CS 161, Lecture 2 Rigorous Analysis of InsertionSort
Mary Wootters
Date: September 25, 2025

1 InsertionSort

In class, we sketched a proof by induction that InsertionSort is correct. In this handout, we will see how to write down this proof rigorously. This level of rigor will be appropriate for a problem set when you are asked to give a formal proof by induction.

Here's one way to implement InsertionSort (which we saw in class and in the pre-lecture exercise):

```
def InsertionSort(A):
   for i in range(1,len(A)):
      current = A[i]
      j = i-1
      while j >= 0 and A[j] > current:
          A[j+1] = A[j]
          j -= 1
      A[j+1] = current
```

2 Correctness of InsertionSort

Once you figure out what InsertionSort is doing (see the slides/lecture video for the intuition on this), you may think that it's "obviously" correct. However, if you didn't know what it was doing and just got the above code, maybe this wouldn't be so obvious. Additionally, for algorithms that we'll study in the future, it *won't* always be obvious that it works, and so we'll have to prove it. So in this handout we'll carefully go through a proof that InsertionSort is correct.

We will do the proof by induction on the number of iterations. Let's go over the informal idea first, and we'll do the formal proof below. Let `A` be our input list, and say that it has size n. Our inductive hypothesis will be that after iteration i of the outer loop, `A[:i+1]` is sorted.[1] This is obviously true after iteration 0 (aka, before the algorithm begins), because the one-element list $A[:1]$ is definitely sorted. Then we'll show that for any k with $0 < k < n$, if the inductive hypothesis holds for $i = k - 1$, then it holds for $i = k$. That is, if it is true that `A[:k]` is sorted after the $k-1$'st iteration, then it is true that `A[:k+1]` is sorted after the k'th iteration. At the end of the day, we'll conclude that `A[:n]` (aka, the whole array) is sorted after the $n-1$'st iteration, and we'll be done.

Formally, the argument goes like this:

Let `A` be the input list of length n.

- **Inductive hypothesis.** After iteration i of the outer loop, `A[:i+1]` is sorted.
- **Base case.** After iteration 0 of the outer loop (aka, before the algorithm begins), the list `A[:1]` contains only one element, and this is sorted.
- **Inductive step.** Let k be an integer so that $0 < k < n$. Suppose that the inductive hypothesis holds for $k-1$, so `A[:k]` is sorted after the $k-1$'st iteration. We want to show that `A[:k+1]` is sorted after the k'th iteration.

[1] An inductive hypothesis like this is sometimes called a *loop invariant,* because it's something that we want to hold (aka, be "invariant") at each iteration of the loop.

Suppose that j^* is the largest integer in $\{0, \ldots, k-1\}$ such that `A[j*] < A[k]`. Then the effect of the inner loop is to turn

$$[A[0], A[1], \ldots, A[j^*], \ldots, A[k-1], A[k]]$$

into

$$[A[0], A[1], \ldots, A[j^*], A[k], A[j^*+1], \ldots, A[k-1]].$$

We claim that this second list is sorted. This is because $A[k] > A[j^*]$, and by the inductive hypothesis, we have $A[j^*] \geq A[j]$ for all $j \leq j^*$, and so $A[k]$ is larger than everything that is positioned before it. Similarly, by the choice of j^* we have $A[k] \leq A[j^*+1] \leq A[j]$ for all $j \geq j^*+1$, so $A[k]$ is smaller than everything that comes after it. Thus, $A[k]$ is in the right place. All of the other elements were already in the right place, so this proves the claim.

Thus, after the k'th iteration completes, `A[:k+1]` is sorted, and this establishes the inductive hypothesis for k.

- **Conclusion.** By induction, we conclude that the inductive hypothesis holds for all $0 \leq i \leq n-1$. In particular, this implies that after the end of the $n-1$'st iteration (after the algorithm ends) `A[:n]` is sorted. Since `A[:n]` is the whole list, this means the whole list is sorted when the algorithm terminates, which is what we were trying to show.

This proof was maybe a bit pedantic: we used a lot of words to prove something that may have been pretty obvious. However, it's important to understand the structure of this argument, because we'll use it a lot, sometimes for more complicated algorithms.