# Advanced Data Structures and Algorithm assignment-1

- S20180010135
- P . Venkatesh

---

1)

By the end of this course i want to increase my problem solving ability in efficient manner. I hope this course will help me to dive deeper into data structures and algorithms showing us the best and efficient way to write an algorithm. We can use the data structures helpful to store the data in the efficient manner.

2)

a)

a)

**Algorithm** :

- Declare two arrays of size n for users and entry time.
- Now traverse the loop from first to last and at each time we can compare each element from entry time in leave time.
- If leave[i] >= enter[j] and enter[i] <= leave[j] are the required possible cases

## Explanation:

So,

```
for( i=0; i< n ; i++)
    for( j=i+1 ; j<n ; j++)
        Comparisons
```

Here the outer loop of this algorithm will run **" n "** times and the inner loop will run **"n-1"** times so it takes

n*(n-1) = n^2 - n = **O(n^2)**

b)

## C - Code :

```c
//algorithm with complexity of O(n^2)
#include <stdio.h>
#include <stdlib.h>
int main(){
   int num;
```

```c
    int enter[10];
    int leave[10];
    printf("\n no. of users\n");
    scanf("%d",&num);
    for(int i=0;i<num;i++){
        printf("user entered time of %d\n ",i+1);
        scanf("%d",&enter[i]);
        printf("time of exit by user %d\n ",i+1);
        scanf("%d",&leave[i]);
    }
    int count=0;
    for(int i=0;i<num;i++){
        for(int j=i+1;j<num;j++){
            if(leave[i]>=enter[j]&&enter[i]<=leave[j]){
                count++;
                printf("(%d,%d) ",i+1,j+1);
            }
        }
    }
    printf("\ntotal pairs : %d \n",count);
}
```

```
venkatesh@m-d:~/academic/cse/adsa$ gcc assignment12a.c
venkatesh@m-d:~/academic/cse/adsa$ ./a.out

no. of users
5
user entered time of 1
 1
time of exit by user 1
 4
user entered time of 2
 2
time of exit by user 2
 5
user entered time of 3
 7
time of exit by user 3
 8
user entered time of 4
 9
time of exit by user 4
 10
user entered time of 5
 6
time of exit by user 5
 10
(1,2) (3,5) (4,5)
total pairs : 3
```

b)

a)

**Algorithm :**

As the main part of this algorithm

I. Sort the entry time array using merge sort algorithm
II. Now search the leave time that are in the range of the entry time and returns array index of that max position it will traverse.

**Explanation :**

**Merge sort** basically a divide and conquer algorithm in which original data is divided into smaller sets of data to sort the array. In merge sort firstly we divide the array into two halves, and these arrays are recursively divided into further subarrays till we get n=subarrays with size one. Then these sub arrays are recursively merged by sorting the array until we reach one array remaining which is our sorted array.

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

Using masters theorem here

a=2,b=2,d=1 => we are getting a=b^d case

So the complexity of the algorithm is O( n^d * logn) => **O(n*logn)**

**Binary search** basically it is used to find the position of specific value in sorted array it works on the principle divide and conquer algorithm. Here he we first go to the middle element of the array and if the mid value is lesser than the target values then we should search in the array having indexes greater than the middle index recursively and viceversa untill we find the value and if we reach the left most or right most part of the array and if we didn't found that value still it returns -1.

T(n)= O(n/2) + O(1)

Using the masters theorem here

a=1,b=2,d=0 => we are getting a= b^d case

So the complexity of the algorithm is O(n^d * logn) => **O(logn)**

To do the same question that we done in 2(a) within **O(n*logn)** time we have to sort the entry time and compare the leave time with entry time that are in the range of entry time. Now we have to do this sorting and searching process in O(n*logn) time for doing that i used merge sort for sorting and binary search for searching each leave time entry time range and return its indexes

So, O(n*logn) + O(n * logn) = O(n * logn)

( Merge sort ) ( for searching n items using binary search )

b)

**C - Code :**

```c
//algorithm with complexity of O(nlog(n))
#include <stdio.h>
#include <stdlib.h>
struct user{
   int id;
   int enter;
   int leave;
};
void merge(struct user users[], int l, int m, int r){
   int i, j, k;
   int n1 = m - l + 1;
   int n2 =  r - m;
   int L[n1], R[n2],L1[10],R1[10],L2[10],R2[10];
   for (i=0;i<n1;i++){
       L[i] = users[l+i].enter;
       L1[i] = users[l+i].leave;
       L2[i] = users[l+i].id;
   }
   for (j=0;j<n2;j++){
       R[j] = users[m+1+j].enter;
       R1[j] = users[m+1+j].leave;
       R2[j] = users[m+1+j].id;
   }
   i = 0;
   j = 0;
   k = l;
   while (i<n1 && j<n2) {
       if (L[i]<=R[j]) {
           users[k].enter = L[i];
           users[k].leave = L1[i];
           users[k].id = L2[i];
           i++;
       }
       else{
```

```
            users[k].enter = R[j];
            users[k].leave = R1[j];
            users[k].id = R2[j];
            j++;
        }
    }
    while (i < n1) {
        users[k].enter = L[i];
        users[k].leave = L1[i];
        users[k].id = L2[i];
        i++;
        k++;
    }
    while (j < n2)
    {
        users[k].enter = R[j];
        users[k].leave = R1[j];
        users[k].id = R2[j];
        j++;
        k++;
    }
}
void mergeSort(struct user users[], int l, int r){
    if (l < r){
        int m = (l+r)/2;
        mergeSort(users, l, m);
        mergeSort(users, m+1, r);
        merge(users, l, m, r);
    }
}
void printArray(struct user users[], int size){
    int i;
    printf("id--enter--leave\n");
    for (i=0; i < size; i++){
        printf("%d %d %d \n",users[i].id,
users[i].enter,users[i].leave);
```

```
    }
}
int binarysearch(struct user users[], int first, int last, int
search){
   int middle = (first+last)/2;
   while (first <= last)
   {
       if(users[middle].enter < search){
           first = middle + 1;
       }
       else if(users[middle].enter == search){
           return (middle);
       }
       else{
           last = middle - 1;
       }
       if(last==first && users[last].enter < search &&
users[last].enter !=search)
           return (last);
       middle = (first + last)/2;
   }
  return -1;
}
void counting(struct user users[],int size){
   int count=0;
   int search;
   for(int i=0;i<size;i++){
       search=users[i].leave;
       int l= binarysearch(users,0,size-1,search);
       if(l>0)
           count=count + l-i;
   }
   printf("\ntotal pairs : %d\n",count);
}
int main(){
   struct user users[10];
```

```
    int num;
    printf("\nno. of users\n");
    scanf("%d",&num);
    for(int i=0;i<num;i++){
        users[i].id=i+1;
        printf("user entered time of %d\n ",i+1);
        scanf("%d",&users[i].enter);
        printf("time of exit by user %d\n ",i+1);
        scanf("%d",&users[i].leave);
    }
    mergeSort(users, 0, num - 1);
    printf("\nSorted array is \n");
    printArray(users, num);
    counting(users,num);
}
```

```
venkatesh@m-d:~/academic/cse/adsa$ gcc assignment12b.c
venkatesh@m-d:~/academic/cse/adsa$ ./a.out

no. of users
5
user entered time of 1
 1
time of exit by user 1
 4
user entered time of 2
 2
time of exit by user 2
 5
user entered time of 3
 7
time of exit by user 3
 8
user entered time of 4
 9
time of exit by user 4
 10
user entered time of 5
 6
time of exit by user 5
 10

Sorted array is
id--enter--leave
1 1 4
2 2 5
5 6 10
3 7 8
4 9 10

total pairs : 3
```

3)

a)

**Loop invariant :**

I. Before starting the inner loop each time we will have a minIndex so that A[minIndex] <= A[i:j-1].

II. Before starting the outer loop each time we will have a sorted array A[0:i-1] with minimum elements in A.

**Inductive hypothesis :**

The loop invariant must holds at the end of $i^{th}$ iteration.

**Base case :**

I. In the first iteration of the inner loop ( j=i+1 ) , so A[0:0] which is A[0] is minimum as there are no more elements. Hence it is true for i=0.

II. In the first iteration of the outer loop we have i=0 here the array left to this index is empty so we can say the loop invariant holds as the empty array is ordered.

**Inductive step :**

I. For inner loop

Suppose A[minIndex] <= A[ i:j-1 ] is true for j=k. From proof by induction if the condition is true for **k** then it should be true for **k+1** element also. So let us verify it for j=k+1. And assume that minIndex is the smallest element in the subarray A[i:k-1]. In the inner loop if A[k+1] < A[minIndex] then the if case is executed and minIndex swap with index of the location j ( here j=k+1) since it is the smallest so the lowest value in A[i:k+1] is A[minIndex]. Hence it is true for j=k+1.

II. For outer loop

Suppose A[0:k] contains smaller elements in the array A in sorted order. From proof by induction if the condition is true for **k** then it should be true for **k+1** element also. So let us verify it for i=k+1 as we already know the inner loop returns the index of minimum values element in A[ i:n ] and it will be swapped with A[i] now A[i] becomes the minimum element and it is larger than A[ 0:k ] because the loop invariant is true at the start of the iteration. So it will simultaneously be the smallest element from i to the right and larger than any of its left. Hence it is true for i=k+1.

**Conclusion :**

Hence by using mathematical induction loop invariant is proved, so the correctness of algorithm is proved.

b)

**C - code :**

```c
#include <stdio.h>
#include <stdlib.h>
void swap(int *i,int *j){
```

```
    int tmp=*i;
    *i=*j;
    *j=tmp;
}
void sort(int arr[],int num){
    int minIndex=0;
    for(int i=0;i<num;i++){
    minIndex = i;
        for(int j = i + 1;j<num;j++){
            if (arr[j] < arr[minIndex]){
                minIndex = j;
                swap(&arr[i],&arr[minIndex]);
            }
        }
    }
    printf("sorted array\n");
    for(int i=0;i<num;i++){
        printf("%d ",arr[i]);
    }
}
int main(){
    int num;
    int arr[10];
    printf("enter no. of values\n");
    scanf("%d",&num);
    printf("enter values\n");
    for(int i=0;i<num;i++){
        scanf("%d",&arr[i]);
    }
    sort(arr,num);
}
```

```
venkatesh@m-d:~/academic/cse/adsa$ gcc assignment13.c
venkatesh@m-d:~/academic/cse/adsa$ ./a.out
enter no. of values
4
enter values
2 3 1 4
sorted array
1 2 3 4 venkatesh@m-d:~/academic/cse/adsa$
```

4)

a)

**Algorithm :**

- Declare variables min and an array of size n
- Now traverse through the array assuming that the first element in the array is smallest(min=arr[o]) .
- And if we find any element smaller than min copy that value to min and traverse through the loop till the end and return the finally updated min value.

b)

We have to traverse through the loop till we reach the end to find the minimum of the array through each and every element. These arrays may not be sorted all the time so for finding the minimum it takes atleast of **O(n)** time complexity.

c) **C-Code :**

```c
#include <stdio.h>
#include <stdlib.h>
int main(){
   int num;
   int arr[10];
   printf("enter array size\n");
   scanf("%d",&num);
   printf("enter values\n");
   for(int i=0;i<num;i++)
       scanf("%d",&arr[i]);
   int min=arr[0];
   for(int i=0;i<num;i++){
       if(arr[i]<min)
       min=arr[i];
   }
   printf("minimum=%d",min);
}
```

```
venkatesh@m-d:~/academic/cse/adsa$ gcc assignment14c.c
venkatesh@m-d:~/academic/cse/adsa$ ./a.out
enter array size
4
enter values
2 3 1 4
minimum=1venkatesh@m-d:~/academic/cse/adsa$
```

c)

**Algorithm :**

If n==1 then

return A[1]

A1 = A[0:n/2]

A2 = A[n/2:n]

return min(findmin(A1),findmin(A2))

def min:

return (a>b:a?b);

d)

Here we can say that **T(n)=2T(n/2)+n** => a=2,b=2,d=1

From masters theorem we get a=b^d

So complexity of this algorithm is O(n^d * logn) => **O(nlogn)**

Comparing both the algorithms we can say that the second algorithm is best for smaller range of inputs and 1st algorithm is better for large range of inputs

d)

**C-Code :**

```c
#include<stdio.h>
#include<stdlib.h>
int minimum(int first, int second){
 return (first < second ? first : second);
}
int find_minimum (int arr[], int n){
   if (n == 1)
       return arr[n - 1];
   int mid = n / 2;
   int arr1[mid];
   int arr2[n - mid];
   for (int i = 0; i < mid; i++)
       arr1[i] = arr[i];
   for (int i = mid; i < n; i++)
```

```
        arr2[i - mid] = arr[i];
    return minimum(find_minimum (arr1, mid), find_minimum (arr2, n -
mid));
}
int main(){
    printf ("enter array size\n");
    int n;
    scanf ("%d", &n);
    int arr[n];
    printf ("enter array elements\n");
      for (int i = 0; i < n; i++){
      scanf ("%d", &arr[i]);
    }
    printf ("minimum=%d", find_minimum (arr, n));
}
```

```
venkatesh@m-d:~/academic/cse/adsa$ gcc assignment14d.c
venkatesh@m-d:~/academic/cse/adsa$ ./a.out
enter array size
4
enter array elements
2 3 1 4
minimum=1venkatesh@m-d:~/academic/cse/adsa$
```

5)

a)

I. **Algorithm :**

1. Declare an array
2. Send the starting and ending index of the array
3. Now send the middle element of the array to find the local_minima
4. If the middle element is smaller than its adjacent element return it
5. Otherwise send the left side subarray if the element left to the middle value is small and viceversa.

II. **Proof :**

**Loop invariant :**

After every recursion loop compares the mid index of the array with its left and right elements of the array and returns it if it is a local-minimum. Otherwise it traverse towards the smaller element to its mid because there exist a local minima towards the array having lesser element to its middle element value.

**Proof by contradiction:**

Let us assume the left/right element of the mid is smaller than the mid and there will be no minimum on the left/right side of the array (contradicting our loop invariant)

Case-1:

If the middle element is local minimum then we returns that value which is contradicting our assumption.

Case-2:

And also from our assumption if we traverse the array towards left/right as the left/right element is smaller than the mid. Now it returns the array towards its left/right side of the mid value based on its smaller value (from our assumption that left/right element of the mid is smaller than the mid). Still we cannot find any local-minimum in the left/right array even we reach the left/right most part of the array and we are getting a contradiction at the last but one element whether it should be a local minimum or the last element of the array should be a local-minimum which is contradicting our assumption. From the above proof by contradiction we can say that there will be a local-minimum exist in the array towards the element lesser than the mid element.

Conclusion:

Hence we proved the loop variant holds using proof by contradiction.

## III. Analyzing the runtime :

T(n) = T(n/2) + C  => a=1,b=2,d=0

Comes under 1st case O(n^d * logn) => **O(logn)** complexity.

Using masters theorem the complexity of the given algorithm takes **O(logn)**

## IV. C - code :

```c
#include <stdio.h>
#include <stdlib.h>
int findlocalminima(int arr[], int start, int end,int length){
   int mid = (start + end) / 2;
   if(start==end)
       return arr[end];
   else if(arr[mid+1] > arr[mid] && arr[mid] < arr[mid-1])
       return arr[mid];
   else if(arr[mid-1] > arr[mid+1])
       return findlocalminima(arr, mid+1,end,length);
   return findlocalminima(arr, start,mid-1,length);
```

```
}
int main(){
   int n,arr[20];
   printf("enter n :");
   scanf("%d",&n);
   printf("enter values : ");
   for(int i=0;i<n;i++){
       scanf("%d",&arr[i]);
   }
   printf("\nlocal minimum : %d",
findlocalminima(arr,0,n-1,n));
}
```

```
1 venkatesh@m-d:~/academic/cse/adsa$ gcc 15a.c
venkatesh@m-d:~/academic/cse/adsa$ ./a.out
enter n :4
enter values : 1 2 0 3

local minimum : 0 venkatesh@m-d:~/academic/cse/adsa$
```

b)

**I. Algorithm :**

1. Declare a 2d-array with g= arr[i][j] where 0<=i,j<=n and divide the 2d-array.
2. If g is smaller than its neighbours then returns its indices
3. Else we compare it with its neighbour elements if it is less than both the neighbour elements then the present minimum element is the local minima else we check which of the two neighbours of minimum elements is smaller
4. And the above 2,3 steps will undergo recursion until we get the local minimum.

**II. Proof :**

1)First we find the minimum element in the first middle and last rows and columns of the matrix.

2)Then we compare it with its neighbour elements if it is less than both the neighbour elements then the present minimum element is the local minima else we check which of the two neighbours of minimum elements is smaller

3)Then we repeat the above steps in the submatrix containing our choosen element.

4)It is sure that the local minima would be found in the sub matrix because it should contain atleast a single minimum in the matrix.

**III. Analyzing the runtime :**

T(n)=T(n/2) + c*n => a=1,b=2,d=1

From masters theorem it comes under a<b^d case

So the complexity of this algorithm is **O(n^d) = O(n)**