1. Consider the following cruel and unusual sorting algorithm, proposed by Gary Miller:

```
Cruel(A[1..n]):
  if n > 1
    Cruel(A[1..n/2])
    Cruel(A[n/2+1..n])
    Unusual(A[1..n])
```

```
Unusual(A[1..n]):
  if n = 2
    else
      for i ← 1 to n/4
        swap A[i + n/4] ↔ A[i + n/2]
      Unusual(A[1..n/2])  ((recurse on left half))
      Unusual(A[n/2+1..n])  ((recurse on right half))
      Unusual(A[n/4+1..3n/4])  ((recurse on middle half))
```

The comparisons performed by Miller’s algorithm do not depend at all on the values in the input array; such a sorting algorithm is called oblivious. 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?]

(b) Prove that Cruel would not correctly sort if we removed the for-loop from Unusual.

(c) Prove that Cruel would not correctly sort if we swapped the last two lines of Unusual.

(d) What is the running time of Unusual? Justify your answer.

(e) What is the running time of Cruel? Justify your answer.

2. Dakshita is putting together a list of famous cryptographers, each with their dates of birth and death: al-Kindi (801–873), Giovanni Fontana (1395–1455), Leon Alberti (1404–1472), Charles Babbage (1791–1871), Alan Turing (1912–1954), Joan Clarke (1917–1996), Ann Caracristi (1921–2016), and so on. She wonders which two cryptographers on her list had the longest overlap between their lifetimes. For example, among the seven example cryptographers, Clarke and Caracristi had the longest overlap of 45 years (1921–1966).
Dakshita formalizes her problem as follows. The input is an array $A[1..n]$ of records, each with two numerical fields $A[i].birth$ and $A[i].death$ and a string field $A[i].name$. The desired output is the maximum, over all indices $i \neq j$, of the overlap length


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

[Hint: Start by splitting the list in half by birth date. Do not assume that cryptographers always die in the same order they are born. Assume that birth and death dates are distinct and accurate to the nanosecond.]
Solved Problems

4. Suppose we are given two sets of \( n \) points, one set \( \{p_1, p_2, \ldots, p_n\} \) on the line \( y = 0 \) and the other set \( \{q_1, q_2, \ldots, q_n\} \) on the line \( y = 1 \). Consider the \( n \) line segments connecting each point \( p_i \) to the corresponding point \( q_i \). Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect, in \( O(n \log n) \) time. See the example below.

Your input consists of two arrays \( P[1..n] \) and \( Q[1..n] \) of \( x \)-coordinates; you may assume that all \( 2n \) of these numbers are distinct. No proof of correctness is necessary, but you should justify the running time.

Solution: We begin by sorting the array \( P[1..n] \) and permuting the array \( Q[1..n] \) to maintain correspondence between endpoints, in \( O(n \log n) \) time. Then for any indices \( i < j \), segments \( i \) and \( j \) intersect if and only if \( Q[i] > Q[j] \). Thus, our goal is to compute the number of pairs of indices \( i < j \) such that \( Q[i] > Q[j] \). Such a pair is called an inversion.

We count the number of inversions in \( Q \) using the following extension of mergesort; as a side effect, this algorithm also sorts \( Q \). If \( n < 100 \), we use brute force in \( O(1) \) time. Otherwise:

- Color the elements in the Left half \( Q[1..\lfloor n/2 \rfloor] \) blue.
- Color the elements in the Right half \( Q[\lfloor n/2 \rfloor + 1..n] \) red.
- Recursively count inversions in (and sort) the blue subarray \( Q[1..\lfloor n/2 \rfloor] \).
- Recursively count inversions in (and sort) the red subarray \( Q[\lfloor n/2 \rfloor + 1..n] \).
- Count red/blue inversions as follows:
  - Merge the sorted subarrays \( Q[1..n/2] \) and \( Q[n/2 + 1..n] \), maintaining the element colors.
  - For each blue element \( Q[i] \) of the now-sorted array \( Q[1..n] \), count the number of smaller red elements \( Q[j] \).

The last substep can be performed in \( O(n) \) time using a simple for-loop:
CountRedBlue(A[1..n]):
  count ← 0
  total ← 0
  for i ← 1 to n
    if A[i] is red
      count ← count + 1
    else
      total ← total + count
  return total

Merge and CountRedBlue each run in \(O(n)\) time. Thus, the running time of our inversion-counting algorithm obeys the mergesort recurrence

\[ T(n) = 2T(n/2) + O(n) \]

(We can safely ignore the floors and ceilings in the recursive arguments.) We conclude that the overall running time of our algorithm is \(O(n \log n)\), as required.

**Rubric:** This is enough for full credit.

In fact, we can execute the third merge-and-count step directly by modifying the Merge algorithm, without any need for “colors”. Here changes to the standard Merge algorithm are indicated in red.

MergeAndCount(A[1..n], m):
  i ← 1; j ← m + 1; count ← 0; total ← 0
  for k ← 1 to n
    if j > n
      B[k] ← A[i]; i ← i + 1; total ← total + count
    else if i > m
      B[k] ← A[j]; j ← j + 1; count ← count + 1
    else if A[i] < A[j]
      B[k] ← A[i]; i ← i + 1; total ← total + count
    else
      B[k] ← A[j]; j ← j + 1; count ← count + 1
  for k ← 1 to n
    A[k] ← B[k]
  return total

We can further optimize MergeAndCount by observing that \(count\) is always equal to \(j - m - 1\), so we don’t need an additional variable. (Proof: Initially, \(j = m + 1\) and \(count = 0\), and we always increment \(j\) and \(count\) together.)
MERGEANDCOUNT2($A[1 \ldots n], m$):

\[
i \leftarrow 1; \quad j \leftarrow m + 1; \quad total \leftarrow 0
\]

for $k \leftarrow 1$ to $n$

if $j > n$

\[
B[k] \leftarrow A[i]; \quad i \leftarrow i + 1; \quad total \leftarrow total + j - m - 1
\]

else if $i > m$

\[
B[k] \leftarrow A[j]; \quad j \leftarrow j + 1
\]

else if $A[i] < A[j]$

\[
B[k] \leftarrow A[i]; \quad i \leftarrow i + 1; \quad total \leftarrow total + j - m - 1
\]

else

\[
B[k] \leftarrow A[j]; \quad j \leftarrow j + 1
\]

for $k \leftarrow 1$ to $n$

\[
A[k] \leftarrow B[k]
\]

return $total$

MERGEANDCOUNT2 still runs in $O(n)$ time, so the overall running time is still $O(n \log n)$, as required.

\[\square\]