1. (a) Prove that CRUEL correctly sorts any input array.

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. *If $A[1..n/2]$ is sorted and $A[n/2+1..n]$ is sorted, then calling* UNUSUAL$(A[1..n])$ *correctly sorts the entire array $A[1..n]$.*

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

$$A_1 := A[1..n/4], \quad A_2 := A[n/4+1..n/2], \quad A_3 := A[n/2+1..3n/4], \quad A_4 := A[3n/4+1..n].$$

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

i. First, consider the smallest $n/4$ elements of A.
   - The smallest $n/4$ elements of A all initially lie in A_1 and A_3.
   - After the for-loop swaps A_2 and A_3, the smallest $n/4$ elements of A lie in A_1 and A_2. Moreover, 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 of A in sorted order.
   - The rest of UNUSUAL does not modify A_1.

ii. Next, consider the largest $n/4$ elements of A.
   - The largest $n/4$ elements all initially lie in A_3 and A_4.
   - After the for-loop swaps A_2 and A_3, the largest $n/4$ elements of A lie in A_3 and A_4. Moreover, 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 of A in sorted order.
   - The rest of UNUSUAL does not modify A_4.

iii. Now 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 last recursive call to UNUSUAL sorts A_2 and A_3, putting each middle element into its correct location.

We conclude that UNUSUAL puts every element of A into its correct location in sorted order. □

The correctness of CRUEL follows immediately by induction, following exactly the same argument as MERGESORT. ■

Rubric: 4 points = 1 for first quarter + 1 for last quarter + 1 for middle half + 1 for final induction. This is more detail than necessary for full credit. This is not the only correct proof.

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

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

Rubric: 1 point; all or nothing. This is not the only counterexample.

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

Solution: With this modification, Cruel($[3,4,1,2]$) returns the array $[3,1,2,4]$. ■

Rubric: 1 point; all or nothing. This is not the only counterexample.

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

Solution: The running time of UNUSUAL satisfies the recurrence $T(n) = 3T(n/2) + O(n)$. Recursion trees, the Master Theorem, and the lecture notes on Karatsuba multiplication all imply that $T(n) = \boldsymbol{O(n^{\lg 3})}$. ■

Rubric: 2 points = 1 for recurrence + 1 for final time bound.

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

Solution: The analysis in part (d) implies that the running time of CRUEL satisfies the recurrence $T(n) = 2T(n/2) + O(n^{\lg 3})$. Recursion trees and the Master Theorem both imply that $T(n) = \boldsymbol{O(n^{\lg 3})}$. ■

Rubric: 2 points. **Everyone gets full credit for this subproblem, because the first solution we posted was incorrect.**

2. For this problem, a *subtree* of a binary tree means any connected subgraph. A binary tree is *complete* if every internal node has two children, and every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the *largest complete subtree* of a given binary tree. Your algorithm should return the root and the depth of this subtree.

Solution: Our algorithm computes and stores a value $CD(v)$ at every node v, equal to the depth of the largest complete subtree ***whose root is v***. (CD stands for 'complete depth'.) This value can be defined recursively as follows:

$$CD(v) = \begin{cases} -1 & \text{if } v = \textsc{Null} \\ \min\{CD(\mathit{left}(v)),\ CD(\mathit{right}(v))\} + 1 & \text{otherwise} \end{cases}$$

The base case $CD(\textsc{Null}) = -1$ ensures that $CD(v) = 0$ for any node v with less than two children. (If we used nodes with less than two children as an explicit base case, the resulting recursive algorithm might not visit every node in T.) This recurrence translates directly into the following recursive algorithm:

```
ComputeCD($v$):
  if $T = \textsc{Null}$
      return $-1$
  else
      $LCD \leftarrow$ ComputeCD($\mathit{left}(v)$)
      $RCD \leftarrow$ ComputeCD($\mathit{right}(v)$)
      $CD(v) \leftarrow min\{LCD, RCD\} + 1$
      return $CD(v)$
```

This algorithm executes a postorder traversal of T, spending $\Theta(1)$ time at each node, so the overall running time is $\Theta(n)$. In fact, we can explicitly describe the algorithm as a postorder traversal as follows:

```
ComputeCDs($T$):
  for each node $v$ in $T$ in postorder:
      if $\mathit{left}(v) = \textsc{Null}$ or $\mathit{right}(v) = \textsc{Null}$
          $CD(v) \leftarrow 0$
      else
          $CD(v) \leftarrow \min\{CD(\mathit{left}(T)),\ CD(\mathit{right}(T))\} + 1$
```

The largest complete subtree of T is the largest complete subtree rooted at some node of T. To find this subtree, we maintain a single global variable and add a wrapper subroutine.

```
LargestCompleteSubtree($T$):
  $maxCD \leftarrow -\infty$
  ComputeCDs($T$)
  return $(maxRoot, maxCD)$
```

```
ComputeCDs($T$):
  for each node $v$ in $T$ in postorder:
      if $\mathit{left}(v) = \textsc{Null}$ or $\mathit{right}(v) = \textsc{Null}$
          $CD(v) \leftarrow 0$
      else
          $CD(v) \leftarrow \min\{CD(\mathit{left}(T)),\ CD(\mathit{right}(T))\} + 1$
          if $CD(v) > maxCD$
              $maxRoot \leftarrow v$; $maxCD \leftarrow CD(v)$
```

This algorithm still runs in $\mathbf{\Theta(n)}$ ***time***. ■

Rubric: 10 points max = 5 for correct algorithm + 3 for proof of correctness + 2 for time analysis. −1 for missing or incorrect base case in the algorithm. This is not the only correct way to describe the algorithm. The proof is more detailed than necessary for full credit.

3. (a) Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$. Describe an algorithm to find the median of the union of A and B in $O(\log n)$ time. Assume the arrays contain no duplicate elements.

Solution: Here is the algorithm:

```
MedianOfUnion($A[1..n], B[1..n]$):
  if $n \le 17$
      compute median($A \cup B$) by brute force
  $m \leftarrow \lceil n/2 \rceil$
  if $A[m] > B[n-m+1]$
      return MedianOfUnion($A[1..m], B[n+1-m..n]$)
  else
      return MedianOfUnion($A[m..n], B[1..n+1-m]$)
```

Correctness: We prove the algorithm correct by induction.

- If $n \le 17$, the algorithm is correct. Brute force is slow, but it works.
- Suppose $n > 17$ and $A[m] > B[n-m+1]$. All $n-m$ elements of $A[m+1..n]$ are larger than the $n+1$ elements of $A[1..m] \cup B[1..n-m+1]$, and thus are larger than median($A \cup B$). Symmetrically, all $n-m$ elements of $B[1..n-m]$ are smaller than median($A \cup B$). Since we discard the same number of elements above and below, we have median($A[1..m] \cup B[n-m+1..n]$) = median($A \cup B$). The arrays $A[1..m]$ and $B[n-m+1..n]$ both have size $m = \lceil n/2 \rceil < n$ (because $n > 1$). Thus, by the induction hypothesis, the recursive call MedianOfUnion($A[1..m], B[n+1-m..n]$) correctly computes median($A \cup B$).
- The final case $n > 17$ and $A[m] < B[n-m+1]$ is similar. The $m-1$ smallest elements of A are all smaller than median($A \cup B$), and the $m-1$ largest elements of B are all larger than median($A \cup B$), so we can safely discard them without changing the median. The remaining arrays $A[m..n]$ and $B[1..n+1-m]$ both have size $n-m+1 = n - \lceil n/2 \rceil + 1 < n$ (because $n > 2$), so we can safely hand the job off to the Recursion Fairy.

Running time: Both recursive calls consider arrays of size at most $n/2+1$, so the running time obeys the recurrence $T(n) \le O(1) + T(n/2+1)$. We can remove the $+1$ in the recursive argument with a domain transformation, giving us the standard binary-search recurrence $T(n) \le O(1) + T(n/2)$. So the algorithm runs in ***$O(\log n)$ time***, as required. ■

Rubric: 4 points = 2 for correct algorithm + 1 for proof of correctness + 1 for time analysis. This is not the only correct solution; in particular, there's nothing special about the number 17. This solution is more detailed than necessary for full credit. Common bugs to watch for (-1 each):

- Throwing out nothing when $n = 2$ or $n = 3$, causing an infinite loop. (This is the main reason for the conservative base case $n \le 17$. Yeah, I could have used $n \le 2$, but why bother?)
- Throwing out *everything* when $n = 2$ or $n = 3$. (Another reason for a conservative base case.)
- Discarding the median itself (which is why I kept both $A[m]$ and $B[n+1-m]$ in both recursive calls).
- Discarding more elements above the median than below, or vice versa, especially when n is even (which is why I used $B[n+1-m]$ instead of $B[m]$ or $B[m+1]$).

(b) Now suppose we are given *three* sorted arrays $A[1..n]$, $B[1..n]$, and $C[1..n]$. Describe an algorithm to find the median element of $A \cup B \cup C$ in $O(\log n)$ time.

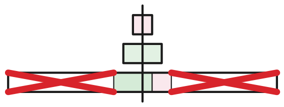
Solution: As in part (a), the main strategy is to repeatedly throw away two equal-size subarrays, one larger than the median and the other smaller than the median. In each iteration, we discard a constant fraction of at least one of the three arrays. Thus, after $O(\log n)$ iterations, all three arrays have constant size, at which point we can find the median by brute force.

The input consists of three sorted arrays $A[1..a]$, $B[1..b]$, $C[1..c]$. Although the original input arrays all have the same size, our recursive calls do not keep the sizes equal, so we must explicitly allow different array sizes. Let $N = a + b + c$ and $M = \lceil N/2 \rceil$. We define the median of $A \cup B \cup C$ to be its Mth smallest element. Without loss of generality, we assume $a \le b \le c$ (otherwise, just permute the variable names).

There are four cases to consider.

(i) ***Suppose $a + b + c \le 100$.*** In this case, we compute median($A \cup B \cup C$) by brute force in $O(1)$ time.

(ii) ***Otherwise, suppose $a + b \le 25$.*** In this case, the algorithm discards only elements of C, as suggested by the following figure.



Each element $C[i]$ has rank between i and $i + (a + b)$ in $A \cup B \cup C$, so

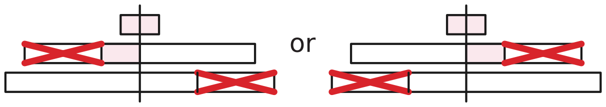
$$C[M - (a + b)] \le \text{median}(A \cup B \cup C) \le C[M].$$

The definition $M = \lceil (a + b + c)/2 \rceil$ implies that $M - (a + b) \ge c - M$. Thus, we can safely discard the top and bottom $c - M$ elements of C without changing the median:

$$\text{median}(A \cup B \cup C) = \text{median}(A \cup B \cup C[c - M + 1..M]).$$

We compute the latter median recursively. The subarray $C[c - M + 1..M]$ has size $2M - c \le N + 1 - c = a + b + 1 \le 26$, so the recursive call fits case (i). Thus, the algorithm also runs in $O(1)$ time in case (ii).

(iii) ***Otherwise, suppose $a \le 5$.*** In this case, we discard the same number of elements from one end of B and the other end of C, as suggested by the following figure.



Let $j = \lceil b/2 \rceil$ and $k = \lceil c/2 \rceil$. As in part (a), there are two subcases to consider:

- Suppose $B[j] < C[k]$. In this case, we discard the bottom $j - a - 1$ elements of B and the top $j - a - 1$ elements of C:

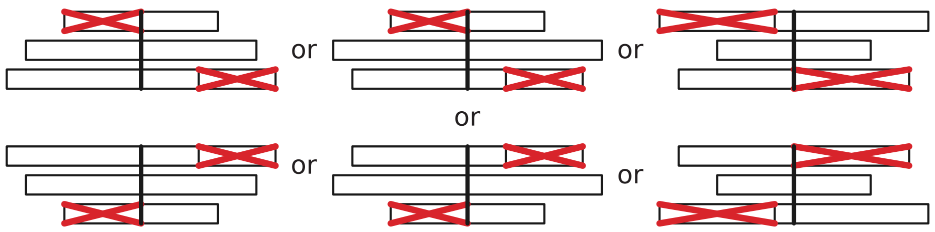
$$\text{median}(A \cup B \cup C) = \text{median}(A \cup B[j - a..b] \cup C[1..c - j + a + 1]).$$

- If $B[j] > C[k]$, we discard the top $(b - j) - a$ elements of B and the bottom $(b - j) - a$ elements of C:

$$\text{median}(A \cup B \cup C) = \text{median}(A \cup B[1..j + a] \cup C[b - j - a + 1..c]).$$

We must have $b > 20 \geq 4a$, since otherwise we would be in case (i) or (ii). Thus, in both subcases, we discard at least one fourth of the elements of B before recursing. It follows that we spend at most $O(\log b) = O(\log N)$ time in case (iii) before falling to case (i) or (ii).

(iv) ***Finally, suppose a > 5.*** In this case, we discard elements from some pair of arrays, as suggested by the following figure:



Let $i = \lceil a/2 \rceil$, $j = \lceil b/2 \rceil$ and $k = \lceil c/2 \rceil$, and sort the elements $A[i]$, $B[j]$, and $C[k]$. There are six nearly identical cases to consider.

- If $A[i] < B[j] < C[k]$, we discard the bottom $i-1$ elements of A and the top $i-1$ elements of C and recurse.
- If $A[i] < C[k] < B[j]$, we discard the bottom $i-1$ elements of A and the top $i-1$ elements of B and recurse.
- If $C[k] < A[i] < B[j]$, we discard the top $b-j$ elements of B and the bottom $b-j$ elements of C and recurse.
- If $C[k] < B[j] < A[i]$, we discard the top $a-i$ elements of A and the bottom $a-i$ elements of C and recurse.
- If $B[j] < C[k] < A[i]$, we discard the top $a-i$ elements of A and the bottom $a-i$ elements of B and recurse.
- If $B[j] < A[i] < C[k]$, we discard the bottom $j-1$ elements of B and the top $j-1$ elements of C and recurse.

In each case, we discard roughly half of one of the three arrays. Thus, we spend at most $O(\log N)$ time in this case before falling to one of the previous cases.

In all four cases, the algorithm runs in $O(\log N)$ time, as required. ■

Rubric: 6 points = 2 for correct algorithm + 2 for proof of correctness + 2 for time analysis. This solution is more detailed than necessary for full credit. This is not the only correct solution; there are at least two other fruitful approaches:

- Apply case (iii) above until the smallest has size $O(1)$. Compute the median of the other two arrays using the algorithm from part (a), modified to handle different-size arrays. Then scan for $\text{median}(A \cup B \cup C)$ in $O(1)$ time.
- Instead of balancing the sizes of the discarded subarrays, describe an algorithm that finds the rth smallest element of $A \cup B \cup C$ for arbitrary r. This variant requires slightly less bookkeeping, but still requires all four cases.

Ignore off-by-one errors in recursive arguments, but watch for the following bugs:

- Assuming (implicitly) that the sizes of the input arrays are equal in recursive calls.
- Incorrectly handling the case where one array has size 1 (or 0) and thus can't shrink any more.

*4. ***Extra credit:*** Describe and analyze an algorithm to reconstruct a full binary tree from the output of Bob's unknown confused traversal algorithms. You may assume that the vertex labels of the unknown tree are distinct, and that every internal node has exactly two children.

Solution (clever recursion): We first establish a few helpful preliminary results. Let $Pre_T[1..n]$, $In_T[1..n]$, and $Post_T[1..n]$ denote the outputs of Bob's traversal algorithms on some full binary tree T with n nodes, or "Bob's traversals of T" for short. For any tree T with more than one node, several observations follow direction from Bob's pseudocode:

- $Pre_T[1]$ and $Post_T[n]$ are the root of T.
- $In_T[1]$ and $Post_T[1]$ are (possibly equal) nodes in the left subtree of T.
- $Pre_T[n]$ and $In_T[n]$ are (possibly equal) nodes in the right subtree of T.

It follows that the root of T is the only label that appears at the start of one traversal (which must be *Pre*) and at the end of a different traversal (which must be *Post*). Thus, *even if we do not know which traversal is which*, we can identify the three traversals and the root of T in $O(1)$ time. This assumption of ignorance is important for recursion.

The root of T is $In_T[r]$ for some index r; our previous observations imply that $1 < r < n$. Then the left subtree of T has $r - 1$ nodes and the right subtree has $n - r$ nodes. Specifically,

- $Pre_T[2..r]$, $In_T[1..r-1]$, and $Post_T[1..r-1]$ are Bob's traversals of the left subtree of T (possibly with the wrong names).
- $Pre_T[r+1..n]$, $In_T[r+1..n]$, and $Post_T[r..n-1]$ are Bob's traversals of the right subtree of T (possibly with the wrong names).

So finally, here is the algorithm. The input consists of the three traversals, possibly permuted.

```
RecoverTree(A[1..n], B[1..n], C[1..n]):
    if n = 1
        v ← new node
        key(v) ← A[1]
        return v
    ⟨⟨— Identify traversals —⟩⟩
    if A[1] = B[n] then Pre ← A; In ← C; Post ← A
    if A[1] = C[n] then Pre ← A; In ← B; Post ← C
    if B[1] = C[n] then Pre ← B; In ← A; Post ← C
    if B[1] = A[n] then Pre ← B; In ← C; Post ← A
    if C[1] = A[n] then Pre ← C; In ← B; Post ← A
    if C[1] = B[n] then Pre ← C; In ← A; Post ← B
    ⟨⟨— Rebuild tree —⟩⟩
    find r such that In[r] = Pre[1]
    v ← new node
    key(v) ← Pre[1]
    left(v) ← RecoverTree(Pre[2..r], In[1..r-1], Post[1..r-1])
    right(v) ← RecoverTree(Pre[r+1..n], In[r+1..n], Post[r..n-1])
    return v
```

The running time of this algorithm depends on how we find the index r (the line in red). The naive implementation is a simple linear scan; the resulting algorithm runs in $\boldsymbol{O(n^2)}$ ***time*** in the worst case. I'll describe two faster methods; one easy to code but difficult to analyze, the other easy to analyze but slightly trickier to code.

Dovetailed linear scan. The simplest way to find r is a direct linear scan of the array *In*, which takes $O(r)$ time. Then the running time of the algorithm is $T(n) = O(r) + T(r-1) + T(n-r)$, which gives us a worst-case running time of $\boldsymbol{O(n^2)}$; specifically, the worst case occurs when $r = n-1$ at every level of recursion. A simple *backward* linear scan of *In* takes $O(n-r)$ time, which leads to the same $O(n^2)$ worst-case running time overall, but with a different worst case $r = 2$. In fact, the worst case for the backward scan is the *best* case for the forward scan, and vice versa.

So what if we dovetail these two searches as follows?

```
for $i \gets 1$ to $n/2$
    if $In[i] = Pre[1]$
        $r \gets i$
        break
    if $In[n-i] = Pre[1]$
        $r \gets n-i$
        break
```

This dovetailed scan requires $O(\min\{r, n-r\})$ time, so the worst-case running time of RECOVERTREES becomes

$$T(n) = \max_{1\le r\le n} \Big\{O(\min\{r, n-r\}) + T(r-1) + T(n-r)\Big\}.$$

Setting $m = \min\{r, n-r\}$ and dropping the -1 in the first recursive argument simplifies this recurrence:

$$T(n) \le \max_{1\le m\le n/2} \Big\{O(m) + T(m) + T(n-m)\Big\}.$$

Notice that if $m = 1$ at every level of recursion, we would get a running time of $O(n)$, and if $m = n/2$ at every level of recursion, we would get a running time of $O(n\log n)$. I claim that $m = n/2$ is essentially the worst case, and thus RECOVERTREES runs in ***$O(n \log n)$ time***.

We can prove this claim by induction as follows. Suppose the non-recursive part of the algorithm takes at most m "steps"; we can multiply by an appropriate constant at the end to get the actual running time. We claim that the algorithm executes at most $cn\ln n$ steps overall, for some constant c. The induction hypothesis implies that

$$T(n) \le \max_{m\le n/2} \Big\{m + cm\ln m + c(n-m)\ln(n-m)\Big\}$$

The expression in braces is maximized when its derivative is zero. (Yep, calculus.)

$$\begin{aligned}\frac{d}{dm}\Big(m + cm\ln m + c(n-m)\ln(n-m)\Big) &= 1 + \ln m + 1 - \ln(n-m) - 1 \\ &= 1 + c\ln\frac{m}{n-m}\end{aligned}$$

This expression is zero when $m = n/(1+e^c)$; to simplify notation, define $\alpha = 1/(1+e^c)$. Plugging that value into our original recurrence gives us

$$\begin{aligned}T(n) &\le n + c\alpha n\ln(\alpha n) + c(1-\alpha)n\ln((1-\alpha)n) \\ &= n + cn\ln n + cn(\alpha\ln\alpha + (1-\alpha)\ln(1-\alpha))\end{aligned}$$

A bit more calculus implies that $\alpha\ln\alpha + (1-\alpha)\ln(1-\alpha)$ is maximized when $\alpha = 1/2$, so

$$T(n) \le n + cn\ln n - (c\ln 2)n$$

So as long as $c > 1/\ln 2 \approx 1.442695$, we have $T(n) \le cn\ln n$, as claimed.

Index dictionary. Another way to implement the red line "find r such that $In[r] = Pre[1]$" is to use a dictionary that records the index of every vertex label in each of the three input traversals. If we build this dictionary in a preprocessing phase, then we can find the index r using a single query.

The idea is especially easy to implement if the vertex labels are actually the integers 1 through n. In that case, we fill three new arrays $PreIndex[1..n]$, $InIndex[1..n]$, and $PostIndex[1..n]$ as follows:

```
for $i \gets 1$ to $n$
    $PreIndex[Pre[i]] \gets i$
    $InIndex[In[i]] \gets i$
    $PostIndex[Post[i]] \gets i$
```

To use these arrays, we need to modify RecoverTree slightly. Instead of passing three subarrays to the Recursion Fairy, we pass four integers:

- The size of the subtree we want to reconstruct.
- The first index in *Pre* for that subtree.
- The first index in *In* for that subtree.
- The first index in *Post* for that subtree.

Here, *Pre*, *In*, and *Post* are the *original* input arrays, not the local variables inside RecoverTree. Adapting our algorithm to use these indices, instead of passing subarrays, is tedious but straightforward. The resulting algorithm runs in ***$O(n)$ time***.

If the labels are *not* the integers 1 through n, we can still map the labels to integers between 1 and $2n$ using an open-addressed hash table (for example, using linear probing). That is, instead of using each vertex label ℓ as an index into *PreIndex* (for example), we use the position of ℓ in the hash table as an index into *PreIndex*. Each access to the hash table requires $O(1)$ expected time, so the overall algorithm runs in ***$O(n)$ expected time***.

If we insist on efficient worst-case behavior, we could use a balanced binary search tree instead of a hash table. Then each dictionary query requires $O(\log n)$ time, so the overall algorithm runs in ***$O(n \log n)$ time***. ■

Rubric: 10 points: Partial credit awarded at Jeff's discretion. This is not the only correct solution.