

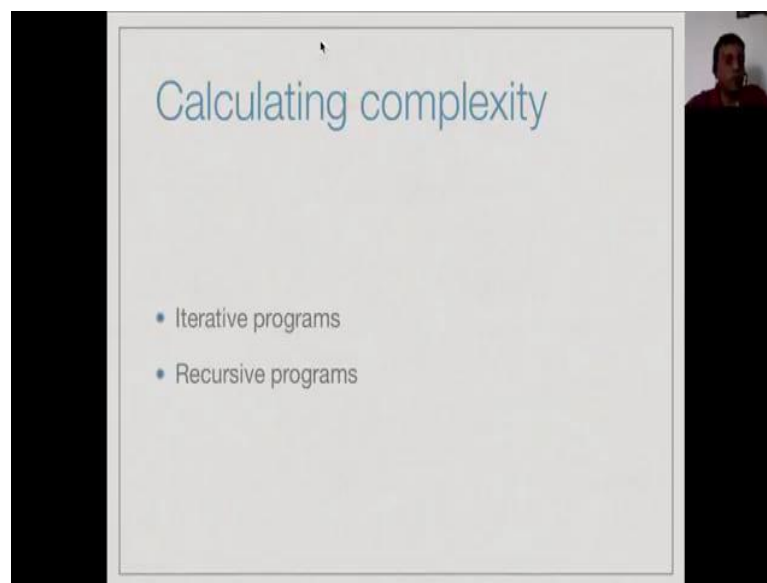
Week - 01

Module - 08

Lecture – 08

The last lecture in this unit, what we are going to do is, actually look at some examples of algorithms and see how to compute their upper nodes.

(Refer Slide Time: 00:12)



So, we will look at two basic classes of algorithms in this unit, in this lecture. So, we will look at some iterative examples and some recursive examples. So, an iterative example will basically be something where there is a loop and a recursive program of course, will be something where you have to solve a smaller problem before you can solve the larger problem. So, you have to recursively apply the same algorithm with smaller input.

(Refer Slide Time: 00:34)

Example 1

- Maximum value in an array

```
function maxElement(A):  
  || maxval = A[0] ←  
  for i = 1 to n-1:  
    if A[i] > maxval:  
      maxval = A[i]  
  || return(maxval)
```

Handwritten annotations:

- $n-1$ steps (next to the for loop)
- comparison (next to $A[i] > \text{maxval}$)
- assignment (next to $\text{maxval} = A[i]$)
- c. operations (next to the if block)
- c. (n-1) basic ops (next to the for loop)
- $O(n)$ (below the for loop)

So, our first example is a very standard problem which you must have done in your basic programming course. Suppose, we want to find the maximum element in an array a . So, what we do is we initially assume that the maximum value is the first value and then we scan the rest of the array and wherever we see a value which is bigger than the current maximum value, max value replace it.

And at the end of this scan we return that value of maxval that we have found which should be the largest value that we shown the entire value. Now, remember that we said that if we have two phases in this case, we have one phase where we do an initialization, we have three phases actually we do an inner loop and then we do a return, it is enough to look at the bottle neck phase, we said that if we have two parts f_1 and f_2 then the order of magnitude of f_1 plus f_2 is the maximum of the order of magnitudes of f_1 and f_2 .

So, in this case it is clear that this loop is what is we want to take the most amount of time. So, it is enough to analyze complexity of this loop. So, now, this loop takes exactly n minus 1 steps. So, the worst case, any input is the worst case, because we must go from beginning to end an order to find the maximum value, we cannot assume anything about where the maximum value lies.

Now, when we are scanning the loop in every iteration we do at least one step. So, this is the comparison, one basic operation and this may or may not happen. So, the assignment happens, if we find the new value A_i which is bigger than maxval. But, since we are

ignoring constants we can treat this as some c operations, some constant number of operations per iterations. So, we have some c times n minus 1 basic operations and if we ignore the c and we ignore this minus 1, overall this algorithm is linear, it takes order n time.

(Refer Slide Time: 02:36)

Example 2

- Check if all elements in an array are distinct

```

function noDuplicates(A):
  for i = 0 to n-1:
    for j = i+1 to n-1:
      if A[i] == A[j]:
        return(False)
    return(True)
  
```

Handwritten notes on the slide:

- For $i=0$, $n-1$ steps
- For $i=1$, $n-2$ steps
- ...
- For $i=n-2$, 1 step
- For $i=n-1$, 0 steps
- Total steps: $0+1+2+\dots+n-2+n-1$ steps
- Sum formula: $\sum_{i=1}^{n-1} i = \frac{(n-1) \cdot n}{2}$
- Complexity: $O(n^2)$

So, let us now move on to an example in which we have two nested loops. So, supposing we are trying to find whether or not an array has all distinct values that is no two values in the array A are the same. So, what we will do is, you will take this array A and then we will compare every A_i and every A_j and if I am ever find an A_i equal to A_j , then I will return false. If I find no such A_i and A_j , then I will not return false I would return true.

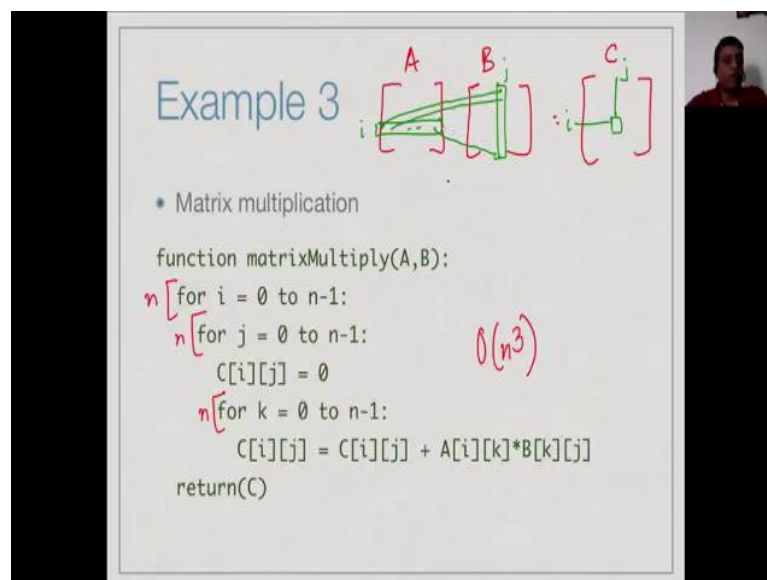
Now, the point is in order to optimize this if I am at position i , then I will only look at elements to it is right. So, I will start with i plus 1 and go to n minus 1 and this will be my range for j . So, in order to not compare A_i , A_j and then A_j , A_i just to avoid this duplicate thing what we have written is for i equal to 0 to n minus 1. So, as I look at each element for j equal to i plus 1 to n minus 1 to it is right, check if A_i is equal A_j .

So, now if I look at the number of times this actually executes, then when i is equal to 0, j varies from 1 to n minus 1. So, there are n minus 1 steps, when i is equal to 1 there are n minus 2 steps and so, on. So, as I go down when i is equal to n minus 1, there will be 1 step, when i is equal to 2 there will be one step. When an i equal n minus 1 the outer loop will terminate, but the inner would not run at all, because you will go from n to n minus 1.

So, overall what we are doing is we are doing 0 plus 1, plus 2, plus n minus 2, plus n minus 1 steps. So, this is a familiar summation, this summation of i is equal to 1 to n minus 1 of i and this you should know is n minus 1 into n, this is a very familiar recurrence and this what we have already seen actually, this is $O(n^2)$. So, we just ignore constant n square by 2 minus n by 2 $O(n^2)$ actually we showed at this theta n square, but for this moment we are only looking at upper bounds.

So, let us say this arguments is $O(n^2)$. So, it is not a trivial, $O(n^2)$ in the sense is not two nested loops of equal size, it is not i equal to 0 to n, j is equal to 0 to n is i equal to 0 to n minus 1 and j equal i plus 1 to n minus 1. But, still this summation 1, 2, 3, 4 up to n is $O(n^2)$ and this is something we will see often. So, useful to remember this.

(Refer Slide Time: 05:10)



Example 3

• Matrix multiplication

```
function matrixMultiply(A,B):
  for i = 0 to n-1:
    for j = 0 to n-1:
      C[i][j] = 0
      for k = 0 to n-1:
        C[i][j] = C[i][j] + A[i][k]*B[k][j]
  return(C)
```

$O(n^3)$

So, this is another example of a nested loops and this is one we choose 3 nested loops. Now, here what we are trying to do is, we are trying to multiply 2 square matrices A and B. So, we have two matrices A and B and we are trying to compute the product C. Now, in this product C, if I want the i, j'th entry, then what I do that look at row i in the first matrix, column j the second matrix and then I have to pair wise I have to do the first entry here, this route I do the first entry that columns I would to multiply those two, then I have to multiply the second entry and. So, on and then I have to multiplied the last entry and then I have to add that.

So, that is what this program is saying. So, this is for each row for i equal 0 to n minus 1,

then for each column j equal to 0 to... So, this is going through all possible entries C_{ij} . Now, I am saying that for this new entry I start for assuming C_{ij} is 0 and then I run through this row k equal to 0 to n minus 1, I look at A_{ik} that is the k th element in the row, B_{kj} the k th element in this column, multiply them and added to C_{ij} . So, this is a loop outer loop of size n , this is another outer loop, inner loop of size n and the inner most loop of size n and this in order n cube. So, this is an natural example of an n cube value.

(Refer Slide Time: 06:33)

Example 4

- Number of bits in binary representation of n

```

function numberOfBits(n):
    count = 1
    while n > 1:
        count = count + 1
        n = n div 2
    return(count)
  
```

Handwritten annotations on the slide:

- A binary tree for $n=9$: $9 \rightarrow 4 \rightarrow 2 \rightarrow 1$.
- The binary representation of 9: 1001 .
- The formula $\log_2 n$ with a note "integer division".
- A note "count = 4" next to the binary representation.
- A note "2x2x1" with an arrow pointing to the binary representation.

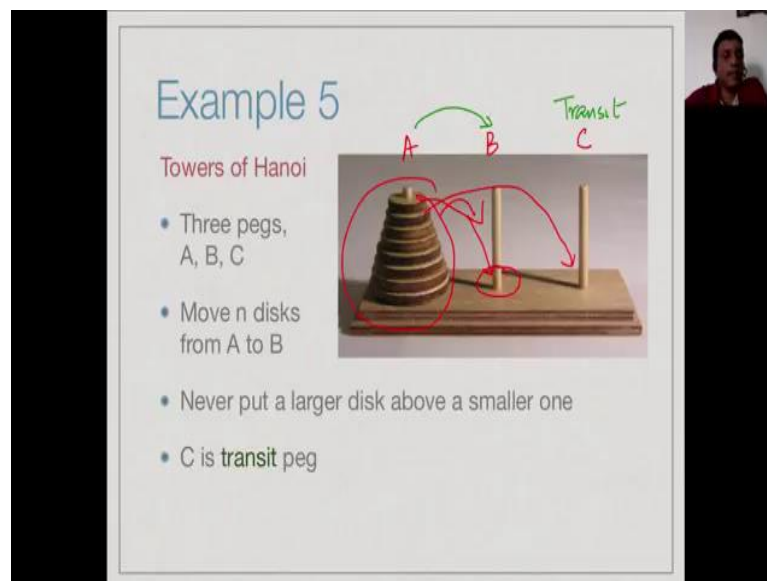
So, our final iterative example is one to find the number of bits in the binary representation of n . So, this is just the same as dividing n by 2 until we reach 1 or 0. So, let us assume that n is a non-negative number. So, n is 0 or 1. So, we assume by send that the number of bits is at least 1 and then. So, long as we have a number which is bigger than 1, we will add one more to the count number of digits and then this is a short form for integer division. So, we will replace n by n by 2.

So, for instant supposing we start with the number like 9, then we will start with count is equal to 1, because that is what I said. When a while n is bigger than 2, I will divide by 2 and add 1 becomes. So, I will replace count make it 2, now make this 4, then I will say that still greater than 1. So, I will make this 3 and I will make this 4 by 2, then I will say this is still bigger than 1. So, I will make this 1, then I make this 4. So, now, I have count equal to 4 and n equal to 1. So, this loop exits and I return count. So, it says that it requires 4 bits to that print number 9 which is correct, because the number 9 and binary it is 1001. So, now what is the complexity of this loop? Well, how many times does this

execute? Well, it will execute as many times it takes for n to come down from its value 2. So, I want n , n by 2, n by 4 etcetera to come down to 1.

So, how many times should I divide n by 2 to reach 1 and this is the same as going backwards, how many times should I multiply 1 by 2 to reach n . So, dividing n by 2 repeatedly to reach 1 is the same as multiplying 1 by 2 repeatedly to reach n and this is nothing but, the definition of the log, what power of 2 reaches n . So, this iterative loop actually though does not decrement by 1, decrements by halving at each time, we can still calculate it explicitly as requiring $\log_2 n$ steps.

(Refer Slide Time: 09:00)

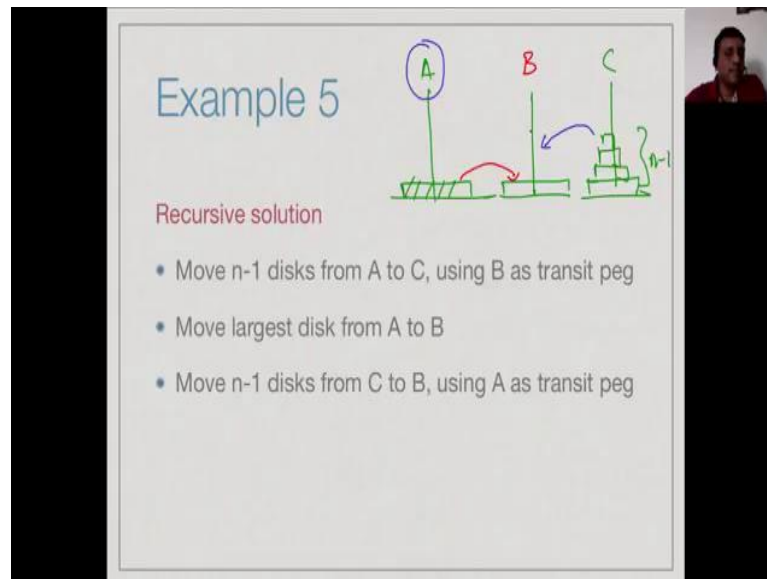


So, we have seen iterative examples of linear time, quadratic time, cubic time, this n , n squared, n cube and also a linear example with $\log n$ time. So, now let us look at one recursive example to see, how we would try to do this when we have the recursive solution. So, we would not look at a formal algorithm, but rather than formal puzzle. So, this is a well known Towers of Hanoi. So, in the towers of Hanoi person we have as we seen this picture here, we have 3 wooden pegs which we will call for the moment A, B and C.

So, we have takes A, B and C and our goal is to move these n disks from A to B. So, the thing that we are not allowed to do is to put a larger disk on a smaller disk. So, if we take the small disk and move it here. So, we move the first disk here, then we must take the second disk and move it there, because we cannot put the second disk on top of the first disk. So, the goal is to do this in an effective way.

So, the actual goal is to move everything from A to B and this is an intermediate thing, because as we saw, we move the first disk from A to B we are stuck, we cannot move anything else (Refer Time: 10:09). So, you must use C as a kind of transit to take temporary obnoxious peg in order to do this job. So, if you have not seen this problem before, you might want to think about it in a spare time, but this is a very classical problem that it has a very standard recursive solution.

(Refer Slide Time: 10:27)



Example 5

Recursive solution

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

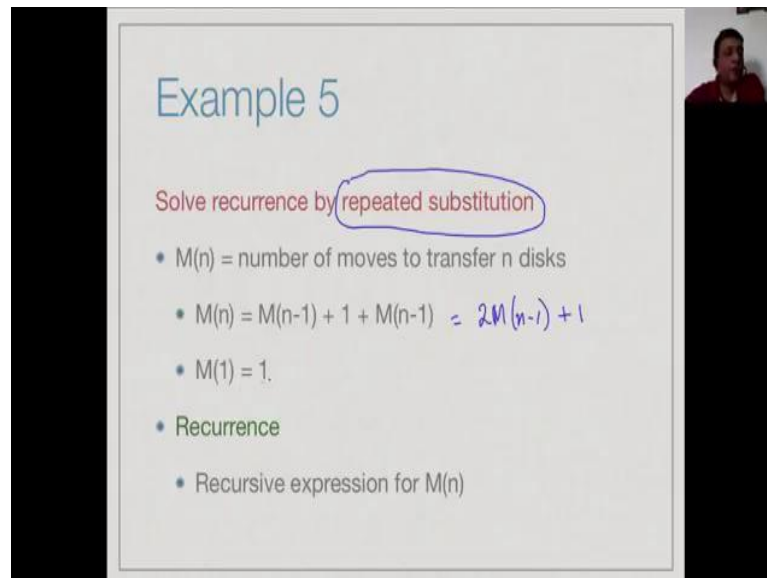
And the standard recursive solution is the following that you first assume that you know how to solve the problem for n minus 1 disks. So, at this moment you want to move n disks from A to B. So, what we do is you first move n minus 1 disks. So, you have on A only the bottom disk left and you have now B empty and you are move all the other n minus 1 disk to C. So, there are now n minus 1 disk here. So, you are assume that you can do this using B as my transit pegs.

So, now I move things from A to C, now what I do, then move this disk here. So, I now have disk here and I no longer have anything here. So, now, I have one biggest disk on B. So, I can put anything on it and I have n minus 1 disk on C. So, what I do is I apply the same algorithm for n minus 1 to move things from here to here using my A as my transit pegs.

So, this the recursive way to solve the problem, you move n minus 1 disks A to C, move the biggest disk from A to B and then move n minus 1 disk, back up C to B. So, the question we want to ask us I suggest, how many times to be move disks in this

procedure?

(Refer Slide Time: 11:37)



Example 5

Solve recurrence by repeated substitution

- $M(n)$ = number of moves to transfer n disks
- $M(n) = M(n-1) + 1 + M(n-1) = 2M(n-1) + 1$
- $M(1) = 1$.
- Recurrence
- Recursive expression for $M(n)$

So, supposing you write M of n to indicate the number of moves we need to transfer n disks from one peg to another peg. So, what we have seen is that in order to transfer n disks, we first transfer n minus 1 disks from A to C , then we transfer one disk from A to B and then n minus 1 disk back from C to B . So, it is M of n minus 1, this is to transfer n disks plus 1 for that 1 disk in an M of n minus 1. So, this I can simplify as 2 times M of n minus 1 plus 1.

So, M of n in general is 2 times M of n minus 1 plus 1 and if I have only one disk to transfer, then there is no problem we can do it directly one steps. So, M of 1 where n is equal 1 is 1. So, this kind of expression of describing M n recursively in terms of smaller values of capital value, this called a recurrence. So, we have a recursive expression for M n , now we have to solve this. So, where we are going to solve this is to use a mostly repeated substitution, we are going to repeatedly use the same rule to simplify this expression, until we reach everything in turns M 1 and then we can plug in the value.

(Refer Slide Time: 13:00)

Example 5

Complexity

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

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

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

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

$$= 2^2(2M(n-3) + 1) + 2 + 1 = 2^3M(n-3) + (4+2+1)$$

$$= \dots$$

$$= 2^kM(n-k) + (2^k - 1)$$

$$= \dots$$

$$= 2^{n-1}M(1) + (2^{n-1} - 1)$$

$$= 2^{n-1} + 2^{n-1} - 1 =$$

$$= 2^n - 1$$

Handwritten notes on the right side of the slide:

- $M(n) = 2M(n-1) + 1$
- $M(n-1) = 2M(n-2) + 1$
- $2^2 - 1$
- $2^3 - 1$
- $n - (n-1) = 1$
- $2 \cdot 2^{n-1} = 2^n$
- $M(n) = 2^n - 1$

So, we start by the basic expression. So, M of n is 2 times in n minus 1 plus 1, now what we do is we substitute for M n minus 1 the same expression in term that n minus 2. So, M n minus 1 by the same expression, it is 2 times M n minus 2 plus 1, because in general for any M we have n minus 2 times M of n minus 1 plus 1. So, this is a general expression. So, we are taking...

So, we do this and we simplified, we get two terms 2. So, we get 2 square coming from this and n minus 2 and then we take 2 times 1 that gives us this 2 plus 1. So, we have just rewritten this as 2 square n minus 2. Now, again if you take this expression 2 times M n minus 2 that becomes 2 times M n minus 3 plus 1 and then this 2 square plus 1 is, this 2 square remember is 4. So, this I get 4 inside and 2 squared times 2, 2 cubes. So, I get 2 cube M n minus 3 plus 2 square plus 2 plus 1.

Now, you can see that if I do this k times I will have 2 to the k M of n minus k . Remember, everywhere I have this and I have this, this is the same number and this is 1 plus 2 plus 4, in next time will be 1 plus 2 plus 4 plus 8. So, this is actually 2 to the k minus 1 to this is nothing but, 2 cube minus 1 it is nothing but, 2 squared minus 1. So, in general after k steps I had this, now when I do this n minus 1 times then n minus n minus 1 is nothing but, 1 n minus n plus 1.

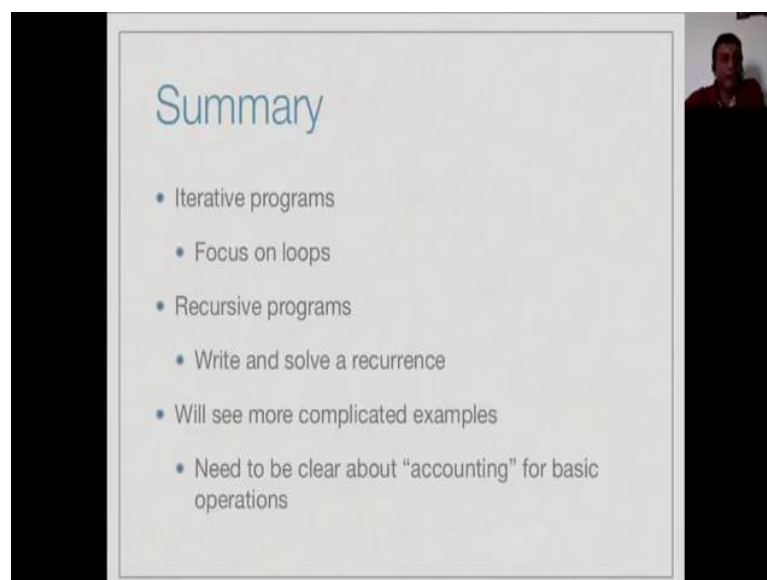
So, if I do this 2 n minus 1 time k n minus 1 then n minus k becomes 1 and this element n minus 1. So, since n minus 1 is 1 I can this omitted from the since. So, I have 2 to the n minus 1 plus 2 to the n minus 1 minus 1. But this is nothing but, 2 times 2 to the n minus

1 which is 2^n . So, I can combine these are 2^n . So, therefore, this skips as by this repeated expansion, substitution whatever you would like to call it.

We have that $M(n) - M(n-1)$ is 2^{n-1} in other words, it takes an exponential number of steps in order to solve this puzzle. So, there is a very famous story by authorship clock, which talks about this some temple word these pegs are there and it has 64 such disks and it says that the world will come to an end when the 64 disk are transfer. So, you can think about how much time it will take to transfer 2^{64} disks, in order to solve the puzzle with 64 disks.

Remember, we said that 2^{30} is about 1 billion. So, this the enormous amount of time I do not think you really need to worry about this as you serious problem if that is read the case.

(Refer Slide Time: 16:00)



So, to summarize we have looked at some examples just to illustrate the flavor of how we apply the concepts we are studied in terms of ago in an actual algorithm, how do you look at an algorithm actually extract it is complexity. So, for in iterative program basically focus on the loops, because the loops I what take of the time and you have some times to be update clever about trying to understand, how many times loops executes, where recursive programs we show on the one example, you will see more as we go along.

But, the main idea is you express the time complexity for program as a recurrence, you write $T(n)$ of n the time taking for n steps, in terms of a smaller value which is obtain from

the recursive call. So, for the Hanoi case we had n and n minus 1. So, in order to solve the problem for n disks we need to solve it problem twice for n minus 1 disks. We will of course, find examples will do not fit any of these, it will not be a simple loop that we can calculate and then we will have to be a little more careful about how we actually count the operations.

So, in a sense actually estimating the efficiency when I do this, it is really like accounting. So, you have kind of keeping track of all the basic operations and you have to do a good job of making sure that you do that to keep track of them in the best possible way. So, that you get an actual picture of the arts.