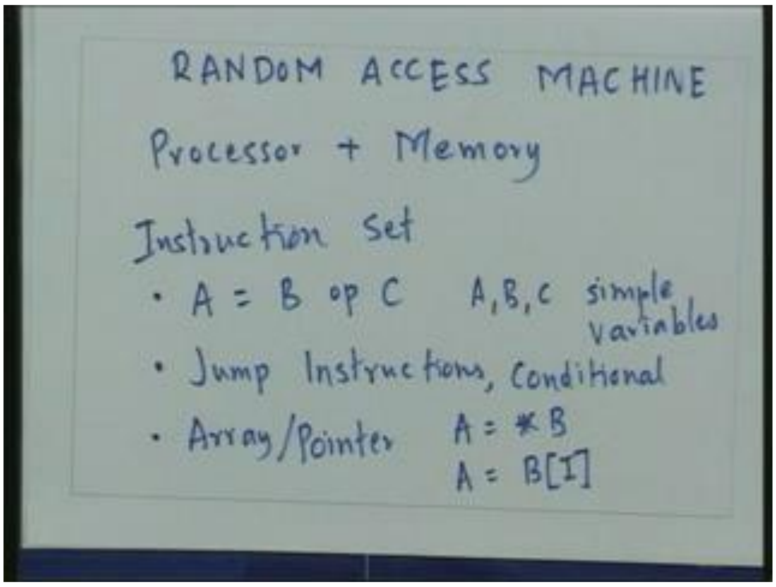
Design and Analysis of Algorithms

Prof. Abhiram Ranade
Department of Computer Science Engineering
Indian Institution of Technology, Bombay

Lecture - 3
Algorithm Analysis Framework – II

Welcome to the course on design and analysis of algorithms. This is the second lecture on the algorithm analysis framework that we are going to be developing algorithm analysis framework part two.

(Refer Slide Time: 00:51)



RANDOM ACCESS MACHINE

Processor + Memory

Instruction Set

- A = B op C   A, B, C simple variables
- Jump Instructions, Conditional
- Array/Pointer   A = *B
  A = B[I]

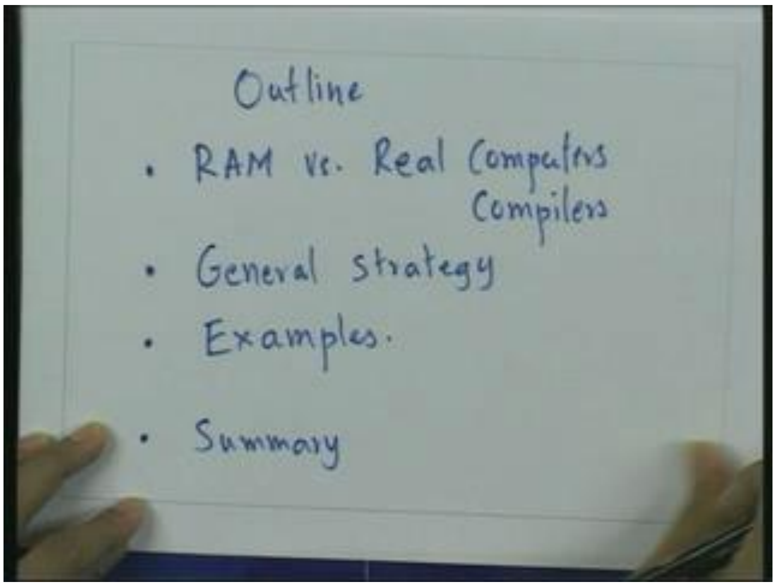
Let me quickly remind you, what we did last time. So last time, we defined an abstract computer model called RAM or the Random Access Machine. This was the model which we said we were going to be using throughout this course. Somewhat implicitly, but at the beginning, we want to make it fairly we want to discuss it in fair detail; so that it is well understood.

So, we said that this model consisted of a processor and a memory. And we said that the data would be stored in memory. The memory would include would be able to contain things like variables structures, arrays the usual, data structures. And there would a separate program memory. Then, we said that the instruction set would be reasonably simple.

So, let me just mention which instructions are important. So, basically the instruction set would be performing operations of the form A equals operator C. So, this would be the first kind of instructions, where A B C are simple variables. This is one kind of instruction. Then, we would have control instructions or jump instructions or control transfer instructions rather. So, jump instructions and conditional jumps.

And then, we would have array operations or array or pointer operations. So, these would be instructions of the form say A equals star B where we are using the C like syntax or it could also be something like A equals B index I. So, this instruction is sort of a standard array axis. And we said last time that these instructions would be single would take a single step would take a single cycle. So, the idea was that we were going to design algorithms. And then, you would reason about what instructions those algorithms would need to execute. And then, we would estimate the total time taken.

(Refer Slide Time: 03:50)



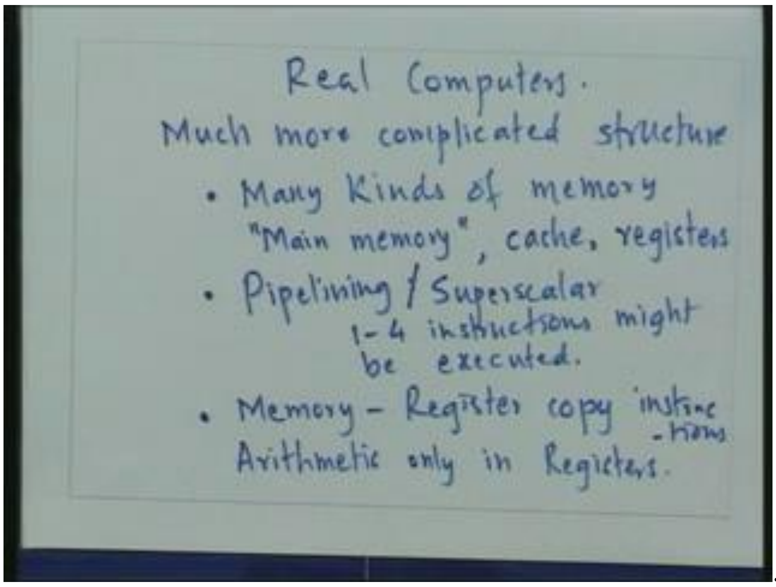
Outline

- RAM vs. Real Computers
  Compilers
- General strategy
- Examples.
- Summary

The goal for today is to do the following. So, let me write that down. So, the outline for today. So, we are going to start by comparing our RAM model with real computers. Actually, it is not just real computers, but also real compilers, because compilers also will make a difference. In this comparison will see what parts of a RAM model are of interest and which parts or which details is really something that we need to define for the sake of completeness. But, which really should not be taken too seriously.

Then, we are going to define our general strategy for algorithm analysis. And then we will take some examples. And after that we will have summary as usual.

(Refer Slide Time: 05:25)



Real Computers.

Much more complicated structure

- Many Kinds of memory
  "Main memory", cache, registers
- Pipelining / Superscalar
  1-4 instructions might be executed.
- Memory - Register copy instructions
  Arithmetic only in Registers.

.

So, let us now come back to our random access machine and try to find out in what sense this differs from real computers. So, first of all unlike our random access machine. In which we said that there was a single processor and a single memory real computer have a much more complicated architecture much more complicated.

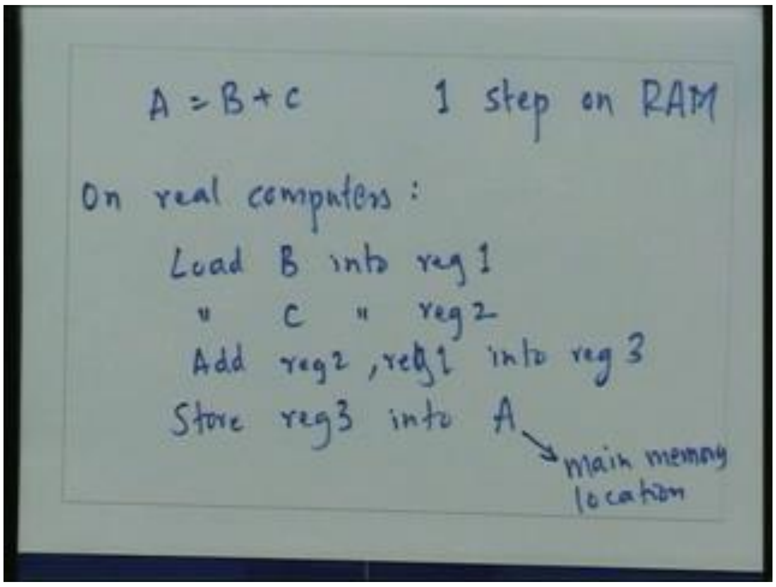
So, for example, there are many different kinds of memory. So, there could be our standard memory which we will call main memory. Then, there could be a memory called cache memory. I am sure you have heard of cache memory. Most computer advertisements do talk about cache memory. There could be memory which is called registers register memory.

Then, a very tricky complication arises in a very tricky idea is used in designing computers these days and that idea is called pipelining, what this means is that? Several instructions might be executed simultaneously at in; they will be at different stages of execution. Not only there is pipelining, but there is also something called superscalar execution.

Essentially this means that say somewhere between 1 and 4 instructions might be simultaneously executed. This is a fairly complicated idea and analyzing this idea is definitely beyond the scope of this course. However, you will see that to a certain extent these complications do not matter.

We said we defined an instruction set for our RAM machine. The instruction set for real computers is somewhat different in the sense that there are instructions which transfer data between different memories in the system. So, there is memory to memory instructions memory or memory to register instructions, memory to register copying instructions. And then arithmetic or any computation is only done in registers. So, as you can see this architecture is much more complicated.

(Refer Slide Time: 08:42)



A = B + C 1 step on RAM

On real computers :

Load B into reg 1

" C " reg 2

Add reg2 , reg1 into reg 3

Store reg3 into A

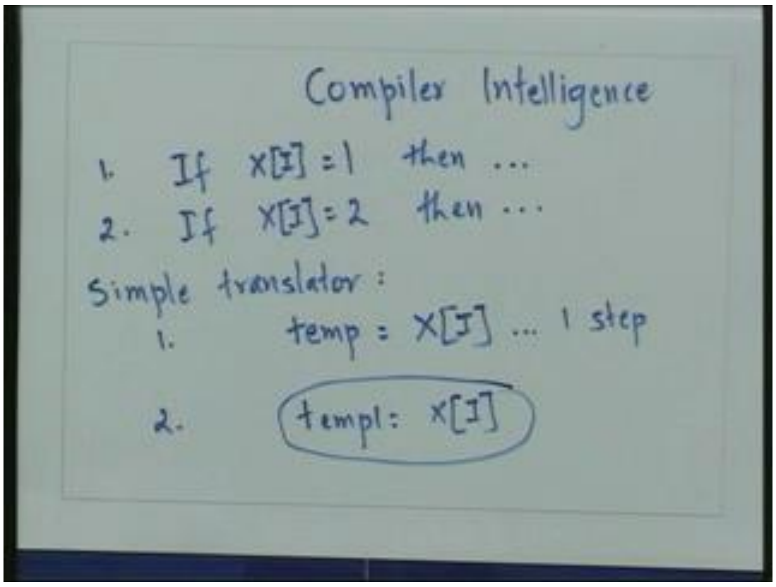
$\searrow$ main memory location

And I will just take one more example just to illustrate what i mean by the use of registers. So, this will drive home one of the points. So, we said that on a RAM suppose you have an instruction like A equals B plus C. So, this really happens in one step on a RAM. Our abstract on our abstract computer, this happens in a single step.

On the real computer this actually needs several steps. So, A might be a will be a variable in memory. So, first you will have to load it into a register. So, on real computers this is what you will do. So, we will load B into say register 1. Then, you will load C into register 2. And then, you will add register 2 register 1 into register 3. So, the value contained in register 2 is added to the value contained in the register 1 and the result is placed in register 3.

And then finally, you will store register 3 into location A of main memory. So, A is a main memory location. Just stress B and C as well. B and C are also main memory locations. But since, we cannot deal directly with data stored in main memory. We will have to deal with we will have to first move them into registers and then deal with them. This is the not the only way in which real computation is different from our idealized memory. There can be differences between real computation and our idealized computation.

(Refer Slide Time: 10:42)



Compiler Intelligence

1. If X[I] = 1 then ...
2. If X[I] = 2 then ...

Simple translator :

1. temp = X[I] ... 1 step
2. temp1 : X[I]

Also because of the use of compilers or because of intelligent compilers, let us take a simple example; say a program which consists of just two lines. So, this is a program say which looks like if X of I equals 1. Then, do something or then go to if X of I equals 2 then do something.

A simple translator for this, a simple translator would operate for this would have to extract the value of I. Because remember, we cannot do conditional operations directly on array elements. We first have to get them into some variable. So, this first statement would be translated into some fragment which will cause X of I to be loaded.

Say temp is the name and here we load X of I. This can be done in one step on a RAM as well as on a real computer very almost certainly. So, this is going to be one of the instructions in translation of or compilation of this first statement. A simple minded translator would also do a similar translation for the second statement which would also require 10 equals or temp 1 equals X of I, because even in the second statement we are accessing X of I.

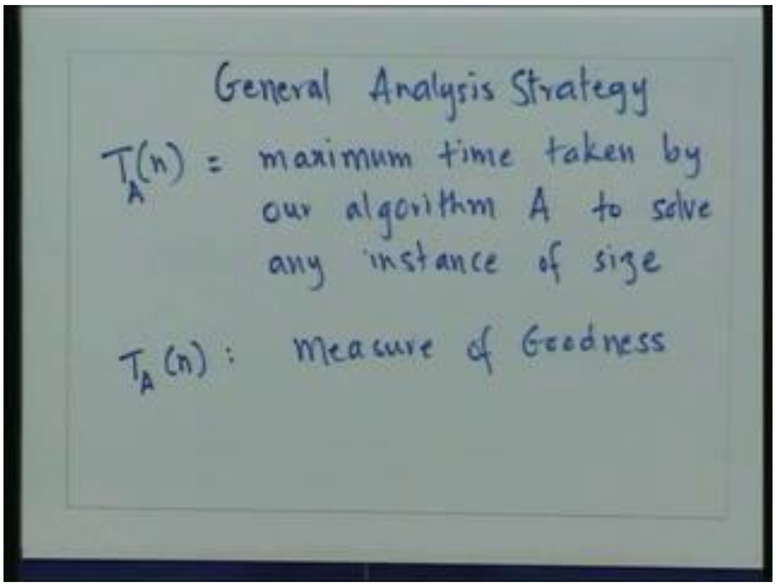
The fact that X of I has not is not changing in between these two statements may or may not be noticed by the compilers. So, as result a high level language statement such as this one or a sequence of high level language statement such as this one could be translated into a different number of statements depending upon the compiler intelligence.

And a simple compiler would translate the first statement and a first statement translation would require a fetching of X of I. And the second statement would be translated separately and that would also require a fetch of X of I. But, an intelligent transfer an intelligent compiler would realize that this fetch is really not needed. Because, it is already there and it can already use the value which is present earlier because that value is not being modified in between.

So, in general, when we write algorithms an intelligent compiler would be between us and the machine on which the algorithm executes. And therefore, we really should be thinking about what are the capabilities of that compiler. So in summary, I would like to say that we have an idealized model of what a computer is. It differs from what a computer what a real computer actually is. It is it is also idealized, because we are not really talking too much.

About the compiler which sets between a high level language algorithm or a high level language program and what actually gets executed on the machine. So in some sense now you might be thinking that it is actually a big surprise that whatever we say about real computers whatever we say about our idealized model applies to real computers as well. And indeed we have to be somewhat careful about how we make this application and we are going to see that next.

(Refer Slide Time: 14:50)



General Analysis Strategy

$T_A(n)$ = maximum time taken by our algorithm A to solve any instance of size

$T_A(n)$: Measure of Goodness

So, now let me indicate our General Analysis Strategy. Our measure for each algorithm is going to be a function T of n, where T of n is going to denote the maximum time taken by the algorithm on the RAM for any instance of size n. So, this is a very crucial definition. T of n is going to be maximum time taken by our algorithm.

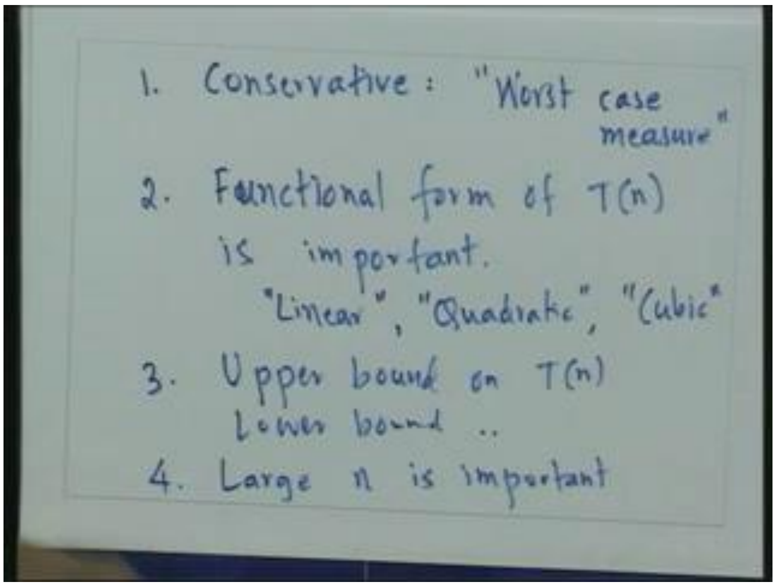
Here, let me call it algorithm A and let me say T sub A of n. Maximum time taken by our algorithm A to solve any instance of size n. So, as we noted in the last lecture, there could be several instances having the same size. So, if you are sorting a data set the size typically could be thought of as a number of values in that data set. And for different values even if the number of values is the same the time taken could be different.

So, when we talk about T of n or T of n for this algorithm A. We are asking, what is the largest time taken, say for sorting or whatever that algorithm is doing for all the input instances of size n and that is what we define as T of n. So, we will have a value which will be defined for T of 1 T of 2 T of 3 T of 4. For every n we will have a value. In principle, we can determine that value by executing our algorithm on all possible instances of size n.

This is only for the definition, but I just wanted to say that just to indicate that our that this definition is very down to earth. And this expression is very well defined. So, T of A T sub A of n is going to be our measure of goodness of this algorithm A. Notice that, we are not saying we are not evaluating a single number for our algorithm. So, you give me an algorithm and you ask me how good is this algorithm.

I am going to tell you here is a function. This is how good this algorithm is. You might want me to tell you a number, but sorry I cannot do that. I am just going to tell you a function. And now for every algorithm that you might give me, I will indicate a certain function. And it is your job to find out which algorithm is actually good by looking at those functions carefully.

(Refer Slide Time: 18:28)



1. Conservative : "Worst case measure"
2. Functional form of $T(n)$ is important.
   "Linear", "Quadratic", "Cubic"
3. Upper bound on $T(n)$
   Lower bound ..
4. Large n is important

So, I would like to make a few more remarks on this definition. The first remark is that this definition is a conservative definition. So, it says that the time that I am going to think is significant for a certain input size is the worst case time is sort of the maximum possible time. I could have said, well let me consider all possible inputs of size n and let me take their average time. This would be a reasonable definition.

However, this averaging of often turns out to be turns out to be difficult to perform. Primarily the reason is that you might enumerate all possible instances of size n. But, those instances may not appear in actual day to day life with equal probability. So unless, they appear with exactly the same probability for all instances taking that average is usually not so significant.

Even if they did appear with equal probability even then taking an average introduces an extra complexity to this which we will deal with once in a while. But by and large, we will stick to this reasonably simple definition which happens to be conservative. So, this is sort of the worst. So, this is a conservative definition in the sense that if I say that T of n is 5.

I know that no instance of size n will take time less than 5 will take time more than 5 and. So, it provides me sort of a very solid guarantee. And since, I am talking about the worst possible time that an instance of size n can take. This measure is also called the worst case measure.

Now, I would like to comment on how the time taken on a RAM relates to real machines. So, we said earlier we pointed out the differences between the real computers and RAMs. And we saw there that a single instruction on the RAM might correspond to several instructions on the real computers and also perhaps vice versa. And also because of compiler technology we might get differences.

So then, the question arises whatever analysis we do? For the RAM is it really of consequence for real computers. So, the point the important point that we will see soon. But, which I would like to just mention right now is that the precise value of T of n. The precise the precise numbers in the expression that we get for T of n. May not be of great consequence of real computers and real computations, what is going to be of consequence; however, is the form of T of n.

Or let me write this as the functional form of T of n, what I mean by the functional form is the following. So, I am going to ask questions such as is T of n linear function of n. By linear I mean functions which are of the form A n plus B or is it a quadratic function of n. By this I mean functions which are of the form A n square plus B n plus C and so on. So, say cubic and other complex functions as well.

But essentially, I am not talking about the precise values of A and B. When, I talk about a function being linear, but I am just talking about the shape of that function. So, they crucial idea is going to be that the shape that this function T of n has is going to be independent of what computer it is on which you run your algorithm.

And since, we are really interested in characterizing the algorithm rather than the computer itself. It is important for us to pick a measure which depends entirely on the algorithm. And in fact, it is going this going to be the shape or this functional form, whether T of n is a linear function or a quadratic function or a cubic function. And we will we will soon see examples of this in a minute.

Very often, we will not be able to exactly estimate T of n. We will not be able to give exact bounds on T of n. Here, we will say something like what is an upper bound or we will say something like what is the lower bound. If the upper bound and lower bound match then great our analysis is complete. But, this does not always necessarily happen.

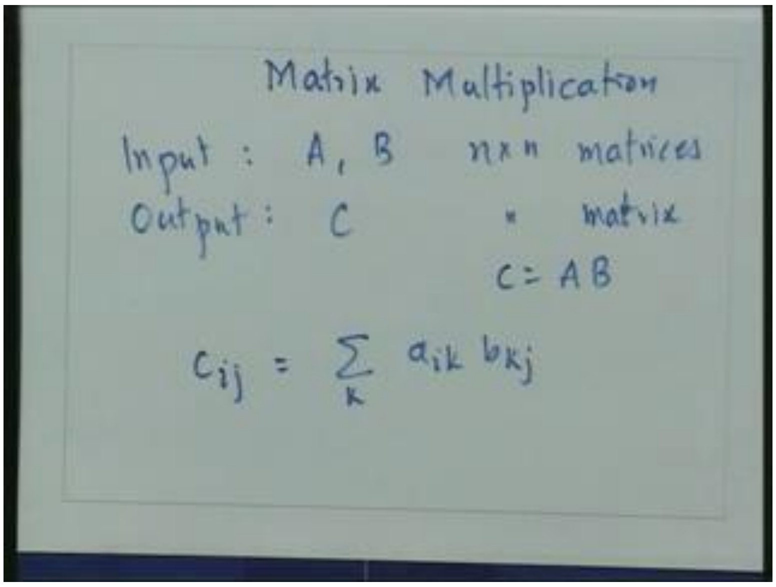
So in that case well, we will have to live with incomplete our work done incomplete. And bridging the difference between the lower bound and the upper bound will be subject of research. Finally, I will also comment that we will mostly be concerned or it is customary to be concerned about getting good estimates of on T of n for large n.

So, we always have in our mind that we are solving large problems. As our problems become larger and larger which algorithm does better that is the question that we should be asking. So, let me answer that question very quickly. If an algorithm has time quadratic, then it is going to take more time than an algorithm which has time linear. Certainly as the problem size n goes to infinity.

And so in the end, what we will end up saying is that a quadratic time algorithm is worse than a linear time algorithm. Does not matter, what the precise constants are whether it is 3X square plus 9 or whether it is 1000X plus 56, because as n goes to infinity as n goes to infinity linear time algorithm is going to take less time eventually as n goes to infinity than a quadratic time algorithm.

And this idea is justified, because typically we use computers to solve large problems. If a problem is small, then either we solve it by hand or it does not take too much time on a computer. Computers are very fast these days. And so, analyzing it is really not of interest. So, this is our overall analysis strategy. And this is a general idea about when we say an algorithm is good. When we say an algorithm is bad and so on.

(Refer Slide Time: 26:21)



Matrix Multiplication

Input : A, B $n \times n$ matrices

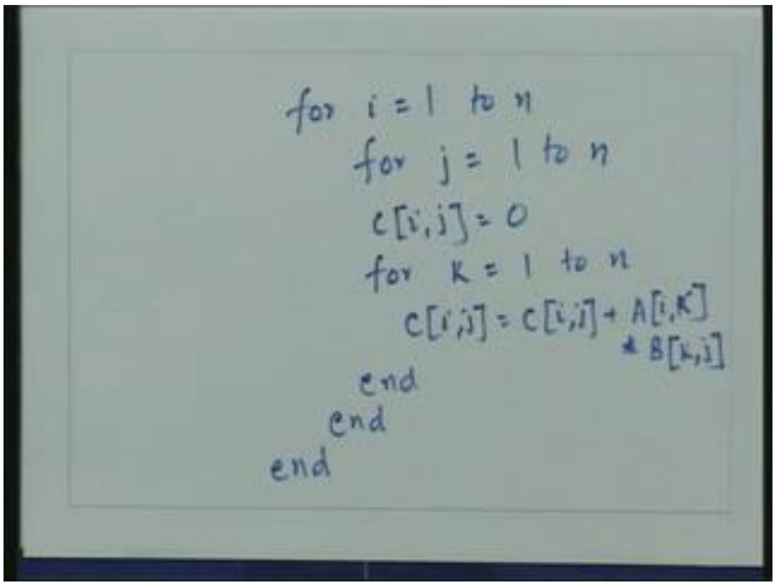
Output : C " matrix

$C = AB$

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

So, let us now take some examples to make these ideas more concrete. The first example is going to be matrix multiplication. Our input is going to be 2n by n matrices say A and B which are n by n matrices. Output is a matrix C which is also A n by n matrix and C will be A times B, where A times B is the usual matrix product. So, in particular C i j is defined as summation over k of a i k b k j. Everybody knows this definition. I am just writing it down just for completeness. There are many algorithms for matrix multiplication. But I am going to consider relatively simple the most natural algorithm which just uses this definition. That is all nothing more.

(Refer Slide Time: 27:30)

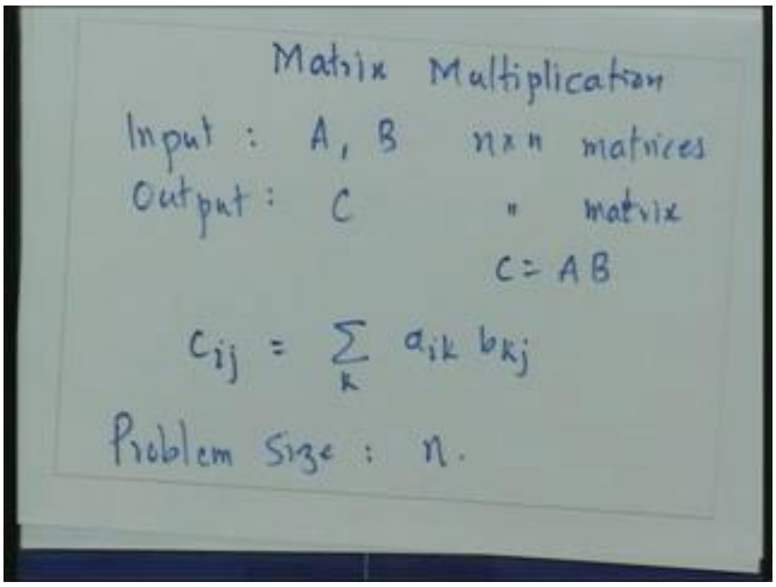


```
for i = 1 to n
    for j = 1 to n
      C[i,j] = 0
      for k = 1 to n
        C[i,j] = C[i,j] + A[i,k] * B[k,j]
      end
    end
end
```

So, here is that simple algorithm. So, we are going to have three loops. So, for I equal 1 to n for j equals 1 to n. And then, we are going to set C i j equals 0 over here. So, C i j is an entry in a two dimensional array n by n of size of dimension n by n. And so, we will assume that all over matrices are represented by two dimensional arrays and they are in memory.

And having set each C i j to 0 we will just use our definition of C i j to calculate the value of it. So, for k going from 1 to n C i j equals C i j plus A i k times B k j. And then we just close all the loops. So, this is our algorithm. We would like to find out how long this takes on our random access machine RAM.

(Refer Slide Time: 29:13)



Matrix Multiplication

Input : A, B $n \times n$ matrices

Output : C " matrix

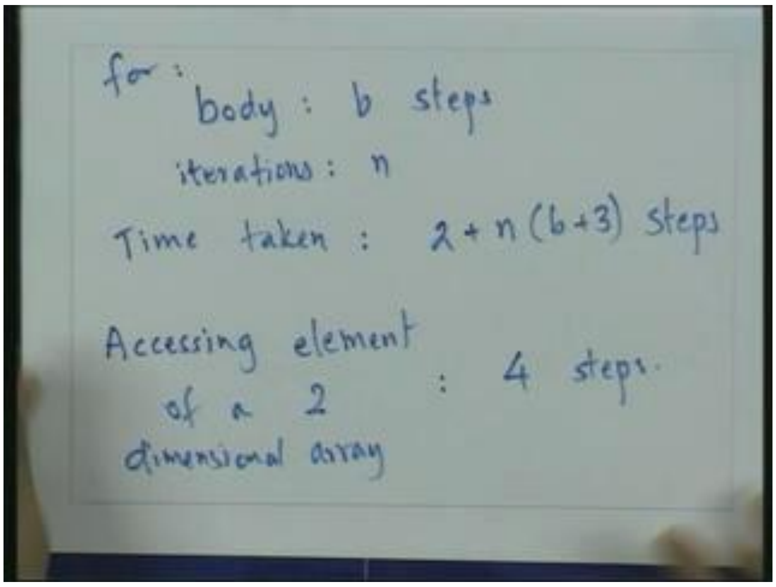
$C = AB$

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

Problem Size : n.

Let me just point out that the problem size can be thought of as just n itself. So, the first thing to note in analyzing this algorithm is that for every instance of the same problem size the time taken is the same. This is not typical of algorithms, but we have chosen this algorithm just to illustrate the analysis ideas. And later on we will see algorithms where the time taken is different and we will analyze those as well. Then, this algorithm also has a very simple structure. Again we will be analyzing more complex structures later on. This basic structure in this algorithm is that of a loop. And we have already seen in the last lecture how to analyze loops.

(Refer Slide Time: 30:05)



for :

body : b steps

iterations : n

Time taken : $2 + n(b+3)$ steps

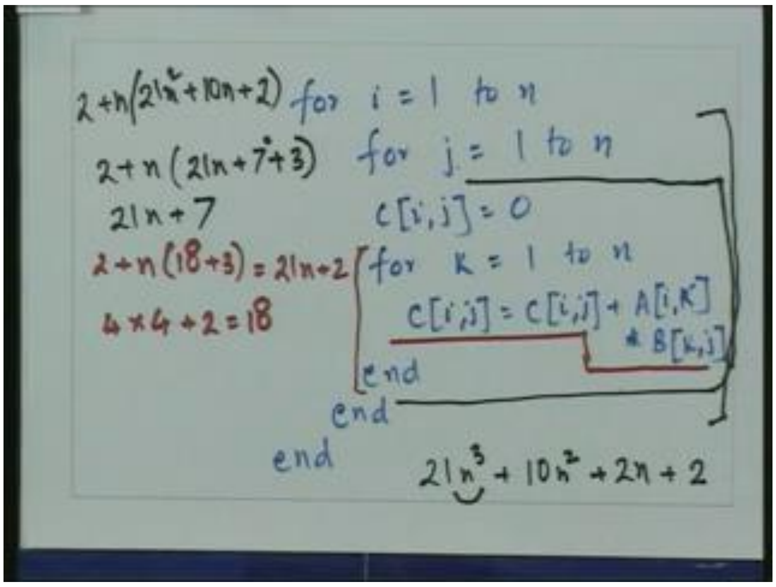
Accessing element of a 2 dimensional array : 4 steps.

So, let us write that down. So, we said last time that if I have a loop in which the body takes b steps and the number of iterations is n. So, we have a for loop in which the body takes b steps and the number of iterations is n. Then, we said that the total time taken is going to be 2 plus n times b plus 3 steps. This is what we concluded in the last lecture.

If you do not remember these numbers 2 and 3 do not worry about it so much. We are in the end going to see that these two numbers are not important. But right now, we are going to do pretty exact analysis. Knowing how to do this exact analysis is important for the sake of completeness and we will do it just once. It might look it might feel a little bit painful, but it is important to do it once. So that, we are sure that we have understood the RAM model completely.

Later on we will see why not, worry about some details is important. And in fact, the second algorithm that we will analyze we will analyze without doing without paying too much attention to all the small detail. But, for now we will stick to exact analysis and will note that for an exact analysis. The time taken for in a for loop is going to be 2 plus n times b plus 3, where b is the time required for the body. The second idea that we needed from last lecture was that if I am accessing a two dimensional array an element of a two dimensional array it takes me 4 steps.

(Refer Slide Time: 32:22)



```
$2+n(21n^2+10n+2)$   for i = 1 to n
$2+n(21n+7+3)$         for j = 1 to n
$21n+7$                       C[i,j] = 0
$2+n(18+3) = 21n+2$   for k = 1 to n
$4 \times 4 + 2 = 18$              C[i,j] = C[i,j] + A[i,k] * B[k,j]
                             end
                         end
                     end
                     $21n^3 + 10n^2 + 2n + 2$
```

So, we will use these ideas to analyze our program our algorithm over here. So, let us look at this statement. So, this statement has four accesses to two dimensional arrays. So, this is one access. This is another access. This is another access and this is another access and then there are two arithmetic operations that it does.

So, the time it takes must be 4 times 4 for the array accesses plus 2. So, 10 cycles totally. So, a single a single execution of this statement itself forget the loops just a single execution of the statement itself will take time 10 steps no matter what the values of C i j A i k and B k j are. But once, we know the time required for this single statement. We can calculate the time required for this outer loop over here this loop.

(Refer Slide Time: 30:05) How do we do that? Well we wrote down that if you have a for loop in which the body takes b steps. And if there are n iterations, then the time taken is 2 plus n times b plus 3. As a result of which this for loop will take time. So, it is 2 plus n is the same over here. It is still has n iterations. B; however is 10 plus 3.

So, it is going to take time 13 and plus 2. 18, so this will take time this. So, in the reverse it will take time 21 and plus 2. But now, this is the time for this particular part. So, we just have to worry about this additional time over here. So this entire portion, all the way from here to here comprises the body of for this loop for the, for loop on j.

This takes an additional 4 steps for this access and 1 step for this assignment. So, this will take time 21 n n plus 7. So, that is the time for the body of this loop. The total time taken for this outer loop now this one over here is again going back to our formula for body the time required for the for a loop in terms of its body and number of iterations.

Well the body of this loop requires time 21 n plus 7. So that is all that we need to plug in into this expression over here. So, if we do that then for this loop we are going to get 2 plus the number of iterations is still n times 21 n plus 7 plus that additional 3. So, this is the time that we are going to get. For this outer loop in j and now all that remains is to do this outer loop in i itself.

So, the total time for the outer loop well we know the time required for the body of that outer loop and that is going to be. So now, it is going to be 2 plus n times whatever time the body of this loop takes. And that is nothing but so if I substitute this I am going to get 21 n squared plus I will get a 10 n out of this and plus I will get a 2. So, this is going to be the time taken for the entire program and. In fact, this is going to be the time taken for our for our algorithm.

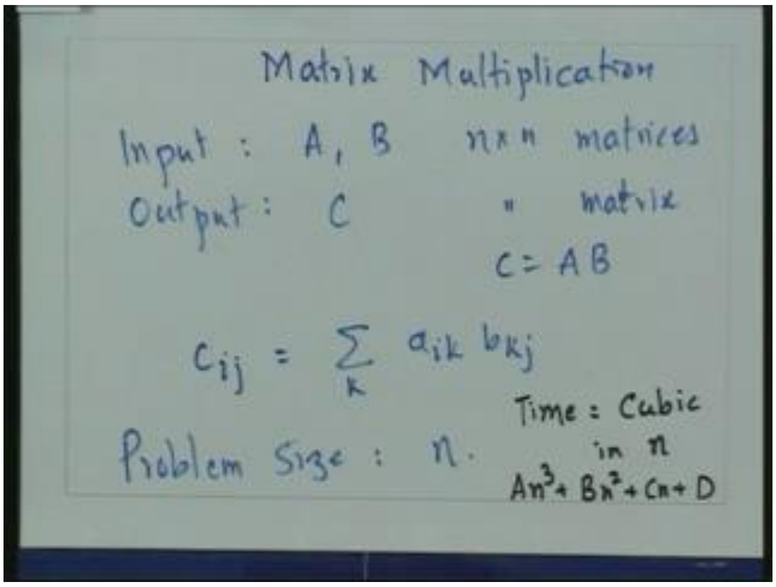
So, let me write that down. So, it is going to be something like 21 n cube, total time is going to be 21 n cube plus 10 n square plus 2 n plus 2 something like that. The important point you will see are not. In fact, these very numbers, but the fact that there is a n cube term in all this. But, anyway unless I have made a mistake in doing this arithmetic the expression should come out to be something like this. Some number times n cube plus some number times n square plus some number times n plus some number.

Now, let me ask you what is going to happen if I execute this algorithm on another computer which has a different instruction set, what well change? Well the time required for each of these individual instructions could change, because that would correspond to a different number of basic instructions on that computer.

There might be some intelligent compiler which might realize that for example, fetching once you fetch this then storing it to the same position you do not need to recalculate where you are going to store it something like that. But, the important point is that no compiler is going to be able to change the fact that this in a loop must execute n times. On no computer and with no compiler this fact is going to change.

Similarly, with no computer and with no compiler, the fact that this outer the second inner most loop is going to execute n times is also going to change or that the first loop is going to execute the outer most loop is going to execute n times. That is neither going to change. So, no matter where we run this program on what compiler we use the time taken is going to involve this n cube term. So, it will be 21 n cube or it could be some 36 n cube or something like that, but there is going to be a n cube term.

(Refer Slide Time: 39:23)



Matrix Multiplication

Input : A, B $n \times n$ matrices

Output : C " matrix

$C = AB$

$c_{ij} = \sum_k a_{ik} b_{kj}$

Problem Size : n.

Time : Cubic in n

$An^3 + Bn^2 + Cn + D$

So, let me write down sort of the major conclusion. So, the main conclusion is that the time is cubic in n. So, what I mean what do I mean by cubic. Let me write that down again that the time is going to be some function of the form A times n cube plus B times n square plus C times n plus D. We do not really care so much about what the values of A B C D are when we say the time taken is cubic in n.

And in fact, saying that it is cubic in n provides us, precisely the level of the level of exactness that we are looking for. Because, we really do not know, what these constants are going to be, unless we look at the precise architecture of a computer. So, if we are going to analyze an algorithm if we are not going to analyze a computer itself. But, if we are going to restrict ourselves to the analysis of the algorithm itself then all that we can say is that the time taken is going to be A n cube plus B n square plus C n plus D.

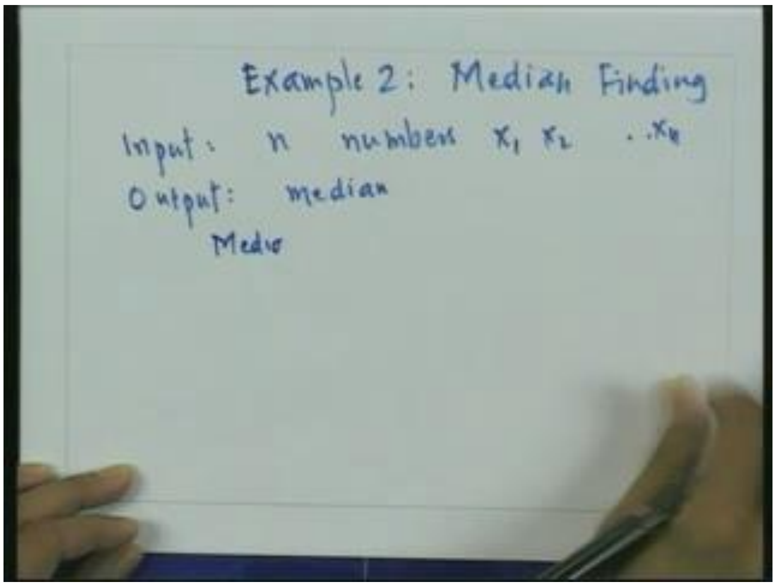
Or make this statement which sounds almost qualitative which is that the time taken is cubic in n. But, this is exactly what we want to be saying in this course. We want to say whether the time taken is cubic or quadratic or linear or something like that because this weak looking statement is exactly the kind of statement which we can justify and which we can claim about all possible computers.

The important point is that although we are making a weak statement. When we say we do not know what the constant A is. We are making at the same time our statement is actually a very strong statement, because we are saying this about all computers. If we made on down this a to be 105, then that statement we would have to make about a specific computer say some specific Pentium whatever computer or some other architecture computer and that is not what we want to make.

We do not want to talk about computers in this course or of precise computer architectures in this course, but we want to talk about algorithms. And therefore, we are going to restrict ourselves to making statements of this kind that. The time taken is cubical and or quadratic and so on. And our methodology is going to be similar to what we followed just down. We are going to execute we are going to mentally think about how long an algorithm executes. And then, we will estimate the time is cubic or quadratic or whatever.

In fact, our methodology is going to be simpler than what we just saw. Because now we know that we are really not interested in whether it is 21 or 2 plus n or whatever 2 plus n square or whatever. We can just say we will just ask about is it cubic is it quadratic. So and will take it in that manner.

(Refer Slide Time: 42:46)



Example 2: Median Finding

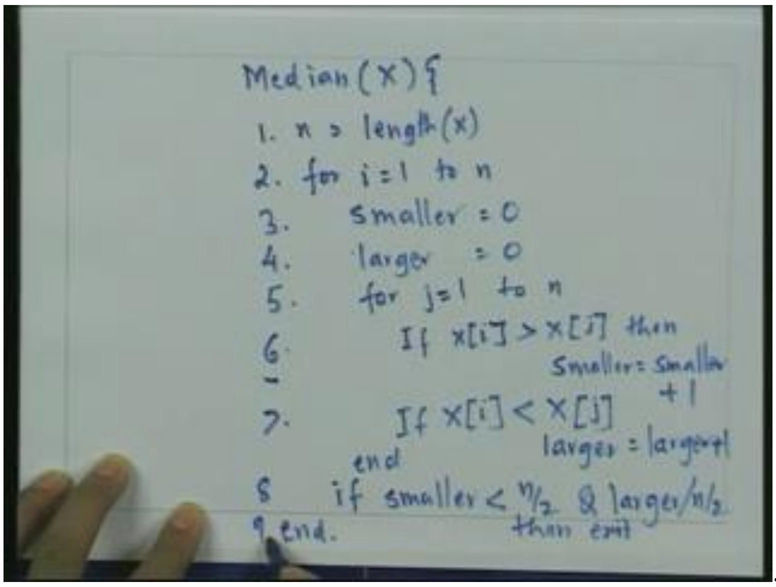
Input: n numbers x_1 x_2 .. x_n

Output: median

Medi

I will take one more example, to illustrate this idea of analysis of algorithms and this example is that of finding the median. The input in this case is n numbers. So, let us call them say X 1 X 2 all the way till X n and our goal is to find out the median. So, the median element is the element which appears in the middle in all those numbers. Or in general, it is defined as that element which is smaller than utmost n over two elements and larger than utmost n over two elements. So, let us first try to it is a very simple program for finding the median. This is not the best possible program, but it is actually it is a very simple program.

(Refer Slide Time: 43:46)



```
Median(X){
1. n = length(X)
2. for i=1 to n
3.     smaller = 0
4.     larger  = 0
5.     for j=1 to n
6.         If X[i] > X[j] then
                              smaller = smaller
                                          +1
7.         If X[i] < X[j]
                              larger = larger+1
       end
8    if smaller < n/2 & larger/n/2
                        than exit
9 end.
```

.

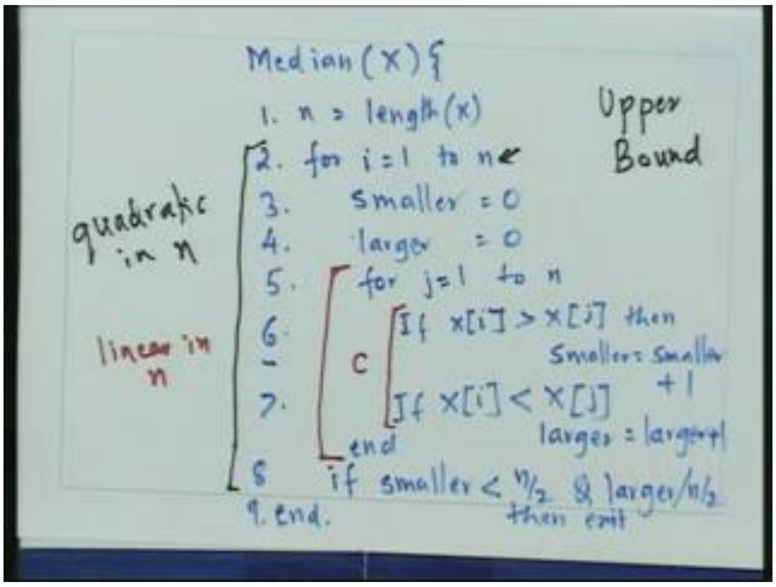
So, let me indicate that. So, let us say let me just write it separately over here. This program will look like this. So, let us assume that the input that is given to us is stored in this array X. So, the idea of the program is going to be I am going to look at successive elements of X. And I am going to check whether it satisfies the median definition, what; that means, is I am going to compare it every other element.

And see if the number of smaller elements is utmost n by 2 and the number of larger elements is also utmost n by 2. If we find that both of these conditions match then we are done. So, here is the code which does this because we are going to say n equals length of X. Then, we are going to have a loop where we consider each element in turn. So, for i equals 1 to n, we are going to say the number of smaller elements is equal to 0.

We are going to initialize the number of larger elements to also 0. Then, we are going to check. So now, we are going to check the i th element with the every j th element. So, for j equal to 1 to n, if X of i is greater than X of j then we will increment smaller. If X of i is strictly less than X of j. Then, we have found an element which is larger. And therefore, will increment the count for larger.

That ends this for loop and at this point we will check, whether we have median in fact. So, the check is going to be something like this. If smaller is less than n b y 2 and larger is also lesser than n by 2, then exit. Because, we have found, otherwise we repeat. We repeat with the rest of the loop. So, this matches this for over here. That is the very simple algorithm.

(Refer Slide Time: 46:50)



```
Median (X) {
1. n = length(x)
2. for i = 1 to n
3.     smaller = 0
4.     larger = 0
5.     for j = 1 to n
6.         If x[i] > x[j] then
               smaller = smaller + 1
7.         If x[i] < x[j]
               larger = larger+1
       end
8      if smaller < n/2 & larger/n/2
           then exit
9. end.
```

Upper Bound
quadratic in n
linear in n
C

Now, the question is how do we analyze it? So, we are going to do that analysis in the abbreviated style which i indicated. So, again we are going to look at the inner most loop. So, here is the inner most loop, how long does this take the body of this take. Well the body of this is going to take some C steps.

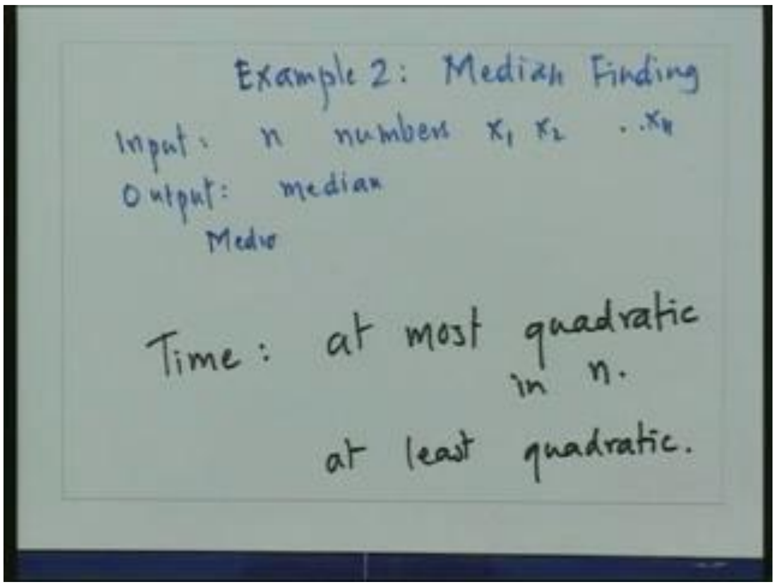
I will leave it to you as an exercise to show that C further will be no larger than 7. But, I am not going to worry about whether it is 7 or 8 or some or whatever it is, but I will just say it is some C. So then, what can I say about the time taken for a single execution of this loop.

Well it executes n times and so I can write down this time as being say some say linear in n, what does linear in n mean. It means that it is going to be some A n plus B sometime which is A n plus B. Now, we will focus our attention on this outer loop, how many times is this outer loop going to be executed. Well we said that it is going to be executed utmost n times, but it could execute fewer than n times as well.

But, let us make a simple assumption a simplifying assumption and say that it never exits early or in the worse case or let us say since we do not know whether it exits earlier or not we will just calculate an upper bound. So, now, we are in the business of calculating an upper bound. Since, we are not being too clever in analyzing when it will when it will exit early if it will exit early.

So, if we make this assumption that it does not exit early, then the time taken for this is going to be linear in the time taken for the body of this loop. But, that itself is linear in n and therefore, the time taken for this loop is going to be quadratic in n. But, that finishes the analysis of the entire algorithm. Because in fact, the time taken for the entire algorithm will have just this extra step and therefore, the time taken is going to be quadratic in n.

(Refer Slide Time: 49:07)



Example 2: Median Finding

Input: n number $x_1\ x_2\ \ldots x_n$

Output: median

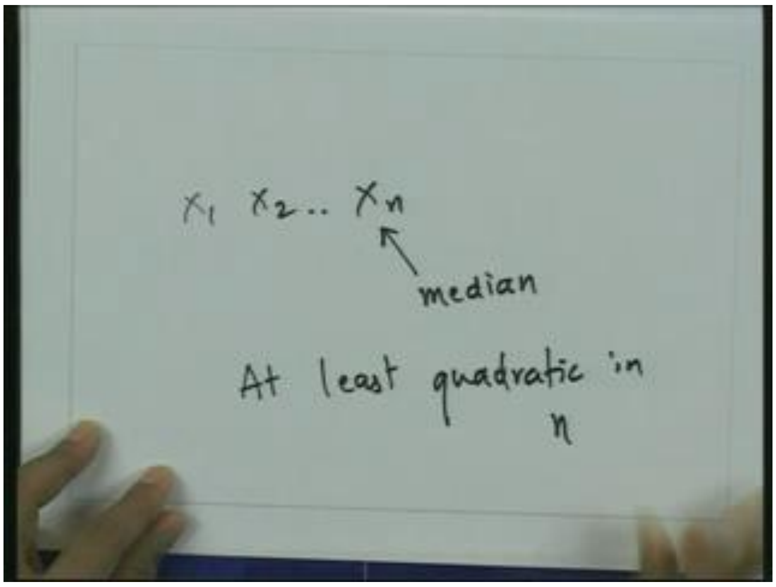
Medie

Time: at most quadratic in n.

at least quadratic.

So, what have we concluded? We have concluded that the time is utmost quadratic in n. So, remember again the time when I say the time I mean a function. So that function is smaller than some function which is quadratic in n. That is what I am saying over here, what do we need to do? In order to complete this analysis we need to ask is there a lower bound we can actually put. So for that, we need to we need to figure out what happens to this exit condition. Is there a situation in which this exit condition is not really met or met very, very few times?

(Refer Slide Time: 50:04)

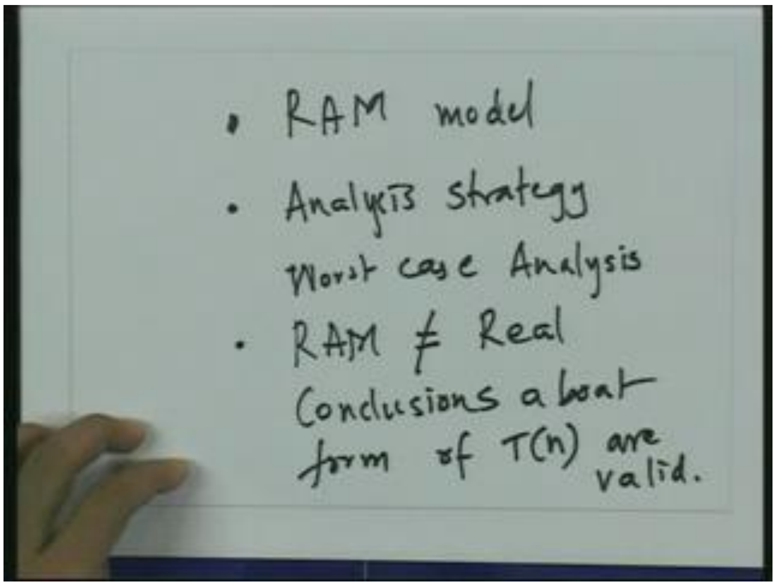


x_1 x_2 .. x_n ← median

At least quadratic in n

So in fact, I will leave this as an exercise for you and that exercise is if X 1 X 2 to X n is the input. And if this last number happens to be the median and all other numbers are distinct. Then, this exit condition will be met only at the very end. And then therefore, we will say that for this particular instance the time taken is going to be at least quadratic in n as well. So, let me write that down. So, this is our conclusion. It is at least quadratic as well as utmost quadratic. This does not mean for every instance it takes time at least quadratic. This just means for the worst instance it takes time at least quadratic.

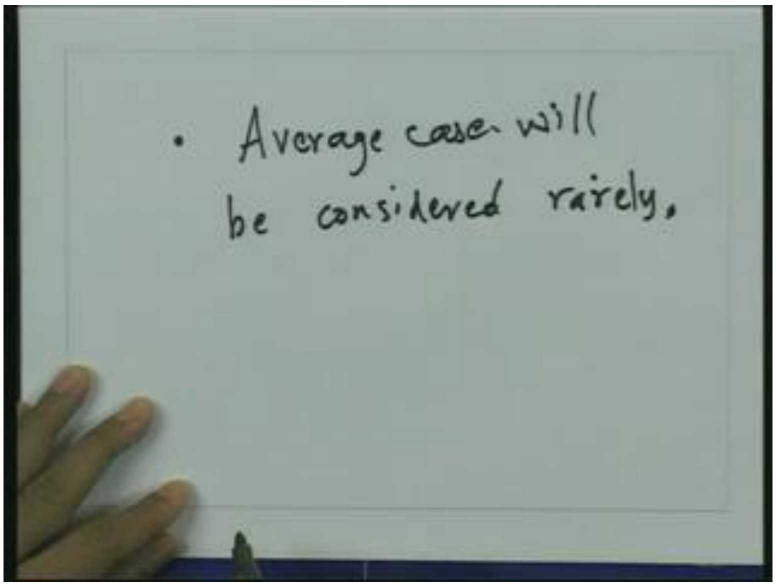
(Refer Slide Time: 51:00)



- RAM model
- Analysis strategy
  Worst case Analysis
- RAM ≠ Real
  Conclusions about form of T(n) are valid.

So, let me now conclude this lecture. So, what have seen in this lecture? In these two lectures well we have seen in the RAM model. Then, we have indicated an analysis strategy. This strategy is based on the notion of worst case analysis. We said that the RAM model is not equal to real computers. But, conclusions about the form of T of n are still valid. And that is what is going to be of importance? And finally, last point. For most of the most of the course this is going to be this is going to be our framework.

(Refer Slide Time: 52:02)



- Average case will be considered rarely.

We will consider worst case, but once in a while we will also look at the average case. So, let me just leave it leave you with this summary and that is the end of the lecture.