FADML

Scribed Lecture - 1

6th January 2025

IIT Kharagpur (PGDBA)

Abhijeet Yadav (24BM6JP01)

Foundation of Algorithm Design

Algorithm design is the art and science of developing a logical and efficient step-by-step procedure to solve computational problems. It plays a foundational role in computer science and software development, providing the blueprint for writing code and solving real-world challenges effectively. Algorithm design goes beyond simply solving a problem—it addresses the **"why"** and **"how"** of creating a solution that is correct, efficient, and practical.

Now we learn how to approach and write efficient algorithms. What is the core foundation of everything?

To address this, we develop a single method that should guide us in our future endeavors to come up with efficient algorithms.

Pavement of solution:

1. Initial Solution

(a) Recursive formulation
(b) Correctness of solution
(c) Analysis

2. Explore Possibilities

(a) Explore recurrent structure
(b) Decomposition
(c) Re-composition Properties

3. Solution refinement

(a) Balancing the Split
(b) Identical Sub-problems

4. Final solution

(a) Traversal of recurring structures
(b) Pruning

5. Data Structuring

(a) Re-use computation
(b) Space complexity

6. Implementation

Only one solution, in most cases, is not perfect. There's always room for improvement. After finding an initial solution, our job is to look for other options, examine them carefully, and create the best possible algorithm. To understand this approach better, let's look at the following example problem.

To find out the largest of n numbers:

To solve the problem of finding the top k-elements, let's first address the simpler task of finding the maximum or minimum element. Several methods can be used for this:

1. Method 1 (M1): Sort the array and pick the desired element.
2. Method 2 (M2): Initialize two variables and iterate through the array, comparing each of the (n−1) elements.

Once we apply both the methods, we find that M2 is faster than M1. This is because, in M1, we are comparing every number with the rest of the numbers in the array. In essence, we are comparing $\binom{n}{2}$ whereas M2 requires only 2n-2 comparisons.

The number of comparisons is used to measure the complexity of an algorithm because, for large n, comparisons make up the majority of the operations. This is called the time efficiency of the algorithm.

Can we make the solution better than this?

M3: Keep comparing two at a time parallelly. Declare winners in the stage and then follow the same method among the winners. Recursively apply this method. We try and count the number of comparisons required for an array of size n to find the maximum, supposing p =[$\log_2 n$],is,

No of comparisons =

$$\frac{n}{2}+\frac{n}{4}+\frac{n}{8}+\cdots+\frac{n}{2^p}=\frac{n}{2^p}(2^p-1)=n-\frac{n}{2^p}\approx n-1 \text{ for large } n.$$

Pseudo code:

Max(L) ← T(n) number of

L = $\{x_1, x_2, x_3, \ldots\ldots\ldots\ldots\ldots, x_n\}$

if (|L| = 1) return x_1 (*cardinality of L = 1)*

L' = L – $\{x_1\}$ ← Decomposition

m' ← max(L') ← Sub-Problem

if ($x_1 > m'$)

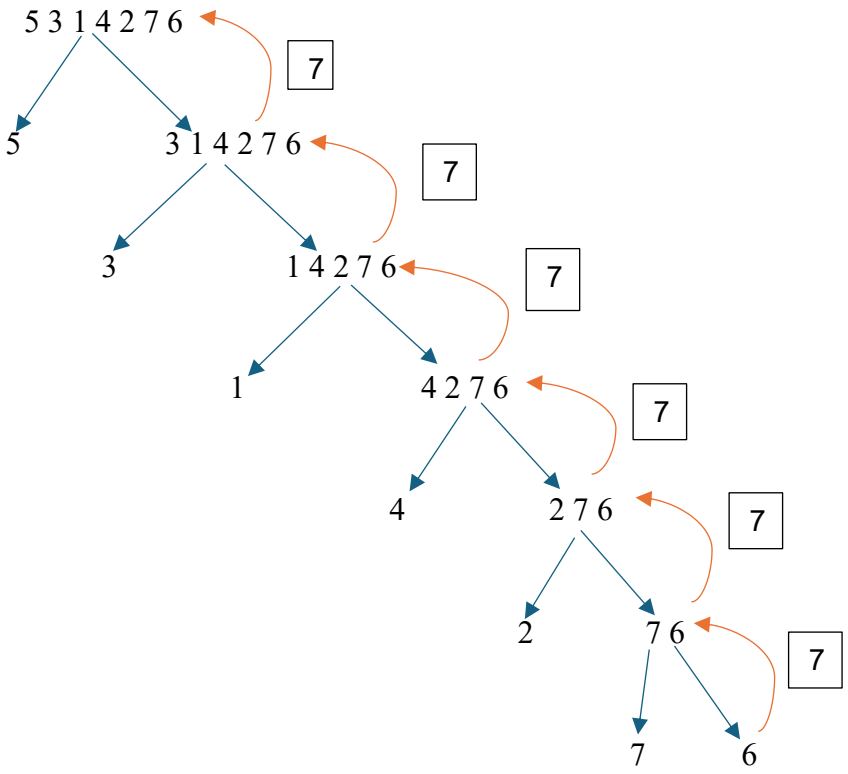
$return\ x_1$

else

$return\ m'$

Recomposition

Example 1:



From the above tree diagram, it is clear that for an array having 7 numbers requires total 6 comparisons.

$T(n) = T(n-1) + 1$ where T(n) = Number of comparisons required

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

$= T(n-3) + 3$

$= T(n-4) + 4$

……

……..

$T(n) = T(1) + (n-1)$

0

$T(n) = n-1$

This decomposition is not general enough. Now, we will try and see if it is the only way to find largest.

Some observations that can be made are,

1. We can remove any element, not just the first.

2. We can bisect the array and compare.

Essentially, we can just divide the array into two non-empty sets L1 and L2.

Method 2: m ← max2(L)

{ if (|L| = 1) return x_1

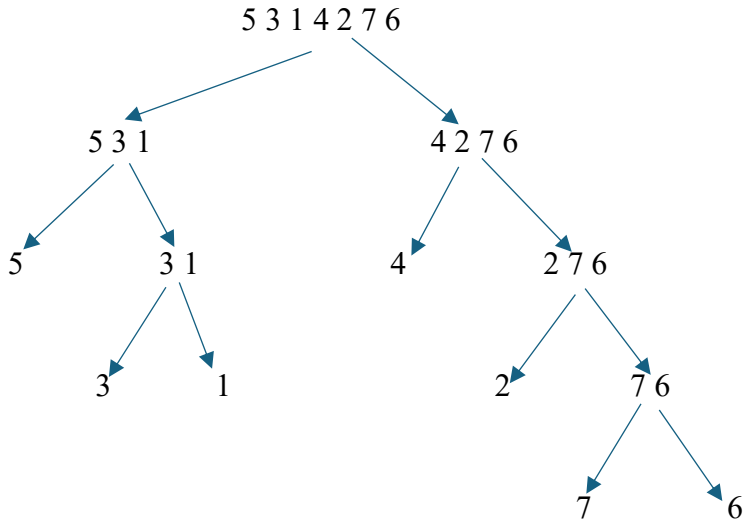
Split L into two non- empty set L_1 and L_2

m_1 ←max2 (L_1)

m_2 ←max2(L_2)

if (m_1>m_2) return m_1 }

Example 2:



In this case also 6 comparisons are required.

$$T(n) = \begin{cases} T(k) + T(n - k) + 1 & n > k \\ 0 & n = 1 \end{cases}$$

Now think of this in terms of a tournament, To declare a sinner among n teams, there needs to be (n-1) comparisons.

Can we do this in less than (n-1) matches? No! Because, in order to eliminate a person, you need a match. So, we have an algorithm to find the maximum (n-1) comparisons and we know that it can't be done in lesser number of comparisons. So, this is optimal.

Now consider the problem: Return the max and min of L

<m, M> ← min Max(L)

If (|L| =1) return (x_1, x_1)

Split L into L_1 and L_2

(m_1, M_1) ← min Max (L_1)

(m_2, M_2) ← min Max (L_2)

If (m1 > m2) {

m← m2

if (M1 > M2)

M ← M1

M ← M2

else { m ←m1

If (M1>M2) M← M1

else M ← M2

T (n) = T(k) + T(n-k) +2

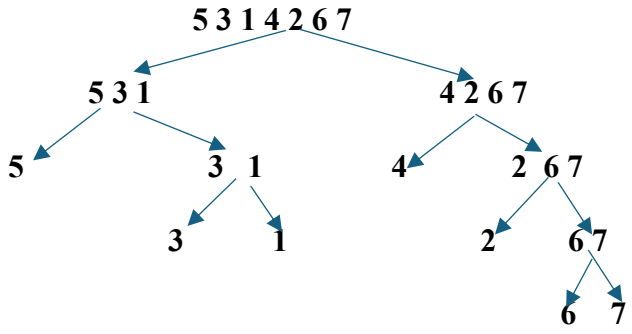
and T(1) = 0

T(n) = T(n-1) + 2

…….

…….

T(n) = T(1) + 2(n-2)



Here we have Proof of Correctness by the Principle of Mathematical Induction.

Here we have changes the base condition to see the result changes?

<m, M> ← min Max(L)

If (|L| =2) , if (x1 > x2) return (x1, x2)

else return (x2 > x1)

Split L into L_1 and L_2

(m_1, M_1) ← min Max (L_1)

(m_2, M_2) ← min Max (L_2)

If (m1 > m2) {

m← m2

if (M1 > M2)

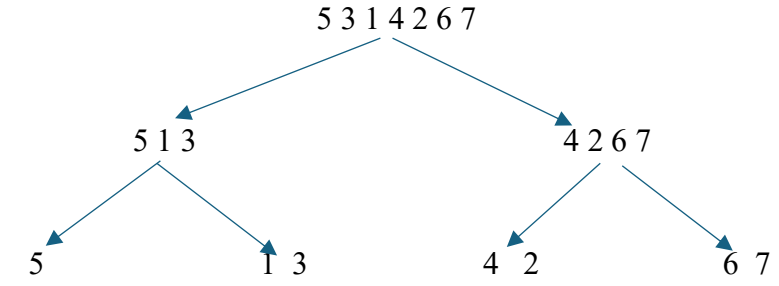
M ← M1

M ← M2

else { m ←m1

If (M1>M2) M← M1

else M ← M2



$$T(n) = \begin{cases} T(n-k) + T(k) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

As can be seen from above, two different ways of splitting the array may produce two different complexities. We observe that when the array size is even, going for a split into two even sized arrays gives us the result in lowest

possible cost. If the array is of odd size, then we must split it into arrays one of size odd and the other of size even anyway.

If we choose k=1, we get T(n) = 2n − 3. Choosing k=2 yields us T(n) = $(3n/2) - 1$ which is evidently better.

So, we see that, in order to get to a better algorithm, we had to open up the tree and design it in such a way that its depth is less. A balance tree will provide the better solution i.e. lesser comparisons will be required the minMax values. Balanced tournament split which minimizes the height of the tournament allows a more efficient algorithm which can be extended to sorting.

Recursive definitions:

max: Any split is fine. Choose 1, n − 1. The number of comparisons is (n – 1).

maxmin: At each level, split the array into two arrays of size 2, n − 2. The number of comparisons is (3n/2) -2.