Lecture 11: In-Place Sorting and Loop Invariants
10:00 AM, Feb 16, 2018

Contents

1 In-Place Sorting 1

2 Swapping Elements in an Array 1

3 Bubble Sort 2

3.1 Example, Unabridged 2

3.2 Example, Optimized 3

3.3 Proofs of Correctness 4

3.3.1 Proving the Invariants 6

3.4 BubbleSort Complexity 7

4 Quicksort 7

4.1 Partitioning 9

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:

```java
x = y;
y = x;
```

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:

```java
T tmp = x;
x = y;
y = tmp;
```

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 \( T \):

```java
T tmp = a[i];
a[i] = a[j];
a[j] = tmp;
```

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;
}
```

Note that swap can return \( void \) because arrays are objects in the heap. If we change the contents inside an array on the heap, they will be visible through existing known names for the object (we’ve seen this before; we’re just reiterating it here).

In-place sorting algorithms rely heavily on swap operations, as we will see in the following two examples.
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>
<tbody>
<tr>
<td><strong>First Pass</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Step 1</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>9</td>
<td>3</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>Step 3</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Step 4</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>3</td>
<td>9</td>
</tr>
</tbody>
</table>

| **Second Pass** |   |   |   |   |   |
| Initial        | 1 | 5 | 7 | 3 | 9 |
| Step 1        | 1 | 5 | 7 | 3 | 9 | don’t swap |
| Step 2        | 1 | 5 | 7 | 3 | 9 | don’t swap |
| Step 3        | 1 | 5 | 3 | 7 | 9 | swap 7 & 3 |
| Step 4        | 1 | 5 | 3 | 7 | 9 | don’t swap |

| **Third Pass** |   |   |   |   |   |
| Initial        | 1 | 5 | 3 | 7 | 9 |
| Step 1        | 1 | 5 | 3 | 7 | 9 | don’t swap |
| Step 2        | 1 | 3 | 5 | 7 | 9 | swap 5 & 3 |
| Step 3        | 1 | 3 | 5 | 7 | 9 | don’t swap |
| Step 4        | 1 | 3 | 5 | 7 | 9 | don’t swap |

| **Fourth Pass** |   |   |   |   |   |
| Initial        | 1 | 3 | 5 | 7 | 9 |
| Step 1        | 1 | 3 | 5 | 7 | 9 | don’t swap |
| Step 2        | 1 | 3 | 5 | 7 | 9 | don’t swap |
| Step 3        | 1 | 3 | 5 | 7 | 9 | don’t swap |
| Step 4        | 1 | 3 | 5 | 7 | 9 | don’t swap |

<table>
<thead>
<tr>
<th><strong>Final Array</strong></th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
</table>
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>
<tbody>
<tr>
<td><strong>First Pass</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Step 1</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Swap 7 &amp; 1</td>
<td></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>Swap 7 &amp; 5</td>
<td></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>Don’t swap</td>
<td></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>Swap 9 &amp; 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| **Second Pass** |   |   |   |   |   |
| Initial         | 1 | 5 | 7 | 3 | 9 |
| Step 1          | 1 | 5 | 7 | 3 | 9 |
| Don’t swap      |   |   |   |   |   |
| Step 2          | 1 | 5 | 7 | 3 | 9 |
| Don’t swap      |   |   |   |   |   |
| Step 3          | 1 | 5 | 3 | 7 | 9 |
| Swap 7 & 3      |   |   |   |   |   |

| **Third Pass** |   |   |   |   |   |
| Initial         | 1 | 5 | 3 | 7 | 9 |
| Step 1          | 1 | 5 | 3 | 7 | 9 |
| Don’t swap      |   |   |   |   |   |
| Step 2          | 1 | 3 | 5 | 7 | 9 |
| Swap 5 & 3      |   |   |   |   |   |

| **Fourth Pass** |   |   |   |   |   |
| Initial         | 1 | 3 | 5 | 7 | 9 |
| Step 1          | 1 | 3 | 5 | 7 | 9 |
| Don’t swap      |   |   |   |   |   |

| **Final Array** | 1 | 3 | 5 | 7 | 9 |
3.3 Proofs of Correctness

How might we prove that this algorithm is correct? Let’s first write the algorithm out formally in code. Here, we put the code inside a class for sorting arrays. Notice that the `ArraySort` class contains the `swap` method, but we have made that method private since it is only meant to be used within the `ArraySort` class.

```java
public class ArraySort {

    /**
     * swaps the values of two positions in an array
     * @param theArray -- the array in which to swap values
     * @param pos1 -- the position of the value to swap with pos2
     * @param pos2 -- the position of the value to swap with pos1
     */
    private static void swap(int[] theArray, int pos1, int pos2) {
        int temp = theArray[pos1];
        theArray[pos1] = theArray[pos2];
        theArray[pos2] = temp;
    }

    /**
     * sorts an array into ascending order, using the bubble sort algorithm
     * @param arr -- the array to sort
     */
    private static void bubbleSort(int[] arr) {
        int outerPass;
        int innerStep;

        for(outerPass = 0; outerPass < arr.length - 1; outerPass++) {
            for(innerStep = 0; innerStep < arr.length - 1 - outerPass; innerStep++) {
                if (arr[innerStep] > arr[innerStep + 1]) {
                    swap(arr, innerStep, innerStep + 1);
                }
            }
        }
    }

    public static void main(String[] args) {
        int[] A2 = new int[] {5, 3, 8, 2, 9, 6};
        System.out.println("Array before is " + Arrays.toString(A2));
        bubbleSort(A2);
        System.out.println("Array after is " + Arrays.toString(A2));
    }
}
```

We have an informal idea of why the algorithm is correct based on our earlier discussion about how the larger elements push rightward (this motivated the optimized algorithm). How could we make that more rigorous?

One way to make argue correctness rigorously is to annotate the code with `assertions` about the contents of data structures or the heap that should hold whenever execution reaches that assertion.
These assertions are typically called *invariants* (because we want them to be true invariably at a particular execution point). To prove correctness, we need to (a) argue that our invariants taken together will prove the correctness of the algorithm, and (b) argue that the invariants hold each time execution passes over the invariant.

Invariants are typically associated with bits of code that can get called multiple times, such as bodies of loops or functions. Here, we will set up one invariant for each `for` loop:

```java
private static void bubbleSort(int[] arr) {
    int outerPass;
    int innerStep;

    for(outerPass = 0; outerPass < arr.length - 1; outerPass++) {
        // INVARIANT #1: arr[len-1-outerPass, ..., len-1] is sorted
        for(innerStep = 0; innerStep < arr.length - 1 - outerPass; innerStep++) {
            if (arr[innerStep] > arr[innerStep + 1]) {
                swap(arr, innerStep, innerStep + 1);
            }
        // INVARIANT #2: all elts in arr[0..innerStep] < arr[innerStep+1]
    }
}
```

Invariant #1 says that the back portion of the array is sorted, while Invariant #2 says that all of the elements in the front portion of the array are smaller than those in the sorted portion.

### 3.3.1 Proving the Invariants

How can we prove that the inner-loop invariant is correct? We argue this by (a) proving it holds the first time execution reaches the invariant point, then (b) proving that if it holds anytime it reaches the invariant point, then it must also hold the *next* time execution reaches that point.

Let’s write this out in detail:

- The first time through the inner loop, `innerStep` is 0 (as is `outerPass`), so Invariant #2 is trivially true

- Assume we’re in a later pass through the inner loop, so Invariant #2 is true for all elts up `innerStep-1`. Specifically:

  \[
  arr[0..innerPass - 1] < arr[innerPass]
  \]

  inside the loop, if \( arr[innerStep] < arr[innerStep + 1] \), then by transitivity

  \[
  arr[0..innerPass - 1] < arr[innerPass]
  \]

  If that weren’t true, we swapped the elts. `innerStep` was already larger than earlier elts, and is not larger than old `innerPass+1`, so Invariant #2 still holds
How about the proof for the outer loop invariant (#1)?

- The first time we enter the loop, outerPass is 0. Invariant #1 asks whether arr[len-1,...,len-1] is sorted. Since a single-element array is sorted, Invariant #1 holds.
- On subsequent passes through the outer loop, we can assume that Invariant #1 held for outerPass-1. We need to prove that

\[
\text{arr}[\text{length} - 1 - \text{outerPass}, ..., \text{length} - 1]
\]

is sorted. Since Invariant #1 held for the previous value of outerPass, we know that

\[
\text{arr}[\text{length} - 1 - \text{outerPass} - 1, ..., \text{length} - 1]
\]

is sorted. To prove that including \(\text{arr}[\text{length} - 1 - \text{outerPass}]\) retains a sorted array, we simply need to prove that the final value in \(\text{arr}[\text{outerPass}]\) is no larger than the elements in \(\text{arr}[\text{length} - 1 - \text{outerPass} - 1, ..., \text{length} - 1]\). The value in \(\text{arr}[\text{outerPass}]\) came from \(\text{arr}[0..\text{arr.length} - 1 - \text{outerPass} - 1]\) in the previous iteration, so by Invariant #2, it was smaller than the elements in the back portion of the array. Thus, the outer invariant still holds.

Do the invariants prove that Bubble Sort yields an array with all of the elements in ascending order?

By the time we finish executing the outer for loop for the last time, outerPass is one shorter than the length of the array; according to Invariant #1, this means that all but the first element are properly sorted. The inner for loop will also have finished, so by Invariant #2, the item in the first position in the array is smaller than the one in the second position. Combining these two statements proves that the entire array is indeed sorted.

### 3.4 BubbleSort Complexity

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.
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:
  
  | 33 | 157 | 18 | 155 | 32 | 17 | 51 |

- 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:

  | 33 | 18 | 32 | 17 | 155 | 157 | 51 |
  | R | R | R | R | L | L | L |

- Step 3: Put the pivot element in between:

  | 33 | 18 | 32 | 17 | 51 | 157 | 155 |
  | L | L | L | L | pivot | R | R |

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

  | 17 | 18 | 32 | 33 | 51 | 155 | 157 |
  | L | L | L | L | pivot | R | R |

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.
Final (Sorted) Array:

| 17 | 18 | 32 | 33 | 51 | 155 | 157 |

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)? There are a couple of approaches; we will show one of them.

### 4.1 Partitioning

Use two indices: left, which is initialized to index the first element of the array, and right, which is initialized to index the last element of the array. The array is then traversed (sans 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 left have values less than the pivot
- All data stored at indices greater than or equal to right have values greater than or equal to the pivot.

Here is a simple iterative algorithm that does this:

1. Increment left if it indexes a cell whose value is less than the pivot value, otherwise leave left in place.
2. Decrement right if it indexes a cell whose value is greater than or equal to the pivot, otherwise leave right in place.
3. If a[left] ≥ pivot > a[right], then swap a[left] and a[right], and then increment left and decrement right.
4. Repeat until left > right, at which point the entire array has been processed.

For example, consider the following array in which the pivot is 33. Initially, left indexes 51 and 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>left</td>
<td>right</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Because 51 ≥ 33 > 32, swap 51 and 32, then increment left and 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>

Now, since 31 < 33, increment left, and since 157 ≥ 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 $181 \geq 33$.

\[
\begin{array}{ccccccc}
32 & 31 & 155 & 18 & 181 & 157 & 51 \\
\hline
\text{left} & \text{right}
\end{array}
\]

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

\[
\begin{array}{ccccccc}
32 & 31 & 18 & 155 & 181 & 157 & 51 \\
\hline
\text{right} & \text{left}
\end{array}
\]

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.

As we did with bubble sort, we could write out the code, annotate it with invariants, and argue that executing the code preserves the invariants. The code is in the source distribution for today’s lecture. We leave the invariants and proof as an exercise for those who want to give this work a try.

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](http://cs.brown.edu/courses/cs018/feedback)