**Batch : T7**

**Practical No. : 1**

**Title of Assignment : Sorting Algorithm**

**Student Name: Om Kulkarni**

**Student PRN: 22510034**

**1)**

**Problem Statement:**

You are given two sorted array, A and B, where A has a large enough buffer at the end to hold B. Write a method to merge B into A in sorted order.

1. Algorithm/Pseudocode :

Initialize Pointers:

Set a pointer i at the end of the elements in array A (excluding the buffer).

Set a pointer j at the end of the elements in array B.

Set a pointer k at the end of the buffer in array A.

Merge Process:

Compare the elements pointed by i and j.

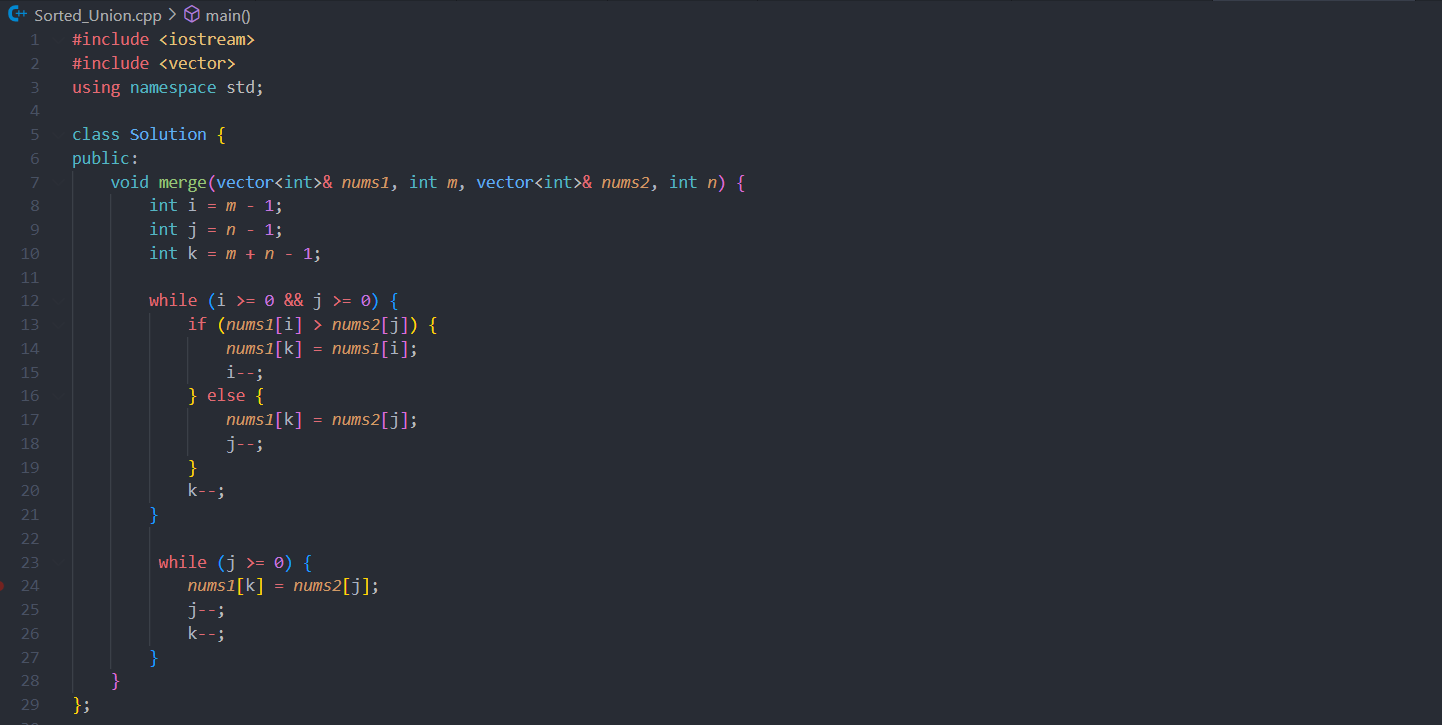
Place the larger element at the position pointed by k and move the pointer (i or j) accordingly.

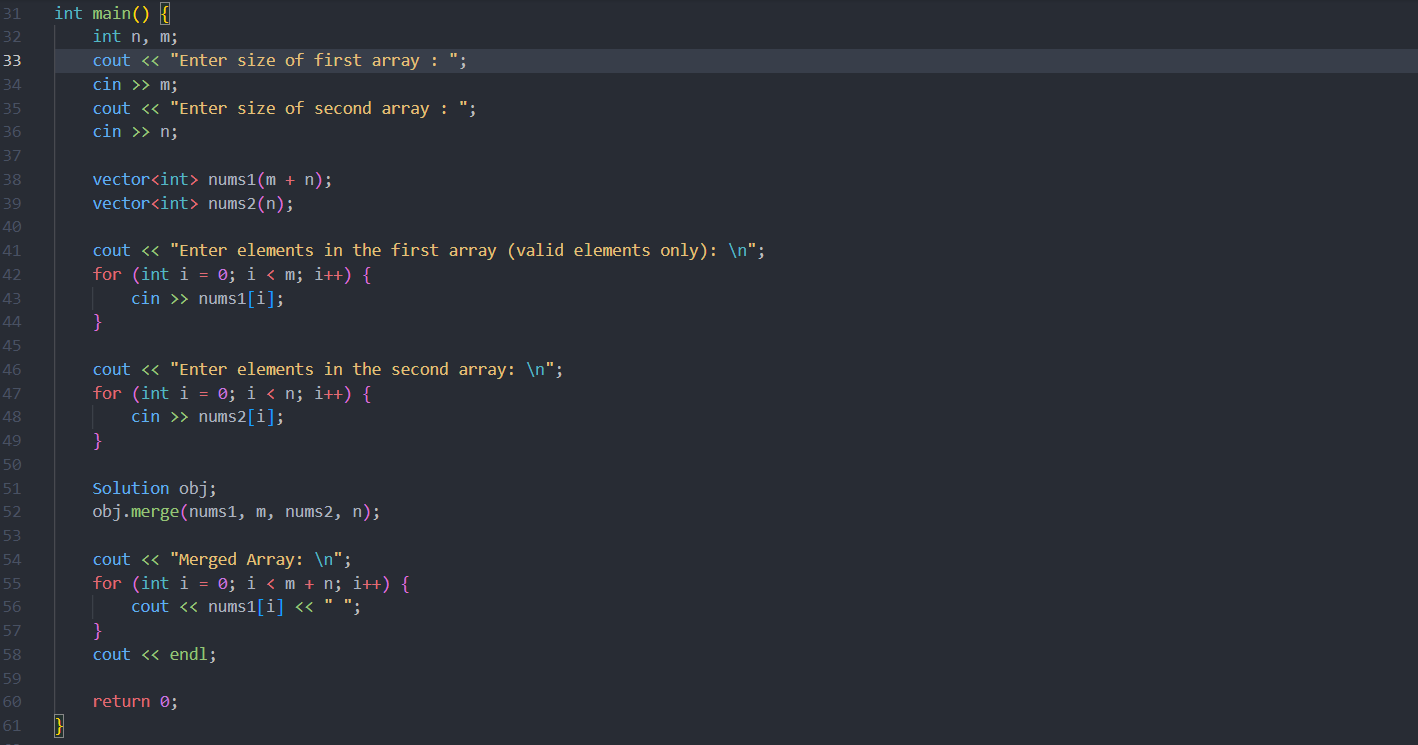
Decrease the pointer k after placing the element.

Handle Remaining Elements:

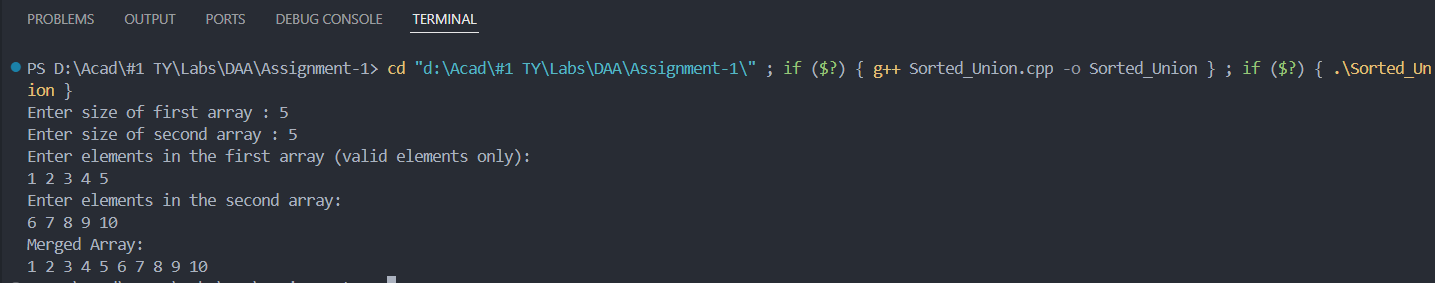
If there are remaining elements in array B, copy them into array A

1. Program Code





1. Output with verity of test cases



1. Analysis in terms of complexity wherever applicable.

**Merging Process:** The while loops iterate over the elements of both arrays. Since each element is compared and placed exactly once, the time complexity is O(m + n), where m is the number of elements in nums1 and n is the number of elements in nums2.

**Problem Statement:**

Write a method to sort an array of string so that all the anagrams are next to each other.

1. Algorithm/Pseudocode

Initialization:

Create an empty map to group anagrams. The keys will be the sorted strings, and the values will be lists of original strings.

Grouping Anagrams:

For each string in the input array:

Sort the string to create a key.

Insert the original string into the map under the sorted string key.

Collecting Results:

Create an empty list to store the results.

For each entry in the map:

Append all strings in the current list to the result list.

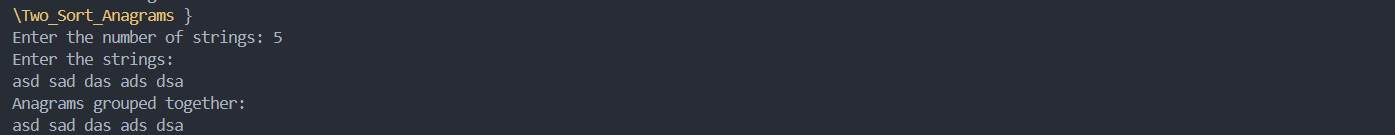
Return the Result:

Return the result list containing all the anagrams grouped together.

2.Program Code



3.Output with verity of test cases



4.Analysis in terms of complexity wherever possible

Time Complexity**:** O(n \* k log k)

We iterate over each string in the input array (O(n) where n is the number of strings).

For each string, we sort its characters (O(k log k) where k is the maximum length of a string).

Insertion and lookup in a hash map are on average O(1), so inserting each string into the map is O(1) per string.

Time Complexity: O(n)

We iterate over each entry in the map and each string in the map values to append them to the result list.

Since there are n strings in total, this operation takes O(n) time.

Time Complexity: O(n \* k log k)

The dominant term is the time taken to sort each string, resulting in O(n \* k log k) overall time complexity.

**Problem Statement:**

Q) Given a sorted array of *n* integers that has been rotated an unknown number of times, write code to find an element in the array. You may assume that the array was originally sorted in increasing order.

EXAMPLE

Input: find 5 in {15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14}

Output: 8 (the index of 5 in the array)

1. Algorithm/Pseudocode

Steps

Initialization:

Initialize left to 0 and right to the last index of the array.

Binary Search:

While left is less than or equal to right:

Calculate the middle index mid.

If the element at mid is the target, return mid.

Determine which half of the array is sorted:

If the left half is sorted (arr[left] <= arr[mid]):

Check if the target is within this range (arr[left] <= target < arr[mid]). If yes, update right to mid - 1.

Otherwise, update left to mid + 1.

If the right half is sorted (arr[mid] <= arr[right]):

Check if the target is within this range (arr[mid] < target <= arr[right]). If yes, update left to mid + 1.

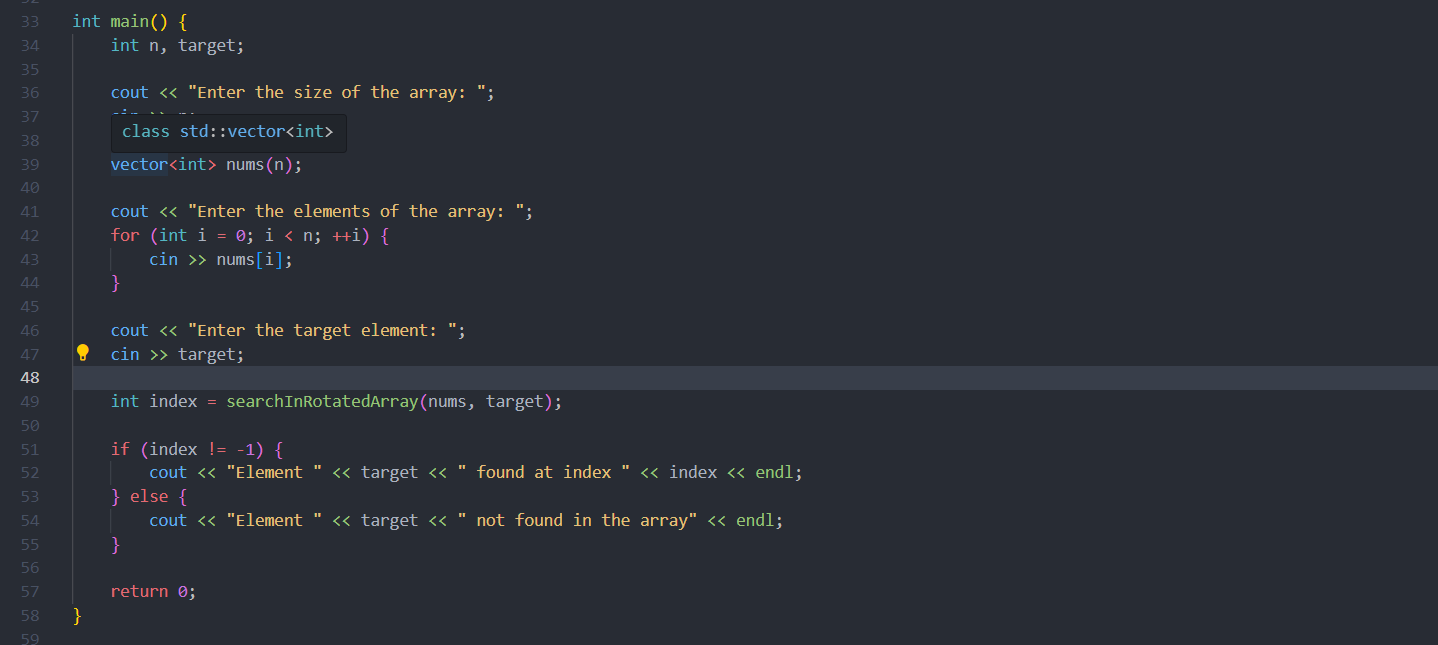
Otherwise, update right to mid - 1.

Target Not Found:

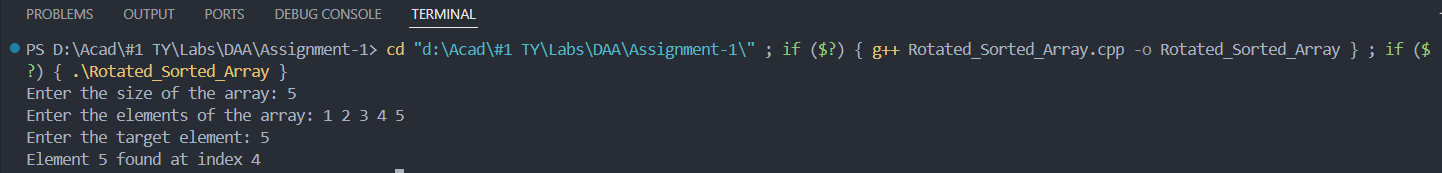
If the loop exits, the target is not in the array. Return -1.

1. Program Code





3. Output with verity of test cases



4. Analysis in terms of complexity wherever applicable.

Time Complexity: O(log n)

The binary search approach ensures that the search range is halved in each iteration, leading to a logarithmic time complexity.

**Problem Statement:**

Imagine you have a 20GB file with one string per line. Explain how you would sort the file.

1.Algorithm/Pseudocode :

Algorithm

Divide the File

Open inputFile for reading.

Set chunkSize to a value that fits into memory.

Initialize chunkCount to 0.

While inputFile has more data:

Read up to chunkSize bytes from inputFile into memory.

Split the data into lines.

Sort the lines in memory.

Write sorted lines to a temporary file named chunk\_<chunkCount>.txt.

Increment chunkCount.

Merge Sorted Chunks

Initialize a priority queue (min-heap).

Open all temporary files.

For each file, push the first line and file handle into the priority queue.

Open the output file sortedFile.txt.

While the priority queue is not empty:

Extract the smallest line from the priority queue.

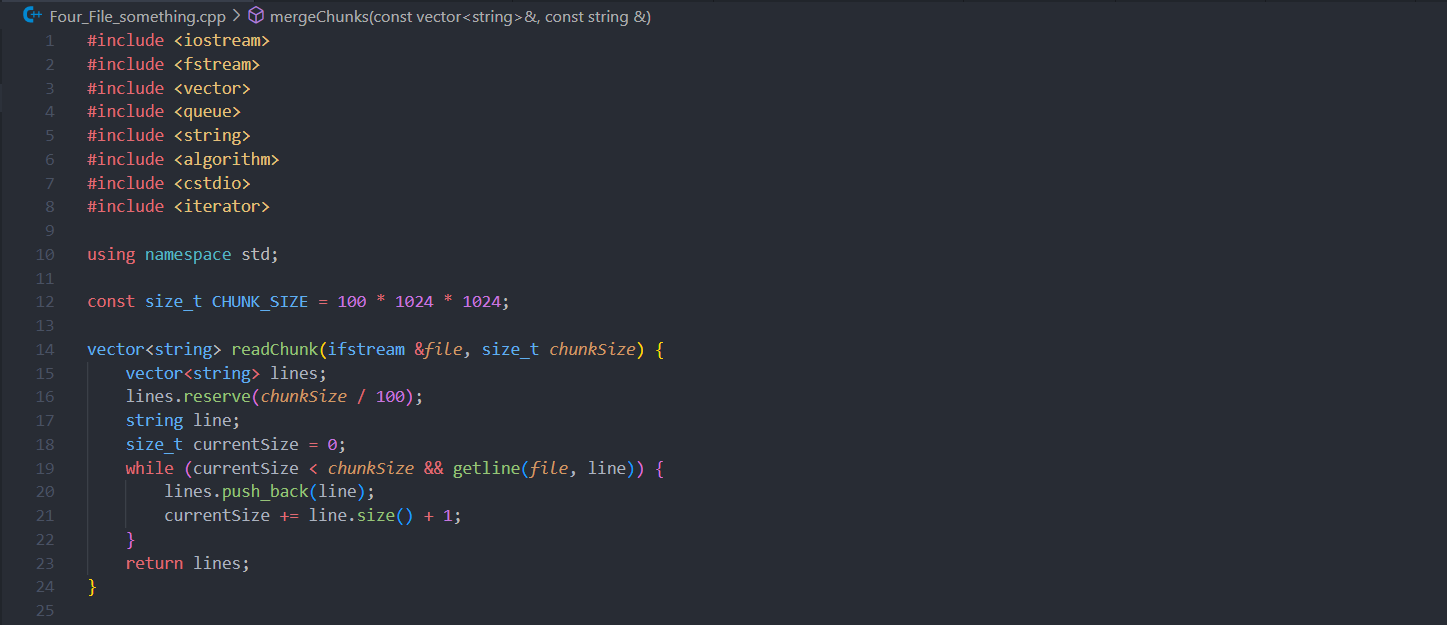
Write the line to sortedFile.txt.

Read the next line from the file corresponding to the extracted line and push it into the priority queue.

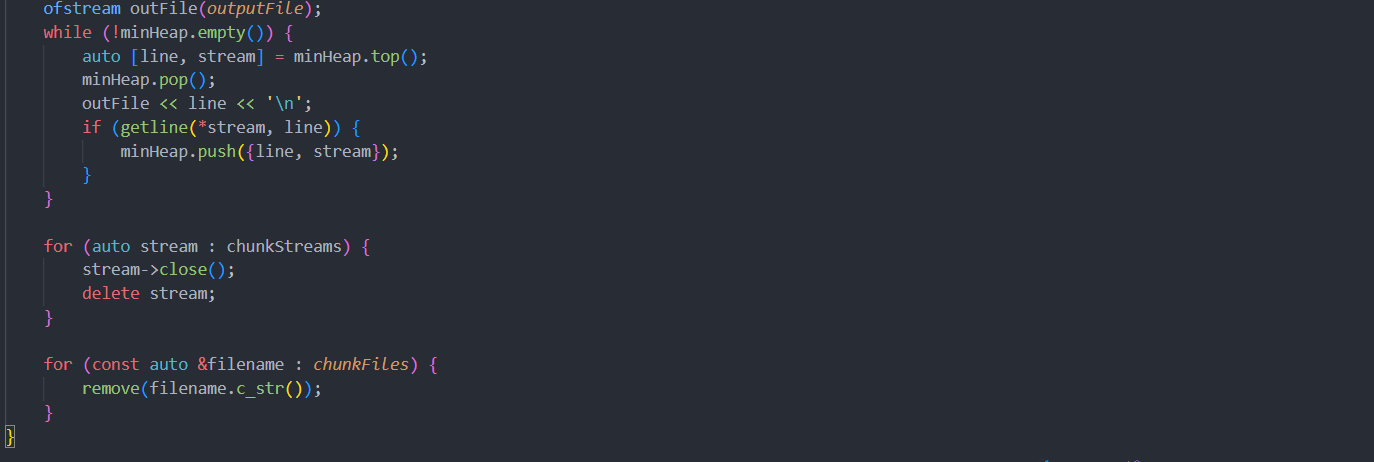
Close all temporary files and the output file.

Delete temporary files if required.

2.Program Code









3.Analysis of the complexity

**Chunk Sorting**:

Time Complexity: O(M log M) for sorting each chunk, where M is the number of lines in the chunk.

Space Complexity: O(M) for storing a chunk in memory.

**Merging Chunks**:

Time Complexity: O(N log k), where NN is the total number of lines and k is the number of chunks (files).

Space Complexity: O(k) for maintaining the priority queue and file handles.

**Problem Statement:**

Given a sorted array of string which is interspersed with empty string, write a method to find the location of a given string.

EXAMPLE

Input: find “ball” in {“at”, “”, “”, “ball”, “”, “”, “car”, “”, “”, “dad”, “”,””}

Output: 4

1.Algorithm/Pseudocode :

Initialization:

Set low to 0 and high to the last index of the array.

Binary Search:

While low is less than or equal to high:

Find the Middle:

Compute the middle index: mid = low + (high - low) / 2.

Skip Empty Strings:

If the string at mid is empty, move left or right to find the nearest non-empty string:

Move left: Set temp to mid and decrement temp until a non-empty string is found or temp is less than low.

Move right: Set temp to mid and increment temp until a non-empty string is found or temp is greater than high.

Check the Middle Value:

If the non-empty string found at temp is equal to the target string, return temp.

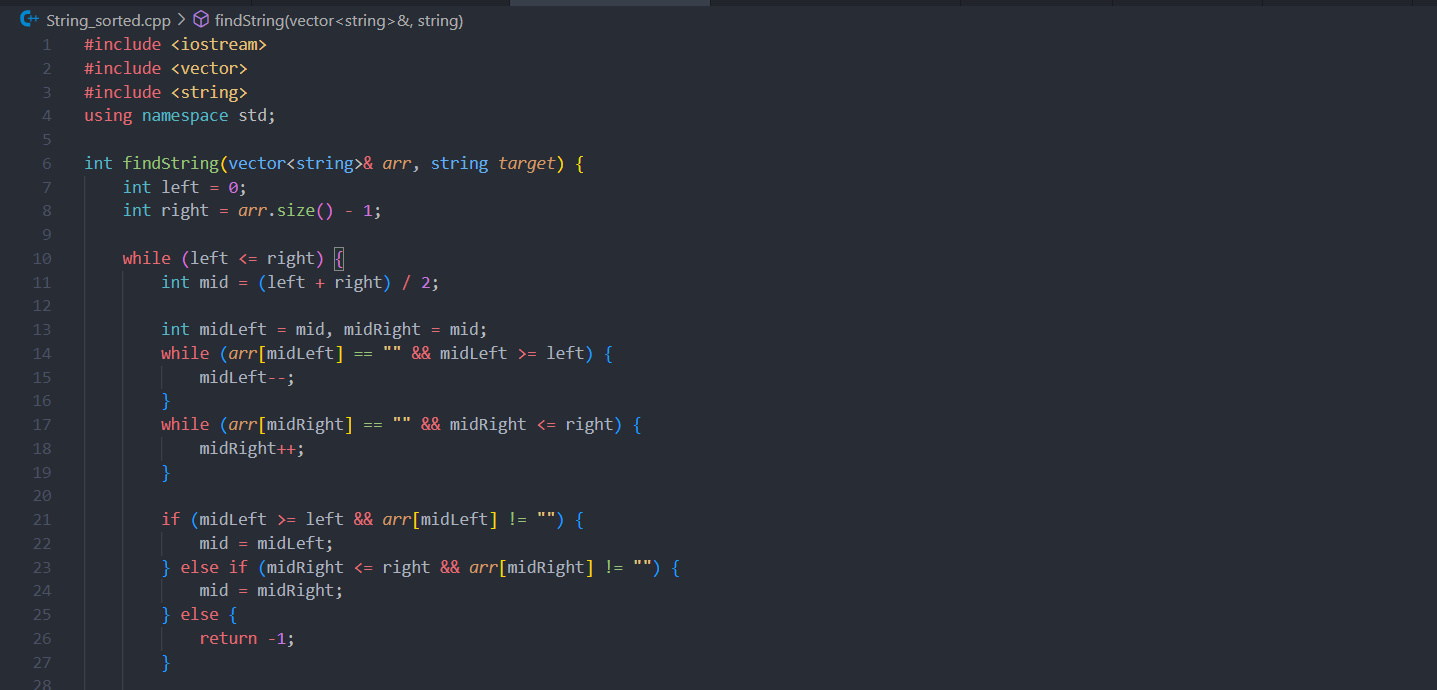
If the non-empty string is less than the target string, adjust the search to the right half: low = temp + 1.

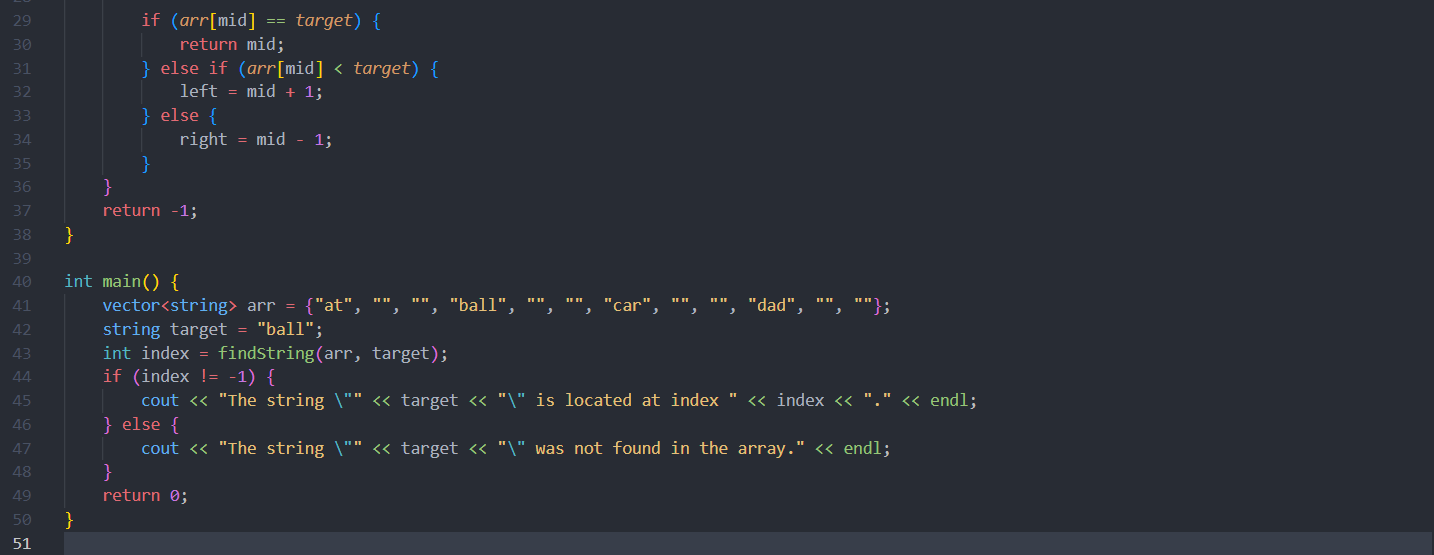
If the non-empty string is greater than the target string, adjust the search to the left half: high = temp - 1.

Return Not Found:

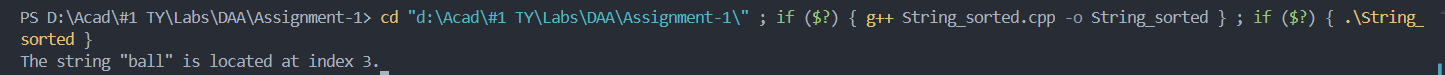
If the loop ends without finding the target string, return -1 indicating that the target string is not in the array.

2. Program Code





3. Output with verify of test cases



4.Analysis in terms of complexity wherever applicable

Binary Search Operations:

Standard Binary Search: A regular binary search algorithm has a time complexity of O(log N), where N is the number of elements in the array.

Skipping Empty Strings:

Finding the Nearest Non-Empty String:

In the worst case, you might need to scan up to O(log N)O(log N)O(log N) elements to find a non-empty string. This is because in a binary search, each step halves the search space. Therefore, you might need to scan approximately log N O(log N) O(log N) elements in the worst-case scenario to skip empty strings and find the next non-empty element.

**Problem Statement:**

Given an M\*N matrix in which each row and each column is sorted in ascending order, write a method to find an element.

1.Algorithm/Pseudocode :

Start from Top-Right Corner:

Begin your search from the top-right corner of the matrix. Let the initial position be (row, col) = (0, N-1) where N-1 is the last column index.

ComparisonandMovement:

While the current position (row, col) is within the matrix bounds:

Compare the element at (row, col) with the target value.

If the element equals the target value, return (row, col) as the position of the target.

If the element is greater than the target value:

Move left to the previous column by updating the column index col to col - 1.

If the element is less than the target value:

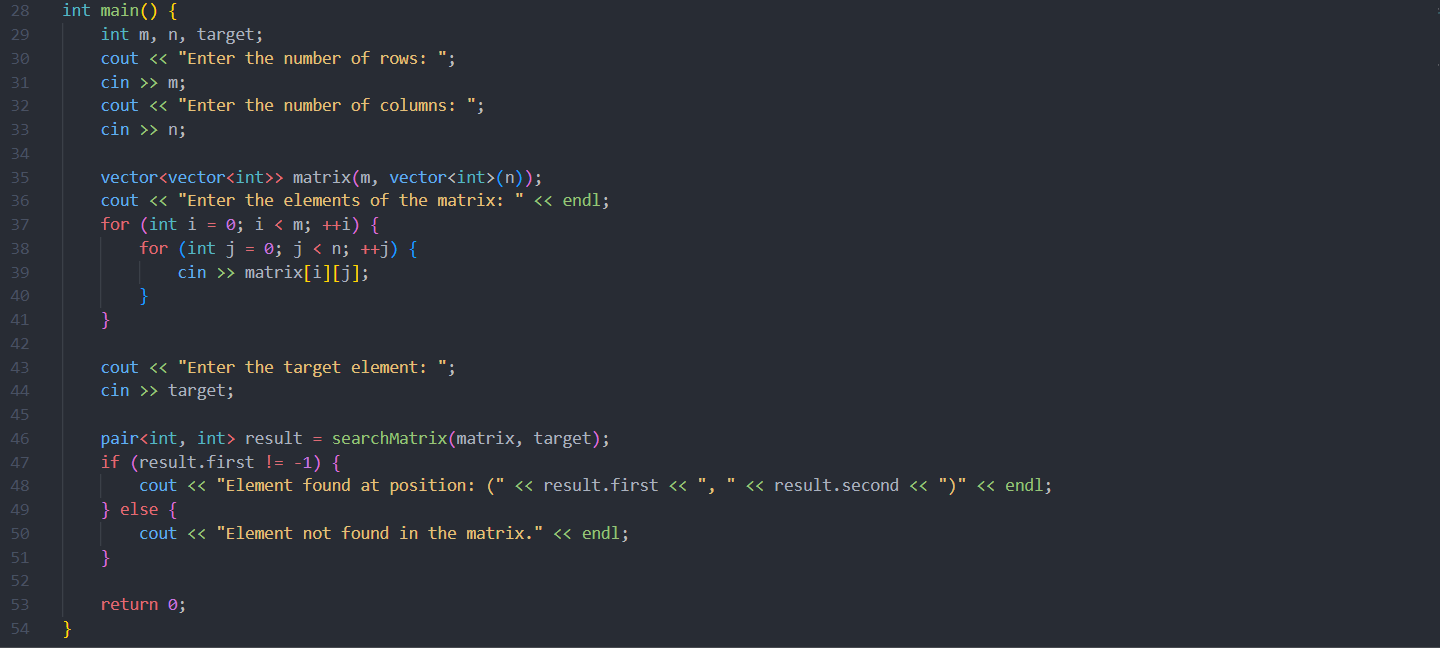
Move down to the next row by updating the row index row to row + 1.

Termination:

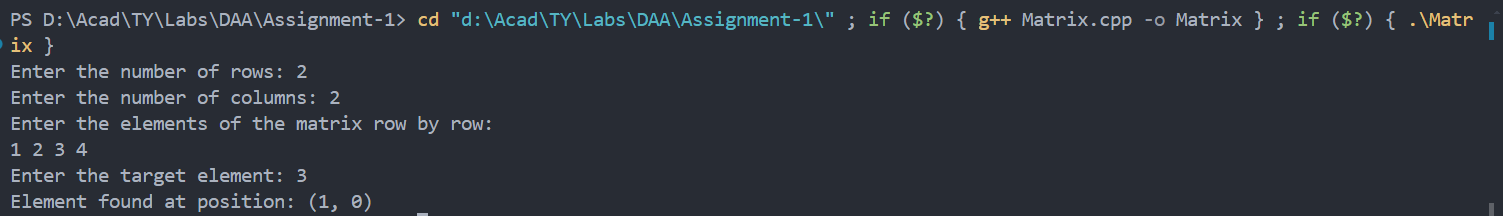
If you move out of matrix bounds (i.e., row becomes greater than or equal to M or col becomes less than 0), the target value is not present in the matrix. In this case, return (-1, -1) to indicate that the element is not found.

2.Program Code





3.Output with verify of test cases



4.Analysis in terms of complexity wherever possible

Time Complexity

The time complexity of the algorithm is O(m + n ), where mmm is the number of rows and n is the number of columns in the matrix.

The algorithm starts at the top-right corner of the matrix.

In each step, it either moves left (decreases the column index) or moves down (increases the row index).

Since it can move left at most n times and down at most mmm times, the maximum number of steps it takes is m + n.

Space Complexity

The space complexity of the algorithm is O(1) since it uses only a constant amount of extra space regardless of the size of the input matrix.

**Problem Statement:**

1.Algorithm /Pseudocode

Sort the Input:

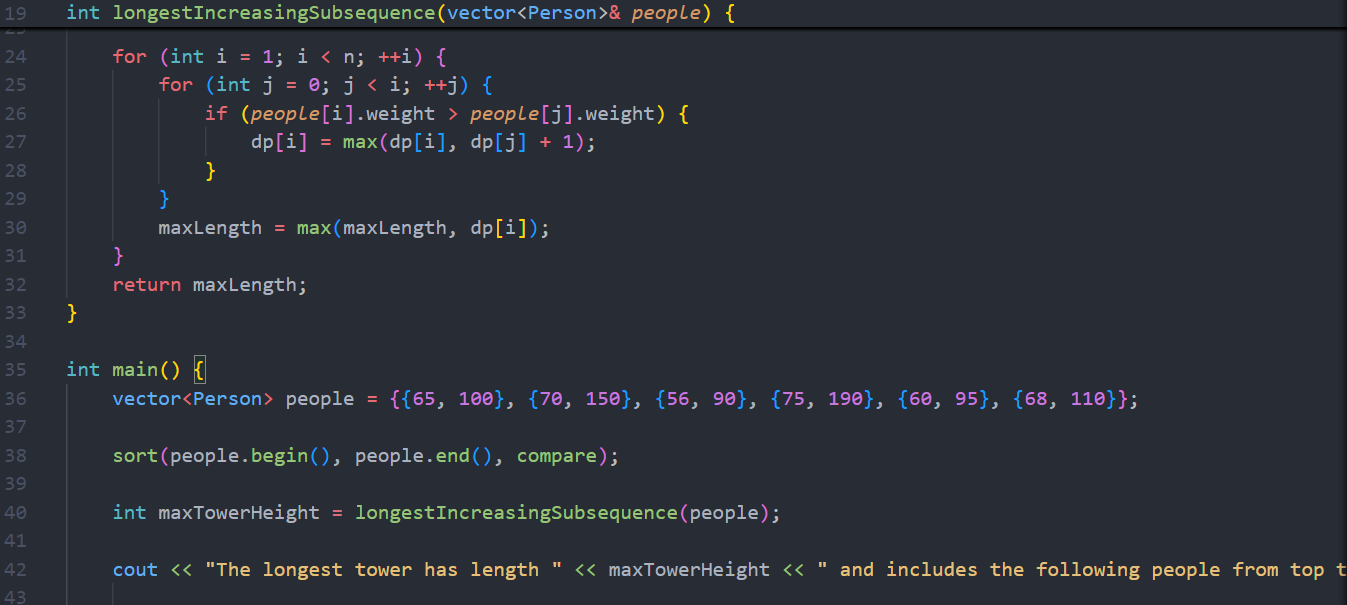
Sort the list of people based on height first. If two people have the same height, sort by weight.

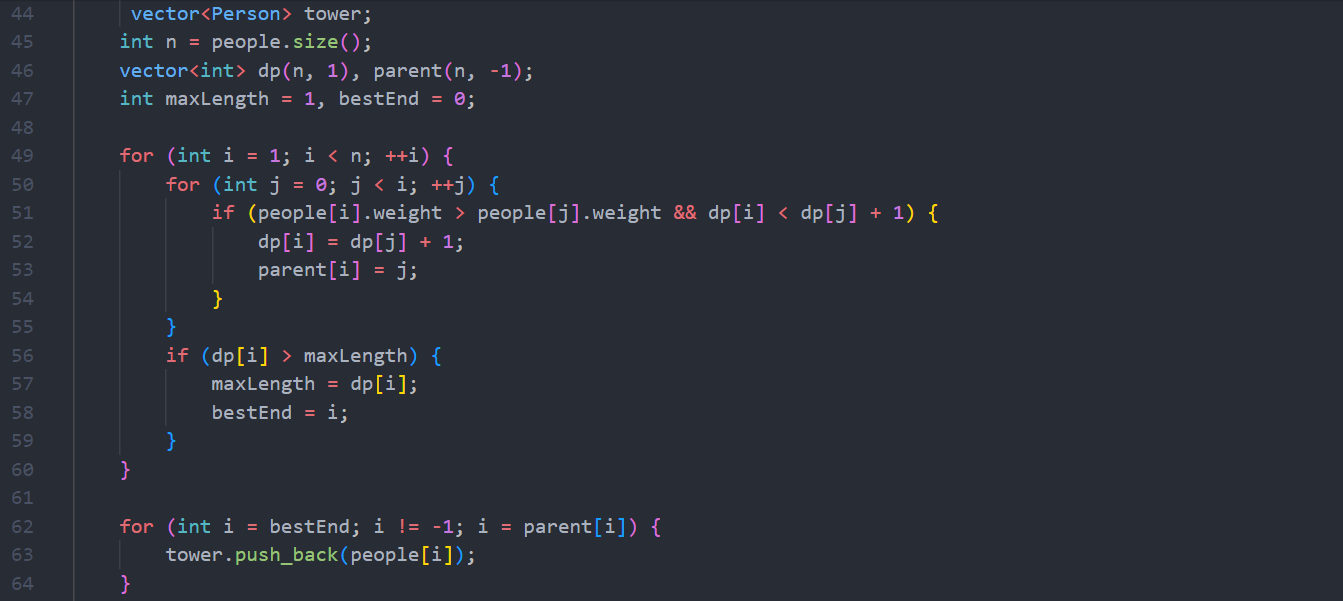
Find the Longest Increasing Subsequence (LIS):

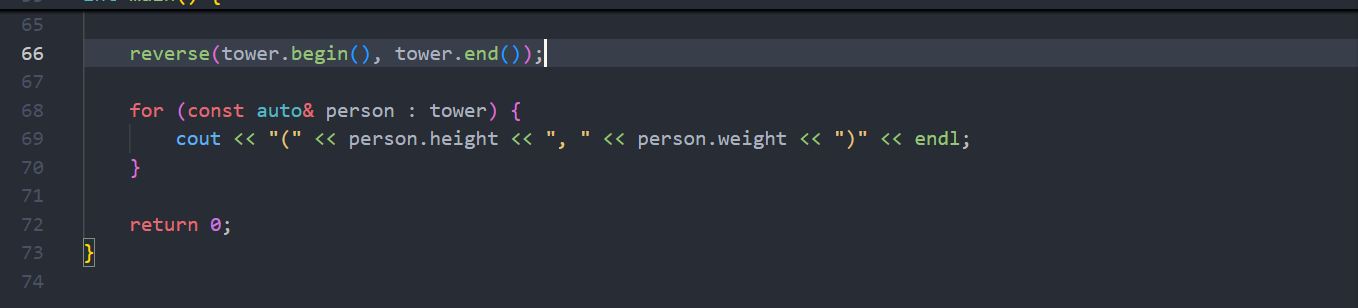
Use dynamic programming to find the longest increasing subsequence of weights in the sorted list of people

2.Program Code

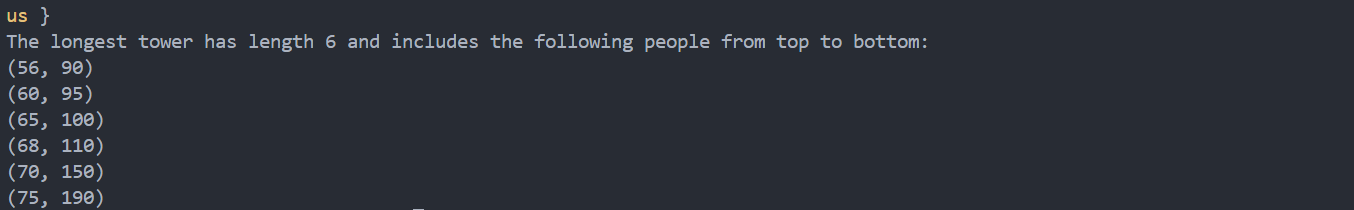








3.Output to verify test cases



4.Analysis in terms of complexity wherever applicable

Time Complexity:

Sorting the list takes O(n log n).

Finding the LIS takes O(n^2) using dynamic programming.

Space Complexity:

O(n) for storing the dp array.

**Problem Statement:**

Imagine you are reading in stream of integers. Periodically, you wish to be able to look up the rank of number *x* (the number of values less than or equal to *x*). Implement the data structures and algorithms to support these operations. That is, Implement the method *track (int x),* which is called when each number is generated, and the method *getRankOfNumber (int x)*, which return the number of values less than or equal to *x* (not including x itself).

EXAMPLE

Stream (in order of appearance) : 5, 1, 4, 4, 5, 9, 7, 13, 3

*getRankOfNumber(1) = 0*

*getRankOfNumber(3) = 1*

*getRankOfNumber(4) =3*

1.Algorithm/Pseudocode

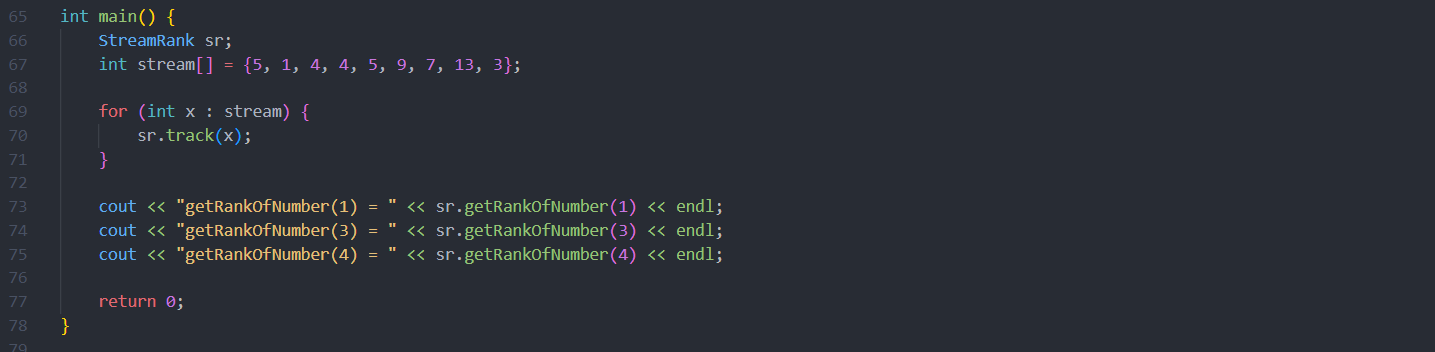
Node Structure: Define a BST node that also stores the size of its left subtree.

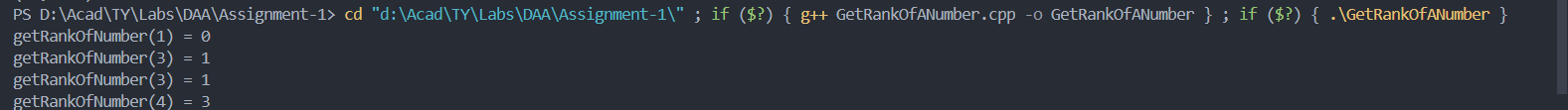
Insertion: Insert elements into the BST while updating the left subtree size.

Rank Query: Traverse the BST to calculate the rank of a given number.

2.Program Code :  



3.Output to verify test cases :  


4.Analysis in terms of complexity wherever possible  
Insertion (track method):

Time Complexity: O(log n) on average if the BST is balanced, but O(n) in the worst case if the BST becomes skewed.

Space Complexity: O(1) for each insert operation, but O(n) for the entire tree to store all elements.

Rank Query (getRankOfNumber method):

Time Complexity: O(log n) on average if the BST is balanced, but O(n) in the worst case if the BST becomes skewed.

Space Complexity: O(1) for each query operation.