Module-1, Part 2

BCSC0012

Designing Techniques of Algorithms, Recurrence Relations, Design & Analysis of Sorting Algorithms, Linear Sorting, Searching and Red Black Tree

Design and Analysis of Algorithms

Dr. Gaurav Kumar
Asst. Prof, CEA
GLA University, Mathura

Why Designing Techniques

- A problem can be solved in various different approaches
- Some approaches deliver much more efficient results than others.
- The algorithm Designing technique is used to measure the effectiveness and performance of the algorithms.

List of several popular design approaches

What we'll briefly understand today

01

Divide and Conquer Strategy

02

Greedy Technique

03

Dynamic Programming

04

Branch and Bound

05

Backtracking Strategy

Divide and Conquer Strategy

What you should know

01

Dividing the problem into sub-problems

02

Solve every subproblem individually, recursively

03

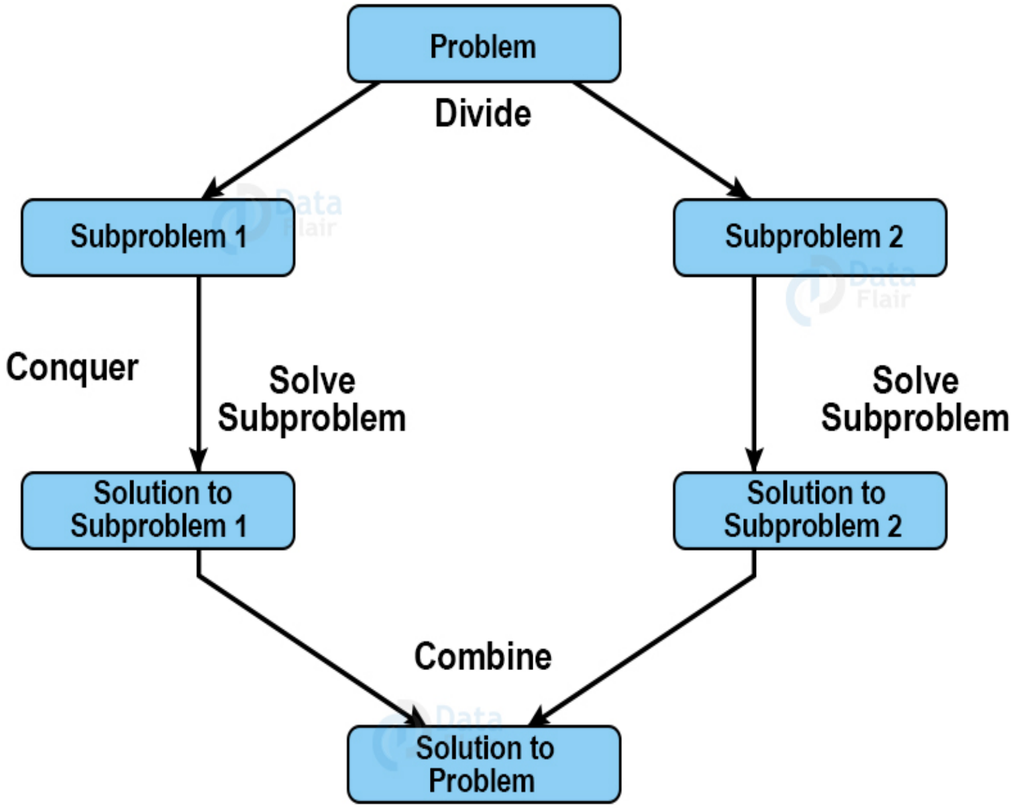
Combining them for the final answer

04

Follows top-down approach

Divide and Conquer Strategy

What you should know



This technique can be divided into the following three parts:

Divide: This involves dividing the problem into smaller sub-problems.

Conquer: Solve sub-problems by calling recursively until solved.

Combine: Combine the sub-problems to get the final solution of the whole problem.

The strategy of designing algorithms has two fundamentals

01

Recurrence formula

02

Stopping Conditions

Divide and Conquer Strategy

What you should know

01

Recurrence Formula/Equation

Function has been defined in terms of same function with some changes in the argument.

02

Stopping Conditions

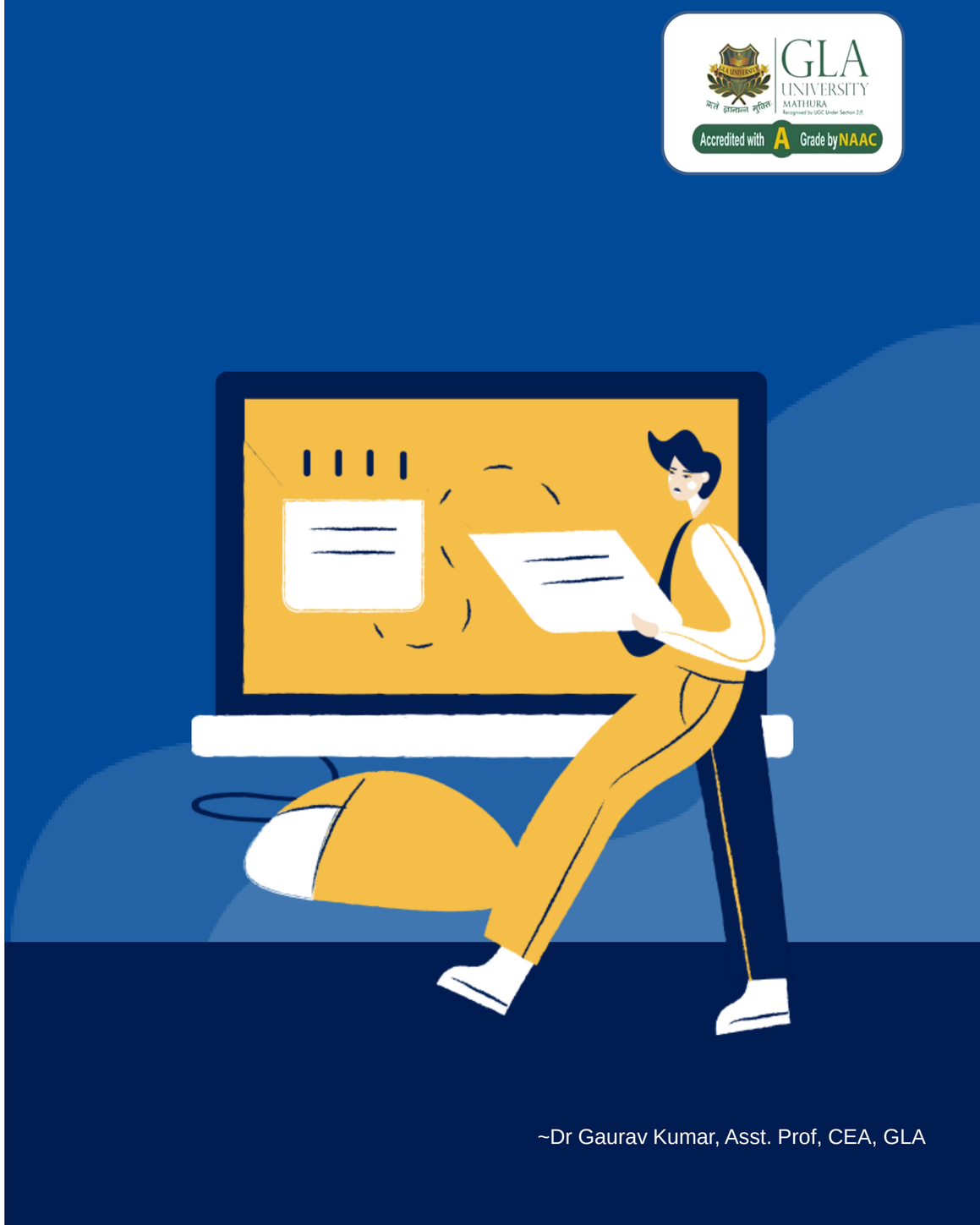
When you divide the problem by divide and conquer strategy, you need to understand for how long you have to keep dividing. You have to put a stopping condition to stop your recursion steps.

~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

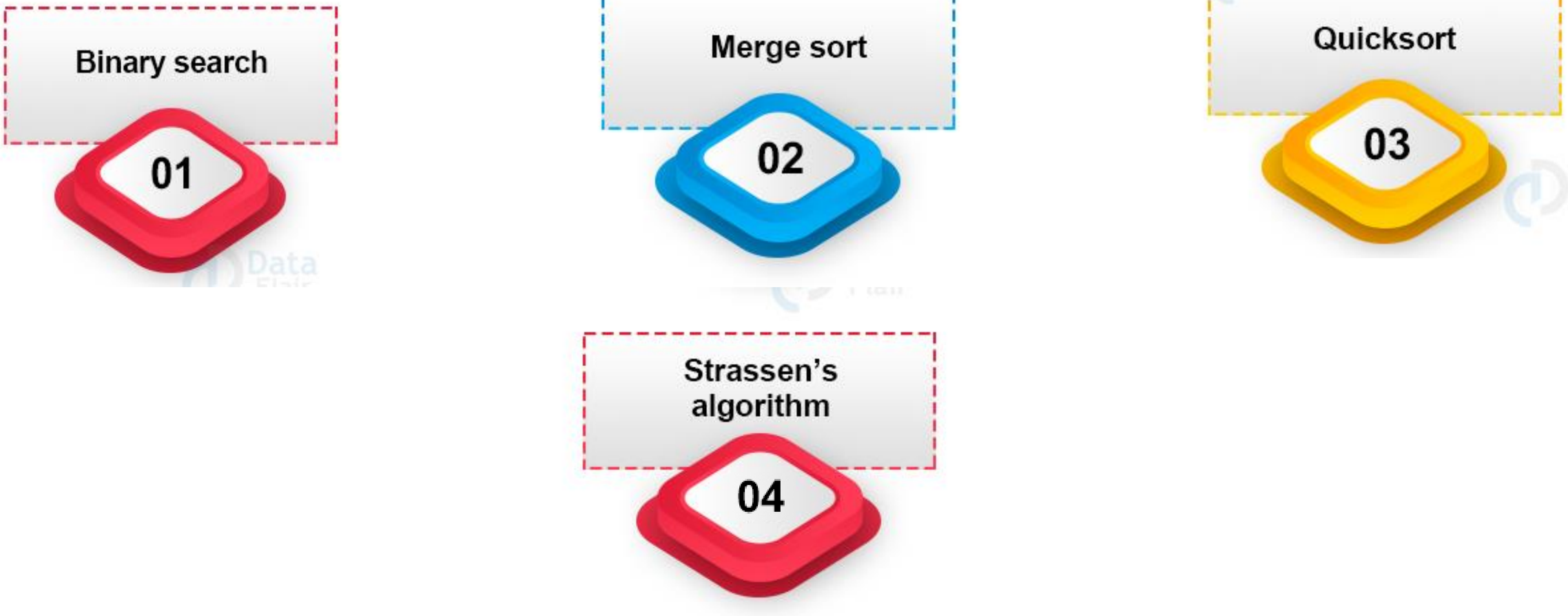
Examples

For Example, the Worst Case Running Time T(n) of the MERGE SORT Procedures is described by the recurrence

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ 2T(n/2) + O(n) & \text{if } n>1 \end{cases}$$



Applications of Divide and Conquer



Advantage of Divide and Conquer Strategy

- Divide and conquer successfully solved one of the biggest problems of the mathematical puzzle world, the tower of Hanoi.
- You might have a very basic idea of how the problem is going to be solved but dividing the problem makes it easy since the problem and resources are divided.
- It Is very much faster than other algorithms.
- The divide and conquer algorithm works on parallelism. Parallel processing in operating systems handles them very efficiently.
- The divide and conquer strategy used cache memory without occupying much main memory. Executing problems in cache memory which is faster than main memory.

Disadvantage of Divide and Conquer Strategy

- Most of the divide and conquer design uses the concept of recursion therefore it requires high memory management.
- It may crash the system if recursion is not performed properly.
- All iterative nature problems can not be solved using Divide and Conquer strategy, but vice versa is true.

Recursion & Recurrence Relations

~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

What does recursion means

- A function calling itself is called **recursion** & corresponding function is called as **recursive function**.

```
fact(int n)
{
   if (n < = 1) // base case
   return 1;
   else
   return n*fact(n-1);
}
```

How recursion works

Example- Sum of first n natural numbers

Approach(1) - Simply adding one by one

```
Algorithm Sum (A, n)
{
  S=0;
  for ( i=0; i<n; i++)
  {
    S= S + A[i];
   }
  return S;
}
```

Approach(2) – Recursive adding

Sum(n) = n+ sum(n-1)

Sum (5) = 5+4+3+2+1

Sum (5) = 5 + Sum(4)

Sum (4) = 4 + Sum(3)

Sum (3) = 3 + Sum(2)

Sum (2) = 2 + Sum(1)

Sum (1) = 1

```
Algorithm Sum (n)
{
 if(n==1)
  return 1;
  else
  return  n + sum(n-1);
}
```

What does recursion relation means

$$\text{Sum}(n) = \begin{cases} 1 & \text{if } n=1 \\ \text{Sum}(n-1) + n & \text{if } n>1 \end{cases}$$

Base/Stopping Condition

- A recurrence relation is an equation or inequality that describes a function in terms of its values on smaller inputs.
- Used to reduce complicated problems to an iterative process based on simpler versions of the problem
- T(n) term is used to define the time complexity in recurrence relation analysis.

There are four methods for solving Recurrence

1. Back Substitution/ Iteration Method
2. Recursion Tree Method
3. Master Method
4. Substitution Method

~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

How to analyze the Recurrence Algorithm

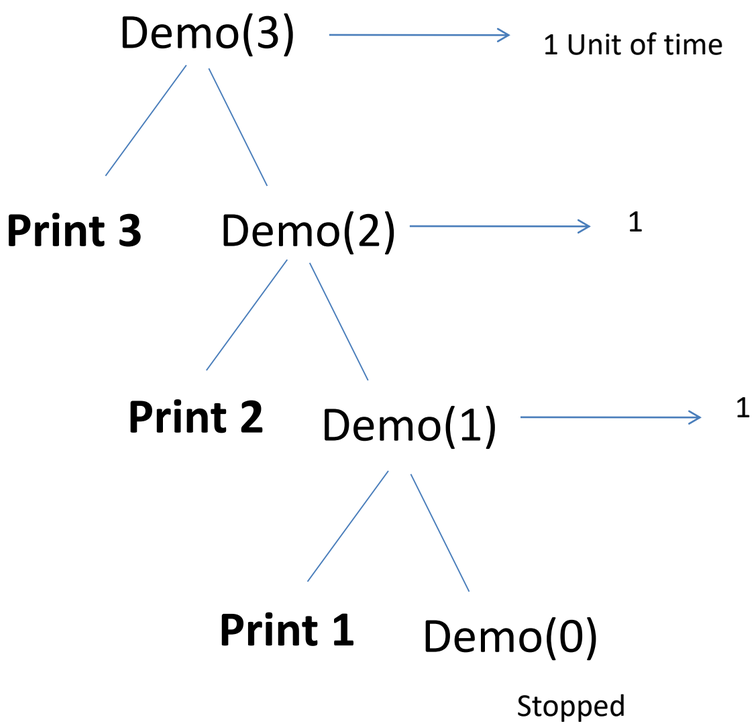
Recursion Tree Method

Draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels

Example- Analyze the time complexity of this algorithm

```
Algorithm Demo(n)
{
  If (n>0)
    {
     Print " n";
     Demo(n-1);
     }
}
```

Lets put n=3



T(n)= n → 0(n)

Number of times Print statements is executed = 3
Total number of function call = 3+1

~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

How to write and solve a Recurrence Equation

Example- Analyze the time complexity of this algorithm

```
Algorithm Demo(n)                    ———— T(n)
{
  If (n>0)
    {
     Print " n";                     ———— 1
     Demo(n-1);                      ———— T(n-1)
     }
}
```

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

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

How to write and solve a Recurrence Equation **(Back Substitution/ Iteration Method)**

$T(n)= T(n-1) + 1$ ——— (1)

Calculate the T(n-1) and then substitute the value

Put n= n-1 in Equation (1)

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

$= T(n-2) + 1$

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

Put n= n-2 in Equation (1)

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

$= T(n-3) + 1$

$T(n)= T(n-3) +1+ 1+1$

Continued for k times

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

Recall this equation

$S(n) = n+ s(n-1)$

If I know s(n-1),
then I can easily calculate s(n)

We know that T(0)=1
(base Condition)

Assume that n-k=0

Then k=n

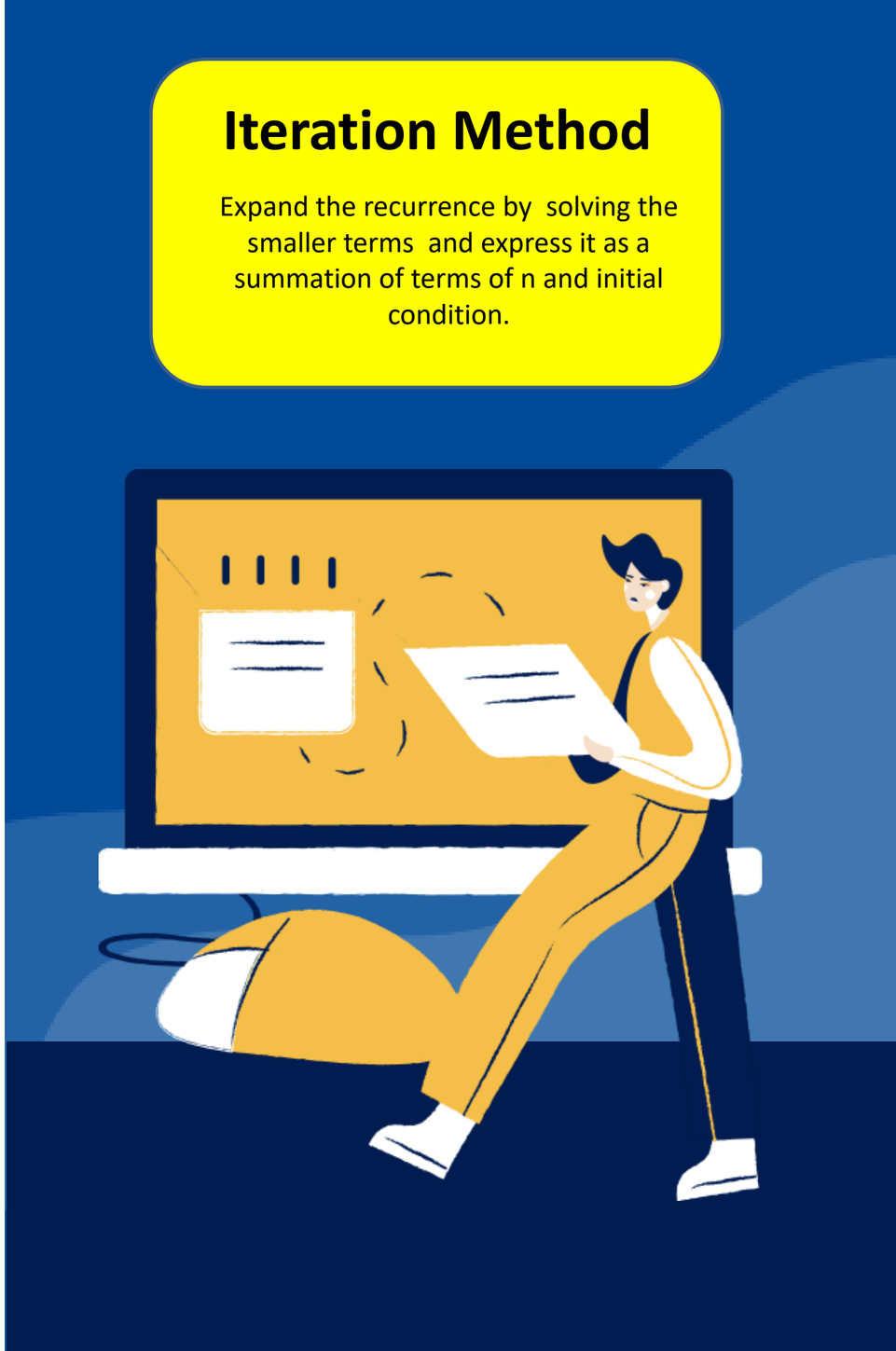
$T(n)= T(0) + k*1$

$T(n)= 1 + n$

$T(n)= O(n)$

Iteration Method

Expand the recurrence by solving the smaller terms and express it as a summation of terms of n and initial condition.



How to write and solve a Recurrence Equation

Master Method (cookbook method)

- Master Method is a direct way to get the solution.
- The master method works for following type of recurrences or for recurrences that can be transformed to following type. (*)

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

where a >= 1 and b >= 1

a = number of sub-problems in the recursion

n/b = size of each sub-problem

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

(*) we can also derive the master theorem for decreasing function.

How to analyze the Recurrence Algorithm

Substitution Method

We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect

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

We guess the solution as T(n) = O(n)

Now we use mathematical induction to prove our guess. We need to prove that T(n) <= cn for all value of n>=n_o and c>0

T(n) = T(n-1) + 1
= c(n-1) + 1 = cn-c+1 = cn+1-c

T(n)= cn+1-c <= cn for all c>=1

Lets Put c=1 1*n+1-1<= 1*n n=n

Put c=2 2*n+1-2<=2*n

2n-1 < 2n

~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

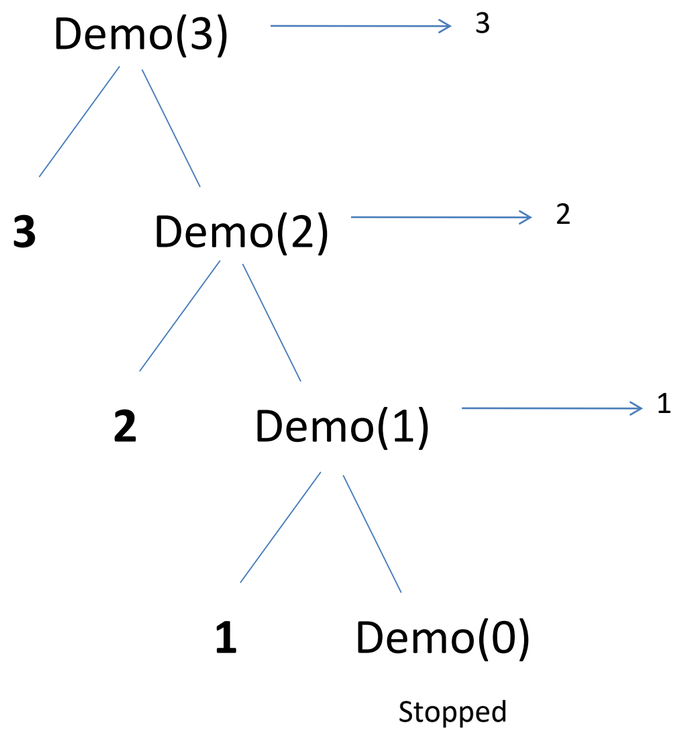
How to analyze the Recurrence Algorithm

```
Algorithm Demo(n)  ------> T(n)
{
  If (n>0)
  {
     for (i=0; i<n; i++)
     {
     print " n";   ------> n
     }
   Demo(n-1);   ------> T(n-1)
   }
}
```

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

Lets put n=3

Recurrence Tree Method



Number of times Print statements is executed =1+2+3

Total number of function call = 3+1

For n elements, number of times Print statements is executed $=1+2+3+4+5+\ldots+n = n(n+1)/2 = n^2$

$T(n)= O(n^2)$

~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

How to analyze the Recurrence Algorithm

Iterative Method

Recurrence Relation $T(n) = \begin{cases} 1 & \text{if } n=0 \\ T(n-1) + n & \text{if } n>0 \end{cases}$ (Base Equation)

Put n= n-1

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

=T(n-2) + n-1

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

Put n= n-2 in the base equation

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

= T(n-3) + n-2

T(n)= T(n-3) +n-2 + n-1 + n

= T(n-4)+ n-3 + n-2+ n-1+ n

Continued for k times

T(n)= T(n-k) +n-k+1 + n-k+2 +…n-1 + n

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

For n=0, T(0)=1, i.e. n-k=0 -> n=k

T(n)= T(0) + n-(n +1)+ n-(n-2) + …..n-1 + n

= T(0) + 1 + 2+ 3+ ……n-1 + n

= T(0) + n (n+1)/2

= 1 + (n^2 + n)/2

= O(n^2)

T(n)= O(n^2)

~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

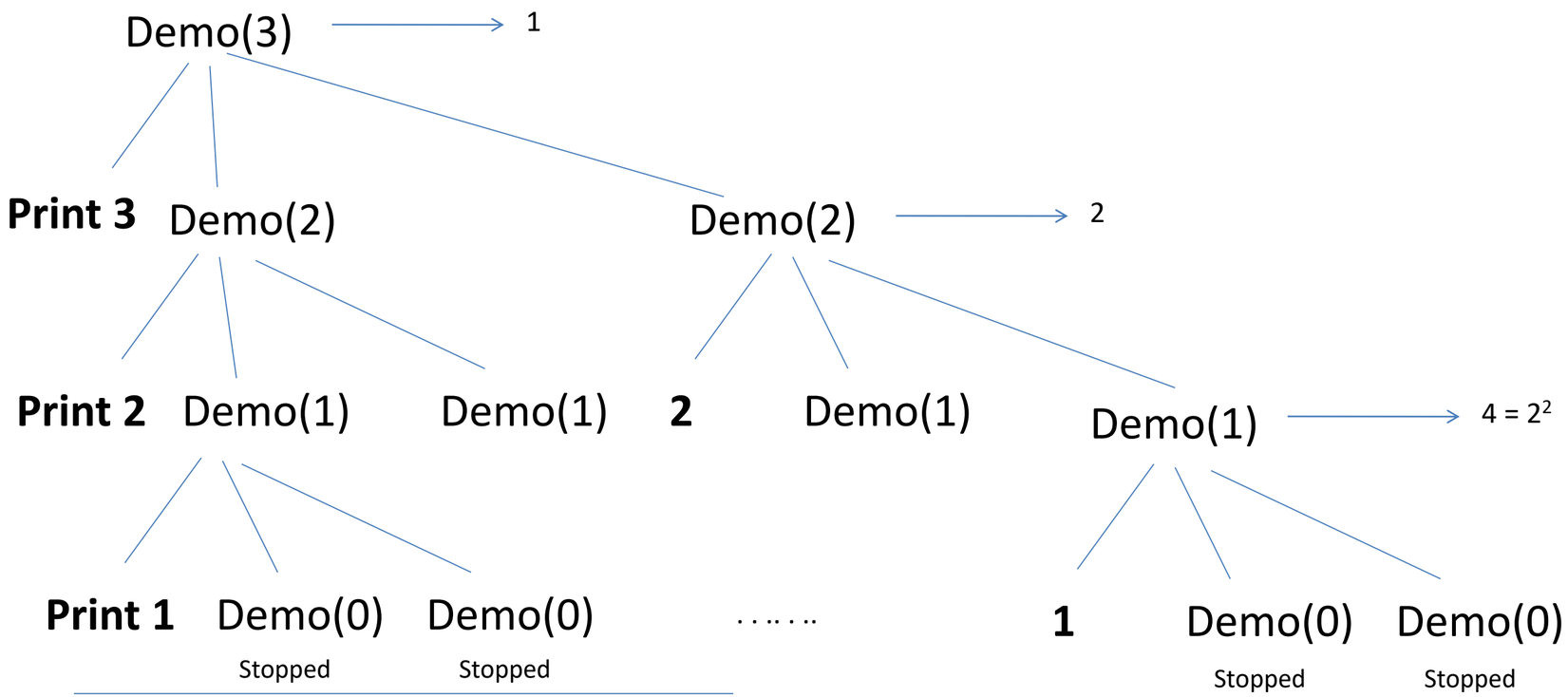
How to analyze the Recurrence Algorithm

```
Algorithm Demo(n)  ------> T(n)
 {
   If (n>0)
   {
    print " n";      ------> 1
    Demo(n-1);       ------> T(n-1)
    Demo(n-1);       ------> T(n-1)
    }
 }
```

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

Lets put n=3

Recurrence Tree Method



Number of times Print statements is executed $=1+2+2^2$

For n elements, number of times Print statements is executed $=1+2+2^2+2^3+2^4+\ldots+2^n=2^{n+1}-1=2^n$

$T(n)= O(2^n)$

Using GP Series formula: $a+ar^2+ar^3+ar^4+\ldots.ar^n= ar^{n+1}-1/r-1$

How to analyze the Recurrence Algorithm

Recurrence Relation

T(n)= 2T(n-1) + 1 for n>0

= 1 for n=0

Iteration Method

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

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

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

= 2^2 (2T(n-3) + 1)+ 2 + 1

= 2^3 T(n-3) + 2^2 + 2 + 1

.

.

.

= 2^k T(n-k) + 2^{k-1} + ….. 2^2 +2 + 1

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

Put T(n-2)= 2T(n-3) + 1

For n=0, T(0)=1, it means that n-k=0 → n=k

$2^n + 2^{n-1}$ + ….. 2^2 +2 + 1 = 2^{n+1} -1 = O(2^n)

~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

Quick Revision of Time Complexity (Decreasing Functions)

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

$= 1$ for $n=0$

$T(n) = O(n)$

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

$= 1$ for $n=0$

$T(n) = O(n^2)$

$T(n) = 2T(n-1) + 1$ for $n>0$

$= 1$ for $n=0$

$T(n) = O(2^n)$

$T(n) = 2T(n-1) + n$ for $n>0$

$= 1$ for $n=0$

$T(n) = O(n.\ 2^n)$

~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

How to analyze the Recurrence Algorithm

(Dividing Functions)

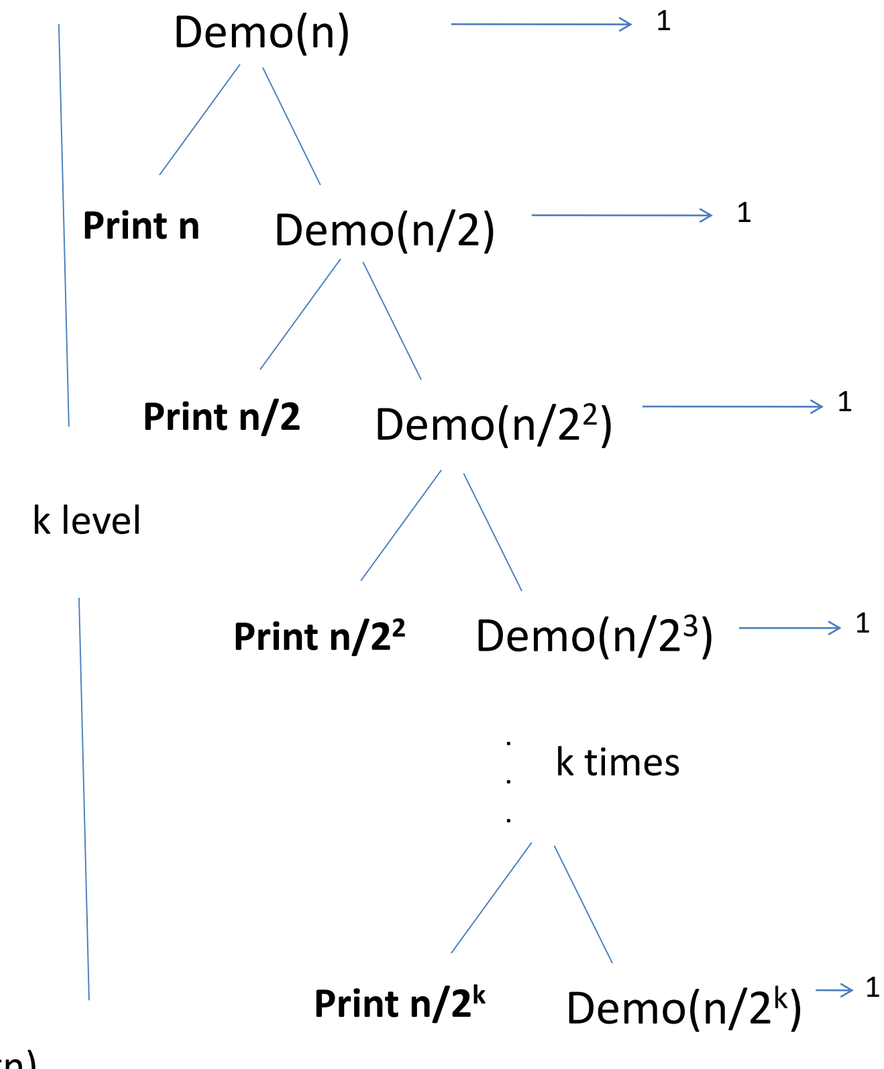
```
Algorithm Demo(n)  ------------------------------------------> T(n)
{
  If (n>1)
    {
     Print " n";  ------------------------------------------> 1
     Demo(n/2);  ------------------------------------------> T(n/2)
     }
}
```

Stopping Condition: $n/2^k=1$

$= n=2^k$

$= k= \log n$ = T(n)= O(logn)

Recurrence Tree Method



~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

How to analyze the Recurrence Algorithm (Dividing Functions)

Recurrence Relation

T(n)= T(n/2) + 1 for n>1

= 1 for n=1

T(n)= O(logn)

Iterative Method

$T(n)= T(n/2) + 1$ Put n=n/2, $T(n/2) = T(n/2^2) +1$

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

$= T(n/2^3) +1 +1 +1$

.

.

.

$= T(n/2^k) + K*1$

$T(n) =T(1) + k$ put $n/2^K=1$

$= 1+k$ $= n=2^K$

$= \log n+1$ $=k= \log n$

$= O(\log n)$

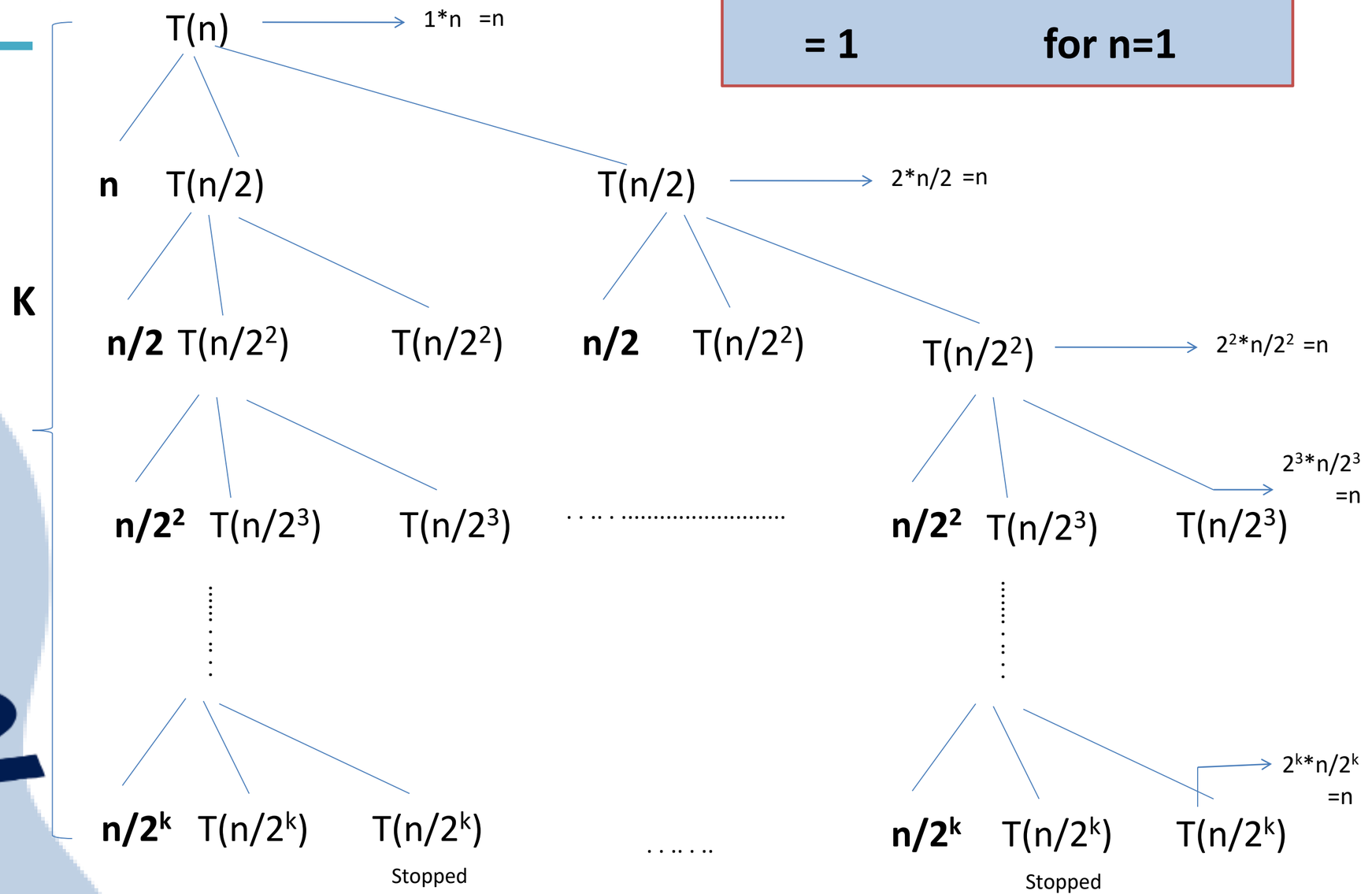
How to analyze the Recurrence Algorithm

(Dividing Functions)

Recurrence Relation

T(n)= 2T(n/2) + n for n>1

= 1 for n=1



How to analyze the Recurrence Algorithm (Dividing Functions)

Recurrence Relation

T(n)= 2T(n/2) + n for n>1

= 1 for n=1

T(n)= n + n+ n+ n+ n+ ……+ n (k times)

= n*k

or

T(n)= number of level or height of tree * time taken at each level

= k* (n)

= n*k

T(n)= n* logn

As we know that stopping condition

$n/2^k=1$

$n=2^k$ (apply log)

=> k = logn

How to analyze the Recurrence Algorithm

(Dividing Functions)

Recurrence Relation

T(n)= T(n/2) + n for n>1

= 1 for n=1

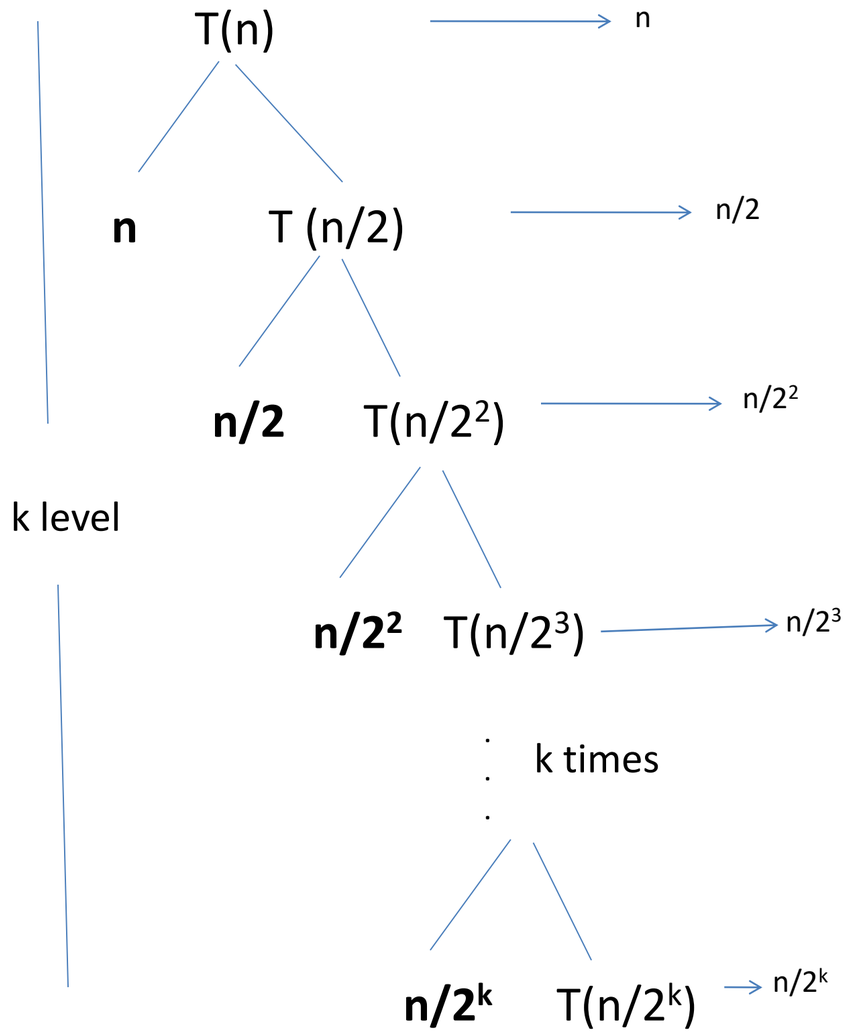
$T(n)= n + n/2 + n/2^2 + n/2^3 + \ldots.. + n/2^K$

$= n (1+1/2 + 1/2^2 + 1/2^3 \ldots\ldots\ldots + 1/2^k)$

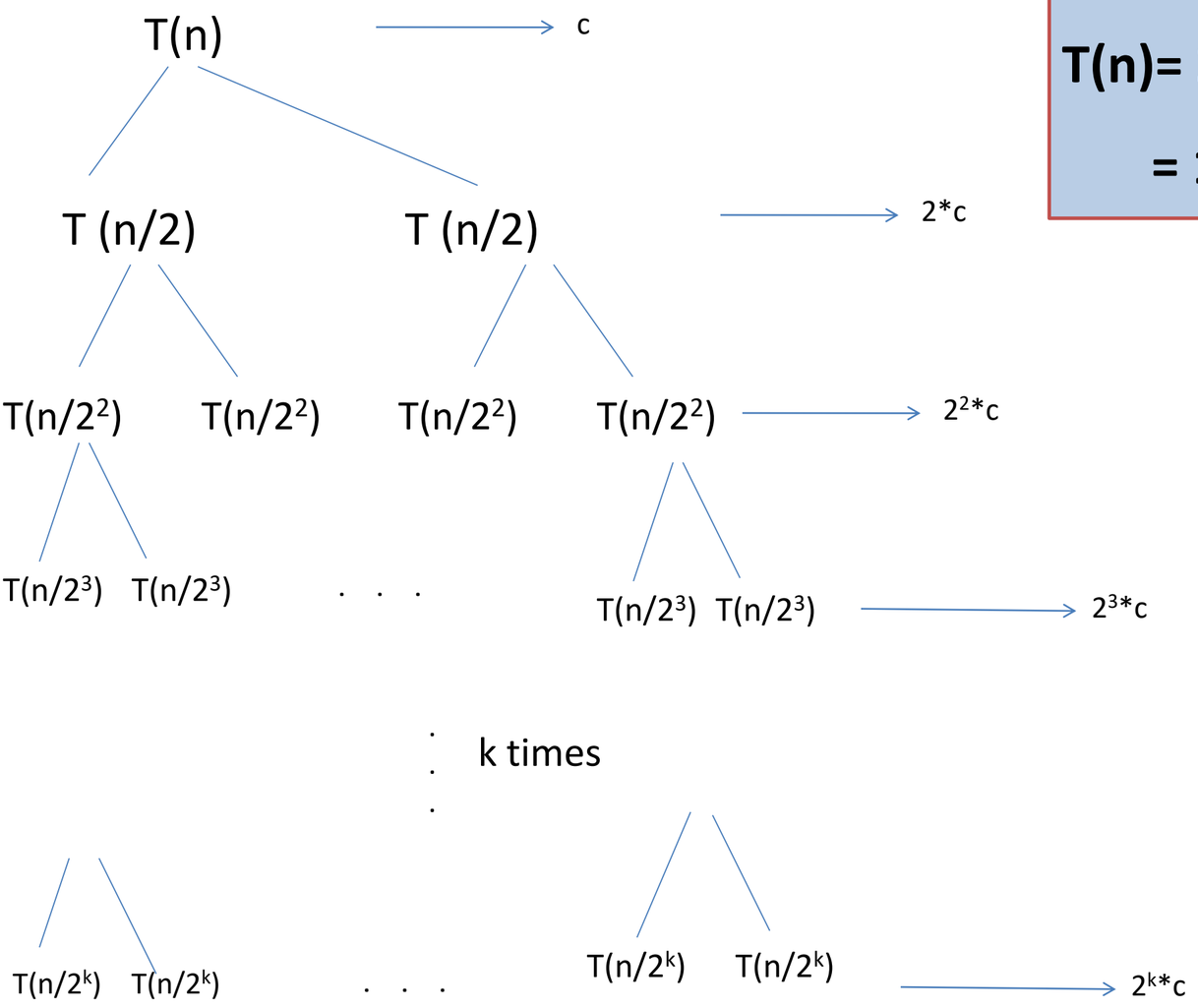
$=n*1$

$=O(n)$

Recurrence Tree Method



How to analyze the Recurrence Algorithm (Dividing Functions)



Recurrence Relation

T(n)= 2T(n/2) + c for n>1

= 1 for n=1

$T(n)= c + 2*c + 2^2*c + 2^3*c + \ldots\ldots + 2^k*c$

$= c\,(1+2 + 2^2 + 2^3\ldots\ldots\ldots + 2^k)$

$= c\,(2^{k+1}-1)$

$= c^* 2^k$

put k =logn

$= c^* 2^{\log n}$

$= c*n \quad (2^{\log n} = n^{\log 2})$

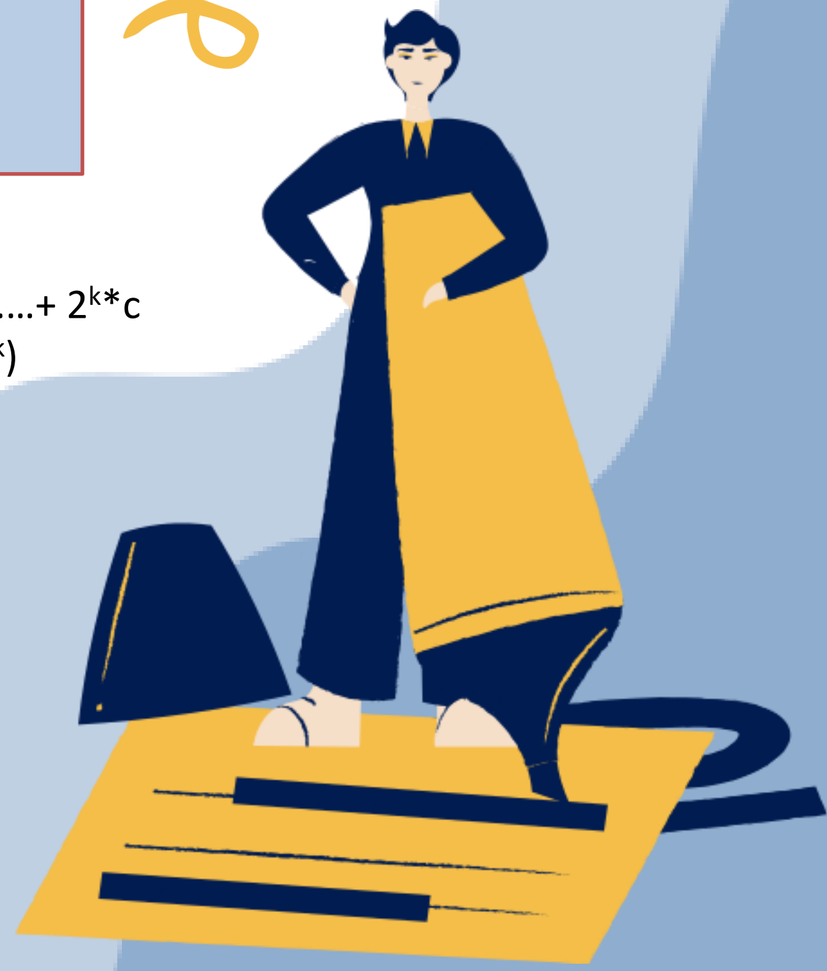
T(n)= O(n)

As we know that stopping condition

$n/2^k=1$

$n=2^k$ (apply log)

=> k = logn



~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

Quick Revision of Time Complexity (Dividing Functions)

T(n)= T(n/2) + 1 for n>1

= 1 for n=1

T(n)= O(logn)

T(n)= 2T(n/2) + n for n>1

= 1 for n=1

T(n)= O(nlogn)

T(n)= T(n/2) + n for n>1

= 1 for n=1

T(n)= O(n)

T(n)= 2T(n/2) +c for n>1

= 1 for n=1

T(n)= O(n)

~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

How to analyze the Recurrence Algorithm (Dividing Functions)

Master Method (cookbook method)

- Master Method is a direct way to get the solution.
- The master method works for following type of recurrences or for recurrences that can be transformed to following type. (*)

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

n = size of input ,

a = number of sub-problems in the recursion , n/b = size of each sub-problem.

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

Here, a ≥ 1 and b > 1 are constants, and f(n) is an asymptotically positive function.

How to analyze the Recurrence Algorithm (Dividing Functions)

Master Method (cookbook method)

$T(n) = aT(n/b) + f(n)$ Calculate $n^{\log_b a}$

Case -1 If $f(n) > n^{\log_b a}$ **Then $T(n) = O(f(n))$**

Case -2 If $f(n) < n^{\log_b a}$ **Then $T(n) = O(n^{\log_b a})$**

Case -3 If $f(n) = n^{\log_b a}$ **Then $T(n) = O(n^{\log_b a} \log n)$ or $O(f(n)^* \log n)$**

How to analyze the Recurrence Algorithm (Using Mater Theorem)

Solve it using Master Theorem

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

a=2, b=2, f(n) = n

Calculate $n^{\log_b a}$

$$n^{\log_2 2} = n$$

f(n) = n is given

Case-3 condition satisfied

Example -1

T(n)= 2T(n/2) + n for n>1

= 1 for n=1

T(n)= O(nlogn)

Case -3 If $f(n) = n^{\log_b a}$

Then T(n) = O(f(n)* logn)

Hence, T(n) = O(f(n)*Logn)

= O(n*logn)

~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

Examples Exercise of Master Theorem

Master Theorem

$T(n) = a\,T(n/b) + f(n)$

Example -2

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

$a=4, b=2, f(n) = n^3$

Calculate $n^{\log_b a} = n^{\log_2 4} = n^2$

$f(n) = n^3$ is given

Case-1 condition is satisfied

Hence, $T(n) = O(f(n))$

$= O(n^3)$

Example -3

$T(n) = 7T(n/2) + n^2$

$a=7, b=2, f(n) = n^2$

Calculate $n^{\log_b a} = n^{\log_2 7}$

$f(n) = n^2$ is given

Case-2 condition is satisfied

Hence, $T(n) = O(n^{\log_b a})$

$= O(n^{\log_2 7})$

~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

Examples Exercise of Master Theorem

Example -4

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

$a=1, b=4, f(n) = n$

Calculate $n^{\log_4 1} = n^0 = 1$

$f(n) = n$ is given

Case-1 condition is satisfied

Hence, $T(n) = O(f(n)$

$= O(n)$

Master Theorem

$T(n) = a\,T(n/b) + f(n)$

Example -5

$T(n) = 4T(7n/3 + 5) + n^2$

$a=4, b=3/7, f(n) = n^2$

Calculate $n^{\log_b a} = n^{\log_4 3/7}$

$f(n) = n^2$ is given

Case-1 condition is satisfied

Hence, $T(n) = O(f(n))$

$= O(n^2)$

Not all recurrence relations can be solved with the use of the master theorem i.e. if

- T(n) is not monotone, ex: T(n) = sin n

- f(n) is not a polynomial, ex: $T(n) = 2T(n/2) + 2^n$

Advance Master Theorem (Extended)

$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$

where n = size of the problem , a>1 and b>1
a = number of sub-problems in the recursion , n/b = size of each sub-problem,

k >= 0 and p is a real number, then,

Case 1- if $a > b^k$, then $T(n) = \theta(n^{\log_b a})$

Case -2 if $a = b^k$, then

(a) if $p > -1$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
(b) if $p = -1$, then $T(n) = \theta(n^{\log_b a} \log\log n)$
(c) if $p < -1$, then $T(n) = \theta(n^{\log_b a})$

Case-3 if $a < b^k$, then

(a) if $p >= 0$, then $T(n) = \theta(n^k \log^p n)$
(b) if $p < 0$, then $T(n) = \theta(n^k)$

Example of Advance Master Theorem

$T(n) = aT(n/b) + \theta(n^k \log^p n)$

Example -1

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

$a = 2, b = 2, k = 1, p = 2$

$b^k = 2$. So, $a = b^k$ [Case 2.(a)]

$T(n) = \theta(n^{\log_b a} \log^{p+1} n)$

$T(n) = \theta(n^{\log_2 2} \log^3 n)$

$T(n) = \theta(n\log^3 n)$

~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

Example of Advance Master Theorem

$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$

Example -2

$T(n) = 3T(n/2) + n^2$

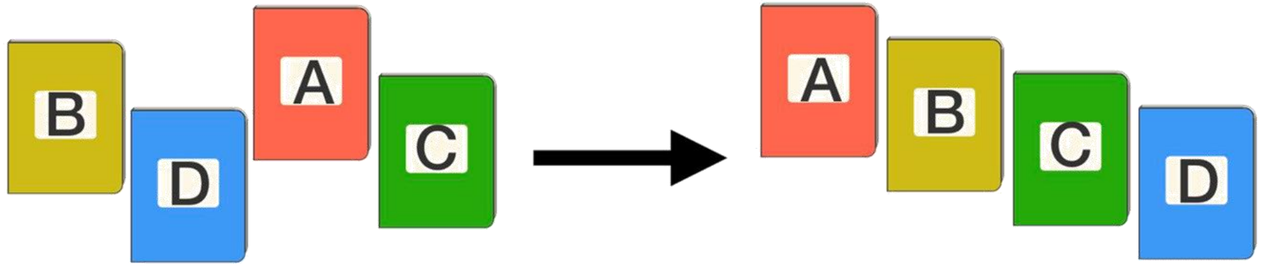
$a = 3, b = 2, k = 2, p = 0$

$b^k = 4$. So, $a < b^k$ and $p = 0$ [Case 3.(a)]

$T(n) = \theta(n^k \log^p n)$

$T(n) = \theta(n^2)$

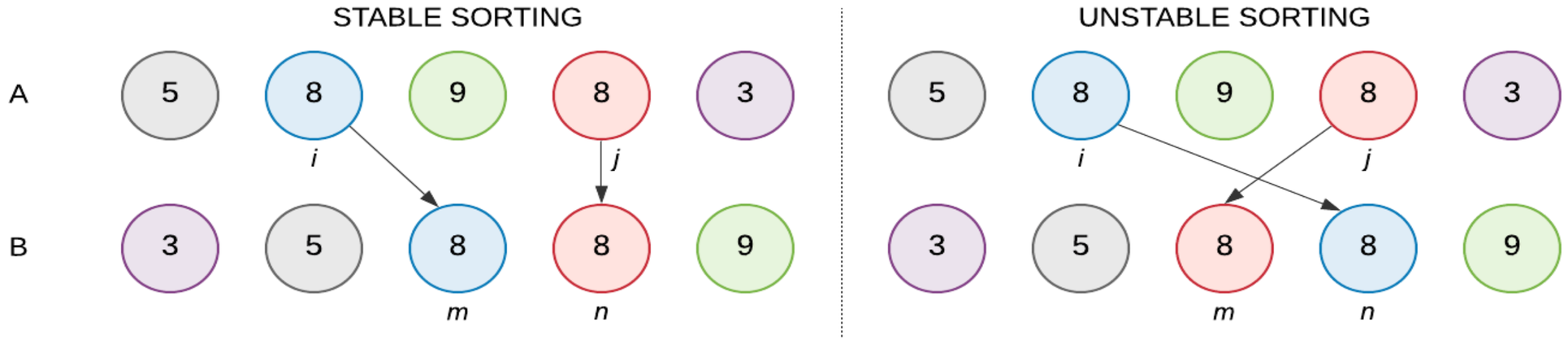
Design and Analysis of Sorting Algorithms



~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

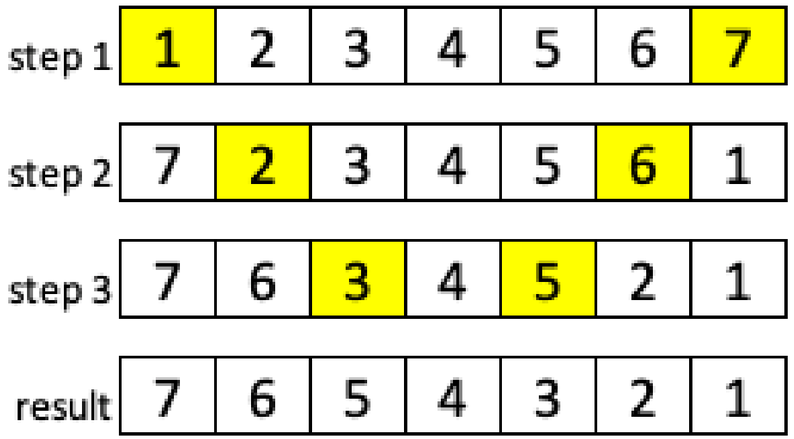
Stable Sorting Algorithm

A sorting algorithm is said to be stable if it maintains the relative order of records in the case of equality of keys.



In-place Sorting Algorithm

An in-place algorithm is an algorithm that does not need an extra space and produces an output in the same memory that contains the data by transforming the input ‘in-place’. However, a small constant extra space used for variables is allowed.



GLA UNIVERSITY
MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

1. Design & Analysis of Insertion Sort Algorithm

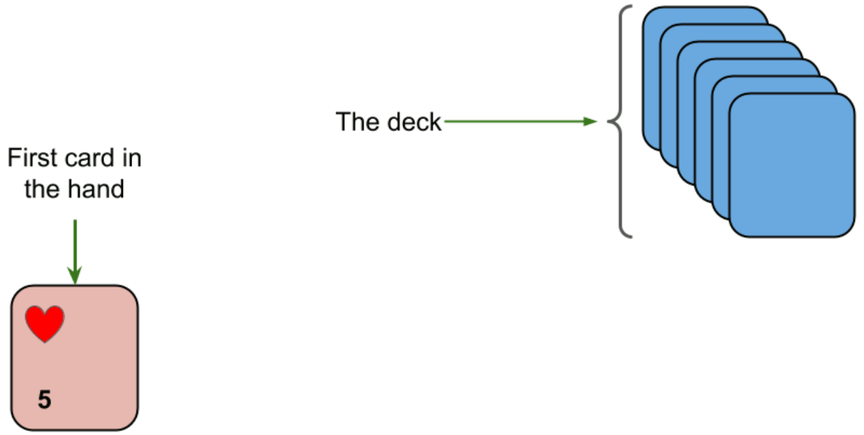
~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

1. Insertion Sort Algorithms

1. Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands.

2. The array is virtually split into a sorted and an unsorted part.

3. Values from the unsorted part are picked and placed at the correct position in the sorted part.



Understanding Insertion Sort

To sort an array of size n in ascending order

6 5 3 1 8 7 2 4

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step2 - Pick the next element, and store it separately in a **key.**

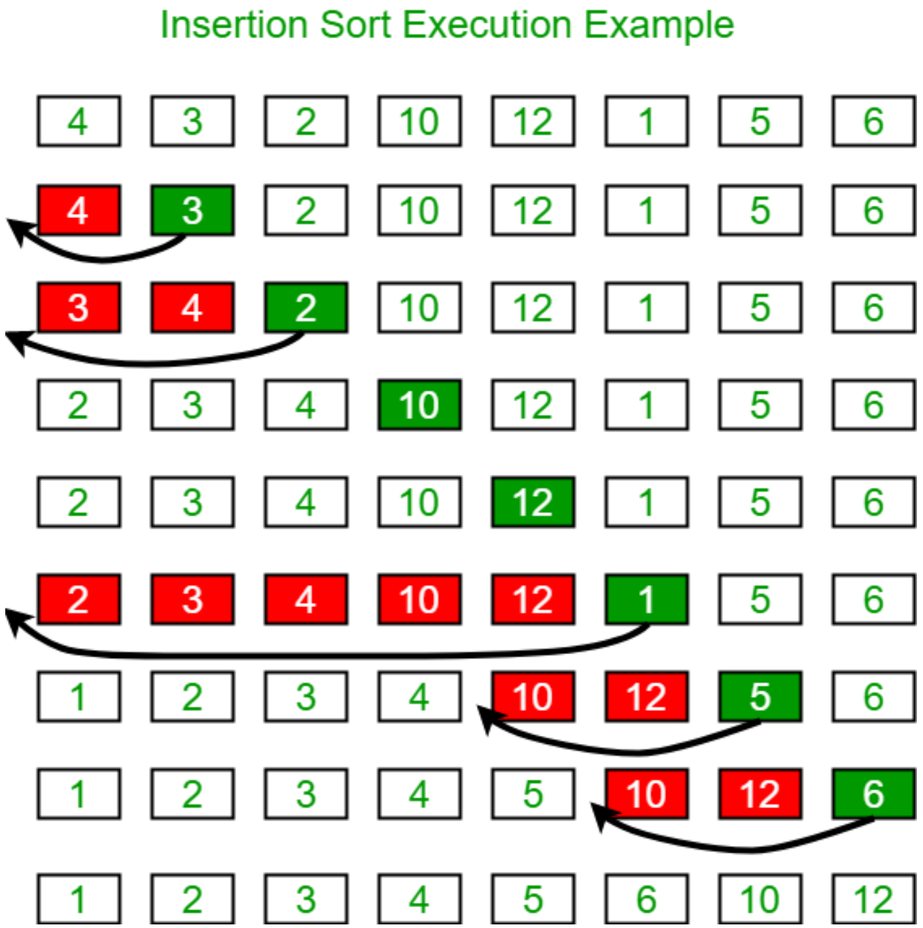
Step3 - Now, compare the **key** with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

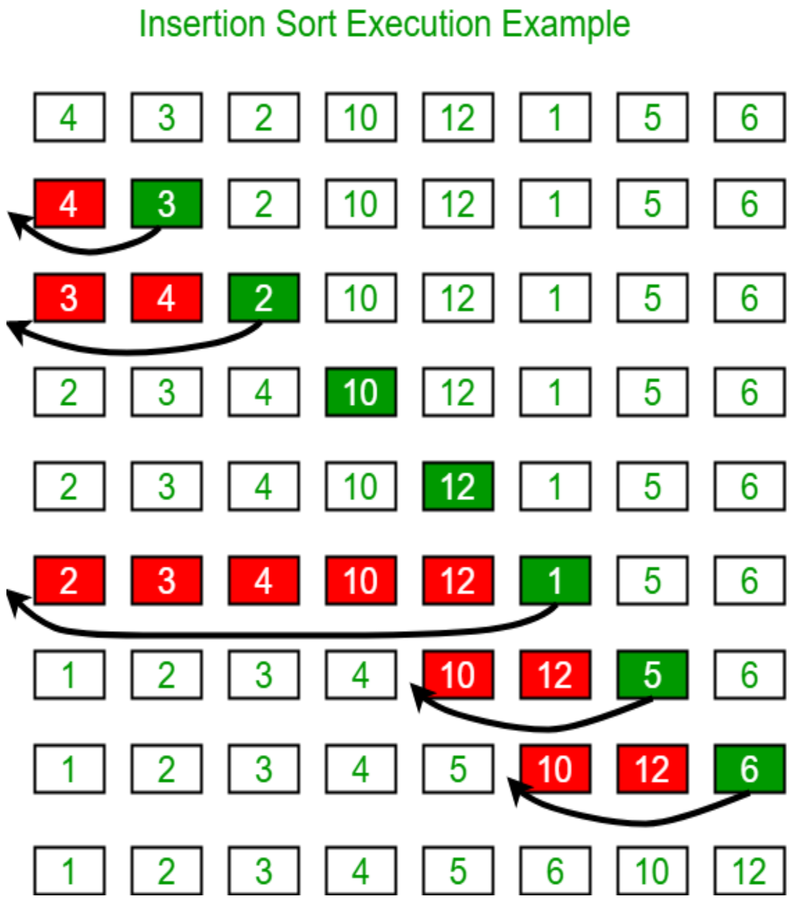
Analysis of Insertion Sort




```
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are
          greater than key, to one position ahead
          of their current position */
        while (j >= 0 && arr[j] > key)
       {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Analysis of Insertion Sort



Best Case (Already Sorted) (Ex- 1,2,3,4,5)

Total Number of Comparison – 1 + 1 + 1 + 1 +… n Times = n

Total Number of Shifting = 0

Total Time = Total Number of Comparison + Total Number of Shifting = 0+n=n

Time Complexity= O(n)

Worst Case (Descending Order) (Ex- 5,4,3,2,1)

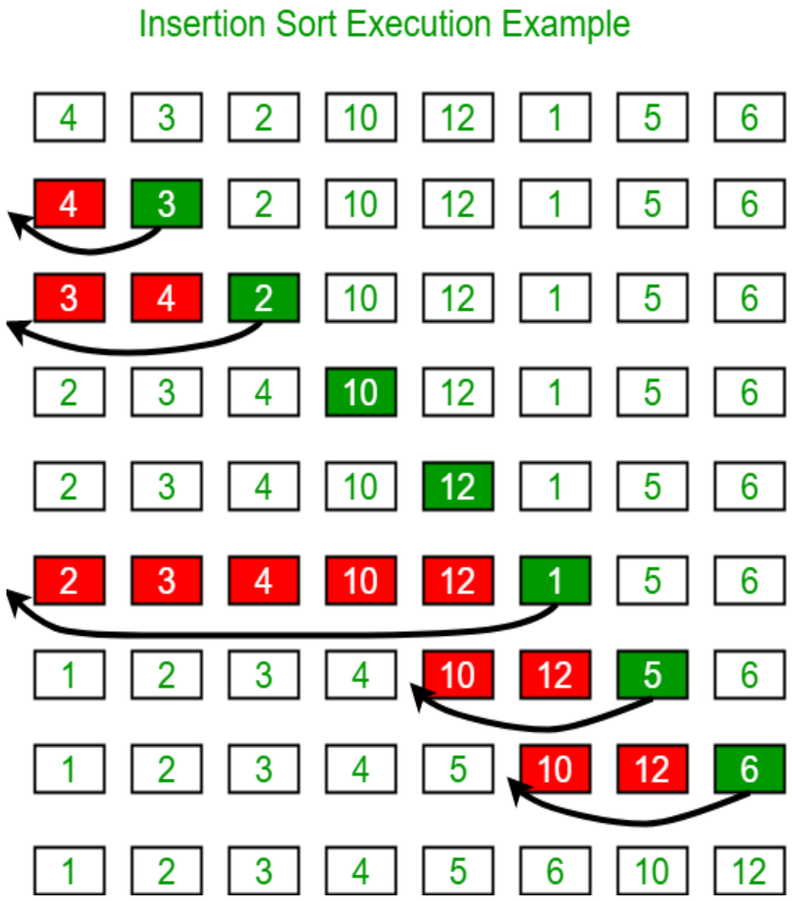
Total Number of Comparison – 1 + 2 + 3 + 4 +… n = n (n+1)/2 =~ n^2

Total Number of Shifting = 1 + 2 + 3 + 4 +… n = n (n+1)/2 =~ n^2

Total Time = Total Number of Comparison + Total Number of Shifting = $n^2+n^2=2n^2$

Time Complexity= $O(n^2)$

Analysis of Insertion Sort



Average Case (Jumbled Order)
(Example of Snapshot)

Total Time = Total Number of Comparison + Total Number of Shifting

Avg Time Complexity= $O(n^2)$

GLA UNIVERSITY
MATHURA
Recognised by UGC Under Section 2(f)
Accredited with A Grade by NAAC

2. Design & Analysis of Shell Sort Algorithm

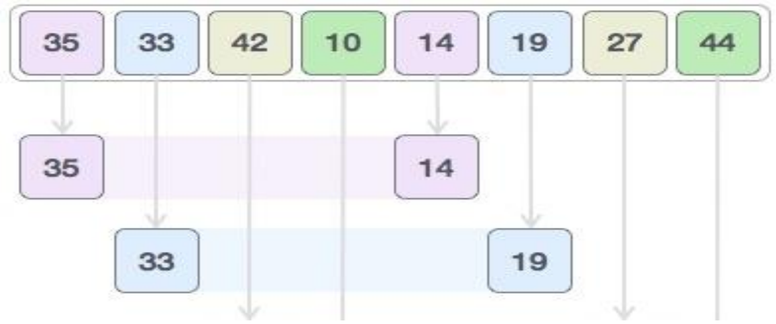
~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

2. Shell Sort Algorithms

- Shell sort is a generalized version of the insertion sort algorithm

- This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

- This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements.



Gap of 4

Understanding Shell Sort Algorithm

We will use very simple technique of N/2, N/4, ...1 to calculate the gaps in our algorithm.

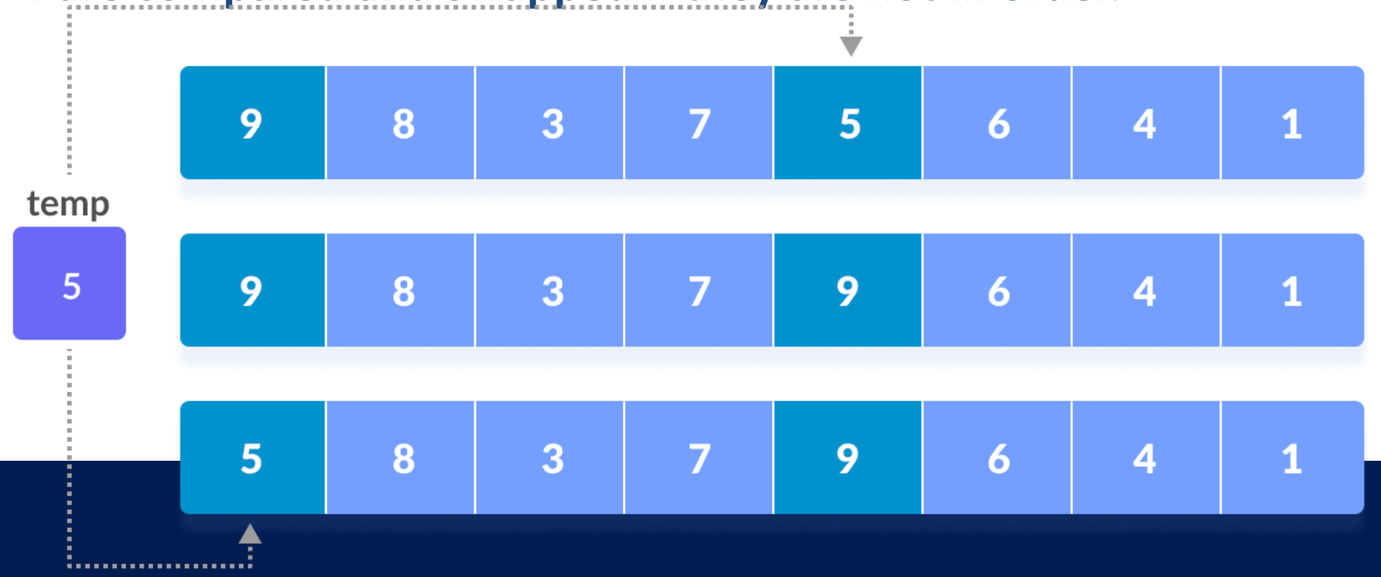
9	8	3	7	5	6	4	1

In Round-1 Loop, if the array size is N = 8 then,

Step-1 the elements lying at the **gap of N/2 = 4 are compared and swapped if they are not in order.**

The 4th element is compared with the 0th element.

If the 4th element is smaller than the 0th one then, swap is performed. i.e. the 4th element is first stored in temp variable and the 0th element (ie. greater element) is stored in the 4th position and the element stored in temp is stored in the 0th position.



Understanding Shell Sort Algorithms

Step-2 This process goes on for all the remaining elements.

Then 5^{th} element is compared with the 1^{st} element.

Then 6^{th}, 7^{th} element is compared with 2^{nd} and 3^{rd} element

Understanding Shell Sort Algorithms

In Round 2 Loop, An gap of N/4 = 8/4 = 2 is taken and again the elements lying at these gaps are sorted.

Then 2nd element is compared with the 0th element.

Then 3rd element is compared with 1st element

Then 4th element is compared with 2nd and 0th element

Then 5th element is compared with 3rd and 1st element and so on…

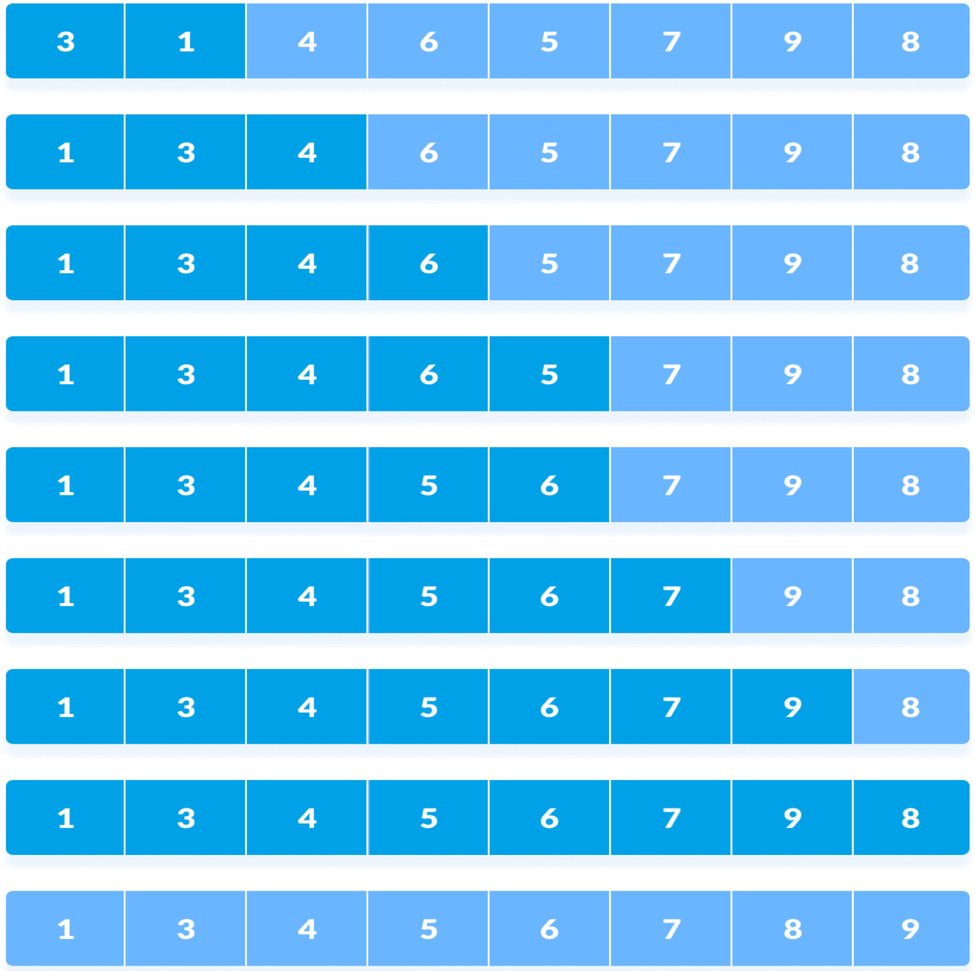
Understanding Shell Sort Algorithms

In Round 3 Loop, Finally, when the gap is N/8 = 8/8 =1 then the array elements lying at the gap of 1 are sorted.

Then 1^{st} element is compared with the 0^{th} element.

Then 2^{nd} element is compared with 1^{st} and 0^{th} element and so on. It will be working as a normal insertion sort.

The array is now completely sorted.



Understanding Shell Sort Algorithm

```
void shellSort( int array[], int n)
{
 for (int gap = n / 2; gap > 0; gap =gap/ 2) // Rearrange elements at each n/2, n/4, n/8, ... gap
 {                                                    // Start with a big gap, then reduce the gap
    for (int i = gap; i < n; i ++)   //Performing Gapped Insertion Sort
    {
      int temp = array[i];
      int j=i-gap;
      while(array [j ] > temp  &&  j >= 0)
      {
        array[j+gap] = array[j ];
         j =j- gap;   // j-=gap
      }
     array[j+gap] = temp;
    }
 }
}
```

```
void insertionSort(int arr[], int n)
{
     int i, key, j;
     for (i = 1; i < n; i++) {
          key = arr[i];
          j = i - 1;

        while (j >= 0 && arr[j] > key)
        {
             arr[j + 1] = arr[j];
             j = j - 1;
         }
          arr[j + 1] = key;
     }
}
```

Complexity	Best Case	Average Case	Worst Case
Time Complexity	$\Omega(n \log(n))$	$\theta(n \log(n)^2)$	$O(n \log(n)^2)$
Space Complexity			$O(1)$

3. Design & Analysis of Quick Sort Algorithm

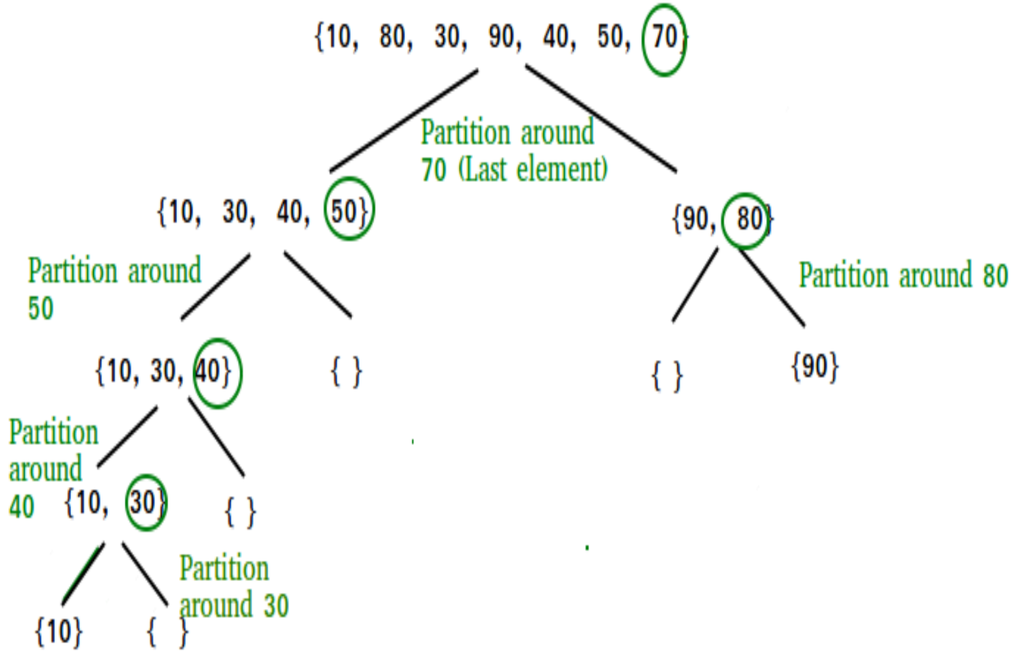
~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

Understanding Quick Sort Algorithm

- Quick sort is based on partitioning of array of data into smaller arrays. (Divide and Conquer Strategy)

- A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

- After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Understanding Quick Sort Algorithm

Choosing the pivot

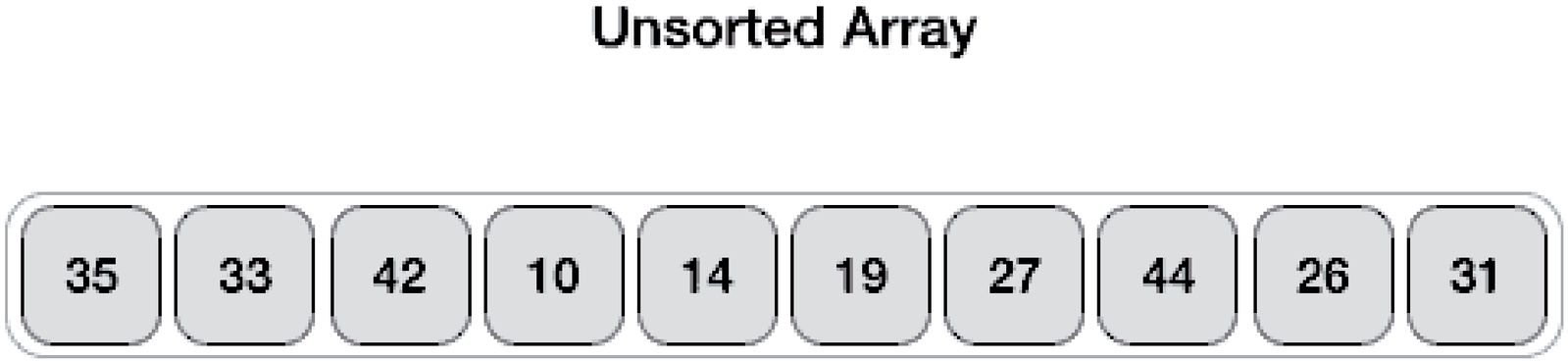
Picking a good pivot is necessary for the fast implementation of quicksort.

Some of the ways of choosing a pivot are as follows –

- Pivot can be random, i.e. select the random pivot from the given array.
- **Pivot can either be the rightmost element of the leftmost element of the given array.**
- Select median as the pivot element.

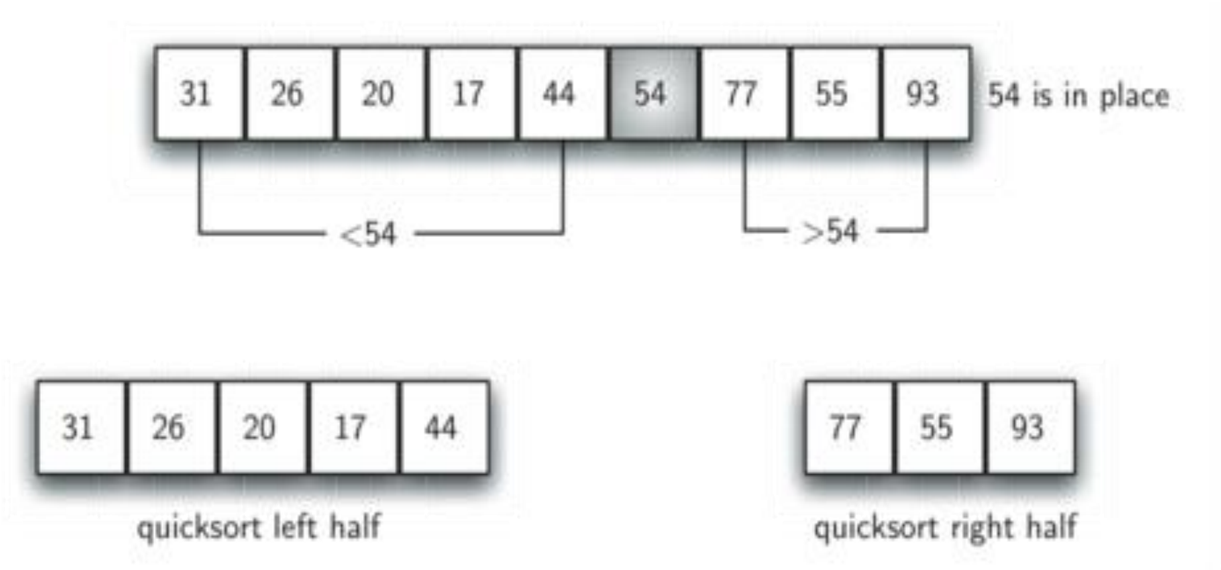
Finding Pivot in Quick Sort Algorithm

Watch this animated image carefully



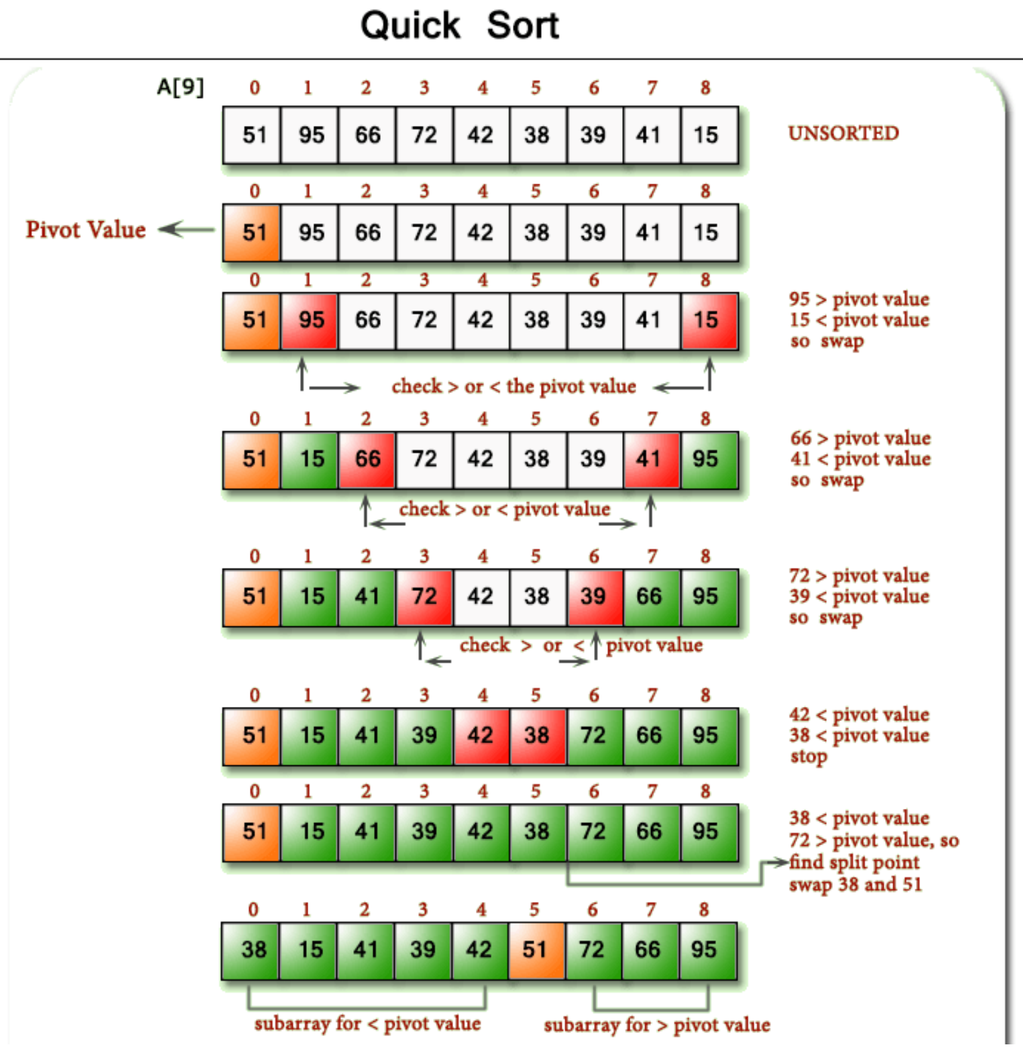
In case, if you still do not understand the concept, then check out slide number 63, and come back again on this slide

Understanding Quick Sort Algorithm




```
QUICKSORT (array A, start, end)
{
  if (start < end)
  {
    p = partition(A, start, end)
   QUICKSORT (A, start, p - 1)
   QUICKSORT (A, p + 1, end)
  }
}
```

Understanding Partition in Quick Sort Algorithm




```
partition(array a, start, end)
{
 pivot=a[start]
 i=start, j=end;
while(i<j)
 {
   do
    {
      i++;
     } while (a[i]<=pivot);
    do
     {
       j--;
      } while (a[j]>=pivot);
    if (i<j)
       swap(a[i], a[j]);
 }
 swap(pivot, a[j]);
 return j;
}
```

Complexity Analysis of Quick Sort Algorithm

```
QUICKSORT (array A, start, end)

{
   if (start < end)
   {
    p = partition(A, start, end)
    QUICKSORT (A, start, p - 1)
    QUICKSORT (A, p + 1, end)
   }
}
```

- Best Case Time Complexity
- Worst Case Time Complexity
- Average Case Time Complexity

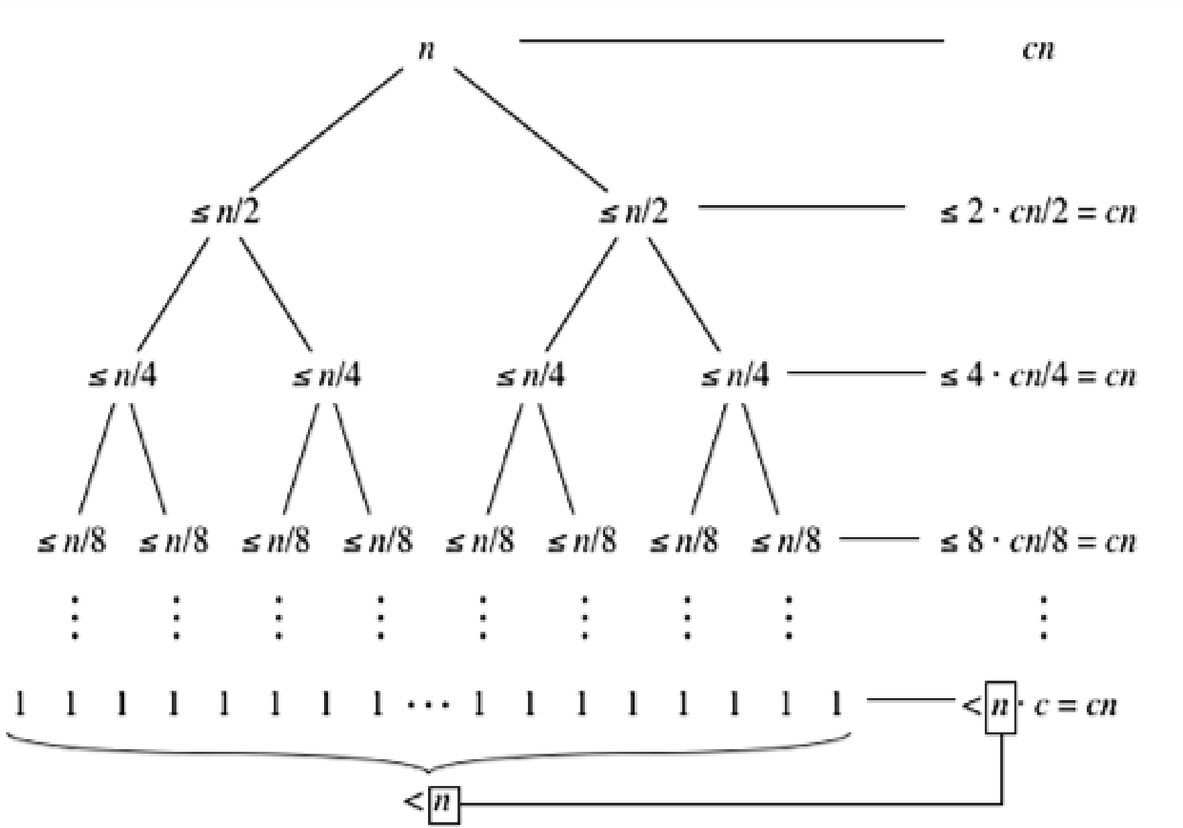
Best Case Complexity Analysis of Quick Sort Algorithm

Pivot is at the Middle of the List

```
QUICKSORT (array A, start, end)

{
  if (start < end)
  {
   p = partition(A, start, end)
   QUICKSORT (A, start, p - 1)
   QUICKSORT (A, p + 1, end)
  }
}
```

This is the Recurrence Tree Method



Best Case Complexity Analysis of Quick Sort Algorithm

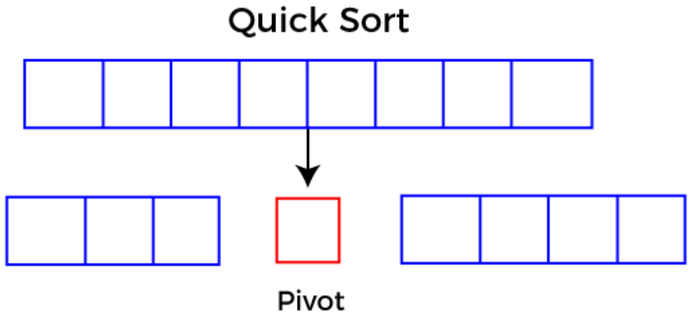
This is the Recurrence Equation Method

```
QUICKSORT (array A, start, end)                    → T(n)

{
  if (start < end)
  {
   p = partition(A, start, end)                    → n
   QUICKSORT (A, start, p - 1)                     → T(n/2)
   QUICKSORT (A, p + 1, end)                       → T(n/2)
  }
}
```

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

Pivot is at the Middle of the List



Best Case Complexity Analysis of Quick Sort Algorithm

Recurrence Relation of Quick Sort

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

$= 1$ for $n=1$

Apply Master Theorem and Solve it

$a=2, b=2$ Calculate $n^{\log_b a} = n^{\log_2 2}$

$= n$

Case 3 is satisfied

Then $T(n) = O(f(n)* \log n)$

$=O(n*\log n)$

Master Method (cookbook method)

$T(n) = aT(n/b) + f(n)$ Calculate $n^{\log_b a}$

Case -1 If $f(n) > n^{\log_b a}$ **Then $T(n) = O(f(n))$**

Case -2 If $f(n) < n^{\log_b a}$ **Then $T(n) = O(n^{\log_b a})$**

Case -3 If $f(n) = n^{\log_b a}$ **Then $T(n) = O(n^{\log_b a} \log n)$ or $O(f(n)* \log n)$**

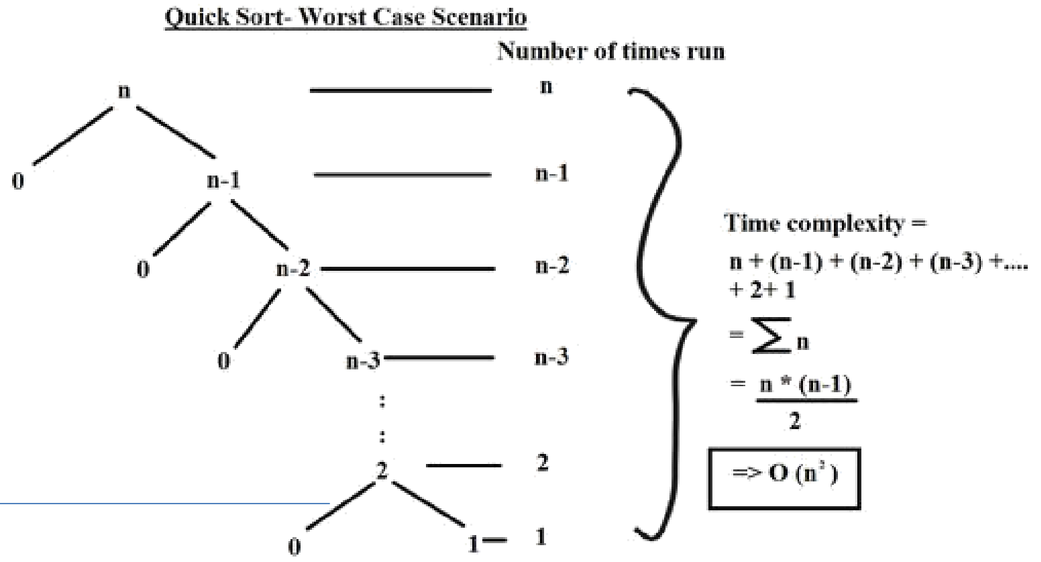
Worst Case Complexity Analysis of Quick Sort Algorithm

When pivot is always at beginning or end of the list

```
QUICKSORT (array A, start, end)                    → T(n)

{
  if (start < end)
  {
    p = partition(A, start, end)                   → n
    QUICKSORT (A, start, p - 1)                    → T(n-1)
    QUICKSORT (A, p + 1, end)                      → T(1)
  }
}
```

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



Worst Case Complexity Analysis of Quick Sort Algorithm

When pivot is always at beginning or end of the list

Recurrence Relation of Quick Sort

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

$= 1$ for $n=1$

Apply Iteration (Back Substitution Method) and Solve it

$T(n)= O(n^2)$

Put n= n-1

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

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

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

Put n= n-2 in the base equation

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

$= T(n-3) + n-2$

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

$= T(n-4)+ n-3 + n-2+ n-1+ n$

Continued for k times

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

Slide Number 22 for reference

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

For n=0, T(0)=1, i.e. n-k=0 -> n=k

$T(n)= T(0) + n-(n+1)+ n-(n-2) + …..n-1 + n$

$= T(0) + 1 + 2+ 3+ ……n-1 + n$

$= T(0) + n(n+1)/2$

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

$= O(n^2)$

$T(n)= O(n^2)$

Average Case Complexity Analysis of Quick Sort Algorithm

```
QUICKSORT (array A, start, end)          → T(n)

{
  if (start < end)
  {
   p = partition(A, start, end)           → n
   QUICKSORT (A, start, p - 1)            → T(n/4)
   QUICKSORT (A, p + 1, end)              → T(3n/4)
  }
}
```

- Pivot can be chosen at any place from array.
- Average time can be calculated by considering the pivot at all possible permutation of array and calculate time taken by every permutation
- We can get an idea of average case by considering the case when partition puts O(n/4) elements in one set and O(3n/4) elements in other set.
- Following is recurrence for this case.

  $T(n) = T(n/4) + T(3n/4) + n$

  $T(n) = O(n*\log n)$

Summary Analysis of Quick Sort Algorithm

Best Case = O(n logn)

Worst Case = $O(n^2)$

Average Case= O(n logn)

Quick sort is **not a stable** algorithm

Quick sort is an **inplace** algorithm which means the numbers are all sorted within the original array itself.

4. Design and Analysis of Merge Sort Algorithm

~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

Understanding Merge Sort Algorithm

- The given unsorted array with **n** elements, is divided into **n** subarrays, each having **one** element, because a single element is always sorted in itself.

- Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.

The concept of Divide and Conquer involves three steps:

1.Divide the problem into multiple small problems.

2.Conquer the sub-problems by solving them. The idea is to break down the problem into atomic sub-problems, where they are actually solved.

3.Combine the solutions of the sub-problems to find the solution of the actual problem.

Watch this animated image carefully

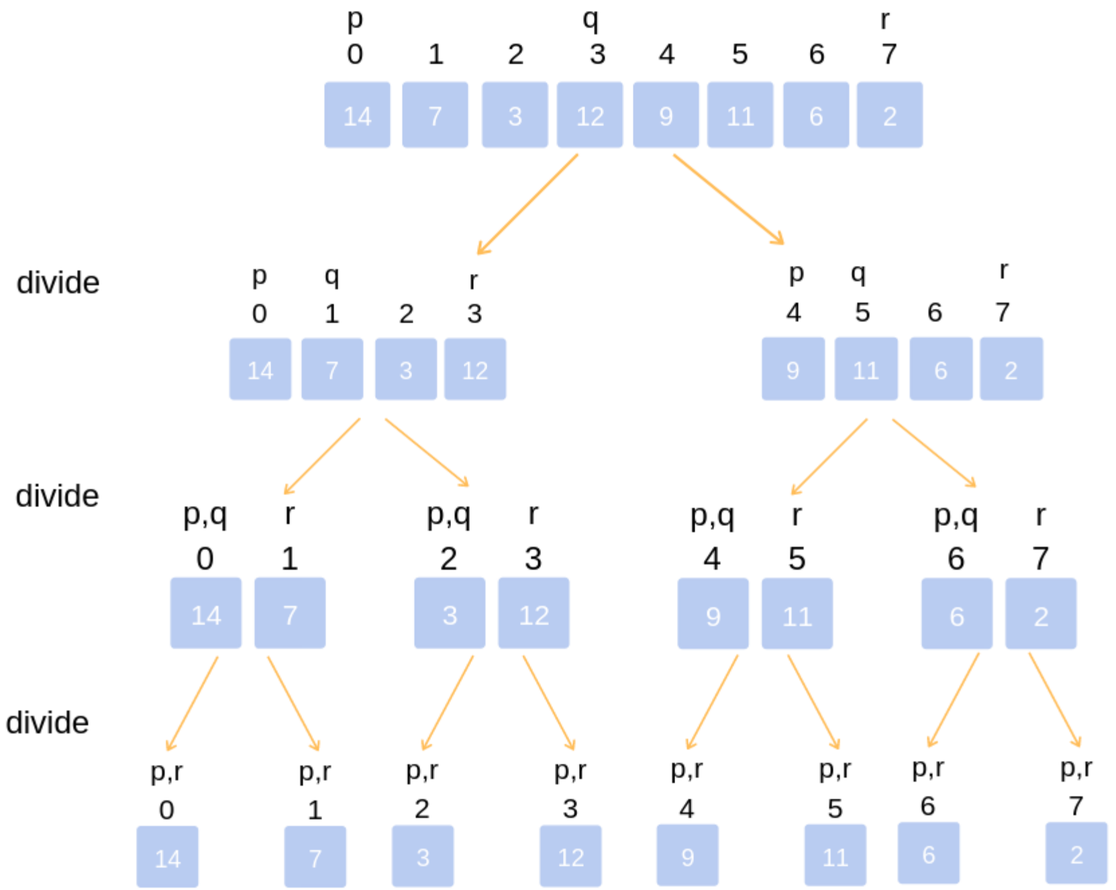
5	3	7	1	0	8	5

mergesort([5, 3, 7, 1, 0, 8, 5])

Understanding Merge Sort Algorithm

Step 1

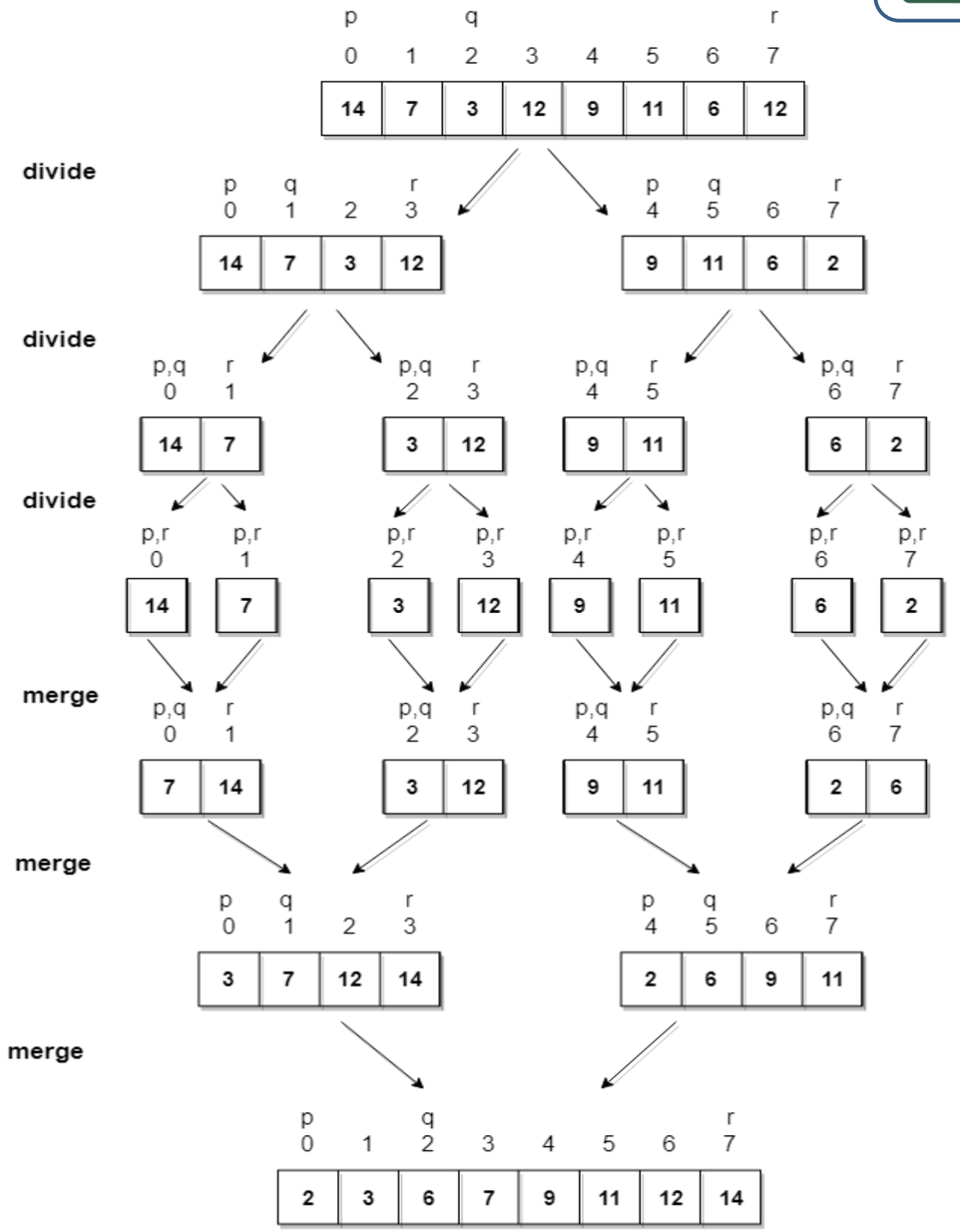
Divide the unsorted list into n sub-lists, each comprising 1 element (a list of 1 element is supposed sorted).



Understanding Merge Sort Algorithm

Step 2

Repeatedly merge sub-lists to produce newly sorted sub-lists until there is only 1 sub-list remaining. This will be the sorted list.



~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

Understanding Merge Sort Algorithm

In merge sort we follow the following steps

1. We take a variable p and store the starting index of our array in this. And we take another variable r and store the last index of array in it.

2. Then we find the middle of the array using the formula (p + r)/2 and mark the middle index as q, and break the array into two subarrays, from p to q and from q + 1 to r index.

3. Then we divide these 2 subarrays again, just like we divided our main array and this continues.

4. Once we have divided the main array into subarrays with single elements, then we start merging the subarrays.

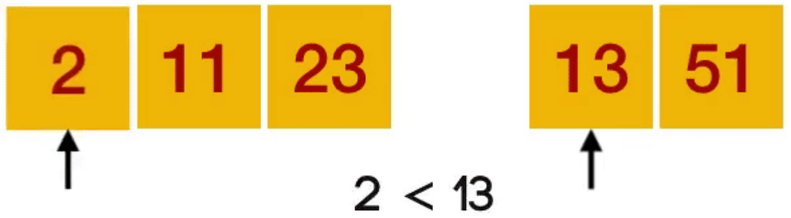
```
void mergeSort(int a[], int
p, int r)
{
int q;
if(p < r)
 {
    q = (p + r) / 2;
    mergeSort(a, p, q);
    mergeSort(a, q+1, r);
     merge(a, p, q, r);
  }
}
```

Understanding Designing of Merge Sort Algorithm

```
void merge(low, mid, high)
{
   i=low
   j= mid+1
   k= high
   while (i < =mid && j <= high)
    {
        if (a[i]<=a[j])
         {
          temp[k] = a [i];
         i++, k++;
        }
       else {
          temp[k] = a[j];
          j++, k++;
       }
   }
```

```
while (i <= mid)  // Copy the remaining elements of the list if there are any
   {
       temp[k] = a[i];
      i++;
      k++;
   }
   while (j <= high) {
      temp[k] = a[j];
      j++;
      k++;
   }
 for(k=low; k<high; k++)
 {
  a[k] = temp[k];
 }
}
```

Understanding merge with animated image



Understanding Time Complexity of Merge Sort Algorithm

```
void mergeSort(int a[], int p, int r)
{
int q;
if(p < r)
 {
    q = (p + r) / 2;
    mergeSort(a, p, q);
    mergeSort(a, q+1, r);
     merge(a, p, q, r);
  }
}
```

mergeSort(a, p, q); → T(n/2)

mergeSort(a, q+1, r); → T(n/2)

merge(a, p, q, r); → n

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

Understanding Time Complexity of Merge Sort Algorithm

Recurrence Relation

T(n)= 2T(n/2) + n for n>1

= 1 for n=1

Time complexity of Merge Sort is O(nLogn) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Sorting In Place: No

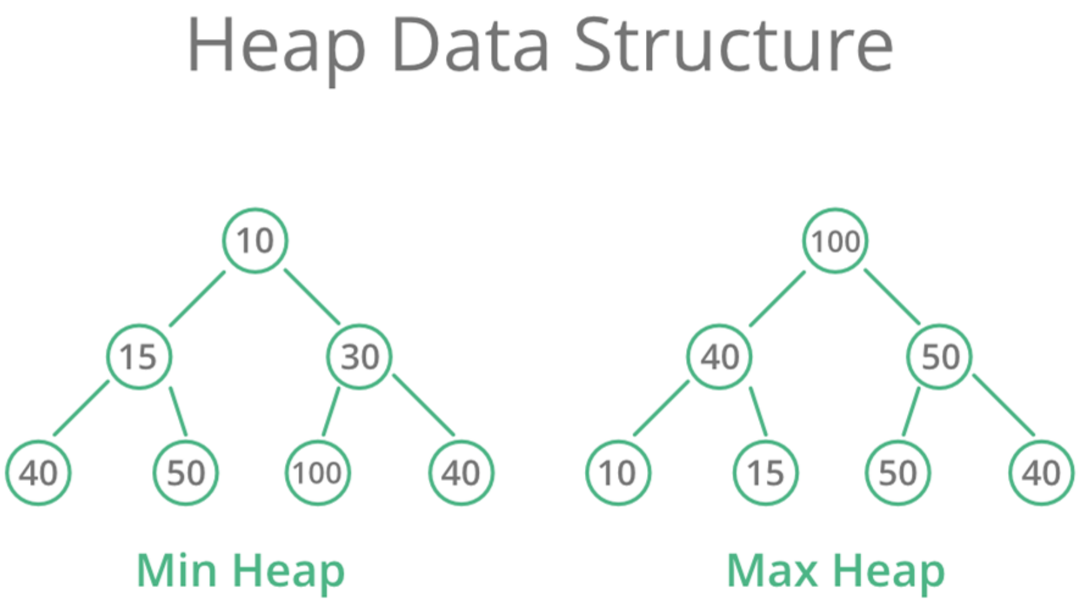
Stable: Yes

5. Design and Analysis of Heap Sort Algorithm

~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

Understanding Designing of Heap Sort Algorithm

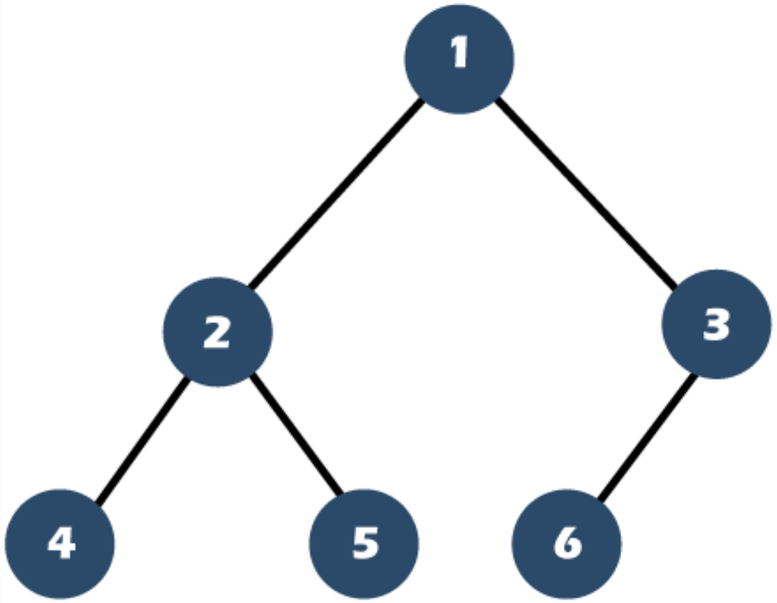
- Heap sort is a comparison-based sorting technique based on **Binary Heap data structure**.
- A Binary Heap is a **Complete Binary Tree** where items are stored in a special order such that the value in a parent node is greater (or smaller) than the values in its two children nodes. The former is called max heap and the latter is called min-heap.
- It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning.
- We repeat the same process for the remaining elements.



Understanding Complete Binary Tree

Complete Binary Tree

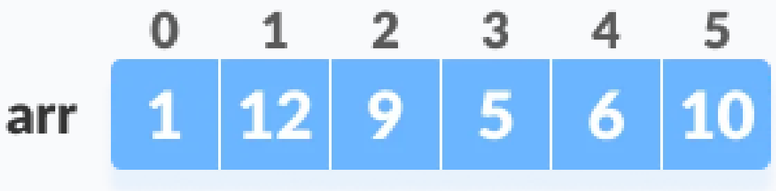
- A binary tree is said to be a complete binary tree when all the levels are completely filled except the last level, which is filled from the left.
- it can be easily represented as an array.
- If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and the right child by 2 * I + 2 (assuming the indexing starts at 0).



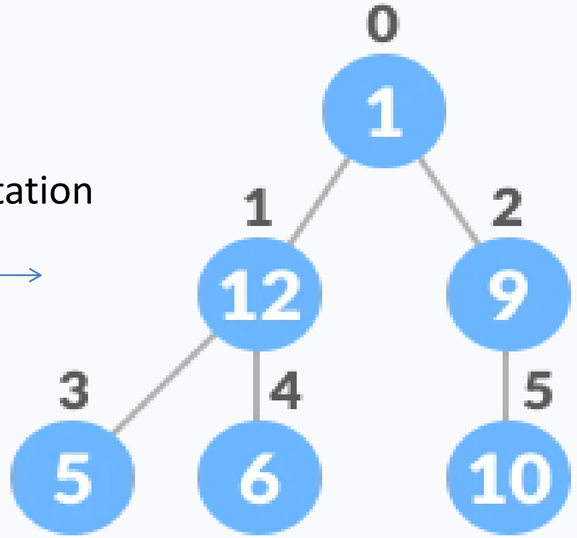
Understanding Designing of Heap Sort Algorithm

Example

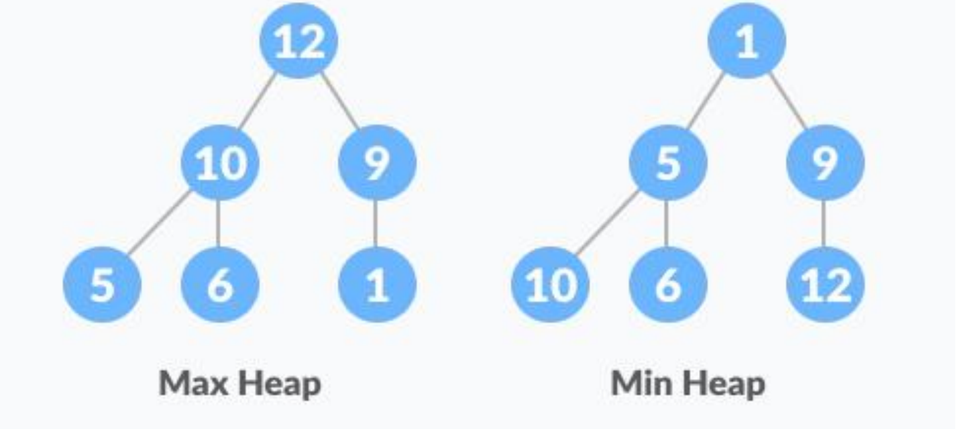
Initial List of elements



Complete Binary Tree Representation



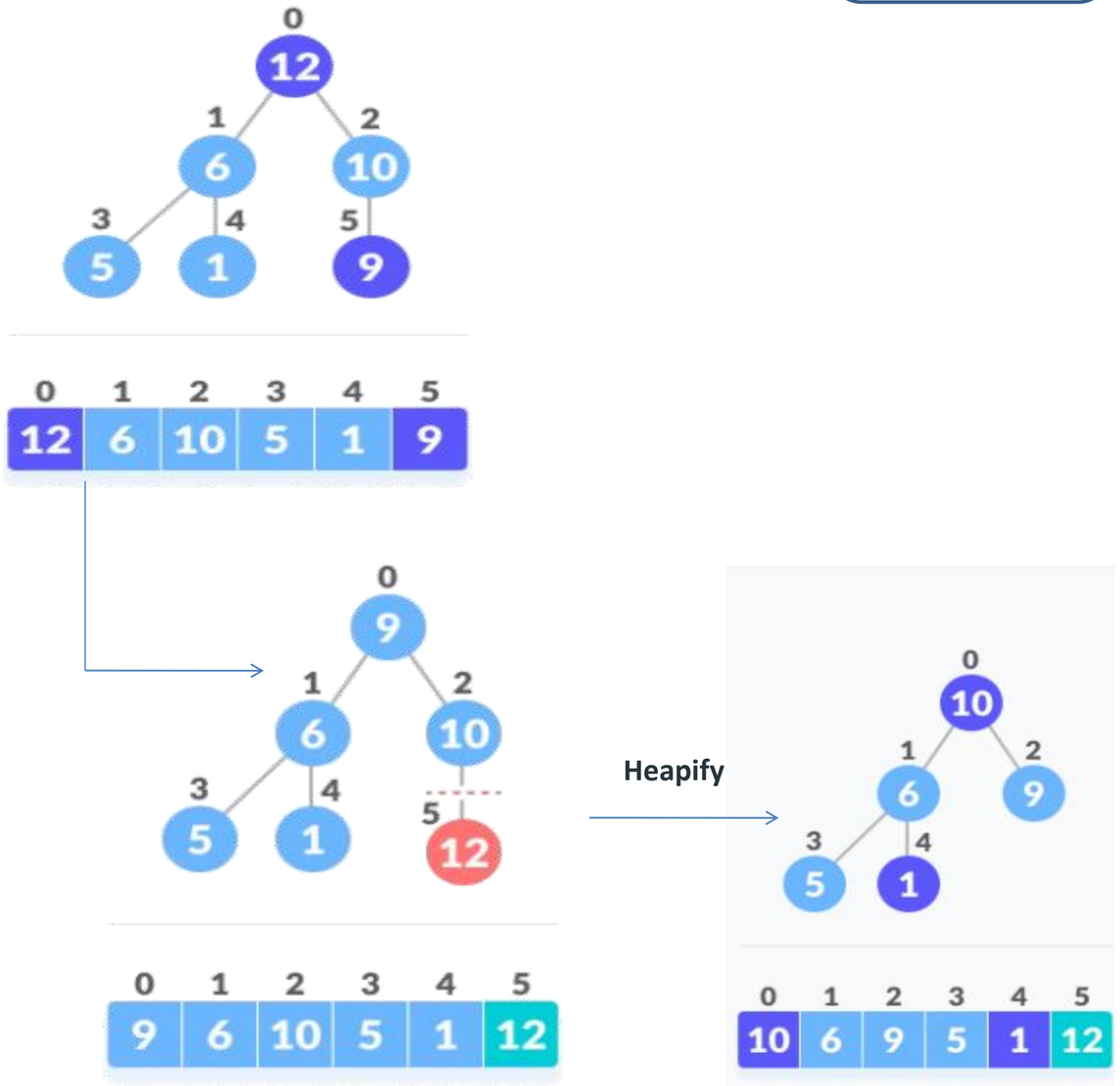
Apply **heapify()** method to build Max or min Heap



Understanding Designing of Heap Sort Algorithm

Heap Sort Algorithm for sorting in **Increasing Order**

1. The tree satisfies Max-Heap property, the largest item is stored at the root node.
2. **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.
3. **Remove:** Reduce the size of the heap by 1.
4. **Heapify:** Heapify the root element again so that we have the highest element at root.
5. The process is repeated until all the items of the list are sorted.



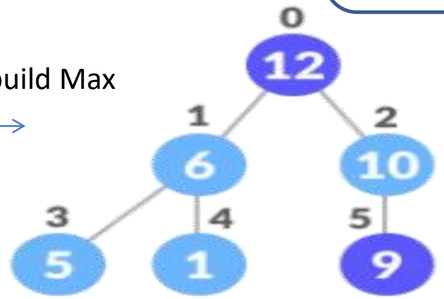
Note: heapify function is used to build or construct the heap.

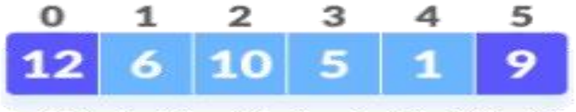
Understanding Time Complexity of Heap Sort Algorithm

```
//Build max heap
for (i = size / 2 - 1; i >= 0; i-
heapify(arr, size, i);

// Heap sort
for (int i = n - 1; i >= 0; i--)
 {
   swap(&arr[0], &arr[i]);
  // Heapify root element to get
     highest element at root again
   heapify(arr, i, 0);
 }
```

heapify() method to build Max





Heap Sort has O(nlog n) time complexities for all the cases (best case, average case, and worst case).

1. Deleting an root element and performing heapify method takes logn unit of times
2. Deleting all n elements will take total time n*logn times

Note: Stable Algorithm: No

In-place: Yes

Comparison of Sorting Algorithms

Sorting Algorithms	Time Complexity		
	Best Case	Average Case	Worst Case
Bubble Sort	Ω(N)	Θ(N^2)	O(N^2)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)
Quick Sort	Ω(N log N)	Θ(N log N)	O(N^2)
Merge Sort	Ω(N log N)	Θ(N log N)	O(N log N)
Heap Sort	Ω(N log N)	Θ(N log N)	O(N log N)

Liner Sorting Algorithm
Count Sort

~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

Understanding Count Sort Algorithm

- Counting Sort Algorithm is an efficient sorting algorithm that is based on non comparison based strategy and can be used for sorting elements within a specific range.

- Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array.

- The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

Input Data

0	4	2	2	0	0	1	1	0	1	0	2	4	2

Count Array

0	1	2	3	4
5	3	4	0	2

Sorted Data

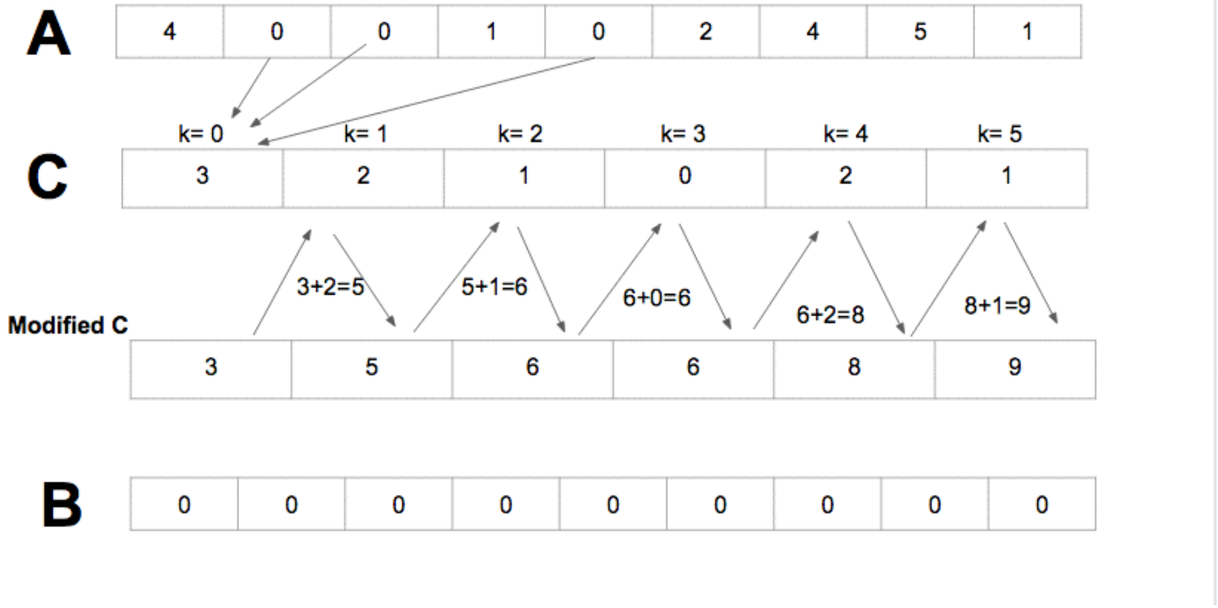
0	0	0	0	0	1	1	1	2	2	2	2	4	4

Understanding Count Sort Algorithm

Counting Sort Algorithm

```
countingSort(array, size)
{
    max <- find largest element in array
    initialize count array with all zeros
     for j <- 0 to size
     find the total count of each unique element
     and store the count at jth index in count array
     for i <- 1 to max
     find the cumulative sum and store it in count array itself
     for j <- size down to 1
     restore the elements to array
     decrease count of each element restored by 1
}
```

Watch this animated image carefully



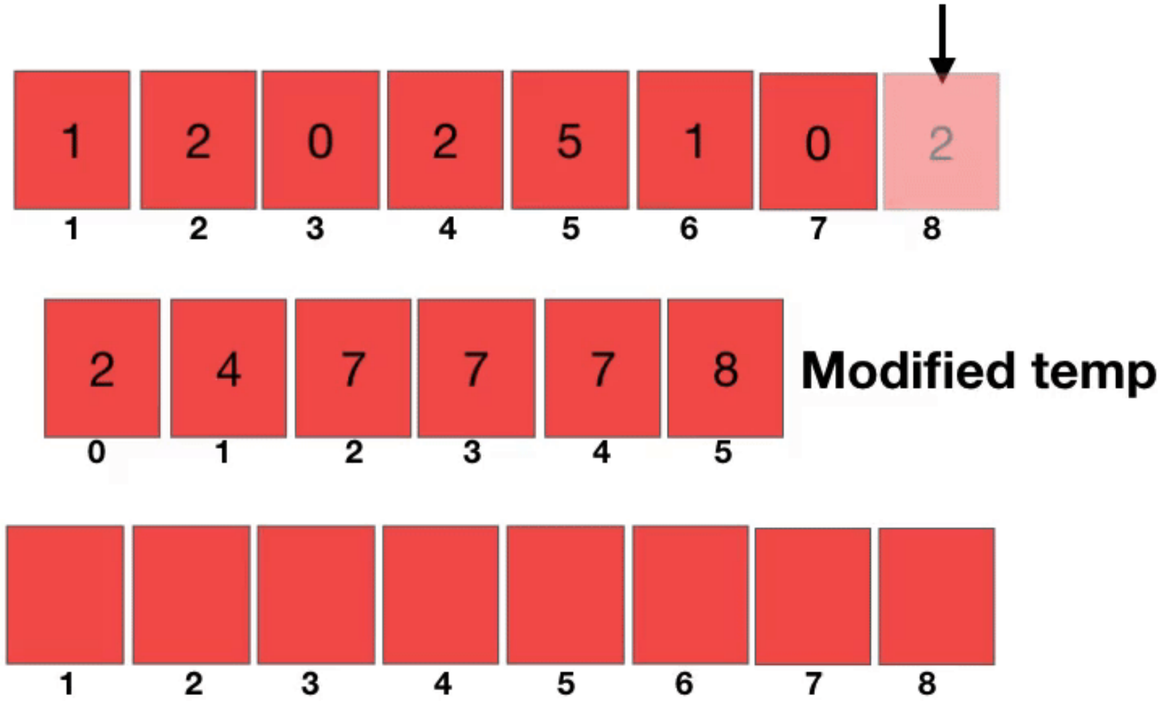
If you do not understand, then do not worry, visit next slide and watch another animated image

Understanding Count Sort Algorithm

Counting Sort Algorithm

```
countingSort(array, size)
{
    max <- find largest element in array
    initialize count array with all zeros
     for j <- 0 to size
     find the total count of each unique element
     and store the count at jth index in count array
     for i <- 1 to max
     find the cumulative sum and store it in count array itself
     for j <- size down to 1
     restore the elements to array
     decrease count of each element restored by 1
}
```

Watch this animated image carefully



Time Complexity= O(n) in all cases

Counting sort is a stable algorithm because the relative order of similar elements are not changed after sorting.

Searching

DFS and BFS

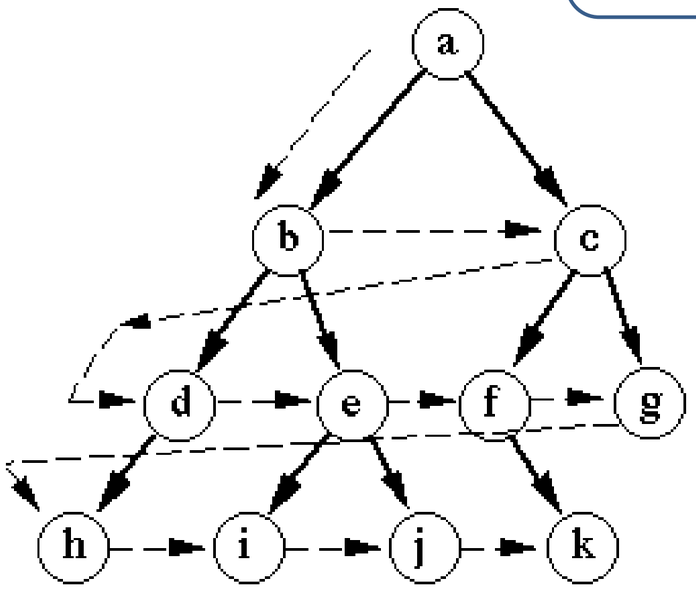
~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

Understanding BFS

- **BFS** stands for **Breadth First Search** is a vertex based technique for finding a shortest path in graph.

- It uses a Queue data structure to remember to get the next vertex to start a search when a dead end occurs in any iteration.

- In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue.



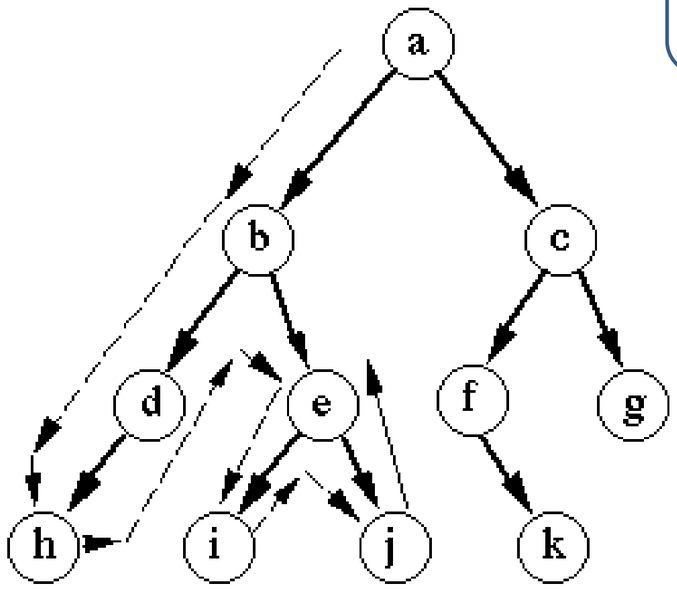
Breadth-first search

Output is: a, b, c, d, e, f, g, h, i,, j, k

Understanding DFS

- **DFS** stands for **Depth First Search** is a edge based technique.

- It uses the Stack data structure to remember to get the next vertex to start a search when a dead end occurs in any iteration



Depth-first search

Output is: a, b, d, h, e, i, j, c, f,, k, g

Understanding the difference between BFS and DFS

Sr. No.	Key	BFS	DFS
1	Definition	BFS, stands for Breadth First Search.	DFS, stands for Depth First Search.
2	Data structure	BFS uses Queue to find the shortest path.	DFS uses Stack to find the shortest path.
3	Source	BFS is better when target is closer to Source.	DFS is better when target is far from source.
4	Suitablity for decision tree	As BFS considers all neighbor so it is not suitable for decision tree used in puzzle games.	DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision. If we reach the conclusion, we won.
5	Speed	BFS is slower than DFS.	DFS is faster than BFS.
6	Time Complexity	Time Complexity of BFS = O(V+E) where V is vertices and E is edges.	Time Complexity of DFS is also O(V+E) where V is vertices and E is edges.

Advance Data Structure

Red Black Tree and its Properties

~Dr Gaurav Kumar, Asst. Prof, CEA, GLA

Understanding Red Black Tree

• A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black).

• These colors are used to ensure that the tree remains balanced during insertions and deletions.

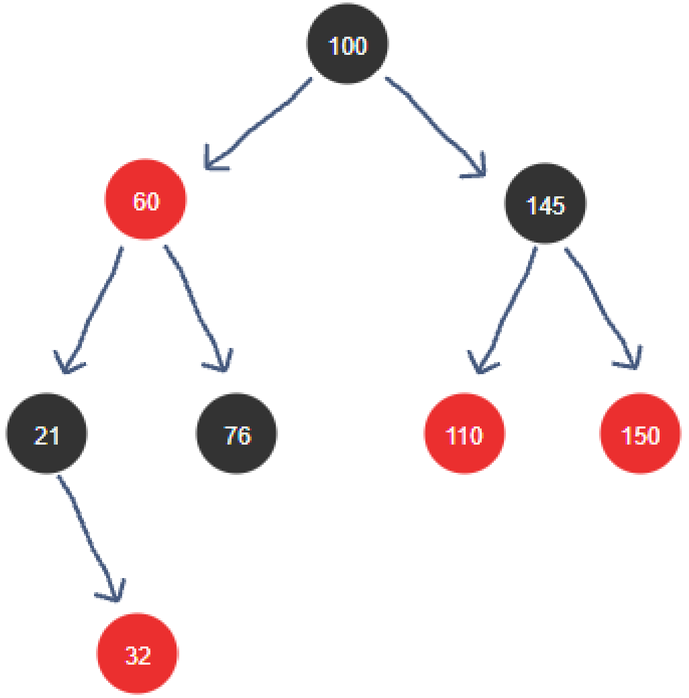
• The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves frequent insertions and deletions, then Red-Black trees should be preferred.

Understanding Rules or Properties of Red Black Tree

- Every node has a color either red or black.
- The root of the tree is always black.
- There are no two adjacent red nodes

(A red node cannot have a red parent or red child).

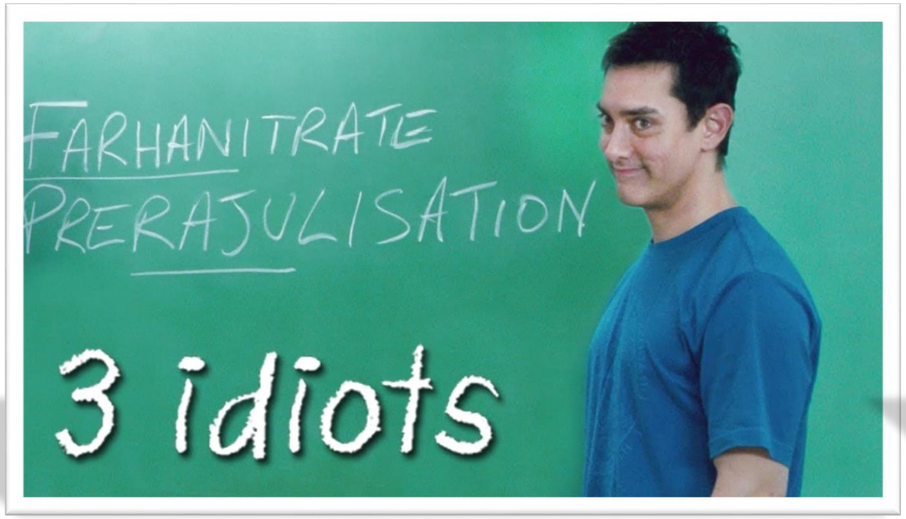
- Every path from a node (including root) to any descendants NULL nodes has the same number of black nod



A correct red-black tree.

Path 1 : 100 → 60 → 21→ 32
Path 2: 100→ 60 → 76 and so on..

Choose Wisely



Happy Learning!

If you have any doubts, or queries , can be discussed in the C-11, Room 310, AB-1.

or share it on WhatsApp 8586968801

if there is any suggestion or feedback about slides, please write it on gaurav.kumar@gla.ac.in

~Dr Gaurav Kumar, Asst. Prof, CEA, GLA