**Sakshi Dhamapurkar**

**Batch-T3**

**PRN:21510073**

**Assignment-1**

Part: 1

**Sorting Algorithm**

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.

*#include* <bits/stdc++.h>

*/\* Sakshi \*/*

using namespace std;

*// Write a method to merge B into A in sorted order.*

void mergeArrays(int *arr1*[], int *arr2*[], int *n1*,int *n2*, int *arr3*[])

{

    int i = 0, j = 0, k = 0;

*while* (i < *n1* && j < *n2*)

    {

*if* (*arr1*[i] < *arr2*[j])

*arr3*[k++] = *arr1*[i++];

*else*

*arr3*[k++] = *arr2*[j++];

    }

*while* (i < *n1*)

*arr3*[k++] = *arr1*[i++];

*while* (j < *n2*)

*arr3*[k++] = *arr2*[j++];

}

int main()

{

    int n1, n2;

    cout<<"Enter n1 ";

    cin >> n1;

    cout<<"Enter n2 ";

    cin>>n2;

    int arr1[n1];

    int arr2[n2];

    cout << "Original array 1 ";

*for* (int i = 0; i < n1; i++)

    {

        cin >> arr1[i];

    }

    cout << endl;

    cout << "Original array 2 ";

*for* (int i = 0; i < n2; i++)

    {

        cin >> arr2[i];

    }

    int arr3[n1];*//as n1>n2*

    mergeArrays(arr1,arr2, n1, n2,arr3);

    cout << endl;

    cout << "Sorted array: ";

*for* (int i = 0; i < n1+n2; i++)

    {

        cout << arr3[i] << " ";

    }

    cout << endl;

*return* 0;

}

//TC -> O(nlogn)

//SC->O(n1+n2)

* **Time Complexity:** The overall time complexity of the provided code is O(sizeA + sizeB), where sizeA is the number of elements in array A, and sizeB is the number of elements in array B.
* **Space Complexity:** The overall space complexity of the provided code is constant, O(1), as it doesn't use additional memory that grows with the input size.

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

*#include*<bits/stdc++.h>

*//𝕊𝕒𝕜𝕤𝕙𝕚 ℂ𝕠𝕕𝕖𝕤 𝕙𝕖𝕣𝕖*

using namespace std;

vector<string> sort\_string\_anagram(vector<string> *array*)

{

    unordered\_map<string, set<string>> anagram;

*for* (string word : *array*)

    {

        string sorted\_word(word);

        sort(sorted\_word.begin(), sorted\_word.end());

        anagram[sorted\_word].insert(word);

    }

    sort(*array*.begin(), *array*.end());

    vector<string> result;

*for* (string word : *array*)

    {

        unordered\_map<string, set<string>>::iterator iter;

        string sorted\_word(word);

        sort(sorted\_word.begin(), sorted\_word.end());

*if* ((iter = anagram.find(sorted\_word)) != anagram.end())

        {

*for* (set<string>::iterator it = (iter->second).begin(); it != (iter->second).end(); ++it)

            {

                result.push\_back(\*it);

            }

            anagram.erase(iter);

        }

    }

*return* result;

}

* **Time Complexity:** O(m \* n \* log n) where m is the number of input strings and n is the average length of the strings.
* **Space Complexity:** O(m \* n), where m is the number of input strings, and n is the average length of the strings.

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)

Search in rotated sorted array

*#include* <bits/stdc++.h>

*// 𝕊𝕒𝕜𝕤𝕙𝕚 ℂ𝕠𝕕𝕖𝕤 𝕙𝕖𝕣𝕖*

using namespace std;

class Solution{

    public :

        int search(vector<int> & *nums*, int *target*){

            int n = *nums*.size();

            int lo=0,hi=n-1,mid;

*while*(lo < hi){

                mid = (hi-lo)/2;

*if*(*nums*[mid] == *target*) *return* mid;

*else* *if*(*nums*[lo] < *nums*[mid]){

*if*(*target* > *nums*[lo] && *target* < *nums*[mid]) hi = mid-1;

*else* lo = mid+1;

                }*else* {

*if*(*target* <= *nums*[hi] && *target* > *nums*[mid]) lo = mid+1;

*else* hi = mid-1;

                }

            }

*if*(*nums*[lo] == *target*) *return* lo;

*if*(*nums*[hi] == *target*) *return* hi;

*return* -1;

        }

};

* **Time Complexity:** O(log n) where n is the number of elements in the input array.
* **Space Complexity:** O(1), constant space usage regardless of input size.

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

Algorithm:

Ans:

Sorting a 20GB file with one string per line is challenging to *do* directly in memory since the file exceeds the available RAM.

To efficiently sort such a large file, you can use an external sorting technique that involves dividing the file into smaller chunks that fit into memory,

 sorting each chunk, and then merging the sorted chunks back together.

The main steps of using Merge Sort *for* external sorting are as follows:

1. Divide into Chunks: Read the 20GB file in smaller chunks that fit into memory. Each chunk should be of manageable size,

such as 100MB or 1GB. For simplicity, let's assume we read 1GB chunks.

2. Sort Each Chunk: For each chunk, apply Merge Sort to sort the strings in memory.

3. Write Sorted Chunks to Temporary Files: After sorting each chunk, write the sorted strings back to temporary files. We may end up with multiple temporary files, one for each sorted chunk.

4. Merge Sorted Chunks: Use a k-way merge algorithm to merge the sorted temporary files into a single large sorted file.

 The k-way merge algorithm works by selecting the smallest string from each temporary file and merging them into a single output file.

We can also use pair wise merge

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* <bits/stdc++.h>

*// sakshi codes here*

using namespace std;

class searchAlgo

{

public:

    int searchh(vector<string> &*list*, string &*str*)

    {

*if* (*list*.empty() || *str*.empty())

*return* -1;

*return* searchRecu(*list*, *str*, 0, *str*.size() - 1);

    }

    int searchRecu(vector<string> &*list*, string &*str*, int *low*, int *high*)

    {

*if* (*low* > *high*)

*return* -1;

        int mid = findMid(*list*, *low*, *high*);

*if* (mid == -1)

*return* -1;

*if* (*list*[mid] == *str*)

*return* mid;

*return* *str*.compare(*list*[mid]) < 0 ? searchRecu(*list*, *str*, *low*, mid - 1) : searchRecu(*list*, *str*, mid + 1, *high*);

    }

    int findMid(vector<string> &*list*, int *low*, int *high*)

    {

        int mid = (*low* + *high*) / 2;

*if* (*list*[mid].empty())

        {

            int l = mid - 1;

            int r = mid + 1;

*while* (l >= *low* && r <= *high*)

            {

*if* (!*list*[r].empty())

                {

                    mid = r;

*break*;

                }

*if* (!*list*[l].empty())

                {

                    mid = l;

*break*;

                }

                l--;

                r++;

            }

*if* (l < *low* || r > *high*)

            {

*return* -1;

            } *// out of bound*

        }

*return* mid;

    }

};

int main()

{

    searchAlgo search;

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

    string str = "ball";

    int result = search.searchh(ans, str);

*if* (result != -1)

        cout << "String found at index " << result << endl;

*else*

        cout << "String not found" << endl;

}

* **Time Complexity:** O(log n) where n is the number of elements in the input array.
* **Space Complexity:** O(1), constant space usage regardless of input size.

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*<bits/stdc++.h>

using namespace std;

class Solution {

public:

    bool searchMatrix(vector<vector<int>>& *matrix*, int *target*) {

      int m=matrix.size();

      int n=matrix[0].size();

      int i=0,j=n-1;

*while*(i<m && j>=0)

      {

*if*(matrix[i][j] == target)

        {

*return* true;

        }

*//target is smaller than current element so move to previous column*

*else*

*if*(matrix[i][j] > target)

          {

            j--;

          }

*else*

        {

          i++;

        }

      }

*return* false;

    }

};

TC: Is O(m + n), where 'm' is the number of rows in the matrix, and 'n' is the number of columns in the matrix.

SC: Is O(1) because it uses a constant amount of extra space for variables (**m**, **n**, **i**, **j**, and **target**) regardless of the size of the input matrix.

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)

*// sort data acc to height O(nlogn)*

*// sort data acc to weight O(nlogn)*

*// find length of LCS from d1 and d2*

*#include* <bits/stdc++.h>

using namespace std;

bool compareByHeight(pair<int, int> &*p1*, pair<int, int> &*p2*)

{

*if* (*p1*.first != *p2*.first)

*return* *p1*.first < *p2*.first;

*else*

*return* *p1*.second < *p2*.second;

}

bool compareByWeight(pair<int, int> &*p1*, pair<int, int> &*p2*)

{

*if* (*p1*.second != *p2*.second)

*return* *p1*.second < *p2*.second;

*else*

*return* *p1*.first < *p2*.first;

}

int LCS(vector<pair<int, int>> &*height*, vector<pair<int, int>> &*weight*)

{

    int n = *height*.size(), m = *weight*.size();

    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

*for* (int i = 1; i <= n; i++) *// Change i-- to i++*

    {

*for* (int j = 1; j <= m; j++) *// Change j-- to j++*

        {

*if* (*height*[i - 1] == *weight*[j - 1])

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

*else*

                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);

        }

    }

*return* dp[n][m];

}

void print(vector<pair<int, int>> &*data*)

{

    vector<pair<int, int>> heightSorted = *data*;

    vector<pair<int, int>> weightSorted = *data*;

    sort(heightSorted.begin(), heightSorted.end(), compareByHeight);

    sort(weightSorted.begin(), weightSorted.end(), compareByWeight);

    int length = LCS(heightSorted, weightSorted);

    cout << "The longest tower is length " << length << " and includes from top to bottom:" << endl;

    int n = heightSorted.size();

    int m = weightSorted.size();

    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

    int i = n, j = m;

    vector<pair<int, int>> tower;

*while* (i > 0 && j > 0)

    {

*if* (heightSorted[i - 1] == weightSorted[j - 1])

        {

            tower.push\_back(heightSorted[i - 1]);

            i--;

            j--;

        }

*else* *if* (dp[i - 1][j] > dp[i][j - 1])

        {

            i--;

        }

*else*

        {

            j--;

        }

    }

    reverse(tower.begin(), tower.end());

*for* (const auto &p : tower)

    {

        cout << "(" << p.first << ", " << p.second << ") ";

    }

    cout << endl;

}

int main()

{

    vector<pair<int, int>> data1 = {

        make\_pair(65, 100),

        make\_pair(70, 150),

        make\_pair(56, 90),

        make\_pair(75, 190),

        make\_pair(60, 95),

        make\_pair(68, 110)};

    int result = LCS(data1, data1);

    cout << "Result: " << result << endl;

    print(data1);

*return* 0;

}

TC->O(n log n + n \* m).

SC->O(n \* m)

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*

Ans:

*//last que*

*#include* <iostream>

using namespace std;

class TreeNode {

public:

    int val;

    int count; *// no of nodes in left subtree, including this node.*

    TreeNode\* left;

    TreeNode\* right;

    TreeNode(int *x*) : val(*x*), count(1), left(nullptr), right(nullptr) {}

};

class BinarySearchTree {

private:

    TreeNode\* root;

    TreeNode\* insert(TreeNode\* *node*, int *x*) {

*if* (!*node*) {

*return* new TreeNode(*x*);

        }

*if* (*x* <= *node*->val) {

*node*->left = insert(*node*->left, *x*);

*node*->count++; *// count++ for left subtree.*

        } *else* {

*node*->right = insert(*node*->right, *x*);

        }

*return* *node*;

    }

    int getRank(TreeNode\* *node*, int *x*) {

*if* (!*node*) {

*return* 0;

        }

*if* (*x* == *node*->val) {

*return* *node*->count - (*node*->right ? *node*->right->count : 0);

        } *else* *if* (*x* < *node*->val) {

*return* getRank(*node*->left, *x*);

        } *else* {

*return* *node*->count + getRank(*node*->right, *x*);

        }

    }

public:

    BinarySearchTree() : root(nullptr) {}

    void track(int *x*) {

        root = insert(root, *x*);

    }

    int getRankOfNumber(int *x*) {

*return* getRank(root, *x*);

    }

};

int main() {

    BinarySearchTree bst;

    bst.track(5);

    bst.track(1);

    bst.track(4);

    bst.track(4);

    bst.track(5);

    bst.track(9);

    bst.track(7);

    bst.track(13);

    cout << "Rank of 4: " << bst.getRankOfNumber(4) << endl; *// print 3 (values <= 4: 1, 4, 4)*

    cout << "Rank of 9: " << bst.getRankOfNumber(9) << endl; *// print 6 (values <= 9: 1, 4, 4, 5, 5, 9)*

*return* 0;

}

**Average Case:**

* TC): O(log N)
* SC: O(N),

**Best Case:**

* TC: O(log N) when the tree is perfectly balanced, making it a height-balanced binary search tree.
* SC: O(N)

**Worst Case:**

* TC: O(N) when the tree is skewed (essentially a linked list), and the depth of the tree becomes N, resulting in linear time complexity.
* SC: O(N)

The average and best cases assume that the tree remains approximately balanced as elements are inserted. The worst-case time complexity occurs when elements are inserted in a way that the tree becomes unbalanced.