CS 157: Assignment 3

Douglas R. Lanman

13 March 2006

In this write-up, we extend quicksort to the task of *fuzzy sorting of intervals.* In many situations the precise value of a quantity is uncertain (e.g., any physical measurement is subject to noise). For such situations we may represent a measurement i as a closed interval $[a_i, b_i]$, where $a_i \leq b_i$. A *fuzzy-sort* is defined as a permutation $\langle i_1, i_2, \ldots, i_n \rangle$ of the intervals such that there exist $c_j \in [a_{i_j}, b_{i_j}]$ satisfying $c_1 \leq c_2 \leq \ldots \leq c_n$.

This document summarizes the design of a fuzzy sorting algorithm which 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 $\Theta(n \lg n)$ 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 $\Theta(n)$ when all intervals overlap at some point.

This document is organized in four sections. In Section 1 we introduce the basic strategy for fuzzy-sorting and a linear-time algorithm to detect the intersection of a set of intervals. In Section 2 we modify randomized quicksort to allow fuzzy sorting. In Section 3 we present a proof of correctness. Finally, in Section 4 we analyze the expected running time of the proposed algorithm.

1 Finding an Overlapping Interval in Linear Time

High-level Description: To begin our analysis, we review the basic operation of QUICKSORT, as presented in [1]. First, note that quicksort utilizes the divide-and-conquer paradigm. It begins by partitioning the input array into two subarrays based on a randomly-selected *pivot* value. These subarrays are then sorted through recursive calls. Since quicksort operates in-place, a sorted array will be produced after the recursive evaluations are completed.

Notice that fuzzy sorting can be trivially implemented by quicksorting the left endpoints (i.e., the a_i's). If we used a randomly-selected left endpoint for the pivot, then this algorithm would achieve the desired worst-case expected running time of $\Theta(n \lg n)$. We can do better than this, however, by realizing that overlapping intervals do not require sorting. That is, for two overlapping intervals i and j, $[a_i, b_i] \cap [a_j, b_j] \neq \emptyset$. In such situations, we can always choose $\{c_i, c_j\}$ (within the intersection of these intervals) such that $c_i \leq c_j$ *or* $c_j \leq c_i$. In general, if n intervals overlap, then any of the $n!$ possible permutations would be a valid fuzzy-sort of the intervals.

Since overlapping intervals do not require sorting, we can improve the expected running time by modifying quicksort to identify overlaps. To this end, we introduce FIND-INTERSECTION.

```
FIND-INTERSECTION($A, B, p, s, a, b$)
  $i \leftarrow$ RANDOM($p, s$)
  exchange $A[i] \leftrightarrow A[s]$
  exchange $B[i] \leftrightarrow B[s]$
  $a \leftarrow A[s]$
  $b \leftarrow B[s]$
  for $i \leftarrow p$ to $s - 1$
      do if $A[i] \leq b$ and $B[i] \geq a$
            then if $A[i] > a$
                    then $a \leftarrow A[i]$
                 if $B[i] < b$                find interval, max(A[i],a) and min(B[i],b)
                    then $b \leftarrow B[i]$
```

Notice that A and B are arrays containing the interval endpoints a_i and b_i, respectively. A non-empty intersection of a subset of intervals from p to s will be determined by Find-Intersection. The right and left endpoints of this intersection are stored in a and b, respectively.

Proof of Correctness: On lines 1 through 3 of Find-Intersection, 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. On lines 4 through 11, we loop through every interval listed in arrays A and B (except the endpoint which is the initial interval). 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 $[a, b] = [a_i, b_i] \cap [a, b]$.

If there exists a value $c_* \in [a_i, b_i], \forall i$, then the intervals overlap at some point. In this situation, the estimated region of overlap $[a, b]$ produced by Find-Intersection will contain c_*. As a result, Find-Intersection will always find a valid region of overlap containing c_* if the intervals completely overlap.

Analysis of Running Time: Find-Intersection has a worst-case running time $\Theta(n)$. Lines 1 through 5 and lines 7 through 11 involve $\Theta(1)$ operations. Line 6 is an iteration over $s - p$ array elements. Since we may evaluate the intersection for $p = 1$ and $s = length(A)$, then the worst-case running time is $\Theta(n)$ – linearly proportional to the size of the input.

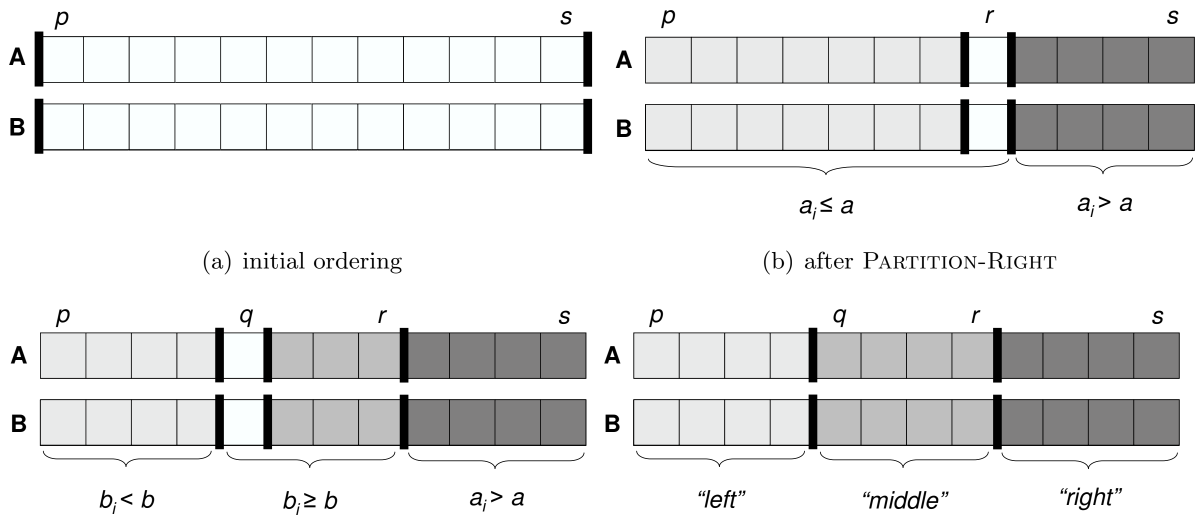
2 In-Place Fuzzy Sorting using Randomized Quicksort

We can extend the pseudocode Quicksort presented in [1] to allow fuzzy sorting using Find-Intersection. Following the strategy introduced in Section 1, we first partition the input array into "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. As 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($A, B, p, s$)
if $p < s$
   then $a \leftarrow b \leftarrow 0$
        Find-Intersection($A, B, p, s, a, b$)
        $r \leftarrow$ Partition-Right($A, B, a, p, s$)
        $q \leftarrow$ Partition-Left-Middle($A, B, b, p, r$)
        Fuzzy-Sort($A, B, p, q - 1$)
        Fuzzy-Sort($A, B, r + 1, s$)
```

As before, A and B are arrays containing the interval endpoints a_i and b_i, respectively. To fuzzy-sort an entire array, one can run Fuzzy-Sort($A, B, 1, length[A]$). As shown in Figure 1(d), the index of the first and last intervals in the "middle" region are given by q and r, respectively. The first and last elements in a subarray are indexed by p and s, respectively.

At this point, we need to determine how to partition the input arrays into "left", "middle", and "right" subarrays *in-place*. Recall that Quicksort partitions an input array into one subarray with values less than or equal to the pivot and another subarray with values greater than the pivot. We can exploit this property by first partitioning the array using a pivot value equal to the left endpoint a found by Find-Intersection, such that $a_i \leq a$. Afterward, we can partition the subarray from p to r using a pivot value equal to the right endpoint b found by Find-Intersection, such that



(a) initial ordering

(b) after Partition-Right

(c) after Partition-Left-Middle

(d) resulting partitions

Figure 1: The three regions maintained within Fuzzy-Sort. Subfigures (a) through (d) show the in-place creation of the "left", "middle", and "right" regions for subarrays $A[p \mathinner{..} s]$ and $B[p \mathinner{..} s]$.

$b_i < b$. These operations are preformed on lines 4 and 5 of Fuzzy-Sort using the following subroutines. In addition, their effect on subarrays $A[p \mathinner{..} s]$ and $B[p \mathinner{..} s]$ are shown in Figure 1.

```
Partition-Right($A, B, a, p, s$)
$i \leftarrow p - 1$
for $j \leftarrow p$ to $s - 1$
    do if $A[j] \leq a$
          then $i \leftarrow i + 1$
               exchange $A[i] \leftrightarrow A[j]$
               exchange $B[i] \leftrightarrow B[j]$
exchange $A[i+1] \leftrightarrow A[s]$
exchange $B[i+1] \leftrightarrow B[s]$
return $i + 1$
```

```
Partition-Left-Middle($A, B, b, p, r$)
$i \leftarrow p - 1$
for $j \leftarrow p$ to $r - 1$
    do if $B[j] < b$
          then $i \leftarrow i + 1$
               exchange $A[i] \leftrightarrow A[j]$
               exchange $B[i] \leftrightarrow B[j]$
exchange $A[i+1] \leftrightarrow A[r]$
exchange $B[i+1] \leftrightarrow B[r]$
return $i + 1$
```

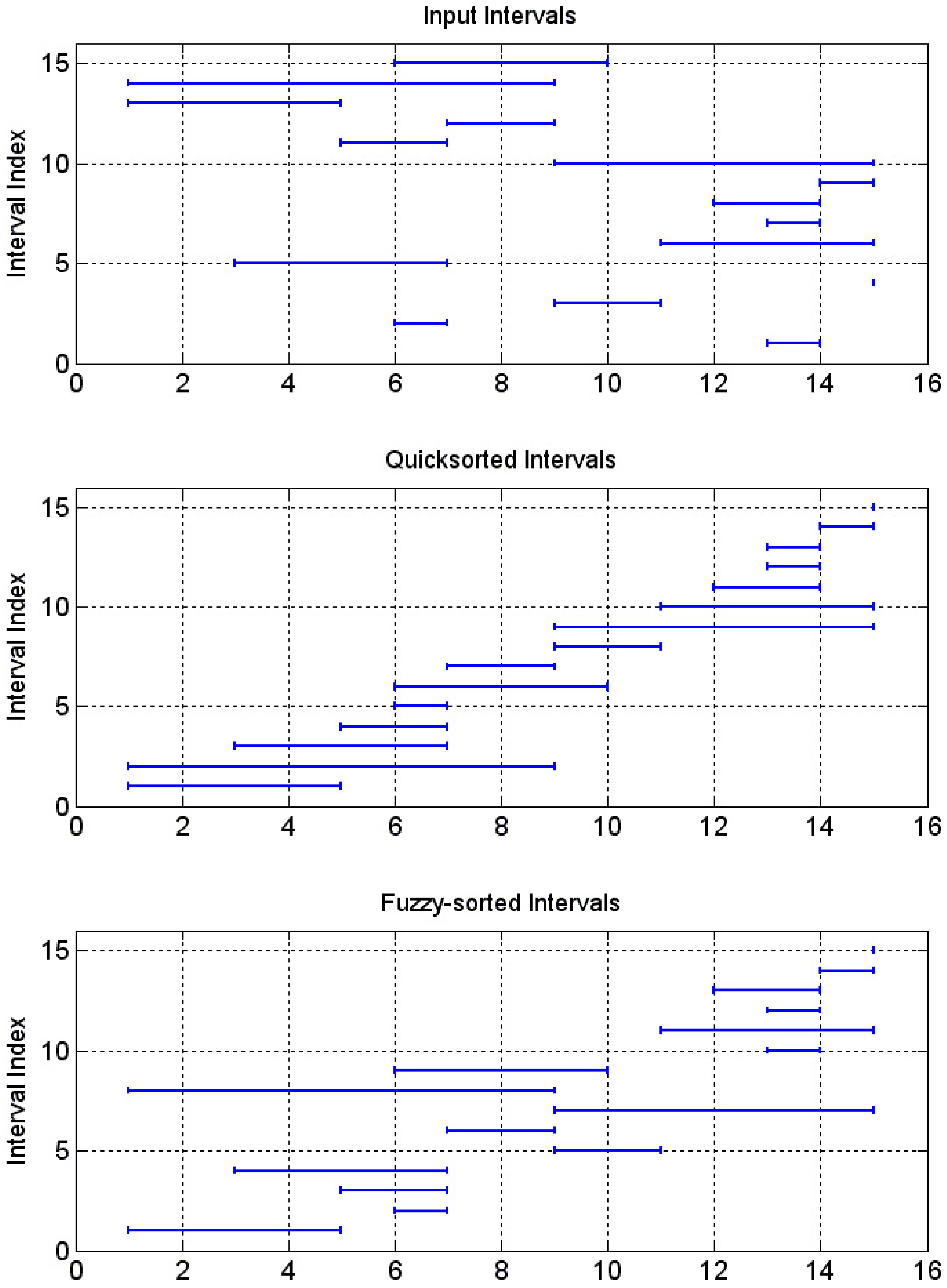


Figure 2: Comparison of Quicksort and Fuzzy-Sort results for the fuzzy interval sorting task. The top figure shows 15 randomly-selected intervals $[a_i, b_i]$ with $\{a_i, b_i\} \in \mathbb{Z}^+$. The middle figure shows the fuzzy-sort produced by quicksorting the left endpoints (i.e., the a_i's). The bottom figure shows a valid fuzzy-sort produced by Fuzzy-Sort. Notice that overlapping intervals can appear in an arbitrary order in the Fuzzy-Sort output.

3 Proof of Correctness

The proposed Fuzzy-Sort algorithm is similar to the randomized quicksort presented in [1]. In fact, Partition-Right and Partition-Left-Middle are nearly identical to the Partition procedure provided on page 146 of [1]. The primary difference, however, is the value of the pivot used to sort the intervals.

Rather than using the value of the left endpoint of the last interval, Partition-Right uses a pivot equal to a, the left endpoint of the region of overlap found by Find-Intersection. If we apply Partition-Right to the subarrays $A[p \mathrel{..} s]$ and $B[p \mathrel{..} s]$, then we produce the result shown in Figure 1(b). That is, we partition the subarray into two regions: the first with $a_i \leq a$ and the second with $a_i > a$. As a result, any interval in the "right" region can be placed in the output arrays *after* all intervals in the "left" or "middle" regions.

After we have completed the first partition, we still need to separate the "middle" intervals (which overlap) from the intervals which are to the left of the region of intersection. This is done by calling Partition-Left-Middle on the subarrays $A[p \mathrel{..} r]$ and $B[p \mathrel{..} r]$. Afterwards, we will have an in-place partitioning of the input arrays into "left", "middle", and "right" regions. This is shown schematically in Figure 1(d).

To complete our proof, we can recall the fact proved in Section 1: overlapping intervals in the "middle" region can appear in any permutation in the output array (so long as they appear after those in the "left" region and before those in the "right"). In conclusion, the recursive calls to Fuzzy-Sort on lines 6 and 7 will ensure that the final array is correctly fuzzy sorted in-place.

4 Analysis of Running Time

Proof of $\mathbf{\Theta(n \lg n)}$ Running Time: Since Fuzzy-Sort is nearly identical to randomized quicksort, we expect it to have a worst-case running time $\Theta(n \lg n)$ for a set of input intervals which do not overlap at any point. First, notice that lines 1 through 3 of Find-Intersection select a random interval as the initial *pivot interval* – similar to the randomly-selected pivot used in [1]. As discussed in Section 1, if the intervals are disjoint, then $[a, b]$ will simply be this initial interval.

Since for this example 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, line 3 is $\Theta(n)$ (as shown in Section 1). Fortunately, since the pivot interval $[a, b]$ is randomly-selected, the expected sizes of the "left" and "right" subarrays are both $\lfloor \frac{n}{2} \rfloor$ – as proven on page 150 of [1]. In conclusion, the reccurrence for the running time is

$$T(n) \leq 2T(n/2) + \Theta(n)$$

which is identical to that of randomized quicksort. As a result, we find that the worst-case expected running time of Fuzzy-Sort is $\Theta(n \lg n)$. For example, the worst-case will occur when the input intervals are disjoint. (QED)

Proof of Best-case $\mathbf{\Theta(n)}$ Running Time: As proven in Section 1, Find-Intersection will always return a non-empty region of overlap $[a, b]$ containing c_* if the intervals all overlap at c_*. For this situation, every interval will be within the "middle" region shown in Figure 1(d). Since the "left" and "right" subarrays will be empty, lines 6 and 7 of Fuzzy-Sort are $\Theta(1)$. As a result, there is no recursion and the running time of Fuzzy-Sort is determined by the $\Theta(n)$ running time required to find the region of overlap. As a result, if the input intervals all overlap at a point, then the expected worst-case running time is $\Theta(n)$. (QED)

5 Conclusion

This write-up has presented a modification of QUICKSORT, as presented in [1], to allow fuzzy sorting of intervals. Utilizing an architecture similar to randomized quicksort, the proposed algorithm has an expected worst-case running time of $\Theta(n \lg n)$. When the intervals all overlap at a point, however, the expected worst-case running time is only $\Theta(n)$. In conlcusion, we find that for the special case of fuzzy intervals, comparison-based sorting can be accomplished in less than $\Theta(n \lg n)$ by expoiting the property that overlapping intervals can appear in any permutation in the output.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill, 2001.