Big Oh, Reading

The proportionality formulae we saw in the previous reading are more to the point in terms of expressing how algorithms scale. But they are hard to write, with that stylized mathy f and the squiggly tilde. We need a better way to write this.

Plus there are considerations of best cases and worst cases and what we'll now have to call an average case, and other similar qualifications. But the most popular way to write proportionality is "big oh". For example, we would write **$O(n^2)$** to express proportionality to n-squared in an average case. Yes, there are big and little ohs and thetas and omegas, but we're not going to get into their subtitles in this course. In a later course after you are familiar with big oh, the variations will make a bit more sense.

Things That Are $O(n^2)$

- Nested for-loop sort of an array or linked list
- "Insertion sort" of a randomly ordered array or linked list
- "Bubble sort" of an array or linked list
- Using an array or linked list for tracking for unique values

$O(n^2)$ is also referred to as **"quadratic timing complexity"**.

Things That Are O(n)

- successful search of a randomly arranged array or linked list
- unsuccessful search of a randomly arranged array or linked list
- counting the number of nodes in a linked list
- finding the last node in a linked list
- queue pop operation when using an array and shifting values
- queue pop operation when removing from the tail of a singly-linked list

O(n) is also referred to as **linear** timing complexity.

Things That Are O(1)

- stack push operation when adding to the end of an array or head of a linked list
- stack pop operation when removing from the end of an array or head of a linked list
- queue push operation when adding to the end of an array or head of a linked list
- queue pop operation when removing from the head of a singly-linked list
- opening a data file for input

O(1) is also referred to as **constant** timing complexity.

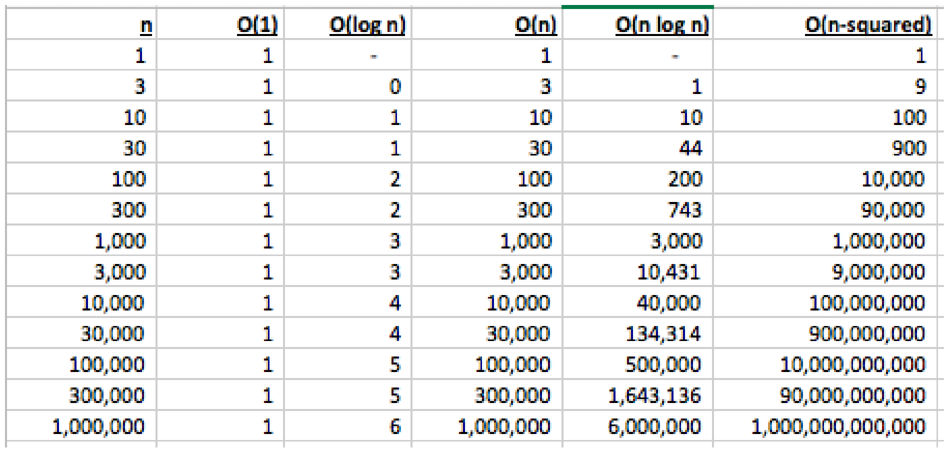
Logarithmic Timing Complexity

Another common big oh is **O(log n)**, called logarithmic timing complexity. It's associated with binary search of an ordered array, as we'll see in a later module of this course. It is between O(1) and O(n), and for very large n, behaves very closely with O(1). So it is very "fast" in big oh terms.

Don't worry too much about the meaning of "log" -- is it base 2 or base e or base 10 or what. Because it does not matter. Remember that $\log_a(x)$ equals $\log_b(x)$ times a *constant*, and we already toss out constants in big oh, so one more constant multiplier makes no difference. Check out **wikipedia.org (https://en.wikipedia.org/wiki/Logarithm#Change_of_base)** for the math, if you wish to know the details, but just remember that the *logarithmic behavior* is what's important.

Loglinear Timing Complexity

Also known as linearithmic timing complexity, **O(n log n)** is associated with "divide and conquer" sorting algorithms, like quicksort, mergesort, and heapsort, all of which we'll study in a later module. It is between O(n) and $O(n^2)$, and for very large n, behaves very close to O(n). The following table demonstrates this:



n	O(1)	O(log n)	O(n)	O(n log n)	O(n-squared)
1	1	-	1	-	1
3	1	0	3	1	9
10	1	1	10	10	100
30	1	1	30	44	900
100	1	2	100	200	10,000
300	1	2	300	743	90,000
1,000	1	3	1,000	3,000	1,000,000
3,000	1	3	3,000	10,431	9,000,000
10,000	1	4	10,000	40,000	100,000,000
30,000	1	4	30,000	134,314	900,000,000
100,000	1	5	100,000	500,000	10,000,000,000
300,000	1	5	300,000	1,643,136	90,000,000,000
1,000,000	1	6	1,000,000	6,000,000	1,000,000,000,000

Other Timing Complexities

Those are the most common big ohs, and you need to become familiar enough with them that you can (1) recognize when an algorithm matches one of them just by applying your reasoning, and (2) determine their big oh analytically by deriving and simplifying a **f(n)** formula.

There are other big ohs, and while they won't be on any quizzes or exams in this course, if you wish to learn more about them, check out this link: **wikipedia.org** (https://en.wikipedia.org/wiki/Big_O_notation) .