CS/Math 240: Introduction to Discrete Mathematics 3/10/2011

Lecture 15 : Review

Instructor: Dieter van Melkebeek Scribe: Dalibor Zelený

DRAFT

Today's lecture serves as a review of the material that will appear on your second midterm exam. We prove correctness of another sorting algorithm, mergesort, and also argue what its running time is. Mergesort is one of the fastest sorting algorithms.

15.1 Mergesort

We saw two examples of simple sorting algorithms, namely selection sort and bubble sort. The next homework assignment asks you to analyze quicksort which is one of the fastest sorting algorithms. Today we analyze another common fast sorting algorithm, mergesort.

15.1.1 Designing the Algorithm

Let's design a recursive algorithm for sorting an array. Recall that for a recursive solution, we need to reduce the problem into smaller instances of the same problem and then somehow combine the solutions. Here is one way to do it for sorting an array.

1. Split the array into two halves.
2. Sort each half.
3. Combine the two sorted halves into one array.

Let's call our sorting algorithm `MergeSort` and let `MergeSortedParts` be the procedure that combines two sorted parts of the array into a sorted version of the whole array.

Example 15.1: Start with the unsorted list $9, 8, 7, 1, 2, 4, 7, 5$. We split it into two lists, $9, 8, 7, 1$ and $2, 4, 7, 5$. We recursively sort each of these lists using `MergeSort`, and obtain the lists $1, 7, 8, 9$ and $2, 4, 5, 7$. Finally, we combine these two lists together into the list $1, 2, 4, 5, 7, 7, 8, 9$ using `MergeSortedParts`. ⊠

Before we describe `MergeSort` and `MergeSortedParts` formally, let's decide what input variables we need. Clearly, we need a variable, say A, for the array we are supposed to sort. Since the recursive calls work with subarrays of A, we need two variables to denote the starting and ending indices of those subarrays. Let's call these variables b for "begin" and e for "end", respectively. This is all information we need for `MergeSort`. We need an additional piece of information for `MergeSortedParts`, namely the index of the last element in the first half of A. Let's use m for this.

Example 15.2: In Example 15.1, the initial call to `MergeSort` has $b = 0$ and $e = 7$. When we split A in half, we get $m = 3$. The first recursive call is then with input $(b, e) = (0, 3)$, and the second recursive call is with input $(b, e) = (4, 7)$. ⊠

Now that we have decided on what inputs we need, we can write the specifications for `MergeSort` and `MergeSortedParts`. We also give a formal description of `MergeSort`.

Algorithm MergeSort(A, b, e)

Input: $b, e \in \mathbb{N}, b \leq e$
Input: A - an array $A[b..e]$ of integers
Side Effect: $A[b..e]$ is sorted in increasing order

(1) **if** $b = e$ **then return**
(2) $m \leftarrow \lfloor (b + e)/2 \rfloor$
(3) `MergeSort(`A, b, m`)`
(4) `MergeSort(`A, $m + 1$, e`)`
(5) `MergeSortedParts(`A, b, m, e`)`

Algorithm: Specification of MergeSortedParts(A, b, m, e)

Input: $b, m, e \in \mathbb{N}$, $b \leq m < e$
Input: A - an array $A[b..e]$ of integers where $A[b..m]$ and $A[m + 1..e]$ are sorted in increasing order.
Side Effect: $A[b..e]$ is sorted in increasing order.

Notice that our two algorithms have no output, and instead have side effects. This means that `MergeSort`, `MergeSortedParts`, and all recursive calls operate on the same array as opposed to getting one array as input and creating a new array that they return as output.

15.1.2 Proof of Correctness

Recall that there are two parts to a correctness proof for a recursive program. To prove partial correctness, we prove the program behaves correctly, assuming it terminates. Second, we prove the program terminates.

To prove partial correctness, we assume that all recursive calls with arguments that satisfy the preconditions return correct values and behave as described by the specification, and use it to show that program behaves as specified, assuming it terminates. We prove that the program terminates by showing that any chain of recursive calls eventually ends and that all loops, if any, terminate after some finite number of iterations.

15.1.2.1 Partial Correctness

In addition to assuming that all recursive calls satisfying the preconditions behave correctly, we also assume that `MergeSortedParts` is correct. We will prove correctness of `MergeSortedParts` later. Now let's use induction to prove partial correctness of `MergeSort`.

For the base case, we consider a one-element array, i.e., the situation when $b = e$. Then $A[b..e]$ is sorted since it only contains one element, so the algorithm does nothing and returns right away, which is the correct behavior.

Now suppose that $b < e$, so $A[b..e]$ contains at least two elements. Then we compute the middle index $m = \lfloor (b + e)/2 \rfloor$.

Since $b \leq e$, we have $b = 2b/2 = \lfloor 2b/2 \rfloor \leq \lfloor (b + e)/2 \rfloor$. This means that $b \leq m$, so the preconditions of the first recursive call to `MergeSort` are satisfied, and this call correctly sorts the subarray $A[b..m]$.

Next, since $b < e$, we have $e = 2e/2 > (e+b)/2 \geq \lfloor (e+b)/2 \rfloor = m$, so $m < e$, which means that $m + 1 \leq e$. Thus, the preconditions of the second recursive call to `MergeSort` are also satisfied, and this call correctly sorts the subarray $A[m + 1..e]$.

Finally, since we showed $b \leq m < e$, and the two subarrays $A[b..m]$ and $A[m+1..e]$ are sorted when we call `MergeSortedParts`, the preconditions of `MergeSortedParts` are satisfied. That means `MergeSortedParts` rearranges the elements of $A[b..e]$ so that they are sorted. This proves partial correctness.

15.1.2.2 Termination

To argue termination, we find a quantity that decreases with every recursive call. One possibility is the length of the part of A considered by a call to `MergeSort`, i.e., $n = e - b + 1$. Now let's argue by induction that `MergeSort` terminates.

For the base case, we have a one-element array. Then $b = e$, and the algorithm terminates in that case without making additional recursive calls.

Now suppose $n \geq 2$. Then $m < e$ by an argument from the proof of partial correctness, so the call on line 3 is with an array of length $n' = m - b + 1 < e - b + 1 = n$. We see that the first call sorts a smaller array than the original one. Similarly, since $m + 1 > m \geq b$, we have $n'' = e - (m + 1) + 1 < e - b + 1 = n$, so the second recursive call sorts an array of size $n'' < n$. This proves that the chain of recursive calls eventually ends, and the algorithm terminates.

As we will see in the next section, we can say more about the values of n' and n''. We will show that $n' = \lceil (e - b)/2 \rceil$ and $n'' = \lfloor (e - b)/2 \rfloor$ (where $\lceil y \rceil$ means y rounded up to the nearest integer, and is read as "the ceiling of y"). But for now, our bounds on n' and n'' are good enough.

15.1.3 Running Time Analysis

Before we discuss the running time of `MergeSort`, let's discuss the elementary operations we use as our measure of complexity. There are various operations used in `MergeSort`, such as recursive calls and mathematical operations used to compute m. There are also some other operations done in `MergeSortedParts` which we have not specified yet. So let's not specify the exact elementary operation, and consider any operation mentioned in this paragraph as an elementary operation (this is common in practice).

Just like we assumed in the proof of correctness that `MergeSortedParts` worked correctly, we now assume that the running time of `MergeSortedParts` on a subarray of length n takes at most $c \cdot n$ elementary operations. We will prove it at the end of lecture.

With this assumption, let's count the number of elementary operations made by `MergeSort`. Let $n = e - b + 1$ like in our proof of termination, and let n' and n'' be the lengths of the arrays in the two recursive calls to `MergeSort`. Since returning is the only operation when the array input to our algorithm has length 1, we have

$$T(n) \leq 2 + T(n') + T(n'') + cn, \qquad T(0) = 1 \tag{15.1}$$

where the 2 comes from the elementary operations on line 2 and cn comes from `MergeSortedParts`.

The quantity $T(n)$ is hard to analyze for two reasons. First, we have an inequality in (15.1) because `MergeSortedParts` may take less time and so could the recursive calls. Second, right now it is possible that n' and n'' are anywhere between 1 and $n - 1$. We deal with these two issues one by one.

The problem of different running times for different arrays of size n makes sense. Sorting an array that is almost sorted should take much less time than sorting an array whose elements are out of order in all kinds of ways. We have encountered a situation like this when we were arguing the complexity of the simple algorithm for finding greatest common divisors. For the simple algorithm

for finding greatest common divisors, we solved this issue by finding a related quantity, namely the maximum number of recursive calls made on inputs (a, b) with $a, b \leq n$. Thus, our final result was an upper bound on the running time. For mergesort, we will also find an upper bound on $T(n)$, and do so by turning the inequality in (15.1) into an equality. We define

$$T^*(n) = 2 + T^*(n') + T^*(n'') + cn, \qquad T^*(0) = 1. \tag{15.2}$$

You can prove by induction that $T(n) \leq T^*(n)$ for all $n \in \mathbb{N}$.

For our second problem, let's find the values for n' and n''. This should be possible since we know how m gets computed on line 2 of `MergeSort`.

First suppose $e - b$ is even, so n is odd. In this case $(e - b)/2$ is an integer, so $m = (b + e)/2$, and

$$n' = m - b + 1 = (e + b)/2 - b + 1 = (e - b)/2 + 1 = (n - 1)/2 + 1 = \lfloor (n - 1)/2 \rfloor + 1.$$

Since n is odd, $\lfloor n/2 \rfloor = \lfloor (n - 1)/2 \rfloor$, and $\lceil n/2 \rceil = \lfloor n/2 \rfloor + 1$. Then $n' = \lceil n/2 \rceil$. By a similar argument, we have $n'' = \lfloor n/2 \rfloor$.

When $e - b$ is odd, n is even, and we get $n' = n'' = \lfloor n/2 \rfloor = \lceil n/2 \rceil$.

We see that in both cases $n' = \lceil n/2 \rceil$ and $n'' = \lfloor n/2 \rfloor$. Thus, we can rewrite (15.2) as

$$T^*(n) = 2 + T^*(\lceil n/2 \rceil) + T^*(\lfloor n/2 \rfloor) + cn, \qquad T^*(0) = 1. \tag{15.3}$$

We now solve the recurrence (15.3). We use a different strategy than the one we've been using so far. Instead of computing a few terms, making a conjecture, and proving it by induction, we calculate the contribution towards $T^*(n)$ for each level in the *recursion tree* for computing $T^*(n)$, and then add up all the contributions to get an expression for $T^*(n)$. What this means is that at each level, we only count the terms from (15.3) that don't correspond to earlier terms in the sequence T^*.

At the first level, we only count $2 + cn$. We ignore the terms $T^*(\lceil n/2 \rceil)$ and $T^*(\lfloor n/2 \rfloor)$ as we will account for them when counting the contributions of levels further down the recursion tree.

At the next level, we count the contributions from the terms $T^*(\lceil n/2 \rceil)$ and $T^*(\lfloor n/2 \rfloor)$ that don't come from other T^* terms. For $T^*(\lceil n/2 \rceil)$ this is $2 + c\lceil n/2 \rceil$, and for $T^*(\lfloor n/2 \rfloor)$ this is $2 + c\lfloor n/2 \rfloor$. Since $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$, the combined contributions of the two are $4 + cn$.

For the level after that, we get a contribution of

$$\begin{aligned} 2 + c\left\lceil \frac{1}{2}\lceil n/2 \rceil \right\rceil + 2 + c\left\lfloor \frac{1}{2}\lceil n/2 \rceil \right\rfloor + 2 + c\left\lceil \frac{1}{2}\lfloor n/2 \rfloor \right\rceil + 2 + c\left\lfloor \frac{1}{2}\lfloor n/2 \rfloor \right\rfloor &= 8 + c\left(\lceil n/2 \rceil + \lfloor n/2 \rfloor\right) \\ &= 8 + cn \end{aligned}$$

We conclude that at every level l of the recursion tree, the total contribution to $T^*(n)$ is $2^l + cn$.

With all the rounding up and down, it may be somewhat difficult to argue how many levels the recursion tree has, which makes adding up the contributions of the individual levels tricky. To avoid this issue, we make a slight modification to (15.3). Seeing that the contribution of each level of the recursion tree is $2^l + cn$, we can get rid of all the rounding in (15.3) and get the same contribution per level. To simplify calculations even further, we drop the constant 2 from (15.3) to get

$$T^*(n) = 2T^*(n/2) + cn, \qquad T^*(0) = 1. \tag{15.4}$$

This is valid since the dominant term is cn, and, as we will see, there are at most $\log n$ levels in the recursion tree, so 2^l doesn't get a chance to dominate cn.

The recursion tree for the computation of $T^*(n)$ from (15.4) is balanced, and it is easier to argue about its height. Note that at level l, all the recursive calls are with arrays of length $n/2^{l-1}$. Recursion stops when $n/2^l < 1$, i.e., after $\log n$ recursive calls. At the bottom level, the algorithm returns, which we view as one elementary operation. Thus, the contribution of every level is cn, except for the last level where the contribution is n. Since there are a total of $\log n$ levels before we reach the bottom level where the algorithm returns, we get $T^*(n) = cn \log n + n = \Theta(n \log n)$.

15.1.4 Review of the Analysis Technique

Let's summarize the technique. We "unroll" the expression for $T^*(n)$, thus getting a recursion tree. For each level of recursion, we find its contribution to $T^*(n)$, and then sum up the contributions over all the levels to get an expression for $T^*(n)$. We point out that this technique is generally applicable to solving recurrences.

We also remark here that the bound on $T^*(n)$ we obtained is very tight. We can construct inputs on which the algorithm takes time $cn \log n$. In particular, one can show for all $n \geq 1$ that

$$cn \lfloor \log n \rfloor + cn - 1 \leq T^*(n) \leq cn \lceil \log n \rceil + cn - 1.$$

It is very uncommon to get such tight bounds.

15.2 The MergeSortedParts Procedure

Now that we have completed the discussion of the mergesort algorithm, let's return back to the subroutine `MergeSortedParts`. So far we have treated it as a black box and made assumptions about its correctness and running time. This may seem like cheating, but it is a good approach to take. Trying to prove a result using some additional assumptions without proof is a good sanity check by which one ensures it even makes sense to continue with the current proof attempt.

Now it's time to write down the `MergeSortedParts` algorithm formally and verify that our assumptions about it were correct.

15.2.1 Designing the Algorithm

Recall that we are given an array A that consists of two sorted parts, say X and Y. Our goal is to rearrange the elements of A so that the entire array is sorted.

The overall idea is that we take the two sorted parts we want to merge, and go through them left to right, adding elements to the merged array one at a time. Since X and Y are sorted, the first element of the merged array should be at the beginning of one of the sorted parts. So we compare the first elements of the two subarrays X and Y, and make the smaller element be the first element in the merged array. If the first element of X was smaller than the first element of Y, the second smallest element of the merged array is either the second element of X or the first element of Y. Again, we take the smaller one of them and put it in the second position in the merged array. If the smaller one was in X again, we look at the third element of X and the first element of Y next to decide what the third smallest element is in the merged array. Otherwise, we look at the second elements of X and Y next to decide what the third smallest element is in the merged array. This process continues until all of the subarray elements have been added to the merged array. If, at some point, we exhaust one of the subarrays, we just copy the rest of the other one at the end of the merged array.

Example 15.3: Consider the two subarrays $X = 1, 7, 8, 9$ and $Y = 2, 4, 5, 7$ from Example 15.1. Here are the steps our algorithm takes:

Step 1 Compare 1 from X and 2 from Y. Since $1 < 2$, place 1 in the first position in A.

Step 2 Compare 7 from X and 2 from Y. Since $2 < 7$, place 2 in the second position in A.

Step 3 Compare 7 from X and 4 from Y. Since $4 < 7$, place 4 in the third position in A.

Step 4 Compare 7 from X and 5 from Y. Since $5 < 7$, place 5 in the fourth position in A.

Step 5 Compare 7 from X and 7 from Y. They are equal, so take the one from X and put it in the fifth position in A.

Step 6 Compare 8 from X and 7 from Y. Since $7 < 8$, place 7 in the sixth position in A.

Step 7 Now we have exhausted all elements in Y, so we take the remaining elements of X and put them at the end of A.

Now A is the array $1, 2, 4, 5, 7, 7, 8, 9$, which is sorted. ⊠

We have already discussed what information we need as input to `MergeSortedParts`. The additional variables we need are the pointers into the two parts of the array we are trying to merge, X and Y, and a pointer into the merged array that tells us where to place the next element. We will use variables i and j for the former, and k for the latter. Finally, we need a temporary copy of $A[b..e]$ to ensure that we don't lose information by overwriting parts of A that have not been added to the merged array yet. Let B be the variable storing that copy. This array B consists of X and Y (X and Y are not variables; we have $X = B[d..m]$ and $Y = B[m + 1..e]$).

Algorithm MergeSortedParts(A, b, m, e)

Input: $b, m, e \in \mathbb{N}$, $b \leq m < e$
Input: A - an array $A[b..e]$ of integers where $A[b..m]$ and $A[m + 1..e]$ are sorted in increasing order.
Side Effect: $A[b..e]$ is sorted in increasing order.

```
(1)  B[b..e] ← A[b..e]
(2)  i ← b; j ← m + 1; k ← b
(3)  while i ≤ m and j ≤ e do
(4)      if B[i] ≤ B[j] then
(5)          A[k] ← B[i]
(6)          i ← i + 1
(7)      else
(8)          A[k] ← B[j]
(9)          j ← j + 1
(10)     end
(11)     k ← k + 1
(12) end
(13) if i ≤ m then A[k..e] ← B[i..m]
```

15.2.2 Proof of Correctness

Recall that to prove correctness of a program, we need to show partial correctness, i.e., that the algorithm behaves correctly assuming it terminates, and we need to show that it terminates.

15.2.2.1 Partial Correctness

Now we prove that `MergeSortedParts` works correctly. We start with partial correctness. That is, we show that if `MergeSortedParts` is given an array $A[b..e]$ and integers $b \leq m < e$ such that the subarrays $A[b..m]$ and $A[m+1..e]$ are sorted, it rearranges the elements so that $A[b..e]$ is sorted.

We copy the initial values of $A[b..e]$ into $B[b..e]$ and now we are free to modify $A[b..e]$ as we see fit. Intuitively, in each step we add to A an element of either $B[b..m]$ or $B[m+1..e]$ that is smallest from among the elements that have not yet been added to A. All elements that have already been added are at most as large as those that have not been added yet. Now the subarray of A consisting of elements that have been added already is $A[b..k-1]$, and this portion of the array A is sorted because we put in smaller elements before putting in any larger elements. Since i and j indicate the positions of the first elements of $B[b..m]$ and $B[m+1..e]$ that still need to be added to A, we decide on the following loop invariant.

Invariant 15.1. *After each iteration of the loop, $A[b..k-1]$ is sorted and consists of elements of $B[b..i-1]$ and $B[m+1..j-1]$. Furthermore, all elements in $A[b..k-1]$ are at most as large as all elements in $B[i..m]$ and $B[j..e]$.*

Here we remark that we could have used this invariant in order to design the Algorithm `MergeSortedParts`. In fact, we did this (although we didn't state it) in the second paragraph of Section 15.2.1 (starting with "The overall idea ...").

We also need an invariant that relates all the pointer variables. Since i is a pointer into $A[b..m]$, we should have $b \leq i \leq m+1$ (if $i = m+1$, the loop terminates). Similarly, we have $m+1 \leq j \leq e+1$. We should also add k to the picture. After k' iterations of the loop, i has changed from b to $b+i'$, j has changed from $m+1$ to $m+1+j'$, and we have $k' = i' + j'$. We then have $i' = i - b$ and $j' = j - (m+1)$. This means that $k = b + k' = b + i' + j' = b + i - b + j - (m+1) = i + j - (m+1)$. Thus, we have the following loop invariant.

Invariant 15.2. *After every iteration of the loop, $b \leq i \leq m+1 \leq j \leq e+1$ and $k = i+j-(m+1)$.*

Now that we have decided on invariants, let's see if we can use them to prove partial correctness of the algorithm. For the loop to terminate, we need $i \geq m+1$ or $j \geq e+1$. Invariant 15.2 tells us that $i \leq m+1$ and $j \leq e+1$. Thus, when the loop terminates, we have either $i = m+1$ or $j = e+1$.

Case 1: $i = m+1$. In this case we have all elements of $B[b..m]$ and $B[m+1..j-1]$ in $A[b..k-1]$ by Invariant 15.1. By Invariant 15.2, we also have $k = i + j - (m+1) = m + 1 + j - (m+1) = j$, so $k = j$. Since we never modified $A[k..e]$, we actually have $A[k..e] = B[j..e]$. Invariant 15.1 tells us that $A[b..k-1]$ is sorted, and all elements of $A[b..k-1]$ are at most as large as all elements of $B[j..e] = A[k..e]$. Note that $B[j..e]$ is sorted, which means $A[k..e]$ is sorted. Since we also said that all elements of $A[b..k-1]$ are upper-bounded by all elements of $A[k..e]$, this means $A[b..e]$ is sorted after the loop terminates. Since $i = m+1$, the last line doesn't execute, and the algorithm terminates with $A[b..e]$ sorted as desired.

Case 2: $j = e+1$. In this case we have all elements of $B[b..i-1]$ and $B[m+1..e]$ in $A[b..k-1]$ by Invariant 15.1. We also see that $A[b..k-1]$ is sorted. Also, all elements of $B[i..m]$ are at least as large as all elements of $A[b..k-1]$, and $B[i..m]$ is sorted. After the loop terminates, $k = i+j-(m+1) = i+e+1-(m+1) = e+i-m$. So there are $e-k+1 = m-i+1$ places to be filled in A, and we fill those places with a copy of $B[i..m]$ (that also consists of $m-i+1$ elements) into $A[k..e]$ to complete the rearrangement of the original A into one that is sorted.

We have managed to prove partial correctness using our invariants. To complete the proof of partial correctness, we prove our two invariants.

Proof of Invariant 15.2. We proceed by induction on the number of iterations of the loop.

When the algorithm begins, that is, after zero iterations, we initialize $i = b$ and $j = m + 1$. Since the preconditions tell us that $b \leq m < e$, this implies that $b \leq i \leq m + 1$. Furthermore, if $m < e$, then $m + 1 \leq e$, and we also have $m + 1 \leq j \leq e + 1$. Finally, in this case $k = b$, and $i + j - (m + 1) = b + (m + 1) - (m + 1) = b$. This proves the base case.

Now assume that the invariant holds after some iteration of the loop. We show that it holds after the next one as well. If $i > m$ or $j > e$, there is nothing to prove because there isn't going to be another iteration of the loop.

Thus, assume $i \leq m$ and $j \leq e$. Inside the loop body, we increase one of i or j by one and leave the other one fixed. We also increase k by one. If i', j' and k' are the values of i, j, and k after the next iteration of the loop, we see that $i' + j' = i + j + 1$ and $k' = k + 1$. Our induction hypothesis tells us that $k = i + j - (m + 1)$, so we have $k' = k + 1 = i + j - (m + 1) + 1 = (i + j + 1) - (m + 1) = i' + j' - (m + 1)$.

Two observations complete the proof of the inductive step. First, since $i \leq m$ and $j \leq e$ and we increase one of i and j by one and leave the other one fixed, we get $i' \leq i + 1 \leq m + 1$ and $j' \leq j + 1 \leq e + 1$. Second, the values of i and j never decrease in `MergeSortedParts`, so $i' \geq b$ and $j' \geq m + 1$. □

Now let's use Invariant 15.2 to prove Invariant 15.1. The proof goes by induction on the number of iterations of the loop.

Proof of Invariant 15.1. When the algorithm begins, that is, after zero iterations, we have $i = b$ and $j = m + 1$, so $B[b..i - 1]$ and $B[m + 1..j - 1]$ are empty arrays. Since $k = b$ at this point, $A[b..k - 1]$ is also the empty array, so $A[b..k - 1]$ definitely contains all elements of the empty arrays $B[b..i - 1]$ and $B[m + 1..j - 1]$. Since it contains no elements, it is also sorted, and all its elements are smaller than all elements of $B[i..m]$ and $B[m + 1..e]$. This proves the base case.

Now assume that the invariant holds after some iteration of the loop. We show that it holds after the next iteration too. We can assume $b \leq i \leq m + 1 \leq j \leq e + 1$ by Invariant 15.2. Furthermore, if $i = m + 1$ or $j = e + 1$, the loop terminates and there is no next iteration and thus nothing to prove.

So let's assume that $b \leq i \leq m$ and $m + 1 \leq j \leq e$. There are two cases to consider.

Case 1: $B[i] \leq B[j]$. In this case we put $B[i]$ into $A[k]$. The new values of i, j, and k are $i' = i + 1$, $j' = j$, and $k' = k + 1$, respectively. Note that $A[b..k - 1]$ is sorted by the induction hypothesis. Furthermore, all elements of $A[b..k - 1]$ are at most as large as any elements in $B[i..m]$, so $B[i]$ is at least as large as all elements of $A[b..k - 1]$. This means that after we place $B[i]$ into $A[k]$, the array $A[b..k] = A[b..k' - 1]$ is sorted.

Next, since $A[b..k-1]$ contains all elements of $B[b..i-1]$ and $B[m+1..j-1]$, and since $A[k] = B[i]$, it follows that $A[b..k' - 1]$ contains all elements of $A[b..i] = A[b..i' - 1]$ and $B[m + 1..j - 1] = B[m + 1..j' - 1]$.

Finally, all elements of $A[b..k - 1]$ are at most as large as all elements of $B[i..m]$ and $B[j..e]$, so they are at most as large as all elements of $B[i'..m]$ and $B[j'..e]$. Now $B[i]$ is at most as large as all elements of $B[i'..m]$ because $B[b..m]$ is sorted and $b \leq i \leq m$. Also, $B[i] \leq B[j]$, $m+1 \leq j \leq e$, and $B[m+1..e]$ is sorted, which means that $B[i]$ is at most as large as all elements of $B[j..e] = B[j'..e]$. Since $A[k] = B[i]$, this means all elements of $A[b..k] = A[b..k' - 1]$ are at most as large as all elements of $B[i'..m]$ and $B[j'..e]$. That proves the invariant is maintained after the next iteration of the loop in this case.

Case 2: $B[i] > B[j]$. The argument for this case is the same as for Case 1, except with the roles of i and j switched. □

15.2.2.2 Termination

To show that `MergeSortedParts` terminates, we need to show that the loop terminates after some number of iterations.

Consider the quantity $i + j$. When the program starts, $i + j = b + m + 1$. In each iteration of the loop, one of i or j increases by one and the other of the two variables doesn't change. Thus, $i + j$ increases by one in every iteration of the loop.

By Invariant 15.2, $i \leq m + 1$ and $j \leq e + 1$. After i increases $(m + 1) - b$ times, we have $i > m$ and the loop terminates. Similarly, after j increases $(e + 1) - (m + 1)$ times, $j > e$ and the loop terminates. Since one of i and j increases in every iteration of the loop, this means that the loop terminates after at most $(m + 1) - b + (e + 1) - (m + 1) = e - b + 1$ iterations. The algorithm terminates after the loop is over and after $B[i..m]$ (if nonempty) gets copied to the end of A.

15.2.3 Running Time Analysis

It should be fairly easy to see that the running time of `MergeSortedParts` is $\Theta(n)$ as we claimed earlier.

Let $n = e - b + 1$ be the length of the array $A[b..e]$.

On the first line we use n elementary operations to make a new copy of the array $A[b..e]$. Initializations on the next line cost us 3 elementary operations, which is negligible. Finally, there are at most n iterations of the loop because i ranges from b to m, j ranges from $m+1$ to e, and one of i and j increases in each iteration of the loop. Thus, after at most $m-b+e-(m+1) = e-b+1 = n$ iterations of the loop, the loop condition is violated and the loop terminates. Moreover, we perform 5 elementary operations in each iteration of the loop, so the entire loop takes $O(n)$ elementary operations. Finally, the last line copies at most n array elements, so it also costs us $O(n)$ elementary operations. Since each step in the algorithm takes $O(n)$ elementary operations, the complexity of `MergeSortedParts` is $O(n)$ as promised.

In fact, the complexity is $\Theta(n)$ because the very first line where we copy A into B takes n elementary operations.