3(a)

Assumption: Each row in this 2Dimensional array represents an interval where, 1st column represents the start time and 2nd column represents the end time interval.

In order to solve this, we divide the given array with n rows into subarrays with n/2 rows until we reach a point where sub-array consists of just one row. Then we merge these subarrays on our way up, in sorted order of their interval if the intervals are not overlapping. If the intervals are overlapping we return "OverLapping" as output else, the array is sorted in the increasing order of its time interval and "NotOverLapping" is returned as the output

Given: Three 2Dimensional array,A which consists of n intervals, temporary arrays L and R which is used to store a copy of left half and right half of the given array A and three integer values left, right and mid which represents the left, right and the middle index that needs to be considered.
Returns: a String based on whether the intervals are overlapping or not

```
MergeToFindOverLap(A, L, R, left, mid, right)
1           g1 =0
2           g2 =0
3           for  i = left  to mid
4                   L[g1][0] = A[i][0]
5                   L[g1][1] = A[i][1]
6                   g1++

7           for  j = mid+1 to right
8                   R[g2][0] = A[j][0]
9                   R[g2][1] = A[j][1]
10                  g2++

11          n1 = 0
12          n2 = 0
13          k = left
14          m1 = mid-left+1
15          m2 = right-mid
16
17          while n1 < m1 and n2 < m2
18
19                  if  L[n1][0]  <  R[n2][0] and L[n1][1]  <=  R[n2][0]
20                          A[k][0] = L[n1][0]
21                          A[k][1] = L[n1][1]
22                          n1++
23
24                  else if L[n1][0]  >=  R[n2][1] and L[n1][1]  >  R[n2][1]
25                          A[k][0] = R[n2][0]
26                          A[k][1] = R[n2][1]
27                          n2++

28                  else
29                          return "OverLapping"

30                  k = k++
```

```
31              while  n1 < m1
32                      A[k][0] = L[n1][0]
33                      A[k][1] = L[n1][1]
34                      n1++
35                      k++

36              while n2 < m2
37                      A[k][0] = R[n2][0];
38                      A[k][1] = R[n2][1];
39                      n2++
40                      k++

41              return "NotOverLapping"
```

```
CheckOverLap (A, L, R, left, right)
1       if left < right
2               mid = ( left + right ) / 2
3               result1 = CheckOverLap (A, left, mid)
4               result2 = CheckOverLap (A, mid+1, right)
5               if  result1 == "OverLapping" or result2 == "OverLapping"
6                       return "OverLapping"
7               else
8                       return MergeSortToFindOverLap(A,left,mid,right)
9        return "NotOverLapping”
```

3 (c) Space and Time efficiency:

We use a temporary arrays L and R where L and R are 2-Dimensinal arrays of size n. Since they are allocated just once before the recursive calls are made,

Space complexity = $\frac{2n}{2} + \frac{2n}{2} = O(2n) = O(n)$

Time Complexity :

Let T(2n) be the time complexity of CheckOverLap function, where 2n is the size of 2Dimensional array. CheckOverLap makes recursive call on two subarrays of size n and Makes call to MergeToFindOverLap function, which involves dividing the array in to two halves. So time complexity will be T(2n/2) = T(n). Step 3 and 4 takes T(n) time execution. Cost of step8 is C_8 *2n ,as it involves merging array of size 2*n Remaining steps take some constant time C which is negligible. So we ignore this constant.

$T(2n) = 2T(n) + 2cn$

$= 2^2 T(n/2) + 2cn + 2cn$

$= 2^3 T(n/2^2) + 2cn + 2cn + 2cn$

:

:

By generalizing we have

$$T(2n) = 2^{i+1} T(n/2^i) + \sum_{i=0}^{\log n} 2cn$$

$(n/2^i) = 1$

$n = 2^i$

$i = \log_2 n$

$T(2n) = 2^{(\log_2 n)+1} T(1) + 2cn(\log n)$

$= n^{\log_2 2} \cdot 2 \cdot T(1) + 2cn(\log n)$

Substituting $\log_2 2 = 1$ and $T(1) = c_x$ where c_x is some constant time

$T(2n) = n + 2cn\log n$

By ignoring all small terms, we have :

$T(2n) = O(n\log n)$

Substituting 2n by N, we have

$T(N) = O(n\log n)$

3 (b) Proof of correctness:

In order to prove the **correctness of CheckOverLap**, we need to prove the correctness of MergeToFindOverLap functionality.

To prove the correctness of MergeToFindOverLap, let's consider the invariant for the internal loop.

Note: we consider (x1, y1) and (x2, y2) as two intervals then (x1, y1) comes first in the order iff x1 < x2 and y1 <= x2.

Invariant: Following invariants hold true at the beginning of the loop

{i} L[k1, k2] and R[u1,u2] contains time intervals of Array A where k1 ={p … m-1} and k2 ={0 … 1} and u1 = {m-1 … q] and u2 = {0 ..1} sorted in the increasing .

{ii} Array A[k1, k2] for k1 ={ p .. k-1} and k2 ={0..1} is sorted.

{iii} L[k1,k2] and R[u1,u2] contains sorted elements where k1 ={i … m-1} and k2={0 … 1} and u1 = {j … q] and u2 = {0 ..1} , i and j are the indices pointing to the first element of the 2dimensional array L and R.

{iv} All the entries in array A[k1, k2] for k1 ={ p .. k-1} and k2 ={0..1} comes before the elements in the arrays L[k1,k2] and R[u1,u2] where k1 ={i … m-1} and k2={0 … 1} and u1 = {j … q] and u2 = {0 ..1}.

If the above mentioned invariants holds true then “NotOverLapping” message is returned, else “Overlapping” message is returned as time periods overlap and no further calls to merge.

Two intervals (x1,y1) and (x2,y2] are overlapping iff x2<x1<y2 or x2<y1<y2 or (x1<x2 and y1>y2)

Initialization: At the very beginning k = 1, i =p and j = m, so the invariant (i) holds, invariant (ii) ,(iii) and (iv) holds as the array considered is empty.

Maintenance: Here let’s assume that all the invariants hold good at the beginning of the while loop where k1 = i and k2= j

Now we need to prove that the invariant hold true at the end of the while loop.

Invariant (i) holds true because L and R arrays are altered inside the while loop.
Invariant (ii) holds true because only one interval entry from either L or R is copied on to A if they are not overlapping.
Invariant (iii) also holds true as we are just pointing i or j to the next position in the same sorted array L and R without making any changes.
Invariant (iv) also holds true as by the assumption we made, the element copied on to the array A from L or R is greater than the existing elements. So L and R will remain sorted even after the appending the new value.

Termination: At the end of while loop all the elements are copies on to array A from L and R if no intervals overlap. Therefore, all the invariants hold true even at the termination of while loop. In case of overlaps, “OverLapping” is returned as Output.

Now Let’s Prove the **correctness of the CheckSort** Functionality By using Induction hypothesis:

Invariant: for all n>=1, CheckSort sorts all the content of array A in the increasing order of their time period. (n is the number of rows in the given 2Dimensional array A)

Initialization: for n =1 we have just one entry, which is sorted

Maintenance: Assume that the invariant holds true for n <= k. Now, we need to prove that CheckSort works for n = k+1.

Checksort divides the given array of size n into two subarrays of size n/2 and sorts the sub array. MergeToFindOverLap function is called to merge these sorted array,

By inductive hypothesis, the subarrays are sorted and since we have proved MergeToFindOverLap works correctly for two sorted arrays. CheckSort holds true for n =k +1

Hence Proved.