1. *See the homework handout for a description of the* CRUEL *and* UNUSUAL *algorithms.* Assume for this problem that the input size n is always a power of 2.

   (a) Prove by induction that CRUEL correctly sorts any input array. *[Hint: Follow the smallest $n/4$ elements. Follow the largest $n/4$ elements. Follow the middle $n/2$ elements. What does* UNUSUAL *actually* ***do****??]*

Solution: The only difference between CRUEL and MERGESORT is that CRUEL calls UNUSUAL instead of MERGE. So to prove that CRUEL correctly sorts, it suffices to prove that UNUSUAL behaves exactly like MERGE.

Lemma 1. *For any array $A[1..n]$ such that n is a power of 2, the subarray $A[1..n/2]$ is sorted, and the subarray $A[n/2+1..n]$ is sorted, calling* UNUSUAL*$(A[1..n])$ correctly sorts the entire array $A[1..n]$.*

Proof: Let $A[1..n]$ be an arbitrary array such that n is a power of 2, and the two subarrays $A[1..n/2]$ and $A[n/2+1..n]$ are each sorted. Assume that UNUSUAL correctly sorts any array whose length is a power of 2 smaller than n and whose first and second halves are sorted.

To simplify notation, we name the four quarters of A as follows:

$$\begin{aligned} A_1 &:= A[1..n/4], \\ A_2 &:= A[n/4+1..n/2], \\ A_3 &:= A[n/2+1..3n/4], \\ A_4 &:= A[3n/4+1..n]. \end{aligned}$$

These names refer to the ***array addresses***, not the ***array contents***. By assumption, the subarrays $A_1 \cup A_2$ and $A_3 \cup A_4$ are initially sorted. We separately track the four quartiles of A through the execution of UNUSUAL(A). There are three cases to consider.

- First, consider the smallest $n/4$ elements of A.
  - The smallest $n/4$ elements all initially lie in the subarrays A_1 and A_3.
  - After the for-loop swaps the contents of A_2 and A_3, the smallest $n/4$ elements of A lie in $A_1 \cup A_2$. Moreover, the subarrays A_1 and A_2 are still sorted.
  - The inductive hypothesis implies that the first recursive call to UNUSUAL sorts $A_1 \cup A_2$. Thus, after this call, A_1 contains the $n/4$ smallest elements in sorted order.
  - The rest of UNUSUAL does not modify A_1.
- Next, consider the largest $n/4$ elements of A.
  - The largest $n/4$ elements all initially lie in the subarrays A_2 and A_4.
  - After the for-loop swaps the contents of A_2 and A_3, the largest $n/4$ elements of A lie in $A_3 \cup A_4$. Moreover, the subarrays A_3 and A_4 are still sorted.
  - The first recursive call to UNUSUAL does not modify A_3 or A_4.

- The inductive hypothesis implies that the second recursive call to UNUSUAL sorts $A_3 \cup A_4$. Thus, after this call, A_4 contains the $n/4$ largest elements in sorted order.
- The rest of UNUSUAL does not modify A_4.

- Finally, consider the $(n/4+1)$th through $(3n/4)$th smallest elements of A, which we call the *middle* elements.
  - The first recursive call to UNUSUAL moves all middle elements out of A_1.
  - The second recursive call to UNUSUAL moves all middle elements out of A_4.
  - At this point, all middle elements lie in the subarrays A_2 and A_3, but possibly in the wrong order. However, A_2 is sorted and A_3 is sorted.
  - The inductive hypothesis implies that the third recursive call to UNUSUAL sorts $A_2 \cup A_3$, putting each middle element into its correct location.

We conclude that UNUSUAL correctly sorts the entire array $A[1..n]$. □

CRUEL's correctness now follows immediately from the correctness of MERGESORT. ■

Rubric: 6 points =
+ 5 for proving UNUSUAL=MERGE (standard induction rubric!)
+ 1 for observing that CRUEL=MERGESORT

(b) Prove that CRUEL would *not* correctly sort if we removed the for-loop from UNUSUAL.

Solution: With this modification, CRUEL([3, 4, 1, 2]) returns [3, 1, 4, 2]. ■

(c) Prove that CRUEL would *not* correctly sort if we swapped the last two lines of UNUSUAL.

Solution: With this modification, CRUEL([3, 4, 1, 2]) returns [1, 3, 2, 4]. ■

(d) State and solve a recurrence for the running time of UNUSUAL.

Solution: The running time of UNUSUAL obeys the recurrence $T(n) = 3T(n/2) + O(n)$. The recursion tree technique (with geometrically increasing level sums)—or the analysis of Karatsuba's algorithm shown in class—gives us the solution $T(n) = \boldsymbol{O(n^{\lg 3}) = O(n^{1.585})}$. ■

(e) State and solve a recurrence for the running time of CRUEL.

Solution: Part (d) implies that the running time of CRUEL obeys the recurrence $T(n) = 2T(n/2) + O(n^{\lg 3})$. The recursion tree technique (with geometrically decreasing level sums) gives us the solution $T(n) = \boldsymbol{O(n^{\lg 3}) = O(n^{1.585})}$. ■

Rubric: 1 point each for parts (b)–(e).

2. Dakshita has an array $A[1..n]$ of records, each with two numerical fields $A[i].birth$ and $A[i].death$ and a string field $A[i].name$. She wants to compute the maximum, over all indices $i \neq j$, of the overlap length

$$\min\{A[i].death,\ A[j].death\} - \max\{A[i].birth,\ A[j].birth\}.$$

Describe and analyze an efficient algorithm to solve Dakshita's problem.

Solution (divide and conquer): We start by sorting the input array A by $birth$. (This only needs to be done once at the beginning, not inside any recursive call.) Then if $n < 1000$, we solve the problem by brute force; otherwise we proceed as follows:

- Recursively find the largest overlap between two intervals in the left subarray $A[1..n/2]$.
- Recursively find the largest overlap between two intervals in the right subarray $A[n/2+1..n]$.
- Find the largest overlap between an interval in the left subarray $A[1..n/2]$ and an interval in the right subarray $A[n/2+1..n]$ as follows:
  - Find the index $1 \le \ell \le n/2$ such that $A[\ell].death$ is as large as possible, by brute force. (No other interval in the left subarray has larger overlap with any interval in the right subarray.)
  - Find the index $n/2 < r \le n$ such that $A[\ell]$ and $A[r]$ overlap as much as possible, by brute force.
- Finally, we return the largest of those three overlaps.

The initial sort takes $O(n \log n)$ time. The running time of the main recursive algorithm obeys the standard mergesort recurrence $T(n) = 2T(n/2) + O(n)$. So the main algorithm also runs in $O(n \log n)$ time. We conclude that the overall algorithm runs in $\boldsymbol{O(n \log n)}$ ***time***. ■

Solution (iterative): Start by sorting the input array by $birth$. Let's call one interval $A[i]$ is *older* than another interval $A[j]$ if $i < j$, or equivalently, if $A[i].birth < A[j].birth$. For each index j, we define two functions:

- $overlap(j) = \max\{|A[i] \cap A[j]| \mid i < j\}$ is the largest overlap between $A[j]$ and an older interval $A[i]$.
- $lastdeath(j) = \max\{A[i].death \mid i < j\}$ is the latest death among all intervals older than $A[j]$.

The function $lastdeath(j)$ satisfies a simple recurrence:

$$lastdeath(j) = \begin{cases} -\infty & \text{if } j = 1 \\ max\{lastdeath(j-1),\ A[j-1].death\} & \text{otherwise} \end{cases}$$

There are two cases to consider:

- If $lastdeath(j) > A[j].death$, then some older interval $A[i]$ completely contains $A[j]$, and so $overlap(j) = A[j].death - A[j].birth$.

- If $lastdeath(j) < A[j].death$, then no older interval completely contains $A[j]$, and so $overlap(j) = lastdeath(j) - A[j].birth$.

In both cases, we can compute both $overlap(j)$ and $lastdeath(j+1)$ in constant time for each j. Here is the resulting algorithm:

```
LargestOverlap(A[1..n]):
  sort A by birth
  maxoverlap ← 0
  lastdeath ← −∞
  for j ← 1 to n
      if lastdeath > A[j].death
          overlap ← A[j].death − A[j].birth
      else
          overlap ← lastdeath − A[j].birth
          lastdeath ← A[j].death
      maxoverlap ← max{overlap, maxoverlap}
  return maxoverlap
```

The overall algorithm runs in ***O(n log n) time***; the running time is dominated by the initial sort. ■

Rubric: 10 points = 2 for initial sort + 6 for other algorithm details (see below) + 2 for time analysis.

- For divide-and-conquer algorithm: 2 for base case (brute force) + 4 for recursive case
- For iterative algorithm: 2 for initialization + 2 for updating *maxoverlap* correctly + 2 for updating *lastdeath* correctly.

These solutions are more detailed than necessary for full credit. These are not the only correct $O(n \log n)$-time algorithms.

Max 3 points for a correct $O(n^2)$-time brute-force algorithm.

No penalty for assuming that at least one pair of intervals actually overlaps.