**Title of assignment: Sorting Algorithm**

Name: Mayur Savale

PRN: 21520010

1. 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.

Ans:

1. **Algorithm: (Pseudocode)**

* Traverse two arrays from back using two index pointers
* Place larger element at the backside
* If one of the array will be completely traversed then other array also will be in the right position

1. **Code snapshots of implementation**

#include <bits/stdc++.h>

using namespace std;

void merge(vector<int>& A,int m,vector<int>& B,int n)

{

int i=m-1,j=n-1,s=m+n;

while(s--)

{

int s1=(i>=0?A[i]:INT\_MIN);

int s2=(j>=0?B[j]:INT\_MIN);

if(s1<=s2)

{

A[s]=s2;

j--;

}

else

{

A[s]=s1;

i--;

}

}

}

int main()

{

int m,n;

cout<<"Enter size of array A: ";

cin>>m;

cout<<"Enter size of array B: ";

cin>>n;

vector<int> A(m+n),B(n);

cout<<"Enter elements of array A\n";

for(int i=0;i<m;i++)

cin>>A[i];

cout<<"Enter elements of array B\n";

for(int i=0;i<n;i++)

cin>>B[i];

merge(A,m,B,n);

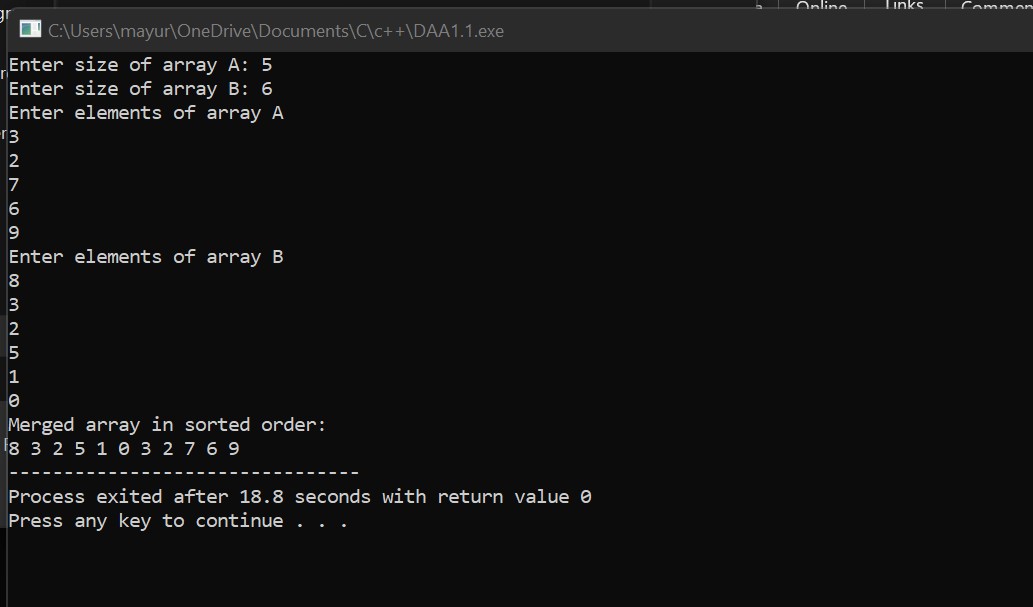
cout<<"Merged array in sorted order:\n";

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

cout<<A[i]<<" ";

}

**Output:**



1. **Complexity of proposed algorithm (Time & Space)**

* Time Complexity: O(m+n)
* Space Complexity: O(m+n) [because it is provided that A has larger buffer at the back]

1. **Your comment (How your solution is optimal?)**

* The proposed algorithm is space efficient
* The alternative solution is by making another array and adding all elements in it and then sorting it. But it would take O(nlogn) Time and O(m+n) space as well.

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

Ans:

1. **Algorithm: (Pseudocode)**

* Make the hashmap of the string using sorted string as a key
* The sorted string version will be same for all anagrams
* Traverse through hashmap and add all values to another array/vector

1. **Code snapshots of implementation**

#include <bits/stdc++.h>

using namespace std;

vector<string> sortAnagrams(vector<string>& stringArray)

{

map<string,vector<string> > mp;

for(string s:stringArray)

{

string st=s;

sort(st.begin(),st.end());

mp[st].push\_back(s);

}

vector<string> answer;

for(auto iterator:mp)

{

for(string s:iterator.second)

answer.push\_back(s);

}

return answer;

}

int main()

{

vector<string> str;

str.push\_back("bal");

str.push\_back("all");

str.push\_back("lal");

str.push\_back("lab");

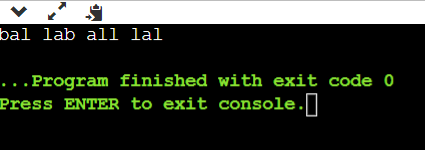
vector<string> ans = s.sortAnagrams(str);

for(string i:ans)

cout << i <<" ";

}

**Output:**



1. **Complexity of proposed algorithm (Time & Space)**

* Time Complexity: O(nmlogm) [considering M as length of largest string]
* Space Complexity: O(n+m)

1. 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)

Ans:

1. **Algorithm: (Pseudocode)**

* Find the pivot Index (Pivot: Index where the array is in descending order For Example for given array pivot index is 4 (Element: 25)).
* So, array from 0 to pivot and pivot to n-1 is in sorted order.
* Now apply binary search on both array and find the index of given element.
* This algorithm works when array values are distinct

1. **Code snapshots of implementation**

#include <bits/stdc++.h>

using namespace std;

int find\_pivot(int a[],int s)

{

if(a[0]<a[s-1])

return 0;

int i=0,j=s-1;

int mid;

while(i<j)

{

mid=(i+j)/2;

if(a[i]<a[mid])

i=mid;

else

j=mid;

}

return mid+1;

}

int binary\_search(int a[],int low,int high,int sv)

{

int mid;

while(low<=high)

{

mid=(low+high)/2;

if(a[mid]==sv)

return mid;

else

{

if(a[mid]<sv)

low=mid+1;

else

high=mid-1;

}

}

return -1;

}

int main()

{

int n;

cout<<"Enter size of array: ";

cin>>n;

int a[n];

int ans=-1;

cout<<"\n Enter elements in array: ";

for(int i=0;i<n;i++)

cin>>a[i];

int sv;

cout<<"\nEnter search value: ";

cin>>sv;

int ind=find\_pivot(a,n);

if(ind==0)

ans=binary\_search(a,0,n-1,sv);

else

{

if(sv>=a[0])

ans=binary\_search(a,0,ind-1,sv);

else

ans=binary\_search(a,ind,n-1,sv);

}

if(ans==-1)

cout<<"\n"<<sv<<" is not present in array";

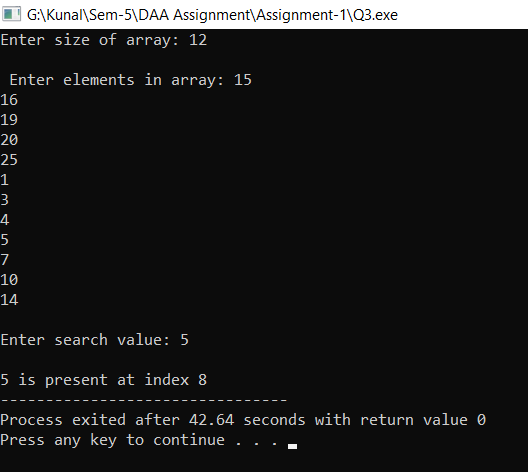
else

cout<<"\n"<<sv<<" is present at index "<<ans;

return 0;

}

**Output:**



1. **Complexity of proposed algorithm (Time & Space)**

* Time Complexity: O(logn)
* Space Complexity: O(n)

1. **Your comment (How your solution is optimal?)**

* This algorithm is an extension of Binary Search Algorithm
* This algorithm works in O(logn)

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

Ans:

1. **Split 20 GB files into K files**

Since we have limited memory, we first need to split 20 GB files into K sub-arrays, each of which contains M integers that can fit into memory available.

1. **Sort K files individually, O(KxMlogM)**

Now, we can sort each chunk with size of M either through quick sort or merge sort. In this case, in place Quick Sort will be more efficient. Running tie of this step will be O(KxMlogM), where K is the number of sub-arrays and M is the number of integers in each sub-array.

1. **Perform the K way merge, O(NlogK)**

Now, we have K partially sorted array of integers, which can be merged through K way merge.

Running time of K way merge will be O(NlogK), where N is the number of integers in 20 GB file and K is the number of partially sorted array.

Overall time complexity of this sorting algorithm will be **O(KxMlogM)**

1. 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

Ans:

1. **Algorithm: (Pseudocode)**

* Use binary search but skip the empty strings when searching
* When we find midpoint go to left until we don’t get non-empty string.
* When we find midpoint go to right until we don’t get non-empty string.
* Decide further search space based on values of left and rightmost non-empty strings

1. **Code snapshots of implementation**

#include <bits/stdc++.h>

using namespace std;

int search\_string(vector<string>& input, string& target,int s,int e)

{

if(s>e)

return INT\_MIN;

int h=(s+e)/2;

int found=INT\_MIN;

if(input[h].length()==0)

{

found=search\_string(input,target,s,h-1);

if(found==INT\_MIN)

found=search\_string(input,target,h+1,e);

}

else

{

if(input[h]==target)

found=h;

else if(input[h].compare(target)<0)

found=search\_string(input,target,h+1,e);

else

found=search\_string(input,target,s,h-1);

}

return found;

}

int main()

{

vector<string> input;

string s;

input.push\_back("at");

input.push\_back("");

input.push\_back("");

input.push\_back("ball");

input.push\_back("");

input.push\_back("");

input.push\_back("car");

input.push\_back("");

input.push\_back("");

input.push\_back("dad");

input.push\_back("");

input.push\_back("");

cout<<"Enter Search String: ";

cin>>s;

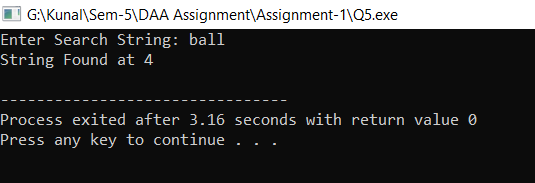
int found=search\_string(input,s,0,input.size()-1);

cout<<"String Found at "<<found+1<<endl;

return 0;

}

**Output:**



1. **Complexity of proposed algorithm (Time & Space)**

* Time Complexity: O(n) [Worst Case] O(logn) [Best Case]
* Space Complexity: O(n)

1. **Your comment (How your solution is optimal?)**

* This algorithm is an extension of binary search.
* This algorithm performs worst when there is more gap of empty strings in given array.

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

Ans:

1. **Algorithm: (Pseudocode)**

* As rows and columns are already sorted try to search from first row fully
* If target is greater than current element then search in next row
* If target is lesser than current elements then search in previous column

1. **Code snapshots of implementation**

#include <bits/stdc++.h>

using namespace std;

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

{

int n=matrix.size(), m=matrix[0].size();

int i=0,j=m-1;

while(i<n && j>=0)

{

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

return {i,j};

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

j--;

else

i++;

}

return {-1,-1};

}

int main()

{

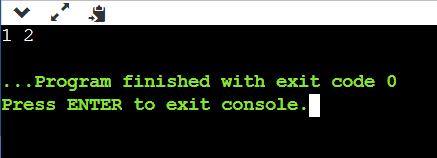
vector<vector<int> > matrix = {{1,2,5},{3,4,7},{5,5,8}};

pair<int,int> p = searchMatrix(matrix,7);

cout << p.first <<" "<< p.second;

}

**Output:**



1. **Complexity of proposed algorithm (Time & Space)**

* Time Complexity: O(m+n) [m-number of columns n-number of rows]
* Space: O(mn)

1. **Your comment (How your solution is optimal?)**

* For naïve brute force approach time complexity for search in matrix will be O(n2)

1. 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)

Ans:

1. **Algorithm: (Pseudocode)**

* We will sort the given people array by their weight
* In case of tie we will sort by using their height
* And will do one traversal for finding valid o=indices of people that can be part of the tower

1. **Code snapshots of implementation**

#include <bits/stdc++.h>

using namespace std;

bool comparator(pair<int,int> &A, pair<int,int> &B)

{

if(A.first < B.first)

return 1;

else if(A.first > B.first)

return 0;

return A.second <= B.second;

}

int maxNumberofPeople(vector<pair<int,int> >& people)

{

sort(people.begin(), people.end(), comparator);

int maxPeople=1;

int previous=0;

for(int i=1;i<people.size();i++)

{

if(people[i].first > people[previous].first && people[i].second > people[previous].second)

{

maxPeople++;

previous =i;

}

}

return maxPeople;

}

int main()

{

vector<pair<int,int> > people;

people.push\_back({65, 100});

people.push\_back({70, 150});

people.push\_back({56, 90});

people.push\_back({75,190});

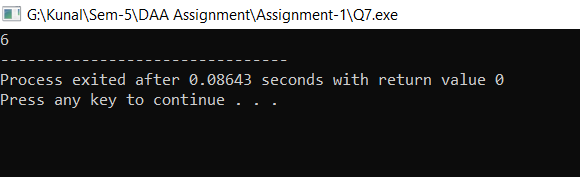
people.push\_back({60, 95});

people.push\_back({68,110});

cout << maxNumberofPeople(people);

}

**Output:**



1. **Complexity of proposed algorithm (Time & Space)**

* Time Complexity: O(nlogn)
* Space Complexity: O(n)

1. **Your comment (How your solution is optimal?)**

* This algorithm works in O(nlogn) time.
* If we try to solve given problem with sorting and with trying each and every subsequence of people to check it is valid or not it may take exponential amount of time and also it can reach higher level space complexity as well.

1. 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:

1. **Algorithm: (Pseudocode)**

* We will use binary search tree for each time insertion of data of stream
* Also we will maintain the number of elements in left subtree while inserting.
* Answer to the getRankOfNumber will be the size of left subtree of a given node

1. **Code snapshots of implementation**

#include <bits/stdc++.h>

using namespace std;

class BSTNode

{

public:

int data;

BSTNode \*left,\*right;

int leftSize;

BSTNode(int data)

{

this->data = data;

left = right = NULL;

leftSize = 0;

}

BSTNode\* insert(BSTNode\* root, int data)

{

if (!root)

return new BSTNode(data);

if (data <= root->data)

{

root->left = insert(root->left, data);

root->leftSize++;

}

else

root->right = insert(root->right, data);

return root;

} int getRank(BSTNode\* root, int x)

{

if (root->data == x)

return root->leftSize;

if (x < root->data)

{

if (!root->left)

return -1;

else

return getRank(root->left, x);

}

else

{

if (!root->right)

return -1;

else

{

int rightSize = getRank(root->right, x);

if(rightSize == -1 ) return -1;

return root->leftSize + 1 + rightSize;

}

}

return -1;

}

};

int main()

{

BSTNode\* root = new BSTNode(5);

root->insert(root,10);

root->insert(root,4);

root->insert(root,4);

root->insert(root,9);

cout << root->getRank(root,9) << endl;

root->insert(root,7);

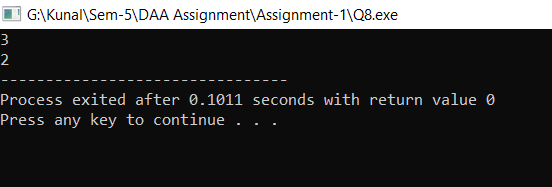
root->insert(root,13);

root->insert(root,3);

cout << root->getRank(root,4);

}

**Output:**



1. **Complexity of proposed algorithm (Time & Space)**

* Time Complexity: O(logn) [for both insert and getRank]
* Space: O(n) [n-number of nodes]

1. **Your comment (How your solution is optimal?)**

* If we try to solve given problem using vector and sorting it after every insertion its worst-case time complexity will be O(n2logn)