[Music]

thank you hello everyone and welcome to an

excellent episode of Dr Dave's data structures modules

for this module that I am creating primarily for my students

that are through the institutions with which I work but hopefully it will help

others as well so this is on Big O and asymptotic

Analysis it's just kind of a short rundown I've provided the

document for you as well so we're just going to go through this

briefly and hopefully it makes some sense and maybe I can clarify some stuff if it's not

uh clear by just reading it so first things first the question is what is

efficiency so the efficiency of an algorithm is the time that it takes to run the algorithm and the amount of

memory that the algorithm causes the program to take up okay so these are our

two major measures of efficiency it would be time and the amount of memory

specifically we're talking about random access memory or Ram so that's what we call the volatile memory the memory that

basically whatever's in it is what's being used by the current programs and if the power goes out typically whatever

was in Ram is just erased so respectively these are called the

complexities are called the time complexity and space complexity of the algorithm for our purposes we're going

to focus on time complexity as it's easier to analyze in order to determine the efficiency of an algorithm you may

be tempted to write the code and run it and then time it for a few different runs to try to see well how fast is this

code or maybe you write multiple examples of the code multiple algorithms that are supposed to

solve the same problem to see which one's the best and that's fine and dandy except for the fact that that could take

up quite a bit of time just to come up with the code and to run it and

everything like that so you have to implement it this type of analysis is valid and it's called

runtime analysis but it has these drawbacks so I briefly mentioned one of them but if you run it on your computer

and then someone else's who has a noticeably higher memory or lower memory and or higher or lower CPU speed amongst

other Hardware factors and also what other programs do they have running um You probably won't even be in the

same ballpark as far as how much time these algorithms are taking if you run the algorithm on the same

computer at different times you could even get significantly different running times because of other programs and

other factors on the same system and of course in order to run an algorithm you have to have implemented it so that

takes a tremendous amount of time as well and if there are many competing algorithms you may find that you have to

implement several to determine which is the fastest so it seems like there should be some sort of motivation for

finding a technique that we can analyze the efficiency of an algorithm before we take the time to actually implement it

and also that it should be independent of a particular device or system or time

that you're running it okay or when you're when you're running it is what I mean in that respect as far as time's

concerned and this is where we get Big O so the most popular technique for

Big O

analyzing the complexity of algorithms is to use Big O analysis also called asymptotic analysis you can refer to the

book now if you're not coming from one of my classes you may not know which book I'm talking about but pretty much every book on data structures and

algorithm is going to have some sort of information on Big O

we will discuss the more practical and calculation based features that will appear on tests or assignments in my

classes and may occur in yours even if I'm not your instructor in order to perform a big O analysis

you should know that the following are the most popular Big O complexity categories

in order from most efficient which means the least complex because we're dealing with what we call complexity to least

efficient which are the most complex or that take the most time given the same size of input okay

um this whole thing is different from what we would call computability computability is the question in

computer science as to whether an algorithm can even solve a problem or not and there are situations where

computability uh where a problem is just simply not computable it's not decidable

um examples you might have if you're taking a more advanced maybe discrete structures discrete mathematics course

or theory of computation if you're joining us from there they will discuss are there certain algorithms that a

computer cannot solve and the answer is yes there are one of the most famous um is the uh the halting problem but

we're not covering that in this lecture this lecture is not dealing with computability we're

assuming that whatever we're dealing with is computable but the computability has to do with theoretical and reality

but complexity has to do with is it practical even if it is computable then

we ask the question is it practical or how how long is it going to take to compute something

so we can think of each of these as a set of algorithms not a single algorithm

or a single solution and I'm using the term set specifically okay so you may have brief understanding

of set theory or maybe you've never touched it before but a set at least a basic notion of a set is that it's it is

itself actually a data structure but it's also a structure that can be used in mathematics and in this

um context what we're dealing with is a maybe a sequence or set or well using

set in the word doesn't help a sequence of items or a collection of items that

don't have any duplicates well in this case it would probably be closer to multi-suts because you could have

duplicates I suppose but you would have this right here Big O of one big O log n

big o v n Etc each of these is considered a set and each set contains different elements just like an array or

some sort of thing you've probably dealt with before some sort of data structure but the question is what are the

elements of these sets and the elements are basically problems and you could even think of the set of algorithms as

well so the algorithms that solve the particular problem are the elements in

this set so different algorithms can be put into one of the above sets

the value of n in any of these is what we call the problem size it's often the size of a collection such as an array

that's being processed in some way Big O of one that is constant times suggests that the time taken for processing with

a particular algorithm is independent of the size of the problem so for example if you have to process 10 elements

versus a million elements the run time of that algorithm if it's a big

O of one will be negligibly different but if it's something like even log in

or n or one of these higher order sets then you would find that the algorithm

actually does take longer to complete so the formal definition of Big O here's

Formal Definition

your mathenese for the evening don't get lost in the sauce here and don't ignore it because this is actually very

important that you understand those so the formal definition of Big O which should clarify the meaning of asymptotic

in the alternative name asymptotic analysis is as follows so we say that a

function f of n now I'm going to read it it's going to sound very math and easy and you may not get it or may not fully

understand it but we'll we'll do examples we'll try to you know work through it and understand what it means

so the formal definition of Big O is that we say that a function f of n okay

we're talking about some sort of function okay um we will talk about what function we're talking about don't think

necessarily in terms of just a programming function um think of it as something that is

growing with the input size so the function of time or function of space is usually what we're dealing with

so we say that a function f of n is Big O of G of n sometimes in a book you

might see the equal sign or you might see a similar to sign or a something like

that just to show that the F of n is in this set I don't like the equal sign because it implies that these are two

equal functions and that's not really what it is Big O is a even though it

looks like it's a function call a big O itself is a set of elements so we say that a function f

of n is Big O of G of n okay and that g of n is usually one of these things okay

so it could be one or log n or G of n equals n g of n equals n log n g of n

equals N squared Etc so it's one of those we say that F of n is Big O of G of n if and only if

the following holds so we said that that function that we are interested in finding the

um finding the Big O of f of n its growth rate is less than or equal to

a constant c times G of n for all values of n greater than some

value n naught okay F of n is less than or equal to c times G of n some constant

scaling this thing in the parentheses it could be one and it could just be this like one of these but it could be like

three or five or seven point eight or something like that so then you'd get if it was linear for example you would have

something like 2N or 5n or 7.5 n or what have you that's that c in front of that

n so we're saying it's being scaled somehow and that's going to cause how fast it grows to change as well but it's still

going the major thing that's going to affect it is which one of these for example an N cubed is going to grow way

faster than an N squared eventually as n gets very very big okay

so we're really interested for values as n gets large so n is the variable

problem size and N Sub Zero is the minimum problem size at which the this inequality holds

so this does not uh this basically puts a restriction or a constraint on this definition here that says that there

could be places where your function is actually the same or bigger than whatever this thing is but at some point

at some point the value n is going to get bigger than uh and not okay so the the size of the

input is going to go past this threshold here and then it's never going to look back it's always this functions always

going to be bigger than the one on the left or said the way it's written this one on the left is always going to be

less than or at most it's going to be equal to this c times G of n on this

side so typically we say that the C is some constant that

could be multiplied by this G of N and N Sub Zero is that threshold that minimum

problem size and we would say that the C can be

multiplied by the specific value of the complexity category to make the inequality true so G of n will typically

be one of those categories listed above one of these on the table the G event specifically is the thing in the

parentheses so big O of G of n would be like Big O of one in which case G of n is one or Big O of G of n would be Big O

of n log n so we would call it linear logarithmic or linear rhythmic is another term for it

um in which case G of n would be n log n or this one over here G of n would be 2 to the power n that's exponential

asymptotic Definition

so don't get lost in the mathenese of the inequality above all it says is your algorithm is represented by a function f

of N and the algorithm is Big O of G of n this is a definition does it belong to that set called a big O of geophen if

and only if it is bound from above by some constant c times that function G of

n so an important note is that we aren't concerned with all values of n just as n

becomes large so there may be values for n for which this inequality is not true as I

mentioned before but once it becomes true while n increases it stays true so this is where

the asymptotic part of the analysis comes from intuitively you can usually just go with

the largest growing term in an expression for the Big O so the following Expressions represent

different algorithms are true okay so are the following Expressions

representing different algorithms sorry so right here if I see 3x 3n squared

plus 5n plus 7. I would normally just say that's Big O of N squared because N

squared is the fastest growing term you don't say it's Big O of N squared plus 5n or 3n squared plus five n you

disregard uh constant coefficients and you disregard the Smalling smaller terms

when we're talking about what set does it belong to because relatively speaking these grow a lot uh slower than the

quadratic term in this case okay and down here this is a simple one so Three

N cubed plus 2 is clearly Big O of n cubed because n cubed is the fastest growing term there 2 doesn't grow at all

the ones down here these are all potentially kind of fast growing terms we've got an N squared here and an N to

the fifth and even an N to the tenth there not sure what kind of algorithm would have 15 n to the 10th but we're

just dealing kind of in generalities at this point but this is the largest growing term of all of them so we would

say this is Big O of n to the tenth and notice it isn't one of our common categories but it is possible because

that would be the largest growing term so that's an intuitive notion of it and that part's relatively easy

um however if you're asked to prove or write a proof of the Big O that's a different story for some of you you

might have just got chills up your spine because you hate proofs but don't worry this is a little bit uh easier than

probably what you're used to you can't just say looks good to me in mathematics math is much more rigorous than a lot of

other disciplines proof in a courtroom for example in law paralegal things like that proof in a courtroom is going to be

quite different from proof that's expected in biology or engineering or mathematics right mathematics is we need

a very rigorous proof engineering we really like to see that things work biology you might have some statistical

proofs or also some practical evidence and then proven a courtroom is just beyond a reasonable doubt if you're

if you're in the United States but don't worry even if you don't like those geometric

proofs that you might have done in Geometry or trigonometry or maybe calculus um the you might have done direct proofs

and inductive proofs and things like that the proofs that we have to provide for Big O are actually fairly simple in

most cases so this is our first thing that we need to do okay we're going to be doing a

couple different things uh in the remainder of the slides but for right now this is what we need to do so we've

got to remember the definition of Big O here identifying the parts again F of n is representing the algorithm we're

analyzing that's usually what we're going to see as a t of n because we're dealing with time so we usually use t s

of n would be for space which is our memory that we use up but often it's going to just be T of N and just

remember the T of n is our kind of General function on this side so T of N and F of n mean the same thing in this

case G of n is on the other hand is going to be the complexity category that we saw

before c is some constant that is going to make this G of n true

for example if I had N squared on this side and I

had N squared on that side that would generally be true but there may be a

situation where you have 2N squared on this side and just N squared on this side because that's one of the gfns that

wouldn't be true so I would have to find a constant that causes this on the right side to actually grow faster than my 2N

squared on this side so you might say 2N squared or 3n squared or 15 N squared and then you would find a corresponding

n a rather N Sub naught that makes that true so n is the size of the problem

and then N Sub naught is the point at which the inequality becomes true in other words at some point that c times G

of n will become and stay larger than F of n forever as n increases

Proofs

so for proof problems on tests and homeworks that I give my students at least you will be given the T of N and

you must prove uh what the Big O should be what the big o is okay prove the Big O the proofs are existential proofs

essentially they're like a a type of direct proof and they just provide existence essentially they show that

something exists meaning you just have to find a value for c and a corresponding value for n naught for

which that inequality right here is true and you should also notice that the

solution is not unique for different values of c you could have different values of n naught and that's

totally fine you just have to find a pair and as soon as you find a pair of N

Sub naughts and a c you have provided enough proof so however I have a pretty handy

technique that is fast and works out for all the polynomial types essentially would work for logarithms and other

things too although those might get a little bit trickier sometimes but typically what I'm going to test my own students on are going to involve

polynomials and subpolynomials so when I say within polynomial we're talking polynomials uh and then also they're

linear and even constant could work but constants a little bit more trivial so typically I'm

going to use polynomials like these so how do we solve these so I use

something called the promotion technique this is a technique I don't know that everyone calls it that but this is a

technique that is used by in many different books and

um at least something similar to it would be used so I like it I like to do it

this way so here are the problems the examples we're going to be working with we have one example where we would say an algorithm has a running time

or a rather a complexity time complexity of Three N squared plus four n plus

three another one has two n to the fifth plus three into the second second Three N squared plus five and then 10 n cubed

which is 10 n to the third plus two so these are all possibilities so my

promotion techniques that I like to promote no pun intended is

copy the T of n on the left and the right side of an inequality a less than

or equal to inequality because what I'm trying to do is I'm trying to get it in this form so

it looks like this part I want my function to it'll it'll be whatever it

is okay the T of n is just going to be on this side you don't change that but I want to manipulate something on this side so that 4 some value or for all

values n greater than a certain point this c times G of n is the format that I get on this side so in other words it's

going to be some sort of constant times a function that is used with the Big O

so for this example you could intuitively just look at it and say well it's going to be some sort of constant

times N squared because that's the largest term that's what I want my G of n is going to be N squared so it's going

to be some sort of c times G of N squared and you have to kind of do it in a

relatively rigorous way just so you don't mess up but it's actually fairly simple so copy

Examples

the T of n on the left to the right of the less than or equal to symbol promote each term on the right to the highest

term combine like terms and now you have the format you need from the definition of Big O that is c times G of n so our

first question so here's our first question T of n equals three N squared plus 4N plus 3

and I have my my T of n which is the F of n that's my function I was dealing

with the runtime uh or not run time rather but the complexity function on this side Three N

squared plus four n plus 3 I was given that and I'm trying to prove it by putting it in the form C some constant

times um some sort of G of n now you can again

easily assume that the G of n is going to be N squared so I'm going to try that I'm promoting all of these terms I

basically copy paste this onto the right side but then promote each of the terms to the highest term so the 4N becomes 4

N squared because the N squared is the highest term 3n squared stays the same and then this 3 just dangling on its own

that also becomes an N squared term so it's 3n squared so now the cool thing is

that since they're all like terms now and we can agree that whatever's on this side is in general at some point going

to grow faster than this side we can just combine them by adding up their constant coefficients so we end up with

t are 10 N squared and that's that c times G of n format that I needed so the

C is 10 and then we want to know where is it true

the N Sub naught is pretty simple so it would not be true for zero

um because this side has a trailing constant and even though these would get canceled out you'd end up with three less than or equal to and then all of

this is zero so we don't want that but one is a value at which it would

definitely be true um past then so this actually proves it we would say we we found a 10 which is C

equals ten or found a c rather that C equals 10 and then we also have n naught

equals one and just for a little bit of a visual if

I bring this over here this is on desmos.com it has our calculator here I

have my T of M which is 3n squared plus 4N

um plus three you will notice that that is the red in here

and even though the red is actually as far as the Y is concerned the red is

actually bigger than the I'm calling it h of n here but this is our c times G of n right here so this is our 10 uh N

squared even though this is larger than our 10 N squared

um at the point from zero we usually don't care about the stuff below zero but at zero this is actually higher

because the hours is at um zero our 10 our 10 of 10 times N

squared is that zero but our Three N squared plus four n plus 3 is actually at three so that's actually bigger than

ours so that's why we need to make that statement that at a certain point it overtakes it now where is that point we

said it was one and it does seem to make sense it's kind of hard to see here but if you zoom in pretty far you'll notice

that at some point see that red goes under

that's the T event that we were talking about and then the blue surpasses it and that happens at the point 110 okay so

that's actually the blue ends up being higher than that and you can kind of look here that blue is remaining larger

than that and we'll do so it'll grow faster forever and ever all right so that should just give you a

visual at least so our question two for this example we

have two n to the fifth plus three N squared plus five again using my promotion technique you can put that

same T event on this side that's our F of N and then we want to get it in the form C some constant times G of n so if

you just copy paste this to the other side you're going to end up with 2 into the fifth three into the fifth five into

the fifth and then you can actually just add those together get 10 n to the fifth and everyone's happy right so this again

ends up being C equals 10 and N Sub naught equals one often usually the the N Sub not for

these polynomials if especially if they have a trailing constant they're gonna

almost always have N Sub Zero equals one but you can solve it this is not the

tech only technique you can use if you have another one you prefer or if you want to make a guess and just look at

this and say well I think three n to the fifth is going to be bigger than this and then just find the point at which three n to the fifth is bigger than that

and then you're good uh question three ten n cubed plus two

this one again 10 n cubed plus two less than or equal to 10 n cubed plus two n cubed that's 12n cubed we end up with

our C equaling 12 and the N Sub naught equals zero

um so if there's a trailing constant uh your N Sub naught is pretty much guaranteed to be a one but remember

these are not unique Solutions the next thing that you should know is how would

these functions or how it functions like the above come to be in the first place so I'm giving you functions and asking

what's the Big O but what does it look like when you compare it to code so here's another technique another skill you need as far

Big O from Code

as my tests and potential assignments are concerned but it's good to know in

any case regardless of who your instructors are what institution you're going to this is

determining the Big O from code so in this one I'm not having you do proofs but you may be asked on tester homework

to look at code or pseudo code and or a segment and then determine what's the

Big O the great thing is that you may not fully use the exact techniques I do but still arrive at the same conclusion

now that might seem kind of crazy or a little bit flaky but it's actually pretty cool

Basic Operations

so when determining time even though things like multiplication take longer at the hardware or what we would

consider the lowest level they can be considered the same type of basic operation as addition or subtraction or

greater than or equal to or not equal to or not or and or anything like that so basic

operations so here's an example the numbers for the following in the

right column indicate the number of basic operations with respect to the size of the problem for Simplicity we

assume the problem size is passed in and is the variable n so what I calculated I

would assume this function okay it's like kind of a pseudo code that looks a lot like Java or C plus

um but for this function I take an input of of N and we assume this is our problem

size and the actual passing in of that input is just one operation one basic

operation for this one this line right here now some of you might look at that and say well that's one operation and

that's fine if you want to do that that's fine as long as it's as long as you can make a reasonable case when you

get to the point where you decide what the Big O is but I say it's 2 because I S I go by the fact that this is a

declaration and then another basic operation maybe is an assignment but you might decide that it's just one and

that's fine too it may even be better who knows so y equals 10 that's another two so I'm

doing the same as the above there now this is where it gets a little bit tricky but pay very close attention to what's going on

and you have to think about the loop itself don't worry about what's in it think about the loop so and how many

operations are being performed we're not talking about how large X and Y get we're not worried about that we're talking about how many times this occurs

and then that will affect what's inside but let's worry about this right now so how many times or how many operations

are just independent of the loop would this be now I said it was two so I'm going to go with 2. and how many

times is this executed in the loop and the reality is that this part of the

loop header is only done once the one in the middle we could say is done end times because it has to keep comparing

it every time it goes back up to the top of the loop and then the one over here where it does the increment that is also done n times

so in total for the loop header the loop itself we have 2 plus n plus n so that

could be looked at as two n plus two now what's inside here

we have how many times this operation is done and what this operation is I would

look at this as an operation like this so it's X an assignment is one operation one basic

operation and the addition that has to be performed to get you the value for x to be reassigned is also another basic

operation and then this whole Loop goes n times so I would say it's that two

basic operations each time and you are doing it in time so it's 2

times n after the loop we increment x one more time here

um you might just say oh that's one basic operation now I could say it's two because it's kind of doing x equals X

plus one so you might decide it's two and that's fine too um return X there's another basic

operation I'm going to call it some people might even leave the return statement out that's okay too

the key is that you got especially the loop is very important but that you're going to get something kind of similar

to what I have so the uh explanations here for the header in case that wasn't clear but in

total we add all of these all the way down so these plus that plus that plus that plus that you get one plus two plus

two plus two plus two n plus two n plus one plus one you end up with four n plus eight so that's our T event and then you

just basically you could prove it if I asked you to but I'm just asking you right now what's the Big O informally

justify it and you could say well the 4N is the largest term so I know it's Big O of N and again you don't say Big O A 4N

because we ignore the constant coefficients and you don't say you don't

say it's n plus eight or four n plus eight because the any smaller terms including trailing constants are also

disregarded okay example two with this one again passing the value in

Example 2 Big O

I'm calling one doing an initialization with the Declaration as two

um in this case there's two Loops we have to be a little bit cautious what's going on here so this outer loop this

happens two times and this is I less than n and then this

is I plus plus each of those is n times so there's two of those

um so that's two n plus two now the inner loop independently the inner loop is this happening twice okay so it is

really what's going on in the outer loop the uh 2N right here this is happening the N

Outer Loop and then this thing happens n times this right here happens N squared times and then this happens

um N squared times as well okay so you end up with this

um 2N squared plus 2N so this thing down here happens N squared times

and then at the bottom you have this return value of one so why is it the two n plus two times N squared because the

inner loop is two n plus two on its own just like the outer loop was but then

what you do is each iteration of the outer loop it will affect what's going on in the inner

loop it's like the J keeps resetting so this actually happens uh n times this

will happen in squared times that will happen in squared times so you end up with the 2N plus 2 the inner part and

it's done n times because that's how many times the outer loop iterates so this is 2N plus 2 times n so that ends

up being the 2N squared plus 2N okay so the full inter the full in iterations

from the inner loop is what causes this to happen now we would then add up all these terms

here you have one plus or all these values rather one plus two plus two n plus two plus two N squared blah blah

and then I put it in um descending order of the power of the

term and I end up with 3n squared plus four n plus six so we would say that

this is a big O of N squared algorithm so that is our Big O I don't think this

sentence is written very well but it's uh so our Big O is N squared and then

the above is our T of n this is our T of n right there all right so that's pretty much what I

had for you I just wanted to explain the document in case there was any lack of clarity and I hope this was useful

thanks [Music]

oh [Music]

English (auto-generated)