

This seems like a good time to catch our breath and review what we've covered so far.

First, a bit of a bio → I self-identify as a “grinder.” Most of my classes weren’t easy, but I just kept persisting until I understood the content. The one exception was Numerical Analysis, the blend of Math and CS was weirdly natural to me.

My Ph.D. is in Applied Mathematics, in the subfield of Numerical Analysis, and specifically is simulated fluid flow.

In grad school, we covered Chebyshev Polynomials in one of my classes. I thought they were the coolest things ever, and that I definitely had never seen them before. Then one day, I was reviewing the notes from my undergrad Numerical Analysis course, and it turns out we’d spent a week on Chebyshev Polynomials!

So, I get it, it’s natural to say, “I’ve never seen that before.” But a lot of the material we’ve covered in this class so far was in your previous coursework, CS 110, CS 240, and MA 116.

What is your method when you get something which needs to be assembled?

- Read directions?
- Just start putting together?

In this class, the notes I follow for lecture are in Moodle, students have the notes they take during lecture, and there are the readings from the textbook. In addition, I have provided class recordings from previous semesters, as a convenience which you may find helpful.

My pedagogical goal for the Homework assignments is to draw students into applying the material by having them read the textbook and review the notes. If students just dive into the problems, they might find they need to take a step back and read the textbook.

“Amount of work,” “rate of growth,” and running time. What do we mean? pg. 32 and 33 in the text:

Let’s now make one more simplifying abstraction: it is the *rate of growth*, or *order of growth*, of the running time that really interests us. We therefore consider only the leading term of a formula (e.g., an^2), since the lower-order terms are relatively insignificant for large values of n . We also ignore the leading term’s constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs. For insertion sort’s worst-case whose worst-case running time is $\Theta(n^3)$. Regardless of the constants hidden by the Θ -notation, there is always some number, say n_0 , such that for all input sizes $n \geq n_0$, the $\Theta(n^2)$ algorithm beats the $\Theta(n^3)$ algorithm in the worst case.

“ θ -notation means ‘roughly proportional when n is large.’” $\theta(n^2)$ means roughly proportional to n^2 when n is large.

Consider the sorts we’ve encountered so far:

- Iterative (loops)
 - Insertion Sort
 - Selection Sort
- Recursive
 - Mergesort

The running time (or “amount of work” or “rate of growth”) is given as

$$T(n) = \theta(\text{some function of } n), \text{ or } O(\) \text{ or } \Omega(\)$$

Determining the running time $T(n)$ (or “amount of work” or “rate of growth”) for an iterative sort usually involves working with series (see below).

Determining the running time $T(n)$ (or “amount of work” or “rate of growth”) for a recursive algorithm involves guessing a solution for a **recurrence relation**, which on page 39 in the text is described as, “*the overall running time of the same algorithm on smaller inputs.*”

You might be asked to find a lower-bound, $\Omega(\)$ for the running time of a recurrence relation.

“Is there a lot of math in this course?”

- Series
- Logarithms
- Exponents
- Quantifiers
- Inductive Proofs

Series: why are they important in an Algorithms and Analysis course?

- They help us describe the running time of algorithms.
- Sigma notation vs. explicitly writing out the series:

A handwritten note on lined paper. It shows the summation formula $\sum_{i=1}^n i = 1 + 2 + \dots + n$. Below the formula, there is a wavy arrow pointing from the term i in the summation to the word "nicknames" written on the line below.

- What we really want is the solution, or value of, this series, in terms of n :
 $\sum_{i=1}^n i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$, and even more, we would like to express this in θ -notation, which would be $\theta(n^2)$.
- Maybe it helps to think of these series as code. What would this look like in Python?

```
sum=0;
for i in range(n+1):
    sum += i
print(sum)
```

- We've evaluated a series to determine the running time of selection sort:

A handwritten note on lined paper. It shows the summation formula $\sum_{i=1}^{n-1} t_i = t_1 + t_2 + t_3 + \dots + t_{n-1}$. Below the formula, there is a question mark above the word "What are these t_i 's?"

The t_i 's are the number of times Line 4 (the most frequently executed line) in the Selection Sort pseudocode would run for a given value of i .

when $i=1$, $t_1 = n$
 $i=2$, $t_2 = n-1$
 \vdots
 \vdots
 $i=n-1$, $t_{n-1} = 2$

This simply gives the series $n + (n-1) + (n-2) + \dots + 3 + 2$.

What would this series be in θ notation?

Logarithms:

First, consider the exponential function:

x	10^x
1	$10^1 = 10$
2	$10^2 = 100$
3	$10^3 = 1000$

What if we invert it, swapping the columns? This is the logarithm, “what exponent of the base gives this value?”

x	$\log_{10}(x)$
10	1
100	2
1000	3

If $\log_{10}(1000) = \log_{10}(10^3) = 3$, what would $\log_{10}(10^b) = ?$

The rules for other bases, such as 2 (2^x and $\lg()$), are the same.

Exponents:

- $8 * 4 = 32$
- $2^3 * 2^2 = 2^5$
- $2^{3+2} = 2^5$

How would you write 2^{n+1} to be $c * 2^n$, where c is a constant?

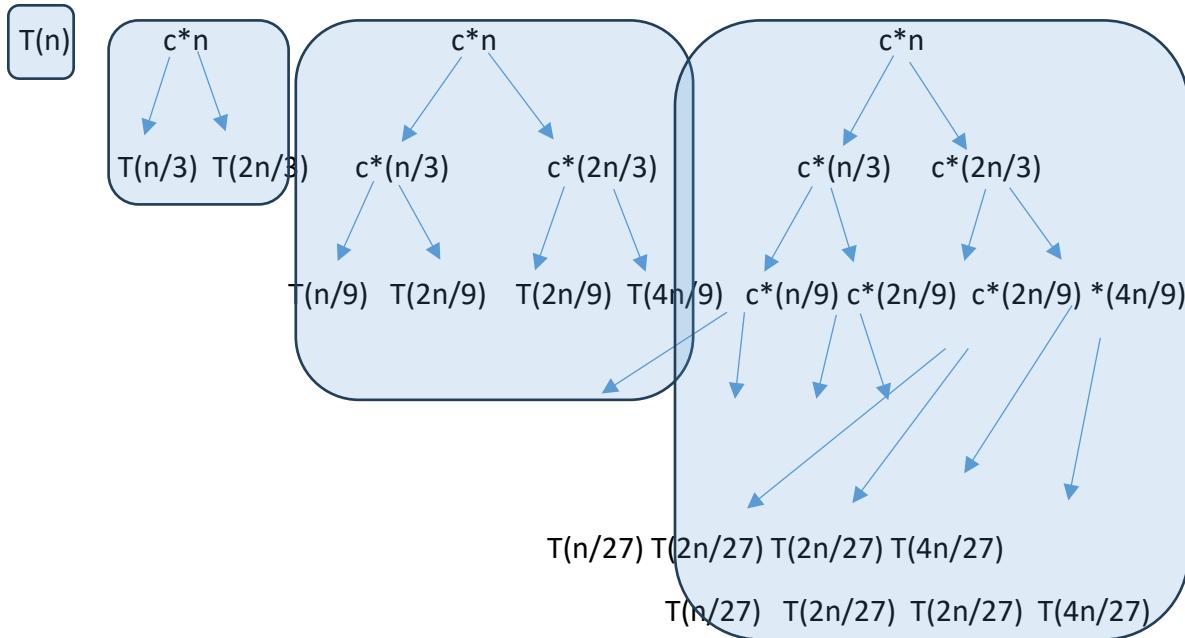
Quantifiers:

- Universal, “for all,” “every,” is the symbol \forall
- Existential, “there exists,” “at least one,” is the symbol \exists
- These come up in the definitions of $\theta()$, $O()$, and $\Omega()$

You may be asked to find just one constant c and one constant n_0 that would make the definition true. You can guess and check, or try some algebra.

- “**Tight Bound**” $\theta(g(n)) = \{ f(n) : \exists \text{ positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall n \geq n_0 \}$
- “**Upper Bound**” $O(g(n)) = \{ f(n) : \exists \text{ positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c \cdot g(n) \quad \forall n \geq n_0 \}$
- “**Lower Bound**” $\Omega(g(n)) = \{ f(n) : \exists \text{ positive constants } c \text{ and } n_0 \text{ such that } 0 \leq c \cdot g(n) \leq f(n) \quad \forall n \geq n_0 \}$

Let's revisit the recurrence $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn$. We use recurrence relations to help us guess what $T(n)$ is as a function of n . We argued that a **lower-bound** for the order of growth/amount of work/running time was $\Omega(n \lg(n))$. We wrote out a sequence of recurrence trees to find the pattern:



Since we were considering the **lower-bound** for the running time, we only worried about the recursive calls that were $\frac{1}{3}$ the size each time (the left side of the trees, the way it's written here), since they would reach problem sizes of 1 the quickest. *There were other recursive calls, more work happening, that we could disregard.*

Let's say we want to consider an **upper-bound** for the order of growth/amount of work/running time for the algorithm associated with the recurrence relation above. In this case we would exclusively consider the recursive calls that were $\frac{2}{3}$ the size each time (the right side of the trees, the way it's written here). The height of the right side of the tree is $\log_{\frac{3}{2}}(n)$, which asymptotically is equivalent to $\log_2(n) = \lg(n)$. But amount of work on each remaining level will not sum to n , since the $\frac{1}{3}$ size recursive calls will have completed. For that reason, we say an upper bound for the above recurrence relation is $O(n \lg(n))$.

On pages 56 and 57, the textbook has a nice discussion of asymptotic notation (meaning we consider functions of n as the problem size n tends to infinity). Remember that we not only have the purely mathematical discussion of upper, tight, and lower bounds, but the discussion of the “shape” of the data that gives the best, average, and worst case running times.

Remember for Insertion Sort that the worst-case is reverse sorted data, since the inner while loop would be executed the maximum number of times for each iteration of the outer loop. Insertion Sort’s best case is already sorted data, since the inner while loop would only be executed once for each iteration of the outer loop.

When you use asymptotic notation to characterize an algorithm’s running time, make sure that the asymptotic notation you use is as precise as possible without overstating which running time it applies to. Here are some examples of using asymptotic notation properly and improperly to characterize running times.

Let’s start with insertion sort. We can correctly say that insertion sort’s worst-case running time is $O(n^2)$, $\Omega(n^2)$, and—due to Theorem 3.1— $\Theta(n^2)$. Although

all three ways to characterize the worst-case running times are correct, the $\Theta(n^2)$ bound is the most precise and hence the most preferred. We can also correctly say that insertion sort’s best-case running time is $O(n)$, $\Omega(n)$, and $\Theta(n)$, again with $\Theta(n)$ the most precise and therefore the most preferred.

But if someone was to ask, “What is Insertion Sort’s running time?” **without any mention of best or worst case**, then the correct answer would be $O(n^2)$.

Here is what we *cannot* correctly say: insertion sort’s running time is $\Theta(n^2)$. That is an overstatement because by omitting “worst-case” from the statement, we’re left with a blanket statement covering all cases. The error here is that insertion sort does not run in $\Theta(n^2)$ time in all cases since, as we’ve seen, it runs in $\Theta(n)$ time in the best case. We can correctly say that insertion sort’s running time is $O(n^2)$, however, because in all cases, its running time grows no faster than n^2 . When we say $O(n^2)$ instead of $\Theta(n^2)$, there is no problem in having cases whose running time grows more slowly than n^2 . Likewise, we cannot correctly say that insertion sort’s running time is $\Theta(n)$, but we can say that its running time is $\Omega(n)$.

Finally, to really drive this into the ground, and repeat from above, the answer to “What is Insertion Sort’s **worst-case** running time?” would be $\theta(n^2)$.