CS 161 (Stanford, Winter 2026) Lecture 2

Adapted From Virginia Williams' lecture notes.
Additional credits: Michael Kim, Ofir Geri, Mary Wootters, Aviad Rubinstein.
Please direct all typos and mistakes to Ellen Vitercik and Moses Charikar.

MergeSort, Recurrences 101, and Asymptotic Analysis

1 Introduction

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

1. Does it work?
2. Does it have good performance?

Today, we'll begin to see how we might formally answer these questions, through the lens of sorting. We'll start, as a warm-up, with InsertionSort, and then move on to the (more complicated) MergeSort. MergeSort will also allow us to continue our exploration of *Divide and Conquer.*

2 InsertionSort

In your pre-lecture exercise, you should have taken a look at a couple of ways of implementing InsertionSort. Here's one:

```
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 us 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 true when $i = 0$ (because the one-element list $A[:1]$ is 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 $\{0, \ldots, i-1\}$ so that `A[j*]` $\leq$ `A[i]`. Then the effect of the inner loop is to turn
  $$[A[0], A[1], \ldots, A[j^*], \ldots, A[i-1], A[i]]$$
  into
  $$[A[0], A[1], \ldots, A[j^*], A[i], A[j^*+1], \ldots, A[i-1]].$$
  We claim that this latter list is sorted. This is because $A[i] \geq A[j^*]$, and by the inductive hypothesis, we have $A[j^*] \geq A[j]$ for all $j \leq j^*$, and so $A[i]$ is larger than or equal to than everything that is positioned before it. Similarly, by the choice of j^* we have $A[i] < A[j^*+1] \leq A[j]$ 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 whole 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? 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? Next, we'll see that MergeSort will scale like $n \log(n)$, which is much faster.

3 MergeSort

Recall the *Divide-and-conquer* paradigm from the first lecture. In this paradigm, we use the following strategy:

- Break the problem into sub-problems.
- Solve the sub-problems (often recursively)
- Combine the results of the sub-problems to solve the big problem.

At some point, the sub-problems become small enough that they are easy to solve, and then we can stop recursing.

With this approach in mind, MergeSort is a very natural algorithm to solve the sorting problem. The pseudocode is below:

```
MergeSort(A):
  n = len(A)
  if n <= 1:
     return A
  L = MergeSort( A[:n/2] )
  R = MergeSort( A[n/2:] )
  return Merge(L, R)
```

Above, we are using Python notation, so $A[: n/2] = [A[0], A[1], \dots, A[n/2-1]]$ and $A[n/2 :] = [A[n/2], \dots, A[n-1]]$. Additionally, we're using integer division, so `n/2` means $\lfloor n/2 \rfloor$.

How do we do the `Merge` procedure? We need to take two sorted arrays, L and R, and merge them into a sorted array that contains both of their elements. See the slides for a walkthrough of this procedure.

```
Merge(L, R):
   m = len(L) + len(R)
   S = [ ]
   for k in range(m):
      if L[i] < R[j]:
         S.append( L[i] )
         i += 1
      else:
         S.append( R[j] )
```

```
        j += 1
return S
```

Note: This pseudocode is incomplete! What happens if we get to the end of L or R? Try to adapt the pseudocode above to fix this.

As before, we need to ask: Does it work? And does it have good performance?

3.1 Correctness of MergeSort

Let's focus on the first question first. As before, we'll proceed by induction. This time, we'll maintain a *recursion invariant* that any time MergeSort returns, it returns a sorted array.

- **Inductive Hypothesis.** Whenever MergeSort returns an array of size $\leq i$, that array is sorted.
- **Base case.** Suppose that $i = 1$. Then whenever MergeSort returns an array of length 0 or length 1, that array is sorted. (Since all arrays of length 0 and 1 are sorted). So the Inductive Hypothesis holds for $i = 1$.
- **Inductive step.** We need to show that if MergeSort always returns a sorted array on inputs of length $\leq i - 1$, then it always does for length $\leq i$. Suppose that MergeSort has an input of length i. Then L and R are both of length $\leq i - 1$, so by induction, L and R are both sorted. Thus, the inductive step boils down to the statement:

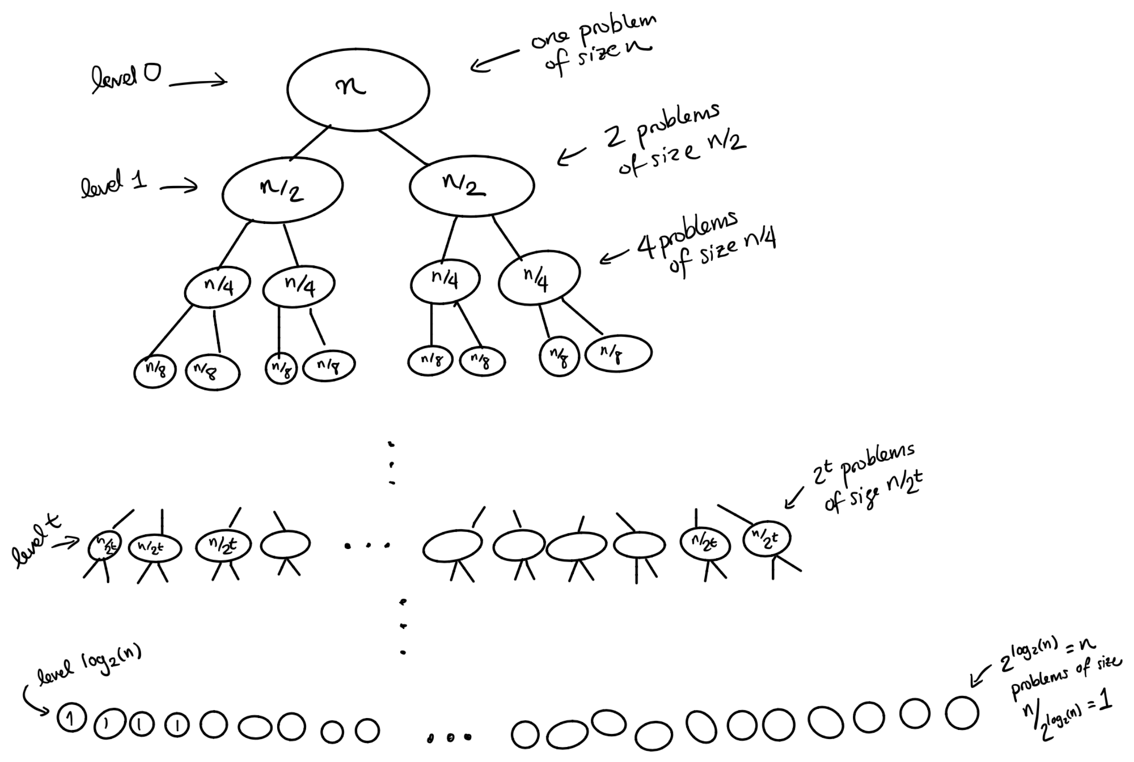
  "When Merge takes as inputs two sorted arrays L and R, then it returns a sorted array containing all of the elements of L, along with all of the elements of R."

  This statement is intuitively true, although proving it rigorously takes a bit of bookkeeping. It takes another proof by induction! Check out CLRS Section 2.3.1 for rigorous treatment.
- **Conclusion.** By induction, the Inductive hypothesis holds for all i. In particular, given an array of any length n, MergeSort returns a sorted version of that array.

3.2 Running time of MergeSort

Finally, we get to our first question in this lecture where the answer may not be intuitively obvious. What is the running time of MergeSort? In the next few lectures, we'll see a few principled ways of analyzing the runtime of a recursive algorithm. Here, we'll just go through one of the ways, which is called the *recursion tree* method.

The idea is to draw a tree representing the computation (see the slides for the visuals). Each node in the tree represents a subproblem, and its children represent the subproblems we need to solve to solve the big sub-problem. The recursion tree for MergeSort looks something like this:



At the top (zeroth) level is the whole problem, which has size n. This gets broken into two subproblems, each of size $n/2$, and so on. At the t'th level, there are 2^t problems, each of size $n/2^t$. This continues until we have n problems of size 1, which happens at the $\log(n)$th level.

Some notes:

- In this class, logarithms will **always** be base 2, unless otherwise noted.
- We are being a bit sloppy in the picture above: what if n is not a power of 2? Then $n/2^j$ might not be an integer. In the pseudocode above, we break a problem of size n into problems of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. Keeping track of this in our analysis will be messy, and it won't add much, so we will ignore it, and for now, we will assume that n is a power of 2. [1]

To figure out the total amount of work, we will figure out how much work is done at each node in the tree, and add it all up. To that end, we tally up the work that is done in a particular node in the tree—that is, in a particular call to MergeSort. There are three things:

1. Checking the base case

[1]To formally justify the assumption that n is a power of 2, notice that we can always sort a *longer* list of length $n' = 2^{\lceil \log_2(n) \rceil}$. That is, we'll add extra entries, whose values are ∞, to the list. Then we sort the new list of length n' and return the first n values. Since $n' \leq 2n$ (why?) this won't affect the asymptotic running time. Also, see CLRS Section 4.6.2 for a rigorous analysis of the original algorithm with floors and ceilings.

2. Making recursive calls (but we don't count the work done in those recursive calls; that will count in other nodes)
3. Running Merge.

Let's analyze each of these. Suppose that our input has size m (so that $m = n/2^j$ for some j).

1. Checking the base case doesn't take much time. For concreteness, let us say that it takes one operation to retrieve the length of A, and another operation to compare this length to 1, for a total of two operations.[2]
2. Making the recursive calls should also be fast. If we implemented the pseudocode well, it should also take a constant number of operations.

   > **Aside:** This is a good point to talk about how we interpret pseudocode in this class. Above, we've written `MergeSort(A[:n/2])` as an example of a recursive call. This makes it clear that we are supposed to recurse on the first half of the list, but it's not clear how we implement that. Our "pseudocode" above is working Python code, and in Python, this implementation, while clear, is a bit inefficient. That is, written this way, Python will *copy* the first $n/2$ elements of the list before sending them to the recursive call. A much better way would be to instead just pass in pointers to the 0'th and $n/2 - 1$'st index in the list. This would result in a faster algorithm, but kludgier pseudocode. In this class, we generally will opt for cleaner pseudocode, as long as it does not hurt the *asymptotic* running time of the algorithm. In this case, our simple-but-slower pseudocode turns out not to affect the asymptotic running time, so we'll stick with this.

   In light of the above **Aside**, let's suppose that this step takes $m + 2$ operations, $m/2$ to copy each half of the list over, and 2 operations to store the results. Of course, a better implementation of this step would only take a constant number of (say, four) operations.
3. The third thing is the tricky part. We claim that the Merge step also takes about m operations.

   Consider a single call to Merge, where we'll assume the total size of A is m numbers. How long will it take for Merge to execute? To start, there are two initializations for i and j. Then, we enter a for loop which will execute m times. Each loop will require one comparison, followed by an assignment to S and an increment of i or j. Finally, we'll need to increment the counter in the for loop k. If we assume that each operation costs us a certain amount of time, say $Cost_a$ for assignment, $Cost_c$ for comparison, $Cost_i$ for incrementing a counter, then we can express the total time of the Merge subroutine

[2]Of course, there's no reason that the "operation" of getting the length of A should take the same amount of time as the "operation" of comparing two integers. This disconnect is one of the reasons we'll introduce big-Oh notation at the end of this lecture.

as follows:

$$2Cost_a + m(Cost_a + Cost_c + 2Cost_i)$$

This is a precise, but somewhat unruly, expression of the running time. In particular, it seems difficult to keep track of lots of different constants, and it isn't clear which costs will be more or less expensive (especially if we switch programming languages or machine architectures). To simplify our analysis, we choose to assume that there is some global constant c_{op} which represents the cost of an operation. You may think of c_{op} as $\max\{Cost_a, Cost_c, Cost_i, \ldots\}$. We can then bound the amount of running time for Merge as

$$2c_{op} + 4c_{op}m = 2 + 4m \text{ operations}$$

Thus, the total number of operations is at *most*

$$2 + (m + 2) + 4m + 2 \leq 11m$$

using the assumption that $m \geq 1$. This is a very loose bound; for larger m this will be much closer to $5m$ than it is to $11m$. But, as we'll discuss more below, the difference between 5 and 11 won't matter too much to us, so much as the linear dependence on m.

Now that we understand how much work is going on in one call where the input has size m, let's add it all up to obtain a bound on the number of operations required for MergeSort. in a Merge of m numbers, we want to translate this into a bound on the number of operations required for MergeSort. At first glance, the pessimist in you may be concerned that at each level of recursive calls, we're spawning an exponentially increasing number of copies of MergeSort (because the number of calls at each depth doubles). Dual to this, the optimist in you will notice that at each level, the inputs to the problems are decreasing at an exponential rate (because the input size halves with each recursive call). Today, the optimists win out.

Claim 1. MergeSort *requires at most* $11n \log n + 11n$ *operations to sort* n *numbers.*

Before we go about proving this bound, let's first consider whether this running time bound is good. We mentioned earlier that more obvious methods of sorting, like InsertionSort, required roughly n^2 operations. How does $n^2 = n \cdot n$ compare to $n \cdot \log n$? An intuitive definition of $\log n$ is the following: "Enter n into your calculator. Divide by 2 until the total is ≤ 1. The number of times you divided is the logarithm of n." This number in general will be significantly smaller than n. In particular, if $n = 32$, then $\log n = 5$; if $n = 1024$, then $\log n = 10$. Already, to sort arrays of $\approx 10^3$ numbers, the savings of $n \log n$ as compared to n^2 will be orders of magnitude. At larger problem instances of 10^6, 10^9, etc. the difference will become even more pronounced! $n \log n$ is much closer to growing linearly (with n) than it is to growing quadratically (with n^2).

One way to argue about the running time of recursive algorithms is to use *recurrence relations*. A recurrence relation for a running time expresses the time it takes to solve an input of size n in terms of the time required to solve the recursive calls the algorithm makes. In particular, we can write the running time $T(n)$ for MergeSort on an array of n numbers as the following expression.

$$\begin{aligned} T(n) &= T(n/2) + T(n/2) + T(\text{Merge}(n)) \\ &\leq 2 \cdot T(n/2) + 11n \end{aligned}$$

There are several sophisticated and powerful techniques for solving recurrences. We will cover many of these techniques in the coming lectures. Today, we can analyze the running time directly.

Proof of Claim 1. Consider the recursion tree of a call to MergeSort on an array of n numbers. Assume for simplicity that n is a power of 2. Let's refer to the initial call as Level 0, the proceeding recursive calls as Level 1, and so on, numbering the level of recursion by its depth in the tree. How deep is the tree? At each level, the size of the inputs is divided in half, and there are no recursive calls when the input size is ≤ 1 element. By our earlier "definition", this means the bottom level will be Level $\log n$. Thus, there will be a total of $\log n + 1$ levels.

We can now ask two questions: (1) How many subproblems are there at Level i? (2) How large are the individual subproblems at Level i? We can observe that at the ith level, there will be 2^i subproblems, each with inputs of size $n/2^i$.

We've already worked out that each sub-problem with an input of size $n/2^i$ takes at most $11n/2^i$ operations. Now we can add this up:

$$\begin{aligned} \text{Work at Level } i &= (\text{number of subproblems}) \cdot (\text{work per subproblem}) \\ &\leq 2^i \cdot 11\left(\frac{n}{2^i}\right) \\ &= 11n \text{ operations.} \end{aligned}$$

Importantly, we can see that the work done at Level i is independent of i – it only depends on n and is the same for every level. This means we can bound the total running time as follows:

$$\begin{aligned} \text{Total number of operations} &= (\text{operations per level}) \cdot (\text{number of levels}) \\ &\leq (11n) \cdot (\log n + 1) \\ &= 11n \log n + 11n \end{aligned}$$

□

This proves the claim, and we're done!

4 Guiding Principles for 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 $[1, 2, 3, 5, 4, 6, 7, 8]$? 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 of Merge and dropped some constant factors and lower-order terms. Is this a problem? In what follows, we'll argue that these are *features*, not bugs, in the design and analysis of the algorithm.

4.1 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. [3]

4.2 Asymptotic Analysis

Throughout our argument about MergeSort, we combined constants (think $Cost_a, Cost_i$, etc.) 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 in MergeSort). 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 to both of these questions is a resounding "No". As the algorithm designer, we want to come up with broadly applicable results, 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 Java, 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

[3] 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 is often much more tricky and requires strong assumptions on the input.

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 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 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.

4.3 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 an input of size n.

"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

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

"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

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

"Big-Theta" Notation:

$T(n)$ is $\Theta(f(n))$ if and only if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$. Equivalently, we can say that

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

We can see that these notations do capture exactly the behavior that we want – namely, to focus on the rate of growth of a function as the inputs get large, ignoring constant factors and lower-order terms. As a sanity check, consider the following example and non-example.

Claim 2. *All degree-k polynomials*[4] *are $O(n^k)$.*

[4]To be more precise, all degree-k polynomials T so that $T(n) \geq 0$ for all $n \geq 1$. How would you adapt this proof to be true for all degree-k polynomials T with a positive leading coefficient?

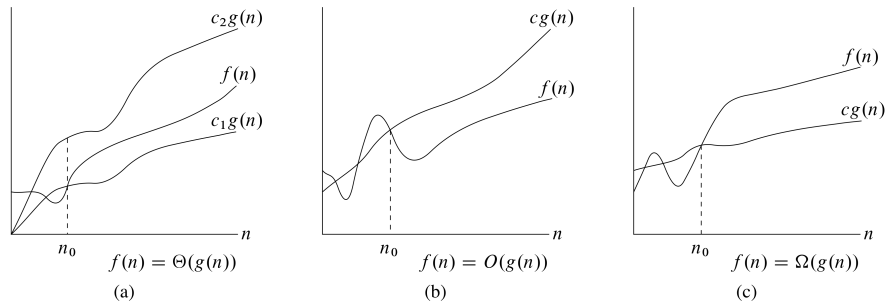


Figure 3.1 from CLRS – Examples of Asymptotic Bounds
(Note: In these examples $f(n)$ corresponds to our $T(n)$ and $g(n)$ corresponds to our $f(n)$.)

Proof. Suppose $T(n)$ is a degree-k polynomial. That is, $T(n) = a_k n^k + \ldots + a_1 n + a_0$ for some choice of a_i's where $a_k \neq 0$. To show that $T(n)$ is $O(n^k)$ we must find a c and n_0 such that for all $n \geq n_0$ $T(n) \leq c \cdot n^k$. (Since $T(n)$ represents the running time of an algorithm, we assume it is positive.) Let $n_0 = 1$ and let $a^* = \max_i |a_i|$. We can bound $T(n)$ as follows:

$$\begin{aligned} T(n) &= a_k n^k + \ldots + a_1 n + a_0 \\ &\leq a^* n^k + \ldots + a^* n + a^* \\ &\leq a^* n^k + \ldots + a^* n^k + a^* n^k \\ &= (k+1)a^* \cdot n^k \end{aligned}$$

Let $c = (k+1)a^*$ which is a constant, independent of n. Thus, we've exhibited c, n_0 which satisfy the Big-Oh definition, so $T(n) = O(n^k)$. □

Claim 3. *For any $k \geq 1$, n^k is not $O(n^{k-1})$.*

Proof. By contradiction. Assume $n^k = O(n^{k-1})$. Then there is some choice of c and n_0 such that $n^k \leq c \cdot n^{k-1}$ for all $n \geq n_0$. But this in turn means that $n \leq c$ for all $n \geq n_0$, which contradicts the fact that c is a constant, independent of n. Thus, our original assumption was false and n^k is not $O(n^{k-1})$. □