Insertion Sort and Induction

We will analyze the runtime of the following algorithm.

Algorithm 1 InsertionSort(A).

```
Input: A is an array of integers. It is indexed 1 to n.
for i = 2 to A.length do
    current = A[i]
    j = i − 1
    while j > 0 and A[j] > current do
        A[j + 1] = A[j]
        j = j − 1
    end while
    A[j + 1] = current
end for
return A
```

First, we analyze the algorithm's runtime. This time, we'll *formally* argue its runtime.

Theorem 1. *Insertion Sort runs in time $O(n^2)$.*

Proof. We iterate i from 2 to n. In each iteration, we conduct assignments (constant time) and a while loop which iterates at most $i-1$ times. Hence the running time $T(n)$ is

$$T(n) \quad \leq \quad \sum_{i=2}^{n} i-1 \quad \leq \quad \frac{n(n-1)}{2} \quad = \quad O(n^2).$$

□

Now, we argue the algorithm's *correctness*—that is, that on *every possible input*, it correctly outputs a sorted version.

Theorem 2. *For any input instance A, Insertion Sort returns an array sorted in ascending order.*

Proof. We show the following by induction on i: At the end of the i^{th} "for" loop iteration, the sub-array $A[1,\ldots,i-1]$ is sorted in ascending order.

Base Case ($i = 2$): When $i = 2$, the sub-array is $A[1]$, a single element, which is sorted.

Inductive Hypothesis: Suppose that for some $i \geq 2$, $A[1,\ldots,i-1]$ is sorted after the i^{th} "for" loop.

Inductive Step ($i+1$): We will need the following Lemma, which needs to be proven separately.

Lemma 1. *Within the "for" loop for a fixed i, let j^{final} be the j at the end of the "while" loop. Then the "while" loop shifts $A[j^{final}+1, i-1]$ to $A[j^{final}+2, i]$ in the same order.*

By the Inductive Hypothesis (IH), $A[1, i-1]$ is sorted. Hence, so are any split at some k into the subarrays $A[1, k]$ and $A[k+1, i-1]$ for $j < i-1$. And then, by Lemma 1, after the "while" loop, so is $A[j+2, i]$.

Consider the execution of the while loop: we enter the while loop for j such that while $A[j] > current$. Hence, for all elements shifted in the while loop $k \in \{j+2, \ldots, i\}$, $A[k] > current$. We exit the while loop when this is no longer true (and we're working with a sorted subarray), hence, for all $k \leq j^{\text{final}}$, $A[k] < current$ (or, alternatively, $j^{\text{final}} = 0$). Hence $A[1, j^{\text{final}}]$, $A[j] = current$, $A[j^{\text{final}}+2, i]$ forms a sorted array of $A[1, i]$. □

Proving Lemma 1: Loop Invariants

Definition 1. A *loop invariant* is something that is true before we start and after every iteration of a loop.

We prove that a loop invariant is true by showing the following three things about it:

- **Initialization:** It is true prior to the first iteration of the loop.
- **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

See CLRS Section 2.1 for details, p.18 in the Third Edition. We use this to prove our Lemma.

Proof of Lemma 1. We prove this as a loop invariant. Before, and after each "while" loop (within the "for" loop for a fixed i), $A[j+1, i-1]$ shifts to $A[j+2, i]$ in the same order.

Initialization: Before the first iteration of the "while" loop, $j = i-1$, so the sub-array in question is empty and thus vacuously shifted.

Maintenance: If our statement holds before an iteration of the loop—that $A[j+1, i-1]$ is shifted to $A[j+2, i]$ in order—then after assigning $A[j+1] = A[j]$, it is now true that $A[j, i-1]$ was shifted to $A[j+1, i]$, and after decrementing j to $j-1$, it is true for our updated j that $A[j+1, i-1]$ was shifted to $A[j+2, i]$.

Termination: When the loop terminates, $A[j^{\text{final}}+2, i]$ contains a sorted array that was previously at $A[j^{\text{final}}+1, i-1]$. □

This completes our proof of Lemma 1, and thus Insertion Sort's correctness!

Exercises:

- Prove that $A[j] < current$ using loop invariants.
- Prove what we proved using induction about the for loop using loop invariants.

Summary

To summarize, from the first two lectures, loop invariants should be the only new skill, and the skills we've reviewed are:

- Pseudocode
- Analyzing Runtime
- Asymptotic Notation (O, Ω, Θ)
- Induction
- Increasing comfort with formal definitions, lemmas, etc.

If you need extra review, please see Lectures 1-6 from DS 120 listed under the resource page on Piazza!

Comparison-Based Lower Bound via A Counting Argument

Note: We did not and will not cover this in class, but it's a very nice proof as to why sorting cannot be faster than $\Omega(n \log n)$.

So Insertion Sort provably always correctly sorts any input array in $O(n^2)$ time! But can we do better? Perhaps we can improve on the $O(n^2)$ running time to get an algorithm that runs in time $O(n)$? To answer this question we need to be more precise about what a "solution" can do. Selection sort inspects the input data using only a single operation: a comparison (i.e. its branching condition is of the form "If $A[i] \leq A[j]$ then. . . "). It is the result of these comparisons (and nothing else) that determines which swaps are performed, which comparisons are performed next, and ultimately which permutation π of the input array A is finally output. That is to say, Insertion Sort operates in the comparison based model of computation:

Definition 2. An algorithm operates in the *Comparison Model* if it can be written as a binary decision tree in which:

1. Each vertex is labelled with a fixed comparison (i.e. $A[i] < A[j]$ for particular i, j)
2. Computation proceeds as a root-leaf path down the tree, branching left if the comparison evaluates to TRUE and right otherwise, and
3. The leaves are labelled with the output of the algorithm (in this case, permutations)

In this model, the running time of the algorithm corresponds to the depth of the tree.

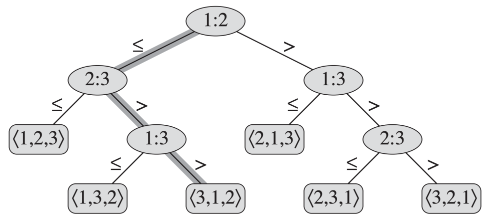

Figure 1: Example decision tree from CLRS.

As it turns out, we can prove an easy lower bound for sorting algorithms in the comparison model. Lower bounds of this sort serve as a guide: either we should not waste effort trying to derive algorithms that improve on the lower bound, or, we should find techniques that step outside of the model in which the lower bound is proven.

Theorem 3. *Any algorithm that solves the sorting problem in the comparison model must have run time at least* $\Omega(n \log n)$.

Proof. The proof is via a nice counting argument. Consider any sorted array A of length n. Consider the $n!$ permutations of A when given as input to our algorithm. It must be that each permutation has a distinct root-to-leaf path in the decision tree—otherwise, all of the comparisons evaluate to the same values, indicating that the input order is identical. Hence, there must be at least $L > n!$ leaves in the algorithm's binary decision tree.

On the other hand, a binary tree of depth d has $L \leq 2^d$ many leaves. Here d is the running time of our algorithm, so by combining these two bounds, we have that:

$$2^d \geq n!$$

taking the log of both sides, we have:

$$d \geq \log(n!) = \Omega(n \log n).$$

□