Fuzzy Sorting

Consider a sorting problem in which we do not know the numbers exactly. Instead, for each number, we know an interval on the real line to which it belongs. That is, we are given n closed intervals of the form $[a\ !\ , b!\ ]$, where $a\ ! \leq b!$

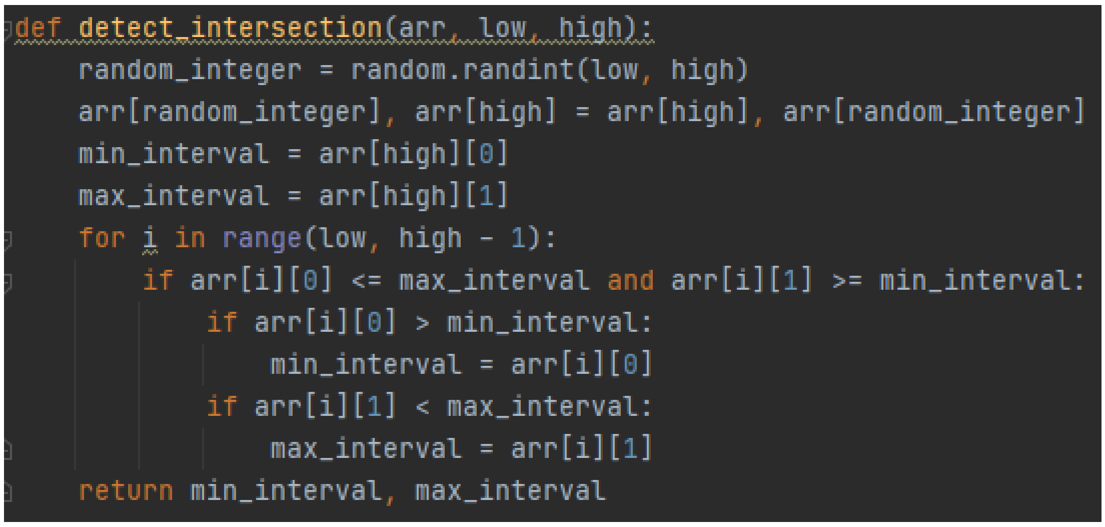
We wish to fuzzy-sort these intervals, i.e., to produce a permutation $\langle i1\ , i2\ , \ldots\ , i\mathrm{n}\ \rangle$ of the intervals such that for $j = 1,2, \ldots , n$ there exist $c\%\in [a\ !\ !\ , b!\ !\ ]$ satisfying $c1 \leq c2 \leq \cdots \leq c\mathrm{n}$

a) Design a randomized algorithm for fuzzy-sorting n intervals. Your algorithm should have the general structure of an algorithm that quicksort's the left endpoints (the $a\ !$ values), but it should take advantage of overlapping intervals to improve the running time. (As the intervals overlap more and more, the problem of fuzzy-sorting the intervals become progressively easier. Your algorithm should take advantage of such overlapping, to the extent that it exists.)

b) Argue that your algorithm runs in expected time $\Theta(n \lg n)$ in general, but runs in expected time $\Theta(n)$ when all of the intervals overlap (i.e., when there exists a value x such that $x \in [a\ !\ , b!\ ]$ for all i). Your algorithm should not be checking for this case explicitly; rather, its performance should naturally improve as the amount of overlap increases.

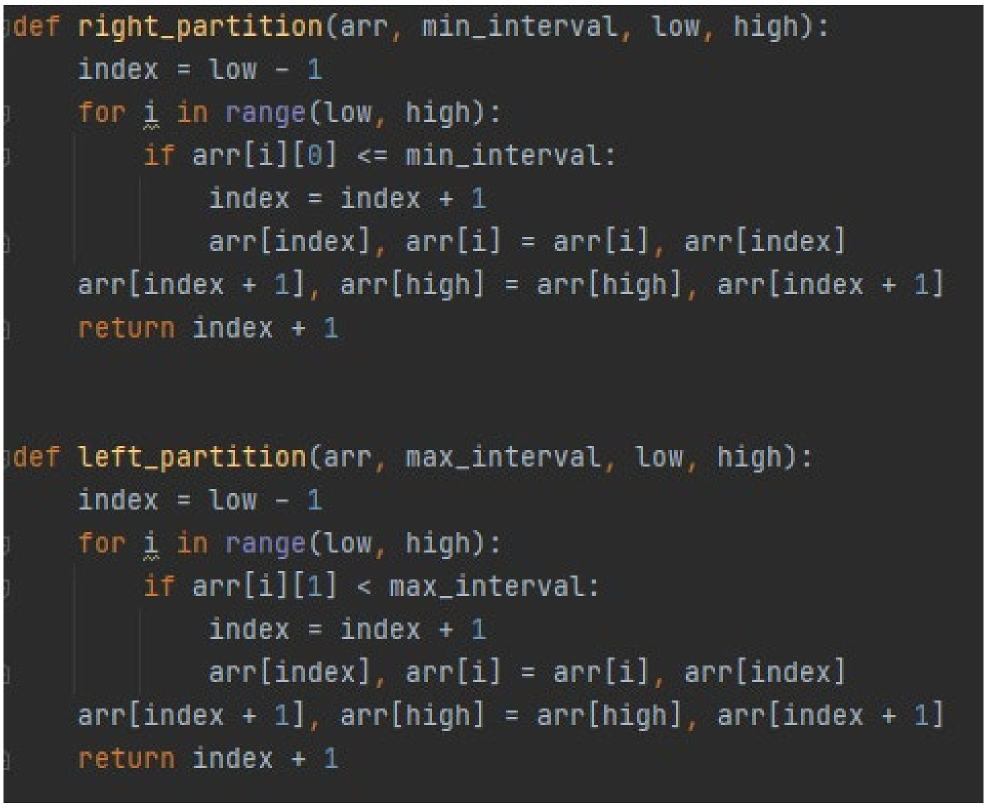
c) Use the two provided files with non-overlapping and overlapping intervals to time your algorithm in each case. Use different values of n (i.e. use the first n intervals in the files), time the execution time of the algorithm, and plot as a function of n. Can you observe $\Theta(n \lg n)$ and $\Theta(n)$ behaviour in your plots?

a)



```python
def detect_intersection(arr, low, high):
    random_integer = random.randint(low, high)
    arr[random_integer], arr[high] = arr[high], arr[random_integer]
    min_interval = arr[high][0]
    max_interval = arr[high][1]
    for i in range(low, high - 1):
        if arr[i][0] <= max_interval and arr[i][1] >= min_interval:
            if arr[i][0] > min_interval:
                min_interval = arr[i][0]
            if arr[i][1] < max_interval:
                max_interval = arr[i][1]
    return min_interval, max_interval
```

The detect_intersection function takes in three arguments: arr, low, and high, which are the list of intervals, the lower index of the list, and the higher index of the list, respectively. It randomly selects an index from the list, swaps the interval at that index with the interval at the high index, and then determines the minimum and maximum values for the overlapping intervals. It returns a tuple containing these minimum and maximum values.

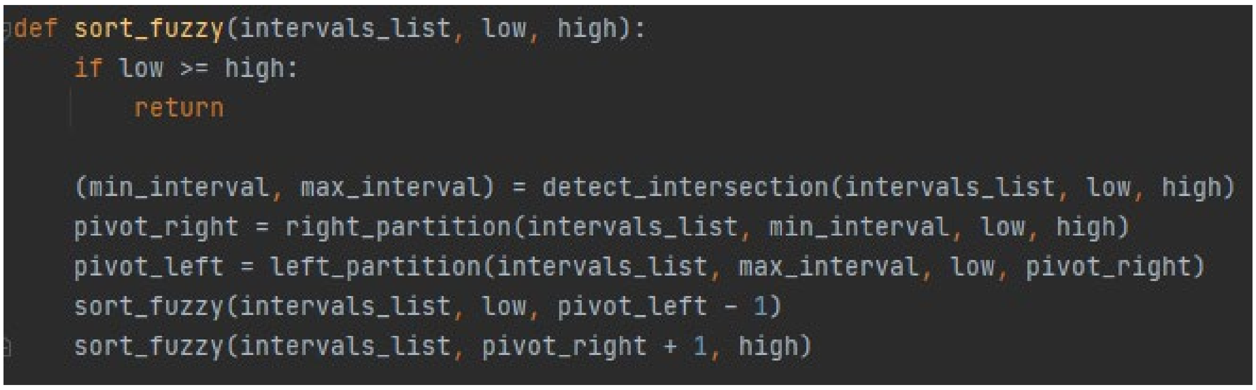


```python
def right_partition(arr, min_interval, low, high):
    index = low - 1
    for i in range(low, high):
        if arr[i][0] <= min_interval:
            index = index + 1
            arr[index], arr[i] = arr[i], arr[index]
    arr[index + 1], arr[high] = arr[high], arr[index + 1]
    return index + 1


def left_partition(arr, max_interval, low, high):
    index = low - 1
    for i in range(low, high):
        if arr[i][1] < max_interval:
            index = index + 1
            arr[index], arr[i] = arr[i], arr[index]
    arr[index + 1], arr[high] = arr[high], arr[index + 1]
    return index + 1
```

The right_partition function takes in four arguments: arr, min_interval, low, and high. It partitions the list into two parts, where the left part contains all intervals with values less than or equal to min_interval, and the right part contains all intervals with values greater than min_interval. It returns the index of the last element of the left partition.

The left_partition function takes in four arguments: arr, max_interval, low, and high. It partitions the list into two parts, where the left part contains all intervals with values less than max_interval, and the right part contains all intervals with values greater than or equal to max_interval. It returns the index of the last element of the left partition.



```python
def sort_fuzzy(intervals_list, low, high):
    if low >= high:
        return

    (min_interval, max_interval) = detect_intersection(intervals_list, low, high)
    pivot_right = right_partition(intervals_list, min_interval, low, high)
    pivot_left = left_partition(intervals_list, max_interval, low, pivot_right)
    sort_fuzzy(intervals_list, low, pivot_left - 1)
    sort_fuzzy(intervals_list, pivot_right + 1, high)
```

The sort_fuzzy function is a recursive function that takes in three arguments: intervals_list, low, and high. It sorts the intervals in the list based on whether they overlap or not. If the length of the list is less than or equal to 1, the function simply returns. Otherwise, it uses the detect_intersection function to find the minimum and maximum values for the overlapping intervals, and then partitions the list using right_partition and left_partition. The function then recursively calls itself on the two sublists created by the partitions.

b)

The detect_intersection() function runs through the list once, so its time complexity is O(n). Both right_partition() and left_partition() functions also loop through the list once, so their worst-case time complexity is also O(n).

The sort_fuzzy() function, despite having a worst-case time complexity of O(n^2), performs better in practice because of the fuzzy sorting algorithm it uses. On average, it has a time complexity of O(nlogn), which is considered a reasonably efficient sorting algorithm.

The recurrence relation for the entire algorithm can be expressed as:

T(n) = 2T(n/2) + O(nlogn).

c)

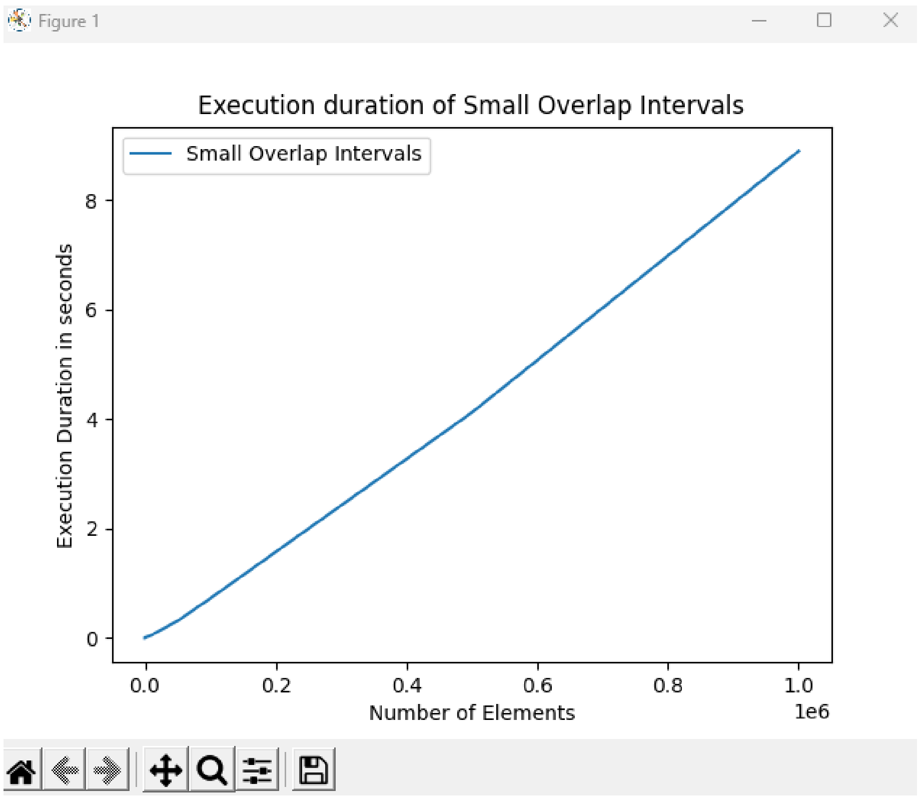
The time complexity of the detect_intersection() function is dependent on the overlap between intervals. If the intervals have a small overlap, the time complexity of the function is O(n), as it loops through the list once.

On the other hand, if the intervals have a large overlap or if all intervals overlap with each other, the time complexity of the function increases to O(nlogn) due to the sorting algorithm used in the code.

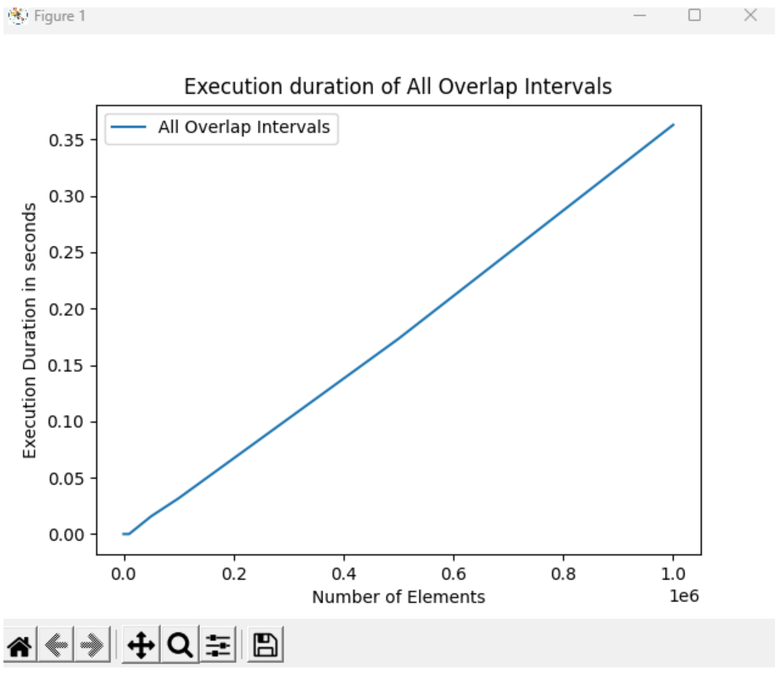
The worst-case time complexity of the sort_fuzzy() function is O(n^2), but in practice, it performs much better due to the fuzzy sorting algorithm.

It has an average time complexity of O(nlogn), making it a reasonably efficient sorting algorithm. Overall, the time complexity of the algorithm varies depending on the overlap between intervals, with smaller overlaps resulting in a lower time complexity of O(n) and larger overlaps resulting in a higher time complexity of O(nlogn).

The following graphs show show both O(nlogn) and O(n) behaviors.



The small overlap Intervals will have the time Complexity of O(n)



The all overlap Intervals will have the time Complexity of O(nlogn)