
        else{
            temp = "";                                                   // initializing temporary string
            while(mystack.top() != '[') {                                // expanding mini-string until we arrive at "["
                temp = mystack.top() + temp;
                mystack.pop();
```

```
        }

        mystack.pop();                                              // removing "["

        // calculating the repcount
        string numberasstring = "";
        while(mystack.size() != 0 &&
             mystack.top() - '0' >= 0 && '9' - mystack.top() >= 0)
        {
            numberasstring = mystack.top() + numberasstring;
            mystack.pop();
        }
        repcount = std::stoi(numberasstring);

        // mini-decoding
        int multitempsize = repcount * temp.size();                 // calculating size after multiplication
        for(int j = 0; j<multitempsize; ++j) {                      // filling up the stack with decoded content
            mystack.push(temp[j%temp.size()]);
        }
    }
}

// creating the final output
string finaloutput;
while(mystack.size()){
    finaloutput = mystack.top() + finaloutput;
    mystack.pop();
}

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

// return
return(0);
```

```
}
```

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

```cpp
int main(){

    // starting timer
    Timer timer;

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

    // setup
    auto   largestnumber  {std::numeric_limits<int>::min()};
    auto   numsset        {std::set<int>(nums.begin(), nums.end())};
    auto   myheap         {std::vector<int>()};
    std::make_heap(myheap.begin(),
                   myheap.end(),
                   std::greater<int>());

```

```cpp
    // going through the histogram
    for(const auto x: numsset){

        // adding element to it
        myheap.push_back(x);
        largestnumber = std::max(largestnumber, x);

        // making a heap out of it
        std::push_heap(myheap.begin(),
                       myheap.end(),
                       std::greater<int>());

        // popping if size is greater
        if (myheap.size() > 3){
            std::pop_heap(myheap.begin(),
                          myheap.end(),
                          std::greater<int>());
            myheap.pop_back();
        }
    }

    // taking the last element
    int finaloutput = largestnumber;
    if (myheap.size() == 3){
        std::pop_heap(myheap.begin(),
                      myheap.end(),
                      std::greater<int>());
        finaloutput = myheap.back();
    }

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

    // return
    return(0);
```

```
}
```

417. Pacific Atlantic Water Flow

There is an m x 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 x 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] = $[r_i, c_i]$ 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 == heights.length
- n == heights[r].length
- $1 \leq$ m, n ≤ 200
- $0 \leq$ heights[r][c] $\leq 10^5$

Code

```cpp
int main(){

    // starting timer
    Timer timer;

    // input-configuration
    auto    heights    {std::vector<std::vector<int>>{
        {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}
    }};

    // setup
    std::vector<int>  tidList;
    auto    finaloutput    {std::vector<std::vector<int>>()};

    // lambda for calculating tid
    auto    fetchtid    =    [&heights](const int row, const int col){
        if (row < 0 || col < 0) throw std::runtime_error("row, col < 0");
```

```
        return col + row * heights[0].size();
    };

    // building marking table
    auto   visitingTable {std::vector<std::vector<int>>(
        heights.size(),
        std::vector<int>(heights[0].size(), 0)
    )};

    // beginning search from pacific
    auto   pacific_pipe   {std::deque<std::vector<int>>()};

    // adding to pacific pipe before running bfs
    for(auto row = 0; row < heights.size(); ++row){
        pacific_pipe.push_back(std::vector<int>({row, 0}));
    }
    for(auto col = 1; col < heights[0].size(); ++col){
        pacific_pipe.push_back(std::vector<int>( {0, col} ));
    }

    auto   checkIfUpstream = [&heights](
        const  auto&  curr_row,
        const  auto&  curr_col,
        const  auto&  next_row,
        const  auto&  next_col){

        // checking if either coordinates are valid
        if (curr_row < 0 || curr_row >= heights.size()) {return false;}
        if (curr_col < 0 || curr_col >= heights[0].size()) {return false;}
        if (next_row < 0 || next_row >= heights.size()) {return false;}
        if (next_col < 0 || next_col >= heights[0].size()) {return false;}

        // checking difference in value
        return heights[next_row][next_col] >= heights[curr_row][curr_col];
    };
```

```
    // running bfs from
    while(pacific_pipe.size()!=0)
    {
        // popping front
        auto    curr    {pacific_pipe.front()}; pacific_pipe.pop_front();
        auto&   row     {curr[0]};
        auto&   col     {curr[1]};

        // ERROR-CHECK: checking if it is already visited
        const  auto    tid     {fetchtid(row, col)};
        if(std::find(tidList.begin(),
                     tidList.end(),
                     tid) != tidList.end())            {continue;}

        // adding to visit-list
        tidList.push_back(tid);

        // marking to dptable
        visitingTable[row][col] += 1;

        // adding the neighbours to the list
        if(checkIfUpstream(row, col, row, col+1)) {pacific_pipe.push_back(std::vector<int>({row, col+1}));}
        if(checkIfUpstream(row, col, row-1, col)) {pacific_pipe.push_back(std::vector<int>({row-1, col}));}
        if(checkIfUpstream(row, col, row, col-1)) {pacific_pipe.push_back(std::vector<int>({row, col-1}));}
        if(checkIfUpstream(row, col, row+1, col)) {pacific_pipe.push_back(std::vector<int>({row+1, col}));}
    }


    // creating atlantic pipe
    auto&  atlantic_pipe {pacific_pipe}; pacific_pipe.clear();
    tidList.clear();

    // filling up atlantic pipe
    for(auto   row = 0; row < heights.size(); ++row)
```

```
        atlantic_pipe.push_back({row, static_cast<int>(heights[0].size() - 1)});
    for(auto   col = 0; col < heights[0].size(); ++col)
        atlantic_pipe.push_back({static_cast<int>(heights.size()-1), col});

    // running bfs
    while(atlantic_pipe.size() != 0)
    {
        // popping front
        auto curr  {atlantic_pipe.front()}; atlantic_pipe.pop_front();
        auto&  row    {curr[0]};
        auto&  col    {curr[1]};

        // checking if visited
        const  auto   tid    {fetchtid(row, col)};
        if (std::find(tidList.begin(),
                      tidList.end(),
                      tid) != tidList.end())  {continue;}

        // adding to tid
        tidList.push_back(tid);

        // marking the table
        visitingTable[row][col] += 2;

        // adding to final output
        if (visitingTable[row][col] == 3)
            finaloutput.push_back(std::vector<int>({row, col}));

        // adding the neighbours to the list
        if(checkIfUpstream(row, col, row, col+1)) {atlantic_pipe.push_back(std::vector<int>({row, col+1}));}
        if(checkIfUpstream(row, col, row-1, col)) {atlantic_pipe.push_back(std::vector<int>({row-1, col}));}
        if(checkIfUpstream(row, col, row, col-1)) {atlantic_pipe.push_back(std::vector<int>({row, col-1}));}
        if(checkIfUpstream(row, col, row+1, col)) {atlantic_pipe.push_back(std::vector<int>({row+1, col}));}

    }
```

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

    // return
    return(0);

}
```

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 = "ABAB", k = 2
   - Output: 4
   - Explanation: Replace the two 'A's with two 'B's or vice versa.
2. **Example 2:**
   - Input: s = "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 exists other ways to achieve this answer too.

Constraints

- $1 \leq \text{s.length} \leq 10^5$

- s consists of only uppercase English letters.
- $0 \leq k \leq$ s.length

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto s {string("ABAB")};
    auto k {2};

    // setup
    auto left                {0};
    auto right               {0};
    auto num_characters_local {0};
    auto finalOutput         {0};
    auto maxelement          {-1};

    std::vector<int> characterCount(26, 0);                              // histogram

    // going through the elements
    for(right = 0; right < s.size(); ++right){

        // adding to count
        characterCount[static_cast<int>(s[right])-65] += 1;

        // setup
        num_characters_local = right - left + 1;

```

```
        // checking if number of repl
        maxelement = *(std::max_element(characterCount.begin(), characterCount.end()));

        if (num_characters_local - maxelement > k ) {

            characterCount[static_cast<int>(s[left])-65] -= 1;            // removing the lef tmost element
            ++left;                                                        // moving left
        }

        // calculating longest
        finalOutput = std::max(finalOutput, right-left+1);

    }

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

    // return
    return(0);

}
```

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

```cpp
// Definition for a Node.
class Node {
public:
    int val;
    vector<Node*> children;

    Node() {}

    Node(int _val) {
        val = _val;
    }

    Node(int _val, vector<Node*> _children) {
        val = _val;
        children = _children;
    }
};
```

```
struct NodeLevelPair
{
    Node*   ptr;
    int     level;
    NodeLevelPair() = default;
    NodeLevelPair(Node* ptr_, int level_): ptr(ptr_), level(level_) {}
};

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto   root   {new Node(1)};
    root->children.push_back(new Node(3));
    root->children.push_back(new Node(2));
    root->children.push_back(new Node(4));
    root->children[0]->children.push_back(new Node(5));
    root->children[0]->children.push_back(new Node(6));

    // setup
    std::vector<std::vector<int>> finaloutput;
    auto   pipe   {std::deque<NodeLevelPair>()};
    pipe.push_back(NodeLevelPair({root, 0}));


    // bfs
    while(pipe.size()!= 0)
    {
        // popping front
        auto   curr   {pipe.front()}; pipe.pop_front();
        auto   curr_level {curr.level};

```

```
        // writing to final output
        if (curr_level < finaloutput.size()) {finaloutput[curr_level].push_back(curr.ptr->val);}
        else   {
            finaloutput.push_back(std::vector<int>({curr.ptr->val}));
        }

        // printing final output
        cout << format("curr->val = {}, finaloutput = {}\n", curr.ptr->val, finaloutput);

        // adding children
        for(const  auto&  child: curr.ptr->children)
            pipe.push_back({child, curr_level + 1});
    }

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


    // return
    return(0);

}
```

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" $\rightarrow$ "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

- $0 \leq$ bank.length ≤ 10
- startGene.length == endGene.length == bank[i].length == 8
- startGene, endGene, and bank[i] consist of only the characters ['A', 'C', 'G', 'T'].

Code

```
void foo(unordered_map<string, vector<string>>& stringtoneighbours,
        vector<string>                        pathsofar,
        bool&                                 foundpath,
        string                                endGene,
        int&                                  finaloutput){

    // checking if the top of the stack has valid neighbours
    string top = pathsofar[pathsofar.size()-1];

    // checking if the current one is thef inal output
    if (top == endGene)
        finaloutput = finaloutput < pathsofar.size() ? finaloutput : pathsofar.size();

    // checking its possible next-states
    auto nextnodes = stringtoneighbours[top];

    // going depth-first
    for(auto x: nextnodes){

        // not considering if it is already in the path
        if(std::find(pathsofar.begin(),
                     pathsofar.end(),
                     x) != pathsofar.end()) continue;
```

```cpp

        // updating path sof far
        auto pathsofar_temp = pathsofar;
        pathsofar_temp.push_back(x);

        // calling function do it
        foo(stringtoneighbours, pathsofar_temp, foundpath, endGene, finaloutput);
    }
}

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

    // starting timer
    Timer timer;

    // input- configuration
    string startGene      = "AACCGGTT";
    string endGene        = "AACCGGTA";
    vector<string> bank   = {"AACCGGTA"};

    // setup
    unordered_map<string, vector<string>> stringtoneighbours;

    // checking if endgene is in the bank
    if (std::find(bank.begin(),
                  bank.end(),
                  endGene) == bank.end()) {cout << format("finalOutput = -1\n");}

    // going through the bank and building neighbours
    bank.push_back(startGene);

    for(auto x: bank){

        // finding valid transactions with the other strings in the bank
```

```
        for(auto y: bank){

            // checking number of string differences between the two
            auto count = 0;
            for(int i =0; i<8; ++i) {if (x[i] != y[i]) ++count;}

            // checking count and adding to valid transactions
            if (count == 1)       {stringtoneighbours[x].push_back(y);}
        }

        // checking if this particular gene can jump to final gene
        // checking number of string differences between the two
        auto count = 0;
        for(int i =0; i<8; ++i) {if (x[i] != endGene[i]) ++count;}

        // checking count and adding to valid transactions
        if (count == 1)       {stringtoneighbours[x].push_back(endGene);}
    }

    // recursion
    vector<string> pathsofar {startGene};
    bool            foundpath  {false};
    int             finalOutput {-1};

    // calling the function
    foo(stringtoneighbours, pathsofar, foundpath, endGene, finalOutput);

    // returning the finaloutput
    if (finalOutput > -1) --finalOutput;

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

    // return
    return(0);
```

```
}
```

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

```cpp
int main(){

    // input- configuration
    vector<char> chars {'a','b','b','b','b','b','b','b','b','b','b','b','b'};

    // going through the character
    int p1            {0};
    char runningchar  {};
    char curr         {};
    int count         {0};
    string finaloutput;

    // going through the inputs
    while(p1<chars.size()){

        // getting curren tchar
```

```
        curr = chars[p1];

        // increasing count
        if (count == 0)                {runningchar = chars[p1]; ++count;}
        else if(curr == runningchar) {++count;}
        else if(curr != runningchar) {
            finaloutput += runningchar; // writing character to current pointer
            if (count != 1)
                finaloutput += std::to_string(count); // increasing write-pointer
            runningchar = curr;
            count = 1;
        }

        // increasing pointer
        ++p1;
    }

    // flushing out
    if (count != 0){
        finaloutput += runningchar; // writing character to current pointer
        if (count != 1)
            finaloutput += std::to_string(count); // increasing write-pointer
    }

    // writing to input
    for(int i = 0; i<finaloutput.size(); ++i){
        chars[i] = finaloutput[i];
    }

    // printing the final output
    cout << format("finaloutput = {}\n", finaloutput);
    cout << "chars = "; fPrintVector(chars);
    cout << format("return-value = {}\n", finaloutput.size());

    // return
```

```
    return(0);

}
```

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
- $1 \leq n \leq 10^5$
- $1 \leq nums[i] \leq n$

Code

```cpp
int main(){

    // starting timer
    Timer timer;

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

    // setup
    auto flags {vector<bool>(nums.size(), false)};

    // going through the array
    for(const auto& x: nums) {flags[x-1] = true;}

    // building final-output
    auto finaloutput  {vector<int>{}};
    for(int i = 0; i<flags.size(); ++i) {if (flags[i] == false) {finaloutput.push_back(i+1);}}

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

    // return
    return(0);

}
```

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 ¡= x ¡= 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

- $1 \leq \text{points.length} \leq 10^5$
- points[i].length == 2
- $-2^31 \leq$ xstart ¡ xend $\leq 2^31 - 1$

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    vector<vector<int>> points{
        {10,16},
        {2,8},
        {1,6},
        {7,12}
    };

    // setup
```

```cpp
    vector< vector<int> > intersections;

    // sorting intervals in case they aren't
    std::sort(points.begin(), \
            points.end(), \
            [](const vector<int>& a, const vector<int>& b){
                return a[0]<b[0];});

    // pushing the first interval to the intersections list
    intersections.push_back(points[0]);
    int finaloutput = 0;

    // going through arrays one by one
    for(int i = 1; i<points.size(); ++i){

        // fetching reference to running intersection
        auto& runningInterval = intersections[intersections.size()-1];

        // fetching incoming interval
        auto incomingInterval = points[i];

        // finding intersections
        if (runningInterval[1] < incomingInterval[0]){
            // no intersection,
            ++finaloutput; intersections.push_back(incomingInterval);
        }
        else{
            // calculating max of start-point
            runningInterval[0] = max(runningInterval[0], incomingInterval[0]);

            // calculating min of end-poitns
            runningInterval[1] = min(runningInterval[1], incomingInterval[1]);
        }
    }
```

```
    // returning results
    cout << format("final-output = {}\n", intersections.size());

    // return
    return(0);

}
```

463. Island Perimeter

You are given row x col grid representing a map where grid[i][j] = 1 represents land and 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: 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: grid = [[1]]
    - Output: 4
3. **Example 3:**
    - Input: grid = [[1,0]]
    - Output: 4

Constraints

- row == grid.length
- col == grid[i].length
- $1 \leq$ row, col ≤ 100
- grid[i][j] is 0 or 1.
- There is exactly one island in grid.

Code

```
bool fCheckCellValidity(const vector<vector<int>>& nums,
                        const vector<vector<bool>>&  flags,
                        vector<int>                  currposition,
                        const std::deque<vector<int>>& pipe){

    // splitting tid into row and column
    auto& rowcurr  {currposition[0]};
    auto& colcurr  {currposition[1]};

    // checking validity
    if (rowcurr < 0 || rowcurr >= nums.size()) {return false;}
    if (colcurr < 0 || colcurr >= nums[0].size()) {return false;}

    // checking visiters register
    if (flags[rowcurr][colcurr] == true)       {return false;}

    // checking if this position is an island or water
    if (nums[rowcurr][colcurr] == 0)           {return false;}

```

```cpp
    // returning false if the currposition is already in the pipe
    if (std::find(pipe.begin(),
                  pipe.end(),
                  currposition) != pipe.end())   {return false;}

    // returning true
    return true;

}
bool fCheckBounds(const vector<vector<int>>&   nums,
                  const int                    rowcurr,
                  const int                    colcurr){

    // checking validity
    if (rowcurr < 0 || rowcurr >= nums.size()) {return false;}
    if (colcurr < 0 || colcurr >= nums[0].size()) {return false;}

    // returning true
    return true;

}
void foo(const vector<vector<int>>& nums,
         vector<vector<bool>>&      flags,
         vector<int>                currposition,
         int&                       runningperimeter,
         std::deque<vector<int>>    pipe){

    // splitting tid into row and column
    auto rowcurr  {currposition[0]};
    auto colcurr  {currposition[1]};

    // checking if current cell has already been visited
    if (nums[rowcurr][colcurr] == 0 || flags[rowcurr][colcurr] == true) {return;}

    // setting the current to true
```

```cpp
    flags[rowcurr][colcurr]   = true;

    // visiteds result in reduced boundaries
    auto amounttosubtract {0};
    if(fCheckBounds(nums, rowcurr, colcurr+1) && flags[rowcurr][colcurr+1] == true) {++amounttosubtract;}
    if(fCheckBounds(nums, rowcurr-1, colcurr) && flags[rowcurr-1][colcurr] == true) {++amounttosubtract;}
    if(fCheckBounds(nums, rowcurr, colcurr-1) && flags[rowcurr][colcurr-1] == true) {++amounttosubtract;}
    if(fCheckBounds(nums, rowcurr+1, colcurr) && flags[rowcurr+1][colcurr] == true) {++amounttosubtract;}

    // unvisited cells add to boundary
    auto amounttoadd      {0};
    if(fCheckBounds(nums, rowcurr, colcurr+1) == false || (fCheckBounds(nums, rowcurr, colcurr+1) &&
        flags[rowcurr][colcurr+1] == false) ) {++amounttoadd;}
    if(fCheckBounds(nums, rowcurr-1, colcurr) == false || (fCheckBounds(nums, rowcurr-1, colcurr) &&
        flags[rowcurr-1][colcurr] == false) ) {++amounttoadd;}
    if(fCheckBounds(nums, rowcurr, colcurr-1) == false || (fCheckBounds(nums, rowcurr, colcurr-1) &&
        flags[rowcurr][colcurr-1] == false) ) {++amounttoadd;}
    if(fCheckBounds(nums, rowcurr+1, colcurr) == false || (fCheckBounds(nums, rowcurr+1, colcurr) &&
        flags[rowcurr+1][colcurr] == false) ) {++amounttoadd;}

    // updating the running perimeter
    runningperimeter += amounttoadd - amounttosubtract;

    // // if next is one, has not been visited, we add to the stack
    if(fCheckCellValidity(nums, flags, {rowcurr, colcurr+1}, pipe)) {pipe.push_back({rowcurr, colcurr+1});}
    if(fCheckCellValidity(nums, flags, {rowcurr-1, colcurr}, pipe)) {pipe.push_back({rowcurr-1, colcurr});}
    if(fCheckCellValidity(nums, flags, {rowcurr, colcurr-1}, pipe)) {pipe.push_back({rowcurr, colcurr-1});}
    if(fCheckCellValidity(nums, flags, {rowcurr+1, colcurr}, pipe)) {pipe.push_back({rowcurr+1, colcurr});}

    // sending it back if thing is empty
    if (pipe.size() == 0) {return;}

    // popping one from the middle
    auto frontentry = pipe.front(); pipe.pop_front();

```

```
    // calling the function on the next
    foo(nums,
        flags,
        frontentry,
        runningperimeter,
        pipe);

}

int main(){

    // starting timer
    Timer timer;

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

    // setup
    vector<vector<bool>> flags(nums.size(),
                               vector<bool>(nums[0].size(),
                                            false));
    auto runningperimeter {0};
    std::deque<vector<int>> pipe;

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

        // finding where the number one is
        auto it    {std::find(nums[i].begin(),
                              nums[i].end(),
```

```
                              1)};

        // continuing if there is no ones here
        if (it == nums[i].end()) {continue;}

        // finding the starting point
        int startingcol  = std::distance(nums[i].begin(), it);

        // launching a search
        foo(nums,
            flags,
            {i,startingcol},
            runningperimeter,
            pipe);

        // breaking
        break;
    }

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

    // return
    return(0);

}
```

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

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto nums  {vector<int>{1,1,0,1,1,1}};

    // setup
    auto p1           {-1};
    auto finaloutput  {-1};

    // going through the nums
    for(int i = 0; i<nums.size(); ++i){
        if (nums[i] == 1){
            if (p1 == -1)  {p1 = i;}                        // updating p1
            auto temp  {i - p1 + 1};                        // calculating size so far
            finaloutput = std::max(temp, finaloutput); // updating finaloutput
        }
        else   {p1 = -1;}
    }

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

    // return
    return(0);

}
```

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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root          {new TreeNode(4)};
    root->left         = new TreeNode(2);
    root->right        = new TreeNode(6);
    root->left->left   = new TreeNode(1);
    root->left->right  = new TreeNode(3);

    // setup
    vector<int> nodevalues;
    std::function<void(const TreeNode*)> foo = [&foo, &nodevalues](
        const TreeNode* root){
        // returning
```

```
        if (root == nullptr) return;

        // going down left
        if (root->left)   {foo(root->left);}

        // adding current-value
        nodevalues.push_back(root->val);

        // going down right
        if (root->right)  {foo(root->right);}

        // returning
        return;
    };

    // calling function
    foo(root);

    // moving through node values
    auto minvalue {std::numeric_limits<int>::max()};
    auto temp     {-1};
    for(int i = 0; i<nodevalues.size()-1; ++i){
        // checking difference
        temp       = std::abs(nodevalues[i]- nodevalues[i+1]);
        minvalue   = std::min(minvalue, temp);
    }

    // returning minvalue
    cout << format("final-output = {}\n", minvalue);

    // return
    return(0);

}
```

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

```

// Definition for a Node.
class Node {
public:
    int val;
    vector<Node*> children;

    Node() {}

    Node(int _val) {
        val = _val;
    }

```

```
    Node(int _val, vector<Node*> _children) {
        val = _val;
        children = _children;
    }
};

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto    root    {new    Node(1)};
    root->children.push_back(new Node(3));
    root->children.push_back(new Node(2));
    root->children.push_back(new Node(4));
    root->children[0]->children.push_back(new Node(5));
    root->children[0]->children.push_back(new Node(6));

    // setup
    int     final_output   {std::numeric_limits<int>::min()};

    // defining
    std::function<void(const Node*, int)>
    fCalculate = [&fCalculate,
                  &final_output](
                      const         Node*    curr_node,
                      int                    curr_level){

        // adding height
        ++curr_level;

        // check if we're at the end
        if (curr_node->children.size() == 0){
            final_output   =   final_output > curr_level ? final_output : curr_level;
```

```
            return;
        }

        // calling function on the rest of them
        for(const  auto&  x:  curr_node->children)  {fCalculate(x,curr_level);}

        // returning
        return;

    };

    // calling the function
    fCalculate(root, 0);

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

    // return
    return(0);

}
```

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

- $1 \leq$ nums.length $\leq 2 * 10^4$
- -1000 $\leq$ nums[i] $\leq$ 1000
- $-10^7 \leq \mathrm{k} \leq 10^7$

Code

```
int main(){

    // starting timer
    Timer timer;

    // input-configuration
    auto nums  {vector<int>{6,4,3,1}};
    auto k     {10};

    // setup
    auto leftSum   {vector<int>(nums.size())};
    std::partial_sum(nums.begin(), nums.end(), leftSum.begin());

    // calculating final-output
    auto finaloutput  {0};
    finaloutput += std::count(leftSum.begin(), leftSum.end(), k);

    // moving from left to right
    for(int i = 0; i < leftSum.size(); ++i){
        for(int j = i+1; j < leftSum.size(); ++j){
            auto sumvalue {leftSum[j] - leftSum[i]};
            if (sumvalue == k) {++finaloutput;}
        }
    }

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

    // return
    return(0);

}
```

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 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

```
bool fAreWeTheSame(TreeNode* root, TreeNode* subroot){

    // checking if root-note values are the same
    if (root == nullptr && subroot == nullptr) return true;
    if (root == nullptr || subroot == nullptr) return false;
    if (root->val != subroot->val)             return false;

    // going through the branches
    auto isLeftBranchEqual   {fAreWeTheSame(root->left,   subroot->left)};
    auto isRightBranchEqual  {fAreWeTheSame(root->right,  subroot->right)};
```

```

    // returning
    return (isLeftBranchEqual && isRightBranchEqual);
}

void f(TreeNode* root, TreeNode* subroot, bool& logicAccum){

    // base-case
    if (root == nullptr || subroot == nullptr) return;

    // checking if they have the same head-value
    if (root->val == subroot->val)
        logicAccum = logicAccum || fAreWeTheSame(root, subroot);

    // sending it back if we found one already
    if (logicAccum) return;

    // moving onto the next
    f(root->left,     subroot, logicAccum);
    f(root->right,    subroot, logicAccum);

    // return
    return;

}

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root         {new TreeNode(3)};
    root->left        = new TreeNode(4);
    root->right       = new TreeNode(5);
```

```
    root->left->left  = new TreeNode(1);
    root->left->right = new TreeNode(2);

    auto subRoot      {new TreeNode(4)};
    subRoot->left     = new TreeNode(1);
    subRoot->right    = new TreeNode(2);

    // setup
    bool logicAccum = false;
    f(root, subRoot, logicAccum);

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

    // return
    return(0);

}
```

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 pi 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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto p1     {vector<int>{0, 0}};
    auto p2     {vector<int>{5, 0}};
    auto p3     {vector<int>{5, 5}};
    auto p4     {vector<int>{0, 5}};

    // setup
    auto finaloutput  {false};
    auto center    = [&p1](auto p){
        return vector<int>{p[0] - p1[0], p[1] - p1[1]};
    };
    auto addvectors = [](auto p00, auto p01){
        return vector<int>{p00[0] + p01[0], p00[1] + p01[1]};
    };

    // changing the coordinate
    auto p1_centered  {center(p1)};
    auto p2_centered  {center(p2)};
    auto p3_centered  {center(p3)};
    auto p4_centered  {center(p4)};

    // checking if its parallelogram
    auto parallelogram_check {false};
    vector<int> first_arm, second_arm;
    if (p4_centered == addvectors(p2_centered, p3_centered))  {
        parallelogram_check = true;
        first_arm         = p2_centered;
        second_arm        = p3_centered;
```

```
    }
    else if (p3_centered == addvectors(p2_centered, p4_centered)) {
        parallelogram_check = true;
        first_arm           = p2_centered;
        second_arm          = p4_centered;
    }
    else if (p2_centered == addvectors(p3_centered, p4_centered)) {
        parallelogram_check = true;
        first_arm           = p3_centered;
        second_arm          = p4_centered;
    }

    // perpendicular check
    auto perpendicular_check {false};
    auto inner_product = std::inner_product(first_arm.begin(), first_arm.end(),
                                            second_arm.begin(),
                                            0);
    perpendicular_check   = inner_product == 0 ? true : false;

    // length check
    auto length_check        {false};
    auto first_arm_length {std::accumulate(first_arm.begin(),
                                            first_arm.end(),
                                            0,
                                            [](auto acc, auto argx){ return acc + argx * argx;})};
    auto second_arm_length {std::accumulate(second_arm.begin(),
                                            second_arm.end(),
                                            0,
                                            [](auto acc, auto argx){ return acc + argx * argx;})};
    length_check   = first_arm_length == second_arm_length ? true : false;
    if (first_arm_length == 0) {length_check = false;}

    // performing final output
    if (parallelogram_check && perpendicular_check && length_check) {finaloutput = true;}

```

```
    // printing
    cout << format("parallelogram_check = {}\n", parallelogram_check);
    cout << format("perpendicular_check = {}\n", perpendicular_check);
    cout << format("length_check     = {}\n", length_check);
    cout << format("final-output     = {}\n", finaloutput);

    // return
    return(0);

}
```

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

- $1 \leq$ flowerbed.length $\leq 2 * 10^4$
- flowerbed[i] is 0 or 1.
- There are no two adjacent flowers in flowerbed.
- $0 \leq n \leq$ flowerbed.length

Code

```cpp
int main(){

    // input- configuration
    vector<int> flowerbed {1, 0, 0, 0, 1};
    int n  {1};


    // going through this
    for(int i = 0; i<flowerbed.size(); ++i){
        if (flowerbed[i] == 0                               &&
            (i == 0 || flowerbed[i-1] == 0)                 &&
            (i == flowerbed.size()-1) || flowerbed[i+1] == 0)
        {
            flowerbed[i] = 1;
            --n;
        }
    }

    // printing final output
    cout << format("Possibility = {}\n", static_cast<bool>(n<=0));

    // return
    return(0);

}
```

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 10-5 of the actual answer will be accepted.

Constraints

- The number of nodes in the tree is in the range [1, 104].
- $-2^{3}1 \leq \text{Node.val} \leq 2^{3}1 - 1$

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root         {new TreeNode(3)};
    root->left         = new TreeNode(9);
    root->right        = new TreeNode(20);
    root->right->left = new TreeNode(15);
    root->right->right = new TreeNode(7);

    // setup


    // running
    // setup
```

```
    vector<double> levelaverages;
    vector<int> levelcount;

    std::function<void(const TreeNode*, int)> foo = [&foo,
                          &levelaverages,
                          &levelcount](
        const TreeNode* root,
        int      currentlevel){

        // checking if valid
        if (root == nullptr) return;

        // increasing level count
        ++currentlevel;

        // adding the sum to the current-value
        while (levelaverages.size() < currentlevel+1)
        {
            // increasing size
            levelaverages.push_back(0);
            levelcount.push_back( 0);
        }

        // adding to sum
        levelaverages[currentlevel] = \
            levelaverages[currentlevel] * \
            (static_cast<double>(levelcount[currentlevel])/static_cast<double>(levelcount[currentlevel]+1)) + \
            (static_cast<double>(root->val) / (levelcount[currentlevel] + 1));

        // increasing count value
        ++levelcount[currentlevel];

        // moving left
        if (root->left)   {foo(root->left, currentlevel);}

```

```
        // moving right
        if (root->right)  {foo(root->right, currentlevel);}

        // returning
        return;

    };


    // going through elements
    foo(root, -1);

    // returning level avaregaes
    cout << format("level-averages = {}\n", levelaverages);


    // return
    return(0);

}
```

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
- $1 \leq k \leq n \leq 10^5$
- $-10^4 \leq nums[i] \leq 10^4$

Code

```cpp
int main(){

    // input- configuration
    vector<int> nums  {1,12,-5,-6,50,3};
    int k             {4};

    // setup
    double windowavg  {std::numeric_limits<double>::min()};

    // calculating the first window average
    for(int i = 0; i<k; ++i) {windowavg += static_cast<double>(nums[i])/static_cast<double>(k);}
    double runningavg {windowavg};

    // going through the rest
    for(int i = k; i < nums.size(); ++i){
        // adding head
        runningavg += static_cast<double>(nums[i])/static_cast<double>(k);

        // substracting tail
        runningavg -= static_cast<double>(nums[i-k])/static_cast<double>(k);

        // finding the bigger-value
        windowavg = windowavg > runningavg ? windowavg : runningavg;
    }

    // printing
    cout << format("largest-average = {}\n", windowavg);

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

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 $\leq$ Node.val $\leq$ 100

Code

```
struct PipeContent
{
    TreeNode*  node;
    unsigned long long      branch_path;
    int         curr_level;
};

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root  {new TreeNode(1)};
    root->left =   new TreeNode(3);
    root->right = new TreeNode(2);
    root->left->left   =   new TreeNode(5);
    root->right->right =  new TreeNode(9);
```

```cpp
    root->left->left->left  =  new TreeNode(6);
    root->right->right->left = new TreeNode(7);


    // setup
    auto   pipe   {std::deque<PipeContent>()};
    auto   levels {std::vector<std::vector<unsigned long long>>()};

    // filling pipe
    pipe.push_back({root, 0, 0});

    // running bfs
    while(pipe.size() != 0)
    {
        // popping front
        auto   front_entry   {pipe.front()}; pipe.pop_front();
        auto&  curr_node     {front_entry.node};
        auto&  branch_path   {front_entry.branch_path};
        auto&  curr_level    {front_entry.curr_level};

        // adding contents to the levels structure
        if(curr_level == levels.size()) {levels.push_back(std::vector<unsigned long long>());}
        levels[curr_level].push_back(branch_path);

        // adding children to pipe
        if (curr_node->left)  {pipe.push_back({curr_node->left, branch_path*2 + 0, curr_level+1});}
        if (curr_node->right) {pipe.push_back({curr_node->right, branch_path*2 + 1, curr_level + 1});}
    }

    // calculating largest size
    auto   finaloutput   {std::numeric_limits<int>::min()};
    for(const  auto&  x: levels){
        // fetching numbers from branch paths
        auto   starting_point    {x[0]};
        auto   ending_point      {x[x.size()-1]};
```

```

        // calculating final output
        finaloutput    =   std::max(finaloutput, static_cast<int>(ending_point - starting_point + 1));
    }

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

    // return
    return(0);

}
```

695. Max Area of Island

You are given an m x 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 = [[0,0,0,0,0,0,0,0]]
    - Output: 0

Constraints

- m == grid.length
- n == grid[i].length
- $1 \leq$ m, n ≤ 50
- grid[i][j] is either 0 or 1.

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto    grid    {std::vector<std::vector<int>>({
            {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}
    })};


    // setup
    auto    curr_island_id    {1};
    auto    global_count      {0};
    auto    pipe              {std::deque<std::vector<int>>()};
    auto    finaloutput       {std::numeric_limits<int>::min()};

    // going through each values
    for(auto row = 0; row < grid.size(); ++row){
        for(auto col = 0; col < grid[0].size(); ++col){

            // continuing if not 1
            if (grid[row][col] != 1) {continue;}

            // clearing pipes and what not
            pipe.clear();
            ++curr_island_id;
            global_count = 0;
            pipe.push_back({row, col});

            // bfs
            while(pipe.size() != 0){

                // popping top
                const   auto    front_entry    {pipe.front()}; pipe.pop_front();
                const   auto&   front_row      {front_entry[0]};
                const   auto&   front_col      {front_entry[1]};

                // checks
                if (front_row < 0 || front_row >= grid.size()) {continue;}
                if (front_col < 0 || front_col >= grid[0].size()) {continue;}
                if(grid[front_row][front_col] != 1) {continue;}
```

```
                // registering
                grid[front_row][front_col] = curr_island_id;
                ++global_count;

                // adding neighbours to pipe
                pipe.push_back({front_row, front_col+1});
                pipe.push_back({front_row-1, front_col});
                pipe.push_back({front_row, front_col-1});
                pipe.push_back({front_row+1, front_col});

            }

            // updating final output
            finaloutput = std::max(finaloutput, global_count);
        }
    }

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

    // return
    return(0);

}
```

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 = nums[1] + nums[2] = 1 + -1 = 0

Constraints:

- $1 \leq \text{nums.length} \leq 10^4$
- $-1000 \leq \text{nums[i]} \leq 1000$

Code

```cpp
int main(){

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

    // setup
    vector<int> numsleft(nums.size(), 0);
    vector<int> numsright(nums.size(), 0);
    int         sumleft    {0};
    int         sumright   {0};
    int         leftindex  {0};
    int         rightindex {0};

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

        // fetching indices
```

```
        leftindex  = i;
        rightindex = nums.size()-1-i;

        // accumulating left and right values
        sumleft    += nums[leftindex];
        sumright   += nums[rightindex];

        if (leftindex < nums.size()-1) {numsleft[leftindex+1] = sumleft;}
        if (rightindex > 0)         {numsright[rightindex-1] = sumright;}
    }

    // finding the mid-element
    int finaloutput {-1};
    for(int i = 0; i<nums.size(); ++i){
        if(numsleft[i] == numsright[i]){
            finaloutput = i;
            break;
        }
    }

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

    // return
    return(0);

}
```

735. Asteroid Collision

We are given an array asteroids of integers representing asteroids in a row. The indices of the asteriod 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

- $2 \leq$ asteroids.length ≤ 104
- $-1000 \leq$ asteroids[i] ≤ 1000
- asteroids[i] != 0

Code

```cpp
int main(){

    // input- configuration
    vector<int> asteroids {5, 10, -5};

    // setup
    vector<int> finaloutput;
    int p;

    // moving through the thing
    while(p<asteroids.size())
    {

        // check current-value
        if (finaloutput.size() == 0 || asteroids[p] > 0){
            finaloutput.push_back(asteroids[p]); ++p;
        }
        else if (asteroids[p]<0){

            auto topvalue = finaloutput.back();
            if (topvalue > std::abs(asteroids[p]))      {;}
            else if (topvalue == std::abs(asteroids[p])) {finaloutput.pop_back();}
            else                                          {finaloutput.pop_back(); finaloutput.push_back(asteroids[p]);}
```

```
            ++p;

            // ensuring the top-value's
            while(finaloutput.size() >= 2 && finaloutput.back() < 0){
                auto negvalue = finaloutput.back(); finaloutput.pop_back();
                topvalue      = finaloutput.back();
                if (topvalue > std::abs(negvalue))      {;}
                else if (topvalue == std::abs(negvalue)) {finaloutput.pop_back();}
                else                                    {finaloutput.pop_back(); finaloutput.push_back(negvalue);}
            }
        }
    }

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


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

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 ith 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

- $1 \leq$ temperatures.length $\leq 10^5$
- $30 \leq$ temperatures[i] ≤ 100

Code

```cpp
int main(){

    // starting timer
    Timer timer;

    // input - configuration
    auto temperatures {vector<int>{73,74,75,71,69,72,76,73}};

    // setup
    vector<vector<int>> tempindexstack;
    auto finaloutput  {vector<int>(temperatures.size(), 0)};

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

        // fetching current-element
        auto& curr = temperatures[i];

        while (tempindexstack.size() != 0 && tempindexstack.back()[0] < curr){

            auto& tempp           = tempindexstack.back()[0];  // fetching top-temperature
            auto& indexx          = tempindexstack.back()[1];  // fetching top-index
            finaloutput[indexx]   = (i - indexx);              // calculating distance
            tempindexstack.pop_back();                         // popping back

        }

        // pushing element-index pair into the stack
        tempindexstack.push_back(vector<int>{curr, i});
    }

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

```
    // return
    return(0);

}
```

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

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto    root        {new TreeNode(90)};
    root->left                  = new TreeNode(69);
    root->left->left            = new TreeNode(49);
    root->left->right           = new TreeNode(89);
    root->left->left->right     = new TreeNode(52);

    // setup
    auto    finaloutput    {std::numeric_limits<int>::max()};
    auto    values         {std::vector<int>()};

    // setting up pipe
```

```
    std::deque<TreeNode*> pipe;
    pipe.push_back(root);
    auto   last_value    {std::numeric_limits<int>::min()};

    // recursive lambda
    std::function<void(const TreeNode*, const int)> InOrderTraversal = [
        &InOrderTraversal,
        &values](const   TreeNode*      root,
                       const    int            parent_value)
    {
        const  auto&  curr_value    {root->val};
        if(root->left)    {InOrderTraversal( root->left, curr_value);}
        values.push_back(curr_value);
        if(root->right)    {InOrderTraversal( root->right, curr_value);}
    };

    // calling the function
    InOrderTraversal(root, std::numeric_limits<int>::min());

    // calculating final output
    finaloutput = std::transform_reduce(
        values.begin() + 1, values.end(),
        values.begin(),
        std::numeric_limits<int>::max(),
        [](int a, int b){ return std::min(a, b); },
        [](int x, int y){ return std::abs(x - y); }
    );

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

    // return
    return(0);

}
```

812. Largest Triangle Area

Given an array of points on the X-Y plane points where 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: 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: points = [[1,0],[0,0],[0,1]]
   - Output: 0.50000

Constraints

- 3 $\leq$ points.length $\leq$ 50
- -50 $\leq x_i, y_i \leq$ 50
- All the given points are unique.

Code

```cpp
int main(){

    // starting timer
    Timer timer;

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

    // setup
    double finaloutput {-1};
    auto calculatearea = [](vector<int> points1, vector<int> points2, vector<int> points3){

        // calculating the area
        int firstcomponent     {points1[0] * (points2[1] - points3[1])};
        int secondcomponent    {points2[0] * (points3[1] - points1[1])};
        int thirdcomponent     {points3[0] * (points1[1] - points2[1])};
        double finaloutput     {static_cast<double>(0.5 * std::abs(firstcomponent + secondcomponent + thirdcomponent))};

        // returning
        return finaloutput;
    };

    // going through the elements
    for(int i = 0; i<points.size(); ++i){
        for(int j = i+1; j<points.size(); ++j){
            for(int k = j+1; k<points.size(); ++k){
                finaloutput = std::max(temp, calculatearea(points[i], points[j], points[k]));
            }
        }
    }

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

```

    // return
    return(0);

}
```

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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto rec1  {vector<int>{4,4,14,7}};
    auto rec2  {vector<int>{4,3,8,8}};

    // checking if all points in rec2 are to the right of xcoordinates
    const auto& top1     {rec1[3]};
    const auto& bottom1  {rec1[1]};
    const auto& left1    {rec1[0]};
    const auto& right1   {rec1[2]};

    const auto& top2     {rec2[3]};
    const auto& bottom2  {rec2[1]};
    const auto& left2    {rec2[0]};
    const auto& right2   {rec2[2]};

    // comparing
```

```
    bool finaloutput  {true};
    if (right1 < left2)       {finaloutput = false;}
    else if(top1 < bottom2)   {finaloutput = false;}
    else if(bottom1 > top2)   {finaloutput = false;}
    else if(right2 < left1)   {finaloutput = false;}

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

    // return
    return(0);

}
```

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 ≤ 1000
- $1 \leq$ m * n ≤ 105
- $-10^9 \leq$ matrix[i][j] $\leq 10^9$

Code

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto    matrix     {std::vector<std::vector<int>>({
        {1,2,3},
        {4,5,6},
        {7,8,9},
        {7,8,9}
    })};

    // setup canvas
    auto    canvas     {std::vector<std::vector<int>>(
        matrix[0].size(),
        std::vector<int>(matrix.size(), 0)
    )};

    // filling the matrix
    for(auto col = 0; col < matrix[0].size(); ++col){
        for(auto row = 0; row < matrix.size(); ++row){
            canvas[col][row]   =   matrix[row][col];
        }
    }

    // return
    return(0);

}
```

883. Projection Area of 3D Shapes

You are given an n x n grid where we place some 1 x 1 x 1 cubes that are axis-aligned with the x, y, and z axes.

Each value v = 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 == grid.length == grid[i].length
- $1 \leq n \leq 50$
- $0 \leq grid[i][j] \leq 50$

Code

```
int main(){

    // starting timer
    Timer timer;

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

    // setup
    auto finaloutput  {0};
    auto xzvector {vector<int>(grid.size(), 0)};
    auto yzvector {vector<int>(grid[0].size(), 0)};

    auto maxheightvalues  {vector<int>{}};
    for(int row = 0; row<grid.size(); ++row){
        for(int col = 0; col<grid[0].size(); ++col){

            // calculating xy-counts
            if (grid[row][col] != 0) {++finaloutput;}

```

```
            // updating xzvector
            xzvector[row]  = std::max(xzvector[row], grid[row][col]);

            // updating yzvector
            yzvector[col]  = std::max(yzvector[col], grid[row][col]);

        }
    }

    // calculating sums
    finaloutput = std::accumulate(xzvector.begin(), xzvector.end(), finaloutput);
    finaloutput = std::accumulate(yzvector.begin(), yzvector.end(), finaloutput);

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

    // return
    return(0);

}
```

892. Surface Area of 3D Shapes

You are given an n x n grid where you have placed some 1 x 1 x 1 cubes. Each value v = 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: grid = [[1,2],[3,4]]
    - Output: 34
2. **Example 2:**
    - Input: grid = [[1,1,1],[1,0,1],[1,1,1]]
    - Output: 32
3. **Example 3:**
    - Input: grid = [[2,2,2],[2,1,2],[2,2,2]]
    - Output: 46

Constraints

- n == grid.length == grid[i].length
- $1 \leq n \leq 50$
- $0 \leq grid[i][j] \leq 50$

Code

```cpp
int main(){

    // starting timer
    Timer timer;

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

    // setup
    auto finaloutput      {0};
    auto neighbourheight  {0};
    auto currheight       {0};

    // constructing lambdas
    auto fCheckBounds = [&grid](const int& row,
                                const int& col){
        if (row < 0 || row >= grid.size())  {return false;}
        if (col < 0 || col >= grid[0].size()) {return false;}
        return true;
```

```
    };
    auto updatefinaloutput = [&finaloutput,
                              &grid](const int& currheight,
                                     const int& neighbourheight){
        if(currheight > neighbourheight){
            finaloutput += currheight - neighbourheight;
        }
    };
    auto fCheckNextCell   = [&grid,
                            &finaloutput,
                            &fCheckBounds,
                            &updatefinaloutput](const int& currheight,
                                                const int& nextrow,
                                                const int& nextcol){
        if (fCheckBounds(nextrow, nextcol)) {
            auto neighbourheight = grid[nextrow][nextcol];
            updatefinaloutput(currheight, neighbourheight);
        }
        else   {finaloutput   += currheight;}
    };

    // goign through the elements
    for(int row = 0; row < grid.size(); ++row){
        for(int col = 0; col < grid[0].size(); ++col){

            // fetching current-height
            currheight = grid[row][col];

            // adding top surfaces
            if (grid[row][col] != 0) {finaloutput += 2;}

            // checking the four directions
            fCheckNextCell(currheight, row, col+1);
            fCheckNextCell(currheight, row-1, col);
            fCheckNextCell(currheight, row, col-1);
```

```
            fCheckNextCell(currheight, row+1, col);

        }
    }

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

    // return
    return(0);

}
```

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 $i \leq k1, k2 \leq j$ with k1 % n == k2 % 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 == nums.length
- $1 \leq n \leq 3 * 10^4$
- $-3 * 10^4 \leq nums[i] \leq 3 * 10^4$

Code

```
int main(){

    // starting timer
    Timer timer;

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

    // setup
    int global_maximum    {nums[0]};
    int global_minimum    {nums[0]};
    int running_max       {0};
    int running_min       {0};
    int total             {0};

    // going through the array
    for(auto x: nums){

        // calculating running maximum and minimum values
        running_max = max(running_max + x, x);
        running_min = min(running_min + x, x);

        // running up the total value
```

```
        total += x;

        // updating global maximum and minumum values
        global_maximum = max(global_maximum, running_max);
        global_minimum = min(global_minimum, running_min);
    }

    // some condition
    if (global_maximum > 0)
        global_maximum = max(global_maximum, total - global_minimum);

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

    // return
    return(0);

}
```

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

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto    root    {new TreeNode(2)};
    root->left =  new TreeNode(2);
    root->right = new TreeNode(2);
    root->left->left  =   new TreeNode(5);
    root->left->right =   new TreeNode(2);

    // setup
    auto    pipe       {std::deque<TreeNode*>()};
    pipe.push_back(root);
    const   auto&   root_value    {root->val};
    auto    final_output   {true};

    // bfs
    while   (pipe.size()!=0)
    {
        // popping front-value
        const   auto&   curr_root     {pipe.front()};
        pipe.pop_front();
        const   auto&   curr_value    {curr_root->val};

        // checking if current-value is different from root-value
        if (curr_value != root_value) {final_output = false; break;}

        // adding children to pipe
        if(curr_root->left)         {pipe.push_back(curr_root->left);}
        if(curr_root->right)        {pipe.push_back(curr_root->right);}
    }

```

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

973. K Closest Points to Origin

Given an array of points where points[i] = [xi, yi] 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: 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: 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

- $1 \leq k \leq \text{points.length} \leq 10^4$
- $-10^4 \leq xi, yi \leq 10^4$

Code

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    vector<vector<int>> points = {
        {3,3},
        {5,-1},
        {-2,4}
    };
    auto k {2};

    // setup
    auto distance = [](auto argx){return std::sqrt(argx[0]*argx[0] + argx[1]*argx[1]);};

    // building heap
    vector<std::pair<int, double>> indexToDistance;

    // Basis on which the heap is built
    auto heap_comparator  = [](const auto& pair_a, const auto& pair_b){
        return pair_a.second < pair_b.second;
    };

    // making heap
```

```
    std::make_heap(indexToDistance.begin(), indexToDistance.end(), heap_comparator);

    //  lambda-function for pushing to heap
    auto pushHeap = [&indexToDistance, &heap_comparator, &k](auto curr_pair){

        indexToDistance.push_back(curr_pair);
        std::push_heap(indexToDistance.begin(),indexToDistance.end(),heap_comparator);

        if (indexToDistance.size() > k){
            std::pop_heap(indexToDistance.begin(),indexToDistance.end(),heap_comparator);
            indexToDistance.pop_back();
        }
    };

    // going through the elements
    for(int i = 0; i<points.size(); ++i){

        // making current-entry
        auto curr  {std::pair{i, distance(points[i])}};

        // adding element to the heap
        pushHeap(curr);
    }

    // building the final output
    auto finaloutput  {vector<vector<int>>{}};
    for(int i = 0; i<indexToDistance.size(); ++i)
        finaloutput.push_back(points[indexToDistance[i].first]);

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

    // return
    return(0);
```

```
}
```

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

```
string fCompareStrings(const string& runningstring,
                      const string& finaloutput)
{

    // copying values
    auto a {runningstring};
    auto b {finaloutput};

    // init
    if (finaloutput.size() == 0) {return runningstring;}

    // going through characters one after the other
    auto numminchars  = std::min(a.size(), b.size());

    for(int i = 0; i<numminchars; ++i){
        if (a[i] < b[i])         {return a;}
        else if (a[i] > b[i])    {return b;}
        else                     {continue;}
    }

    // if we're here, it means that the first minchars are the same
    if (a.size() < b.size())     {return a;}
    else                         {return b;}

}
```

```cpp
void foo(TreeNode* root,
         string    runningstring,
         string&   finaloutput){

    // sending it back
    if (root == nullptr) return;

    // adding current character to string
    runningstring = std::string(1, static_cast<char>(root->val + 'a')) + runningstring;

    // calling the function on the two children
    if (root->left == nullptr && root->right == nullptr){
        finaloutput = fCompareStrings(runningstring, finaloutput);
    }
    else{
        // calling the children
        foo(root->left,   runningstring, finaloutput);
        foo(root->right,  runningstring, finaloutput);
    }

    // returning
    return;
}

int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto root          {new TreeNode(0)};
    root->left         = new TreeNode(1);
    root->right        = new TreeNode(2);

    root->left->left   = new TreeNode(3);
```

```
    root->left->right = new TreeNode(4);

    root->right->left = new TreeNode(3);
    root->right->right = new TreeNode(4);


    // setup
    string finaloutput = "";

    // calling the function
    foo(root, "", finaloutput);

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

    // return
    return(0);

}
```

1004. Max Consecutives 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 = [1,1,1,0,0,0,1,1,1,1,0], k = 2
    - Output: 6
    - Explanation: [1,1,1,0,0,1,1,1,1,1,1] Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.
2. **Example 2:**
    - Input: nums = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], k = 3
    - Output: 10
    - Explanation: [0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1] 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

```cpp
int main(){

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

    // seutp
    int zerocounter {0};
    int p1          {0};
    int running     {-1};
    int maxlength  {-1};

    // running
    for (int i = 0; i<nums.size(); ++i){
        // incrementing zeros
        if (nums[i] == 0) {++zerocounter;}

        // printing
        cout << format("substring = ");
        fPrintVector(std::vector<int>(nums.begin() + p1, nums.begin() + i + 1));

        // moving tail
        if (zerocounter == k+1){
            while(nums[p1] != 0)  {++p1;}
            ++p1;
            --zerocounter;
        }

        // assessing lengths
        running    = i - p1+1;
        maxlength  = running > maxlength ? running : maxlength;
    }

```

```
    // printing
    cout << format("max-length = {}\n", maxlength);

    // return
    return(0);

}
```

1037. Valid Boomerang

Given an array points where points[i] = [xi, yi] 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
- $0 \leq xi, yi \leq 100$

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    vector<vector<int>> points {
        {82,23},
        {98,88},
        {94,72}
    };

    // setup
    vector<int> vecAB {points[1][0] - points[0][0], points[1][1] - points[0][1]};
    vector<int> vecAC {points[2][0] - points[0][0], points[2][1] - points[0][1]};

    // lambda
    auto fCalculateInnerProduct = [](const vector<int>& vecAB,
                                     const vector<int>& vecAC){
        auto modAB {std::sqrt(vecAB[0]*vecAB[0] + vecAB[1]*vecAB[1])};
        auto modAC {std::sqrt(vecAC[0]*vecAC[0] + vecAC[1]*vecAC[1])};

        if (modAB == 0 || modAC == 0) {return static_cast<double>(1);}

        auto dotproduct {vecAB[0] * vecAC[0] + vecAB[1] * vecAC[1]};
        return dotproduct/(modAB * modAC);
    };

    // calculating innerproduct
    auto innerproduct = fCalculateInnerProduct(vecAB, vecAC);

    // checking
    auto finaloutput  {true};
```

```
    if (std::abs(1 - std::abs(innerproduct)) < 1e-7) {finaloutput = false;}

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

    // return
    return(0);

}
```

1091. Shortest Path in Binary Matrix

Given an n x 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 == grid.length
- n == grid[i].length
- $1 \leq n \leq 100$
- grid[i][j] is 0 or 1

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto   grid   {std::vector<std::vector<int>>({
        {0,0,0},
        {1,1,0},
        {1,1,0}
    })} ;

    // setting up shortest paths
    auto   spath  {std::vector<std::vector<int>>(
        grid.size(),
        std::vector<int>(grid[0].size(), std::numeric_limits<int>::max())
    )};

    // running bfs
    auto   pipe   {std::deque<std::vector<int>>()};
    pipe.emplace_back(std::vector<int>({0,0,1}));
```

```
// lambda for checking validity
auto   fCheckValidity =  [&grid,&spath](auto  row,auto   col){
    if (row < 0 || row >= grid.size())  {return false;}
    if (col < 0 || col >= grid[0].size()) {return false;}
    if (grid[row][col] == 1)            {return false;}
    return true;
};

// running bfs
while (pipe.size() != 0 && fCheckValidity(0,0))
{

    // popping vector from the front
    const  auto   front_entry   {pipe.front()}; pipe.pop_front();
    const  auto&  row    {front_entry[0]};
    const  auto&  col    {front_entry[1]};
    const  auto&  pathlength {front_entry[2]};

    // checking the spath
    if (pathlength >= spath[row][col])  {continue;}
    else                                {spath[row][col] = pathlength;}

    // adding the neighbours to the path
    if(fCheckValidity(row, col+1))      {pipe.push_back(std::vector<int>({row, col+1, pathlength + 1}));}
    if(fCheckValidity(row-1, col+1))    {pipe.push_back(std::vector<int>({row-1, col+1, pathlength + 1}));}
    if(fCheckValidity(row-1, col))      {pipe.push_back(std::vector<int>({row-1, col, pathlength + 1}));}
    if(fCheckValidity(row-1, col-1))    {pipe.push_back(std::vector<int>({row-1, col-1, pathlength + 1}));}

    if(fCheckValidity(row, col-1))      {pipe.push_back(std::vector<int>({row, col-1, pathlength + 1}));}
    if(fCheckValidity(row+1, col-1))    {pipe.push_back(std::vector<int>({row+1, col-1, pathlength + 1}));}
    if(fCheckValidity(row+1, col))      {pipe.push_back(std::vector<int>({row+1, col, pathlength + 1}));}
    if(fCheckValidity(row+1, col+1))    {pipe.push_back(std::vector<int>({row+1, col+1, pathlength + 1}));}
}
```

```

    // printing finaloutput
    auto   finaloutput   {spath[spath.size()-1][spath[0].size()-1]};
    finaloutput = finaloutput == std::numeric_limits<int>::max() ? -1 : finaloutput;

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

    // return
    return(0);

}
```

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

- $1 \leq$ text1.length, text2.length ≤ 1000
- text1 and text2 consist of only lowercase English characters.

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto text1        {string("bsbininm")};
    auto text2        {string("jmjkbkjkv")};

    // setup
    auto entryvalue   {-1};
    vector<vector<int>> dptable;

    // creating dp-table
    for(int i = 0; i<text1.size()+1; ++i)
        dptable.push_back(vector<int>(text2.size()+1, 0));

    // filling dptable
    for(int row = text1.size()-1; row>=0; --row){
        for(int col = text2.size()-1; col >= 0; --col){

            // checking if current-values are the same
            if (text1[row] == text2[col]) {entryvalue = 1 + dptable[row+1][col+1];}
            else                         {entryvalue = std::max(dptable[row+1][col], dptable[row][col+1]);}

```

```
            // storing to dptable
            dptable[row][col] = entryvalue;

        }
    }

    // printing the dptable
    cout << format("final-output = {}\n", dptable[0][0]);

    // printing matrix
    fPrintMatrix(dptable);

    // return
    return(0);

}
```

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

- $1 \leq$ arr.length ≤ 1000
- $-1000 \leq$ arr[i] ≤ 1000

Code

```
int main(){

    // input- configuration
    vector<int> arr {-3,0,1,-3,1,1,1,-3,10,0};

    // building histogram
    unordered_map<int, int> histogram;
    for(const auto& x: arr){
        if (histogram.find(x) == histogram.end()) {histogram[x] = 1;}
        else                                       {++histogram[x];}
    }

    // building a set out of the histogram
    std::set<int> uniquecounts;
    bool finaloutput  {true};
    int uniquecounts_prevsize = 0;
    for(const auto& x: histogram){
        uniquecounts.insert(x.second);
        if (uniquecounts.size() == uniquecounts_prevsize) {finaloutput = false; break;}
        uniquecounts_prevsize = uniquecounts.size();
    }

    // printing
    cout << format("histogram.size() = {}\n", histogram.size());
    cout << format("finaloutput = {}\n", finaloutput);
    cout << format("uniquecounts = "); for(const auto& x: uniquecounts) {cout << x << ",";}


    // return
    return(0);

}
```

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

- $2 \leq$ coordinates.length ≤ 1000
- coordinates[i].length == 2
- $-10^4 \leq$ coordinates[i][0], coordinates[i][1] $\leq 10^4$
- coordinates contains no duplicate point.

Code

```
int main(){

    // starting timer
    Timer timer;

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

    // setup
    vector<vector<int>> geometric_vectors(coordinates.size()-1);
    std::transform(coordinates.begin()+1,
                   coordinates.end(),
                   geometric_vectors.begin(),
                   [&coordinates](auto argx){return vector<int>{argx[0] - coordinates[0][0], argx[1] - coordinates[0][1]};}
                    );

    // calculating slope
    vector<double> geometric_slopes(geometric_vectors.size());
    std::transform(geometric_vectors.begin(),
                   geometric_vectors.end(),
                   geometric_slopes.begin(),
                   [](auto argx){
                    if (argx[0] == 0) {return std::numeric_limits<double>::max();}
                    else              {return static_cast<double>(argx[1])/static_cast<double>(argx[0]);}
                   });

```

```
    // checking if all elements are same
    auto areallsame = std::adjacent_find(geometric_slopes.begin(),
                                         geometric_slopes.end(),
                                         std::not_equal_to<>()) == geometric_slopes.end();

    cout << format("coordinates = {}\n", coordinates);
    cout << format("vectors   = {}\n", geometric_vectors);
    cout << format("geometric_slops = {}\n", geometric_slopes);
    cout << format("final-output = {}\n", areallsame);

    // return
    return(0);

}
```

1266. Minimum Time Visiting All Points

On a 2D plane, there are n points with integer coordinates points[i] = [xi, yi]. 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: points = [[1,1],[3,4],[-1,0]]
   - Output: 7
   - Explanation: One optimal path is [1,1] → [2,2] → [3,3] → [3,4] → [2,3] → [1,2] → [0,1] → [-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
- $-1000 \leq$ points[i][0], points[i][1] ≤ 1000

Code

```cpp
int main(){

    // starting timer
    Timer timer;

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

    // setup
    auto fCalculateDistance = [](const vector<int>& coordinate_A,
                                 const vector<int>& coordinate_B){

        auto xdif         {std::abs(coordinate_A[0] - coordinate_B[0])};
```

```
        auto ydif         {std::abs(coordinate_A[1] - coordinate_B[1])};
        auto common       {std::min(xdif, ydif)};
        auto distleft     {std::max(xdif, ydif) - std::min(xdif, ydif)};
        return distleft + common;
    };

    // calculating distances
    auto finaloutput  {std::inner_product(points.begin(),
                                          points.end()-1,
                                          points.begin()+1,
                                          0,
                                          std::plus<int>(),
                                          fCalculateDistance)};

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

    // return
    return(0);

}
```

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

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto   original          {new TreeNode(7)};
    original->left           =   new TreeNode(4);
    original->right          =   new TreeNode(3);
    original->right->left    =   new TreeNode(6);
    original->right->right   =   new TreeNode(19);

    auto   cloned          {new TreeNode(7)};
    cloned->left           =   new TreeNode(4);
    cloned->right          =   new TreeNode(3);
    cloned->right->left    =   new TreeNode(6);
    cloned->right->right   =   new TreeNode(19);

    auto   target     {original->right};

    // setup
    auto   finaloutput    {static_cast<TreeNode*>(nullptr)};
    auto   found      {false};
```

```cpp

    // setting up lambda
    std::function<void(const TreeNode*, TreeNode*)>
    seachLambda = [&seachLambda,
                   &target,
                   &found,
                   &finaloutput](
            const  TreeNode*  curr_root,
            TreeNode*  curr_root_parallel
        )
    {
        // quitting if already found
        if(found == true)            {return;}

        // checking if current-value is target
        if(curr_root == target)      {finaloutput = curr_root_parallel; found = true; return;}

        // passing control to the children
        if (curr_root->left != nullptr) {seachLambda(curr_root->left, curr_root_parallel->left);}
        if (curr_root->right != nullptr) {seachLambda(curr_root->right, curr_root_parallel->right);}

        // returning
        return;
    };

    // call the function
    seachLambda(original, cloned);

    // return
    return(0);

}
```

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 ith 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 ith 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,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 == candies.length
- $2 \leq n \leq 100$
- $1 \leq$ candies[i] ≤ 100
- $1 \leq$ extraCandies ≤ 50

Code

```
int main(){

    // input- configuration
    vector<int> candies {2,3,5,1,3};
    int extraCandies  {3};

    // setup
    auto iter = std::max_element(candies.begin(), candies.end()); // finding max value
    vector<bool> finaloutput;

    // going through all values
    for(int i = 0; i<candies.size(); ++i){
        if ((candies[i]+extraCandies) >= *iter) {finaloutput.push_back(true);}
        else                                    {finaloutput.push_back(false);}
    }

```

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

    // return
    return(0);

}
```

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

```cpp
int main(){

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

    // setup
    int zerocounter   {0};
    int p1            {0};
    int running       {0};
    int maxlength     {0};

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

        // incrementing zero-count
        if(nums[i] == 0)  {++zerocounter;}

        // updating length
        if (zerocounter <= 1){
            running    = i-p1;
            maxlength  = running > maxlength ? running : maxlength;
        }
        else {
            // moving tail
            while(nums[p1] != 0)  {++p1;}
```

```
            ++p1; --zerocounter;

            // updating length
            running    = i - p1;
            maxlength  = running > maxlength ? running : maxlength;
        }
    }

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


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

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 $\Longrightarrow$ 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 $\Longrightarrow$ bbcbaa (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” -¿ ”acb”
     - Apply Operation 1: ”acb” -¿ ”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

- $1 \leq$ word1.length, word2.length ≤ 105
- word1 and word2 contain only lowercase English letters.

Code

```
int main(){

    // input- configuration
    string word1  {"cabbba"};
    string word2  {"abbccc"};

    // setup
    bool finaloutput  {false};

```

```
    // building histogram
    unordered_map<char, int> histogram1, histogram2;
    for(const auto x: word1){
        if(histogram1.find(x) == histogram1.end()) {histogram1[x] = 1;}
        else                                       {++histogram1[x];}
    }
    for(const auto x: word2){
        if(histogram2.find(x) == histogram2.end()) {histogram2[x] = 1;}
        else                                       {++histogram2[x];}
    }

    // checking if one can be obtained from the other just using shuffling
    if (histogram1 == histogram2) {
        finaloutput = true;
        cout << format("(same histogram) finaloutput = {}\n", finaloutput);
        return 0;
    }

    // if number of keys are different
    if (histogram1.size() != histogram2.size()){
        finaloutput = false;
        cout << format("(different number of keys) finaloutput = {}\n", finaloutput);
        return 0;
    }


    // checking if keys match
    bool keysmatch {true};
    {
        auto it_histogram1 = histogram1.begin();
        auto it_histogram2 = histogram2.begin();
        vector<char> keys1, keys2;
        while(it_histogram1 != histogram1.end()){
            keys1.push_back(it_histogram1->first);
            keys2.push_back(it_histogram2->first);
```

```
            ++it_histogram1;
            ++it_histogram2;
        }
        std::sort(keys1.begin(), keys1.end());
        std::sort(keys2.begin(), keys2.end());
        if (keys1 == keys2) {keysmatch = true;}
        else                {keysmatch = false;}
    }

    // checking if counts match
    bool countsmatch  {false};
    {
        vector<int> counts1, counts2;
        for(const auto& x: histogram1) {counts1.push_back(x.second);}
        for(const auto& x: histogram2) {counts2.push_back(x.second);}
        sort(counts1.begin(), counts1.end());
        sort(counts2.begin(), counts2.end());
        if (counts1 == counts2) {countsmatch = true;}
    }

    // producing the final output
    if (keysmatch && countsmatch) {finaloutput = true;}
    else                          {finaloutput = false;}

    // return
    return(0);

}
```

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

```
int main(){

    // input- configuration
    vector<int> nums
        {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,
    int k              {77};

    // setup
    unordered_map<int, int> histogram;
    int count  {0};

    // go through list
    for(int i = 0; i<nums.size(); ++i){

        // fetching current value
        int curr   {nums[i]};
        int comp   {k - curr};

        // updating histogram
        if (histogram.find(comp) == histogram.end() || histogram[comp] == 0)
        {
            // in case where complement hasn't even been entered
            if (histogram.find(curr) != histogram.end()) {++histogram[curr];}
            else                                         {histogram[curr] = 1;}
        }
        else if(histogram[comp] > 0)
        {
```

```
            --histogram[comp];
            ++count;
        }
    }

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

    // return
    return(0);

}
```

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

- $1 \leq$ word1.length, word2.length ≤ 100
- word1 and word2 consist of lowercase English letters.

Code

```
int main(){

    // input- configuration
    string word1   {"ab"};
    string word2   {"pqrs"};

    // setup
    int p1 {0};
    int p2 {0};
    string finaloutput;

    // going through
    while(p1<word1.size() || p2 < word2.size()){
        // pushing to final output
        if(p1<word1.size())  {finaloutput += word1[p1++];}
        if (p2<word2.size()) {finaloutput += word2[p2++];}
```

```
    }

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

    // return
    return(0);

}
```

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

- $1 \leq n \leq 2 * 10^5$
- $0 \leq$ edges.length $\leq 2 * 10^5$
- edges[i].length == 2
- $0 \leq$ ui, vi $\leq$ n - 1
- ui $\neq$ vi
- $0 \leq$ source, destination $\leq$ n - 1
- There are no duplicate edges.
- There are no self edges.

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    const  auto   n   {3};
    // const  auto    edges   {std::vector<std::vector<int>>{
    const  auto    edges   {std::vector<std::vector<int>>{
        {0,1},
        {1,2},
        {2,0}
    }};
```

```
    const  auto   source     {0};
    const  auto   destination {2};

    // building hashmap
    unordered_map<int, vector<int>> hmap;

    // going through the inputs
    for(const  auto&  x: edges){
        // adding to hashmap
        if (hmap.find(x[0]) == hmap.end()) hmap[x[0]] = std::vector<int>({x[1]});
        else                             hmap[x[0]].push_back(x[1]);
        if (hmap.find(x[1]) == hmap.end()) hmap[x[1]] = std::vector<int>({x[0]});
        else                             hmap[x[1]].push_back(x[0]);
    }

    // setting up for bfs launch
    auto   visited    {std::vector<bool>(n, false)};
    auto   found      {false};
    auto   pipe       {std::deque<int>()};
    pipe.push_back(source);

    // peforming bfs launch
    while(pipe.size() != 0 && found == false){

        // popping from the front
        auto   front_object   {pipe.front()}; pipe.pop_front();

        // printing
        cout << format("curr = {} | destination = {}\n", front_object, destination);

        // checking if current point is destination and updating found
        if (front_object == destination) {found = true; break;}

        // checking if the current point has been visited already. if yes,c ontinue
        if (visited[front_object] == true) {continue;}
```

```
        else                                {visited[front_object] = true;}

        // else add the neighbours to the pipe
        for(const  auto&  x: hmap[front_object])
            pipe.push_back(x);
    }

    // printing hashmap
    PRINTLINE
    cout << format("final-output = {}\n", found);




    // return
    return(0);

}
```

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 ith 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

- $1 \leq \text{bombs.length} \leq 100$
- $\text{bombs[i].length} == 3$
- $1 \leq x_i, y_i, r_i \leq 10^5$

Code

```
int main(){

    // starting timer
    Timer timer;

    // input- configuration
```

```cpp
    vector<vector<int>> bombs {
        {1,2,3},
        {2,3,1},
        {3,4,2},
        {4,5,3},
        {5,6,4}
    };

    // setup
    unordered_map<int, vector<int>> treemap;

    auto checkOverlap = [&treemap](const vector<int>& source,
                                   const vector<int>& destination,
                                   const int& source_index,
                                   const int& destination_index){

        auto relative_vector     {vector<double>{static_cast<double>(source[0] - destination[0]),
                                                 static_cast<double>(source[1] - destination[1])}};
        auto dist_between_centers {std::sqrt(std::inner_product(relative_vector.begin(),
                                                                relative_vector.end(),
                                                                relative_vector.begin(),
                                                                0.00))};
        // dist_between_centers = std::sqrt(dist_between_centers);

        // adding to hashmap
        if (dist_between_centers <= source[2]){
            if (treemap.find(source_index) == treemap.end())  {treemap[source_index] = vector<int>{destination_index};}
            else                                              {treemap[source_index].push_back(destination_index);}
        }
        if(dist_between_centers <= destination[2]){
            if (treemap.find(destination_index) == treemap.end()) {treemap[destination_index] = vector<int>{source_index};}
            else                                              {treemap[destination_index].push_back(source_index);}
        }

        return;
```

```
    };

    // going through the combinations
    for(int i = 0; i<bombs.size(); ++i){
        for(int j = i+1; j<bombs.size(); ++j){
            checkOverlap(bombs[i], bombs[j], i, j);
        }
    }

    // building dfs
    int finaloutput  {-1};
    std::function<void(vector<int>&, int, int&)>
    dfs   = [&dfs,
            &treemap](vector<int>& pathsofar,
                    int            currindex,
                    int&           finaloutput)
    {


        // sending it back if its already visited
        if (std::find(pathsofar.begin(), pathsofar.end(), currindex) != pathsofar.end()) {return;}

        // adding current-element to path
        pathsofar.push_back(currindex);
        finaloutput = finaloutput > (int)pathsofar.size() ? finaloutput : (int)pathsofar.size();

        // movign to the rest
        for(const auto& x: treemap[currindex]) {dfs(pathsofar, x, finaloutput);}

        // returning
        return;

    };
```

```cpp

    // calling the function
    for(int i = 0; i<bombs.size(); ++i) {
        vector<int> temp;
        dfs(temp, i, finaloutput);
    }

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

    // return
    return(0);

}
```

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

- $1 \leq$ nums1.length, nums2.length ≤ 1000
- $-1000 \leq$ nums1[i], nums2[i] ≤ 1000

Code

```
int main(){

    // input- configuration
    vector<int> nums1 {1,2,3};
    vector<int> nums2 {2,4,6};

    // setup
    std::set<int>     set1(nums1.begin(), nums1.end());
    std::set<int>     set2(nums2.begin(), nums2.end());
    vector<int>       firstinput;
    vector<int>       secondinput;
    vector<vector<int>> finaloutput;

    // filling first entry
    for(const auto& x: set1){
        if(set2.find(x) == set2.end()) {firstinput.push_back(x);}
    }

```

```
    // filling second entry
    for(const auto& x: set2){
        if (set1.find(x) == set1.end()) {secondinput.push_back(x);}
    }

    // pushing to the final output
    finaloutput.push_back(firstinput);
    finaloutput.push_back(secondinput);

    // printing
    cout << format("first input = "); fPrintVector(firstinput);
    cout << format("second input = ");fPrintVector(secondinput);

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

2249. Count Lattice Points Inside a Circle

Given a 2D integer array circles where circles[i] = [xi, yi, ri] represents the center (xi, yi) and radius ri of the ith 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

- $1 \leq$ circles.length ≤ 200
- circles[i].length $== 3$
- $1 \leq$ xi, yi ≤ 100
- $1 \leq$ ri $\leq$ min(xi, yi)

Code

```
int main(){

    // starting timer
    Timer timer;

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

    // setup

    auto checkIfWithin = [](auto x_center,
                            auto y_center,
                            auto radius,
                            auto x,
                            auto y){

        auto relative_coordinate {vector<int>{x - x_center, y - y_center}};
        auto distance            {std::sqrt(std::pow(relative_coordinate[0],2)+
```

```
                                            std::pow(relative_coordinate[1],2))};
        return distance <= radius;
    };

    // going through the points
    auto xmin  {std::numeric_limits<int>::max()};
    auto xmax  {std::numeric_limits<int>::min()};
    auto ymin  {std::numeric_limits<int>::max()};
    auto ymax  {std::numeric_limits<int>::min()};

    // going through the circles
    for(const auto& x: circles){
        xmin    = std::min(xmin, x[0]-x[2]);
        xmax    = std::max(xmax, x[0]+x[2]);
        ymin    = std::min(ymin, x[1]-x[2]);
        ymax    = std::max(ymax, x[1]+x[2]);
    }

    // searching through the grid
    auto finaloutput  {0};
    for(int i = xmin; i<=xmax; ++i){
        for(int j = ymin; j<=ymax; ++j){

            // pritning the points
            for(const auto& circle: circles){
                auto temp00    {checkIfWithin(circle[0],
                                            circle[1],
                                            circle[2],
                                            i,
                                            j)};
                if (temp00) {++finaloutput; break;}
            }

        }
    }
```

```

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

    // return
    return(0);

}
```

2352. Equal Row And Column Pairs

Given a 0-indexed n x 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 == grid.length == grid[i].length

- $1 \leq n \leq 200$
- $1 \leq grid[i][j] \leq 10^5$

Code

```
int main(){

    // input- configuration
    vector<vector<int>> grid;
    grid.push_back(vector<int>({3, 2, 1}));
    grid.push_back(vector<int>({1, 7, 6}));
    grid.push_back(vector<int>({2, 7, 7}));

    // setup
    int nrows  {static_cast<int>(grid.size())};
    int ncols  {static_cast<int>(grid[0].size())};
    int nelems {nrows};
    int counts {0};

    // going through the values

    for(int i = 0; i<nrows; ++i){
        for(int j = 0; j<ncols; ++j){

            // comparing row and column
            bool same  {true};
            for(int k = 0; k<nelems; ++k){
                if(grid[i][k] != grid[k][j]) {same = false;}
            }

            // increasing count
            if (same)  {++counts;}
```

```
        }
    }

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

    // return
    return(0);

}
```

2390. Removing Starts 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

- $1 \leq \text{s.length} \leq 10^5$
- s consists of lowercase English letters and stars *.
- The operation above can be performed on s.

Code

```
int main(){

    // input- configuration
    string s   {"leet**cod*e"};

    // going through the inputs
    std::string finalOutput;
    for(auto x: s){
        if (x == '*') finalOutput.pop_back();
        else finalOutput.push_back(x);
    }

    // returning output
    cout << format("final-output = {}\n", finalOutput);
```

```

    // return
    return(0);

}
```

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

```cpp
int main(){

    // starting timer
    Timer timer;

    // input- configuration
    auto n {4};

    // setup
    auto finaloutput      {0};
    if (n == 1)           {finaloutput = 0;}
    else if (n%2 == 0)    {finaloutput = n/2;}
    else                  {finaloutput = n;}

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

    // return
    return(0);

}
```