Lecture 5: In-Place Sorting

Feb 3, 2017

Contents

1 In-Place Sorting

2 Swapping Elements in an Array

3 Bubble Sort
   3.1 Example, Unabridged
   3.2 Example, Optimized
   3.3 BubbleSort Complexity

4 Quicksort
   4.1 Partition, First Approach
   4.2 Partition, Second Approach

5 Proof of Correctness
   5.1 Partition, First Approach
   5.2 Partition, Second Approach

Objectives

By the end of these notes, you will know:

- how in-place sorting algorithms work

By the end of these notes, you will be able to:

- swap elements in an array
- sort an array in-place using bubble sort
- sort an array in-place using quicksort

1 In-Place Sorting

In our never-ending search for fast sorting algorithms in CS 17, you learned about insertion sort, selection sort, quicksort, and mergesort. We used those algorithms to sort lists—immutable lists, in
particular, so that the product of our sorting procedures was always a brand new list (in a brand new memory location), in sorted order.

Today, we are going to discuss in-place sorting of arrays. Rather than create a new array, we are going to sort the elements in an array by moving them around within the array’s allocated memory locations (and a tiny bit more space). We will demonstrate this idea using two algorithms, one new, bubble sort, and one old, quicksort. To get started, let’s explore the idea of swapping array elements.

2 Swapping Elements in an Array

Suppose we want to swap two elements in an array. Let’s first try swapping the data stored in two variables, say \( x \) and \( y \). As a first attempt, we might try doing something like this:

\[
\begin{align*}
    x &= y; \\
    y &= x;
\end{align*}
\]

But this is incorrect. Why? Because when \( x \) is set to the value of \( y \), the original value of \( x \) has been lost. Indeed, in the second statement \( y \) is set to its own value! For example, if \( x \) is 17 and \( y \) is 18, then setting \( x \) to \( y \) sets \( x \) to 18, but now setting \( y \) to \( x \) also sets \( y \) to 18, its own value!

To get around this problem, we introduce a temporary variable, as follows:

\[
\begin{align*}
    T \ tmp &= x; \\
    x &= y; \\
    y &= \tmp;
\end{align*}
\]

Here, \( T \) is an arbitrary type, but necessarily the type of both \( x \) and \( y \).

Now before overriding the value of \( x \), its value is stored in \( \tmp \). Then \( x \) is set to to \( y \), as above, but now \( y \) is set to \( \tmp \), instead of \( x \), which actually sets \( y \) to the original value of \( x \).

Here is the same idea applied to an array \( a \) of type \( T \):

\[
\begin{align*}
    T \ tmp &= a[i]; \\
    a[i] &= a[j]; \\
    a[j] &= \tmp;
\end{align*}
\]

Swapping array elements is such a common activity that it often makes sense to have on hand a swap helper function like this one:

```java
public static void swap(int[] a, int i, int j) {
    int tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
}
```
This function might look surprising to you. Why is the return type \texttt{void}? Shouldn’t we be returning the modified array? After all, we just went through all that trouble modifying it!

The answer to this conundrum is that array variables actually store memory addresses. Consequently, when an array is passed to a function as an argument, the value that is passed is a memory address. So if an element of that array is modified, the modification persists! That means that the change is not only local to the callee function (e.g., \texttt{swap}), which is natural; but, in addition, the change is seen in the caller function (e.g., a sort function), which sends the array as an argument.

It is in precisely this way that an in-place sorting algorithm operates. It makes repeated calls to a \texttt{swap} function that re-orders adjacent elements of an array, and those changes persist in the sort function itself.

### 3 Bubble Sort

Bubble sort is rarely used in practice, but it is a classic sorting algorithm, so if no other reason than historical, it is useful to be familiar with it. It is a simple idea really: it works by repeatedly making passes through the array to be sorted, comparing adjacent elements, and swapping them (in place) if they are not already in sorted order.

#### Notation:

Throughout this discussion, we let \( A \) denote an array of length \( n \); further, we let \( A[0, \ldots, i] \) denote the first \( i + 1 \) elements of \( A \), and \( A[i + 1, \ldots, n - 1] \) denote the last \( n - (i + 1) \) elements of \( A \). For example, if \( A \) is the array \( \{7, 1, 5, 9, 3\} \) of length \( n = 5 \), and if \( i = 2 \), then \( A[0, \ldots, i] \) is the subarray \( \{7, 1, 5\} \), while \( A[i + 1, \ldots, n - 1] \) is the subarray \( \{9, 3\} \).

#### 3.1 Example, Unabridged

Here is how bubble sort—the unabridged version—sorts the array \( A = \{7, 1, 5, 9, 3\} \).

<table>
<thead>
<tr>
<th>Initial Array</th>
<th>7</th>
<th>1</th>
<th>5</th>
<th>9</th>
<th>3</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>1</th>
<th>5</th>
<th>9</th>
<th>3</th>
<th>—</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step 1</strong></td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>9</td>
<td>3</td>
<td><strong>swap 7 &amp; 1</strong></td>
</tr>
<tr>
<td><strong>Step 2</strong></td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td><strong>swap 7 &amp; 5</strong></td>
</tr>
<tr>
<td><strong>Step 3</strong></td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td><strong>don’t swap</strong></td>
</tr>
<tr>
<td><strong>Step 4</strong></td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>3</td>
<td>9</td>
<td><strong>swap 9 &amp; 3</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>5</th>
<th>7</th>
<th>3</th>
<th>9</th>
<th>—</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step 1</strong></td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>3</td>
<td>9</td>
<td><strong>don’t swap</strong></td>
</tr>
<tr>
<td><strong>Step 2</strong></td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>3</td>
<td>9</td>
<td><strong>don’t swap</strong></td>
</tr>
<tr>
<td><strong>Step 3</strong></td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td><strong>swap 7 &amp; 3</strong></td>
</tr>
<tr>
<td><strong>Step 4</strong></td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td><strong>don’t swap</strong></td>
</tr>
</tbody>
</table>
### Third Pass

<table>
<thead>
<tr>
<th>Initial</th>
<th>1</th>
<th>5</th>
<th>3</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>don't swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 2</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>swap 5 &amp; 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 3</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>don't swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 4</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>don't swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Fourth Pass

<table>
<thead>
<tr>
<th>Initial</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>don't swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 2</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>don't swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 3</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>don't swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 4</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>don't swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Final Array**

| 1 | 3 | 5 | 7 | 9 |

Bubble sort consists of not just one, but two, loops: an outer loop, which performs “passes;” and an inner loop, which “steps” through the array and swaps adjacent elements as necessary.

In this example (and in the worst case, which we discuss later), the number of passes is $n - 1$, one less than the number of elements in the array. In this unabridged version, the number of steps is also $n - 1$, because there are $n - 1$ pairs of adjacent elements in an array of length $n$.

The contents of the array shown at each step $i$ are the contents after each step $i$. The elements shown in red are those that were considered for a swap. (You can tell whether or not they were actually swapped by looking at their order in the previous step.)

The elements shown in blue are elements that are sorted. We note that after $i$th pass, the last $i$ elements of the array are sorted.

### 3.2 Example, Optimized

It is not actually necessary to carry out all the steps shown in the unabridged version of the algorithm. On the contrary, during pass $i$, it is only necessary to consider swaps among the first $n - i$ elements. Why? Because the last $i$ elements are already sorted! For example, during the second pass, it is only necessary to consider swaps among the first four elements; the lone last element is already sorted! This observation motivates the following abridged version of bubble sort:

<table>
<thead>
<tr>
<th>Initial Array</th>
<th>7</th>
<th>1</th>
<th>5</th>
<th>9</th>
<th>3</th>
</tr>
</thead>
</table>

**First Pass**

<table>
<thead>
<tr>
<th>Initial</th>
<th>7</th>
<th>1</th>
<th>5</th>
<th>9</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>swap 7 &amp; 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 2</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>swap 7 &amp; 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 3</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>don’t swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 4</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>swap 9 &amp; 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


**Exercise:** Are there further ways to optimize bubble sort?

### 3.3 BubbleSort Complexity

In your homework this week, you will convince yourselves (formally) that bubble sort works. Here, we will explore its run time.

What does a bad case look like for bubble sort? Is an array that is already sorted a bad case for bubble sort? No, it is not. What about an array in reverse sorted order? Yes, this is bad, but why exactly? Which are the problematic entries in the array? Are some worse than others? The answer to this question is yes. But which?

In the unabridged version, bubble sort makes \( n \) passes, and each pass is \( O(n) \): i.e., linear in the length of the array, since it steps through the entire array. The run time is quadratic:

\[
\sum_{i=1}^{n} n - 1 = n(n - 1) \in \Theta(n^2)
\]

In the abridged version, bubble sort makes \( n \) passes, each of which is only \( O(n - i) \): i.e., linear in the number of unsorted entries in the array. The run time is again quadratic:

\[
\sum_{i=1}^{n} n - i = \sum_{i=1}^{n} i = \frac{n(n + 1)}{2} \in \Theta(n^2)
\]

Although its run time is quadratic, bubble sort does have one advantage over other, faster sorting algorithms: it can detect when an array is sorted, and terminate early. Whereas an obvious way to implement bubble sort might be as two nested *for* loops, a smarter alternative is to nest a *for* loop inside a *while* loop, and to terminate the *while* loop as soon as the array is sorted (i.e., as soon as there are no further swaps).
4 Quicksort

Quicksort is the perhaps the sorting algorithm most widely used in practice. However, it is not the implementation of quicksort that you learned in CS 17 that is most widely used. Rather, it is an in-place version, which we will discuss presently.

Recall the quicksort algorithm, as applied to a sequence: e.g. a list or an array.

1. **Base Case**: If the sequence is size 1, return. (It’s sorted!)

2. **Pick a pivot**: Choose a pivot element. (Any element in the sequence will do, but some might be preferable to others.)

3. **Partition**: Use the pivot to split up the rest of the sequence into two smaller sequences, one with values less than the pivot and the other with values greater than or equal to the pivot.\(^1\)

4. **Recur**: Recur on the two smaller sequences, then return the concatenation of the (now sorted) first sequence, the pivot, and the (now sorted) second sequence.

Today we will discuss an implementation of this very same algorithm, but using mutable arrays rather than immutable lists. That is, we won’t create a new array with each recursive call; instead we will continually modify the same array, until it is sorted.

Here is an example of quicksorting an array, in-place:

- **Initial Array**:

  \[
  \begin{array}{ccccccc}
  33 & 157 & 18 & 155 & 32 & 17 & 51 \\
  \end{array}
  \]

- **Step 1**: Choose 51 as the pivot element.

- **Step 2**: Move the elements less than 51 to the left part of the array, and move the elements greater than or equal to 51 to the right part of the array:

  \[
  \begin{array}{cccccccc}
  33 & 18 & 32 & 17 & 155 & 157 & 51 \\
  L & L & L & L & R & R & pivot \\
  \end{array}
  \]

- **Step 3**: Put the pivot element in between:

  \[
  \begin{array}{cccccccc}
  33 & 18 & 32 & 17 & 51 & 157 & 155 \\
  L & L & L & L & pivot & R & R \\
  \end{array}
  \]

  and then recursively sort the left and right parts of the array:

  \[
  \begin{array}{cccccccc}
  17 & 18 & 32 & 33 & 51 & 155 & 157 \\
  L & L & L & L & pivot & R & R \\
  \end{array}
  \]

- **Final (Sorted) Array**:

  \[
  \begin{array}{ccccccc}
  17 & 18 & 32 & 33 & 51 & 155 & 157 \\
  \end{array}
  \]

\(^1\)Alternatively, you could split up the rest of the sequence into two, with values less than or equal the pivot in one, and values greater than the pivot in the other.
Interestingly, when we sort an array in place (and, more generally, when you do any sort of in-place operation), we need not return anything. The effect of quicksorting an array in place is not to produce a new sorted array; rather, it is to modify the given unsorted array.

The most interesting part of in-place quicksort implementation is the partitioning scheme. How would we go about moving all the elements less than the pivot to the left part of the array, and all the elements greater than or equal to the pivot to the right part of the array, in place (and efficiently)?

### 4.1 Partition, First Approach

Use two indices, `split` and `current`, both initialized to index the first element of the array. Traverse the array (sans the pivot element), by incrementing the `current` index, while maintaining the following properties:

- All data stored at indices less than `split` have values less than the pivot value.
- All data stored at indices greater than or equal to `split` and less than `current` have values greater than or equal to the pivot value.

Here is a simple iterative algorithm that achieves this:

1. If \(a[current] < pivot\), then swap \(a[current]\) and \(a[split]\), and increment `split`.
2. Increment `current`.
3. Repeat until `current` indexes beyond the last element of the array, at which point the entire array has been processed.

For example, consider the following array, in which the pivot is 33. Initially, `split` and `current` are 0, meaning they both index 51.

<table>
<thead>
<tr>
<th>51</th>
<th>17</th>
<th>155</th>
<th>18</th>
<th>157</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>split</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>current</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Since 51 > 33, increment `current`, but not `split`.

<table>
<thead>
<tr>
<th>51</th>
<th>17</th>
<th>155</th>
<th>18</th>
<th>157</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>split</td>
<td>current</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Next, since 17 < 33, swap 17 and 51, and this time increment both `split` and `current`.

<table>
<thead>
<tr>
<th>17</th>
<th>51</th>
<th>155</th>
<th>18</th>
<th>157</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>split</td>
<td>current</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Next, 155 > 33, so increment only `current`.

<table>
<thead>
<tr>
<th>17</th>
<th>51</th>
<th>155</th>
<th>18</th>
<th>157</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>split</td>
<td>current</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

But 18 < 33, so swap 18 and 51 (and increment both indices).
Now, \(157 > 33\), so increment only \(\text{current}\).

But \(32 < 33\), so swap \(32\) and \(155\) (and increment both indices).

Finally, \(\text{current}\) has traversed the entire array, so the entire array has been processed. Observe: the values to the left of \(\text{split}\) are less than the pivot, and the values to the right of \(\text{split}\) are greater than or equal to the pivot.

### 4.2 Partition, Second Approach

Use two indices: \(\text{left}\), which is initialized to index the first element of the array, and \(\text{right}\), which is initialized to index the last element of the array. The array is then traversed \((\text{sans} \text{ the pivot element})\) by moving the left index to the right and the right index to the left, while maintaining the following properties:

- All data stored at indices less than \(\text{left}\) have values less than the pivot.
- All data stored at indices greater than or equal to \(\text{right}\) have values greater than or equal to the pivot.

Here is a simple iterative algorithm that does this:

1. Increment \(\text{left}\) if it indexes a cell whose value is less than the pivot value, otherwise leave \(\text{left}\) in place.

2. Decrement \(\text{right}\) if it indexes a cell whose value is greater than or equal to the pivot, otherwise leave \(\text{right}\) in place.

3. If \(a[\text{left}] \geq \text{pivot} > a[\text{right}]\), then swap \(a[\text{left}]\) and \(a[\text{right}]\), and then increment \(\text{left}\) and decrement \(\text{right}\).

4. Repeat until \(\text{left} > \text{right}\), at which point the entire array has been processed.

For example, consider the following array in which the pivot is 33. Initially, \(\text{left}\) indexes 51 and \(\text{right}\) indexes 32.

<table>
<thead>
<tr>
<th>51</th>
<th>31</th>
<th>155</th>
<th>18</th>
<th>181</th>
<th>157</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{left})</td>
<td>(\text{right})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Because \(51 \geq 33 > 32\), swap 51 and 32, then increment \(\text{left}\) and decrement \(\text{right}\).
Now, since $31 < 33$, increment left, and since $157 \geq 33$, decrement right.

<table>
<thead>
<tr>
<th>32</th>
<th>31</th>
<th>155</th>
<th>18</th>
<th>181</th>
<th>157</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>right</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Since $155 > 33$, left cannot be further incremented, unless there is first a swap. But right can be decremented, since $181 \geq 33$.

<table>
<thead>
<tr>
<th>32</th>
<th>31</th>
<th>155</th>
<th>18</th>
<th>181</th>
<th>157</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>right</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

And now there should be a swap, since $155 \geq 33 > 18$. After swapping, increment left ($18 < 33$) and decrement right ($155 > 33$).

<table>
<thead>
<tr>
<th>32</th>
<th>31</th>
<th>18</th>
<th>155</th>
<th>181</th>
<th>157</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>left</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

At this point, left exceeds right, meaning the entire array has been processed. Observe: the values to the left of left are less than the pivot, while values to the right of right are greater than or equal to the pivot.

## 5 Proof of Correctness

Now that we understand how to implement a partition function for in-place quicksort, it’s time to prove that our approaches correct.

### 5.1 Partition, First Approach

**First Loop Invariant** Recall the first loop invariant:

- All data stored at indices less than split have values less than the pivot value.

We restate this invariant using mathematical notation as follows:

**Claim:** For all values of current, for all $j \in \{start, \ldots, split−1\}$, $arr[j] < pivot$.

**Proof:** The proof is by induction on current.

**Basis:** The value of split is initialized to 10, so before the loop is entered, this invariant (for all $j \in \{start, \ldots, split−1\}$, $arr[j] < pivot$) is vacuously true.

**Step:** Assume the induction hypothesis: after iteration $current−1$, for all $j \in \{start, \ldots, split−1\}$, $arr[j] < pivot$.

Two cases arise.

**Case 1:** $arr[current] \geq pivot$. In this case, the only thing that happens inside the loop is that current is incremented.

As the value of split is unchanged, and likewise so are the contents of arr, the claim follows immediately from the induction hypothesis.

**Case 2:** $arr[current] < pivot$. In this case, $arr[current]$ and $arr[split]$ are swapped, and then the values of both split and current are incremented.
We know by the induction hypothesis that for all \( j \in \{\text{start}, \ldots, \text{split} - 1\} \), \( \text{arr}[j] < \text{pivot} \). After \( \text{arr}[\text{current}] \) and \( \text{arr}[\text{split}] \) are swapped, \( \text{arr}[\text{split}] \) is also less than \( \text{pivot} \), since \( \text{arr}[\text{current}] \) was assumed to be less than \( \text{pivot} \). So, after \( \text{split} \) is incremented, once again for all \( j \in \{\text{start}, \ldots, \text{split} - 1\} \), \( \text{arr}[j] < \text{pivot} \).

**Second loop Invariant**  
Recall the second loop invariant:

- All data stored at indices greater than or equal to \( \text{split} \) and less than \( \text{current} \) have values greater than or equal to the pivot value.

We restate this invariant using mathematical notation as follows:

**Claim:** For all values of \( \text{current} \), \( j \in \{\text{split}, \ldots, \text{current} - 1\} \), \( \text{arr}[j] \geq \text{pivot} \).

**Proof:** The proof is by induction on \( \text{current} \).

**Basis:** The value of \( \text{current} \) is initialized to \( \text{split} \), so before the loop is entered, this invariant (for all \( j \in \{\text{split}, \ldots, \text{split} - 1\} \), \( \text{arr}[j] \geq \text{pivot} \)) is vacuously true.

**Step:** Assume the induction hypothesis: after iteration \( \text{current} - 1 \), for all \( j \in \{\text{split}, \ldots, \text{current} - 1\} \), \( \text{arr}[j] \geq \text{pivot} \).

Two cases arise:

**Case 1:** \( \text{arr}[\text{current}] \geq \text{pivot} \). In this case, the only thing that happens inside the loop is that \( \text{current} \) is incremented.

As the contents of \( \text{arr} \) are unchanged, and \( \text{arr}[\text{current}] \) is already greater than or equal to \( \text{pivot} \), the claim follows immediately.

**Case 2:** \( \text{arr}[\text{current}] < \text{pivot} \). In this case, \( \text{arr}[\text{current}] \) and \( \text{arr}[\text{split}] \) are swapped, and then the values of \( \text{split} \) and \( \text{current} \) are both incremented.

Two further cases arise based on the relationship between the values of \( \text{split} \) and \( \text{current} \). Note that the value of \( \text{split} \) can never exceed that of \( \text{current} \). So it is always the case that \( \text{split} \leq \text{current} \).

Now:

**Case 2a:** \( \text{split} = \text{current} \). The invariant (for all \( j \in \{\text{split}, \ldots, \text{split} - 1\} \), \( \text{arr}[j] \geq \text{pivot} \)) is vacuously true.

**Case 2b:** \( \text{split} < \text{current} \). We know by the induction hypothesis that for all \( j \in \{\text{split}, \ldots, \text{current} - 1\} \), \( \text{arr}[j] \geq \text{pivot} \). After \( \text{arr}[\text{current}] \) and \( \text{arr}[\text{split}] \) are swapped, \( \text{arr}[\text{current}] \) is also greater than or equal to \( \text{pivot} \), since \( \text{arr}[\text{split}] \) was assumed to be greater than or equal to \( \text{pivot} \) (by the induction hypothesis). However, \( \text{arr}[\text{split}] \) is less than \( \text{pivot} \), since \( \text{arr}[\text{current}] \) was assumed to be less than \( \text{pivot} \). But not to worry ... after \( \text{split} \) and \( \text{current} \) are incremented, once again for all \( j \in \{\text{split}, \ldots, \text{current} - 1\} \), \( \text{arr}[j] \geq \text{pivot} \). \( \diamond \)

**Termination**  
The first partitioning scheme is guaranteed to terminate because either \( \text{current} \) is initialized to be greater than or equal to \( \text{arr}.\text{length} \), and the loop is never entered; or the loop is entered, in which case \( \text{current} \) is incremented unconditionally, so its value will necessarily reach that of \( \text{arr}.\text{length} \).

Furthermore, upon termination, two invariants hold:
For all \( j \in \{\text{start}, \ldots, \text{split} - 1\} \), \( \text{arr}[j] < \text{pivot} \).

For all \( j \in \{\text{split}, \ldots, \text{arr.length} - 1\} \), \( \text{arr}[j] \geq \text{pivot} \).

Therefore, once the pivot is swapped with \( \text{arr}[\text{split}] \), the array has been partitioned around the pivot value. ⊓

### 5.2 Partition, Second Approach

#### Left Loop Invariant

Recall the left loop invariant:

- All data stored at indices less than \( \text{left} \) have values less than the pivot value.

We restate this invariant using mathematical notation as follows:

**Claim:** After iteration \( i \), for all \( j \in \{\text{start}, \ldots, \text{left} - 1\} \), \( \text{arr}[j] < \text{pivot} \).

**Proof:** The proof is by induction on \( i \).

**Basis:** The value of \( \text{left} \) is initialized to 10, so before the loop is entered, this invariant (for all \( j \in \{\text{start}, \ldots, \text{start} - 1\} \), \( \text{arr}[j] < \text{pivot} \)) is vacuously true.

**Step:** Assume the induction hypothesis: after iteration \( i - 1 \), for all \( j \in \{\text{start}, \ldots, \text{left} - 1\} \), \( \text{arr}[j] < \text{pivot} \).

There are three cases.

**Case 1:** \( \text{arr}[\text{left}] < \text{pivot} \). In this case, the value of \( \text{left} \) is incremented.

We know by the induction hypothesis that for all \( j \in \{\text{start}, \ldots, \text{left} - 1\} \), \( \text{arr}[j] < \text{pivot} \). Appending to this list \( \text{left} \), for which it is assumed that \( \text{arr}[\text{left}] < \text{pivot} \), by incrementing the value of \( \text{left} \) yields for all \( j \in \{\text{start}, \ldots, \text{left} - 1\} \), \( \text{arr}[j] < \text{pivot} \).

**Case 2:** \( \text{arr}[\text{right}] \geq \text{pivot} \) and \( \text{arr}[\text{left}] \geq \text{pivot} \). In this case, only the value of \( \text{right} \) is decremented.

The value of \( \text{left} \) is unchanged, as are the contents of \( \text{arr} \). Hence, the claim follows immediately from the induction hypothesis.

**Case 3:** \( \text{arr}[\text{left}] \geq \text{pivot} \) and \( \text{pivot} > \text{arr}[\text{right}] \). In this case, the values of \( \text{arr}[\text{left}] \) and \( \text{arr}[\text{right}] \) are swapped, and then the value of \( \text{left} \) is incremented and the value of \( \text{right} \) is decremented.

We know by the induction hypothesis that for all \( j \in \{\text{start}, \ldots, \text{left} - 1\} \), \( \text{arr}[j] < \text{pivot} \). After \( \text{arr}[\text{left}] \) and \( \text{arr}[\text{right}] \) are swapped, \( \text{arr}[\text{left}] \) is also less than \( \text{pivot} \), since \( \text{arr}[\text{right}] \) was assumed to be less than \( \text{pivot} \). So, after \( \text{left} \) is incremented, once again for all \( j \in \{\text{start}, \ldots, \text{left} - 1\} \), \( \text{arr}[j] < \text{pivot} \). ⊓

#### Right Loop Invariant

Recall the right loop invariant:

- All data stored at indices greater than \( \text{right} \) have values greater than or equal to the pivot value.
We restate this invariant using mathematical notation as follows:

**Claim:** After iteration $i$, for all $j \in \{right + 1, \ldots, end\}$, $\text{arr}[j] \geq \text{pivot}$.

**Proof:** The proof is by induction on $i$ (which is represented by the variable current in our code).

**Basis:** The value of right is initialized to end, so before the loop is entered, this invariant (for all $j \in \{right + 1, \ldots, end\}$, $\text{arr}[j] \geq \text{pivot}$) is vacuously true.

**Step:** Assume the induction hypothesis: after iteration $i - 1$, for all $j \in \{right + 1, \ldots, end\}$, $\text{arr}[j] \geq \text{pivot}$.

There are three cases.

**Case 1:** $\text{arr}[\text{left}] < \text{pivot}$. In this case, only the value of left is incremented.

The value of right is unchanged, as are the contents of arr. Hence, the claim follows immediately from the induction hypothesis.

**Case 2:** $\text{arr}[\text{right}] \geq \text{pivot}$ and $\text{arr}[\text{left}] \geq \text{pivot}$. In this case, the value of right is decremented.

We know by the induction hypothesis that for all $j \in \{right + 1, \ldots, end\}$, $\text{arr}[j] \geq \text{pivot}$. Prepending to this list right, for which it is assumed that $\text{arr}[\text{right}] \geq \text{pivot}$, by decrementing the value of right yields for all $j \in \{right + 1, \ldots, end\}$, $\text{arr}[j] \geq \text{pivot}$.

**Case 3:** $\text{arr}[\text{left}] \geq \text{pivot}$ and $\text{pivot} > \text{arr}[\text{right}]$. In this case, the values of arr[left] and arr[right] are swapped, and then the value of left is incremented and the value of right is decremented.

We know by the induction hypothesis that for all $j \in \{right + 1, \ldots, end\}$, $\text{arr}[j] < \text{pivot}$. After arr[left] and arr[right] are swapped, arr[right] is also greater than or equal to pivot, since arr[left] was assumed to be greater than or equal to pivot. So, after right is decremented, once again for all $j \in \{right + 1, \ldots, end\}$, $\text{arr}[j] < \text{pivot}$.

**Termination** The second partitioning scheme is guaranteed to terminate. If left is initialized to a value that is greater than right, the loop is never entered, so the function terminates trivially.

Otherwise, left <= right, and the loop is entered. Then, one of three mutually exclusive cases is encountered. In the first case left is incremented; in the second, right is decremented; and in the third, left is incremented and the right is decremented. In other words, during every iteration of the loop, progress is made towards to termination condition, in which left > right.

Furthermore, upon termination, two invariants hold:

- For all $j \in \{start, \ldots, right\}$, $\text{arr}[j] < \text{pivot}$.

- For all $j \in \{left, \ldots, end\}$, $\text{arr}[j] \geq \text{pivot}$.

Therefore, once the pivot is swapped with arr[right], the array has been partitioned around the pivot value.
Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form:

http://cs.brown.edu/courses/cs018/feedback