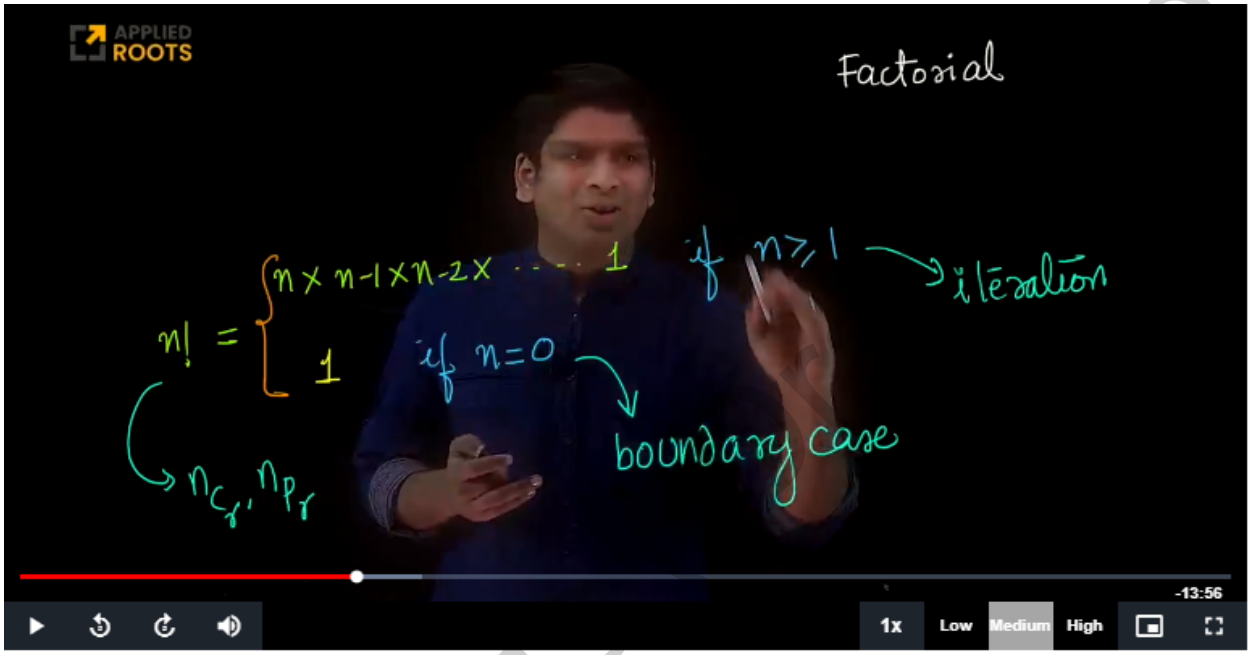
11.1 Introduction to Recursion

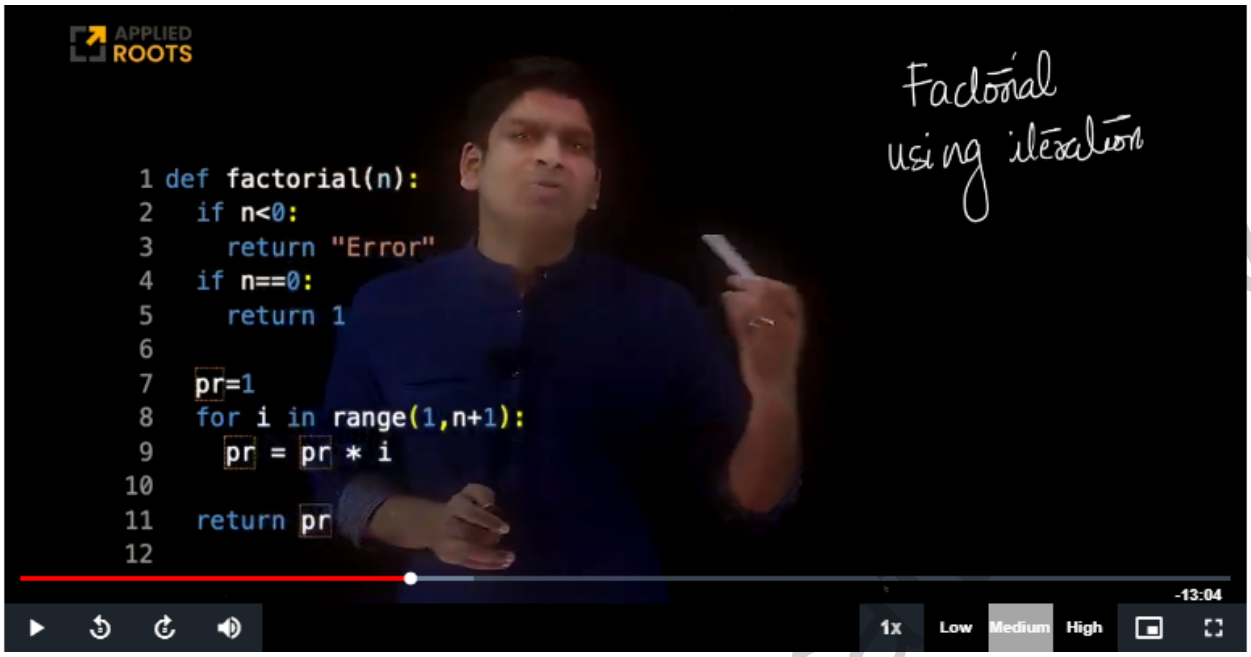
In this video we will see how we can convert factorial algorithm iterative implementation to recursive implementation.

Please look at the formulation of factorial of a number n using iterative approach:



Timestamp: 5:42

Factorial of a number n is implemented using iterative programming as below:



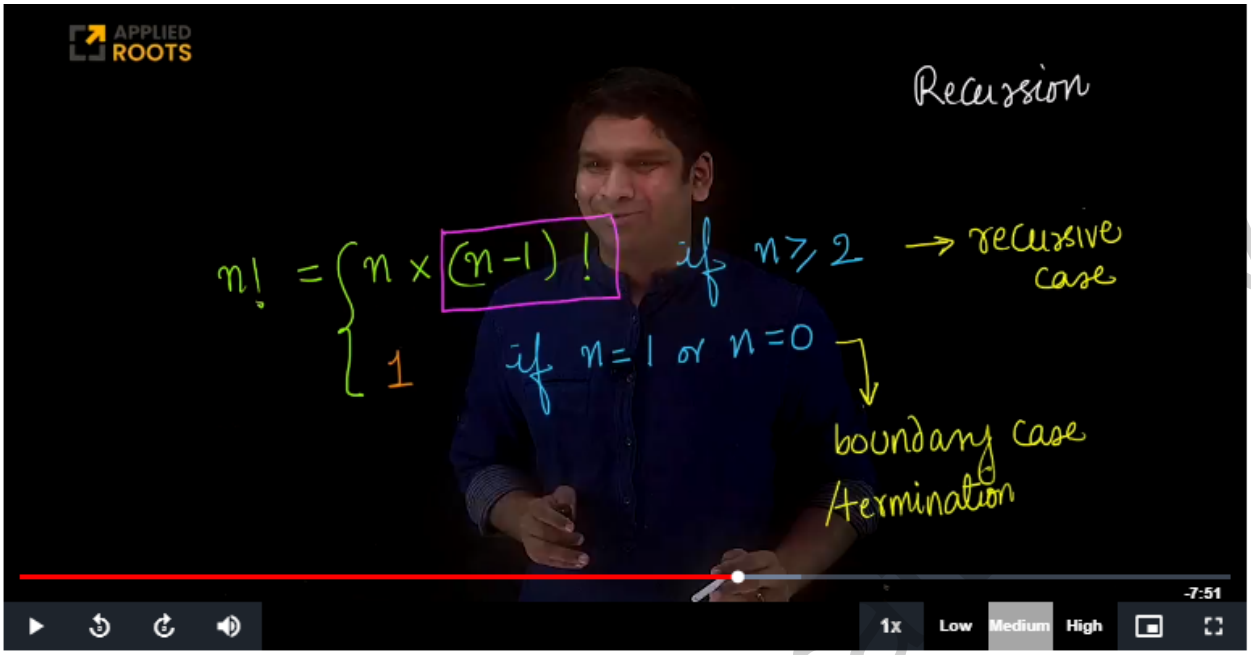
TImestamp: 6:14

Notice that factorial does not exist for negative number hence we are doing error handing by displaying “Error”.

Notice that n=0 is a boundary case hence we are handling it by returning 1 without any computation.

For n>=1, we are implementing an iterative algorithm that stores the result 1*2*....*n.

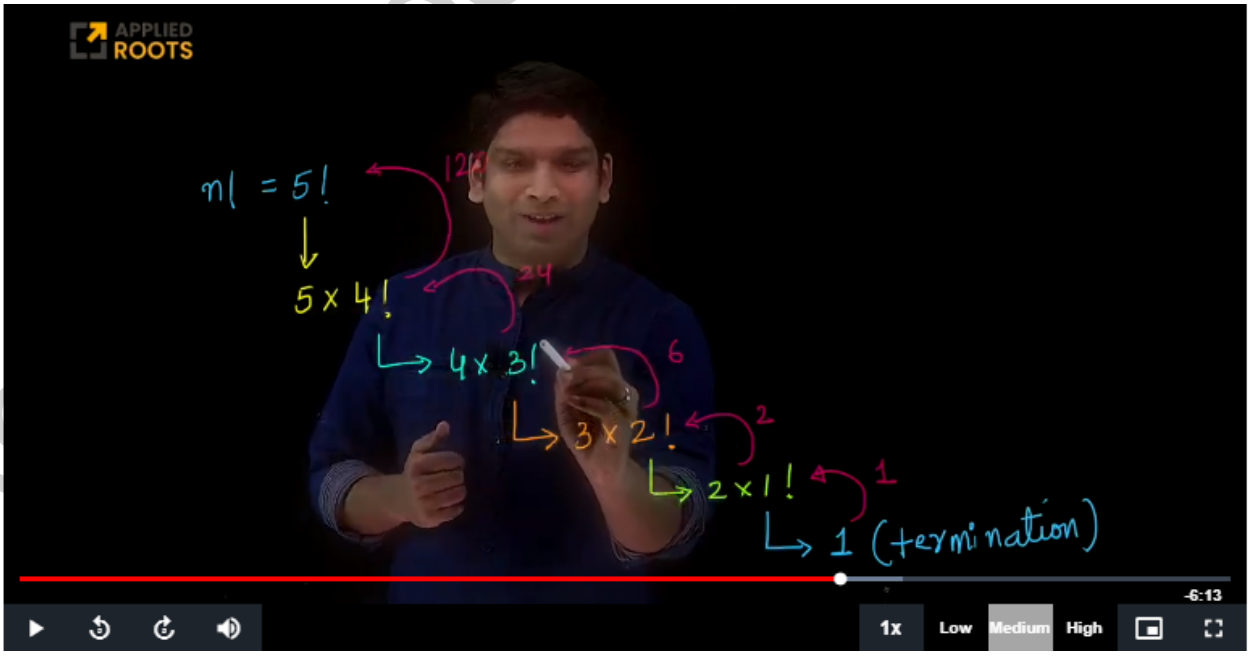
The factorial of a number n using recursive approach is as follows:



Timestamp: 11:25

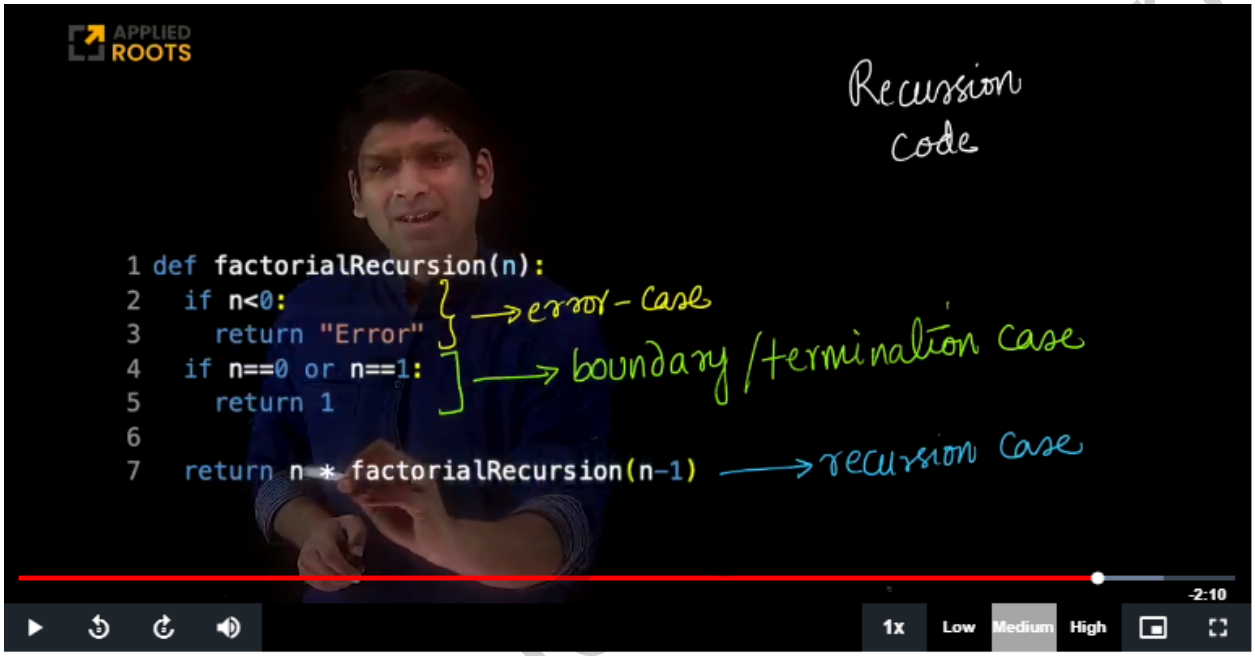
Notice that recursion is followed by breaking up a bigger problem into smaller subproblem(s) and using these smaller subproblems to solve the bigger problem.

As shown in the above figure, n! can be computed by calculating (n-1)! and then multiplying it by n. So the problem of calculating n! has now become the problem of calculating (n-1)! and just multiplying it by n.



Timestamp: 13:05

Notice that in the above figure and in our formulation n=0 and n=1 are called termination or boundary cases since as shown in the above figure, we keep on splitting our bigger problem i.e computation of 5! Into smaller subproblem such as 4! and so on until we reach 1! And then we begin to trace back and solve our bigger problems hence n=0 and n=1 are called boundary or termination cases.

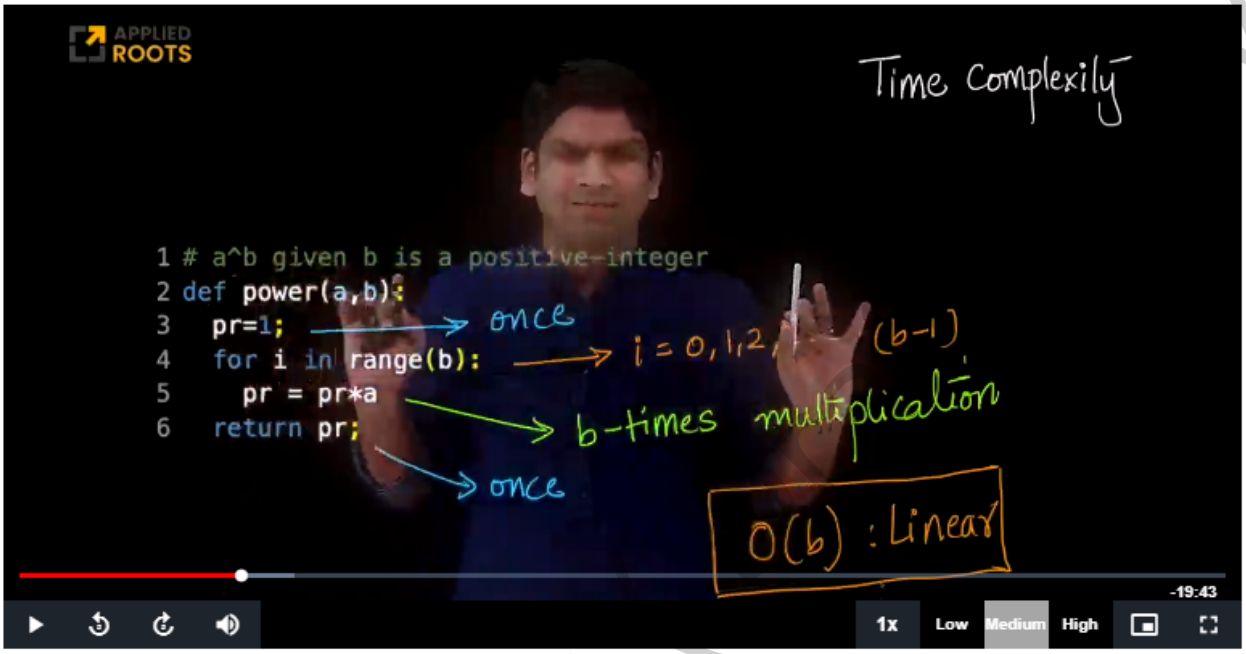


Timestamp: 17:06

In the above figure we can see how to implement factorial using recursion. Notice how we handled our error and boundary cases and how to defined our recursion case.

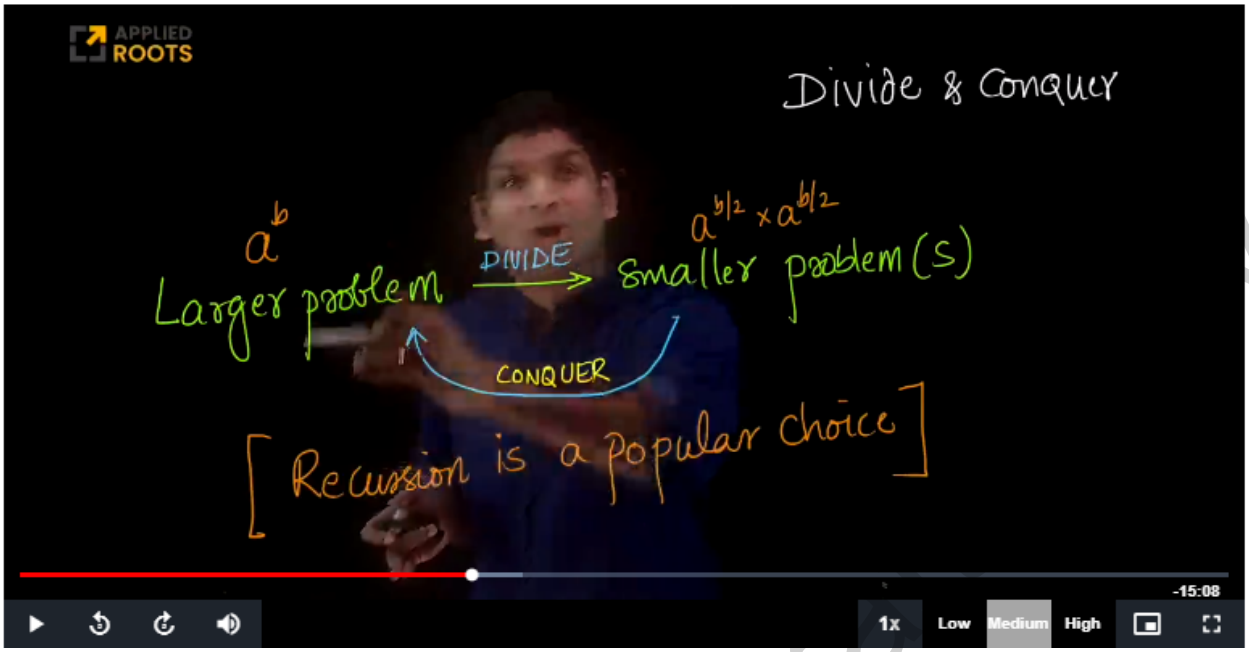
11.2 Power(a,b) using recursion

In this video, we look at how we can calculate a^b using recursion.



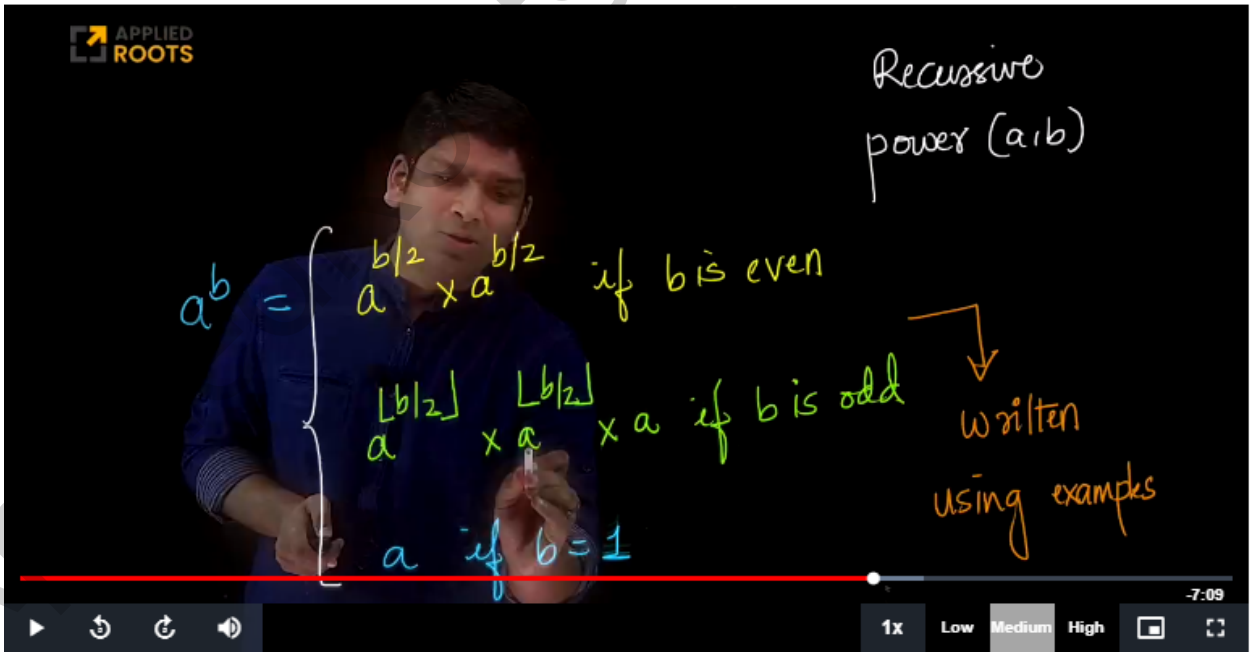
TImestamp: 4:27

In the above figure we can see how can implement b power of a number a (a^b) using iterative algorithm in O(b) time complexity.



Timestamp: 9:02

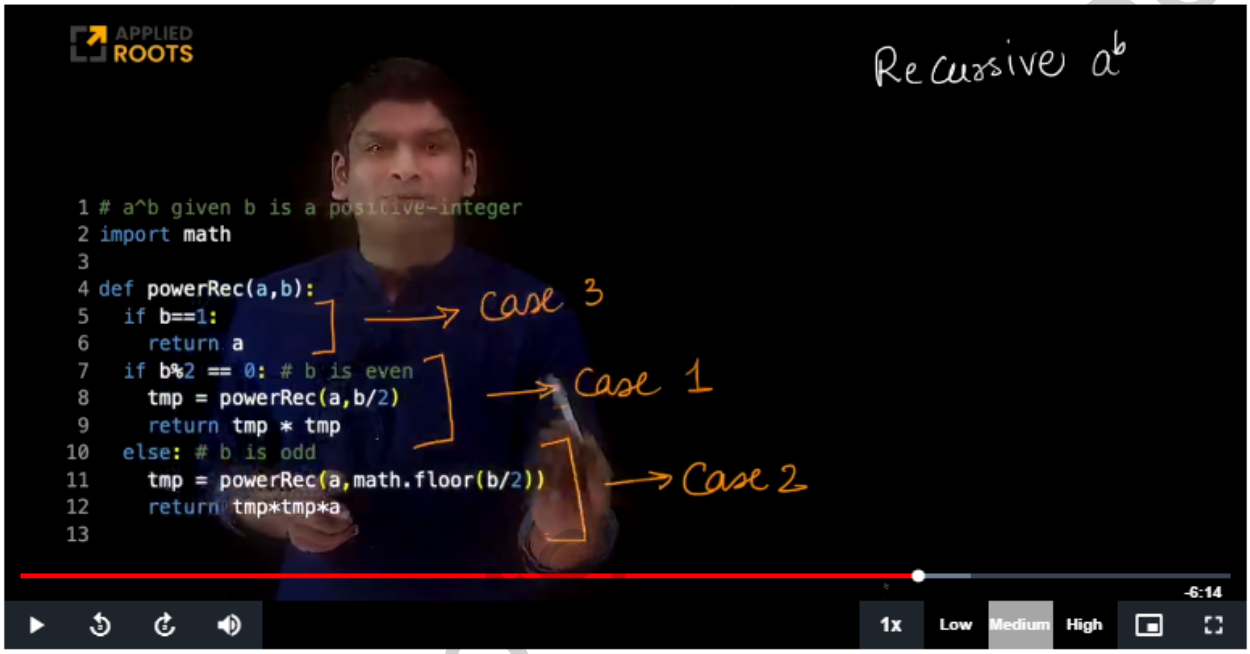
Divide and Conquer is a popular strategy using which we can divide a larger problem into smaller subproblems (divide) and combining them in some form to solve the larger problem(conquer). For implementing this strategy in code, recursion is a popular choice.



Timestamp: 17:02

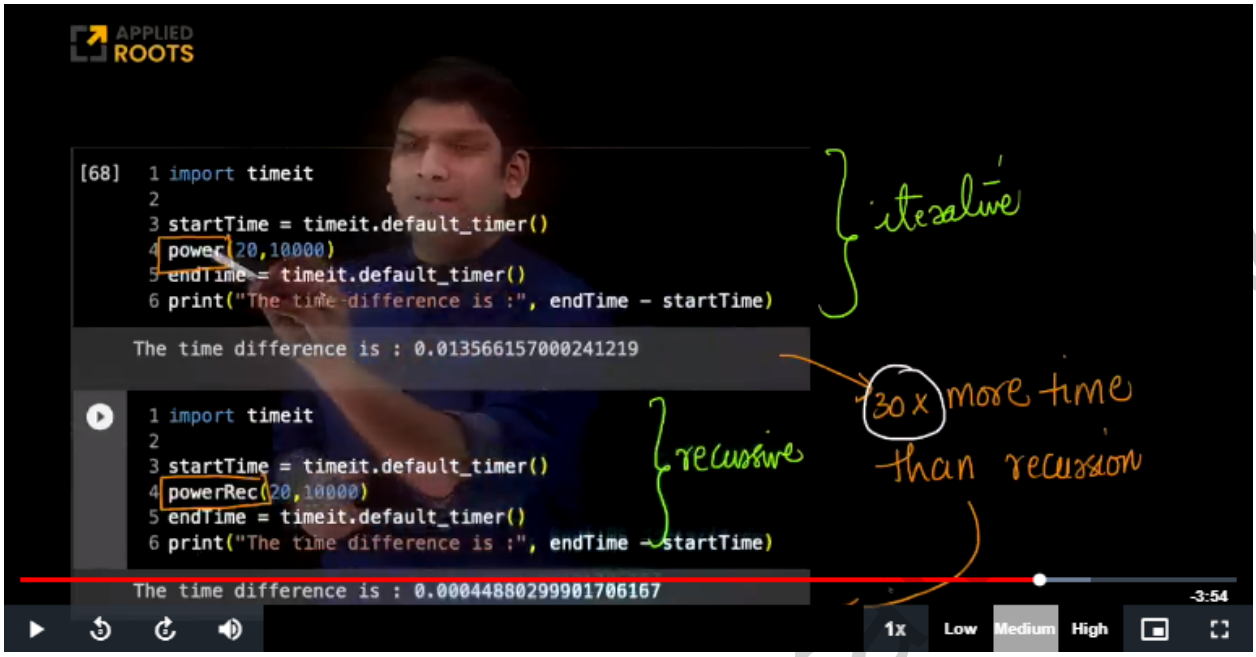
a^b can be solved using recursion as shown above. Notice that ⌊x⌋ for a number indicates floor of x meaning an integer less than x and closer to x. For ex: ⌊2.5⌋ is 2.

Please try for some example values of b, to check it works. Notice that we are solving the bigger problem of calculating a^b by dividing into smaller subproblems of calculating $a^{b/2}$ and $a^{b/2}$ (incase if b is even, it holds similarly when b is odd). Using this strategy once you compute $a^{b/2}$, you need not calculate the second $a^{b/2}$ but can use it directly many times once computed. To understand more let us look at the code implementation.



Timestamp: 17:57

Notice that once we computed $a^{b/2}$ we are simply storing it in tmp variable and using it to compute a^b thus saving performing many multiplications. Notice how we wrote recursion for different cases and how we handled boundary case.

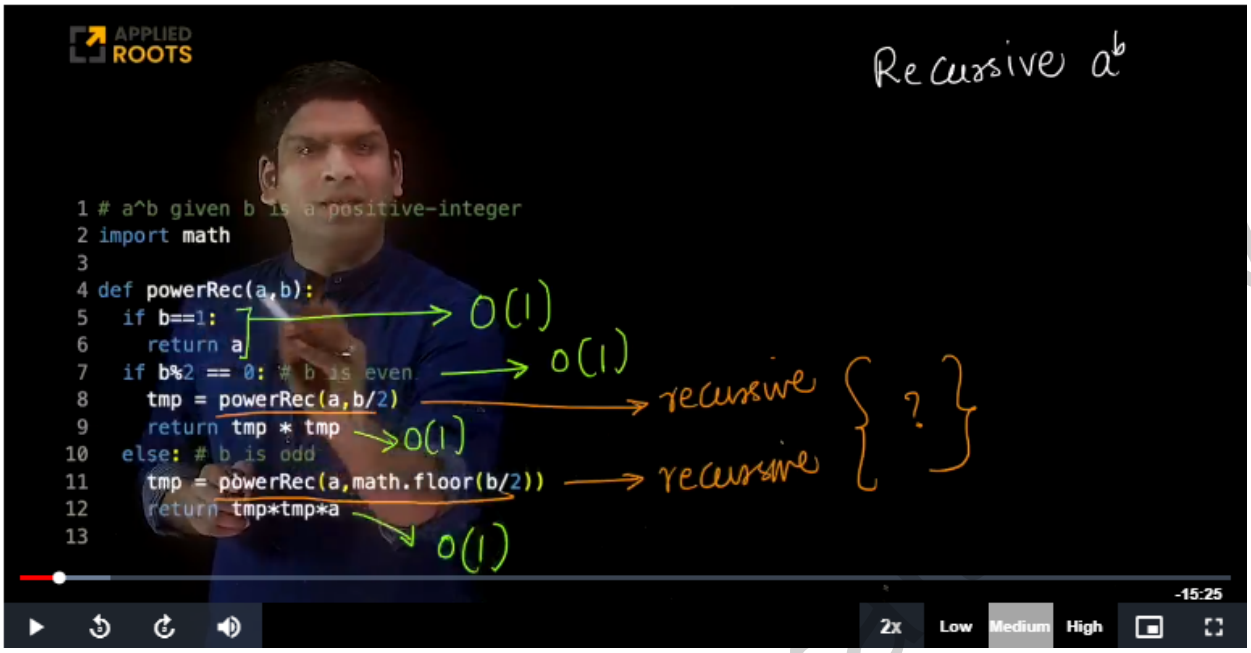


Timestamp: 20:!7

Notice that the time taken for solving the problem iteratively took 0.0135 time where as recursively it took 0.0004 time. This 30x time saving comes from our how we divided our problem into subproblems and using them instead of recomputing and recursion helped us implement it.

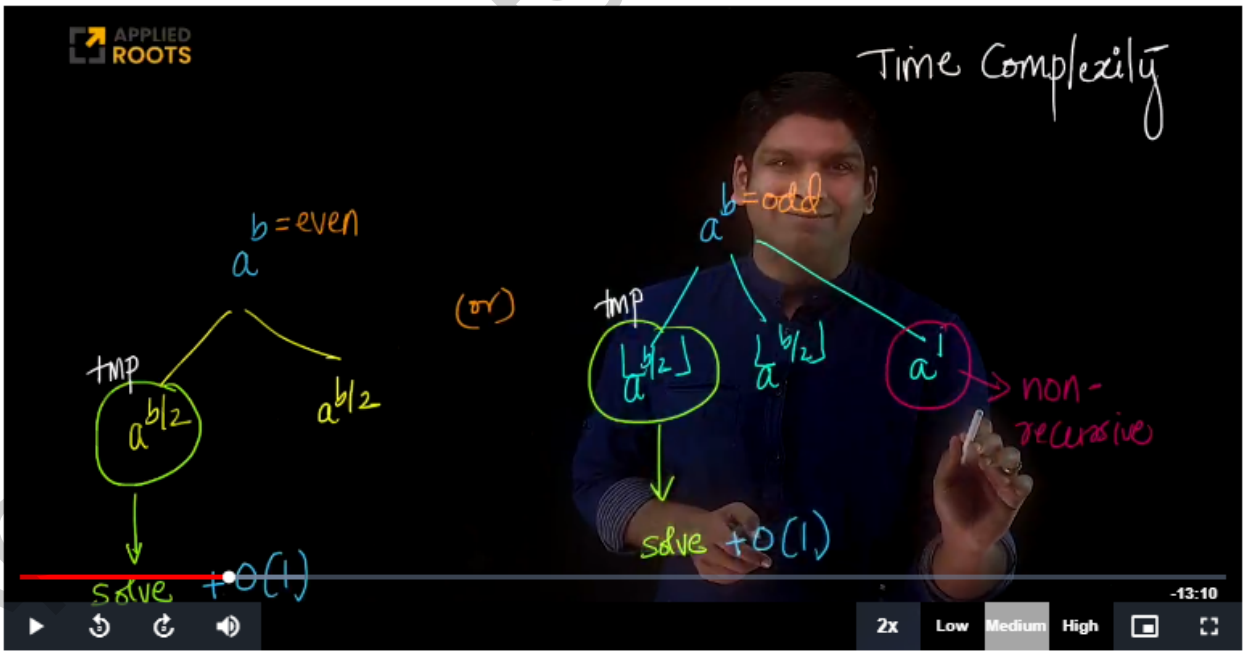
11.3 Asymptotic Analysis using Recursion Tree method

In this video we will perform the time complexity analysis of the above recursive solution to calculate a^b using recursion tree method.



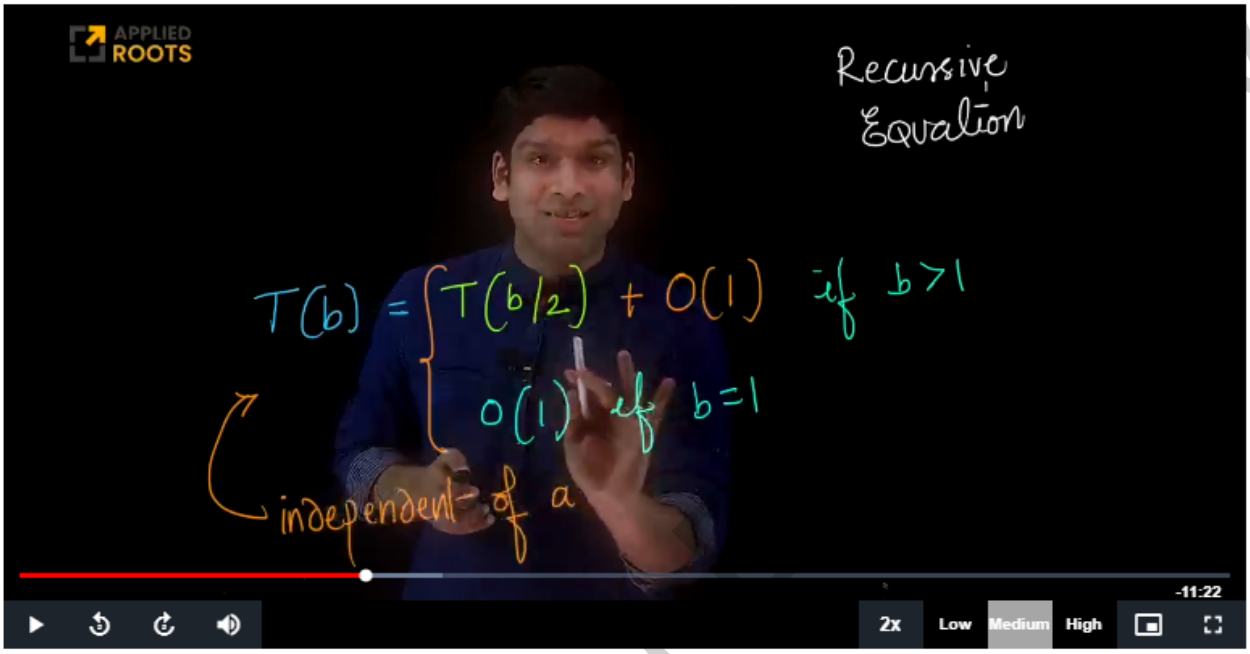
Timestamp: 0:24

Notice in the above figure, in the recursive implementation of a^b, the boundary case when b=1 takes O(1) time and code in line 7,9,12 take O(1) time each. Let us understand the time taken for recursion.



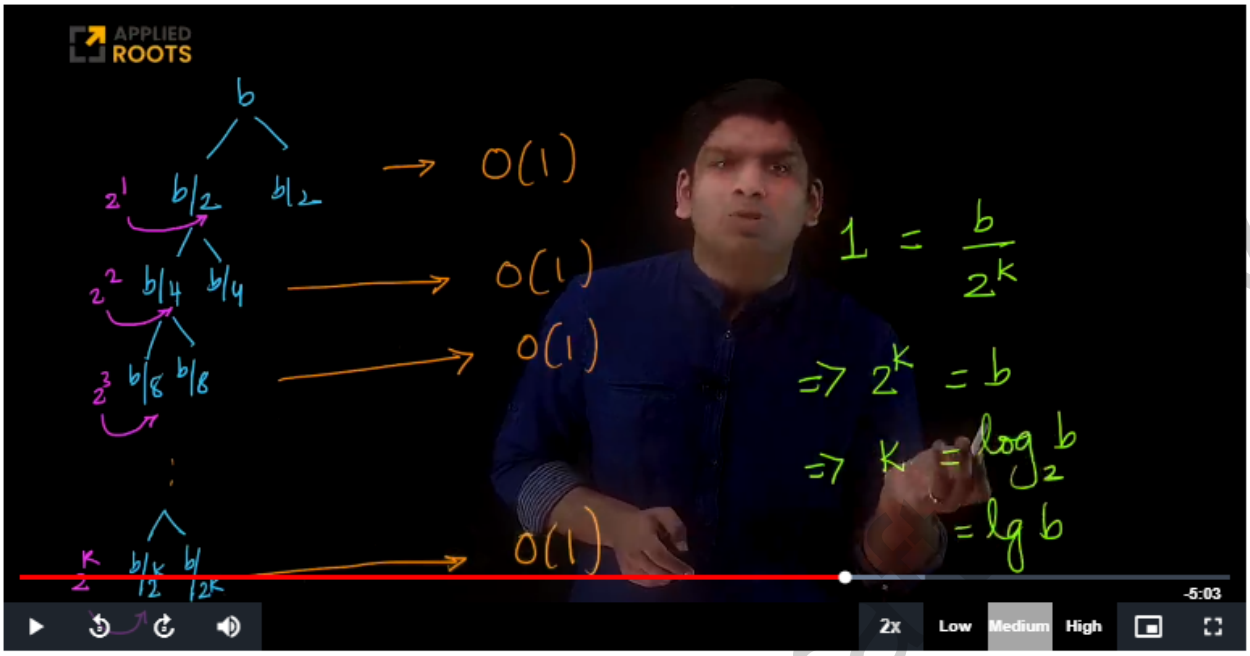
Timestamp: 2:46

Notice that in the above figure the amount of time taken for calculating a^b is the time required for solving $a^{b/2}$ and O(1) time for calculating tmp*tmp (the same can be extended when b is odd since there is only an extra multiplication with a which takes O(1)).



Timestamp: 4:34

Since the time required to solve the problem is independent of a and depends only on b. It can be expressed as in the above figure.

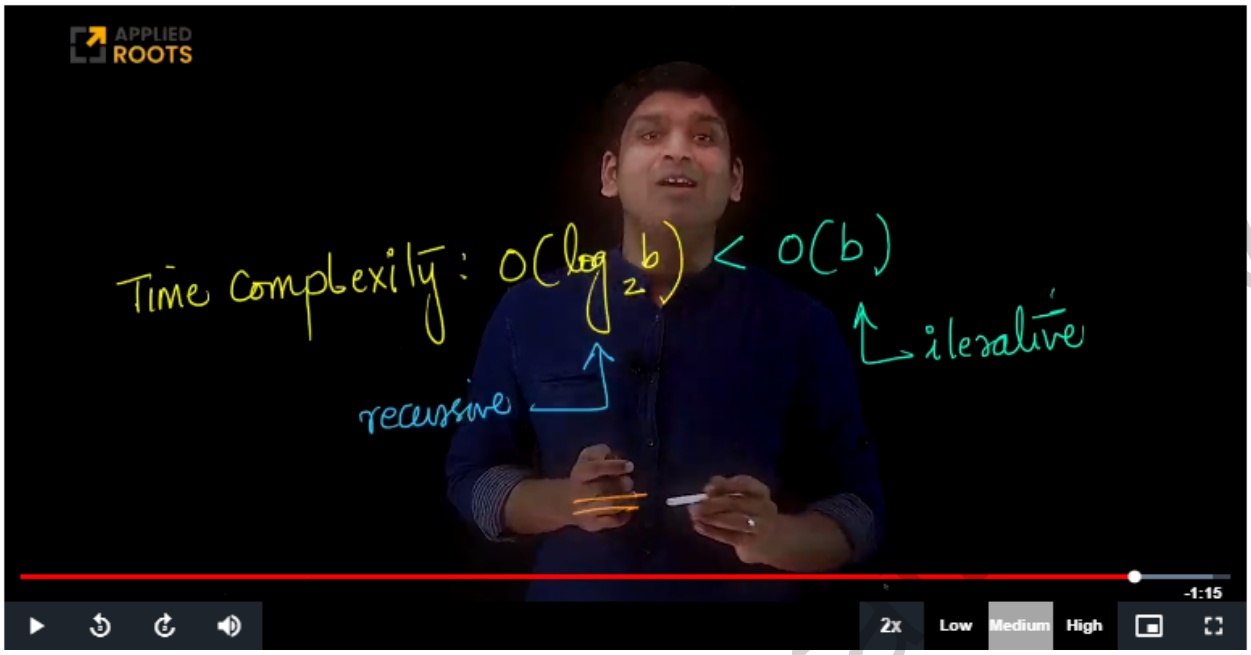


Timestamp: 10:51

Notice that in the above tree representation of our recursive function, at each level we are taking only O(1) time for that level only plus the time taken for the lower levels. The problem of solving a^b is divided into solving $a^{b/2}$ and multiplying with itself, the problem of solving $a^{b/2}$ is further divided into solving $a^{b/4}$ and multiplying with itself and so on.

This process is repeated for K times until we reach the boundary condition where $b/2^k=1$, solving which gives K=lg(b) (note: lg(b) is symbol for representing $\log_2 b$)

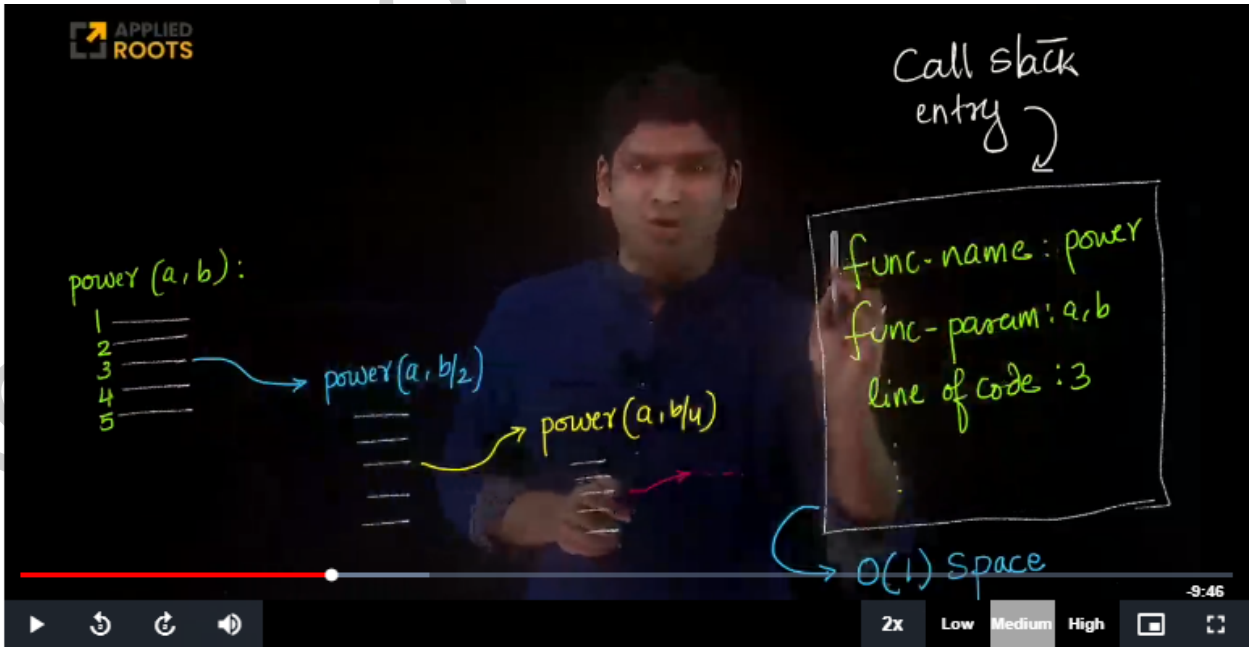
Hence this process is repeated for K times each taking O(1) time and hence the time complexity of recursive implementation of a^b is K*O(1) which O(K) which is O(lgb)



Timestamp: 14:40

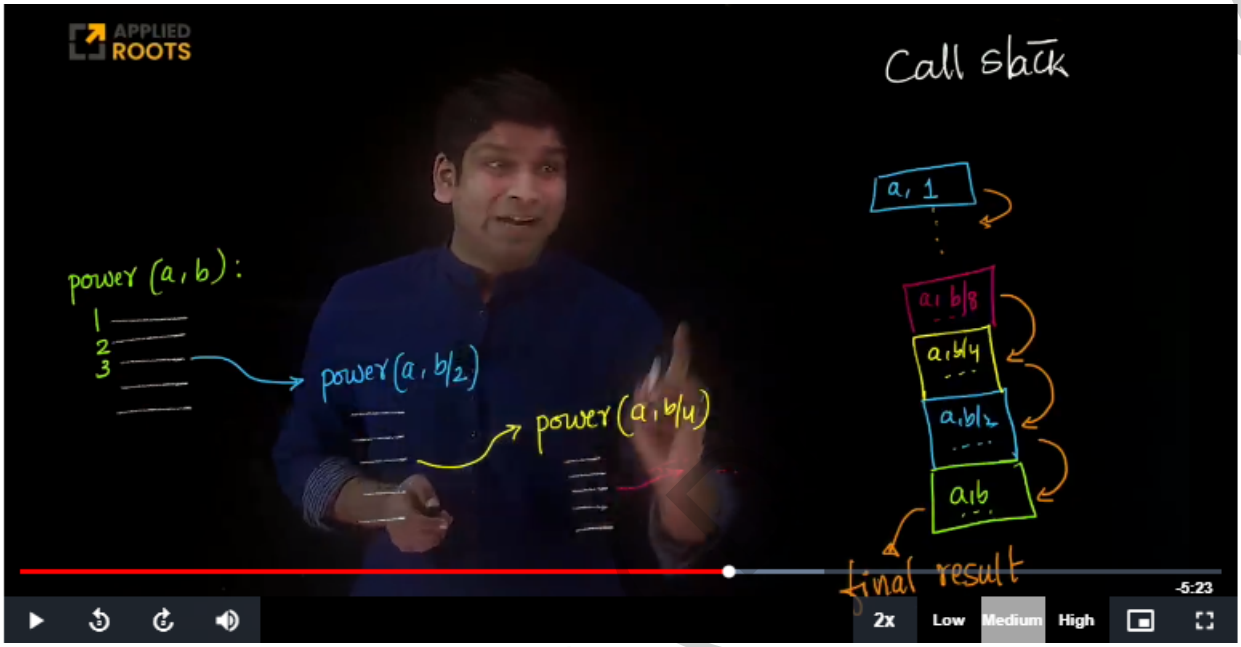
Notice from the above figure that the time complexity of our recursive implementation is less than the time complexity of iterative implementation there by getting us our required speed up of 30x as in the previous video.

11.4 Call-Stack for Recursion



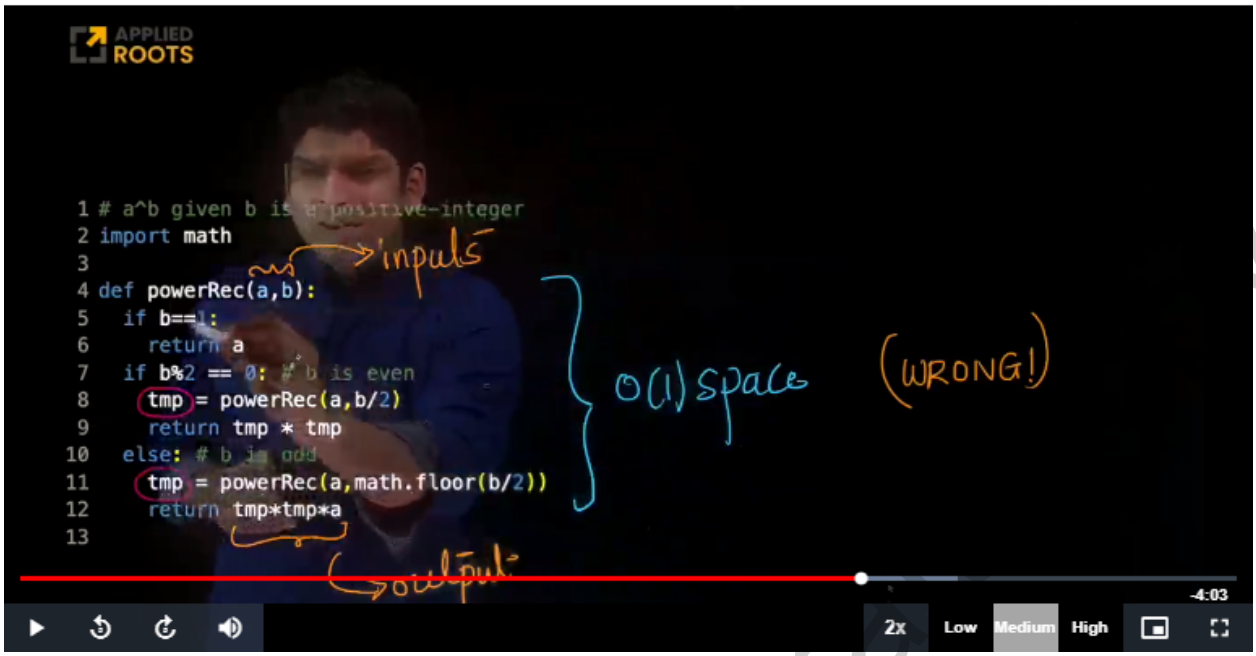
Timestamp: 3:23

Notice as in the above figure for each recursion of our program, a space called call stack entry is created which contains the function name, the function parameters, lines of code,etc which occupies O(1) space.



Timestamp: 7:46

Notice that as in the above figure, for each recursion call a new call stack entry corresponding to the new recursion call is added on top of the previous call stack entry as a stack until the boundary condition is reached. Once the boundary condition is reached and when it comes of out that recursive call, the call stack entry corresponding to that recursive call is deleted until the final result is calculated.



Timestamp: 9:06

When we look at our recursive program it is easy to get confused that we are using O(1) space since it looks like we are only creating an extra variables tmp but this is not the case. We have to account for each call stack entry because even though we are not explicitly writing code to create these call stack entries programming languages such as python implicitly create these entries.

Since in our previous video, we learned that our recursive program has K=lgb recursive calls and since space complexity for each of them is O(1). The total space complexity for our recursive program becomes O(lgb). Remember that for our iterative algorithm the space complexity is just O(1) since we are only creating extra space for storing i and pr.

Hence,
Time complexity of iterative : O(b)
Space complexity of iterative: O(1)

Time complexity of recursive: O(lg b)
Space complexity of recursive: O(lg b)

Hence there is a trade-off between space and time complexity for iterative implementation and recursive implementation. The reason for choosing recursive implementation over the iterative implementation is since ram sizes are growing each day hence space complexity shouldn't be an issue but we want low latency application hence time complexity is much more important.

11.6 Merge-Sort - Why?

Having learnt insertion sort, we know that its worst case complexity is $O(n^2)$, if we can come up with any other sorting algorithm that can solve our sorting problem with a better time complexity it would save our time. Hence we learn merge sort that has O(nlgn) time complexity. Along with reducing time complexity it is also known to work with large sizes of data that cannot be stored in ram but are present in hard disk. Sorting these large sizes of data using insertion sort is not easy.

11.7 Merge-Sort - Intuition

Timestamp: 13:46

As shown in the above figure, merge sort follows divide and conquer approach.
Every divide and conquer approach has two stages ‘divide’: the stage in which bigger problems are divided into smaller subproblems and ‘conquer’: the stage in which these smaller subproblems are solved and combined to get the final result.

In merge sort algorithm in divide stage the given array is broken continuously into smaller arrays of size nearly(when odd size) half until the size of each becomes 1. These smaller arrays are

then combined so that each combination is sorted, so as more and more pieces are combined the greater the size of the sorted array until the final result when the entire array is sorted.

Let us look at how these smaller arrays are combined. Let us say we have smaller arrays A=[5,6] and B=[1,3] let us say we store the result in array C.

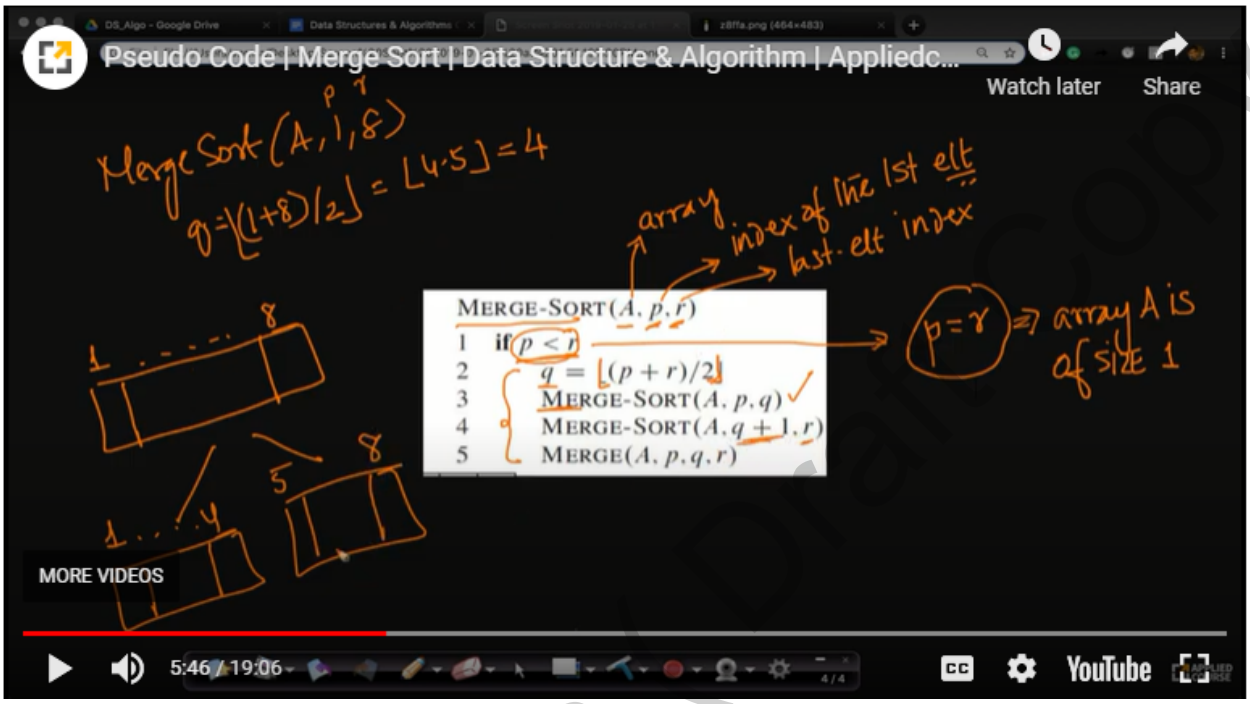
First we compare the first elements of A and B and the smaller one is stored in C.
So now A=[5,6] , B=[3] and C=[1].

Next we compare again the first elements of A and B and the smaller one is stored in C.
So now A=[5,6] , B=[] and C=[1,3].

Now since there are no elements in B all the elements in A are added to C.
So now C=[1,3,5,6]. This is done using markers i,j for arrays A,B to note the first element in each array.

Merge sort names comes from how we are merging these smaller arrays to former larger arrays. Please refer to the merge sort gif in wikipedia to understand more.

11.8 Merge-Sort - Pseudo Code



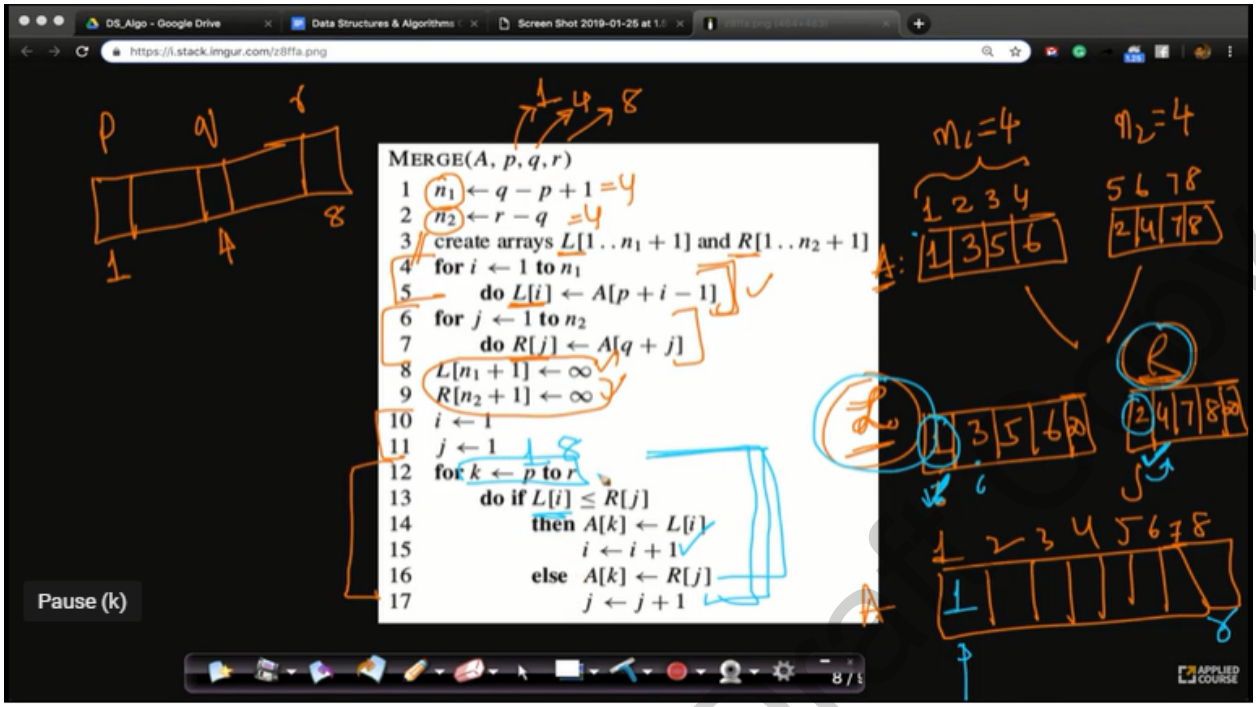
Timestamp: 5:46

The pseudo code for merge sort looks as in the above figure.
A is the given array to be sorted, p is the index of the first element and r is the index of the last element.

So p<r condition checks whether the size of the array is 1 (since if array contains only one element then p=r). If the size of array is 1 then it is already sorted otherwise we have to perform the following operations.

i) merge sort on the first half of the array
ii) merge sort on the second half of the array
iii) merging both the halves.

The below figure shows the pseudo code for merge operation.



Timestamp: 16:42

N1,n2 store the size of each smaller array.

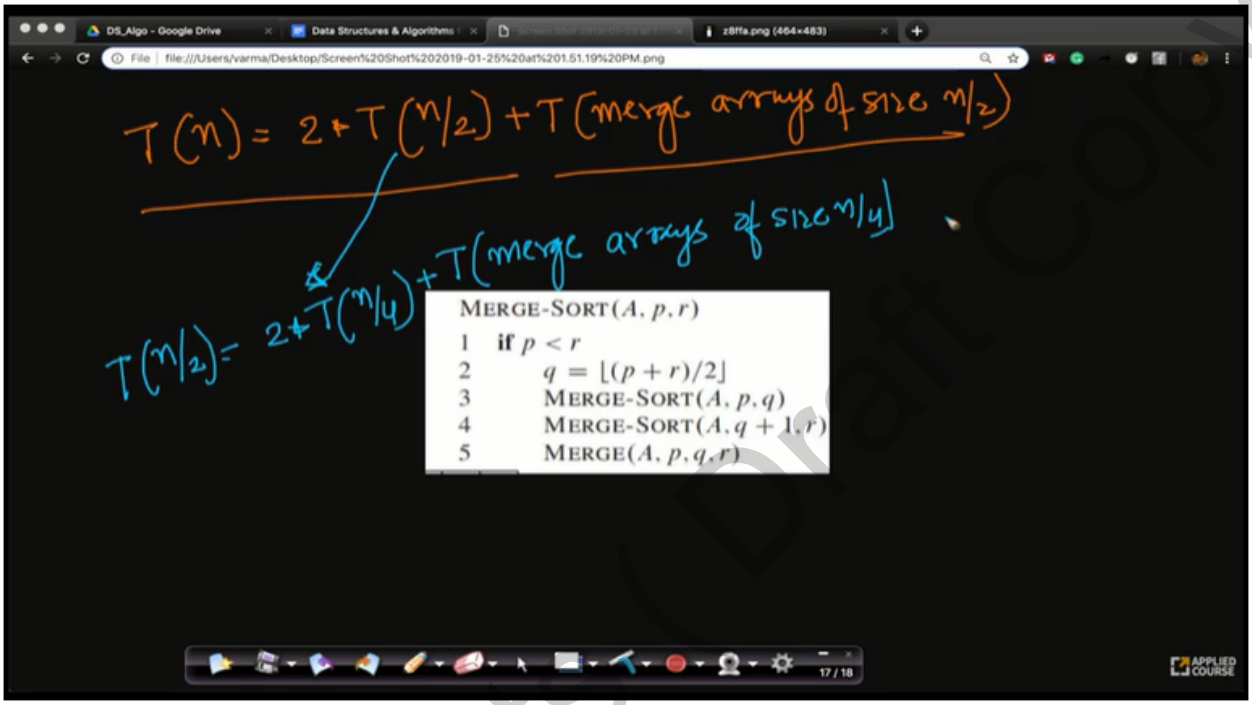
We are creating separate arrays for left half and right half using the elements of the main array so that we don't overwrite the main array

We are adding infinity at the end of both arrays so that when one of the array completes we still have to compare the firsts of both arrays and add the least one. Since infinity is greater than all numbers so once one array is completed we can continuously add elements from the other array.

We fill the original array starting from index p with the smaller of firsts of both arrays until we have filled index r.

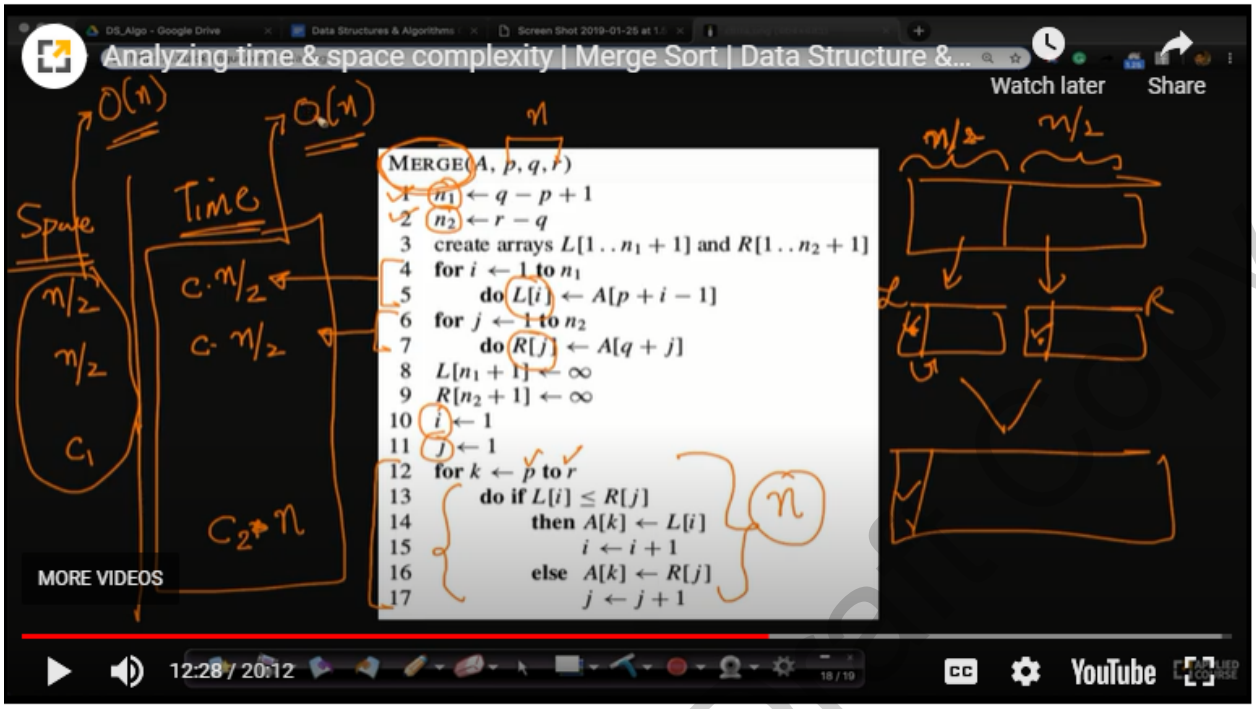
The bigger picture is how merge sort is dividing bigger array to smaller arrays and then combining them in sorting order thus sorting the bigger array.

11.9 Merge-Sort - Analysing time and space complexity



Timestamp: 6:40

As shown in the figure, due to the recursive equation of merge sort, the time take for sorting array of size n T(n) is the sum of time taken to sort each of the smaller arrays 2* T(n/2) (T(n/2) each due to size n/2) as well as the time taken for merging two arrays of size n/2 T(merge arrays of size n/2), similarly it can be extended to arrays of size n/4 and so on.

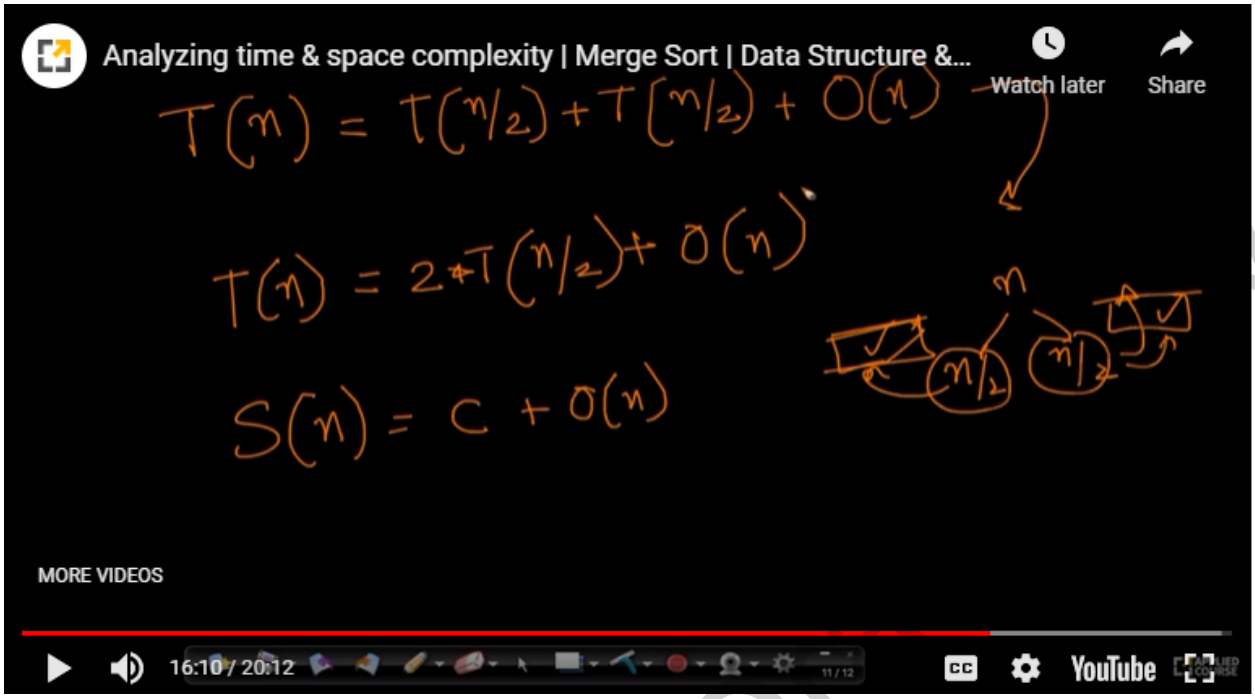


Timestamp: 12:28

From the above figure we can see that for merging two arrays of size n/2, we need some constant c*n/2 time for creating array L, c*n/2 time for creating array R and for running the loop some constant c2*n times to fill the original array A.

For space complexity the extra space that is created is for some variable such i,j which O(1) and for creating arrays L, R each of size n/2. So the space complexity is 2*O(n/2) +O(1) which O(n).

Hence the total time complexity for merge operation is O(n) while the space complexity is also O(n).



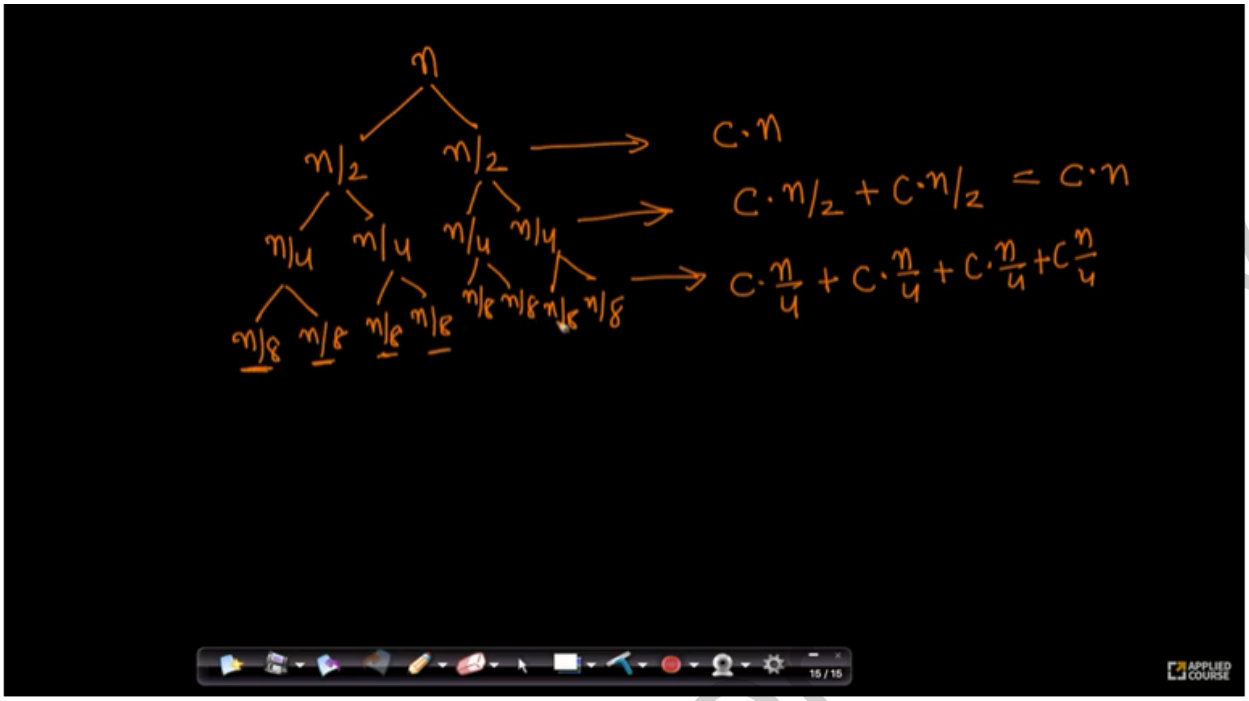
Timestamp: 16:10

So the total space complexity of merge sort is O(1)+O(n) =O(n) (O(1) for storing q in the pseudo code).

The total time complexity becomes

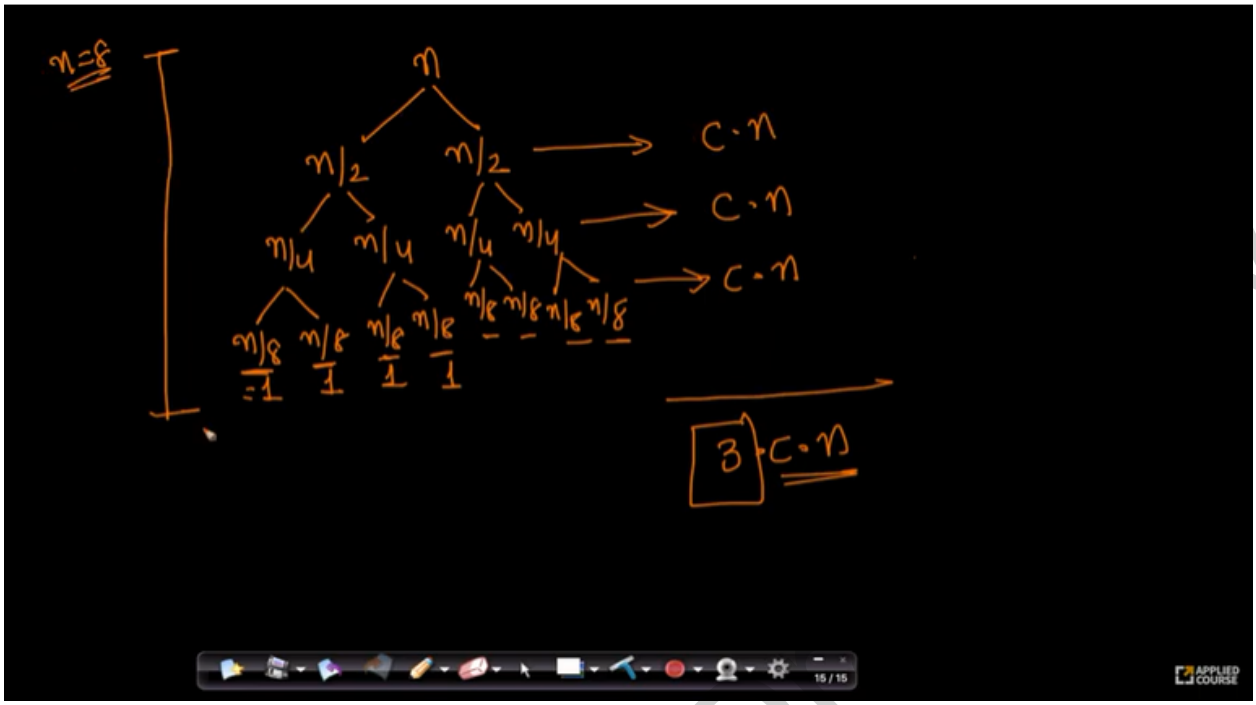
T(n)=2*T(n/2) + O(n)

11.10 Merge-Sort - Recursion Tree Method



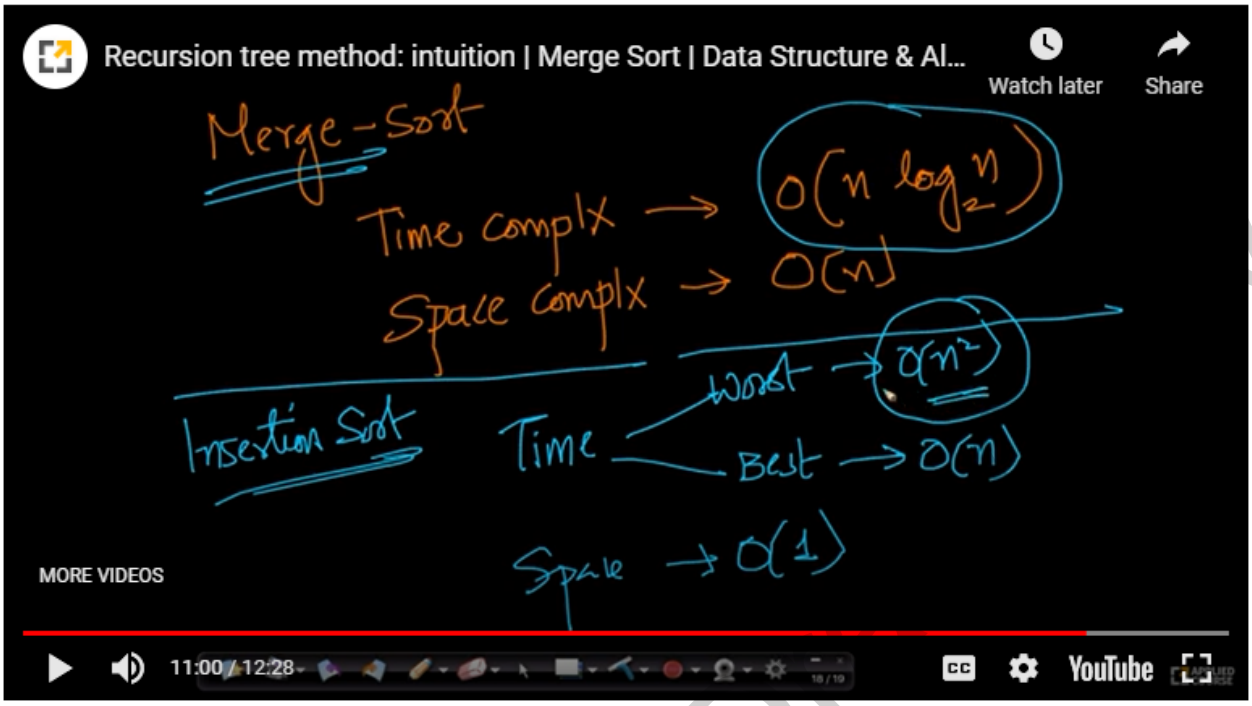
Timestamp: 3:08

From the above figure we can see that at each stage for merge operation the time taken c*n. For example at level 3 when merging 4 arrays of size n/4, we have to merge first two arrays in c*n/2 time and the rest two arrays in c*n/2 time. Hence the total time at that level = c*n/2 +c*n/2 =c*n.



Timestamp: 4:47

We can see from the above figure that for 8 elements we have three levels, similarly if we can check for 16 elements we have 4 levels. If we can observe that if we have n elements then we have $\log_2 n$ levels. Hence the total time complexity for merge sort is lgn*O(n) =O(nlgn) and space complexity is O(n).



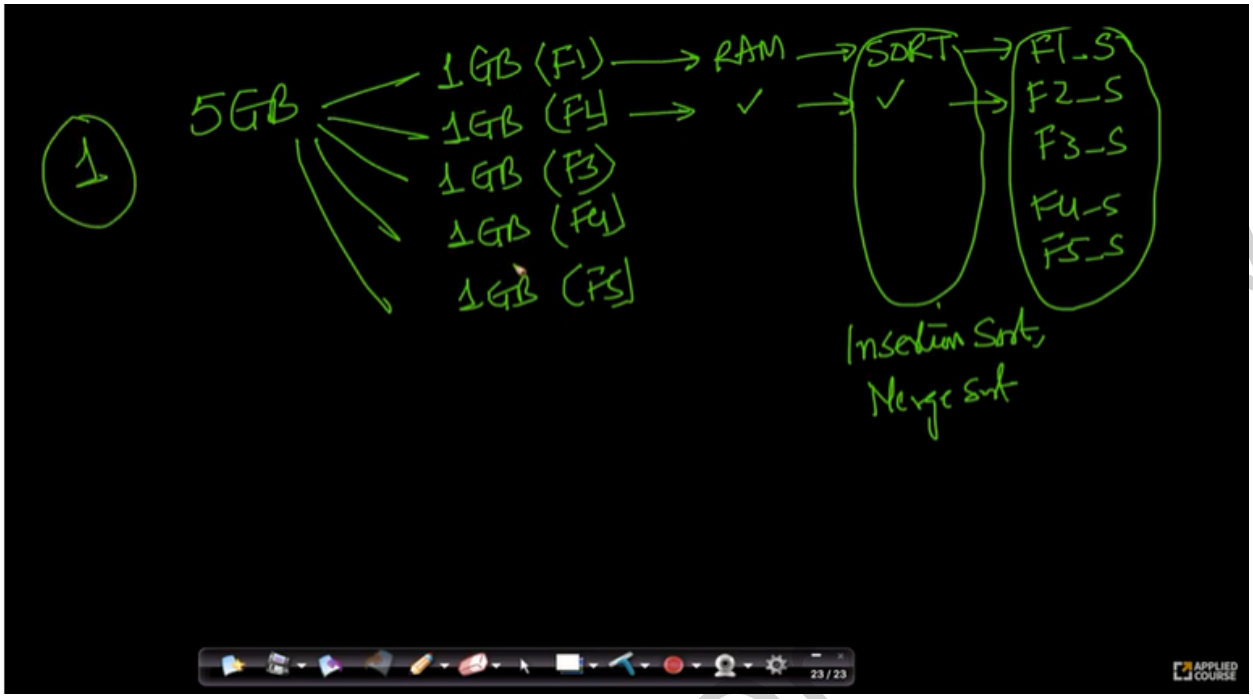
Timestamp: 11:00

Comparing insertion sort and merge sort, the best case time complexity of insertion sort is better than merge sort but the worst case time complexity of insertion sort is worse than that of merge sort.

The space complexity of insertion sort is O(1) while the space complexity of merge sort is O(n). We have to choose the appropriate sorting algorithms based on requirement.

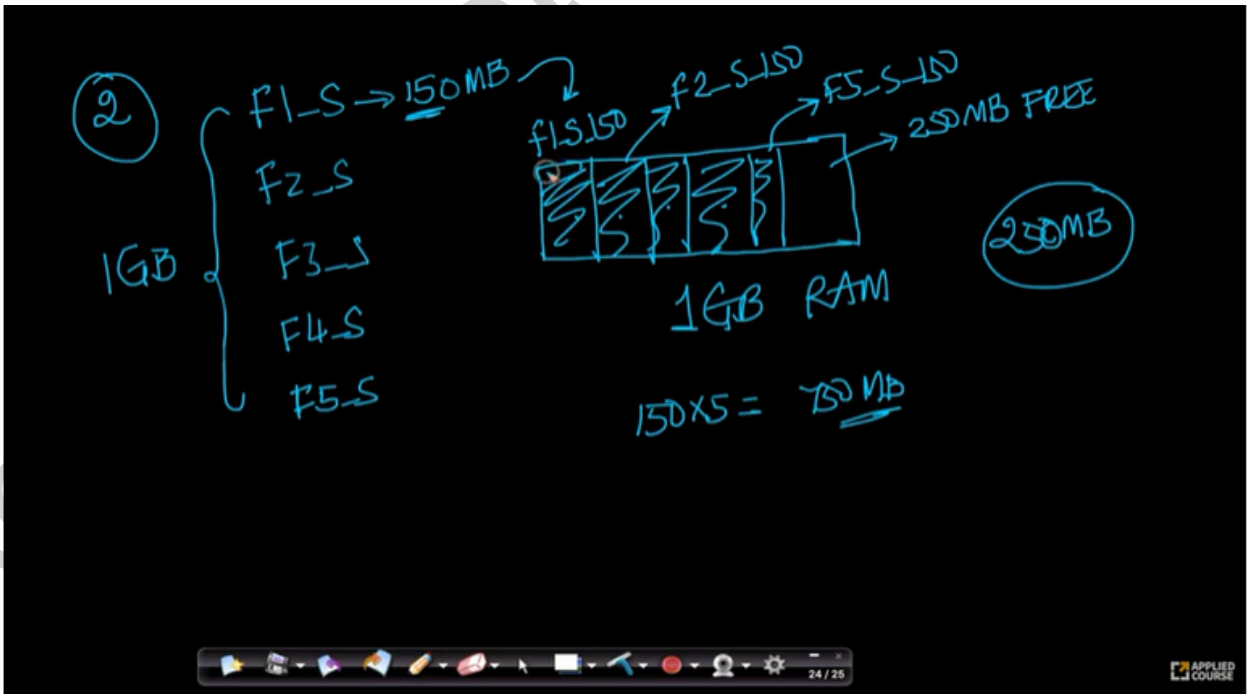
11.11 Merge-Sort - External Sorting

External Sorting is done when the entire data that needs to be sorted cannot be loaded in the ram. In this case we will use external storage such as hard disk hence the name External Sorting.



Timestamp: 7:50

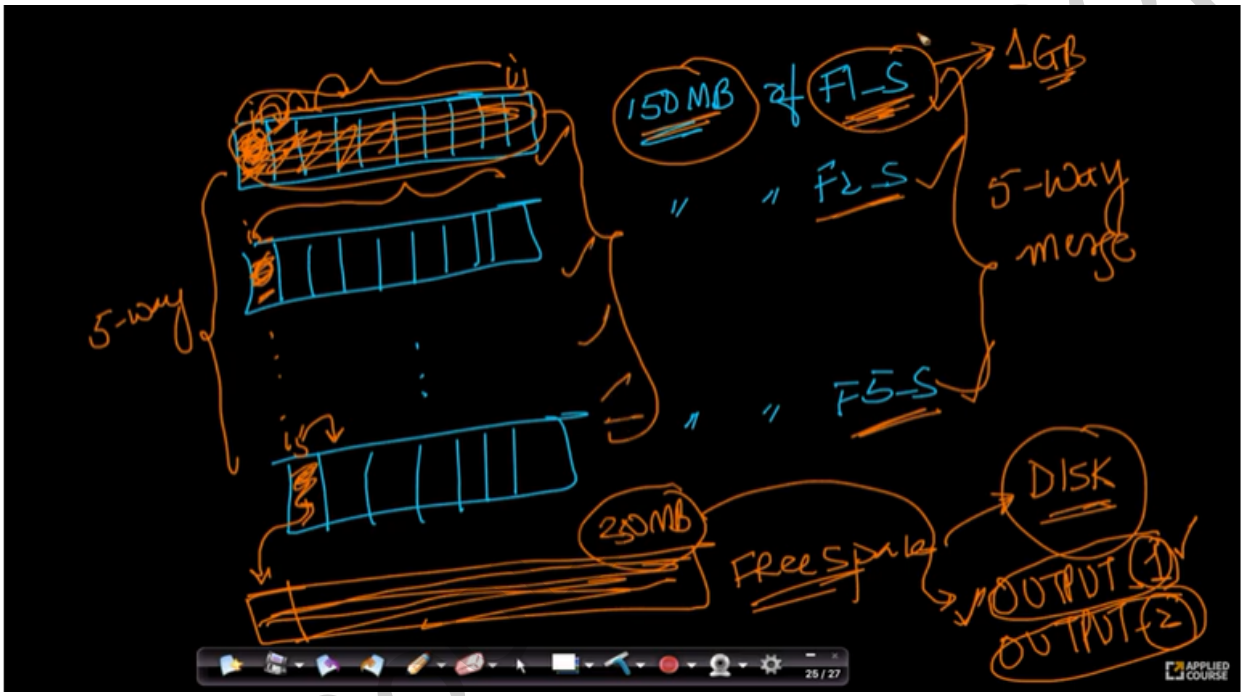
Step1) Suppose if you have a 5 gb file the first thing we have to do is to split the 5gb data into smaller arrays that can fit into the memory and apply any sorting algorithm (considering the space complexity) to sort them individually and store in files F1_s,F2_s,F3_s,F4_s,F5_s.



Timestamp: 10:57

step2) Suppose we have 1gb of memory, then since we have already sorted parts of data and stored in files each of size 1gb in the disk. We can now pick the first 150mb of data from each file and load it in memory and use the remaining 1000-150*5=250 mb of memory to perform our operations. We now perform 5-way merging using these already sorted 5 arrays.

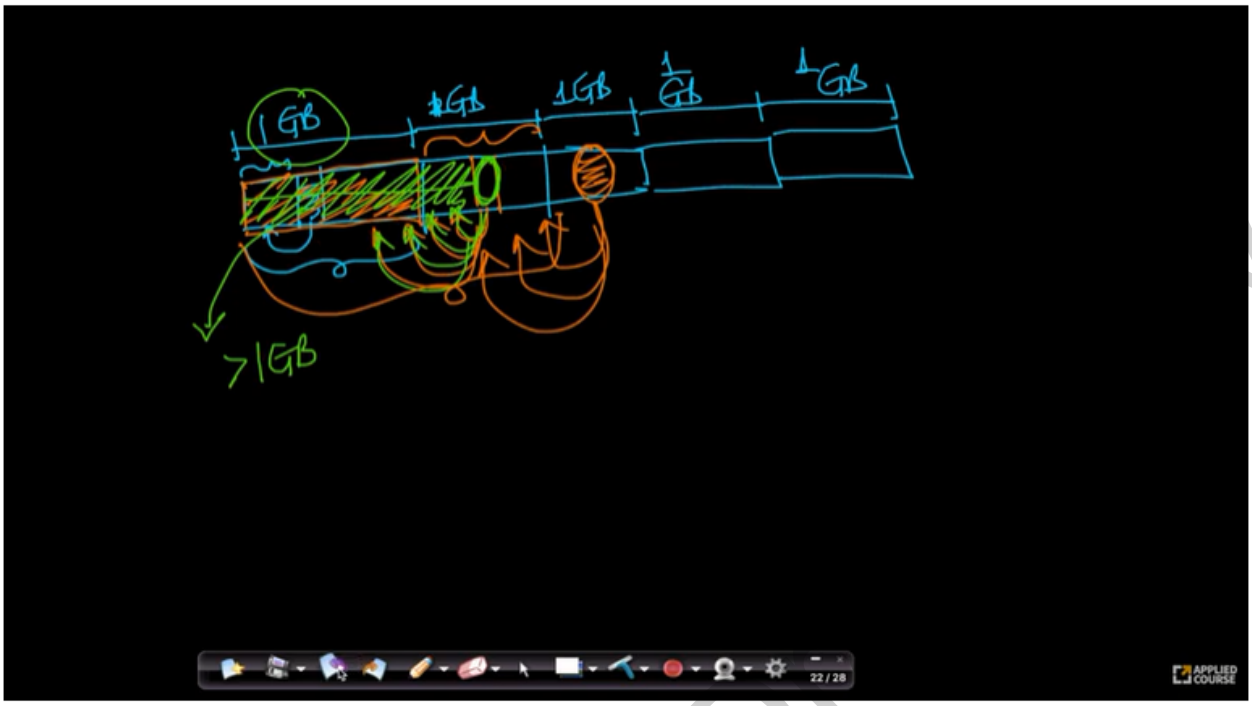
5-way merging is done similar to how merge is done in merge sort. But instead of combining 2 arrays we combine 5 arrays by combining first elements of each arrays and picking the smaller one and placing it in the output array which occupies the above said 250 mb of memory and updating the markers on the array.



Timestamp: 17:41

So once we find these 250 mb of sorted elements, this array is stored to disk and the 250 mb memory is freed. We then again start filling the next 250 mb and again store it in to disk.

Once all the elements in any of the 150mb array are done, we can load the next 150mb of data from the corresponding file. This continues until all the data of all the files is now finally merged and the output is stored in the disk.



Timestamp: 18:56

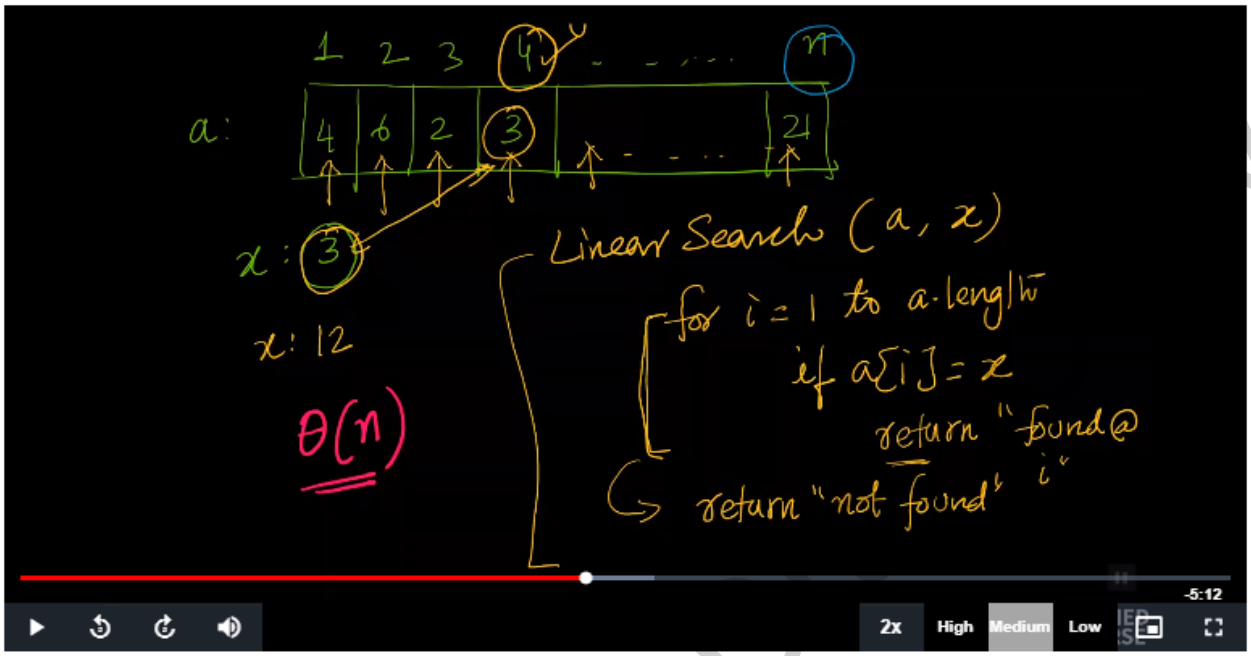
Insertion sort cannot perform external sorting easily because with insertion sort if the data is divided and sorted. Then each of the elements in the second array should again be compared with each element in the first array the way insertion sort works.

So insertion sort is not an external sort but merge sort using k-way merging is an external sorting algorithm because of its divide and conquer approach.

Please also refer to this link https://en.wikipedia.org/wiki/External_sorting#External_merge_sort.

External sort is helpful in real-world scenarios like sorting all of amazon products by price.

11.12 Linear Search



Timestamp: 4:34

Search is the problem of finding whether an element x is present in an array a. Linear search is a simple searching algorithm that compares each element of the array a with the search the element x as shown in the above figure.

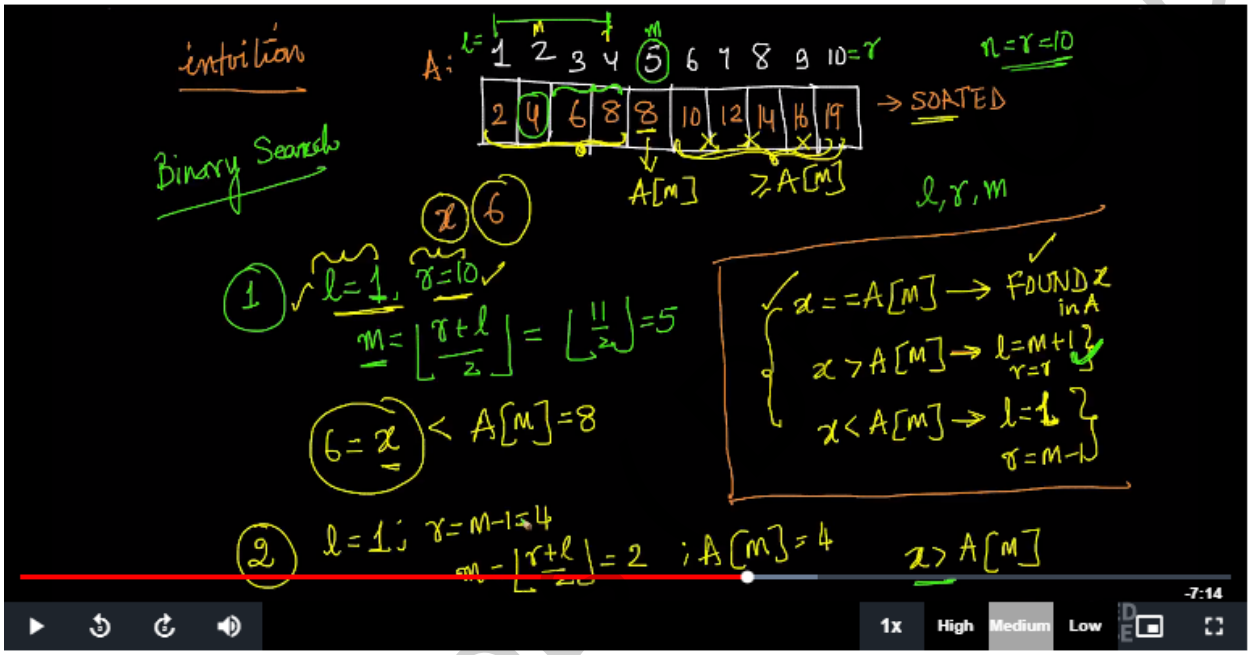
If the element is present in the array we will return the index at which the element x is present in the array a. If x is not present in the array a then it will return array is not found.

There is an another variation of the problem where we want to return all the indices where x is present. For that problem we may need to maintain another array to which we will add the indices at which x is present as we move along the array a comparing each element with x.

So the time complexity of linear search is Θ(n) where n is the size of the array since we loop through each element of the array and the space complexity of linear search is Θ(1) since we are only one additional looping variable i apart from the data given.

Linear search can work on linked list as well as we are comparing the elements with the search element x sequentially.

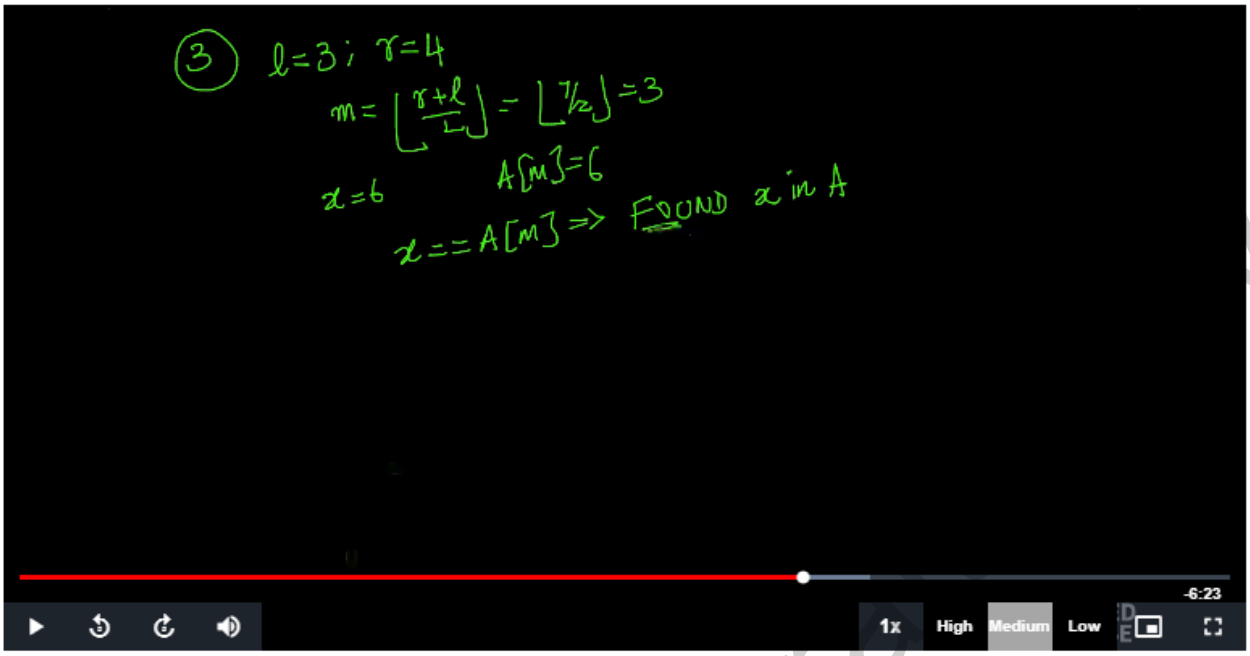
11.13 Binary Search -- intuition



TImestamp: 10:54

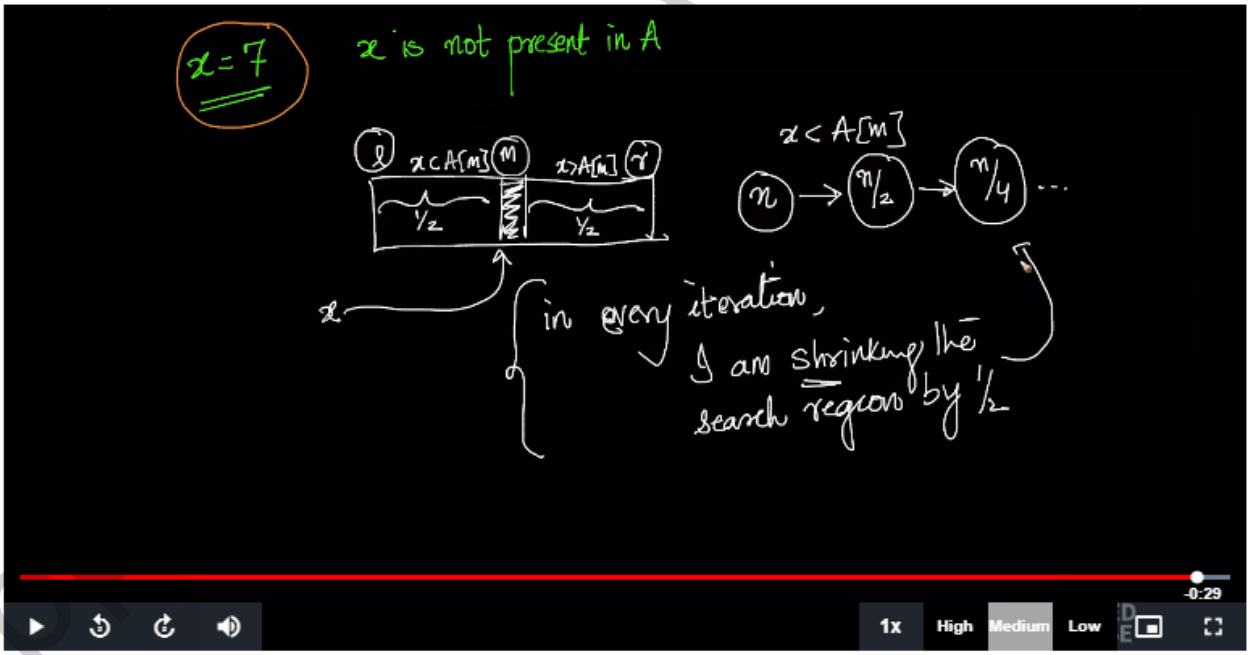
To perform binary search we need the array A to be sorted. To search an element x in an array A in binary search at each iteration
i) we will check whether the middle element in the array is the search element x.
ii) If the middle element is not the search element, then we will compare whether the search element is greater than the middle element, it it is greater then we will search the right half of the array.
iii) If the search element is less than the middle element then we will search the left half of the array.

We repeat this process until we find the element or we are only left with one element array and the element is not the search element in which case the search element is not present in the array.



Timestamp: 11:45

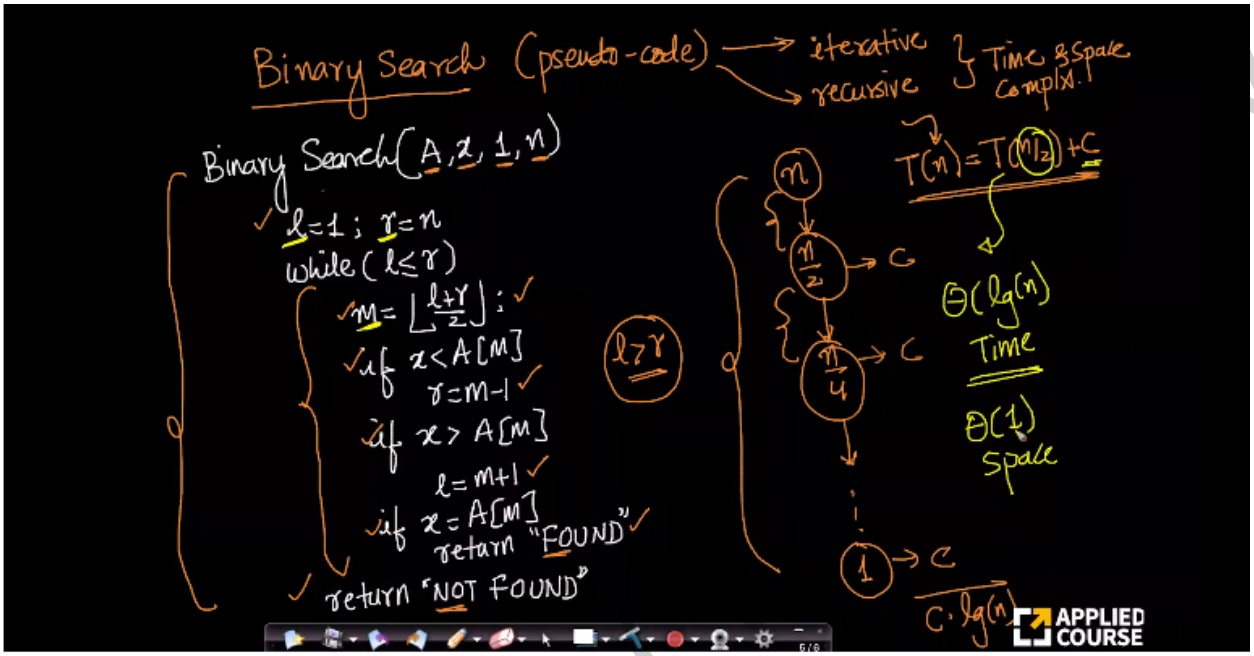
Please work through the example in the above figure where search element x=6.



Timestamp: 17:39

Notice that for every iteration we are shrinking our search region by half as shown in the figure.

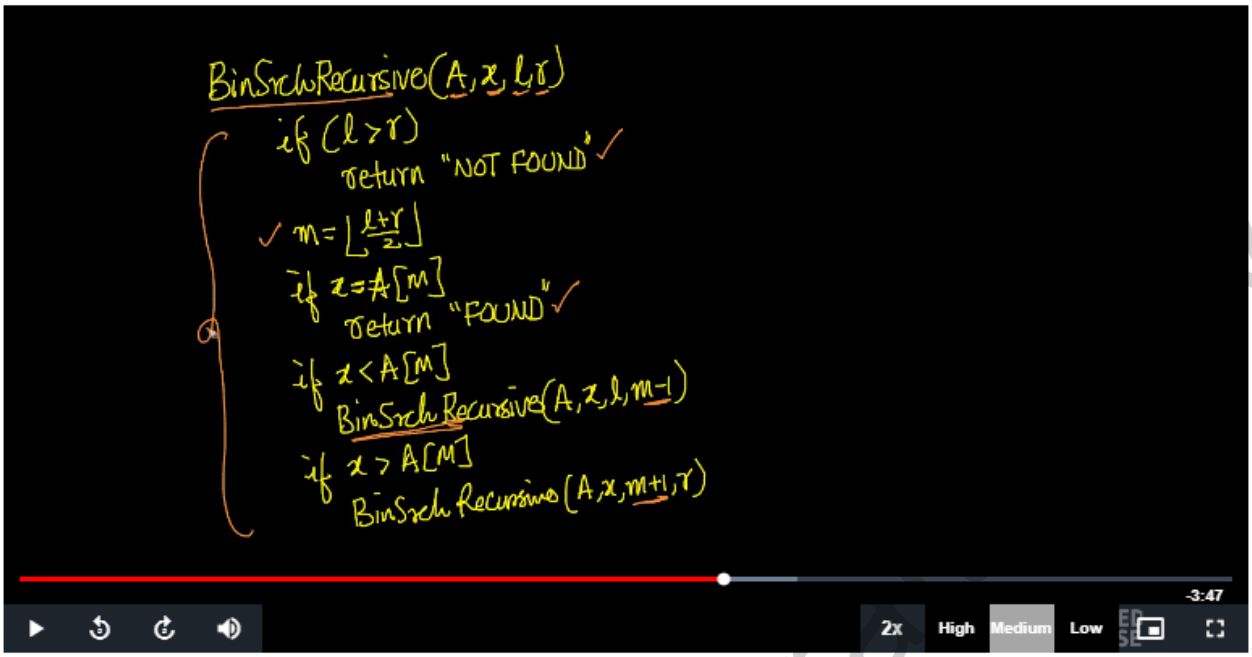
11.14 Binary Search -- Pseudo Code



Timestamp: 4:27

In the above figure we can see that at iteration of the loop, we are performing constant amount of operations such as calculating the middle index m, comparing search element with the middle element,etc hence each iteration the time complexity is Θ(1). Notice that since at each iteration we are reducing the search space to half until we reach 1, hence the number of iterations for searching an array with n elements is lgn. Hence the total time complexity is Θ(lgn).

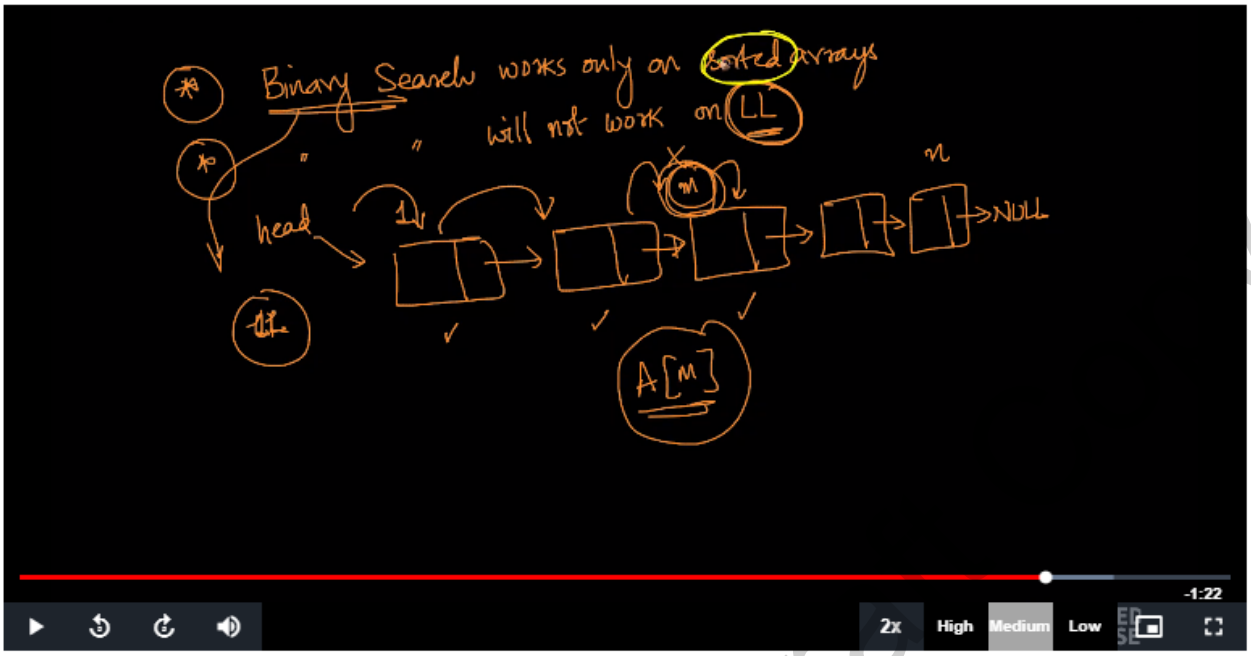
Notice that throughout the entire pseudo code we are only creating additional variables like l,r,etc which doesn't depend on n, hence the space complexity is Θ(1).



Timestamp: 5:16

The above figure shows the pseudo code for the recursive implementation of binary search. Note that at each recursive call the amount of time taken for a search space of n is the sum of constant time taken for checking the if conditions and the time taken for the next recursive call with search space n/2.
$T(n) = T(n/2) + \Theta(1)$

Solving this should once again give us the time complexity of binary search to be $\Theta(\lg n)$.

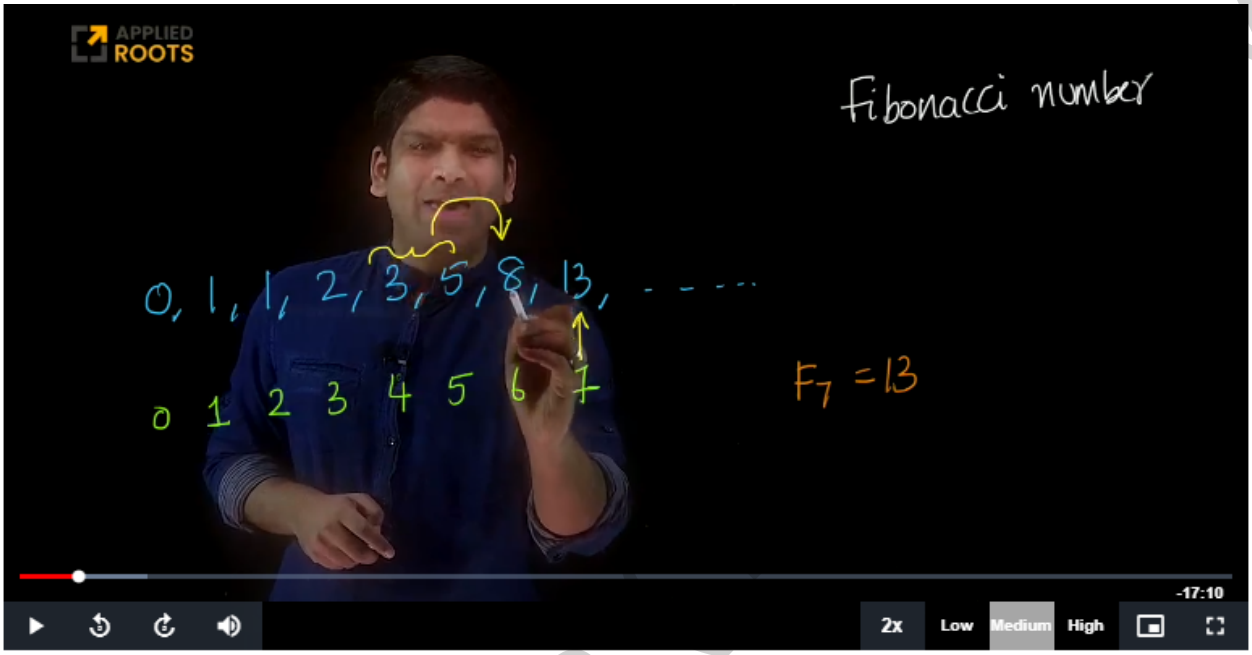


Timestamp: 7:41

Binary search will work only on sorted array since in binary search we reduce our search space by half after comparing with the middle element because if the array is sorted and the search element is less than the middle element(let's say) then the element can only be found in the first half. Since it is less than the middle element there is no way it can be less than the elements that are greater than the middle element.

Binary search doesn't work in the case of linked list since in binary search using arrays we are directly accessing the middle element of the array in constant time, we cannot do the same incase of linked list.

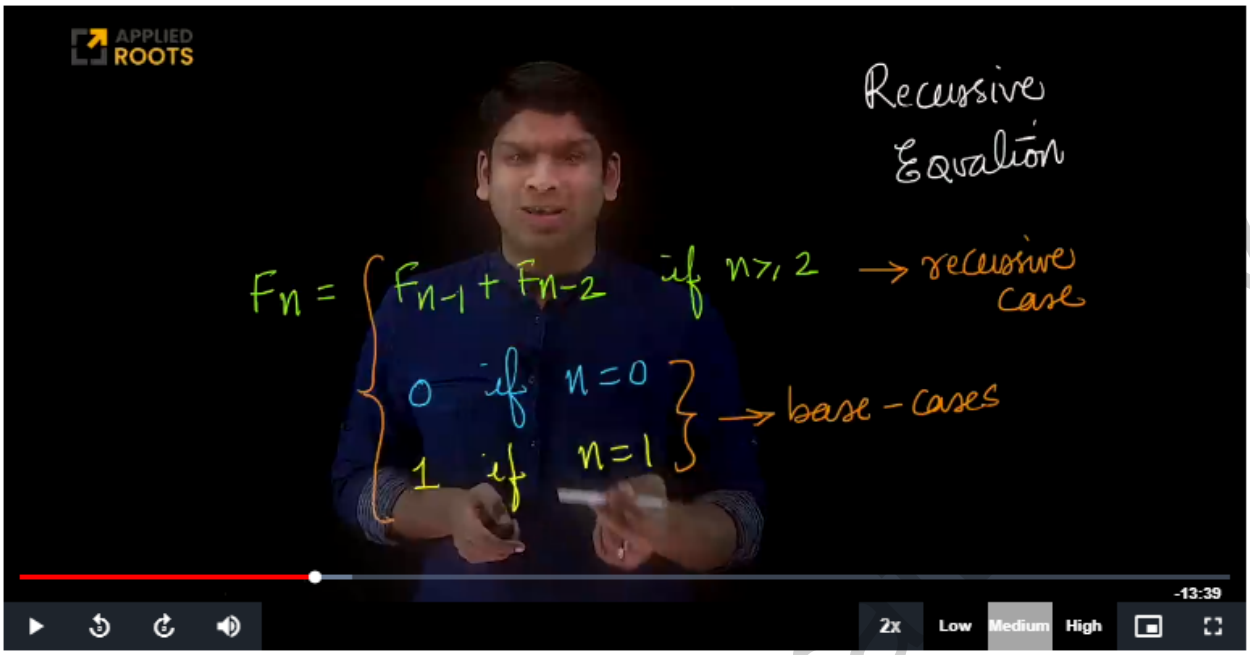
11.16 Fibonacci numbers using Recursion



Timestamp: 1:16

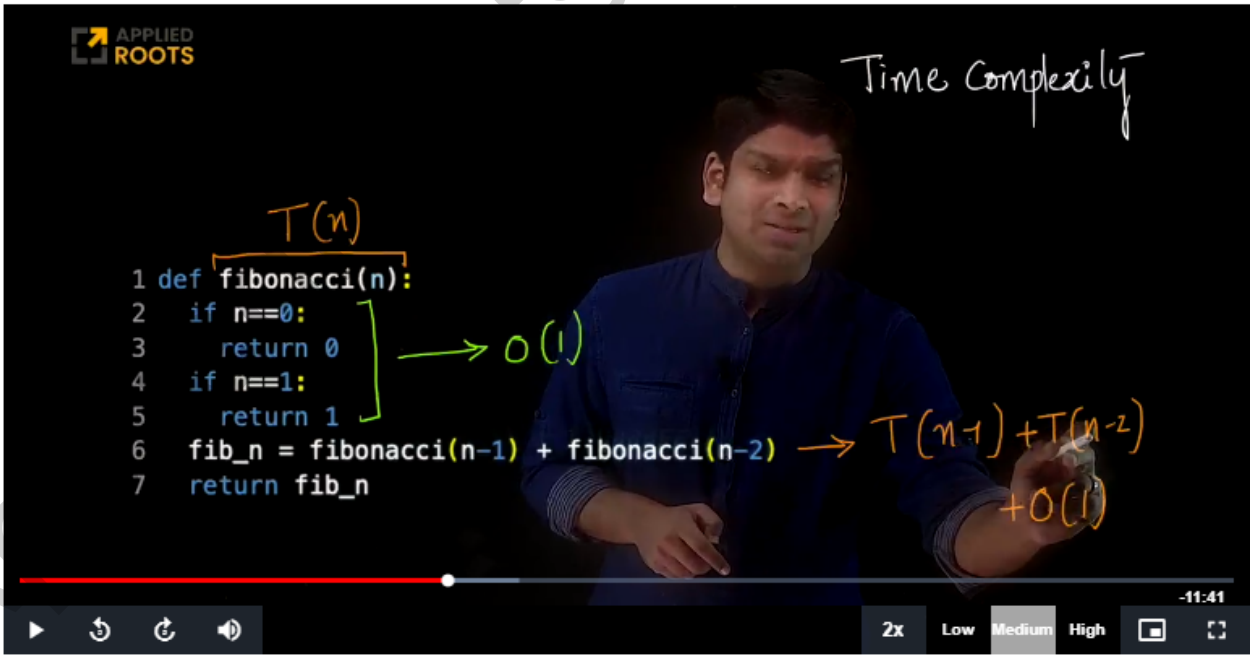
The above sequence is of numbers is called Fibonacci series and the 0th fibonacci number is 0 and is indicated by F_0, the 1st fibonacci number is 1 and is indicated by F_1. If you notice from the 3rd Fibonacci number, the fibonacci number $F_i = F_{i-1} + F_{i-2}$.

The fibonacci numbers occurs in nature in many forms. Please use wikipedia for the same.



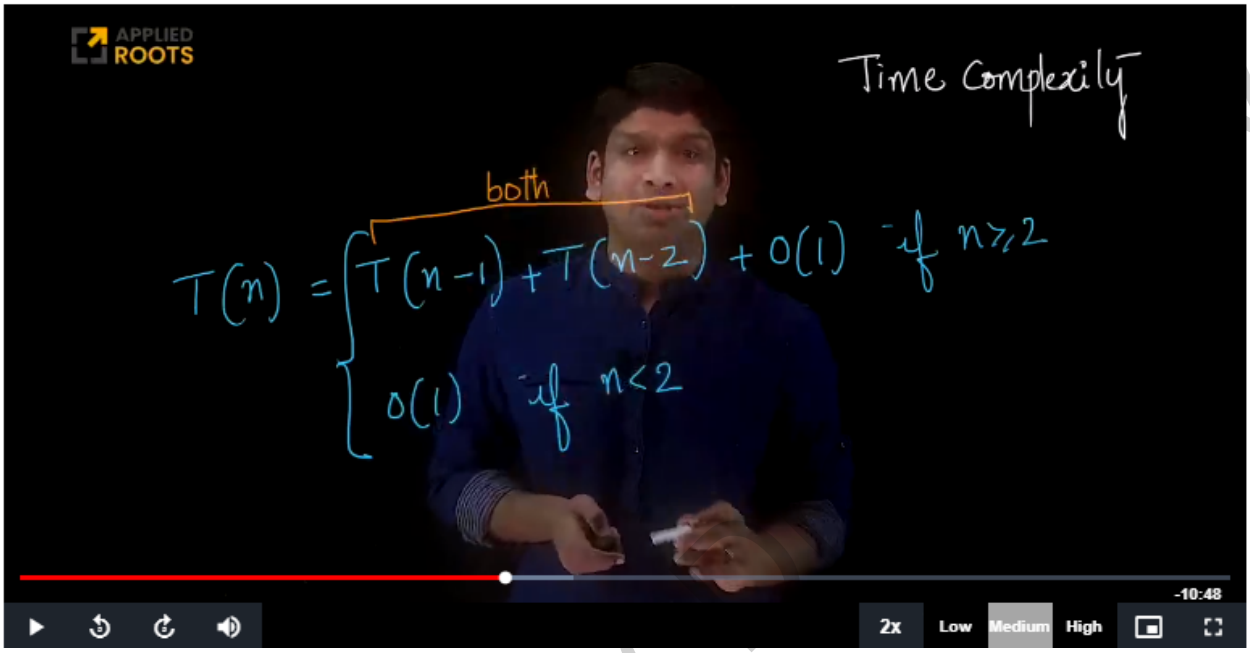
Timestamp: 4:24

The nth Fibonacci number can be found using the above recursive equation. Notice for the base cases 0 and 1 we need not use any recursive call. We have two base cases and one recursive case.



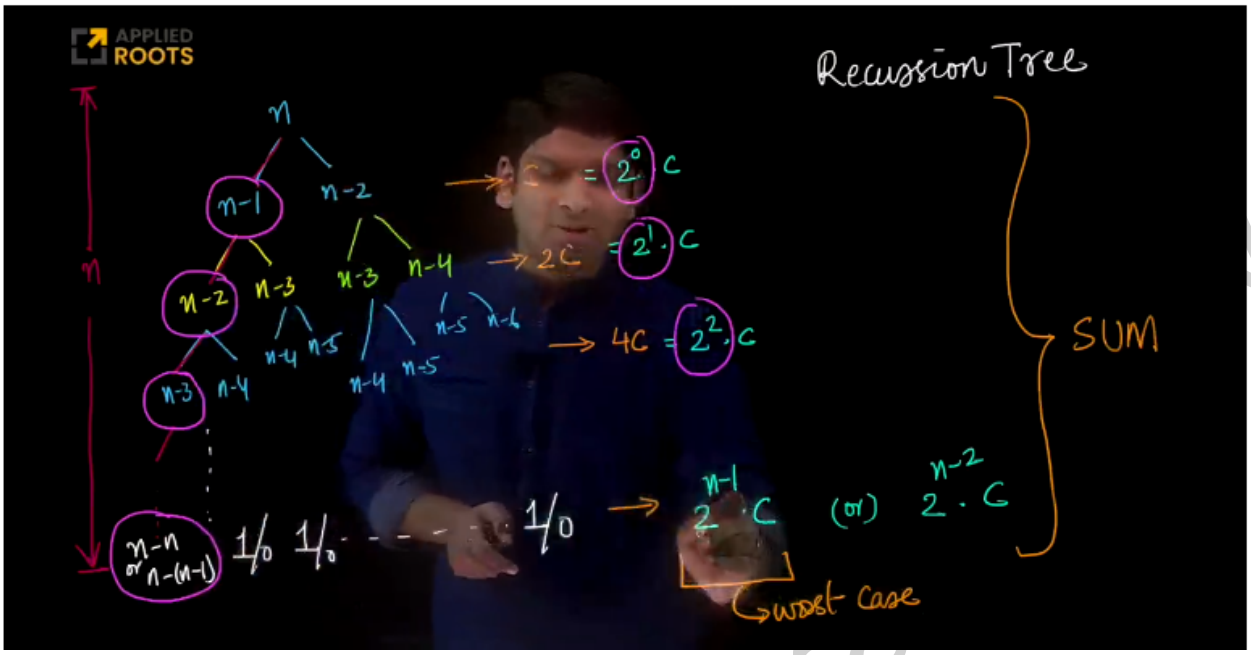
Timestamp: 6:22

The above figure shows to calculate the nth fibonacci number. Notice that for each recursive call we take O(1) time to check the base cases and the time taken for two recursive calls for calculating n-1 and n-2 fibonacci number respectively.



Timestamp: 7:15

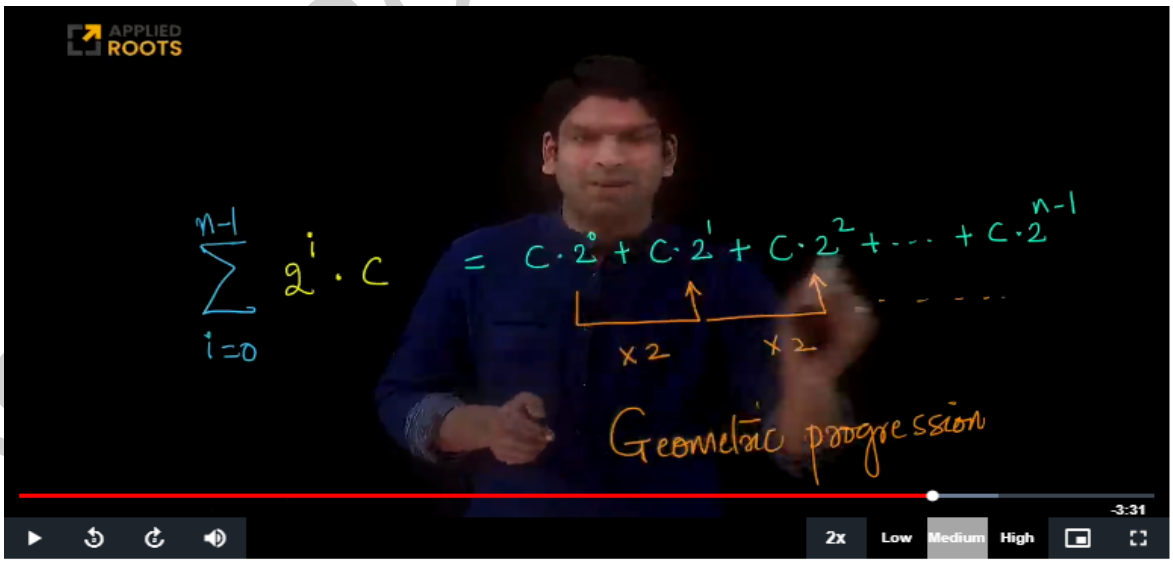
The recursive equations looks as above. For the base case we take T(n)= O(1) time else T(n) =T(n-1) + T(n-2) +O(1), O(1) time for addition of the previous two fibonacci numbers and for checking base conditions.



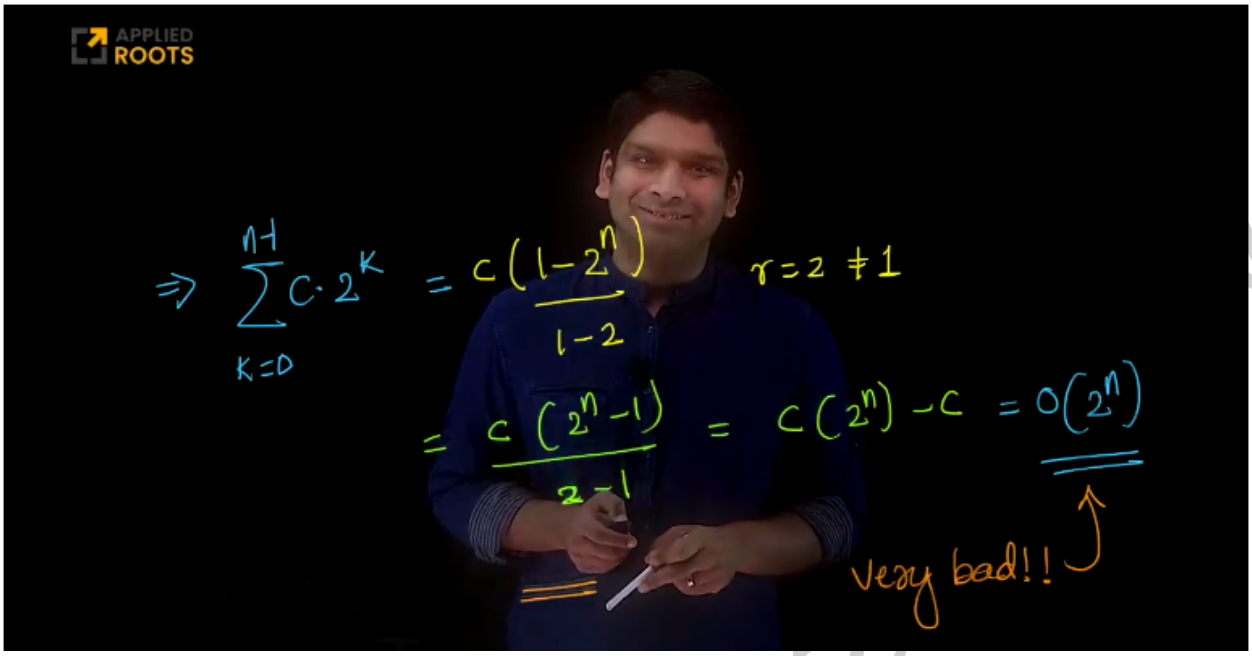
Timestamp: 13:08

Notice that using recursion tree as in the above picture we can find that the number of levels are n since we see that the leftmost element decreases by 1 each level until we reach base cases either 0 or 1.

Notice that each level i if the first element is n-ii, we are need to do 2^{i-1}*c number of operations. Summing the total time taken at all levels we can find that the sum follows geometric progression.



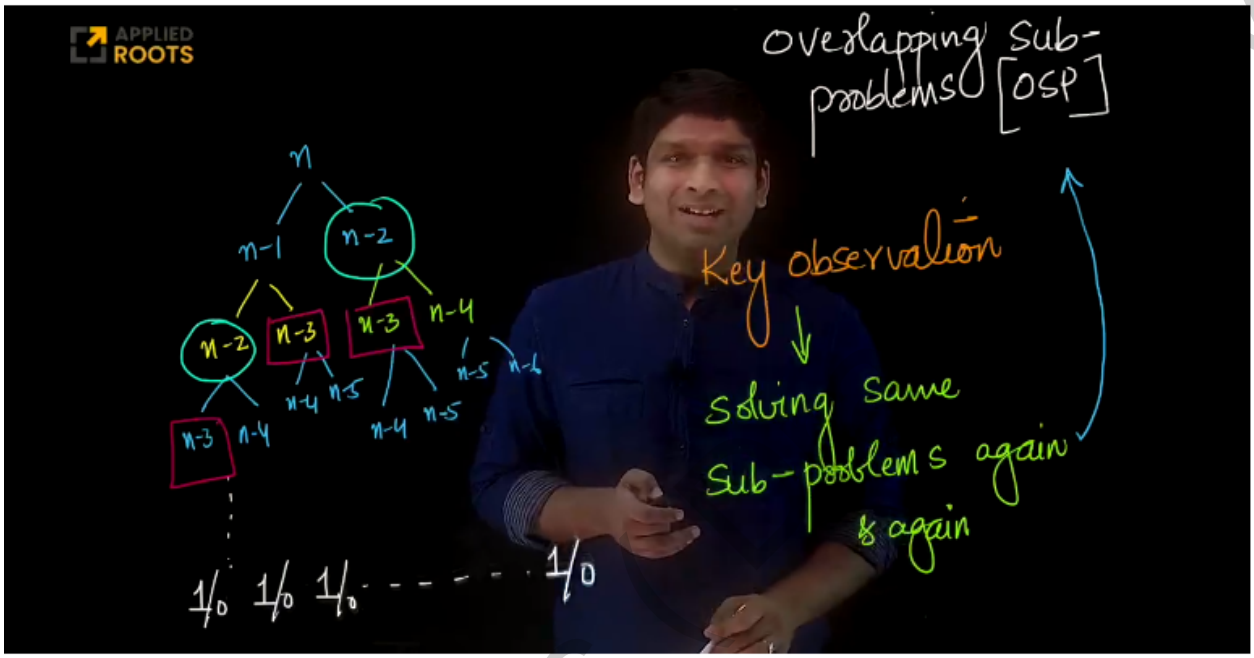
Timestamp: 14:32



Timestamp: 17:40

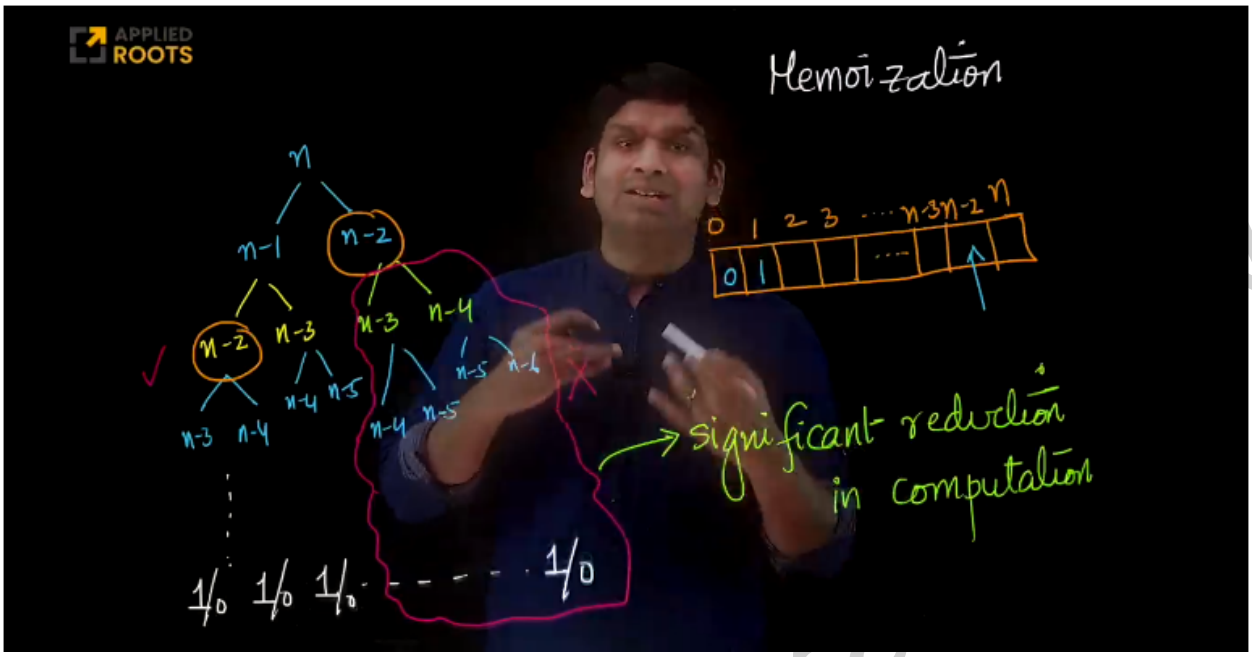
Solving the geometric progression, we can find that the time complexity of finding nth fibonacci number is $O(2^n)$ which is worse. In the next video, we will see how we can improve the time complexity.

11.17 Dynamic Programming (For Fibonacci numbers)



Timestamp: 2:59

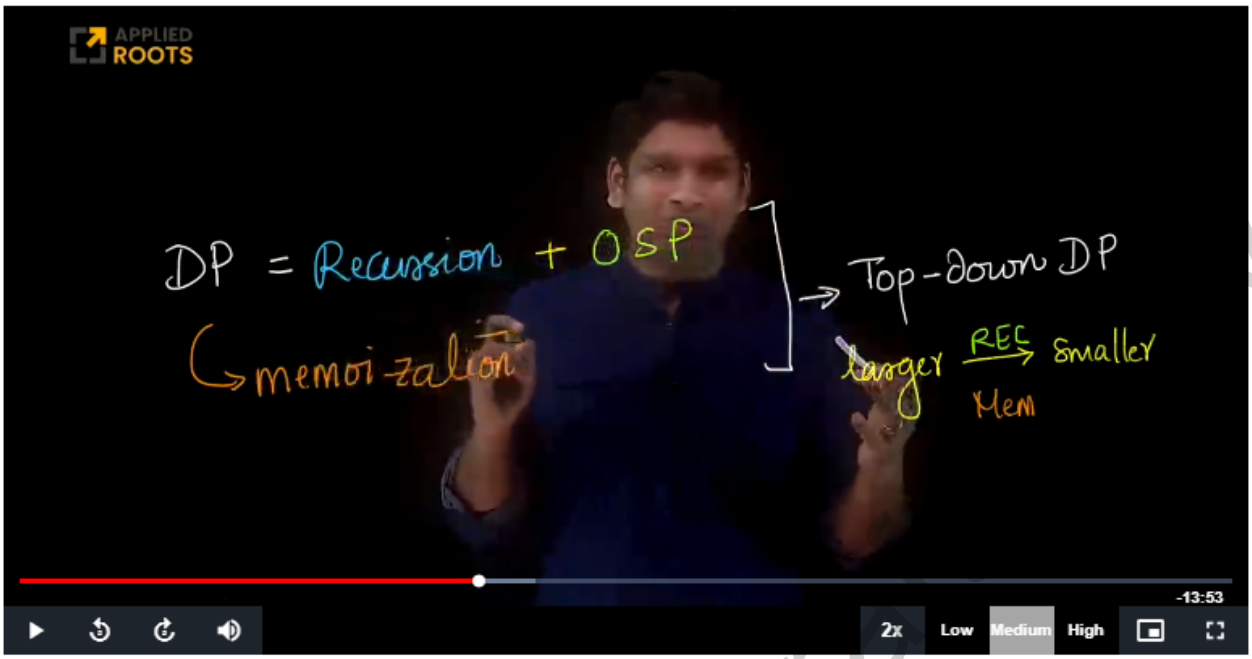
Notice in the recursion tree while calculating the n^{th} Fibonacci number, we are calculating fibonacci numbers such as n-2, n-3 many many times throughout the recursion tree. This issue of solving the same subproblems again and again to solve a larger subproblem is called overlapping sub-problems [osp]. One strategy to overcome this is to use a technique called memoization.



Timestamp: 7:32

So the technique called memoization is to store the result of solving subproblem into a data structure such as array and using it directly instead of solving it again. For example in our problem of finding nth Fibonacci number we can store the n-2 th fibonacci number once computed in the left subtree in an array at index n-2 and directly use it when we need it in the right subtree.

So for calculating any subproblem we will first check whether it is already present in our array, if it is then we will directly use it else we will compute it and store in the array and we will use it from next time directly from the array.

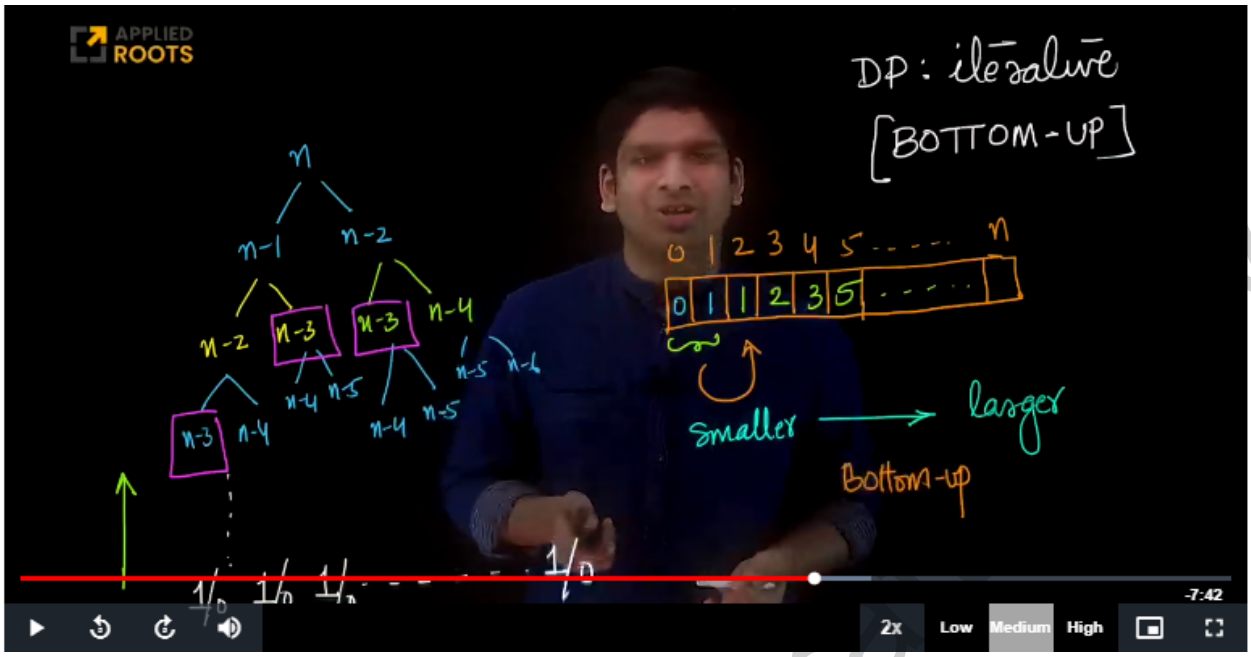


Timestamp: 8:29

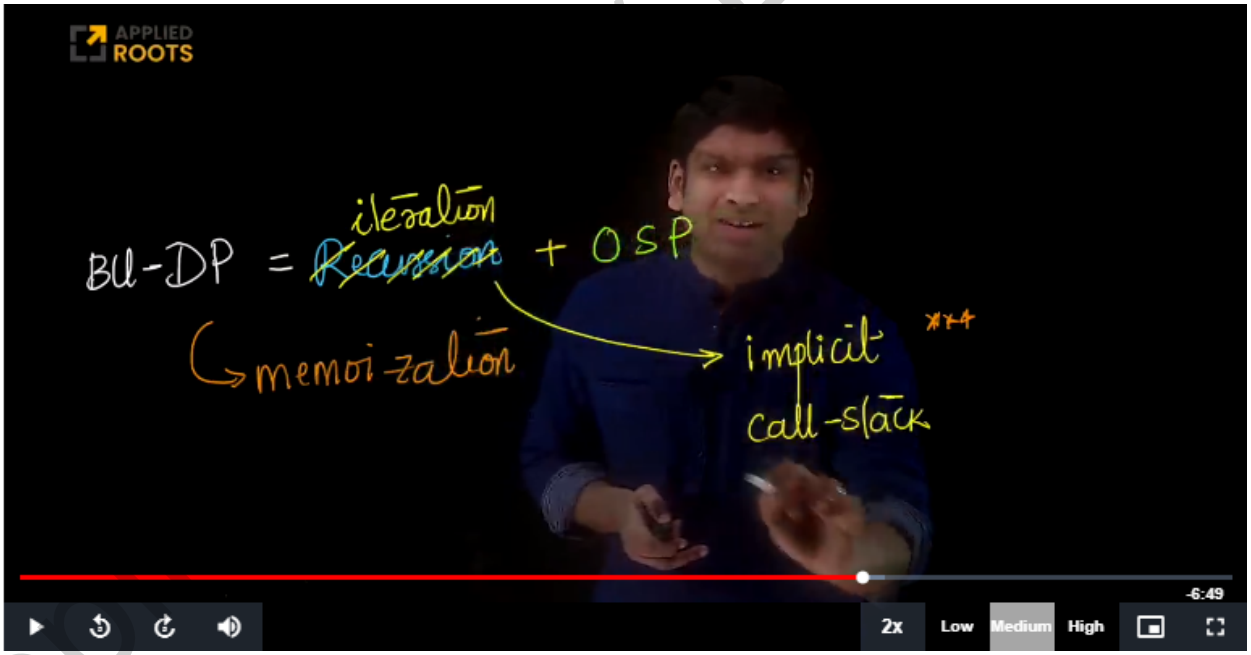
When we have a recursive subproblem and we noticed over lapping subproblems and if we use memoization to solve the problem then it is called dynamic programming (DP)

There are two types of dynamic programming

a) Top down approach, in which we will solve larger problem by recursively solving smaller subproblems and use memoization. Recursion is often referred to as optimal substructure.
b) Bottom up approach, where we will populate the datastructure (in this case array) by solving the subproblems and storing them and using them to solve the larger subproblems as shown in the below figure.

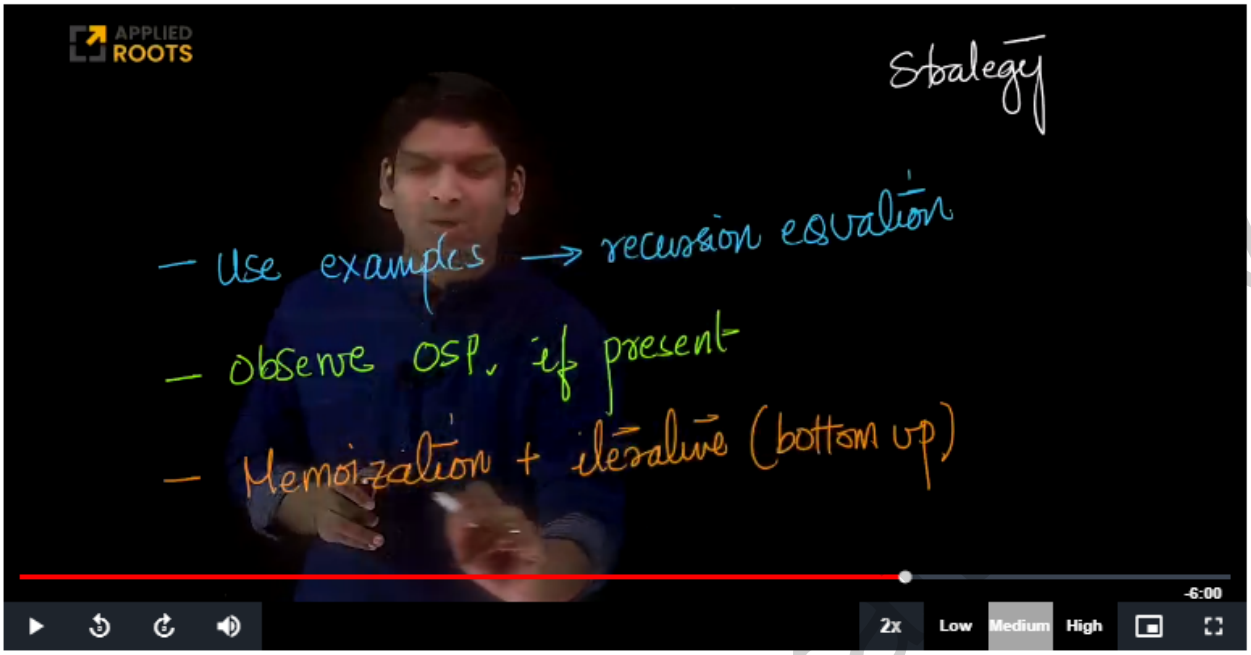


Timestamp: 14:40



Timestamp: 15:33

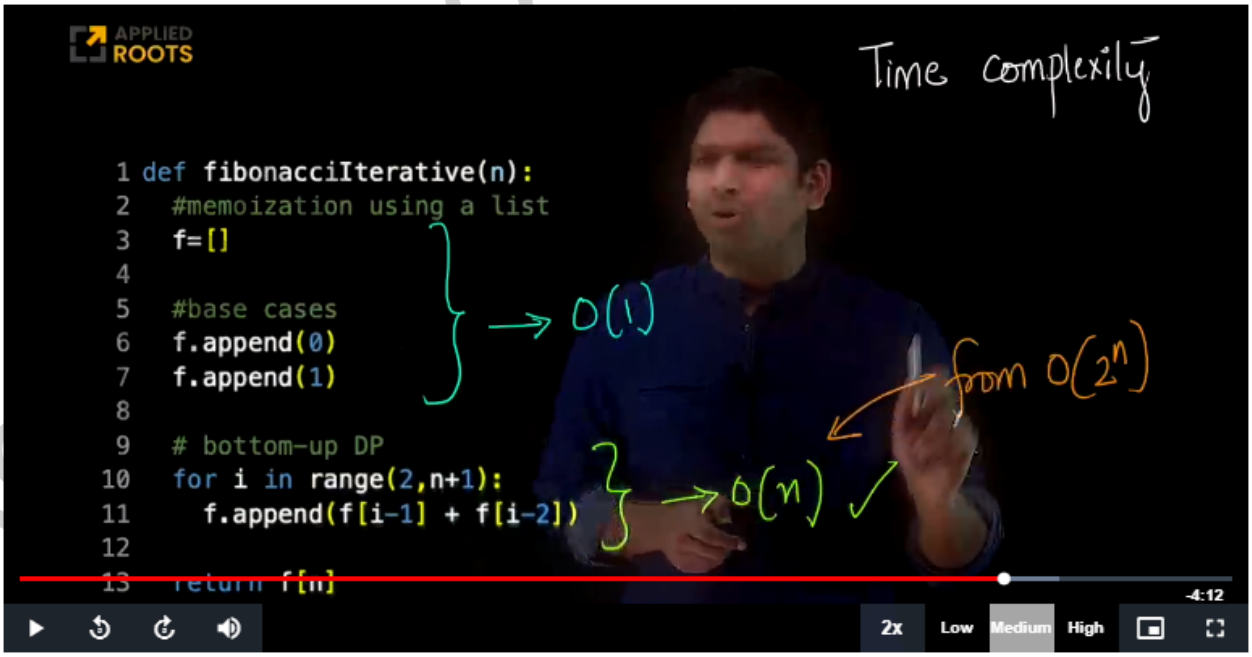
Memoization is often referred to as tabulation when we were are implementing bottom up DP. We often use bottom-up approach due to the implicit call stack caused by using recursion in the top-down approach.



Timestamp: 16:22

The strategy for solving a problem is to use examples and find its recursive equation.
Check if it contains overlapping subproblem (OSP), if it is present then use memoization and iterative i.e bottom up DP.

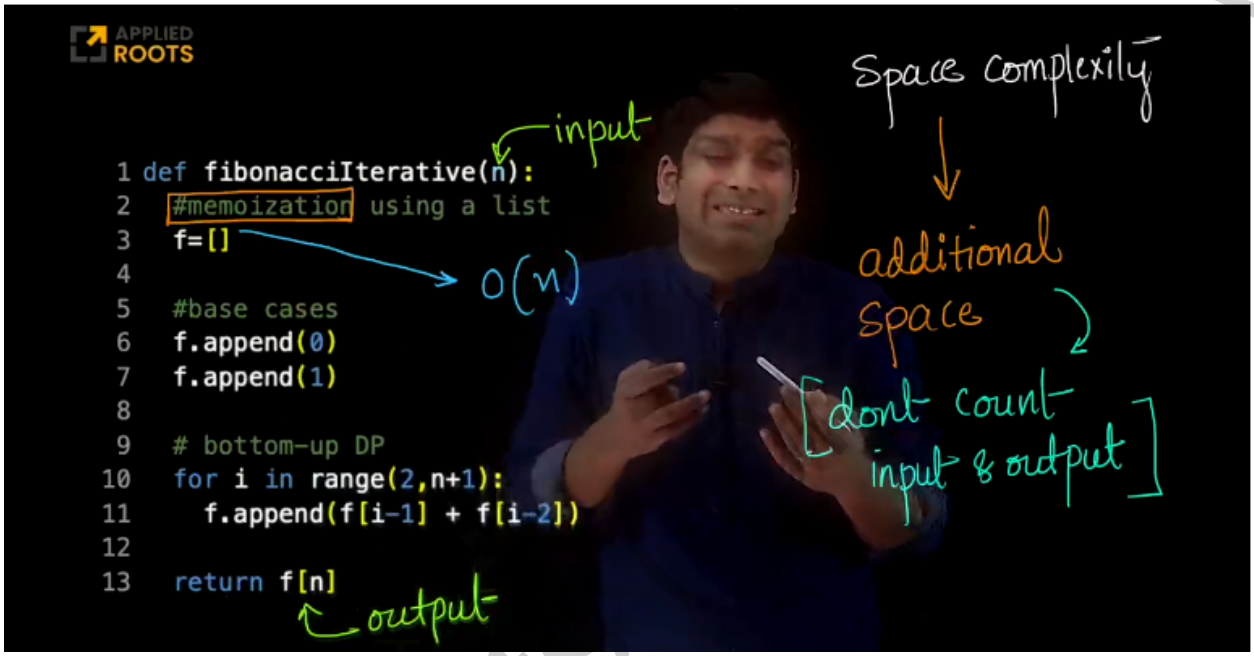
Let us look at the code and time complexity for bottom-up DP approach.



Timestamp: 18:10

If we can observe from the above figure our iterative DP is taking O(1) time for populating our array with base cases 0 and 1 and taking O(n) time for calculating nth fibonacci number. So the total time complexity for our iterative DP for calculating the nth fibonacci number is O(n) which is a significant drop from the time complexity $O(2^n)$ of our recursive algorithm with no memoization.

Let us look at the space complexity of iterative DP for our problem.

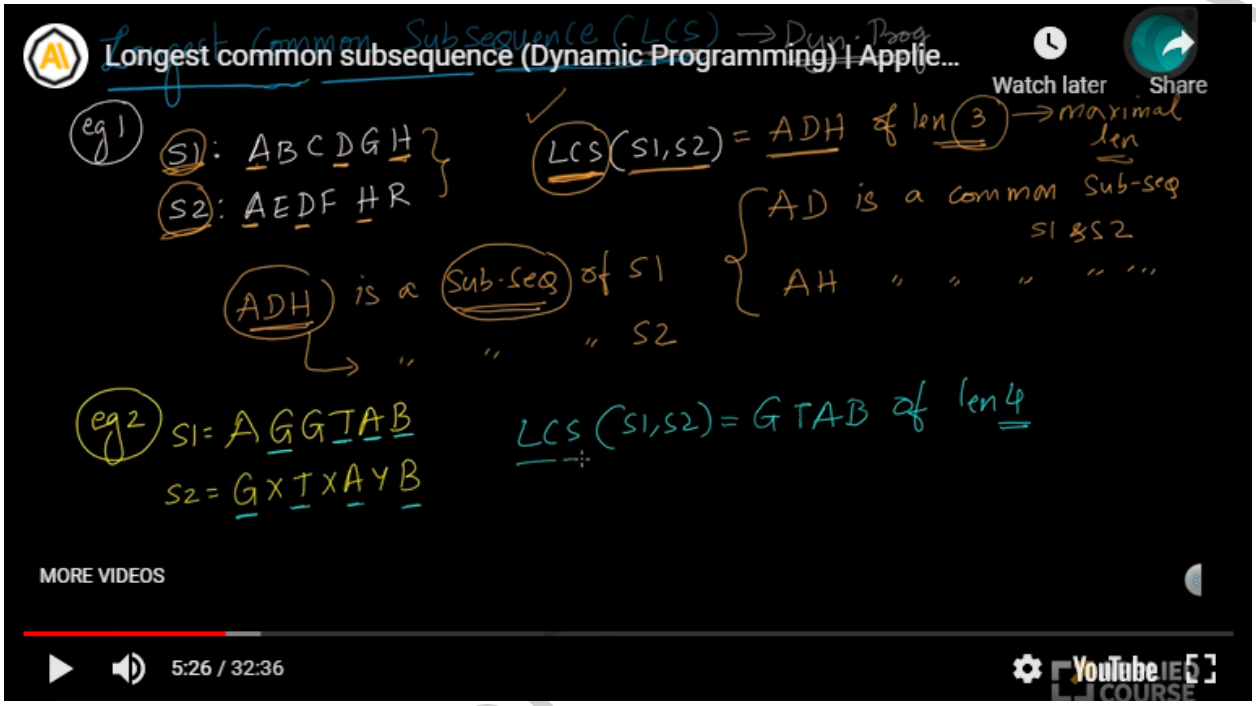


Timestamp: 19:16

Notice that our space complexity has become O(n) due to memoization since we have to store the results of smaller subproblems to solve larger problems. Our space complexity increased from O(1) time (although recursion uses call stack memory implicitly) in recursive approach with no memoization to O(n) time with iterative approach using memoization.

Notice that by using iterative bottom up approach we are able to make time complexity come down from $O(2^n)$ to O(n) with increase in explicit space complexity from O(1) to O(n). But this is completely fine because our ram sizes will increase considerably overtime and we are more concerned about time rather than memory.

11.18 Longest Common Sub-Sequence(LCS) using DP

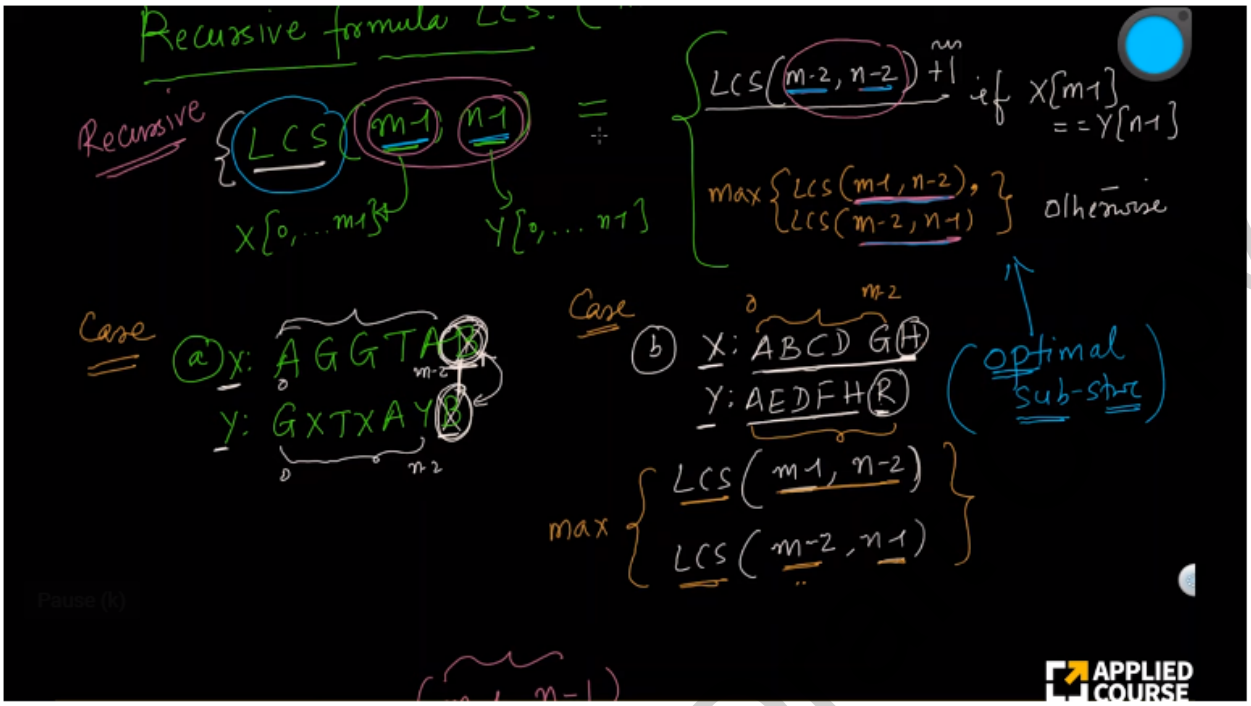


Timestamp: 5:26

Longest common subsequence is the problem of finding the longest subsequence that is common between the two string. Please note that subsequence unlike substring doesn't have a condition that all the elements in the subsequence should be neighbours rather maintaining the same order as in the original sequence or string is enough to be a subsequence.

In the above figure, for eg1 ADH of length 3 is the longest common subsequence whereas for eg2 GTAB of length 4 is the longest common subsequence.

LCS has applications in genomics where we have to find the longest common subsequence between two DNA. It also has applications in finding difference between two files, identifying plagiarism, etc. That is why it is important to solve LCS problem.

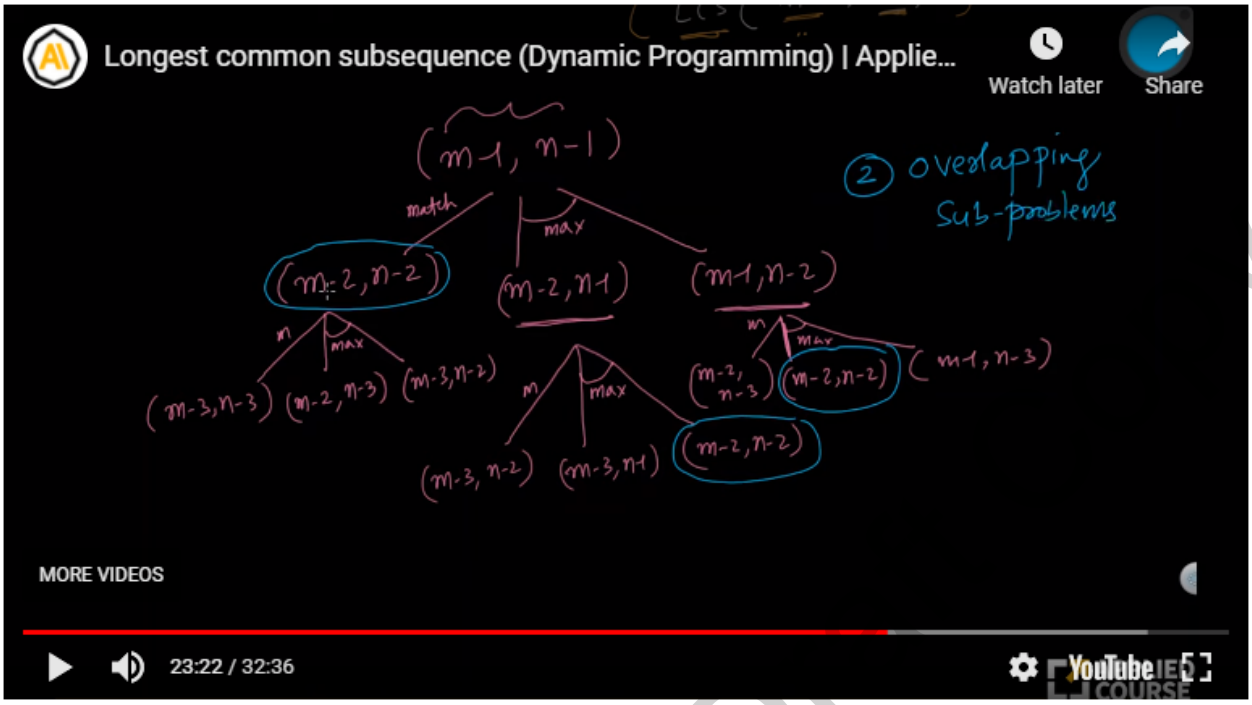


Timestamp: 22:56

The recursive formulation of LCS is as above. The key intuition for this:

a) As shown in case a in the above figure, if the last character in both the sequences match then the character is bound to be in the lcs of the sequences. Hence we will just add the length 1(length of the last character) and compute the lcs of the remaining part of both the sequences.
b) As shown in case b in the above figure. If the last character in both sequences doesn't match then at least one of them is bound to be miss in the final lcs of the sequences, hence we will have two recursive calls excluding the last character of one sequence and including the last character of the other sequence in one call and vice-versa in the other.

Notice in the above figure there is an optimal substructure when we are trying to solve our bigger problem by dividing into smaller subproblems using recursion.



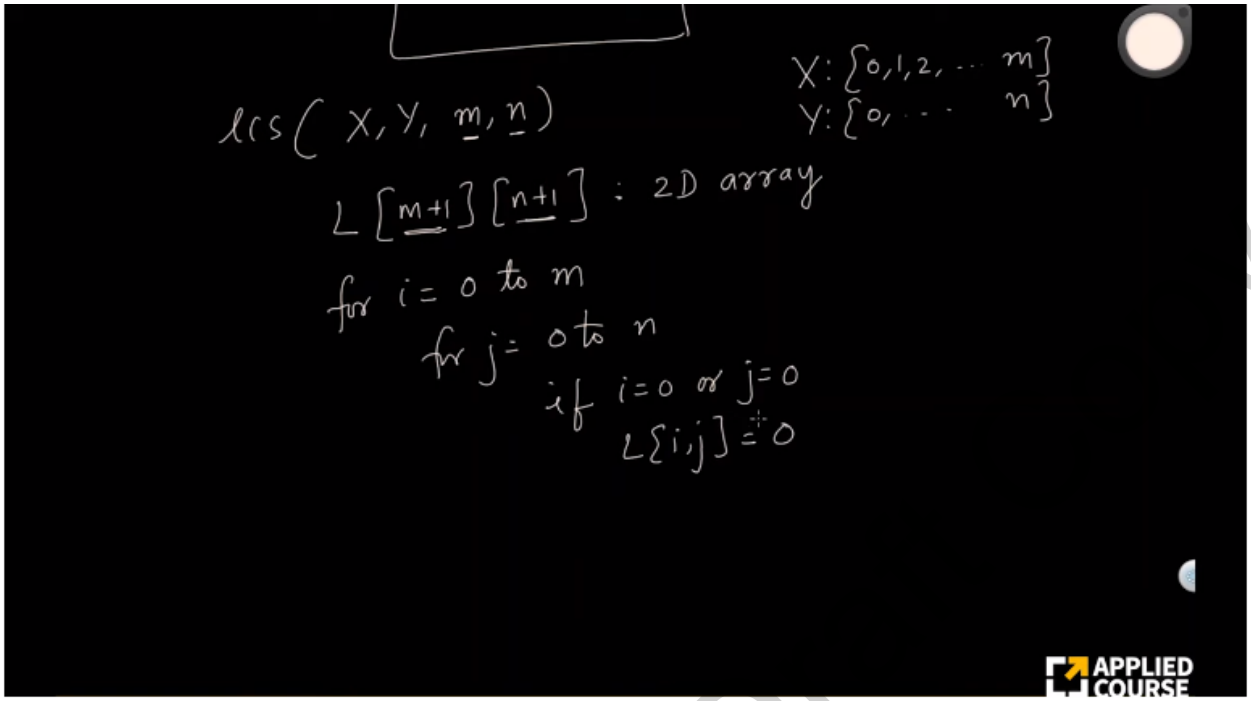
TImestamp: 23:22

In the above figure we can notice in the recursion tree of our solution we have overlapping subproblems as highlighted by the blue boxes.

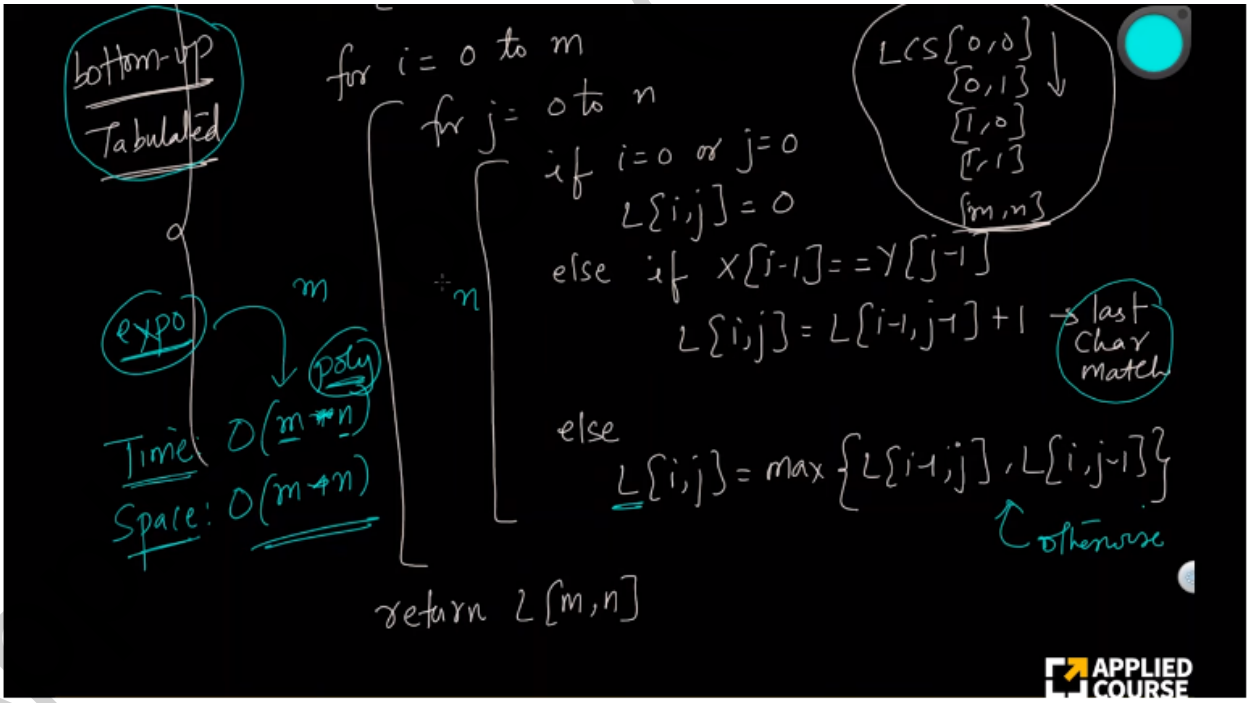
If we notice that in the worst case if our last characters of both the sequences doesn't match at each level we have to solve two subproblems and solve the max between them. So at level 1 we will be solving 2 subproblems, then at level 2 we will be solving 4 subproblem and so on and at last level h we will be performing 2^h sub problems and the no of levels h is m+n since the recursion tree continues until both (m-m,n-n=0) happens on all branches.

Hence at last level we have 2^{m+n} subproblems to solve and each takes O(1) time. So the total time complexity of the recursive algorithm is $O(2^{m+n})$.

But since we have both optimal substructure and overlapping sub problems we can solve this problem using bottom-up dynamic programming as shown in the below figures.



Timestamp: 28:49



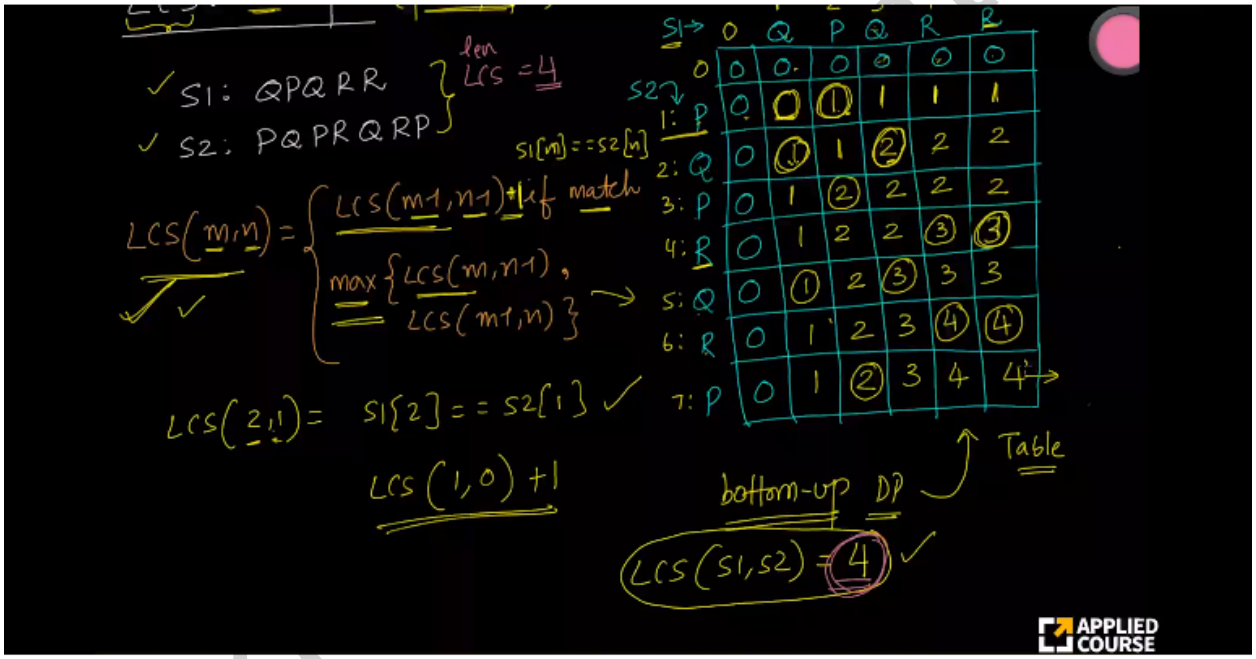
Timestamp: 31:59

Since we are following bottom-up approach when either of the sequences have length 0 then we are making LCS is 0 then we check whether the last character is same and call recursion accordingly.

If we notice the time complexity of the algorithm is O(m*n) since we are using two for loops of length m and n and we are performing O(1) operations for memoization, max etc within each iteration.

Hence using dynamic programming we are able to bring down an exponential time of complexity $O(2^{m+n})$ to O(m*n) for solving our LCS problem.

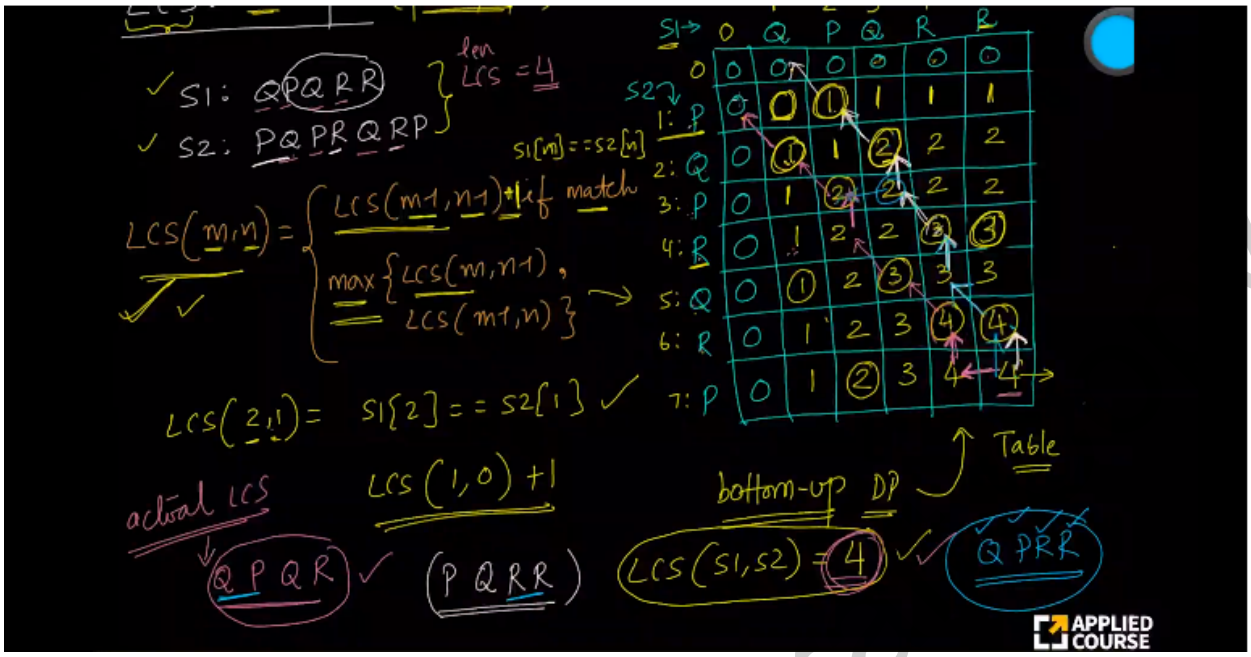
11.19 LCS: Worked out example



Timestamp: 11:02

Please workout the above example of finding the LCS between sequences S1: QPQRR and S2: PQPRQRP.

Note that when there is a match between the column character and the row character at (m,n) we add 1 to the value at (m-1,n-1) and store in (m,n) otherwise we take the max of (m-1,n) and (m,n-1) and store in (m,n).



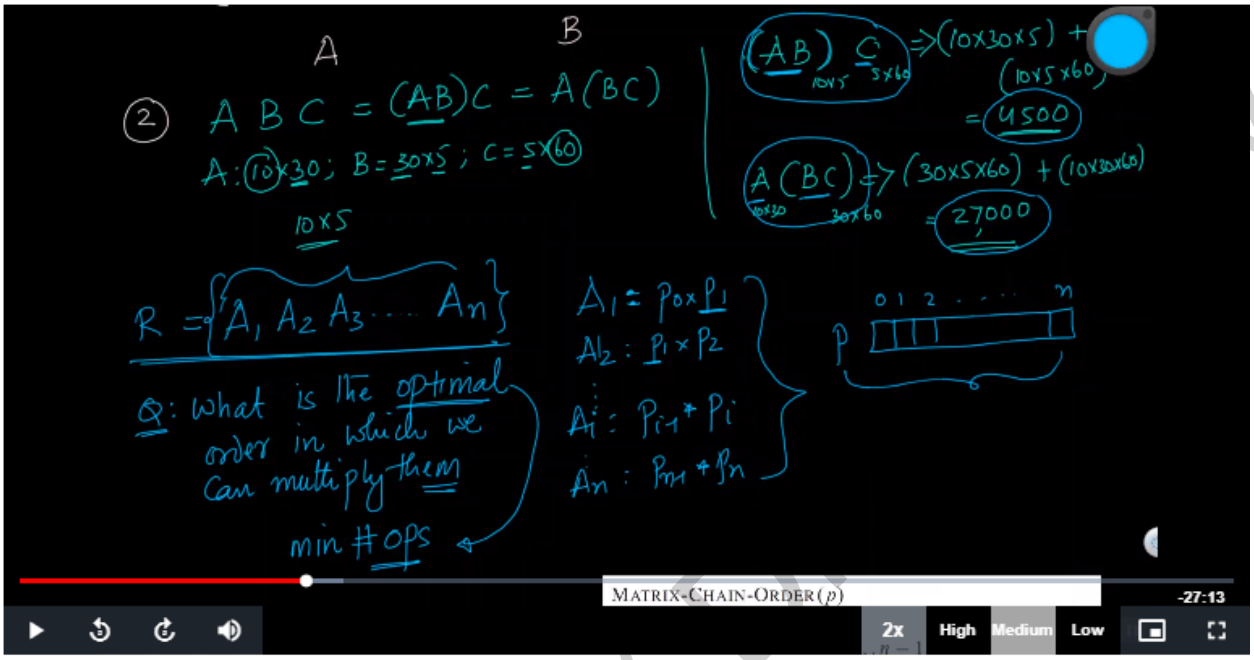
Timestamp: 18:11

From the above figure we can see that we can even trace back through our table to get the actual LCS along with length of LCS.

Starting from the end result of 4, we traceback following from which cell we got the value 4, notice that the end result 4 is calculated using the max of (7,5) or (6,6) hence we can take either paths. If we take path (6,6) which contains 4 we got 4 by incrementing the value at (5,5) by 1 since there is match at (6,6) between R and R. Likewise we can trace back to the start.

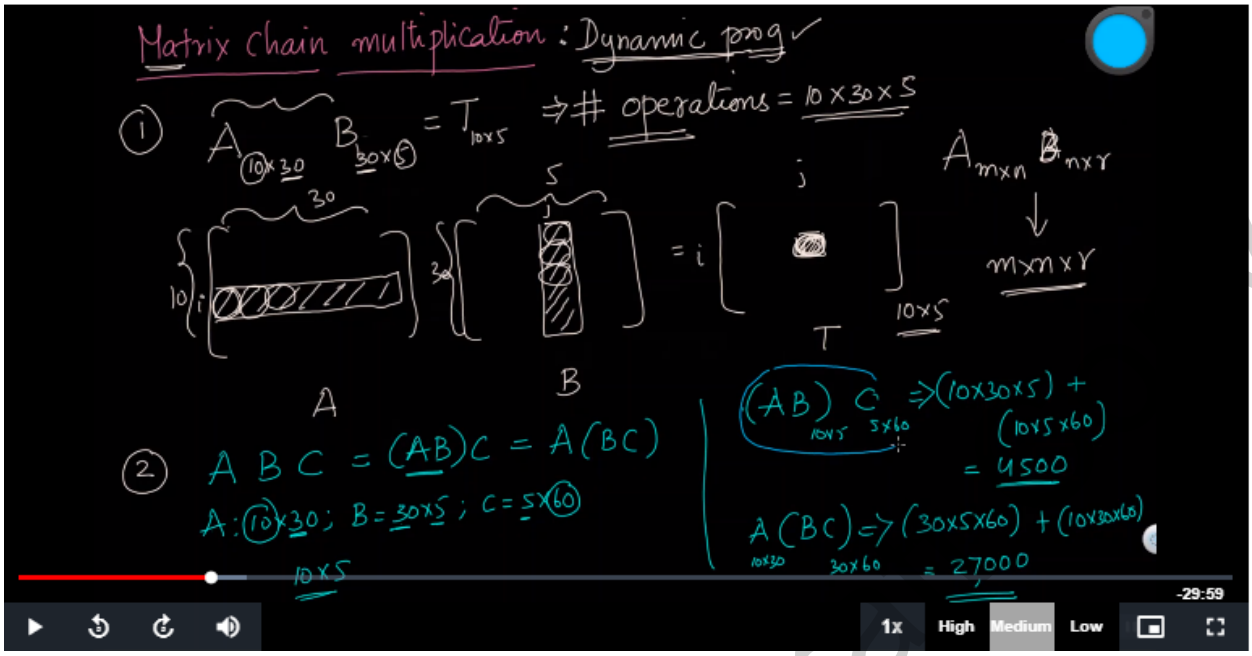
By taking multiple paths we can end up with different LCS which are all correct. Please work it out by following different color paths and match with the exact color LCS in the above figure.

11.20 Matrix Multiplication using DP



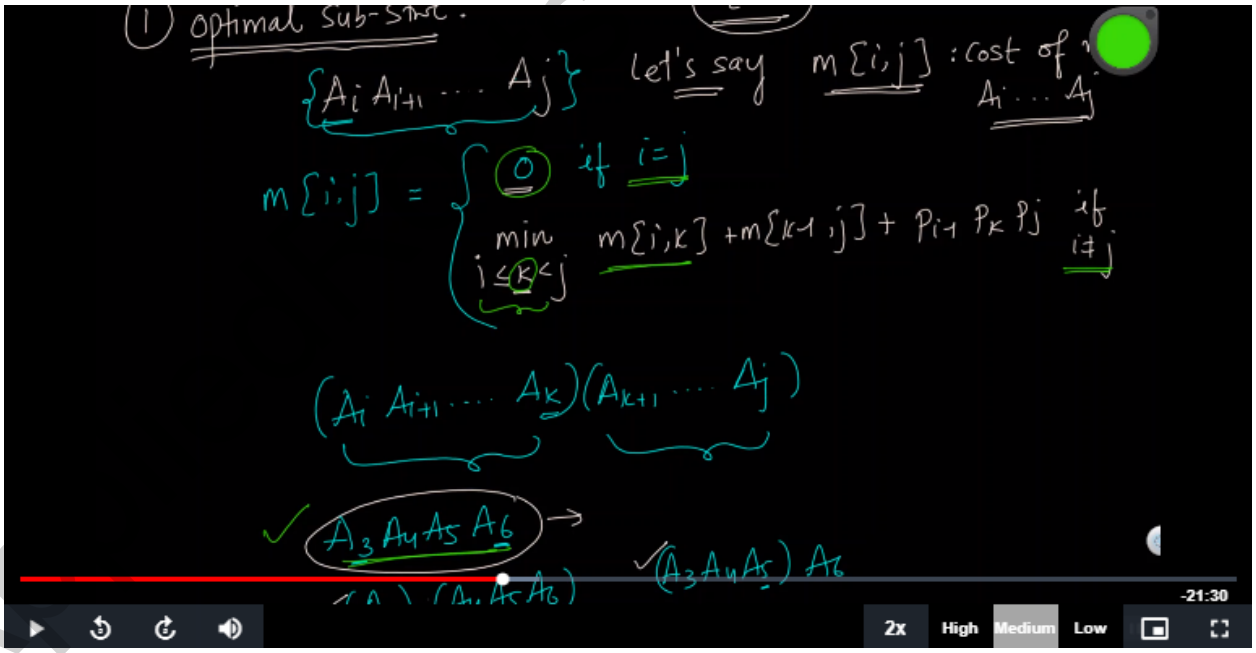
Timestamp: 08:26

Matrix multiplication is the problem of finding which matrices to multiply first in the multiplication sequence of matrices A1*A2*A3*A4 ..*An given matrix like of A_i to be P_{i-1} x P_i. For example multiplying matrix sequence A*B*C can be multiplied in two ways i)By first computing A*B and then (A*B)*C ii) By first computing B*C and then A*(B*C). The way which reduces the number of multiplications has to be chosen to reduce the number of operations and save time.



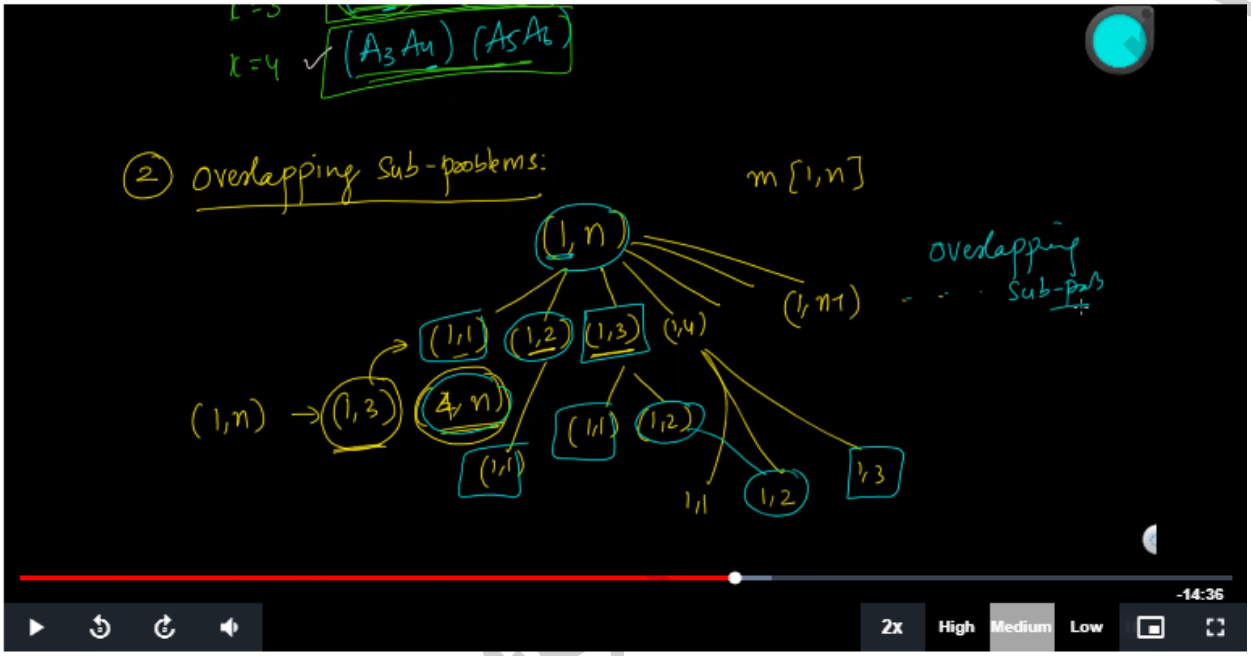
Timestamp: 5:40

As shown in the above figure choosing to perform A*B then (A*B)*C reduced the time taken 6 times for the given matrix sizes. Notice that to multiply matrices of size m*n and n*r the number of multiplications to perform is m*n*r.



Timestamp: 14:09

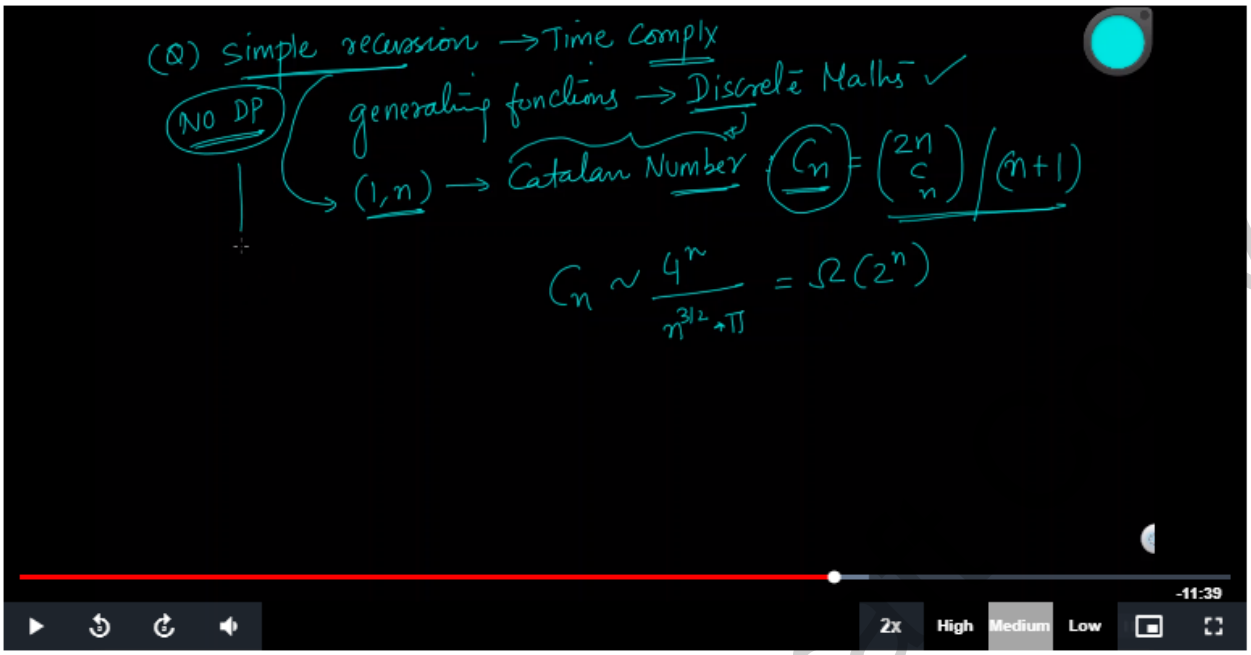
Given an array sequence Ai Ai+1 … Aj . Let's say m[i,j] is the cost of multiplying the sequence of array Ai to Aj. Then we can solve the problem to find m[i.j] by first addressing the base case. When i=j it means that there is only one array hence no multiplications are required hence cost m[i,j]=0 otherwise for all possible values of k such that i<=k<j we find which value of k minimizes the combined cost of multiplying first k arrays and the next n-k arrays and the number of multiplications to multiply these resultant two arrays.



Timestamp: 21:03

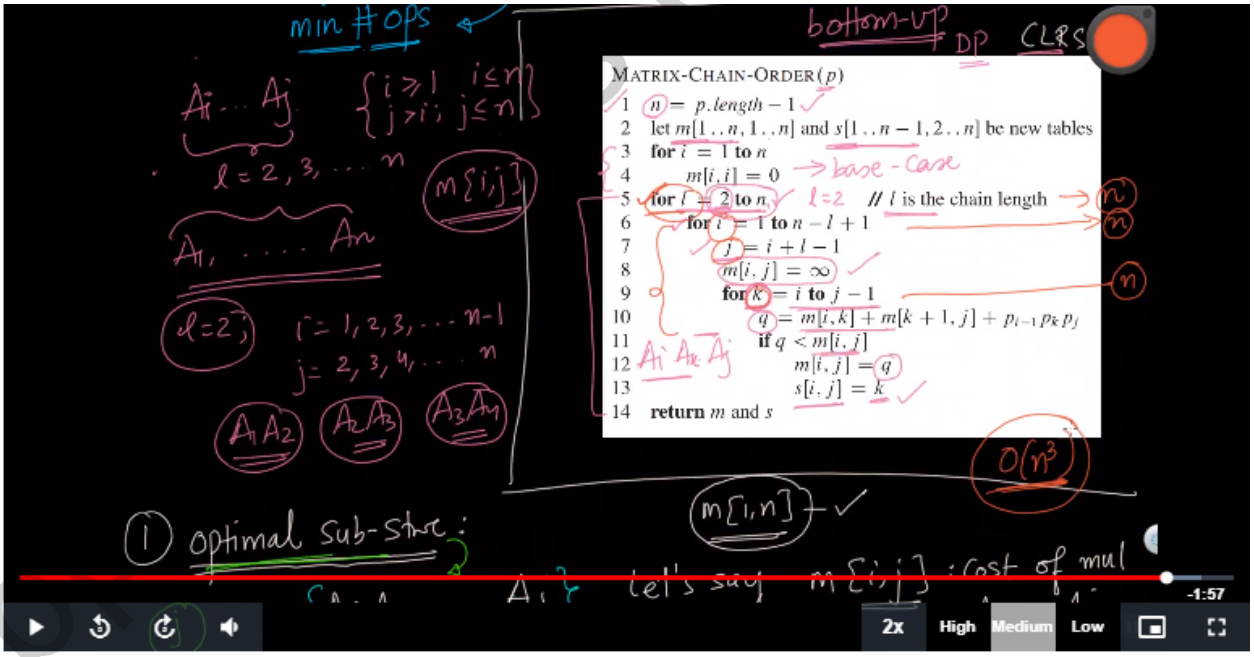
Notice that as in the above figure the recursion tree for the problem contains the overlapping subproblems as shown in the figure. Since we have optimal substructure and overlapping subproblem we can solve this problem using DP.

In normal recursive implementation without DP it takes $\Omega(2^n)$ time the proof of which will explained in discrete maths. Note that the time complexity to solve the matrix multiplication problem for n+1 arrays is given by the catalan number n.



Timestamp: 24:00

Let us see the below pseudo code to solve matrix multiplication problem using bottom-up DP.



Timestamp: 33:42

n stores the length of the array since p (the array containing matrix sizes) has n+1 elements.

We create matrix m whose element m[i,j] store the minimum cost for multiplying matrices from Ai to Aj. We also create matrix s whose elements s[i,j] stores the value of k that splits the sequence of matrices into i...k and k+1...j.

We store 0 in all elements of m with same i and j since the cost of multiplication given only a single array is 0 since no multiplications are required.

We first start with trying to multiply array sequences of length 2 and figure out which two arrays to multiply first. Suppose we were given array A1,A2,A3,A4 then we have to find out whether multiplying (A1,A2) or (A2,A3) or (A3,A4) is optimal. Then as L increases we will look at combining larger length sequences.

So when l=2, i loops from 1 to n-1 and j takes value 2 when i=1 and when i=2 j takes value 3 and so on. So when i=n-1 j=n. k takes values 1 when i=1 and j=2, k takes only value 2 when i=2 and j=3.

So when i=1 and j=2, k takes value 1, so q becomes m[1,1] +m[2,2] + pop1p2 = pop1p2 since we know that m[i,i]=0 so in this case the total cost is the cost to multiply matrices A1 and A2.

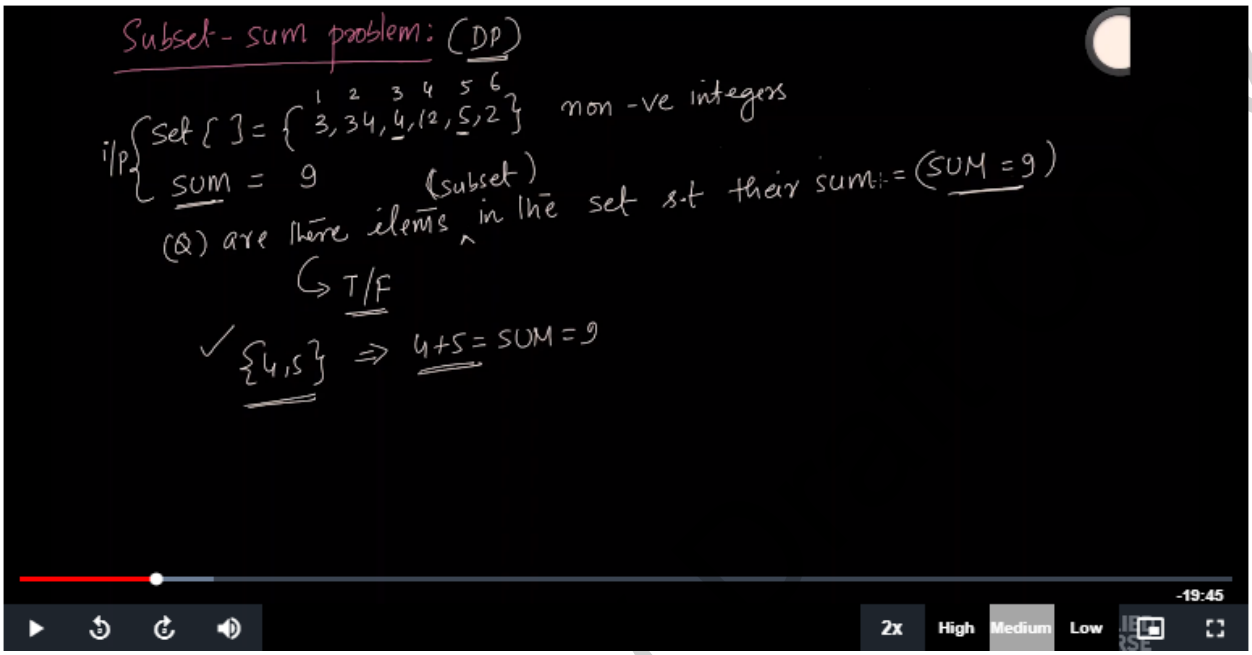
For different values of l, for different combinations of i and j (based on l) m[i,j] stores the minimum cost to multiply to array sequence Ai..Aj by first calculating Ai...Ak and Ak+1 ...Aj and s stores the value k by which to split the array sequence Ai...Aj.

Please workout using the table similar to how we solve our lcs problem with sample sequence of arrays A1,A2,A3,A4,A5 with some examples sizes.

Notice that three loops each for l, i and k each of them runs for at most n times and inside each iteration we do O(1) operations of comparing and storing and adding. Hence the total time complexity is $O(n^3)$.

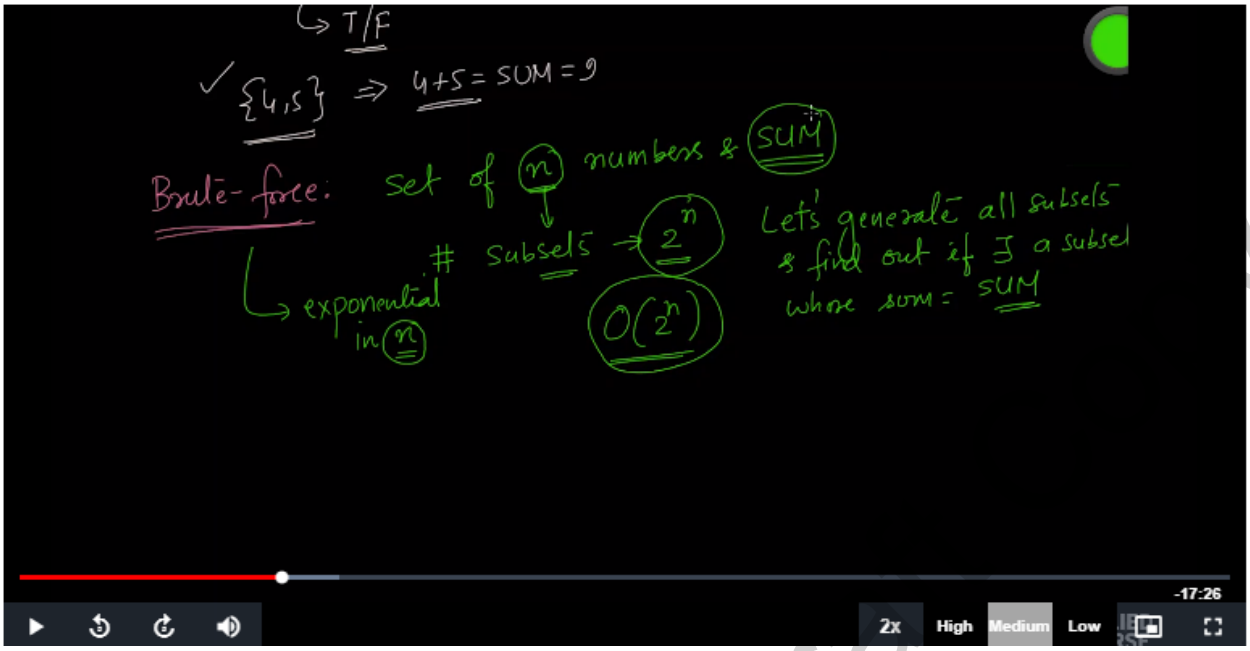
Notice that we came down from $O(2^n)$ time complexity using normal recursive approach to $O(n^3)$ time complexity using bottom-up dynamic programming approach. Please go to 11.26 for a solved example.

11.21 Subset- Sum problem using DP



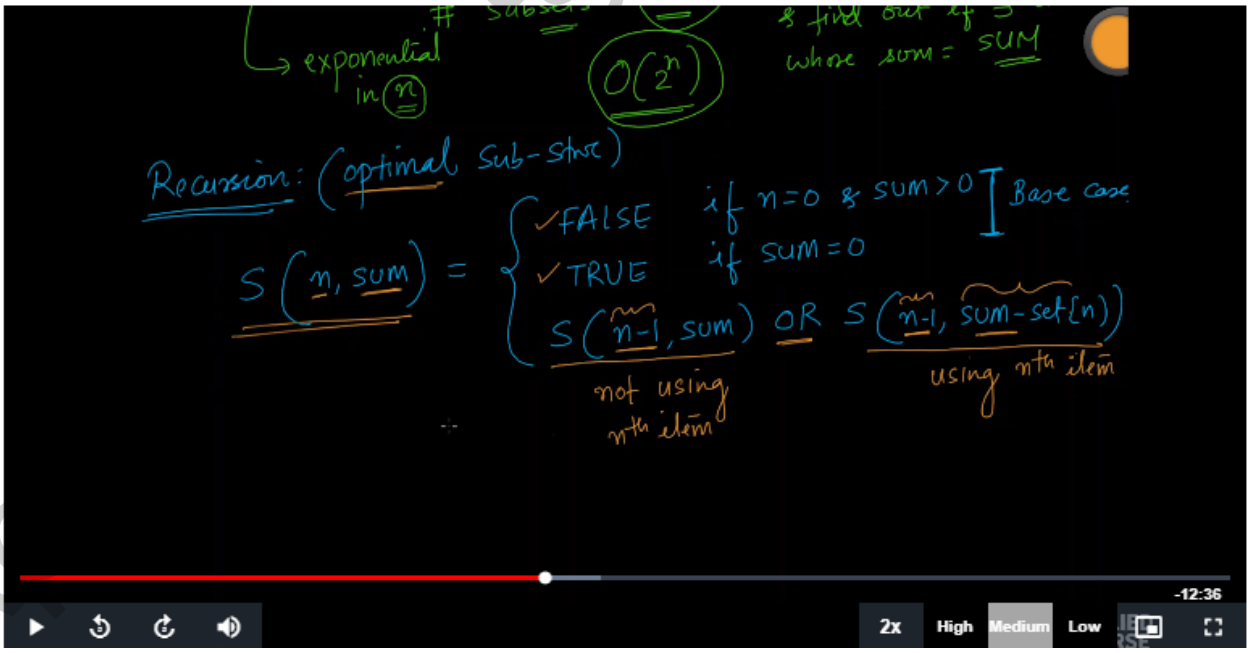
Timestamp: 2:31

Subset sum problem is the problem of finding whether a sum of subset of elements from a given set of non-negative integers will be equal to the sum given. In the example in the figure, given a set of elements of non-negative integers set ={3,34,4,12,5,2} we want to find whether it contains any subset whose sum of elements is 9. {4,5} is a subset whose sum is 9=sum hence there exists a subset.



Timestamp: 4:50

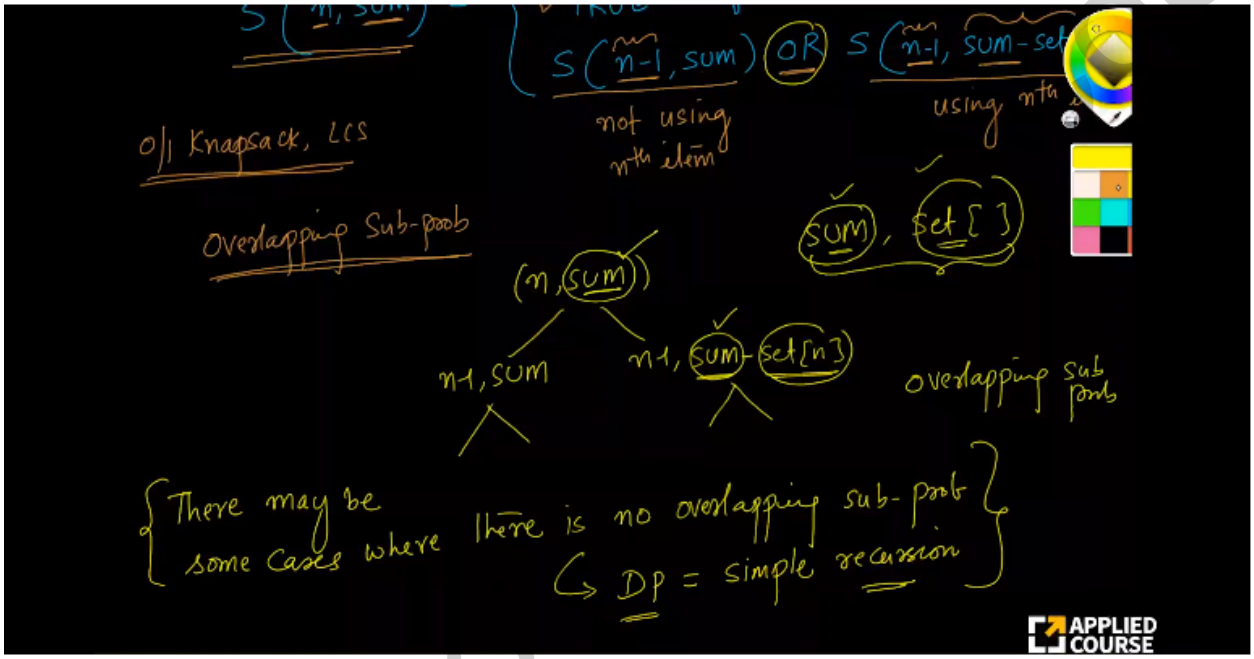
The brute force approach of solving this problem is to take each subset and check whether the sum of the elements of each subset is equal to the given sum. Since there $O(2^n)$ subsets the time complexity will be exponential in terms of n.



Timestamp: 9:40

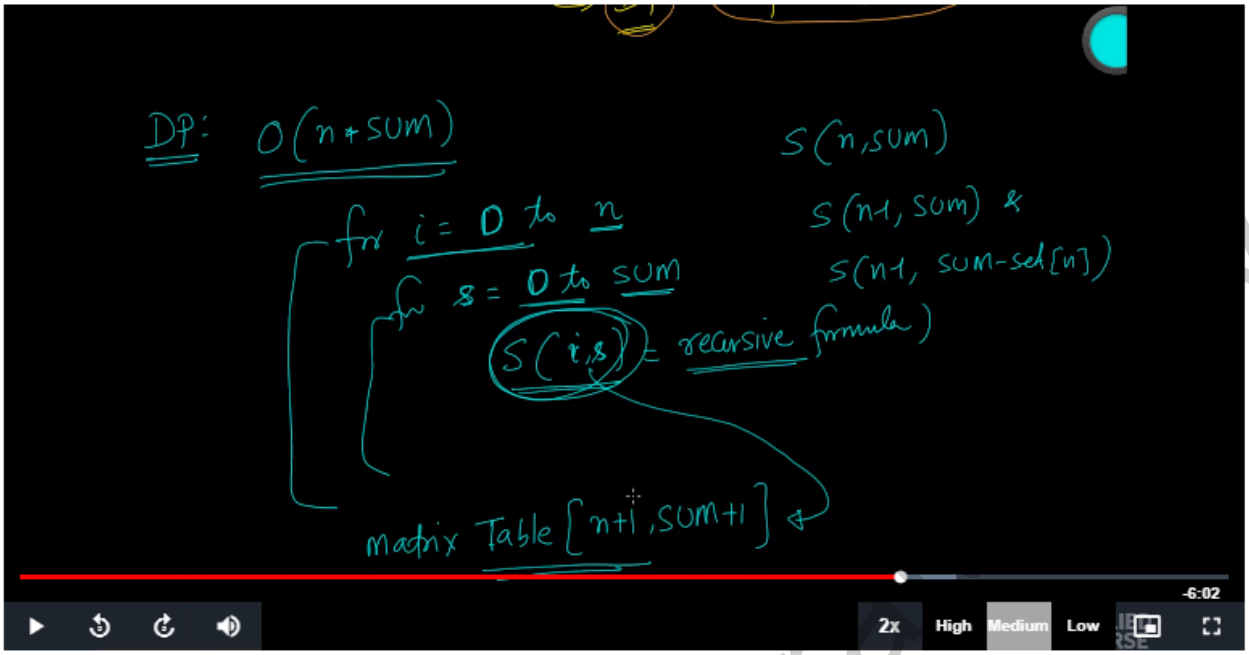
The problem can be solved using recursion as above. Notice that in the base case if we have no elements in the array then we can't find a subset that matches the sum if it greater than 0. If the sum is zero then any subset can satisfy the condition hence we return true.

For the recursive case, there are two cases i) if the last element is not present in the result subset then the problem can be solved by taking the set without the last element ii) If the last element is present in the subset then the remaining sum can be found using all elements in the set without this element. Notice that this recursion equation has optimal substructure.



Timestamp: 12:59

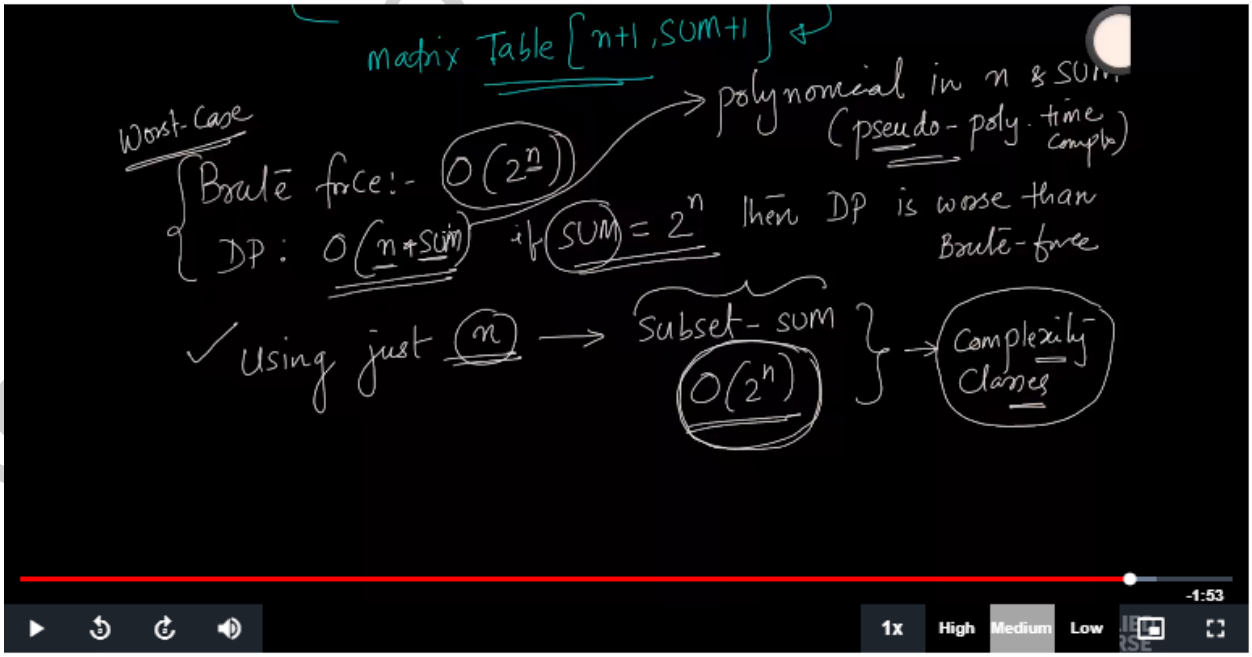
We can find from the recursion tree that there exist overlapping subproblems. But in some cases there might not be a overlapping subproblem then it becomes simple recursion but if it exists then we can use DP.



Timestamp: 16:14

As shown in the above figure, for implementing DP we create a matrix with number of rows to be number of elements and number of columns as sum and we loop through n and sum filling our matrix using the recursion equation.

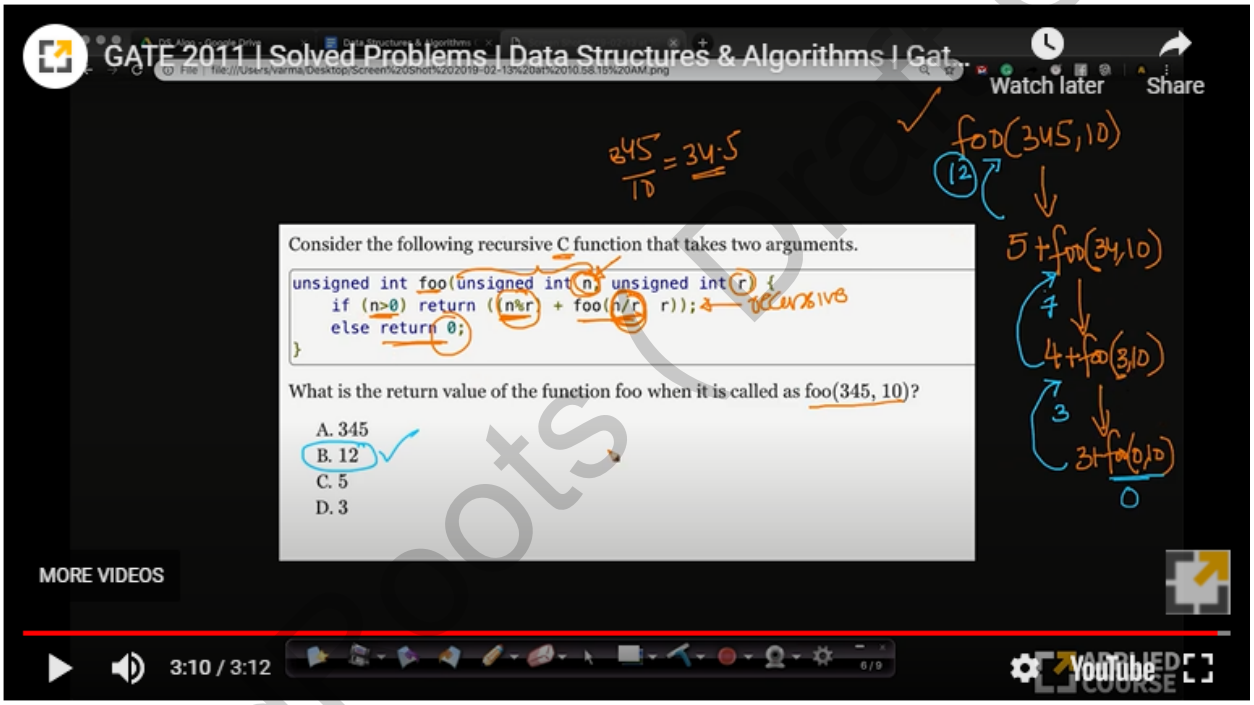
Since we are using two loops then performing O(1) computation inside them our time complexity of subset sum problem using DP is O(n*sum).



Timestamp: 20:23

Although O(n*sum) is polynomial in n and sum, sum in worst case can still be a very large number such in $O(2^n)$ hence our DP time complexity can become $O(n2^n)$ which is worse than brute force. Hence no matter what the algorithm we use the time complexity still remains worse. Hence the subset problem belongs to a special case of problems called complexity classes.

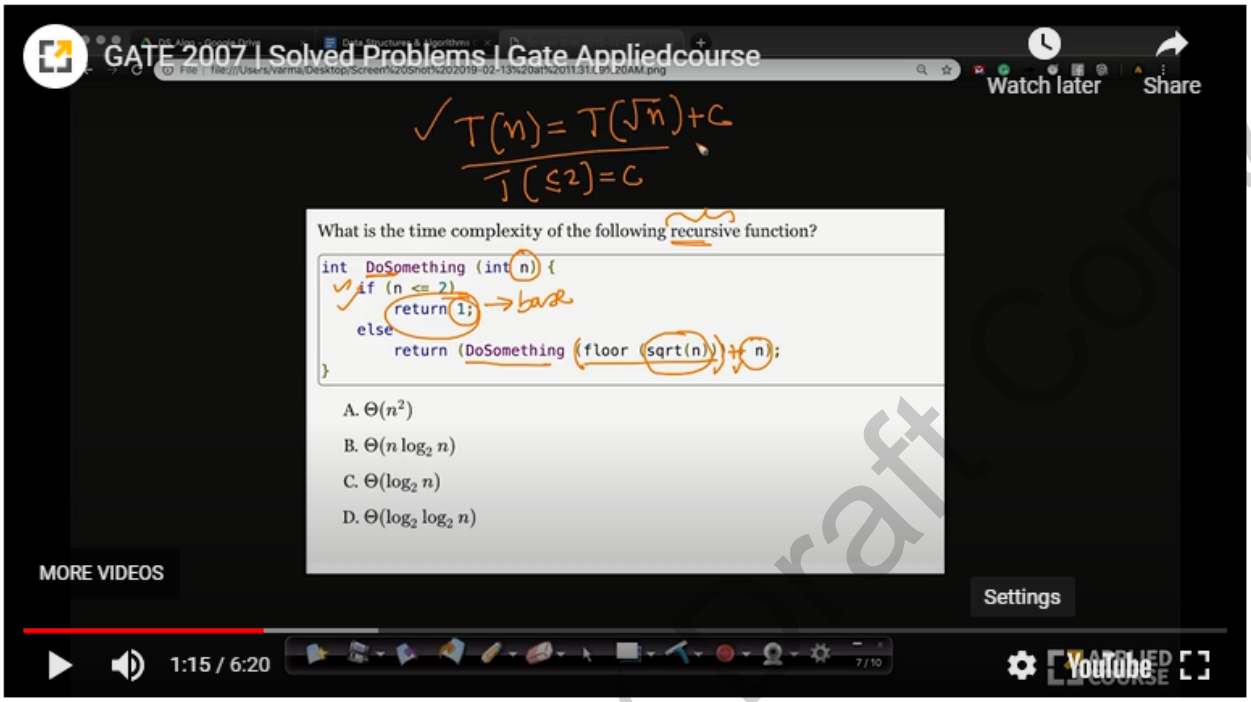
11.23 Solved Problem - 1



Timestamp: 3:10

You can solve the above problem by going through each recursion call.
First when foo is called on 345, n%10 [here r=10] gives 5 and foo is then called on 34, n%10 gives 4 and foo is then called on 3, n%10 gives 3 and then foo is called on 0 since the condition is not satisfied foo returns 0 to its previous call, the previous calls adds its value of n%r=3 to this 0 and returns 3 to the previous call and so on to get the final result 12.

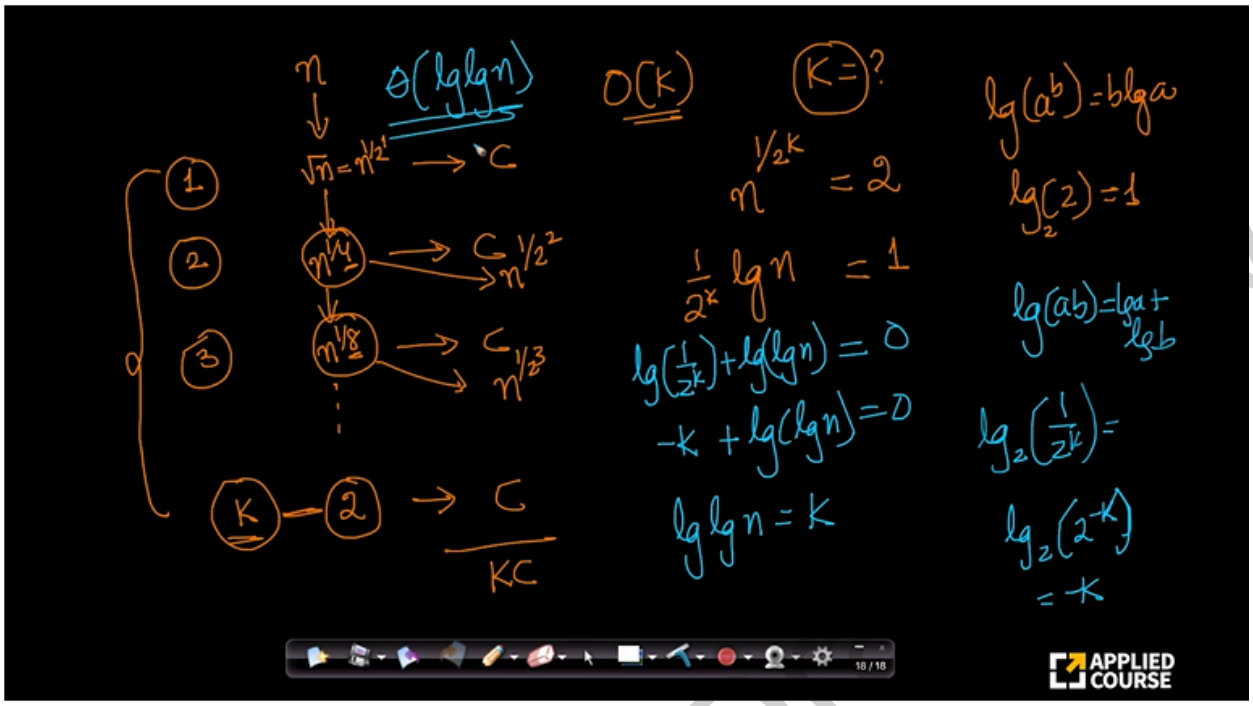
11.24 Solved Problem - 2



TImestamp: 1:15

We can observe from the given question in the above figure that when n<=2 we have the base case and we just simply return 1 but if n is greater than 2 we have the recursion case of calling the function on floor of sqrt of n and adding n. Since we are doing constant Θ(1) work in if and in adding n at each recursive call, the recursion equation is as mentioned in the above figure.

We will solve the recurrence equation using the recursion tree method as below.



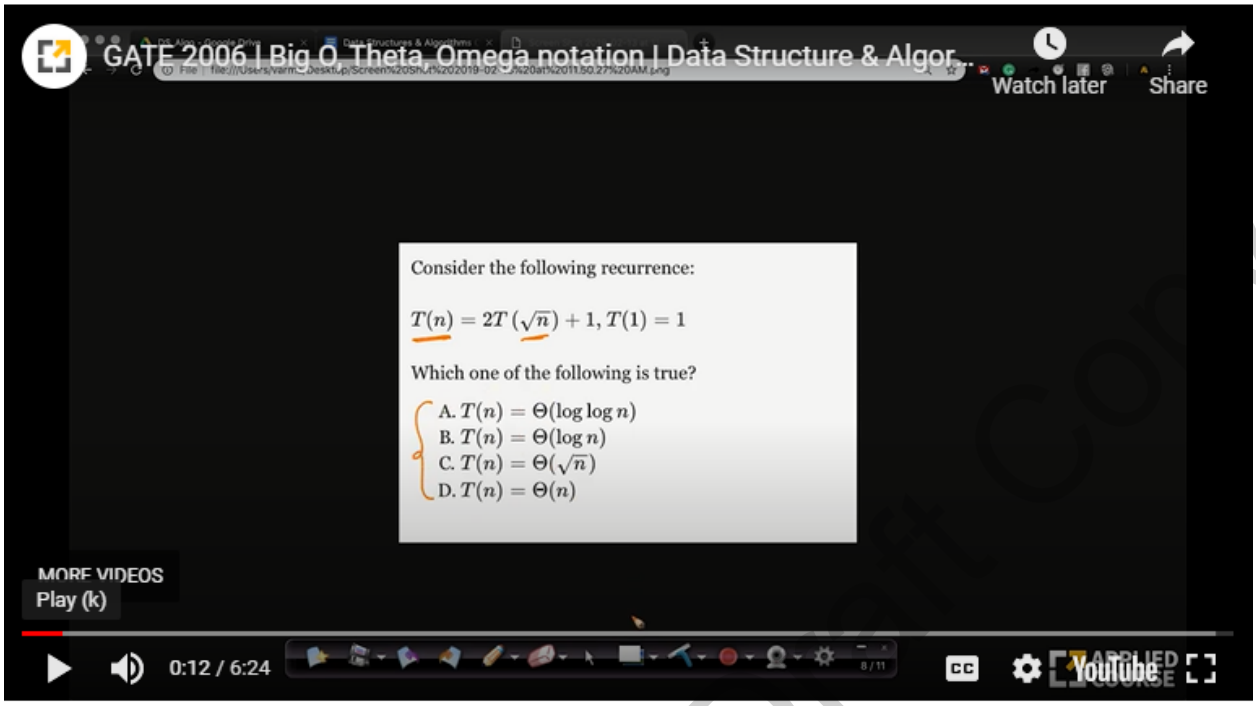
Timestamp: 5:39

We can notice in the recursion tree, for the 1st recursive is called on $n^{1/2}$, the 2nd recursive is called on $n^{1/4}$ (¼ is 2^2) so if k is the last level then the kth recursive call is called on $n^{1/2 \text{ power } k}$. Since the termination condition is n at that level should be less than 2, we have the below equation.

$n^{1/(\ 2 \text{ power } k)} = 2$

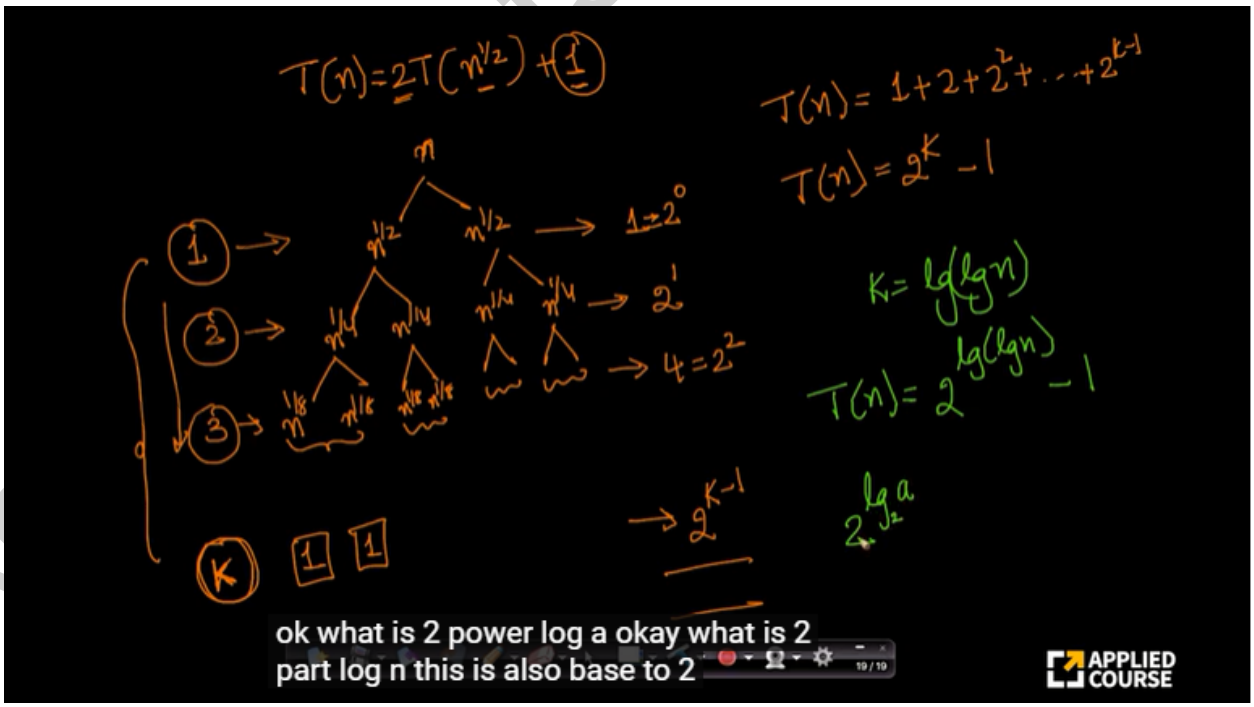
Solving the eqn by taking log as required as in the above figure, we will get K=lg lg n. Hence the total time complexity of the program is lglgn* Θ(1) which is Θ(lglgn) option D.

11.25 Solved Problem - 3



Timestamp: 0:12

In the question in the figure above notice that the base condition is when n=1. Using the given recursive equation we can come up with the recursion tree as below.



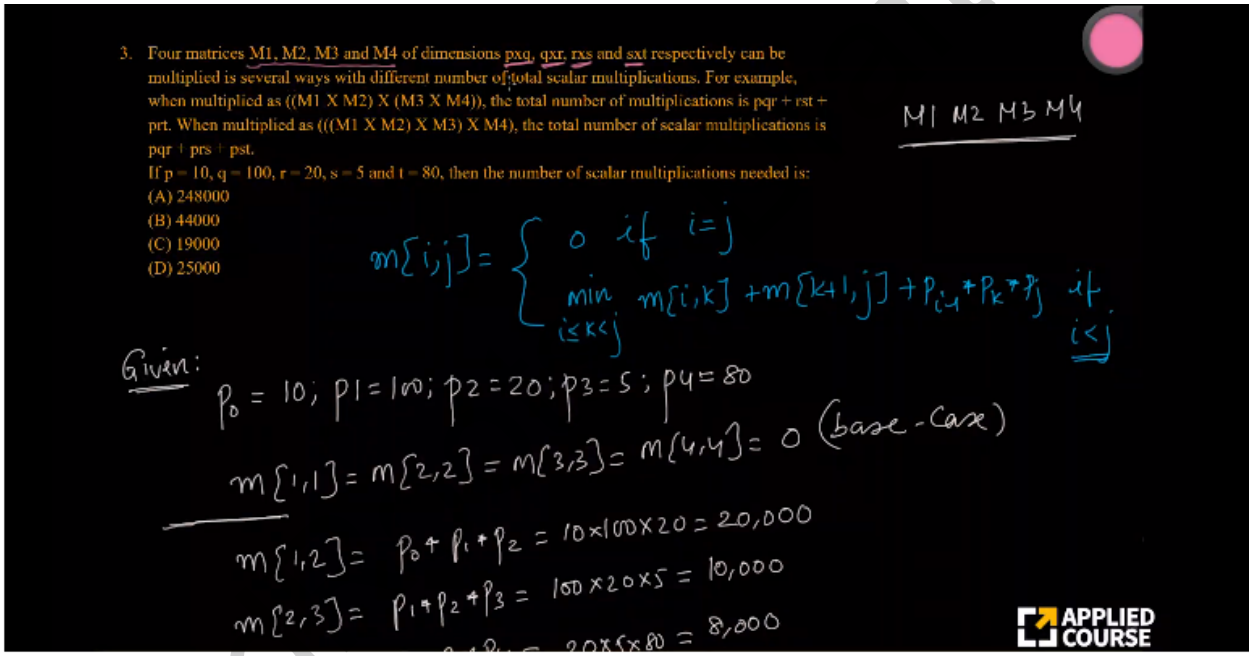
Timestamp: 5:28

As shown in the above figure at each level l we perform 2^{l-1}, due to 2^{l-1} recursive calls at each level l. Having solved the previous problem in 11.24 we know that number of levels k in such a recursion tree is lglgn. Hence we have the time complexity T(n) = 1+2+4+...2^{k-1} as shown in the above figure.

We know the sum of such geometric series is 2^k -1. Here since k=lglgn. The total time taken will be θ(lgn) since 2^{lglgn} = lgn because 2^{lga} =a.

Hence the correct option is b.

11.26 Solved Problem - 4



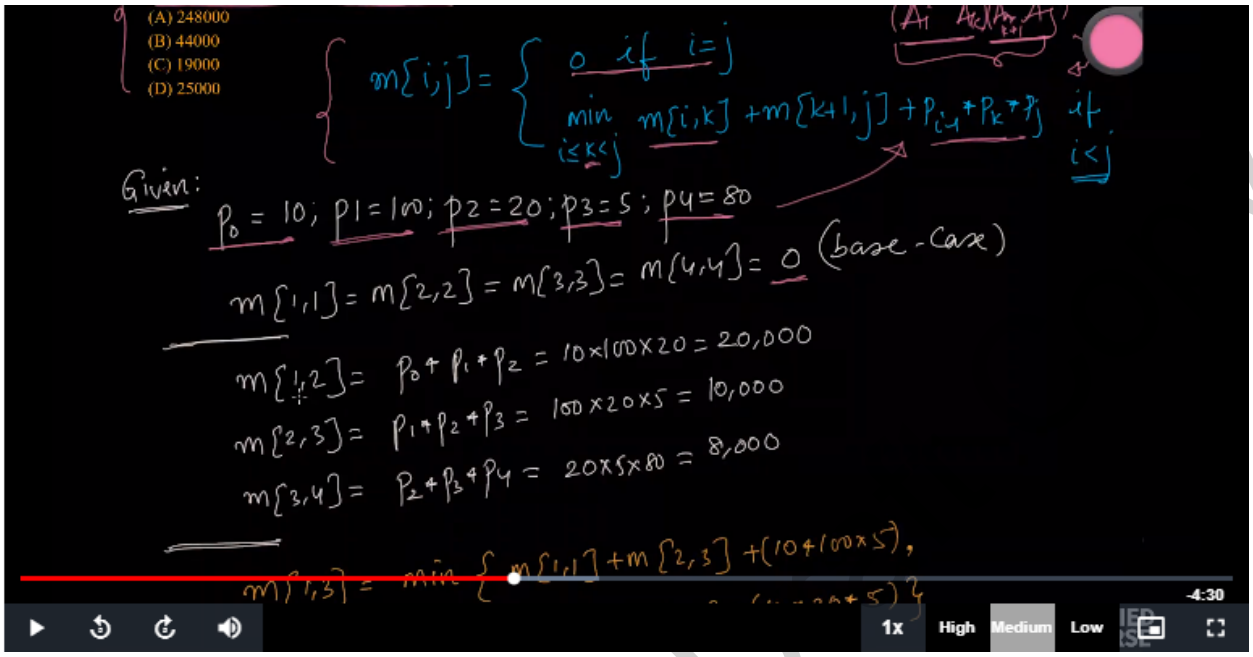
Timestamp: 0:19

This problem is an example of the matrix multiplication problem discussed in 11.20 hence it has the same recurrence equation as shown in the above figure.

As discussed in the matrix multiplication problem, we start with lesser chain lengths and increase the chain length to match the entire sequence.

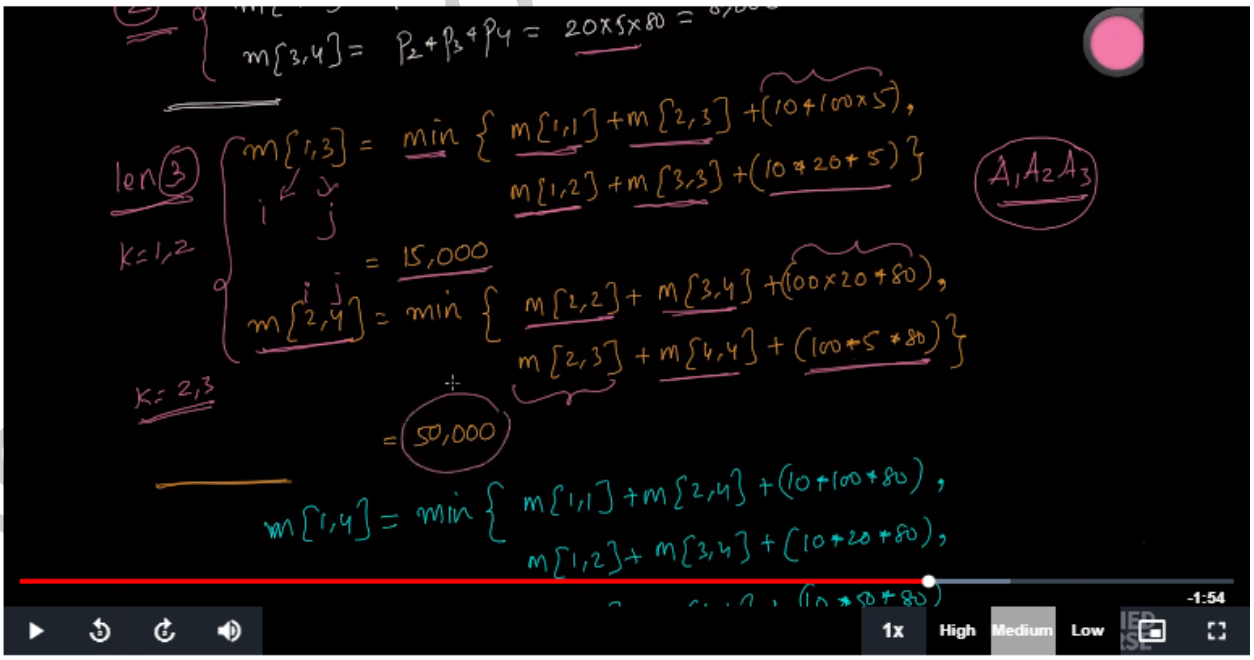
So start with length 1 sequences, since we know that m[i,i]=0 since they are no multiplications required if we have only one matrix, m[1,1]=m[2,2]=m[3,3]=m[4,4]=0 which is the base case.

Next we will be dealing with length 2 sequences and compute m[1,2], m[2,3],m[3,4] as below:

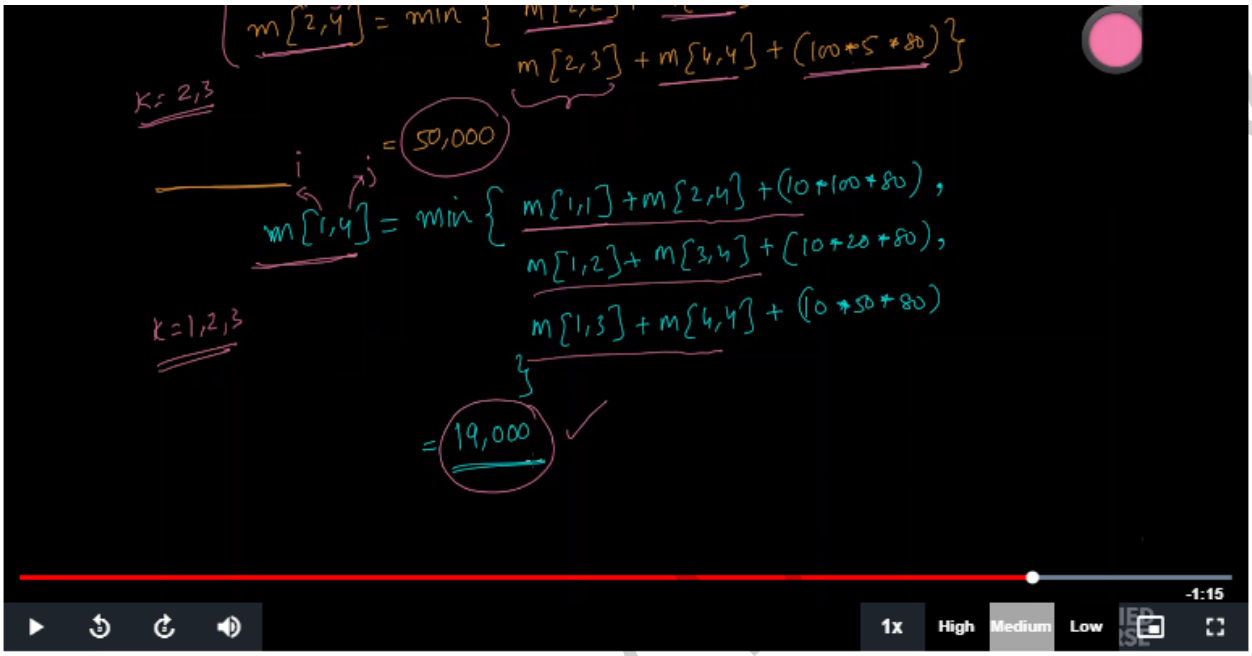


Timestamp: 3:05

Next we will be dealing with matrix chain of length 3, from here we have to choose the way of multiplying the sequences which reduces the number of total multiplications in that chain. Please take a look at the below figure.



Notice in the above figure that we took the number of multiplications for computing m[1,3] as the minimum of finding m[1,3] by splitting into m[1,2] and m[2,3] and combining them or by splitting m[1,3] by splitting into m[1,1] and m[1,3] and combining them.

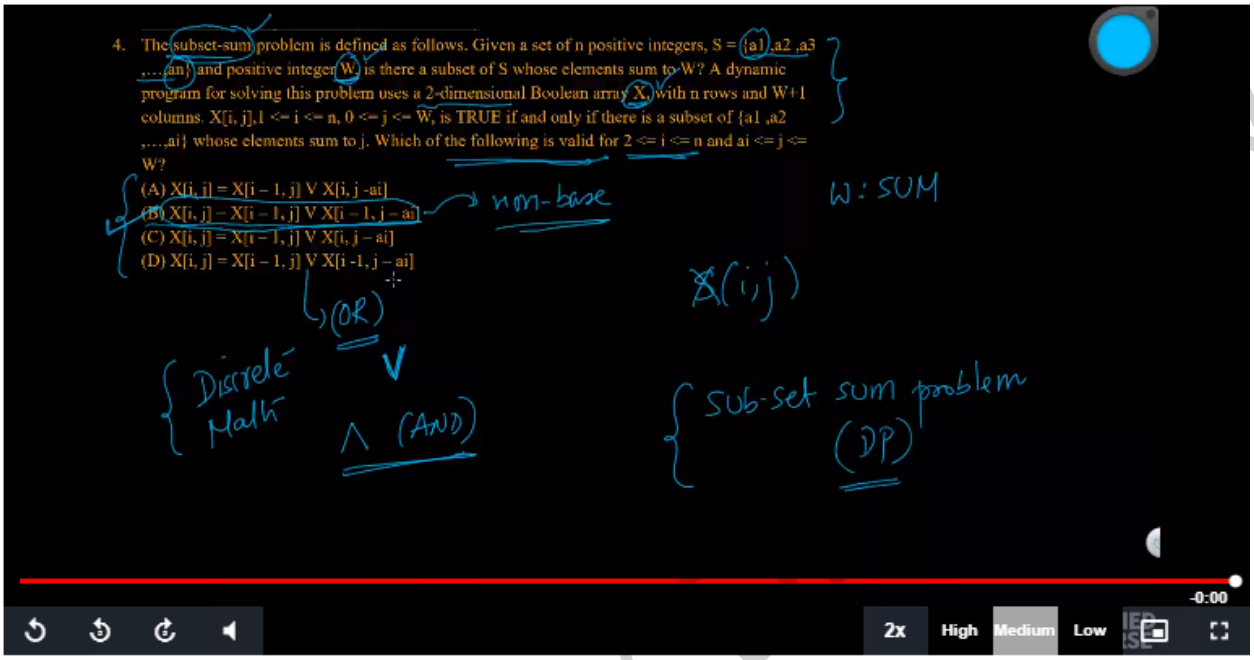


Timestamp: 6:20

Now we finally compute chain of length 4, which will cover the entire sequence.
We choose m[1,4] as the minimum number of multiplications required among splitting m[1,4] into m[1,1] and m[2,4] and the cost of combining them or splitting m[1,4] into m[1,2] and m[3,4] and the cost of combining them or splitting m[1,4] into m[1,3] and m[4,4] and the cost of combining them.

We found it to be 19000 which is the minimum number of multiplications required to compute M1*M2*M3*M4 hence the answer is option C.

11.27 Solved Problem - 5



Timestamp: 4:38

The question in crisp is asking the recurrence relation of our subset sub problem with given sum as w.

The correct answer is option b in the below options.

A. $X[i,j]=X[i-1,j] \vee X[i,j-ai]$

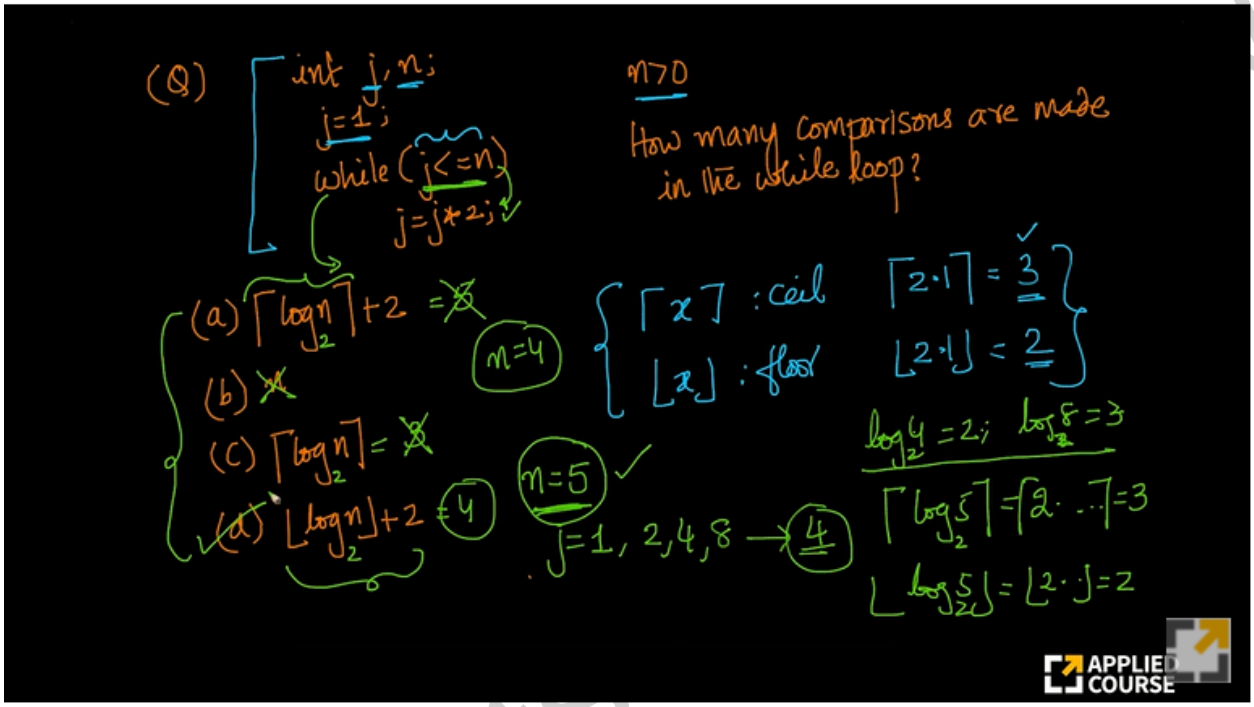
B. $X[i,j]=X[i-1,j] \vee X[i-1,j-ai]$

C. $X[i,j]=X[i-1,j] \wedge X[i,j-ai]$

D. $X[i,j]=X[i-1,j] \wedge X[i-1,j-ai]$

Option B is correct since for computing X[i,j] we check it as the or condition of X[i-1,j] which signifies that in the end subset, element i is not present and X[i-1,j-ai] which signifies that element i is present in the end subset so we can just check whether we can get sum j-ai from the remaining i-1 elements.

11.28 Solved Problem - 6



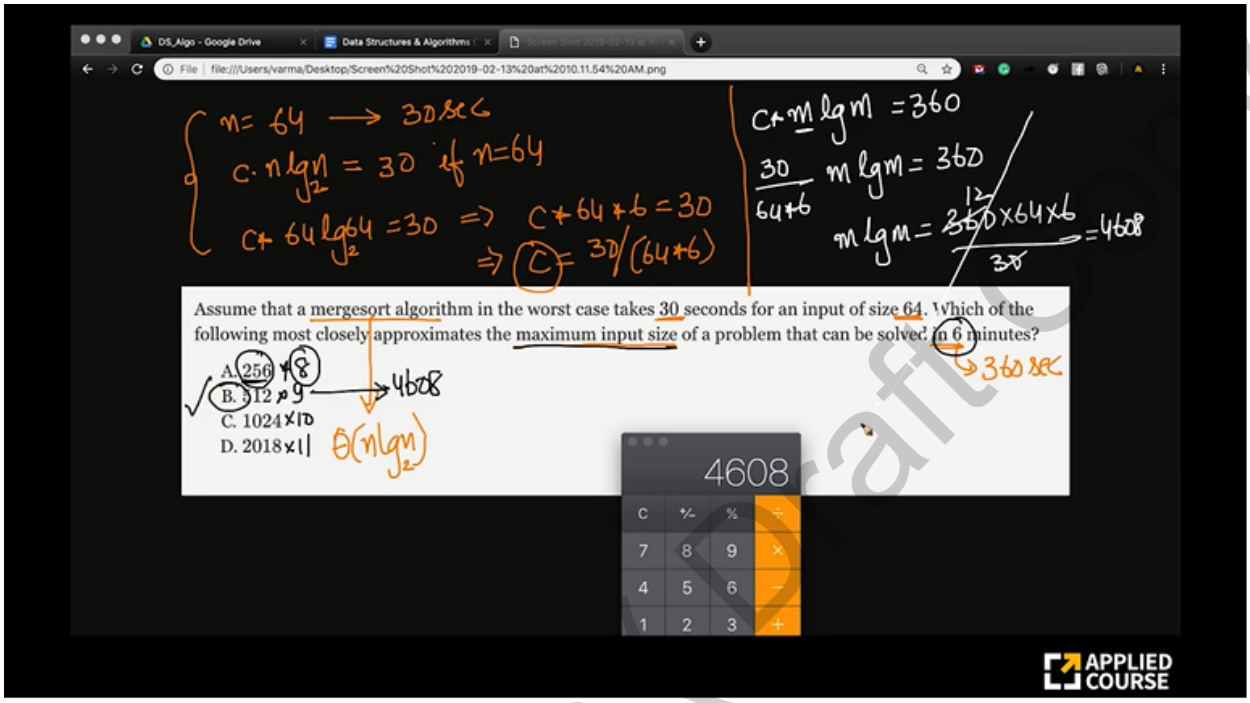
Timestamp: 4:20

The question is to find the number of comparison made in the while loop given in the above figure. Since the options are not asking for time complexity but for the exact answer. We will have to try it for an example n and try it out.

Suppose if we take n=5 then for j=1 it checks makes one comparison, for j=2 it makes the second comparison, for j=4 it makes the third comparison, for j=8 we make the fourth comparison and the condition fails hence there are no more comparison.s

Checking 4 against each options only option d satisfies, hence no of comparisons are as mentioned in the option d.

11.29 Solved Problem - 7



Timestamp: 4:01

In the question in the above figure we have to find out how much is the input size that can be sorted in 6 minutes using merge sort, given that we were able to solve an input size of 64 using merge sort in 30 seconds.

We know that merge sort takes c*nlgn time for some constant c, since we were given that for input size 64 we took that 30 seconds, so solving c*64lg64 = 30 should give us c=30/(64*6).

Now let us say the required max input size is m, then given it takes time 6 minutes = 6*60=360 seconds. We know that c*m*lgm = 360 -> 30/(64*6)*m*lgm =360, solving this by substituting the given options should give the required size of 512. Hence the answer is option B.