Course: Introduction to Graph Algorithms

Professor: C Pandu Rangan

Department: Computer Science and Engineering

Institute: IISc

Week: 01

Lecture 02 Principles of Algorithms (Part 2)

Namaskara, we continue our discussions on the basic principles of algorithm. By principles, we mean the basic set of concepts and ideas that act as building blocks for the entire body of knowledge. For algorithms we have seen that the notion of computational problem, the input process and output, the notion of algorithm itself and the notions like input size which is addressing the problem of scale and finally the concept of complexity. Complexity means resource utilization. How much resource is consumed by a computational process, that is called the complexity. If the resource is time we refer that as time complexity.

Time complexity is a measure, that is determined by looking at the specification of the algorithm. Time complexity is determined by looking into the specification or description of algorithm. Running time is determined by the execution of the code that is using the algorithm. The computer program is written based on the algorithm that are specified and that physical code when you execute, whatever the physical time it takes is called the running time.

So the code might be written in C, C++, Java, Python, it depends, and also the programmers skill, therefore running time is depending on various other factors as well and the system, the processor and the facilities that are available, operations that are available, effective translation by the compiler. Therefore, while running time is very important and it is a key resource that we would like to cut down or optimize, getting an idea about that in its full form is very complex. So we just look into the process and then identify the operation count. Therefore operation time complexity is a complexity measure and this is simply an integer, positive integer it is a number of operations that you do, therefore this is a simple positive integer. Remember that instruction count is just a positive integer, alright.

Now, operation count, when you run the algorithm, imagine, right, it is a thought experiment, assume that the algorithm works on an input you go through the steps of the algorithm, execute that and until you obtain an output you work out the steps and find out how many instructions, how many times certain instructions are executed, find the

instruction count, okay. Now on which input? there are a lot of inputs, in fact there are a lot of inputs of same size. And instruction count may vary even among the inputs of the same size. Intuitively, you can imagine that simply saying the instruction count for sorting 100 numbers. What is the instruction here? In order to sort, you are not going to take square root, division, multiplication, none of those kind of things.

You are going to do comparison. So, example sorting and instruction is comparison. How many comparisons would you do to sort? Let us say 100 numbers, but then there are several sequences of 100 numbers on which the sorting algorithm should work. You can intuitively feel that, if the sequence of 100 numbers is nearly sorted, with very little work probably you can fix and obtain the sorted sequence. Much less number of comparisons may be needed and you might be able to obtain the output.

But if the sequence is thoroughly scrambled, it is a random order. In order to sort you have to compare lot of elements and reposition them and so on, quite a bit of labour is needed to arrive at a sort. So the input size may be 100 but different inputs may call for different number of operations to be performed to produce the corresponding output. So which inputs instruction count I should take to act as a measure, to act as a time complexity measure? This is where the concept of worst case complexity is introduced. For each input I, let us take an operation, it could be plus, it could be multiplication, it could be comparison, it could be some other operation, that depends on the algebraic system or the abstract data type.

So, you choose an operation and then you find out how many times that operation is performed. Assuming that a particular input is taken, how many operations are performed to produce a corresponding output. You call that as instruction count for I

(IC(I))

There may be several inputs. Consider for example sorting of 100 numbers.

There are actually infinitely many sequences of 100 numbers, because the numbers can be any numbers, any set of number, any number can be there in the sequence, there are 100 elements but the elements can have a lot of options. Therefore there are actually infinitely many sequences of 100 numbers. For each such sequence, there is a corresponding instruction count, for each input there is a corresponding instruction count. What is the maximum in that, what is the maximum value among all the instruction counts of a given size. okay, so worst case complexity is written as, worst case complexity parameterized by n is maximum over all I belong to I_n, what is I_n? I_n is a set of all inputs of size n - instruction count of I.

$$WCC(n) = \max_{I \in I_n} (IC(I))$$

So there are several inputs for each input there is a corresponding instruction count, find a maximum. Okay so here what is I_n, I_n is the set of all inputs of size n. I have defined this for the input size represented by a single parameter. You can extend this notion and idea for inputs are measured by several parameters, 2 or 3 parameters. It will be a function of the input size parameters and this is called, this is function, associating n to the maximum value right it is a function.

$$WCC(n) = \max_{I \in I_n} (IC(I))$$

It is called complexity function or worst case complexity function.

WCC(n)

The complexity function relates an input size to the maximum instruction count, whatever is that number. Every n is associated with another number, so this called the complexity function. You can write it as *f(n)*, the general functional notation. So efficiency of the algorithm is judged by the complexity function, efficiency of an algorithm is judged by its complexity function.

Comparisons are done between two algorithms for their efficiency by comparing their complexity functions. So I have an algorithm A that is taking $7n^2$ plus 9n plus 10 steps

$$A \longrightarrow 7n^2 + 9n + 10$$

and I have algorithm B that is taking 10n plus 23 steps.

$$B \longrightarrow 10n + 23$$

For an input of size n I have two algorithms in my hand and the worst case complexity of A is $7n^2$ plus 9n plus 10 and the worst case complexity of B is 10n plus 23, which algorithm would you choose to implement? you have to write a code based on which algorithm you would produce a code that would run on computers, you could easily see you would lean on B, right? In general we need a mathematical way to determine which algorithm is better. Since we are typically using computers for processing large amount of data, we would like to know which function is better or which algorithm is better by

considering their complexity functions for very large n, okay. So if algorithm A has a complexity function f of n, worst case complexity function f of n, algorithm B has worst case complexity g of n.

A — $7n^2 + 9n + 10$ — $f(n)$

B — $10n + 23$ — $g(n)$

How do I compare f(n) and g(n), especially for large values of n? How do we compare f(n) and g(n) values for large values of n? Okay, because, when I am churning out a large amount of data, which algorithm will take less time when implemented? Okay. This is very important for large values. If you would like to mathematically understand the properties of the function, when the parameter value becomes larger and larger, it is called study of asymptotics of the function. You want to look into the behavior of the function n as n goes to infinity, you would like to compare that with the behavior of the function g(n) as n going to infinity.

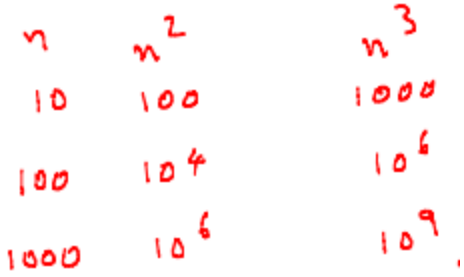
$f(n)$, $n \to \infty$

$g(n)$, $n \to \infty$

So f(n) and g(n), you let n go to infinity and then see which one is going to take larger values, which one will take less values.

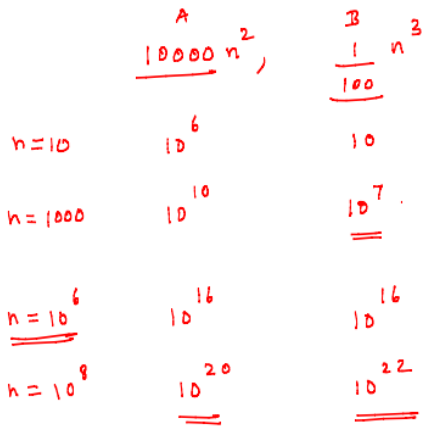
That is how we would like to compare, as n becomes larger and larger, the behavior of f(n) and behavior of g(n) is to be mathematically examined and then we can compare and then we can conclude about which complexity function is a better complexity function, in other words which algorithm is going to be more efficient. The algorithm may appear less efficient for small values of n. We are not going to be unduly worried about that. We are interested in looking into the behavior of these two functions f(n) and g(n) as n goes to infinity, okay. So this is where we are going to look into the asymptotics of complexity functions.

Asymptotics of complexity functions will comment on what is known as growth rate. Asymptotics of complexity functions are helpful in determining growth rate of complexity functions. So, as n becomes larger and larger, how the values of f(n) is growing, how the values of f(n) becomes bigger and bigger, okay. So if you take $n^2$and n^3you can see that when n equal to 10 this is 100, this is 1000. When n equal to 100, this is 10 power 4 and this is 10 power 6, this is million, it is only 10000, whereas that is million when this is 1000, n^2 is 10 power 6 this is 10 power 9.



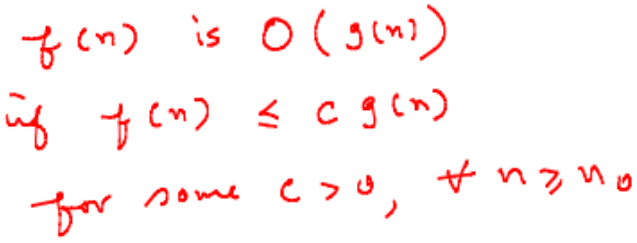
So as n becomes larger and larger, n^2 is also becoming larger and larger, n^3 is also becoming larger and larger. Which one is growing, which one is becoming larger at a more rapid rate between these two? You can see that n^3 is growing faster than n^2. In fact this is the reason why even if we have two complexity functions, one is 10,000 n^2another one is 1 by 100 n^3, even if I have two functions like this, because of the rapid growth of n^3 this value will eventually catch and then move ahead, it will grow faster, it will have bigger values. So, for example, n equal to 10, this is 10 power 4 time, this is 10 power 6, this is 10 power 4 and 10 square is 10, this is 10 cube by 100 is only 10. So for n equal to 10, for example if these are complexity functions, it is a thought experiment okay I am not giving any algorithm I am just telling you that if these are the two complexity functions.

Suppose algorithm A has complexity 10,000 n^2 and algorithm B has complexity 1 by 100 n^3, which algorithm would you choose? Which is taking smaller values? so for n equal to, I have 10 power 6 and for n equal to 1000, this is 10 power 10 and this is 10 power 7. So even up to 1000 you can see that this algorithm is consuming less time, it is taking less number of instructions than the other, but as n becomes larger and larger, we are interested in problem of scale, so as n tends to infinity, when n becomes let us say 10 power 6, you can see the difference. So n^2 will be 10 power 12 plus 4 this will be 10 power 16 and this will be 10 power 18 this is also 10 power 16 so at this point they seem to kind of become equal, but if it is 10 power 8 you can see that 10 power 16 plus this is 10 power 20 whereas this one will be 10 power 22 you can see that this is become larger. So as n becomes larger and larger, n^3 function is going to become much larger than n^2 function, whatever is the constant. The constants actually do not matter.



Asymptotically, that is when n becomes very large, this is going to be more than the other. 1 by 100 n^3 will eventually become larger than 10,000 n^2. That means asymptotically, algorithm A is better than algorithm B. If this is the complexity of algorithm A and if this is the complexity of algorithm B asymptotically. But for smaller values of n for example 100 or something like that, you can see that B is finishing the job with less number of instructions.
B is finishing the job with less number of instruction, A consumes more instruction. But, asymptotically A is going to take less number of instruction than B okay and this is what we are interested. So for small values of n, for small values of input size, we are not unduly bothered about the differences between them, but as the parameter becomes larger and larger which one will take smaller values and which one will take higher values, I would like to determine this by a comparison. This is where we have the big O notation and theta notations come into picture, okay.

A function f(n) is said to be big O(g(n)) if f(n) is less than or equal to c times g(n) for some c greater than 0 and for all n greater than or equal to n o.



I say this is of order n, so this big O because of this is like upper bound, it is an upper bound, okay, because in my definition I have f(n) less than or equal to c times because constants really do not affect as far as the growth rate of concerned with, as n tends to infinity. I had one function with constant 10,000 another function constant 1 by 100, still 1 by 100 n^3, because of the growth rate, because it is very fast growing, it beats 10000 n^2 at some point and after that it will be always larger. Therefore eventually B becomes,

inefficient A becomes the winner, A becomes more efficient. In order to get a clear idea of this comparison in a mathematical way, we first introduce the notion of big O. So for example 7 n^2 plus 9n plus 10 is big O n^2 why it is big O n^2, simply add these constants 7 plus 9 16, 16 plus 10, 26; you can see that 7 n^2plus 9 n plus 10 is less than or equal to 7 n^2 plus 9 n^2 plus 10 n^2 which is equal to 26 n^2.

$7n^2 + 9n + 10$ is $O(n^2)$.

$7n^2 + 9n + 10 \leq 7n^2 + 9n^2 + 10n^2$

$= 26n^2$.

Since $7n^2$ plus 9n plus 10, is less than or equal to $26n^2$, for all n greater than or equal to 1, $7n^2$ plus 9n plus 10 is big O n^2;

$(7n^2 + 9n + 10) \leq 26n^2, \quad \forall n \geq 1,$

$7n^2 + 9n + 10$ is $O(n^2)$

From the definition therefore you can say that it is of order n^2. $7n^2$ plus 9n plus 10 is of order n^2, big O of n^2, okay. The growth rate of n^2 dominates or bigger than the growth rate of $7n^2$ plus 9n plus 10, less than or equal to, okay, the growth rate. Instead of upper bound, we would like to have a tight upper bound. That is because, you can see that $7n^2$ plus 9n plus 10 is big O of n^4 also, not only it is big O of n^2, it is big O of n^4.

$7n^2 + 9n + 10$ is $O(n^4)$

Very simple, this is because, $7n^2$ plus 9n plus 10 is less than or equal to $7n^4$ plus $9n^4$ plus $10n^4$. Therefore, $7n^2$ plus 9n plus 10 is less than or equal to 26n power 4, this is a constant, this is the threshold.

$$7n^2 + 9n + 10 \leq 7n^4 + 9n^4 + 10n^4$$

$$7n^2 + 9n + 10 \leq 26\, n^4 \; ; \quad n \geq 1$$

Therefore it meets the definition of big O, therefore it is big O of n power 4, because it is an upper bound, this upper bound is a very loose upper bound, okay. What we are interested in is a kind of a exact or matching or a tight upper bound, that is given by theta notation. Theta notation tells you that both of them have got the same growth rate, in other words I have f(n) is theta g(n) if some c1 times g(n) is less than or equal to f(n) less than or equal to some c2 times g(n) for c1, c2 greater than 0 and for all n greater than or equal to n0. Beyond some point this will happen, that means it is lower bound and upper bound, which means it is same, the growth rate is same.

$f(n)$ is $\Theta(g(n))$ if

$$c_1 g(n) \leq f(n) \leq c_2 g(n).$$

for $c_1, c_2 > 0$. for $\forall\, n \geq n_0$

For example, we have $3n^2$ is less than or equal to $7n^2$ plus 9n plus 10, which is less than or equal to $26n^2$, both on lower bound and upper bound.

$$3n^2 \leq 7n^2 + 9n + 10 \leq 26n^2$$

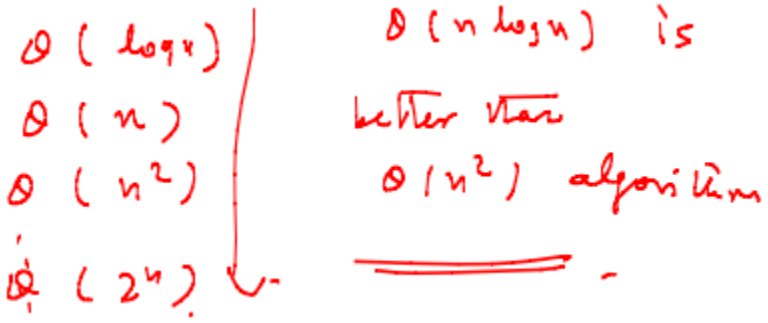
It is sandwiched between, they have the same growth rate. $7n^2$ plus 9n plus 10 is theta n^2, the growth rate is same as n^2.

$$7n^2 + 9n + 10 \text{ is } \Theta(n^2)$$

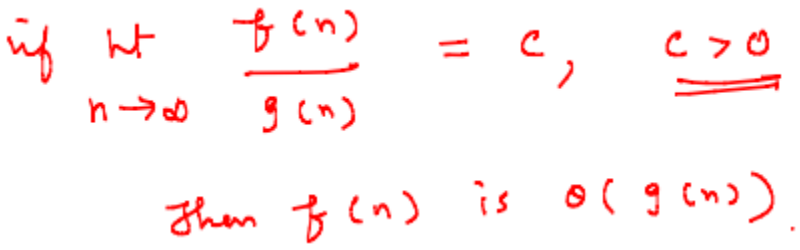
$n^4$and all have higher growth rate, it is an upper bound, but this is a tight bound. When I say f(n) is theta g(n) that means the growth rate g and f are same, they are the same growth rate.

$f(n)$ is $\Theta(g(n))$

In the earlier case one has got a higher growth rate than the other. We always prefer an algorithm whose complexity function, whose worst case complexity has as low a growth rate as possible. Okay, so if you have a function, it could be 7 on log n plus 10, they all have the growth rate theta log n. You may have certain algorithm whose complexity is theta n, some algorithms with complexity n^2 and so on and theta 2 power n. Now theta n log n algorithm is better than theta n^2 algorithm.



Because n log n has got smaller growth rate than n^2. We can determine the growth rates of, we can compare the growth rates by using the inequalities in the definition or you can use calculus. If limit n tending to infinity of f(n) by g(n) is a constant and c is greater 0 than f(n) is theta g(n).



If limit n tending to infinity of g(n) by f(n) is either c greater than, c that is greater than or equal to 0 right, and it could be even infinity. You can write this in terms of f(n) by g(n) and it should go to 0, okay, always.

$$\text{if} \quad \lim_{n \to \infty} \frac{f(n)}{g(n)} = c, \quad c \geq 0$$

We will write in terms of that. c could be here you can see that c must be definitely greater than 0. Here c could be even 0, in this case f(n), so g(n) has got same or higher growth rate okay.

$$f(n) \text{ is } O(g(n)).$$

This basic definition is enough for us to compare two algorithms, find out which one has got poor growth rate. We can compare the complexity functions and then determine which one is more efficient and which one is less efficient. The principles and mathematics of asymptotics is used to compare two algorithms, okay. These are the basic building blocks and form the basic set of principles of algorithms. I stop at this point, we continue our discussions in our next session, thank you.