VLSI Physical Design with Timing Analysis

Dr. Bishnu Prasad Das

Department of Electronics and Communication Engineering

Indian Institute of Technology, Roorkee

Week 01

Lecture 03

Complexity Analysis for Algorithms

Welcome to the VLSI Physical Design with Timing Analysis course. In this lecture, we will discuss about Complexity Analysis of Algorithms. The content of this lecture is that first of all we will discuss about the algorithms and then we will discuss data structure which is needed to implement algorithms, then complexity analysis through asymptotic notations will also be explained. Then finally, we will discuss about NP-hardness with two different category of NP-hardness like polynomial time algorithms and NP algorithms. So first we discuss about the algorithms. So algorithms is basically it is try to solve a problem. So let us say if you have a problem, any problem statement let us say we do a linear search or sorting. So, any problem, so we write some step of instructions to solve that problem. So that finite state of instruction is called the algorithm. So, it should give a output after a stipulated time frame. Whenever you give some input to that algorithm, it will give a desired output for which purpose we have written that algorithm. So, there are several algorithms are there. One is called greedy algorithm. The greedy algorithm tries to find the solution of the problem which is basically locally optimum. It gets the solution in less time. However, it does not go into the global optimum. For example, I will give a example here. Let us say I have a graph like this. So let us say I will start from this point. If I go by greedy algorithm, I will come to this point. So, this is the solution if I go by the greedy algorithm. However actual solution, global solution is remaining here, but we get the solution in less time. So, the advantage of greedy algorithm is that it finds a solution in less time, but it is locally optimum. What is divide and conquer? If you have any problem statement that you divide into subproblems, then those subproblems are disjoint in nature. So, there is no overlap between those subproblems. Then each of the subproblems will solve independently. Then you can combine the solution of each one of them to find the final solution. That is called divide and conquer algorithm.

Then comes the dynamic programming algorithm. In this dynamic programming algorithm, the main problem is divided into subproblems. However, there is an overlap

between the subproblems. So, in that case, the divide and conquer approach is not suitable. Reason being that it does those overlapped problem solution multiple times. But the dynamic programming, so solve those solution and store it in net tabular format and utilize that later whenever it is needed. So dynamic programming solves those type of problem statement in less time. Similarly, integer linear programming that is used in routing algorithms in VLSI physical design. So, let us come to the data structure. So, data structure is basically method of storing, organizing the data such that we can efficiently implement the algorithm properly. So, here we need to basically how we can access the data properly and modify the data properly. So basic data structures whatever we use in day-to-day implementations are stack, link list, queue, tree and graph. So those are basically used most often in the implementation of the algorithms. So, here in VLSI physical design, we have different types of little bit different types of data structure which uses the same framework, but the implementation when you go for VLSI physical design data structures, we use link list of blocks, bin-based method, neighbor pointer and corner stitching. So, these are more advanced data structure basically focusing towards the requirement of VLSI physical design. So, this is more basically important for VLSI physical design implementations.

So, let us discuss why we should discuss about the time and space complexity. Time complexity and space complexity. Time complexity is that how much time it will take to solve any problem. So, in that case, we need to basically find the performance of that algorithm. Let us say I have algorithm A1; I have algorithm A2. So, I need to compare both of them. Which one is better for my problem statement? So, in that case, I need to check what is the time it takes for A1 and what is the time it takes for A2 which takes less time that I use in my implementations. So, it is used for performance evaluation and also, we need to based on that we can choose a particular algorithm. And one more thing is resource management. Resource means space, memory requirement.

Whenever let us say I have a algorithm A1, it takes some memory space M1 and let us say I have A2 which takes memory space M2. So, this M1 and M2 we need to check which take less memory. Because it also sometimes the memory requirement is huge, so space requirement will be more. So basically, to reduce that space complexity problem, we need to also consider the space requirement whenever you are implementing any algorithm. So overall optimization of any kind of problem statement we need to analyze the algorithm in terms of time and space complexity.

So, whenever you basically implement or using an algorithm for a particular problem statement that runtime of the algorithm will depend upon type of input. So, the runtime will change based on the type of the input. Let us say I have one particular input which will pass through all the steps of the algorithm and let us say another input which is going through only one part of the algorithm. So, in that case the type of input will change the runtime of the algorithm. So, it is basically it is not possible to evaluate the runtime of the algorithm depending upon the type of the inputs.

And it will also depends upon the machine also, which machine you are implementing the algorithm. Let us say I have a very fast machine; it will execute the algorithm in faster manner compared to the machine which is slow in nature. So, in that case it is not right way of doing the comparison. Similarly, we can use different programming languages like MATLAB, Python or C or C++ to implement the algorithm. So, this says that the runtime of the algorithm depends upon the type of the input, machine and programming language. So, we need to find out a unique method where we can find the runtime of the algorithm which does not depend upon all these things. So let us take an example here. So, you have a linear search. Here what we are doing, we have a array is there which is X which is having n element and in that array we are finding whether that key is there or not. So, we have a for loop, which is running for n plus 1 time. So, because 0 to n minus 1 is n, however the last time it will go and check whether it is n minus 1 or not that is why it is n plus 1. So, then the line number 2 will run for n times, then the return i will run for the one time. So, then the last return will run for the one time. So, the total complexity, total time to execute the algorithm is 2n plus 3. So total time to execute this algorithm is 2n plus 3. So here what we have to do, we have to remove the constant factors. So, after you remove the constant factor, runtime becomes 2n. Then we need to ignore the coefficients, the coefficients the 2 we have to remove. Now I have n. So, the remaining term is called the order of growth that is n.

So here basically this is order of the growth of this algorithm is order of n. So, this previous algorithm whatever it is there, it is order of n. So then let us take some example. Let us say I have 70n log n. If I have a algorithm which takes 70n log n which we can also express as order of n log n where we are dropping the coefficient 70.

Similarly, here

$$8n^2 + 2n + 10$$

So, in this case the lower order terms we remove and also the coefficient 8 we remove. So, this is order n square. So, this order n square is basically is the complexity of this algorithm. So, whenever you are doing this kind of analysis, we remove the lower order terms and we remove the coefficient in the higher order terms to find the order of the algorithm.

So basically, there is a popular way of analyzing the algorithm which does not depend upon the type of the inputs. It does not depend upon the programming language which we are using to implement that one. It also does not depend upon the machine. It does not depend upon the inputs. It does not depend upon the programming language. It does not depend upon the machine. So, in those that type of analysis is called asymptotic analysis. So, we will discuss about this asymptotic analysis in detail in this lecture. So, we have three different type of asymptotic analysis.

We will discuss one at a time. The first one is the Big-Oh notation. The first asymptotic analysis is called the Big-Oh notation which says about the upper bound on the asymptotic behavior of the function. The Big-Oh denotes the upper bound. The upper bound means that I have a function f of n and this function is given to me. Basically, I need to express in terms of Big-Oh of g of n if there exist two constant c and n0 such that 0 less than equal to f of n less than equal to c of g of n for all n greater than equals to n0. What does it mean? So, you have a function f of n which is plotted here. So, this graph is plotted here, and I have another graph is there c of g of n and I need to find two things. One is c and one is n0. So, these two I need to find out and both are positive constant.

They should not be negative. They should be positive constant such that my f of n is less than equals to c of g of n. Okay. So, this then we can represent this one as Big-Oh of g of n.

So let us take an example here. I have a function f of n. This is a function and I have another function which is g of n which is n equal to n cube. So, we need to find n0 and c such that your f of n is less than equal to this condition should be satisfied. We need to find out c and n0 such that my f of n is bounded by 0 and c of g of n. So, in this case this is my f of n and this I need to this is my f of n and I need to find c of g of n.

Okay. I need to find c and n such that this inequality holds good. So, if you can go by here if I take n0 equals to 1, I will get c as 133. If I n0 equal to 10, I will get 8.62. So, I can say basically f of n is Big-Oh of g of n. If this inequality holds good, I can write f of n is Big-Oh of g of n and g of n is basically Big-Oh of n cube.

So, now we go to the omega notation. This omega notation basically tells about the asymptotic lower bound. We discussed about the asymptotic upper bound. Now we are discussing about the asymptotic lower bound and here I have a function f of n is given to me and I have another function g of n is given to me. I need to find two positive constants c and n0. I need to find two positive constants c and n0 such that my c of g of n is less than f of n for all n greater than equals to n0. So, this inequality should holds good. So, in this case if you can see here, I have the function. This is my function here which is plotted, and your x axis is actually size of the input. This n is input size. This n is input size and y is the value of the function. So, I need to find this n0 above which your f of n is greater than c of g of n. So, then you can write f of n as omega of g of n. So now you take this example here. I have a function f of n here. I have a function g of n here. Now I need to find out two constant which is basically n0 and c and which is this is the constant such that this inequality holds good. So, then I can say my f of n is omega of g of n or omega of n cube.

So now we will go into the third type of asymptotic notation which is theta. Similarly, the theta is my third type of notation. One is called asymptotic tight bound. So, what asymptotic tight bound means that it will be if your f of n is theta of g of n if and only if

my f of n is big O of g of n and f of n is omega of g of n. If both conditions holds good then f of n is the theta of g of n. So, I need to write this one. So, f of n is omega of g of n. I need to find three things here. I need to find three positive constant c1, c2 and n0. All the three are positive such that my this inequality holds good. So then if you can see here after n0 this f of n is bounded by c1 g of n and c2 g of n. So, then f of n is called theta of g of n.

So, we can take this example because we have already proven that your f of n is a big O of g of n and f of n is also omega of g of n. Hence, I can from the same argument I can say that my f of n should be theta of g of n or theta of n cube or we can find out those constant c1, c2 and n0 such that my this inequality holds good. I have this c1 should be 7, c2 is 133 and n0 is 1. So, if this is the case then my f of n is theta of g of n equal to theta of n cube.

So, we discussed about big O notation. We discussed about the omega notation, and we discussed about the theta notation. Big O is my asymptotic upper bound and omega is my asymptotic lower bound and theta is my asymptotic tight bound. So, after discussing all this let us discuss how my different types of complexity of different algorithms, how it grows actually with input size and time your time it requires to execute that one. So here if you can say your input size is x axis is your input size and y axis is your time.

So, first one is O of 1 which is constant time algorithm. This is called constant time algorithm. So, next one is O of log n. This is called logarithmic algorithm, algorithm which is logarithmic in nature whose time complexity is logarithmic in nature. Then you have big O of n which is linear time complexity.

It is the time increases with the n. It is proportional to n. So, it is linear time complexity. This is n log n. So, it is basically takes more time than linear time algorithms n log n. Then you have big O of n square which is or order of n square which is quadratic time complexity.

So, we have basically big O of 1 which takes very less time. Then you have order of log n. Then comes my order of n. Then comes my order of n log n. Then comes my order of n square. So, if I can have multiple different algorithms, so then if I have two different algorithms which is basically takes less time is the constant time algorithm will take less time.

Then comes the big O of log n. Then you have linear time algorithm. Then n log n. Then quadratic complexity algorithm. Then you have last one is basically order of 2 to the power n which is exponential time complexity which is basically order of 2 to the power n. So, this takes very last time exponential time complexity. It grows exponentially with number of input size. So here we just give an example of each one of them. So your O of log n is your binary search. Whenever you are doing binary search it is O of log n and you have

linear search is O of n. Order of n log n is your merge search or insertion search. It takes order of n log n. Then the bubble search will take order of n square. Then you have order of 2 to the power n is your SAT problem and Knapsack problems. So, these two are basically exponential time complexity problem and it takes very long time to solve because your number of input increases and your time will increase by order of 2 to the power n.

So let us discuss some of the algorithms. How it is basically one algorithm how it can be implemented in two different time complexity we can discuss. Here basically we are doing sum of n natural numbers. We are adding n natural numbers. So here the inputs are n integer numbers. Then the output is sum of the first n natural numbers. So, the n is the input to the algorithm and output is the sum of first n natural numbers. So here if you can see the first one is initialization step. It will run for one time. Then the second step is attempt n plus 1 because as I told you it runs for n times but the finally it has to go to check that whether it is i equal to n or not that is why it is n plus 1. Now you have basically a equal to a plus xi, which is we are doing the accumulation operation here which is running for n times. So, and the last return runs for one time. So, the total time it takes is f of n equal to 2n plus 3 which we can which we discussed earlier we drop the basically constant and the coefficients after that f of n is order of g of n.

So, this is basically your complexity of the algorithm. So, this also it is called omega of n and basically theta of n. So now we same problem we can solve in order of one algorithm by doing this basically using this formula which runs for one time. It inputs the number of inputs basically sum of n natural numbers then it will solve this in one equation. It does not require any kind of iterations. So that is why this complexity of this algorithm is basically f of n equal to 2 which is f of n equal to order of 1. f of n is also omega of 1, f of n is equal to theta of 1. Now we will go into another example 2. We are basically looking into the searching algorithm. In this searching algorithm I have input n element basically array with n element and the output is basically based on the key whatever it is given it is there or not we need to check it. So I have this x which is a array and n is the number of element in the array and the key is whether that element is there in the array or not. So let us we have this linear search. Basically, linear search means we need to check that whether the key is 56 is there or not. The first element we checked it is not there. Second element we checked it is not there. Third element we checked it is not there. Fourth, so sixth element we find the key is there. So in this case what I found is that in this case what we found is that we need to search each and every element. So then the time complexity will depend upon worst case will go till the last element in the array. So that is why it is called basically order of n complexity. Complexity of this algorithm is order of n. Then we have basically this is a binary search algorithm. We have a key here 56. I need to check that whether this 56 is there or not. Here there is a condition is there in case of binary search your array should be sorted earlier. But in case of this one in this linear search this is unsorted array. This is unsorted integers actually.

But in case of binary search your array should be sorted. So let us say this is a sorted array with 10 elements and I need to search whether the 56 is there or not. So, I have 10 elements. So I need to check in the middle. The middle is fourth element and whether this fourth element is greater than 56 or lesser than 56. If it is greater than 56 then I need to look into in this section. So, looks that it is 23 and it is 56 is greater than basically 23. So, I need to look into the array in this area. Now I will take the middle of that one which is 7 and here I can find in this one the 56 is greater than 55. So, what I have to go again I need to go in this area. In this area I will check whether my element is there or not. I finally found in 56. So how many checks I need to do here. I just check for three times. So, I check at 4, I check for 7, then I check for 8. So, three checks I can find the element in the array. So this is called order of log n complexity. This algorithm, binary search algorithm complexities order of log n base 2.

In this slide we will discuss about the classes of algorithm. So, there are different classes of algorithm which is used to solve different highly compute intensive problems. So basically, here in this case we have classified into different category. One is P. P means that polynomial time algorithm which can find the solution, we can efficiently find the solution of the problem statement in polynomial time. The NP is basically non-deterministic polynomial time where you cannot find a solution of the algorithm in polynomial time. However, if there is any solution basically if we can guess a solution, we can verify that in polynomial time. So, verifying the solution is basically is takes less time than finding a solution for problem statement. Then you have NP complete which does a benchmarking between the NP hard and NP. So, it is a borderline for the level of difficulty of the problem between your NP and NP hard. So now first we will discuss about the polynomial time algorithm, the P, the problems which is solvable in polynomial time. So, where your complexity is basically order of n to the power k where k is some constant, n is the input size. So, we discussed many algorithm like linear search, then whatever the mod sort, quick sort all are polynomial time algorithms. So here we discuss about that NP that non-deterministic polynomial time algorithm which where we cannot find a solution in polynomial time.

These problems are very hard to solve. And the only way one way to find it is that guess some solution for this problem, and we can verify that the solution is correct from that problem in correct or wrong for that problem statement in polynomial time. The main idea here is that we cannot find a solution for this problem statement if the problem is NP problem and we can guess some solution for that problem statement and that solution we can verify in polynomial time. So, then there is a procedure called reduction. This reduction is what is happening here is that here you have one problem A and there is another problem basically B is there. So basically we find an instance of A and we can transfer or refresh that instance to the instance in B and basically so that the solving instance

of B is also same as solving a instance of A. So let us say I have a instance of problem A and I basically transpose or refresh that problem to another instance of problem B. Then if let us say I find a solution to the problem in the instance of B then that leads to solving the problem A in polynomial time. So, this is called reduction. So, NP-hard algorithms are the algorithms where we have let us say you assume a problem statement X. That problem statement X is basically all the problems in the NP set are reducible to that problem A in polynomial time then that problem is called the NP-hard problem. Then here if you can see the Venn diagram, we have a polynomial time algorithms. We have NP non-deterministic polynomial time algorithm. Basically, this is the set for non-deterministic polynomial time algorithm. This is the non-deterministic polynomial time algorithm. Then we have basically NP-hard and NP-hard then the NP-complete is the intersection of this. So, NP-intersection means a problem X is NP-complete if X means belongs to NP and it also belongs to NP-hard.

So, this set is your NP-complete which is the benchmarking between the NP and NP-hard. So, if you have problems in NP-hard problems how we can solve it? Basically, most optimization problems in VLSI physical designs are NP-hard and we are looking for polynomial time algorithms which basically how we can solve this in polynomial time. So is it a solution is needed even if it is not optimal due to the practical nature of your physical design automation flow. Basically we need to find out a method even if your problem statement is NP-hard how we can find a solution to that problem and we can solve that problem in reasonable time frame such that our complete chip design flow can be done in a stipulated time. So basically, we have lots of problems in the VLSI physical design which is the NP-hard.

We can solve those problems in four different types of algorithms are there to solve these NP-hard problems. One is the exponential algorithms then special case algorithm then approximation algorithm and then heuristic driven algorithms. In case of exponential algorithm we divide the problem into sub problems small problems and which is solvable in polynomial time and after the let us say I have a problem P, I can break that into P1, P2 … Pn then I will find a solution of each one of them S1, S2 … Sn in polynomial time because the problem size is smaller now I can find a solution in polynomial time. Then I can merge all of them using another technique which will not take exponential time. So, then I can find a solution of the problem statement, and which is taking less time. So, this even if you are not getting into the global solution but will come closer to the global solution. Then we have a special case algorithm where we are basically for example your graph coloring for a general graph is NP-complete but in case of VLSI physical design some specific graphs are there where this graph coloring is can be solved in polynomial time. So, when your color graph coloring k equal to 2 when the k is 2 number of color is 2 then that will be solvable in polynomial time.

Then we have approximation algorithm this is very interesting. Actually, solving a problem will take very long time and if we follow that method we cannot send the chip for fabrication. So, what we do is that we do not look for the optimal solution but we can look for a near optimal solution is good enough for our doing the VLSI physical design problem.

So, in that case what you do we have phi and phi star. The phi is a solution of the solution produced by the algorithm and phi star is the solution of the optimal solution of the problem. So, this phi is not the actual solution is one of the solution and phi star is the optimal solution. So, this we define some gamma here phi by phi star where we can basically say that the phi is closer to phi star. For example, I will give you example here let us say we have different types of spice engines are there.

One is called each spice which is more accurate it takes long time. But it will give you accurate result when you have smaller designs it is possible to find the solution in less time. But let us say I have a finesim which basically solve the problem in less time but the solution is not accurate. So, a spice is more time, and the solution is accurate. So, depending upon our requirement if I want to do a very bigger simulation and I want the result to be functionally correct I want the functional evaluation of my design I do not look for the very accurate picosecond delay. So, I can run the fine sim to run that and find the solution in less time and verify my design in less time. So, this is one example of approximate algorithm. Then we have different types of heuristic driven algorithms are there which are used to solve this NP complete problems. Basically, even if you are not able to get globally optimal solutions, you can get the solution in less time. So, it has low time and space complexity, and it produces near optimal solution in realistic time frame and it has a very good average case performance. For example, your simulator handling, we will discuss in future slides is one of the heuristic driven algorithm.

Thank you very much for listening to me.