Batch: T3

Assignment No.: 1

Title of Assignment: Sorting Algorithm

Student Name: Piyush Dawkhare

PRN: 23510056

Problem Statement 1:

Q) 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. Technologies/libraries/algorithm used:

* #include algorithms
* #include <bits/stdc++.h> //This library includes these two libraries
* #include <vector>

1. Related theory or any related information:

* We think in the manner that gives us better view of the array
* 1st when we take array A we give have to make sure, it has enough space for when we merge the array B into it.
* At the start we give garbage value or 0’s on place of extra space, or there is no need to give any value and just keep the that part of the empty.
* And then when merging we merge in the sorted order.

1. Algorithms:

* The algo for this is very simple, 1st take the input of the two arrays,
* Make sure they are sorted, if not sorted sort them
* Make sure the array in which we are merging both the arrays has enough space
* Either you can make a function or just work in the same function for finding this
* We require the 2 arrays which we are merging and the size of these two
* Make 3 pointer:

1. Pointer at last point of A
2. Pointer at last point of B
3. Pointer at last index of A

* Start a loop for till B has any element left
* Now we move simultaneously in both the arrays and arranging the elements and merging these two arrays together in array A.

1. Code:
2. // Q) You are given two sorted array, A and B, where A has a large enough buffer at the end to hold B.
3. // Write a method to merge B into A in sorted order
4. #include <bits/stdc++.h>
5. using namespace std;
6. // Merges array B into A assuming A has enough space
7. int mergeSortedArrays(vector<int>& A, int m, vector<int>& B, int n) {
9. int i = m - 1; // Pointer at last element of real data in A
10. int j = n - 1; // Pointer at last element of B
11. int k = m + n - 1; // Pointer at last index of A (including buf
12. // If B still has elements left
13. while (j >= 0) {
14. if (i >= 0 && A[i] > B[j])
15. {
16. A[k] = A[i];
17. i--;
18. }
19. else
20. {
21. A[k] = B[j];
22. j--;
23. }
24. k--;
25. }
26. for (int i = 0; i < m + n; i++) {
27. cout << A[i] << " ";
28. }
29. return 0;
30. }
31. int main ()
32. {
33. vector<int > A = {1,2,3,4,5,0,0,0,0,0};
34. vector<int > B = {6,7,8,9,10};
35. int m=5;
36. int n= B.size();
37. mergeSortedArrays(A, m, B, n);
38. return 0;
39. }

5. Input:

A = [1,2,3,4,5,0,0,0,0,0]

B = [6,7,8,9,10]

6. Output:

1 2 3 4 5 6 7 8 9 10

Problem Statement 2:

Q) 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. Technologies/libraries/algorithm used:

* #include <iostream> → for input and output
* #include <vector> → to store list of strings
* #include <algorithm> → used for sorting each string
* #include <unordered\_map> → used to group words based on sorted key

1. Related theory or any related information:

* **Anagrams** are words that have the same letters but arranged in different order.  
  Example: eat, tea, ate are anagrams of each other.
* To check if two words are anagrams → we sort the letters of both words. If sorted forms are equal, they are anagrams.
* Here, we are asked to **arrange all anagrams next to each other** in the array.
* For this problem, the trick is:
* Use a map (dictionary) where the **key** is the sorted version of the word.
* The **value** is a list of all words that match that key.
* Finally, we put all the words back into the array grouped together.

1. Algorithms:

* Take input list of words (array of strings).
* Create a map (unordered\_map) to group anagrams.
* For every string:

1. Make a copy of the string.
2. Sort its characters → this becomes the key.
3. Put the original string inside the map at that key.

* After processing all words, the map will have groups of anagram
* Clear the original array.
* Go through the map and put all grouped words back into the array.
* Print the final array → anagrams are now placed next to each other.

1. Code:

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

#include <iostream>

#include <vector>

#include <algorithm>

#include <unordered\_map>

using *namespace* std;

*void* groupAnagrams(vector<string>*&* *strs*) {

    unordered\_map<string, vector<string>> anagramMap;

    for (*const* string& str : *strs*) {

        string key = str;

        sort(key.begin(), key.end());

        anagramMap[key].push\_back(str);

    }

*strs*.clear();

    for (*const* *auto*& pair : anagramMap) {

        for (*const* string& anagram : pair.second) {

*strs*.push\_back(anagram);

        }

    }

}

*int* main() {

    vector<string> strs = {"eat", "tea", "tan", "ate", "nat", "bat"};

    groupAnagrams(strs);

    for (*const* string& str : strs) {

        cout << str << " ";

    }

    cout << endl;

    return 0;

}

5. Input:

Strs = {"eat", "tea", "tan", "ate", "nat", "bat"}

6. Output:

bat tan nat eat tea ate

Problem Statement 3:

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. Technologies/libraries/algorithm used:

* #include <iostream> → for input and output
* #include <vector> → to store the rotated sorted array
* **Binary Search Algorithm** → modified version to handle rotated arrays

1. Related theory or any related information:

* A **rotated sorted array** is an array that was originally sorted but then rotated (shifted) at some pivot point.  
  Example: Sorted array [1,2,3,4,5,6] rotated becomes [4,5,6,1,2,3].
* In a rotated array, **one half (left or right) is always sorted**.
* We can use **binary search** to search efficiently in O(log n) time by checking which half is sorted and narrowing the search space accordingly.
* If the target is in the sorted half, we move in that direction, otherwise we move to the other half.

1. Algorithms:

* Take input: rotated sorted array and the target element.
* Set two pointers → left at start and right at end.
* Run a loop while left <= right:
  + Find the middle index mid.
  + If arr[mid] == target, return the index.
  + If **left half is sorted** (arr[left] <= arr[mid]):
    - If target lies between arr[left] and arr[mid], move right = mid - 1.
    - Else, move left = mid + 1.
  + Else (means **right half is sorted**):
    - If target lies between arr[mid] and arr[right], move left = mid + 1.
    - Else, move right = mid - 1.
* If element is not found, return -1.

1. Code:

*#include <iostream>*

*#include <vector>*

*using namespace std;*

*int searchRotatedArray(const vector<int> &arr, int target)*

*{*

*int left = 0, right = arr.size() - 1;*

*while (left <= right)*

*{*

*int mid = left + (right - left) / 2;*

*if (arr[mid] == target)*

*return mid;*

*// Left half is sorted*

*if (arr[left] <= arr[mid])*

*{*

*if (target >= arr[left] && target < arr[mid])*

*right = mid - 1;*

*else*

*left = mid + 1;*

*}*

*// Right half is sorted*

*else*

*{*

*if (target > arr[mid] && target <= arr[right])*

*left = mid + 1;*

*else*

*right = mid - 1;*

*}*

*}*

*return -1; // Not found*

*}*

*int main()*

*{*

*vector<int> arr = {15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14};*

*int target = 5;*

*int index = searchRotatedArray(arr, target);*

*if (index != -1)*

*cout << "Element found at index: " << index << endl;*

*else*

*cout << "Element not found" << endl;*

*return 0;*

*}*

5. Input:

arr = {15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14};

target = 5

6. Output:

Element found at index: 8

Problem Statement 4:

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

1. Technologies/libraries/algorithm used:

* **External Sorting Algorithm** → because the file is too large to fit into memory (20GB is bigger than typical RAM).
* **Merge Sort** (External Merge Sort) → commonly used for sorting very large files.
* **File handling** → to read and write chunks of data.

1. Related theory or any related information:

* When the data is **larger than RAM**, we cannot load the whole file into memory and sort directly.
* In such cases, we use **external sorting**.
* The main idea:
* Break the file into smaller **chunks** that can fit into memory.
* Sort each chunk individually (using normal sort).
* Save the sorted chunks back to disk as temporary files.
* Finally, **merge** all sorted chunks together (like the merging step in merge sort).
* This method is efficient and commonly used in databases, search engines, and big data systems

1. Algorithms:

* Open the 20GB file.
* Read a chunk of the file that fits into memory (for example, 100MB at a time).
* Sort this chunk in memory using a standard sort (like quicksort or mergesort).
* Write the sorted chunk back to disk as a temporary file.
* Repeat steps 2–4 until the whole file is split into sorted chunks.
* Now perform a **k-way merge** on all these sorted chunks:
* Open all temporary files simultaneously.
* Use a min-heap (priority queue) to always pick the smallest current line among all files.
* Write that line into the final output file.
* Move the pointer forward in the file from which that line was taken.
* Continue until all temporary files are fully merged
* The final file is completely sorted.

Problem Statement 5:

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. Technologies/libraries/algorithm used:

* #include <iostream> → for input/output
* #include <vector> → to store the array of strings
* #include <string> → for working with string type
* **Modified Binary Search Algorithm** → adapted to handle empty strings

1. Related theory or any related information:

* Normally, binary search works well on a sorted array because every element has a defined value for comparison.
* Here the problem is: the array is sorted, but it contains **empty strings ("")** scattered in between.
* If we directly apply binary search and land on an empty string, we can’t decide whether to move left or right.
* So the trick is:
* If the middle element is an empty string, **look left and right** until you find the closest non-empty string.
* Then continue binary search as usual.
* This allows us to still use the efficiency of binary search with only a small adjustment.

1. Algorithms:

* Take input: a sorted array of strings with empty strings inside, and the target string.
* Apply binary search with left and right pointers.
* Find middle index mid.
* If arr[mid] is empty:
  + Move left and right outward until you find a non-empty string.
  + If none found, return -1.
* Compare arr[mid] with the target:
  + If equal → return mid (index found).
  + If target is smaller → search left half.
  + If target is larger → search right half.
* Continue until element is found or range becomes empty.
* If not found, return -1.

1. Code:

*// Q) 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*

*#include <iostream>*

*#include <vector>*

*#include <string>*

*using namespace std;*

*int sparseSearch(const vector<string>& arr, int left, int right, const string& target) {*

*if (left > right) return -1;                // Base case: range empty*

*int mid = (left + right) / 2;*

*// If mid is empty, find the nearest non-empty string*

*if (arr[mid].empty()) {*

*int l = mid - 1, r = mid + 1;*

*while (l >= left || r <= right) {*

*if (l >= left && !arr[l].empty()) {*

*mid = l;*

*break;*

*}*

*if (r <= right && !arr[r].empty()) {*

*mid = r;*

*break;*

*}*

*l--; r++;*

*}*

*// If no non-empty string found*

*if (arr[mid].empty()) return -1;*

*}*

*// Compare and continue binary search*

*if (arr[mid] == target) {*

*return mid;*

*}*

*else if (arr[mid] > target) {*

*return sparseSearch(arr, left, mid - 1, target);*

*} else {*

*return sparseSearch(arr, mid + 1, right, target);*

*}*

*}*

*int main()*

*{*

*vector<string> words = {"at", "", "", "ball", "", "", "car", "", "", "dad", "", "", ""};*

*string required = "ball";*

*int index = sparseSearch(words,0,12,required);*

*if(index!=-1)*

*cout<<words[index]<<" Found at index "<<index<<endl;*

*else*

*cout << "Not Found" << endl;*

*return 0;*

*}*

5. Input:

Words = ["at", "", "", "ball", "", "", "car", "", "", "dad", "", "", ""]

target = “ball”

6. Output:

"ball" Found at index 3

Problem Statement 6:

Q) *Given an M\*N matrix in which each row and each column is sorted in ascending order, write a*

1. Technologies/libraries/algorithm used:

* #include <iostream> → for input/output
* #include <vector> → to store the matrix
* **Search algorithm in a sorted 2D matrix** → starts from top-right and eliminates row/column step by step

1. Related theory or any related information:

* The matrix given is special:
* Each row is sorted in ascending order (left to right).
* Each column is sorted in ascending order (top to bottom).
* A simple approach (checking all elements) would take O(M\*N) time.
* A better approach:
* Start from the **top-right corner** of the matrix.
* At each step:
  + If the element is equal to the target → found.
  + If the element is larger than target → move **left** (since going left gives smaller numbers).
  + If the element is smaller than target → move **down** (since going down gives larger numbers).
* This reduces the search to O(M + N) instead of O(M\*N).

1. Algorithms:

* Take the matrix and target as input.
* Start from top-right element: (row=0, col=last column).
* Repeat until you go out of matrix bounds:
  + If matrix[row][col] == target → return (row, col).
  + If matrix[row][col] > target → move left (col--).
  + Else (matrix[row][col] < target) → move down (row++).
* If you exit the loop without finding, return (-1, -1) meaning not found.

1. Code:

*// Q) Given an M\*N matrix in which each row and each column is sorted in ascending order, write a*

*// method to find an element*

*#include <iostream>*

*#include <vector>*

*using namespace std;*

*pair<int, int> searchMatrix(const vector<vector<int>> &matrix, int target)*

*{*

*int rows = matrix.size();*

*if (rows == 0)*

*return {-1, -1};*

*int cols = matrix[0].size();*

*int row = 0, col = cols - 1;*

*while (row < rows && col >= 0)*

*{*

*if (matrix[row][col] == target)*

*return {row, col};*

*else if (matrix[row][col] > target)*

*col--; // Move left*

*else*

*row++; // Move down*

*}*

*return {-1, -1}; // Element not found*

*}*

*int main()*

*{*

*vector<vector<int>> mat = {*

*{10, 20, 30, 40},*

*{15, 25, 35, 45},*

*{27, 29, 37, 48},*

*{32, 33, 39, 50}};*

*int target = 29;*

*pair<int, int> pos = searchMatrix(mat, target);*

*if (pos.first != -1)*

*cout << "Found at (" << pos.first << ", " << pos.second << ")\n";*

*else*

*cout << "Not Found\n";*

*}*

5. Input:

mat = {

        {10, 20, 30, 40},

        {15, 25, 35, 45},

        {27, 29, 37, 48},

        {32, 33, 39, 50}};

target = 29

6. Output:

Found at (2,1)’

Problem Statement 7:

Q) A circus is designing a tower routine consisting of people standing atop one another’s shoulders. For practical and aesthetic reasons, each person must be both shorter and lighter than the person

below him or her. Given the heights and weight of each circus, write a method to compute the largest

possible number of people in such tower.

EXAMPLE:

Input(ht,wt): (65, 100) (70, 150) (56, 90) (75,190) (60, 95) (68, 110).

Output: The longest tower is length 6 and includes from top to bottom:

(56, 90) (60, 95) (65, 100) (68, 110) (70, 150) (75, 190)

Q) 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. Technologies/libraries/algorithm used:
   * #include <iostream> → for input/output
   * #include <vector> → to store list of people
   * #include <algorithm> → used for sorting and reversing the sequence
   * **Dynamic Programming (LIS – Longest Increasing Subsequence)** → applied on weights after sorting by height
2. Related theory or any related information:

* The problem is similar to building the **longest increasing subsequence**, but with two conditions:
  1. Each person must be **shorter** than the one below.
  2. Each person must also be **lighter** than the one below.
* To solve:
  1. First, **sort people by height** (and if heights are equal, sort by weight).
  2. Then apply **LIS (Longest Increasing Subsequence)** on the **weights**.
* Why it works:
  1. Sorting by height ensures that heights are always increasing.
  2. Applying LIS on weights ensures both conditions (shorter + lighter) are satisfied.
* This is a classical DP + sorting problem, also known as the “Circus Tower Problem.”

1. Algorithms:

* Take input: a list of people with (height, weight).
* Sort them by height (and by weight if heights are equal).
* Initialize a DP array dp[i] = length of longest tower ending at person i.
* For each person i from 1 to n-1:
  + Compare with all previous persons j < i.
  + If weight[j] < weight[i], then update:  
    dp[i] = max(dp[i], dp[j] + 1)
  + Also keep a prev array to reconstruct the actual sequence.
* Track the maximum length and its ending index.
* Reconstruct the sequence of people by going backwards using the prev array.
* Reverse the result to get the tower from **top to bottom**.
* Print the tower and its length.

1. Code:

*// Q) A circus is designing a tower routine consisting of people standing atop one another’s shoulders.*

*// For practical and aesthetic reasons, each person must be both shorter and lighter than the person*

*// below him or her. Given the heights and weight of each circus, write a method to compute the largest*

*// possible number of people in such tower.*

*// EXAMPLE:*

*// Input(ht,wt): (65, 100) (70, 150) (56, 90) (75,190) (60, 95) (68, 110).*

*// Output: The longest tower is length 6 and includes from top to bottom:*

*// (56, 90) (60, 95) (65, 100) (68, 110) (70, 150) (75, 190)*

*#include <iostream>*

*#include <vector>*

*#include <algorithm>*

*using namespace std;*

*// Define a struct for person*

*struct Person*

*{*

*int height;*

*int weight;*

*Person(int h, int w) : height(h), weight(w) {}*

*};*

*// Custom comparison: sort by height, and if equal, by weight*

*bool compare(const Person &a, const Person &b)*

*{*

*if (a.height == b.height)*

*return a.weight < b.weight;*

*return a.height < b.height;*

*}*

*// Function to find longest tower*

*vector<Person> longestTower(vector<Person> &people)*

*{*

*// Step 1: Sort by height and weight*

*sort(people.begin(), people.end(), compare);*

*// Step 2: Apply LIS on weight*

*int n = people.size();*

*vector<int> dp(n, 1);    // dp[i] = length of LIS ending at i*

*vector<int> prev(n, -1); // To reconstruct path*

*int maxLen = 1, maxIndex = 0;*

*for (int i = 1; i < n; i++)*

*{*

*for (int j = 0; j < i; j++)*

*{*

*if (people[j].weight < people[i].weight && dp[j] + 1 > dp[i])*

*{*

*dp[i] = dp[j] + 1;*

*prev[i] = j;*

*}*

*}*

*if (dp[i] > maxLen)*

*{*

*maxLen = dp[i];*

*maxIndex = i;*

*}*

*}*

*// Reconstruct the longest sequence*

*vector<Person> result;*

*for (int i = maxIndex; i != -1; i = prev[i])*

*result.push\_back(people[i]);*

*reverse(result.begin(), result.end());*

*return result;*

*}*

*int main()*

*{*

*vector<Person> people = {*

*{65, 100}, {70, 150}, {56, 90}, {75, 190}, {60, 95}, {68, 110},{100,200}};*

*vector<Person> tower = longestTower(people);*

*cout << "Longest tower length: " << tower.size() << endl;*

*cout << "Tower from top to bottom:" << endl;*

*for (const auto &p : tower)*

*cout << "(" << p.height << ", " << p.weight << ")" << endl;*

*return 0;*

*}*

5. Input:

(65, 100) (70, 150) (56, 90) (75,190) (60, 95) (68, 110)

6. Output:

Longest tower length: 6

Tower from top to bottom:

(56, 90) (60, 95) (65, 100) (68, 110) (70, 150) (75, 190)

Problem Statement 8:

Q) *Q) 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. Technologies/libraries/algorithm used:

* #include <iostream> → for input/output
* #include <vector> → to store the stream of numbers
* **Basic array traversal** (your code checks rank by scanning from the end)

1. Related theory or any related information:

* The problem is about maintaining a stream of numbers and being able to answer queries like:  
  **“What is the rank of number x?”**
* The **rank** of a number is the count of numbers **less than or equal to x** in the stream.
* Example stream: 5, 1, 4, 4, 5, 9, 7, 13, 3
  + Rank of 1 = 0 (no number less than 1)
  + Rank of 3 = 1 (only 1 is less)
  + Rank of 4 = 3 (numbers ≤ 4 are 1, 3, 4)
* The code given works by scanning the vector and checking the position of x, then printing its rank (index-based).
* However, this version only finds the **last occurrence index** instead of true “rank” count.
* For correct rank, one should count all numbers less than or equal to x.

1. Algorithms:

* Read the size n of the stream.
* Input n numbers into a vector arr.
* Read the number x whose rank we want to find.
* Start from the end of the array and scan backwards:
  + If arr[i] == x, print i+1 as the rank (index-based position).
  + Mark that number as found and break.
* If no match is found, print "Number Not Found".

1. Code:
2. *// Q) Imagine you are reading in stream of integers. Periodically, you wish to be able to look up the rank*
3. *// of number x (the number of values less than or equal to x). Implement the data structures and*
4. *// algorithms to support these operations. That is, Implement the method track (int x), which is called*
5. *// when each number is generated, and the method getRankOfNumber (int x), which return the number*
6. *// of values less than or equal to x (not including x itself).*
7. *// EXAMPLE*
8. *// Stream (in order of appearance) : 5, 1, 4, 4, 5, 9, 7, 13, 3*
9. *// getRankOfNumber(1) = 0*
10. *// getRankOfNumber(3) = 1*
11. *// getRankOfNumber(4) =3*
12. #include <iostream>
13. #include <vector>
14. using *namespace* std;
15. *int* main()
16. {
17. *int* n;
18. cin >> n;
19. vector<*int*> arr(n);
20. for (*int* i = 0; i < n; i++)
21. {
22. cin >> arr[i];
23. }
24. *int* x;
25. cout << "Enter Number to Know Rank" << endl;
26. cin >> x;
27. *bool* check = false;
28. for (*int* i = n - 1; i >= 0; i--)
29. {
30. if (arr[i] == x)
31. {
32. cout << "Rank of " << x << " is " << i + 1 << endl;
33. check = true;
34. break;
35. }
36. }
37. if (!check)
38. {
39. cout << "Number Not Found" << endl;
40. }
41. return 0;
42. Input:

n = 9

Stream: 5 1 4 4 5 9 7 13 3

x = 4

1. Output:

Rank of 4 is 4