# Transcript: Big O Intro with Jim Ashe

*The following transcript is a verbatim account of the video or audio file accompanying this transcript.*

Speaker #1 (Narrator):

WGU's IT Audio Series, flexible, portable, profound.

Speaker #2 (Jim Ashe):

Hi. I'm Dr. Jim Ashe, onetime Math, Computer Science and now Capstone course faculty. In this podcast, I want to discuss the Big O notation and the context of C949, C950, and C960, that's data structures and algorithms parts 1 and 2 and Discrete Math 2 respectively. What's up with Big O? That's the worst-case time complexity notation. What the heck with the Haiti's with the neuroglia, is it? It's just a way to categorize code and functions according to their worst-case growth as input gets really big. An upper bound for its worst possible behavior for very very large inputs, a lot like limits in calculus. Categorizing is something we do with, well, everything, not always according to growth, but something we care about. Like cars, we say pickup trucks, minivans, sports cars, according to design and functionality. Or animals, reptiles, mammals, fish according to how they evolved or whatever. I don't know, I'm a mathematician, not a biologist. Of course, for a Big O, we have a mathematically precise way to categorize stuff which you'll find lots about everywhere. But here I'll provide an intuitive explanation which will be really all you need for our WGU courses. We have some code that prints out an in-long list of names, a 100 names a 100 things to do, 200 names 200 things to do, 300 names 300 things to do, etc. As the list gets longer, there's more work to do. You increase N, you increase the workload by the same amount. We can express this as a function, 1 times N. Hence, y code and math functions are often discussed hand-in-hand. Now for everyone's name three times, John, John, John, three times the work. How does it grow? A 100 names then you have 300 things to do, 200 names, then you have 600 things to do. But the way it grows is very similar to one end. It's linear, so we put this job into the same category, ON. ON is a set, but they say somewhat poorly with this the notation, 3N is equal to ON. But what they really mean is that 3N is in the set ON. Three 3N is in ON, 5N is in ON, a 1000N is in ON, one million or whatever N is just ON. What about something like 3N plus 10,000? The code does 10,000 things and then prints the list three times. Compare 3N to 3N plus 10,000. At first, that 10,000 really makes a big difference. For N equals to 100 for example, it's 300 versus 10,300. But with Big O, we're only worried about what happens as N gets really big like limits in calculus. For N equal to a million, it's 300 million versus 300 million and 10,000. Well, that's hardly any difference. Imagine the graphs of these two functions and keep zooming out. Eventually that 10,000 isn't noticed. Eventually the lines are indistinguishable, so we just drop the 10,000 and we put them in the same category. 3N is ON, 3N plus 10,000 is ON. Any set of instructions, let's say we're going to do one thing, or we're going to do 10,000 things, we're going to do 10 million things, those are all called constant time and then are all O1. Because it doesn't grow at all when N gets bigger even or no matter how big it starts out as, it doesn't grow, at least not with respect to N. When looking at bigger, we only consider the stuff that has the most impact as N grows. What about N squared? It goes for N equal to 2 to 4, N equal 3 to 9, 4 goes to 16, 10 goes to a 100, 100 to, count my zeros, 10, 000, a thousand goes to a million. It grows a lot faster than N. N squared is a slower category than N. You end up doing more work, that's what your code does. We say it's ON squared. What about 3N squared? 10N squared? Similar to above, they're all just ON squared. We dropped that little stuff. 10N squared plus 3N plus 100,000, 10N squared has a most impact so we drop the little stuff, we dropped the a 100,000, we dropped the 3N. It's just 10N squared, which is just ON squared. It simplifies things, really that's what we like to do. I know it doesn't seem like that at first, but it really does simplify things when you categorize things this way. Just say, oh this code is N squared or polynomial time. They strip down a lot about how you describe something. If you're given a function or equation in C960 Discrete Math 2, look for the most impactful part and just drop the little stuff. What if you're given code like in C949 or C960? You might see that on the assessments, look for loops, whiles and for loops. Remember this is the worst-case scenario, so we assume that reachable code runs. To print N names, you might say XN, this list, print X, that's going to be as long as the list, so ON. For a loop embedded in a loop, say X in list. Then for X in list, and this is important, you have to imagine this it's either going to be indented or they're going to put it in parentheses. Let me clear that that second for loop is inside the first one. That's as long as the first list N, and then you do N stuff each time you do one of those Ns, that's N times N or N squared. It's very important that I said it was embedded in there. A loop, in a loop, in a loop, that's N cubed. Though it might get a little cute and put a function inside a loop. Say a for loop of some function called F inside, and you're told as worst-case time complexity is O log N. They'll say F is O log N. The loop does N things in each time you do a log N, so that's N times log N. The Big O is N log N. Logs are inverses of exponents. For example, 10^3 is a 1,000, so log base 10 of 1,000 is equal to 3. That's for base 10. I didn't mention the base before because they don't matter for Big O when you're dealing with logs. Just as 3N and 300N are both in ON with the logs, you just leave the base off and you just see O log N. It's easier to think of logs using base 10 or two. For example, log base 2 of 16 is 4, since 2^4 is 2, 4, 8, 16. I counted four, right? It's having four steps to have 16 down to 1. You go with 16, you would go 8, 4, 2, then 1, that's four steps, which is why you see log N associated with sorting algorithms which split list like merge sort for example. It's fast, faster than ON, which is mostly what you need to know about log N. To sum things up, for C949, C960 and C950, you'll need to determine the Big O of some given code. C950, it'll be your code. For this you just ignore the little stuff. A set of instructions that always runs regardless of your size of N is O1, constant time. A loop is ON for our while. A loop inside a loop is N squared, a loop inside a loop inside a loop is N cubed. There you go N cubed. For a given function, as you might see in C960, identify the biggest, lowest, most impactful part and ignore the rest. You'll be able to organize from fast to slow, left to right respectively. O1 is really fast. This is which probably always going to drop. O log N and then ON and then ON squared, then ON cubed. Getting slower we have the O2^N, N is N the exponent at that point. Then of course, a very very slow, N factorial. For C949, you'll also need to memorize the Big O for the major sorting algorithms. For example, what's the Big O bubblesort? ON squared. Well, you just need to know that then you spin your brain. Write those down and memorize that. C960 also did similar stuff with big Omega, the best case, the lower bound and big Theta, which is between the lower bound and the upper bound of the Big O. That wraps it up. I hope that helps set a foundation for understanding Big O, the context of our courses. If you have any questions of course, your course instructor is your best resource for more details and practice materials. Thank you for listening.

Speaker #1 (Narrator):

WGU, a new kind of view.