Design and Analysis of Algorithm

KCS503

Instructor: Md. Shahid

DETAILED SYLLABUS		3-1-0
Unit	**Topic**	**Proposed Lecture**
I	**Introduction:** Algorithms, Analyzing Algorithms, Complexity of Algorithms, Growth of Functions, Performance Measurements, Sorting and Order Statistics - Shell Sort, Quick Sort, Merge Sort, Heap Sort, Comparison of Sorting Algorithms, Sorting in Linear Time.	**08**
II	**Advanced Data Structures:** Red-Black Trees, B – Trees, Binomial Heaps, Fibonacci Heaps, Tries, Skip List	**08**
III	**Divide and Conquer** with Examples Such as Sorting, Matrix Multiplication, Convex Hull and Searching. **Greedy Methods** with Examples Such as Optimal Reliability Allocation, Knapsack, Minimum Spanning Trees – Prim's and Kruskal's Algorithms, Single Source Shortest Paths - Dijkstra's and Bellman Ford Algorithms.	**08**
IV	**Dynamic Programming** with Examples Such as Knapsack. All Pair Shortest Paths – Warshal's and Floyd's Algorithms, Resource Allocation Problem. Backtracking, Branch and Bound with Examples Such as Travelling Salesman Problem, Graph Coloring, n-Queen Problem, Hamiltonian Cycles and Sum of Subsets.	**08**
V	**Selected Topics:** Algebraic Computation, Fast Fourier Transform, String Matching, Theory of NP-Completeness, Approximation Algorithms and Randomized Algorithms	**08**

Text books:

1. Thomas H. Coreman, Charles E. Leiserson and Ronald L. Rivest, "Introduction to Algorithms", Printice Hall of India.
2. E. Horowitz & S Sahni, "Fundamentals of Computer Algorithms",
3. Aho, Hopcraft, Ullman, "The Design and Analysis of Computer Algorithms" Pearson Education, 2008.
4. LEE "Design & Analysis of Algorithms (POD)",McGraw Hill
5. Richard E.Neapolitan "Foundations of Algorithms" Jones & Bartlett Learning
6. Jon Kleinberg and Éva Tardos, Algorithm Design, Pearson, 2005.

Preface

Dear AKTU University Students,

I am excited to present these comprehensive notes for the "Design and Analysis of Algorithms" course tailored specifically for your academic journey. These notes aim to serve as a valuable companion, offering clear explanations, insightful examples, and practical insights to aid your understanding of algorithmic principles. Whether you're preparing for exams or deepening your grasp of key concepts, these notes are crafted to enhance your learning experience. Wishing you a successful and enriching exploration of algorithm design.

Best regards,

Md shahid (Assistant professor, MIET, MEERUT)

2^{nd} edition

Year-2023

The author can be reached at shahid.55505@gmail.com.

Unit-01

Algorithm—It is a combination of a sequence of finite steps to solve a computational problem.

Program— A program, on the other hand, is a concrete implementation of an algorithm using a programming language.

Properties of Algorithm

- It should terminate after a finite time.
- It should produce at least one output.
- It should take zero or more input externally.
- It should be deterministic (unambiguous).
- It should be language independent.

Example to differentiate between algorithm and program

Add()
{

1. input two numbers a and b
2. sum a and b and store the result in c
3. return c

}

The above example is an algorithm as it follows its properties.

```
While(1)
    {
      printf("MIET");
    }
```

The above example is not an algorithm as it will never terminate. Though it is a program.

The main algorithm design techniques

1. Divide and conquer technique
2. Greedy technique
3. Dynamic programming
4. Branch and bound
5. Randomized
6. Backtracking

Note— The most basic approach to designing algorithms is the brute force technique, where one attempts all possible solutions to a problem and opts for the successful one. All computational problems can be solved through the brute force method, though often not achieving noteworthy efficiency in terms of space and time complexity.

For example, search for an element in a sorted array of elements using linear search.

Steps required to design an algorithm

1.**Problem Definition**: Clearly understand the problem you need to solve. Define the input, output, constraints, and objectives of the algorithm.

2.**Design Algorithm**: Choose an appropriate algorithmic technique based on the nature of the problem, such as greedy, divide and conquer, dynamic programming, etc.

3.**Draw Flowchart**: Create a visual representation of your algorithm using a flowchart. The flowchart helps to visualize the logical flow of steps.

4.**Validation**: Mentally or manually walk through your algorithm with various inputs to verify its correctness. Ensure it produces the expected results.

5.**Analyze the Algorithm**: Evaluate the efficiency of the algorithm in terms of time complexity (how long it takes to run) and space complexity (how much memory it uses).

6.**Implementation (Coding)**: Translate your algorithm into actual code using a programming language of your choice. Write clean, well-organized code that follows best practices.

Q. Define 'algorithm,' discuss its main properties, and outline the steps for designing it.

Q. What do you mean by an algorithm, and what are the main algorithm design techniques?

Analysis of Algorithms

The efficiency of an algorithm can be analyzed at two different stages: before implementation (A Priori Analysis) and after implementation (A Posteriori Analysis).

A Priori Analysis— This is a theoretical analysis of an algorithm's efficiency before it's actually implemented. The analysis assumes that certain factors, such as processor speed and memory, remain constant and do not affect the algorithm's performance. It involves evaluating an algorithm based on its mathematical characteristics, such as time complexity (Big O notation) and space complexity. It provides insights into how an algorithm will perform under ideal conditions and helps in comparing different algorithms in terms of their theoretical efficiency.

A Posteriori Analysis— This is an empirical analysis that occurs after an algorithm has been implemented in a programming language and executed on an actual computer. The algorithm is tested and executed on a target machine, and actual statistics like running time and space required are collected. A Posteriori Analysis provides a more realistic view of how an algorithm performs in a real-world setting, considering hardware characteristics, compiler optimizations, and other factors. It helps validate the theoretical analysis and may reveal unexpected performance issues or bottlenecks.

Note—Writing a computer program that handles a small set of data is entirely different from writing a program that takes a large number of input data. The program written to handle a big number of input data must be algorithmically efficient in order to produce the result in reasonable time and space.

Asymptotic Analysis of Algorithms

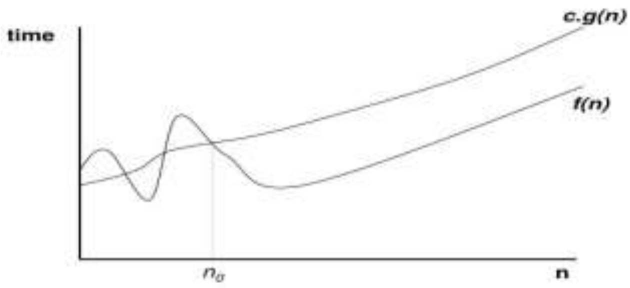
Asymptotic analysis of algorithms is a method used to analyze the efficiency and performance of algorithms as the input size grows towards infinity. It focuses on understanding how an algorithm's performance scales with larger inputs and provides a way to express the upper (worst-case) and lower bounds (best-case) on its execution time or space complexity. The primary goal of asymptotic analysis is to identify the algorithm's growth rate, which helps in making comparisons between different algorithms and determining their suitability for various problem sizes.

Asymptotic Notations:

1. **O** [Big-oh] (upper bound)
2. **Ω** [Big-omega] (lower bound)
3. **ϴ** [Big-theta] (tight bound)
4. **o** [small-oh] (Not tightly upper bound)
5. **w** [small omega] (Not tightly lower bound)

1. **Big-oh Notation (O)** : It is used to describe the upper bound or worst-case performance of an algorithm in terms of its time complexity or space complexity. It provides an estimate of the maximum amount of time or space an algorithm can require for its execution as the input size increases.

   Note: Most of the time, we are interested in finding only the worst-case scenario of an algorithm (worst case time complexity). Big O notation allows for a high-level understanding of how an algorithm's efficiency scales without getting into specific constants or lower-order terms.



$f(n) = O(g(n))$

We say that

f(n) = O g(n) if and only if

f(n) <= c . g(n) for some c >0 after n >= n_o >=0

Question: Find out upper bound for the function f(n) = 3n+2.

Solution:

Steps:

1. We know that definition of upper bound is f(n) <= c. g(n).
2. f(n) = 3n + 2 (Given)
3. We need to find out c and g(n).
4. If we choose c=5 and g(n) = n, then 3n+2 <= 5*n.
5. For c=5 and n_0 = 1 (starting value of "n"), f(n)<=c. g(n).
6. Therefore, f(n) = O(n)

Note: Other functions that are larger than f(n), such as n^2, n^3, nlogn, etc., will also serve as upper bounds for the function f(n). However, we typically consider only the closest upper bound among all possible candidates, as the remaining upper bounds are not as useful.

1. Given f(n) = $2n^2$ + 3n + 2 and g(n) = n^2 , prove that f(n) = O g(n).

Solution:

Steps:

1. We have to show $f(n) <= c. g(n)$ where c and n_0 are some positive constants for all n which is >= n_0
2. Now, find out the value of c and n_0 such that the equation-(1) gets true.

$$2n^2 + 3n + 2 <= c. (n^2) \text{ ....(1)}$$

If we put c = 7 (note— we can take any positive value for c), then

$2n^2 + 3n + 2 <= 7 n^2$

Now, put n=1 which is n_0 (starting value for input n)

7 <= 7 [True]

Hence, when c=7 and $n_0 = 1$, f(n) = O g(n) for all n which is > = n_0

2. Given $f(n) = 5n^2 + 6n + 4$ and $g(n) = n^2$, then prove that f(n) is $O(n^2)$.

Solution:

f(n) will be $O(n^2)$ if and only if the following condition holds good:

$f(n) <= C. g(n)$ where C is some constant and $n>=n_0 >=0$

$5n^2 + 6n + 4 < = C. n^2$

If we put C=15 and $n_0 = 1$, then we get

15 <= 15 (which is true.)

It means f(n) is $O(n^2)$ where C=15 , and $n_0 =1$

Note: We have to find out c and n_0 (starting value of input n) to solve such a question.

3. Solve the function $f(n) = 2^n + 6n^2 + 3n$ and find the big-oh (O) notation.

Steps:

1. Find out the greatest degree of "n" from f(n), which is big-oh.
2. Prove it using the formula $f(n) <= O(g(n))$.

Solution:

Big-oh (upper bound) of $f(n) = 2^n + 6n^2 + 3n$ will be 2^n iff

$f(n) <= c.\ 2^n$ for some constant $c > 0$ and $n >= n_0 >= 0$

$2^n + 6n^2 + 3n < c.\ 2^n$

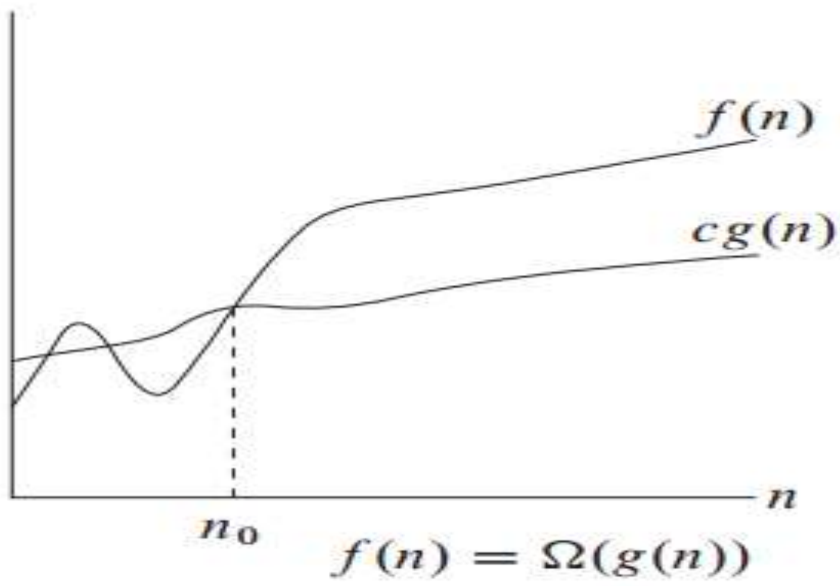
If we put c=11 and $n_0 =1$, then we get

11 <= 22 (It is true.)

It means big-oh of f(n) is 2^n when c=11 and $n_0 = 1$

2. Big-omega Notation (Ω): It is used to describe the lower bound or best-case performance of an algorithm in terms of its time complexity or space complexity. It provides an estimate of the minimum amount of time or space an algorithm can require for its execution as the input size increases.

T



We say that

f(n) = Ω g(n) if and only if

f(n) >= c . g(n)   for some c >0 after n >= n_o >=0

For example :

1. Given f(n) = 3n + 2 and g(n) = n, then prove that f(n) = Ω g(n)

Solution

1. We have to show that f(n) >= c. g(n) where c and n_0 are some positive constants for all n which is >= n_0
2. 3n + 2 >= c . n
3. When we put c=1
4. 3n +2 >= n
5. Put n = 1
6. 5 > = 1 [True]
7. Hence, when we put c=1 and n_0=1, f(n)= Ω g(n).

2. Solve the function: $f(n) = 3^n + 5n^2 + 8n$ and find the big omega(lower bound) notation.

Solution :

Steps:

1. Find out the smallest degree of n from f(n). This will be the value for lower bound (best case for the function f(n).
2. Use the formula to find out c and n_0 to prove your claim.

$f(n) = \Omega(n)$ iff $f(n) >= C.\ n$ where c is some constant and $n>=n_0>=0$

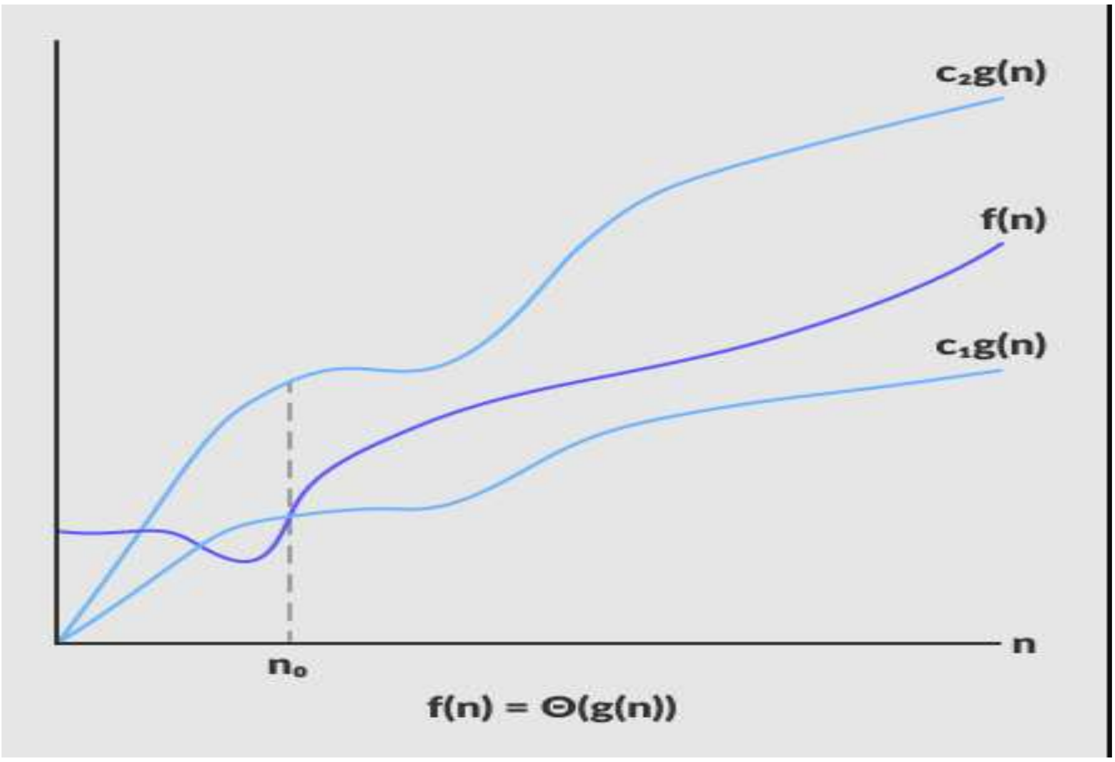
$3^n + 5n^2 + 8n >= c.\ n$

If we put c=16 and $n_0 = 1$, then we get

16 >= 16 (holds good)

It is means lower bound (Ω) for the given function $f(n)= 3^n + 5n^2 + 8n$ is n.

3. Big Theta Notation (Θ): It's the middle characteristics of both Big O and Omega notations as it represents the **lower and upper bound** of an algorithm.



We say that

f(n) = **ɵ g(n) if and only if**

c1.g(n) <= f(n) <= c2.g(n) for all n >=n_0>=0 and c >0

For example

1. **Given f(n) = 3n +2 and g(n) = n, prove that f(n) = ɵ g(n).**

Solution

1. We have to show that c1.g(n) <= f(n) <= c2.g(n) for all n >=n_0>=0 and c >0
2. c1.g(n) <= f(n) <= c2.g(n)
3. c1. n <= 3n+2 <= c2.n
4. Put c1 =1, c2 = 4 and n=2 , then 2 <= 8 <=8 [True]

5. Hence, f(n) = **ɵ g(n) where c1=1,c2=4 and n_0=2.**

3. Solve the function: $f(n) = 27n^2 +16$ and find the Tight (average bound) bound it.

Solution:

1. If we have upper bound (big-oh) and lower bound (big omega) of f(n) equal, that's when we can call it Theta-notation of f(n).
2. Use the formula c1. g(n) <= f(n) <= c2. g(n)

Let's check if n^2 is Theta or not.

$27n^2 +16 <= c1. n^2$ (for upper bound)

If we put c1 = 43 and n=1, then we get

43 < = 43 (holds good)

Now check for the lower bound

$27n^2 +16 >= c2. n^2$

If we put c1 = 43 and n=1, then we get

43 > = 43 (hold good)

Since upper and lower bounds are the same for the given function $f(n)= 27n^2 + 16$, n^2 is Tight- bound for the function f(n).

Q: Find out upper, lower and average bounds for the function $f(n) = 3n+2$.

Sol^n:

For upper bound:

$$f(n) \leq c.g(n)$$

$$\boxed{3n+2 \leq 5n} \Rightarrow \text{True}$$

$c=5$ and $n_0 = 1$ (starting value of "n")

$g(n)$

$$\therefore f(n) = O(n)$$

For lower bound:

$$f(n) \geq c.g(n)$$

$$\boxed{3n+2 \geq c.n} \Rightarrow \text{True}$$

$C = 1$ and $n_0 = 1$ (starting value of "n")

$g(n)$ [Even for $n_0=0$ it is true]

$$\therefore f(n) = \Omega(n)$$

Since $O(n) = \Omega(n)$

$$\boxed{\therefore f(n) = \Theta(n)}$$

4. Small-oh [o] : We use o-notation to denote an upper bound that is not asymptotically tight whereas big-oh (asymptotic upper bound) may or may not be asymptotically tight.

We say that

f(n) = o(g(n)) if and only if

0<= f(n) < c. g(n) for all values of c which is >0 and $n>=n_0>0$

Or

Lim f(n)/g(n) = 0

$n \to \infty$

For example

1. Give $f(n) = 2n$ and $g(n) = n^2$, prove that $f(n) = o(g(n))$

Solution

$\lim 2n/n^2$
$n \to \infty$

$\lim 2/n$
$n \to \infty$

$\lim 2/\infty \quad = 0$
$n \to \infty$

Hence, $f(n) = o(g(n))$

5. **w [small omega] :** We use w-notation to denote a lower bound that is not asymptotically tight.

We say that

$f(n) = w(g(n))$ if and only if

$0 <= c. g(n) < f(n)$ for all values of c which is >0 and $n>=n_0>0$

Or

$\lim f(n)/g(n) = \infty$
$n \to \infty$

For example

1. Given $f(n)= n^2/2$ and $g(n)= n$, prove that $f(n) = w(g(n))$.

Solution

$$\lim_{n \to \infty} n^2/2 /n$$

$$\lim_{n \to \infty} n/2$$

$$\lim_{n \to \infty} \infty /2 \quad = \infty$$

Hence, $f(n) = w(g(n))$

Question. Why should we do asymptotic analysis of algorithms?

It is crucial for several reasons:

Efficiency Comparison: It allows us to compare and evaluate different algorithms based on their efficiency. By analyzing how an algorithm's performance scales with input size, we can select the most suitable algorithm for a given problem.

Algorithm Design: Asymptotic analysis guides the design of new algorithms. It helps in making informed design choices to optimize algorithms for various use cases.

Resource Management: Understanding the resource requirements of algorithms as input size grows helps allocate computational resources efficiently, preventing bottlenecks.

Scalability: It provides insights into how algorithms will perform as data sizes increase, ensuring that systems can handle larger inputs efficiently.

Question. Order the following functions by their asymptotic growth, and justify you answer: f1=2^n, f2= $n^{3/2}$, f3=nlog n, f4= $n^{\log n}$.

$$\left.\begin{aligned} f_1 &= 2^n \\ f_2 &= n^{3/2} \\ f_3 &= n\log n \\ f_4 &= n^{\log n} \end{aligned}\right] \Rightarrow \text{Given functions}$$

Sol^n: I) f_1 and f_2

Apply "log" both sides

$\log 2^n$ and $\log n^{3/2}$

$n\log 2$ and $\frac{3}{2}\log n$

$\log 2 \rightarrow = 1$ [$\because \log_2 2 = 1$]

n and $\frac{3}{2}\log n$

Now, put $\boxed{n = 2^{128}}$

2^{128} and $\frac{3}{2}\log 2^{128}$

and $\frac{3}{\cancel{2}} \times \overset{64}{\cancel{128}}$ [$\because \log_2 2 = 1$]

2^{128} and 192

$\therefore \boxed{f_1 > f_2}$

<u>Now</u> f_1 and f_3

2^n and $n\log n$

Apply log both sides

$\log 2^n$ and $\log(n\log n)$

n and $\log n + \log\log n$ $\boxed{\because \log(a*b) = \log a + \log b}$

put $\boxed{n = 2^{128}}$

2^{128} and $\underbrace{\log 2^{128}}_{\Downarrow} + \log(\log 2^{128})$

2^{128} and $128 + \log 128$

2^{128} and $128 + 7$

$\therefore \boxed{f_1 > f_3}$

<u>Now</u> f_1 and f_4

2^n and $n^{\log n}$

Apply log both sides

$\log 2^n$ and $\log(n^{\log n})$

n and $\log n \log n$

put $\boxed{n = 2^{128}}$

2^{128} & $(\log 2^{128})(\log 2^{128})$

2^{128} and 7×7

2^{128} and 49

$\boxed{f_1 > f_4}$

Now f_4 and f_3

$n^{\log n}$ and $n\log n$

Apply log both sides

$\log(n^{\log n})$ and $\log(n\log n)$

$\log n \log n$ and $\log n + \log\log n$

put $n = 2^{128}$

128×128 and $128 + 7$

$\therefore \boxed{f_4 > f_3}$

Now, f_2 and f_3

$n^{3/2}$ and $n\log n$

$\frac{3}{2}\log n$ and $\log n + \log\log n$

$\boxed{n = 2^{128}}$

$\frac{3}{\cancel{2}} \times \overset{64}{\cancel{128}}$ and $128 + 7$

192 and 135

$\therefore \boxed{f_2 > f_3}$ Ans

$\therefore \boxed{f_1 > f_4 > f_2 > f_3}$

Complexity of Algorithms

1. Time complexity
2. Space complexity

Algorithms can be broadly categorized into two main groups based on their structure and approach:

1. Iterative algorithms (having loop(s))
2. Recursive algorithms (having recursion)

Note: In the 'a priori' analysis of algorithms, the RAM (Random Access Machine) model is used for analyzing algorithms without running them on a physical machine.

The RAM model has the following properties:

- **A simple operation (+ , \ , * , - , = , &&, ||, if) takes one-time step.**
- **Loops are comprised of simple/primitive/basic operations.**
- **Memory is unlimited and access takes one-time step.**

Note: In the last step of the 'a priori' analysis of algorithms, asymptotic notation is commonly used to characterize the time complexity of an algorithm. Asymptotic notation provides a concise way to describe how the performance of an algorithm scales as the input size becomes very large. It allows us to focus on the most significant factors affecting the algorithm's efficiency and ignore constant factors and lower-order terms.

Q. What are the key characteristics of the RAM model?

Time complexity (running time)

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes Ci steps to execute and executes "n" times will contribute (Ci * n) to the total running time.

Note: "C_i (i=1,2,3 …, n)" indicates constant unit of time.

```
A(n)
{                                   Cost        Times
   int i;                           C1            1
   for( i = 1; i <=n ; i++)         C2          n+1
       printf("MIET");              C3           n
}
```

$$T(n) = C1*1 + C2*(n+1) + C3*n$$

After eliminating constant terms, we get the time complexity in terms of n.

O(n); linear time

Q. **With a suitable example, define the term "running time" of an algorithm.**

Time Complexity of Iterative Algorithms

Note— When an algorithm contains an iterative control construct such as a while or for loop, we can express its running time as the sum of the times spent on each execution of the body of the loop.

Note— If a program doesn't have loop(s) as well as recursion, then it takes O (1)- constant running time.

```
A()
{
    pf("MIET");        // one-time step
    pf("MIET");       // one-time step
    pf("MIET");       // one-time step
}
```

1+1+1= 3 (it's a constant.)

Note— We can define a constant running time using either O (1) or O(C).

Pattern-01 One loop and increment/decrement is by 1

```
A(n)
{
for(i=1 ; i<=n; i++)     → n+1
    pf("MIET");          →  n-1
}
```

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

= 2n+2

T(n) = O(n) [We remove all constant terms and consider only highest degree of n for the running time.]

Pattern-02 One loop and increment/decrement is not by 1

In this case, we need to calculate the number of iterations carefully.

```
A (n)
{
   int i;
   for( i= 1;  i<n ; i= i*2){
         pf("MIET");
       }
}
```

As loop is not getting incremented by one, we will have to carefully calculate the number of times "MIET" will be executed.

i= 1, 2, 4, . . . , n

After Kth iterations, "i" gets equal to "n":

i= 1, 2, 4, . . . , n

Iterations= $2^0, 2^1, 2^2, \ldots, 2^k$

$2^k = n$

Convert it into logarithmic form… [If $a^b = c$, we can write it $\log_a{}^c = b$]

$k = \log_2 n$

O(logn)

```
A(n)
{
    int  i , j;
    for( i = 1 to n)
        for(j = 1 to n)
            pf("MIET");    //It
will be printed n^2 times .
}
```

Time complexity is $O(n^2)$

```
A(n)
{
    int  i , j , k;
    for( i = 1 to n)
        for(j = 1 to n)
            for(k = 1 to n)
                pf("MIET");//n^3
}
```

Time complexity is $O(n^3)$

Pattern-4 When there is a dependency between the loop and the statements in the body of the loop.

```
A(n)
{
1.  int i = 1, j = 1;
2.  while( j <= n)
  {
3.  i++;
```

```
4.   j = j + i;
5.  pf("MIET"); // We need to know the no of times it'll be printed
   }
}
```

Solution:

We have to find out the number of times "MIET" will be printed to know the time complexity of the above program. We can see that there is a dependency between the line number 2 (while loop) and 4(the value of "j" which in turns depends on "i").

i = 1, 2, 3, 4, … k

j = 1, 3, 6, 10 … k(k+1)/2 [sum of the first "K" natural numbers]

k(k+1)/2 = n+1 [when the value of "n" gets n+1, condition gets false]

k^2 = n [We eliminate constant terms, consider only variable.]

k =√n Time complexity is O(√n)

Pattern05: When there is a dependency between loops (having more than one loop)

Note – We have to unroll loops in order to find out the number of times a particular statement gets executed.

```
A(n)
{
    int  i, j, k;

        for(i = 1; i <= n; i++)
         {
            for(j = 1; j <= i; j++)
             {
                for(k = 1; k <= 100; k++)
                    {
                      Pf("MIET");
                    }
             }
         }
    }
```

There is a dependency between the second and the first loop; therefore, we will have to unroll the loops to know the number of times "MIET" will be printed.

i = 1	i = 2	i = 3	...	i = n
j = 1	j = 2	j = 3		j = n
k = 1*100	k = 2 * 100	k = 3* 100		k = n* 100

1*100 + 2*100 + 3*100 + . . . + n*100

100(1+ 2+3 +…n)

100(n(n+1)/2) = 50*n^2 + 50*n

Time complexity = O(n^2) [We remove constant terms and lower order terms.]

Q. Write a function to compute x^n in logarithmic time complexity using an iterative approach.

By S. Khan

```
int Power(int x, int n)
{
    int result = 1;
    While (n > 0)
    {
        if (n % 2 == 1)
        {
            result = result * x;
        }
        x = x * x;
        n = n / 2;
    }
    return result;
}
```

making problem size half each time the condition n>0 gets true.

Running Time : $T(n) = O(\log_2 n)$

<u>Time Complexity of Recursive Algorithms</u>

To find out the time complexity of recursive programs, we have to write a recurrence relation for the given program and then solve it using one of the following methods:

1. Iteration or Back substitution method
2. Recursion-tree method
3. Master method
4. Forward substitution method (Substitution Method)
5. Changing variable method

Let's learn how to write a recurrence relation for the given program having a recursive call/function.

Note: Each recursive algorithm or program must have a stopping condition (also known as an anchor condition or base condition) to halt the recursive calls.

```
A(n)
{
 if (n > 0) // stopping condition for the recursive call
   {
      pf("MIET");
      A(n-1);  // Calling itself( recursive function)
    }

  }
```

We assume that T(n) is the total time taken to solve A(n) , where n is the input. It means that this T(n) is split up among all statements inside the function i.e., time taken by all instructions inside a function is equal to T(n).

Note: "if" and "print" take constant amount of time step as per the RAM model, and we can use either 1 or C to indicate it. When "if- condition" gets false, it again takes constant amount of time — (one-time step).

Recurrence relation for the above program is given below:

$$T(n) = \begin{cases} T(n-1) + 1 & \text{when } n>0 \\ 1 & \text{When } n = 0 \text{ (stopping condition)} \end{cases}$$

#Mixed (iterative + recursive)

```
A(n)
{
  If(n>0)                              ...... 1
  {
    for(i=1; i<=n; i++)                  .... n+1
      {
        pf("MIET");                     ...... n
      }
   A(n-1);                               ....T(n-1)
   }
}
```

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

#Factorial of a number

```
fact(n)
{
   if(n<=1)
```

```
   return 1;
else
   return n*fact(n-1); // here "*" takes constant time step
}
```

Note: Multiplication and other instructions in green will take a constant amount of time. Left side of the * is the first operand, cannot be included in the equation.

$$T(n) = T(n-1) + c \quad \text{when } n>1$$
$$= \quad 1 \qquad \text{when } n <=1$$

#Fibonacci number Fn

```
fib(n)
{
  if (n== 0 || n==1)
     return n;
  else
    return fib(n-1)+ fib(n-2);
}
```

$$T(n) = T(n-1)+ T(n-2) + 1 \text{ when } n >1$$
$$1 \text{ unit of time} \quad \text{when } n<=1$$

1. Iteration method (backward substitution) for solving recurrences

The Iteration Method, also known as Backward Substitution, is a technique used to solve recurrence relations and determine the time complexity of algorithms. This method involves iteratively substituting a recurrence relation into itself, moving backward towards the initial conditions or base cases, until a pattern or closed-form solution emerges.

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

Note— When solving a recurrence relation to determine the time complexity, our goal is to address the initial term given in T(…), which represents a sub-problem. We utilize the base condition to simplify the T() term.

$T(n) = T(n-1) + 1$ ………………….(1)

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

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

Back substitute the value of T(n-1) into (1)

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

$T(n) = T(n-2) + 2$ …………………(2)

Now, substitute the value of $T(n-2)$ into (2)

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

$T(n) = T(n-3)+3$ ……………………….(3)

.

.

.

$= T(n-k)+k$ [Assume $n-k = 0$ so, $n= k$]

$= T(n-n)+n$

$= T(0) + n$ [$T(0) = 1$ is given]

$= 1+ n$

$$T(n) = O(n)$$

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

$T(n) = T(n-1) + n$ ……………………….(1)

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

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

Substituting the value of $T(n-1)$ into (1)

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

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

Substituting the value of $T(n-2)$ into (2)

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

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

.

.

.

$T(n) = T(n-k)+(n-(k-1))+ (n-(k-2))+. . . +(n-1) + n$ ………..(4)

Assume that $n-k = 0$

Therefore $n = k$

In place of k, substitute "n" in the equation (4)

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

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

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

$= O(n^2)$

Solve the recurrence using back substitution method :

$T(n) = 2T(n/2) +n$ [previous year question]

Base condition is not given in the question; therefore, we assume that when n=1 , it takes 1 unit of time.

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

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

$T(n/4)= 2T(n/8) + n/4$

Substituting the value of T(n/2) into (1), we get

$T(n) = [\ 2\ (\ 2T(n/4)+n/2)+ n]$

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

Substituting the value of T(n/4) into (2), we get

$T(n) = [4\ (2T(n/8) + n/4) + 2n$

$= 2^3\ T(n/8) + n + 2n$

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

.

.

.

$= 2^k\ T(n/2^k) + k*n$ ....(4)

Assume that $(n/2^k) = 1$, then

$2^k = n$

$\log_2 n = k$

$T(n) = 2^{\log_2 n}\ T(n/n) + n*\log n$

$T(n) = n\ T\ (1) + n\ \log n$ $\qquad$ [Since $2^{\log_2 n} = n$]

= n+ nlogn= O(nlogn)

V.V.I

V.V.I Q solve the following recurrence using iteration (back substitution) method:

$$\boxed{T(n) = T(n-1) + n^4} \quad \text{—— (1)}$$

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

Put the value of $T(n-1)$ into eq — (1)

$$T(n) = T(n-2) + (n-1)^4 + n^4 \quad \text{—— (2)}$$

Put the value of $T(n-2)$ into eq^n — (2)

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

$$\vdots$$

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

Assume $n-k=0 \therefore k=n$

$$\therefore T(n-n) + (n-(n-1))^4 + (n-(n-2))^4 \dots n^4$$

$$T(0) + (1)^4 + (2)^4 + \dots n^4$$

when T(0) it takes 1 unit of time (Base condition)

$$1 + ((1)^4 + (2)^4 + \dots n^4)$$

$$1 + \quad \text{Sum} = \frac{n}{30}(n+1)(2n+1)(3n^2+3n+1)$$

Remove all constant terms and Consider largest value in term of "n".

$$\boxed{T(n) = O(n^5)}$$

Q The recurrence relation $T(n) = 7T(\frac{n}{2}) + n^2$ describes the execution time of an algorithm A. A's competitor algorithm, let A', has execution time $T'(n) = aT'(\frac{n}{4}) + n^2$. What is the greatest integer value of "a", for which A' is asymptotically faster than A?

Sol^n: $A = T(n) = 7T(\frac{n}{2}) + n^2$ —— (I)

$$T(\tfrac{n}{2}) = 7T(\tfrac{n}{4}) + (\tfrac{n}{2})^2$$

$$T(n) = 7\left[7T(\tfrac{n}{4}) + (\tfrac{n}{2})^2\right] + n^2$$

$$= 49T(\tfrac{n}{4}) + \tfrac{7}{4}n^2 + n^2$$

$$= 49T(\tfrac{n}{4}) + n^2(\tfrac{7}{4} + 1)$$

$$T(n) = \underline{49}T(\tfrac{n}{4}) + n^2(\tfrac{11}{4}) \text{ —— (II)}$$

$$T'(n) = \underline{a}T'(\tfrac{n}{4}) + n^2 \cancel{\tfrac{11}{4}}$$

At a=49, both algos have the same time complexity, so at a=48, A' will be asymptotically faster than A.

V.V.I

Lecture-09 (DAA) By S.Khan (1)

Q The recurrence $T(n) = 7T(n/3) + n^2$ describes the running time of an algorithm A. ~~Another~~ competing algorithm B has a running time of $S(n) = aS(\frac{n}{9}) + n^2$. What is the smallest integer value of "a" such that A is asymptotically faster than B?

Sol^n: $\Rightarrow A = T(n) = 7T(\frac{n}{3}) + n^2$ ——— (1)

$$T(\tfrac{n}{3}) = 7T(\tfrac{n}{9}) + (\tfrac{n}{3})^2$$

$$A = T(n) = 7[7T(n/9) + (\tfrac{n}{3})^2] + n^2$$

$$= 49T(n/9) + \tfrac{7n^2}{9} + n^2$$

$$A = T(n) = 49\,T(\tfrac{n}{9}) + (\tfrac{16}{9})n^2$$

$$B = S(n) = a\,S(\tfrac{n}{9}) + n^2$$

At $a = 49$, both A and B are the same.

At $a = 50$, A will be asymptotically faster than B

2. **Recursion-Tree Method for Solving Recurrences**

Type- 01 (Reducing function)

Steps:

1. Make T(n) the root node.
2. Draw the tree for two to three levels to calculate the cost and height.
3. If the cost at each level is the same, multiply it by the height of the tree to determine the time complexity.
4. If the cost at each level is not the same, try to identify a sequence. The sequence is typically in an arithmetic progression (A.P.) or geometric progression (G.P.).

Lecture-06

By
S.Khan

1

② Recursion-tree Method to solve "Recurrences".

$\to$ cost (time)

①

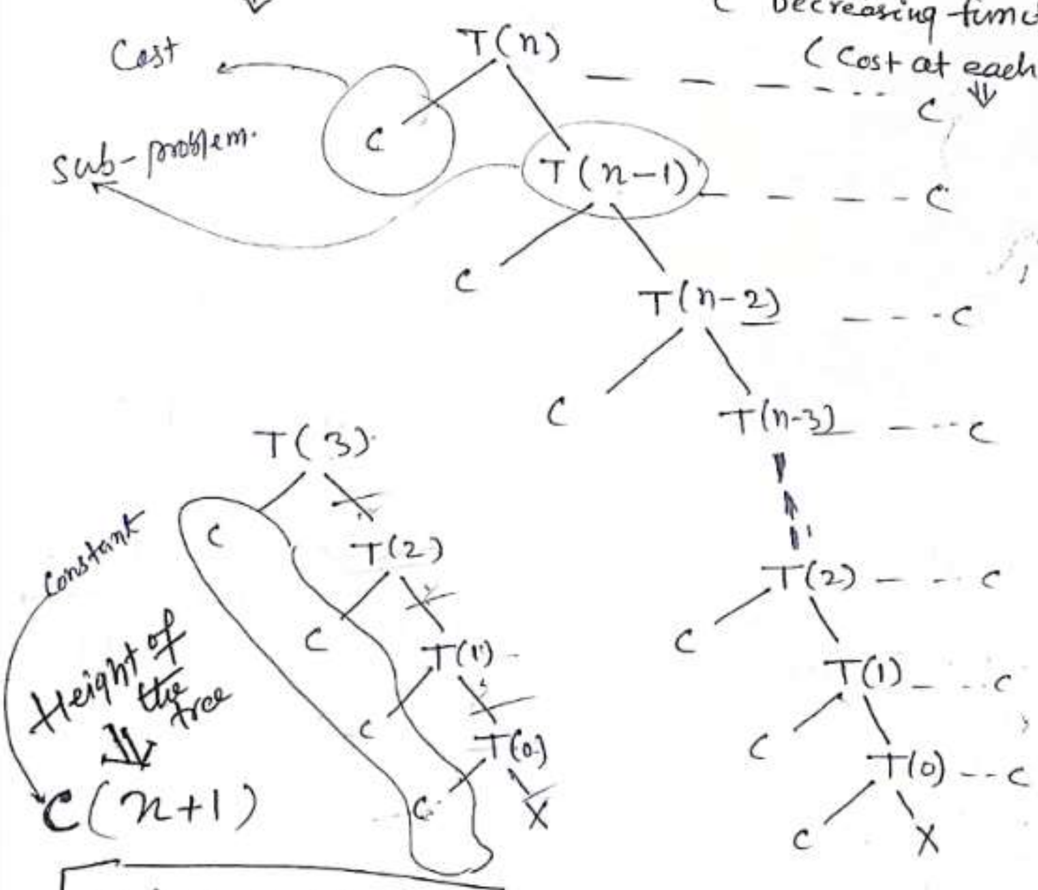
$$T(n) = \begin{cases} T(n-1) + c & ; n > 0 \\ 1 & ; n = 0 \end{cases}$$

$n = 0 \Leftarrow$ stopping condition (Base cond.) for recursive call.

$\hookrightarrow T(0) = 1$

$n-1 \hookrightarrow$ size of sub-problem (Decreasing function)

Sol^n :-

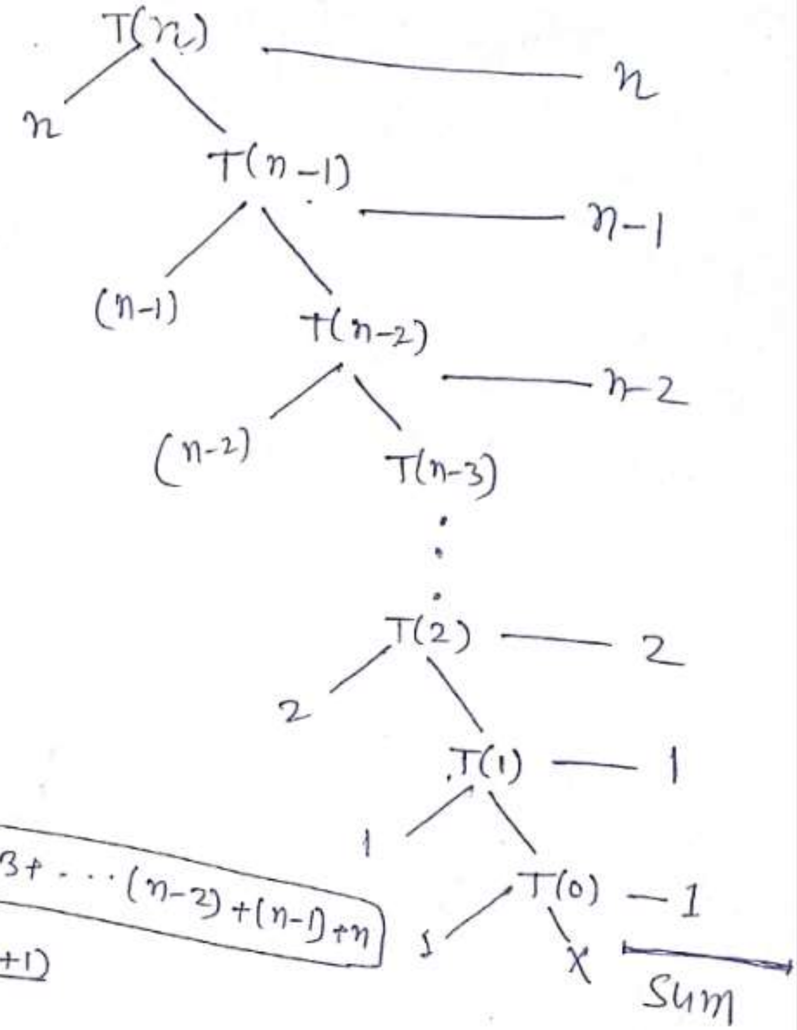


$$\boxed{T(n) = O(n)}$$

Note :- When "Cost" at each level is same, just find out the height of the tree and multiply with the cost to find out running time.

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

Sol^n :-



$T(n) = 1 + \boxed{1 + 2 + 3 + \cdots (n-2) + (n-1) + n}$

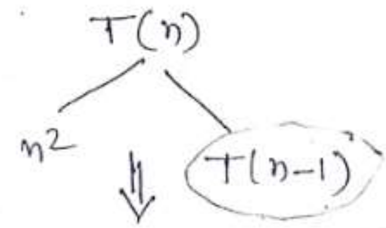
$= 1 + \frac{n(n+1)}{2}$

$= 1 + \frac{n^2 + n}{2}$

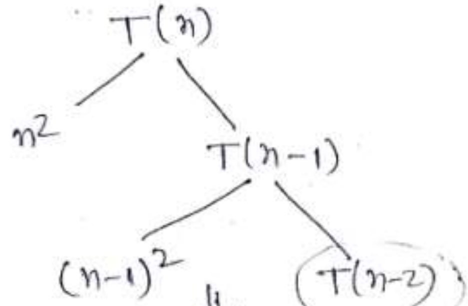
$T(n) = O(n^2)$

(3)

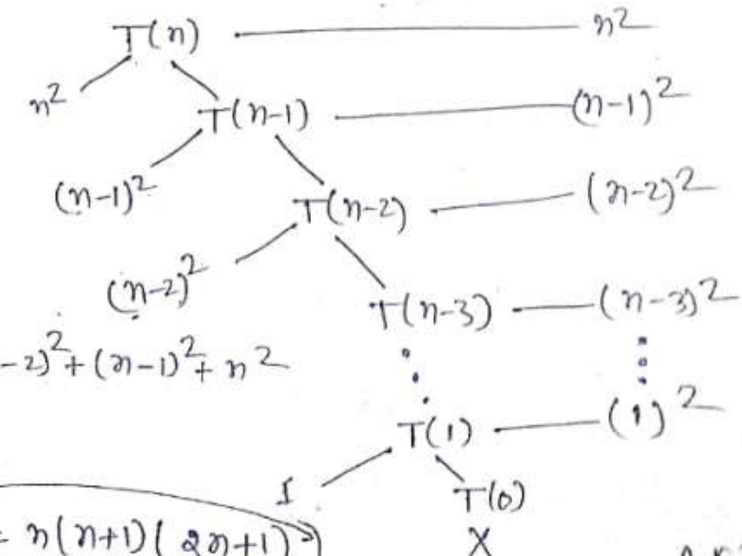
$$T(n) = \begin{cases} T(n-1) + n^2 & ; n > 0 \\ 1 & ; n = 0 \end{cases}$$



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



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



$$= 1^2 + \cdots + (n-2)^2 + (n-1)^2 + n^2$$

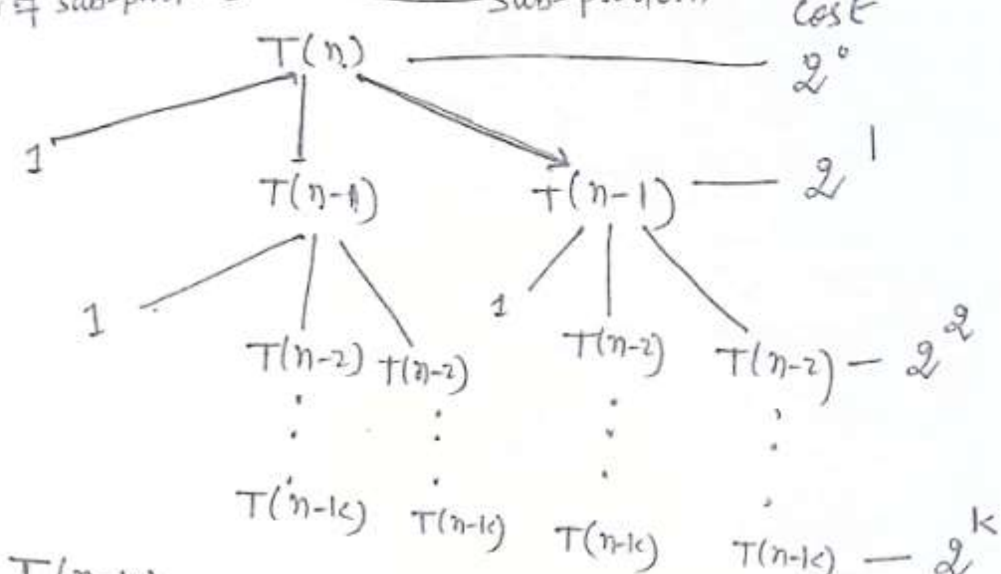
$$\boxed{T(n) = O(n^3)}$$

$$\because \; S_n = \frac{n(n+1)(2n+1)}{6}$$

↳ sum of squares of first n natural no.

(4) $\boxed{T(n) = 2T(n-1) + 1 \; ; n > 0 \text{ and } T(0) = 1}$

number of sub-problems — 2; sub-problem — $n-1$; cost — 1



$\because \; T(n-k) = 0$

$\therefore \; \underline{\underline{k = n}}$

$$T(n) = 2^0 + 2^1 + 2^2 + \cdots + 2^k \qquad \text{Terms} = k+1, \; r = 2$$

$$= \frac{1(2^{k+1} - 1)}{2 - 1} \qquad \left[\because \text{Common ration}(r) = 2\right]$$

$$= 2^{k+1} - 1 \qquad \left[S_n = \frac{a(r^n - 1)}{r - 1}\right] \text{ G.P}$$

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

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

$$\boxed{T(n) = O(2^n)} \qquad [\because k = n]$$

Type-2 (Dividing function- when there is more than one sub-problem, and the size of each sub-problem is the same.)

Steps:

1. **Make the last term the root node.**
2. **Draw the tree for two to three levels to calculate the cost and height.**
3. **If the cost at each level is the same, multiply it by the height of the tree to determine the time complexity.**
4. **If the cost at each level is not the same, try to identify a sequence. The sequence is typically in an arithmetic progression (A.P.) or geometric progression (G.P.).**

Note: If the size of sub-problem is only one, follow Type-1 approach only.

Lecture 07

By S.Khan (1)

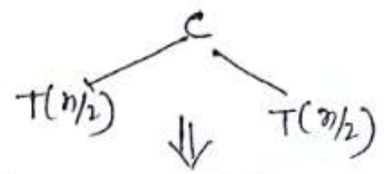
Type 02 → (Dividing functions)

Steps :

1. Make the 1st term the root node.
2. keep drawing till you get a pattern among all levels
3. The pattern is typically an A.P or G.P
4. Add the work done at all levels to get time complexity.

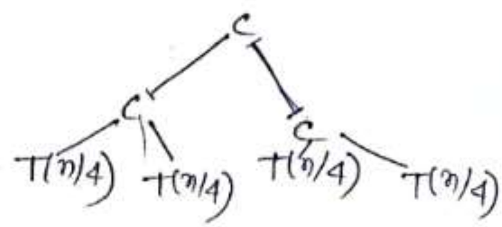
number of sub problems — size of each sub problem

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



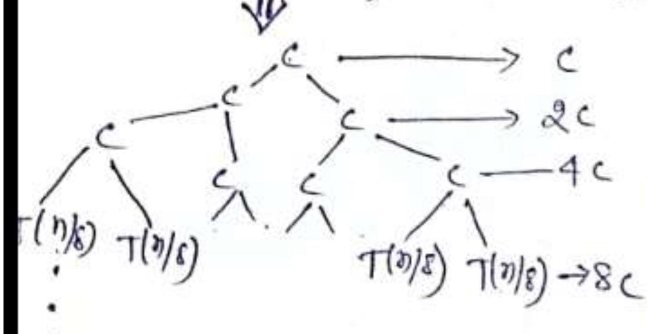
$\Downarrow$

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



$T(n/4) = 2T(n/8) + c$

$\Downarrow$



$T(n/2^k) = 1$ $\rightarrow 2^k c$

height of (# levels)

$c + 2c + 4c + \cdots + 2^k c$

$c(1 + 2 + 4 + \cdots + 2^k)$

$c(2^0 + 2^1 + 2^2 + \cdots + 2^k)$

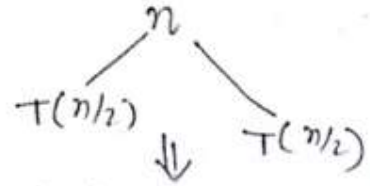
$c\left(\frac{1(2^{k+1}-1)}{(2-1)}\right)$

$c(2n-1)$ $\left[\because 2^k = n,\ n \times 2^1 = 2n\right]$

$\boxed{T(n) = O(n)}$

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

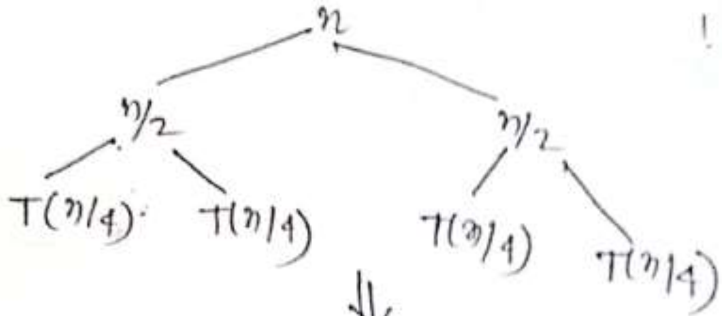
sol^n :]



$\Downarrow$

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

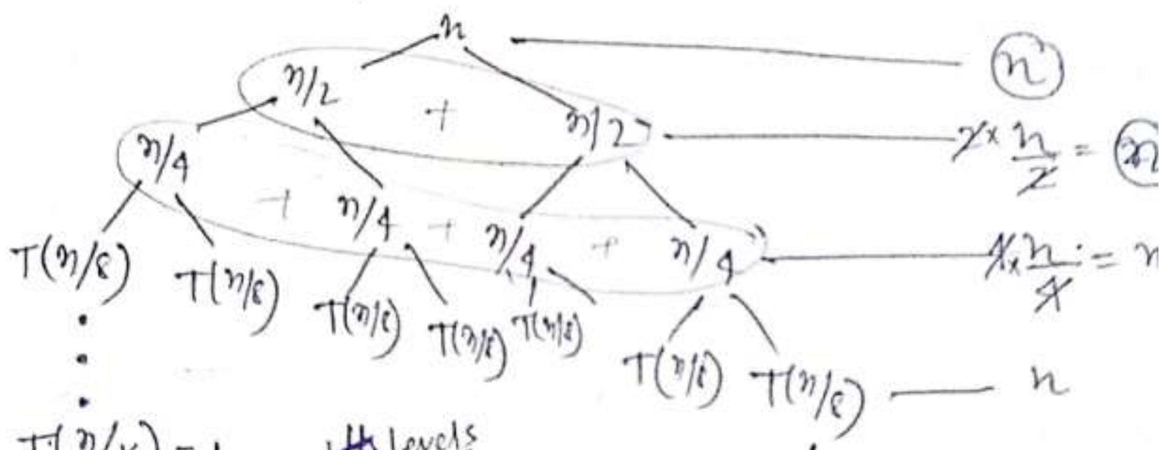
$\Downarrow$



$\Downarrow$

$T(n/4) = 2T(n/8) + n/4$

$\Downarrow$



$T(n/2^k) = 1$ # levels

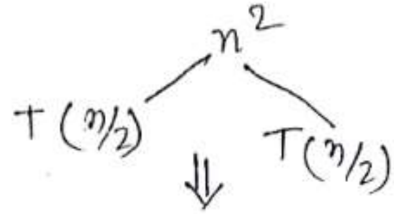
$2^k \times \frac{n}{2^k}$

$K = \log n$ and cost at each level is n

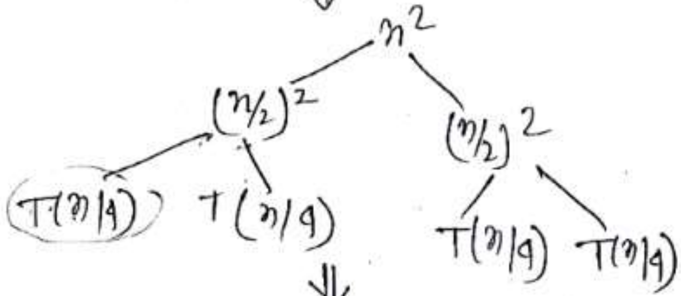
$\therefore \boxed{O(n \log n)}$

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

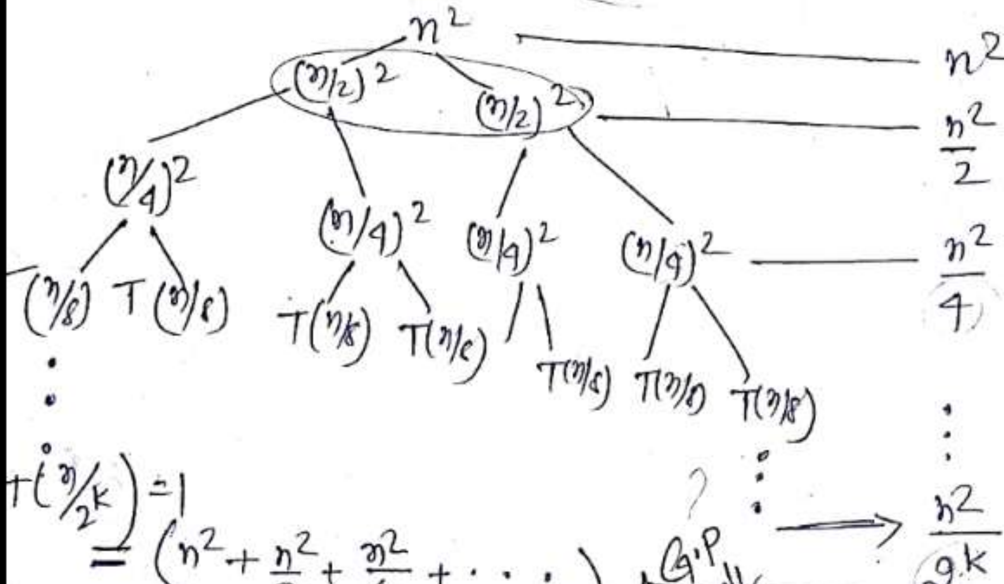
Sol^n:



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



$$T(n/4) = 2T(n/8) + \left(\frac{n}{4}\right)^2$$



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

$= \left(n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \cdots\right)$ G.P.

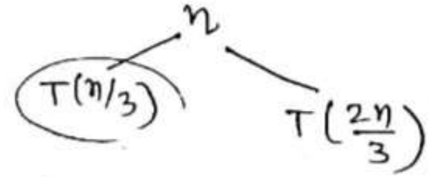
$= n^2\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right)$ $S_\infty = \frac{1}{1-r}$ if $r < 1$

$= \frac{n^2}{(1 - 1/2)} = \boxed{O(n^2)}$ here $r = 1/2$

Type—03 : (When the size of sub-problems is not equal)

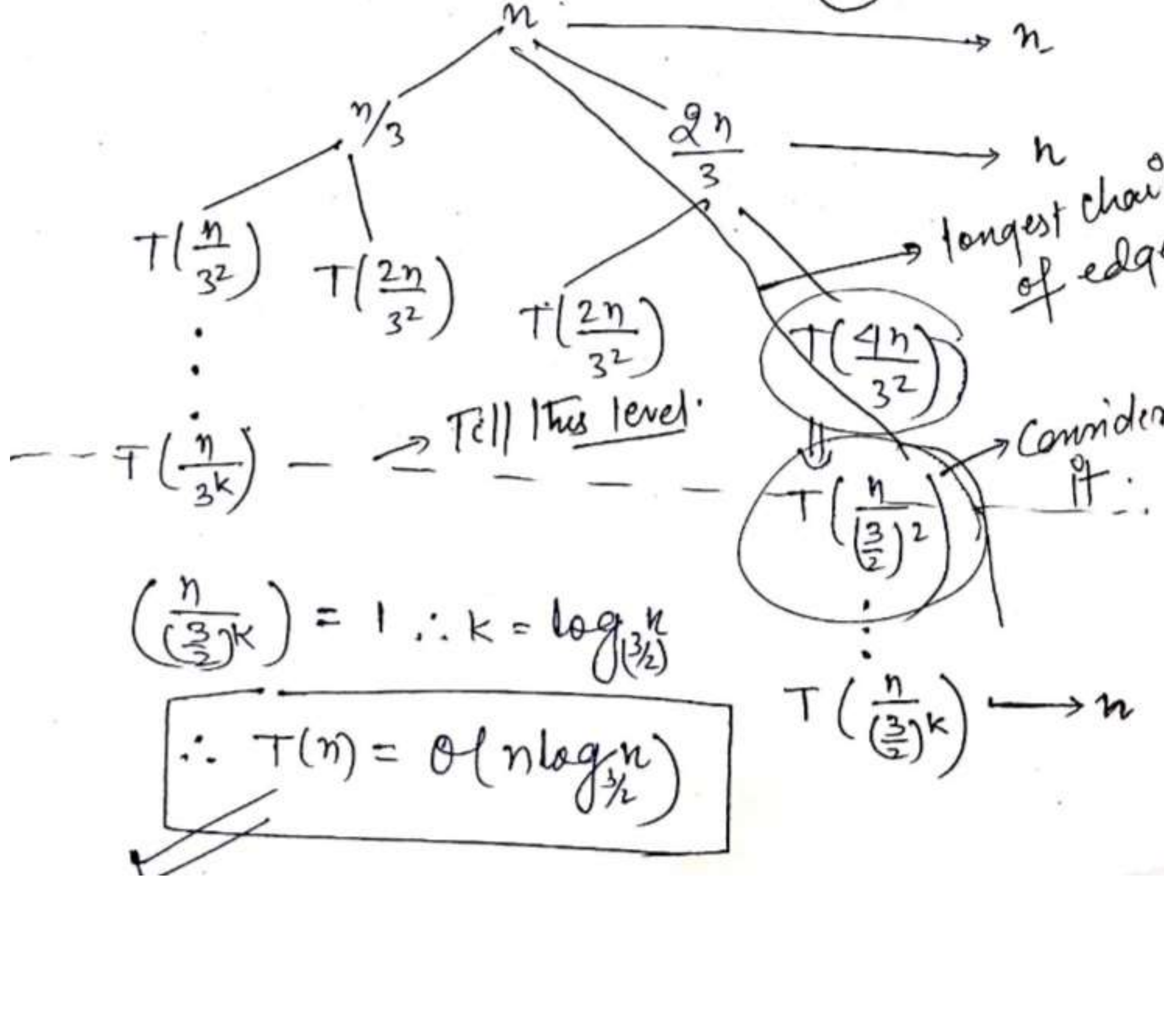
(I)
$$T(n) = \begin{cases} T(n/3) + T\left(\frac{2n}{3}\right) + n & ; n > 1 \\ 1 & ; n = 1 \end{cases}$$

Sol^n :



$$T\left(\frac{n}{3}\right) = T\left(\frac{n}{3^2}\right) + T\left(\frac{2n}{3^2}\right) + \frac{n}{3}$$ — new root of the left child

$$T\left(\frac{2n}{3}\right) = T\left(\frac{2n}{3^2}\right) + T\left(\frac{4n}{3^2}\right) + \frac{2n}{3}$$ — new right root.



$$\left(\frac{n}{\left(\frac{3}{2}\right)^k}\right) = 1 \therefore k = \log_{(3/2)} n$$

$$\boxed{\therefore T(n) = \theta(n \log_{3/2} n)}$$

Q. Solve the following recurrences:

i. $T(n) = T(n-1) + n^4$ using iteration method

ii. $T(n) = 3T(n/4) + cn^2$ using recursion tree method

V.V.I 20/01/22

(1)

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

3 → no of Sub-problems

$n/4$ → size of Sub-problem

$\underline{\underline{Sol^n}}$:

cn^2 → Cost at each level cn^2

$T(n/4)$ $T(n/4)$ $T(n/4)$

$$\boxed{T(n/4) = 3T\left(\frac{n}{4^2}\right) + c\left(\frac{n^2}{16}\right)}$$

3 → Times of Sub-problem

$\frac{n}{4^2}$ → Sub-problem size

$c\left(\frac{n^2}{16}\right)$ → Root

⇓

cn^2 → cn^2

$c\left(\frac{n^2}{16}\right)$ $c\left(\frac{n^2}{16}\right)$ $c\left(\frac{n^2}{16}\right)$ → $\frac{3}{16}n^2 * c$

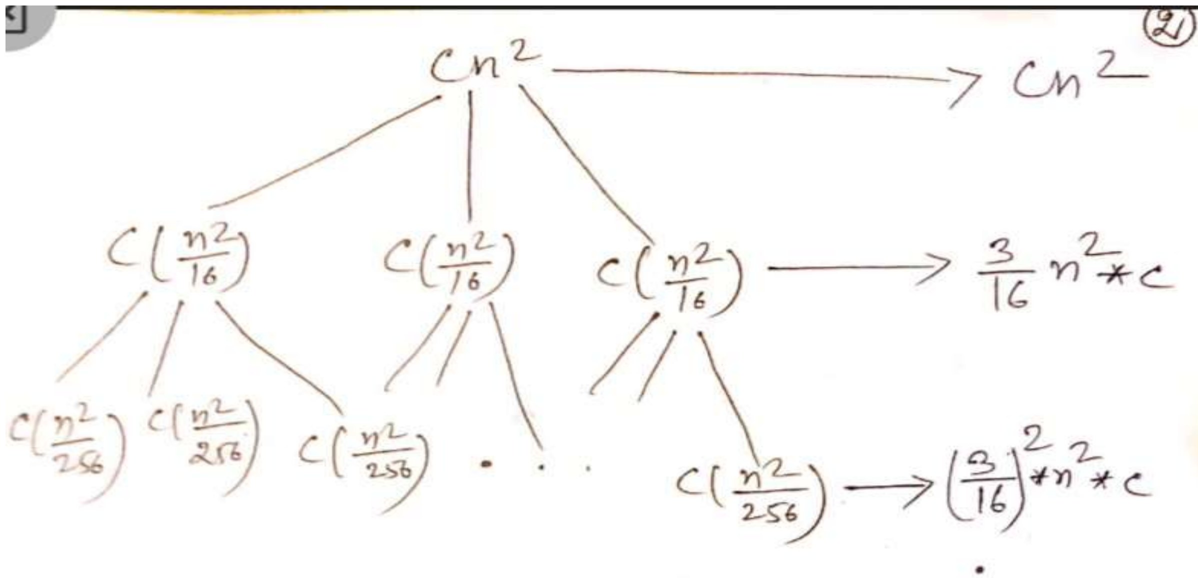
$T\left(\frac{n}{4^2}\right)$ $T(n/4^2)$ $T\left(\frac{n}{4^2}\right)$ $T\left(\frac{n}{4^2}\right)$ $T(n/4^2)$ $T(n/4^2)$ $T(n/4^2)$ $T(n/4^2)$ $T(n/4^2)$

⇓

$$\boxed{T\left(\frac{n}{16}\right) = 3T\left(\frac{n}{64}\right) + c\left(\frac{n^2}{256}\right)}$$

$\frac{n}{64}$ → Sub-problem size

$c\left(\frac{n^2}{256}\right)$ → Root



As the cost at each level is different, we need to find out a series to get the result.

$$T(n) = cn^2\left(1 + \left(\frac{3}{16}\right) + \left(\frac{3}{16}\right)^2 + \cdots \infty\right)$$

in G.P

$$r = \frac{3}{16}$$

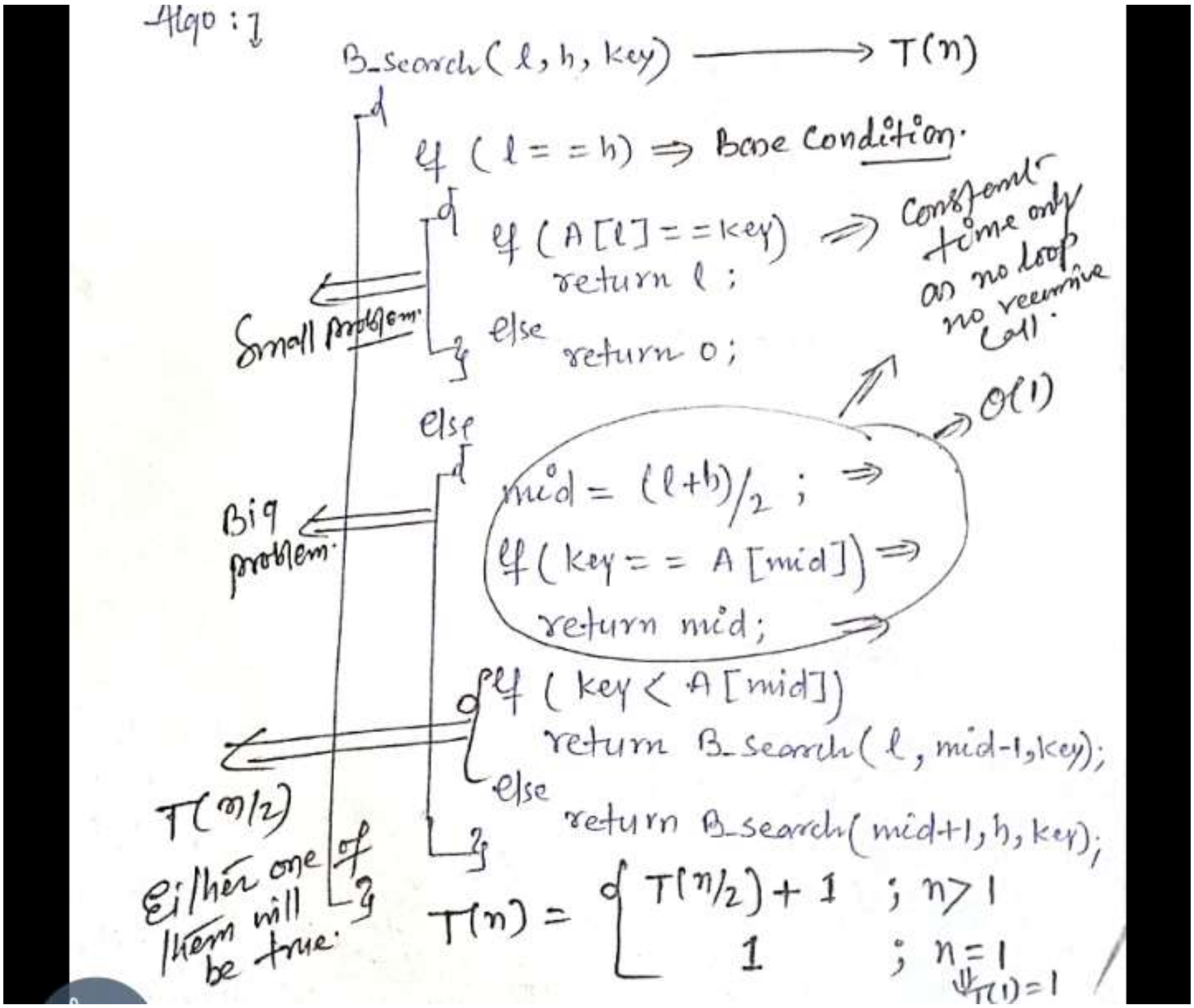
$$\therefore T(n) = \underbrace{c}_{\text{Constant}} n^2 \underbrace{\left(S_n = \frac{1}{1-r}\right)}_{\text{Constant}}$$

$\boxed{\frac{1}{1-r} \text{ if } r<1}$

$$\therefore \boxed{T(n) = \Theta(n^2)}$$

Q Explain Binary Search Algorithm. Also solve its recurrence relation.

It is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).



Algo:

```
B_Search(l, h, key) ----> T(n)
{
   if (l == h) => Base Condition.
   {
      if (A[l] == key)      => Constant time only
         return l;             as no loop
      else                     no recursive call.
         return 0;
   }
   else                                        => O(1)
   {
      mid = (l+h)/2 ;
      if (key == A[mid])
         return mid;
      if (key < A[mid])
         return B_Search(l, mid-1, key);
      else
         return B_search(mid+1, h, key);
   }
}
```

Small problem

Big problem

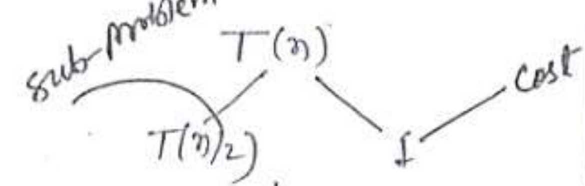
T(n/2)

Either one of them will be true.

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

$T(1) = 1$

Using recursion-tree method.



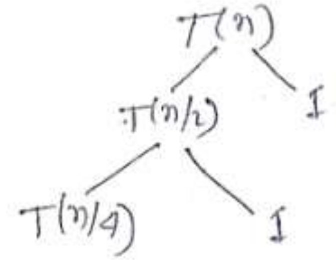
$\Downarrow$

Put $T(n/2)$ into eq^n — ①

$$T(n/2) = T(\tfrac{n}{4}) + 1$$

cost

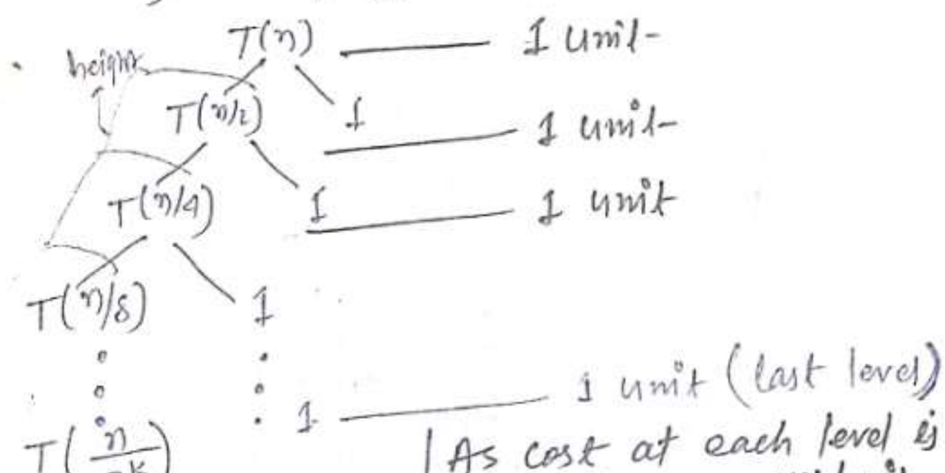
Note: Make root $T(n)$ as the sub-problem is only one time. If the number of sub-problems was more than one time, then we made cost (1) root.



Put $T(n/4)$ into eq^n — ①

$$T(n/4) = T\left(\frac{n}{8}\right) + 1$$

cost



height $\dfrac{n}{2^K} = 1$ (from base condition)

$\therefore K = \log_2 n$

As cost at each level is same, just multiply it with the height of the tree to get time complexity

$$\boxed{T(n) = O(\log n)}$$

V.V.I

Solve using recursive tree method

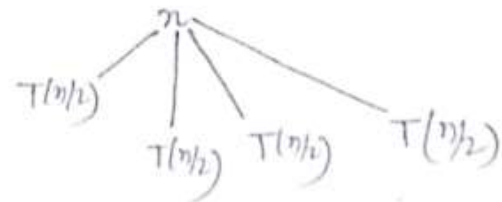
subproblem

cost

no of subproblems

$$T(n) = 4T(n/2) + n \quad \text{and} \quad T(1) = 1 \longrightarrow (1)$$

sol^n:

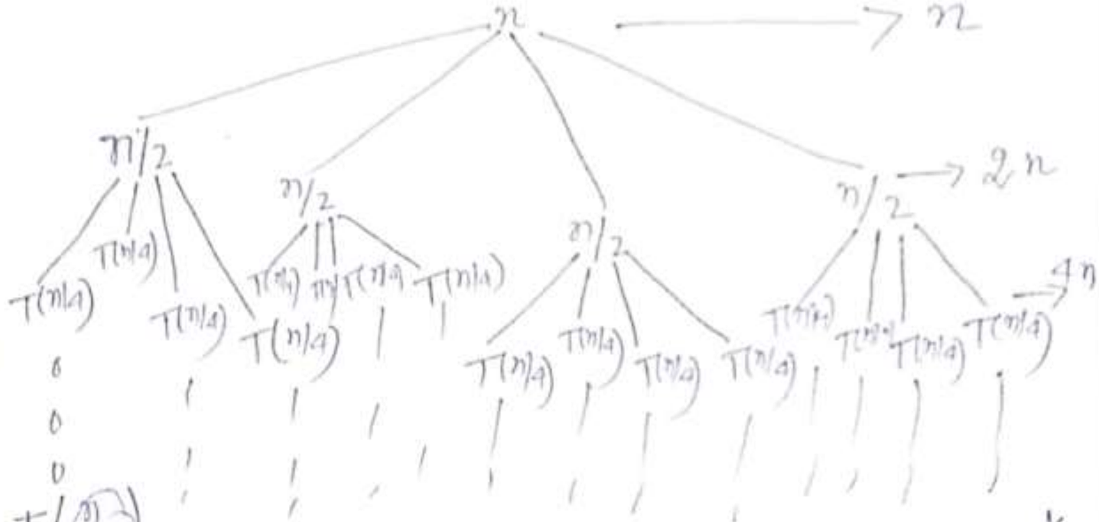


at the place of n

Put T(n/2) in equation $\longrightarrow$ (1)

$$T(n/2) = 4T\left(\frac{n}{4}\right) + \frac{n}{2}$$

Cost



$T\left(\frac{n}{2^K}\right)$ $\quad \frac{n}{2^K} = 1 \quad \therefore \boxed{2^K = n}$

$\longrightarrow 2^k * n$

last level

$$T(n) = n + 2n + 4n + \cdots + 2^k n$$
$$= n(1 + 2 + 4 + \cdots + 2^k)$$
$$= n(2^0 + 2^1 + 2^2 + \cdots + 2^k)$$

G.P term = k+1
r = 2

$$= n\left[\frac{1 \times (2^{k+1} - 1)}{2 - 1}\right] = n(2^K) \quad \because 2^K = n$$
$$= n \times n = \boxed{O(n^2)}$$

3 Master Theorem to solve recurrences

Note—Effective for the university exam

Question. State Master Theorem and find time complexity for the recurrence relation T(n) = 9 T(n/3) +n.

Solution— Let a >= 1 and b > 1 be constants, let f(n) be a function , and let T(n) be defined on the nonnegative integers by the recurrence

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

Where we interpret n/b to mean either floor value of (n/b) or ceiling value of (n/b). Then T(n) has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a-\epsilon})$ for some constants $\epsilon > 0$, then $T(n)= \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) <= c(f(n))$ for some constant c <1 and all sufficiently larger n, then $T(n)= \Theta(f(n))$.

Given: a= 9, b= 3 and f(n) = n

Now we need to calculate $\mathbf{n^{\log_b a}}$ as it's the common term in all 3-cases of the Master Theorem.

$n^{\log_b a} = n^{\log_3 9} = n^2$ (It is clearly bigger than f(n), which is n)

Case-1 can be applied; therefore, $T(n) = \Theta(n^2)$.

Question. State Master Theorem and find time complexity for the recurrence relation T(n) = T(2n/3) +1.

Solution— Given: a = 1, b= 3/2 and f(n) =1

Now we need to calculate $\mathbf{n^{\log_b a}}$ as it's the common term in all 3-cases of the Master Theorem.

$\mathbf{n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1}$ **[Since, $\mathbf{\log^1 = 0}$]**

As the result of $\mathbf{n^{\log_b a}}$ is equal to f(n) in the question, we can apply the second case (for tight bound/average case).

T(n)= $T(n) = \Theta(n^{\log_b a} \log n)$

=$T(n) = \Theta(\log n)$

Question. State Master Theorem and find time complexity for the recurrence relation T(n) = 3 T(n/4) +nlogn.

Solution— Given: a = 3, b= 4 and f(n) =nlogn

Now we need to calculate $\mathbf{n^{\log_b a}}$ as it's the common term in all 3-cases of the Master Theorem.

$\mathbf{n^{\log_b a} = n^{\log_4 3} = n^{0.793}}$

Since, nlogn = $\Omega(\mathbf{n^{\log_4 3+\epsilon}})$ where $\epsilon = 0.2$

Case-3 applies if we can show that

af(n/b) <= c.f(n)

3(n/4)log(n/4) <= (3/4)nlogn for c = ¾

By case-3, T(n) = ϴ(nlogn)

Note—Effective for the competitive exam

Lecture 08   By S.Khan (1)

(3) Master theorem to solve recurrences.

Form of Master theorem

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k \log^p n)$$

$a \geq 1$; $b > 1$; $k \geq 0$ and p is a real number.

1. If $a > b^k$, then $\boxed{T(n) = \Theta(n^{\log_b a})}$

2. If $a = b^k$
   - (a) $p > -1$ then $T(n) = \Theta(n^{\log_b a} \cdot \log^{p+1} n)$
   - (b) $p = -1$ then $T(n) = \Theta(n^{\log_b a} \cdot \log \log n)$
   - (c) $p < -1$ then $T(n) = \Theta(n^{\log_b a})$

3. If $a < b^k$
   - (a) $p \geq 0$ then $\boxed{T(n) = \cancel{\Theta(n^{\log_b a}.} \; \Theta(n^k \log^p n)}$
   - (b) $p < 0$ then $T(n) = O(n^k)$

For example :

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

$a = 2$, $b = 2$, $k = 1$ and $p = 0$

$a = b^k$ (true)

(2)(a) $T(n) = \boxed{\Theta(n \log n)}$

V.V.S Use master method to solve the following recurrence relation:

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

(b) $T(n) = 10T(n/3) + 17n^{1/2}$

Solⁿ:

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

$a = 1,\ b = 2,\ k = 1$ and $p = 0$

$a < b^k$ (True)

$p \geq 0 \therefore\ \theta(n^k \log^p n)$

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

$\boxed{T(n) = \theta(n)}$

(b) $T(n) = 10T\left(\frac{n}{3}\right) + 17n^{1/2}$

$a = 10,\ b = 3,\ k = \frac{1}{2}$ and $p = 0$

$a > b^k$ True.

$\therefore T(n) = \theta(n^{\log_b a})$

$\boxed{T(n) = \theta(n^{\log_3 10})}$

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

$a = 1$, $b = 2$, $k = 0$ and $p = 0$

$a = b^k$

(2)(a) $\Theta(n^{\log_b^a} \cdot \log^{p+1} n)$

$\Theta(n^0 \cdot \log n)$

$\boxed{\Theta(\log n)}$

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

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

$a < b^k$

(3)(a) ~~$\Theta(n^{\log}$~~

$\Theta(n^k \log^p n)$

$\Theta(n^2 \log^0 n)$

$\boxed{\Theta(n^2)}$ $[\because \log^0 n = 1]$

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

$a = 4$, $b = 2$, $k = 2$ and $p = 0$

$a = b^k$

(2)(a) $\Theta(n^{\log_b^a} \cdot \log^{p+1} n)$

$\boxed{\Theta(n \log n)}$

(v) $T(n) = 16T(n/4) + n$

$a = 16$, $b = 4$, $k = 1$ and $p = 0$

$a > b^k$

(1) $\therefore$ $\Theta(n^{\log_b^a})$

$\boxed{\Theta(n^2)}$

*** (vi) $T(n) = 2T(n/2) + n\log n$

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

$a = b$

(2)(a) $\Theta(n^{\log_b^a} \cdot \log^{p+1} n)$

$\boxed{\Theta(n \log^2 n)}$

*** (vii) $T(n) = 2T(n/2) + n/\log n$

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

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

$a = b^k$

(2b) ~~$\Theta(n^{\log_b^a} \cdot \log^{p+1} n)$~~

~~$\Theta(n \log^0 n)$~~

$= \boxed{\cancel{\Theta(n)}}$

$p = -1$

$\Theta(n^{\log_b^a} \cdot \log \log n)$

$\boxed{\Theta(n \log \log n)}$

Practice questions

Ans:

(1)	$T(n) = 4T(n/2) + \log n$	$\Theta(n^2)$
(2)	$T(n) = \sqrt{2}\, T(n/2) + \log n$	$\Theta(\sqrt{n})$
(3)	$T(n) = 2T(n/2) + \sqrt{n}$	$\Theta(n)$
(4)	$T(n) = 3T(n/2) + n$	$\Theta(n^{\log_2 3})$
(5)	$T(n) = 3T(n/3) + \sqrt{n}$	$\Theta(n)$
(6)	$T(n) = 4T(n/2) + Cn$	$\Theta(n^2)$
(7)	$T(n) = 3T(n/4) + n\log n$	$\Theta(n\log n)$

Note:

This theorem is valid for dividing functions.

$$T(n) = T(n/_{2,3,4,a}) + \cdots$$

Lecture 09

By S. Khan

(1)

(Master Theorem for Decreasing Function)

$$\boxed{\begin{array}{l} T(n) = aT(n-b) + f(n) \\ a>0,\ b>0 \text{ and } f(n) = O(n^k) \text{ where } k \geq 0 \end{array}}$$

Case (I) If $a = 1$ then $O(n \times f(n))$

Case (2) If $a > 1$ then $O(a^{n/b} \times f(n))$

Case (3) If $a < 1$ then $O(f(n))$

Examples :

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

$a = 1,\ b = 1$ and $f(n) = 1$

Case (1) $\boxed{O(n)}$

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

$a = 1,\ b = 1$ and $f(n) = n$

Case (1) $\boxed{O(n^2)}$

Examples

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

$a = 1$, $b = 1$ and $f(n) = \log n$

Case (1) $\boxed{T(n) = O(n \log n)}$

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

$a = 2$, $b = 2$ and $f(n) = 1$

Case (2) $O(2^{n/2} \times 1)$

$\boxed{T(n) = O(2^{n/2})}$

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

$a = 3$, $b = 1$ and $f(n) = 1$

Case 2 $\boxed{O(3^n)}$

(6) $2T(n-1) + n$

$a = 2$, $b = 1$, and $f(n)$

Case 2 $O(2^n \times n)$

$\boxed{T(n) = O(n \times 2^n)}$

4. Substitution method for solving recurrences [Forward Substitution method]

Lecture 10

By S.Khan (1)

(4) Substitution Method to solve recurrences :-
(forward- substitution method)

The substitution method for solving recurrences comprises two steps:

1. Guess the form of the solution
2. Use mathematical induction to find the "constants" and show that solution works.

base condition

Example :- (1) $T(n) = 2T(n/2) + n$, $T(1) = 1$ — (A)

Step 1 : Guess the solution.

$$\boxed{T(n) = O(n\log n)} \quad \text{— (1)}$$

Step 2 : Now we have to prove that our assumption is true — using mathematical induction.

By P.M.I (Principle of mathematical induction)

$T(n) \le c \cdot n\log n$ from eqn — (1)

Now put $n = 1$ in eqn — (1)

$T(1) \le c \cdot 1 \log 1$

$1 \le c \times 0$

$1 \le 0$ (False)

equation (1) fails for $n = 1$; therefore we have to check for $n = 2$ . . .

find its value from (A)

Now put $n = 2$ in eqn — (1)

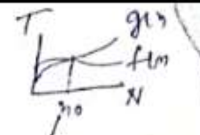
$T(2) \le c \cdot 2\log 2$

From eqn — (A) $T(2) = 2T(2/2) + 2$

$T(2) = 2T(1) + 2$ [$\because T(1) = 1$]

$T(2) = 4$

$\boxed{4 \leq 2c}$ $\left[\because 2\log 2 = 2\right]$



→ It is true for $c \geq 2$

and $c \geq 2$

It means initial value of "n" is $2(n_0)$; therefore, it should be true for $3, 4, \ldots, K$.

It means $T(K) \leq c \cdot k\log k$ $\boxed{\text{when } 2 \leq K \leq n}$

When we move from (2 to n) value of K some where in the middle, we get $K = (n/2)$ (As we need to remove $T(n/2)$ from eqn (A))

$$T(n/2) \leq c \cdot \frac{n}{2}\log n/2 \quad \text{—— (2)}$$

(value of $T(n/2)$)

Now put (2) into (A), we get

$$T(n) \leq 2\left(c \cdot \frac{n}{2}\log n/2\right) + n$$

$$T(n) \leq \cancel{2}c\frac{n}{\cancel{2}}\log n/2 + n$$

$$T(n) \leq cn\log n/2 + n$$

$$T(n) \leq nc(\log n - \log 2) + n$$

$$T(n) \leq nc(\log n) + n$$

$$T(n) \leq cn\log n + n$$ Bigger than "n"

$$T(n) \leq c\, n\log n$$ we remove constant term(s)

$$\boxed{T(n) = O(n\log n)}$$ proved

(2)

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

Step1 : Guess the solution.

$$T(n) = O(n^2) \text{ ——— } (1)$$

Step2 : Now we have to prove that our assumption is true using mathematical induction.

By P.M.I :

$$T(n) \leq c \cdot n^2 \text{ from eq}^n\text{-}(1)$$

Now put $n = 1$ in eq^n-(1)

$$T(1) \leq c \cdot (1)^2$$

$$T(1) \leq c$$

$$T(1) = T(1-1) + 1 \text{ from } (A)$$

$$= T(0) + 1$$

$$= 1 + 1$$

$$T(1) = 2$$

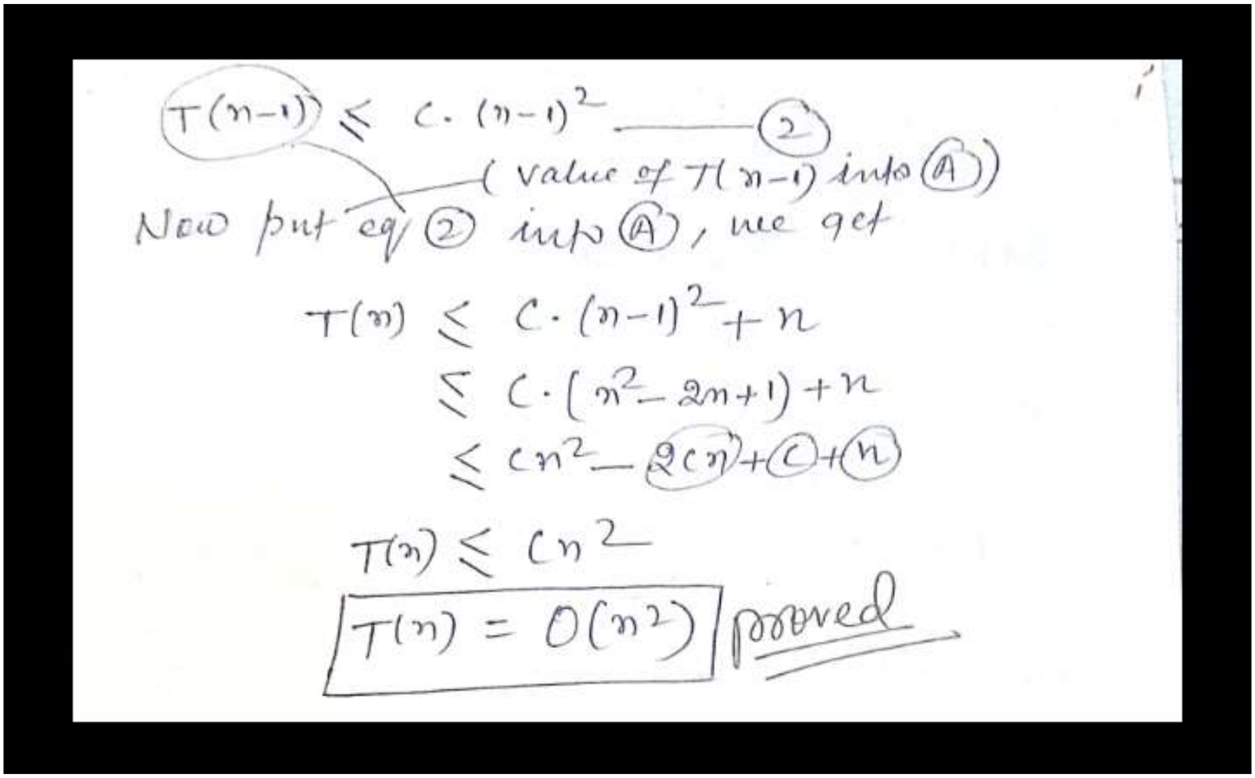
$\boxed{2 \leq c}$ True for $c \geq 2$

It means $n_0 = 1$ and $c \geq 2$ eq-(1) is true.

It should be true for $1, 2, 3, \cdots k$

$$T(k) \leq c \cdot k^2 \; ; \; \boxed{1 \leq k \leq n}$$

when we move forward from 1 to n, somewhere, we get $k = n-1$ (As we need to remove $T(n-1)$ from eq^n (A)



$T(n-1) \leq c \cdot (n-1)^2$ ——— (2)

(value of $T(n-1)$ into (A))

Now put eq (2) into (A), we get

$$T(n) \leq c \cdot (n-1)^2 + n$$
$$\leq c \cdot (n^2 - 2n + 1) + n$$
$$\leq cn^2 - 2cn + c + n$$
$$T(n) \leq cn^2$$
$$\boxed{T(n) = O(n^2)}$$ proved

Q. Solve by substitution method (Forward substitution method):

a. $T(n) = n * T(n-1)$ if $n>1$; $T(1) = 1$ …… [A]

Solution:

Step 1: Guess the solution

$T(n) = O(n^n)$ ……. [1] [You can easily get it using iteration method.]

Step 2: Now, we have to prove that our assumption is true using property of mathematical induction.

$T(n) <= c. n^n$ from equation-[1]

Now, put n=1 in equation-[1]

$T(1) <= c. 1$

$1 <= c.1$ [True for $c>=1$, $n_0 = 1$]

It should be true 1, 2, 3, . . ., k

$T(k) <= c. k^k$ [$1<= k <= n$]

When we move forward from 1 to n somewhere we get k = n-1.

$T(n-1) <= c. (n-1)^{(n-1)}$ …………………… [2]

Now, put the value of T(n-1) into equation [A].

$T(n) <= n * c. (n-1)^{(n-1)}$

$<= c * n * (n-1)^{(n-1)}$

$<= c * n * n^n$ [if n-1 = n]

$<= cn * n^n$ [we consider only bigger term]

$<= n * n^n$ [We remove constant term(s)]

$<= n^{n+1}$

$<= n^n$

Hence, $T(n) = O(n^n)$ proved

Lecture 11 — By S. Khan (1)

Analysing Space Complexity of Iterative and recursive algorithms

$$\boxed{\text{Space Complexity} = \text{input size} + \text{Extra Space (taken by variables)}}$$

→ for iterative algorithms / programs

Note: We just consider extra space taken by an algorithm to compute its space complexity when considering iterative version of algorithms.

For example: (input) — A → Array, n → size

```
fun(A, n)
{
    int i;
    for (i=1; i<n; i++)
        A[i] = 10;
}
```

→ only one extra variable.

Space complexity = $\boxed{O(1)}$

```
fun(A, n)
{
    int i;
    int B[n];
    for (i=1 to n)
    {
        B[i] = A[i];
    }
}
```

$(1+n) \Rightarrow \boxed{O(n)}$

```
fun(A, n)
{
    int B[n,n];
    int i, j;
    for (i=1 to n)
        for (j=1 to n)
            B[i,j] = A[i,j]
}
```

$(n^2+2) \Rightarrow \boxed{O(n^2)}$

Space Complexity of recursive Programs

We have two methods to compute space complexity of recursive algorithms/Programs:

(I) Tree Method (use it when the program is small)
(II) Stack Method (use it when the program is big)

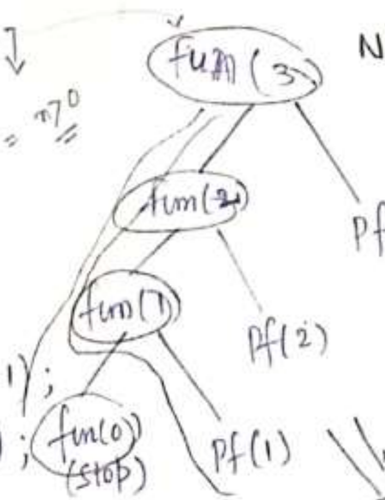
(i) Tree Method :

We draw a recursion-tree and then find out the maximum depth of the tree, which is proportional to the space complexity of recursive programs.

For example :

```
fun(n)
{
    if (n>=1)
    {
        fun(n-1);
        pf(n);
    }
}
```

$= n \geq 0$



Note: Max^m depth of the tree is the number of nodes along the longest path from the root node down to the farthest leaf node.

$\Rightarrow$ Max^m depth of the tree is 4. $= \underbrace{3}_{I/P}+1$

If input is "n", then maximum depth of the tree will be $(n+1)$.

remove constant

$$\boxed{\therefore \text{ Space complexity} = O(n)}$$

(ii) Stack Method to compute Space Complexity of recursive programs :- ②

We just count stack size and multiply with constant amount of ~~time~~ space taken by each recursive call to compute space complexity of recursive programs.

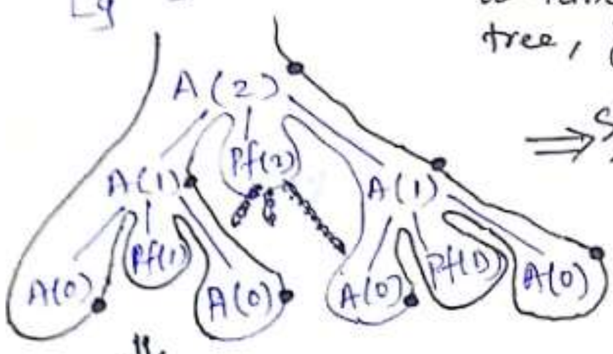
For example :-

```
A(n)
{
  if(n>=1)
  {
    A(n-1);
    Pf(n);
    A(n-1);
  }
}
```

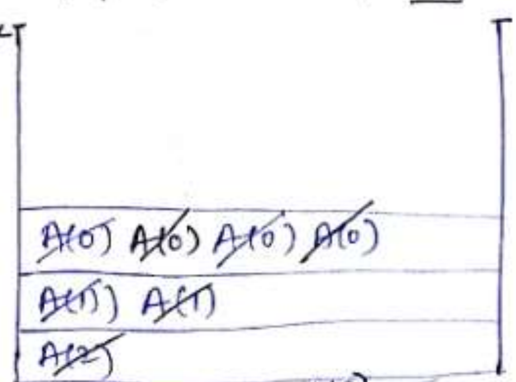
$\Rightarrow$ Steps:

1. Create a recursion - tree
2. use the tree to perform push and pop operations as discussed below:

When we encounter a function call for the first time, just push it inside stack, and when we encounter a function for the last time in the tree, just pop it from stack.



$\Rightarrow$ Stack



$\Downarrow$

max^m depth = 3 (Input+1)

$\boxed{O(n)}$

let every recursive call take "k" cell then space complexity

$= (n+1)k$

Depth of stack

$= \boxed{O(n)}$ (Remove constant)

Depth $= 3\,((2+1)) = (n+1)$ (I/P)

$= \boxed{O(n)}$ Space complexity

Sorting Algorithms and their Analysis

Lecture 12

(sorting algorithms)

By S. khan

(I)

(1) Insertion Sort
(2) Shell sort
(3) Quick sort (v.v.I)
(4) Merge sort
(5) Heap sort (v.v.I)

⟹ working + algo + analysis
Time
Space

(1) Insertion Sort

Insertion_Sort (A)
1. For j = 2 to A.length
2. key = A [j]
3. // Insert A [j] into the sorted sequence A [1...j-1]
4. i = j - 1
5. while i > 0 and A[i] > key
6. A[i+1] = A[i]
7. i = i - 1
8. A [i+1] = key

Working of Insertion Sort Algorithm:

$\boxed{A[\,] = \{8, 5, 9, 0, 7\}}$

The Number of elements is 5; Therefore, we need 4 passes to sort them out.

1	2	3	4	5
8	5	9	0	7

K = 5

First Pass: The first two elements of the given array are compared using Line num 5 of insertion sort algorithm

1	2	3	4	5
8	5	9	0	7

i j Key = 5

$\boxed{\text{while } i>0 \text{ and } A[i] > \text{Key}}$ ⇓ 8, ⇓ 5

↳ True

As the condition is true, swap 8 and 5 using the line numbers 6, 7 and 8.

So, for now, 5 is sorted in a sorted sub-array.

5	8	9	0	7

Second Pass: (Move to the next 2 ele.)

1	2	3	4	5
5	8	9	0	7

i j Key = 9

$\boxed{\text{while } i>0 \text{ and } A[i] > \text{Key}}$ ⇓ 8, ⇓ 9

↳ false

As the condition gets false, line number 8 of algo will be used to place 9 to its original position only.

So, 8 is also sorted in the sorted sub-array along with 5

		9	0	7

3rd Pass:

Now, move to the next two elements and compare them

5	8	9	0	7

i j Key = 0

$\boxed{\text{while } i>0 \text{ and } A[i] > \text{Key}}$ ⇓ 9, ⇓ 0

↳ true.

As the condition is true, move 9 to the place of 0 and decrement i by one.

		9	9	7

i Key = 0

$\boxed{\text{while } i>0 \text{ and } A[i] > \text{Key}}$ ⇓ 8, ⇓ 0

↳ True

As the condition is true, move 8 to the next cell and $i = i-1$

5	8	8	9	7

i Key = 0

Still condition is true, move 5 to the next cell and $i = i-1$.

5	5	8	9	7

$i = 0$ Key = 0

$\boxed{\text{while } i>0 \text{ and } A[i] > \text{Key}}$ ⇓ 0

↳ false

As the condition is false, line number 8 is used to place the value of key to its exact location.

~~0~~	~~5~~	~~8~~	9	7

Fourth Pass: Move to the next two elements:

0	5	8	9	7
			i	j

key = 7

while i > 0 and A[i] > key
9 7

True

As the condition is true, move 9 to the next cell and decrement i by one using the line number 6

0	5	8	9	9
		i		

key = 7

while i > 0 and A[i] > key
8 7

True

As the condition still holds good, move 8 to the next cell and decrement i by one.

0	5	8	8	9
	i			

key = 7

while i > 0 and A[i] > key
5 7

false

As the condition is not true, use line number 8 to place the value of key to its correct location.

~~0~~	~~5~~	~~7~~	~~8~~	~~9~~

$\Longrightarrow$ sorted

Analysis of Insertion Sort

As there is an indirect dependency between inner loop and out loop, we need to unroll inner loop to know how many times it runs (iterates) for its (time comp

** Best case scenario ( ) :

When elements of an array is already sorted in ascending order.

1	2	3	4	5

$\boxed{\text{while } i > 0 \text{ and } A[i] > key}$ → we remove constant term

↳ This line will never be true, but it will run $(n-1)$ times due to outer loop; therefore, time complexity in the best case is $\boxed{\Theta(n)}$.

$\boxed{\Omega(n)}$

*** Worst case scenario :

when elements of an array is sorted in decending order.

5	4	3	2	1

comparison	movements
1	1
2	2
3	3
⋮	
$n-1$	$n-1$

$= \frac{n(n-1)}{2}$

$\boxed{O(n^2)}$

*** Average case :

$\boxed{\Theta(n^2)}$

** Space complexity $= O(1)$

↳ In-place algorithm

Shell-Sort Algorithm

Shell - Sort

```
Shell_sort(A, n)
{
    for ( gap = n/2 ; gap>0 ; gap = gap/2)
    {
        for ( j = gap ; j<n ; j++)
        {
            for ( i = j - gap ; i>=0 ; i = i-gap)
            {
                if ( A[i+gap] > A[i])
                    break;
                else
                    swap( A[i+gap], A[i])
            }
        }
    }
}
```

Working of Shell-Sort Algorithm :

20	30	14	18	32	6	9	4	2
0	1	2	3	4	5	6	7	8

Pass one : $n = 9$, $gap = 9/2 = 4$

0	1	2	3	4	5	6	7	8
20 (i)	30	14	18	32 (j)	6	9	4	2

No swapping will be done as A[j] > A[i] True.

0	1	2	3	4	5	6	7	8
20	30 (i)	14	18	32	6 (j)	9	4	2

A[j] > A[i] false, swap.

0	1	2	3	4	5	6	7	8
20	6	14 (i)	18	32	30	9 (j)	4	2

A[j] > A[i] false, swap

0	1	2	3	4	5	6	7	8
20	6	9	18 (i)	32	30	14	4 (j)	2

A[j] > A[i] false, swap.

0	1	2	3	4	5	6	7	8
20	6	9	4	32 (i)	30	14	18	2 (j)

A[j] > A[i] false, swap

0	1	2	3	4	5	6	7	8
20	6	9	4	2	30	14	18	32

Here, we will have to check left side of i at the gap of 4 as well. Here, 2 > 20 so, swap it.

2	6	9	4	20	30	14	18	32

↳ Array after 1st Pass gets over.

Second Pass:-

0	1	2	3	4	5	6	7	8
2	6	9	4	20	30	14	18	32

Now, $gap = \frac{4}{2} = 2$

A[j] > A[i] True, no swapping will be done.

0	1	2	3	4	5	6	7	8
2	6	9	4	20	30	14	18	32

A[j] > A[i] false, swap.

0	1	2	3	4	5	6	7	8
2	4	9	6	20	30	14	18	32

A[j] > A[i] True, no swap.

0	1	2	3	4	5	6	7	8
2	4	9	6	20	30	14	18	32

Again no swapping will be done as A[j] > A[i]

0	1	2	3	4	5	6	7	8
2	4	9	6	20	30	14	18	32

A[j] > A[i] false, swap.

0	1	2	3	4	5	6	7	8
2	4	9	6	14	30	20	18	32

And also check left side of "i" at the gap of 2. Here value of i is greater than 2 & less than its index, so no further swapping will be done.

0	1	2	3	4	5	6	7	8
2	4	9	6	14	30	20	18	32

Swap

0	1	2	3	4	5	6	7	8
2	4	9	6	14	18	20	30	32

Also check left side of "i" at the gap of 2. Here swapping won't be done.

0	1	2	3	4	5	6	7	8
2	4	9	6	14	18	20	30	32

A[j] > A[i] True, no swapping

0	1	2	3	4	5	6	7	8
2	4	9	6	14	18	20	30	32

↳ Array after second pass gets over.

Third Pass:-

$gap = \frac{2}{2} = 1$

0	1	2	3	4	5	6	7	8
2	4	9	6	14	18	20	30	32

Once gap gets equal to one, it works like insertion sort.

0	1	2	3	4	5	6	7	8
2	4	9	6	14	18	20	30	32

0	1	2	3	4	5	6	7	8
2	4	9	6	14	18	20	30	32

Swap and also check left side of i at the gap of one.

0	1	2	3	4	5	6	7	8
2	4	6	9	14	18	20	30	32

No swapping

0	1	2	3	4	5	6	7	8
2	4	6	9	14	18	20	30	32

No swapping

0	1	2	3	4	5	6	7	8
2	4	6	9	14	18	20	30	32

No swapping

2	4	6	9	14	18	20	30	32

No swapp

2	4	6	9	14	18	20	30	32

No sw

0	1	2	3	4	5	6	7	8
2	4	6	9	14	18	20	30	32

→ sorted

Analysis of Shell-sort (4)

Time Complexity:

Best Case $\Omega(n\log n)$ [Array already sorted]
Worst Case $O(n^2)$
Average Case $\Theta(n\log n)$

Space complexity:
As it is an in-place algorithm, it will take constant amount of space in the memory.

$\boxed{O(1)}$

*** In-place Algorithms:
An algorithm that does not need an extra space equal or more than to its input size. However a small constant extra space is allowed. Such algorithms are: Bubble sort, selection sort, insertion sort, shell-sort, Quick sort, etc.

*** Stable Algorithm:
An algorithm that preserves the order of elements even if two elements are same.

Quick Sort Algorithm

Lecture – 13

By S. Khan

(1)

Quicksort – Algorithm

Array — first index — last index (pivot element index)

```
QUICKSORT(A, p, r)
1. if p < r
2.     q = PARTITION(A, p, r)
3.     QUICKSORT(A, p, q-1)
4.     QUICKSORT(A, q+1, r)
```

Last index of the array (A)

```
PARTITION(A, p, r)
1. x = A[r]
2. i = p-1
3. For j = p to r-1
4.     if A[j] ≤ x
5.         i = i+1
6.         exchange A[i] with A[j]
7. exchange A[i+1] with A[r]
8. return i+1
```

first index of the array A

$\boxed{O(n)}$ $\left[\because 0 \text{ to } n-1 = O(n)\right]$

running time for partition procedure

he key to the algorithm is the PARTITION procedure, which rearranges the sub-array A[P...r] in place. t always selects an element $x = A[r]$ as a pivot element around which to partition the sub-array A[P.

0 (p)	1	2	3	4	5	6	7 (Pivot (r))
2	8	7	1	3	5	6	4

i j

Working of Partion Procedure (2)

$A = \{2, 8, 7, 1, 3, 5, 6, 4\}$

(A)

p							r
0	1	2	3	4	5	6	7
2	8	7	1	3	5	6	4
j							

i (before index 0), $x = 4$

$\boxed{\text{If } A[j] \leq x}$ ⇒ True ($2 \leq 4$)

$i = i + 1$ $(-1 + 1 = 0)$ and exchange $A[i]$ with $A[j]$ and $j = j+1$

(B)

0	1	2	3	4	5	6	7 (r)
2	8	7	1	3	5	6	4
i	j						

$x = 4$

$\boxed{\text{If } A[j] \leq x}$ ⇒ false (8, 4)

just increment "j" by one

(C)

0	1	2	3	4	5	6	7
2	8	7	1	3	5	6	4
i		j					

$x = 4$

$\boxed{\text{If } A[j] \leq x}$ ⇒ false; $j = j+1$ (7, 4)

(D)

0	1	2	3	4	5	6	7
2	8	7	1	3	5	6	4
i			j				

$x = 4$

$\boxed{\text{If } A[j] \leq x}$ ⇒ True; $i = i+1$ (1, 4)

Exchange $A[i]$ and $A[j]$ & $j = j+1$

(E)

0	1	2	3	4	5	6	7
2	1	7	8	3	5	6	4
	i			j			

$x = 4$

$\boxed{\text{If } A[j] \leq x}$ ⇒ True (3, 4)

$i = i + 1$

(F)

0	1	2	3	4	5	6	7
2	1	3	8	7	5	6	4
		i			j		

$x = 4$

$\boxed{\text{If } A[j] \leq x}$ ⇒ false; $j = j+1$ (5, 4)

(G)

0	1	2	3	4	5	6	7
2	1	3	8	7	5	6	4
		i				j	

$x = 4$

$\boxed{\text{If } A[j] \leq x}$ ⇒ false; $j = j+1$

(H)

0	1	2	3	4	5	6	7 (r)
2	1	3	8	7	5	6	4
		i					j

$x = 4$

"j" has reached "r", which makes the condition in the loop false.

$\boxed{\text{For } j = p \text{ to } (r-1)}$ ⇒ false ($7 - 1 = 6$)

Control goes out of the loop body and then executes the line number 7 (Exchange $A[i+1]$ with $A[r]$) and returns $(i+1)$

(I)

0	1	2	3 ($i+1$)	4	5	6	7
2	1	3	4	7	5	6	8

left sub array — sorted one element — Right sub-array.

Now again apply Partition procedure on sub-arrays one by one to sort all elements.

Analysis of Quick sort

Time complexity :

(1) for the best case :

$$T(n) = 2T\left(\frac{n}{2}\right) + \underbrace{O(n)}_{\text{running time for partition procedure}}$$

By master theorem,

$a = 2,\ b = 2,\ k = 1$ and $p = 0$

$\boxed{a = b^k}$ True.

Apply first case $p > -1$

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

$$\boxed{T(n) = \theta(n \log n)}$$

(2) for the worst case : when the "pivot" element is sorted in such a way that it comes either at the first index or last index of the array.

$$T(n) = \underbrace{T(n-1)}_{\text{Decreasing function}} + n$$

By master theorem

$\because a = 1$ & $f(n) = n$
$= O(n * f(n))$
$= O(n^2)$

$$\boxed{T(n) = \theta(n^2)}$$

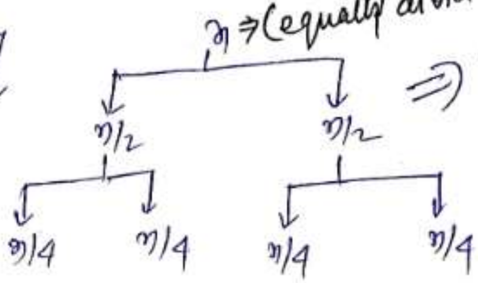
(3) For the average case time complexity :

$$\boxed{\theta(n \log n)}$$

Space Complexity :- (using tree method)

Best Case :-

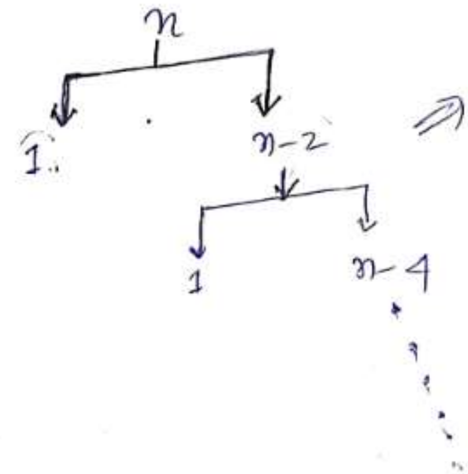
$\boxed{\Omega(\log_2 n)}$



⇒ max^m depth of this tree is $(\log_2 n)$

Worst case :-
(unbalanced)

$\boxed{O(n)}$



⇒ max^m depth is "n"

V.V.I

Merge. sort Algorithm
(out-of-Place algorithm)

```
Merge-sort (A, P, r)            --> T(n)
    if P < r
        q = ⌊(P+r)/2⌋
        Merge-sort (A, P, q)        --> T(n/2)
        Merge-Sort (A, q+1, r)      --> T(n/2)
        Merge(A, P, q, r)           --> Θ(n)
```

```
Merge (A, P, q, r)
    n1 = q - P+1
    n2 = r - q
    let L[1..n1+1] and R [1... n2+1] be new arrays
    For i = 1 to n1
        L[i] = A [P+i-1]
    For j = 1 to n2
        R[j] = A [q+j]
    L [n1+1] = ∞
    R [n2+1] = ∞
    i = 1
    j = 1
    For k = p to r                  --> Θ(n)
        if L [i] ≤ R [j]
            A [k] = L[i]
            i = i+1
        else A [k] = R [j]
            j = j+1
```

Working of Merge Procedure :]

p = 1, q = 4, $q+1$ = 5, r = 8

1	2	3	4	5	6	7	8
0	4	6	7	1	3	5	8

left sub-array (sorted) Right sub-array (sorted)

$n_1 = q - p + 1 (4 - 1 + 1)$
$= 4$ elements

copy

$n_2 = r - q (8 - 4)$
$= 4$ elements

copy

New arrays :

$L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$

L

0	4	6	7	∞

↑i

R

1	3	5	8	∞

↑j

(A)

A =

1	2	3	4	5	6	7	8

↑k

$\boxed{\text{if } L[i] \le R[j]}$ ⇒ True

$A[k] = L[i]$
$i = i + 1$ and $k = k + 1$

(B) L

~~0~~	4	6	7	∞

↑i

R

1	3	5	8	∞

↑j

A =

1	2	3	4	5	6	7	8
0							

↑k

$\boxed{\text{if } L[i] \le R[j]}$ ⇒ false

else $A[k] = R[j]$
$j = j + 1$ and $k = k + 1$

(C) L

~~0~~	4	6	7	∞

↑i

R

~~1~~	3	5	8	∞

↑j

A =

1	2	3	4	5	6	7	8
0	1						

↑k

$\boxed{\text{if } L[i] \le R[j]}$ ⇒ false

else $A[k] = R[j]$
$j = j + 1$ and $k = k + 1$

(D) L =

~~0~~	4	6	7	∞

↑i

R =

~~1~~	~~3~~	5	8	∞

↑j

A =

1	2	3	4	5	6	7	8
0	1	3					

↑k

$\boxed{\text{if } L[i] \le R[j]}$ True

$A[k] = L[i]$
$i = i + 1$ and $k = k + 1$

(E) L

~~0~~	~~4~~	6	7	∞

↑i

R

~~1~~	~~3~~	5	8	∞

↑j

A

1	2	3	4	5	6	7	8
0	1	3	4				

↑k

$\boxed{L[i] \le R[j]} \Rightarrow$ false

else

$A[k] = R[j]$

$j = j+1$ and $k = k+1$

(F) L $[\cancel{0} \mid \cancel{1} \mid 6 \mid 7 \mid \infty]$ R $[\cancel{1} \mid \cancel{3} \mid \cancel{5} \mid 8 \mid \infty]$

1	2	3	4	5	6	7	8
0	1	3	4	5			

$\boxed{\text{if } L[i] \le R[j]} \Rightarrow$ True

$A[k] = L[i]$

$i = i+1$ and $k = k+1$

(G) L= $[\cancel{0} \mid \cancel{1} \mid \cancel{6} \mid 7 \mid \infty]$ R= $[\cancel{1} \mid \cancel{3} \mid \cancel{5} \mid 8 \mid \infty]$

1	2	3	4	5	6	7	8
0	1	3	4	5	6		

$\boxed{\text{if } L[i] \le R[j]} \Rightarrow$ True

$A[k] = L[i]$

$i = i+1$ and $k = k+1$

(H) L= $[\cancel{0} \mid \cancel{1} \mid \cancel{6} \mid \cancel{7} \mid \infty]$ R= $[\cancel{1} \mid \cancel{3} \mid \cancel{5} \mid 8 \mid \infty]$

A=

0	1	3	4	5	6	7	

(2)

$\boxed{\text{if } L[i] \le R[j]} \Rightarrow$ false

else

$A[k] = R[j]$

$j = j+1$ and $k = k+1$

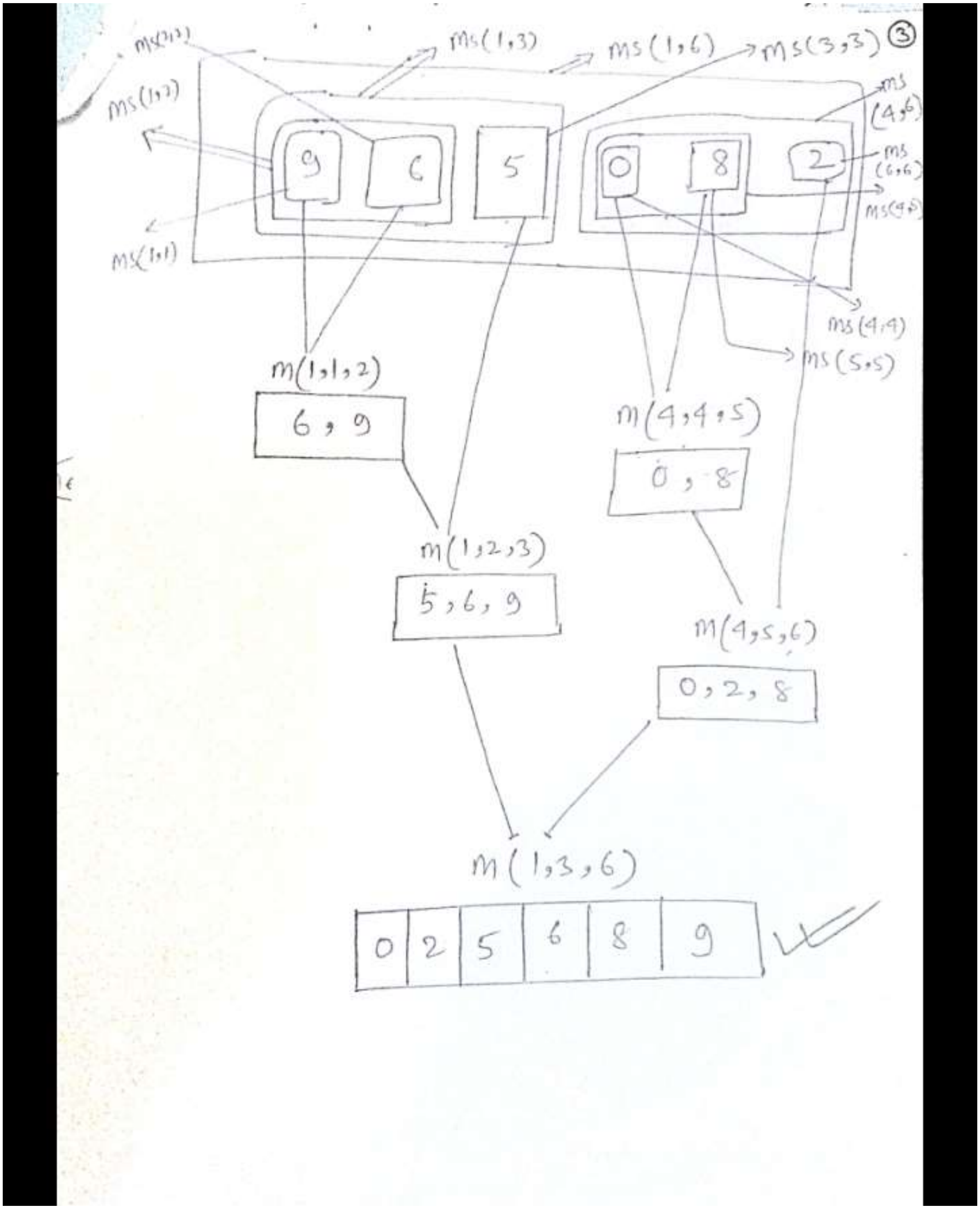
(I) L $[\cancel{0} \mid \cancel{1} \mid \cancel{6} \mid \cancel{7} \mid \infty]$ R $[\cancel{1} \mid \cancel{3} \mid \cancel{5} \mid \cancel{8} \mid \infty]$

1	2	3	4	5	6	7	8
0	1	3	4	5	6	7	8

sorted in ascending order.

k (out of lap)

Apply Merge sort on the array { 9, 6, 5, 0, 8,5} and also write down its time complexity



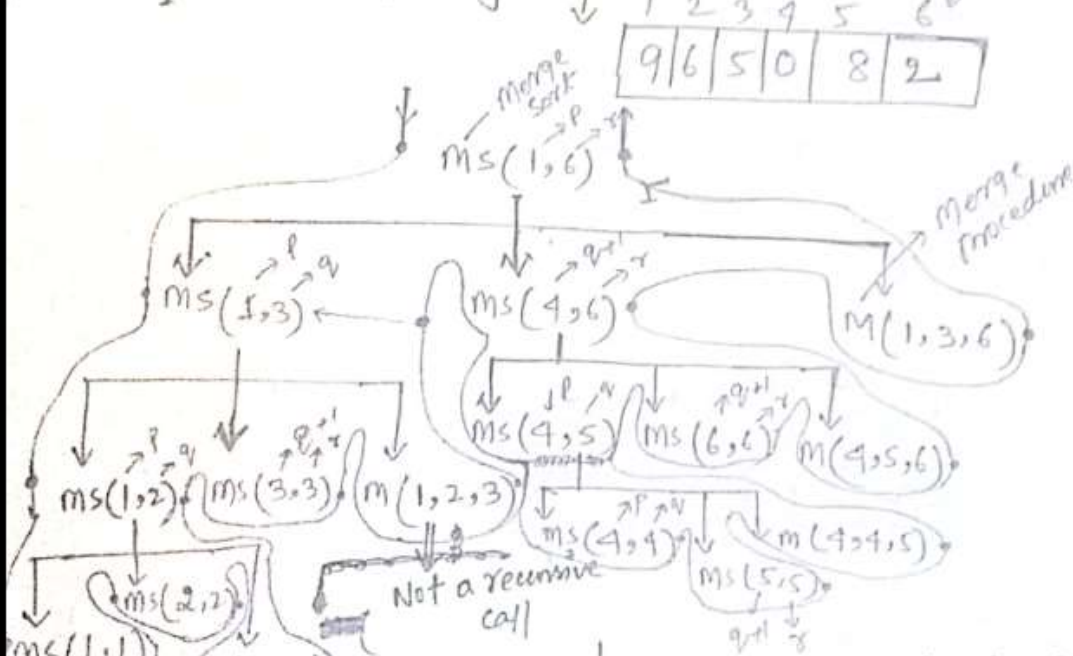
Analysis of Merge Sort Algo

Time Complexity :-

$$T(n) = 2 * T\left(\frac{n}{2}\right) + O(n)$$

$\boxed{\theta(n \log n)}$ → Best as well as Worst case.

** Space Complexity :-



Space required for merge procedure is $O(n)$. And space required for recursive calls is $O(\log n)$

∴ $\boxed{\text{Space complexity} = O(n)}$

Since $O(n) > O(\log n)$

Heap Sort Algorithm

S. Khan

Heap Sort Algorithm (In-place Algo)

```
HEAPSORT(A)
BUILD_MAX_HEAP(A)                        => O(nlogn)
For i = A.length down to 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size - 1
    MAX_HEAPIFY(A, 1)
```

Time ↓ $O(n\log n)$

Space ↓ $O(1)$ ↳ if MAX-HEAPIFY() is implemented using loop.

```
BUILD_MAX_HEAP(A)
A.heap-size = A.length
For i = ⌊A.length/2⌋ down to 1
    MAX_HEAPIFY(A, i)
```

Time ↓ ⇒ $O(n)$

Space ↓ $O(\log n)$

```
MAX_HEAPIFY(A, i)
l = LEFT(i) → 2i
r = Right(i) → 2i+1
If l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
If r ≤ A.heap-size and A[r] > A[largest]
    largest = r
If largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

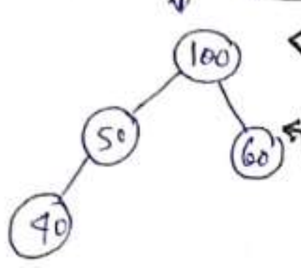
⇒ Time ↓ $O(\log n)$

Space ↓ $O(\log n)$

Heap : (Binary) It is an almost complete binary tree.
↳ a kind of binary tree

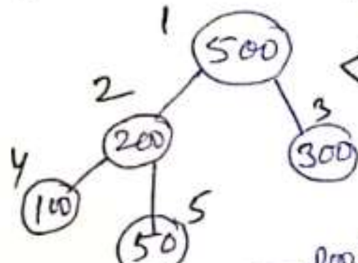
For example :

100	50	60	40



⇐ Insertion takes place level by level and left to right order at each level. It is not a complete binary tree as its all leaves do not have the same depth.

Max-heap : the key present at the root node must be greater than or equal among the keys present at all of its children. For example,



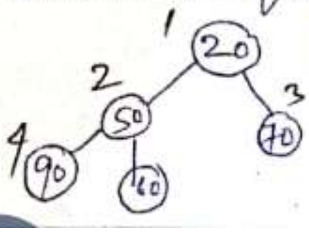
⇐ we have three roots (1, 2, 3) in the tree, and all of them have greater key than their children.

Roots / Root of the tree:

- Node – 1 : key = 500, which is greater than all.
- Node – 2 : key = 200, which is greater than its children.
- Node – 3 : key = 300, which is greater than its children

Min-heap : the key present at the root node must be less than or equal among the key present at all of its children

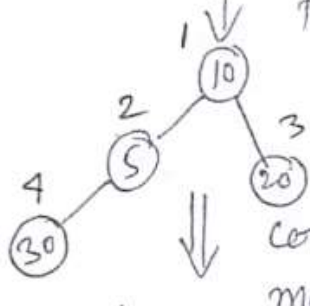
20	50	70	90	60



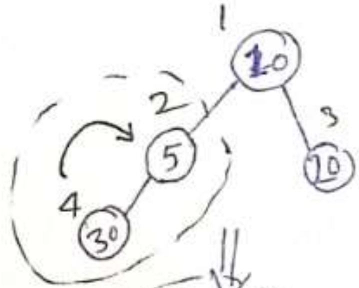
⇐ Roots : node 1 : Its key is the least among all.
node 2 : Its key is the least among its children--

Illustrate the operation of HEAPSORT on the array $A = \{10, 5, 20, 30\}$.

Sol^n: Convert it into an almost binary tree



Convert it into max-heap using max-hepify procedure. We start from a non-leaf node which has greater node ~~number~~ index.

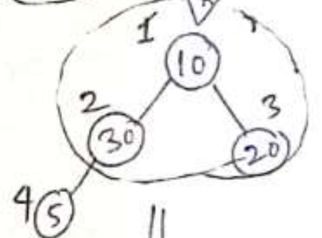


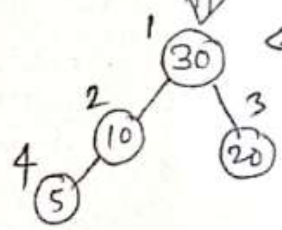
formula to find out non-leaves:

$\lfloor \frac{n}{2} \rfloor$ to 1 (n = no of nodes)

$\lfloor \frac{5}{2} \rfloor$ to 1

$\boxed{2, 1}$ ⟹ Non-leaf nodes.

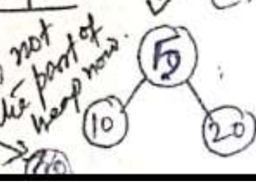




⟸ max-heap

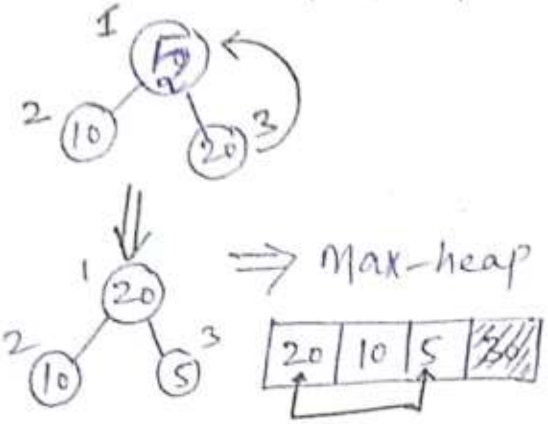
1	2	3	4
30	10	20	5

Step 2: Exchange root with the last node.

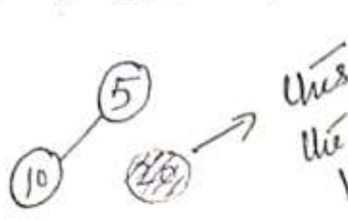


This is not the part of heap now.

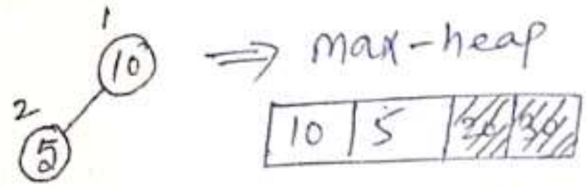
Step 3 :- Convert it into max-heap



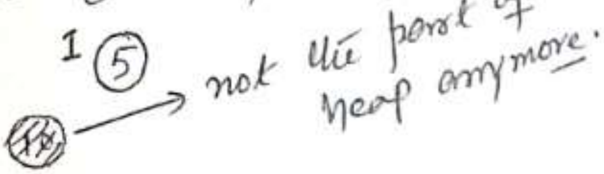
Step 4 :- Exchange root with the last node.

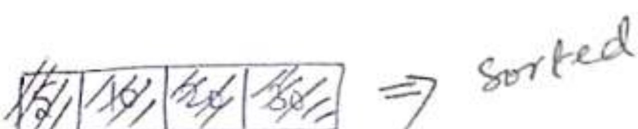


Step 5 :- Convert it into max-heap.



Step 6 :- Exchange root with the last node.





Heapsort runs from $i = \boxed{A.length \text{ to } 2}$ index only; therefore when we have only one node left, it is already sorted.

Q What do you understand by a stable sort? Name two stable sort algorithms.

A sorting algorithm is said to be stable if two objects having equal keys appear in the same order in sorted output as they appear in the input data set. For example, Insertion and Counting sorts.

Q. Define in-place sorting algorithm.

It is a type of sorting algorithm that rearranges the elements of an array or list without requiring additional memory space proportional to the size of the input.

Q Describe the difference between the average-case and the worst-case analysis of algorithm, and give an example of an algorithm whose average-case running time is different from worst case analysis.

Q. Compare sorting algorithms in a tabular form.

Algorithm	**Average Case**	**Worst Case**	**Memory**	**Stable**
Bubble Sort	n^2	n^2	1	yes
Insertion sort	n^2	n^2	1	yes
Shell sort	-	$n\log^2 n$	1	No
Merge sort	nlogn	nlogn	n	yes
Heapsort	nlogn	nlogn	1	No
Quicksort	nlogn	n^2	logn	Depends

Sorting in linear time O(n)

[Non-comparison-based sorting algorithms]

Non-comparison-based sorting algorithms do not rely on pairwise comparisons between elements to determine their relative order. Instead, they exploit properties of the input data, such as the range of values, to achieve efficient sorting. These algorithms are often used when the range of possible input values is known and limited, making them more efficient than comparison-based algorithms in certain scenarios. Here are some common non-comparison-based sorting algorithms:

1. Count sort
2. Radix sort
3. Bucket sort

Counting-Sort Algorithm

Counting Sort assumes that each of the "n" input elements is an integer in the range from 0 to "k," where "k" is an integer representing the range of values.

Counting-Sort-Algorithm

Counting-Sort(A, B, k)

1. let c[0...k] be a new array // Auxiliary array
2. For i = 0 to k
3. c[i] = 0 // Initialize array "c" with all 0
4. For j = 1 to A.length
5. c[A[j]] = c[A[j]] + 1 // update array "c".
6. For i = 1 to k
7. c[i] = c[i] + c[i-1] // Sum the array "c" [Prev] + [current]
8. For j = A.length downto 1
9. B[c[A[j]]] = A[j] // Resultant array "B"
10. c[A[j]] = c[A[j]-1]

Analysis:

$$\boxed{\begin{aligned} \text{Time} &= O(n) \\ \text{Space} &= O(n) \end{aligned}}$$

Q. Write down the Counting-Sort algorithm and illustrate the operation of Counting Sort on the array A = {6, 4, 8, 4, 5, 1}.

Solution:

Counting-Sort Algorithm

Counting-Sort (A, B, k)

1. let c[0...k] be a new array // Auxiliary array
2. For i = 0 to k
3. c[i] = 0 // Initialize array "c" with all 0
4. For j = 1 to A.length
5. c[A[j]] = c[A[j]] + 1 // update array "c".
6. For i = 1 to k
7. c[i] = c[i] + c[i-1] // Sum the array "c" [Prev] + [Current]
8. For j = A.length downto 1
9. B[c[A[j]]] = A[j] // Resultant array "B"
10. c[A[j]] = c[A[j]-1]

Analysis: ↓

$$\boxed{\begin{aligned} \text{Time} &= O(n) \\ \text{Space} &= O(n) \end{aligned}}$$

Sol^n: Given A =

6	4	8	4	5	1

B =

⇒ Resultant array

Step-01 A =

1	2	3	4	5	6
6	4	8	4	5	1

K = 8 (max^m ele) ⟹ 0 to 8 (Range)

Step-02 C =

0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0

Step-03 – update the array c using line no 4 and 5.

for j = 1 to A.length

C [A[j]] = C [A[j]] + 1

C [A[1]] = C [A[1]] + 1

C [6] = C [6] + 1

C [6] = 0 + 1

C [6] = 1

⟹ In the same way, update the array "C".

C =

0	1	2	3	4	5	6	7	8
0	1	0	0	~~1~~ 2	1	1	0	1

Step-04 Sum the array c's elements using the line numbers 6 and 7.

First ele same and then [prev] + [current]

C =

0	1	2	3	4	5	6	7	8
0	~~1~~ 0	1	1	~~3~~ ~~2~~ 1	~~4~~ 3	5	5	~~6~~ 5

Step-05 (Resultant Array) ⟹ using Line no 9 and 10.

1	2	3	4	5	6
1	4	4	5	6	8

Radix-Sort Algorithm

Lecture-17

By S.Khan

Radix Sort

Base

Radix_sort (A, d)

// Each key in A[1...n] is a d-digit integer

// Digits are numbered 1 to d from right to left

for i = 1 to d

use a stable sort algo to sort "A" on digit "i".

Illustrate the operation of Radix-sort on the array {804, 26, 5, 64, 52, 1}.

Step 1: As the largest number (804) contains three digits, make other numbers three digits by appending zeros(s).

Hundreds place, Tens Place, unit place

8	0	4
0	2	6
0	0	5
0	6	4
0	5	2
0	0	1

Step 2: Take the unit place and apply a "stable sort".

Tens place

0	0	1
0	5	2
8	0	4
0	6	4
0	0	5
0	2	6

Relative ordering of the same elements does not change after sorting.

Step 3: Take the Tens place and apply the stable sort.

Hundreds place

0	0	1
8	0	4
0	0	5
0	2	6
0	5	2
0	6	4

Step 4: Take the hundreds place and apply the Stable sort.

0	0	1
0	0	5
0	2	6
0	5	2
0	6	4
8	0	4

⟹ sorted.

If we have more than one elements in the array which are same, then after sorting their relative ordering does not change. For example.

same elements — sorted

2_A	4	1	2_B

⟹

1	2_A	2_B	4

left, Right — left, Right → Relative ordering is same

Analysis:

Time:

$$\underbrace{d}_{\text{digits} \Rightarrow \log_b l}\left(O(\underbrace{n}_{\text{i/p}}+\underbrace{b}_{\text{base}})\right) \text{ if } \underbrace{l}_{\text{largest number}} = O(n^k)$$

then $\cancel{k}\log_b n \cdot O(n+b)$

$$\Theta \boxed{(n\log_b n)}$$

Bucket-Sort Algorithm

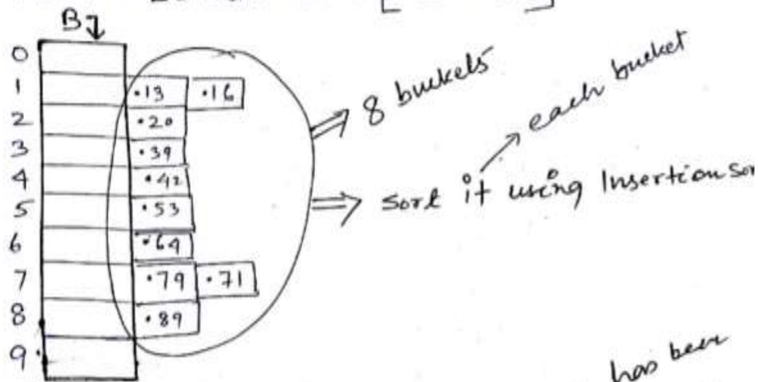
```
Bucket_Sort(A)
let B[0...n-1] be a new array
n = A.length
for i = 0 to n-1
    make B[i] an empty list
for i = 1 to n                              i/p elements.
    insert A[i] into list B[⌊n*A[i]⌋]
for i = 0 to n-1
    sort list B[i] with insertion sort
Concatenate the lists B[0], B[1], ..., B[n-1] in order.
```

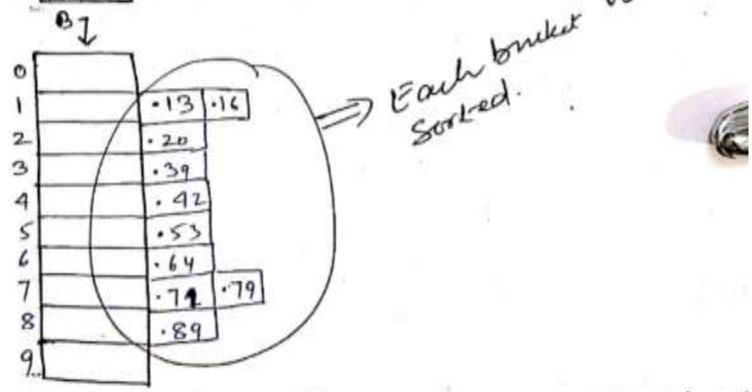
$i =$	1	2	3	4	5	6	7	8	9	10 (n)
$A =$	.79	.13	.16	.64	.39	.20	.89	.53	.71	.42

For $i = 1$ to n

Insert $A[i]$ into list $B[\underbrace{\lfloor n \cdot A[i] \rfloor}_{\text{floor}}]$

$B[\lfloor 10 \times .79 \rfloor] \Downarrow B[\lfloor 7.9 \rfloor] \Downarrow B[7]$





Now, concatenate the lists $B[0], B[1], \ldots, B[n-1]$ in order

.13	.16	.20	.39	.42	.53	.64	.71	.79	.89

Unit-02

Red-Black Tree

It is a Binary Search Tree (BST) with one extra bit of storage per node: its color, which can be either red or black.

Properties of RB Tree:

1. Every node is either red or black.
2. The root is black.
3. Every leaf node (NIL) is black.
4. If a node is red, then both its children are black. It means we need to avoid red-red (RR) conflict.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

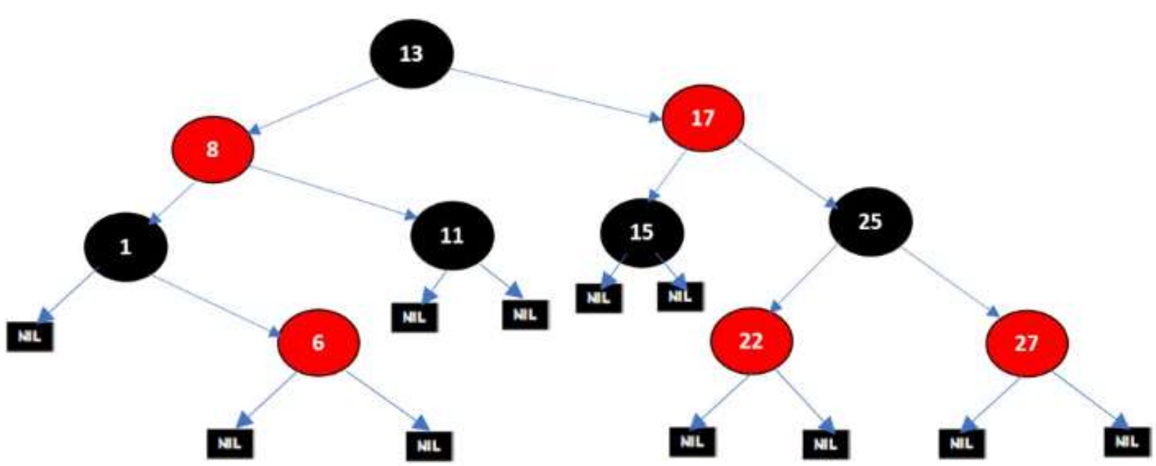


Fig – Red Black Tree

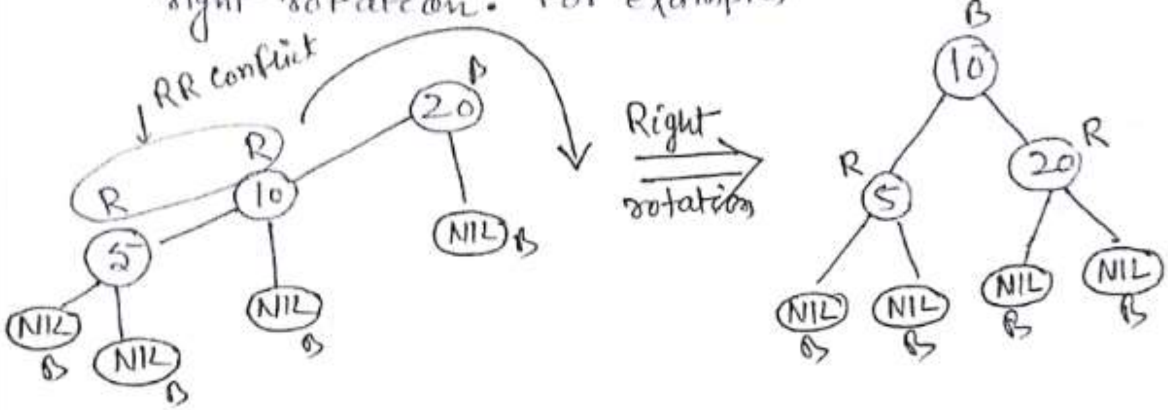
Q1. Explain various rotations in an RB Tree.

We have four types of rotations in RB tree like AVL tree:

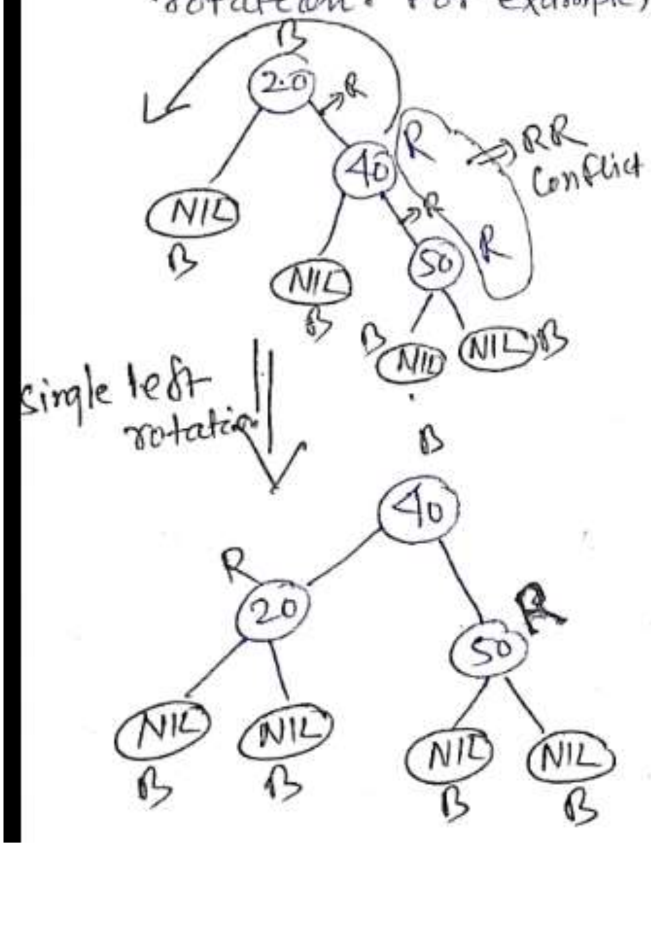
1. Left-Left (LL) problem: Needs a single right rotation
2. Right-Right (RR) problem : Needs a single left rotation
3. Left-Right (LR) problem: Needs one left and then one right rotations.
4. Right-Left (RL) problem: Needs one right and one left rotations.

Note: We have to recolor only those nodes which are involved in rotation. When dealing with RL or LR rotation, we have to recolor only last rotated nodes.

(I) Left-Left (LL) Problem : It requires a single right rotation. For example,

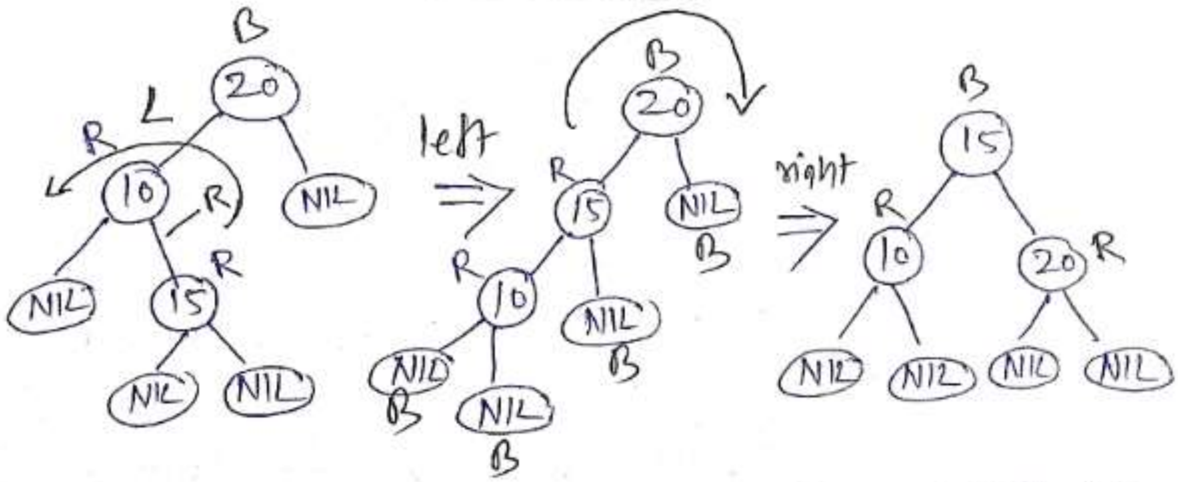


(2) Right-Right (RR) Problem : It requires a single left rotation. For example,



(3) Left-Right Problem :
(L-R)
It requires left rotation and then right rotation. For example.

LR = L R

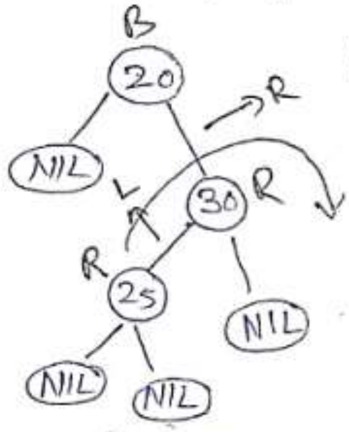


Note: ⇒ Last rotated nodes (15) and (20) are recolored.

(4) Right-left (RL) Problem:

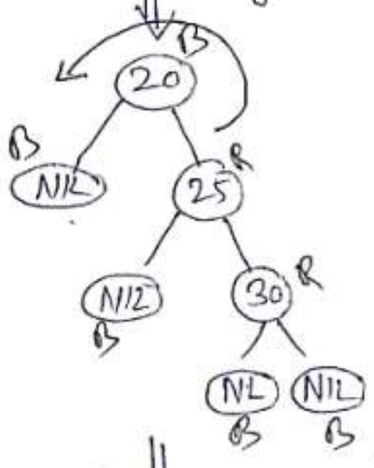
RL = RL

It requires one right rotation and then one left rotation. For example.

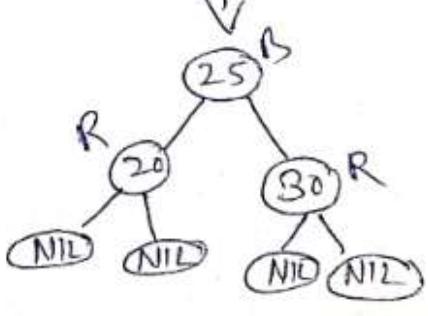


Note: Two adjacent Rs are not allowed.
⇓
Called RR Conflict.

⇓ Right rotation



⇓ ——— left rotation



Q2. Compare the properties of AVL tree with RB Tree.

Basis of comparison	Red Black Trees	AVL Trees
Lookups	Red Black Trees has fewer lookups because they are not strictly balanced.	AVL trees provide faster lookups than Red-Black Trees because they are more strictly balanced.
Colour	In this, the color of the node is either Red or Black.	In this, there is no color of the node.
Insertion and removal	Red Black Trees provide faster insertion and removal operations than AVL trees as fewer rotations are done due to relatively relaxed balancing.	AVL trees provide complex insertion and removal operations as more rotations are done due to relatively strict balancing.
Storage	Red Black Tree requires only 1 bit of information per node.	AVL trees store balance factors or heights with each node thus requiring storage for an integer per node.
Searching	It does not provide efficient searching.	It provides efficient searching.
Uses	Red-Black Trees are used in most of the language libraries like map, multimap, multiset in C++, etc.	AVL trees are used in databases where faster retrievals are required.
Balance Factor	It does not gave balance factor	Each node has a balance factor whose value will be 1,0,-1
Balancing	Take less processing for balancing i.e.; maximum two rotation required	Take more processing for balancing

Q3. What are advantages of an RB Tree?

Advantages:

Balanced Structure: Ensures efficient operations (O(log n)) by self-balancing the tree.

Predictable Performance: Rules maintain consistent performance regardless of data.

Fast Insertions/Deletions: Efficient for frequent changes.

Sorted Order: Supports sorted data and range queries.

Search Efficiency: Quick search operations due to balanced structure.

Q4. Write an algorithm for insertion of keys into an RB Tree and also insert the following keys <5,16,22, 25, 2,10,18,30,50,12,1> into an empty RB Tree.

Algorithm for insertion

1. If the tree is empty, create a new node as the root node with the color “black”.
2. If the tree is not empty, insert the new node with the color “red”.
3. If the parent of the new node is “black”, exit.
4. If the parent of new node is “red”, check the color of parent’s sibling (the uncle of the newly inserted node).

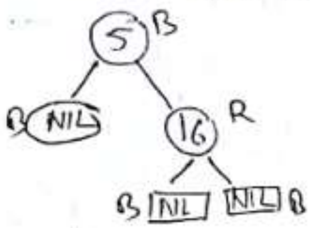
   4(a) If its color is black or Null (no uncle), perform suitable rotations and recolor only the last rotated nodes.
   4(b) If its (uncle’s) color is red, recolor the following nodes:
   1. Uncle
   2. Parent
   3. Grandfather (if it is not the root of the tree).

Check if the tree is an RB tree or not. If not, consider the grandfather as a newly inserted node and repeat step -4.

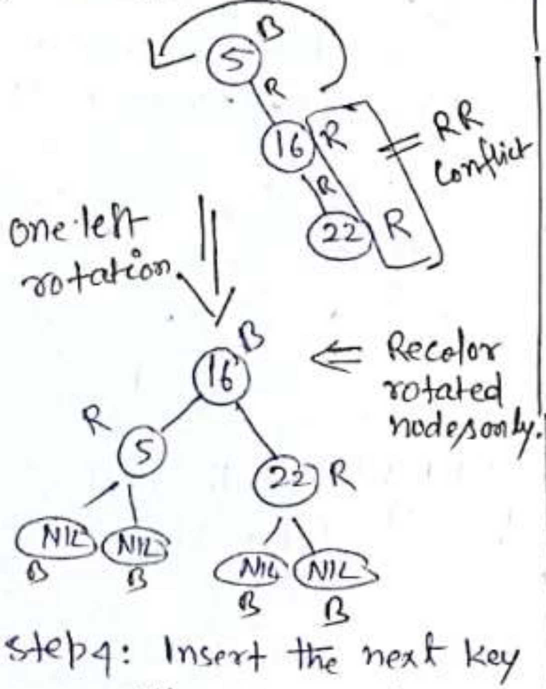
Insert the following keys in an empty RB tree.
↳ $\langle 5, 16, 22, 25, 2, 10, 18, 30, 50, 12, 1 \rangle$

Step 1: Insert 5 in an empty RB tree with color black.

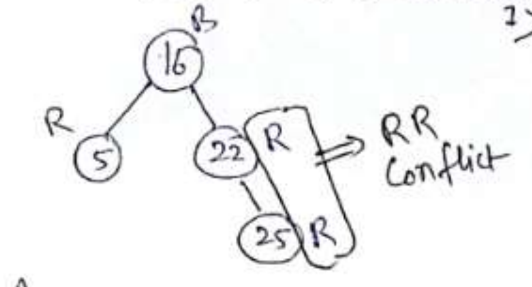
Step2: Insert the next element(16) with color red.



Step 3: Insert the next element (22) with color red.



Step4: Insert the next key 25 with color red.

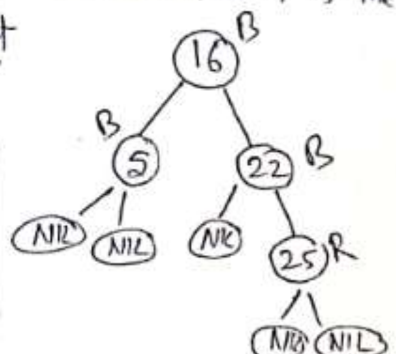


As uncle (5) is red, recolor the following:

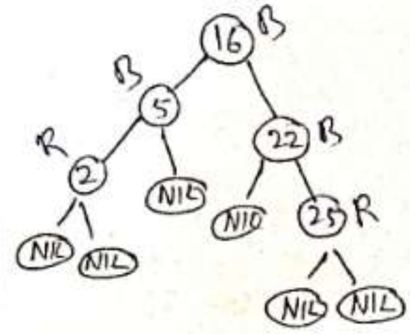
(I) Parent

(II) Uncle

(III) Grandfather will not be recolored as it is the root of the tree.

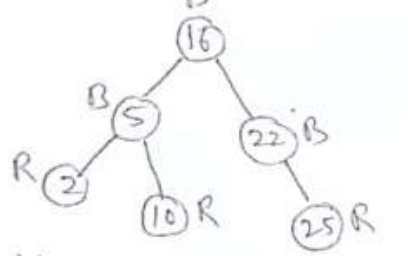


Step 5: Insert the next key (2) with color red.



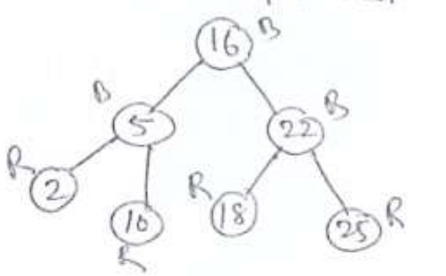
No change as no conflict(RR).

Step 6: Insert the next key (10) with color red.



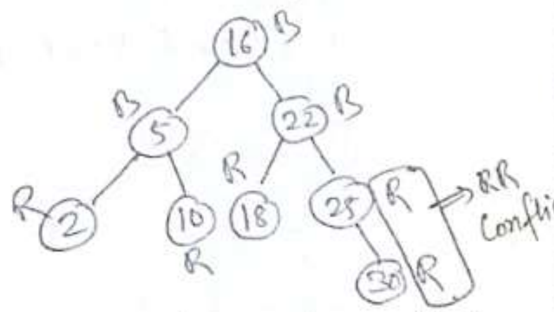
No Change as no RR conflict.

Step 7: Insert the next key (18) with color red.



No change as no RR Conflict

Step 8: Insert the next key (30) with color red.

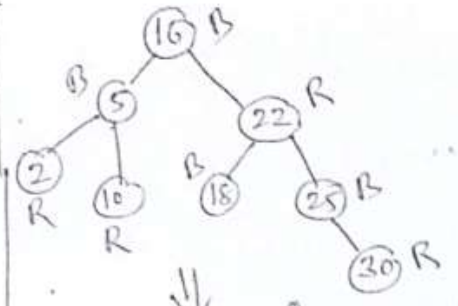


Uncle of 30 is red(18), recolor the following:

(i) Parent (25)

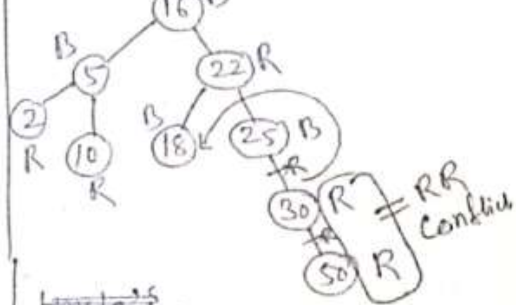
(ii) Uncle (18)

(iii) Grandfather (22)



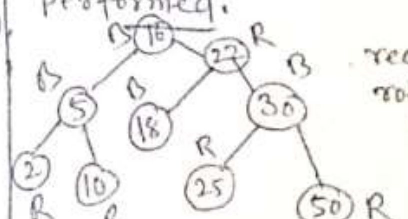
⇓ Having no conflict now.

Step 9: Insert the next key (50) with color red.



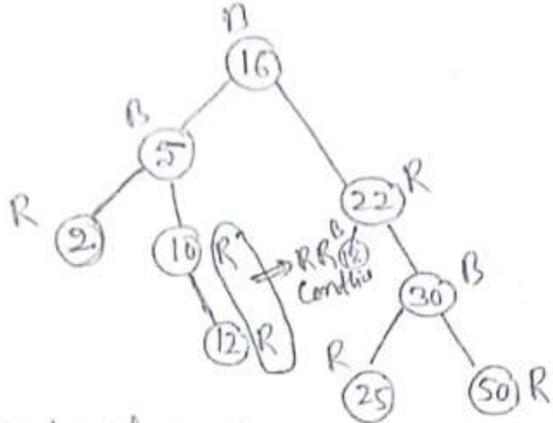
~~Uncle's~~

As 50 has no uncle (NULL), rotation will be performed.

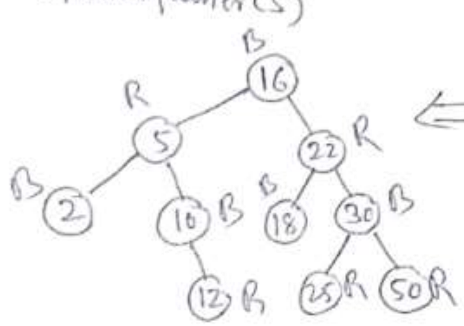


recolor only rotated nodes.

Step 10: Insert the next key (12) with color red.

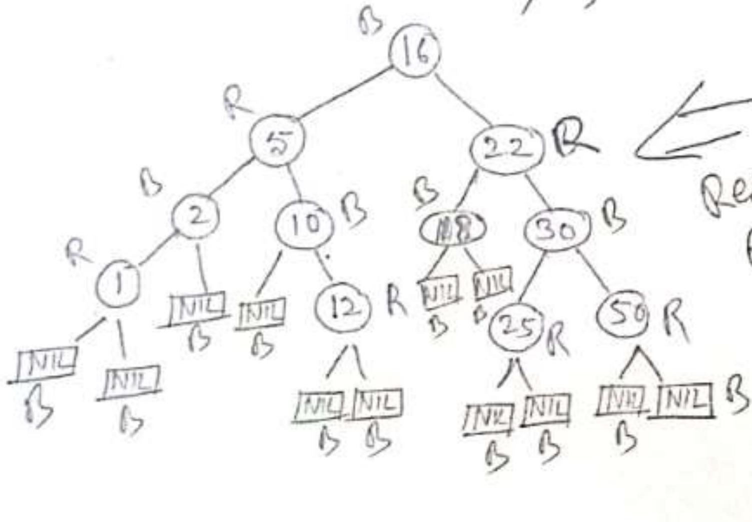


As uncle of 12 is red (2), just recolor the following:
(i) uncle (2)
(ii) Parent (10)
(iii) Grandfather (5)



⟸ No RR conflict now.

Step 11. Insert the next key (1) with color red.



⟸ Resultant RB tree

Deletion in RB-Tree

Steps:

Step 1: Choose the node to be deleted like in BST. Binary Search tree.

Step 2: Perform the Standard Case:

Case 1: If the node to be deleted is red, just delete it and exit.

Case 2: If Double Black (DB) is root, remove DB.

Case 3: If DB's sibling is black, and both its nephews are also black, then

(a) Remove Double Black (DB).

(b) Make sibling red

(c) If parent was originally red, make it black else, make parent DB and reapply suitable case.

Case 4: If DB's sibling is red, then

(a) Swap color of DB's parent & DB's sibling.

(b) Rotate parent in DB's direction.

(c) Reapply a suitable case (as DB still exits)

Case 5: If DB's sibling is black, far nephew is black and near nephew is red, then:

(a) Swap color of DB's sibling and DB's near nephew.

(b) Rotate DB's sibling in the opposite direction of DB. (c) Apply case-6.

Case 6: If DB's sibling is black and far nephew is red, then

(a) Swap color of DB's parent & DB's sibling

(b) Rotate DB's parent in DB direction

(c) Change color of red nephew to black.

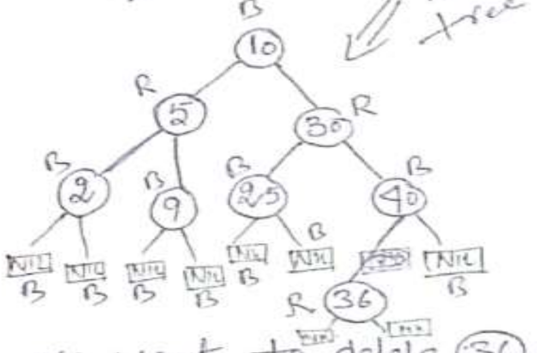
6(d) Remove DB

Illustrate all 6 cases with suitable examples.

Case 1 [If the node to be deleted is red, just delete it and exit.]

Ex:

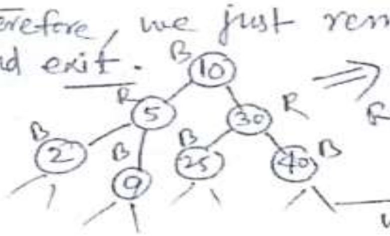
RB tree



We want to delete (36), which is a leaf node according to Binary search tree (BST)

We reach node 36 using BST and delete it as it is a leaf node of the BST.

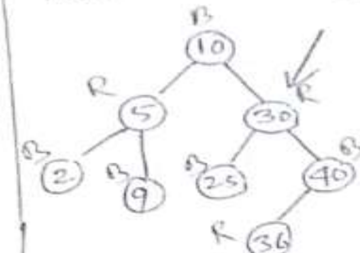
Case 1 follows here, because color of the node 36 is red. therefore, we just remove it and exit.



New RB-tree

understand

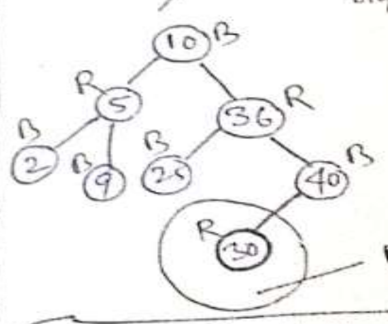
Now delete (30).



As 30 is not a leaf node, so we can not directly delete it according to the deletion rule of BST. In such a case, we replace the node either by inorder predecessor or by inorder successor of the node

inorder predecessor of 30 is 25 [Biggest ele in the left sub-tree]

inorder successor of 30 is 36 [Smallest element in right sub tree]



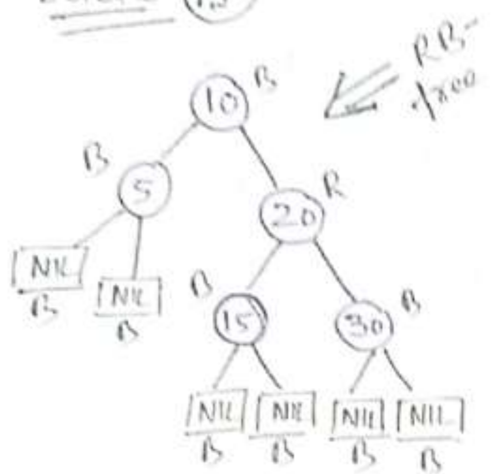
Delete it

Case 1 follows here too

↳ deleted node is (red)

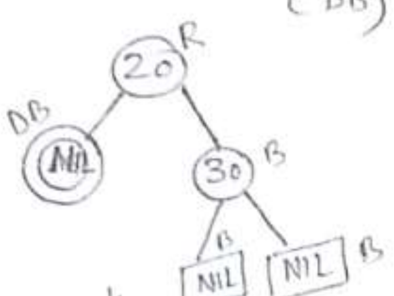
Case (03)

Delete (15)



Here, node to be deleted is "black" and both its sibling nephews are also black.
When we replace 15, which is black by its inorder predecessor or inorder successor [NIL], it becomes double black.

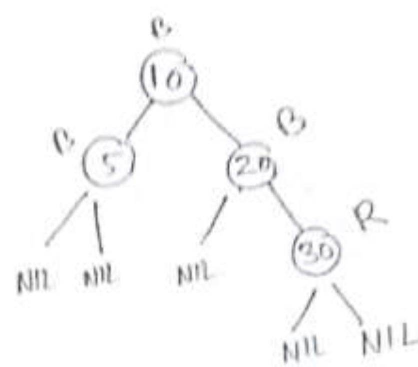
 ⇒ double back (DB)



⇓

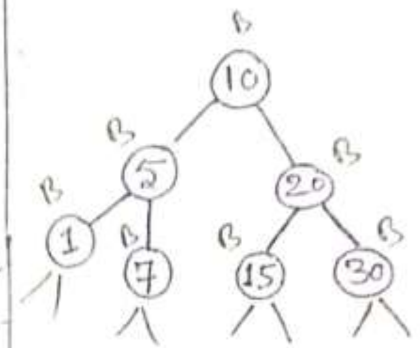
(a) Remove DB
(b) Make sibling (30) red
(c) If Parent is originally red, make it black.

(2)

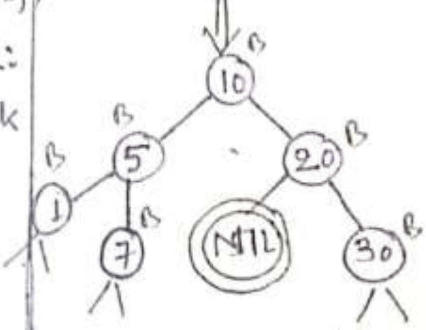


Case 03

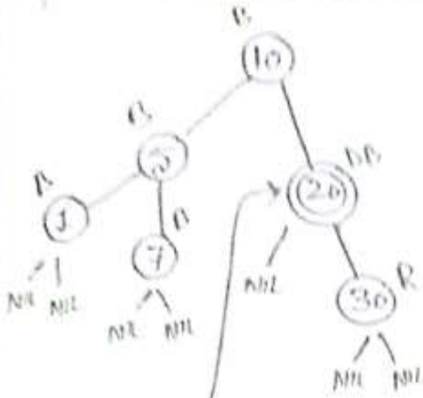
Delete (15)



⇓



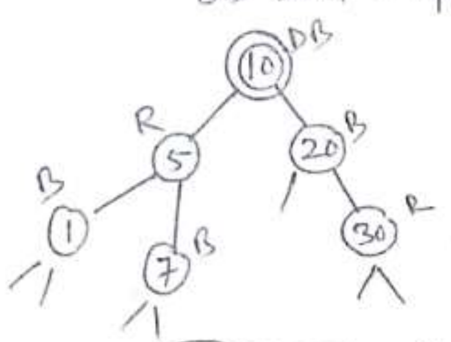
(a) Remove DB
(b) Make sibling red
(c) Make Parent DB and reapply suitable case.
(Since Parent (20) is Black).



We have got another double black node, so we need to get rid of it also.

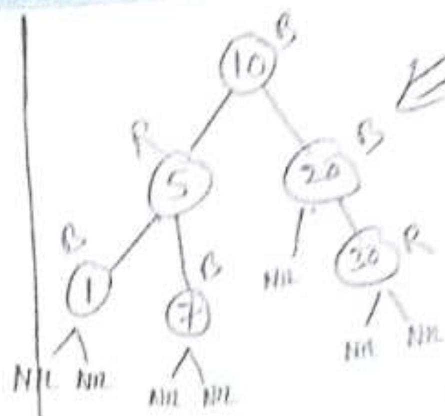
Again Case-03 will be applied.
↳ DB's (20) sibling is black (5) and both its nephews (1) and (7) are also black, then

(a) Remove DB (make it single black)
(b) Make sibling red
(c) Make parent of (20) DB and reapply suitable case

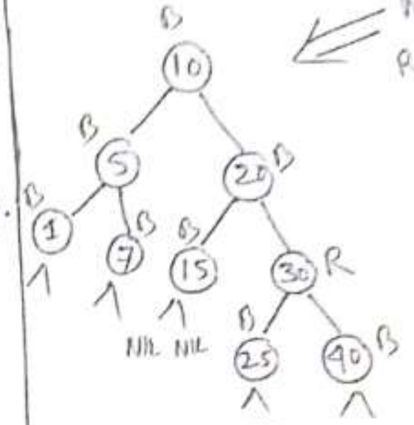


Now $\boxed{\text{Case-2}}$ will be applied.
As DB is root, remove DB — make it single black.

RB tree

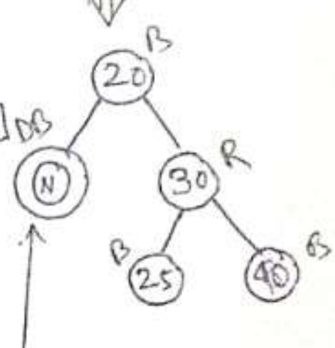


New RB tree

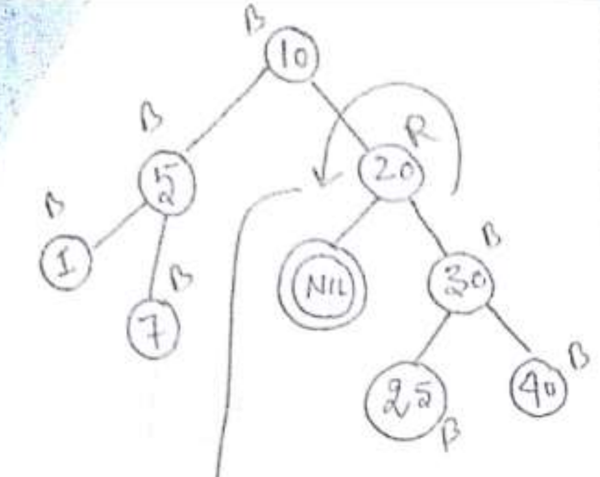


Delete (15)

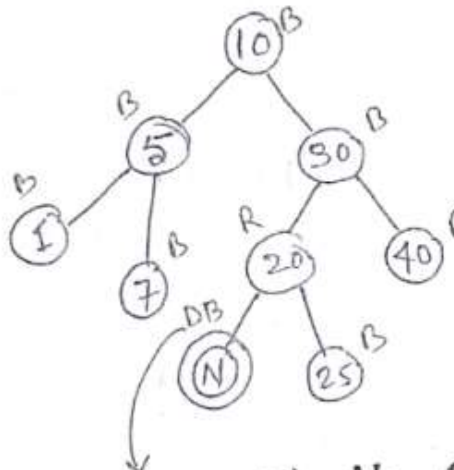
⇓



Case 4: DB's sibling is red, Then:
(a) Swap color of DB's parent & DB's sibling.
(b) Rotate parent in DB's direction
(c) Reapply a suitable case.

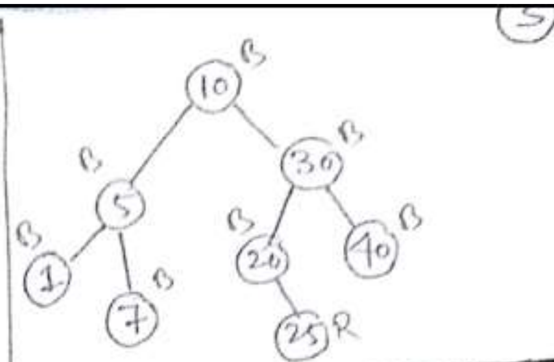


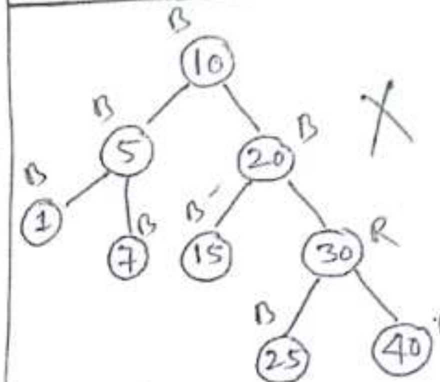
Now, rotate parent in DB's direction.

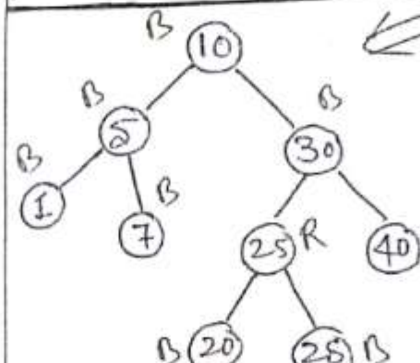


DB still exits. Case-3 will be applied here: I

(a) Remove DB (make it single B)
(b) Make sibling red.
(c) Make parent black.



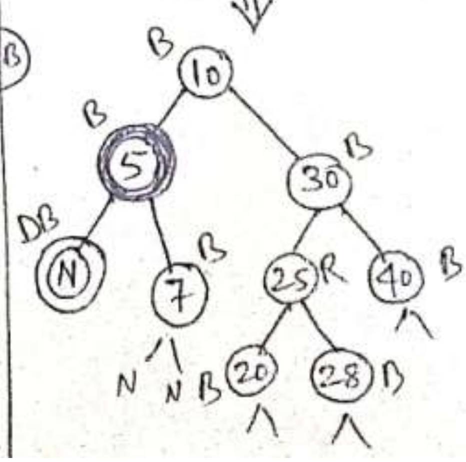




⇐ A new RBT

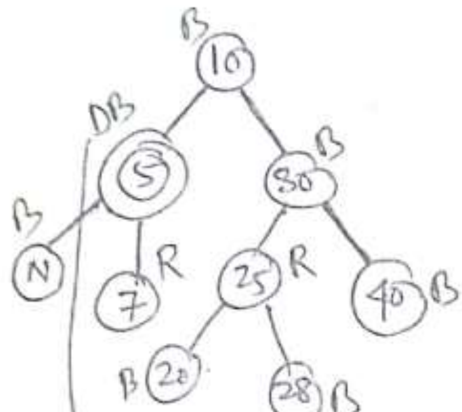
Delete (1)

⇓



Case-3 will be applied:

(a) Remove DB
(b) Make sibling red
(c) Make parent DB
and reapply suitable case.

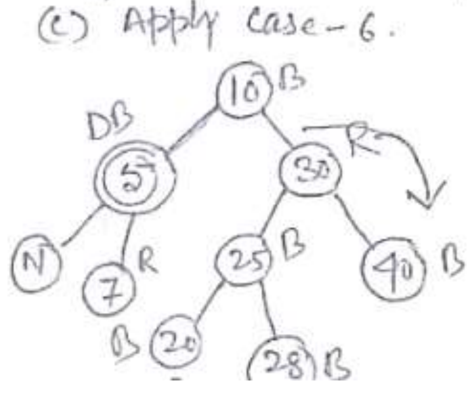


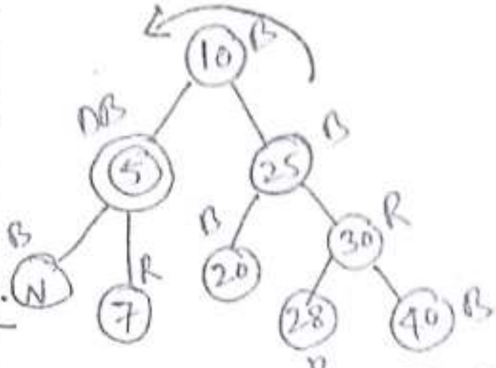
⇓

Case-5 will be applied

Since DB's sibling (30) is black, far nephew is black (40) and near nephew (25) is red.

(a) Swap color of DB's sibling and DB's near nephew.
(b) Rotate DB's sibling in the opposite direction of DB
(c) Apply Case-6.

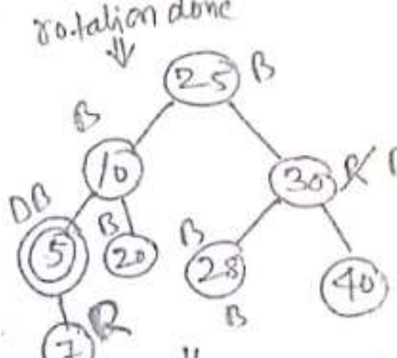




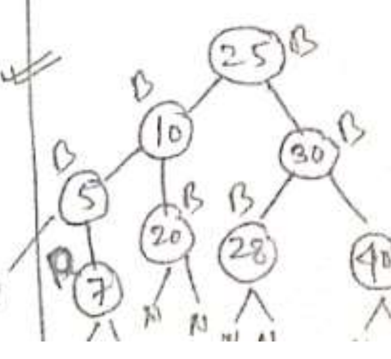
Now apply (Case-6) [Note- Every time, we apply case-5, we have to apply case-6 as well].

(a) Swap color of DB's parent and DB's sibling (Here, both have same color)
(b) Rotate DB's parent in DB direc
(c) Change color of red nephew
(d) Remove DB (Make it single B).

rotation done



⇓ Remove DB (make it single)



resultant RB-tree

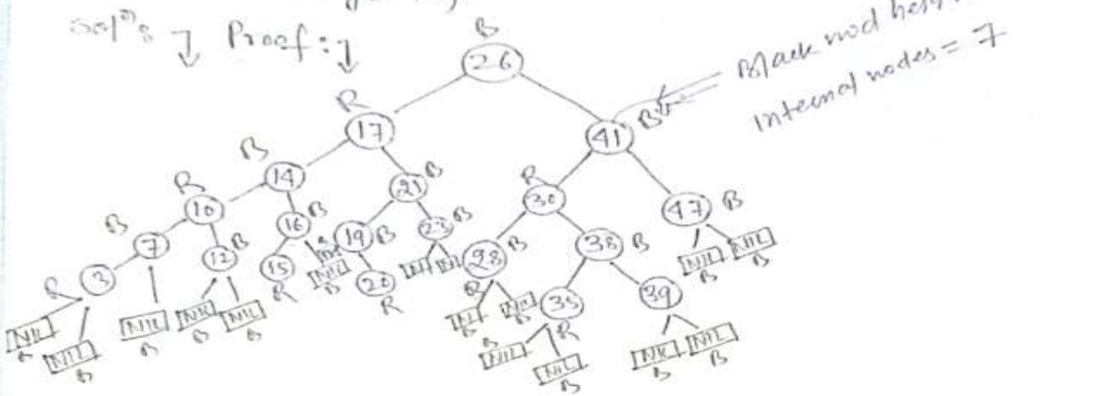
Q4. Explain about double black node problem in RB tree.

When a black node is deleted and replaced by a black child, the child is marked as double black. The main task now becomes to convert this double black to single black.

Q5. Construct an RB Tree, and let h be the height of the tree and n be the number of internal nodes in the tree. Show that $h <= 2\log_2(n+1)$.

Lemma: A red-black tree with "n" internal nodes has height at most $2\log(n+1)$.

Sol^n:] Proof:]



* The subtree rooted at any node "x" contains at least $\boxed{2^{bh(x)} - 1}$ internal nodes. [bh ⇒ Black node height]

Let's verify it. Consider node 41, which has bh = 2 (as it has two black node in its ~~sub~~ simple path from 41 to NIL)

$2^2 - 1 = 3$ true as it is ≤ 7 (internal nodes of 41.)

* We prove this claim by induction on the height of x. If the height of x is 0, then x must be a leaf, and the subtree rooted at x indeed contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes.

* For the inductive step, consider a node x that has positive height and is an internal node with two children; then each child has a black-height of either bh(x) or bh(x)−1, depending on whether its color is red or black, respectively.

Since the height of a child of x ~~is~~ any node less than the height of x itself, we can apply the inductive hypothesis to conclude that each child has at least $2^{bh(x)-1}-1$ internal nodes.

Thus, the subtree rooted at x contains at least

$$(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1 = 2^{bh(x)}-1 \text{ internal nodes}$$

To complete the proof of the lemma, let "h" be the height of the tree. According to property 4 of RB tree, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently, the black-height of the root must be at least h/2; thus

$$n \geq 2^{h/2}-1.$$

Moving the 1 to the lef-hand side and taking logarithms on both sides yields:

$$\log(n+1) \geq h/2 \qquad [\because \log_2 2 = 1]$$

$$\text{or } \boxed{h \leq 2\log(n+1)} \text{ proved.}$$

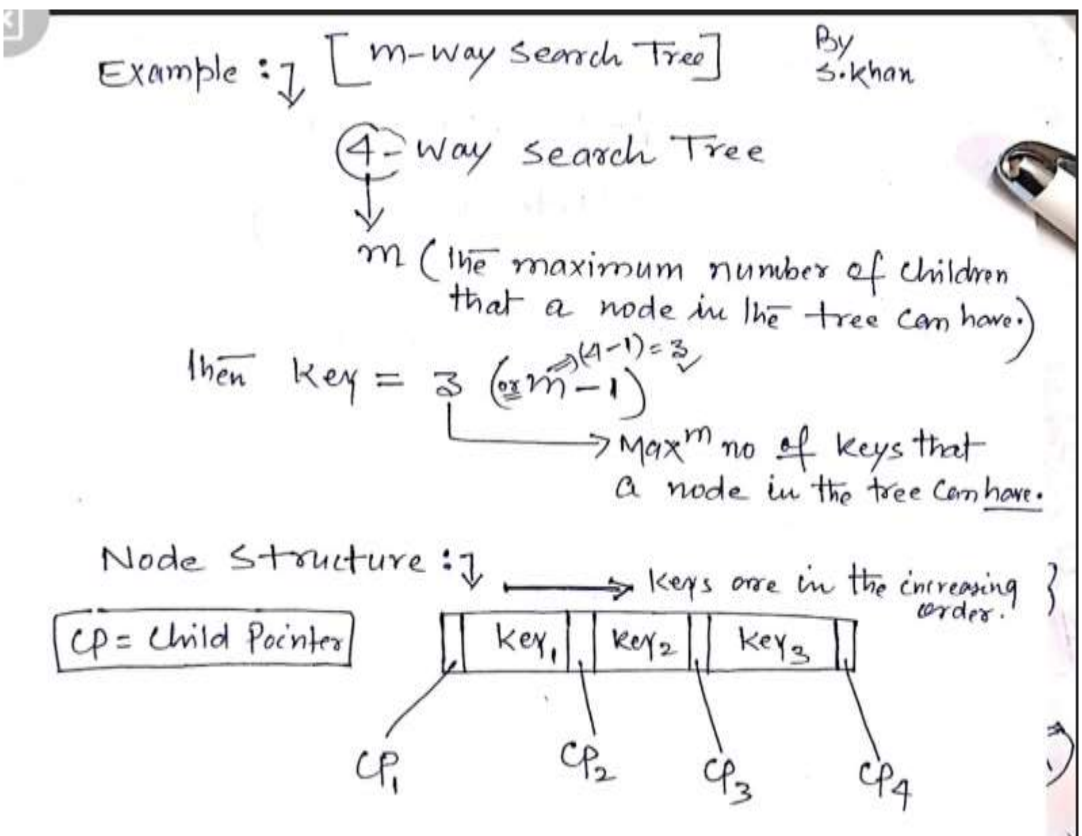
B-Trees

A B-tree is a self-balancing m-way search tree with the following restrictions:

1. The root node must have at least two children.
2. Every node, except for the root and leaf nodes, must have at least ⌈m/2⌉ children.
3. All leaf nodes must be at the same level.
4. The creation process is performed bottom-up.

Properties of m-way search tree:

1. m (degree/order): Maximum number of children (or child pointers)
2. m-1 keys : Maximum keys per node
3. All keys are arranged in ascending order.



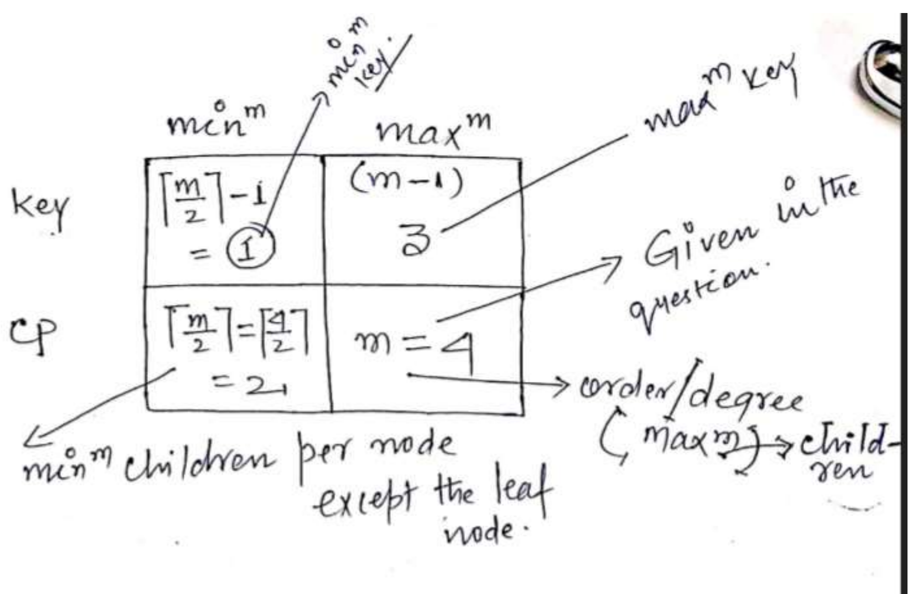
Note— An m-way search tree, also known as an m-ary search tree, does not have strict height control during its construction. Depending on the order (m) and the specific keys inserted, an m-way search tree can grow to a height of "n," where "n" is the number of keys inserted into the tree. To address this issue and maintain a more balanced structure, B-trees were introduced. B-trees are a type of m-way tree with specific restrictions and properties designed to keep their height under control and ensure balanced branching.

Insertion of keys in a B-tree

Type-01 We are given some keys with a degree.

For example: insert the keys: 12, 21, 41, 50, 60, 70, 80, 30, 36, 6, 16 into an empty B-tree with degree 4.

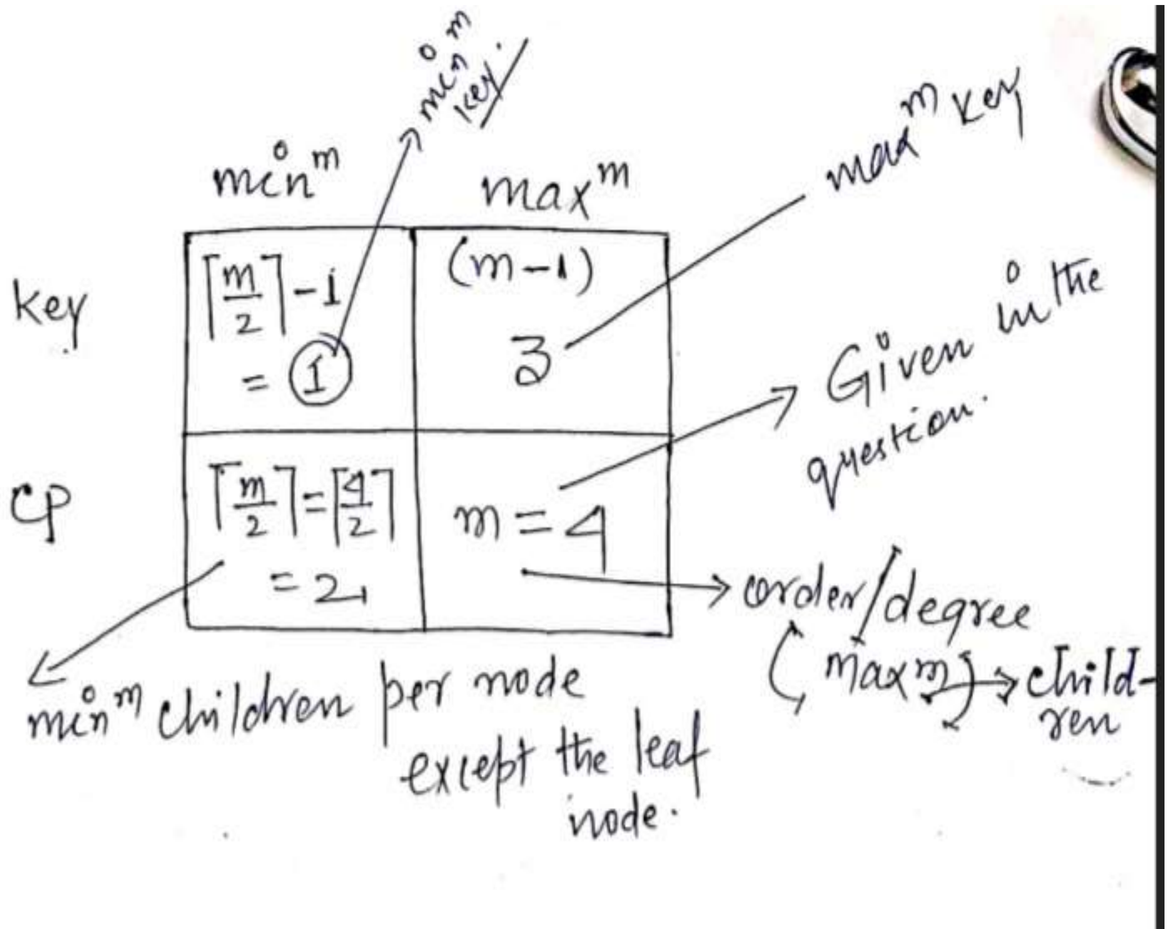
Create a table following the properties of the B-tree



Type-02 We are given some keys with a maximum degree.

For example: Using the maximum degree m= 4, insert the following sequence of integers 10, 25, 20, 35, 30, 55, 40, 45, 50, 60, 75, 70, 65, 80, 85,90 into an initially empty B-Tree.

Create a table following the properties of the B-tree

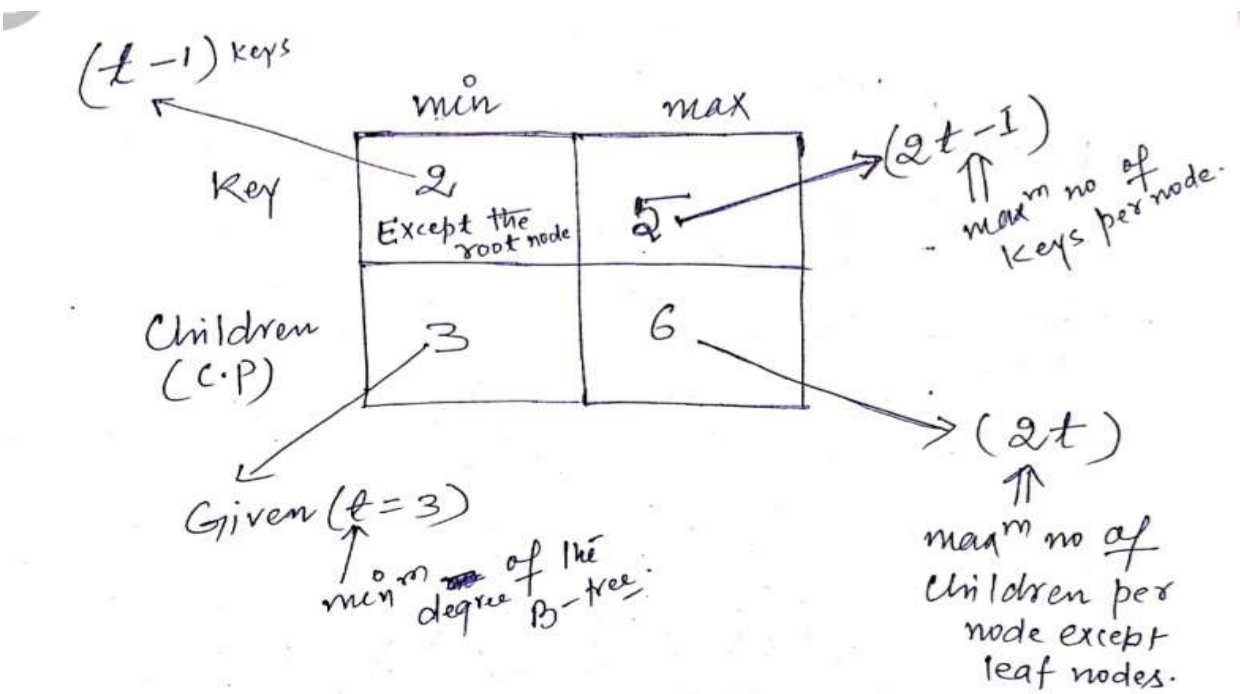


Type-03 We are given some keys with a minimum degree.

For example: Using the minimum degree t= 3, insert the following sequence of integers 10, 25, 20, 35, 30, 55, 40, 45, 50, 60, 75, 70, 65, 80, 85,90 into an initially empty B-Tree.

Create a table following the properties of the B-tree

Note— The maximum number of keys that can be stored in a particular node of a B-tree with a minimum degree of t is 2t-1. Therefore, based on the question, the maximum number of keys per node is 5.

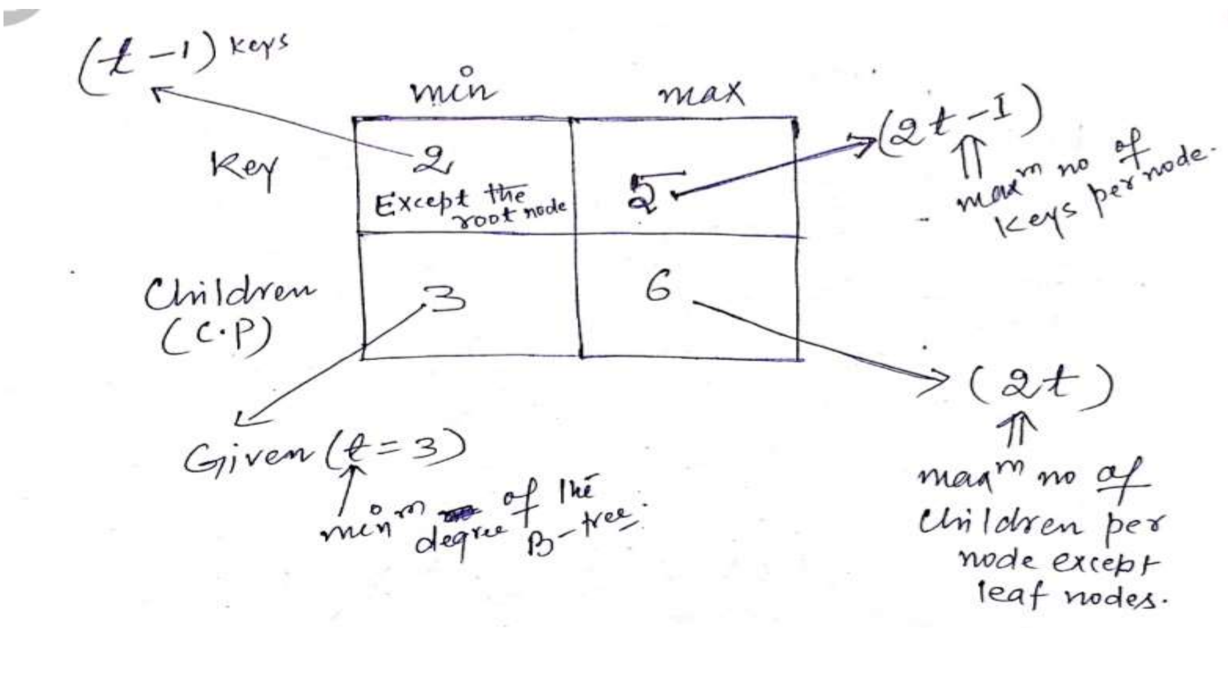


Note—The simplest B-tree occurs when t=2. Every internal node then has either 2, 3, or 4 children, and we have a 2-3-4 tree.

	min	max
Key	1	3
Children	2 (Given "t")	4

Q. Using the minimum degree t= 3, insert the following sequence of integers 10, 25, 20, 35, 30, 55, 40, 45, 50, 60, 75, 70, 65, 80, 85,90, 100,110,120, 112, 114, 120 into an initially empty B-Tree.

Solution—



I

Step-01: Insert the first integer, '10', into the B-tree. Since the tree is initially empty, it becomes the root node.

10

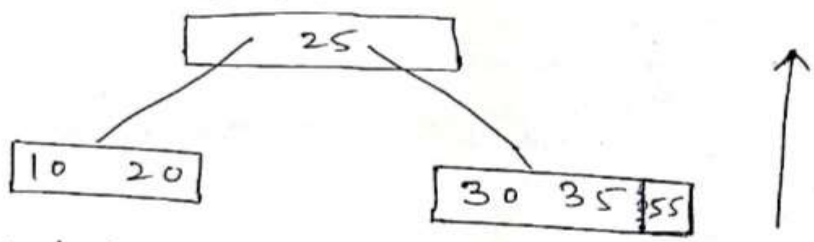
Step-02: Insert the next integer, '25'. Since the root node is not full (max^{m} keys allowed is '5') insert it in the increasing order. → see table

10 25

Step-03: In the same way, we can insert the next 3 keys.

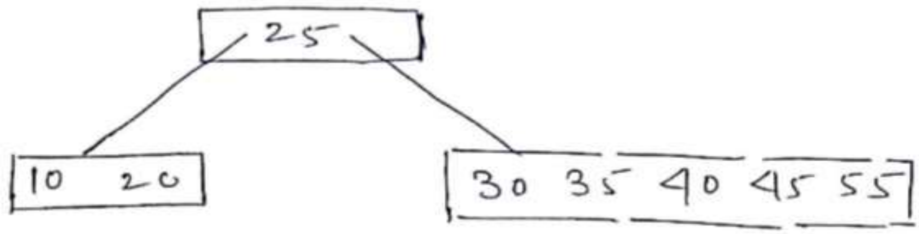
10 20 25 30 35

Step-04: Insert '55'. The root node is now full, so we need to split it up. The middle key (25) moves up to become the new root, and the left and right parts become two children.

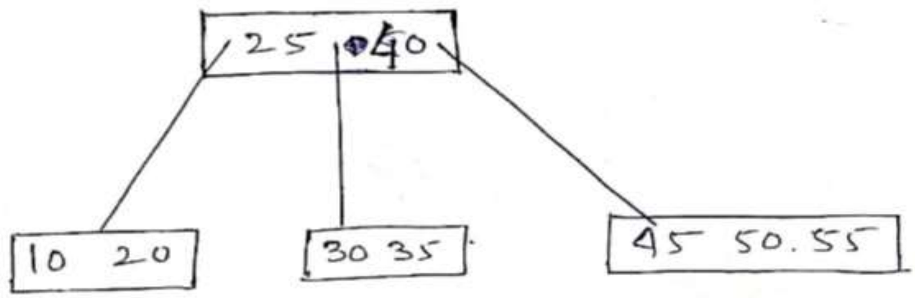


Step-05: Insert 40 and 45. We follow bottom up approach, so, we can insert the next two keys (40 & 45) in the right child of the root node.

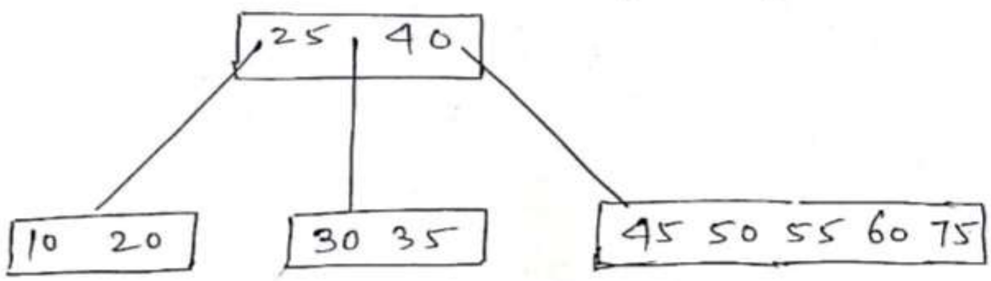
21



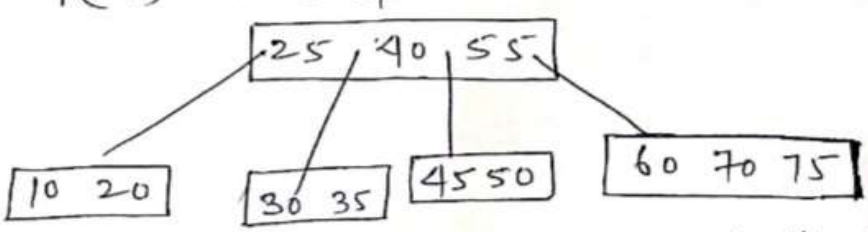
Step-06 : Insert 50. The node is now full (5 keys), so we need to split it. The middle key (40) moves up :-



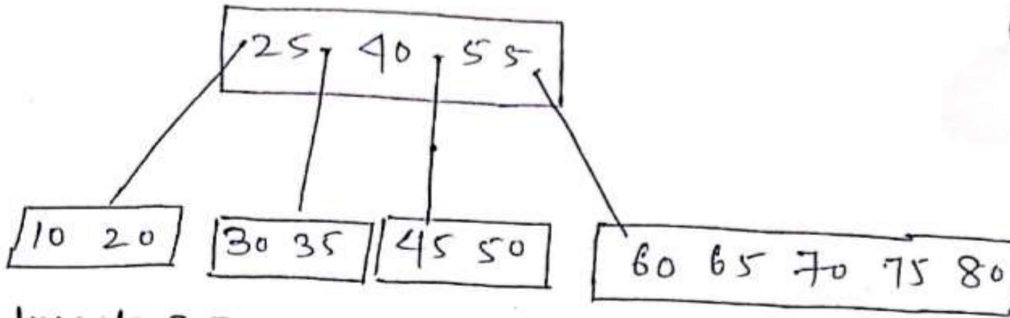
Step-07 : Insert the next two keys (60 and 75).



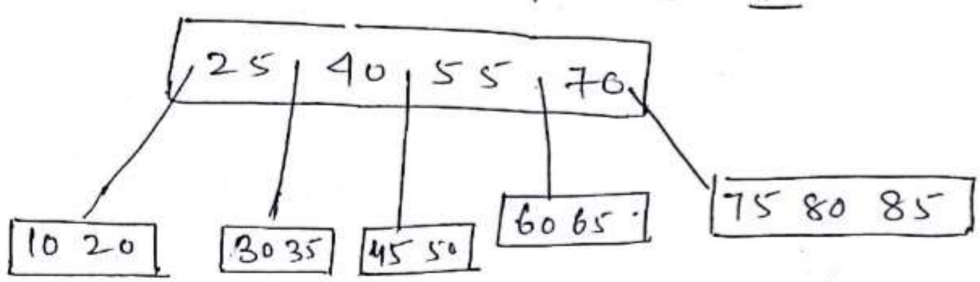
Step-08 : Insert 70. We need to split the node. The middle Key (55) moves up.



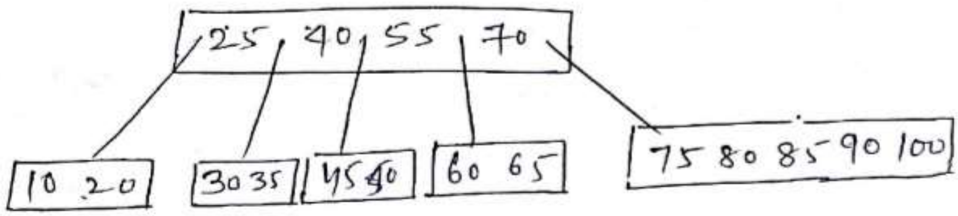
Step-9 : Insert the next two keys (65 and 80) without any issue.



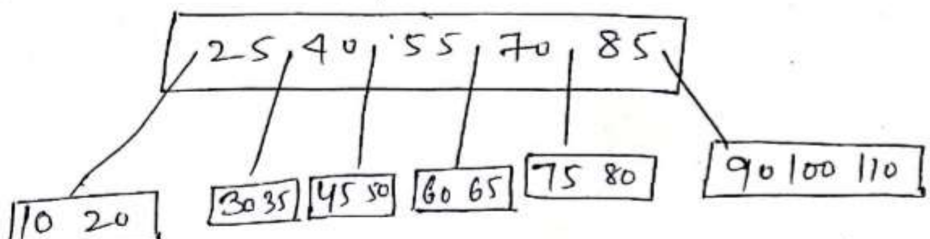
Step-10: Insert 85. we need to split the node.



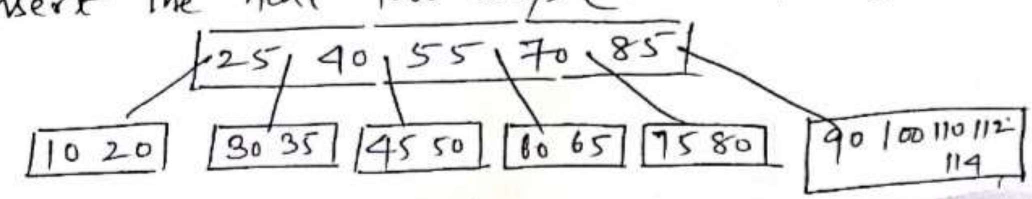
step 11: Insert the next two keys (90 and 100).



Step 12: Insert the next key 110. we need to ~~insert~~ split.



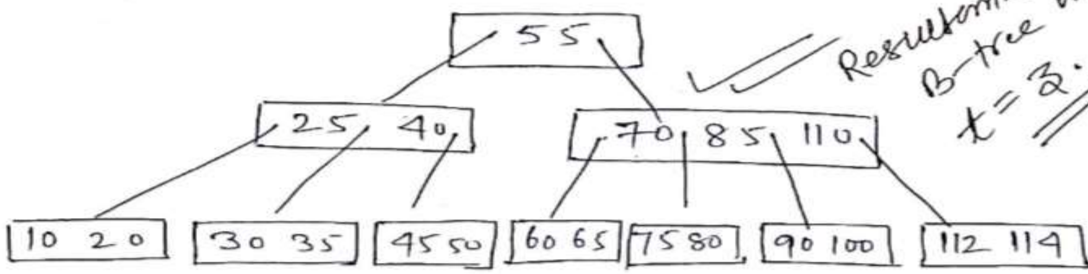
step 13: Insert the next two keys (112 and 114).



Step 14: Insert 120. We need to split the node. 110 will move up.

root node ⇒ | 25 40 55 70 85 | 110

⇒ It is also full, so we need to split it also.



Resultant B-tree with $t = 3$.

⇒ Root node can have one key.

⇒ All nodes except the root node have min^m 2 keys and max^m 5.

⇒ Minimum children for each node except the root and leaf is 3 and max^m 6.

Q. Using the minimum degree t= 2, insert the following sequence of integers 12, 25, 42, 50, 60, 70, 80, 28, 36, 14, 18 into an initially empty B-Tree.

Solution—

	min	max
Key	1 $(t-1)$	3 $(2t-1)$
Children	2 (Given $t=2$)	4 $(2t)$

(I)

Step-01: Insert 12 :

[12]

Step-02: Insert 25 :

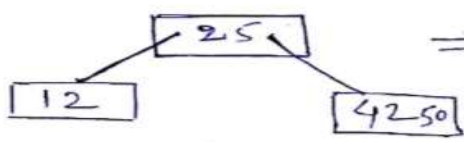
[12 25]

Step-03: Insert 42 :

[12 25 42] ⇒ Max^m keys allowed 3 (full)

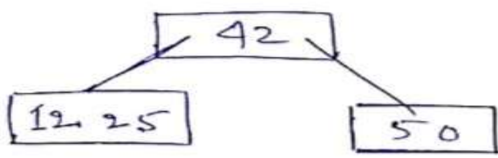
Step-04: Insert 50 :

We need to split the root node.



⇒ Right biased

or

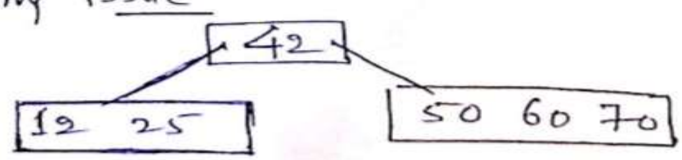


⇒ left biase

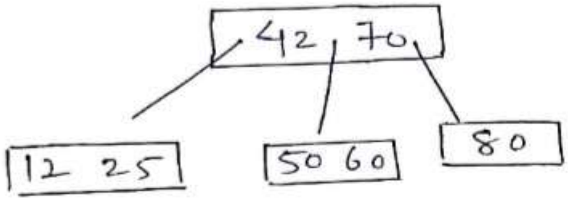
Both are ok.

Tree is created up-word.

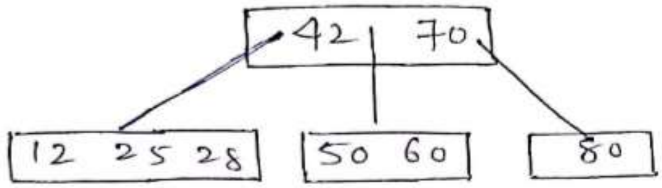
Step-05: Insert ~~the~~ next ~~three~~ two Keys (60 and 70) without any issue.



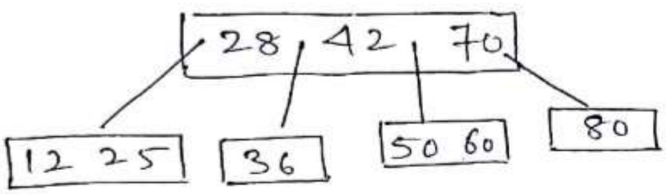
Step-06: Insert 80 : We need to split the node.



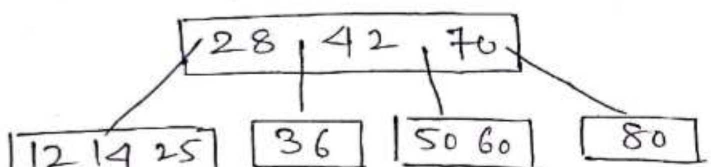
Step 07: Insert 28.



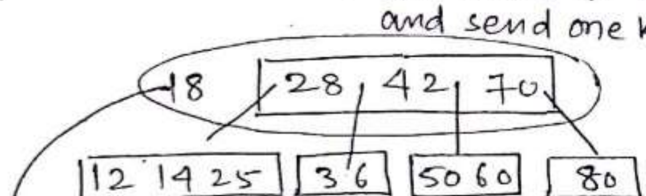
Step 08: Insert 36. We need to split the node.



Step 09: Insert 14.

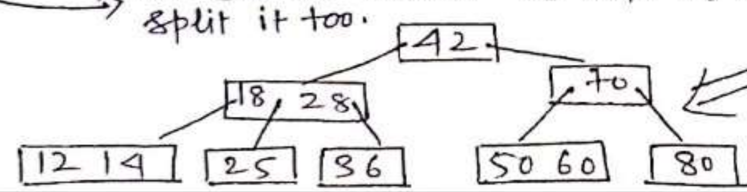


Step 10: Insert 18: We need to split the left most node and send one key to the root node.



or is already full (3 keys)

We cannot insert 18 into the root node, so split it too.



Resultant B-tree with $t = 2$.

Insert the following key: 12, 21, 41, 50, 60, 70, 80, 30, 36, 6, 16 into an empty B-tree with degree 4 (maxm).

Soln: Given.

Max Degree (m) = 4
Keys = m-1 = 3

It means each node can have four children and three keys (maxm).

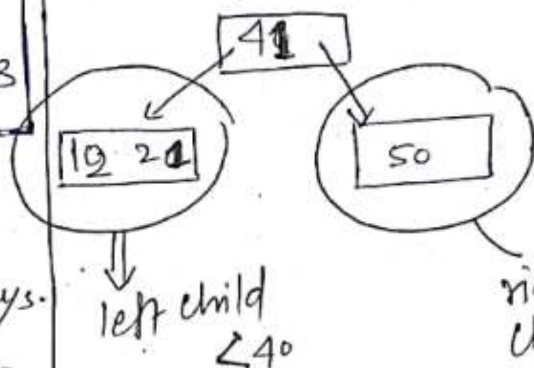
Step I: We can insert 12, 21 and 41 in the first node as it can have three keys.

12 21 41

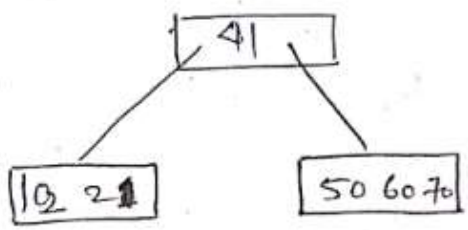
Each key must be arranged in ascending order only.

Step 2: Now, the next element 50 cannot be inserted in the first node as it has already reached its maximum capacity of holding three keys.

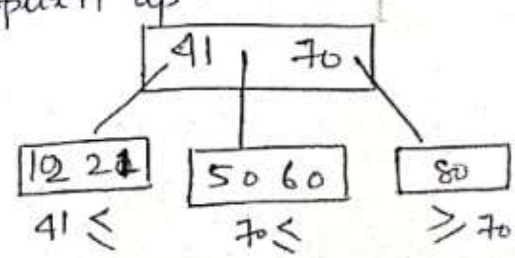
In such a case, we have to split up the node and send one of the keys on the root.



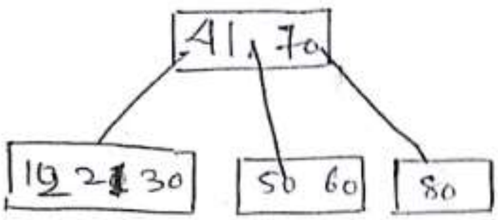
Step 3: Now, insert the next elements 60 and 70.



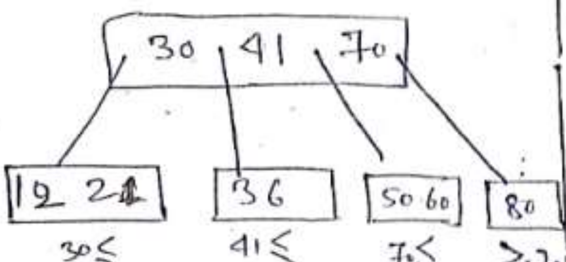
Step 4: Now, the next element 80 cannot be inserted in the right ~~child~~ node; therefore, split it up.



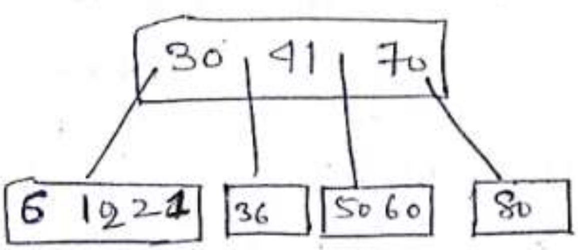
Step 5: Now, insert the next element 30. As it is less than 41, it will be inserted in the first child node.



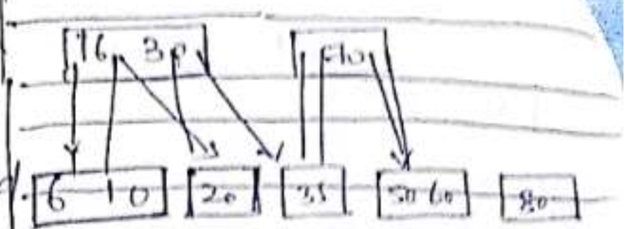
Step 6: Now, the next element 36 cannot be inserted in the first child node, so split it up.

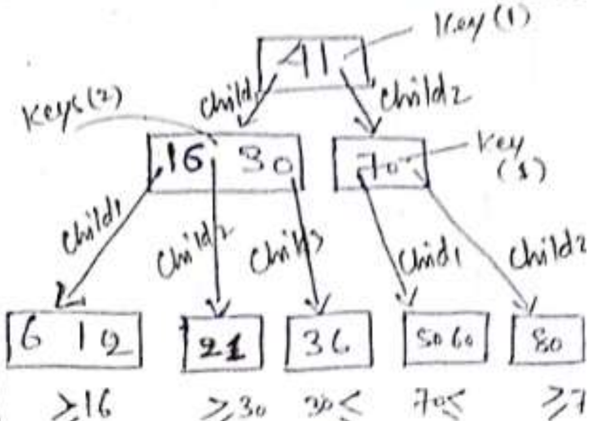


Step 7: Now, insert 6.



Step 8: Now, insert 16. we need to split the first child node.





⇓ This is three level of indexing created using B-tree data structure.

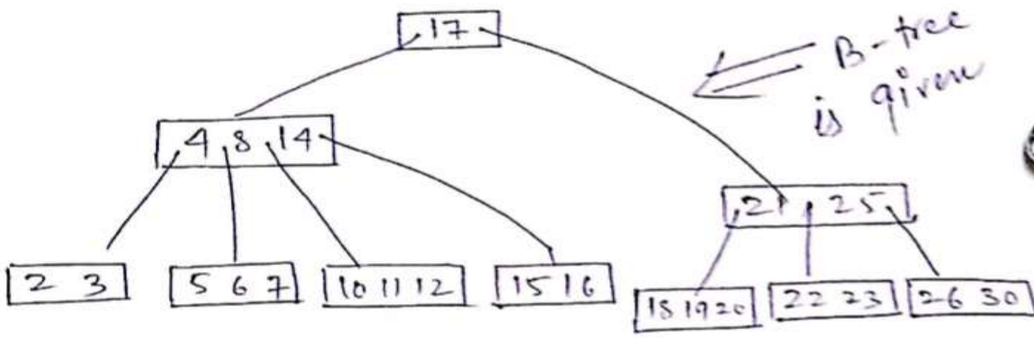
For $m = 6$, we have

	min^m	max^m
Key	$\lceil \frac{m}{2} \rceil - 1$ = 2 [Exact value]	$m-1$ = 5
CP	$\lceil \frac{m}{2} \rceil = \lceil \frac{6}{2} \rceil$ = 3	$m = 6$

CP → Child pointer

Deletion of keys in a B-tree

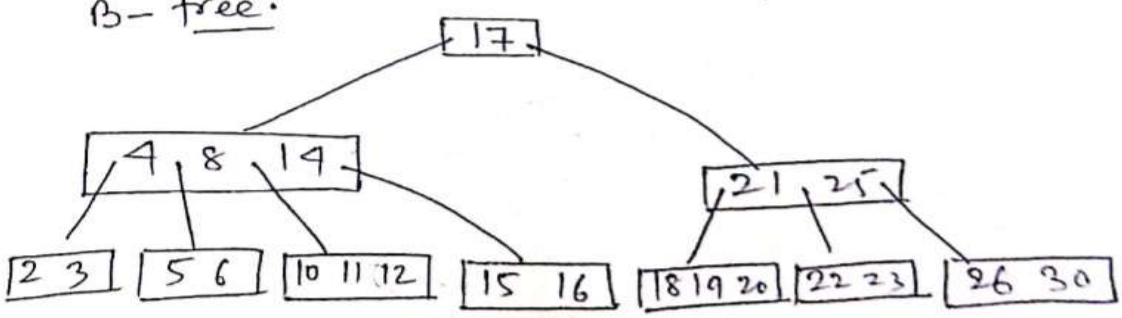
Delete the keys : 7, 14, 8, 5, 3 ~~whe~~ m = 6. (max degree)



Sol^n : ↓ For m = 6, we have the following properties:

	min^m	max^m
Key	2	5
Children (CP)	3	m = 6 → Given

Step-01: We need to delete 7, which can be directly deleted without violating the properties of B-tree.

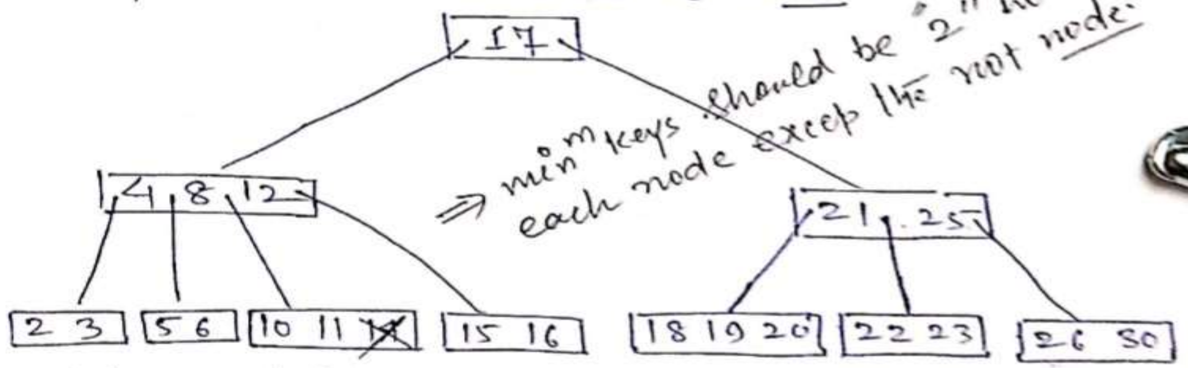


Step-02: Delete (14) :↓ We cannot directly delete a non-leaf node. We need to swap the key of the node to be deleted and its inorder predecessor or inorder successor.

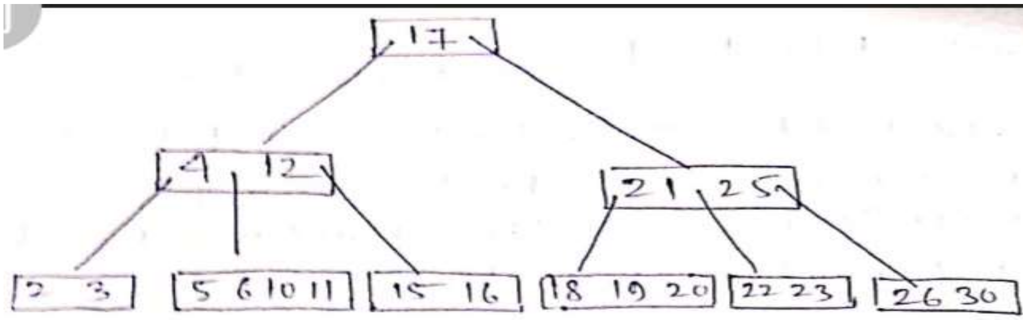
Inorder Predecessor: The longest key value of the left sub-tree.

Inorder Successor: The smallest key value of the right sub-tree.

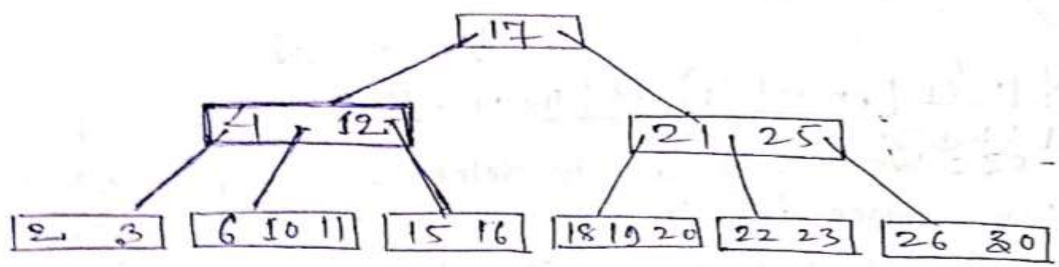
"14" will be replaced with 12 and 12 with 14. And now, 14 can be deleted without violating the properties of B-tree mentioned in the table.



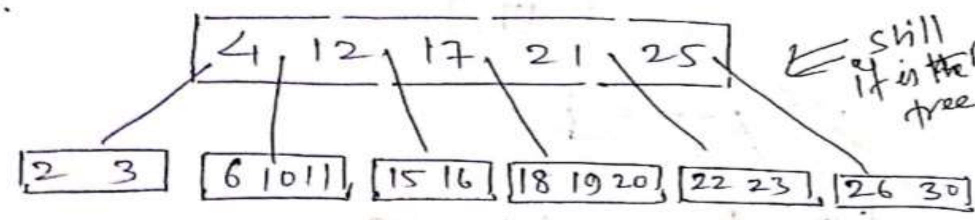
Step-03: Delete 8 :↓ Case is same as the step-02, but here if we try to replace 8 with either inorder predecessor or successor, then after deleting the key the particular leaf node is left with only one key, which violates the properties of B tree mentioned in the table. In such a case, we delete the key and merge both of its children.



Step-04: Delete 5: we can directly delete it without violating the B-tree properties.

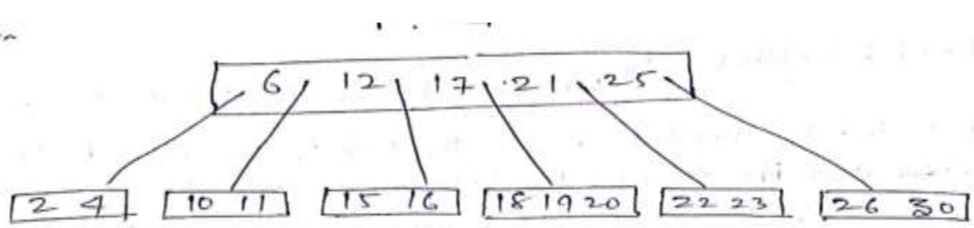


Now, check if we can shrink the B-tree. If yes, do it.



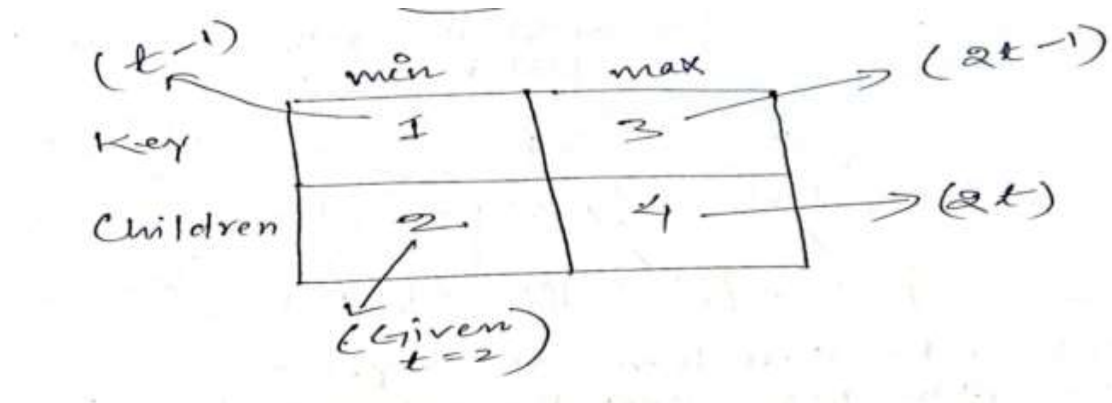
still it is the B-tree.

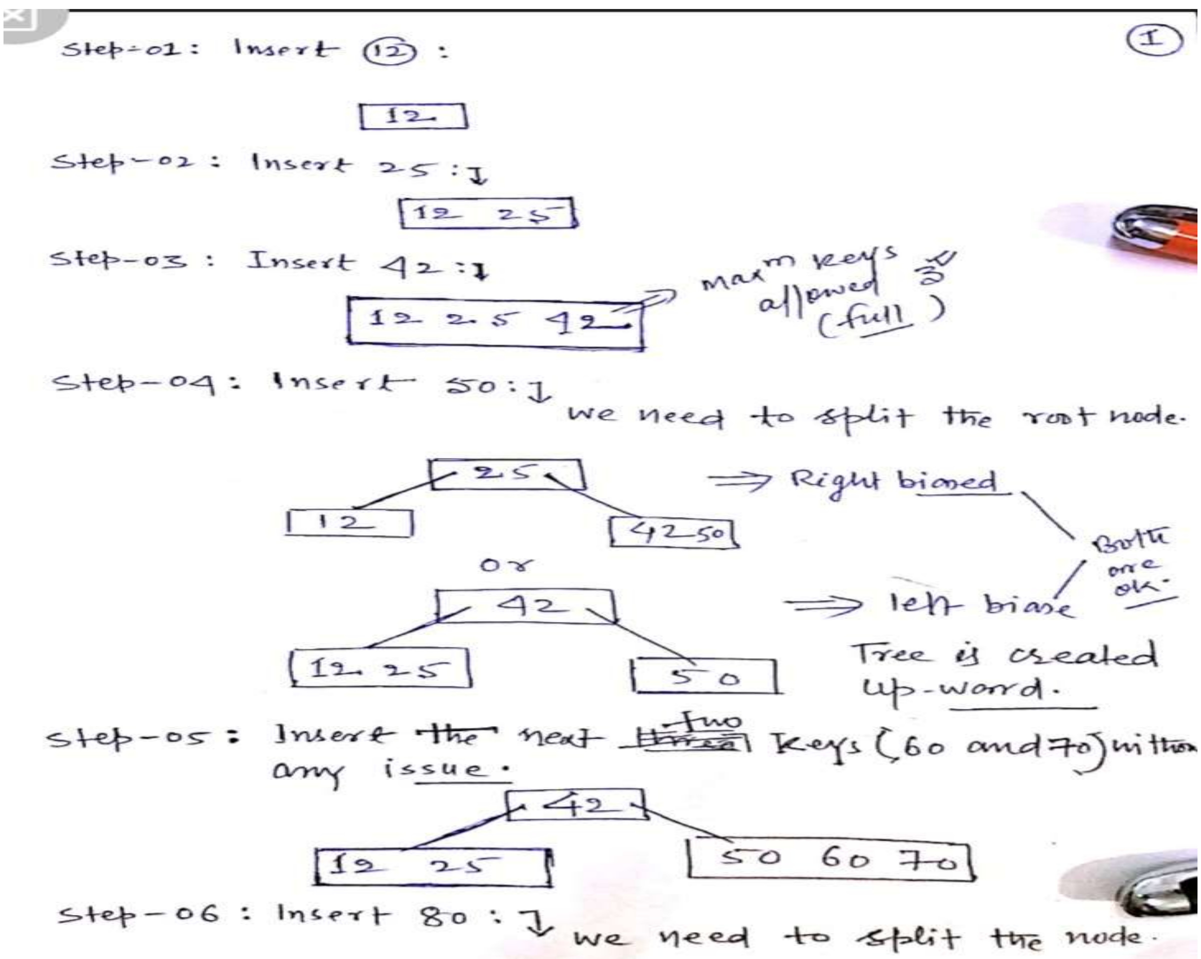
Step-05: Delete 3: we cannot delete 3 as it will violate min^m key property of the B-tree. In such a case, we borrow a key from the parent node and send one neighbor key on the parent node. First, we delete 3 by replacing it with 4 and then send 6 on the root.

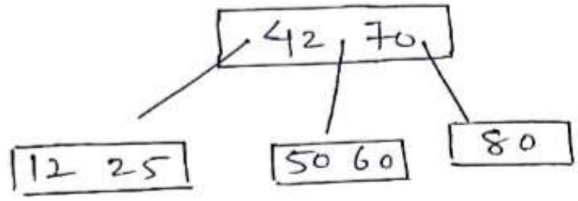


Resultant B-tree with degree 6

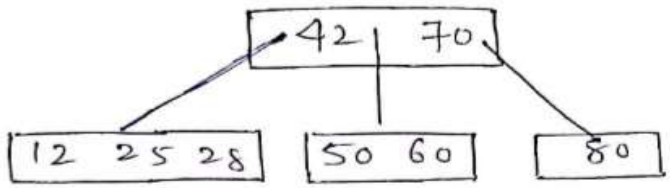
Q. Using the minimum degree t = 2, insert the following sequence of integers: 12, 25, 42, 50, 60, 70, 80, 28, 36, 14, 18 into an initially empty B-Tree. Now, delete 60, 18, and 14.



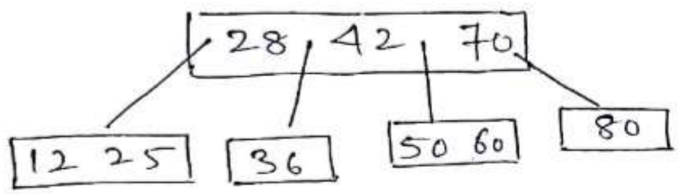




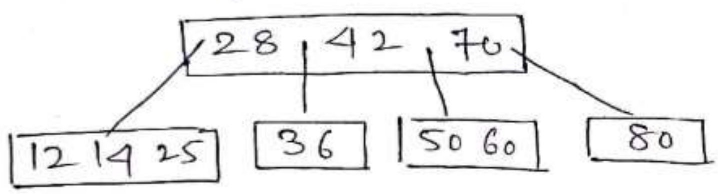
Step 07: Insert 28.



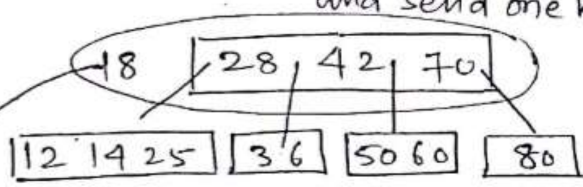
Step 08: Insert 36. We need to split the node.



Step 09: Insert 14

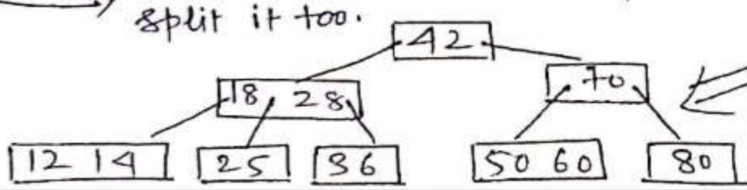


Step 10: Insert 18: We need to split the left most node and send one key to the root node.



gr is already full (3 keys)

We cannot insert 18 into the root node, so split it too.

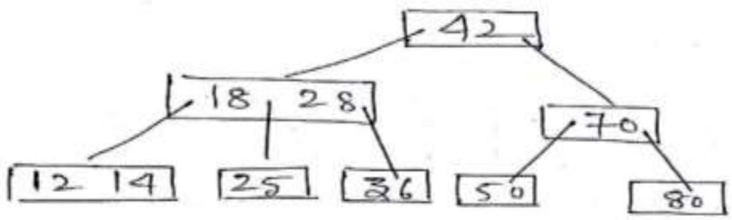


Resultant B-tree with $t = 2$.

Deletion

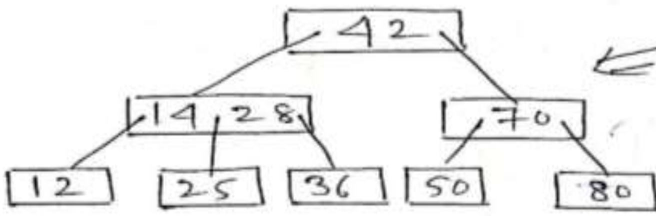
1. Delete 60:

We can directly delete 60 without violating the properties of B-tree mentioned in the table.



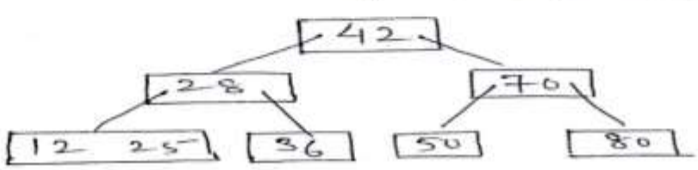
2. Delete 18:

We cannot directly delete a non-leaf node's key. In this case, we swap the key to be deleted and its either in-order predecessor or successor.

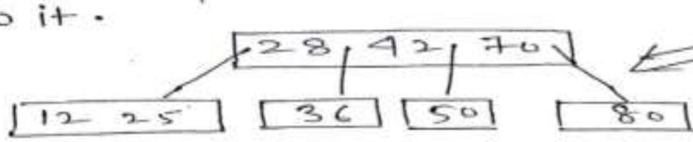


Here, we consider inorder predecessor of the key(18), which is 14 (largest element in the left child)

3. Delete 14: We cannot directly delete 14 as it will violate the B-tree property. In this case we dete the key (14) and merge both of its children.



Now, Check if we can shrink the tree. If yes, we do it.



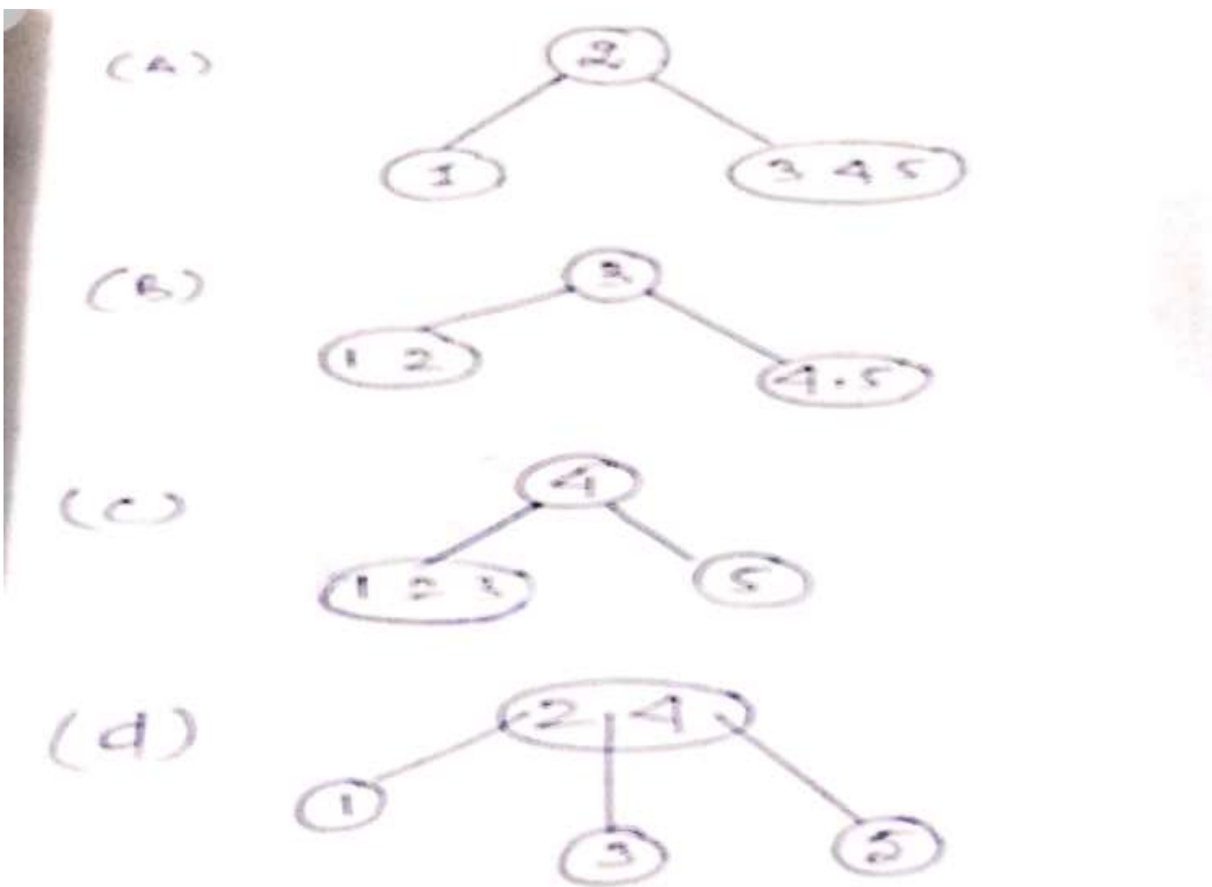
Resultant B-tree with min^m degree 2.

Q. Why don’t we allow a minimum degree of t=1 in B-trees?

Allowing a minimum degree of t=1 in B-trees would fundamentally change their structure and behavior in ways that make them less efficient and undermine some of their key advantages. The choice of a minimum degree of t>=2 in B-trees is a deliberate design decision to maintain the desirable properties and performance characteristics that make B-trees valuable for a wide range of applications involving sorted data storage and retrieval.

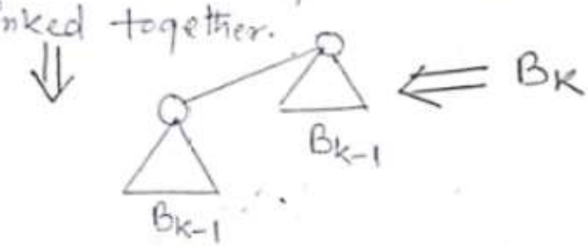
Q. Show all legal B-Trees of minimum degree 2 that represent <1,2,3,4,5>

Q.



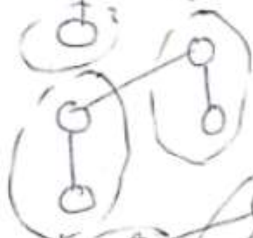
Q7. Define Binomial Tree and mention its properties.

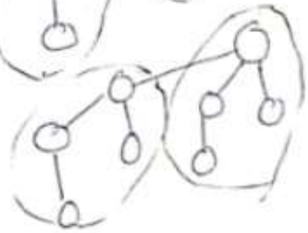
* The Binomial tree "B_K" is an ordered tree defined recursively.
* The Binomial tree B_0 Consists of a single node.
* The Binomial tree B_k Consists of two Binomial trees B_{k-1} that are linked together.



$B_0 \Rightarrow$ ○ (Consists of a single node)

$B_1 \Rightarrow$  (consists of two sing nodes)

$B_2 \Rightarrow$  (Consists of two B_1)

$B_3 \Rightarrow$  (Consists of two B_2)

.
.
.

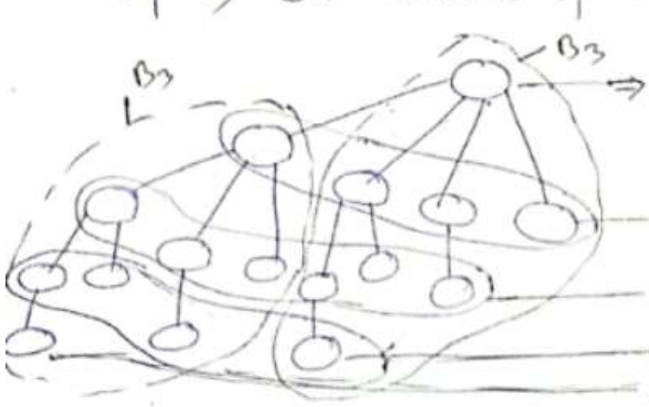
$B_K \Rightarrow$ (Consists of two B_{K-1} Binomial trees)

Properties of Binomial tree (B_K):

1. There are 2^k nodes.
2. The height of tree is k.
3. There are exactly ${}^{k}C_i$ nodes at depth i for $i = 0, 1, 2, \ldots, k$.
4. The root has degree k, which is greater than other nodes.
5. If i, the children of the root, are numbered from left to right by $k-1, k-2, \ldots 0$, then child i is the root of a subtree B_i.

Let's verify the above properties using a Binomial tree B_4.

$B_4 \Rightarrow$ It consists of two B_3 Binomial trees:



depth

$\Rightarrow i = 0 \quad {}^{4}C_0 = 1$

$i = 1 \quad {}^{4}C_1 = 4$

$i = 2 \quad {}^{4}C_2 = 6$

$i = 3 \quad {}^{4}C_3 = 4$

$i = 4 \quad {}^{4}C_4 = 1$

1. 2^k nodes $= 2^4 = 16$
2. Height $= 4$
3. ${}^{k}C_i$ nodes at depth i for $i = 0, 1, 2, k$

$$\hookrightarrow \frac{k!}{i!\,(k-i)!}$$

4. Degree of the root = 4, which is greater than other nodes. ②

5. K = 4 for the root of tree and 3, 2, 1, and 0 for all children from left to right, then child 3 is the root of a sub tree, child 2 is the root of a subtree, child 3 is the root of a sub-tree and ~~the~~ root 0 is the root of a subtree.

Q8. Define Binomial heap, write an algorithm for union of two Binomial heaps and also write its time complexity.

Lecture 22 By S. Khan ①

Binomial Heap

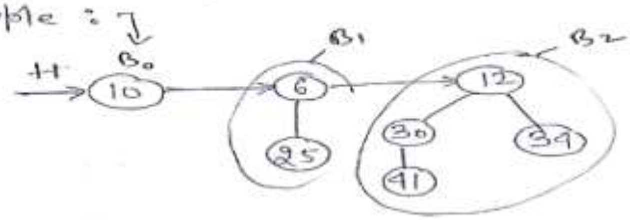
A Binomial heap "H" is a set of Binomial trees that satisfies the following Binomial heap properties:

(i) Each Binomial tree in H obeys the min/max heap property.

(ii) For a non-negative integer k, there is at most one Binomial tree B_k in H. It means no Binomial tree is repeated in H.

(iii) Degrees of all Binomial trees are in increasing order in H.

Example:



⇒ Binomial trees B_0, B_1 and B_2 are min-heaps.

⇒ All Binomial trees are distinct (B_0, B_1, B_2)

⇒ Degrees of Binomial trees:

B_0 = 0
B_1 = 1 ⟹ increasing order
B_2 = 2

Binomial min Heap Union operation ②

Algo :

1. Merge the root lists of Binomial min heaps H_1 and H_2 into a single linked list that are stored in increasing order of their degrees.
2. Link the roots of equal degree until at most one root remains of each degree.

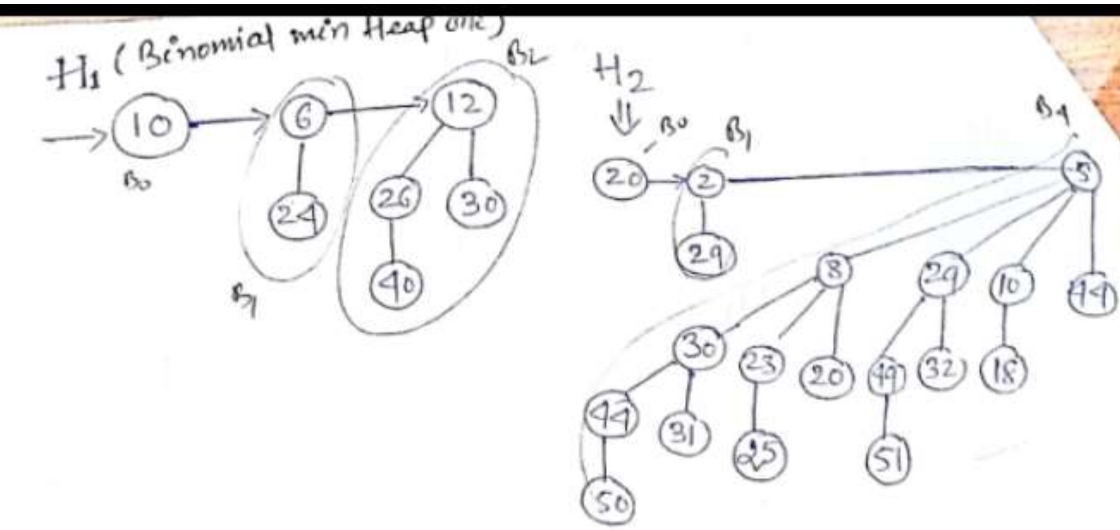
   (a) If (degree[X] ≠ degree [next-X] or degree [X] = degree[next-X] = degree [sibling[next-X]]) , then move the pointers one position right.

   (b) If (degree[X] = degree [next-X] ≠ degree[sibling[next-X]])

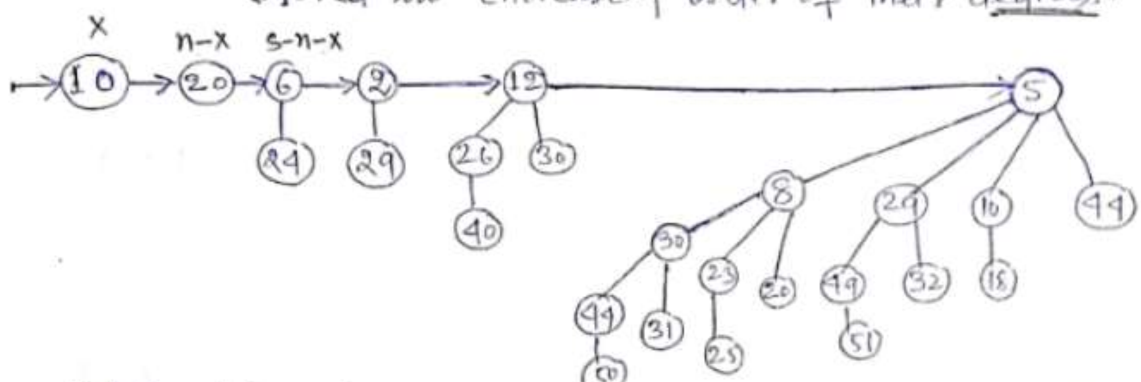
   (i) If key(x) < key(next-x), then make (next-x) as left most child of X.

   (ii) If key(X) > key(next-x), then make X as left-most child of (next-x)

Example :



Step 1 : Merge the root lists of Binomial min heaps H_1 and H_2 into a single linked list that are stored in increasing order of their degrees.



Step 2: Link the roots of equal degree until at most one root remains of each degree.

degree of x $= 0$

degree of next of x $= 0$ $\quad 0 = 0 \neq 1$

degree of sibling of next of x $= 1$

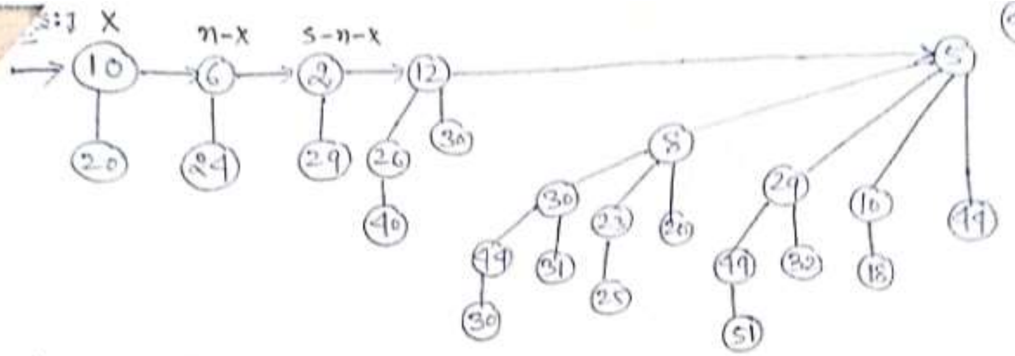
(b) If (degree [x] = degree [next-x] $\neq$ degree [sibling[next-x]])

$\Downarrow$ True.

(i) If $\underset{10}{\text{key}(x)} < \underset{20}{\text{key}(\text{next}-x)}$ True

make (next-x) as left most child of x.

(9)



Degree of X = 1
Degree of n-X = 1
Degree of s-n-X = 1

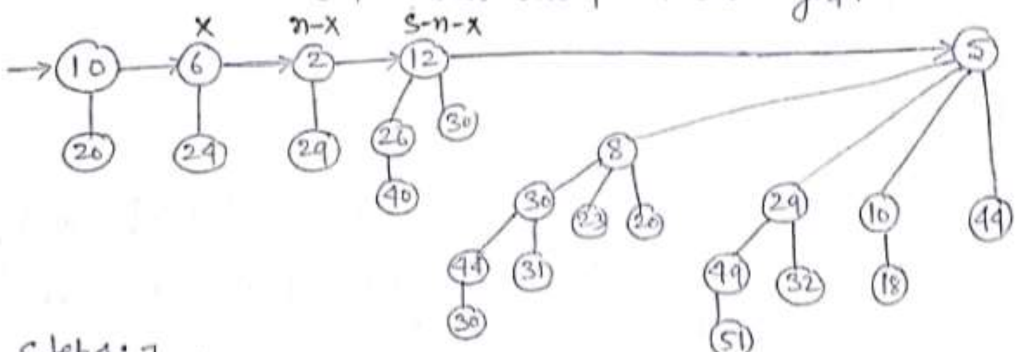
$\boxed{1 = 1 = 1}$

↳ Degree of these three Binomial heaps are same.

②ⓐ If (degree [x] ≠ degree [next-x] or
degree [x] = degree [next-x] = degree [sibling [next-x]])

⇓ True.

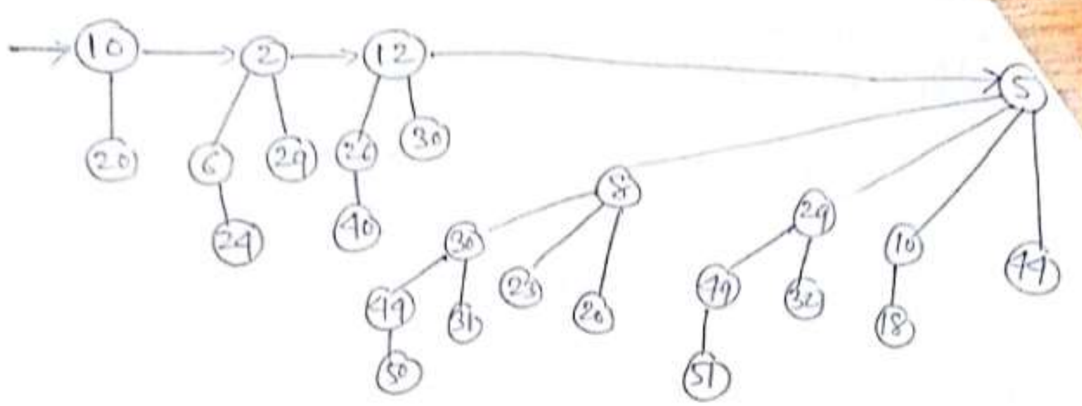
move the pointers one position right.



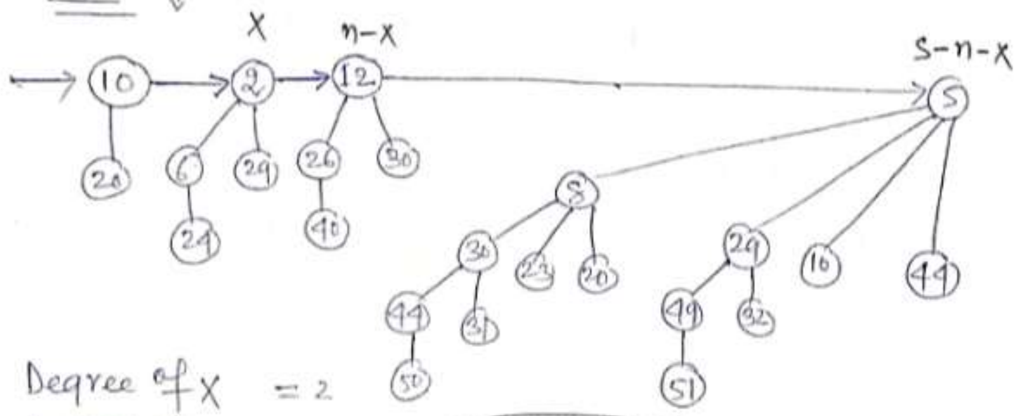
Step 4: Degree of X = 1
Degree of n-X = 1
Degree of S-n-X = 2

$\boxed{1 = 1 \neq 2}$

②ⓑ If (degree [x] = degree [next-x] ≠ degree [sibling [next-x]])
(ii) If key(x) > key (next-x), make X as left-most child of (next-x).



Step 5 :



Degree of $x = 2$

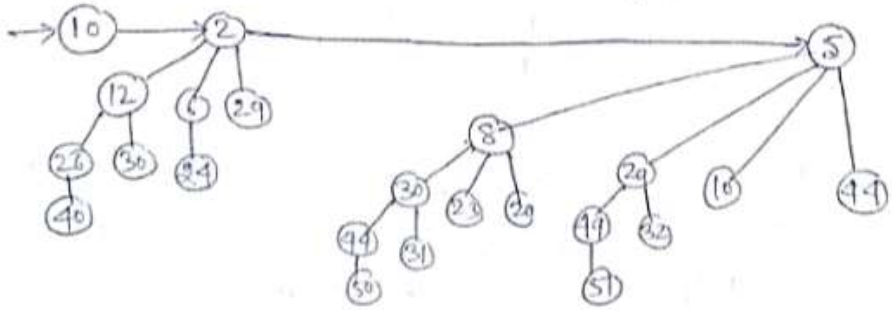
Degree of $n\text{-}x = 2$

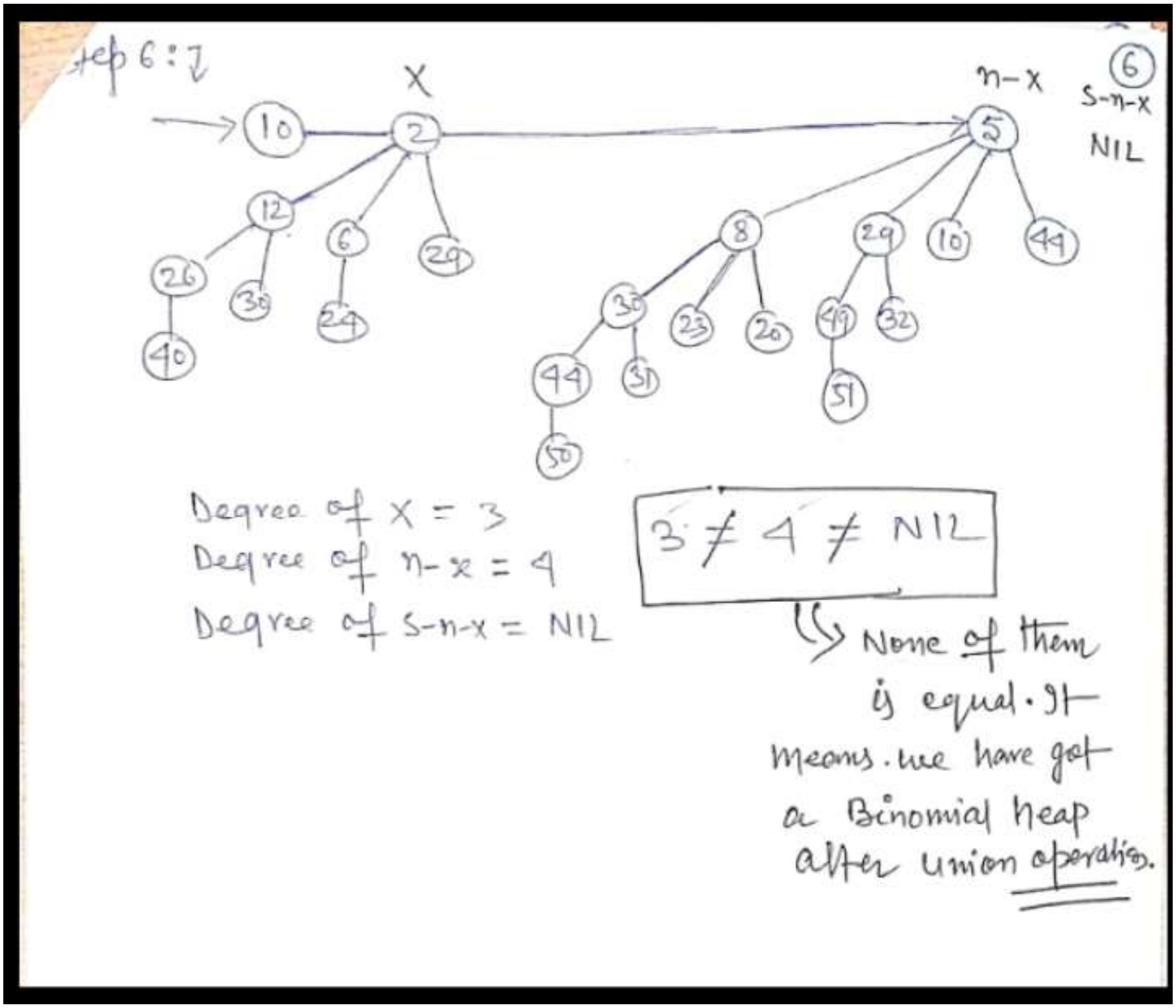
Degree of $s\text{-}n\text{-}x = 4$

$$2 = 2 \neq 4$$

②⑤ If (degree [x] = degree [next-x] ≠ degree [sibling [next-x]])

(i) If key(x) < key (next-x), make (next-x) as left most child of x.





Time complexity : O(Log(n))

Q9. Design a Binomial heap for a given A, A= [7, 2, 4, 17, 1, 11, 6, 8,15,10, 20]

Steps:

1. Create a Binomial heap H^1 containing a new element (key).
2. Apply union operation between the two Binomial min heaps H and H^1.

Sol^n: ↓

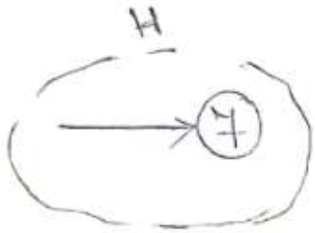
Step 1: ↓

Initially, we have an empty Binomial min-heap H. A new node, what we create after inserting a new element, is by-default a Binomial heap H'.

Insert 7:

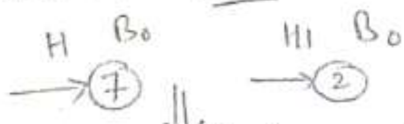
H empty ⟶ 7 ⇒ H'

⇑ Perfor union b/w H and H'



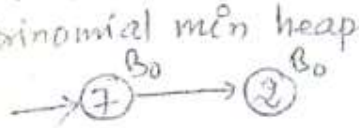
⇓ This is a Binomial heap with one node and having zero child. It means that this Binomial heap Contains only one Binomial tree (B_0).

Now, Insert 2

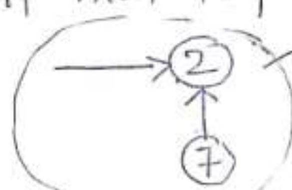


⇓ Perform union

Merge the root lists of Binomial min heaps

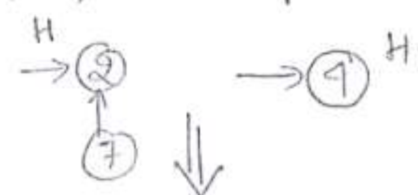


As degrees are same, make it min heap.



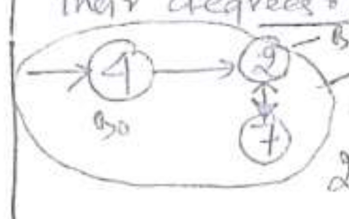
Binomial heap having one Binomial tree of degree one (B_1)

Now, insert 9.

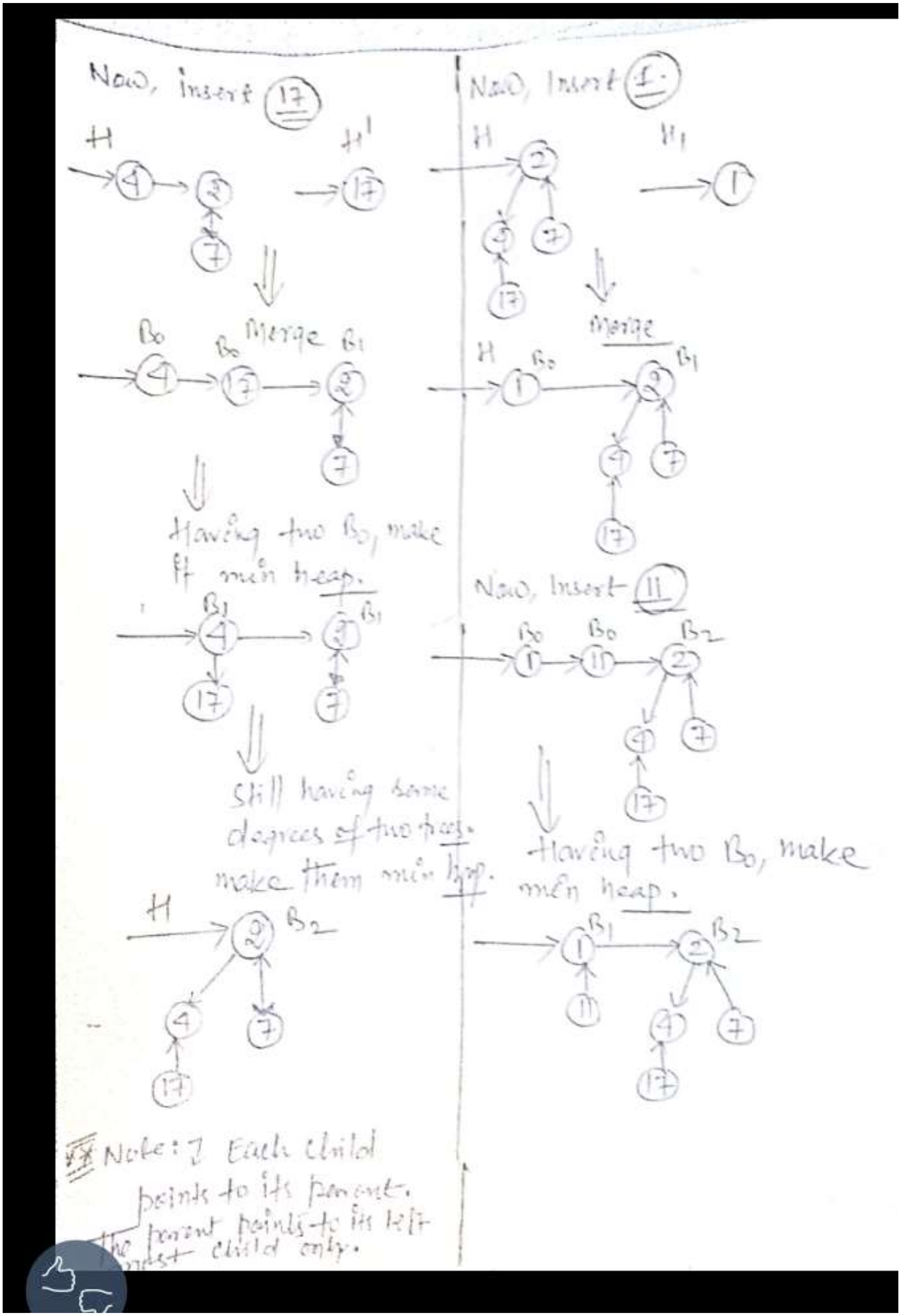


⇓

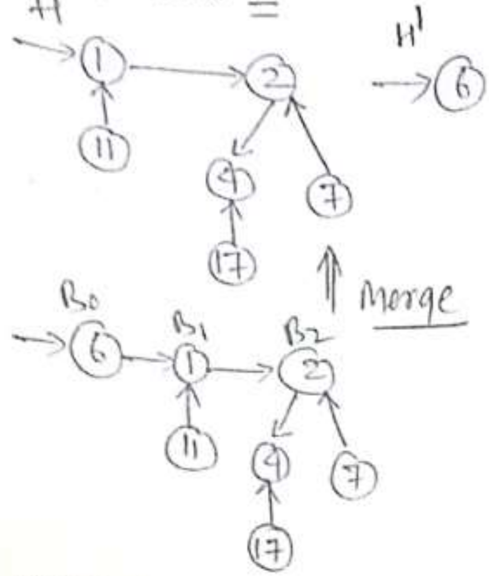
Merge the root lists of Binomial min heaps in the increasing order of their degrees.



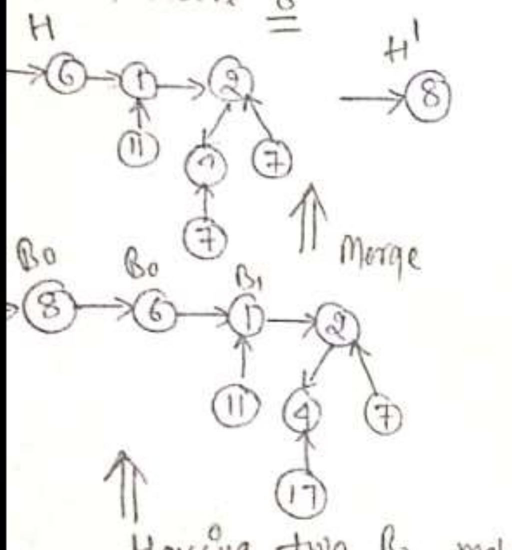
Binomial heap Containing 2 Binomial trees B_0 & B_1


Now, insert 17
H
H'
Merge
B0
B0
B1
Having two B0, make it min heap.
B1
B1
Still having same degrees of two trees. make them min heap.
H
B2
Note: Each child points to its parent. The parent points to its left most child only.
Now, Insert 1.
H
H1
Merge
H
B0
B1
Now, Insert 11
B0
B0
B2
Having two B0, make min heap.
B1
B2

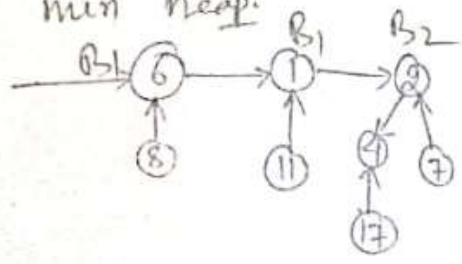
Now, Insert 6



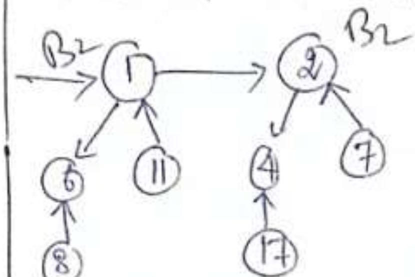
Now, Insert 8



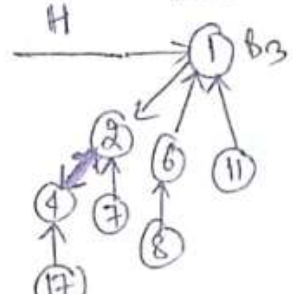
Having two B_0s, make min heap.



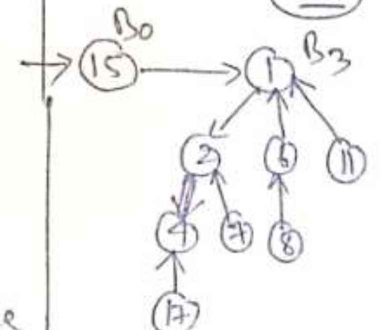
Still having two B_1, make Binomial min heap.



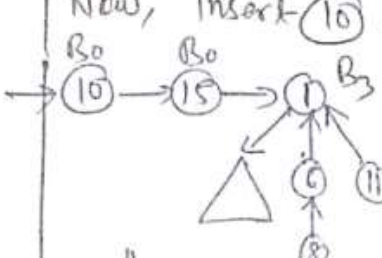
Still having two B_2, make the min heap.



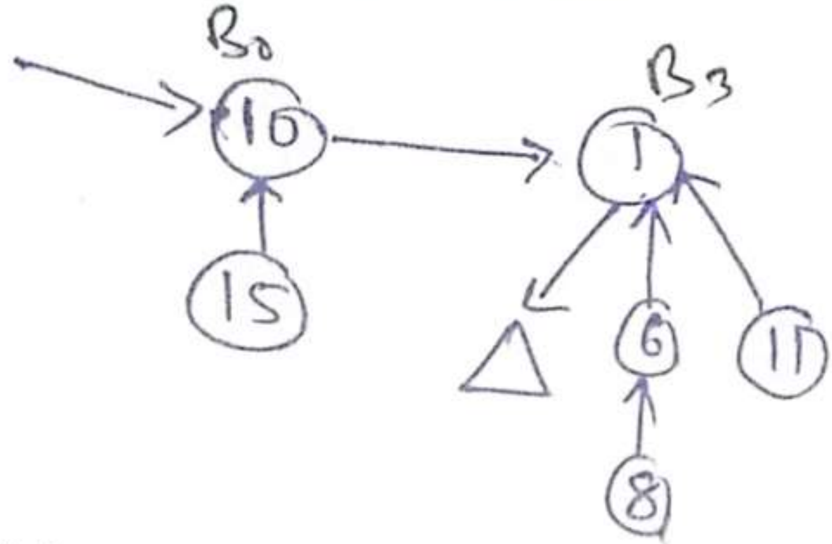
Now, Insert 15.



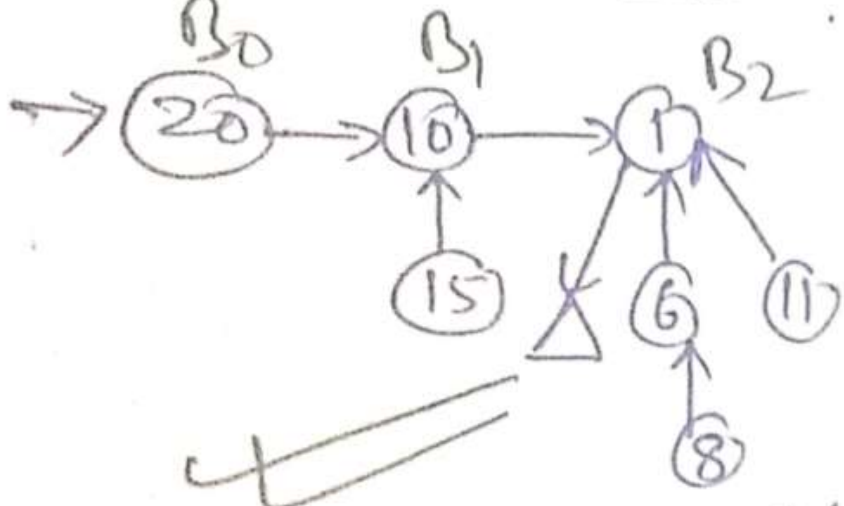
Now, Insert 10.



⇓ Having two B



Now, Insert 20..



Resultant Binomial min heap.

Q. Write down the algorithm for Decrease key operation in Binomial Heap also write its time complexity.

```
BINOMIAL-HEAP-DECREASE-KEY(H, x, k)
if k > key[x]
    then error "new key is greater than current key"
key[x] = k
y = x
z = p[y]
while z != NIL and key[y] < key[z]
    do exchange key[y] and key[z]
        If y and z have satellite fields, exchange them, too.
        y = z
        z = p[y]
```

Running time T(n) = O(logn)

Q10. Define Fibonacci heap and also compare the complexities of Binary heap, Binomial heap and Fibonacci heap.

By S.Khan (1)

Lecture - 24

Fibonacci Heap

Definition :- It is a collection of trees with each tree following the heap ordering property (either min or max heap).

Properties of Fibonacci Heap:

(i) Trees may be in any order in the root list.

(ii) A pointer to the minimum element of the heap is always maintained.

(iii) Siblings are connected through a circular doubly linked list.

(iv) Each child points to its parent.

(v) Each parent points to any one child.

(vi) Each tree of order n has at least F_{n+2} nodes in it.

Reason to be called it as fibonacci heap.

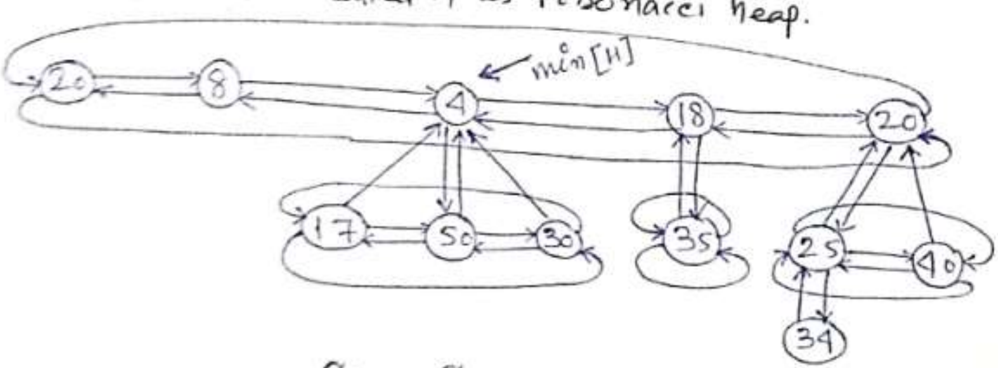


fig: Fibonacci Heap

Procedure	Binary Heap	Binomial Heap	Fibonacci Heap
Make Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
Min	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$
Extract min	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Union	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
Decrease key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
Delete	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

Fig: comparison of heaps. Amortized running time

⇒ Fibonacci heaps are used to implement the priority queue element in Dijkastra's algorithm.

Q11. Explain the extracting minimum node operation of Fibonacci heap with example.

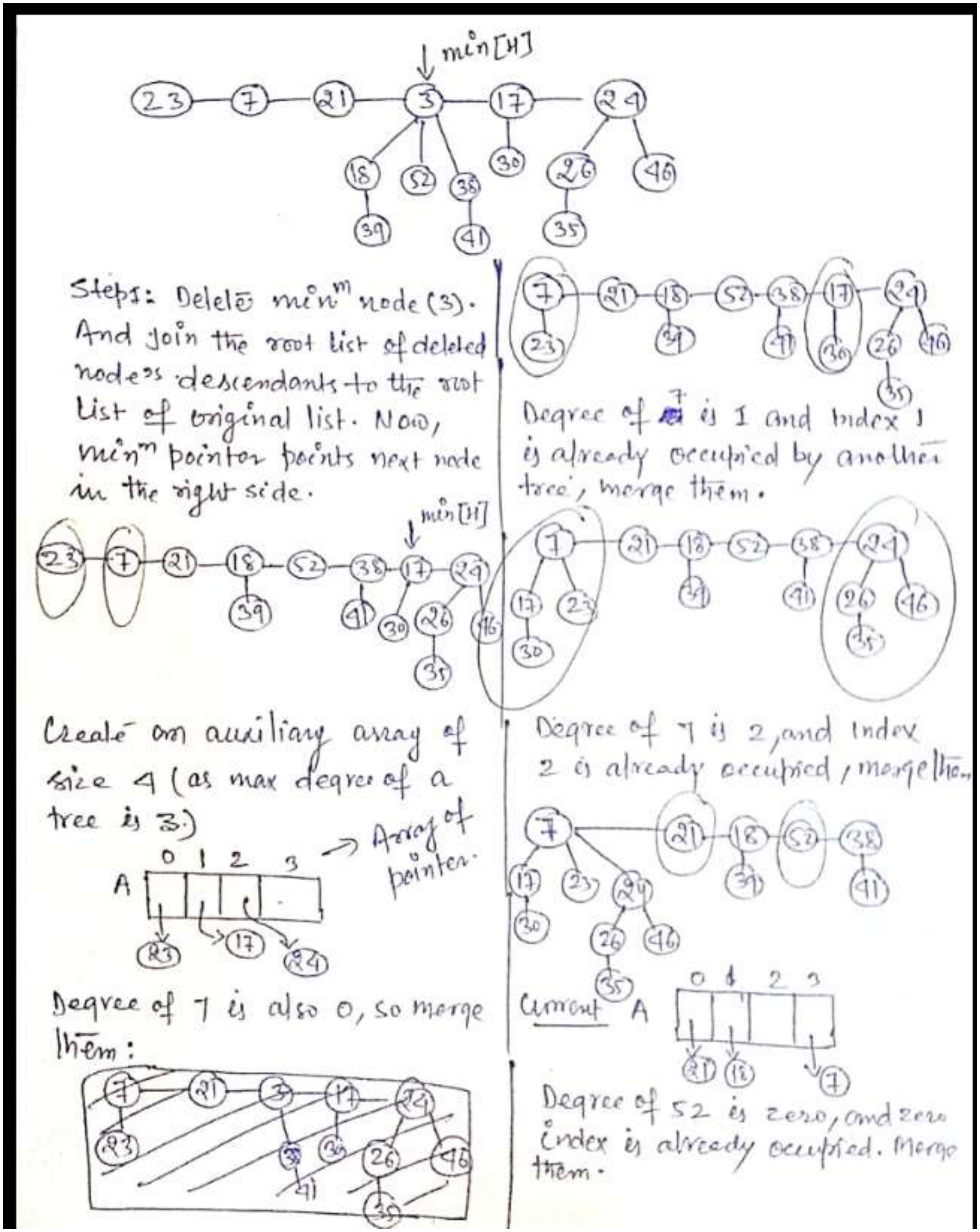
This operation is accomplished by deleting the minimum key node and then moving all its children to the root list. It uses the process called "consolidate" to merge the trees having same degree.

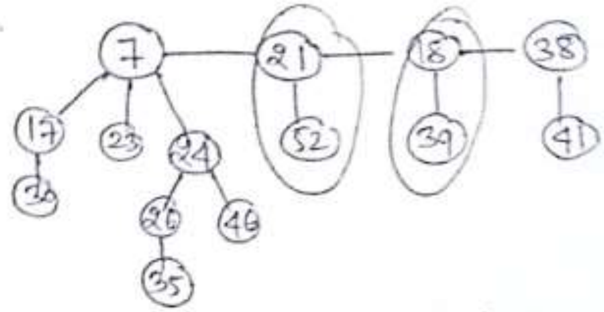
Steps:

1. Delete the minimum node .
2. Join the root list of deleted node's descendants to the root list of original root list.
3. Traverse left to right in the root list
   3.a Find new minimum.
   3.b Merge trees having same degree.

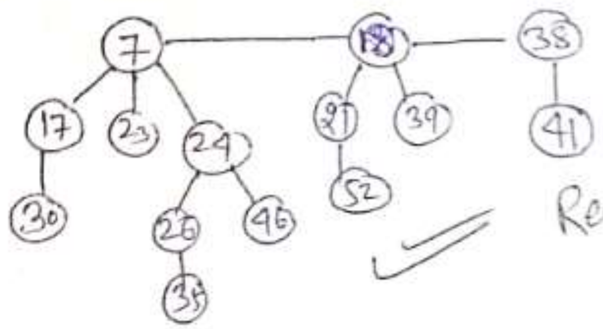
4. Stop after having every tree with unique degree.

Example-


min[H]
Steps: Delete minm node (3). And join the root list of deleted node's descendants to the root list of original list. Now, minm pointer points next node in the right side.
Degree of 7 is 1 and index 1 is already occupied by another tree, merge them.
min[H]
Create an auxiliary array of size 4 (as max degree of a tree is 3.)
A
Array of pointer
Degree of 7 is also 0, so merge them:
Degree of 7 is 2, and index 2 is already occupied, merge them.
Current A
Degree of 52 is zero, and zero index is already occupied. Merge them.



Degree of 21 is 1, and index 1 is already occupied. Merge them.



Resultant fibonacci heap.

Q12. Define Skip List and Trie with example.

Skip List

(5)

Definition :-

It is a probablistic data structure with a hirarchy of sorted linked lists. Subsequent Layers of linked lists are subsets of the original sorted linked list only.

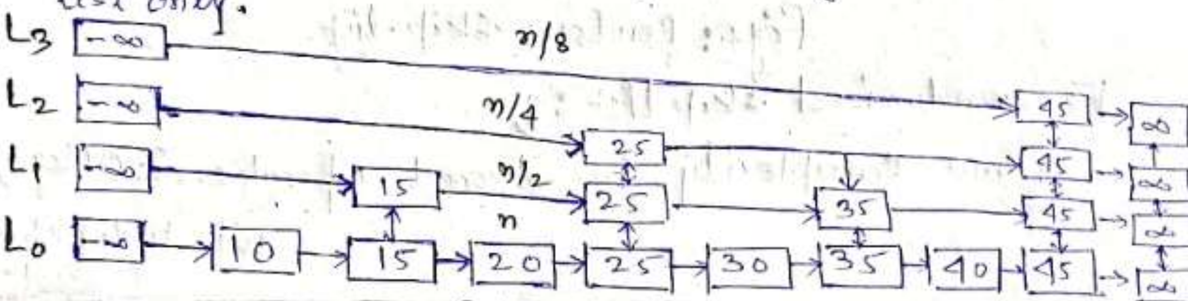


Fig1: A Perfect skip list

Number of levels in skip list $\approx \log_2 n$

Time complexity for the search operation $= O(\log n)$
(2 * no of level)

Perfect skip list :- A skip list where we promote alternate elements and has $\log_2 n$ levels is called a perfect skip list. Fig I is an example of a Perfect SL.

Random skip list :- A skip list where we promote random elements from the original sorted linked list is called Random skip list.

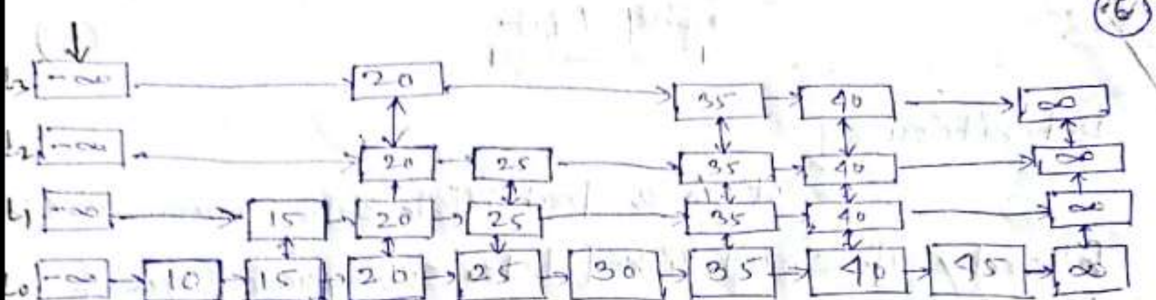


Fig2: Random skip-list.

For randomized skip list :

Time complexity for search operation $\cong O(\log n)$ with high probability.

Time complexity to insert an element in a skip list $= O(\log n)$

Perfect

Time complexity to delete an element from a perfect skip list $= O(\log n)$

Space complexity $= O(n \log n)$

Q. Explain the Search operation in Skip list with a suitable example. Also write its algorithm.

By S.Khan (2)

Trie
(Digital tree / Prefix tree)

Definition :

It is a tree used for storing and searching a specific key from a set. We generally use tries to store strings. Using it, search complexities can be reduced to key length. The word Trie is derived from reTRIEval, which means finding something or obtaining it.

Properties :

1. It is a tree.
2. It stores a set of strings.
3. Every node can have at most 26 children
4. * Every node except root node stores a letter of English alphabet.
5. Each path from the root to any node represents a word or string.

Operations :

1. Search
2. Insert
3. Delete

→ $O(L)$ — L: length of the key.

Applications :

1. Prefix search
2. Dictionary
3. Spell Checker

Construct a Trie and Compressed Trie for the set of strings: <DOG, DONE, CAT, CAN>.

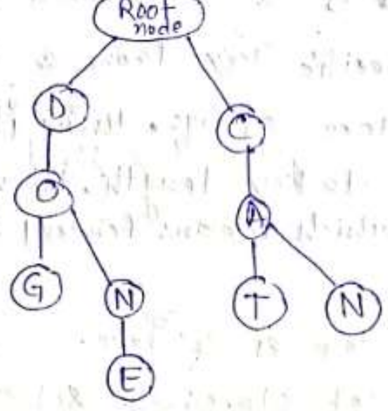


fig: Trie for the given strings.

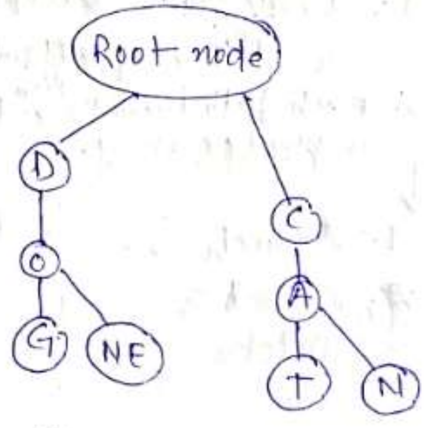


fig: compressed Trie

Q. Insert the following strings into the initially empty trie: DOG, DONE, CAT, CAN, RIGHT, DO, JUG, DAA, CA, CAME. Then delete the strings DO, CAME, and RIGHT from it.

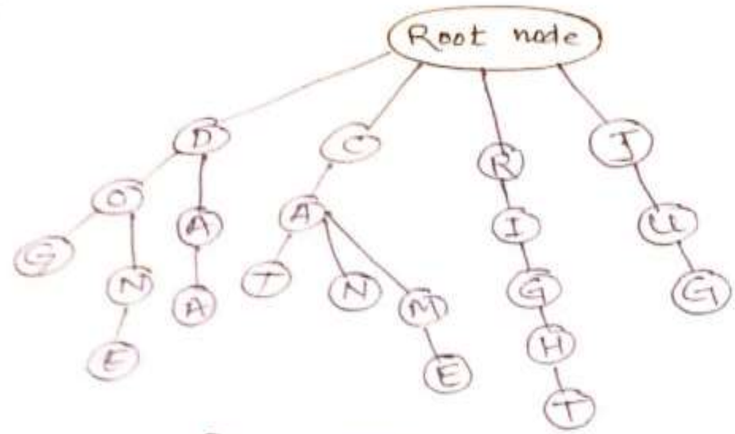


fig01: Trie

Deleting DO, CAME and RIGHT

Soln: I we delete as long as it does not affect other string(s), and deletion starts from the right most letter of a string.

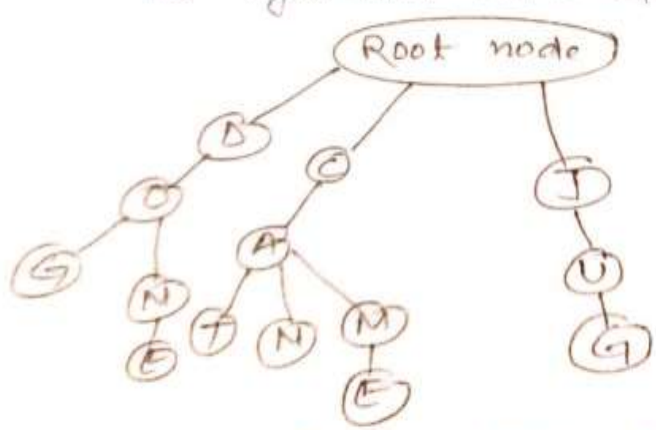


fig02: Trie after deletion

Q14. Insert strings < ten, test, car, card, nest, next, tea, tell, park, part, see, seek, seen> in an empty Trie data structure and also compress the Trie.

Insert strings : < ten, test, car, card, nest, next, tea, tell, park, part, see, seek, seen> in an empty Trie data structure and also compress the Trie.

soln: ↓

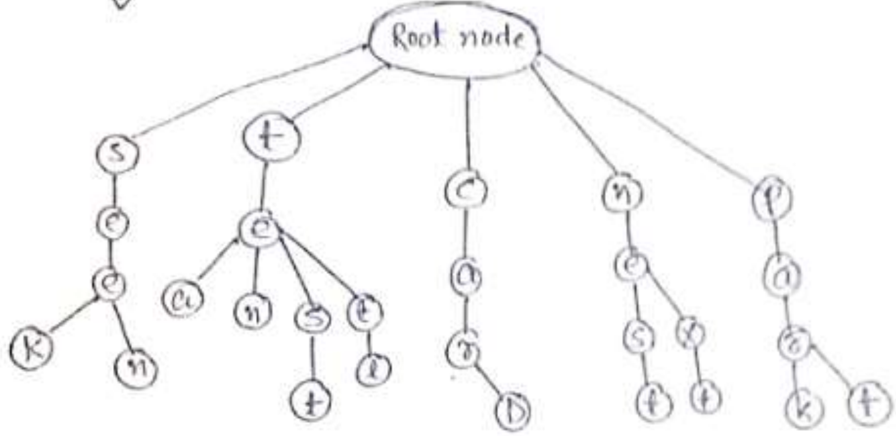


fig1. Trie

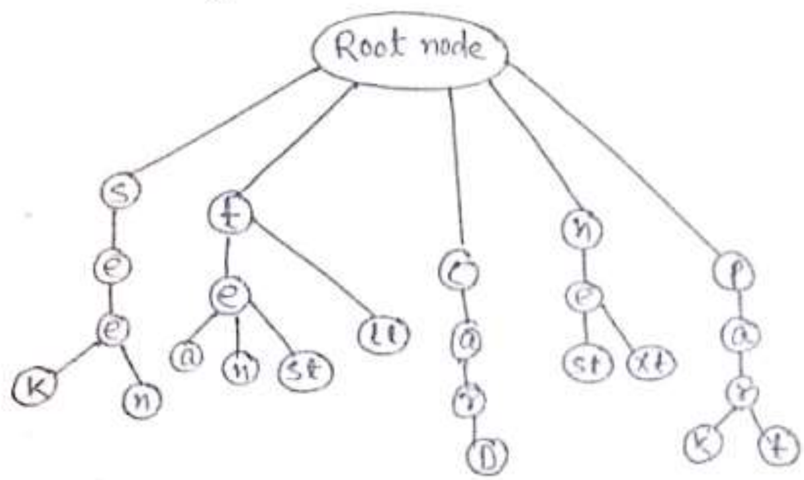


Fig2: Compressed Trie

Unit-03

Divide and Conquer: It is one of the algorithm design techniques in which the problem is solved using the divide, conquer and combine strategy.

Divide: This involves dividing the problem into smaller sub-problems. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible.

Conquer: This involves solving sub-problems by calling them recursively until solved.

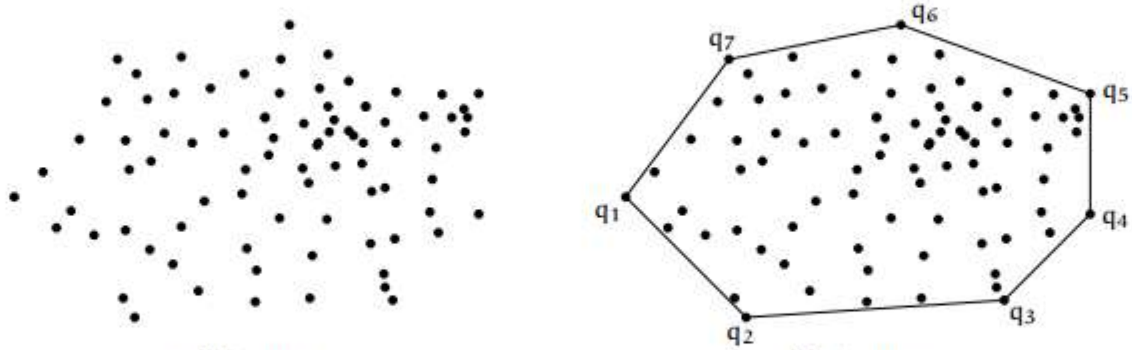
Combine: When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution to the original problem.

Following are some standard algorithms that follow Divide and conquer approach:

1. Quick sort
2. Merge sort
3. Strassen's algorithm for matrix multiplication
4. Convex Hull algorithm
5. Closest pair of points

Q1. Describe the Convex-Hull problem with a suitable example.

Given a set of points, a Convex Hull is the smallest convex polygon containing all the given points.



(a) Input. (b) Output.

Quickhull Algorithm [Divide and Conquer Algorithm]

Let P[0…n-1] be the input array of points. Following are the steps for finding the convex hull of the points.

1. Find the point with the minimum X-coordinate. Let's say, min_x and similarly the point with the maximum X-coordinate, max_x.
2. Make a line joining these two points, say L. This line divides the whole set into two parts. Take both parts one by one and proceed further.

3. For a part, find the point P with the maximum distance from line L. P forms a triangle with the points min_x and max_x.
4. The above step divides the problem into two sub-problems, which are solved recursively. Now, the line joining the points P and min_x and the line joining the points P and max_x are new lines, and the points residing outside the triangle are the set of points. Repeat line number 3 till there is no point left with the line. Add the endpoints of this point to the convex hull.

Example:

Find a convex hull of a set of points given below.

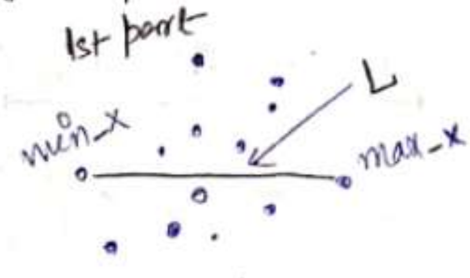
Example :

Find a convex hull from a set of points given below:

Sol^n :

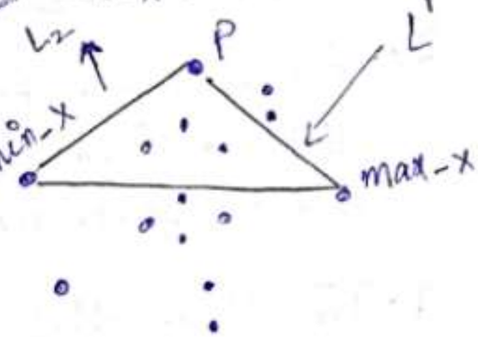
Step 1: Find the point with minimum x-coordinate (min_x) and similarly the points with maximum x-coordinate (max_x)

Step 2: Make a line (L) joining these two points.

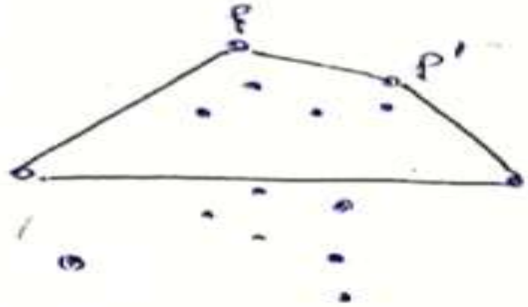


2nd part :

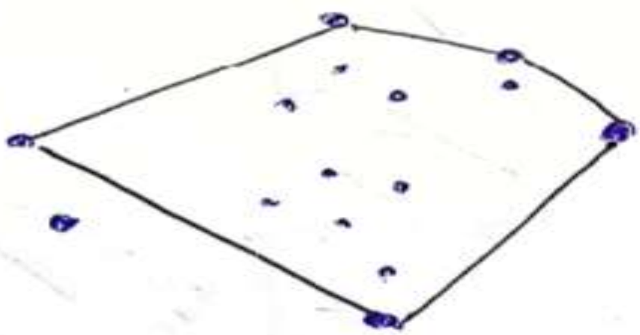
Step 3: For the 1st part, find the point P with maximum distance from the line L. P form a $\triangle$ with min_x and max_x.

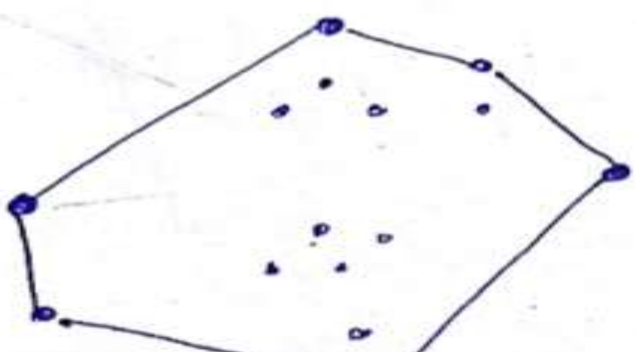


Step 4: Now, we have two new lines : L_1 and L_2. Now repeat the line – 3.



Step 5: Do the same with the second part.





↳ convex hull

Matrix Multiplication

Matrix Multiplication

$$\boxed{A_{p\times q} * B_{q\times r} = C_{p\times r}}$$

$$\underbrace{\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}_{2\times 2}}_{A} \times \underbrace{\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}_{2\times 2}}_{B} = \underbrace{\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}_{2\times 2}}_{C}$$

$c_{11} = a_{11} \times b_{11} + a_{12} * b_{21}$.

$c_{12} = a_{11} \times b_{12} + a_{12} \times b_{22}$

$c_{21} = a_{21} \times b_{11} + a_{22} \times b_{21}$

$c_{22} = a_{21} \times b_{12} + a_{22} \times b_{22}$

```
For( i=0 ; i<n ; i++)  => n+1 = (n)
{                                    *
  For( j=0 ; j<n ; j++)  => n+1 = (n)     O(n^3)
  {                                  *
      C[i][j] = 0;
      For( k=0 ; k<n ; k++)  => n+1 = (n)
      {
         C[i][j] = C[i][j] + A[i][k] * B[k][j];
      }
  }
}
```

All loops are independent of each other, so ... How many times this statement will be executed ⇓ Time Complexity

How can we apply Divide and Conquer technique to solve two matrices' multiplication of order more than 2x2 ?

$$A = \left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right]_{4\times 4} \qquad A_{11},\ A_{12},\ A_{21},\ A_{22}$$

$$B = \left[\begin{array}{cc|cc} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ \hline B_{31} & B_{32} & B_{33} & B_{34} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{array}\right]_{4\times 4} \qquad B_{11},\ B_{12},\ B_{21},\ B_{22}$$

```
MM(A, B, n)
{
   if (n ≤ 2)
   {
      C11 = A11 * B11 + A12 * B21
      C12 = A11 * B12 + A12 * B22
      C21 = A21 * B11 + A22 * B21
      C22 = A21 * B12 + A22 * B22
   }
   else
   {
      MM(A11, B11, n/2) + MM(A12, B21, n/2)
      MM(A11, B12, n/2) + MM(A12, B22, n/2)
      MM(A21, B11, n/2) + MM(A22, B21, n/2)
      MM(A21, B12, n/2) + MM(A22, B22, n/2)
   }
}
```

$$T(n) = \begin{cases} 8T(n/2) + n^2 & n > 2 \\ 1 & n \le 2 \end{cases}$$

using master theorem:

$a = 8, b = 2, k = 2, p = 0$

$a > b^k$ Case-I

$\therefore \Theta(n^{\log_b a})$

$\Theta(n^{\log_2 8})$

$= \boxed{\Theta(n^3)}$

Note: If we apply simple method to multiply two matrices or apply D&C, we get $\Theta(n^3)$ running time.

[* Strassen's Matrix Multiplication *]

$P = (A_{11} + A_{22}) * (B_{11} + B_{22})$

$Q = (A_{21} + A_{22}) * B_{11}$

$R = A_{11} * (B_{12} - B_{22})$

$S = A_{22} * (B_{21} - B_{11})$

$T = (A_{11} + A_{12}) * B_{22}$

$U = (A_{21} - A_{11}) * (B_{11} + B_{12})$

$V = (A_{12} - A_{22}) * (B_{21} + B_{22})$

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$

$$T(n) = \begin{cases} 7T(n/2) + n^2 & n > 2 \\ 1 & n \le 2 \end{cases} \quad \Theta(n^{2.81})$$

V.V.I

Show all the steps of Strassen's matrix multiplication algorithm to multiply the following matrices.

$$A = \begin{bmatrix} 1 & 3 \\ 7 & 5 \end{bmatrix} \quad B = \begin{bmatrix} 6 & 7 \\ 3 & 8 \end{bmatrix}$$

Sol^n:

$A_{11} = 1$ | $B_{11} = 6$
$A_{12} = 3$ | $B_{12} = 7$
$A_{21} = 7$ | $B_{21} = 3$
$A_{22} = 5$ | $B_{22} = 8$

$P = (A_{11} + A_{22}) * (B_{11} + B_{22})$
$\Rightarrow 84$

$Q = (A_{21} + A_{22}) * B_{11}$
$\Rightarrow 72$

$R = A_{11} * (B_{12} - B_{22})$
$\Rightarrow -1$

$S = A_{22} * (B_{21} - B_{11})$
$\Rightarrow -15$

$T = (A_{11} + A_{12}) * B_{22}$
$\Rightarrow 32$

$U = (A_{21} - A_{11}) * (B_{11} + B_{12})$
$\Rightarrow 78$

$V = (A_{12} - A_{22}) * (B_{21} + B_{22})$
$\Rightarrow 32$

$C_{11} = P + S - T + V$
$\Rightarrow 150$

$C_{12} = R + T$
$\Rightarrow 31$

$C_{21} = Q + S$
$\Rightarrow 57$

$C_{22} = P + R - Q + U$
$\Rightarrow 89$

$$C = \begin{bmatrix} 150 & 31 \\ 57 & 89 \end{bmatrix}$$

Q2. Compare and contrast BFS and DFS. How do they fit into the decrease and conquer strategy?

Decrease and Conquer Strategy: The name 'divide and conquer' is used only when each problem may generate two or more sub-problems. The name 'decrease and conquer' is used for the single sub-problem class. The Binary search rather comes under decrease and conquer because there is one sub-problem. Other examples are BFS and DFS.

BFS (Breadth First Search)

1. It is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.
2. It uses Queue data structure.
3. It is more suitable for searching vertices closer to the given source.
4. It requires more memory.
5. It considers all neighbors first and therefore not suitable for decision-making trees used in games or puzzles.

DFS (Depth First Search)

1. It is also a traversal approach in which the traversal begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.
2. It uses stack data structure.
3. It is more suitable when there are solutions away from source.
4. It requires less memory.

5. It is more suitable for game or puzzle problems. We make a decision, and the then explore all paths through this decision. And if this decision leads to win situation, we stop.

Greedy Method of Algorithm Design

As the name suggests it builds up a solution piece by piece locally, always choosing the next piece that offers immediate benefit. The main function of this approach is that the decision is taken on the basis of the currently available information.

Note— A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

Pseudo code of Greedy Algorithm

```
Greedy(arr[], n)
{
  Solution = 0;
  for i=1 to n
   {
       x = select (arr[i]);
       if feasible(x)
         {
          Solution = solution + x;
         }
    }
 }
```

Initially, the solution is assigned with zero value. We pass the array and number of elements in the greedy algorithm. Inside the for loop, we select the element one by one and checks whether the solution is feasible or not. If the solution is feasible, we add it to the solution.

Applications of Greedy Algorithm

- It is used in finding the shortest path.
- It is used to find the minimum spanning tree using the prim's algorithm or the Kruskal's algorithm.
- It is used in a job sequencing with a deadline.
- This algorithm is also used to solve the fractional knapsack problem.

Let's try to understand some terms used in the optimization problem.

Suppose we want to travel from Delhi to Mumbai as shown below:

Problem (P): Delhi(D) → Mumbai (M)

There are multiple solutions to go from D to M. We can go by walk, bus, train, airplane, etc., but there is a constraint in the journey that we have to travel this journey within 16 hrs. If we go by train or airplane then only, we can cover this distance within 16 hrs. Therefore, we have multiple solutions to this problem, but only two solutions satisfy the constraint, which are called feasible solutions.

If we say that we have to cover the journey at the minimum cost, then this problem is known as a minimization problem.

Till now, we have two feasible solutions, i.e., one by train and another one by air. Since travelling by train cost us minimum, it is an optimal solution. The problem that requires either minimum or maximum result is known as an optimization problem. Greedy method is one of the strategies used for solving the optimization problem. A Greedy algorithm makes good local choices in the hope that the solution should be either feasible or optimal.

Q3. Describe optimization problem, feasible solution and optimal solution.

Optimization problem – An optimization problem refers to a computational problem where the goal is to find the best solution from a set of possible choices (called feasible solutions). The objective is to either maximize or minimize a specific criterion while adhering to a set of constraints. We have the following methods to solve optimization problems:

1. Greedy
2. Dynamic programming
3. Branch and bound

Feasible solution - Most of the problems have ‘n’ inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. A problem can have many feasible solutions.

“A feasible solution satisfies all constraints of the problem.”

Optimal solution - It is the best solution out of all possible feasible solutions.

Q4. What is principle of optimality?

A problem is said to satisfy the Principle of Optimality if the sub solutions of an optimal solution of the problem are themselves optimal solutions for their subproblems. For example, the shortest path problem satisfies the principle of optimality.

Q5. Differentiate between Greedy approach and Dynamic programming approach.

Greedy Approach:

1. We make a choice that seems best at the moment in the hop that it will lead to global optimal solution.
2. It does not guarantee an optimal solution.
3. It takes less memory.
4. Fractional Knapsack is an example of Greedy approach.

Dynamic Programming Approach:

1. we make a decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution.
2. It guarantees an optimal solution.
3. It takes more memory.
4. 0/1 Knapsack is an example of Dynamic programming approach.

Q6. Find the optimal solution for the fractional knapsack problem with n=7 and a knapsack capacity of m=15, where the profits and weights of the items are as follows: (p1, p2, . . . , p7) = (10, 5, 15, 7, 6, 18, 3) and (w1, w2, . . . , w7) = 2, 3, 5, 7, 1, 4, 1, respectively.

Capacity of knapsack(bag) = 15

We have to put objects in the bag (knapsack) such that we should get maximum profit.

Selection of the object can be entirely (x=1), in fraction (0<=x<=1) or not selected (x=0).

Object	Profits	Weights	P/W
1	10	2	5
2	5	3	1.6
3	15	5	3
4	7	7	1
5	6	1	6
6	18	4	4.5
7	3	1	3

We select an object according to its P/W ratio. An object with maximum P/W ratio will be selected first and then second maximum P/W ratio and so on.

Remaining = Capacity of Knapsack- Weight of the selected object

Final table according to decreasing P/W ratio

Objects	Profits(P)	Weights	P/W	Remaining	Selection(X)
5	6	1	6	15-1=14	1
1	10	2	5	14-2=12	1
6	18	4	4.5	12-4=8	1
3	15	5	3	8-5=3	1
7	3	1	3	3-1=2	1
2	5	3	1.66	2-2=0	2/3
4	7	7	1	0	0

Object-4 is not selected; therefore, value of x for this object is zero. Object-2 is selected only 2 units out of 3 units, so its value for x is (2/3).

$$\text{Profit} = X_i * P_i$$

Profit = 1* 6 + 1*10 + 1*18 + 1*15 + 1*3+ (2/3) * 5 + 0*7
= 6 + 10 +18+ 15+ 3+3.3 +0
= 55.3

Job Sequencing with Deadlines Problem

We are given a set of n jobs. Associated with job i is an integer deadline $d_i >= 0$ and a profit $p_i > 0$. For any job i, the profit p_i is earned if and only if the job is completed by its deadline. The objective is to maximize the total profit by scheduling jobs in a way that they meet their deadlines.

The constraints in this problem include:

Each job has a specific deadline by which it must be completed.
Each job takes one unit of time to complete. This one unit of time may be equal to one hour, one day, one week, or one month.
Only one machine is available for processing jobs.
We can only work on one job at a time.
We cannot exceed the specified deadlines.
No preemption.

Q. Using a greedy method, find the optimal solution for the "job sequencing problem with deadlines" with n = 4, where (p1, p2, p3, p4) = (100, 10, 15, 27) and (d1, d2, d3, d4) = (2, 1, 2, 1).

sol^n: ↓

one unit of time.

Processing Sequence

Time slots according to max^m deadlines ⟹

0 1 2

one unit of time, one unit of time

Constraints:
(i) Each job completes in one unit of time with no preemption.
(ii) Deadline should not be violated.

Step 01: Pick a job with the highest profit and complete it as late as possible.

job-1 has the highest profit with 100 and its deadline is 2 units of time. It means we can complete it either in the first slot or in the second slot. we place it in the second slot according to algo.

	J_1

0 1 2

Step-02: Pick the next job with the highest profit. The job 4 has the second highest profit with deadline 1. It means we have to complete it in the one unit of time only, so place it in the first slot.

J_4	J_1

0 1 2

Profit = 100 + 27
= $\boxed{127}$ ← optimal sol^n.

Q. Identify all solutions satisfying the constraints for the “job sequencing with deadlines” problem with n = 4, where (p1, p2, p3, p4) = (100, 10, 15, 27) and (d1, d2, d3, d4) = (2, 1, 2, 1).

Solution— We know that we can have multiple solutions for an optimization problem, but feasible solutions are only those that satisfy all constraints of the problem. Therefore, we need to identify all feasible solutions for this problem.

	Feasible solutions	Processing sequence	value
1	(1, 2)	2, 1	110
2	(1, 3)	1,3 or 3,1	115
3	(1, 4)	4, 1	127
4	(2, 3)	2, 3	25
5	(3, 4)	4, 3	42
6	(1)	1	100
7	(2)	2	10
8	(3)	3	15
9	(4)	4	27

optimal

⇓ 9 feasible solutions

Activity Selection Problem

The activity selection problem involves selecting a maximum number of non-overlapping activities from a given set of activities, each with a **start time** and **finish time**. The goal is to choose a set of activities that do not overlap in time and, therefore, can all be completed.

The constraints in this problem include:

You can only perform one activity at a time.
The activities must be selected in a way that none of them overlap in time.

Q. Using Greedy method, find an optimal solution to the activity selection problem with the following information:

Activity	A1	A2	A3	A4	A5	A6	A7	A8
Start	1	0	1	4	2	5	3	4
Finish	3	4	2	6	9	8	5	5

Solution—

sol^n: ↓

Step 01: Sort the activities in increasing order of their finishing time:

i	Sorted Activity	A_3	A_1	A_2	A_7	A_8	A_4	A_6	A_5
S_i	Start	1	1	0	3	4	4	5	2
F_i	Finish	2	3	4	5	5	6	8	9

Step 02: Select the first activity

✓ $\{A_3\}$

Step 03: Select the next activity if its starting time is "$\geq$" to the previously selected activity and repeat this step until all activities are considered.

Next activity $= A_1$

Start time of A_1 $(SA_1) = 1$

Finish time of A_3 $(FA_3) = 2$

$\boxed{\underset{1}{SA_1} \geq \underset{2}{FA_3}} \Rightarrow$ false

We move to the next activity (A_2).

$\boxed{\underset{0}{SA_2} \geq \underset{2}{FA_3}} \Rightarrow$ false

We move to the next activity (A_7).

✓ $\boxed{\underset{3}{SA_7} \geq \underset{2}{FA_3}} \Rightarrow$ True

include A_7 to the result and move to the next activity.

$\boxed{SA_8 \geq FA_7} \Rightarrow$ false

$SA_8 = 4$, $FA_7 = 5$ → previously selected activity

We move to the next activity (A_4).

$\boxed{SA_4 \geq FA_7} \Rightarrow$ false.

$SA_4 = 4$, $FA_7 = 5$

We move to the next activity (A_6).

$\boxed{SA_6 \geq FA_7}$ True

$SA_6 = 5$, $FA_7 = 5$

include A_6 to the sol^n and move to the next activity (A_5) :

$\boxed{SA_5 \geq FA_6}$ — false

$SA_5 = 2$, $FA_6 = 8$ → previously selected activity.

Selected Activity	A_3	A_7	A_6
Start	1	3	5
Finish	2	5	8

Graphs

1

A graph is a collection of vertices (also called nodes) and edges.

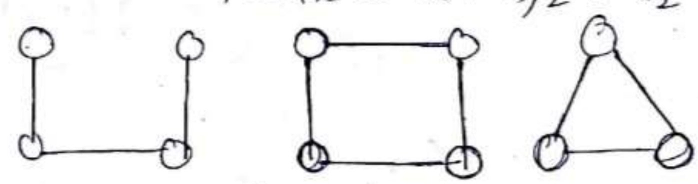
We have two types of graphs on the basis of whether they allow loops or multiple edges between the same pair of vertices :

(1) Simple Graph
(ii) Multigraph

(1) Simple Graphs :

* There is at most one edge between any pair of vertices. Min^m is zero (NULL graph)
* There are no loops (edges that connect a vertex to itself).
* Max^m no of edges $= n(n-1)/2$ or $^nC_2$.

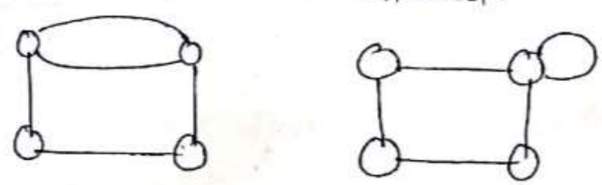
Example :



(ii) Multigraph :

* It allows multiple edges between the same pair of vertices.
* Loops are allowed.

Example :



Complete Graph G(V,E)

(21)

* It is a type of simple graph in which every pair of distinct vertices is connected by an edge.

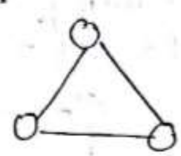
Characteristics of a Complete Graph:

(i) The number of edges with "n" vertices is given by the formula $E = \frac{n(n-1)}{2}$ or ${}^{n}C_2$

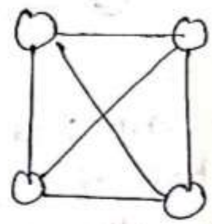
(ii) Every vertex is connected to every other vertex in the graph.

(iii) It is a simple graph, meaning it has no loops or multiple edges between the same pair of vertices.

Example: ↓



$= \frac{n(n-1)}{2} = \frac{3 \times \cancel{2}}{\cancel{2}} = 3$ edges



$= \frac{n(n-1)}{2} = \frac{\cancel{4}^{2} \times 3}{\cancel{2}} = 6$ edges

or

${}^{4}C_2 = \frac{4!}{2!\,(4-2)!} = \frac{4 \times 3 \times \cancel{2!}}{2! \times \cancel{2!}} = 6$ edges

Spanning Tree (ST)

3

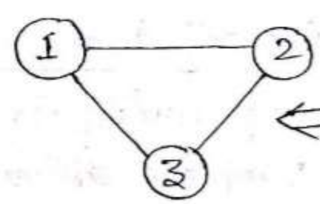
"A spanning tree is a subset of graph G that covers all vertices with the minimum possible number of edges."

Characteristics of a spanning Tree

1. It includes all the vertices from the original graph.
2. It is acyclic, meaning no loops or no cycles.
3. It is connected, meaning all vertices are reachable from every other vertex through a series of edges.

undirected graph ⇒

unweighted graph ⇒



⇐ this is a labeled graph

⇐ complete graph
⇓
$\frac{n(n-1)}{2}$ edges

vertices = $\{1, 2, 3\}$

Edges = $\{(1,2), (1,3), (2,3)\}$

Fig: A complete graph with 3 vertices

Following are the spanning trees of the above G.

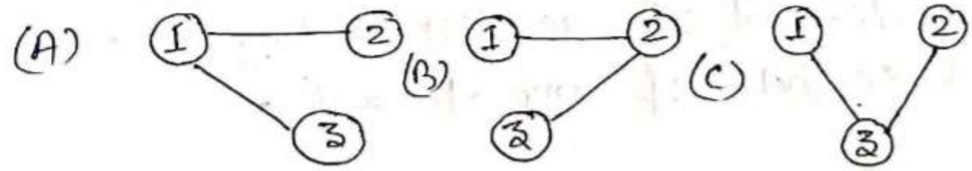


Fig: Three possible spanning trees of the above G.

**Note: In a connected, undirected graph with "n" vertex, a spanning tree will always have exactly "n-1" edges.

*** Counting of Maximum Number of Spanning Trees of a Given Graph.

⇒ For a "complete (K_n) graph", if it has "n" vertices, then we have n^{n-2} ~~vert~~ spanning trees.

⇒ this formula is valid only for the complete graph.

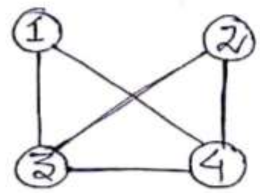
what about if the graph is not complete?

Ans: → using Kirchoff theorem, we can find the all possible spanning trees of either ~~Connected~~ complete or not complete graph.

Kirchoff theorem for counting all possible ST_s of a G.

1. Construct an adjacency matrix for the given graph.
2. Diagonal zeroes are replaced with the degree of the nodes.
3. Non-diagonal 1's are replaced with "−1".
4. Non-diagonal 0's are left as they are.
5. Find co-factor of any element.

Q Find maximum numbers of spanning trees out of the graph given below: (5)



undirected G ✓
un-weighted G ✓
non-complete G ✓
labeled G ✓
simple G ✓

solⁿ: ↓

1. Construct an adjacency matrix for the given G.

$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ 1 & 0 & 0 & 1 & 1 \\ 2 & 0 & 0 & 1 & 1 \\ 3 & 1 & 1 & 0 & 1 \\ 4 & 1 & 1 & 1 & 0 \end{array}$$

→ principal diagonal

2. update the following info of the adjacency mat:

principal diagonal

(a) Diagonal 0's ⇒ Replaced with the degree of the nodes.

(b) Non-diagonal 1's ⇒ Replaced with "−1".

(c) Non-diagonal 0's are left as they are.

⇓ Any zero values that are not on the diagonal remain unchanged.

$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ 1 & 2 & 0 & -1 & -1 \\ 2 & 0 & 2 & -1 & -1 \\ 3 & -1 & -1 & 3 & -1 \\ 4 & -1 & -1 & -1 & 3 \end{array}$$

3. Find co-factor of any element to find the number of STs possible.

$$
\begin{array}{c|cccc|}
 & 1 & 2 & 3 & 4 \\
1 & 2 & 0 & -1 & -1 \\
2 & 0 & 2 & -1 & -1 \\
3 & -1 & -1 & 3 & -1 \\
4 & -1 & -1 & -1 & 3
\end{array}
$$

Let's find co-factor of A[1][1].

$$+(2)\begin{vmatrix} 3 & -1 \\ -1 & 3 \end{vmatrix} - (-1)\begin{vmatrix} -1 & -1 \\ -1 & 3 \end{vmatrix} + (-1)\begin{vmatrix} -1 & 3 \\ -1 & -1 \end{vmatrix}$$

$$2(9-(-1\times-1)) + (-1\times3-(-1\times-1)) - (1-(3\times-1))$$

$$2(9-1) + (-3-1) - (1+3)$$

$$2\times8 - 4 - 4$$

$$16 - 8 = 8$$

Total no of spanning Trees

ST

Q7. Define spanning tree and minimum spanning tree with an example.

Spanning Tree – It is a subset of graph G having all its vertices covered with minimum possible number of edges. If there are 'n' vertices in an undirected connected graph, then every possible spanning tree out of this graph has "n-1" edges. It does not have a cycle.

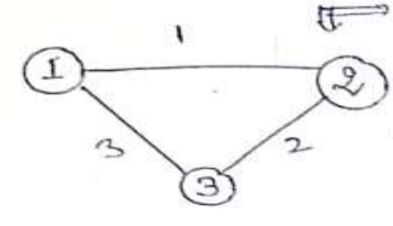
Minimum Spanning Tree (MST) – An MST for a weighted, connected, undirected graph is a spanning tree having a weight less than or equal to the weight of every other possible spanning tree.

Example:

⇐ connected graph.

G:

undirected ⇒ weighted graph



⇐ This is a labeled graph.

fig 1: A complete graph with 3 vertices (K_3).

vertices = { 1, 2, 3 }

Edges = { (1,2), (1,3), (3,2) }

⇓ All possible spanning trees out of G.

As G has three vertices, every possible ST will have only two edges.

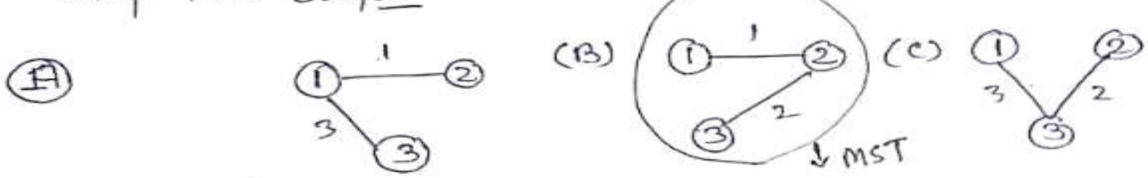


fig 2: Three possible spanning trees out of G.

Note: For a $\boxed{\text{Complete graph}}$, if it has "n" vertices then we have $\boxed{n^{n-2}}$ spanning trees.

(B) is the minimum spanning tree for G because its cost is least (3) among other STs.

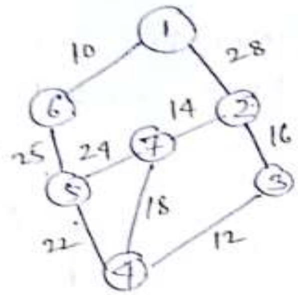
Prim's Algorithm for finding an MST

```
Prim( E, cost, n, t) {
// E is the set of edges. cost is (nxn) adjacency matrix
// MST is computed and stored in array t[1:n-1, 1:2]
1. let (k, l) be an edge of minimum cost in E
2. mincost = cost[k, l];
3. t[1,1] = k ; t[1,2] = l;
4. For i = 1 to n
5.        if ( cost[i, l] < cost [i, k]) then near[i] = l ;
6.        else near[i] = k ;
7.      near[k] = near [l] = 0
8. For(i = 2 to n-1)
9. let j be an index such that near[j] ≠ 0 and
10. cost[j, near[j]] is minimum ;
11. t[i,1] = j , t[i,2] = near[j];
12. mincost = mincost + cost[j, near [j]];
13. near [j] = 0
14. For k = 1 to n
15. if ((near[k] ≠ 0 and (cost[k, near[k]] > cost[k, j ]) then
        near [k] = j ;
    }
```

Q8. Give an example of an MST using Prim's algorithm for a connected graph.

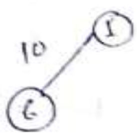
Example : { MST using Prim's algo }

$\boxed{\Theta(n^2)}$ # vertices

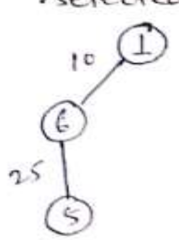


⟸ 7 vertices

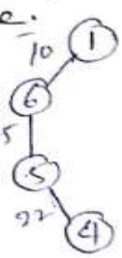
Step 1: Select minimum cost edge

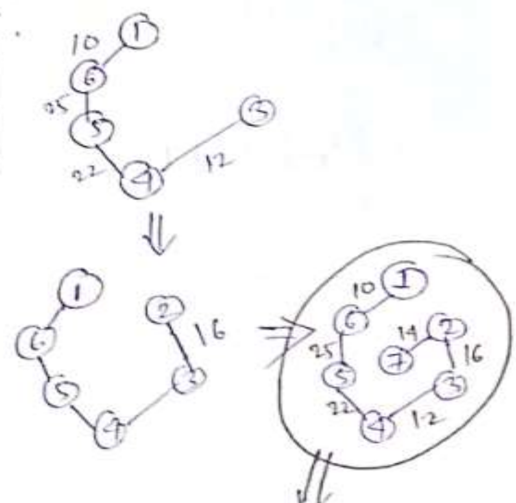


Step 2: Select another minimum cost edge which is ~~already~~ connected to already selected vertices.



Follow step 2 to select the next minimum cost edge.





Cost $\boxed{99}$ Ans

MST with 6 edges.

Kruskal's Algorithm for finding an MST

Algorithm:

1. Sort all the edges in increasing order of their weight.

2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.

3. Repeat step#2 until there are (V-1) edges in the spanning tree.

Single Source Shortest-Paths Problem

1. Dijkstra's Algorithm (Greedy)
2. Bellman-Ford Algorithm (Dynamic)

Given a graph G = (V, E), we want to find a shortest path from a given source vertex s ϵ V to each vertex v ϵ V.

1. Dijkstra's Algorithm for Single Source Shortest Paths

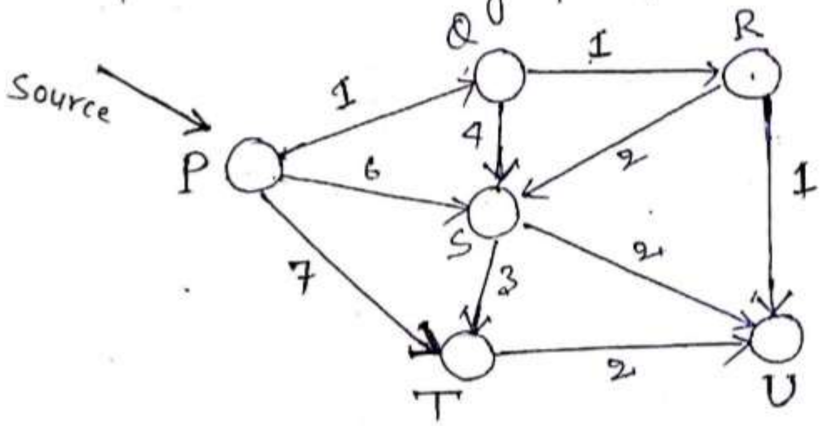
Dijkstra's algorithm solves the single source shortest-paths problem on a weighted graph G = (V , E) for the case in which all edge weights are nonnegative. It works for directed as well as undirected graph. It may or may not work with negative edge weights.

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = Ø
Q = G.V
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

(1)

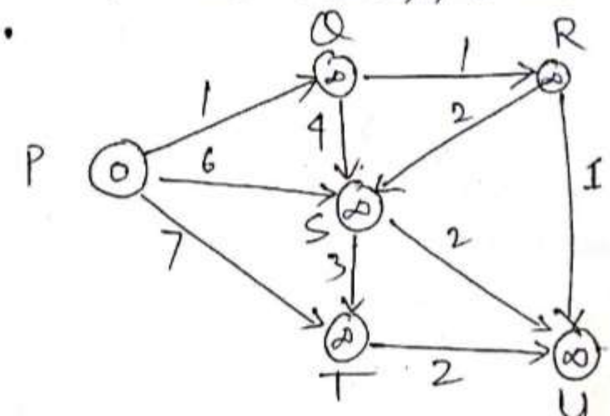
Q. Apply the greedy single source shortest path algorithm on the graph given below:

(Dijkstra's Algo)



Sol^n:

Step-01 The source vertex is P, and distance of P to P is zero. We do not know the distance between P to all other vertices, so we mark them with ∞.



$\infty \Rightarrow$ Not Known

Step-02: Now, consider all outgoing edges from the vertex P and perform "relaxation".

Relaxation:

$$\text{If } (\text{distance}[u] + \text{cost}(u,v) < \text{distance}[v]) \Rightarrow \text{True}$$

then

$$\text{distance}[v] = \text{distance}[u] + \text{cost}(u,v)$$

For P to Q edge:

$u = 0$, $cost(u,v) = 1$ and $v = \infty$

$\therefore \boxed{0+1 < \infty}$ True.

Update the value of Q with 1.

For P to S edge:

$\boxed{0+6 < \infty}$ True

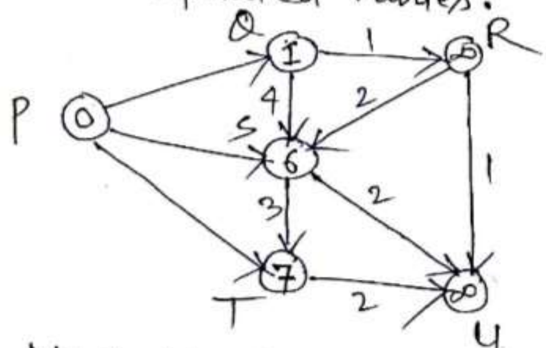
Update the value of S with 6.

For P to T edge:

$\boxed{0+7 < \infty}$ True

Update the value of T with 7.

We have got the graph with updated values.



Now, we have to pick the vertex with the least value from (Q, S, T), which is Q.

Step-03: Now, consider all outgoing edges from the vertex Q and perform "relaxation". 21

We have two edges going from Q, which are QR and QS.

For Q to R edge:

$\boxed{1+1 < \infty}$ True

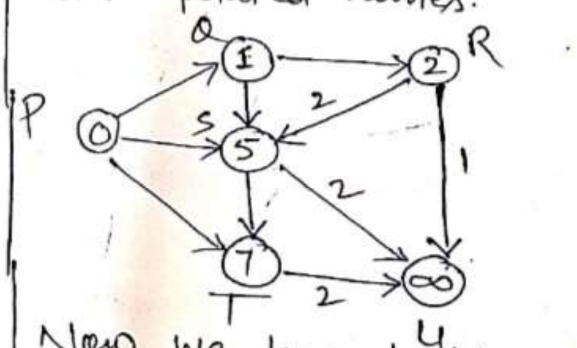
Update the value of R with 2.

For Q to S edge:

$\boxed{1+4 < 6}$ True

Update the value of S with 5.

Again we have the graph with updated values.



Now, we have to pick the vertex with the least value from (R, S, T), which is R.

Step 04: Now, consider all outgoing edges from the vertex "R" and perform "relation". (3)

We have two outgoing edges from R, which are RS and RU.

For R to S edge:

$\boxed{2+2 < 5}$ True

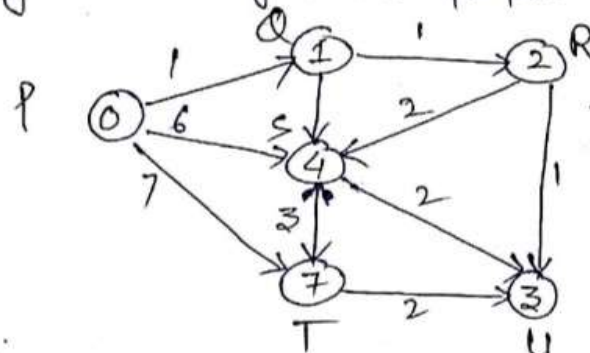
update the value of S with 4.

For R to U edge:

$\boxed{2+1 < \infty}$ True

update the value of U with 3.

Again we got the graph with updated values.



Now, we have to pick the vertex with the least value from (S, T, U), which is ~~S~~ "U".

"U" does not have any outgoing edge, so we consider the next smallest vertex, which is "S".

"S" has two outgoing edges, but both of them are already the least:

S to T edge

$\boxed{4+3 < 7}$ false

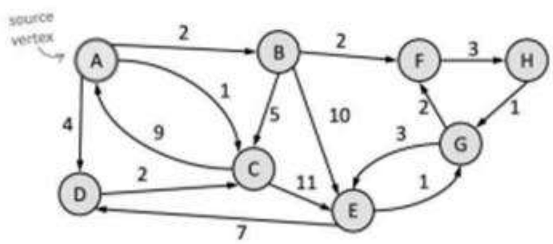
S to U edge

$\boxed{5+2 < 3}$ false

∴ P to Q = 1
P to S = 4
P to T = 7
P to R = 2
P to U = 3

⇒ Five vertices have least distance from "P"

Q. Apply the greedy single source shortest path algorithm on the graph given below.



2. Bellman-Ford Algorithm (Dynamic)

BELLMAN-FORD(G, w, s)

{

1. INITIALIZE-SINGLE-SOURCE (G, s)
2. for $i = 1$ to $|G.V| - 1$ $\Rightarrow (V-1)$
3. for each edge $(u,v) \in G.E \Rightarrow (E)$ } $O(VE)$
4. RELAX(u, v, w) – O(1)
5. For each edge $(u,v) \in G.E \Rightarrow O(E)$
6. if $v.d > u.d + w(u,v)$
7. return FALSE
8. return TRUE

}

$\boxed{O(VE)}$

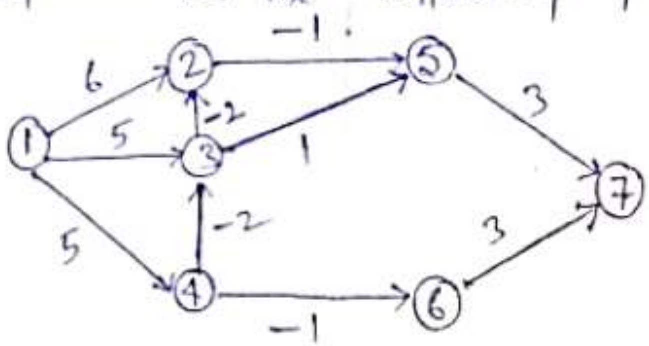
Note:-

(I) It can detect –ve weight cycle.

(II) It is applicable for –ve weight edges.

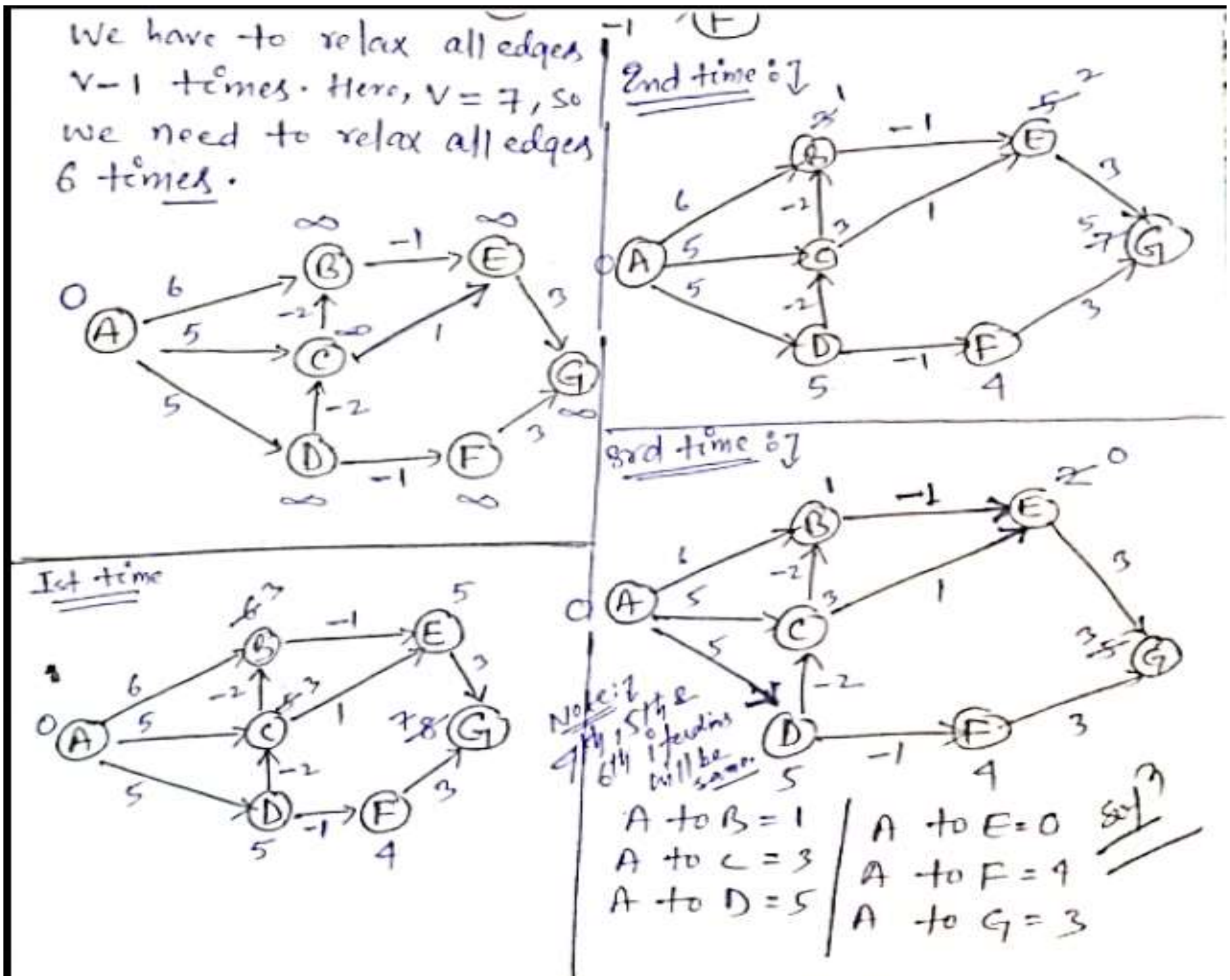
(III) we relax all edges till "V–1" times.

Apply the Dynamic Programming single source shortest path algorithm on the following graph:



EdgeList → [(1,2), (1,3), (1,4), (2,5), (3,2), (3,5), (4,3), (4,6), (5,7), (6,7)]

⇓ we need to relax all edges 6 times as the graph has 7 vertices.



Unit-04

Dynamic Programming

It is one of the algorithm design techniques used to solve optimization problems. It is mainly an optimization over plain recursion. Wherever we encounter a recursive solution with repeated calls for the same inputs, we can optimize it using Dynamic Programming. The idea is to store the results of subproblems so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

Principle of Optimality : The principle of optimality, developed by Richard Bellman, is the basic principle of dynamic programming. It states that in an optimal sequence of decisions, each subsequence must also be optimal.

Memoization : It is the top-down approach (start solving the given problem by breaking it down) . If we want, we can use this approach in Dynamic programming as well, but we generally use iterative method (tabulation method), which is the bottom-up approach, in Dynamic programming.

Let's try to understand Dynamic Programming approach with a suitable example.

Find Fibonacci term using plain recursion (recursive program).

Fibonacci series : 0 1 1 2 3 5 . . .
Fn: 0 1 2 3 4 5 . . . (Fibonacci terms)
F3 term= 2 , F1 term=1 , F4 term=3, etc.

```
int fib(int n)
 {
    if(n<=1)
      return n;
   return fib(n-2) + fib(n-1);
 }
```

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

Time complexity (Upper bound)

$T(n) = 2T(n-1) + 1$ [Since $T(n-1)$ is almost equal to $T(n-2)$]

Using master method for decreasing functions, we get the time complexity $O(2^n)$, which is exponential.

Now, try to observe repeated recursive calls for the same argument (input value) using a recursive tracing tree.

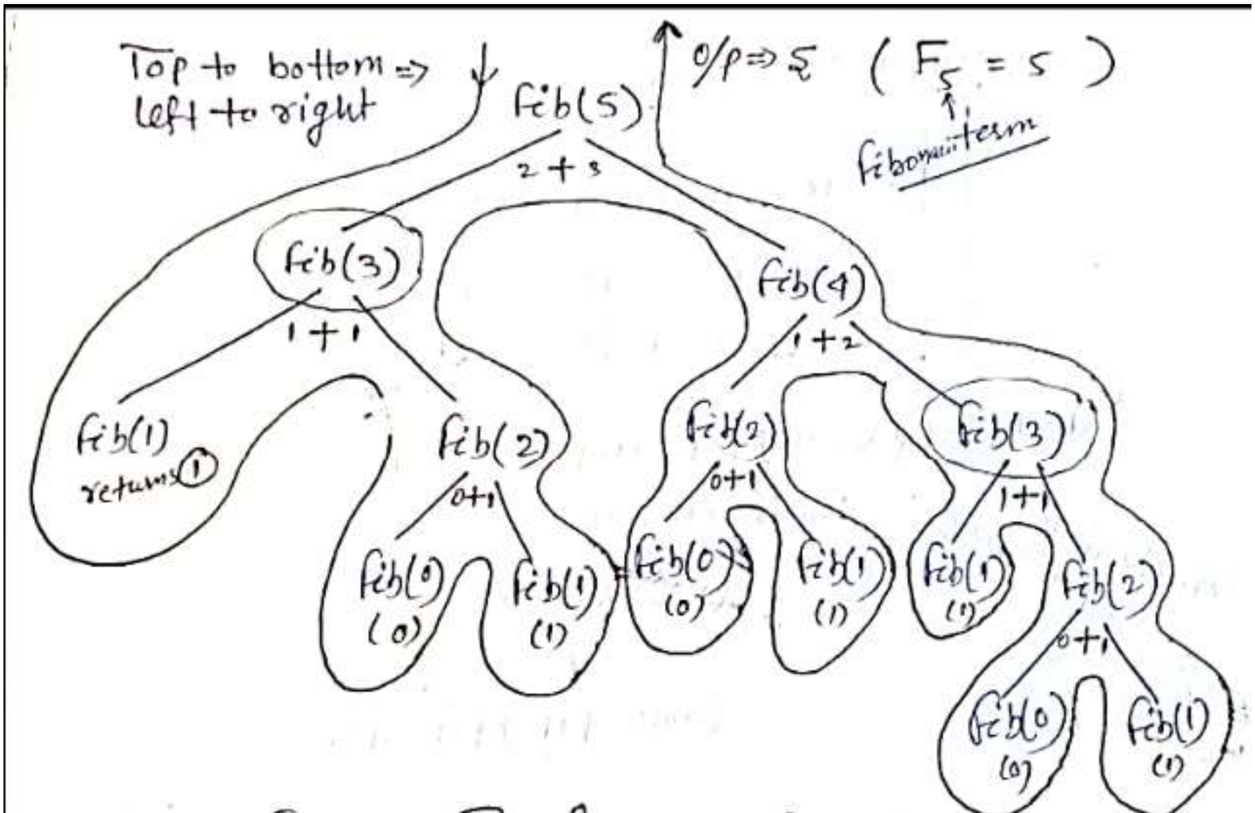


Fig 1: Tracing tree for fib(5)

Count of Repeated Recursive calls in fig 1:

fib(3) – 2 times repeated, fib(2) – 3 times repeated, fib(1) – 5 times repeated, and fib(0) -- 3 times repeated

We have got repeated recursive calls for the same input. It makes this approach have exponential running time. It is where Dynamic Programming approach comes into the picture, which reduces time complexity drastically by avoiding repetitive computation for the same recursive call.

Find Fibonacci term using memoization (Dynamic Programming Approach).

```
int F[20];    // Global array

int fib(int n)      // Function definition
     {

          if(n <= 1)
             return n;
          if(F[n] != -1)
             return F[n];

         F[n] = fib(n-2) + fib(n-1); // recursive call
         Return F[n];
     }

void main(void)
 {
   int  i, result=0;

   for( i=0 ; i< 20 ; i++)
        F[i] = -1;
   result = fib(5);
   printf("%d", result);   }
```

From the above example, we can observe the following points:

If we use memoization method to solve the same problem, we don't have to go for repetitive computation for the same recursive calls. It means for fib(5), we have to compute recursive function calls only 6 times (fib(5), fib(4), fib(3), fib(2), fib(1) and fib(0)).

If we generalize it for fib(n), the number of recursive calls will be n+1.It means time complexity will be O(n)- linear.

Note – We generally don't use the memoization method in Dynamic programming as it consumes more space due to recursion.

Note – Memoization follows top-down approach.

Iterative Method (tabulation method) for the Same Problem [bottom-up approach]

```
int F[20];

int fib(int n)
{
    if(n <=1)
      {
       return n;
      }
  F[0]=0;
  F[1]=1;
  for(int i = 2; i<=n; i++)
     {
         F[i]= F[i-2] + F[i-1];
     }
return F[n];
}
```

1. 0/1 Knapsack Problem

The knapsack problem deals with putting items in the knapsack based on the value/profit of the items. Its aim is to maximize the value inside the bag. In 0-1 Knapsack, you can either put the item or discard it; there is no concept of putting some part of an item in the knapsack like fractional knapsack.

Q. Find an optimal solution to the 0/1 Knapsack instances n=4 and Knapsack capacity m=8 where profits and weights are as follows p= {1, 2, 5, 6} and w = {2, 3, 4, 5}

Note – If weights are not given in the increasing order, then arrange them in the increasing order and also arrange profits accordingly.

The matrix (mat[5][9]) will contain 9 columns (as capacity (m) = 8 is given) and 5 rows (as n= 4 is given)

P_i = profits

W_i = weights

i = Objects

Formula to fill out cells : mat[i, w] = max (mat[i-1, w], mat[i-1, w-weight[i]+ p[i])

Short-cut to fill the table

1. Fill the first row and the first column with zero.

2. For the first object, check the weight (w_i) of the first object, which is 2. We have capacity w=2, so place profit of this object in the cell having capacity of 2 units (mat[1][2]=1). So far, we have only one object to consider , so we can put the first object (i = 1) having 2 units of weight (w_1 = 2) in the knapsack having capacity (w)

3,4,5,6,7 and 8 units. Therefore, fill mat[1][3], mat[1][4], mat[1][5], mat[1][6], mat[1][7] and mat[1][8] with 1.

3. For the cell(s) left side of the current cell, we just consider the maximum value between left side and above of the current cell. For example, for the left side of mat[1][2], we need to pick max(mat[1][1], and mat[0][2]), which is 0. Therefore, place zero in the mat[1][1].

4. For the second object, weight is given 3 units. Now, we can consider two objects (1 and 2) together. The second object having 3 units of weight can be placed in the cell [2][3] having 3 units of capacity. Both objects together have 5 units of weight, which can be placed in the cells [2][5], [2][6], [2][7] and [2][8] having 5 units of capacity. For the cell [2][2], pick max(mat[2][1], mat[1][2]) which is 1. And follow the same for the cell [2][4].

5. For the third object, 4 units of weight is given. Now, we can consider three objects (1,2, and 3 objects) together . Weight of the third object is 4 units , so we can place its profit (5) in the cell [4][4] having 4 units of capacity. Objects 2 and 3 together have 7 units of weight and 7 units of profit (5+2), so we can place them in the cell [3][7] having 7 units of capacity. Object 1 and 3 together have 6 units of weight, so we can place them in the cell [3][6] having 6 units of capacity. To fill out remaining cells , follow above steps.

Maximum profits = 8 (placed in the last cell of the matrix)

Selection of objects $X_i = X_1\ X_2\ X_3\ X_4$ (0 1 0 1)

Only objects 2 and 4 have been placed in the knapsack to gain maximum profit.

Instances/ Object(i)

Capacity(w)

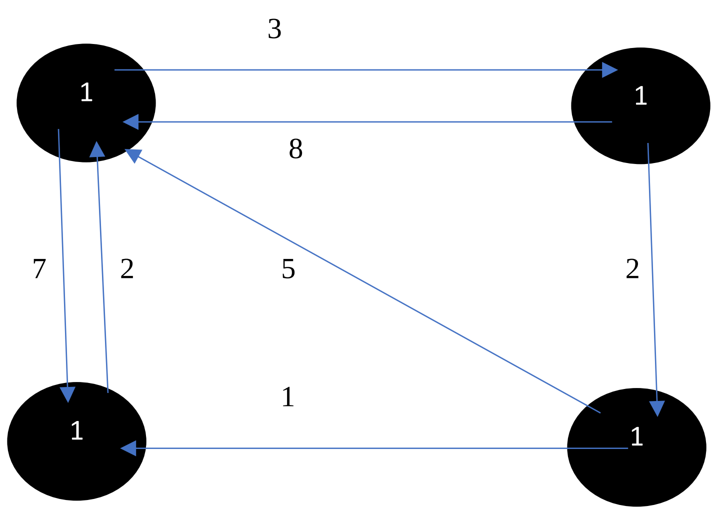
P_i	W_i		0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5	6	6	7	8

2. Single Source Shortest Path using Bellman-Ford Algorithm (Dynamic Programming)

Kindly refer unit-03 notes

3. All Pairs Shortest Path (Floyd-Warshall Algorithm)

Apply Floyd-Warshall algorithm on the below graph:



(1)

cost matrix

$$A^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & 2 & \infty \\ 3 & 5 & \infty & 0 & 1 \\ 4 & 2 & \infty & \infty & 0 \end{array}$$

cost matrix

Note: use "∞" for not having a direct path.

** Note - As we have 4 vertices, we need 4 matrices to solve this problem.

$$\Rightarrow A^1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & & \\ 3 & 5 & & 0 & \\ 4 & 2 & & & 0 \end{array}$$

For A^1, we copy the first row and the first column from A^0 and also the left diagonal.

for the rest of values, we use the below formula.

$$\boxed{A^k[i,j] = \min\{A^{k-1}[i,j],\ A^{k-1}[i,k] + A^{k-1}[k,j]\}}$$

for $A^1[2,3]$

$$A^1[2,3] = \min\left(A^0[2,3],\ A^0[2,1] + A^0[1,3]\right)$$
$$= \min(\ 2\ ,\ 8+\infty)$$
$$\boxed{A^1[2,3] = 2}$$ — update in A^1 matrix

for $A^1[2,4]$

$$\min\left(A^0[2,4],\ A^0[2,1] + A^0[1,4]\right)$$
$$(\ \infty\ ,\ \ 8 + 7)$$
$$\boxed{A^1[2,4] = 15}$$ — update in A^1

for $A^1[3,2]$

$$\min\left(\overset{\to\infty}{A^0[3,2]},\ \overset{\to 5}{A^0[3,1]} + \overset{\to 3}{A^0[1,2]}\right)$$
$$\boxed{A^1[3,2] = 8} \rightarrow$$ update in A^1

$A^1[3,4] = ?$

$\min(A^0[3,4], A^0[3,1]+A^0[1,4])$

$\min(1, 5+7)$

$\boxed{A^1[3,4] = 1}$ — update

$A^1[4,2] = ?$

$\min(A^0[4,2], A^0[4,1]+A^0[1,2])$

$\min(\infty, 2+3)$

$\boxed{A^1[4,2] = 5}$ — update

$A^1[4,3] = ?$

$\min(A^0[4,3], A^0[4,1]+A^0[1,3])$

$\infty \qquad 2+\infty$

$\boxed{A^1[4,3] = \infty}$ — update.

We get A^1 :

$$A^1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & 2 & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & \infty & 0 \end{array}$$

$$\Rightarrow A^2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & & \\ 2 & 8 & 0 & 2 & 15 \\ 3 & & 8 & 0 & \\ 4 & & 5 & & 0 \end{array}$$

For the second matrix A^2, we copy values of the second row and the second column and also the left diagonal.

Again use the same formula to update the second matrix

we get the second matrix:

$$A^2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 5 & 7 \\ 2 & 8 & 0 & 2 & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$

$$\Rightarrow A^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & & 5 & \\ 2 & & 0 & 2 & \\ 3 & 5 & 8 & 0 & 1 \\ 4 & & & 7 & 0 \end{array}$$

For the 3rd matrix, copy 3rd row and 3rd column from the 2nd matrix and also the left diagonal.

Again use the same formula to update missing values in A^3. We get the 3rd matrix

$$A^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 5 & 6 \\ 2 & 7 & 0 & 2 & 3 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$

$$\Rightarrow A_4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & & & 6 \\ 2 & & 0 & & 3 \\ 3 & & & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$

copy the 4th row and 4th column from A^3 and also the left diagonal.

Again use the same formula to update the matrix <u>A4.</u>

$$A^K[i,j] = \min\left(A^{K-1}[i,j],\ A^{K-1}[i,K] + A^{K-1}[K,j]\right)$$

we get the resultant matrix like this

$$A^4 = \begin{array}{c|c|c|c|c|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 5 & 6 \\ \hline 2 & 5 & 0 & 2 & 3 \\ \hline 3 & 3 & 6 & 0 & 1 \\ \hline 4 & 2 & 5 & 7 & 0 \\ \hline \end{array}$$

source

1 to 1 = 0
1 to 2 = 3
1 to 3 = 5
1 to 4 = 6

source

2 to 1 = 5
2 to 2 = 0
2 to 3 = 2
2 to 4 = 3

source

3 to 1 = 3
3 to 2 = 6
3 to 3 = 0
3 to 4 = 1

4 to 1 = 2
4 to 2 = 5
4 to 3 = 7
4 to 4 = 0

when source vertex is <u>4.</u>

4. Matrix Chain Multiplication Problem

We are given n matrices A1, A2, …. , An and asked in what order these matrices should be multiplied so that it would take a minimum number of computations to derive the result.

Two matrices are called compatible only if the number of columns in the first matrix and the number of rows in the second matrix are the same. Matrix multiplication is possible only if they are compatible. Let A and B be two compatible matrices of dimensions p x q and q x r. Each element of each row of the first matrix is multiplied with corresponding elements of the appropriate column in the second matrix.

The total number of multiplications required to multiply matrix A and matrix B is p x q x r.

Suppose dimension of two matrices are :

A1 = 5 x 4

A2 = 4 x 3

Resultant matrix will have 15 elements (5 rows and 3 columns), and each element in the resultant matrix is derived using 4 multiplications. It means 60 (5 x 4 x 3) multiplications are required.

We cannot multiply A2 = (4 x 3) and A1 = (5 x 4) as column of A2 and row of A1 are different. Therefore, we can parenthesize A1 and A2 in one way only i.e., (A1 x A2).

Suppose dimension of three matrices are :

$A_1 = 5 \times 4$
$A_2 = 4 \times 6$
$A_3 = 6 \times 2$

1. In how many ways can we parenthesize them?
2. How many multiplications are required to derive the resultant matrix?

   Formula to find out all valid combinations: $1/n \times {}^{2(n-1)}C_{n-1}$

   For n=3
   $1/3 \times {}^4C_2$
   $1/3 \times 4! \;/\; 2! * (4-2)!$
   $1/3 \times 4 \times 3 \times 2! \;/\; 2! * 2!$
   $1/3 \times 4 \times 3 \;/\; 2!$
   = 2 (We can parenthesize these three matrices only in two ways.)

A. A1 (A2 X A3) [First possible order of multiplication]
   (5 x 4) { (4 x 6) (6 x2) } [Here last two matrices require 48 multiplications]
   (5 x 4) (4 x 2) [Here two matrices require 40 multiplications]
   Total 88 multiplications are required.

B. (A1 X A2) A3 [Second possible order of multiplication]
   { (5 X 4) (4 X 6)} (6 X 2)
   (5 X 6) (6 X 2)
   Total 180 multiplications are required.

The answer of both multiplication sequences would be the same in the resultant matrix having 5 rows and 2 columns, but the numbers of multiplications are different. This leads to the question- what order should be selected for a chain of matrices to minimize the number of multiplications to reduce time complexity?

(I)

Consider the following four matrices. Find out optimal parenthesization of matrix chain multiplication.

$$\underset{5\times4}{A_1} * \underset{4\times6}{A_2} * \underset{6\times2}{A_3} * \underset{2\times7}{A_4}$$

$\underline{\text{Sol}^n}$:-

As $n = 4$, we can parenthesize these matrices in 5 different ways. We get it using the below formula:

$$\boxed{\frac{1}{n} \times {}^{2(n-1)}C_{n-1}} = \frac{1}{4} \times {}^{6}C_{3} = \frac{1}{4} \times \frac{6\times5\times4\times3!}{3!\times3!} = 5$$

All 5 solutions:

1. $\underset{5\times6}{(A_1 * A_2)} \cdot \underset{6\times7}{(A_3 * A_4)}$ ✓
2. $\underset{5\times4}{A_1} \; \underset{4\times2}{(A_2 * A_3)} \; \underset{2\times7}{A_4}$ ✓
3. $\underset{5\times2}{(A_1 * A_2 * A_3)} \cdot \underset{2\times7}{A_4}$ ✓
4. $\underset{5\times4}{A_1} \; \underset{4\times7}{(A_2 * A_3 * A_4)}$ ✓
5. $\underset{5\times4}{A_1} \left(\underbrace{\underset{4\times6}{A_2} \left(\underset{6\times7}{A_3 \times A_4} \right)}_{4\times7} \right)$ ✓

Out of these 5, we have to find out which one takes the least number of multiplications to desire the resultant matrix using dynamic programming.

Step 01: Draw two matrices having dimension 4×4.

matrix m

	1	2	3	4
1				
2				
3				
4				

matrix S

	1	2	3	4
1				
2				
3				
4				

Step 02: Order of matrix chain multiplication.

order $P < \underset{P_0}{5}, \underset{P_1}{4}, \underset{P_2}{6}, \underset{P_3}{2}, \underset{P_4}{7} >$

Step 03: Fill the left diagonal of the matrix "m" with zero.

← matrix "m"

	1	2	3	4
1	0	120	88	
2		0	48	104
3			0	84
4				0

matrix S

	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

Step 04: Fill right side of diagonal using the following two formulae:

For the "m" matrix.

$$m[i,j] = \min_{i \le k < j} \{ m[i,k] + m[k+1,j] + P_{i-1} P_k P_j \}$$

For the "s" matrix.

$$S[i,j] = K$$

$$m[1,2] = \min_{i \le k < j} \{ m[i,k] + m[k+1,j] + P_{i-1} P_k P_j \}$$

Here, $i=1, j=2$ so k will be "1"

$$= \{ m[1,1] + m[2,2] + P_0 P_1 P_2 \}$$

$$= \{ 0 + 0 + 5 \times 4 \times 6 \}$$

$$= 120$$

For the "s" matrix, value of $S[1,2]$ will be 1 as the value of K is equal to 1.

For $m[2,3]$, the value of K will be 2 as it should be <3 ~~or~~ equal to 2.

$k=2$

$$m[2,3] = \min \{ m[2,2] + m[3,3] + P_1 P_2 P_3 \}$$

$$= \{ 0 + 0 + 4 \times 6 \times 2 \}$$

$$= 48$$

For the "s" matrix:

$$S[2,3] = 2$$

For $m[3,4]$: $i=3, j=4$ & $k=$

$$m[3,4] = \min \{ m[3,3] + m[4,4] + P_2 P_3 P_4 \}$$

$$= \{ 0 + 0 + 6 \times 2 \times 7 \}$$

$$= 84$$

For the "s" matrix:

$$S[3,4] = 3$$

For $m[1,3]$, $i=1$, $j=3$ and $k = 1, 2$ because $\boxed{i \le k < j}$.

$$m[1,3] = \min_{k=1 \text{ and } 2} \begin{cases} m[1,1] + m[2,3] + P_0 P_1 P_3 \\ 0 + 48 + 5\times4\times2 = 88 \quad \text{min.} \\ \text{and} \\ m[1,2] + m[3,3] + P_0 P_2 P_3 \\ 120 + 0 + 5\times6\times2 = 180 \end{cases}$$

For the "s" matrix:

$S[1,3] = 1$ because at $k=1$, we get minimum value

For $m[2,4]$, $i=2$, $j=4$ and $k = 2$ and 3.

$$m[2,4] = \min_{i \le k < j} \begin{cases} \underset{0}{m[2,2]} + \underset{84}{m[3,4]} + \underset{4\ 6\ 7}{P_1 P_2 P_4} = 152 \\ \underset{48}{m[2,3]} + \underset{0}{m[4,4]} + \underset{4\ 2\ 7}{P_1 P_3 P_4} = 104 \checkmark \end{cases}$$

For the "s" matrix:

$S[2,4] = 3$ because at $k=3$, we get min value.

For $m[1,4]$, $i=1$, $j=4$ and $K = 1, 2,$ and 3.

$$m[\overset{i}{1},\overset{j}{4}] = \min_{i \le k < j} \begin{cases} \underset{0}{m[1,1]} + \underset{104}{m[2,4]} + \underset{5\ 4\ 7}{P_0 P_1 P_4} = 244 \\ \underset{120}{m[1,2]} + \underset{84}{m[3,4]} + \underset{5\ 6\ 7}{P_0 P_2 P_4} = 414 \\ \underset{88}{m[1,3]} + \underset{0}{m[4,4]} + \underset{5\times2\times7}{P_0 P_3 P_4} \\ 88 + 70 = 158 \end{cases} \text{min}$$

For the "s" matrix.

$S[1,4] = 3$ because at $K=3$, we get min value.

Step 05: Follow the matrix "s" to get optimal parenthesization.

$(A_1 * A_2 * A_3) * A_4$

$\boxed{((A_1) * (A_2 * A_3)) * A_4}$ → optimal soln.

V.V.I

Backtracking

By S.khan

(1)

Definition: It is one of the algorithm design technique. It uses a brute force approach for finding the desired solutions. The brute force approach tries out all the possible solutions and chooses the best/desired solutions. The term backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions.

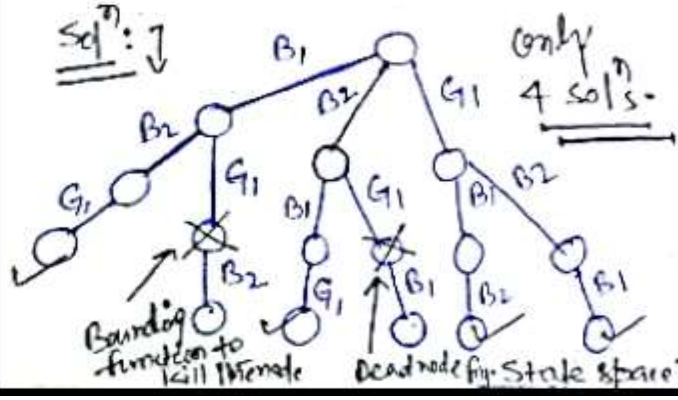
* Problems solved by backtracking approach have some constraints. If the solutions satisfy these constraints, then we consider them as solutions.

* Solutions are generated using "State space tree", which use DFS.

Example to understand Backtracking Approach

Q. There are three students (2 boys and 1 girl) and three chairs. We have to arrange these 3 students on the three chairs. There is only one constraint that the girl should not sit in the middle.

Sol^n:



B_1	B_2	G_1
B_2	B_1	G_1
G_1	B_1	B_2
× B_1	G_1	B_2
× B_2	G_1	B_1
G_1	B_2	B_1

$n! = 3! = 6$

= 6 solutions without satisfying the constraint.

4 sol^n Ans.

V.V.I

Sum of Subsets Problem (using Backtracking)

sum = 50 &

Consider the sum of subsets problem, $n = 4$, and $w_1 = 10$, $w_2 = 20$, $w_3 = 30$ and $w_4 = 40$. Find solutions to the problem using backtracking.

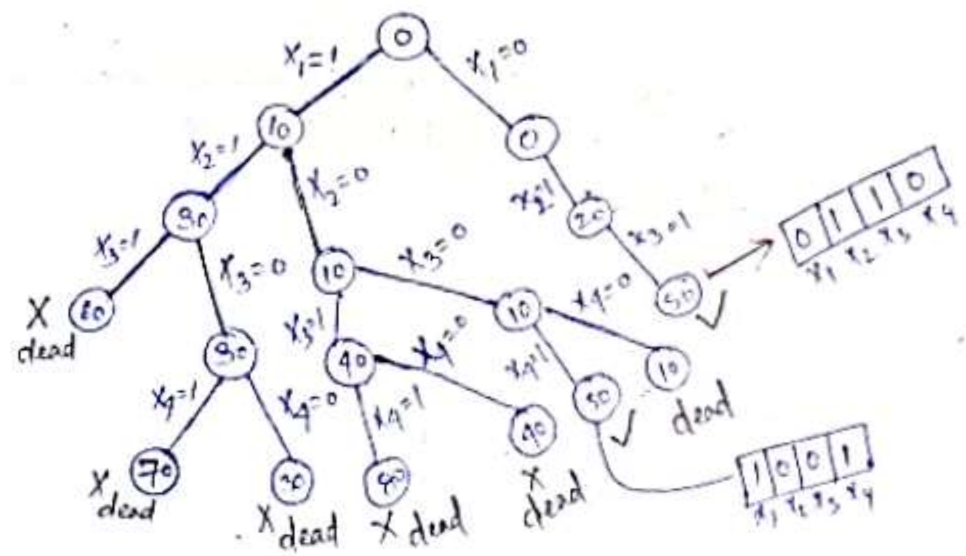
Solⁿ :

$\{10, 20, 30, 40\}$ → x_1 x_2 x_3 x_4

Sum = 50

Solⁿs: $\{20, 30\}$ & $\{10, 40\}$

State space Tree :



V.V.I

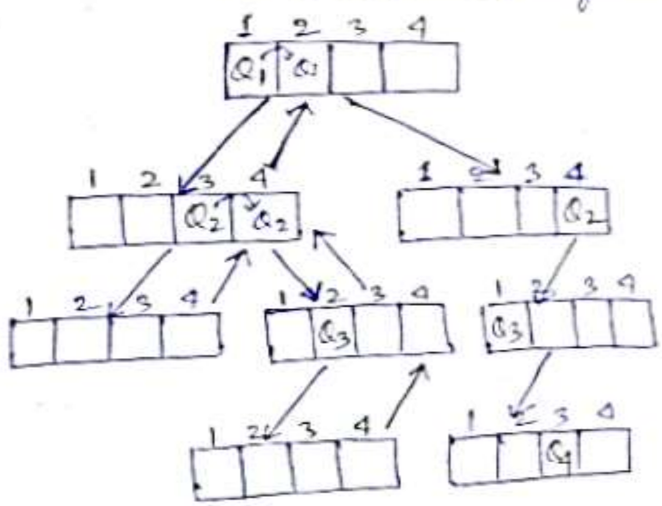
N- Queens problem
(Back-tracking)

(3)

N- Queens problem is to place n-queens in such a manner on an n×n chessboard that no queens attacks each other by being in the same row, column or diagonal.

Example :- 4 queens problem

Bounding function :- No two queens should be in the
(Condition)
same :
row, column or diagonal



$Q_1 = (1,2)$
$Q_2 = (2,4)$
$Q_3 = (3,1)$
$Q_4 = (4,3)$

(mirror image.)

Two solⁿ are possible.

18/11/2024

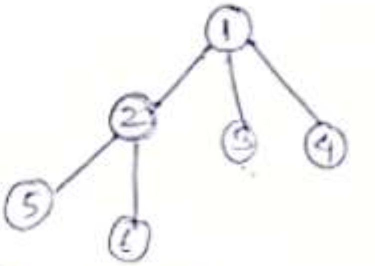
Branch and Bound

By SKhan (1)

It is an algorithm design technique used to solve optimization problems. Like Back tracking, it also generates a state space tree, but it uses BFS approach to generate the SST.

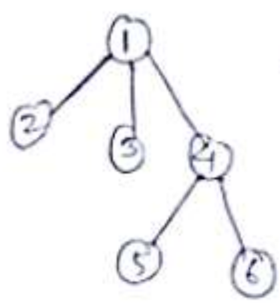
Classification of Branch and Bound Problems:

(1) FIFO Branch and Bound :



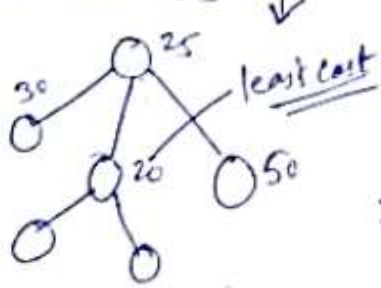
⇒ first in first out BB

(2) LIFO BB :



⇒ last-in-first-out BB

(3) Least cost BB :



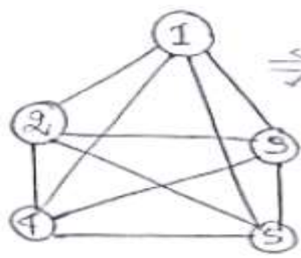
⇒ least cost BB

⇒ It uses priority queue.

Travelling Salesman Problem using Least Cost Branch and Bound

In this Problem, a salesman must visits n cities. We can say that salesman wishes to make a tour, visiting each city exactly once and finishing at the city he starts from. The goal is to find a tour of minimum cost. We assume that every two cities are connected. We can model the cities as a complete graph of n vertices, where each vertex represents a city.

Q Find the solution of following TSP using LC Branch and Bound.



Cost matrix

	1	2	3	4	5	min value for each row
1	∞	20	30	10	11	10
2	15	∞	16	4	2	2
3	3	5	∞	2	4	2
4	19	6	18	∞	3	3
5	16	4	7	16	∞	4
						21

Sol:

Step 01: Find reduced cost matrix by subtracting the min value in each row from each element in that row, this is called row reduction. We also do the column reduction.

Matrix after row reduction:

	1	2	3	4	5
1	∞	10	20	0	1
2	13	∞	14	2	0
3	1	3	∞	0	2
4	16	3	15	∞	0
5	12	0	3	12	∞
min value for each column	1	0	3	0	0 = 4

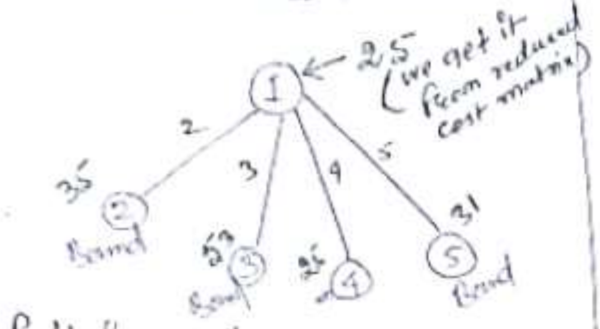
Matrix after column reduction:

	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

Reduced cost matrix

21+4=25

Let's start from node 1.



Path "1 to 2" at node 2:

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & \infty & \infty & 11 & 2 & 0 \\ 3 & 0 & \infty & \infty & 0 & 2 \\ 4 & 15 & \infty & 12 & \infty & 0 \\ 5 & 11 & \infty & 0 & 12 & \infty \end{array}$$

Make all entries for row 1 and col 2 ∞. Also for [2,1], make it ∞.

reduced cost = 0 → As all rows & cols have at least 0.

$$Cost(2) = 25 + 10 + 0 = 35$$

[1,2] check in the reduced cost matrix.

Path 1 to 3 at node 3:

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & 12 & \infty & \infty & 2 & 0 \\ 3 & \infty & 3 & \infty & 0 & 2 \\ 4 & 15 & 3 & \infty & \infty & 0 \\ 5 & 11 & 0 & \infty & 12 & \infty \end{array}$$

11 (reduction of column 1)

Make row 1 and col 3 ∞. And also for [3,1], make it ∞. Rest of entries will be same as in reduced cost matrix.

$$Cost(3) = 25 + 17 + 11 = 53$$

[1,3] → check in reduced cost matrix.

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & 1 & \infty & \infty & 2 & 0 \\ 3 & \infty & 3 & \infty & 0 & 2 \\ 4 & 4 & 3 & \infty & \infty & 0 \\ 5 & 0 & 0 & \infty & 12 & \infty \end{array}$$

Path 1 to 4 at node 4:

Make 1st row and 4th col as ∞ and also [4,1] as ∞. Rest entries are done from reduced cost matrix only.

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & 12 & \infty & 11 & \infty & 0 \\ 3 & 0 & 3 & \infty & \infty & 2 \\ 4 & \infty & 3 & 12 & \infty & 0 \\ 5 & 11 & 0 & 0 & \infty & \infty \end{array}$$

reduced cost = 0

$$Cost(4) : 25 + 0 + 0 = 25$$

[1,4]

Path 1 to 5 at node 5:

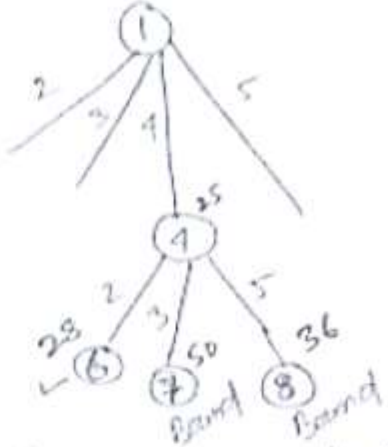
$$\begin{array}{c|ccccc|c} & 1 & 2 & 3 & 4 & 5 & \\ \hline 1 & \infty & \infty & \infty & \infty & \infty & \\ 2 & 12 & \infty & 11 & 2 & \infty & 2 \\ 3 & 0 & 3 & \infty & 0 & \infty & \\ 4 & 15 & 3 & 12 & \infty & \infty & 3 \\ 5 & \infty & 0 & 0 & 12 & \infty & \end{array}$$

5

$$Cost(5) : 25 + 1 + 5 = 31$$

[1,5]

As the cost of node 4 is the least, we consider it only to expand.



Path 1, 4 to 2 at node 6.

Consider the matrix 1 to 4. make row_1, col_4, row_4 and col_2 ∞ and also [2,1] as ∞.

$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & \infty & \infty & 11 & \infty & 0 \\ 3 & 0 & \infty & \infty & \infty & 2 \\ 4 & \infty & \infty & \infty & \infty & \infty \\ 5 & 11 & \infty & 0 & \infty & \infty \end{array}$$

reduced cost of this matrix is zero

Cost(6) : $\underset{\uparrow \text{ cost of } [4]}{25} + \underset{\uparrow [4,2]}{3} + 0 = 28$

Path 1, 4, 3 at node 7.

make ∞ to :
row_1, col_4
row_4, col_3
row_3, col_1 ([3,1])

Consider 1 to 4 matrix. (2)

$$\begin{array}{c|ccccc|c} & 1 & 2 & 3 & 4 & 5 & \\ \hline 1 & \infty & \infty & \infty & \infty & \infty & \\ 2 & 12 & \infty & \infty & \infty & 0 & \\ 3 & \infty & 3 & \infty & \infty & 2 & 2 \\ 4 & \infty & \infty & \infty & \infty & \infty & \\ 5 & 11 & 0 & \infty & \infty & \infty & \\ \hline & 11 & & & & & \end{array}$$

reduced cost = 11 + 2 = 13

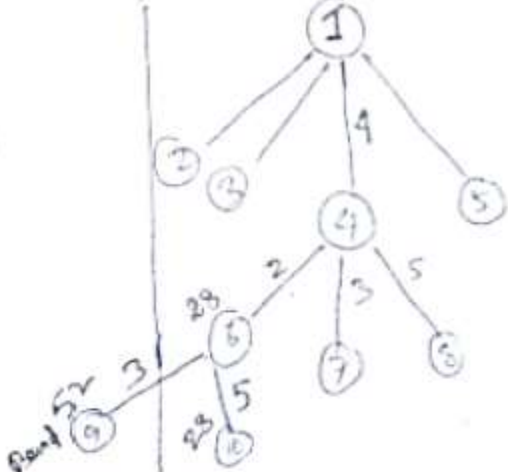
Cost(7) : $25 + \underset{\uparrow [4,3]}{12} + 13 = 50$

Path 1, 4, 5 at node 8

$$\begin{array}{c|ccccc|c} & 1 & 2 & 3 & 4 & 5 & \\ \hline 1 & \infty & \infty & \infty & \infty & \infty & \\ 2 & 12 & \infty & 11 & \infty & \infty & 11 \\ 3 & 0 & 3 & \infty & \infty & \infty & \\ 4 & \infty & \infty & \infty & \infty & \infty & \\ 5 & \infty & 0 & 0 & \infty & \infty & \end{array}$$

reduced cost = 11 ✓

Cost(8) : $25 + \underset{\uparrow [4,5]}{0} + 11 = 36$



(9)

Path 1, 4, 2, 3 at node 9.

Consider matrix having paths from 1, 4, to 2 at node 6 as this matrix produces cost 28 for the node 6.

	1	2	3	4	5	
1	∞	∞	∞	∞	∞	
2	∞	∞	∞	∞	∞	
3	∞	∞	∞	∞	2	2
4	∞	∞	∞	∞	∞	
5	11	∞	∞	∞	∞	11

reduced cost = 13

$Cost(9) : 28 + 11 + 13 = 52$

[2,3]

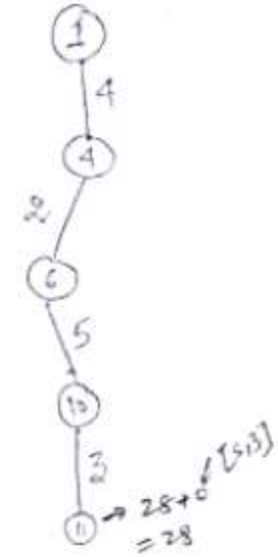
Path 1, 4, 2, 5 at node 10.

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	0	∞	∞	∞	∞
4	∞	∞	∞	∞	∞
5	∞	∞	0	∞	∞

reduced cost = 0

$Cost(10) : 28 + 0 + 0 = 28$

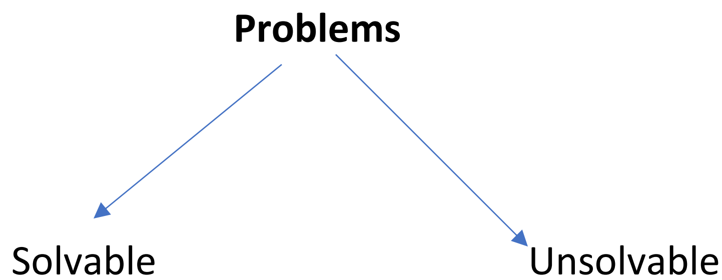
[2,5]



Paths → 1 → 4 → 2 → 5 → 3

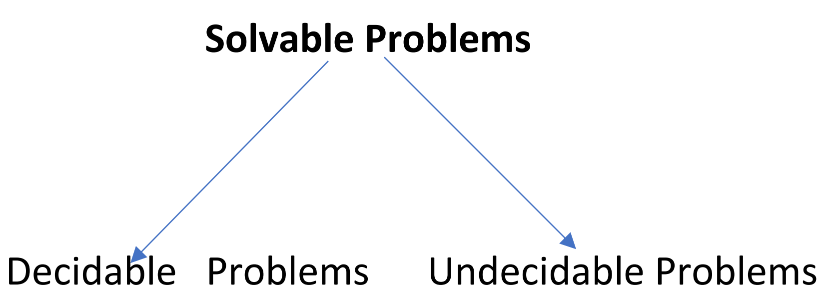
Soln

Unit-05



Solvable problems - A problem is said to be solvable if we know either there exists a solution or we are able to prove mathematically that the problem cannot be solved.

Unsolvable problems – A problem is said to be unsolvable if we know neither there exists a solution nor we are able to prove mathematically that the problem cannot be solved. It means that in the future we will have all problems currently in unsolvable domain in solvable domain for sure. For example, time complexity of Shell sort.



Decidable Problems – A problem is said to be decidable if we are able to predict the time to solve the problem. It means that we have an algorithm as well as procedure to solve the problem. For example, sorting problem.

Undecidable Problems – A problem is said to be undecidable if we are not able to predict the time to solve the problem. It means that we have only procedure to solve the problem but not an algorithm- which is used to predict the time. For example, if I ask ," Is it possible to become the PM of India?" Answer is yes as we have a certain procedure to become the PM of India, but the time for this problem cannot be predicted.

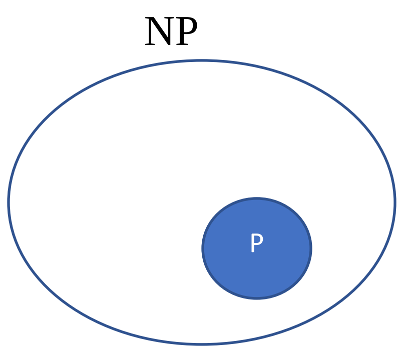
Q 1. Explain the complexity classes P, NP, NPC and NP hard. How are they related to each other?

P class – P stands for polynomial. It is a set of problems which can be solved as well as verified in polynomial time. Linear Search O(n) , Binary Search O(logn), Merge Sort O(nlogn), Heap Sort O(nlog), etc., are the examples of algorithms which solve the problem in polynomial time.

Note - Whatever algorithms we studied before dynamic programming belong to P class only.

NP class – NP stands for non-deterministic polynomial. It is a set of decision problems for which there exists a polynomial time verification algorithm. For example, for TSP , so far (we don't know about future), we have been unable to find out any polynomial time solution but then, given a solution of a TSP , we can verify it in polynomial time.

Note – If a problem belongs to P, then by default, it also belongs to NP because it can be verified in polynomial time, but vice versa does not hold good.



As of now, NP minus P (NP-N) problems have been unable to be solved in polynomial time. We don't know if these problems (TSP, 0/1 Knapsack, etc.) can be solved in polynomial time in the future or not.

NP hard class – If every problem in NP can be polynomial time reducible to a problem "A", then 'A' is called NP hard. If "A" could be solved in polynomial time, then by default, every problem in NP would become P.

NP complete class – A problem is said to be NP complete if it is NP as well as NP hard.

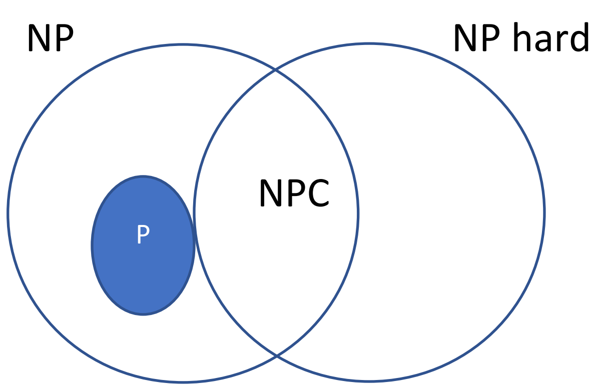


Fig - How P, NP, NP hard and NP complete are related to each other.

Q2. What are approximation algorithms? What is meant by P (n) approximation algorithms? Discuss approximation algorithm for Vertex cover problem.

An approximation algorithm is a way of dealing with NP-completeness for an optimization problem. This technique does not guarantee the best solution. The goal of the approximation algorithm is to come as close as possible to the optimal solution in polynomial time.

Some examples of the Approximation algorithm :

1. The Vertex Cover Problem
2. Travelling Salesman Problem
3. The Set Covering Problem
4. The Subset Sum Problem

If an algorithm reaches an approximation ratio of P (n), then we call it a P (n)-approximation algorithm.

C = Cost of solution
C*= Cost of optimal solution

- For a maximization problem, $0 < C < C^*$, and the ratio of C^*/C (approximation ration) gives the factor by which the cost of an optimal solution is larger than the cost of the approximate algorithm.
- For a minimization problem, $0 < C^* < C$, and the ratio of C/C^* gives the factor by which the cost of an approximate solution is larger than the cost of an optimal solution.

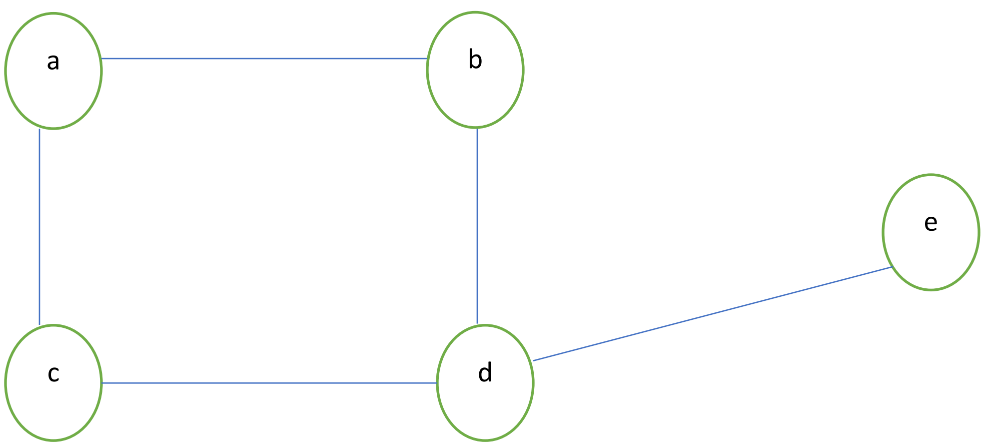
Vertex Cover Problem – Given an undirected graph, the vertex cover problem is to find minimum size vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. It is a minimization problem because we have to find a set of vertices containing minimum number of vertices covering all edges of the given undirected graph.

Approximation algorithm for vertex cover problem

```
APPROX-VERTEX-COVER(G)
1  C = ∅
2  E′ = G.E
3  while E′ ≠ ∅
4      let (u, v) be an arbitrary edge of E′
5      C = C ∪ {u, v}
6      remove from E′ every edge incident on either u or v
7  return C
```

[Important for your university exam]

Example –



Solution –

Line 1. C = solution set, which is empty in the beginning.

Line 2. E' = { (a, b), (a, c), (c, d), (b, d), (d, e)} // set of edges

Line 3. While E' != empty

Line 4. Add an arbitrary edge from E' in the solution set on line 5.

Suppose we consider (a, b) from E', then

Line 5. C = C U { a, b }

C = { a, b } // As of now solution set contains two vertices

Line 6. Remove every edge incident on either a or b vertex. Therefore,

Remove the following edges from E ' :

(a, b), (b, c), and (a, c) [These three edges are incident on a and b.]

Now, E ' = { (c, d), (b, d), (d, e) }

As E ' is not empty, add another arbitrary edge from E ' in the solution set C. Let's take the edge (c, d) now.

C = { a, b, c, d}

Using line number 6, remove every edge incident on either vertex c and d. Therefore, remove (c, d), (b, d) and (d, e) from E '. E ' is empty now, so return C using line number 7, which contains four vertices a, b, c, and d.

C is a set of minimum number of vertices, which covers all edges.

Randomized algorithm – Algorithms using random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm. For example, in Randomized Quick Sort, we use a random number to pick the next pivot (or we randomly shuffle the array). Typically, this randomness is used to reduce time complexity or space complexity in other standard algorithms.

String Matching Algorithms

String matching algorithms, sometimes called string searching algorithms, are an important class of string algorithms that try to find a place where one or several strings (also called pattern) are found within a large string or text. For example,

If text array = T [1 . . . n] and pattern array = P[1 . . . m], then we have to find out P in T. Therefore, length of P must be less than or equal to T. Both the pattern and searched text belong to ∑ (set of alphabets) , and it can contain either English alphabet (finite set) or binary number (0 and 1).

We have the following algorithms to search a patter in the given text:

1. Naive String-Matching Algorithm.
2. Rabin-Karp String Matching Algorithm
3. Finite Automata String Matching Algorithm
4. Knuth-Morris-Pratt (KMP) String Matching Algorithm

Naive String-Matching Algorithm

The naive approach tests all the possible placement of Pattern P [1.......m] relative to text T [1......n]. We try shift s = 0, 1.......n-m successively and for each shift s. Compare T [s+1.......s + m] to P [1......m].

Q. Reframe an algorithm for naive string matcher?

Algorithm

NAIVE-STRING-MATCHER (T, P)

```
1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n -m
4. do if P [1...m] = T [s + 1....s + m]
5. then print "Pattern occurs with shift" s
```

Find an example on the next page.

Lecture 41

By S. khan ①

Naive string Matching

Example.

Text → $T = \{ babababbabaa \} = 10$ (length)

pattern → $P = \{ babba \} = 5$

Sol^n:

S (shift) = T − P = 10 − 5 = 5

we can shift til 5.

Step 1:

T	b	a	b	a	b	b	a	b	a	a
	↓	↓	↓	X						
s=0, p→	b	a	b	b	a					

Step 2:

T→	b	a	b	a	b	b	a	b	a	a
s=1		↓	X							
P→		b	a	b	b	a				

Step 3:

	b	a	b	a	b	b	a	b	a	a
S=2			↓	↓	↓	↓	↓			
P →			b	a	b	b	a			

Ans: s = 2

At s = 2, we have got a match.

Step 4:

T	b	a	b	a	b	b	a	b	a	a
S = 3				X↓						
P →				b	a	b	b	a		

Step 5: Now, S = 4.

	b	a	b	a	b	b	a	b	a	a
					↓	X↓				
					b	a	b	b	a	

Step 6: Now last valid iteration s = 5

	b	a	b	a	b	b	a	b	a	a
						↓	↓	↓	X	
						b	a	b	b	a

Rabin-Karp String Matching Algorithm

The Rabin–Karp algorithm is a string-searching algorithm created by Richard M. Karp and Michael Rabin (1987) that uses hashing to find an exact match of a pattern string in a text. They suggest the hash function by choosing a random prime number q and calculate p[1 ….. m] mod q.

Algorithm

```
RABIN-KARP-MATCHER(T, P, d, q)
n = T.length
m = P.length
h = d^(m-1) mod q
p = 0
t_0 = 0
for i = 1 to m                  // preprocessing
    p = (dp + P[i]) mod q
    t_0 = (d t_0 + T[i]) mod q
for s = 0 to n - m              // matching
    if p == t_s
        if P[1..m] == T[s + 1..s + m]
            print "Pattern occurs with shift" s
    if s < n - m
        t_(s+1) = (d(t_s - T[s + 1]h) + T[s + m + 1]) mod q
```

Q 3. For q = 11, how many valid and spurious hits are found for the given Text and Pattern:

T = 3141592653589793

P = 26

Solution –

Step -1 Find p mod q

26 % 11 = 4

Step -2 As "p" contains a 2 digits number, find mod 11 of each 2 digits number from T as follows:

T = 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3

31 mod 11 = 9
14 mod 11 = 3
41 mod 11 = 8
15 mod 11 = 4
59 mod 11 = 4
92 mod 11= 4
26 mod 11 = 4
65 mod 11 = 10
53 mod 11 = 9
35 mod 11 = 2
58 mod 11 = 3
89 mod 11 = 1
97 mod 11 = 9
79 mod 11 = 2
93 mod 11 = 5

We consider only the number which gives us 4 after performing mod 11. Therefore, we have :

15 mod 11 = 4
59 mod 11 = 4
92 mod 11= 4
26 mod 11 = 4[Here, 26 is equal to P (pattern), so it is a valid hit]

Three spurious hits in yellow and one valid hit in red.

Finite Automata Based String Matching Algorithm

For a given pattern P, we construct a string-matching automaton in a preprocessing step before using it to search the text string.

Algorithm

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
n = T.length
q = 0
for i = 1 to n
    q = δ(q, T[i])
    if q == m
        print "Pattern occurs with shift" i − m
```

For the pattern p = abcd

Prefixes of P = a, ab, abc, abcd [started from the left side]

Suffixes of P = d, cd, bcd, abcd [started from the right side]

In order to specify the string-matching automaton corresponding to a given pattern p[1 . . . m], we first define an auxiliary function σ , called suffix function. σ(x) is the length of the longest prefix of p that is also a suffix of x. For example,

For the pattern p = abab and x = aba

σ(x) = ?

"a" is a suffix of x as well as a prefix of p

"aba" is a suffix of x as well as prefix of p.

Length of "a" = 1

Length of "aba" = 3

Therefore, σ(x) = 3

Example – T = {abababacaba} and p ={ababaca}

Solution – Pattern length (m) = 7

Number of states = m+1 = 8

Q = { q_0, . . ., q_7 }

∑ = { a, b, c }

Prefixes of P ={ a, ab, aba, abab, ababa, ababac, ababaca}

Now, we can create a transition table for P using suffix function σ.

Transition function δ : Q x ∑ → Q

δ(q_0 , a) = σ(a) = 1 (since "a" is a prefix in P and a's length is 1)

δ(q_0 , b) = σ(b) = 0 (No transition as "b" is not a prefix in P)

δ(q_0 , c) = σ(c) = 0 (No transition as "c" is not a prefix in P}

δ(q_1 , a) = σ(aa) = 1 (only single "a" is a prefix in P)

δ(q_1 , b) = σ(ab) = 2 ("ab" is a prefix in P and its length is 2)

δ(q_1 , c) = σ(ac) = 0 (No transition as "ac" is not a prefix in P)

δ(q_2 , a) = σ(aba) = 3 ("aba" is a prefix in P and its length is 3)

δ(q_2 , b) = σ(abb) = 0 (None of its suffixes (b, bb and abb) is in P)

δ(q_2 , c) = σ(abc) = 0 (None of its suffixes (c, bc and abc) is in p)

$\delta(q_3, a) = \sigma(abaa) = 1$ (Among suffixes (a,aa,baa,abaa) only "a" is a prefix in P and its length is 1)
$\delta(q_3, b) = \sigma(abab) = 4$ ("abab" is a prefix in p and its length is 4)
$\delta(q_3, c) = \sigma(abac) = 0$(among all suffixes (c, ac, bac, abac), none is there in p; therefore, no transition)

$\delta(q_4, a) = \sigma(ababa) = 5$ ("ababa" is a prefix in p and its length is 5)
$\delta(q_4, b) = \sigma(ababb) = 0$ (among all suffixes , none is there in p)
$\delta(q_4, c) = \sigma(ababc) = 0$ (among all suffixes , none is there in p)

$\delta(q_5, a) = \sigma(ababaa) = 1$(among all suffixes (a, aa, baa, abaa, babaa, ababaa) only "a" is prefix in p and its length is 1)
$\delta(q_5, b) = \sigma(ababab) = 4$ (among all suffixes (b, ab, bab, abab, babab, ababab) the longest prefix "abab" is in p and its length is 4)
$\delta(q_5, c) = \sigma(ababac) = 6$ ("ababac" is a prefix in p and its length is 6)

$\delta(q_6, a) = \sigma(ababaca) = 7$ ("ababaca" is a prefix in p and its length is 7)
$\delta(q_6, b) = \sigma(ababacb) = 0$
$\delta(q_6, c) = \sigma(ababacc) = 0$

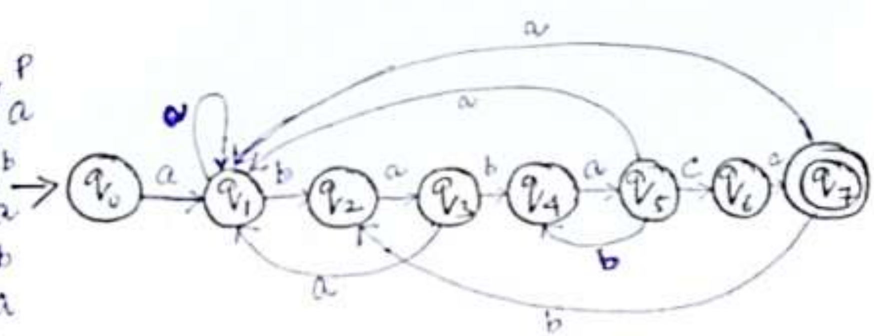
$\delta(q_7, a) = \sigma(ababacaa) = 1$(among all suffixes(a, aa, caa,….) only "a" is in p)
$\delta(q_7, b) = \sigma(ababacab) =2$ (among all suffixes (b, ab, cab, …) only ab is in p)
$\delta(q_7, c) = \sigma(ababacac) = 0$ (None of the prefixes is there in p)

I/p

	a	b	c	P
→ q_0	1	0	0	a
q_1	1	2	0	b
q_2	3	0	0	a
q_3	1	4	0	b
q_4	5	0	0	a
q_5	1	4	6	c
q_6	7	0	0	a
* q_7	1	2	0	

→ It does not give us any new state

(a) Transition Table



(b) State transition diagram

$m = 7$

Index

i	1	2	3	4	5	6	7	8	9	10	11
Text — T[i]	a	b	a	b	a	b	a	c	a	b	a
(q) state 0	1	2	3	4	5	4	5	6	$\boxed{7}$	2	3

If $q == m$

Pattern occurs with shift $i - m = 11 - 7 = 2$

Knuth-Morris-Pratt (KMP) String Matching Algorithm

KMP is a linear time string matching algorithm. It uses concept of prefix and suffix to generate Π table.

KMP-MATCHER (T, P)

```
n ← length [T]
m ← length [P]
Π← COMPUTE-PREFIX-FUNCTION (P)
q ← 0          // numbers of characters matched
for i ← 1 to n// scan S from left to right
do while q > 0 and P [q + 1] ≠ T [i]
do q ← Π [q]         // next character does not match
If P [q + 1] = T [i]
then q ← q + 1       // next character matches
If q = m                             // is all of p matched?
then print "Pattern occurs with shift" i - m
q ← Π [q]                       // look for the next match
```

COMPUTE- PREFIX- FUNCTION (P)

```
m ←length [P]        //'p' pattern to be matched
Π [1] ← 0
k ← 0
for q ← 2 to m
do while k > 0 and P [k + 1] ≠ P [q]
do k ← Π [k]
If P [k + 1] = P [q]
    then k← k + 1
Π [q] ← k
Return Π
```

Q 4. Compute the prefix function Π for the pattern ababbabbabbababbabb when the alphabet is ∑ = { a, b }.

Π – It is also called the longest prefix which is same as some suffix (LPS).

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
p	a	b	a	b	b	a	b	b	a	b	b	a	b	a	b	b	a	b	b
Π	0	0	1	2	0	1	2	0	1	2	0	1	2	3	4	5	6	7	8

Note – Use any short-cut trick to prepare LPS or Π table in the exam.

Fast Fourier Transform (FFT)

An FFT algorithm computes the discrete Fourier transform (DFT) of a sequence or its inverse DFT in time O(nlogn).

All the best 😊