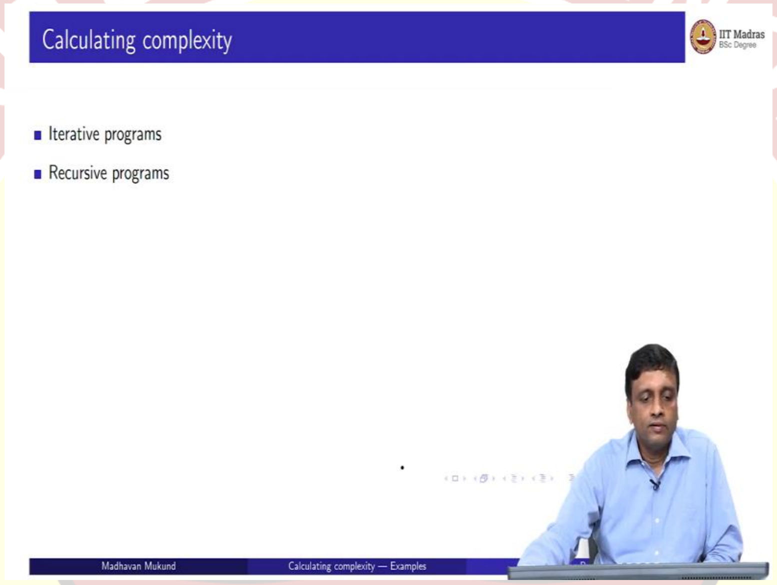
IIT Madras

ONLINE DEGREE

Programming, Data Structures and Algorithms using Python
Professor Madhvan Mukund
Calculating Complexity

So, we have seen how to express orders of magnitude using this big O notation. So, before we get into some actual algorithms, let us look at some examples of how we would look at very simple pieces of code, and try to estimate the complexity of the code.

(Refer Slide Time: 00:27)

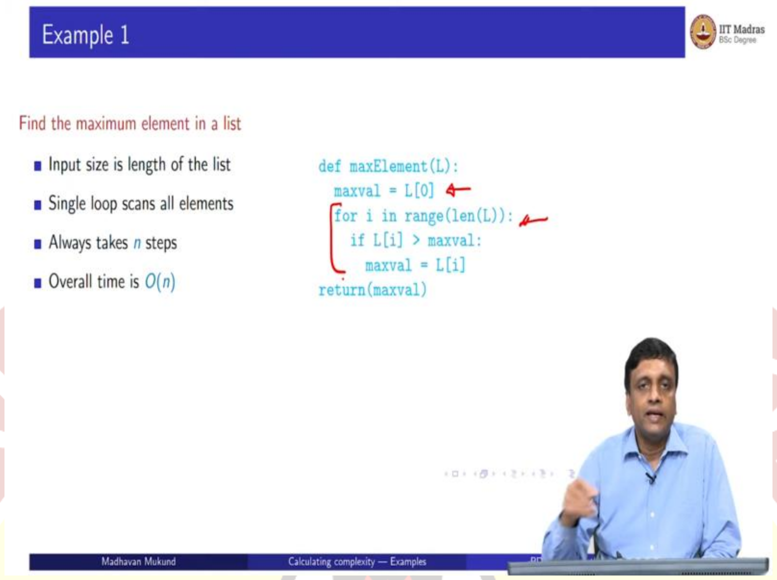


Calculating complexity

- Iterative programs
- Recursive programs

So, typically there will be two types of code that we look at; either it will be an iterative program, where we run some loops perhaps nested loops. Or it will be a recursive program, where the program calls itself. So, let us start with some examples of iterative programs.

(Refer Slide Time: 00:43)



Example 1

Find the maximum element in a list

- Input size is length of the list
- Single loop scans all elements
- Always takes n steps
- Overall time is $O(n)$

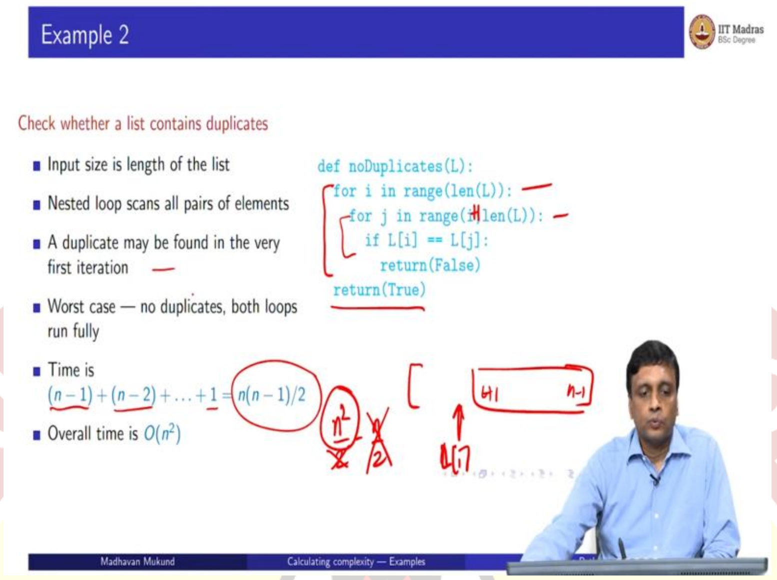
```
def maxElement(L):
  maxval = L[0]
  for i in range(len(L)):
    if L[i] > maxval:
      maxval = L[i]
return(maxval)
```

So, here is a very simple program that we all know, which is how to find the maximum in an in a list. So, what we do is we start assuming that the beginning of the list; the first element is the maximum. And then we scan through all the elements, and wherever we find a value, which is bigger than the current maximum; we updated to that value.

So, in one scan of the list, we just keep the track of maximum value that we see. So, the complexity of this is very easy to estimate. So, first of all we have to calculate what is the input size. So, a natural notion of input size for this is the length of the list; because obviously we have to find the maximum and a longer list, it will take more time.

And so, it is reasonable to say that the input list size or the input list length is the parameter of interest. Now, this is a single loop, so there is a for which goes through all the elements in the list. And is going to always scan all the elements, because we have no particular information about this list; we do not know whether the maximum value occurs in the beginning or in the middle or in the end. So, this is an example of a function that will always run that loop n times, where n is the size of the input list. So, this has worst case order n, but it is also in some sense every case order n.

(Refer Slide Time: 02:00)



Example 2

Check whether a list contains duplicates

- Input size is length of the list
- Nested loop scans all pairs of elements
- A duplicate may be found in the very first iteration
- Worst case — no duplicates, both loops run fully
- Time is $(n-1)+(n-2)+\ldots+1 = n(n-1)/2$
- Overall time is $O(n^2)$

```
def noDuplicates(L):
  for i in range(len(L)):
    for j in range(i+1,len(L)):
      if L[i] == L[j]:
        return(False)
  return(True)
```

Madhavan Mukund Calculating complexity — Examples

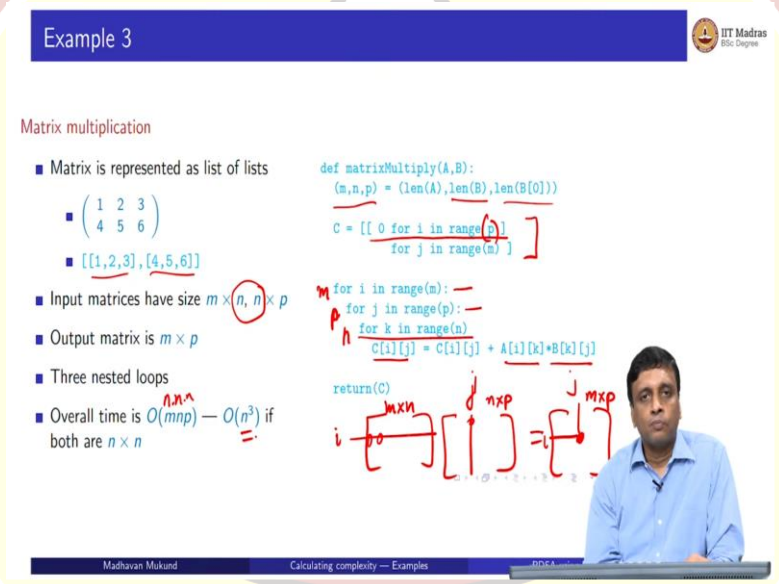
Now, let us look at the slightly more complicated question. We have a list of elements which is a no particular order, and we want to find out if the list has two duplicate elements; that is at least two elements which are the same. So, for this we run a nested loop. So, what we do is we scan through every element; so we have this whole list. So, when we are looking at element A Li, we compare we compare this to everything to the right of i. So, we have i plus 1 up to n minus 1; so that is the nested loop. So, for every i from 0 to n minus 1, for every j from i to n minus 1; so, maybe this should be i plus 1. From i plus 1 to n minus 1 we check whether Li is equal to Lj.

If we ever find that Li equal to Lj, we will return False. If we find no Li is equal to Lj, then we would return True. So, this now is a nested loop; so there is an outer loop which runs n times, and there is an inner loop which runs. The first time it will run n minus 1 times, because it run from 1. If the outer loop is at 0, it will run from 1 to n minus 1; if the outer loop is at 1, it will run from 2 to n minus 1 and so on. So, it is going to run, the inner loop is going to run n minus 1 times the first time; then n minus 2 times the second time and so on. And finally, the last time is going to run once; and effectively there is going to be a zeroth thing, which we will not worry about.

So, if you add up this then this turns out to be n into n minus 1 by 2. So, we saw last time that n into n minus 1 by 2, we just discard the lower powers of n. So, this is n squared by 2 minus n by 2; so we get rid of this, and we get rid of the constants. And we can declare that this particular nested loop has worst case complexity order n squared. Now, when is it going to reach at worst case is going to reach worst case if it is finally going to return True. That is it has to scan through every pair and decide that no pair is the same; so, there could be much better situations. So, there could be situations where the very first element L0 is equal to L1.

Then write at the beginning of this nested loop, you will find that it is False and return. So, there could be good situation, there could be bad situation. But as we said we are looking at the worst case situation. And in the worst-case situation here is one where there are no duplicates, and it is going to take n times order n squared.

(Refer Slide Time: 04:22)



Example 3

IIT Madras BSc Degree

Matrix multiplication

- Matrix is represented as list of lists
  - $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$
  - [[1,2,3],[4,5,6]]
- Input matrices have size $m \times n$, $n \times p$
- Output matrix is $m \times p$
- Three nested loops
- Overall time is $O(\overset{n.n.n}{mnp})$ — $O(n^3)$ if both are $n \times n$

```
def matrixMultiply(A,B):
  (m,n,p) = (len(A),len(B),len(B[0]))

  C = [[ 0 for i in range(p) ]
         for j in range(m) ]

  for i in range(m):
    for j in range(p):
      for k in range(n)
        C[i][j] = C[i][j] + A[i][k]*B[k][j]

  return(C)
```

Madhavan Mukund Calculating complexity — Examples

Let us look it at more computational examples; so let us look at the problem of matrix multiplication. So, if you remember matrix multiplication, we take two matrices which are of compatible size; and then we have to multiply them to produce an element here. So, if this element here is in row i and column j, then we take row i in this matrix. We take column j in this matrix, and we multiply pairwise every element A i k with B k j. So, that is how matrix multiplication happens. So, in particular if this has m times n that is m rows and n columns; and this has n rows and p columns, then the output is going to have m rows and p columns.

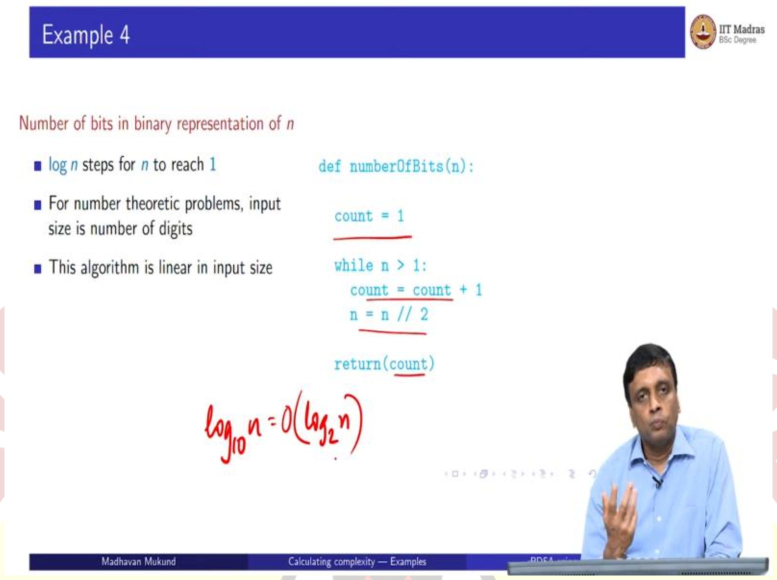
So, the input matrices have to have this compatibility; the number of columns of the first matrix must be equal to number of rows in second matrix. And the final output matrix will have the same number of rows as the first matrix, is the same number of columns is a second matrix. So, we will now use a very simple representation of matrices. So, we will take a matrix like this and represent each row as a list. So, the matrix as a whole is the list of list; so each row is the list, each row is one list. So, 1, 2, 3 is the first list; the second row is 4, 5, 6 is the second list; and the whole matrix is a list of these rows.

So, now if you look at the code what we have to do is first extract m, n, and p. So, we are going to assume that A and B can be multiplied, so the number of rows in A, which is a number of elements at the top level of the list is m. The number of columns in A is equal to number of rows in B. So, we can take the number of rows in B as n, and the number of columns in B is the number of entries in the first element of B. Because any of these rows has an equal number of entries; so we can just pick the first one. So, B0 will be the first row of B, and the number of entries in that the length of that will be p. So, we having got m, n and p; we have to setup the output to be a list of m rows with p columns.

So, this is p zeros, 0 for i in range p will setup a list of p zeros; and then we took m of them, so we get m rows of p0. So, this is our initial output matrix and now we do this update rule. We take each for each C i j, we run k over all n; and we take A i k times B k j and add it to the current value of C i j. So, we go over the number of rows in A and the number of columns in B. So, this now is three nested loops, so this is going to take m iterations; this is going to take n iterations and this is going to take. Sorry, this is going to take p iterations, and this is going to take n iterations; so, m times n times p.

So, the number of rows in A times a number of columns in B, times the number of columns in A, which is the same as the number of rows in B. So, we are going to take m times n times p. And if we are actually multiplying square matrices that is m is equal to n is equal to p; then this is going to be n times n times n or n cube. So, here we see a natural example of a problem which is actually higher than n squared; so, far we saw nested loops of n squared. Now, here is a problem which is actually n cubed.

(Refer Slide Time: 07:43)



Example 4

Number of bits in binary representation of n

- $\log n$ steps for n to reach 1
- For number theoretic problems, input size is number of digits
- This algorithm is linear in input size

```
def numberOfBits(n):

  count = 1

  while n > 1:
    count = count + 1
    n = n // 2

  return(count)
```

$\log_{10} n = O(\log_2 n)$

Madhavan Mukund Calculating complexity — Examples

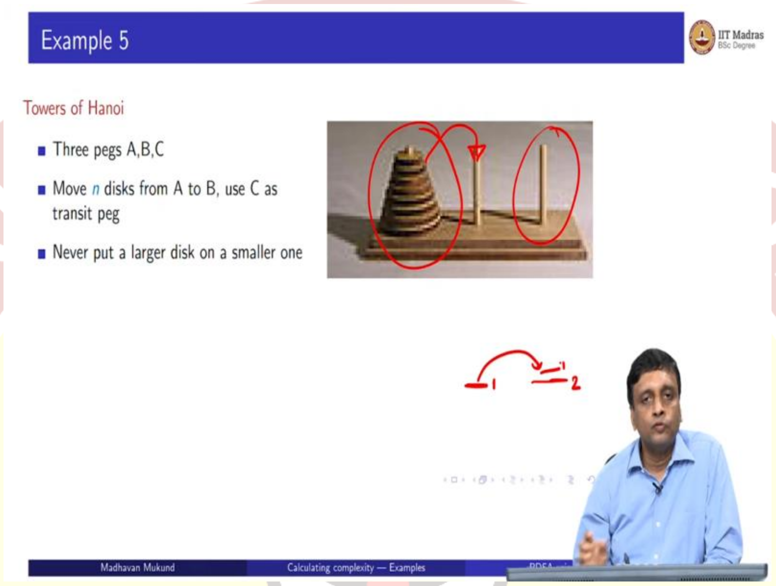
At the other end of the spectrum, supposing we want to find out the number of bits or number of binary digits in the binary representation of n. So, the usual way we do this is we keep dividing by 2, and we get each bit as we go along. So, we can just keep a counter initialize it to 1, and we just want to see how many times we can divide by 2 till we reach 1.

So, we keep incrementing every time we divide by 2, and we stay above 1; and finally, we return this counter. So, this is what we said before when we looked at this binary search and so on; you know when we were halving the interval. The number of times it takes to (have) n down to 1 is basically log n; so, this is going to take order log n steps.

So, for a list type of algorithm, where the input is of size is n; a function which takes order of log n would be considered very efficient. But we said that first number theoretic problems like this, we should not count n, the number n itself as the input size. We should rather look at how much space it takes to write down n, which is roughly the number of digits in n. So, we are saying really that the number of digits in n is proportional to log of n; and the output is taking time log n to compute. So, this is actually a linear function, even though it looks like it is taking log n time.

Log n itself is the size of the input and that something we have to keep here. Now, one thing you realize is that we might be writing it in decimal and taking it out in the binaries. So, if you say log 10 of n and log 2 of n; you might ask what is the connection between these two. It is not difficult to see that these are connected by a constant factor. So, if it is log anything of n to any base, its order of log any other base to n; so, this is also log of n. So, the input is say in base 10, so the size is a number of digits written in in decimal. The output is logarithmic to base 2; but they are both of the same order of magnitude.

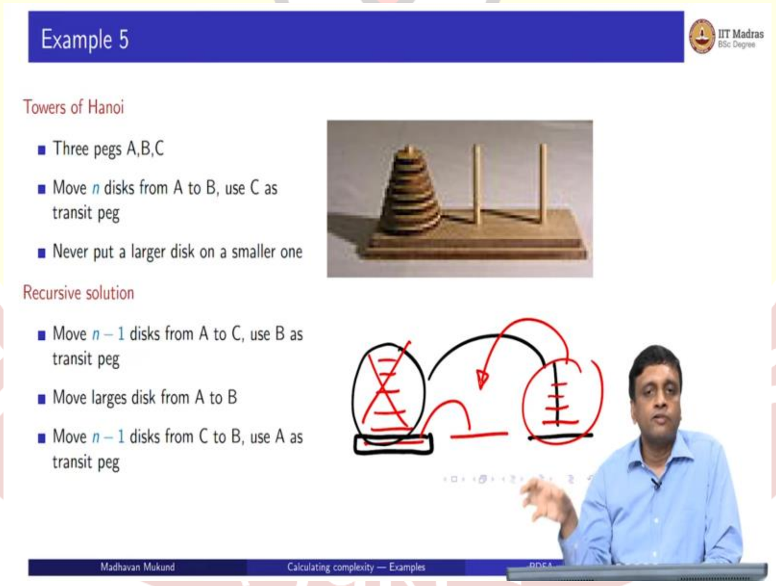
(Refer Slide Time: 09:45)



Example 5

Towers of Hanoi

- Three pegs A,B,C
- Move n disks from A to B, use C as transit peg
- Never put a larger disk on a smaller one



Example 5

Towers of Hanoi

- Three pegs A,B,C
- Move n disks from A to B, use C as transit peg
- Never put a larger disk on a smaller one

Recursive solution

- Move $n-1$ disks from A to C, use B as transit peg
- Move larges disk from A to B
- Move $n-1$ disks from C to B, use A as transit peg

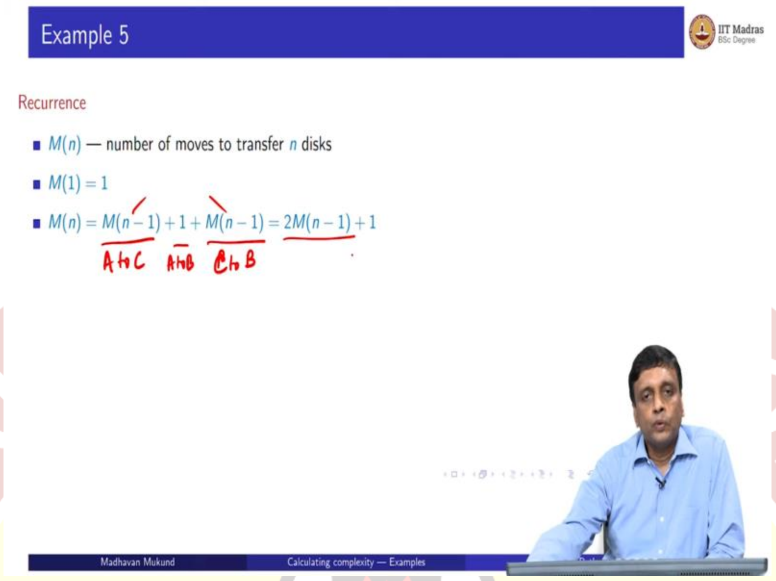
Such a final example let us look at a very famous problem; which is like a puzzle. So, you are supposed to take these discs which are of different diameter, and are arranged in decreasing order of size. And it is supposed to transfer them from one peg to another peg. But the constraint is you never allowed to put a bigger peg on top of the smaller peg. So, supposing you move the smallest one here, then you cannot put the next biggest one on top of it. So, you have to move biggest one to the other one. So, now the question is how are you going to transfer these n pegs from A to B? So, you are given a third peg.

So, since you are given a third peg, in this situation what you can do, is supposing this is smallest peg, this is the next biggest peg. Now, you can move this here and create a pile of two on that peg. So, this is how you can use the third peg in order to make space to move things around. So, there is a very well-known recursive solution to this, and that says that I have these n pegs. Now, if I keep this peg at the bottom, this largest peg at the bottom; then anything can sit on top of it. Because it is a largest disc at the bottom, anything can sit on top of it. So, I can as well assume that this disc never moves.

So, now I take these n minus 1 pegs which are above, and I transfer them to the third one. Now, this is a smaller problem. And effectively I have all three pegs at my disposal, because I do not have to use this one. When I gets to the bottom, I can just put anything on top of it. So, having done this, now I have I have moved this n minus things here; now, I can move this biggest one to the peg that I wanted to be. And now I can again solve the same problem, recursively move this n minus 1 things here. So, I can move n minus 1 pegs from A to C which was supposed to be the temporary peg, thinking of B as my intermediate peg.

So, I do not use B as my final peg; then I moved my biggest disc to the final peg. And then I can now again use the original thing A as intermediate thing, and move everything back from C to B. So, I have moved to n minus 1 things from A to C, one disc from A to B, and n minus disc from C to B.

(Refer Slide Time: 12:14)

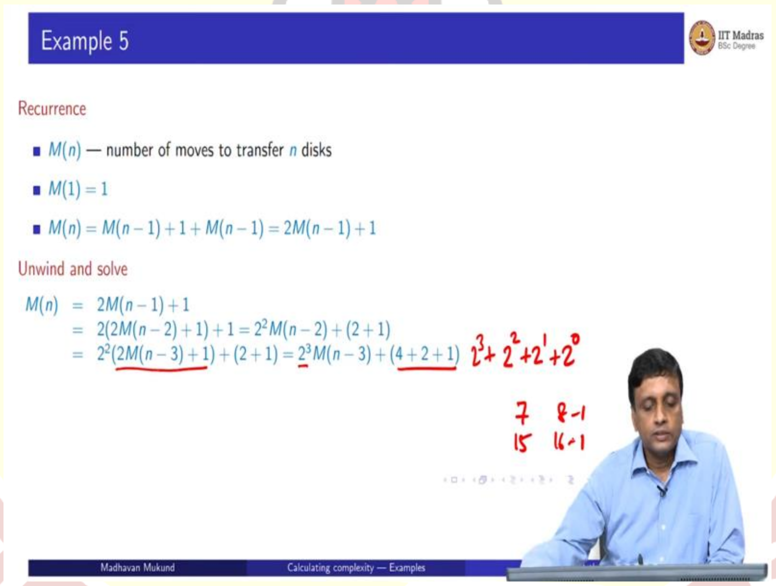


Example 5

Recurrence

- $M(n)$ — number of moves to transfer n disks
- $M(1) = 1$
- $M(n) = \underbrace{M(n-1)}_{A \text{ to } C} + \underbrace{1}_{A \text{ to } B} + \underbrace{M(n-1)}_{C \text{ to } B} = \underline{2M(n-1)+1}$

Madhavan Mukund | Calculating complexity — Examples



Example 5

Recurrence

- $M(n)$ — number of moves to transfer n disks
- $M(1) = 1$
- $M(n) = M(n-1) + 1 + M(n-1) = 2M(n-1) + 1$

Unwind and solve

$$\begin{aligned} M(n) &= 2M(n-1)+1 \\ &= 2(2M(n-2)+1)+1 = 2^2M(n-2)+(2+1) \\ &= 2^2(\underline{2M(n-3)+1})+(2+1) = \underline{2^3}M(n-3)+(\underline{4+2+1}) \quad 2^3+2^2+2^1+2^0 \end{aligned}$$

$7 \quad 8-1$

$15 \quad 16-1$

Madhavan Mukund | Calculating complexity — Examples

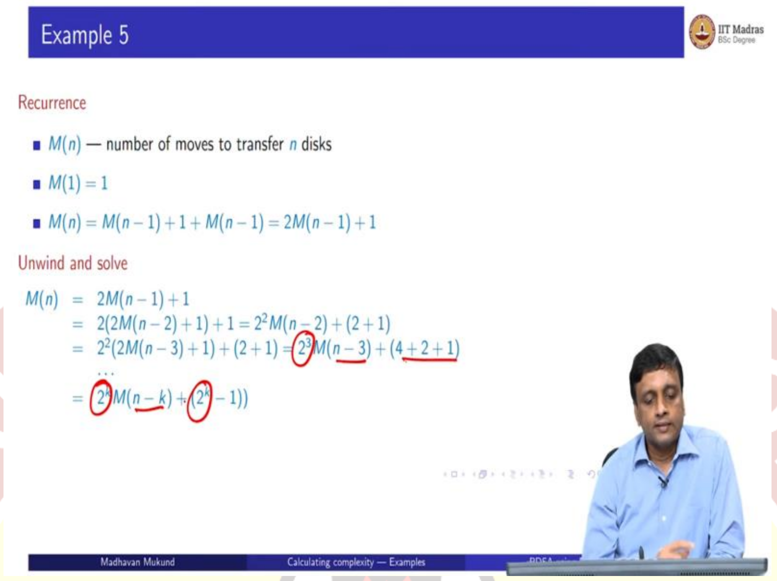
So, when I have a recursive solution like this, then the best way to express the complexity is to think of it, as a recursive formulation or a recurrence. So, let me say that the number of moves M, for an input of size n; if I have to move n discs M of n, it is a function of n. So, if I am moving one disc to move I can always move it; means one disc means I start with three pegs and only one disc. Then I can move it in one move; I do not have any constraints, so M of 1 is 1. On the other hand, if I have more than one disc, then I have to obey this recursive solution. So, what I do is I first move this is from A to C, I move n minus 1 disc from A to C. then I moved this one disc from A to B, and then moved n minus 1 disc back from C to B.

So, I have to solve this n minus 1 problem twice, and in between I have move one disc. So, if I combine this and this which are the same, I get two times n minus 1 plus 1. This gives me a kind of recursive dependence of M of n on M of n minus 1. And the usual way to solve this is to expand or unwind the recurrence by substituting. So, I start with M of n being two times n minus 1, n minus M of n minus 1 plus 1. And now I can do the same thing to; I do not know what M of n minus 1 is. But I can apply this same formula to that. So, I get two times M of n minus 2 plus 1, so this comes from expanding this, so, what is inside this bracket.

And I have this outer two and I have this plus 1. So, if I expand this out again and kind of regroup; this 2 times 2 becomes 2 squared. These 2 times 1, plus 1 is here; and I have of course M times M of n minus 2. So, this is the kind of cleaned up version of this one step expansion. So, now I do I expand this again; I will get something in terms of M of n minus 3. So, I will get 2 times M of n minus 3 plus 1; and again if I combine and clean up, I will get 2 squared times two is 2 cube times of M of n minus 1, n minus 3. And then I will get two squared times 1 4, and I already I have a 2 and I have a 1.

So, 4 plus 2 plus 1 and if you read this carefully, it is actually 2 squared plus 2 to the 1 plus 2 to the 0. So, each time I will get a higher power of 2; next time I will get plus 2 to the 3. And if you know this or you should check this, 1 plus 2 plus 4 is 7, which is 8 minus 1. If I add 8 to that I get 15, which is 16 minus 1. So, if I add up the powers of two up to n, I get 2 to the power n plus 1 minus 1.

(Refer Slide Time: 15:02)



Example 5

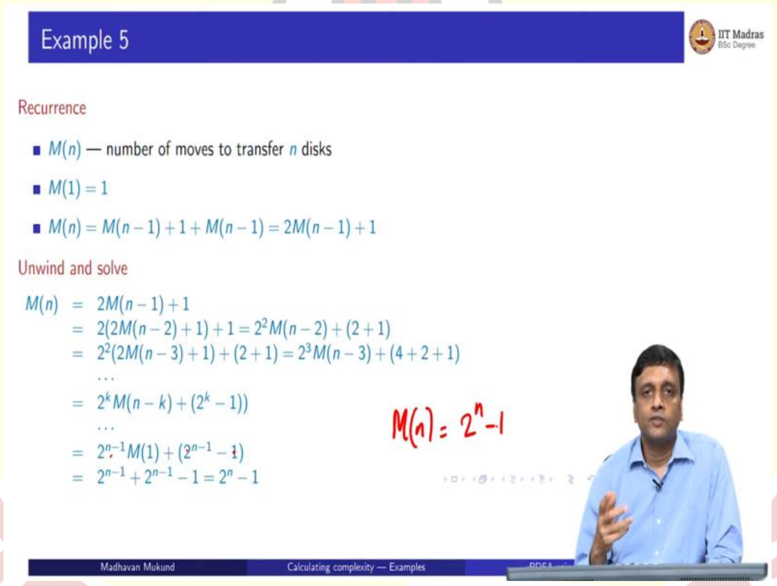
Recurrence

- $M(n)$ — number of moves to transfer n disks
- $M(1) = 1$
- $M(n) = M(n-1) + 1 + M(n-1) = 2M(n-1) + 1$

Unwind and solve

$$\begin{aligned} M(n) &= 2M(n-1)+1 \\ &= 2(2M(n-2)+1)+1 = 2^2M(n-2)+(2+1) \\ &= 2^2(2M(n-3)+1)+(2+1) = \boxed{2^3}M(\underline{n-3})+(4\underline{+2+1}) \\ &\dots \\ &= \boxed{2^k}M(\underline{n-k})+(\boxed{2^k}-1)) \end{aligned}$$

Madhavan Mukund Calculating complexity — Examples PDSA



Example 5

Recurrence

- $M(n)$ — number of moves to transfer n disks
- $M(1) = 1$
- $M(n) = M(n-1) + 1 + M(n-1) = 2M(n-1) + 1$

Unwind and solve

$$\begin{aligned} M(n) &= 2M(n-1)+1 \\ &= 2(2M(n-2)+1)+1 = 2^2M(n-2)+(2+1) \\ &= 2^2(2M(n-3)+1)+(2+1) = 2^3M(n-3)+(4+2+1) \\ &\dots \\ &= 2^kM(n-k)+(2^k-1)) \\ &\dots \\ &= 2^{n-1}M(1)+(2^{n-1}-1) \\ &= 2^{n-1}+2^{n-1}-1 = 2^n-1 \end{aligned}$$

$M(n) = 2^n - 1$

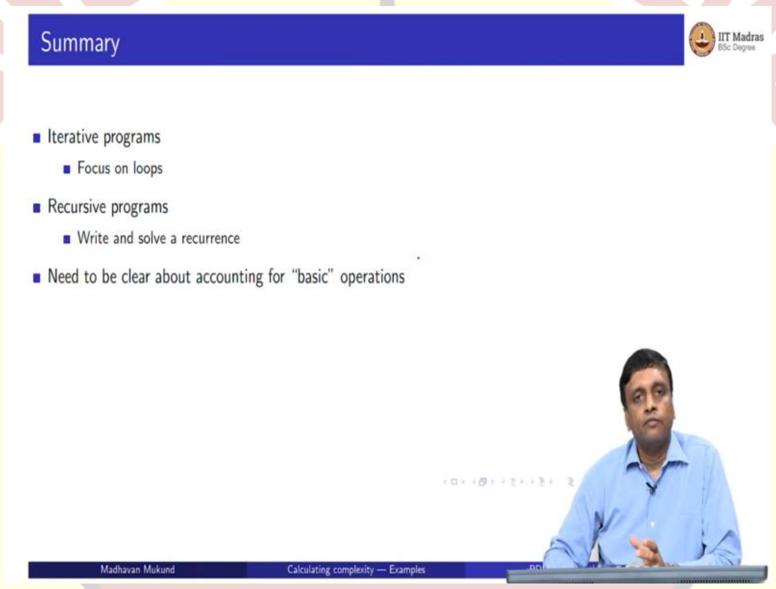
Madhavan Mukund Calculating complexity — Examples PDSA

So, I should use that fact to say that if I expand this k times, this 2 to the 3 will become 2 to the k. This n minus 3 will become n minus k; and this summation will be 2 to the power k minus 1. Because I would have 2 to the 0, plus 2 to the 1, up to 2 to the k minus 1.

So, this becomes the general form after k steps of substituting this recurrence; and eventually when I reach 1, I will have 2. If I do this n minus 1 times, I will have n minus minus 1, which is equal to 1. So, after n minus 1 steps if I take to be n minus 1, then I will get M of 1, and this k is again n minus 1. So, I have 2 to the n minus 1 times M of M of 1, plus 2 to the n minus n minus 1. But what is M of 1? M of 1 is just 1; M of 1 was known to be 1. So, this is just 2 to the n minus 1, plus 2 to the n minus 1 minus 1. So, is this plus this minus 1, which is two times this is 2 to the n minus 1.

So essentially, I have now by kind of laboriously substituting and coming down to the base case, I have argued that this tower of Hanoi solution actually takes exponential time. So, if I give you n discs is going to take you, 2 to the power of n minus 1 moves to execute that recursive solution, in order to transfer all the discs.

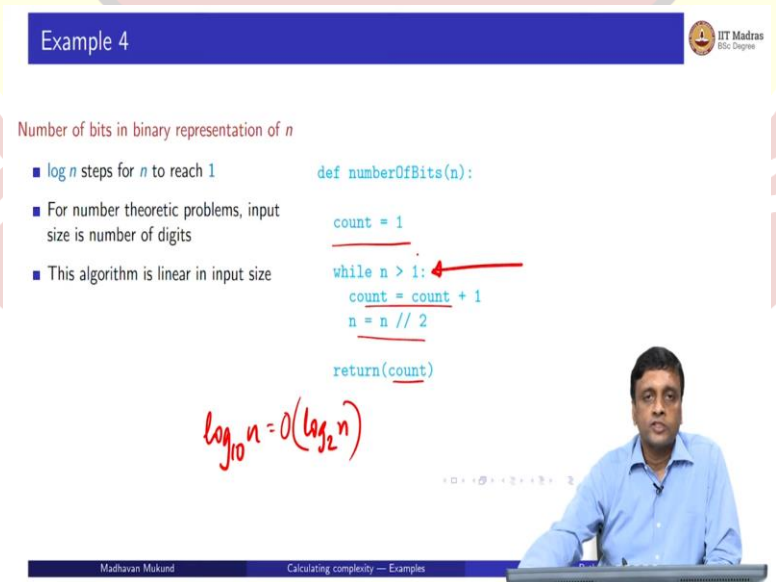
(Refer Slide Time: 16:27)



Summary

- Iterative programs
  - Focus on loops
- Recursive programs
  - Write and solve a recurrence
- Need to be clear about accounting for "basic" operations

Madhavan Mukund   Calculating complexity — Examples



Example 4

Number of bits in binary representation of n

- $\log n$ steps for n to reach 1
- For number theoretic problems, input size is number of digits
- This algorithm is linear in input size

```
def numberOfBits(n):
  count = 1
  while n > 1:
    count = count + 1
    n = n // 2
  return(count)
```

$\log_{10} n = O(\log_2 n)$

Madhavan Mukund   Calculating complexity — Examples

So, to summarize what we have seen is that if we are looking at iterative programs with loops, then what we really need to understand is how many times the loops execute. If there are no loops, then it is just a straight line of code is equals to take the constant amount of steps. So, any interesting program will have a loop which will depend on the input. We will run through the loop; if it is a for loop, it is a very easy to calculate. Because we know exactly how many times loop is going to execute. If it is while loop, then we have to be a little bit more careful to calculate how many times it will take for the while to become 1.

So, this is something that we saw, for instance, when we looked at that number of digits thing; so, here we had a while loop. So, when we have a while loop, then we cannot tell a priory that is going to take n steps. So, we have to figure out how long it is going to take for that condition to become False and the loop terminates. So, if we have an iterative program, we are basically looking at the loops. If we have a recursive program, then we have to understand the recurrence. So, we have to say, how the time complexity for the full input depends on the recursive or the inductive parts into which you are breaking it; and then we have to solve this recurrence.

So, we typically sol