Lecture Transcript
Running Time of Program: Empirical And Analytical Method

Hi, and welcome to this next lecture on Data Structures and Algorithms. In today's session we'll discuss the concept of running time of a program. Most specifically, we'll introduce you how you could analyze an algorithm based on its running time. We'll consider two notions of running time, one is empirical and that will lead us to the analytical method. So let's take a canonical example and that of search. We'll illustrate through illustrate the canonical example of searching element 'e' in an array 'S'. You know 'e' could be anywhere in this. We will make an assumption that 'S' is sorted and we'll assume that sorting is based on the element. The key value of the elements. So, the running time will really depend on how long the sequence 'S' is. Also, the running time could depend on the position of the example. So how do you characterize this running time as a function of input size and the position of the element? So, to empirically find the running time, you need to do the following one you need to run the program and this is based on the choice of the algorithm. We'll discuss two algorithms for finding such an element.

Now you'll need to run the program corresponding to that algorithm not once but multiple times and that two with a large number of different kinds of inputs. What is the difference in the inputs? Well, we change in input size and position of 'e' will have different inputs. So, these two are the different input parameters we could consider for every run you'll need to compute the time to run the wall clock time and also compute average times across runs. Now, moment I say wall clock time you should be bothered about other interwining factors. So, to do the empirical analysis of running time need to eliminate or more generally normalized all extraneous factors. Now, what are the extraneous factors? It could have background processes running at affect the wall clock times so you should have no background processes running. The other thing is if you have different programs that you want to compare you should be talking about comparable data structures or comparable data types. What does this mean? Well, we will consider int and float to be comparable types. However, an int array is certainly not comparable either of this.

So, we need comparable data types. We'll also assume that implementation is single threaded and we'll also assume that there are no ideal time because of some scheduling problems single threaded and no wasted cycles. Finally, you might want to compare algorithms, implementations of different algorithms so certainly you want to ensure that the algorithms are implemented on programs and that are in comparable languages and this programs are run on operating systems that are comparable. Even after all of this, you would know what kind of system generated interrupts or any other extraneous factors related to temperature and so on might affect the running time of your program. So you need to also average across multiple runs for each program.

One good practice is to run virtual machines which have guaranteed resources. Obviously, you need to run this virtual machines on very high capacity servers. So new profile the running times of different implementations and plot them one against the other you could make some observations of the time complexity.

So we have taken this example of searching what is array? The binary search basically operates on sorted arrays assume the sorted array and as you increase the array size and profile the time required in certain units such as microseconds as a function of increasing array size. You find that an algorithm such as binary search grows very slowly with the size of the array that is linear search as the name says grows linearly. Now in the simple case, is there a way we can talk off one algorithm being always faster than the other. So notice that there are regions lower down here for which you might expect the linear search to be faster than binary search. But how do you know that should be only based on impressum on experiments. You could also plot the time required as a function of the position element being searched for and this, you don't expect to be how very informative but let's take a short nevertheless.

So the element e in the case of linear search as you increase a position it's position of 'e', this is the time. You would expect the time to grow linearly assuming left-right scan to the position. How about binary search? Well, with binary search, if the element happens to be exactly in the middle of 'S' certainly you take very less time. You would then invoke binary search on each end so there would be sometimes some more time required depending on where the element is expected to be. Of course, the time would again grow as you move to either side and so on. So depending on the resolution the length of the list you would expect the very zigzag and so on. I want both to get completed, this is something you can try. Now if you treated the position of the element e as a variable , is there any insight that you might get ? Would it be a good idea to average the performance of an algorithm across multiple positions of elements? Certainly yes. We want to average but there is nothing specific you will get by varying the time as a function of the position. There are some issues with such empirical analysis.

The primary issue is that instead of reasoning about the outcome before the experiment, we seem to be reasoning after experimentation. Is there nothing that we expect from a particular algorithm. In fact, this whole method is too time consuming, reasoning, post hoc reasoning is very time consuming because there is limit to the number of algorithms, there are number of runs of and implementation that you can ever make also there are several hidden factors despite our best efforts to normalize them, for example, the hardware, the compiler etc. Most importantly is that it ignores the essential behavior of the algorithm. Binary search is supposed to be efficient on a sorted array but what does efficient mean? So, you want to capture the essential behavior of the algorithm. However, an empirical analysis might ignore this while focusing on the unnecessary details and this actually no substitute to analysis on paper because you get a foundational understanding of any system based on pen and paperwork. Thus how caveat such analysis you need to do serious modelling on the system. That is what we will cover in this lecture and sub following lectures.

So our model for algorithm analysis is as follows. We are going to assume that we have a processor that sequentially processes instructions and that we have infinite memory to hold any auxiliary data. All basic instructions take one unit of time. This may not be the case and practice. We assume that additions, subtraction, multiplication, division, bit operations, comparison, assignment,

etc. all takes same time. We know from practice that multiplication and division could be a lot more expensive than addition and subtraction. However, you could multiply a number x by 2, by basically shifting bit shifting x to left by 1 bit position. In fact, you can do the same thing for kx. kx is shifting x to left by k positions. So we'll be assume that these are all to be treated on the same scale and we will make these assumptions for both integer and floating point data types. So this process we are ignoring discrete times, memory hierarchies, paging, context switching and so on. So this is really moving away from our messy empirical setup. So let's consider this algorithm A for searching element e in a sequence S. So recall that we are first interested in a linear scan algorithm. So there is a linear scan algorithm, which takes as input element e and sequence S and as the name says its scans every element, checks if the element is the number that you are searching for num here is e.

You found it breaks else it continuous the scan. So any for loop you are making one increment at a time the cost is one. You making array accesses and comparisons simultaneously this overall cost for the comparison is two and then both assignment and break statement will incur a cost of one. So the overall cost will basically depend on where the element is found. So, in the case of successful search, you will keep doing these comparisons and the increments till the element is found and if the element were found at the end of the array will have to do these up for n or n-1 steps. The assignment steps, the last two assignment steps being called exactly once. So what exactly, so summing up the time for successful search will incur a cost of two. This is for the if statement, you make an array access as well as comparison. This will be encoded plus one time. You will also have the increment operating n+1 times. In addition, you have this cost associated with termination the one and the one. So overall you'll find the cost to be 4n+5. How about an unsuccessful search, which means you actually had to go until the end of the sequence S. So in such a case, you never come across the found and break statements. So you'll have to do this until the end, increment the array access and the comparison. So the cost incurred here is basically 4N + 2. So please note the comparison here is between 4 times n+5 and 4 times N+2. So how you compute the average number of instructions executed by program A? And we talk of average, you mean what?

We mean average across all possible values of the position of num or the element e, while holding the length of the list as a constant. So this is the kind of average you can interested in and why is it so? Well, we've already seen that looking at the time as a function of num is not really meaningful. How a looking at the time is the function length is meaningful, so you hold the length constant average access all values of num, change the length and average across all values of num and so on and basically, we expect to see the empirical results but anticipate them analytically and we would expect the linear curve to go up like this as we had seen already. And the logarithmic curve to move very slowly. You would like the binary search to operate like this. But we would like this to be evaluated or validated on a large number of a possible sequence as S with different positions of num in order to get the expected or the average performance of both algorithms.

Thank you.