CS5800: Algorithms Spring 2018
Assignment 2.3

Saptaparna Das

January 29, 2018

3.(a) Algoritm:
1. Sort all the intervals depending on their start.We can use merge sort here.
2. Linearly check if for any interval, its end time is less than start time of previous interval.

Pseudo code:

```
//Assume the input is a zero indexed array of intervals,
//sorted in ascending order according to their start times
hasOverlap(input[0,1...,n-1]){
        for(i=1 to length){
                if(input[i].start<input[i-1].end)
                        return true
        }
        return false
}
```

(b) We can see, for every iteration of the for loop, input[0..i-1] does not have any overlapping intervals. So, we can use this property as loop invarient. To show the program works correctly, we need to show three things:

Initialization: We need to show that loop invarient holds before first iteration of the loop.
When i=0, the sub-array input[0..i-1] will have only one interval. So there is no overlap, which shows loop invarient is true before loop starts.
Maintenance: We need to show that loop invarient holds for each iteration of the loop.
In each iteration we check, if that interval overlaps with previous interval. Lets assume before i^{th} iteration, input[0...i-1] doesn't contain any overlapping intervals.
There are two cases now. If the current interval is not overlapping the previous interval, then input[0...i-1] doesn't have any overlapping intervals and after current iteration input[o...i] will have same property.
If current interval is overlapping the previous one, in that case the function returns true and stops searching.
Termination: We need to show loop invarient holds after loop termination.
The loop terminates in two conditions. Either i = length or for any iteration, current interval overlaps previous interval. In later case, we have already discussed hasOverlap works correctly. In first case, when all elements are scanned, i becomes n. It exits the loops and returns false. So, in that case also i[0..n-1] subarray doesn't conatin any overlaps.

Hence, hasOverlap works correctly for all possible inputs.

(c) Analysis of efficiency:

There are two things going on here, which can contribute in the time complexity. First, the input array is sorted using merge sort.
In merge sort, as we know the divide step takes constant amount of time. Because it just finds the mid point.
Let T(n) is time taken by merge sort to sort an array of n elements. The conquer step recursively sorts 2 sub-arrays of size n/2 (approximately). So, it takes 2T(n/2) time.
The merge step, merges n elements in linear time. So it takes $\Theta(n)$ time. We can write, $T(n) = 2T(n/2) + \Theta(n)$

Now, applying Master theorem, a=2,b=2, f(n)=O(n). $n^{\log_b a} = n^1 = n$. So, the complexity is $\Theta(n \log n)$

For the second part, the algorithm is scanning the elements linearly. So, at most there will be n comparisions. So, again we can say, time complexity for hasOverlap function is linear. i.e. $\Theta(n)$
Hence, the time complexity of whole algorithm is, $\Theta(n \log n) + \Theta(n)$. Since, $\Theta(n \log n)$ is greater than $\Theta(n)$, we can drop the second part. So, its $\Theta(n \log n)$.
For space complexity, hasOverlap() doesn't use any extra space but the algorithm internally calls merge sort, which uses

extra O(n) space. So, space complexity would be O(n).