COMP 285 (NC A&T, Spr '22) Lecture 1

Adapted From Virginia Williams' lecture notes. Additional credits: J. Su, W. Yang, Gregory Valiant, Mary Wootters, Aviad Rubinstein, Sami Alsheikh.

Insertion Sort, Proofs, and Formal Big-Oh

1 Introduction

In general, when analyzing an algorithm, we want to know two things.

1. Does it work?
2. How fast is it?

Today, we'll begin to see how we might formally answer these questions, through the lens of sorting. We'll start with InsertionSort to make sure we focus not on the algorithm itself, but on the newly introduced concepts.

2 InsertionSort

There are many ways to implement `InsertionSort`. You can see several C++ implementations here: https://www.onlinegdb.com/edit/7iNReAeY-n. For the purposes of lecture, we'll stick to pseudo-code for now. Here's a possible implementation of `InsertionSort`.

```
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
```

Let's ask our two questions: does this algorithm work, and does it have good performance?

2.1 Correctness of InsertionSort

Once you figure out what `InsertionSort` is doing (see the slides 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. To warm us up for those proofs, let's carefully go through a proof of correctness of `InsertionSort`.

We'll do the proof by maintaining a *loop invariant*, in this case that after iteration i, then $A[:i+1]$ is sorted. This is obviously true when $i = 0$ (because the one-element list $A[:1]$ is definitely sorted) and then we'll show that for any $i > 0$, if it's true for $i-1$, then it's true for i. At the end of the day, we'll conclude that $A[:n]$ (aka, the whole thing) is sorted and we'll be done.

Formally, we will proceed by induction.

- **Inductive hypothesis.** After iteration i of the outer loop, $A[:i+1]$ is sorted.
- **Base case.** When $i = 0$, $A[:1]$ contains only one element, and this is sorted
- **Inductive step.** Suppose that the inductive hypothesis holds for $i-1$, so $A[:i]$ is sorted after the $i-1$'st iteration. We want to show that $A[:i+1]$ is sorted after the i'th iteration.

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

  $$[A[0], A[1], \dots, A[j^*], \cdots, A[i-1], A[i]]$$

  into

  $$[A[0], A[1], \dots, A[j^*], A[i], A[j^*+1], ..., A[i-1]].$$

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

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

- **Conclusion.** By induction, we conclude that the inductive hypothesis holds for all $i \leq n-1$. 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 whose list is sorted when the algorithm terminates, which is what we were trying to show.

The above 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.

2.2 Running time of InsertionSort

The running time of `InsertionSort` is about n^2 operations. To be a bit more precise, at iteration i, the algorithm may have to look through and move i elements, so that's about $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ operations. We're not going to stress the precise operation count, because we'll argue that the end of the lecture that we don't care too much about it. The main question that we have, is, can we do asymptotically better than n^2 2? That is, can we come up with an algorithm that sorts an arbitrary list of n integers in time that scales less than n^2? For example, like $n^{1.5}$, or $n\log(n)$, or even n?

You should recall from COMP 280 that a much better algorithm exists, which is called `MergeSort` and which has running time $O(n\log(n))$.

3 Guiding Principles of Algorithm Design and Analysis

After going through the algorithm and analysis, it is natural to wonder if we've been too sloppy. In particular, note that the algorithm never "looks at" the input. For instance, what if we received the sequence of numbers $[1, 2, 3, 5, 4, 6, 7, 8]$? Clearly, there is a "sorting algorithm" for this sequence that only takes a few operations, but `MergeSort` runs through all $\log n + 1$ levels of recursion anyway. Would it be better to try to design our algorithms with this in mind? Additionally, in our analysis, we've given a very loose upper bound on the time required and dropped a number of constant factors and lower order terms. Is this a problem? In what follows, we'll argue that these are actually features, not bugs, in the design and analysis of the algorithm.

3.1 Worst-Case Analysis

One guiding principle we'll use throughout the class is that of Worst-Case Analysis. In particular, this means that we want any statement we make about our algorithms to hold for every possible input. Stated differently, we can think about playing a game against an adversary, who wants to maximize our running time (make it as bad as possible). We get to specify an algorithm and state a running time $T(n)$; the adversary then chooses an input. We win the game if even in the worst case, whatever input the adversary chooses (of size n), our algorithm runs in at most $T(n)$ time.

Note that because our algorithm made no assumptions about the input, then our running time bound will hold for every possible input. This is a very strong, robust guarantee [1].

[1] In the case where you have significant domain knowledge about which inputs are likely, you may choose to design an algorithm that works well in expectation on these inputs (this is frequently referred to as Average-Case Analysis). This type of analysis of ten much more tricky, and requires strong assumptions on the input

3.2 Asymptotic Analysis

Throughout our arguments about running time, we combined constants and gave very loose upper bounds (like being okay with a naive implementation of our pseudocode, or with this very wasteful upper bound 11m on the work done at a subproblem). Why did we choose to do this? First, it makes the math much easier. But does it come at the cost of getting the "right" answer? Would we get a more predictive result if we threw all these exact expressions back into the analysis? From the perspective of an algorithm designer, the answer is to both of these questions is a resounding "No". As an algorithm designer, we want to come up with results that are broadly applicable, whose truth does not depend on features of a specific programming language or machine architecture. The constants that we've dropped will depend greatly on the language and machine on which you're working. For the same reason we use pseudocode instead of writing our algorithms in C++, trying to quantify the exact running time of an algorithm would be inappropriately specific. This is not to say that constant factors never matter in applications (e.g. I would be rather upset if my web browser ran 7 times slower than it does now) but worrying about these factors is not the goal of this class. In this class, our goal will be to argue about which strategies for solving problems are wise and why.

In particular, we will focus on *Asymptotic Analysis*. This type of analysis focuses on the running time of your algorithm as your input size gets very large (i.e. $n \to +\infty$). This framework is motivated by the fact that if we need to solve a small problem, it doesn't cost that much to solve it by brute-force. If we want to solve a large problem, we may need to be much more creative in order for the problem to run efficiently. From this perspective, it should be very clear that $11n(\log n + 1)$ is much better than $n^2/2$. (If you are unconvinced, try plugging in some values for n.)

Intuitively, we'll say that an algorithm is "fast" when the running time grows "slowly" with the input size. In this class, we want to think of growing "slowly" as growing as close to linear as possible. Based on this this intuitive notion, we can come up with a formal system for analyzing how quickly the running time of an algorithm grows with its input size.

3.3 Asymptotic Notation

To talk about the running time of algorithms, we will use the following notation. $T(n)$ denotes the runtime of an algorithm on input of size n.

3.3.1 "Big-Oh" Notation:

Intuitively, Big-Oh notation gives an upper bound on a function. We say $T(n)$ is $O(f(n))$ when as n gets big, $f(n)$ grows at least as quickly as $T(n)$. Formally, we say

$$T(n) = O(f(n)) \iff \exists c, n_0 > 0 \text{ s.t } \forall n \geq n_0, 0 \leq T(n) \leq c \cdot f(n)$$

3.3.2 "Big-Omega" Notation:

Intuitively, Big-Omega notation gives a lower bound on a function. We say $T(n)$ is $\Omega(f(n))$ when as n gets big, $f(n)$ grows at least as slowly as $T(n)$. Formally, we say

$$T(n) = O(f(n)) \iff \exists c, n_0 > 0 \text{ s.t } \forall n \geq n_0, 0 \leq c \cdot f(n) \leq T(n)$$

3.3.3 "Big-Theta" Notation:

Intuitively, Big-Theta notation gives both a lower and upper bound on a function. We say $T(n)$ is $\Theta(f(n))$ if and only if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

$$T(n) = O(f(n)) \iff \exists c_1, c_2, n_0 > 0 \text{ s.t } \forall n \geq n_0, 0 \leq c_1 f(n) \leq T(n) \leq c_2 f(n)$$

We will cover each of these definition in more detail next lecture.