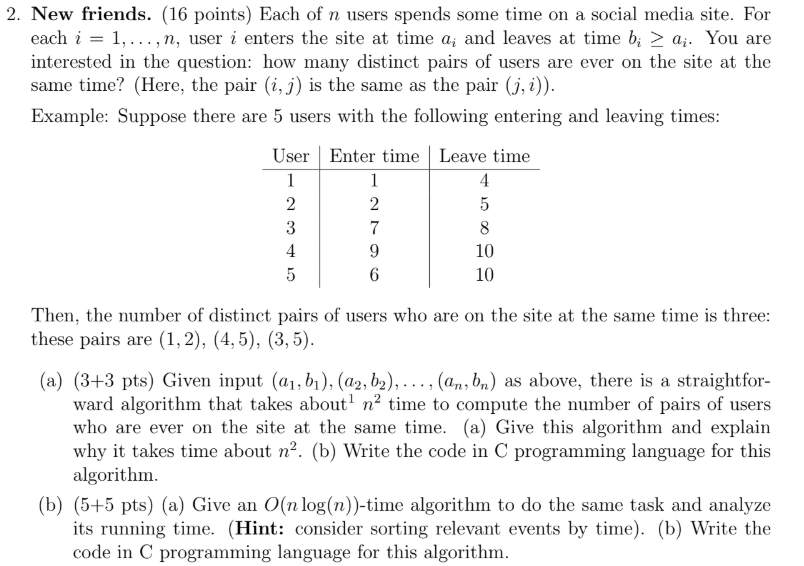


I hope to advanced algorithms and improve my problem solving skills to understand, analyze and solve such problems that usually came in challenges, contests, interviews, etc. This will also help me in future as this course will make us realize the bad coding mistakes people usually make and how to prevent our self from doing the same.



(a). Algorithm

Algorithm **I**: Time Complexity O(n^2)

This Program will check how many unique no of pairs of students will be

available onilne based on thier entry and exit times

The Algorithm Uses Two Nested Loops Which Go over N times

**Input**:

n <= total no of students

people <= [(entry, exit), ...] Array defining a students' entry/exit time

no\_of\_pairs <= 0

**FOR** i from 0 to n - 1:

**FOR** j from (i+1) to n - 1:

**IF** (person1.entry < person2.exit && person2.entry < person1.exit):

Increment no\_of\_pairs by 1

(a). C Program:

#include <stdio.h>

typedef struct person{

int entry;

int exit;

} Person;

int main(int argc, char const \*argv[]){

int noOfPeople;

scanf("%d", &noOfPeople);

Person people[noOfPeople];

for (int i = 0; i < noOfPeople; i++){

scanf("%d %d", &people[i].entry, &people[i].exit);

}

int distinctPairs = 0;

for (int i = 0; i < noOfPeople; i++){

for (int j = i + 1; j < noOfPeople; j++){

if(people[i].entry < people[j].exit && people[j].entry < people[i].exit){

distinctPairs++;

}

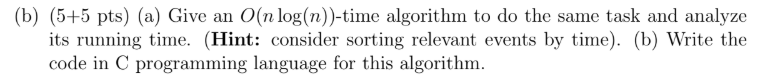
}

}

printf("Possible distinct pairs are %d", distinctPairs);

return 0;

}



1. . Prerequisite Algorithms:

1-Binary Search

Algorithm **BINARYSEARCH**: Time Complexity O(log(n))

This Program will search for the greatest no. less than or equal

to the Value given in the Sorted Array

**BINARYSEARCH**(Array, val)

l <= 0, r <= Array.Length -1

**WHILE** l < r:

**GET** Middle Element and compare with val

mid <= l + (r - l) / 2;

**IF** val is less than Array[mid]:

**SEARCH** for val in right half of sorted Array:

l <= mid + 1

**ELSE**

**SEARCH** for val in left half of sorted Array:

r <= mid

**IF** Array[l] == val:

RETURN l

**ELSE**

RETURN l - 1

2-Merge Sort

Algorithm **MERGESORT**: Time Complexity O(n\*log(n))

**MERGESORT**(arr[], l, r)

**IF** l == r

**RETURN**

**ELSE**

Find the middle point to divide the array into two halves:

middle m <= (l+r)/2

Do **MERGESORT** for first half:

Call **MERGESORT**(arr, l, m)

Do MERGESORT for second half:

Call **MERGESORT**(arr, m+1, r)

MERGE the two halves sorted in step 2 and 3:

Call **MERGE**(arr, l, m, r)

Algorithm MERGE: Time Complexity O(n)

**MERGE**(Array1, Array2, l, m, r):

// Array1.Length is m - l + 1, Array2.Length is r - m

Create a Duplicate array of Total Length:

Temp <= Array of Size [ Array1.Length + Array2.Length ]

index <= 0

Initialize Index for both Array:

i <= 0, j <= 0

**WHILE** index < Temp.Length:

**IF** any Array becomes empty

**IF** i == Array1.Length :

Temp[index++] <= Array2[j++]

**IF** j == Array2.Length :

Temp[index++] <= Array1[i++]

**CONTINUE**

INSERT the lower Element:

**IF** Array1[i] < Array2[j]:

Temp[index++] <= Array1[i++]

**ELSE**

Temp[index++] <= Array2[j++]

(b) Algorithm

Algorithm II: Time Complexity **O(n\*log(n))**

This Program will check how many unique no of pairs of students will be

available online based on their entry and exit times

The First Part of the Problem User MERGESORT which takes **O(n\*log(n))** Time

In Second Part we BINARYSEARCH the Value in Array over a loop so

net Complexity will be **O(n) \* O(log(n))** => **O(n\*log(n))**

Input:

n <= Total no of students

Array defining a student structure with entry/exit time

people <= [(entry, exit), ...]

no\_of\_pairs <= 0

Apply MERGESORT on people Array

**FOR** i from 0 to n - 1:

GET Value When the ith person Leaves

val <= people[i].exit

Find How many people were logged in before ith person left

found\_index <= **BINARYSEARCH**(people, val)

Add No. of Pairs of the respective person

**Increment** no\_of\_pairs by found\_index - i

(b) C Program

#include <stdio.h>

typedef struct person {

int entry;

int exit;

} Person;

int binarySearchPersonEntry(Person Array[], int val, int start, int end)

{

while (start <= end) {

int m = start + (end - start) / 2;

if (Array[m].entry == val)

return m;

if (Array[m].entry < val)

start = m + 1;

else

end = m - 1;

}

return start - 1;

}

int compare(Person p1, Person p2){

if (p1.entry < p2.entry) {

return 1;

}

else {

if (p1.exit < p2.exit) {

return 1;

}

else {

return 0;

}

}

}

void mergeSort(Person Array[], int start, int end)

{

if (start == end) {

return;

}

int mid = (start + end - 1) / 2;

mergeSort(Array, start, mid);

mergeSort(Array, mid + 1, end);

// Merging Arrayay

int length = end - start + 1;

Person Temp[length];

for (int i = 0; i < length; i++){

Temp[i] = Array[i + start];

}

int i = 0, j = 0, m = mid - start + 1;

for (int c = start; c <= end; c++){

if(i >= m){

Array[c] = Temp[m + j++];

}

else if (j + mid >= end)

{

Array[c] = Temp[i++];

}

else if (compare(Temp[i], Temp[m + j]))

{

Array[c] = Temp[i++];

}

else

{

Array[c] = Temp[m + j++];

}

}

}

int main(int argc, char const \*argv[])

{

int noOfPeople;

scanf("%d", &noOfPeople);

Person people[noOfPeople];

for (int i = 0; i < noOfPeople; i++){

scanf("%d %d", &people[i].entry, &people[i].exit);

}

mergeSort(people, 0, noOfPeople - 1);

int distinctPairs = 0;

int begin = 0, end = 0, current\_online = 0;

for (int i = 0; i < noOfPeople; i++){

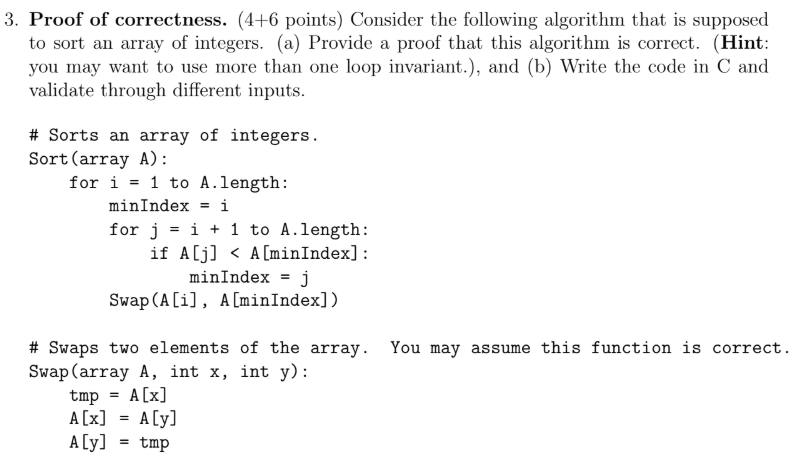
int k = binarySearchPersonEntry(people, people[i].exit - 1, i+1, noOfPeople-1) - i;

distinctPairs = k;

}

printf("No. of distinct pairs are: %d.\n", distinctPairs);

}



**Inductive Hypothesis:** After iteration of *i* in the outer loop the array A[0 : I] will be sorted.

**Base Case:** At beginning of outerLoop *i = 1* The Array A[0:1] will containsingle element which is already sorted since it is the only element in the Array.

**Inductive Step:** Suppose that the inductive hypothesis holds for *i-1* so A[0 : i-1] is sorted and all the elements in Array A[i : end] are larger than all elements in the Previous Array, after the i-1th iteration. We want to show that Array A[0 : i] is sorted with having the least i members of Array A after the *ith* iteration.

Suppose that *kth* element is the smallest integer in the remaining Unsorted Array A[i : end]. Then the effect of the inner loop is to swap the position of ith element with kth element, which will convert the existing array

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A[0] | A[1] | … | A[i-1] | A[i] | … | A[k] | … | A[end] |

Into following:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A[0] | A[1] | … | A[i-1] | A[k] | … | A[i] | … | A[end] |

We can now claim that the following list is sorted:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A[0] | A[1] | A[2] | … | A[i-1] | A[k] |

This is because the first i-1 elements are already sorted and Since all the remaining elements were larger than all the elements in Array A[0 : i-1]. Hence A[k] > A[i-1]. Also Since A[k] was the smallest in the Array A[i : end] hence all the remaining elements A[i+1: end] are larger than A[k] Satisfying the second condition. Since Both the conditions are satisfied and element A[k] is in the right place, this proves the claim.

Thus, After the ith iteration completes the Array A[0: i] is sorted and this establishes the inductive hypothesis for i.

Hence the Correctness of Algorithm is Proved.

(b)- C Program

#include <stdio.h>

void swap(int \*a, int \*b){

\*a = \*a + \*b;

\*b = \*a - \*b;

\*a = \*a - \*b;

}

void sort(int arr[], int length){

for (int i = 0; i < length; i++){

int pos = i;

for (int j = i + 1; j < length; j++){

if(arr[j] < arr[pos]){

pos = j;

}

}

if(i != pos){

swap(&arr[i], &arr[pos]);

}

}

}

int main(){

int n;

printf("Enter length: ");

scanf("%d", &n);

int arr[n];

printf("Enter numbers: ");

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

scanf("%d", &arr[i]);

}

sort(arr, n);

printf("Sorted Array: ");

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

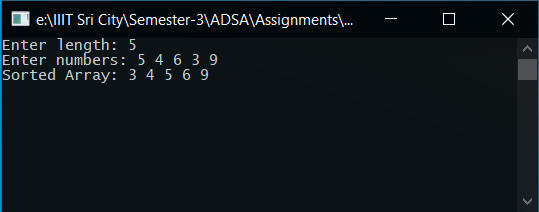
printf("%d ", arr[i]);

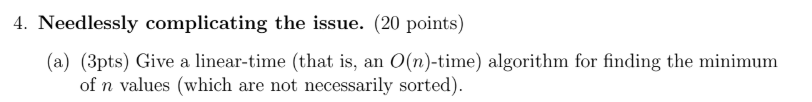
}

printf("\n");

}

Output





(A)- Algorithm

**Algorithm**: Time Complexity **O(n)**

This Program will Search the Array for the least element and will return it

The Algorithm Uses a Single Loop in which we will match each element

with the minimum element and update the minimum element if turns out to be smaller

Since Every element is matched once the complexity will be **O(n)**

Input:

n <= total no of students

arr <= [...] Array of Numbers

**DEFINE** first element as the least

min = arr[0]

**FOR** i from 1 to n:

**IF** arr[i] is less than min:

**DEFINE** min as arr[i]

min <= arr[i]

RETURN min

(b)

**Algorithm**: Time Complexity **O(n)**

This Program will Search the Array for the least element and will return it

The Algorithm Uses a Single Loop in which we will match each element

with the minimum element and update the minimum element if turns out to be smaller

Since Every element is matched once the complexity will be **O(n)**

Input:

n <= total no of students

arr <= [...] Array of Numbers

**DEFINE** first element as the least

min = arr[0]

**FOR** i from 1 to n:

**IF** arr[i] is less than min:

**DEFINE** min as arr[i]

min <= arr[i]

RETURN min