ENPM809X: Data and Algorithms

Bharadwaj Chukkala

PROJECT 1

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 $[a_i, b_i]$ where $a_i \leq b_i$.

We wish to fuzzy-sort these intervals, i.e., to produce a permutation $(i_1, i_2, ..., i_n)$ of the intervals such that for $j = 1, 2, ..., n$ there exist $c_j \in [a_{i_j}, b_{i_j}]$ satisfying $c_1 \leq c_2 \leq ... \leq c_n$

(a) Design a randomized algorithm for fuzzy-sorting n intervals. Your algorithm should have the general structure of an algorithm that quicksorts the left endpoints (the a_i 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 becomes 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 $O(nlog(n))$ in general, but runs in expected time $O(n)$ when all of the intervals overlap (i.e. when there exists a value x such that $x \in [a_i, b_i]$ 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 $O(nlog(n))$ and $O(n)$ behavior in your plots?

My answer summarizes the design of a fuzzy sorting algorithm that operates in-place on two arrays containing the interval endpoints. The proposed algorithm has the general structure of randomized quicksort. As a result, its expected running time is $O(nlog(n))$ for disjoint intervals. The presence of overlapping intervals is detected and reduces the expected running time. In fact, the proposed algorithm has an expected running time of $O(n)$ when all intervals overlap at some point.

1 Problem (a)

1.1 Find-Intersection

- As a starting point, I will review the fundamental process of Quicksort. Initially, it's important to acknowledge that Quicksort implements the divide-and-conquer approach. It initiates the algorithm by dividing the input array into two sub-arrays using a pivot value chosen at random. These sub-arrays are subsequently sorted by means of recursive function calls. Due to the process being recursive, the in situ operations will eventually return a sorted array.
- We need to understand that fuzzy sorting can be trivially implemented by quick-sorting the left endpoints (a_i) of the list of overlapping intervals. When we do this, we will be randomly selecting a pivot point and doing this will give us a worst case expected running time of $O(nlog(n))$.

- To achieve a better standing expected run time of $O(n)$, we need to realize that when the intervals are overlapping, there is no explicit need for sorting. Meaning, for two overlapping intervals i and j, $[a_i, b_i] \cap [a_j, b_j] \neq \emptyset$. In such situation we can choose $c_i, c_j \ni c_i \leq c_j$ or $c_j \leq c_i$.
- Since overlapping intervals do not require sorting, we can improve the expected running time by modifying quicksort to identify overlaps. This is how it i modified the quicksort algorithm.

```
FIND-INTERSECTION(A, p, r)
    rand = RANDOM(p, r)
    exchange A[rand] with A[r]
    a = A[r].a
    b = A[r].b
    for i = p to r - 1
        if A[i].a <= b and A[i].b >= a
            if A[i].a > a
                a = A[i].a
            if A[i].b < b
                b = A[i].b
    return (a, b)
```

Explanation:

- This algorithm is used as an effort to find an overlapping interval in linear time.
- We select a random pivot interval as the initial region of overlap $[a, b]$.
- If the intervals $[a_i, b_i]$ are disjoint, then the estimated region of overlap will simply be this randomly-selected interval.
- We loop through every interval listed in arrays A and B and at each iteration, we determine if the current interval overlaps the running estimate of the region of overlap. If it does, we update the region of overlap as $[a, b] = [ai, bi] \cap [a, b]$.
- Since we evaluate the intersection from 1 to A.length() by looping through the whole array (n iterations), the `Find-Intersection` algorithm has a worst-case running time of $O(n)$

1.2 Fuzzy-Sort

- Extending the Quicksort algorithm to allow fuzzy sorting using `Find- Intersection`.
- We create three partitions of the input array: "left", "middle", and "right" subarrays. The "middle" subarray elements overlap the interval $[a, b]$ found by `Find-Intersection`. As a result, they can appear in any order in the output.
- Just like we do in quicksort, we recursively call Fuzzy-Sort on the "left" and "right" subarrays to produce a fuzzy sorted array in place. The following pseudocode implements these basic operations.

```
FUZZY-SORT(A, p, r)
    if p < r
        (a, b) = FIND-INTERSECTION(A, p, r)
        t = PARTITION-RIGHT(A, a, p, r)
        q = PARTITION-LEFT(A, b, p, t)
        FUZZY-SORT(A, p, q - 1)
        FUZZY-SORT(A, t + 1, r)
```

Explanation:

- Using the `Find-Intersection` function we find out the interval in which middle sub-array elements overlap.
- We use `Partition-Right` to get a right sub-array from p to r using a pivot value equal to the left endpoint a which gives us t that is the pivot index in a sorted right array.
- We use `Partition-Left` to get a left sub-array from p to t using a pivot value equal to the right endpoint b which gives us q that is the pivot index in a sorted left array.
- Then using t and q, we recursively call `Fuzzy-Sort` to sort both the partitions.

1.3 Partitioning

- The key to the algorithm is the PARTITION procedure, which rearranges the sub-array $A[p, ..., r]$ in place.
- This partitioning step is important in fuzzy sorting because it helps to efficiently sort an array of elements based on their "fuzziness" or degree of similarity to a reference value.
- By comparing each element to a reference value and partitioning the array accordingly, the algorithm can quickly group similar elements together and separate them from dissimilar ones.

1.3.1 Right Partition

```
PARTITION-RIGHT(A, a, p, r)
    i = p - 1
    for j = p to r - 1
        if A[j].a <= a
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    return i + 1
```

Explanation:

- p and r represent the start and end indices of a sub-array within A that we want to sort, and a is a value that we use for comparison with the elements in the sub-array.
- The algorithm partitions the sub-array $A[p...r]$ into two parts based on the comparison with a. It does this by iterating over the sub-array from left to right using a loop variable j.
- For each element $A[j]$, if $A[j].a$ is less than or equal to a, we swap it with the element at index $i + 1$, where i is a variable initialized to $p - 1$. This effectively places element $A[j]$ in the first part of the partition.
- At the end of the loop, we swap the element at index $i + 1$ with the element at index r, which ensures that the partition is properly divided between the two parts. Finally, we return the index $i + 1$, which is the position of the first element in the second part of the partition.

1.3.2 Left Partition

```
PARTITION-LEFT(A, b, p, t)
    i = p - 1
    for j = p to t - 1
        if A[j].b < b
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[t]
    return i + 1
```

Explanation:

- This implementation is very similar to `Right-Partition`, the difference is we are using p and t as start indices trying to partition the left sub-array $A[p...t]$ into two parts.

2 Problem (b)

2.1 Analysis of Running Time

2.1.1 Proof of Worst Case Running Time: $O(nlog(n))$

- We expect `Fuzzy-Sort` to have a worst-case running time of $O(nlog(n))$ for a set of input intervals that do not overlap each other.

- Looking at the lines 2 through 3 of `Find-Intersection`, we can see that the algorithm selects a random interval as the initial pivot interval just like it happens in `Quick-Sort`.
- Recall that if the intervals are disjoint, then the final $[a, b]$ values will simply be this initial interval.
- Assuming the worst case scenario, which is that there are no overlaps, the "middle" region created by lines 4 and 5 of `Fuzzy-Sort` will only contain the initially-selected interval.
- In general, `Find-Intersection` function has a running time of $O(n)$, and this is because the pivot interval [a,b] is randomly selected, the expected sizes of the "left" and "right" subarrays are both $\frac{n}{2}$.
- Piecing the information we have together, the recurrence of the running time is:

$$T(n) \leq 2T(\tfrac{n}{2}) + O(n) = O(nlog(n))$$

This running time is identical to that of randomized `Quick-Sort`. As a result, we find that the worst-case expected running time of `Fuzzy-Sort` is $O(nlog(n))$.

2.1.2 Proof of Best Case Running Time: $O(n)$

- We expect Fuzzy-Sort to have a best-case running time of $O(nlog(n))$ for a set of input intervals that overlap each other.
- The `Find-Intersection` will always return a non-empty region of overlap $[a, b]$ containing x if the intervals all overlap at x. For this situation, every interval will be within the "middle" region since the "left" and "right" sub-arrays will be empty, and lines 6 and 7 of `Fuzzy-Sort` are $O(1)$.
- As a result, there is no recursion, and the running time of `Fuzzy-Sort` is determined by the $O(n)$ running time required to find the region of overlap. Therefore, if the input intervals all overlap at a point, then the expected worst-case running time is $O(n)$. This can be considered as a best case running time when we are using fuzzy sort and comparing with non-overlapping intervals.

3 Problem (c)

3.1 Comparison of Execution times of Small and All Overlapping intervals

- $O(n)$ means that the algorithm's runtime grows linearly with the size of the input. This means that if the input size doubles, the runtime of the algorithm will also double.
- $O(nlog(n))$ means that the algorithm's runtime grows at a rate that is proportional to n multiplied by the logarithm of n. This means that if the input size doubles, the runtime of the algorithm will more than double.

3.1.1 Case and Analysis

- To understand the behavioural difference better, let us suppose we have one algorithm, and two different sets of data. Upon implementing the algorithm for both the data, we notice that there is a difference in execution times for the same algorithm when the data changes.
- This difference lets us estimate the better and worse case running times for the same algorithm, giving us loose bounds of the time complexity.
- We notice that the algorithm with $Dataset - 1$ has a time complexity of $O(n)$ and $Dataset - 2$ has a time complexity of $O(nlog(n))$.
- The stark difference can be visualized on a plot of both those time complexities.

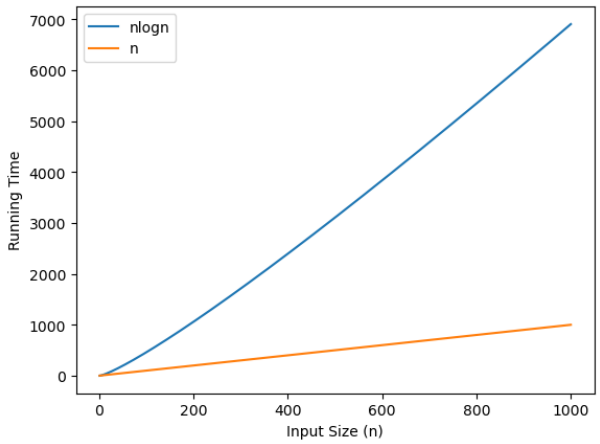

Fig 2: O(n) vs O(nlog(n))

3.1.2 Case Example: `Fuzzy-Sort`

- `Fuzzy-Sort` is the example of the case that we discussed above and also the problem at hand.
- Analogous to the case, here we have two datasets *Small Overlap* and *All Overlap*. Both of which upon implementing `Fuzzy-Sort` on them will give us varying execution times for different values of n.
- When we look at the plot below, the comparison of execution times of `Fuzzy-Sort` for All overlapping and Small overlapping intervals, we can see how sorting of All overlapping intervals has a time complexity closer to $O(n)$ and sorting Small overlapping intervals has a time complexity closer to $O(nlog(n))$.
- We should note that, if we are sorting a small array with only a few elements, the execution time of both datasets may be comparable. However, as the size of the array grows larger, the execution time for Small overlap increases rapidly when compared to All overlap.
- In conclusion, **We observe that** sorting data with Small Overlapping Intervals tends to have the behaviour of the worst case complexity, $O(nlog(n))$. And, sorting data with All overlapping intervals tends to have the behaviour of best case complexity, $O(n)$.

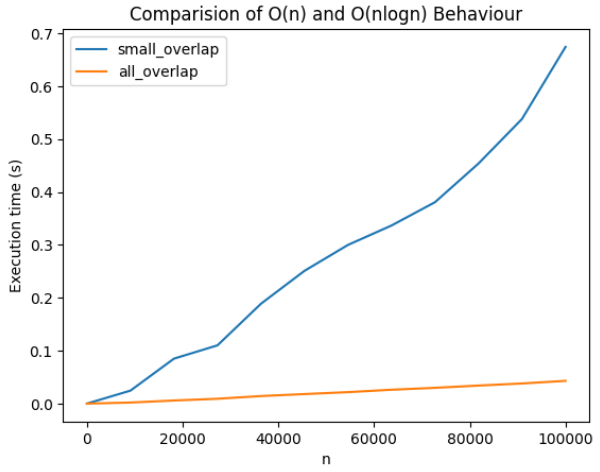

Fig 2: Small Overlap $O(nlog(n))$ and All Overlap $O(n)$

3.1.3 Get Run Times & Plot

```

n_values = Linspace[1, 100000, 12]

    GET_RUN_TIMES(n_values, A)
        for n in n_values
            arr = A[0 to n]
            start = time.start
            Do Fuzzy-Sort(arr)
            end = time.end
            execution_time = start - end
            execution_times-> append execution_time
        return execution_times

Plot(x= n_values, y= execution_times)

# Do for both sets of data
```

4 Resources

- Github link to code files

**********Fin**********