Analysis of Insertion Sort

Like selection sort, insertion sort loops over the indices of the array. It just calls insert on the elements at indices $1,2,3,\ldots,n-1$. Just as each call to **indexOfMinimum** took an amount of time that depended on the size of the sorted subarray, so does each call to insert. Actually, the word "**does**" in the previous sentence should be "**can**," and we'll see why.

Let's take a situation where we call insert and the value being inserted into a subarray is less than every element in the subarray. For example, if we're inserting 0 into the subarray [2, 3, 5, 7, 11], then every element in the subarray has to slide over one position to the right. So, in general, if we're inserting into a subarray with k elements, all k might have to slide over by one position. Rather than counting exactly how many lines of code we need to test an element against a key and slide the element, let's agree that it's a constant number of lines; let's call that constant c. Therefore, it could take up to $c \cdot k$ lines to insert into a subarray of k elements.

Suppose that upon every call to insert, the value being inserted is less than every element in the subarray to its left. When we call insert the first time, $k=1$. The second time, $k=2$. The third time, k=3k=3. And so on, up through the last time, when $k=n-1$. Therefore, the total time spent inserting into sorted subarrays is

$$c \cdot 1 + c \cdot 2 + c \cdot 3 + \cdots c \cdot (n-1) = c \cdot (1+2+3+\cdots+(n-1))$$

That sum is an arithmetic series, except that it goes up to n−1 rather than n. Using our formula for arithmetic series, we get that the total time spent inserting into sorted subarrays is

$$c \cdot (n-1+1)((n-1)/2) = cn^2/2 - cn/2$$

Using big-Θ notation, we discard the low-order term $cn/2$ and the constant factors c and 1/2, getting the result that the running time of insertion sort, in

this case, is $\Theta(n^2)$.

Can insertion sort take less than $\mathbf{\Theta(n^2)}$ time? The answer is yes. Suppose we have the array [2, 3, 5, 7, 11], where the sorted subarray is the first four elements, and we're inserting the value 11. Upon the first test, we find that 11 is greater than 7, and so no elements in the subarray need to slide over to the right. Then this call of insert takes just constant time. Suppose that every call of insert takes constant time. Because there are n-1n−1 calls to insert, if each call takes time that is some constant cc, then the total time for insertion sort is $\mathbf{c \cdot (n-1)}$, which is $\mathbf{\Theta(n)}$, not $\mathbf{\Theta(n^2)}$.

Can either of these situations occur? Can each call to **insert** cause every element in the subarray to slide one position to the right? Can each call to insert cause no elements to slide? The answer is yes to both questions. A call to insert causes every element to slide over if the key being inserted is less than every element to its left. So, if every element is less than every element to its left, the running time of insertion sort is $\mathbf{\Theta(n^2)}$. What would it mean for every element to be less than the element to its left? The array would have to start out in reverse sorted order, such as [11, 7, 5, 3, 2]. So a reverse-sorted array is the worst case for insertion sort.

How about the opposite case? A call to **insert** causes no elements to slide over if the key being inserted is greater than or equal to every element to its left. So, if every element is greater than or equal to every element to its left, the running time of insertion sort is $\mathbf{\Theta(n)}$. This situation occurs if the array starts out already sorted, and so an already-sorted array is the best case for insertion sort.

What else can we say about the running time of insertion sort? Suppose that the array starts out in a random order. Then, on average, we'd expect that each element is less than half the elements to its left. In this case, on average, a call to insert on a subarray of **k** elements would slide **k/2** of them. The running time would be half of the worst-case running time. But in asymptotic notation, where constant coefficients don't matter, the running time in the average case would still be $\mathbf{\Theta(n^2)}$, just like the worst case.

What if you knew that the array was "almost sorted": every element starts out at most some constant number of positions, say 17, from where it's supposed

to be when sorted? Then each call to insert slides at most 17 elements, and the

time for one call of insert on a subarray of $\mathbf{k}$ elements would be at most 17·c. Over all $\mathbf{n-1}$ calls to insert, the running time would be $\mathbf{17 \cdot c \cdot (n-1)}$, which is $\mathbf{\Theta(n)}$, just like the best case. So insertion sort is fast when given an almost-sorted array.

To sum up the running times for insertion sort:

- Worst case: $\mathbf{\Theta(n^2)}$.
- Best case: $\mathbf{\Theta(n)}$.
- Average case for a random array: $\mathbf{\Theta(n^2)}$
- "Almost sorted" case: $\mathbf{\Theta(n)}$.

If you had to make a blanket statement that applies to all cases of insertion sort, you would have to say that it runs in $\mathbf{O(n^2)}$ time. You cannot say that it runs in $\mathbf{\Theta(n^2)}$ time in all cases, since the best case runs in $\mathbf{\Theta(n)}$ time. And you cannot say that it runs in $\mathbf{\Theta(n)}$ time in all cases, since the worst-case running time is $\mathbf{\Theta(n^2)}$.