

CSE 221 Algorithms: Ch3

Instructor: Saraf Anika

CSE, SoSET, EDU

Spring 2022

Divide & Conquer

- A **divide and conquer algorithm** is a strategy of solving a large problem by
 - breaking the problem into smaller sub-problems ✓
 - solving the sub-problems, and ✓
 - combining them to get the desired output. ✓
- To use the divide and conquer algorithm, **recursion** is used.
- Here are the steps involved:
 - **Divide**: Divide the given problem into sub-problems using **recursion**.
 - **Conquer**: Solve the smaller sub-problems recursively. If the sub-problem is small enough, then solve it directly.
 - **Combine**: Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.
- Application:
 - Binary Search
 - Merge Sort
 - Quick Sort
 - Strassen's Matrix multiplication
 - Karatsuba Algorithm

Divide & Conquer Cont.

- Time Complexity

$$T(n) = aT(n/b) + f(n), \quad \text{where,}$$

n = size of input

a = number of subproblems in the recursion

n/b = size of each subproblem. All subproblems are assumed to have the same size.

$f(n)$ = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

- **Divide and Conquer Vs Dynamic approach**

The divide and conquer approach divides a problem into smaller subproblems; these subproblems are further solved recursively. The result of each subproblem is **not stored** for future reference, whereas, in a dynamic approach, the result of each subproblem is stored for future reference.

Maximum Sum Subarray Problem(MSS) ✓

- Subarray: An ordered contiguous part of an array. May contain only 1 element to n-number of elements. It has to be contiguous.
- Two approaches to solve this problem:
 - DAC
 - DP(Kadane's Algorithm)
- What if all elements of the array are negative? (**Ans:** It is guaranteed that at least one element in the array would be non-negative)
- Is it allowed to modify the array? (**Ans:** Sure, but a subarray is an ordered contiguous segment of the original array)

DAC Approach

- Divide the array into two equal parts : create 2 subarrays: element no. equal: Left Right
- Recursively calculate the maximum sum for *left* and *right* subarray
- To find cross-sum:-
 - Iterate from **mid** to the **starting part** of the left subarray and at every point, check the maximum possible sum till that point and store in the parameter **lsum**.
 - Iterate from **mid+1** to the **ending point** of right subarray and at every point, check the maximum possible sum till that point and store in the parameter **rsum**.
 - Add **lsum** and **rsum** to get the **cross-sum**
- Return the maximum among (left, right, cross-sum)

DAC Approach: Pseudocode

Input: A[8] = { -2, -5, 6, -2, -3, 1, 5, -6 }
Output:
MSS = 7
Max Subarray = { 6, -2, -3, 1, 5 }

```
● // The original values would be low = 0 and high = n-1
● int maxSubarraySum (int A[], int low, int high) {
    ○ if (low == high)
        ■ return A[low]
    ○ else {
        ■ int mid = low + (high - low)/2
        ■ int left sum = maxSubarraySum (A, low, mid)
        ■ int right sum = maxSubarraySum (A, mid+1, high)
        ■ int crossing Sum = maxCrossingSum (A, low, mid, high)
        ■ return max (left_sum, right_sum, crossing_sum)
    }
}
● int maxCrossingSum (int A[], int l, int mid, int r) {
    ○ int sum = 0
    ○ int lsum = INT MIN
    ○ for (i = mid to l) {
        ■ sum = sum + A[i]
        ■ If (sum > lsum)
            ● lsum = sum
    }
    ○ sum = 0
    ○ int rsum = INT MIN
    ○ for (i = mid+1 to r) {
        ■ sum = sum + A[i]
        ■ If (sum > rsum)
            ● rsum = sum
    }
    ○ return (lsum + rsum)
}
● }
```

Recursion

left

Right

Cont.

- Time Complexity: `maxSubArraySum()` is a recursive method and time complexity can be expressed as following recurrence relation.

$$T(n) = 2T(n/2) + \Theta(n)$$

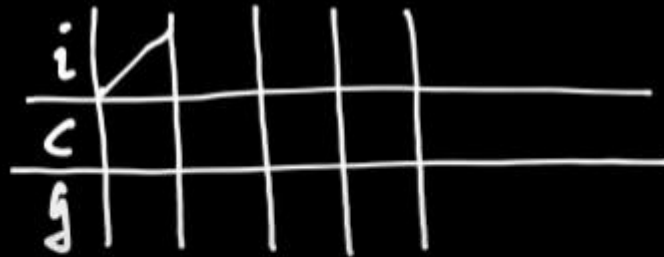
- We can simply form that the height of the **maxSubarraySum** tree is $\log n$ as it is a binary tree and each phase will break down n number of elements. so, the time complexity is $O(n \log n)$.

DP(Kadane's Algorithm)

- Max_current = c
- Max_global = g

Kadane's Algorithm (Code)

A = [-2, 3, 2, -1]



```
def kadane(A):  
    max_current = max_global = A[0]  
    for i from 1 to length(A) - 1:  
        max_current = max(A[i],  
                           max_current + A[i])  
        if max_current > max_global  
            max_global = max_current  
    return max_global
```


DP(Kadane's Algorithm) Cont.

- Time Complexity: $O(n)$ time. Therefore the Kadane's algorithm is better than the Divide and Conquer approach.

Kadane's Algorithm (Code)

$A = [-2, 3, 2, -1]$

Handwritten annotations for the array A :

- Index 2 is circled in green.
- Indices 1 and 2 are circled in orange.
- Indices 1 and 2 are boxed in orange, with the value 2 written above the box.
- Indices 1 and 2 are boxed in orange, with the value 3 written below the box.

i	1	2	3
c	-2	3	5
g	-2	3	5

```
def kadane(A):  
    max_current = max_global = A[0]  
    for i from 1 to length(A) - 1:  
        max_current = max(A[i],  
                           max_current + A[i])  
        if max_current > max_global  
            max_global = max_current  
    return max_global
```

Merge Sort

- Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.
- The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e. $p == r$.
- After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.

Merge Sort Cont.

- Pseudocode ✓

```
MergeSort(arr[], l, r) {
```

```
    if (r > l)
```

```
        1. Find the middle point to divide the array into two halves:
```

```
            middle m = l + (r-l)/2
```

```
        2. Call mergeSort for first half:
```

```
            Call mergeSort(arr, l, m) →
```

```
        3. Call mergeSort for second half:
```

```
            Call mergeSort(arr, m+1, r) →
```

```
        4. Merge the two halves sorted in step 2 and 3:
```

```
            Call merge(arr, l, m, r) ↘
```

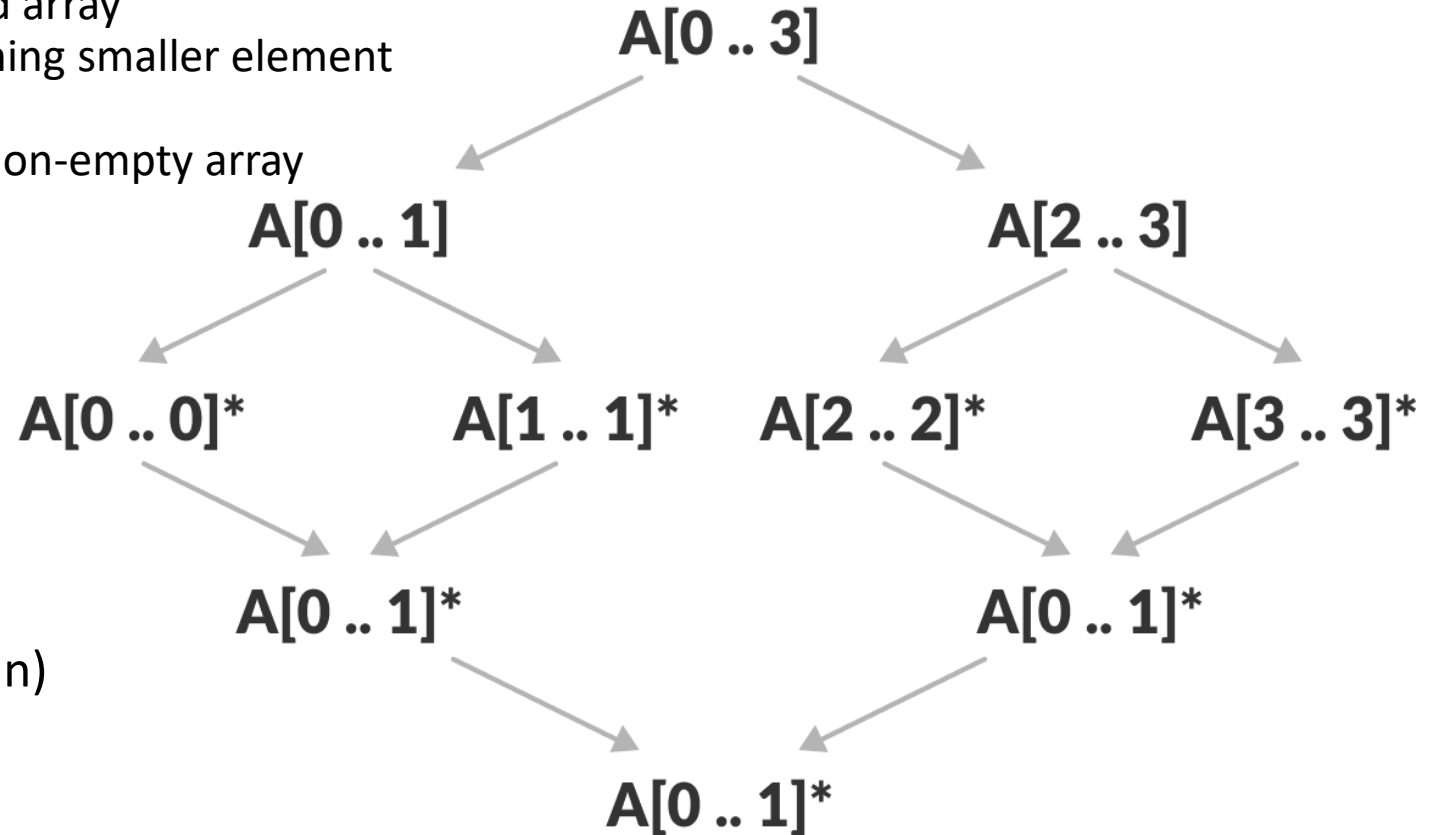
```
    }
```

Merge Sort Cont.

- Have we reached the end of any of the arrays?
 - No:
 - Compare current elements of both arrays
 - Copy smaller element into sorted array
 - Move pointer of element containing smaller element
 - Yes:
 - Copy all remaining elements of non-empty array

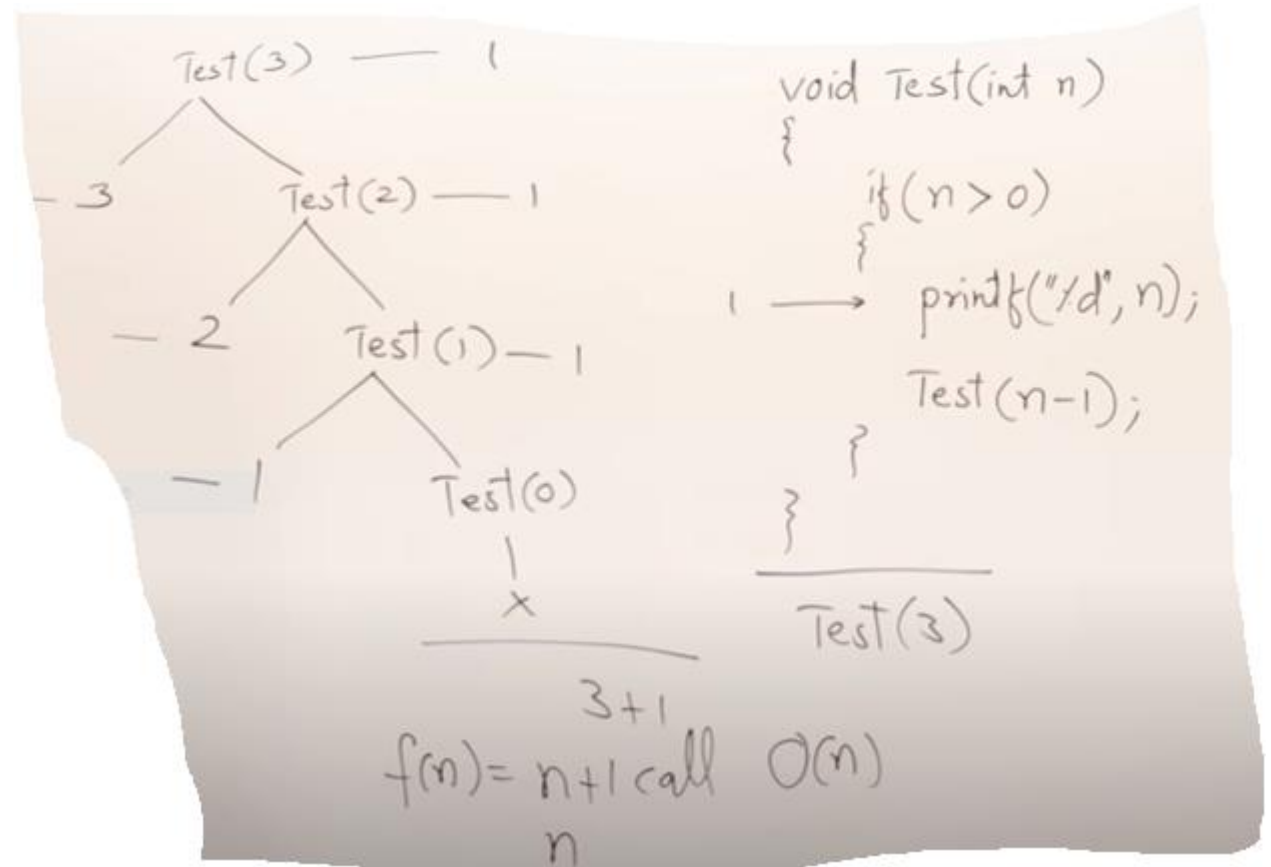
- **Time Complexity**

- Best Case Complexity: $O(n \cdot \log n)$
- Worst Case Complexity: $O(n \cdot \log n)$
- Average Case Complexity: $O(n \cdot \log n)$

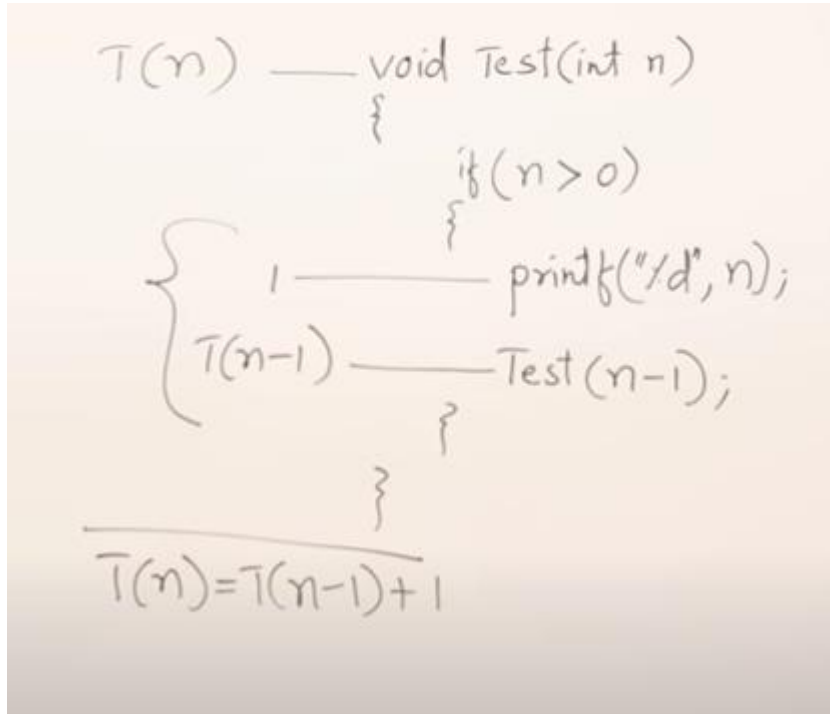


Recurrence Relation Decreasing Functions

- $T(n) = T(n-1) + 1$
- A simple C program is given
- Breakdown of recursive function, Test(n) is shown in the figure
- Test(n) function is called n times, so Time Complexity is $O(n)$



Recurrence Relation Dividing Functions Cont.



Handwritten code for a recursive function `Test` and its recurrence relation:

```
T(n) — void Test(int n)
      {
          if (n > 0)
          {
              printf("%d", n);
              T(n-1);
          }
      }
```

Below the code, the recurrence relation is written:

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

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

$$T(n) = T(n-1) + 1, \quad T(n-1) = T(n-1) + 1$$

Substitute $T(n-1)$, $n = T(n-2)$

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

$$T(n) = [T(n-2-1) + 2] + 1 = T(n-3) + 3$$

⋮
⋮
⋮
⋮

Continue k times

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

Assume,

$$n-k = 0, \quad n=k$$

Substituting n,

$$T(n) = T(n-n) + n = T(0) + n = 1 + n$$

Time Complexity $O(n)$

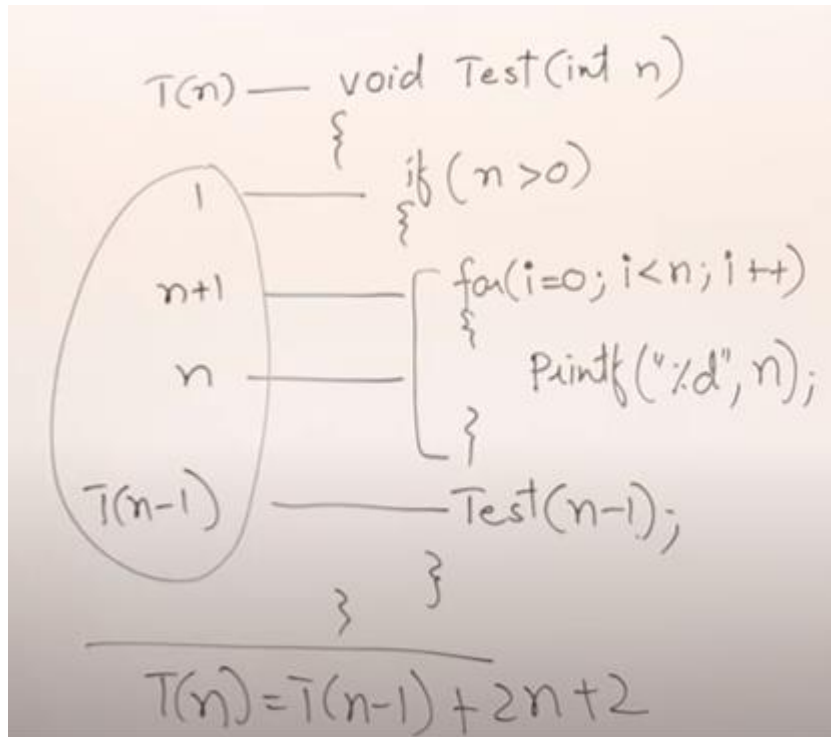
Recurrence Relation Decreasing Functions

Cont.

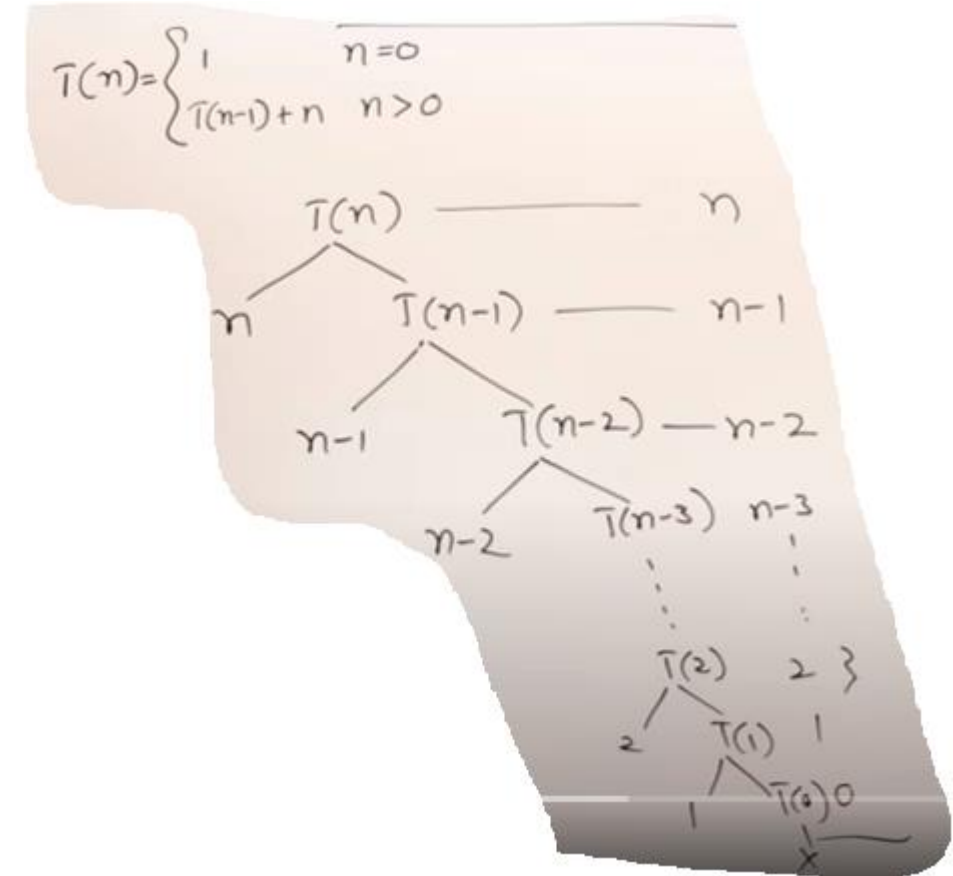
- $T(n) = T(n-1) + n$

$$0+1+2+\dots+n-1+n = n(n+1)/2$$

$$T(n) = n(n+1)/2 = O(n^2)$$



$2n+2$ can be considered as n to avoid complication



Recurrence Tree

Recurrence Relation Decreasing Functions Cont.

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

$$T(n) = T(n-1) + n \dots \dots \dots \mathbf{(1)}$$

Substitute $T(n-1)$,

$$T(n) = [T(n-1-1) + n-1] + n = T(n-2) + (n-1) + n \dots \dots \dots \mathbf{(2)}$$

$$T(n) = [T(n-2-1) + n-2] + n-1 + n = T(n-3) + (n-2) + (n-1) + n \dots \dots \dots \mathbf{(3)}$$

.

.

Continue k times, substitute constant,

$$T(n) = T(n-k) + (n-(k-1)) + (n-(k-2)) + \dots \dots \dots + (n-1) + n \dots \dots \dots \mathbf{(4)}$$

Assume,

$$n-k = 0, n=k$$

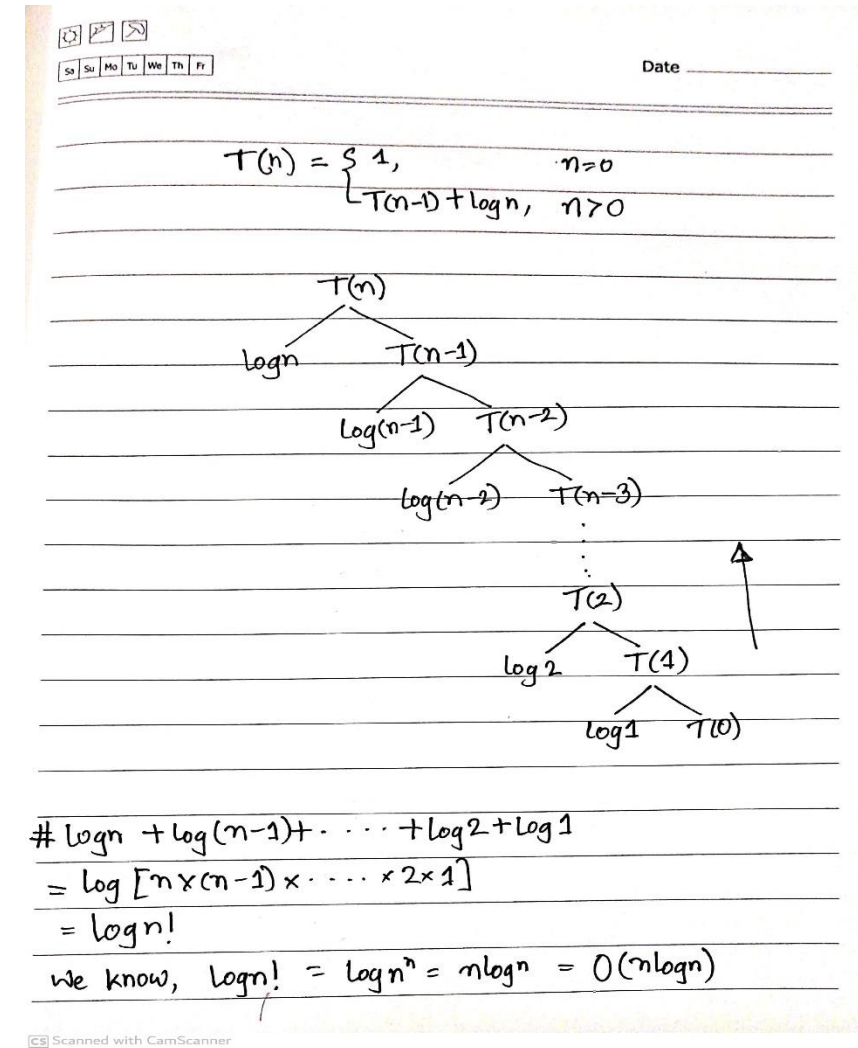
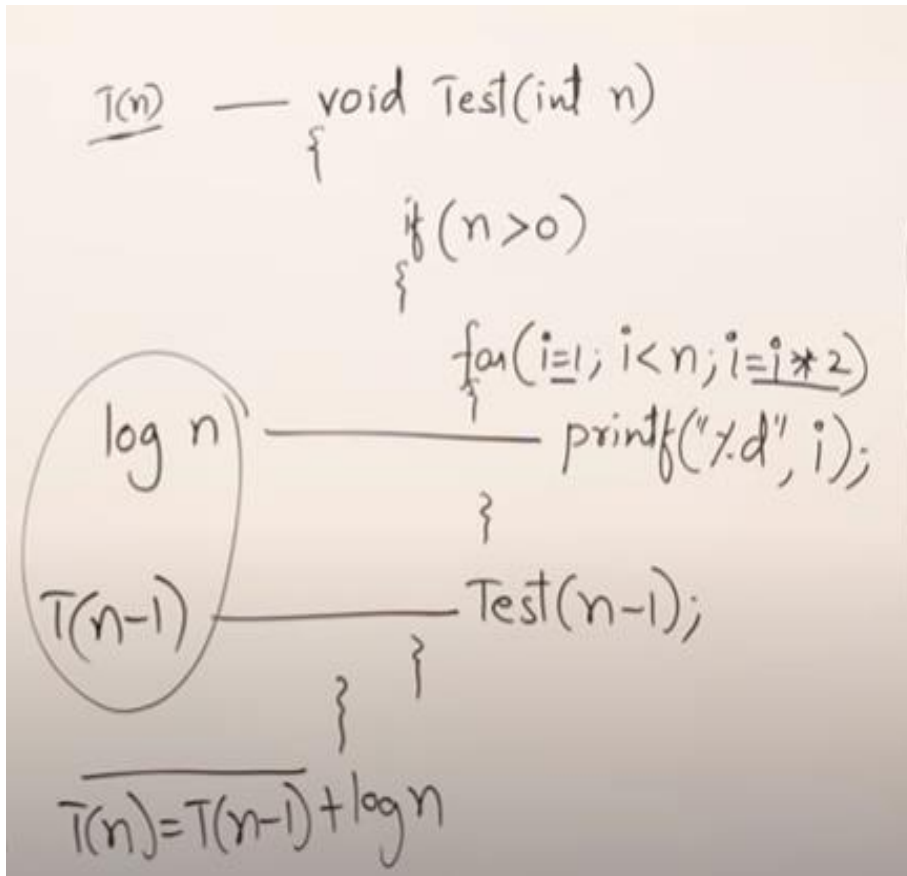
Substitute value of n in **(4)**

$$\begin{aligned} T(n) &= T(n-n) + (n-(n-1)) + (n-(n-2)) + \dots \dots \dots + (n-1) + n \\ &= T(0) + 1 + 2 + \dots \dots \dots + (n-1) + n = 1 + n(n-1)/2 = n^2 \end{aligned}$$

Time Complexity $O(n^2)$

Recurrence Relation Decreasing Functions Cont.

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



Recurrence Relation Decreasing Functions Cont.

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

$$T(n) = T(n-1) + \log n \dots \dots \dots (1)$$

Substitute $T(n-1)$,

$$T(n) = T(n-2) + \log(n-1) + \log n \dots \dots \dots (2)$$

$$T(n) = T(n-3) + \log(n-2) + \log(n-1) + \log n \dots \dots \dots (3)$$

.

.

Continue k times, substitute constant,

$$T(n) = T(n-k) + \log 1 + \log 2 + \dots + \log(n-1) + \log n \dots \dots \dots (4)$$

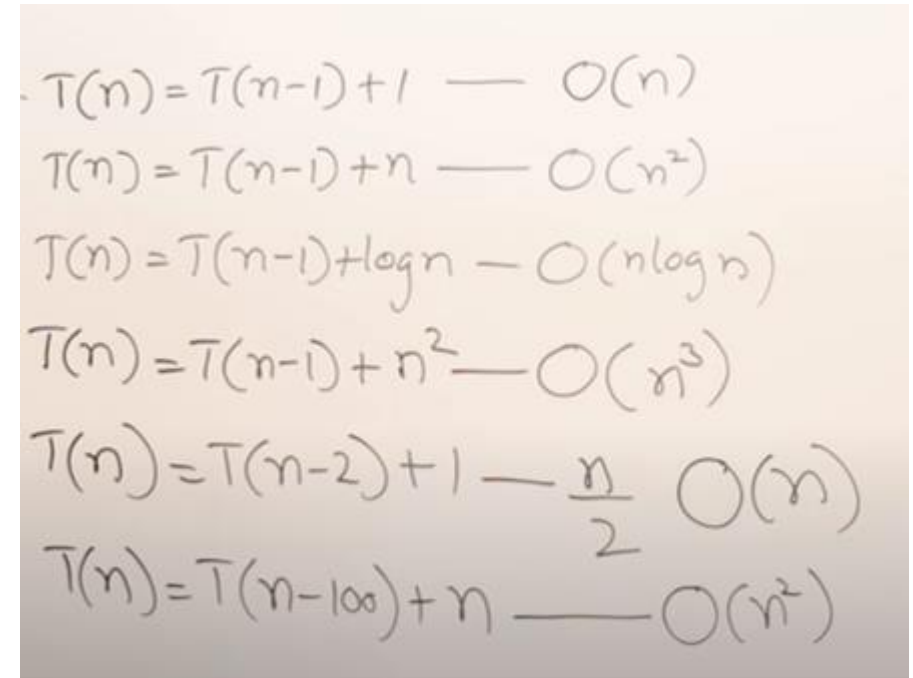
Assume,

$$n-k = 0, n=k$$

Substitute value of n in (4)

$$T(n) = T(0) + \log n! = 1 + \log n! \text{ [Since } n! = 1 \times 2 \times 3 \times \dots \times n, \log(n!) = \log(1) + \log(2) + \log(3) + \dots + \log(n)\text{]}$$

Upper bound for $n! = O(n^n)$, so Time Complexity $O(n \log n)$

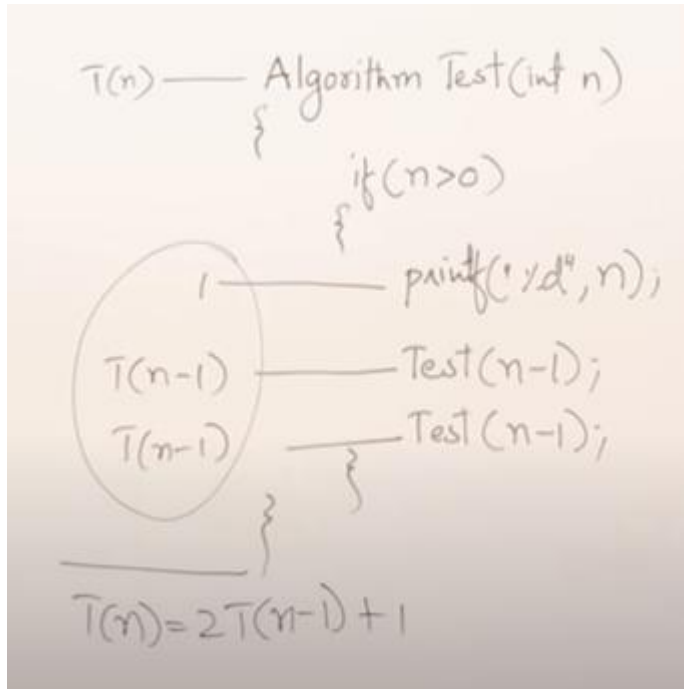


Handwritten notes showing recurrence relations and their time complexities:

- $T(n) = T(n-1) + 1 \rightarrow O(n)$
- $T(n) = T(n-1) + n \rightarrow O(n^2)$
- $T(n) = T(n-1) + \log n \rightarrow O(n \log n)$
- $T(n) = T(n-1) + n^2 \rightarrow O(n^3)$
- $T(n) = T(n-2) + 1 \rightarrow \frac{n}{2} \rightarrow O(n)$
- $T(n) = T(n-100) + n \rightarrow O(n^2)$

Recurrence Relation Decreasing Functions Cont.

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



$$T(n) = \begin{cases} 1 & n=0 \\ 2T(n-1) + 1 & n>0 \end{cases}$$
$$1 + 2 + 2^2 + 2^3 + \dots + 2^k = 2^{k+1} - 1$$
$$a + ar + ar^2 + ar^3 + \dots + ar^k = a \frac{(r^{k+1} - 1)}{r - 1}$$
$$a=1 \quad r=2 \quad = 1 \frac{(2^{k+1} - 1)}{2 - 1}$$
$$= 2^{k+1} - 1$$
$$n-k=0; n=k; 2^{n+1} - 1 \quad O(2^n)$$

Recurrence Relation Decreasing Functions Cont.

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

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

$$T(n) = 2T(n-1) + 1 \quad \text{--- (1)}$$

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

$$T(n) = 2^2 T(n-2) + 2 + 1 \quad \text{--- (2)}$$

$$= 2^2 [2T(n-3) + 1] + 2 + 1$$

$$T(n) = 2^3 T(n-3) + 2^2 + 2 + 1 \quad \text{--- (3)}$$

:

$$T(n) = 2^k T(n-k) + 2^{k-1} + 2^{k-2} + \dots + 2^2 + 2 + 1$$

Assume $n-k=0$
 $n=k$

$$= 2^n T(0) + \underline{1 + 2 + 2^2 + \dots + 2^{k-1}}$$

$$= 2^n \times 1 + 2^k - 1$$

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

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

$$O(2^n)$$

Recurrence Relation Decreasing Functions Cont.

- Masters Theorem Decreasing Function

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

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

- $T(n) = T(n-1) + 1 \dots O(n)$
- $T(n) = T(n-1) + n \dots O(n^2)$
- $T(n) = T(n-1) + \log n \dots O(n \log n)$
- $T(n) = 2T(n-1) + 1 \dots O(2^n)$
- $T(n) = 3T(n-1) + 1 \dots O(3^n)$
- $T(n) = 2T(n-1) + n \dots O(n2^n)$
- $T(n) = 2T(n-2) + 1 \dots O(2^{n/2})$

- Find out:

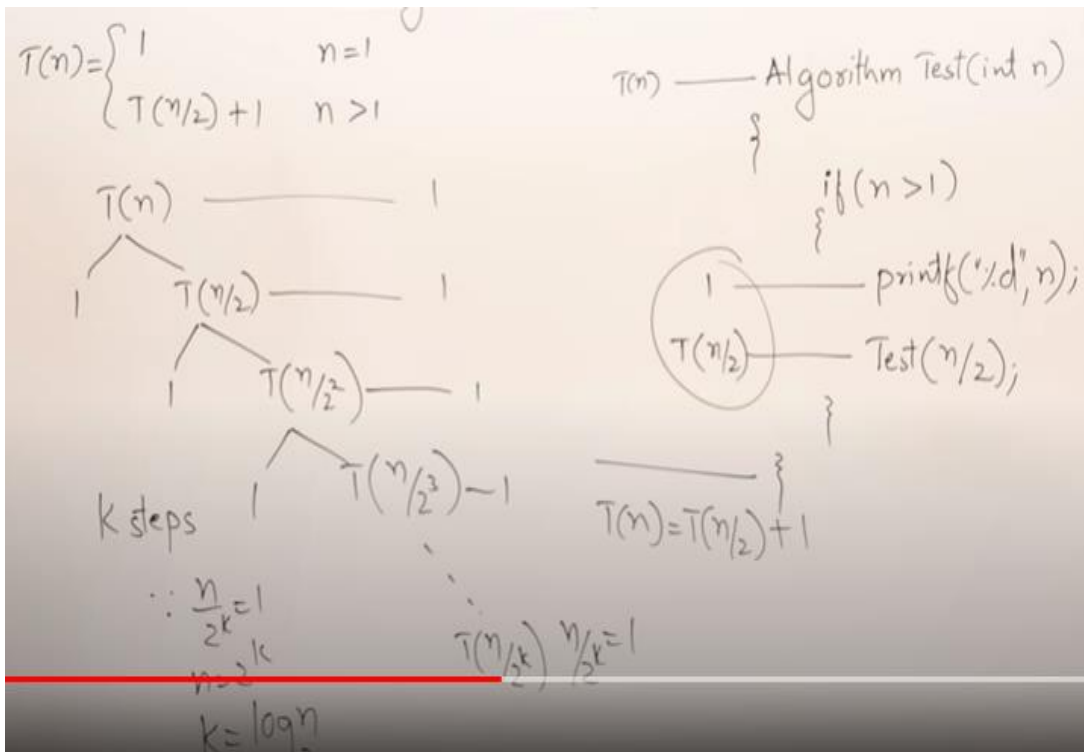
- $T(n) = T(n-1) + n^2 \dots ?$
- $T(n) = 2T(n-50) + n \dots ?$
- $T(n) = T(n-100) + \log n \dots ?$

- CASE:

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

Recurrence Relation Dividing Functions

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



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

$$T(n) = T(n/2^k) + k$$

Assume $\frac{n}{2^k} = 1$

$\therefore n = 2^k$ and $k = \log n$

$$T(n) = T(n/2) + 1 \quad \text{--- (1)}$$

$$T(n) = [T(n/2) + 1] + 1$$

$$T(n) = T(1) + \log n$$

$$T(n) = T(n/2) + 2 \quad \text{--- (2)}$$

$$T(n) = 1 + \log n$$

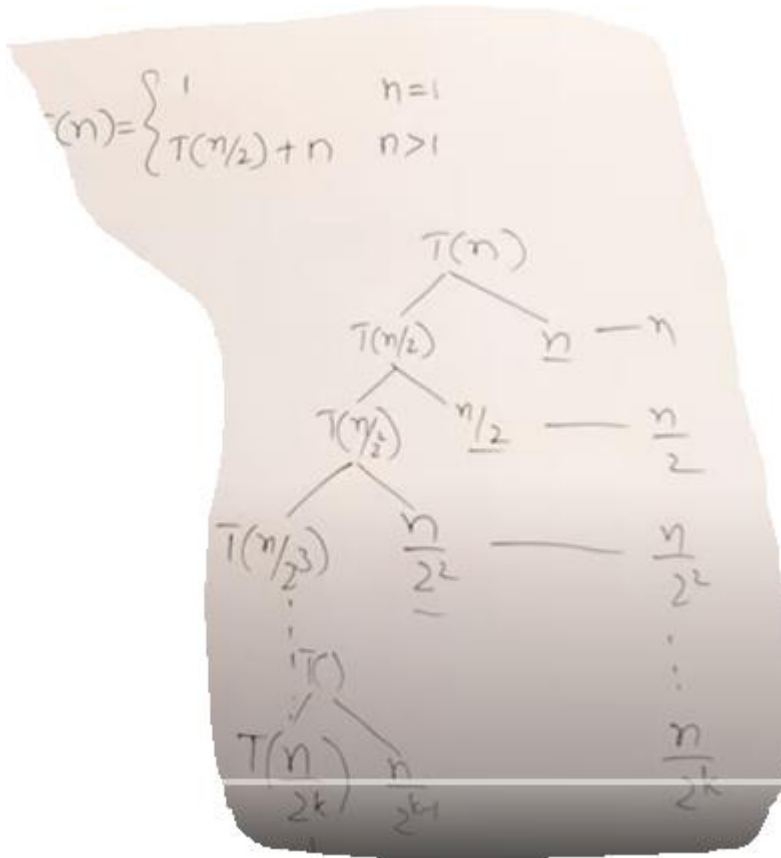
$$T(n) = T(n/2) + 3 \quad \text{--- (3)}$$

$$O(\log n)$$

$$T(n) = T(n/2) + k \quad \text{--- (4)}$$

Recurrence Relation Dividing Functions Cont.

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



Recurrence Relation: $T(n) = n + \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \dots + \frac{n}{2^k}$

$T(n) = n \left[1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \dots + \frac{1}{2^k} \right]$

$= n \left(\sum_{i=0}^k \frac{1}{2^i} \right) = 1$

$= n \times 1$

$T(n) = n$

$O(n)$

A diagram illustrating the geometric series $\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \dots$. It shows a circle divided into segments of decreasing size, representing the terms of the series. The first segment is labeled $\frac{1}{2}$, the second $\frac{1}{2^2}$, and the third $\frac{1}{2^3}$. The series is shown to converge to 1.

Recurrence Relation Dividing Functions Cont.

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

$n=1$
 $T(n)=T(n/2)+n \quad n>1$

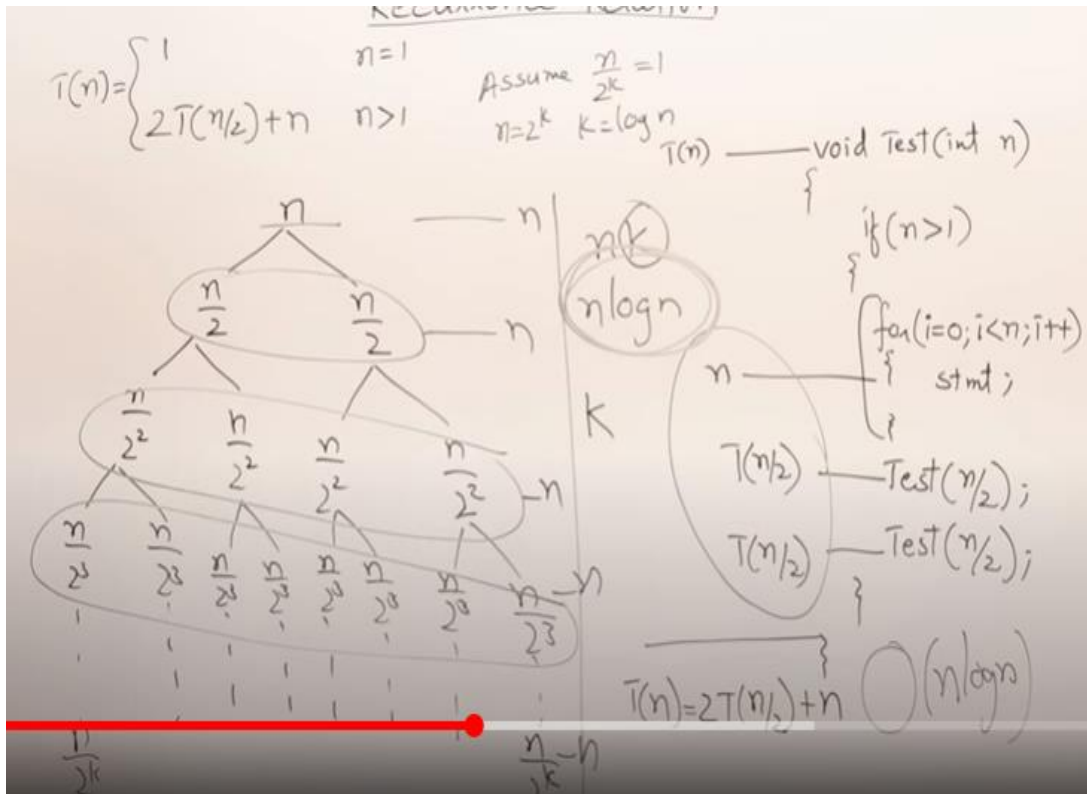
$T(n) = T(n/2) + n \quad \text{--- (1)}$
 $T(n) = \left[T(n/2) + \frac{n}{2} \right] + n$
 $T(n) = T(n/2) + \frac{n}{2} + n \quad \text{--- (2)}$
 $T(n) = T(n/2) + \frac{n}{2} + \frac{n}{2} + n$
 \vdots
 $T(n) = T(n/2^k) + \frac{n}{2^{k-1}} + \frac{n}{2^{k-2}} + \dots + \frac{n}{2} + n \quad \text{--- (3)}$

$T(n) = T(n/2^k) + \frac{n}{2^{k-1}} + \dots + \frac{n}{2} + n$
 Assume $\frac{n}{2^k} = 1$
 $\therefore n = 2^k$ and $k = \log n$

$T(n) = T(1) + n \left[\frac{1}{2^{k-1}} + \frac{1}{2^{k-2}} + \dots + \frac{1}{2} + 1 \right]$
 $T(n) = 1 + n[1+1]$
 $T(n) = 1 + 2n$
 $T(n) = O(n)$

Recurrence Relation Dividing Functions Cont.

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



Recurrence Relation Dividing Functions Cont.

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

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

$$T(n) = 2T(n/2) + n \quad \text{--- ①}$$

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

$$T(n) = 2 \left[2T(n/2^2) + \frac{n}{2} \right] + n$$

$$T(n) = 2^2 T(n/2^2) + n + n \quad \text{--- ②}$$

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

$$T(n) = 2^2 \left[2T(n/2^3) + \frac{n}{2^2} \right] + 2n$$

$$T(n) = 2^3 T(n/2^3) + 3n \quad \text{--- ③}$$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

Assume $T\left(\frac{n}{2^k}\right) = T(1)$

$$\therefore \frac{n}{2^k} = 1 \quad n = 2^k$$

$$k = \log n$$

$$② \quad T(n) = 2^k T(1) + kn$$

$$T(n) = n \times 1 + n \log n$$

$$O(n \log n)$$

Masters Theorem for Dividing Function

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

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

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

Case 2: if $\log_b a = k$

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

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

if $p < -1$ $O(n^k)$

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

Example of CASE 1:

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

$$a = 2$$

$$b = 2$$

$$\log_2 2 = 1 > k = 0$$

$$f(n) = O(1)$$

$$= O(n^0 \log^0 n)$$

$$k = 0, p = 0$$

Case 1: $O(n^1)$

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

$$\log_2 4 = 2 > k = 1, p = 0$$

$$O(n^2)$$

- Complexity of $T(n) = 8T(n/2) + n \dots ??$
- Complexity of $T(n) = 8T(n/2) + n^2 \dots ??$
- Complexity of $T(n) = 8T(n/2) + n \log n \dots ??$

Masters Theorem for Dividing Function Cont.

- Example of CASE 2:

$$T(n) = 4T(n/2) + n^2$$
$$\log_2^4 = 2 \quad k=2$$
$$O(n^2 \log n)$$

When $P > -1$

$$T(n) = 4T(n/2) + n^2 \log^2 n$$
$$\log_2^4 = 2 \quad k=2$$
$$O(n^2 \log^3 n)$$

$$T(n) = 2T(n/2) + \frac{n^1}{\log n}$$
$$\log_2^2 = 1 \quad k=1 \quad p=-1$$
$$O(n \log \log n)$$

When $P = -1$

$$T(n) = 2T(n/2) + n^1 \log^{-2} n$$
$$\log_2^2 = 1 \quad k=1 \quad p=-2$$
$$O(n)$$

When $P < -1$

Masters Theorem for Dividing Function Cont.

- Example of CASE 3:

$$T(n) = 2T(n/2) + n^2$$
$$\log_2^2 = 1 < k = 2$$
$$O(n^2)$$

When $P \geq 0$

$$T(n) = 2T(n/2) + n^2 \log^2 n$$
$$\log_2^2 = 1 < k = 2$$
$$O(n^2 \log^2 n)$$

$$T(n) = 4T(n/2) + n^3 / \log n$$

$$\log_2^4 = 2 < k = 3, P < 0$$

$$O(n^3)$$

When $P < 0$

Recurrence Relation for Root Function

```
T(n) — void Test(int n)
      {
        if (n > 2)
        {
          stmt;
          T(√n) — Test(√n);
        }
      }


---


T(n) = T(√n) + 1
```

$$T(n) = \begin{cases} 1 & n=2 \\ T(\sqrt{n}) + 1 & n>2 \end{cases}$$
$$T(n) = T(\sqrt{n}) + 1$$
$$T(n) = T(n^{\frac{1}{2}}) + 1 \quad \text{--- ①}$$
$$T(n) = T(n^{\frac{1}{2^2}}) + 2 \quad \text{--- ②}$$
$$T(n) = T(n^{\frac{1}{2^3}}) + 3 \quad \text{--- ③}$$
$$\vdots$$
$$T(n) = T(n^{\frac{1}{2^k}}) + k \quad \text{--- ④}$$

Assume $n = 2^m$

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

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

$$\therefore \frac{m}{2^k} = 1$$
$$m = 2^k \text{ and } k = \log_2 m$$
$$\therefore n = 2^m \quad m = \log_2 n$$
$$k = \log \log_2 n$$
$$O(\log \log_2 n)$$

Quicksort(DAC)

- Quicksort is a sorting algorithm based on the divide and conquer approach where

- An array is divided into subarrays by selecting a **pivot element** (element selected from the array).

While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.

- The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
 - At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Quicksort(DAC) Cont.

Pseudocode:

```
/* low --> Starting index, high --> Ending index */
```

```
quickSort(arr[], low, high)
```

```
{
```

```
    if (low < high)
```

```
    {
```

```
        /* pi is partitioning index, arr[pi] is now  
        at right place */
```

```
        pi = partition(arr, low, high);
```

```
        quickSort(arr, low, pi - 1); // Before pi
```

```
        quickSort(arr, pi + 1, high); // After pi
```

```
    }
```

```
}
```

```
/* This function takes last element as pivot, places the pivot element at its  
correct position in sorted array, and places all smaller (smaller than pivot) to  
left of pivot and all greater elements to right of pivot */
```

```
partition (arr[], low, high)
```

```
{
```

```
    // pivot (Element to be placed at right position)
```

```
    pivot = arr[high];
```

```
    i = (low - 1) // Index of smaller element and indicates the  
                // right position of pivot found so far
```

```
    for (j = low; j <= high - 1; j++)
```

```
    {
```

```
        // If current element is smaller than the pivot
```

```
        if (arr[j] < pivot)
```

```
        {
```

```
            i++; // increment index of smaller element
```

```
            swap arr[i] and arr[j]
```

```
        }
```

```
    }
```

```
    swap arr[i + 1] and arr[high])
```

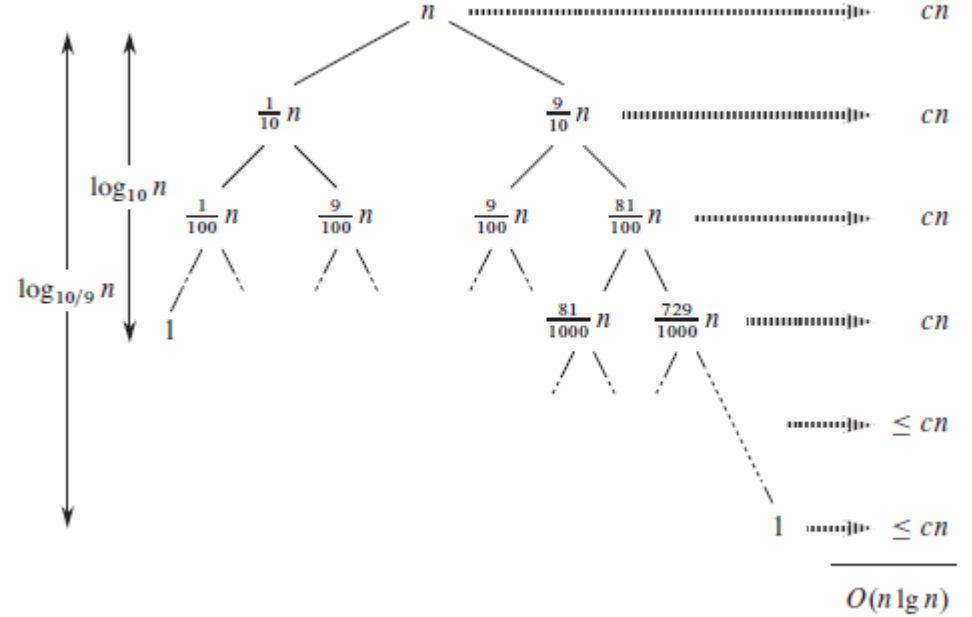
```
    return (i + 1)
```

```
}
```


Quicksort(DAC) Cont.

Analysis:

Best Case Time Complexity: (When PIVOT is in the middle) $n \log n$every step is almost n and height of the tree is $\log n$



Worst Case Time Complexity:
(When PIVOT is the first element)

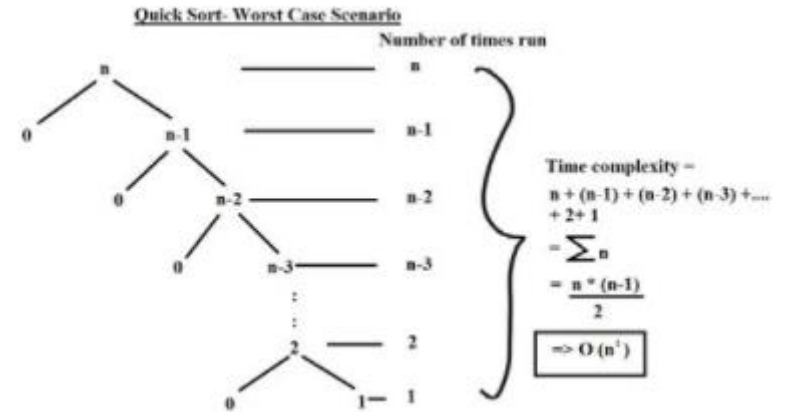
Median: medium element of the sorted list

Each step takes starting from $n, n-1, n-2, \dots, 3, 2, 1$: $n(n+1)/2$

 $O(n^2)$

To improve the Time Complexity:

- Pivot at the middle
- Pivot can be any random number



Quicksort(DAC) Cont.

i/p arr[] = {10, 80, 30, 90, 40, 50, 70}

Indices: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70

Initialize index of smaller element, i = -1

Traverse elements from j = low to high-1

- j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 0

arr[] = {10, 80, 30, 90, 40, 50, 70} **// No change as i and j are same**

- j = 1 : Since arr[j] > pivot, do nothing **// No change in i and arr[]**

- j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 1

arr[] = {10, 30, 80, 90, 40, 50, 70} **// We swap 80 and 30**

- j = 3 : Since arr[j] > pivot, do nothing

// No change in i and arr[]

- j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 2

arr[] = {10, 30, 40, 90, 80, 50, 70} **// 80 and 40 Swapped**

- j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

i = 3

arr[] = {10, 30, 40, 50, 80, 90, 70} **// 90 and 50 Swapped**

We come out of loop because j is now equal to high-1.

Finally we place pivot at correct position by swapping

arr[i+1] and arr[high] (or pivot)

arr[] = {10, 30, 40, 50, 70, 90, 80} **// 80 and 70 Swapped**

Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

Binary Tree(Heap)

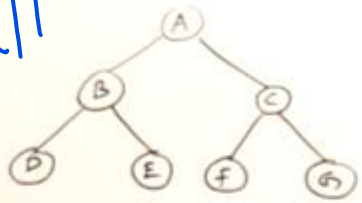
Heap Tree

- The leftmost tree has 0,1,2 height(h) starting from bottom
- Each Full Binary tree has maximum $2^{h+1} - 1$ nodes

Full Binary Tree

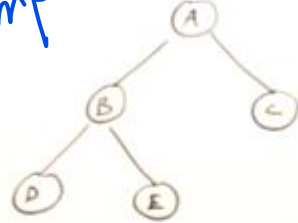
Complete Binary Tree

Full



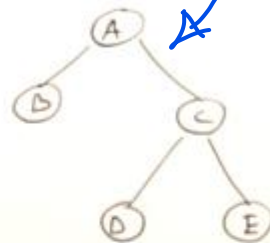
T	A	B	C	D	E	F	G
	1	2	3	4	5	6	7

Complete



T	A	B	C	D	E
	1	2	3	4	5

* Neither full binary tree or complete binary tree



T	A	B	C	-	-	D	E
	1	2	3	4	5	6	7

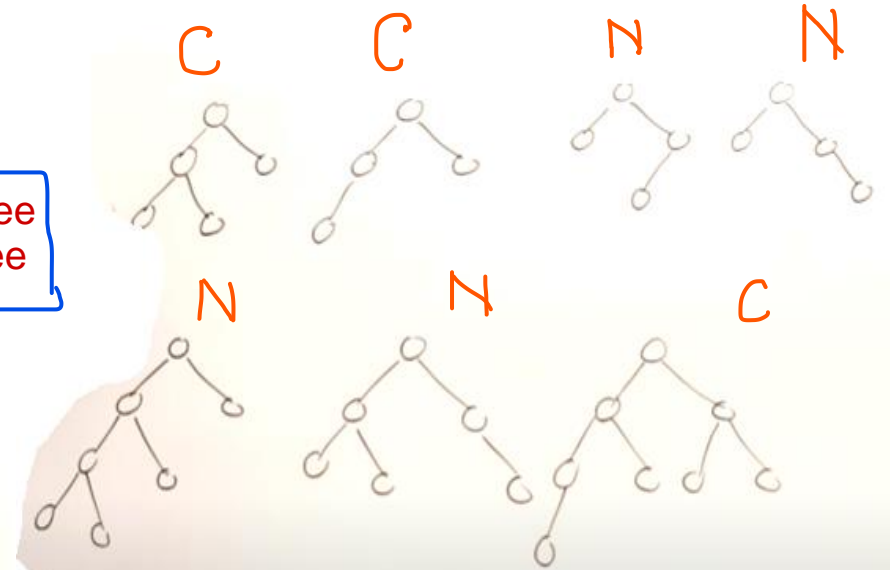
if a Node is at index - i

its left child is at $2*i$

its right child is at $2*i+1$

its parent is at $\lfloor \frac{i}{2} \rfloor$

Array Representation of Binary Tree



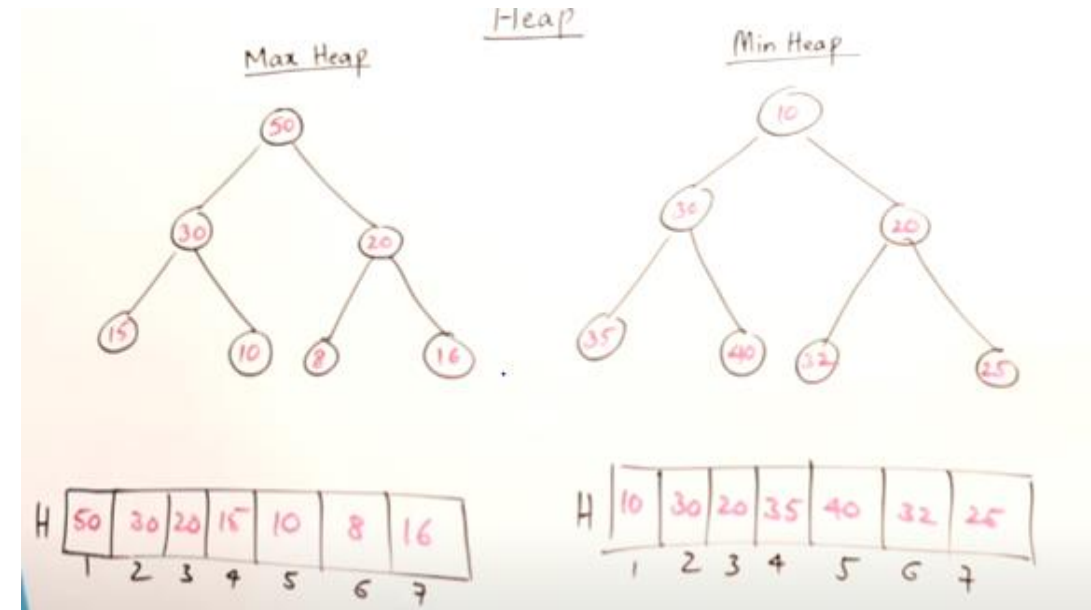
Binary Tree(Heap)

Learning how to write the heap sort algorithm requires knowledge of two types of data structures - arrays and trees.

The initial set of numbers that we want to sort is stored in an array e.g. [10, 3, 76, 34, 23, 32] and after sorting, we get a sorted array [3,10,23,32,34,76].

Heap sort works by visualizing the elements of the array as a special kind of complete binary tree called a heap.

A complete binary tree has an interesting property that we can use to find the children and parents of any node.

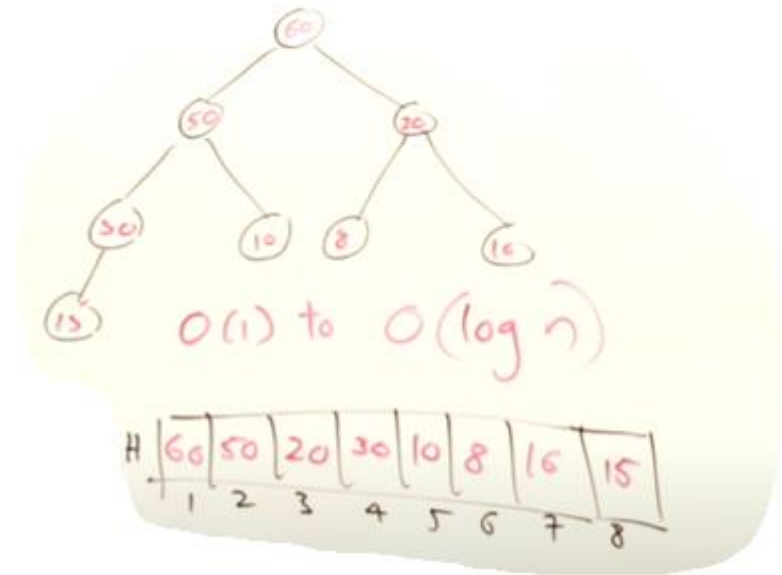
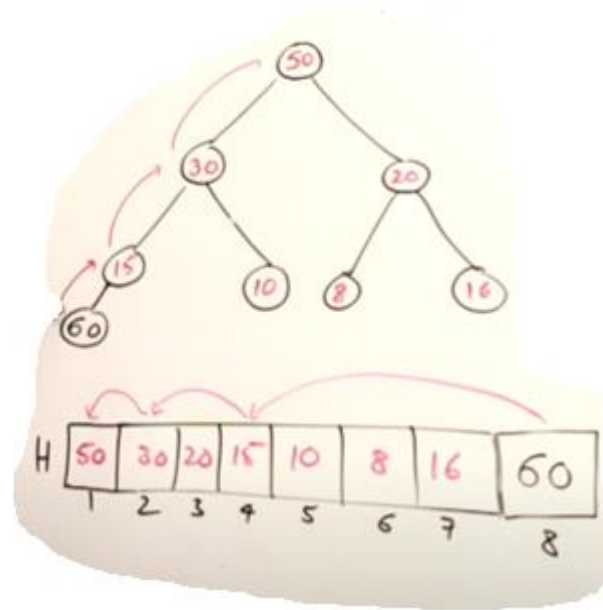


Binary Tree(Heap)

MAX HEAP ADD: Adjustment is done from

Leaf towards Root

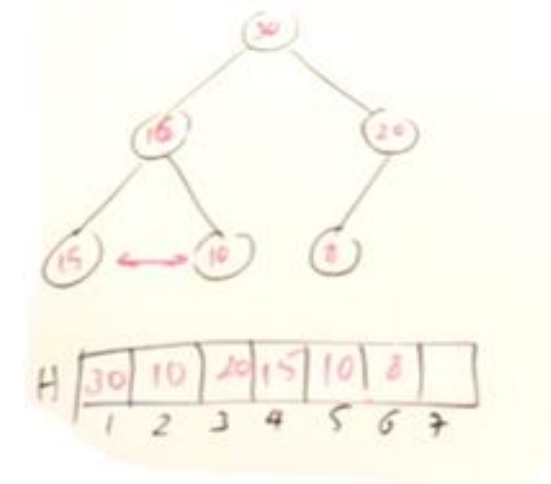
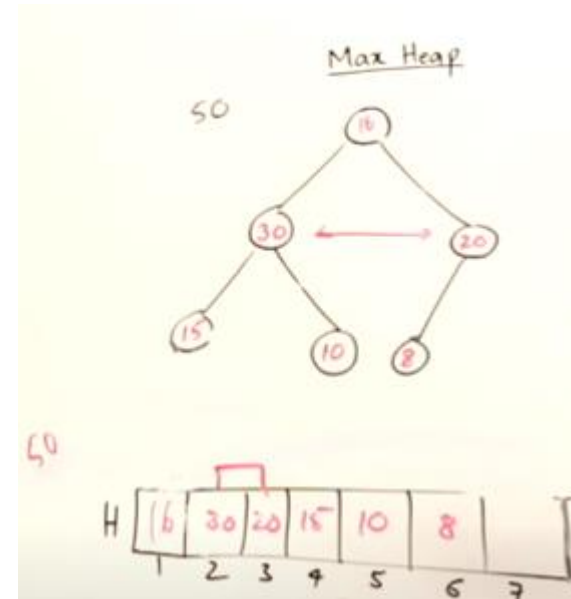
Time Complexity: $O(1)$ to $O(n \log n)$



MAX HEAP DELETE: Adjustment is done from Root

towards Leaf

Time Complexity: $O(n \log n)$

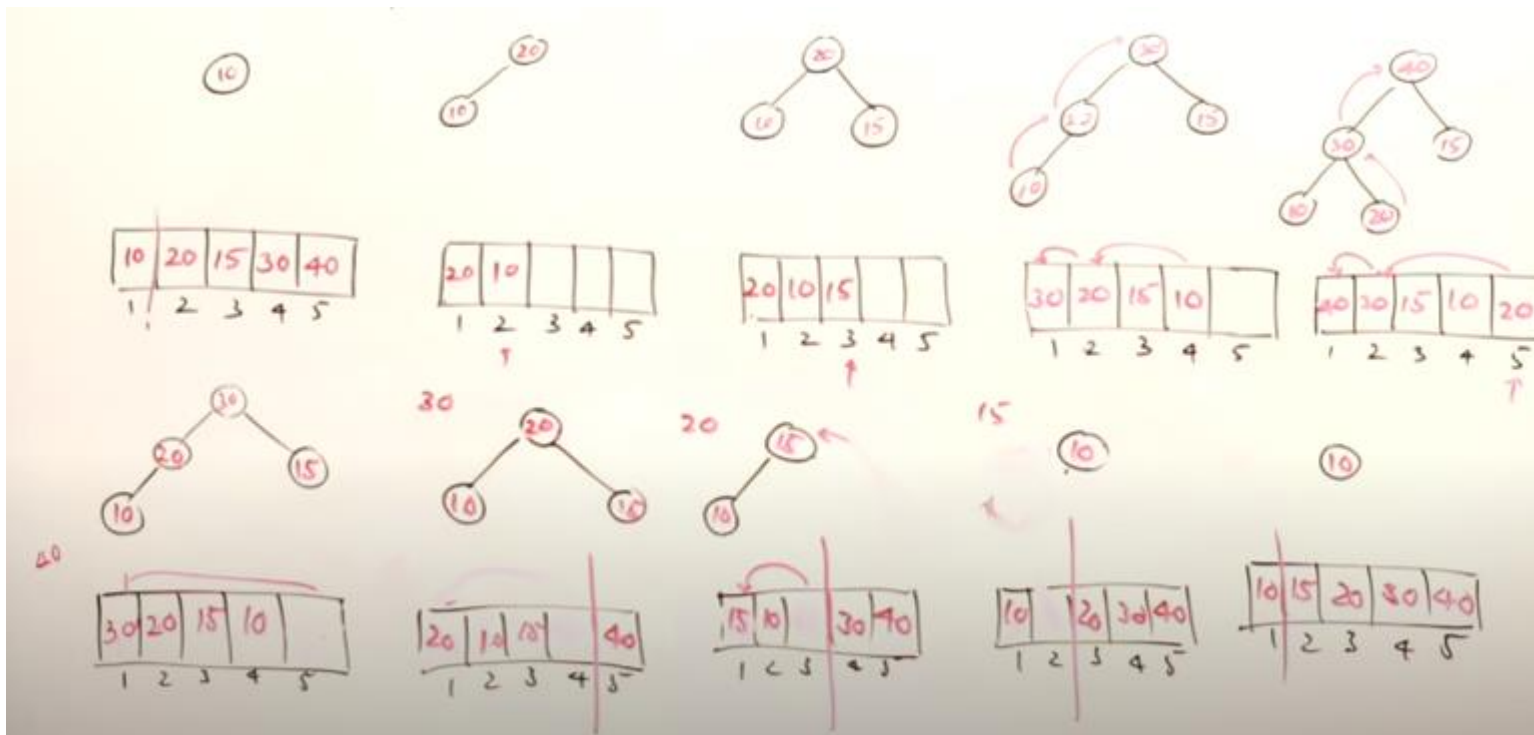


Binary Tree(Heap)

For Heapsort,

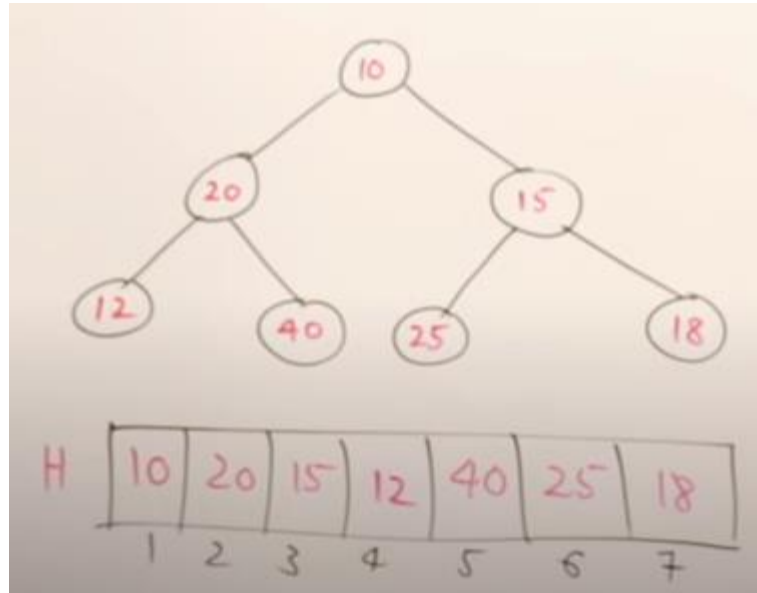
- ✓ First Create the heap(max) from given array.
- ✓ Then delete one by one element from heap to form a sorted array.

Time Complexity: $n \log n + n \log n = O(n \log n)$



Heapify

- Starting from a complete binary tree, we can modify it to become a Max-Heap by running a function called heapify on all the non-leaf elements of the heap.
- Since heapify uses recursion, it can be difficult to grasp.
- In the figure, it is a complete B.T. but not Max Heap. We need to form that using Heapify method. Start from the last element of array and go upwards.



Heap Priority Queue

Steps:

- Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.
- Swap: Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.
- Remove: Reduce the size of the heap by 1.
- Heapify: Heapify the root element again so that we have the highest element at root.
- The process is repeated until all the items of the list are sorted.

