Data Structures and Algorithms

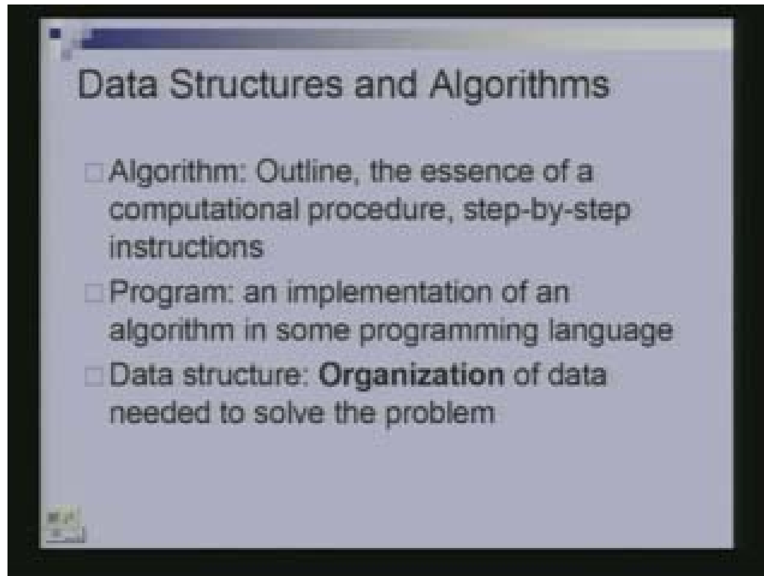
Dr. Naveen Garg
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 1

Introduction to Data Structures and Algorithms

Welcome to data structures and algorithms. We are going to learn about some basic terminologies regarding data structures and the notations that you would be following in the rest of this course. We will begin with some simple definitions. An algorithm is an outline of the steps that a program or any computational procedure has to take. A program on the other hand is an implementation of an algorithm and it could be in any programming language. Data structure is the way we need to organize the data, so that it can be used effectively by the program.

(Refer Slide Time: 1:27)



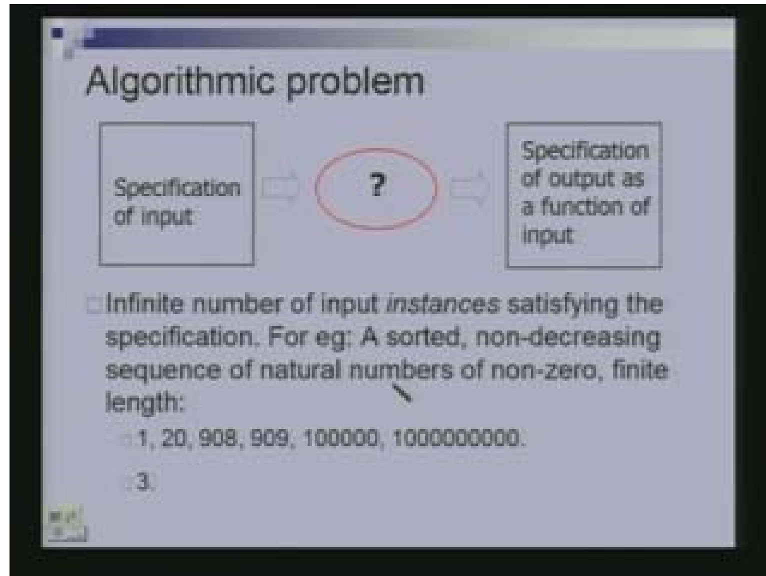
Hope you are all familiar with certain data structures, an array or a list. In this course you will be seeing a lot of data structures and you will see how to use them in various algorithms. We will take a particular problem, try to solve it and in the process develop data structures. The best way of organizing the data, associated with that problem. What is an algorithmic problem? An algorithmic problem is essentially, that you have a certain specifications of an input and specify what the output should be like. Here is one specification. A sorted, non decreasing sequence of natural numbers of non-zero, finite length. For example:

- 1,20,908,909,100000,1000000000
- 3.

This is a completely specified input. Above are the two examples of input, which meets the specification and I have not given any output specification.

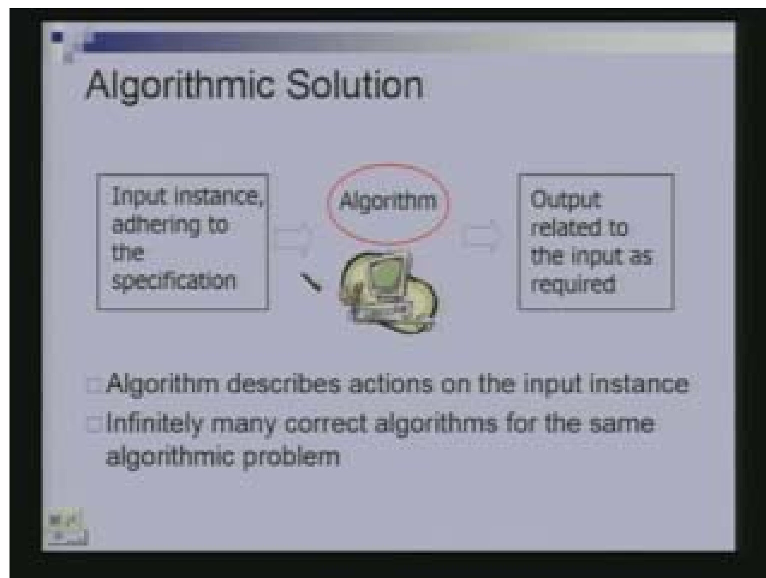
What is an instance? A sorted, non-decreasing sequence of natural numbers of non-zero, finite length forms an instance. Those two examples are the instances of the input. You can have any possible number of instances that may take sequence of sorted, non-decreasing numbers as input.

(Refer Slide Time: 2:29)



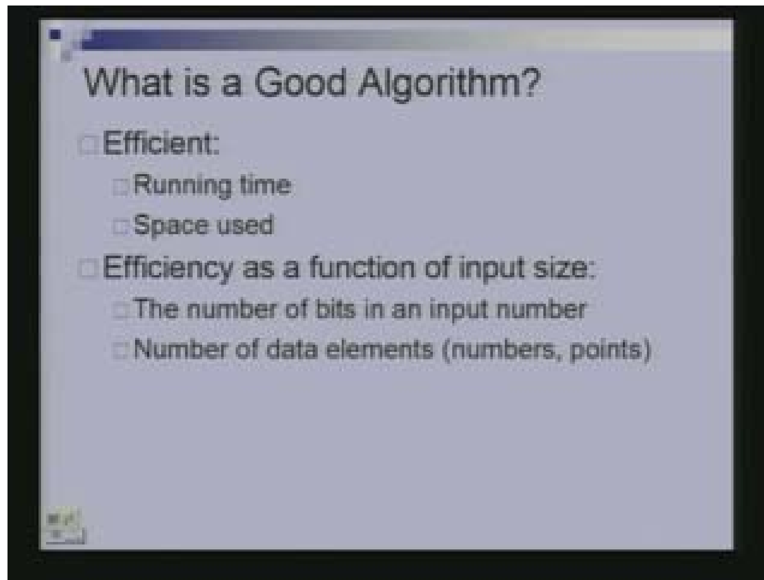
An algorithm is essentially, describing the actions that one should take on the input instance to get the specified output. Also there can be infinitely many input instances and algorithms for solving certain problem. Each one of you could do it in a different way.

(Refer Slide Time: 3:33)



That brings the notion of good algorithm. There are so many different algorithms for solving a certain problem. What is a good algorithm? Good algorithm is an efficient algorithm. What is efficient? Efficient is something, which has small running time and takes less memory. These will be the two measures of efficiency we will be working with. There could also be other measures of efficiency.

(Refer Slide Time: 3:53)



But these are the only two things we will be considering in this course. We would be spending more time on analyzing the running time of an algorithm and we will also spend some time on analyzing the space. We would be interested in the efficiency of algorithms, as a function of input size.

Clearly you can imagine that, if I have a small input and my algorithm or a program running on that input will take less amount of time. If the input becomes 10 times larger, then the time taken by the program may also increase. It may become 10, 20 or 100 times. It is this behavior of increase in the running time, with the increase in the size of input would be of our interest.
Let us see the slide.

(Refer Slide Time: 5:20)

Measuring the Running Time

How should we measure the running time of an algorithm?

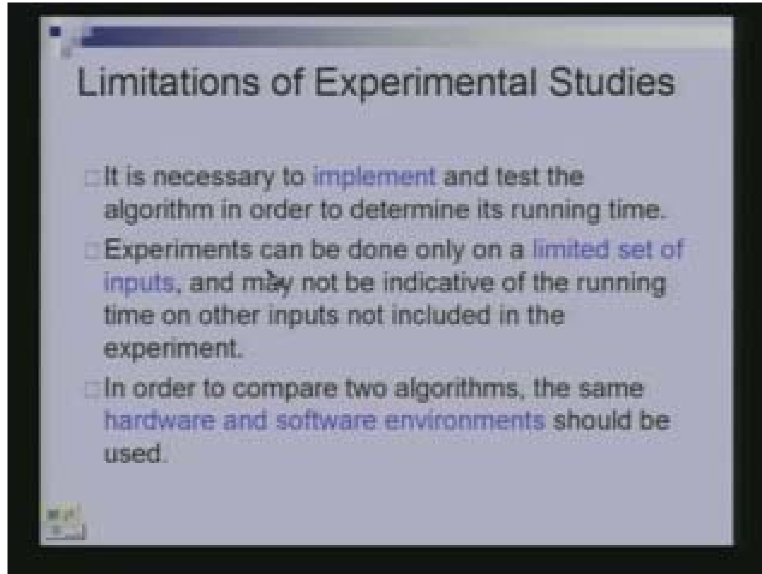
Experimental Study
- Write a program that implements the algorithm
- Run the program with data sets of varying size and composition.
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time.

How does one measure the running time of an algorithm? Let us look at the experimental study. You have a certain algorithm and you have to implement the algorithm, which means you have to write a program in a certain programming language.

You run the program with varying data sets in which some are smaller, some are of larger data sets, some would be of some kinds and some would be of different kinds of varying composition. Then you clock the time the program takes and clock does not mean that you should sit down near stopwatch. Perhaps you can use the system utility like System. Current Time Millis (), to clock the time that program takes and then from that you try to figure out, how good your algorithms is. That is what one would call as the experimental study of the algorithm.

This has certain limitations, let us see them in detail. First you have to implement the algorithm in which we will be able to determine how good your algorithm is. Implementing it is a huge overhead, where you have to spend considerable amount of time. Experiments can be done only on a limited set of inputs. You can run your experiment on a small set of instances and that might not really indicate the time that your algorithm is taking for other inputs, which you have not considered in your experiment.

(Refer Slide Time: 6:23)

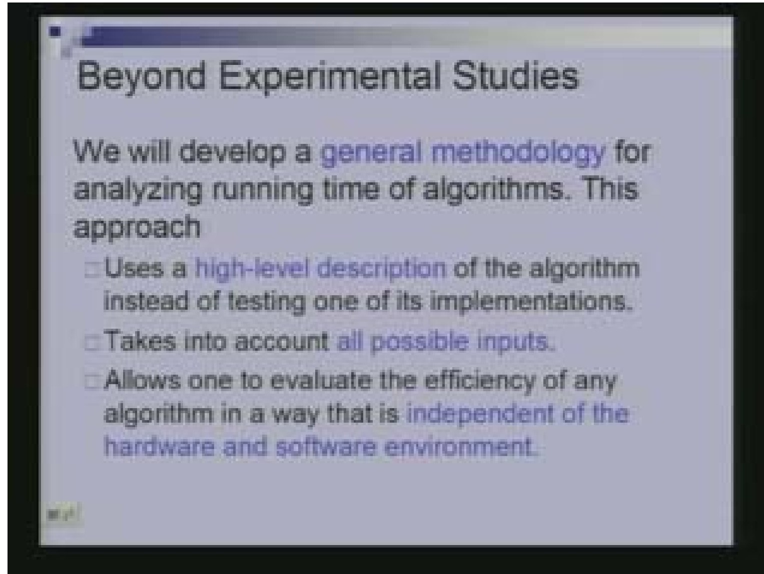


If you have two algorithms and you have to decide, which one is better. You have to use exactly the same platforms to do the comparison. Platform means both the hardware and software environment. Because as you can imagine, different machines would make a difference, in fact even the users who are working on that system at that particular point would make a difference on the running time of an algorithm. It becomes very messy, if you have to do it this way. Hence same hardware and software environments should be used.

What we are going to do in the part of this course? In this very first lecture, we have to develop the general methodology, which will help us to analyze running time of algorithms. We are going to do it as follows: First we are going to develop a high level description of an algorithm. The way of describing an algorithm and we are going to use this description to figure out the running time and not to implement it to any system.

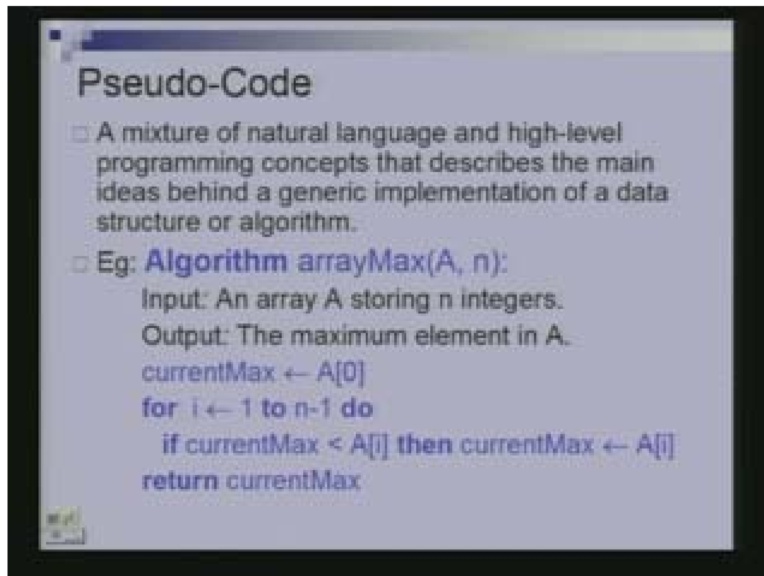
A methodology would help us to take into account of all possible input instances and also it will allow us to evaluate the efficiency of the algorithm in a way that it is independent of the platform we are using.

(Refer Slide Time: 7:38)



Pseudo-code is the high level description of an algorithm and this is how we would be specifying all our algorithms for the purpose of this course.

(Refer Slide Time: 8:23)



Here is an example of pseudo code and you might have seen this in earlier courses also. What is this algorithm doing? This algorithm takes an array A, which stores an integer in it and it is trying to find the maximum element in this array. Algorithm array Max (A, n) The above mentioned example is not a program, because the syntax is wrong. But it is a pseudo code which is a mixture of natural language and some high-level programming concepts.

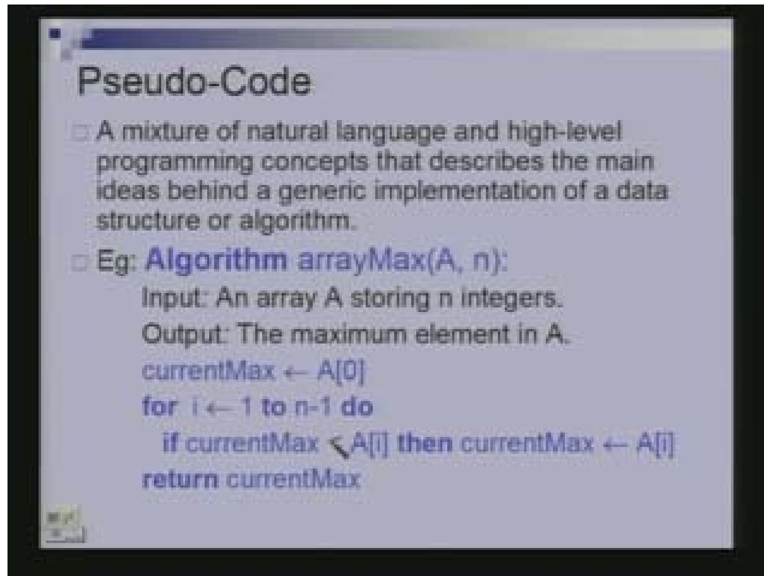
I am going to use a for loop, do loop, if-then-else statement and a while loop. But I will not bother about whether there should be a semicolon or a colon, because they are required for the compiler. But for our understanding, what the program is doing is clear. In the beginning it keeps track of the maximum variable in a variable called current max which is initialized to the first element of the array. Current Max ← A [0] Then it is going to run through the remaining element of the array, compare them with the current maximum element. If the current maximum element is less than the current element, then it would update the current max. A[i] becomes the new max and then when the loop terminates we would just return current max.

If current Max < A[i] then current Max ← A[i]

return current Max

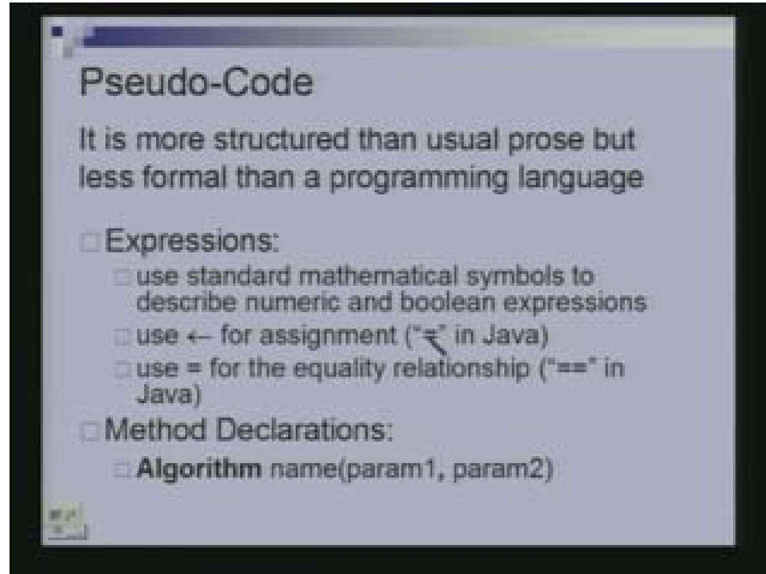
It is a very simple algorithm but just with this pseudo-code, you are able to understand what it is doing. This will not run on any computer since it is the pseudo-code, but it conveys the idea or the concepts.

(Refer Slide Time: 8:48)



Thus pseudo-code is more structured than usual prose, but it is less formal than a programming language. How pseudo-code will look like? We will use standard numeric and boolean expressions in it.

(Refer Slide Time: 10:33)

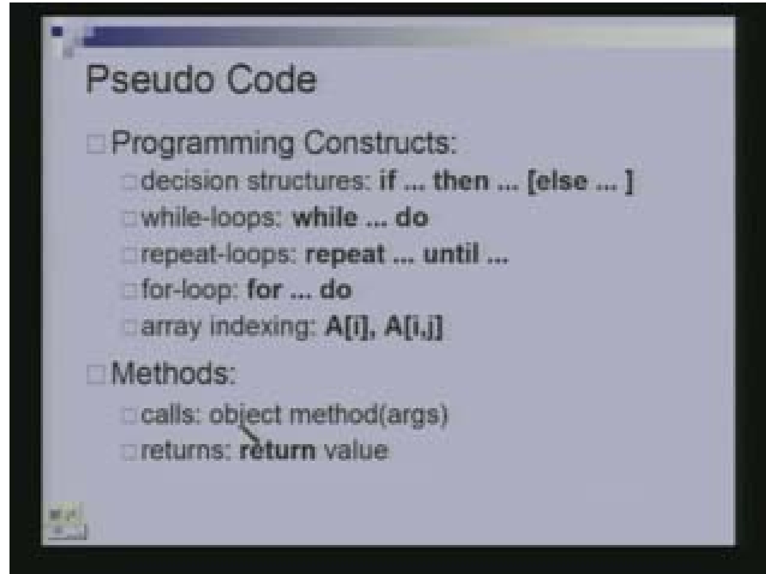


Instead of the assignment operator which is '=' in java, I will use ← and instead of the equality operator, an equality relationship in java which is '= =' the same in C, I will just use '='. I will declare methods with the algorithmic name and the parameter it takes. Algorithm name (param 1, param2)

I will use all kinds of programming construct like if …then statement, if …then… [else] statement, while … do, repeat …until, for … do and to index array I will say A[i], A [i, j]. It should be clear in what it is doing.

I will use return when the procedure terminates and return value will tell about the value returned by the particular procedure or a function. returns: return value When I have to call a method, I will specify that with the name of the method and the argument and the object used. calls: object method (args)

(Refer Slide Time: 11:22)



Object specifies the type of the value returned by the particular method. You will see more of this, when we come across more pseudo-code. How do we analyze algorithms? First we identify what are the primitive operations in our pseudo-code. What is a primitive operation? It is a low level operation. Example is a data movement in which I do an assignment from one to another, I do a control statement which is a branch (if… then …else) subroutine call or return. I do arithmetic operations or logical operations and these are called as a primitive operation.

- Data movement (assign)
- Control (branch, subroutine call, return)
- Arithmetic an logical operations (e.g. addition, comparison)

In my pseudo code, I just inspect the pseudo code and count the number of primitive operations that are executed by an algorithm. Let us see an example of sorting. The input is some sequence of numbers and output is a permutation of the sequence which is in non decreasing order. What are the requirements for the output? It should be in non-decreasing order and it should be the permutation of the input.

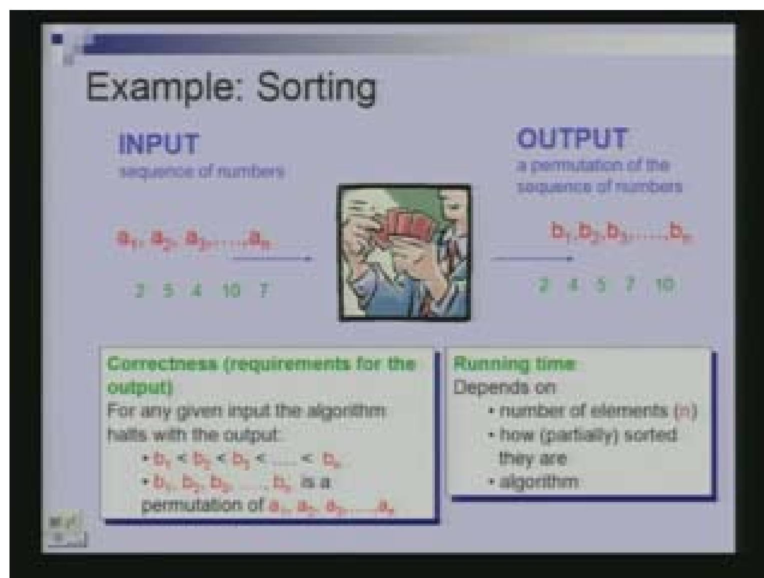
(Refer Slide Time: 12:37)

Analysis of Algorithms

- **Primitive Operation:** Low-level operation independent of programming language. Can be identified in pseudo-code. For eg:
  - Data movement (assign)
  - Control (branch, subroutine call, return)
  - arithmetic an logical operations (e.g. addition, comparison)
- By inspecting the pseudo-code, we can count the number of primitive operations executed by an algorithm.

Any set of numbers which are in non-decreasing order does not make an output. Algorithm should sort the numbers that were given to it and not just produce the sequence of numbers as an increasing order. Clearly the running time depends upon, number of elements (n) and often it depends upon how sorted these numbers are. If they are already in sorted order then the algorithm will not take a long time. It also depends upon the particular algorithm we use. The running time would depend upon all these things. The first sorting technique we use is the one that you have used very often.
Let us say when you are playing game of cards.

(Refer Slide Time: 13:11)



What is the strategy you follow, when you are picking up a set of cards that have been dealt out to you? You like to keep them in a sorted order in your hand. You start with the empty hand and you pick up the first card, then you take the next card and insert it at the appropriate place.

(Refer Slide Time: 14:45)

Suppose if I have some five cards in your hand already, let us say 2, 7, 9, jack and queen. Then I am getting 8, so I am going to put it between 7 and 9. That is the right place it has to be placed in. I am inserting it at the appropriate place and that is why this technique is called insertion sort. I keep on doing this, till I have picked up all the cards and inserted in the appropriate place.

Let us see the pseudo-code for insertion sort. I will give an array of integers as input and output is a permutation of the original numbers, such that it is sorted. The output is also going to be in the same array.

$A[1] \leq A[2] \leq _ \leq A[n]$

This is the input, output specification. I am going to have 2 variables or indices i and j. The array is going to be sorted from a [1] through a [j-1]. The element should be inserted at the jth Location, which is the right place to insert. Clearly j has to vary from 2-n.

For j← 2 to n do

(Refer Slide Time: 15:43)

I am going to look at j^{th} element and I put that in key. Key ← A[j] I have to insert A [j] or the key in to the sorted sequence which is A [1] through A [j-1]. i.e. A [1_j-1] I am going to use the index i to do this. What is index i going to do? Index i is going to run down from j-1 down to 1. We have to decrease index i, which we are doing in the while… do loop.

It starts with the value j-1. I have to insert 7 and i am going to move 9 to 7th location, because 9 is greater than 7. Then I compare 7 with 8 and 8 is still greater than 7, so I will move it right. Then I compare 7 with 6. As 6 is smaller than 7, I would put 7 in the appropriate place.

I run through this loop, till I find an element which is less than a key. Key is the element which I am trying to insert. This loop will continue while the element, which I consider is more than key and this loop will terminate, when I see an element which is less than key or the loop will terminate when I reach i=0. While i >0 and A[i] > key do A [i+1] ← A[i] That means I have moved everything to the right and I should insert the element at the very first place and I am just shifting the element one step to the right. Do A [i+1] ← A[i]

Note that I have to insert 7 at the right place, so I shift 9 right to 1 step. 9th location becomes empty, then I shift 8 to 1 step, so this 8th location becomes empty and now I put 7 there. i + 1 is the index, which would be the empty location eventually and i put the key there. A [i+1] ← key All of you can implement it. May be you would have implemented it in a slightly different way, that would give you a different program, but the algorithm is essentially the same. You are going to find the right place for the element and insert it. Let us analyze this algorithm.

(Refer Slide Time: 19:34)

Analysis of Insertion Sort

	cost	times
for j←2 to n do	c_1	n
key←A[j]	c_2	n-1
Insert A[j] into the sorted sequence A[1..j-1]	0	n-1
i←j-1	c_3	n-1
while i>0 and A[i]>key	c_4	$\sum_{j=2}^{n} t_j$
do A[i+1]←A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i--	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i+1] Å key	c_7	n-1

Total time = $n(c_1+c_2+c_3+c_7) + \sum_{j=2}^{n} t_j \ (c_4+c_5+c_6) - (c_2+c_3+c_5+c_6+c_7)$

I have put down the algorithm on the left. There is a small mistake in the last line of the slide, where there should be a left arrow. Please make a correction on that.

A [i+1] ← A key

Let us count.

Key ← A[j]

I ← j-1

These are all my primitive operations. I am comparing i with 0 and I am comparing A[i] with key, also I take and, so there are three primitive operations.

while i >0 and A[i] > key

Each of the operation takes a certain amount of time, depending upon the computer system you have. $C_1, C_2, C_3, C_4, C_5, C_6$ just represent the amount of time taken for these operations and they can be in any units. I am counting the number of times, each of these operations is executed in this entire program.

Why this operation is done n times? I start by assigning j =2 then assign 3, 4,5,6,7 and go up to n. Then when I increment it once and check that there is one more, so I have counted it as n times. There might be small errors in n and n + 1, but that is not very important. Roughly n times we need to do this operation.

(Refer Slide Time: 19:34)

How about this operation? Key ← A[j] I am going to do exactly n-1 times once for 2, once for3, once for 4 up to n. That is why this operation is being done up to n-1 times. Just leave the comment statement. Again the operation will be done exactly n-1 times.

We have to look at how many times I come to this statement. While I >0 and A[i] > key t_j - Counts the number of times I have to shift an element to the right, when I am inserting the jth card in to my hand. In the previous example when I am inserting 7, I had to shift 2 elements 8 and 9. t_j is going to count that quantity and that is the number of times I am going to reach A[i] part of my while loop. While I >0 and A[i] >key

I will be checking this condition for many times. For one iteration or for the j_{th} iteration of this for loop, I am going to reach this condition for t_j times. The total number of times I am saying that condition is the sum of t_j as j goes from 2 to n.

$$\sum_{j=2}^{n} t_j$$

while I >0 and A[i] > key
do A[i+1] ← A[i]

Every time I see (A[i] >key) condition I also come to A[i], because the last time I see the statement I would exit out of this condition. That is why this is t_j -1 where j going from 2 to n.

$$\sum_{j=2}^{n} (t_{j-1})$$

A [i+1] ← A key. This statement here is not a part of the while loop rather it is a part of the for loop as it is done exactly n-1 times as the other statement. If you knew about the constants then the total time taken by the procedure can be computed. You do not know

what t_j is. t_j is quantity which depends upon your instance and not problem. Problem is in the sorting. The instance is a set or a sequence of numbers that have given to you. Thus t_j depends upon the instance.

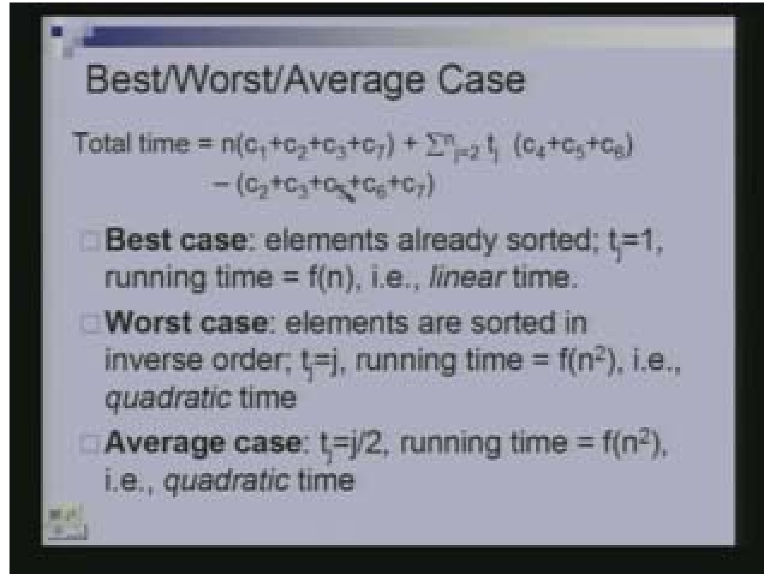
Let us see the difference that t_j makes. If the input was already sorted, then t_j is always 1(t_j=1). I just have to compare the element with the last element and if it is larger than the last element, I would not have to do anything. t_j is always a 1 if the input is already in increasing order.

What happens when the input is in decreasing order? If the input is in decreasing order, then the number that I am trying to insert is going to be smaller than all the numbers that I have sorted in my array. What am I going to do? I am going to compare with the 1st element, 2nd element, 3rd element, 4th element and all the way up to the 1st element. When I am trying to insert the jth element, I am going to end up in comparing with all the other j elements in the array. In that case when t_j is equal to j, note that the quantity becomes its summation of j, where j goes from 2 to n. It is of the kind n^2 and the running time of this algorithm would be some constant time n^2 plus some other constant times n minus some other constant.

$$n(c_1 + c_2 + c_3 + c_7) + \sum_{j=2}^{n} t_j (c_4 + c_5 + c_6) - (c_2 + c_3 + c_5 + c_6 + c_7)$$

Thus the behavior of this running time is more like n^2. We will come to this point later, when we talk about asymptotic analysis but this is what I meant by $f(n^2)$. On the other hand in the best case when t_j=1, the sum is just n or n-1 and in that case the total time is n times some constant plus n-1 times some constant minus some constant which is roughly n times some constant. Hence this is called as linear time algorithm.

(Refer Slide Time: 24:36)



On an average what would you expect? In the best case you have to compare only against one element and in the worst case you have to compare about j elements. In the average case it would compare against half of those elements. Thus it will compare with $\frac{j}{2}$, even when the summation of $\frac{j}{2}$ where j goes from 2 to n, this will be roughly by $\frac{n^2}{4}$ and it behaves like n^2 . This is what I mean by the best, worst and average case. I take the size of input, suppose if I am interested in sorting n numbers and I look at all possible instances of these n numbers.

(Refer Slide Time: 26:47)

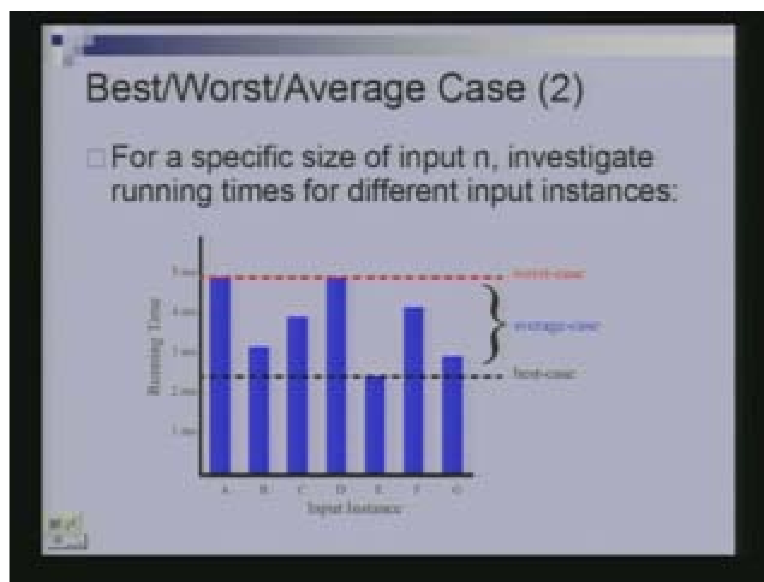
Best/Worst/Average Case

Total time = $n(c_1+c_2+c_3+c_7) + \sum_{j=2}^{n} t_j \, (c_4+c_5+c_6)$

$- (c_2+c_3+c_5+c_6+c_7)$

- **Best case**: elements already sorted; $t_j=1$, running time = f(n), i.e., *linear* time.
- **Worst case**: elements are sorted in inverse order; $t_j=j$, running time = $f(n^2)$, i.e., *quadratic* time
- **Average case**: $t_j=j/2$, running time = $f(n^2)$, i.e., *quadratic* time

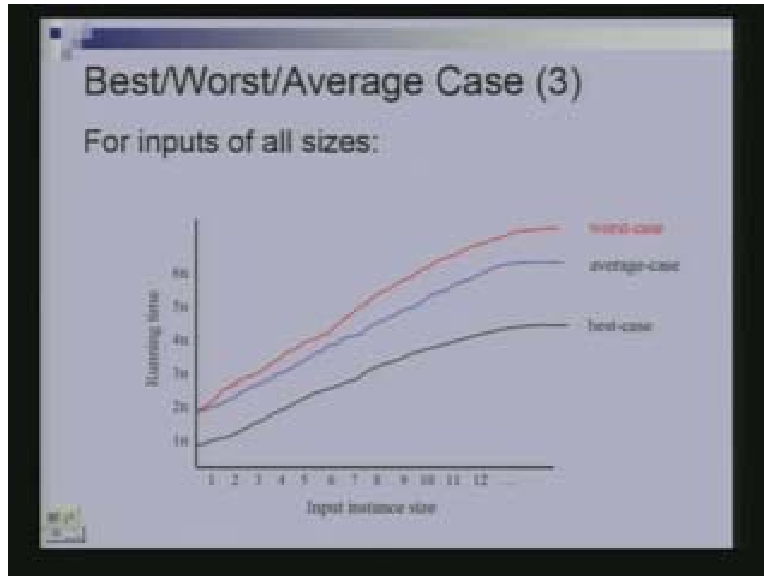
(Refer Slide Time: 27:08)



It may be infinitely many, again it is not clear about how to do that. What is worst case? The worst case is defined as the maximum possible time that your algorithm would take for any instance of that size. In the slide 27:08, all the instances are of the same size. The best case would be the smallest time that your algorithm takes and the average would be the average of all infinite bars. That was for the input for 1size of size n, that would give the values, from that we can compute worst case, best case and the average case. If I would consider inputs of all sizes then I can create a plot for each inputs size and I could figure out the worst case, best case and an average case. Then I would get such a monotonically increasing plots. It is clear that as the size of the input increases, the time

taken by your algorithm will increase. Thus when the input size becomes larger, it will not take lesser time.

(Refer Slide Time: 28:18)



Which of this is the easiest to work with? Worst case is the one we will use the most. For the purpose of this course this is the only measure we will be working with. Why is the worst case used often? First it provides an upper bound and it tells you how long your algorithm is going to take in the worst case.

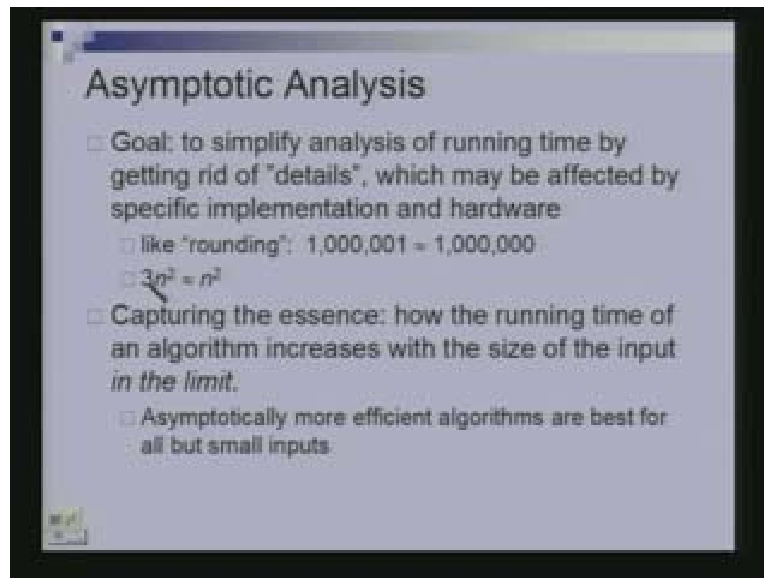
(Refer Slide Time: 28:50)

Best/Worst/Average Case (4)

- **Worst case** is usually used: It is an upper-bound and in certain application domains (e.g., air traffic control, surgery) knowing the **worst-case** time complexity is of crucial importance
- For some algorithms **worst case** occurs fairly often
- **Average case** is often as bad as the **worst case**
- Finding **average case** can be very difficult

For some algorithms worst case occurs fairly often. For many instances the time taken by the algorithm is close to the worst case. Average case essentially becomes as bad as the worst case. In the previous example that we saw, the average case and the worst case were n^2. There were differences in the constant but it was roughly the same. The average case might be very difficult to compute, because you should look at all possible instances and then take some kind of an average. Or you have to say like, when my input instance is drawn from a certain distribution and the expected time my algorithm will take is typically a much harder quantity to work and to compute with.

(Refer Slide Time: 30:36)



The worst case is the measure of interest in which we will be working with. Asymptotic analysis is the kind of thing that we have been doing so far as n and n^2 and the goal of this is to analyze the running time while getting rid of superficial details.

We would like to say that an algorithm, which has the running time of some constant times n^2 squared is the same as an algorithm which has a running time of some other constant times n^2,because this constant is typically something which would be dependent upon the hardware that your using.

$$3n^2 = n^2$$

In the previous example c_1, c_2 and c_3 would depend upon the computer system, the hardware, the compiler and many factors. We are not interested to distinguish between such algorithms. Both of these algorithms, one which has the running time of $3n^2$ and another with running time n^2 have a quadratic behavior. When the input size doubles the running time of both of the algorithm increases four fold.

That is the thing which is of interest to us. We are interested in capturing how the running time of algorithm increases, with the size of the input in the limit. This is the crucial point here and the asymptotic analysis clearly explains about how the running time of this algorithm increases with increase in input size within the limit.

(Refer Slide Time: 32:20)

Let us see about the "big-oh" O-notation. If I have functions $f(n)$, g (n) and n represents the input size. f (n) measures the time taken by that algorithm. f (n) and g (n) are non-negative functions and also non-decreasing, because as the input size increases, the running time taken by the algorithm would also increase. Both of these are non-decreasing functions of n and we say that f (n) is O (g (n)), if there exist constants c and n_0, such that f (n) $\leq$ c times of g (n) $\geq$ n_0.

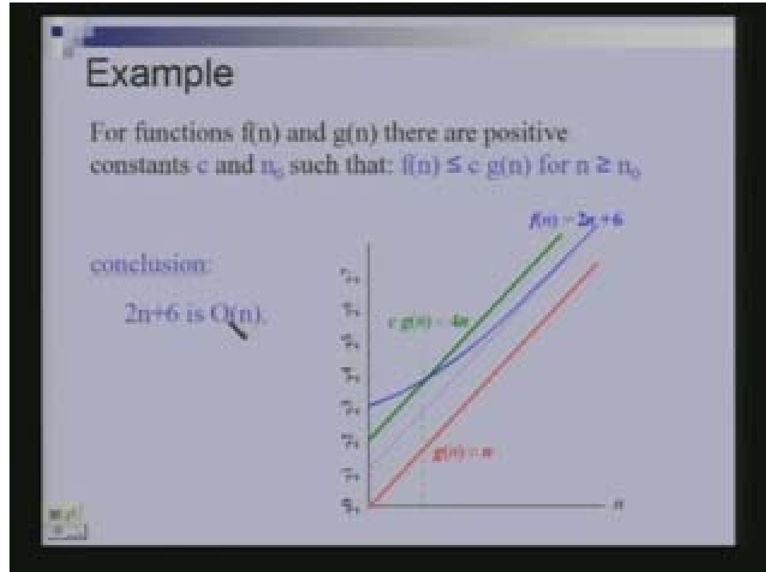
f (n) =O(g(n)

f (n) $\leq$ c g(n) for n $\geq$ n_0

What does it mean? I have drawn two functions. The function in red is f (n) and g (n) is some other function. The function in green is some constant times of g (n). As you can see beyond the point n_0, c (g (n)) is always larger than that of f (n). This is the way it continues even beyond. Then we would say that f (n) is O (g (n) or f (n) is order (g (n)).

f (n) = O (g(n))

(Refer Slide Time: 33:56)



Few examples would clarify this and we will see those examples. The function f (n) =2n+6 and g (n) =n. If you look at these two functions 2n+6 is always larger than n and you might be wondering why this 2n+6 is a non-linear function. That is because the scale here is an exponential scale. The scale increases by 2 on y-axis and similarly on x-axis. The red colored line is n and the blue line is 2n and the above next line is 4n. As you can see beyond the dotted line f (n) is less than 4 times of n. Hence the constant c is 4 and n_0 would be this point of crossing beyond which 4n becomes larger than 2n+6.

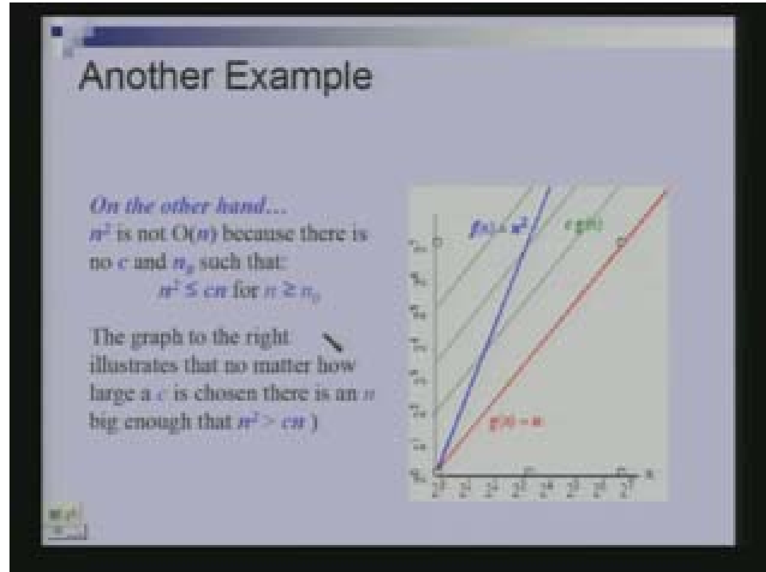
At what point does 4n becomes larger than 2n+6. It is three. So n_0 becomes three. Then we say that f (n) which is 2n+6 is O (n).

2n+6 = O (n)

Let us look at another example. The function in red is g (n) which is n and any constant time g (n) which is as same scale as in the previous slide. Any constant time g (n) will be just the same straight line displaced by suitable amount. The green line will be 4 times n and it depends upon the intercept, but you're n^2 would be like the line which is blue in color. So there is no constant c such that $n^2 <$ c (n).

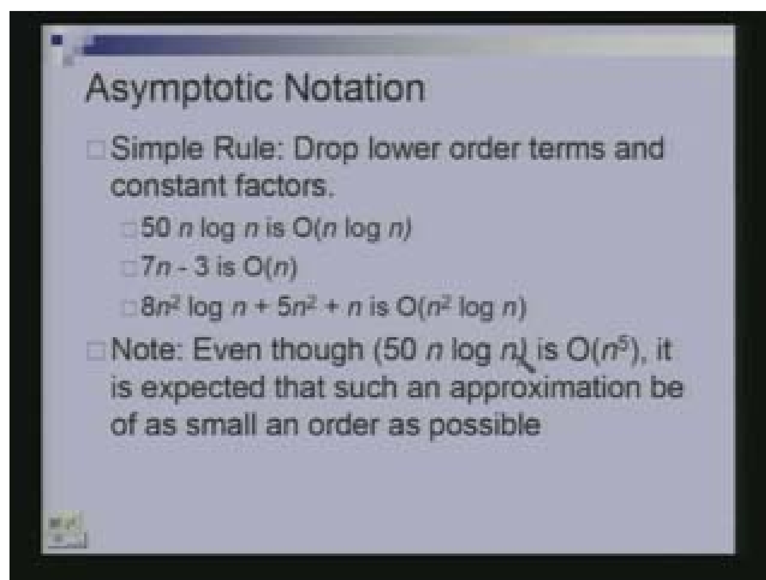
Can you find out a constant c so that $n^2 <$ c (n) for n more than n_0. We cannot find it. Any constant that you choose, I can pick a larger n such that this is violated and so it is not the case that n^2 is O (n).

(Refer Slide Time: 35:55)



How does one figure out these things? This is the very simple rule. Suppose this is my function 50 n log n, I just drop all constants and the lower order terms. Forget the constant 50 and I get n log n. This function 50 n log n is O (n log n). In the function 7n-3, I drop the constant and lower order terms, I get 7n-3 as O (n).

(Refer Slide Time: 37:01)



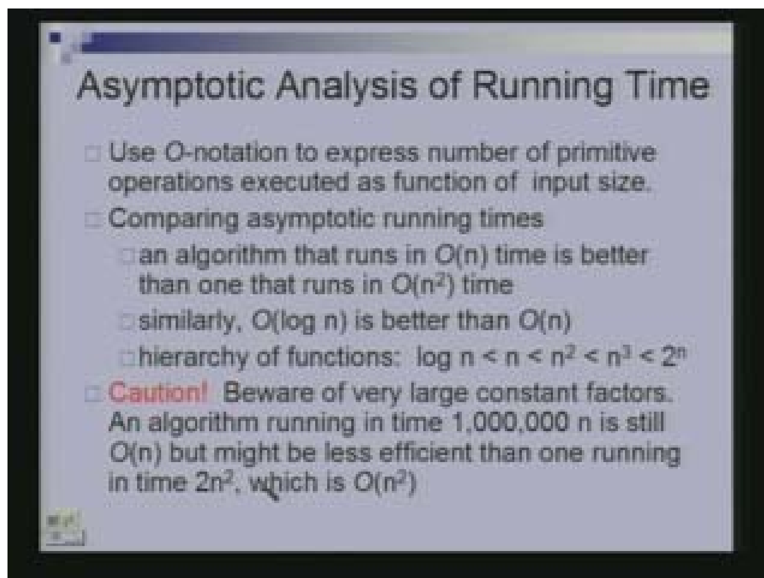
I have some complicated function like $8n^2$ log n+ $5n^2$ +n in which I just drop all lower order terms. This is the fastest growing term because this has n^2 as well as log n in it. I just drop n^2, n term and also I drop my constant and get n^2 log n. This function is O (n^2 log n). In the limit this quantity ($8n^2$ log n+$5n^2$+n) will be less than some constant

times this quantity (O (n^2 log n)). You can figure out what should be the value of c and n_0, for that to happen.

This is a common error. The function 50 n log n is also O (n^5). Whether it is yes or no. It is yes, because this quantity (50 n log n) in fact is ≤ 50 times n^5 always, for all n and that is just a constant so this is O(n^5). But when we use the O-notation we try and provide as strong amount as possible instead of saying this statement is true we will rather call this as O (n log n)). We will see more of this in subsequent slides.

How are we going to use the O-notation? We are going to express the number of primitive operations that are executed during run of the program as a function of the input size. We are going to use O-notation for that. If I have an algorithm which takes the number of primitive operations as O (n) and some other algorithm for which the number of primitive operations is O (n^2). Then clearly the first algorithm is better than the second. Why because as the input size doubles then the running time of the algorithm is also going to double, while the running time of O (n^2) algorithm will increase four fold.

(Refer Slide Time: 39:10)



Similarly our algorithm which has the running time of O (log n) is better than the one which has running time of O (n). Thus we have a hierarchy of functions in the order of log n, n, n^2, n^3, 2^n.

There is a word of caution here. You might have an algorithm whose running time is 1,000,000 n, because you may be doing some other operations. I cannot see how you would create such an algorithm, but you might have an algorithm of this running time. 1,000,000n is O (n), because this is ≤ some constant time n and you might have some other algorithm with the running time of $2n^2$.

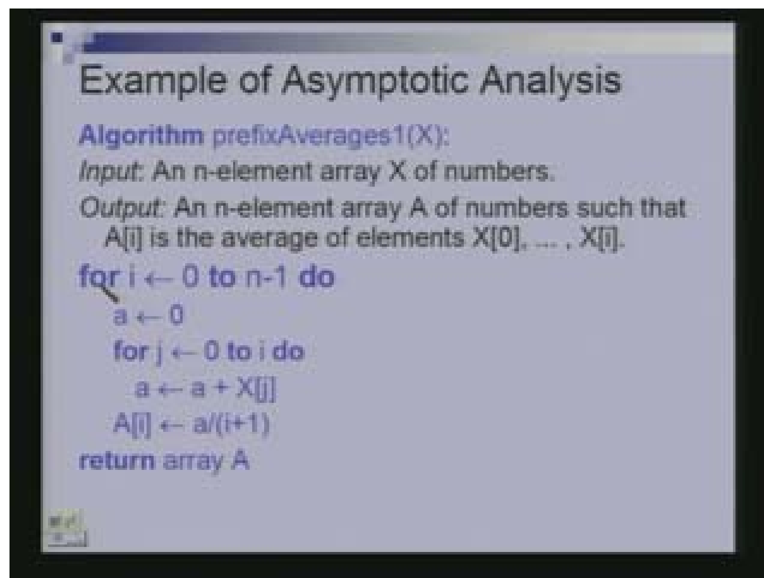
Hence from what I said before, you would say that 1,000,000 n algorithm is better than $2n^2$. The one with the linear running time which is O (n) running time is better than O (n^2). It is true but in the limit and the limit is achieved very late when n is really large. For small instances this $2n^2$ might actually take less amount of time than your 1,000,000 n. You have to be careful about the constants also.

We will do some examples of asymptotic analysis. I have a pseudo code and I have an array of n numbers sitting in an array called x and I have to output an array A, in which the element A[i] is the average of the numbers X [0] through X[i]. One way of doing it is, I basically have a for loop in which I compute each element of the array A. To compute A [10], I just have to sum up X [0] through X [10], which I am doing here.

For j ← 0 to I do
A ← a + X[j]
A[i] ← a/ (i+1)

To compute A [10], i is taking the value 10 and I am running the index j from 0-10. I am summing up the value of X from X [0] - X [10] in this accumulator a and then I am eventually dividing the value of this accumulator with 11, because it is from X [0] to X [10]. That gives me the number I should have in A [10]. I am going to repeat this for 11,12,13,14 and for all the elements.

(Refer Slide Time: 41.34)



It is an algorithm and let us compute the running time. This is one step. It is executed for i number of times and initially i take a value from 0,1,2,3 and all the way up to n-1. This entire thing is done n times. This gives you the total running time of roughly n^2.

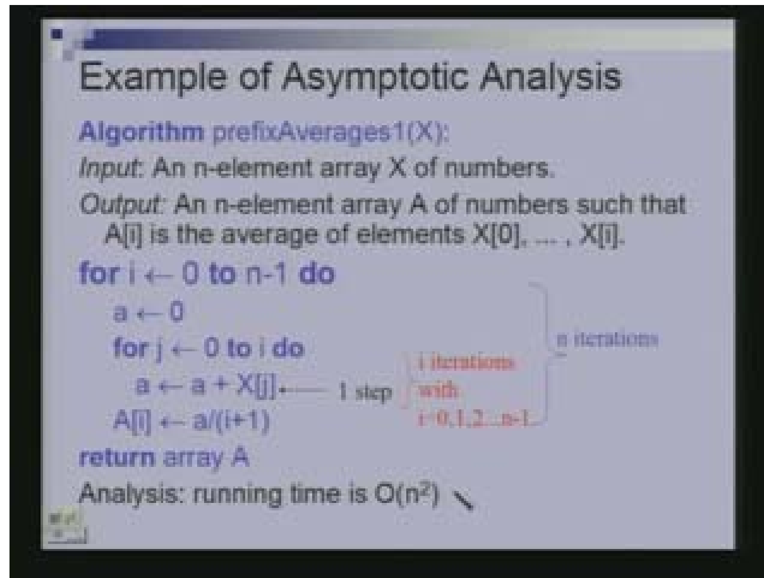
a ← a+ X[j]

This one step is getting executed n^2 times and this is the dominant thing. How many times the steps given below are executed?

A[i] ← a/ (j+1)

a ← 0 These steps are executed for n times. a ← a + X[j] But the step mentioned above is getting executed roughly for some constant n^2 times. Thus the running time of the algorithm is O (n^2). It is a very simple problem but you can have a better solution.

(Refer Slide Time: 43:19)



What is a better solution? We will have a variable S in which we would keep accumulating the X[i]. Initially S=0. When I compute A[i], which I already have in S, X [0] through X [i-1] because they used that at the last step. That is the problem here.

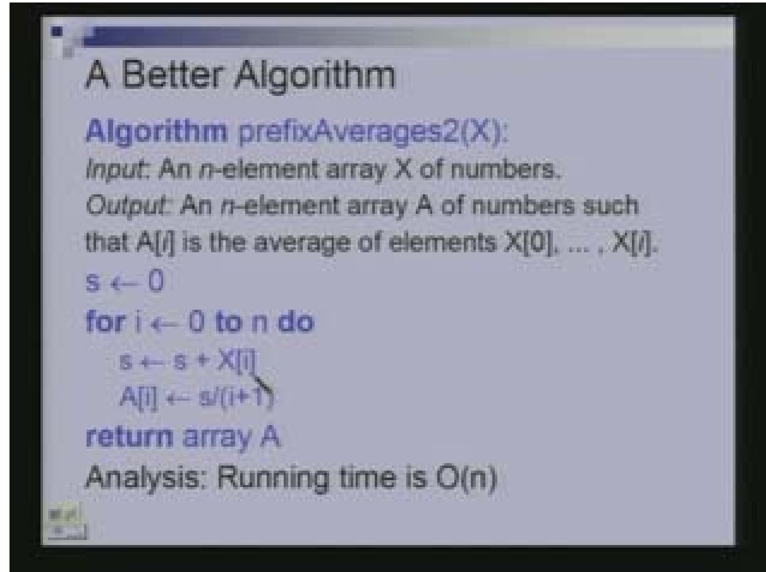
a ← a +X[j]

Every time we are computing X. First we are computing X [0] + X [1], then we are computing X [0] + X [1] +X [2] and goes on. It is a kind of repeating computations. Why should we do that? We will have a single variable which will keep track of the sum of the prefixes. S at this point (s← s+x[i]), when I am in the ith run of this loop has some of X [0] through X [i-1] and then some X[i] in it. To compute i^{th} element, I just need to divide this sum by i +1.

S ← S +X[i]
A[i] ← S/ (i+1)

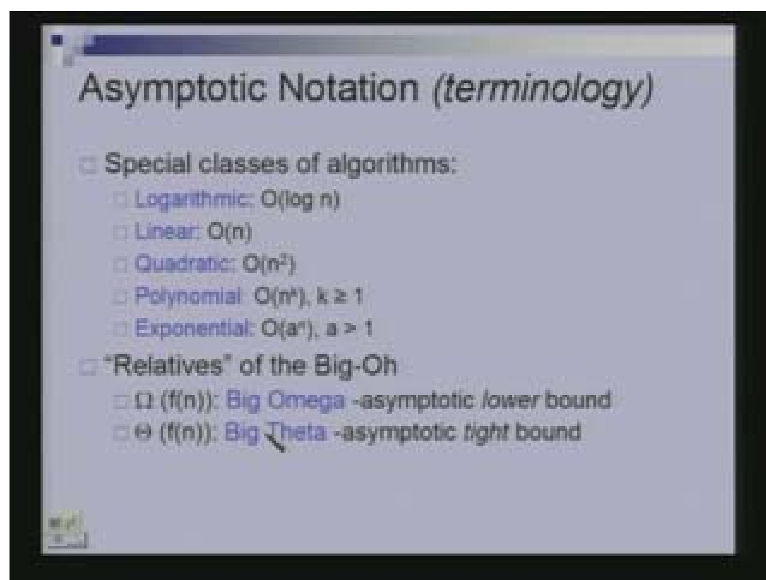
I keep this accumulator(S) around with me. When I finish the ith iteration of this loop, I have an S, the sum X [0] through X[i]. I can reuse it for the next step.

(Refer Slide Time: 44:15)



How much time does this take? In each run of this loop I am just doing two primitive operations that makes an order n times, because this loop is executed n times. I have been using this freely linear and quadratic, but the slide given below just tells you the other terms I might be using.

(Refer Slide Time: 46:01)



Linear is when an algorithm has an asymptotic running time of O (n), then we call it as a linear algorithm. If it has asymptotic running time of n^2, we called it as a quadratic and logarithmic if it is log n. It is polynomial if it is n^k for some constant k.

Algorithm is called exponential if it has running time of a^n, where a is some number more than 1. Till now I have introduced only the big-oh notation, we also have the big-omega notation and big-theta notation. The "big-Omega" notation provides a lower bound. The function f (n) is omega of g (n),

$f(n) = \Omega(g(n))$

If constant time g (n) is always less than f(n), earlier that was more than f(n) but now it is less than f(n) in the limit, beyond a certain n_0 as the picture given below illustrates.

$c\,g(n) \le f(n)$ for $n \ge n_0$

f (n) is more than c (g(n)) beyond the point n_0. That case we will say that f (n) is omega of g (n).

$f(n) = \Omega(g(n))$

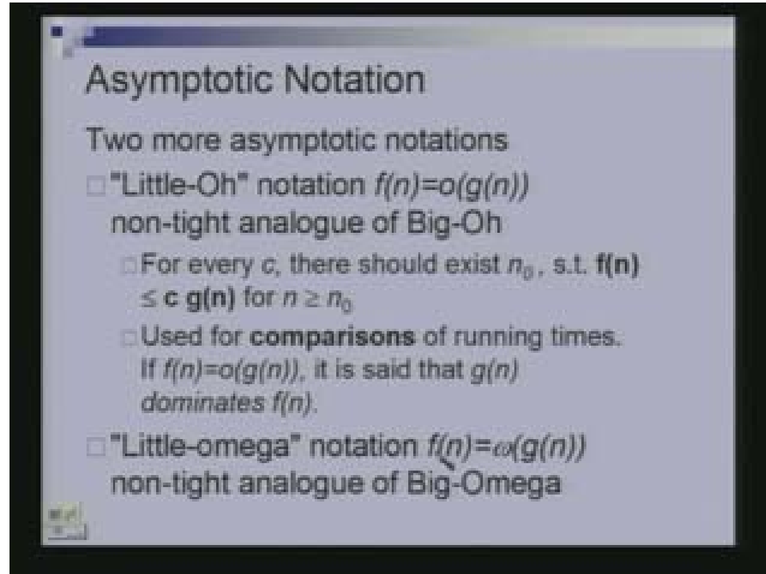
(Refer Slide Time: 47:51)

Asymptotic Notation

- The "big-Theta" Θ–Notation
  - asymptotically tight bound
  - $f(n) = \Theta(g(n))$ if there exists constants c_1, c_2, and n_0, s.t. $c_1\,g(n) \le f(n) \le c_2\,g(n)$ for $n \ge n_0$
- $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- $O(f(n))$ is often misused instead of $\Theta(f(n))$

In θ notation f (n) is θ(g (n), if there exist constant C_1 and C_2 such that f (n) is sandwiched between C_1 g (n) and C_2 g (n). Beyond a certain point, f (n) lies between 1 constant time g (n) and another constant time of g (n). Then f (n) is θ (g (n)) where f (n) grows like g (n) in the limit. Another way of thinking of it is, f (n) is θ(g (n)). If f (n) is O (g (n)) and it also Ω(g (n). There are two more related asymptotic notations, one is called "Little-oh" notation and the other is called "Little-omega" notation. They are the non-tight analogs of Big-oh and Big-omega. It is best to understand this through the analogy of real numbers.

(Refer Slide Time: 48:48)

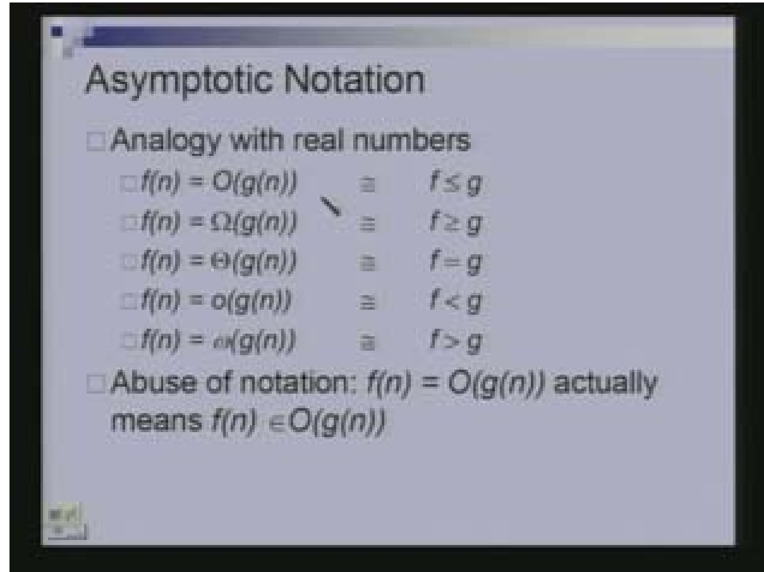


When f (n) is O (g (n)) and the function f is less than or equal to g or f (n) is less than c (g (n). The analogy with the real numbers is when the number is less than or equal to another number. Ω is for $\geq$ and θ is for =. θ(g (n) is function and f=g are real numbers.

If these are real numbers, you can talk of equality but you cannot talk of equality for a function unless they are equal. Little-oh corresponds to strictly less than g and Little-omega corresponds to strictly more. We are not going to use these, infact we will use Big-oh. You should be very clear with that part.

The formal definition for Little-oh is that, for every constant c there should exist some n_0 such that f (n) is < c (g(n) for n > n_0. f (n) $\leq$ c (g(n)) for n $\geq$ n_0 How it is different from Big- oh? In that case I said, there exist c and n_0 such that this is true. Here we will say for every c there should exist an n_0.

(Refer Slide Time: 49:12)

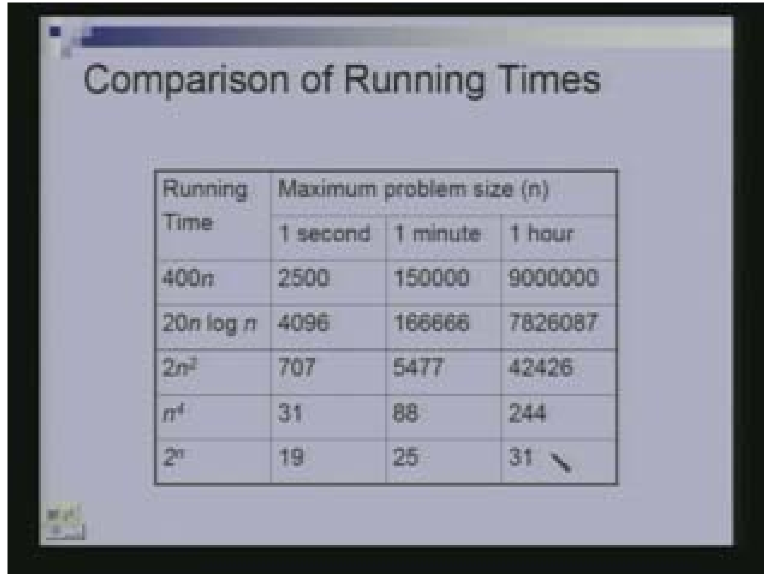


The slide which is below defines the difference between the functions. I have an algorithm whose running times are like 400n, 20n log n, $2n^2$, n^4 and 2^n. Also I have listed out, the largest problem size that you can solve in 1 second or 1 minute or 1 hour. The largest problem size that you can solve is roughly 2500.

Let us say if you have 20n log n as running time then the problem size would be like 4096. Why did you see that 4096 is larger than 2500, although 20n log n is the worst running time than 400n, because of the constant. You can see the differences happening. If it is $2n^2$ then the problem size is 707 and when it is 2^n the problem size is 19.

See the behavior as the time increases. An hour is 3600seconds and there is a huge increase in the size of the problem you solve, if it is linear time algorithm. Still there is a large increase, when it is n log n algorithm and not so large increase when it is an n^2 algorithm and almost no increase when it is 2^n algorithm. If you have an algorithm whose running time is something like 2^n, you cannot solve for problem of more than size 100. It will take millions of years to solve it.

(Refer Slide Time: 50:52)



This is the behavior we are interested in our course. Hence we consider asymptotic analysis for this.