ADVANCED DATA STRUCTURES AND ALGORITHMS
FALL 2019
ASSIGNMENT - 1

Submitted By -

Sayam Kumar
S20180010158
Sec - A

Ans - 1.

1. **The skills that I wish to learn -** The major skills that I wish to learn is to understand a problem in a better way. I should be able to come up with such an implementation of the problem that will be time as well as space efficient. All the algorithms that are implemented in real life like B, B+ trees, graphs, dynamic programming should be taught as these will be helpful in the software industry.

2. **Expected benefits -** This will help me in improving my performance in various coding competitions which are being hosted on codechef, hackerrank etc. I will be able to visualize properly how the real life softwares are implemented. I will also be able to crack coding interviews of the top-notch companies.

Ans - 2. a) The question states to find the users which are ever on the site at the same time. The algorithm for this problem in $O(n^2)$ complexity is based on brute force approach. We will check each and every possible combination of pair of users. If the entering time of the user is before the leaving time of another user, we have found a pair. Repeatedly performing the task gives all the possible combinations.

Pseudo Code →

```
no_of_users ← int( input() )
users ← int( input() ) // a 2-D array storing entering and leaving time of users.
```

```
for i ← 0, no_of_users:
        for j ← i+1, no_of_users:
                if ( users[ i ][ 1 ] >= users[ j ][ 0 ] )
                        print( i + 1, j + 1 )
```

This brute force approach has time complexity O(n^2) because for the first iteration, the inner loop runs n-1 times. For the second iteration of the outer loop, the inner loop runs n-2 times. We continue the same process and sum up time taken by all the iterations -

Time complexity = (n-1) + (n-2) + (n-3) + … + 2 + 1

Now reversing the sum = 1 + 2 + 3 + ... + (n-3) + (n-2) + (n-1)

So, the final ans = n*(n-1)/2;

Taking the Big O(ans), the time complexity via this approach is O(n^2).

The algorithm explained above is implemented in C language →

```c
#include<stdio.h>
int main()
{
   int no_of_users;
   printf("Enter the no of users: ");
   scanf("%d", &no_of_users);
   printf("Enter the entry & leaving time of %d
users:\n",no_of_users);
   int users[no_of_users][2];
   for(int i = 0; i < no_of_users; i++)
       scanf("%d %d", &users[i][0], &users[i][1]);
   printf("The index of users at the same time are:\n");
   for(int i = 0; i < no_of_users; i++)
   {
       for(int j = i + 1; j < no_of_users; j++)
       {
           // checking each and every pair
           if (users[i][1] >= users[j][0])
               printf("%d %d\n", i + 1, j + 1);
```

```
            }
        }
}
```

b) An efficient way to solve this problem is to apply the concept of divide and conquer algorithm. We will implement merge sort algorithm in which we divide the problem into subproblems of the same size and then conquer all the subproblems via merging them all. The sort should accomplish the ordering of users with respect to their entry time. Then we iterate over all the leaving time of users and find its correct place using binary search algorithm in the entering time. The concept behind this is when we have found position of leaving time in the entering time aray, then all the users marked at positions before this, will be at the same time.

Running Time Analysis →

1. By making use of Master Theorem →

1. As at each step of the merge sort, we divide the problem into 2 subproblems of size half of the initial problem. Now, dividing the array by a factor of 2 repeatedly gives log(n) levels. For each level, we wait till the merging of its subproblems. The time complexity for merging is O(n) as we are iterating over the final array while comparing the two merged sorted arrays. Till now, sorting of array has complexity as O(n*log(n)).
2. Now according to the question, we need to count the number of pairs of users online at the same time. For each user, we analyse the position of leaving time in the sorted array of entry time using binary search and increase the counter by the same difference. As for each user, we are searching using binary search, the time complexity for this task is O(n*log(n)).
3. So the overall complexity of this problem is O(n*log(n)) + O(n*log(n)). The first n*log(n) for sorting and the other one is for implementing binary search for each element.

2. Using Substitution Method →

Let T(n) be the time taken by the recurrence relation for merge sort. By using substitution →

T(n) = 2*T(n/2) + n

= 4*T(n/4) + 2n

= 8*T(n/8) + 3n

For t^{th} level:

T(t) = 2^t*T($n/2^t$) + 3t

By summing the work done on all levels → T(n) = n*1 + $\log_2$ (n)* n

So, the time complexity for merge sort = O(n*log(n))

Conclusion → The overall complexity of this problem is O(n*log(n)).

b) The implementation of the above described algorithm is written in C language as follows →

```c
#include<stdio.h>
void merge(int a[][2],int left_index,int middle_index,int right_index)
{
    int n1 = middle_index - left_index + 1;
    int n2 = right_index - middle_index;
    int left_part[n1][2], right_part[n2][2];
    // copy the left part
    for(int i=0;i<n1;i++)
    {
        left_part[i][0] = a[i + left_index][0];
        left_part[i][1] = a[i + left_index][1];
    }
    // copy the right part
    for(int i=0;i<n2;i++)
    {
```

```
        right_part[i][0] = a[middle_index + 1 + i][0];
        right_part[i][1] = a[middle_index + 1 + i][1];
    }

    // compare values in i, j and fill in final array
    int i = 0, j = 0, k = left_index;
    // looping over the final array
    while(i < n1 && j < n2)
    {
        if (left_part[i][0] < right_part[j][0])
        {
            a[k][0] = left_part[i][0];
            a[k][1] = left_part[i][1];
            i++;
        }

        else
        {
            a[k][0] = right_part[j][0];
            a[k][1] = right_part[j][1];
            j++;
        }
        k++;
    }
    while(i < n1)
    {
        a[k][0] = left_part[i][0]; // if some elements are left
        a[k][1] = left_part[i][1];
        i++;
        k++;
    }
```

```
    while(j < n2)
    {
        a[k][0] = right_part[j][0];
        a[k][1] = right_part[j][1];
        j++;
        k++;
    }
}

void merge_sort(int a[][2], int left_index, int right_index)
{
   if (left_index < right_index)
   {
       int middle_index = left_index + (right_index - left_index)/2;
       merge_sort(a, left_index, middle_index); // left half
       merge_sort(a, middle_index + 1, right_index); //right half
       merge(a, left_index, middle_index, right_index); // merge
   }
}

int main()
{
   int no_of_users;
   printf("Enter the no of users: ");
   scanf("%d", &no_of_users);
   printf("Enter entry & leaving time of %d users:\n",no_of_users);
   int users[no_of_users][2];
   for(int i = 0; i < no_of_users; i++)
       scanf("%d %d", &users[i][0], &users[i][1]);
   merge_sort(users, 0, no_of_users-1);
   int count = 0;
```

```
   for(int i=0;i<no_of_users;i++)
   {
       int leaving_time = users[i][1];
       int low = 0, high = no_of_users-1, mid;
       while (low < high)
       {
           mid = (low + high)/2;
           if (users[mid][0] == leaving_time)
               break;
           else if (users[mid][0] > leaving_time)
               high = mid;
           else
               low = mid + 1;
       }
       int position = ( high + low )/2;

       if (leaving_time > users[position][0])
        {
            //to insert at the last position
            position += 1;
        }
        count += (position - i -1);
   }
    printf("No of pairs of users online at same time are
%d.\n",count);
}
```

Ans - 3. **a)** The pseudo code given in the question is the algorithm for selection sort. In this algorithm, for each iteration, we select the minimum element of the array and swap it with the current index. Finally after all the iterations, we will have the array sorted in ascending order.

Pseudo Code →

```
Swap( int x, int y )
        temp ← x
        x ← y
        y ← temp
Sort ( A, no_of_elements )
        for i ← 0, no_of_elements:
                minIndex ← i
                for j ← i+1, no_of_elements:
                        if ( A[ j ] < A[ minIndex ] )
                                minIndex ← j
                Swap( A[ i ], A[ minIndex ] )
no_of_elements ← int( input() )
A ← int( input().split() ) // A is the array of elements
Sort ( A, no_of_elements )
print( A )
```

Proof →

Loop Invariant → The minimum element of the remaining array gets appended to the sorted array.

Inductive Hypothesis → Appending the minimum element of remaining array to sorted array, leading to sorting of the entire array in ascending array.

Base Case → An array with a single element is already sorted.

Inductive Step → Let's denote the minimum element in the remaining array by min after k^{th} step. This element is greater than all the elements inserted in sorted array.

Now, to prove for k+1 →

In the $(k+1)^{th}$ iteration, we select the minimum element in the unsorted array. This element is greater than the minimum element found in the k^{th} iteration because otherwise, this minimum element gets selected in k^{th} iteration. This means that A[k+1] >= A[k] for all k<=k+1 ie k=[1,2,3, … ,k]. So, the loop invariant is true for all the iterations. Therefore by induction, the selection of

minimum element and appending it to left part leads to entire sorting of the array.

Hence proved!

b) The code for selection sort written in C language, is given below →

```c
#include<stdio.h>
void Swap(int *n1, int *n2)
{
   // swapping the values stored in n1 and n2
   int temp = *n1;
   *n1 = *n2;
   *n2 = temp;
}

void Sort(int A[], int no_of_elements)
{
   for(int i = 0; i < no_of_elements; i++)
   {
       int minIndex = i; // assign i to minIndex
       for(int j = i + 1; j < no_of_elements; j++)
       {
           // find the smallest element
           if (A[j] < A[minIndex])
           {
               minIndex = j;
           }
       }
```

```
        // swap with the smallest element
        Swap(&A[i], &A[minIndex]);
    }
}

int main()
{
    int no_of_elements;
    printf("Enter the no of elements: ");
    scanf("%d", &no_of_elements);
    int A[no_of_elements];
    printf("Enter the elements: ");
    for(int i = 0; i < no_of_elements; i++)
    {
        scanf("%d", &A[i]);
    }
    Sort(A, no_of_elements);
    printf("Sorted array is ");
    for(int i = 0; i < no_of_elements; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
}
```

Validating through different inputs -

```
Sayam at My_MacBook in assignment_1
$ gcc selection_sort.c

Sayam at My_MacBook in assignment_1
$ ./a.out
Enter the no of elements: 10
Enter the elements: 10 9 8 7 6 5 4 3 2 1
Sorted array is 1 2 3 4 5 6 7 8 9 10

Sayam at My_MacBook in assignment_1
$ ./a.out
Enter the no of elements: 5
Enter the elements: 5 4 6 3 7
Sorted array is 3 4 5 6 7

Sayam at My_MacBook in assignment_1
$ ./a.out
Enter the no of elements: 6
Enter the elements: -1 -3 15 -6 7 19
Sorted array is -6 -3 -1 7 15 19

Sayam at My_MacBook in assignment_1
$
```

Ans - 4. a) The linear time algorithm for finding the minimum element in the array will be iterating over all the elements and updating min variable whenever a minimum element with respect to the previous one is found.

Pseudo Code →

```
no_of_elements ← int( input() )
A ← int(input().split() ) // input the elements of the array
min = A[ 0 ]
for i← 1, no_of_elements
        if A[ i ] < min
                min = A[ i ]
```

print(min)

b) It is true that in order to find the minimum of n items, any algorithm must do at least n operations in the worst case. There are multiple options like using a linear search, first sort and then take the first element, divide recursively by a factor of 2 and then start comparing. In any case, we need to compare at least n times. So, the given statement is true.

c) The implementation of the above described algorithm is written in C language as follows →

```
#include<stdio.h>
int main()
{
   int no_of_elements;
   printf("Enter the no of elements: ");
   scanf("%d", &no_of_elements);
   int A[no_of_elements];
   printf("Enter the elements: ");
   for( int i = 0; i < no_of_elements; i++ )
   {
       scanf("%d", &A[i]);
   }
   int min = A[0];
   for( int i = 1; i < no_of_elements; i++ )
   {
       if ( A[i] < min )
       {
           min = A[i];
```

```
        }
    }
    printf("Minimum element of the array is %d.\n", min);
}
```

d) Given the algorithm →
Input: List A = { a_1, … , a_n } of n items
Output: min_i { a_1, … , a_n }
if n=1 then
    return A[0]
A_1 = A[0: n/2]
A_2 = A[n/2: n]
return min(findMinimum(A_1), findMinimum(A_2))
The blank should be filled with A[0] because if there is only one element left in the array, that itself serve as minimum element. This is because, we are only considering the minimum of both left and right subarrays.

e) Running Time Analysis →

1. The algorithm is dividing the entire array into parts of two recursively and thereby time taken for this purpose is $\log_2(n)$.
2. From bottom to top, comparison of left and right subarrays is made to check for minimum element. This has just constant time complexity.
3. But in order to divide the array and copy the elements take O(n) complexity.
4. By keeping in view, all the above mentioned three aspects, the total time taken will be c*n + log(n) + k where c and k are constants.

Conclusion → The running time of this algorithm is O(n). The solution of part a is based on a linear search but this algorithm is of divide and conquer type. But both these algorithms has time complexity of O(n).

f) The implementation of the above described algorithm is written in C language as follows →

```c
#include<stdio.h>
int min(int a, int b)
{
   if (a > b)
       return b;
   return a;
}

int findMinimum(int A[], int no_of_elements)
{
   if (no_of_elements == 1)
   {
       return A[0];
   }
   int n1 = (no_of_elements+1)/2;
   int n2 = no_of_elements - n1;
   int A1[n1], A2[n2];
   for(int i = 0; i < n1; i++)
   {
       A1[i] = A[i];
   }
   for(int i = 0; i < n2; i++)
   {
       A2[i] = A[n1 + i];
   }
   return min(findMinimum(A1, n1), findMinimum(A2, n2));
}
```

```
int main()
{
   int no_of_elements;
   printf("Enter the no of elements: ");
   scanf("%d", &no_of_elements);
   int A[no_of_elements];
   printf("Enter the elements: ");
   for( int i = 0; i < no_of_elements; i++ )
   {
       scanf("%d", &A[i]);
   }
   int minimum = findMinimum(A, no_of_elements);
      printf("Minimum element of the array is %d.\n",
minimum);
}
```

Ans - 5. a) The main task is to find a local minimum in the array provided that all elements are distinct. For this, I have implemented a recursive algorithm which compares the middle of array with mid + 1 and mid - 1. The program moves towards the minimum of these three and the rest of the array is rejected. At each step, the size of array is reduced by two.

Pseudo Code →

```
int findLocalMinimum(int A[], int left, iny right )
mid = (left+right)/2
if left==right
        return A[left]
elif mid==left
        if A[mid] < A[mid+1]
```

```
            return A[mid]
        else return A[mid+1]
elif A[mid+1] is smaller
        return findLocalMinimum(A, mid+1, right)
elif A[mid-1] is smaller
        return findLocalMinimum(A, left, mid)
else
        return A[mid]

no_of_elements ← int( input() )
A ← array of n distinct integers
min ← findLocalMinimum(A, 0, no_of_elements-1)
print(min)
```

b) Proof of Correctness → Since we are dividing the entire array by a factor of 2, I will formally prove the algorithm using induction.

Inductive Hypothesis → The recursive function returns the local minimum at mid of left and right indices.

Base Case → When there is only one element in the array, it will serve as local minimum.

Inductive Step → Since all the elements in the array are distinct, as stated by the question, taking the mid as (left+right)/2, and start comparing with the element at the index mid-1 and mid+1.
If mid is smaller than both, then it is the local minimum. Otherwise check for the element at mid-1 and mid+1.
If A[mid] > A[mid-1] and A[mid+1] > A[mid-1], we move the direction towards left part of the array and reject the other half by making high = mid.
But if A[mid] >A[mid+1] and A[mid-1] > A[mid+1], we move the direction towards right part of the array and reject the other half by making low =

mid+1. Now by recursively calling the findLocalMinimum function, as all elements are distinct, we find the local minimum by moving either left or right.

Hence Proved!

c) Running Time Analysis →

As we are rejecting the other half of the array and moving the direction of program towards the smaller element, the recurrence relation for this algorithm can be written as →

$T(n) = T(n/2) + c$

This constant c is the time complexity for various comparisons in the if-else conditions .

Now solving futher the recurrence relation, T(n) can be written as →

$T(n) = T(n/2) + c$

$T(n) = T(n/4) + c + c$

$T(n) = T(n/8) + 3c$

$T(n) = T(n/16) + 4c$

…

$T(n) = T(n/2^{\log(n)}) + c \log(n)$ [because, we have $\log_2(n)$ levels]

$T(n) = 1 + c \log(n)$

So, $T(n) = O(\log(n))$

Conclusion → The running time of the above described algorithm is O(log(n)).

d) The implementation of the above described algorithm is written in C language as follows →

```c
#include<stdio.h>


int findLocalMinimum(int A[], int left, int right)
{
   int mid = (left+right)/2;
```

```
    if (left==right)
    {
        return A[left];
    }
    else if (mid==left)
    {
        if (A[mid] < A[mid+1])
            return A[mid];
        else return A[mid+1];
    }
    else if (A[mid+1] <= A[mid] && A[mid+1] <= A[mid-1])
    {
        // mid + 1 is smaller
        return findLocalMinimum(A, mid+1, right);
    }
    else if (A[mid-1] <= A[mid] && A[mid-1] <= A[mid+1])
    {
        // mid - 1 is smaller
        return findLocalMinimum(A, left, mid);
    }
    else if (A[mid] <= A[mid-1] && A[mid]<= A[mid+1])
    {
        // mid is local min
        return A[mid];
    }
    return 0;
}
```

```c
int main()
{
   int no_of_elements;
   printf("Enter the no of elements: ");
   scanf("%d", &no_of_elements);
   int A[no_of_elements];
   printf("Enter the elements: ");
   for( int i = 0; i < no_of_elements; i++ )
   {
       scanf("%d", &A[i]);
   }
   int minimum = findLocalMinimum(A, 0, no_of_elements-1);
   printf("Local min element of array is %d.\n",minimum);
}
```

B) Finding local minimum in the n*n grid

a) For achieving O(n) complexity, we keep dividing the size of array by size of 2 like a window and keep recursively checking for local minimum element.

Pseudo Code →

check (no_of_elements, A, x, y)

check for the boundaries and return 1 if it is a local minimum.

findMinimum (no_of_elements, A, start, left, right)

First find the minimum of window formed by first row, first column, last row, last column, middle row, and middle column.

Keep a track if any of the elements in the window is itself a local minimum.

Check for the indices of minimum element found in window and decide

which quadrant to choose. And proceed with recursion.

main()
no_of_elements ← int(input())
A ← 2-d array
print(findMinimum(no_of_elements, A, 0, no_of_elements, no_of_elements))

b) Proof →
Loop invariant → The local minimia found in a window is always less than its parent window.
Base Step → For 1*1 grid, the local minima is the single element itself.
Inductive Hypothesis → Since the flow of the program is towards the local minima, we keep on reducing the size of n*n grid by a factor of 2 and moving to the suitable quadrant. In this process, we will always be having local minima less than the previous one. So the loop invariant is maintained.
Hence we can state that reducing the size of grid will always result in a local minimum.
Hence proved!
c) Running Time Analysis →
For this recusive relation, let's denote the time taken to find local minimum of a n*n grid is T(n).
T(n) = T(n/2) + c(n) where c is a constant
T(n) = T(n/4) + c*n/2 + c*n
…
T(n) = T(1) + c (1 + 2 + 4 + … + n/2 + n)
So, the T(n) = O(n)
Hence, the time complexity of the above described algorithm is O(n).

d) The implementation of the above described algorithm is written in C language as follows →

```
#include<stdio.h>

int check(int no_of_elements, int A[][no_of_elements], int x, int y)
```

```
{
   // checking if the number is a local min
   int flag1=0, flag2=0, flag3=0, flag4=0;
   if (x-1 >= 0)
       flag1=1;
   if (y-1 >= 0)
       flag2=1;
   if (x+1 < no_of_elements)
       flag3=1;
   if (y+1 < no_of_elements)
       flag4=1;
   if (flag1)
   {
       if (flag2)
       {
           if (flag3)
           {
               if (flag4)
                    return (A[x][y]<A[x-1][y] && A[x][y]<A[x+1][y] &&
A[x][y]<A[x][y-1] && A[x][y]<A[x][y+1]);
                   return (A[x][y]<A[x-1][y] && A[x][y]<A[x+1][y] &&
A[x][y]<A[x][y-1]);
           }
           if (flag4)
                   return (A[x][y]<A[x-1][y] && A[x][y]<A[x][y+1] &&
A[x][y]<A[x][y-1]);
           return (A[x][y]<A[x-1][y] && A[x][y]<A[x][y-1]);
       }
       if (flag3)
       {
           if (flag4)
```

```
                    return (A[x][y]<A[x-1][y] && A[x][y]<A[x+1][y] &&
A[x][y]<A[x][y+1]);
        }
        if (flag4)
            return (A[x][y]<A[x-1][y] && A[x][y]<A[x][y+1]);
    }
    if (flag2)
    {
        if (flag3)
        {
            if (flag4)
                    return (A[x][y]<A[x][y-1] && A[x][y]<A[x+1][y] &&
A[x][y]<A[x][y+1]);
            return (A[x][y]<A[x][y-1] && A[x][y]<A[x+1][y]);
        }
    }
    if (flag3)
        return (A[x][y]<A[x][y+1] && A[x][y]<A[x+1][y]);
    return 1;

}

int findMinimum(int no_of_elements, int A[][no_of_elements], int
start, int left, int right)
{
    int min = A[0][0], x = 0, y = 0, flag = 0;
    // first_row
    for(int i=start;i<right;i++)
    {
        if (check(no_of_elements, A, start, i))
            return A[start][i];
```

```
        if (A[start][i]<min)
        {
            min = A[start][i];
            x=start;
            y=i;
        }
    }
    // last_row
    int factor = (left+start)%2?1:0;
    for(int i=start;i<right;i++)
    {
        if (check(no_of_elements, A, left-factor, i))
            return A[left-factor][i];
        if (A[left-factor][i]<min)
        {
            min = A[left-factor][i];
            x=left-factor;
            y=i;
        }
    }
    // first_column
    for(int i=start;i<left;i++)
    {
        if (check(no_of_elements, A, i, start))
            return A[i][start];
        if (A[i][start]<min)
        {
            min = A[i][start];
            x=i;
            y=start;
        }
```

```
}
//last_column
factor = (right+start)%2?1:0;
for(int i=start;i<left;i++)
{
    if (check(no_of_elements, A, i, right-factor))
        return A[i][right-factor];
    if (A[i][right-factor]<min)
    {
        min = A[i][right-factor];
        x=i;
        y=right-factor;
    }
}
// middle_row
int row = (left-start)/2;
for(int i=start;i<right;i++)
{
    if (check(no_of_elements, A, row, i))
        return A[row][i];
    if (A[row][i]<min)
    {
        min = A[row][i];
        x = row;
        y = i;
    }
}
// middle_column
int column = (right-start)/2;
for(int i=start;i<left;i++)
{
```

```
        if (check(no_of_elements, A, i, column))
            return A[i][column];
        if (A[i][column]<min)
        {
            min = A[i][column];
            x=i;
            y=column;
        }
    }
    // global min found in window
    if (x < (left+start)/2)
    {
        if (y < (right+start)/2)
                return findMinimum(no_of_elements, A, 0, (left+start)/2,
(right+start)/2); // 1
            return findMinimum(no_of_elements, A, (left+start)/2, left,
(right+start)/2); // 2
    }
    else
    {
        if (y < (right+start)/2)
        {
                        findMinimum(no_of_elements, A, (right+start)/2,
(left+start)/2, right); // 3
        }
            return findMinimum(no_of_elements, A, (left+start)/2, left,
right); // 4
    }
}

int main()
```

```
{
   int no_of_elements;
   //printf("Enter the no of elements: ");
   scanf("%d", &no_of_elements);
   int A[no_of_elements][no_of_elements];
   //printf("Enter the elements: ");
   for( int i = 0; i < no_of_elements; i++ )
   {
       for( int j = 0; j < no_of_elements; j++ )
           scanf("%d", &A[i][j]);
   }
     int minimum = findMinimum(no_of_elements, A, 0, no_of_elements,
no_of_elements);
   printf("Local minimum element of the array is %d.\n", minimum);
}
```

If any one wants to test my code, I would recommend to download them from my drive folder. It contains all the C programs written in this assignment. https://drive.google.com/drive/folders/13-IvRnKEl8dyxTDmIYObJbgz_SF5imw5?usp=sharing