# Class Notes

02 October 2022 09:35

## Algorithms

1. What is an algorithm : Finite sequence of unambiguous step by step instructions followed to accomplish a given task
2. Properties of Algorithm :
    a. Input, Output,
    b. **Finiteness**: Must terminate after finite number of steps
    c. **Definiteness**: The steps of the algorithm must be precisely defined. Each instruction must be clear and unambiguous
    d. **Effectiveness**: The operations of the algorithm must be basic enough to be put down on pencil and paper
3. Analysis of Algorithms:
    a. Primitive Operation method : Using RAM model, compute the number of primitive operations, such as increment, assignment, comparison etc
    b. Basic Operation method : Identify the operations basic operation, and see how many times that is executed
    c. We always look at the worst case time of an algorithm

## Time Complexity and Notations

1. What is best and worst case ?
2. Asymptotic notation : Used to denote how running time increases with the input size, particularly as the size of the input increases without bound or reaches infinity
3. Asymptotic notation is used to compare algorithms based on the order of growth of their basic operations

### Notations

1. O(g(n)) is the set of all functions with a lower or same order of growth as g(n) (to within a constant multiple, as n goes to infinity). It's also the upper bound of time complexity.
2. Big Omega(g(n)), stands for the set of all functions with a higher or same order of growth as g(n) (to within a constant multiple, as n goes to infinity). It's also the lower bound of time complexity
3. Big theta(g(n)) is the set of all functions that have the same order of growth as g(n) (to within a constant multiple, as n goes to infinity). It's the same order as the algorithm we are analyzing.

### Proving Time complexity in Notations

1. Big O notation example: Solve problems using f(n), g(n) and ceiling function. Need to do some mathematical jugglery and induction to prove
2. Big O notation proof : Not required
3. Big Omega notation example: Solve problems

**Correctness of Algorithm**
An algorithm is said to be correct if, for every input instance, it halts with the correct output

## Analyzing Recursive Algorithms

### Iterative method

1. Analyze function in terms of basic operations and write in function format.
2. $$T(n) = \begin{cases} \boxed{3} & \text{if } n=1 \\ \underline{T(n-1)+7} & \text{otherwise} \end{cases}$$

   Decompose the function and write in terms of I and N. Substitute boundary condition and come up with a polynomial

### Recursion Tree

1. Write the recursion like a tree with base condition at the top and branches beow

2. No: of nodes at level h $= 3^h$
$$= 3^{\log_4 n}$$
$$= \underline{n^{\log_4 3}}$$
3. Solve a lot of problems

### Master Method

1. Case 1 : f(n) grows polynomially slower than $n^{logba}$
2. Case 2 : f(n) and $n^{logba}$ grow at similar rates
3. Case 3 : f(n) grows polynomially faster than $n^{logba}$

Case 2 Generalization

Solve a lot of problems

## Abstract Data Type

This is basically an API, only provides the methods and structures required to access a data structure for eg. For a stack, push(), pop() are ADT's

1. A method for achieving abstraction for data structures and algorithms
2. ADT = model + operations
3. Describes what each operation does, but not how it does it
4. An ADT is independent of its implementation

### Simple ADT's

1. Stack
2. Queue
3. Vector
4. Lists
5. Sequences
6. Iterators

These are called linear data structures

## Stack

Stack ADT's

1. Push()
2. Pop()
3. Supporting methods
    a. Size() returns the number of objects in the stack
    b. isEmpty() returns boolean indicating if stack is empty
    c. Top() return value of top object on the stack
4. Pseudo code implementation of Stack in an Array : See PPT

### Applications

1. Procedure calls
2. Recursion

### Problem

Solve stock span problem

## Queue

Queue ADT's

1. Enqueue()
2. Dequeue()
3. Supporting methods
    a. Size()
    b. Isempty()
    c. Front()
4. Pseudo code implementation of Queue via Array, look at the PPT. works with two variables, f and r

### Problem

Implement a queue using two stacks

## Singly Linked List

SLL ADT's

1. Size()
2. isEmpty()
3. First, last, after
4. replaceElement, swapElements
5. insertFirst, insertLast
6. insertAfter
7. insertBefore
8. Remove

## Doubly Linked List

Single linked list with pointer to traverse in reverse direction

Same ADT's as Singly Linked List

## Position ADT

We abstract the Linked List as just a list with the positions of elements

Method for ADT

1. Element() - returns the element stored at this position

## List ADT

Using position ADT, we can build a List ADT with the following methods

1. L.first() - Return the position of the first element of S
2. L.last() - Return the position of the last element of S
3. L.isFirst(p) - Return a bool indicating whether given position is the first one in the list
4. L.before(p) - Return the position of the element of S preceding the one at position p
5. L.after(p), L.size(), L.isEmpty()
6. L.replace(p, e) - Replace the element at position p with e, returning the element formerly at position p
7. L.swap(p, q) - Swap the elements stored at position p and q
8. L.add_First ( e ) - Insert a new element e into S as the first element
9. L.add_last (p) - Insert a new element e into S as the last element
10. L.add_before(p, e) - Insert a new element e into S before position p in S
11. L.add_after(p, e) - Insert a new element e into S after position p in S

## Vector ADT

A Vector stores a list of elements:

1. Access is via rank/index
2. elementAtRank( r )
3. replaceAtRank(r, o)
4. insertAtRank(r, o)
5. removeAtRank(r )
6. Size(), isEmpty()

Time complexity of Vector ADT's - See PPT

## Trees

Applications

1. Org charts
2. File Systems
3. Compiler Design/Text Processing
4. Searching Algorithms
5. Evaluating a mathematical expression

### Terminology

1. Root
2. Internal node : Node with at least one child
3. External node : a.k.a leaf (node without children)
4. Ancestors of a node: parent, grandparent, great-grandparent etc
5. Depth of a node: Number of ancestors excluding the node itself
6. Height - Number of descendants along the longest path
7. Descendant of a node : Child, grand-child etc
8. Degree of a node : Number of its children

### Tree ADT

1. Size() - Returns number of nodes in the tree
2. isEmpty()
3. Iterator elements()
4. Iterator Positions()
5. Position root()
6. Position parent(p) Return the parent of node p
7. Children(p) - Return the children of node v
8. isInternal()
9. isExternal()
10. isRoot()
11. swapElements(v, w)
12. replaceElement(v, e)

### Tree Traversals

1. Inorder (left, root, right)
2. Preorder (root, left, right) - Root of the tree is first
3. Postorder (left, right, root) - Root of the tree is last

### Problems

1. Solve questions on Tree Traversal
2. Find preorder traversal using in-order and post-order traversal and construct tree
3. We can construct full binary tree from pre-order and post-order traversals. Discuss with an example

## Binary Tree

**Full vs Complete**

A full binary tree is a tree in which every node other than the leaves has two children

In complete binary tree, all levels except possibly the last are completely filled and nodes are as left as possible

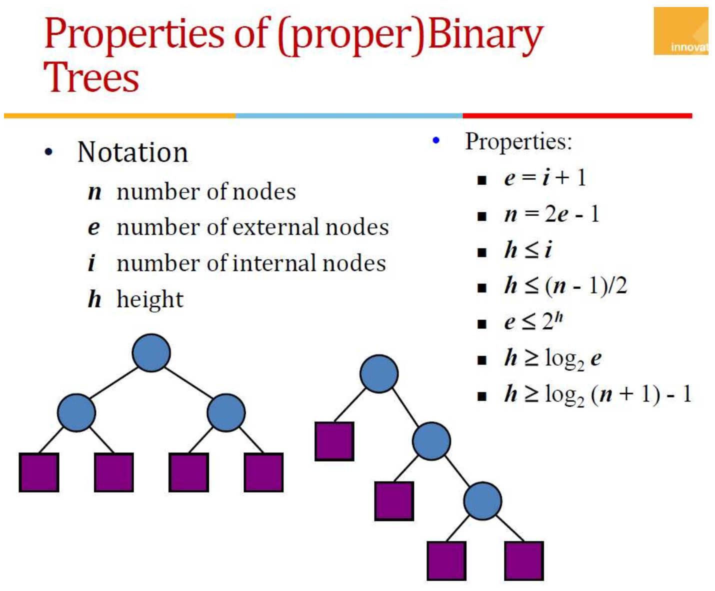
Perfect Binary Tree - Full + Complete + Symmetry

1. Each internal node has two children
2. A binary tree has at most two children
3. Children of a node are an ordered pair

Application :

1. Arithmetic expression tree
2. Decision Tree

**Properties of a proper binary tree**



## Properties of (proper)Binary Trees

- Notation
  - $n$ number of nodes
  - $e$ number of external nodes
  - $i$ number of internal nodes
  - $h$ height
- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \le i$
  - $h \le (n - 1)/2$
  - $e \le 2^h$
  - $h \ge \log_2 e$
  - $h \ge \log_2 (n + 1) - 1$

## Vector based representation of Binary tree

Nodes are ordered as root - I, leftchild - 2i, rightchild - 2i + 1

- Operation Time

| | |
|---|---|
| – positions, elements | O(n) |
| – swap Elements, replaceElement | O( 1 ) |
| – root, parent, children | O( 1 ) |
| – leftChild, rightChild, sibling | O ( 1 ) |
| – islnternal , isExternal, isRoot | O( 1 ) |

Methods

– Fast and Simple

– Space inefficient if the height of the tree is large

## Linked List based representation of Binary tree

- Operation Time
- Size()
- isEmpty()
- swapElements(v,w)
- replaceElement(v,e)

O(1)

- Positions()
- Elements()

O(n)

## Applications

1. Data base indexing
2. Binary Space Partitioning in video games
3. Path finding algorithms
4. Huffman coding
5. Heaps

6. Set and Map in C++
7. Syntax tree

## Heaps

1. Heap is a binary tree that stores collection of keys at it's nodes and that satisfies two additional properties
    a. A relational property defined in terms of the way keys are stored in T and
    b. A structural property defined in terms of T itself
2. A total order relation on the keys is given , for example by a comparator

Heap order property (relational property)

1. Min heap and max heap

A heap is always a complete binary tree i.e. other than leaf nodes, all nodes are fully filled and each nodes are as left as possible

### Methods

1. **Insertion into heap**: Put into the last node and restore heap order property (upheap)
2. **RemoveMin from heap (min heap)**: Swap root with last node in the heap and then restore heap order property

### Problems

1. Find possible last deleted nodes from heap
2. Write an algorithm to find the kth largest element of an array using
    a. Min Heap
    b. Max heap

**Heapify operation** : Check the children and apply heap order property, swap elements as required. It can be built top-down or bottom-up. Complexity is O(n), since leaf nodes do not require any comparisons or swaps and number of comparisons and swaps reduces logarithmically as we go up the tree

## Priority Queue ADT

Methods

1. insertItem (k, o)
2. removeMin()
3. minKey()
4. minElement()
5. Size(), isEmpty()

Applications

1. Standby flyers
2. Auctions
3. Stock Market

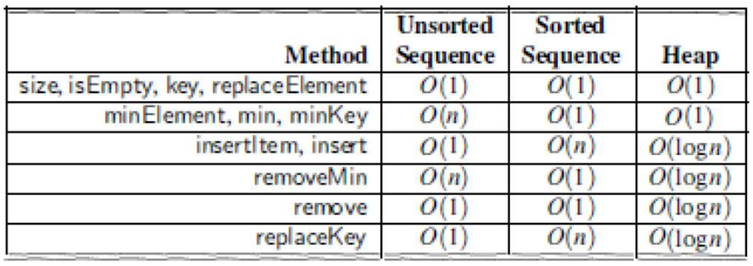
## Heap Sort

From an unsorted array, create a sorted array : Run Heapify operation and build a heap, now remove the root node (by swapping) and then running upheap() operation. Repeat until a sorted array has been created.

### Problems

1. Given sample array : Show how to run max-heapify and build max-heap
2. Related problem to above
3. Explain why runtime of heap-sort algorithm is O (n log n)
4. Implement priority queue implementation using an unordered sequence

**Comparison of different priority queue implementations**



| Method | Unsorted Sequence | Sorted Sequence | Heap |
|---|---|---|---|
| size, isEmpty, key, replaceElement | $O(1)$ | $O(1)$ | $O(1)$ |
| minElement, min, minKey | $O(n)$ | $O(1)$ | $O(1)$ |
| insertItem, insert | $O(1)$ | $O(n)$ | $O(\log n)$ |
| removeMin | $O(n)$ | $O(1)$ | $O(\log n)$ |
| remove | $O(1)$ | $O(1)$ | $O(\log n)$ |
| replaceKey | $O(1)$ | $O(n)$ | $O(\log n)$ |

## Dictionary ADT

Methods
1. findElement(k) - Find element with key value K
2. insertItem(k, e) - Insert element e with key k
3. removeElement(k)
4. Size(), isEmpty
5. Keys(), elements() - Iterators

Special element (NO_SUCH_KEY) is known as a sentinel

### Dictionary implementation using Hash Tables
A hash table consists of
- Array (called table) of size N - (Bucket Array)
- Hash function h

Collision : When two keys hash to the same bucket

Hash code map -- h1 --> maps keys to integers. For example if the keys are strings then some function which involves ASCII or Unicode values

Compression map -- h2 --> maps the hash code to an integer within the bucket array

h(x) = h2(h1(x))

Goal of hash function is to disperse the keys in a random way

## Collision Handling

### Separate Chaining
When collisions are handled as a kind of linked list mapped to the cell where collision happens

Load factor = n/N where n is the number of entries in the bucket array and N is the total capacity

### Open Addressing
When collisions are stored in the same bucket, and not on a separate list, it's called Open Addressing.

There are multiple variants within this
1. **Linear Probing :** When each cell is probed after a collision and first empty cell is used to store the value. Disadvantage is that collided entries tend to bunch together. h(k) = (k)%N , h1(k) = (k+1)%N
2. **Quadratic probing** : h(k) = (k % N), h1(k) = (k + I + I2) % N, h2(k) = (k + I + i4) % N
3. **Double Hashing** : Choose some other hash function which re-hashes the key. A common choice is h2(k) = q - k mod q

## Ordered Dictionary
Dictionary with an order relation between the keys in the dictionary

Operations
- FindElement(k)
- insertItem(k, e)
- removeElement(k)
- closestKeyBefore(k)

- closestElemBefore(k)
- closestElemAfter(k)
- First() return entry with smallest key
- Last() return entry with largest key

Application:
1. Flight databases
2. Any database which allows query based on a criteria

**Implementation using Sorted Array**

**Performance**:
1. findElement takes O (log n) using binary search
2. insertItem takes O(n) time since we need to shift items
3. removeElement takes O(n) time

## Problems
Solve problems on ordered dictionary from internet

## Binary Search Tree
A binary search tree is a binary tree storing keys (or key-element pairs) at its internal nodes and satisfying the following property

Let u, v, and w be three nodes such that u is in theleft subtree of v and w is in the right subtree of v.
We have key(u) <= key(v) <= key(w)

In-order traversal of the BST visits the keys in increasing order

Successor : Smallest number which is larger than the key
Predecessor : Largest number smaller than the key

If X has two children then its in-order predecessor is the maximum value in its left subtree and its in-order successor the minimum value in its right subtree

**Finding predecessor**
1. Left subtree's right most child
2. If left subtree does not exist then predecessor is one of the ancestors. Travel up until you see a node which is right child of it's parent. The parent of such a node is the predecessor

**Finding Successor**
1. If right subtree is not null then successor lies in right subtree. Get right subtrees leftmost child
2. If right subtree is null, then successor is one of the ancestors. Travel up using the parent pointer until you see a node which is the left child of it's parent. The parent of such a node is the successor

**Deleting a node**
1. Deleting a leaf node
2. Deleting a node with 1 child : Remove the node and replace it with its child
3. Deleting a node with 2 children : Replace the node with it's successor or predecessor

A balanced tree is a tree where every leaf is "not more than a certain distance" away from the root than any other leaf.

Add a rule to the BST definition that will maintain a logarithmic height for the tree

**Height balance property**
For every internal node v of T, the heights of the children of v can differ by at most 1

**Leftsize** - number of nodes in its left subtree

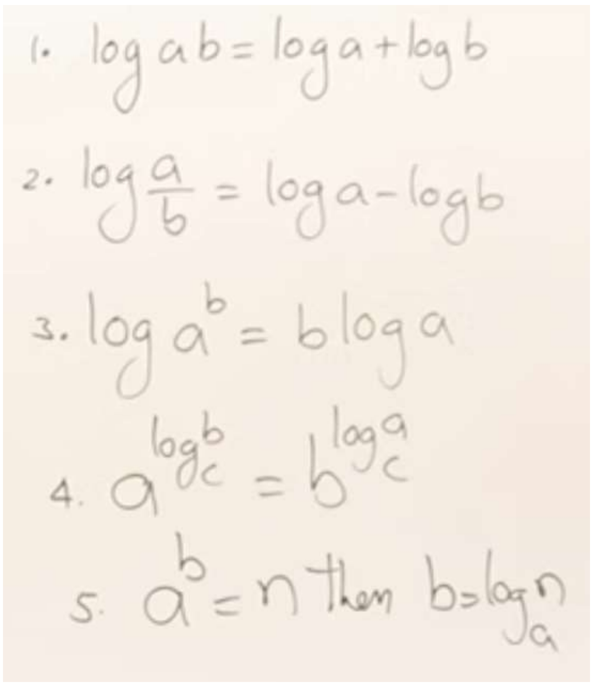
**Rank** : Rank of an element is its position in in-order traversal, also the value of leftsize

## Problems

1. Find the kth smallest element in a BST
2. Find least common ancestor and time complexity of finding it

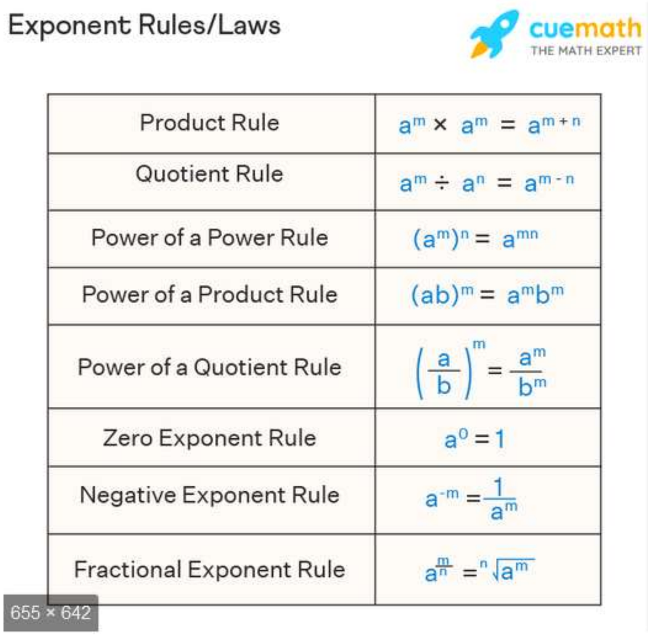
## Additional Data

### Log formulas for comparison



1. $\log ab = \log a + \log b$
2. $\log \frac{a}{b} = \log a - \log b$
3. $\log a^b = b \log a$
4. $a^{\log_c b} = b^{\log_c a}$
5. $a^b = n$ then $b = \log_a n$

### Exponent Formulas for comparison

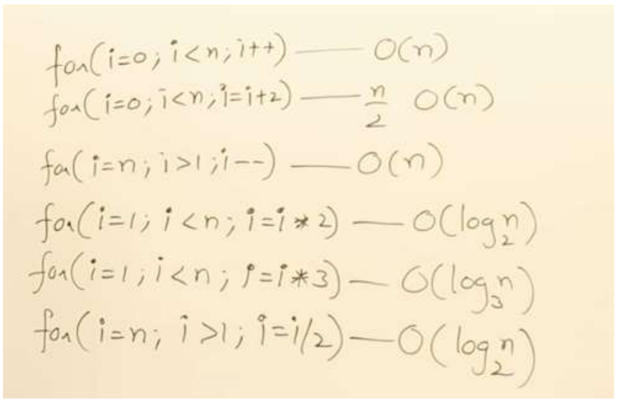


Exponent Rules/Laws

| | |
|---|---|
| Product Rule | $a^m \times a^m = a^{m+n}$ |
| Quotient Rule | $a^m \div a^n = a^{m-n}$ |
| Power of a Power Rule | $(a^m)^n = a^{mn}$ |
| Power of a Product Rule | $(ab)^m = a^m b^m$ |
| Power of a Quotient Rule | $\left(\frac{a}{b}\right)^m = \frac{a^m}{b^m}$ |
| Zero Exponent Rule | $a^0 = 1$ |
| Negative Exponent Rule | $a^{-m} = \frac{1}{a^m}$ |
| Fractional Exponent Rule | $a^{\frac{m}{n}} = \sqrt[n]{a^m}$ |

cuemath
THE MATH EXPERT
655 × 642

### Time Complexity of common loops



for(i=0; i<n; i++) — $O(n)$

for(i=0; i<n; i=i+2) — $\frac{n}{2}$ $O(n)$

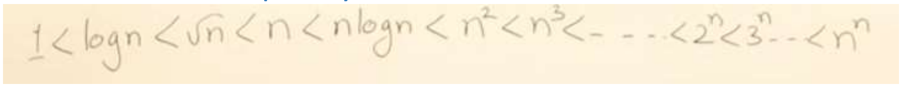
for(i=n; i>1; i--) — $O(n)$

for(i=1; i<n; i=i*2) — $O(\log_2 n)$

for(i=1; i<n; i=i*3) — $O(\log_3 n)$

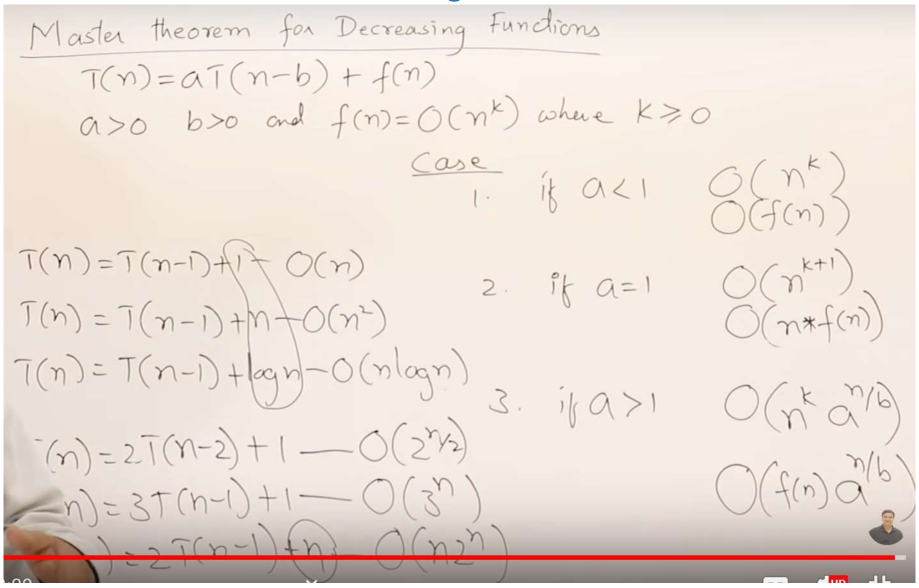
for(i=n; i>1; i=i/2) — $O(\log_2 n)$

## Order of time complexity



$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots < 2^n < 3^n \ldots < n^n$

## Masters Theorem for Decreasing Functions



Master theorem for Decreasing Functions

$T(n) = aT(n-b) + f(n)$

$a>0$ $b>0$ and $f(n) = O(n^k)$ where $k \geq 0$

Case

1. if $a<1$ $O(n^k)$ $O(f(n))$
2. if $a=1$ $O(n^{k+1})$ $O(n*f(n))$
3. if $a>1$ $O(n^k a^{n/b})$ $O(f(n)a^{n/b})$

$T(n) = T(n-1) + 1$ — $O(n)$

$T(n) = T(n-1) + n$ — $O(n^2)$

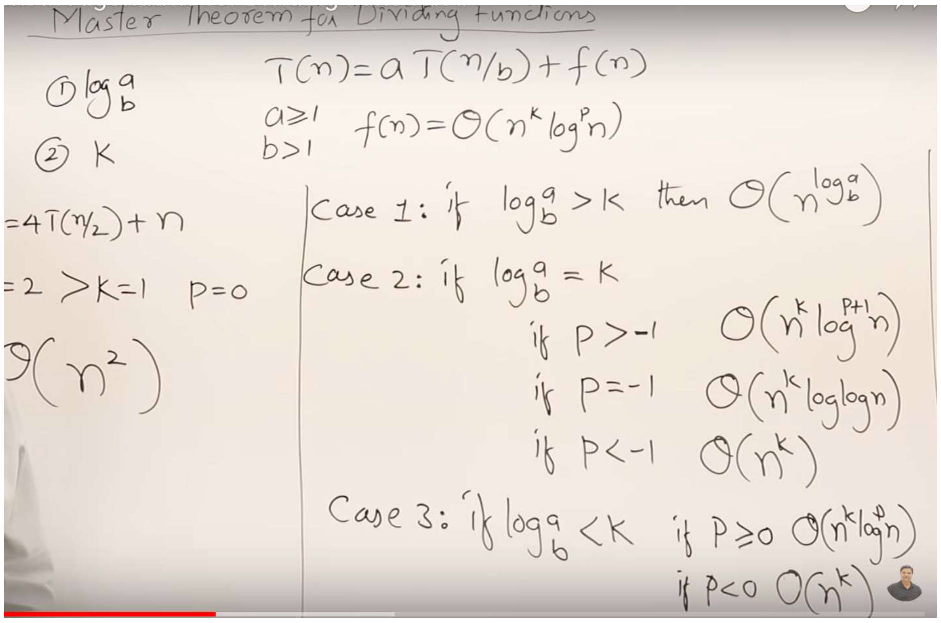
$T(n) = T(n-1) + \log n$ — $O(n\log n)$

$(n) = 2T(n-2) + 1$ — $O(2^{n/2})$

$n) = 3T(n-1) + 1$ — $O(3^n)$

$) = 2T(n-1) + n$ — $O(n2^n)$

## Masters theorem for Dividing functions



Master Theorem for Dividing Functions

① $\log_b a$

② $K$

$T(n) = aT(n/b) + f(n)$

$a \geq 1$
$b > 1$ $f(n) = \Theta(n^k \log^p n)$

Case 1: if $\log_b a > k$ then $\Theta(n^{\log_b a})$

Case 2: if $\log_b a = k$

if $p > -1$ $\Theta(n^k \log^{p+1} n)$

if $p = -1$ $\Theta(n^k \log\log n)$

if $p < -1$ $\Theta(n^k)$

Case 3: if $\log_b a < k$ if $p \geq 0$ $\Theta(n^k \log^p n)$

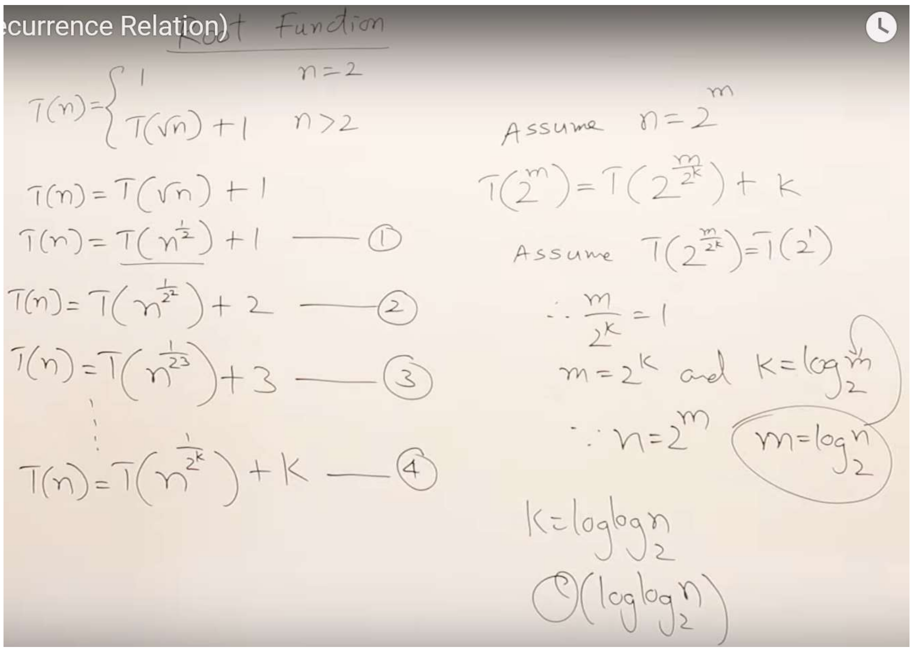
if $p < 0$ $O(n^k)$

$=4T(n/2) + n$

$=2 > k=1$ $p=0$

$\Theta(n^2)$

## Solution for Root



ecurrence Relation)

Root Function

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

$T(n) = T(\sqrt{n}) + 1$

$T(n) = \underline{T(n^{\frac{1}{2}})} + 1$ —— ①

$T(n) = T(n^{\frac{1}{2^2}}) + 2$ —— ②

$T(n) = T(n^{\frac{1}{2^3}}) + 3$ —— ③

$\vdots$

$T(n) = T(n^{\frac{1}{2^k}}) + K$ —— ④

Assume $n = 2^m$

$T(2^m) = T(2^{\frac{m}{2^k}}) + K$

Assume $T(2^{\frac{m}{2^k}}) = T(2^1)$

$\therefore \frac{m}{2^k} = 1$

$m = 2^k$ and $K = \log_2 m$

$\therefore n = 2^m$ $\quad \boxed{m = \log_2 n}$

$K = \log \log_2 n$

$O(\log \log_2 n)$