

Insertion Sort

Lecturer: Arran Stewart

Outline

- ▶ What is insertion sort?
- ▶ How does it work?
- ▶ What is the algorithmic complexity of insertion sort?

Insertion sort

- ▶ *Insertion sort* is an algorithm which takes a list of numbers and sorts them.
 - ▶ It can be used to sort things other than numbers, too – but we will stick to numbers for the moment.
- ▶ Recall that an *algorithm* is a step-by-step procedure for accomplishing a task.

The basic idea

- ▶ Imagine that you have a pile of cards on your desk, each with a number, that are in no particular order.
- ▶ If you wanted to sort them, one way you might do it is this:
 - ▶ Pick a card out of the pile, and add it to a *new* pile (which we'll call the "sorted pile").
 - ▶ Now pick a second card out of the pile, and put it in the correct order with the one card you've got in the sorted pile.
 - ▶ Now pick a third card, and do the same.
 - ▶ ... and repeat this; until you've got no more cards in the unsorted pile.

The basic idea

- ▶ Effectively, we are maintaining two lists: one of sorted, processed cards, which we have looked at, and one of unsorted, unprocessed cards, which we haven't.

An example

- ▶ We will consider how this works with an example.
- ▶ Suppose we start with a list of six unsorted numbers – 3, 7, 4, 6, 8, and 5.
- ▶ We'll consider all six elements of the list to be our *unsorted* portion.

Java code for insertion sort

We will look at the code for insertion sort contained in the Java file `SortingAlgorithms.java`.

The `insertionSort` method looks like this:

```
14 public static void insertionSort(short[] arr) {
15     for (int j = 1; j < arr.length; j++) {
16         short key = arr[j];
17         int i = j - 1;
18         while (i >= 0 && arr[i] > key) {
19             arr[(i + 1)] = arr[i];
20             i = i - 1;
21         }
22         arr[i + 1] = key;
23     }
24 }
```

Java code for insertion sort

Some points about the code:

- ▶ `short[] arr` is the array we are sorting.
- ▶ Java has a few different sorts of numbers – `short`, `int`, and `long` for instance.
 - ▶ They differ from each other in *how large* the largest number they can hold is.
- ▶ The variables `i` and `j` are used here to contain *indexes* into the array `arr`.

Java code for insertion sort

- ▶ The variable `j` acts as a sort of boundary between the *unsorted* portion and the *sorted* portion.
- ▶ It is the index of the first element of the *unsorted* portion – i.e., the element which have just taken from the unsorted portion, and are trying to decide where to insert into the sorted portion.
- ▶ The `for` loop shows us that initially, the unsorted portion starts at position 1; and each time through the loop, moves one to the right.

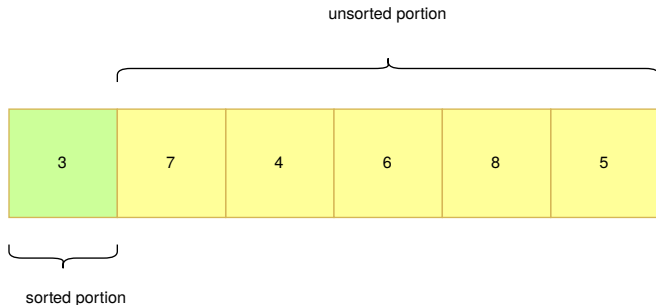
Java code for insertion sort

- ▶ Within the `for` loop is a `while` loop, which uses the variable `i`.
- ▶ We can imagine that `i` points to a spot in the *sorted* portion, and we are trying to decide whether the element we are looking at should be inserted there.
- ▶ The code shows that, initially, we consider the spot directly to the left of the element.
- ▶ And if that's not the right spot – we keep moving one spot to the left.
- ▶ We can think of the `arr[(i + 1)] = arr[i]` line as follows:
 - ▶ Each time we move `i` to the left – we shift the element “beneath” us to the right, so that we are making space in the unsorted portion for the new element to go.

Diagram of insertion sort

- ▶ Let's consider a diagram of how insertion sort works.

Diagram of insertion sort (outer loop run 0 times)



- ▶ At the very start of the algorithm, before the outer loop as even run once: we can already consider the 1st element to be in the *sorted* portion.
- ▶ i.e., We have a sorted portion of length 1.

Diagram of insertion sort (outer loop run 1 time)

- ▶ Then we will start the outer loop:
 - ▶ j will be set to 1, and the element we are trying to put into the “sorted” portion will be `arr[j]`, which is 7.
 - ▶ We will discover that 7 is bigger than 3, so the sorted portion is already in correct order.
- ▶ And at the end of the first run of the outer loop, the array looks like this:

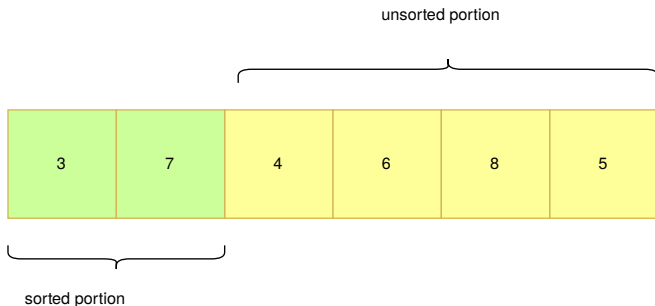


Diagram of insertion sort (outer loop run 2 times)

- ▶ Then we will run the outer loop a second time:
 - ▶ j will be set to 2, and the element we are trying to put into the “sorted” portion will be $\text{arr}[j]$, which is 4.
- ▶ The inner loop will compare 4 with the element to its left, and discover that 4 and 7 are out of order.
- ▶ At the end of the second run of the outer loop, the array looks like this:

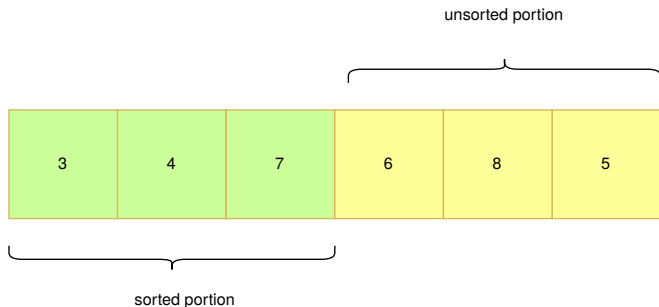


Diagram of insertion sort (outer loop run 3 times)

- ▶ And we continue with a third run of the outer loop, where j is 3 and the element we are trying to put into the sorted portion is 6.
- ▶ Try running the `insertionSort` method in Eclipse, using the debugger to see what is happening in the inner loop.
- ▶ After the third run of the outer loop, the array looks like this:

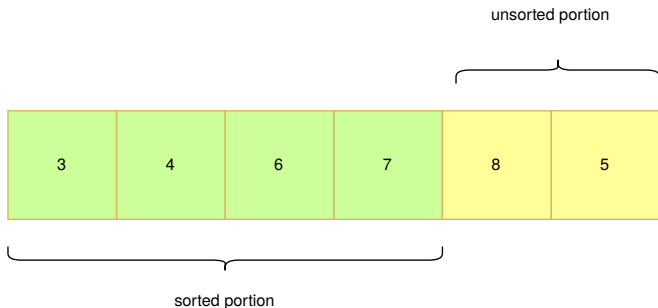


Diagram of insertion sort (outer loop run 4 times)

- After the fourth run of the outer loop, the array looks like this:

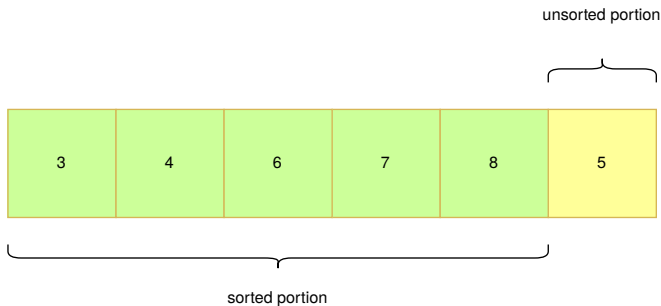
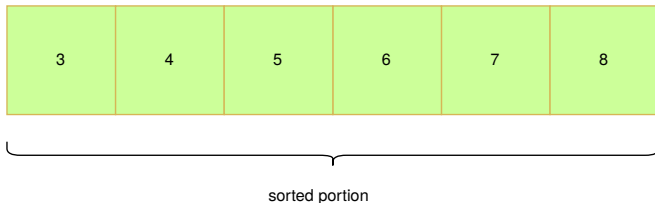


Diagram of insertion sort (outer loop run 5 times)

And finally, after the fifth run of the outer loop, all the elements are in the “sorted” portion, and the algorithm ends, with the array looking like this:



Complexity of insertion sort

- ▶ How long does it take to run insertion sort?
- ▶ We will first ask how long it takes to run in the *worst* case. (i.e., what is the longest possible time it could take.)
- ▶ We had two loops in our algorithm with index variables j and i .
- ▶ We had to “move” j over all the elements in the unsorted portion, shifting it one to the right each time.
- ▶ And within the sorted portion – if our list was *completely* out of order – we might have to move the i index through all the elements in the sorted portion.

Complexity of insertion sort

- ▶ If we have a loop over an array of size n , and each time through that loop, we have *another* loop over the array, we might think that the complexity of our algorithm is $O(n^2)$.
- ▶ And this is usually the case.
- ▶ But remember: our sorted and unsorted portions were never as long as the *whole* array – except at the end, when the “sorted” portion consisted of the whole array.
- ▶ The outer loop will indeed run n times.
- ▶ But – in the worst case – the inner loop will run 1 time, then 2 times, then 3 times, and so on – up until the last execution of the outer loop, when it will run `arr.length - 1` times. (Which is $n - 1$ times, since we are assuming our input is of size n .)

Complexity of insertion sort

- It turns out there is a simple formula for

$$1 + 2 + 3 + \dots + (n - 2) + (n - 1)$$

- The formula is:

$$\frac{n \times (n - 1)}{2}$$

which is equivalent to

$$\frac{n^2 - n}{2}$$

or

$$\frac{n^2}{2} - \frac{n}{2}$$

Complexity of insertion sort

- ▶ As n “tends towards infinity”, how will

$$\frac{n^2}{2} - \frac{n}{2}$$

behave?

- ▶ Well, the $\frac{n^2}{2}$ part will become much larger than the $\frac{n}{2}$. (In fact, as n approaches infinity, the $\frac{n}{2}$ part of the formula will become a vanishingly small proportion of it.)
- ▶ So the answer is that as n tends towards infinity, the value of the formula will approach

$$\frac{n^2}{2}$$

Complexity of insertion sort

- ▶ And *that* means that insertion sort has a worst-case running time of $O(n^2)$, pronounced “Big ‘O’ n squared”.
- ▶ This is sometimes called *quadratic running time*.

Best-case performance

- ▶ What about if we ran insertion sort on a list that was already sorted?
- ▶ In that case, the inner loop wouldn't need to run at all – the “sorted” portion would simply increase each time through the outer loop.
- ▶ So we would be able to sort the loop (in the best case) in n steps –
so insertion sort has a *best-case* performance of $O(n)$.