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

This PDF provides classroom notes covering the syllabus of design and analysis of algorithms for AKTU, Lucknow, UP, India.

Unit-01

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

Properties of Algorithm

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

Steps required to design an algorithm

1. Problem definition
2. Design algorithm (choose from one of the available techniques)
3. Draw flow chart
4. Testing
5. Analyze the algorithm(Time complexity and space complexity)
6. Implementation (Coding)

The main algorithm design techniques

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

Brute Force technique

The simplest algorithm design technique is brute force technique. Almost all problems can be solved by brute force approach, although generally not with appreciable space and time complexity.

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

```
Void main(void){
        int arr[]={1, 2, 3, 4, 5},i;
        int search=4;
        char flag='0';
        for(i=0;i<5;i++)
            {
              If(arr[i]==search)
              flag=1;
              break;
             }
      if(flag==1)
          Printf("found");
      else
         Printf("not found");
  }
```

Analysis of Algorithms

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following –

- **A Priori Analysis** − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other

factors, for example, processor speed, are constant and have no effect on the implementation.

- **A Posteriori Analysis** − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

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

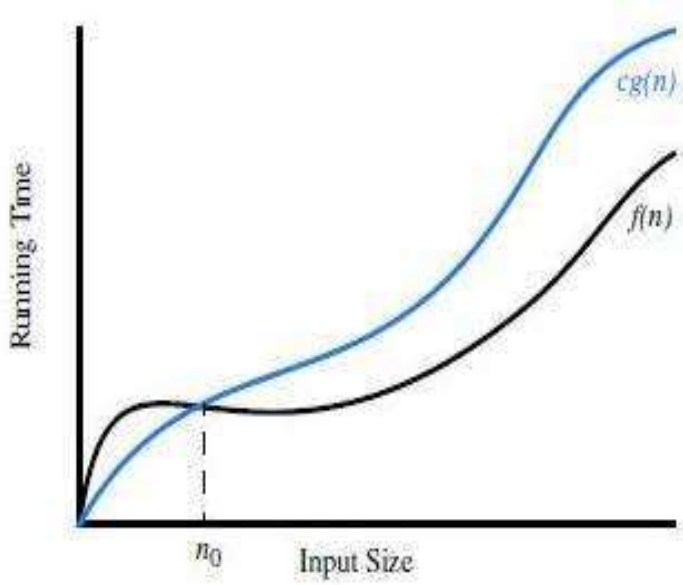
Asymptotic analysis of an algorithm refers to defining the mathematical boundation of its run-time performance. Using asymptotic analysis, we can conclude the best case, average case, and worst-case scenarios of an algorithm.

Asymptotic Notations:

1. **O** [Big-oh] (upper bound/ worst case scenario)
2. **Ω** [Big-omega] (lower bound / best case scenario)
3. **ɵ** [Big-theta] (tight bound / average case scenario)
4. **o** [small-oh] (Not tightly upper bound)
5. **w** [small omega] (Not tightly lower bound)

1. **Big-oh Notation (O)** : It denotes the upper bound of the runtime of an algorithm. Big O Notation's role is to calculate the longest time an algorithm can take for its execution, i.e., it is used for calculating the worst-case time complexity of an algorithm.

   Note: Most of the time, we are interested in finding only the worst-case scenario of an algorithm (worst case time complexity).



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

For example :

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.

$$2n^2 + 3n + 2 <= c. (n^2) \ldots\ldots\ldots(1)$$

If we put c = 7 then

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

Now, put n=1 which is n_0

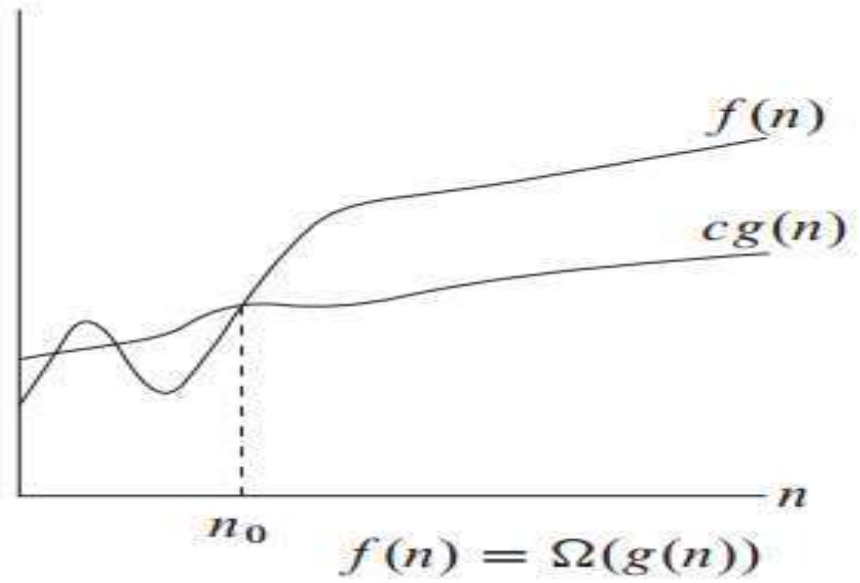
7 <= 7 [True]

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

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

2. **Omega Notation (Ω):** It represents the lower bound of the runtime of an algorithm. It is used for calculating the best time an algorithm can take to complete its execution, i.e., it is used for measuring the best-case time complexity of an algorithm.

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 :

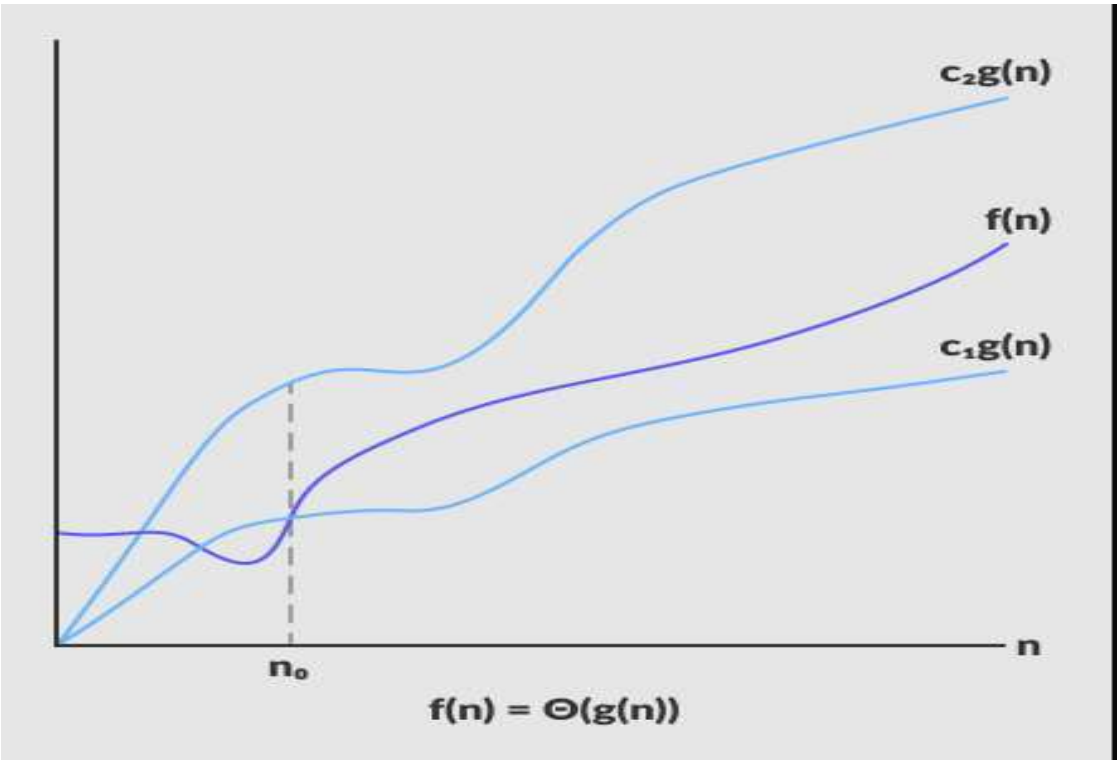
1. Given f(n) = 3n + 2 and g(n) = n, then prove that f(n) = Ω g(n)

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).

3. **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.**

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

$$\lim_{n \to \infty} f(n)/g(n) = 0$$

For example

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

Solution

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

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

$$\lim_{n\to\infty} 2/\infty = 0$$

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

$$\lim_{n\to\infty} f(n)/g(n) = \infty$$

For example

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

Solution

$$\lim n^2/2 /n$$

$n \to \infty$

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

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

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

Complexity of Algorithms

1. Time complexity
2. Space complexity

Algorithms can be broadly divided into two groups :

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

Time complexity

Time complexity of algorithms: The time complexity of an algorithm is not equal to the actual time required to execute a particular code but "the number of times a particular statement executes".

Have a close look at the below code to understand the above statement in green.

Note : " C1,C2, …, Cn" indicates constant time.

```
#include<stdio.h>

Void main(void)
{                                   Cost        times
   int i,  n = 20;                  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 time complexity in terms of n; (n) - linear time

Note : 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 operations.**
- **Memory is unlimited and access takes one-time step (one unit of time).**

Time complexity of iterative programs

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

Pattern-01 When there is no dependency between loops

```
A()
{
    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()
{
    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-02 When there is a dependency between loops (having more than one loop) or statements inside a loop and the loop

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

```
A()
{
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 line number 2 (while loop) and 4(the value of "j" inside "while loop" depends on "j" on line number-4).

i = 1, 2, 3, 4, … k

j = 1, 3, 6, 10 … k(k+1)/2 [the sum of 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.]

$k = \sqrt{n}$ Time complexity is $O(\sqrt{n})$

```
A()
{
    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 second and 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

$1*100 + 2*100 + 3*100 + \ldots + n*100$

$100(1+ 2+3 +\ldots n)$

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

Time complexity = $O(n^2)$ [We remove constant terms and smaller terms as well.]

```
A()
{
   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 K^{th} 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)

Time complexity of recursive programs

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

1. Iteration method (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/program must have a stopping condition (anchor condition/base condition) to stop recursive call.

```
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”- comparison and “print”- memory access take constant amount of time, and we can use either 1 or C to indicate it. When “if-condition” gets false, it takes again constant amount of time.

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=0; 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
}
```

Note: Left side of the * is the first operand, cannot be included in the equation.

$T(n) = T(n-1) + c$ when $n>1$

$= 1$ when $n <=1$

#Fibonacci number Fn

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

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

$$1 \text{ unit of time when } n <= 1$$

1. Iteration method (backward substitution) for solving recurrences

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

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

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

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

$T(n) = O(n)$

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

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 (base condition) , 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)

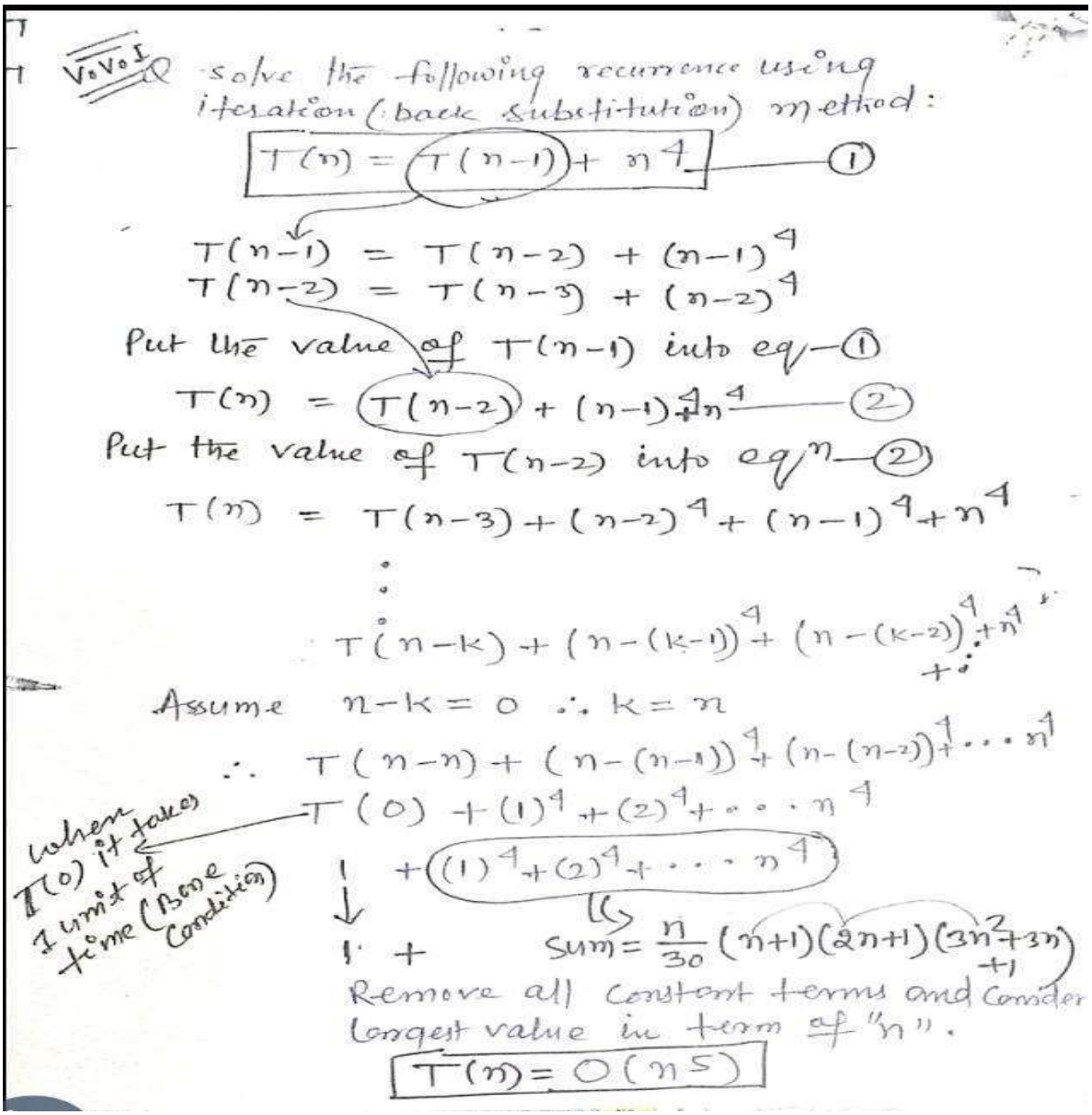
Assume that $(n/2^k) = 1$, then

$2^k = n$

$\log_2 n = k$

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

$= n + n\log n = O(n\log n)$



Recursion-tree method for solving recurrences

Type- 01 (Reducing function)

Steps:

1. Make T(n) as the root node
2. Draw the tree till two to three levels to calculate the cost and height
3. If the cost at each level is same, multiply it with the height of tree to get time complexity.
4. If the cost at each level is not same, try to find out a sequence . The sequence is generally in A.P or G.P.

Lecture – 06

By S. Khan

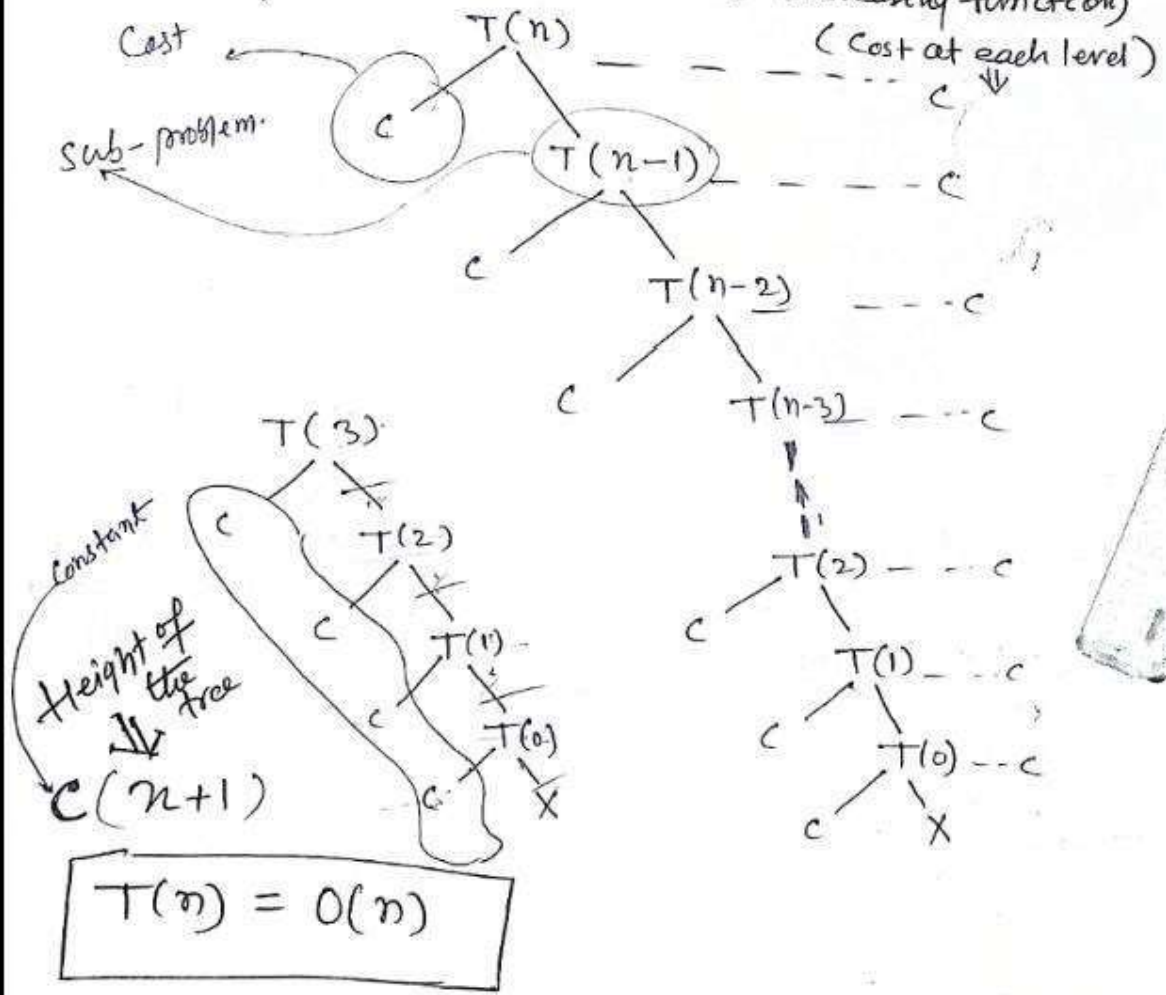
(2) Recursion-tree Method to solve "Recurrences".

(1)

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

- $T(n-1) + c$ → cost (time)
- $n = 0$ ⇐ stopping condition (Base cond.) → $T(0) = 1$ for recursive call.
- $T(n-1)$ → size of sub-problem (Decreasing function)

So, n :

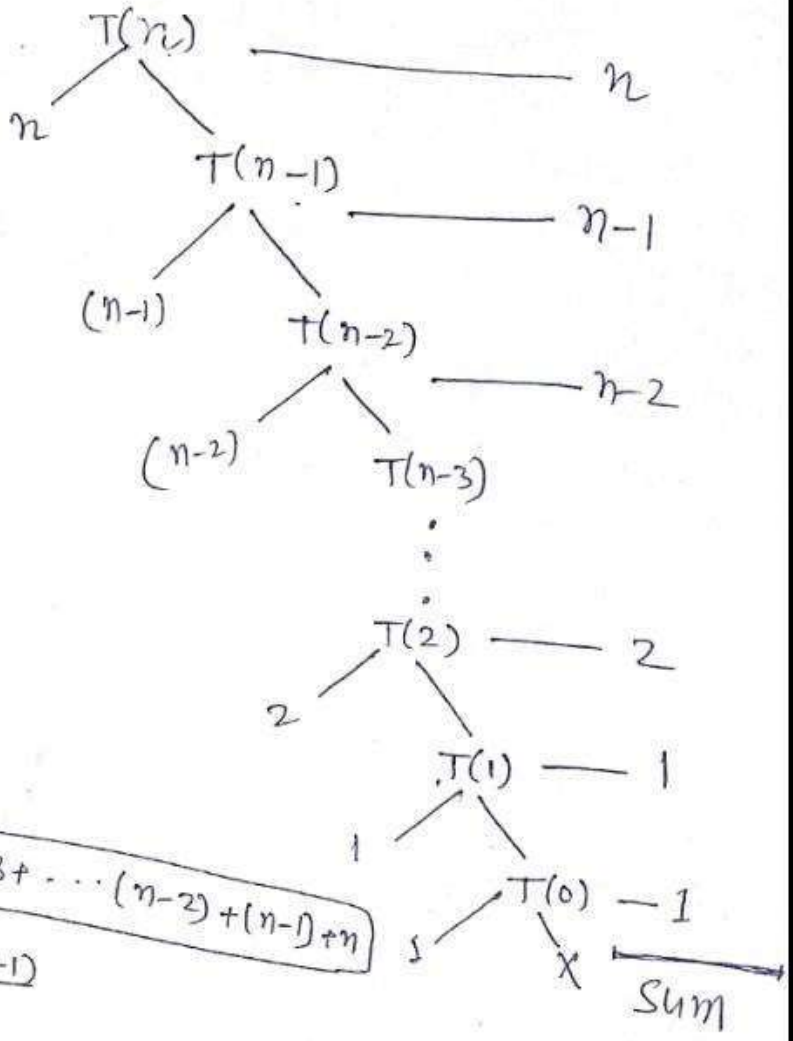


$$T(n) = O(n)$$

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 :↓



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

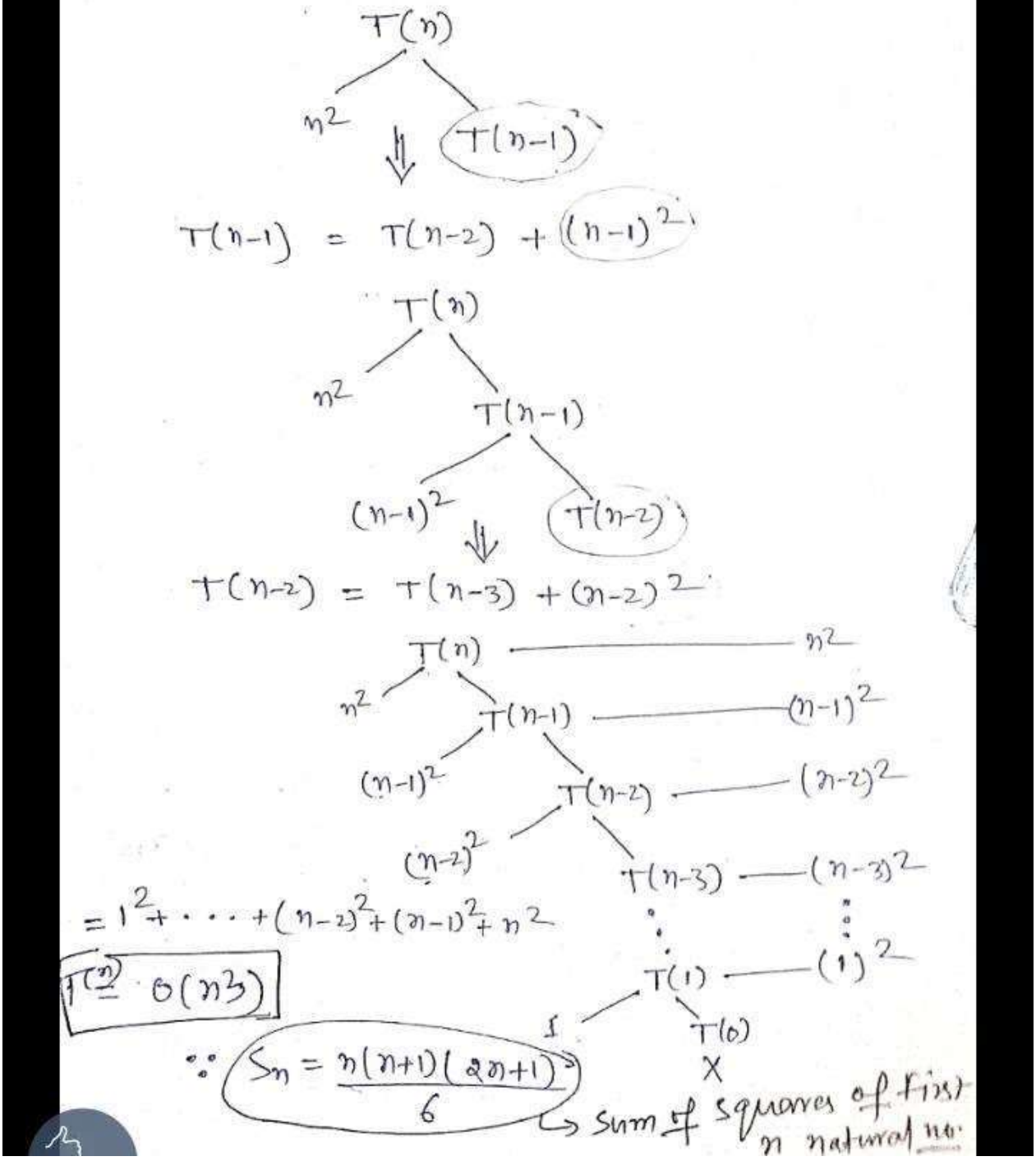
$= 1 + \frac{n(n+1)}{2}$

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

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

(3)

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



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

$T(n) = O(n^3)$

$\because S_n = \frac{n(n+1)(2n+1)}{6}$ → Sum of squares of first n natural no.

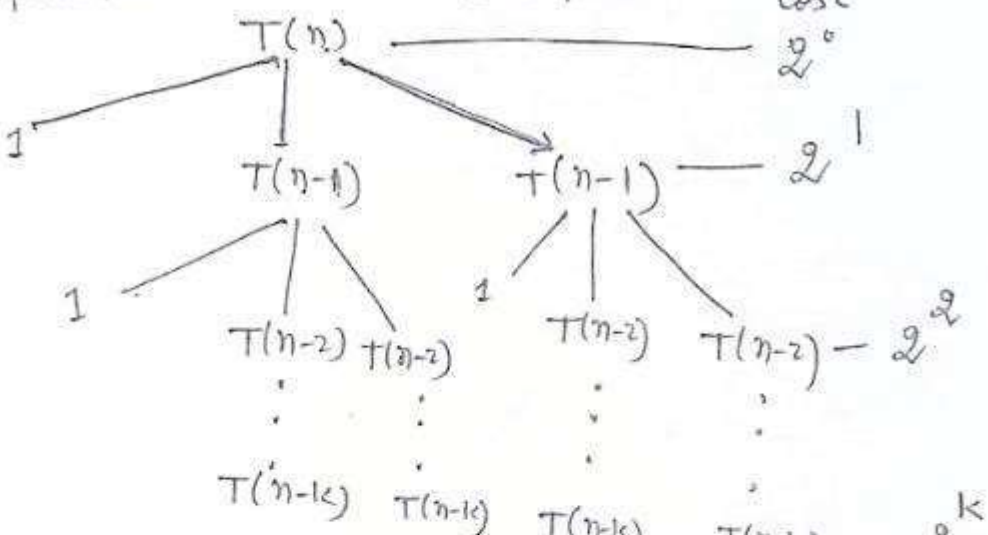
(4)

$T(n) = 2T(n-1) + 1 \; ; n>0$ and $T(0)=1$

cost = 1

number of sub-problems = 2

sub-problem = $n-1$



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

$\therefore k = n$

Terms = $k+1$, $r = 2$

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

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

$$= 2^{k+1} - 1$$

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

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

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

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

Type-2 (Dividing function when the size of sub-problems is more than one and same)

Steps:

1. Make the last term as the root node
2. Draw the tree till two to three levels to calculate the cost and height
3. If the cost at each level is same, multiply it with the height of tree to get time complexity.
4. If the cost at each level is not same, try to find out a sequence . The sequence is generally in A.P or G.P.

Lecture 07

By (1) S. Khan

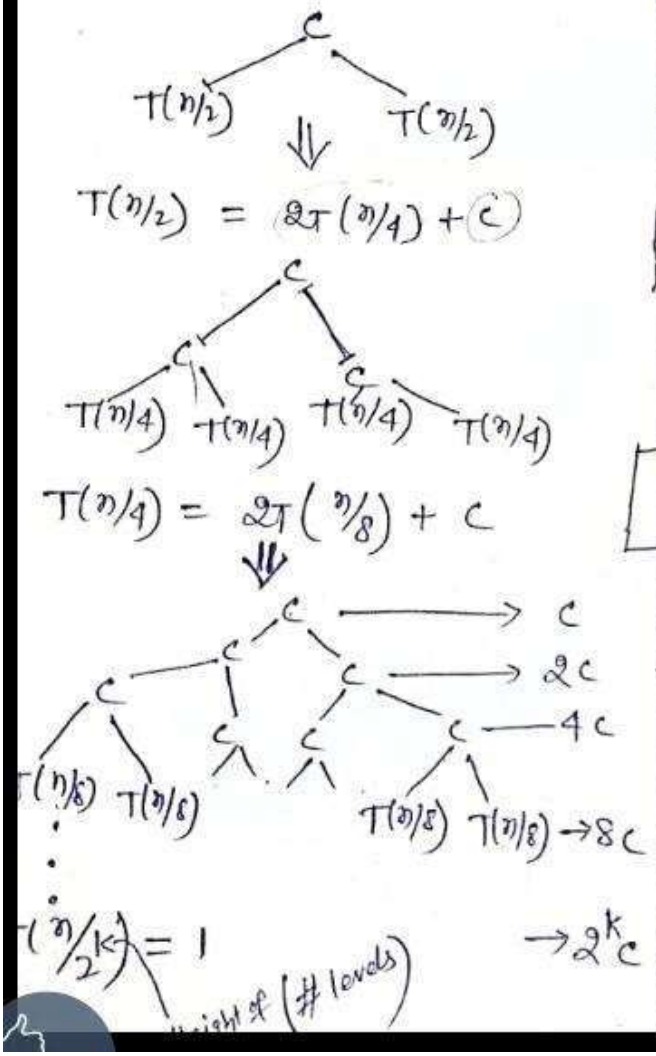
Type 02 → (Dividing functions)

Steps:

1. Make the first 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.

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

(number of sub problems, size of each sub problem)



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

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

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

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

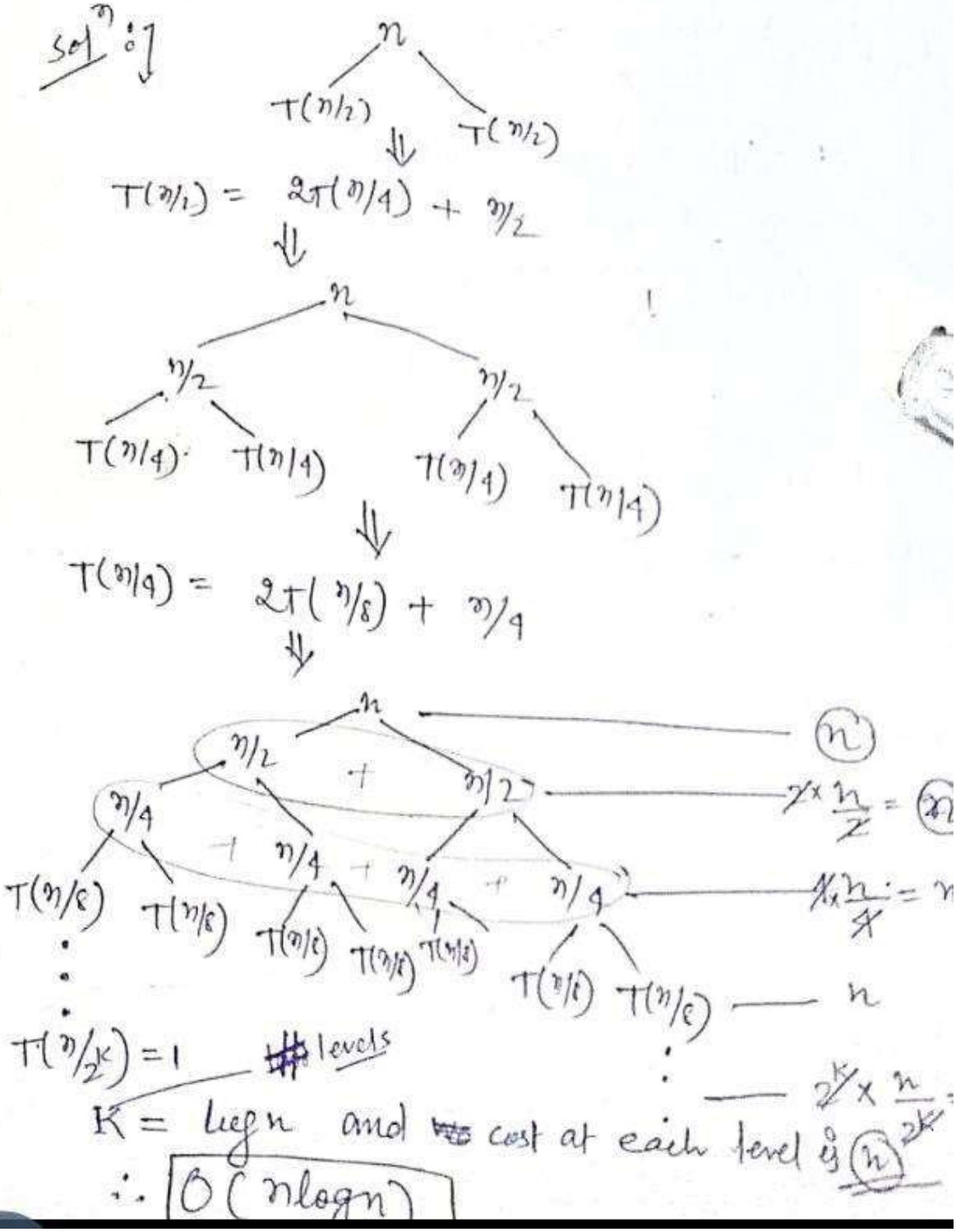
$c(2n - 1)$ $\left[\because 2^k = n \right.$

$\left. n \times 2^1 = 2n \right]$

$T(n) = O(n)$

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

Sol :]

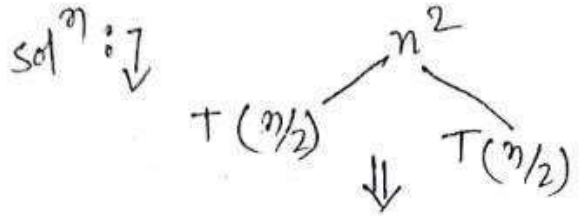


$K = \log n$ and cost at each level is n

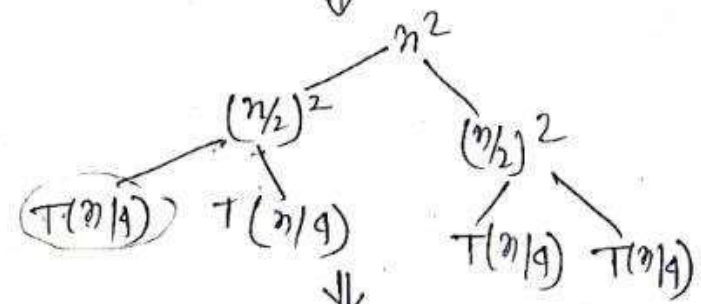
$\therefore$ $O(n \log n)$

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

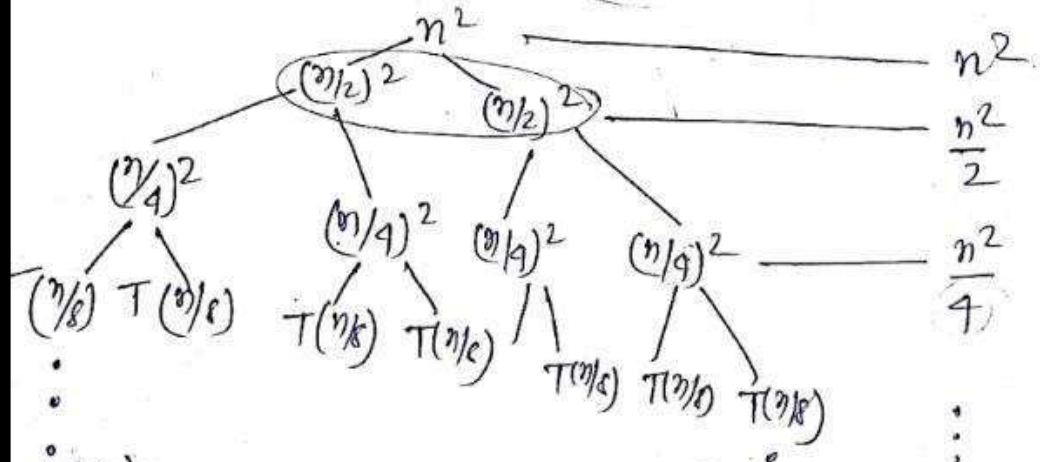
Sol^n :



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



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



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

$= n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \cdots$

$= n^2\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right)$

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

G.P $\Downarrow$ $S_\infty = \frac{1}{1-r}$ if $r < 1$

here $r = 1/2$

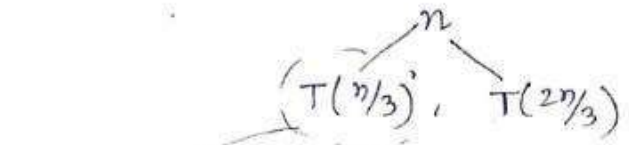
Type 03 :- (when the size of sub-problems is not same)

$$T(n) = T(n/3) + T(2n/3) + n \quad ; n>1 \text{ & } T(1)=1$$

eq^n (1)

↳ size of sub-problems is not equal.

Note: To get the height of tree, we consider the longest chain of edges.

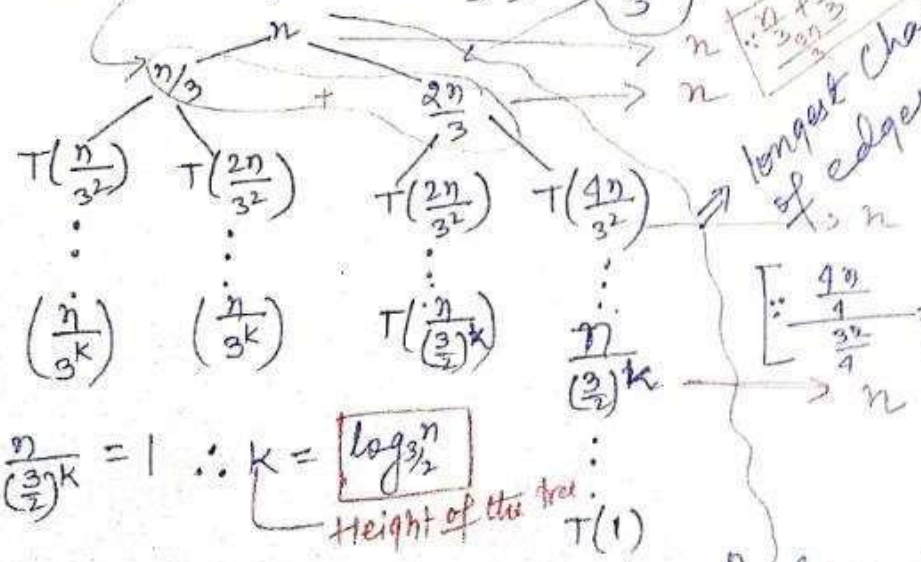


Now Put the value of $T(n/3)$ and $T(\frac{2n}{3})$ into eq^n (1) & get the following sub equations:

$$T(n/3) = T(\frac{n}{3^2}) + T(\frac{2n}{3^2}) + (\frac{n}{3})$$ → new left root

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

$\because \frac{n}{3} + \frac{2n}{3} = n$



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

Height of the tree

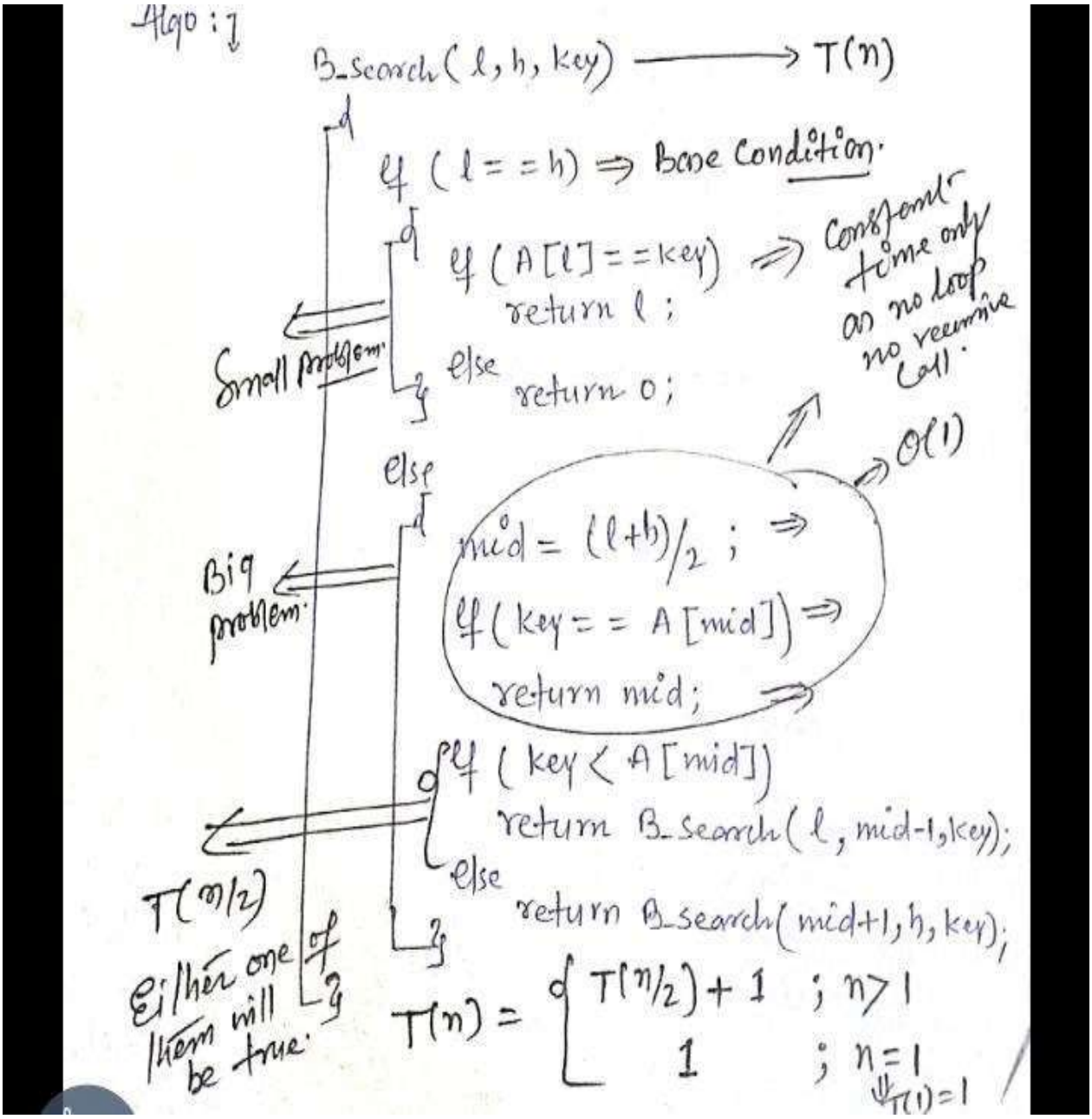
T(1)

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

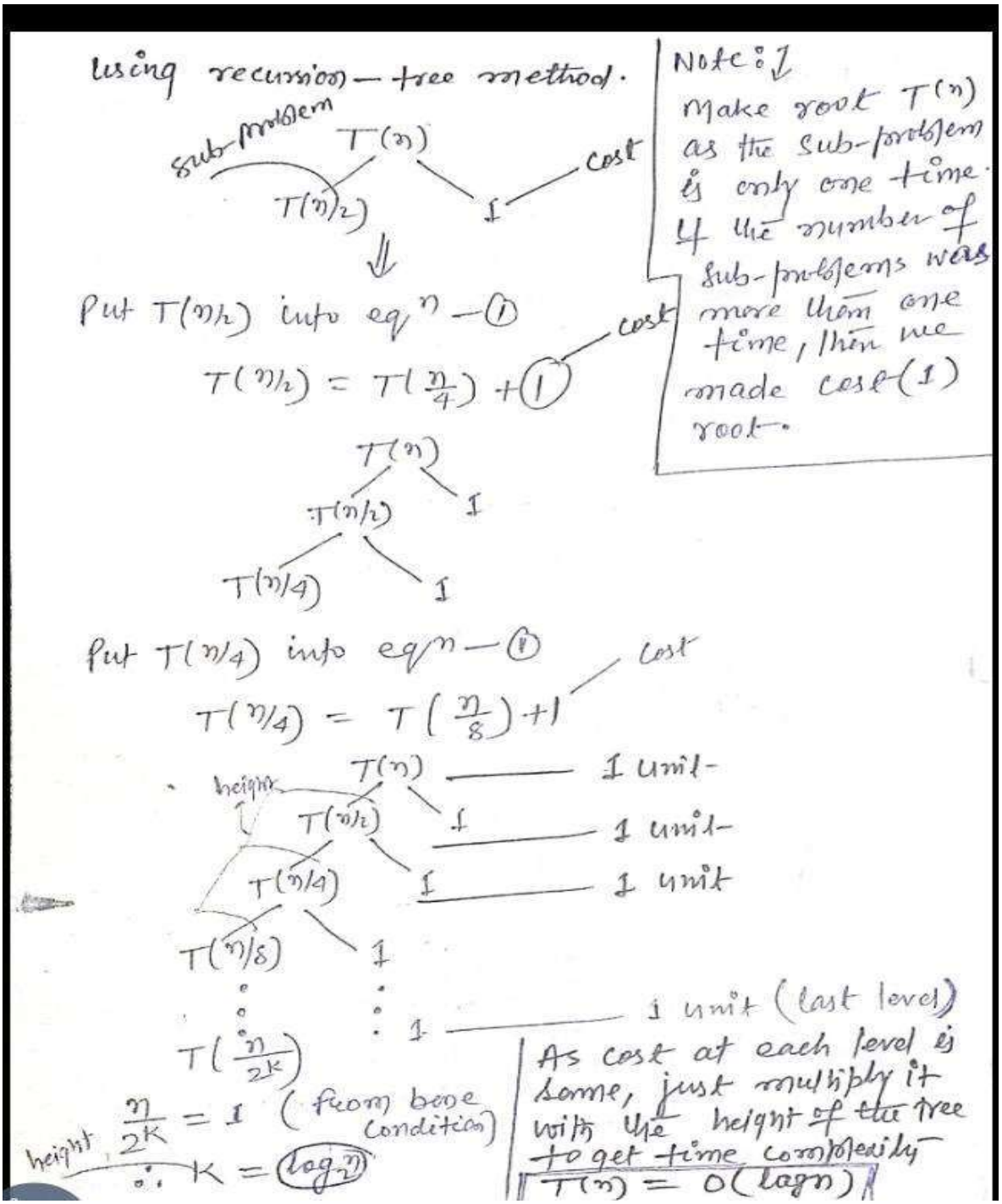
$$T(n) = O(n \log_{3/2} n)$$ Ans

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).



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



Q.No 1

Solve using recursive tree method. cost

$T(n) = 4T(n/2) + n$ and $T(1) = 1$

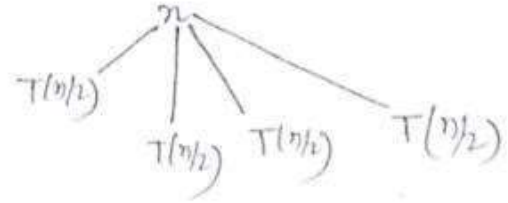
(1)

no of sub-problems

sub-problem

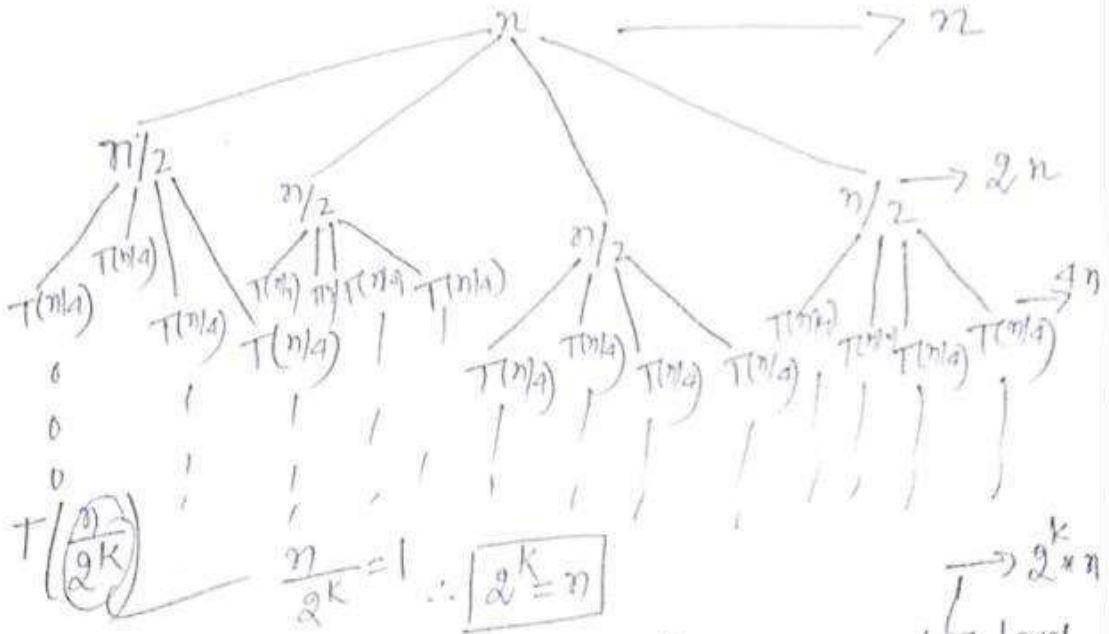
Sol^n :

at this place of n



Put $T(n/2)$ in equation → (1)

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



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

$\rightarrow 2^k \times n$

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

$r = 2$

$\because 2^k = n$

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

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

3 Master Method to solve recurrences

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 numb er.

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 $\boxed{T(n) = \Theta(n^k \log^p n)}$

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

Sol^n: 2

(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) ~~$\theta(n^{\log_b a})$~~

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

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

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

(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 $p = -1$

$a = b^k$

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

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

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

$\rightarrow p = -1$

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

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

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.

$T(n) = T(n/2, 3, 4, a) + \dots$

Lecture 09

By S. Khan (1)

(Master Theorem for Decreasing Functions)

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

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

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

Example

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

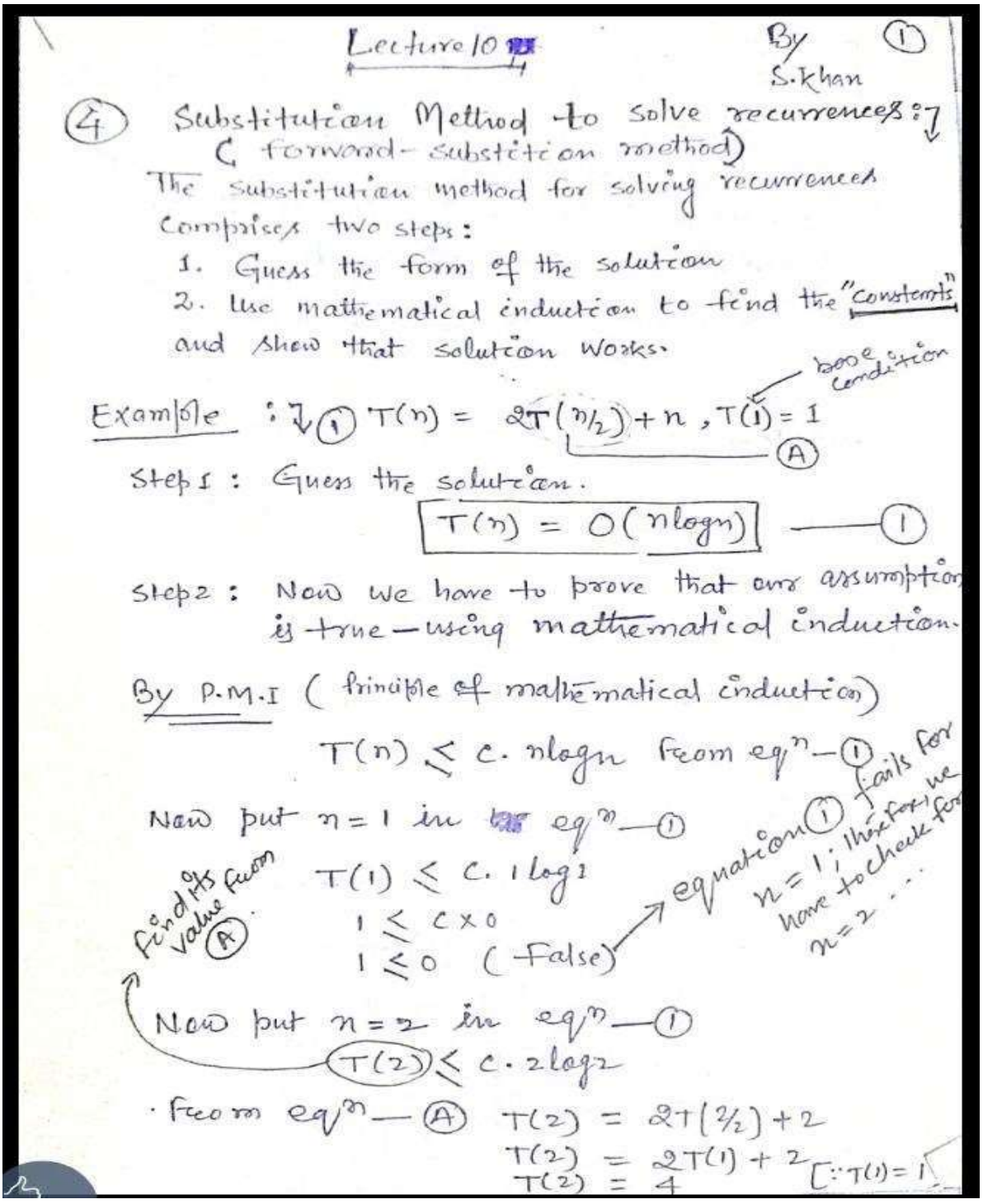
(6) $2T(n-1) + n$

$a = 2, b = 1$, 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-substition 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.

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

Step 1 : Guess the solution.

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

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

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

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

Now put $n = 1$ in eqn — (1)

Find its value from (A)

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

$1 \leq c \times 0$

$1 \leq 0$ (False) → equation (1) fails for $n = 1$; therefore we have to check for $n = 2$ ...

Now put $n = 2$ in eqn — (1)

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

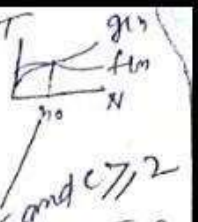
From eqn — (A) $T(2) = 2T(2/2) + 2$

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

$T(2) = 4$

$4 \leq 2c$ [$\because 2\log 2 = 2$]

→ 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, ..., k.

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

value of K

when we move from (2 to n) some where in the middle, we get $K = (n/2)$ (As we need to remove $T(n_1)$ from eq^n (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

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

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

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

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

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

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

$$T(n) \leq cn\log n$$ — we remove constant terms)

$$T(n) = O(n \log n)$$ — proved

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

Steps : Guess the solution.

$$T(n) = O(n^2) \longleftarrow (1)$$

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

By P.M.I :

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

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

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

$T(1) \leq c$

$T(1) = T(1-1) + 1$ from (A)

$= T(0) + 1$

$= 1 + 1$

$T(1) = 2$

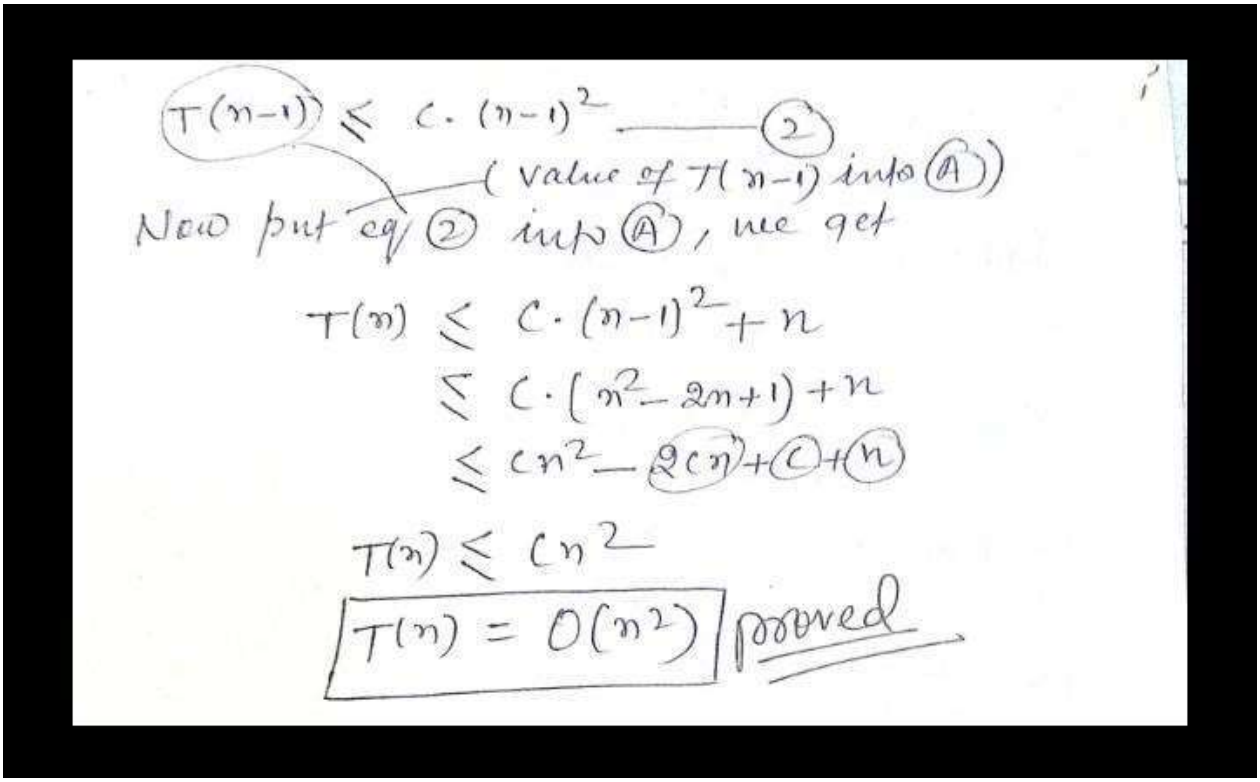
$2 \leq c$ True for $c \geq 2$

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

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

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

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)



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

Analysing Space Complexity of Iterative and recursive algorithms

Space Complexity = input size + 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:

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

→ only one extra variable.

Space complexity = O(1)

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

(1 + n)

⇓

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];
}
```

(n² + 2)

⇓

O(n²)

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)

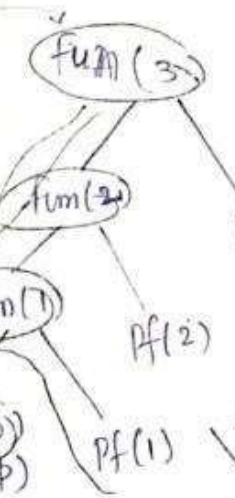
(i) Tree Method :-

↓

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);
  }
}
```



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

Maxm depth of the tree is (4). I/P = 3+1

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

remove constant

∴ 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 space taken by each recursive call to compute space complexity of recursive programs.

For example: ↓

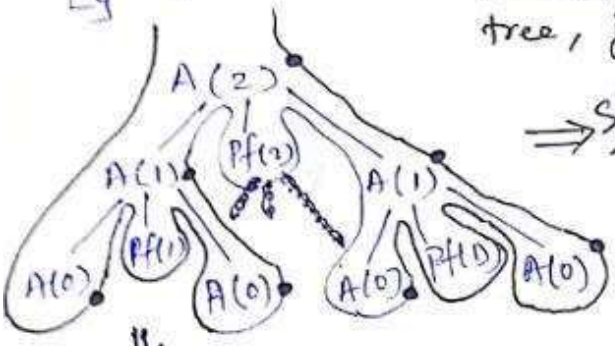
```
A(n)
{
  if(n>=1)
  {
    A(n-1);
    Pf(n);
    A(n-1);
  }
}
```

⇒

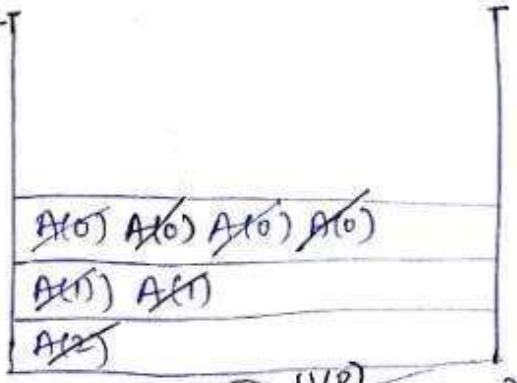
Steps:

1. Create a recursion - tr...
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.



⇒ Stack



max^m depth = 3 (input+1)

O(n)

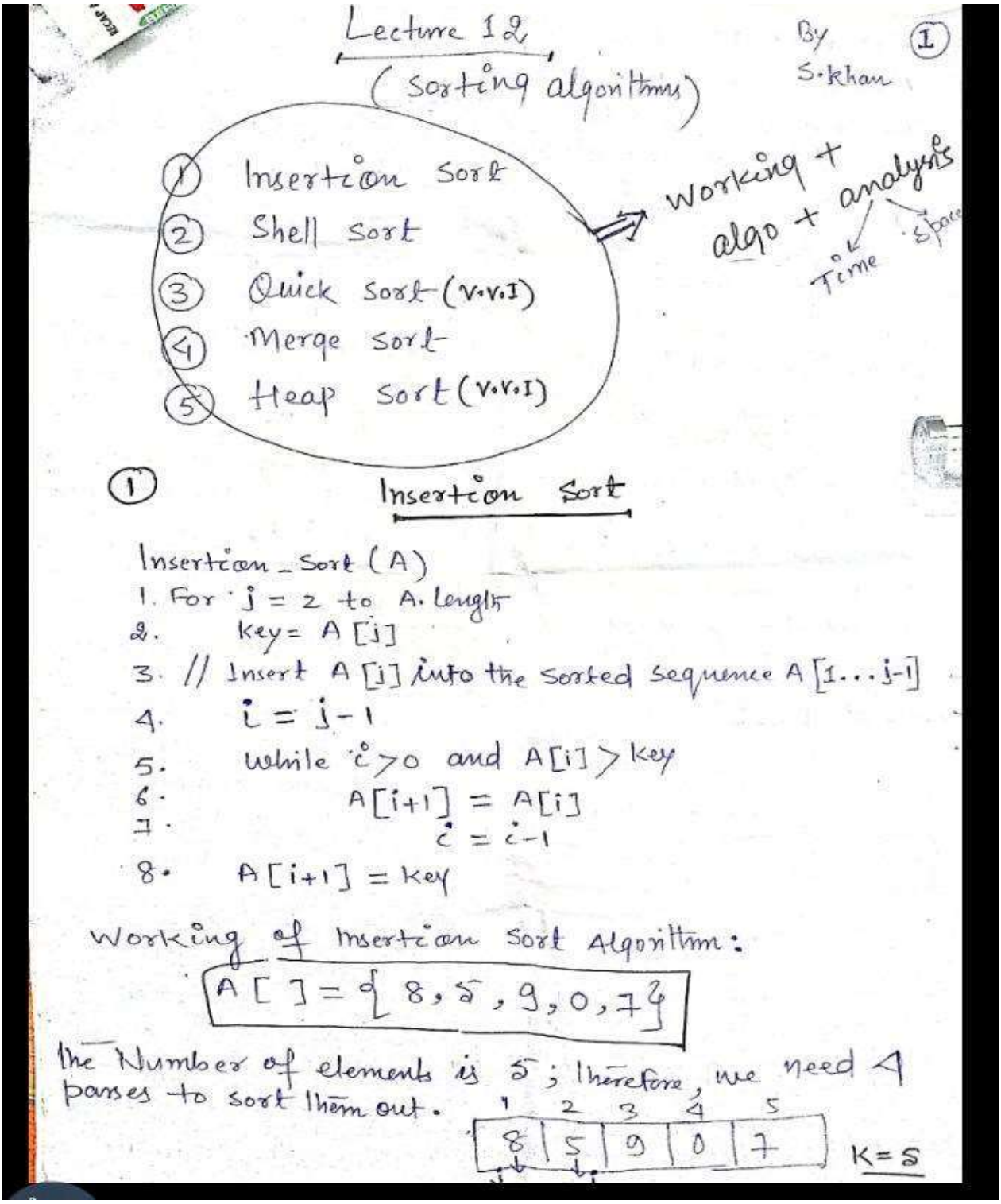
Let every recursive call take a "k" cell then space complexity = ((n+1))k = O(n) (Remove constant)

Depth of stack

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

= O(n) Space complexity

Sorting Algorithms and their Analysis



Lecture 12

(Sorting algorithms)

By S. Khan (1)

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:

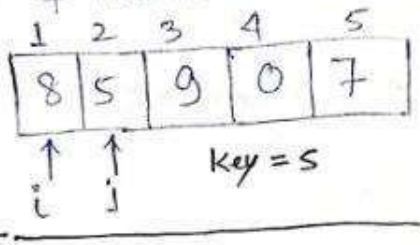
A[ ] = {8, 5, 9, 0, 7}

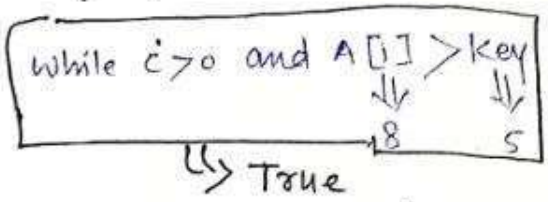
The Number of elements is 5; therefore, we need 4 passes to sort them out.

1	2	3	4	5
8	5	9	0	7

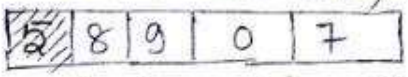
K = 5

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

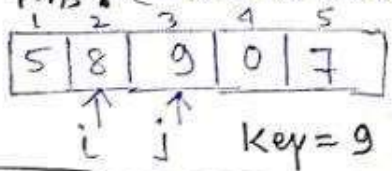


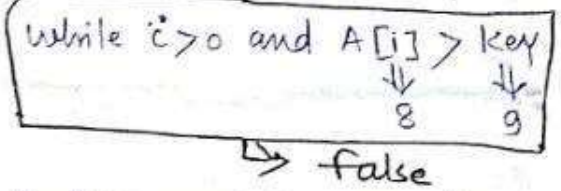


As the condition is true, swap 8 and 5 using the line numbers 6, 7 and 8. So, for now, 5 is sorted in a sorted sub-array.



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

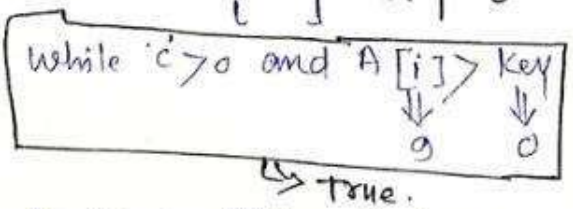




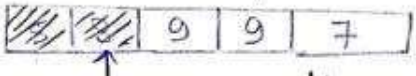
As the condition gets false, line number 8 of algo will be used to place 9 to its original position only.
So, 8 is also sorted in the sorted sub-array along with 5

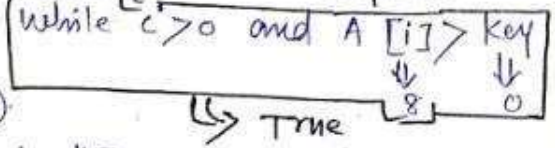
3rd Pass:

Now, move to the next two elements and compare them

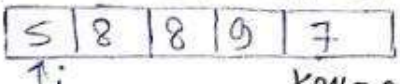


As the condition is true, move 9 to the place of 0 and decrement i by one.

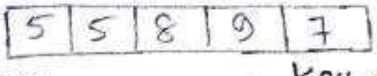


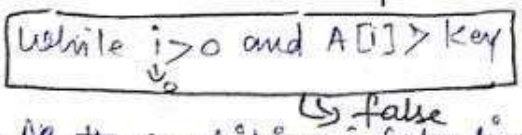


As the condition is true, move 8 to the next cell and i = i-1

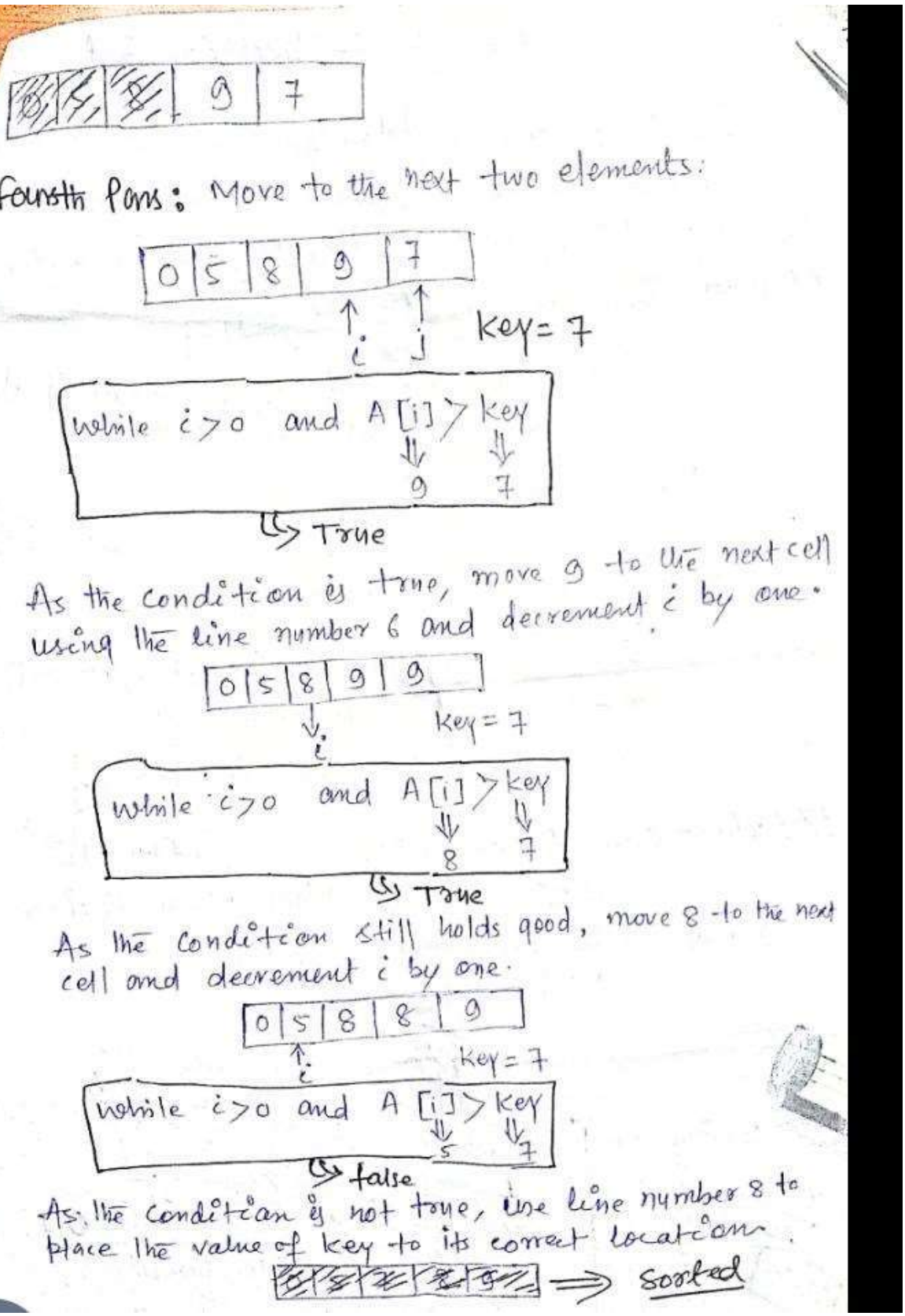


Still condition is true, move 5 to the next cell and i = i-1.

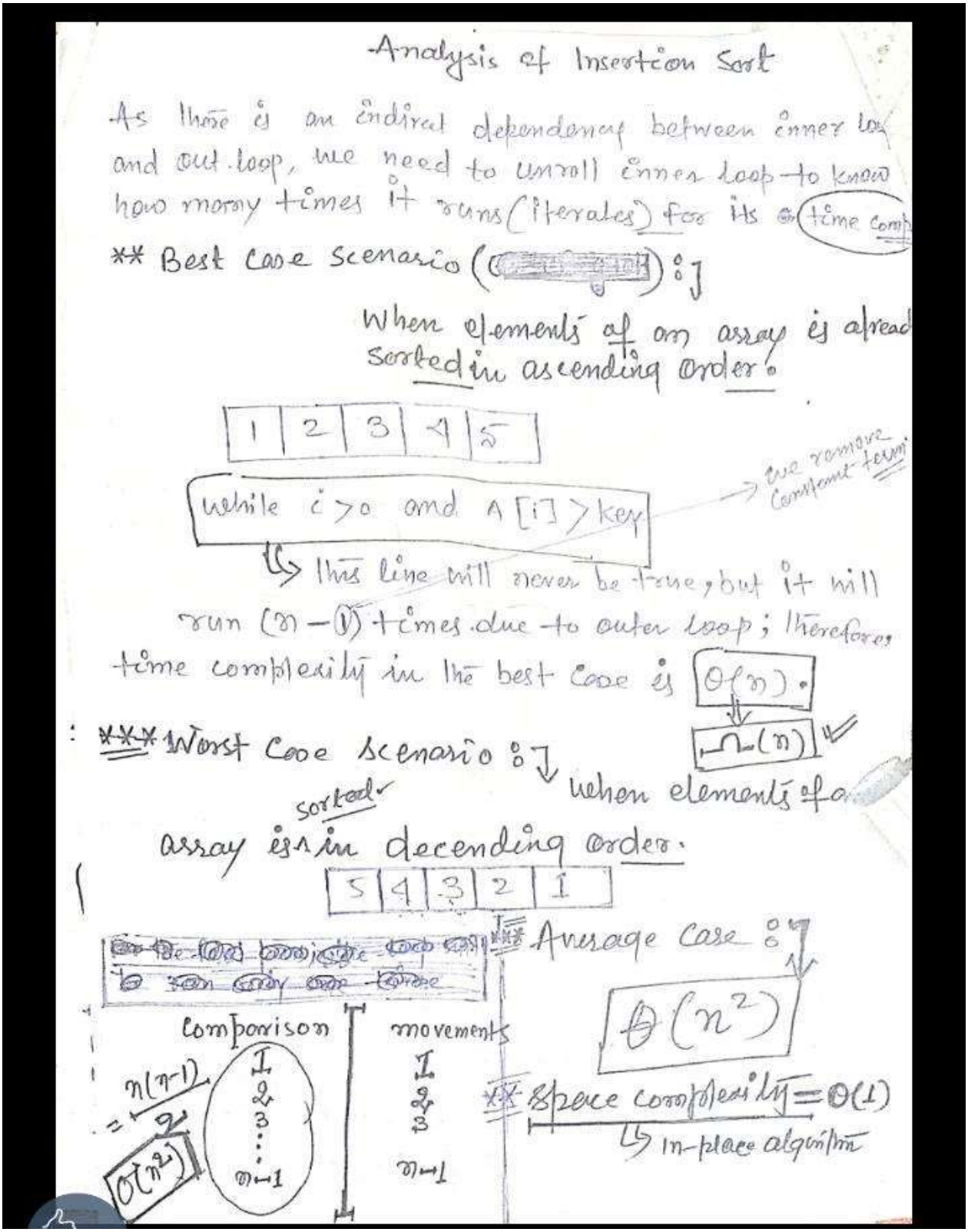




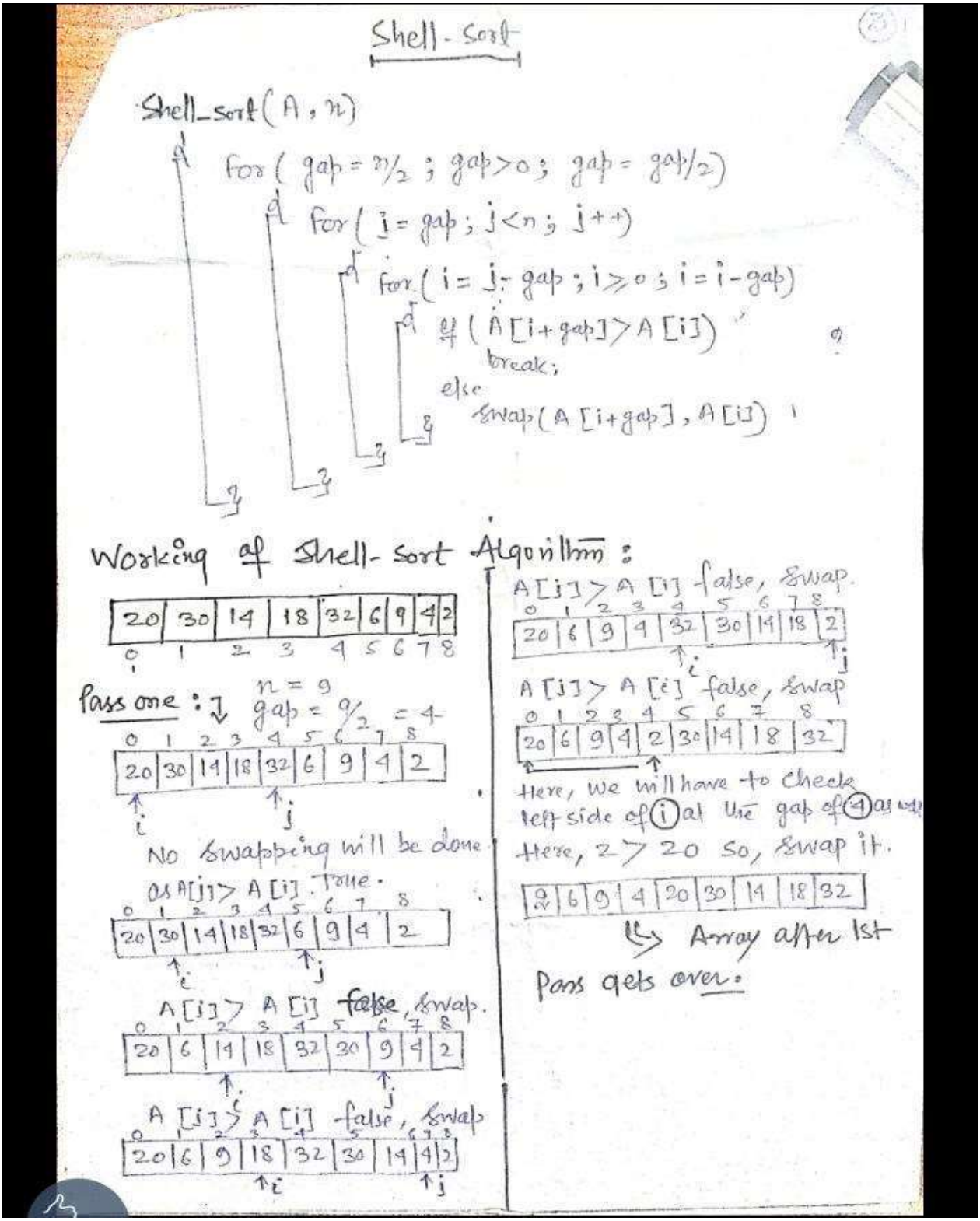
As the condition is false, line number 8 is used to place the value of key to its exact locati


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 using the line number 6 and decrement i by one.
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.
sorted

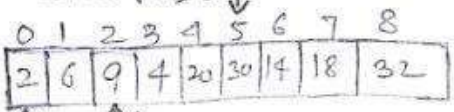
Analysis


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
while i > 0 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 Θ(n).
Ω(n)
*** Worst Case Scenario :)
when elements of a sorted array is in decending order.
5 4 3 2 1
Comparison
n(n−1)/2
O(n²)
1 2 3 ... n−1
movements
1 2 3 ... n−1
*** Average Case :)
Θ(n²)
** Space complexity = O(1)
In-place algorithm

Shell-Sort Algorithm

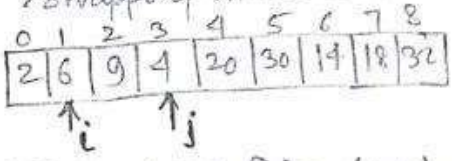


Second Pass: ↓

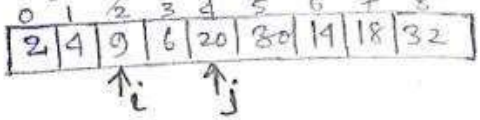


Now, gap = $\frac{4}{2}$ = 2

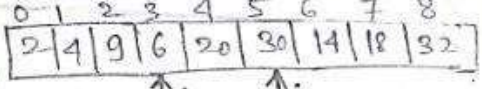
A[i] > A[i] True, no swapping will be done.



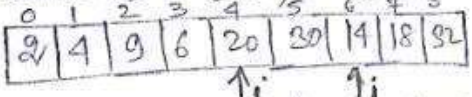
A[j] > A[i] false, swap.



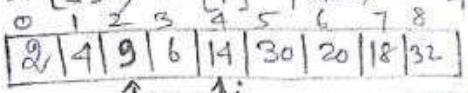
A[j] > A[i] True, no swapp



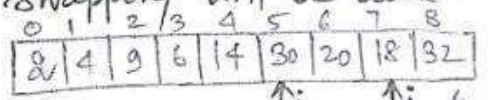
Again no swapping will be done as A[j] > A[i]



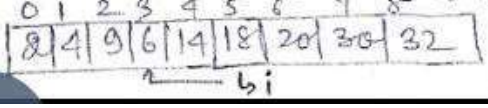
A[j] > A[i] false, swap



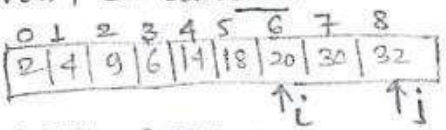
And also check left side of "i" at the gap of 2. Here value of i is greater than 2 less than its index, so no further swapping will be done.



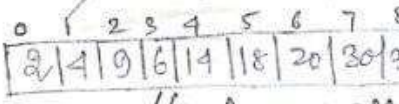
swap



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



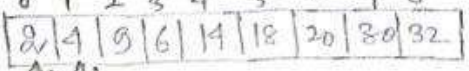
A[j] > A[i] True, no swapping



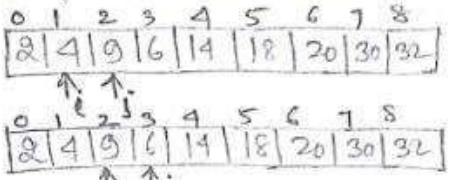
↳ Array after second pass gets over.

Third Pass: ↓

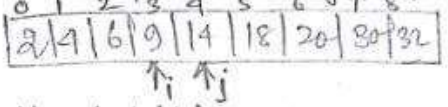
gap = $\frac{2}{2}$ = 1



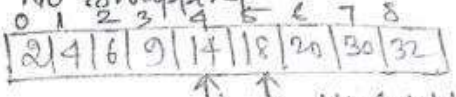
Once gap gets equal to one, it works like insertion sort.



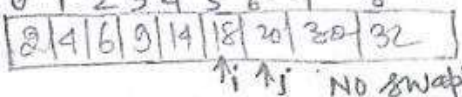
swap and also check left side of i at the gap of one.



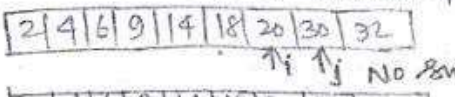
No swapping



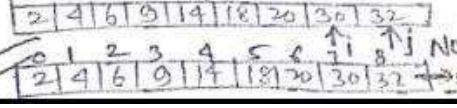
No swapping



No swapp



No sw



→ Sorte

Analysis of Shell-sort (4)

Time complexity:

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

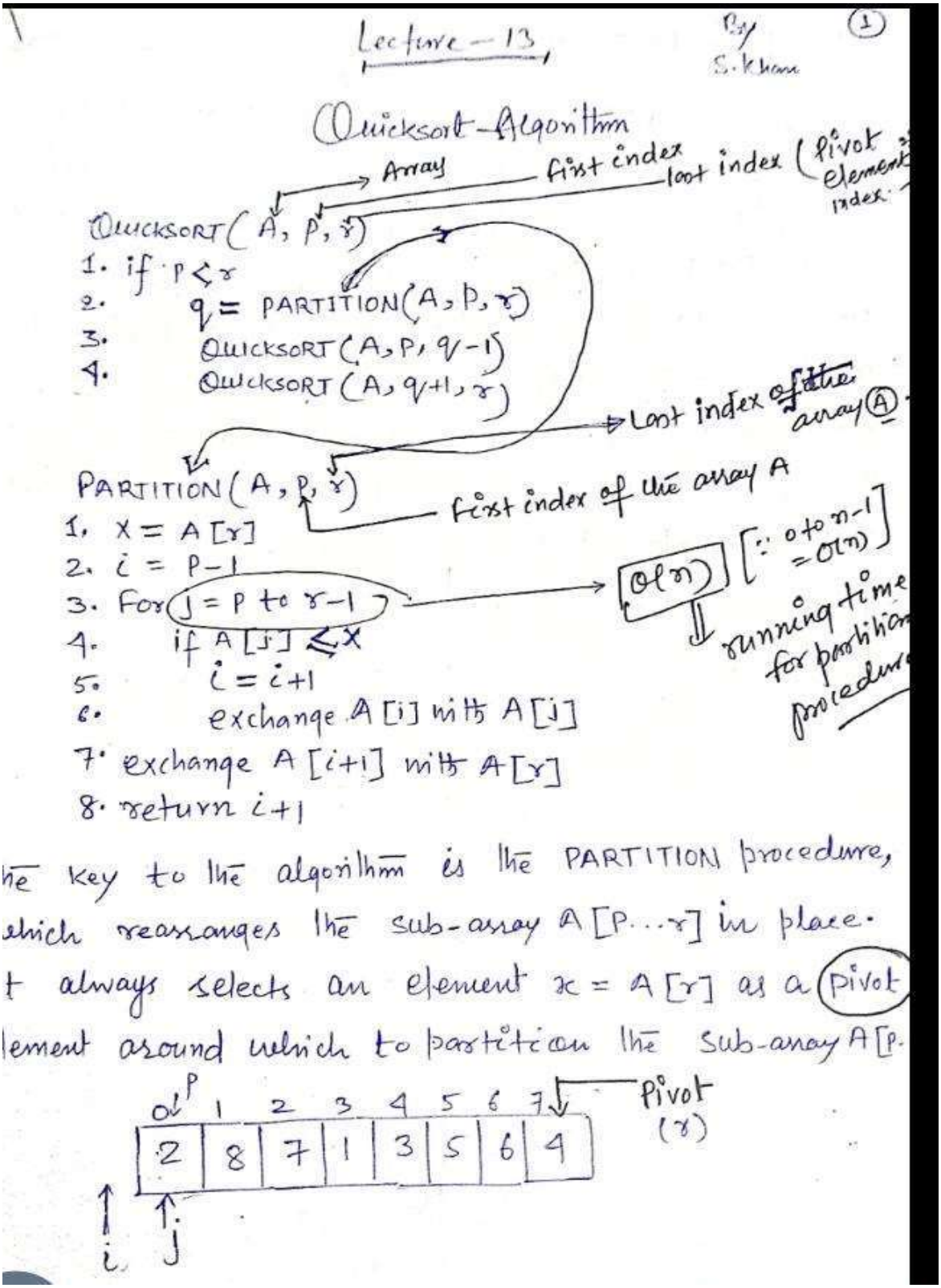
Space complexity: ↓ As it is an in-place algorithm, it will take constant amount of space in the memory.

$O(1)$ ↓

*** 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



Working of Partion Procedure (2)

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

(A)

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

i ↑ , ↑ j ; x = 4

if A[j] ≤ x ⇒ True (2 ≤ 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

if A[j] ≤ 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

if A[j] ≤ 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

if A[j] ≤ 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

(F)

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

↑i ↑j ; x = 4

if A[j] ≤ x ⇒ false; j = j + 1 (7 ≤ 4)

(G)

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

↑i ↑j ; x = 4

if A[j] ≤ x ⇒ false; j = j + 1 (5 ≤ 4)

(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.

For j = p 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	4	5	6	7
2	1	3	4	7	5	6	8

sorted one element

left sub array

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 :-

$T(n) = 2T(\frac{n}{2}) + O(n)$ ← running time for partition procedure.

By master theorem,

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

$a = b^k$ True.

Apply first case $p > -1$

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

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

(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.

$T(n) = T(n-1) + n$ ← Decreasing function.

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 :-

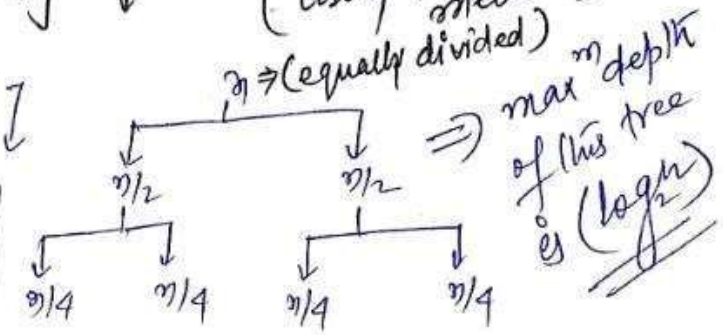
$\boxed{\theta(n \log n)}$

(3)

Space Complexity :] (using tree method)

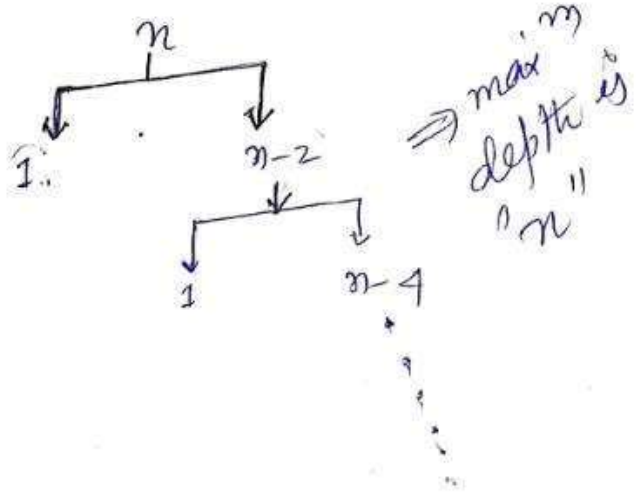
Best Case :]

$\Omega(\log_2 n)$



Worst case :] (Unbalanced)

$O(n)$



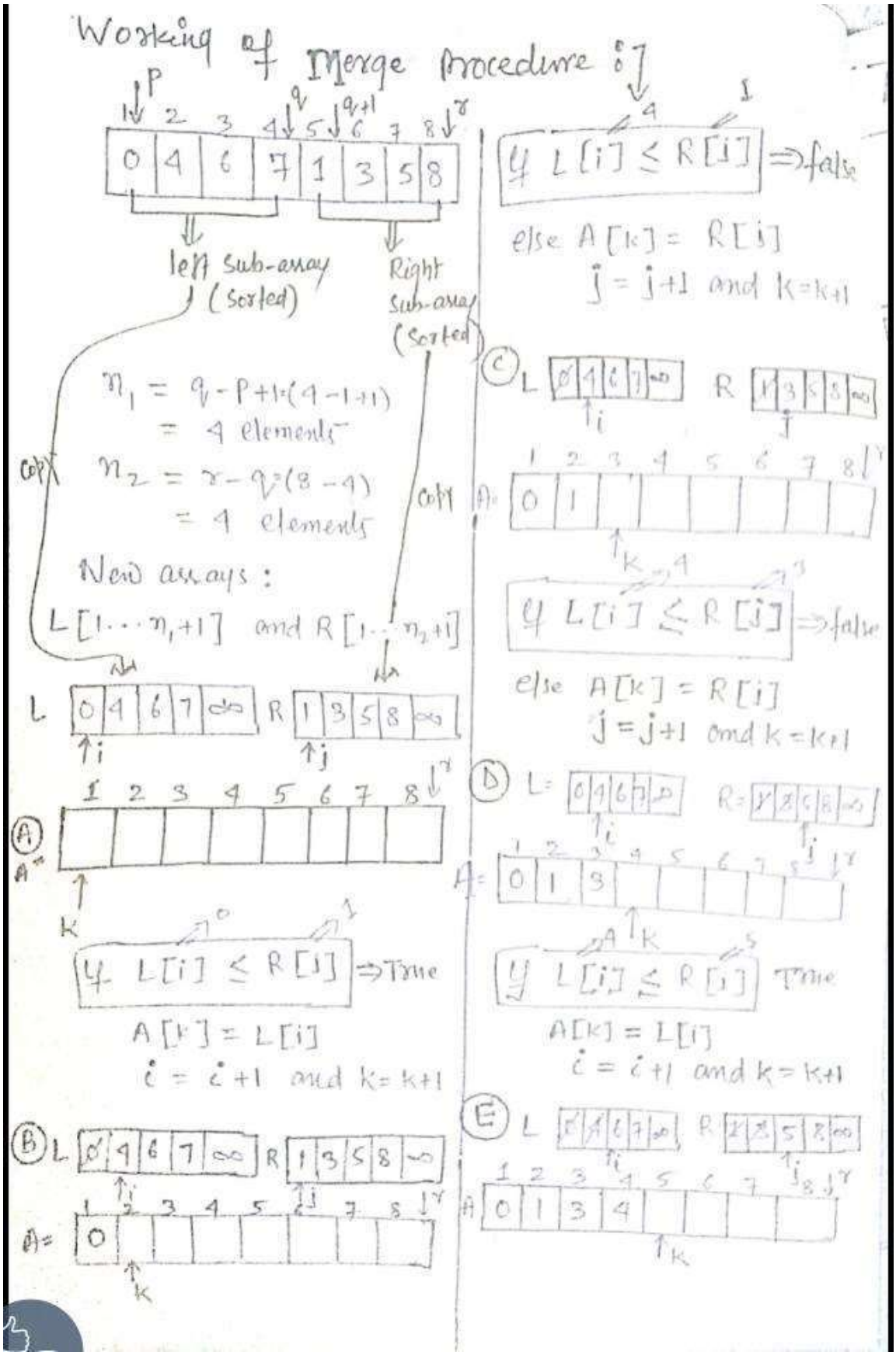
Merge Sort Algorithm

(Out-of-Place algorithm)

```
Merge-Sort(A, P, r)  ⟶ T(n)
1. if P < r
2.     q = ⌊(P+r)/2⌋
3.     Merge-Sort(A, P, q)      ⟶ T(n/2)
4.     Merge-Sort(A, q+1, r)    ⟶ T(n/2)
5.     Merge(A, P, q, r)        ⟶ Θ(n)
```

```
Merge(A, P, q, r)
1   n1 = q - P + 1
2   n2 = r - q
3   Let L[1..n1+1] and R[1...n2+1] be new arrays
4   For i = 1 to n1
5       L[i] = A[P+i-1]
6   For j = 1 to n2
7       R[i] = A[q+j]
8.  L[n1+1] = ∞
9.  R[n2+1] = ∞
10  i = 1
11  j = 1
12  For k = P to r                ⟶ Θ(n)
13      If L[i] ≤ R[j]
14          A[k] = L[i]
15              i = i+1
16      else A[k] = R[j]
17              j = j+1
```

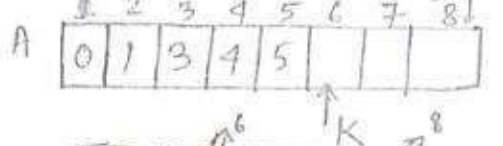
Working of Merge Procedure :



if L[i] ≤ R[j] ⇒ false

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

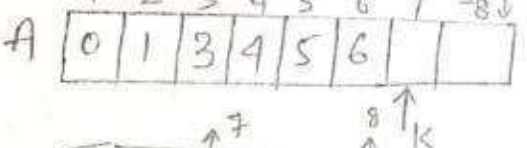
(F) L: 0 1 3 7 ∞ R: 2 3 5 8 ∞



if L[i] ≤ R[j] ⇒ True

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

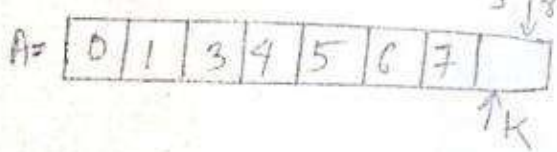
(G) L: 0 1 3 7 ∞ R: 2 3 5 8 ∞



if L[i] ≤ R[j] ⇒ True

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

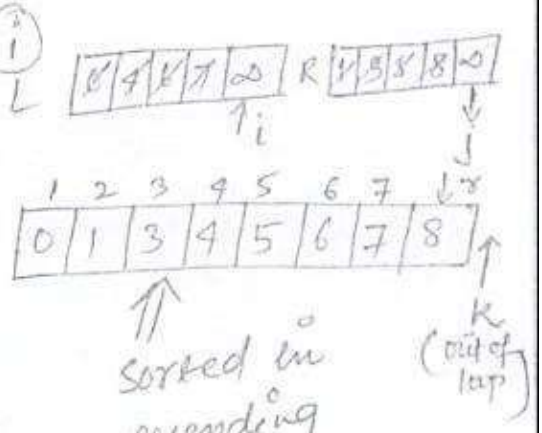
(H) L: 0 1 3 7 ∞ R: 2 3 5 8 ∞



(2)

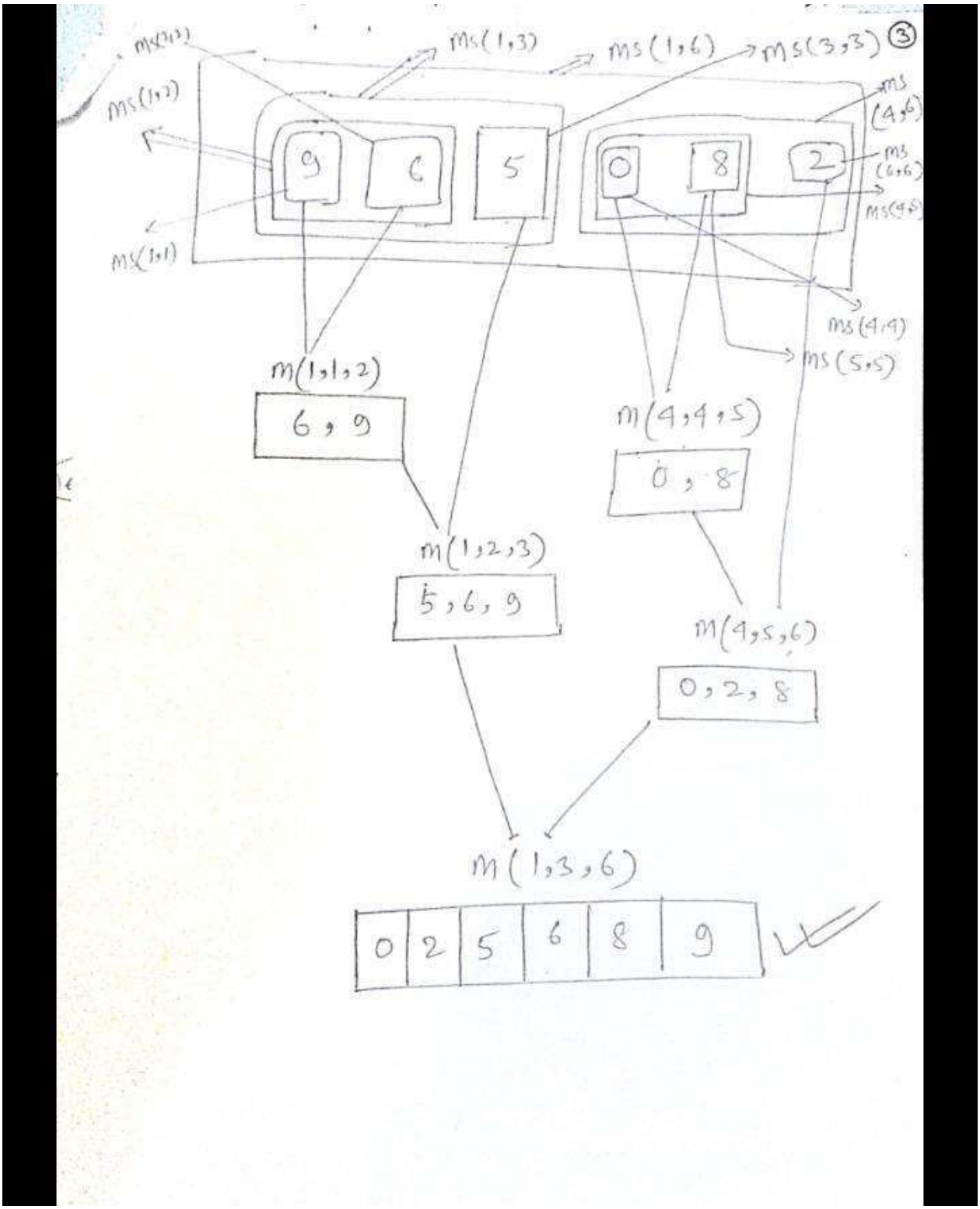
if L[i] ≤ R[j] ⇒ false

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



Sorted in ascending order.

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



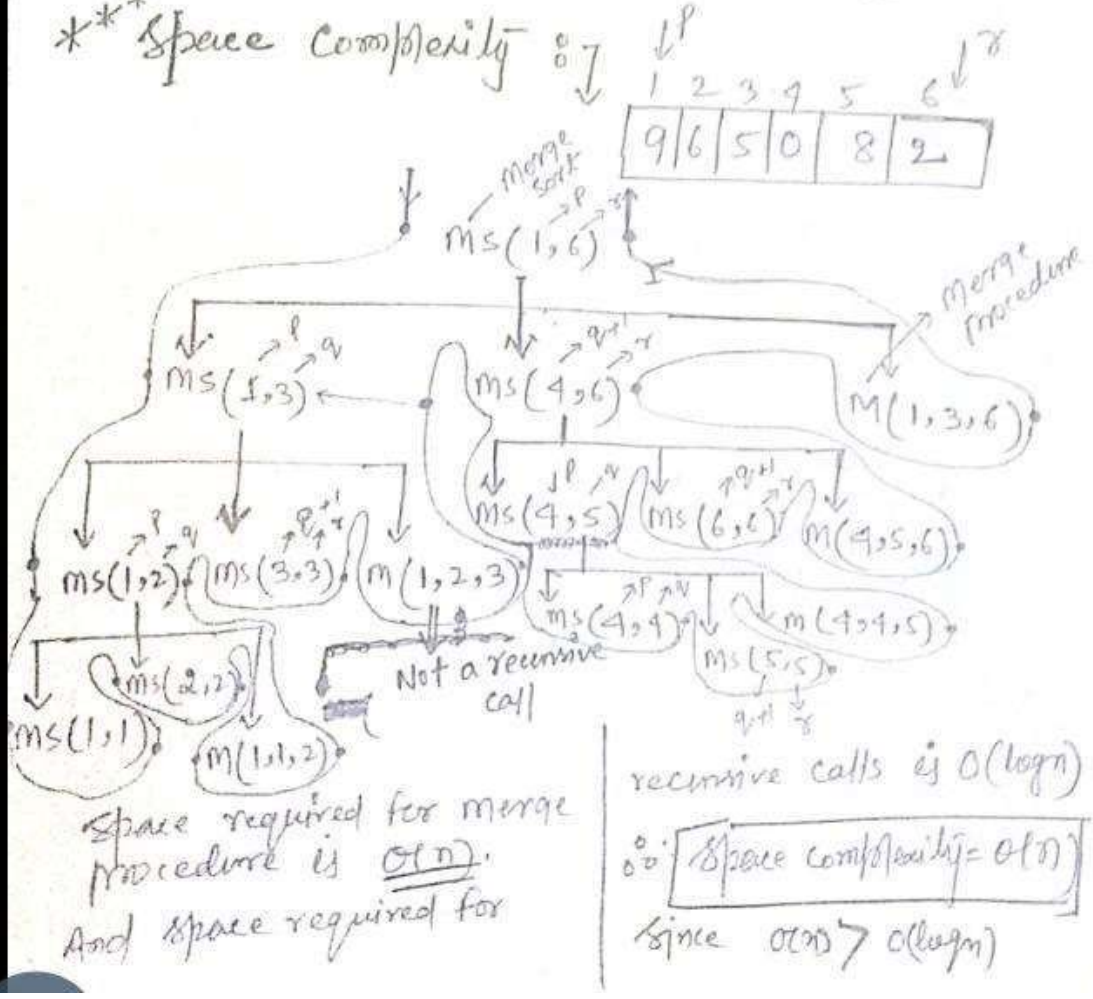
Analysis of Merge Sort Algo

Time Complexity :-

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

$\theta(n \log n)$ → Best as well as Worst Case.

*** Space Complexity :-



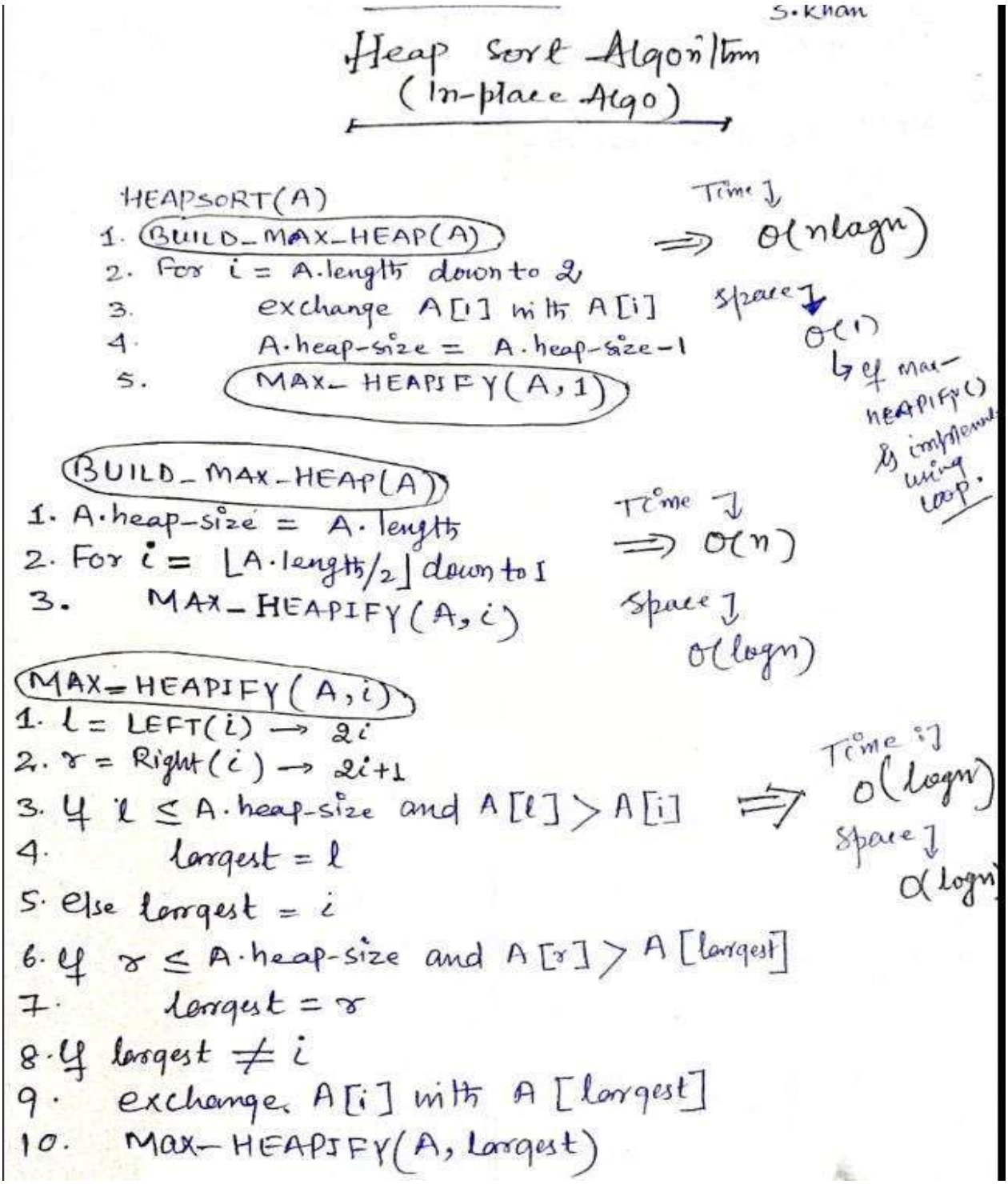
Space required for merge procedure is O(n).

And space required for recursive calls is O(log n)

∴ Space complexity = O(n)

since O(n) > O(log n)

Heap Sort Algorithm

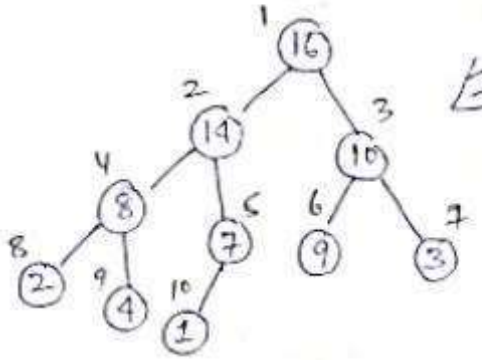


Heap (Binary) :↓

The binary heap data structure is an array object that we can view as a nearly/almost complete binary tree. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left upto a point. For example,

Array ⇒

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1



last level is not fully filled.

fig: A max-heap viewed as a nearly complete binary tree

** Complete Binary Tree (CBT):↓ It is a binary tree, in which insertion takes place level by level and all the nodes are filled from left to right manner. Its last level might/might not be fully filled.

** Almost Complete Binary Tree (ACBT):↓ It is also a complete binary tree, but its last level is not fully filled.

There are two kinds of binary heaps:

(i) Max-heaps
(ii) Min-heaps

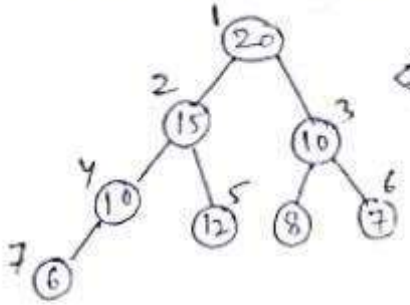
(i) max-heap :

In a max-heap, the max-heap property is that for every node 'i' other than the root;

$A[PARENT(i)] \geq A[i]$

($PARENT(i)$ → parent of i)

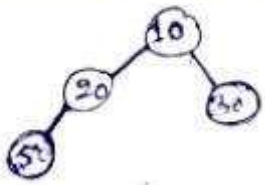
, that is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself.



⇐ Max-heap

(ii) Min-heap :

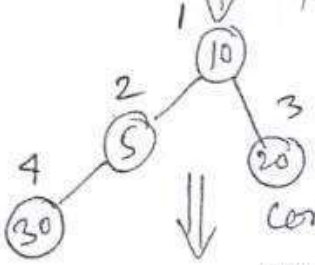
It is organized in the opposite way; the min-heap property is that for every node 'i' other than the root, $A[PARENT(i)] \leq A[i]$. The smallest element in a min-heap is at the root.



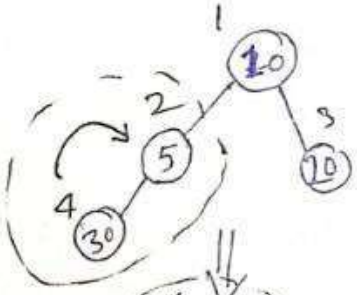
⇐ Min-heap

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

So" :] Convert it into an almost binary tree



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

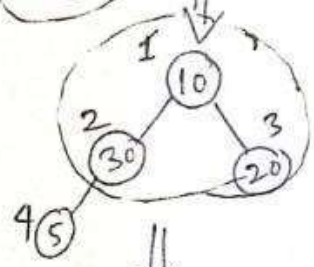


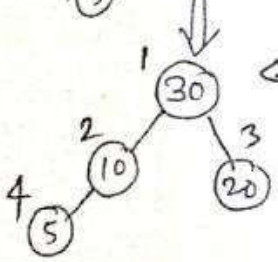
formula to find out non-leaves:

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

$\lfloor \frac{5}{2} \rfloor$ · to 1

2, 1 ⟹ Non-leaf nodes.

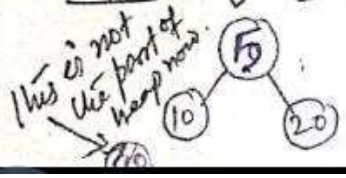




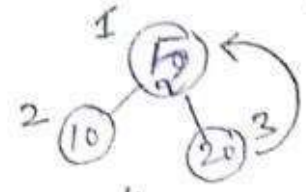
⟸ max-heap

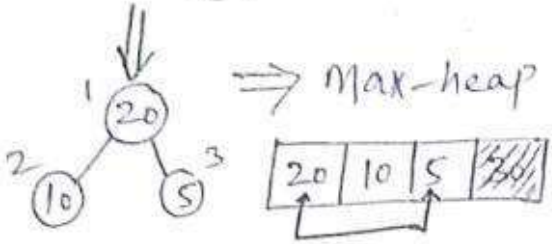
1	2	3	4
30	10	20	5

Step 2 :] Exchange root with the last node.

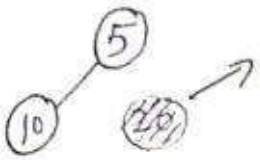


step3 : ↓ Convert it into max-heap

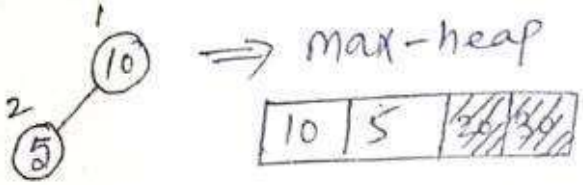




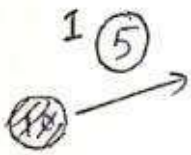
step4 : ↓ Exchange root with the last node.



step5 : ↓ convert it into max-heap.



step6 : ↓ Exchange root with the last node.



Heapsort runs from $i = A.length$ to 2 index only; because 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.

Sorting in linear time O(n)

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

Lecture-16

By (1) Sakhan

Counting-Sort-Algorithm

Counting-Sort (A, B, k) (A: Array, B: Array, k: range)

```
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 zeros
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 [Pre] + [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 :

Time = $O(n)$

Space = $O(n)$ [$\because 2n = O(n)$]

Illustrate the operation of Counting-sort on the array $A = \{4, 0, 2, 1\}$

Initial set-up using Line numbers 1 to 3. : Range (4)

	1	2	3	4
A =	4	0	2	1

K

C ⇒ Create an array with size 5 (0 to 4)

	0	1	2	3	4
C =	0	0	0	0	0

K

⇓

Initialize it with all zeros.

Update array "c" using line numbers 4 and 5

For j = 1 to A.length

{ c [A[j]] = c [A[j]] + 1 }

c [A[1]] = c [A[1]] + 1

c [4] = c [4] + 1

Index number 4 of the array "c" will be updated by one. c =

	0	1	2	3	4
	0	0	0	0	1

→ It will run till index 4 of the array (A) The condition

If 2nd iteration

Also, c [A[2]] = c [A[2]] + 1

⇓

c [0] = c [0] + 1

⇓

Zero index of the array "c" will be incremented by 1.

	0	1	2	3	4
C =	1	0	0	0	1

3rd iteration:

c [A[3]] = c [A[3]] + 1

⇓

c [2] = c [2] + 1

⇓ Second index of the array "c" will be incremented by one.

	0	1	2	3	4
C =	1	0	1	0	1

4th iteration (last)

c [A[4]] = c [A[4]] + 1

⇓

c [1] = c [1] + 1

⇓ 1st index number of the array c will be incremented by one.

	0	1	2	3	4
C =	1	1	1	0	1

Now, again update the array C using the line numbers (7) and (8).

For i = 1 to k

C[i] = C[i] + C[i-1]

(current, previous)

	0	1	2	3	4
C =		1	2	3	3	4

Now, make the resultant array B using the line numbers (8), (9) and 10.

For j = A.length downto 1

B[C[A[j]]] = A[j]

C[A[j]] = C[A[j]] - 1

→ these two lines help us get the sorted array B. The loop runs from 4 to 1.

B[C[A[j]]] = A[j] (j = 4)

B[C[A[4]]] = 1

B[C[3]] = 1

B[2] = 1

↳ put "1" at the place of index B[2].

and decrement C[3] by 1

	1	2	3	4
B =		1		

(2)

and iteration of

	1	2	3	4
	1	1	3	3	4

(j = 3)

B[C[A[j]]] = A[j] (j = 3)

B[C[A[3]]] = 2

B[C[2]] = 2

B[3] = 2

↳ put 2 at B[3]

	1	2	3	4
B =		1	2	

and decrement C[2] by one.

	0	1	2	3	4
C =	1	1	2	3	4

3rd iteration: (j = 2)

B[C[A[j]]] = A[j]

B[C[A[2]]] = 0

B[C[0]] = 0

B[1] = 0

↳

	1	2	3	4
B =	0	1	2	

and decrement C[0] by 1

	0	1	2	3	4
C =	0	1	2	3	4

Lecture - 17 (1)

By S. Khan

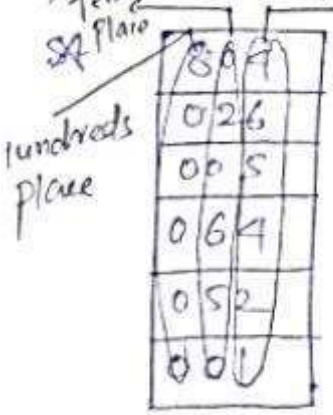
Base

Radix Sort

```
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 to sort
    "A" on digit "i".
```

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

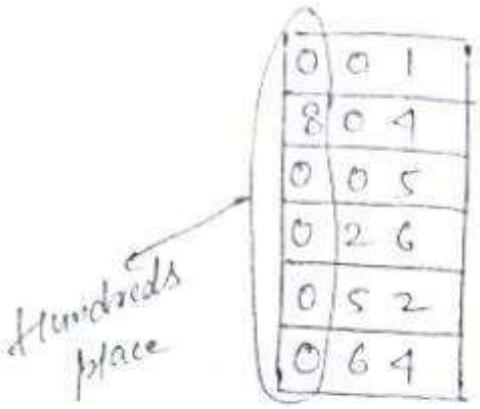
Step 1: As the largest number (804) contains three digits, make all the numbers three digits by appending zeroes.



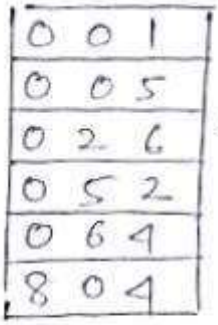
Step 2: Take the unit place and apply a "stable sort".

⇓ Relative ordering of the same element does not change after sorting

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

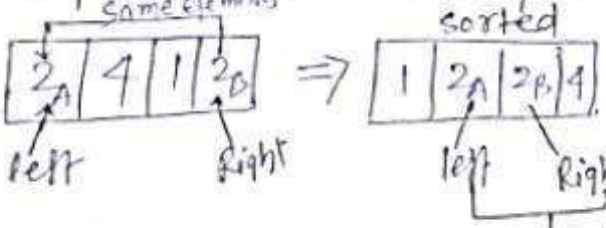


Step 1: Take the hundreds place and apply the stable sort.



⇒ 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:



Analysis:

Time:

$d(O(n+b))$ if $l = O(n^k)$

digits $\log_b^l$ base n largest number

then $\times \log_b^n (O(n+b))$

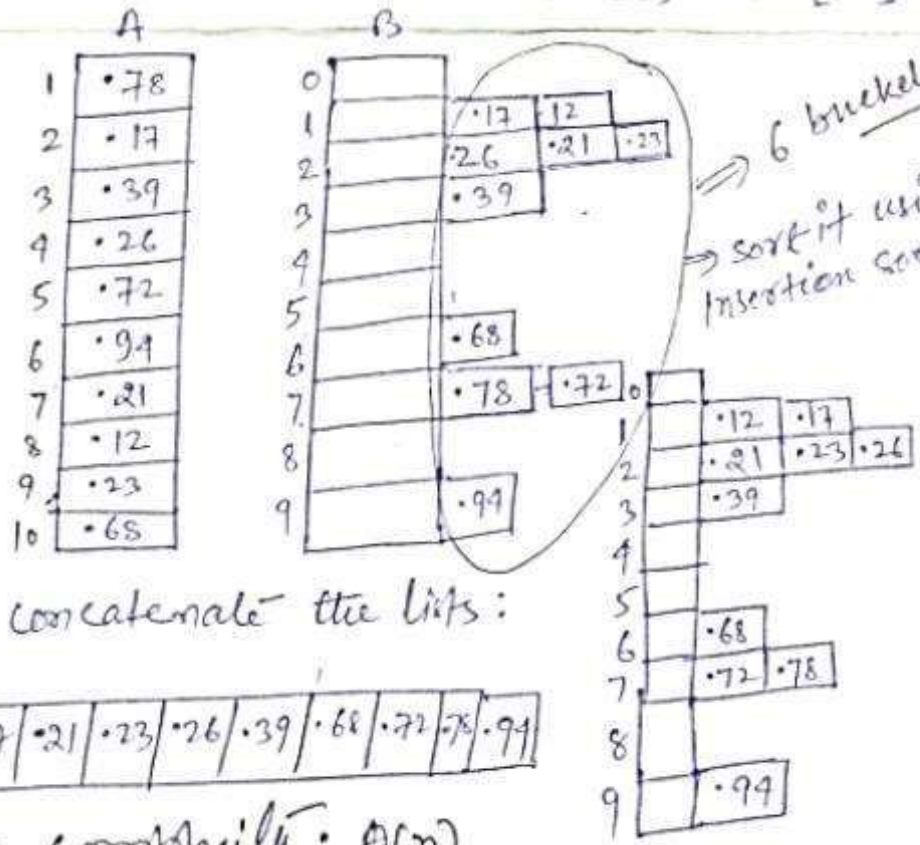
$\Theta(n \log_b^n)$

Bucket-Sort Algorithm

```
Bucket_Sort(A)
1. Let B[0...n-1] be a new array
2. n = A.length
3. For i = 0 to n-1
4.     make B[i] an empty list
5. For i = 1 to n
6.     insert A[i] into list B[⌊n·A[i]⌋]
7. For i = 0 to n-1
8.     sort list B[i] with insertion sort
9. Concatenate the lists B[0], B[1], ..., B[n-1] in order
```

n/2 elements.

Ex 1



Now, concatenate the lists:

.12	.17	.21	.23	.26	.39	.68	.72	.78	.94

Time complexity: $\Theta(n)$

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.

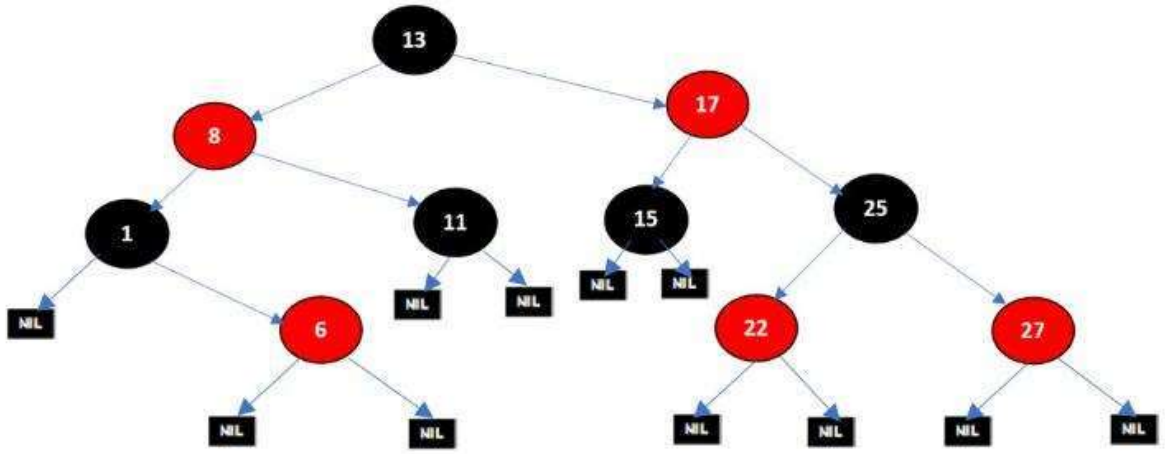


Fig – Red Black Tree

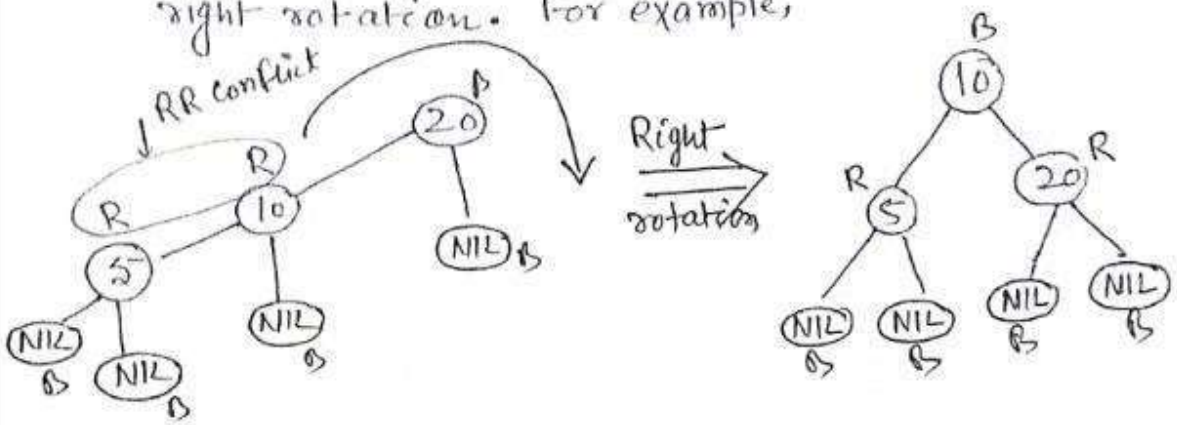
Q1. Explain various rotations in an RB Tree.

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

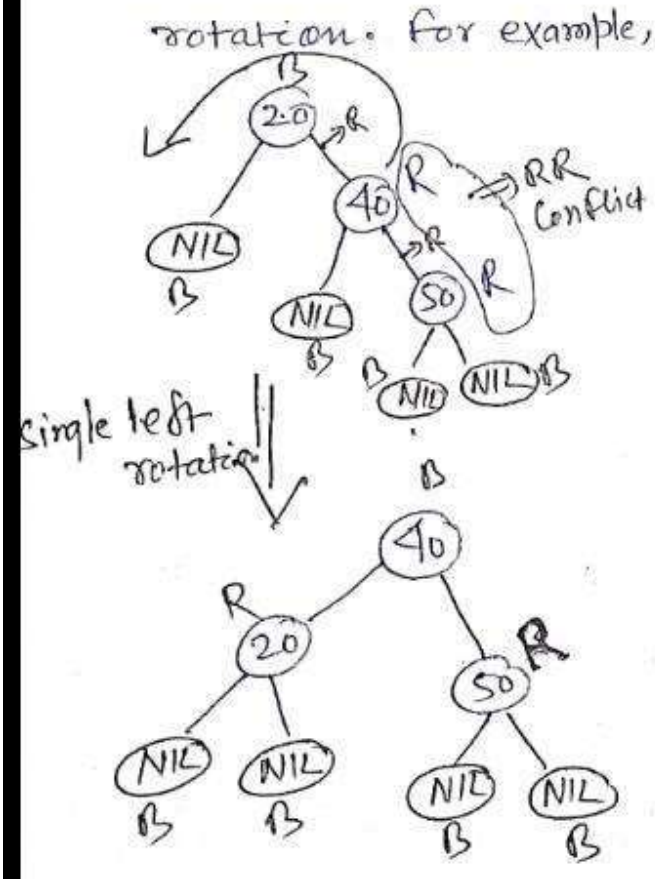
1. Left-Left (LL) problem: Needs a single right rotation
2. Right-Right (RR) problem : Needs a single left rotation
3. Left-Right (LR) problem: Needs one left and then one right rotations.
4. Right-Left (RL) problem: Needs one right and one left rotations.

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.

(1) Left-Left (LL) Problem :- It requires a single right rotation. For example,

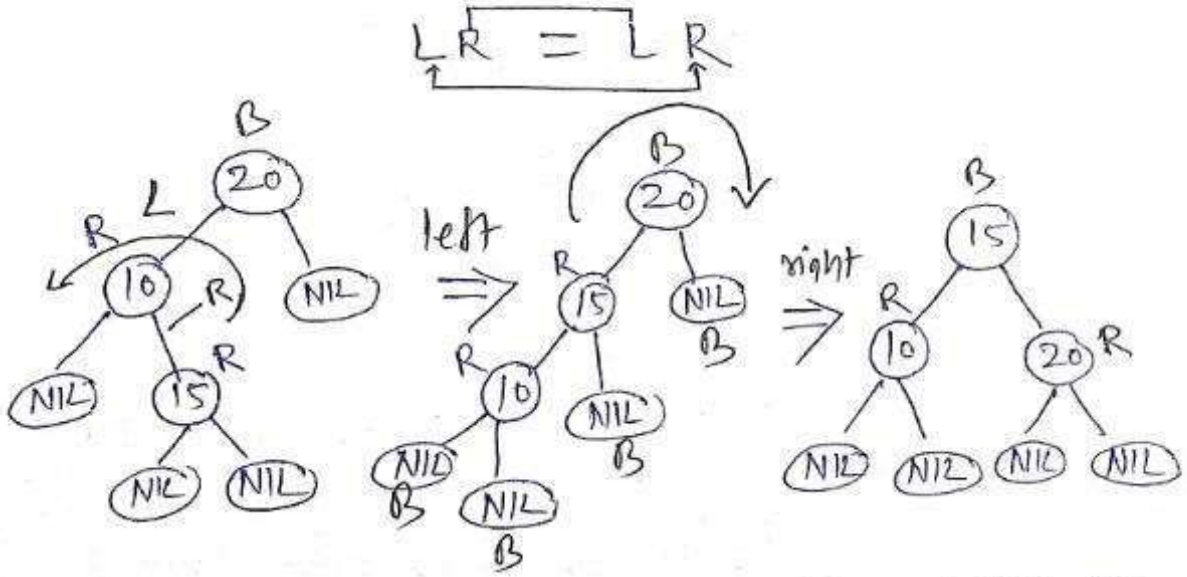


(2) Right-Right (RR) Problem :- It requires a single Left rotation. for example,



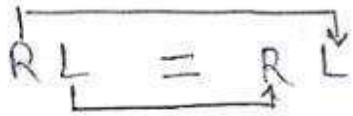
(3) Left-Right Problem :
(L-R)

It requires left rotation and then right rotation. For example.

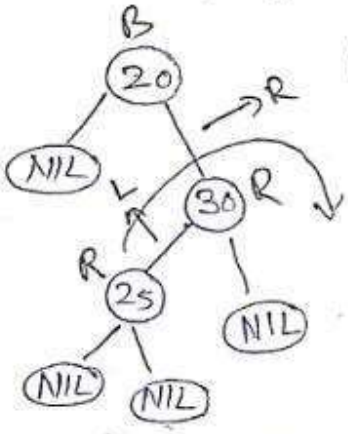


Note: ⇒ Last rotated nodes (15) and (20) are recolored.

④ Right-left (RL) problem:

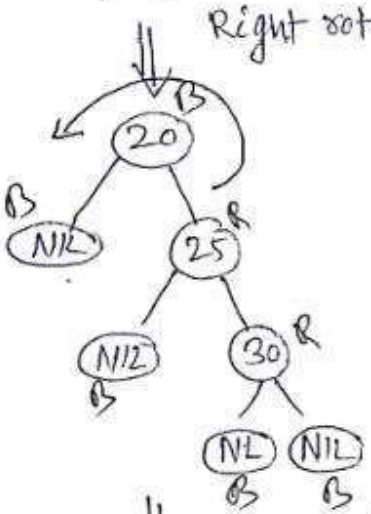


It requires one right rotation and then one left rotation. For example.

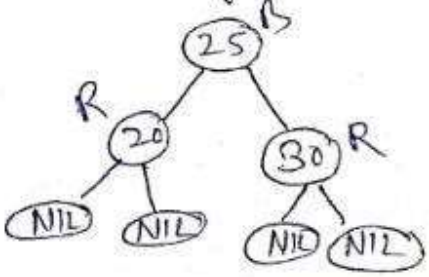


Note: Two adjacent Rs are not allowed.
⇓
Called RR Conflict.

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. Write an algorithm for insertion of keys in an RB Tree and also insert the following keys <5,16,22, 25, 2,10,18,30,50,12,1> in an empty RB Tree.

Algorithm for insertion

1. If the tree is empty, create a new node as the root node with color “black”.
2. If the tree is not empty, then insert the new node with color “red”.
3. If the parent of the new node is “black”, then exit.

4. If the parent of new node is “red”, then check the color of parent’s sibling, which is the uncle of the newly inserted node.

   4(a) If its color is black or Null (no uncle), then make suitable rotation(s) and recolor the last rotated nodes only.
   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.

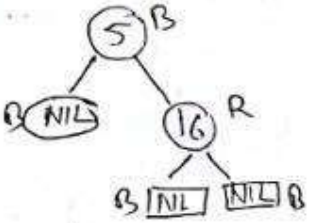
Again, check if the tree is an RB tree or not. If not, then consider grandfather as a newly inserted node and apply line number -4 again.

Insert the following keys in an empty RB tree.

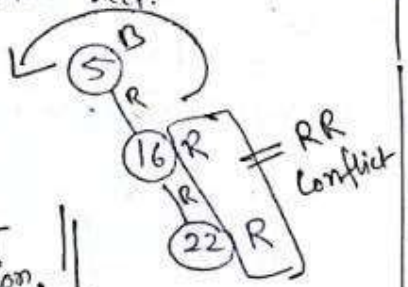
↳ <5, 16, 22, 25, 2, 10, 18, 30, 5, 12, 7>

Step 1 : Insert 5 in an empty RB tree with color black.

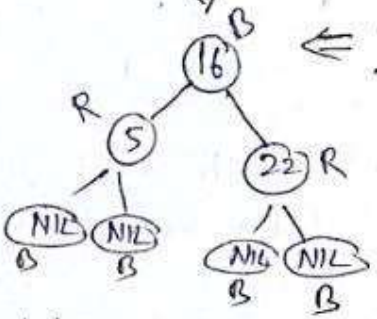
Step 2 : Insert the next element (16) with color red.



Step 3 : Insert the next element (22) with color red.

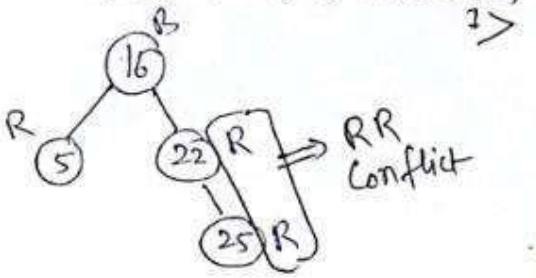


one left rotation



⇐ Recolor rotated nodes only.

Step 4 : Insert the next key 25 with color red.

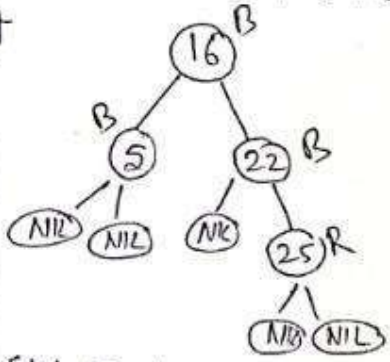


As uncle (5) is red, recolor the following:

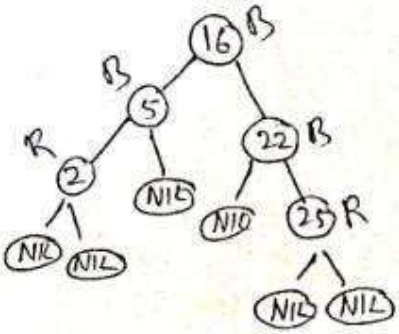
(i) Parent

(ii) Uncle

(iii) Grandfather will not be recolored as it is the root of the tree.

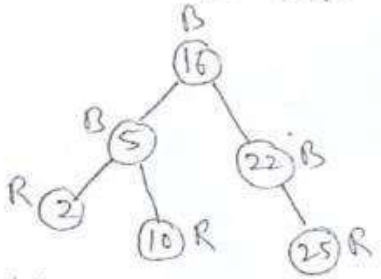


Step 5 : Insert the next key (2) with color red.



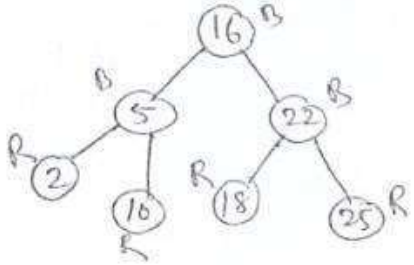
No change as no conflict (RR).

Step6: Insert the next key (10) with color red.



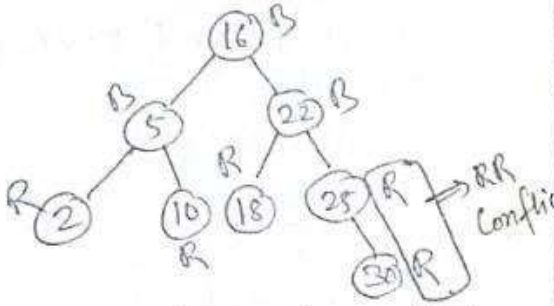
No Change as no RR conflict.

Step7: Insert the next key (18) with color red.

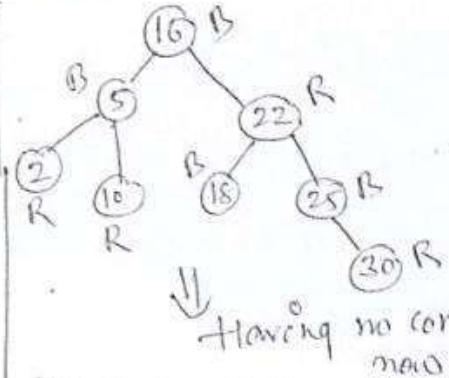


No Change as no RR conflict

Step8: Insert the next key (30) with color red.

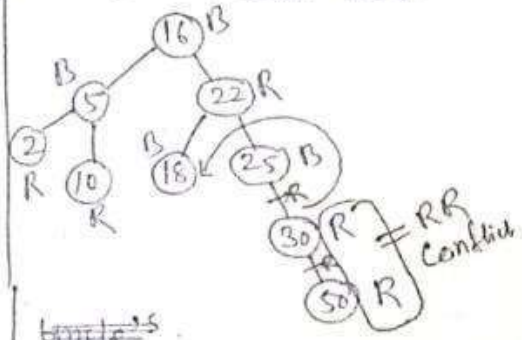


Uncle of 30 is red(18), recolor the following:
(I) Parent(25)
(II) Uncle(R)
(III) Grandfather(22)



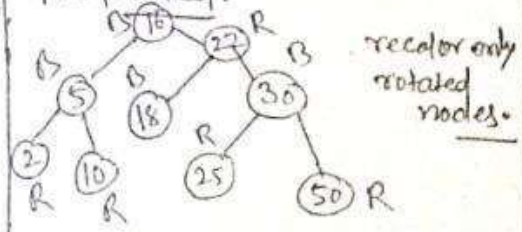
Having no conflict now.

Step9: Insert the next key (50) with color red.



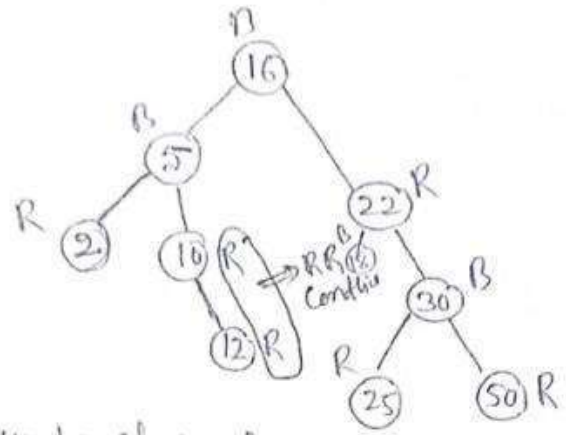
~~Uncle's~~

As 50 has no uncle (NULL), rotation will be performed.



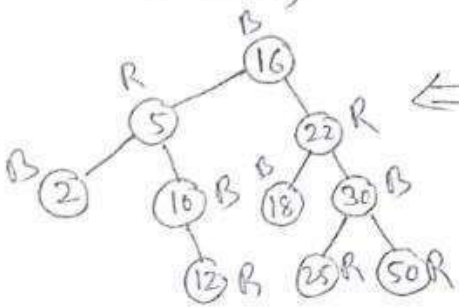
recolor only rotated nodes.

Step10: Insert the next key (12) with color red.



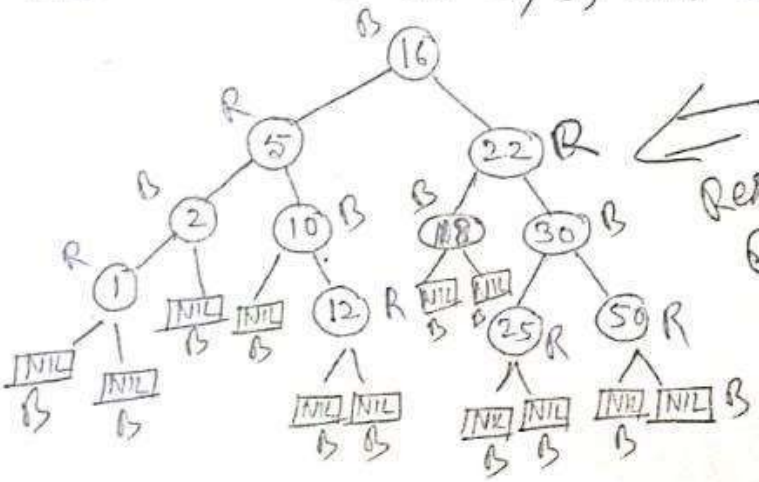
As uncle of 12 is red (2), just recolor the following:

(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:

Binary search tree

Step 1: Choose the node to be deleted like in BST.

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 colors 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

6(d.) Remove DB

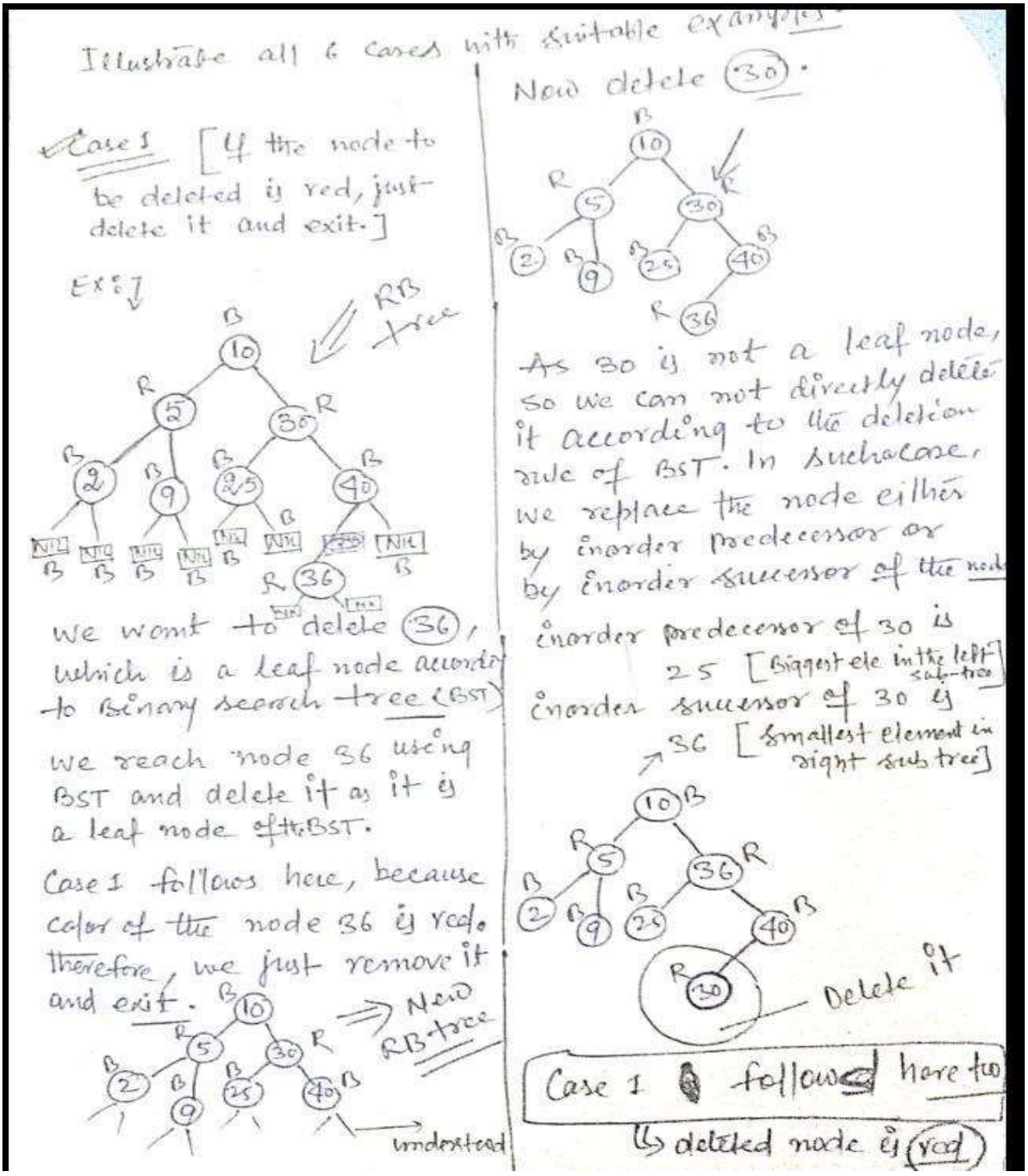
(a) Swap Colors of DB's Parent & DB's Sibling

(b) Rotate DB's Parent in DB direction

(c) Change Color of red nephew to black.

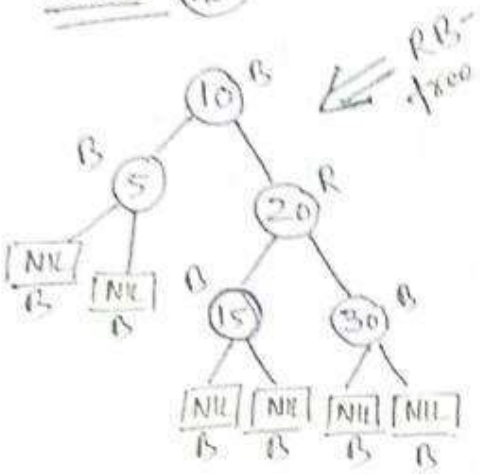
Q4. Explain about double black node problem in RB tree.

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.



Case 03

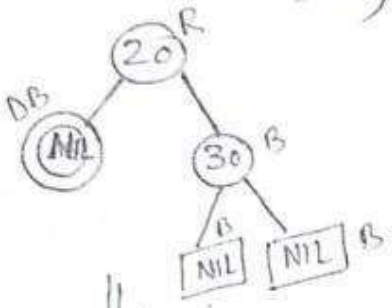
Delete (15)



Here, node to be deleted is "black" and both its sibling nephews are also black.

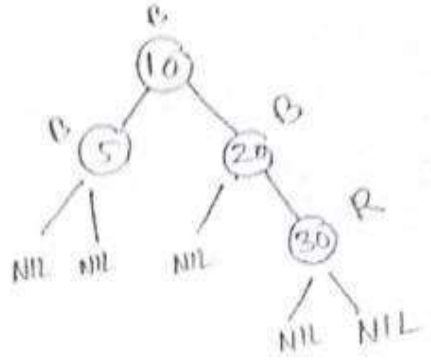
When we replace 15, which is black by its inorder predecessor or inorder successor [NIL], it becomes double black:

(O) ⇒ double back (DB)



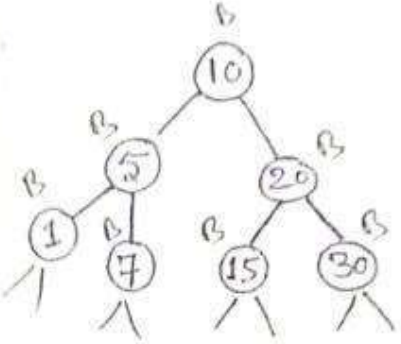
(a) Remove DB
(b) Make sibling (30) red
(c) If Parent is originally red, make it black.

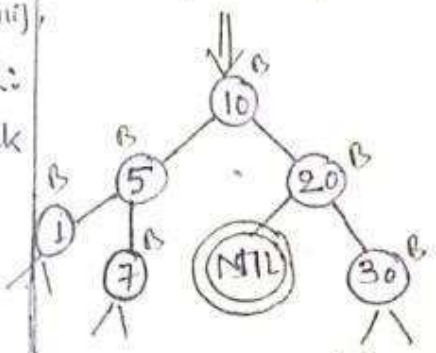
(2)



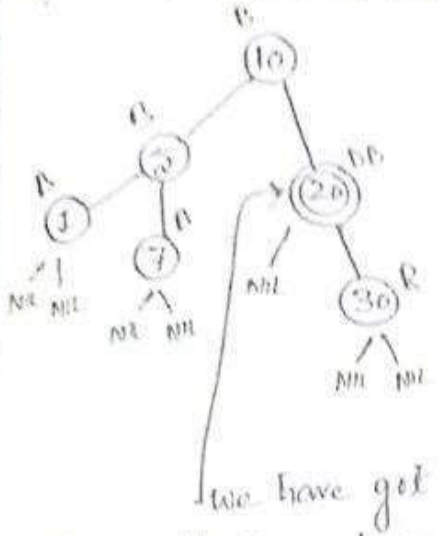
Case 03

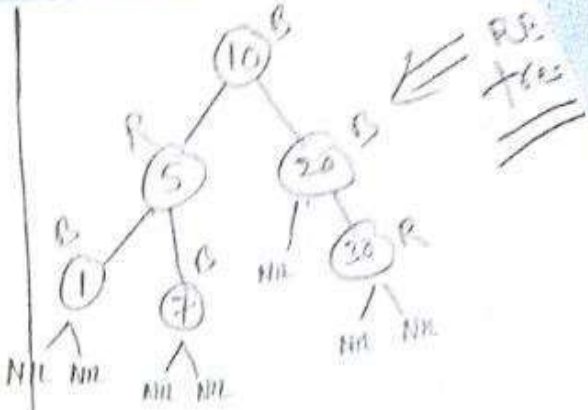
Delete (15)





(a) Remove DB
(b) Make sibling red
(c) Make Parent DB and reapply suitable case. (Since Parent (20) is Black).

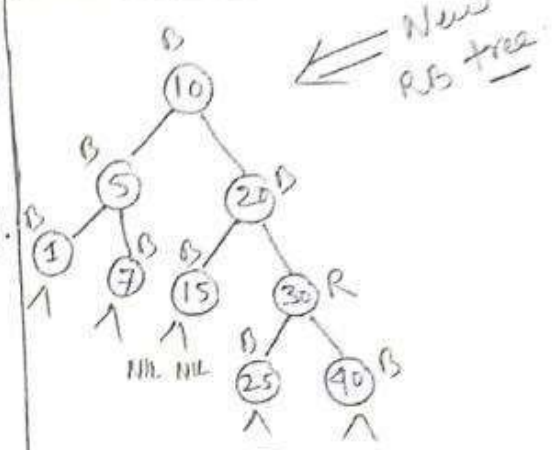




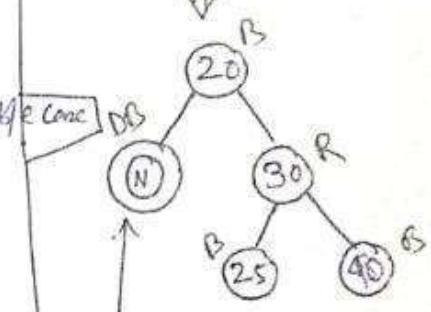
We have got another double black node, so we need to get rid of it also.

Again Case-03 will be applied.
↳ DB's (20) Sibling is black (5) and both its nephews (1) and (7) are also black, then

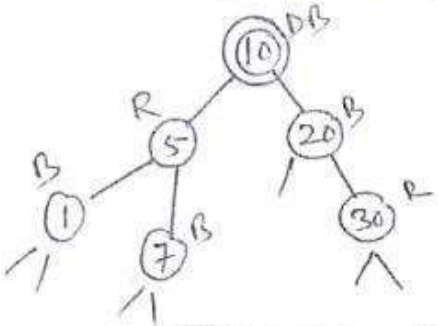
(a) Remove DB (make it single black)
(b) Make sibling red
(c) Make parent of (20) DB and reapply suitable case



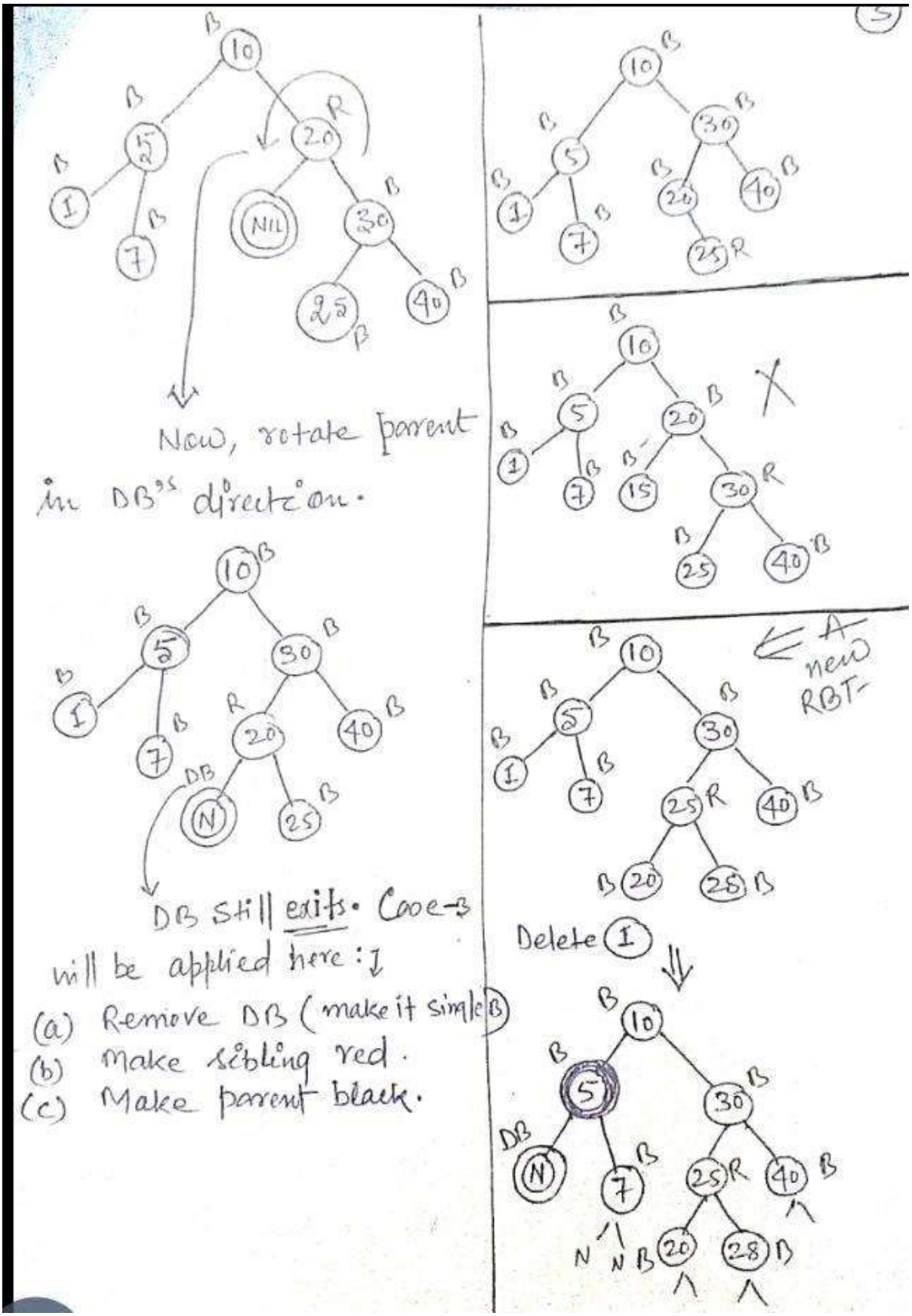
Delete (15)



Case 4: DB's sibling is red
Then:
(a) Swap color of DB's parent & DB's sibling.
(b) Rotate Parent in DB's dir.
(c) Reapply a suitable case.



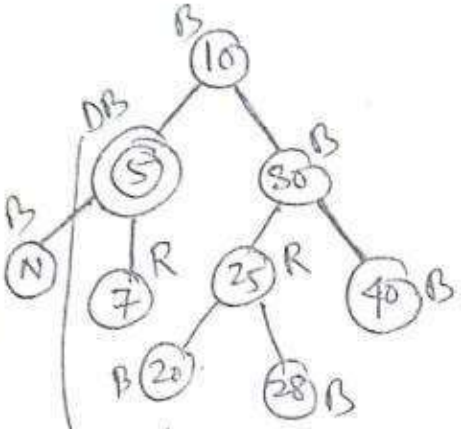
Now Case-2 will be applied.
As DB is root, remove DB-make single black.


Now, rotate parent in DB's direction.
DB still exits. Case-3 will be applied here: I
(a) Remove DB (make it simple)
(b) Make sibling red.
(c) Make parent black.
A new RBT.
Delete (1)

Case-3 will be applied:

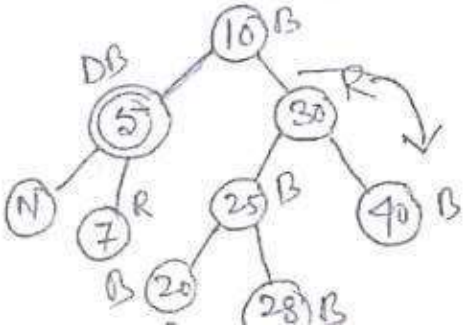
(a) Remove DB
(b) Make sibling red
(c) Make parent DB

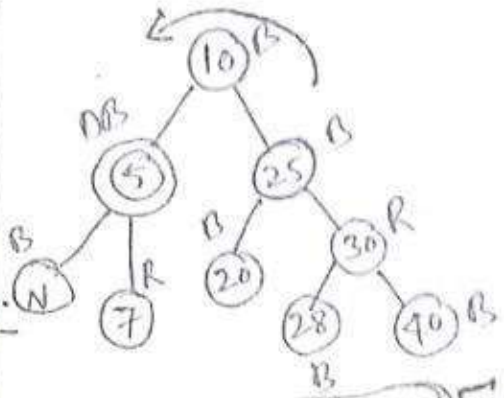
and reapply suitable case.



Case-5 will be applied since DB's sibling (30) is black, for nephew is black (40) and near nephew (25) is red.

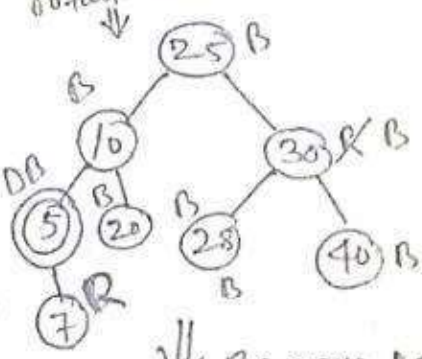
(a) Swap color of DB's sibling and DB's near nephew.
(b) Rotate DB's sibling in the opposite direction of DB
(c) Apply case-6.



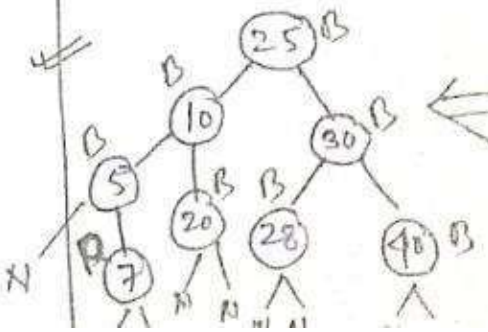


Now apply Case-6 [Note- Every time, we apply case-5, we have to apply case-6 as well].

(a) Swap color of DB's Parent and DB's sibling (Here, both have same color)
(b) Rotate DB's parent in DB direction
(c) Change color of red nephew
(d) Remove DB (Make it single B).

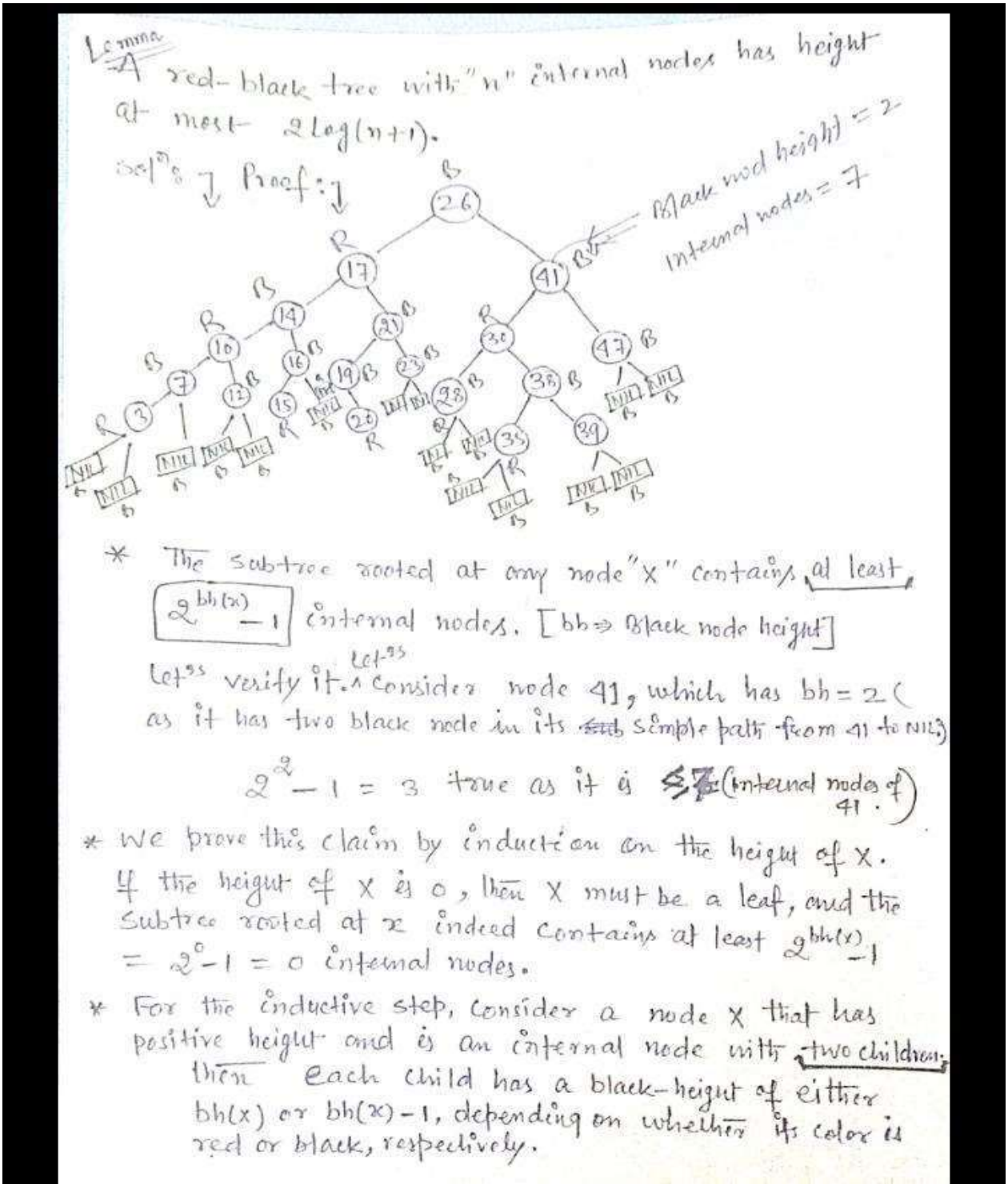


Remove DB (make it single)



resultant RB-tree

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: ↓

Black node height = 2
Internal nodes = 7

* The subtree rooted at any node "x" contains at least $2^{bh(x)} - 1$ internal nodes. [bh ⇒ Black node height]

Let's verify it. Let's consider node 41, which has bh = 2 (as it has two black node in its simple path from 41 to NIL)

$2^2 - 1 = 3$ true as it is ≤ (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 (any node) is 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

$$\left(2^{bh(x)-1} - 1\right) + \left(2^{bh(x)-1} - 1\right) + 1 = 2^{bh(x)} - 1 \text{ internal nodes.}$$

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 left-hand side and taking logarithms on both sides yields:

$$\log(n+1) \geq h/2 \qquad [\because \log_2 2 = 1]$$

or $\boxed{h \leq 2\log(n+1)}$ proved.

Lecture 20

by S. khan

B-Trees :

B-trees are balanced search trees designed to work well on disks or other direct access secondary storage devices. Many database systems use B-trees or variants of B-trees to store information.

Definition :

B-tree is an "m-way search tree" with the following restrictions:

1. The root node must have at least two children. It means that it can have even one key.
2. Every node except root (leaf) nodes must have at least $\lceil \frac{m}{2} \rceil$ children.
3. All leaf nodes must be at the same level.

/* m-way search trees */

It has the following properties:

(a) m (degree/order) $\Rightarrow$ max^m number of children. (Child pointer)

(b) m − 1 keys $\Rightarrow$ At most

(c) Keys are in the increasing order.

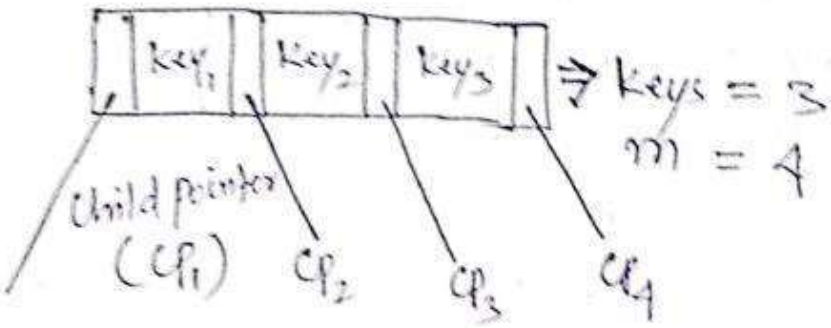
Example 01 : 4- way search tree

$m = 4$ (child pointers)

$key = 3$ $(m-1)$

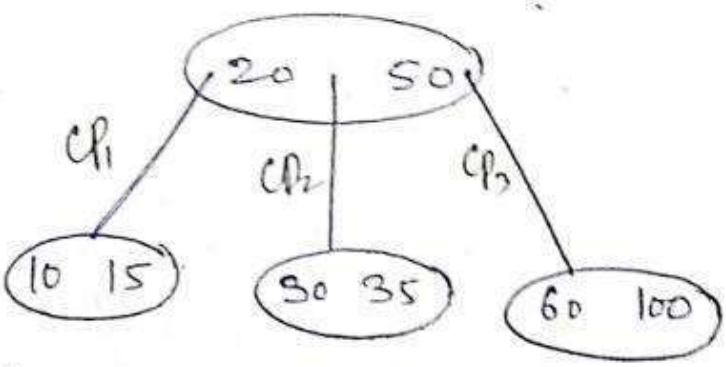
At most.

Node structure :



Example 02 :

3- way search tree (3-way ST)



An m-way ST does not have control over its construction as it can grow till "n" height. It can have minimum even one key in each node leading to a tree having height "n" that's why we moved to B-tree, which is an m-way tree with some restrictions to keep its height under check.

Insert the following key: 12, 21, 41, 50, 60, 70, 80, 30, 36, 6, 16 into an empty B-tree with degree 4 (max^m).

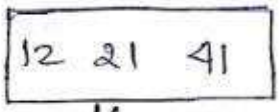
Sol^n:

Given.

Max Degree (m) = 4
Keys = m-1 = 3

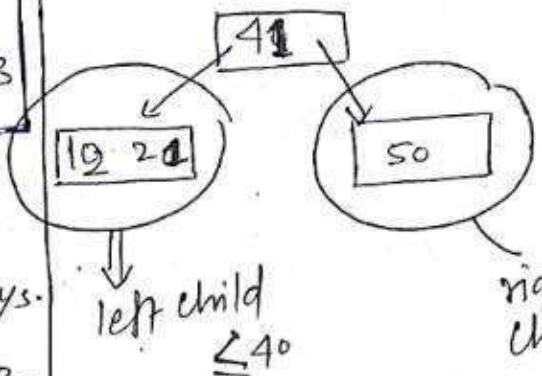
↳ 9t means each node can have four children and three keys. (max^m)

Step1: We can insert 12, 21 and 41 in the first node as it can have three keys.

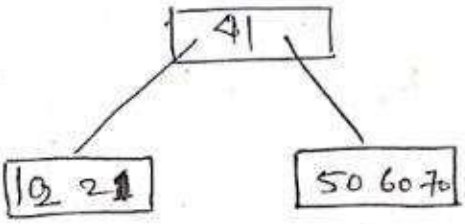


↳ Each key must be arranged in ascending order only.

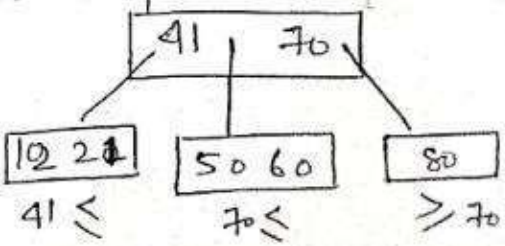
Step2: Now, the next element 50 cannot be inserted in the first node as it has already reached its maximus capacity of holding three keys. In such a case, we have to split up the node and send one of the keys on the root.



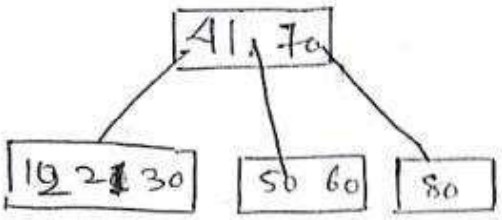
Step3: Now, insert the next elements 60 and 70.



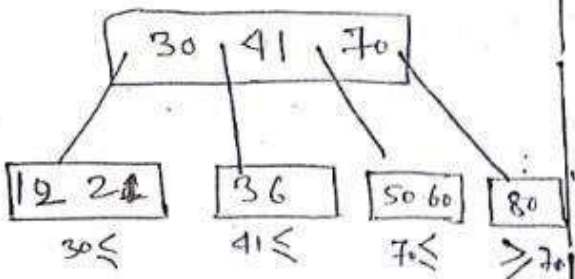
Step4: Now, the next element 80 cannot be inserted in the right ~~child~~ node; therefore, split it up.



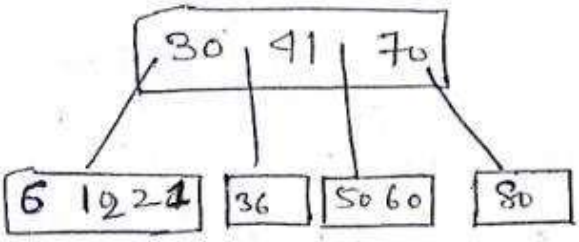
Step 5: Now, insert the next element 30. As it is less than 41, it will be inserted in the first child node.



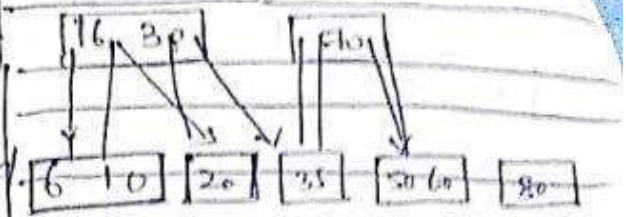
Step 6: Now, the next element 36 cannot be inserted in the first child node, so split it up.

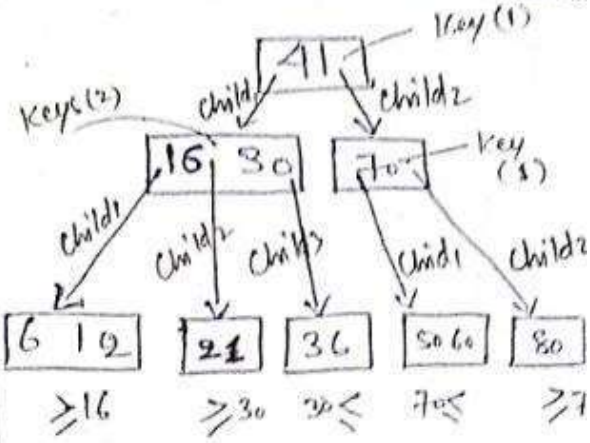


Step 7: Now, insert 6.



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.

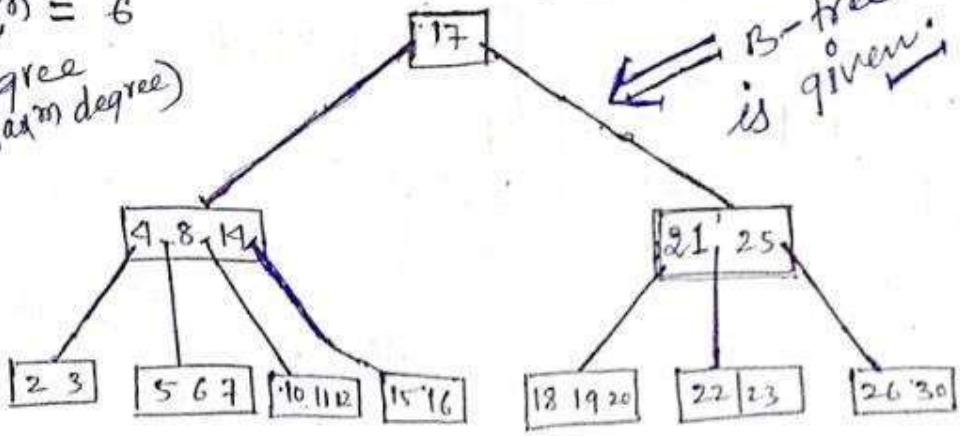
** For $m = 6$, we have

	min^m	max^m
Key	$\lceil \frac{m}{2} \rceil - 1$ (2) [Except root]	$m-1 = (5)$
cp	$\lceil \frac{m}{2} \rceil = \lceil \frac{6}{2} \rceil$ (3)	$m = 6$

cp → Child pointer

Deletion in B-trees

Delete the following keys: 7, 14, 8, 5, 3 where $m = 6$ / degree (Max^m degree)



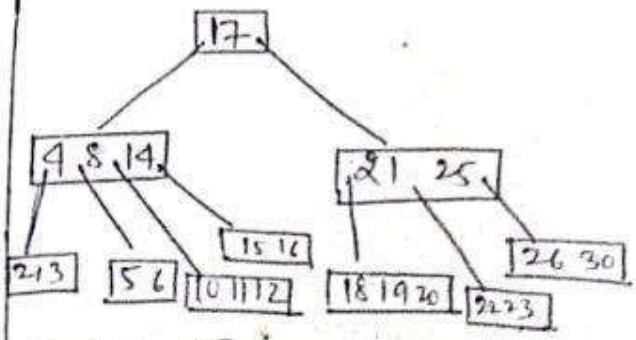
B-tree is given.

Sol^n : For m (max^m degree) $= 6$, we have the following properties:

	min^m	max^m
Key	$\lceil \frac{m}{2} \rceil - 1$ = (2) [except root]	$m-1$ (5)
Children (CP)	$\lceil \frac{6}{2} \rceil$ = (3)	m = (6)

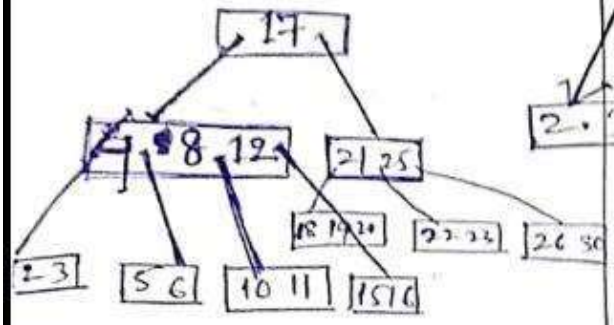
Delete (7)

We can easily delete 7 from the given B-tree without violating the properties of B-tree. Minimum key except root node should be (2).



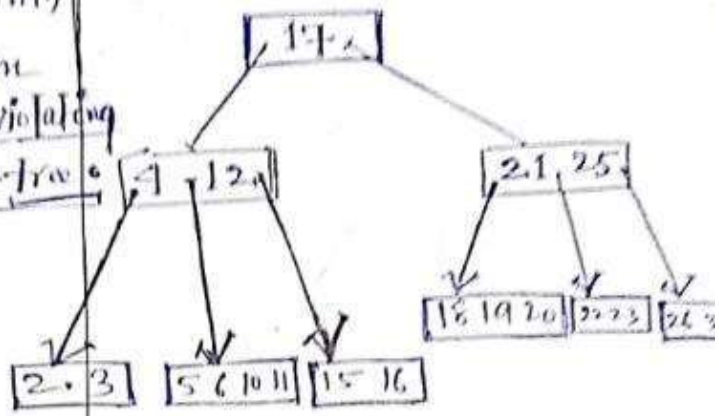
Delete (14): we cannot directly delete a non-leaf node key. We need to swap the keys of the node to be deleted and

its in-order predecessor (largest key value of the left sub-tree) = 10
14 will be replaced with 12, and then 12 can be deleted without violating the properties of B-tree.

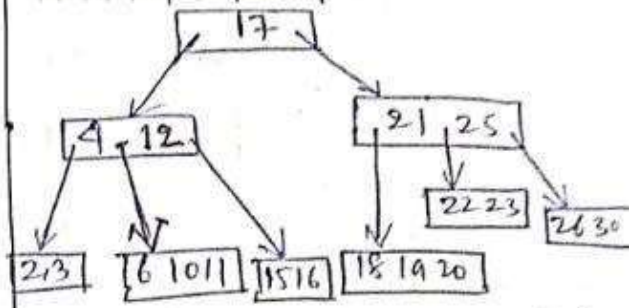


Delete (8)
we cannot ~~a non-~~ leaf directly delete a non-leaf key, but we can replace it with its inorder ~~fore~~ predecessor or inorder successor. ~~If~~ we ~~do~~ cannot replace 8 with either its inorder predecessor (10) or successor (12) as it will violate minimum key property, which must be (2) in this B-tree.

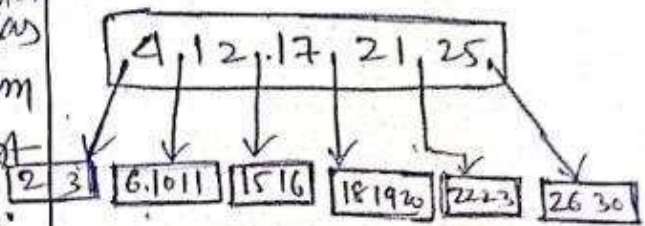
In such a case, we del. the key and merge with of its children.



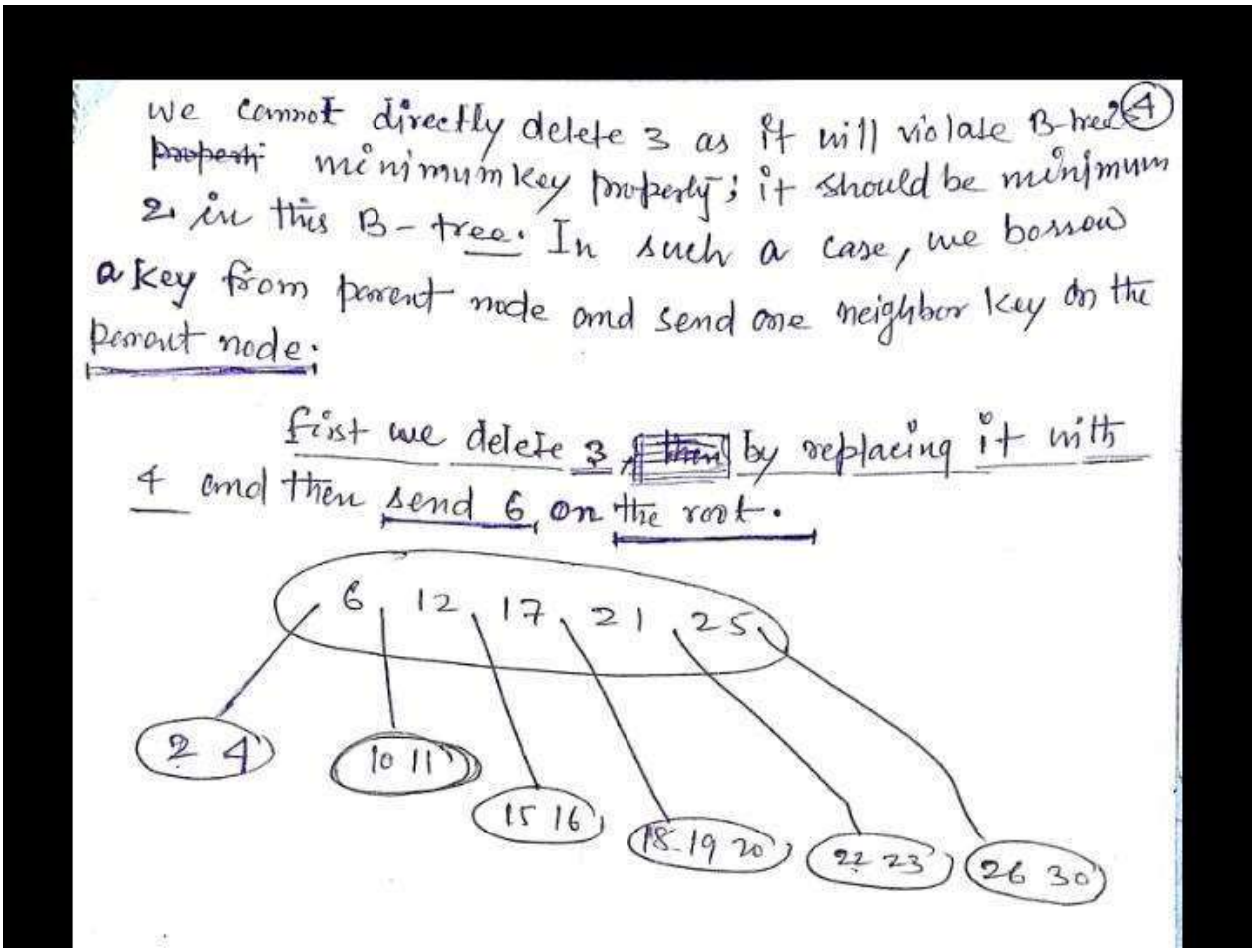
Delete (5) : we can easily delete it as it is in the leaf node and does not violate min key property.



Now, check if we can shrink this B-tree. If yes, do it.



Delete (3)

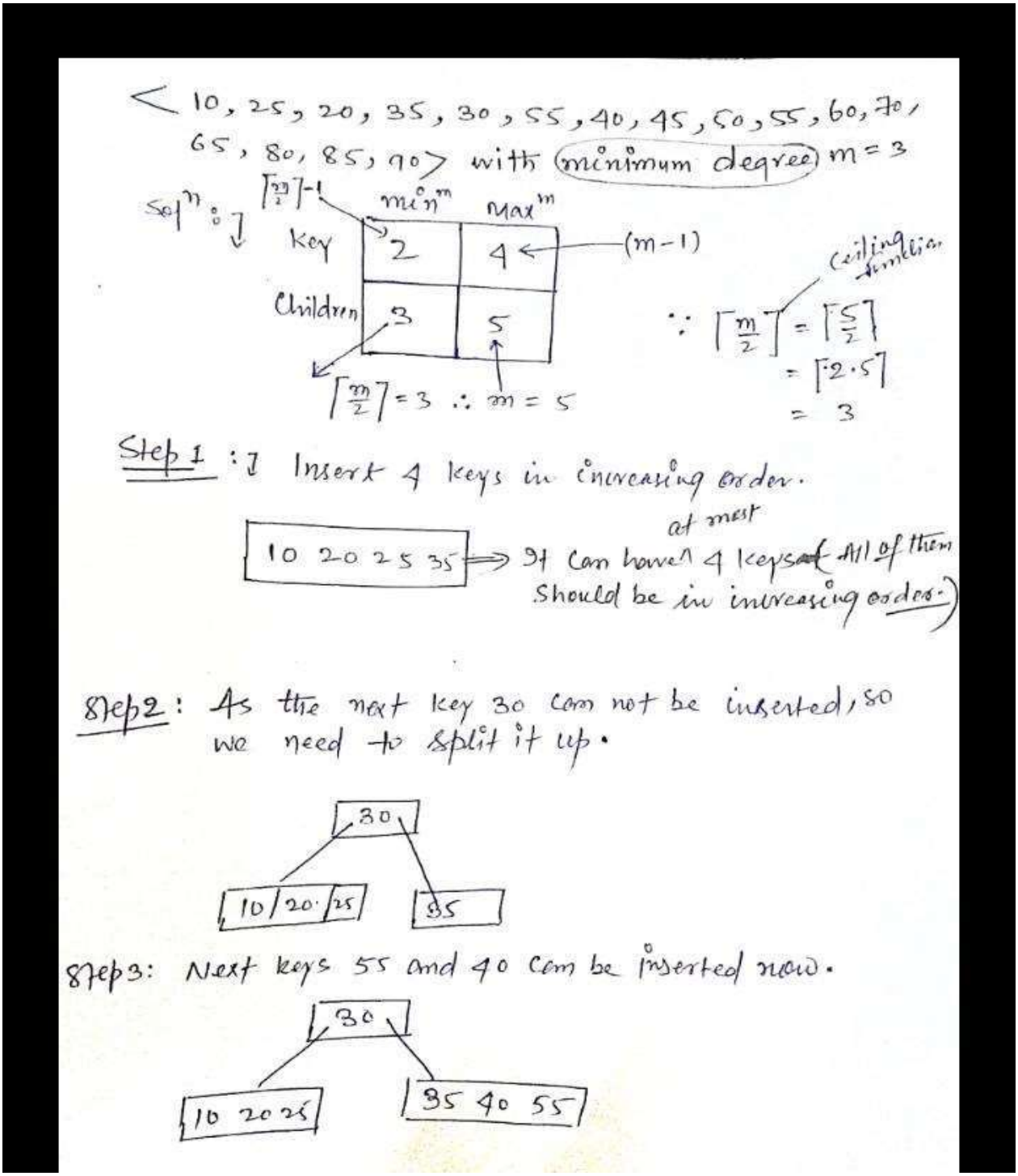


Q6. Define B-Tree with its properties.

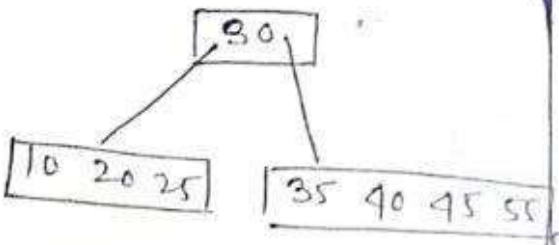
B tree is an m-way search tree with the following restrictions:

1. The root node must have at least two children.
2. Every node except root and leaf nodes must have at least ⌈ m/2 ⌉ children.
3. All leaf nodes must be at the same level.
4. Creation process is bottom up.

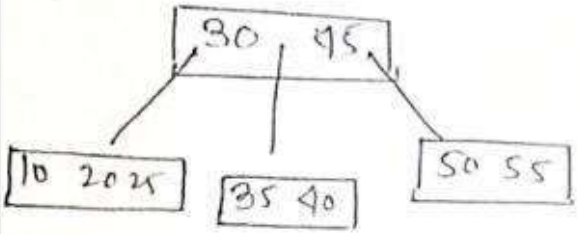
Q6. Using the minimum degree m= 3, insert the following sequence of integers 10, 25, 20, 35, 30, 55, 40, 45, 50, 55, 60, 75, 70, 65, 80, 85,90 in an initially empty B-Tree.



Step 4: The next key is 45, which can be inserted.
* Max^m keys capacity for a node is (4).

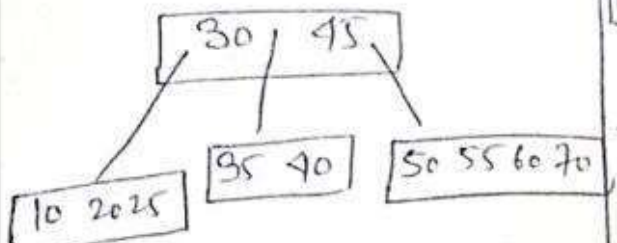


Step 5: The next key is 50. Split right child.

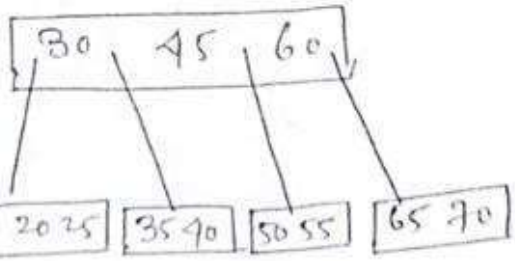


Step 6: The next key is 55, which is duplicate, not allowed.

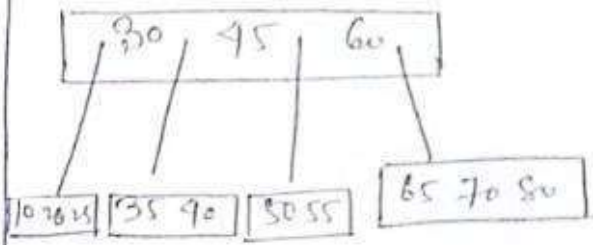
Step 7: The next two keys 60 and 70 can be inserted.



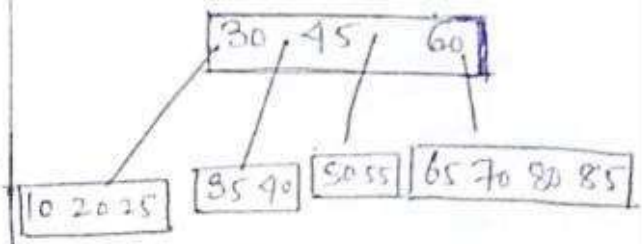
Step 8: The next key is 65. Split node.



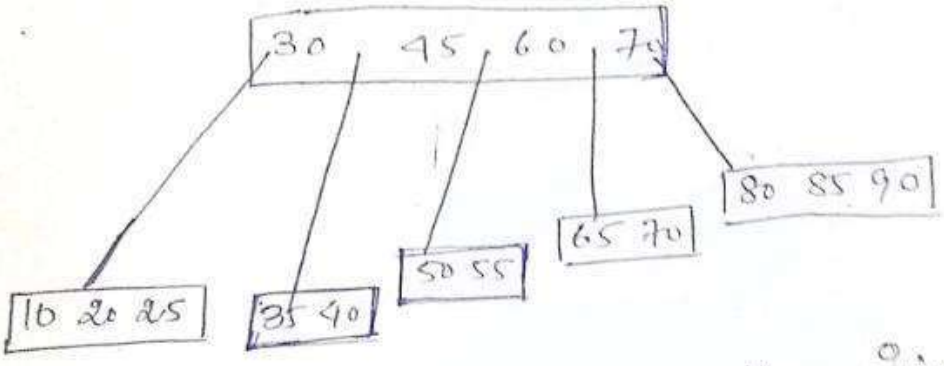
Step 9: The next key is 80. Insert it.



Step 10: The next key is 85. Insert it.



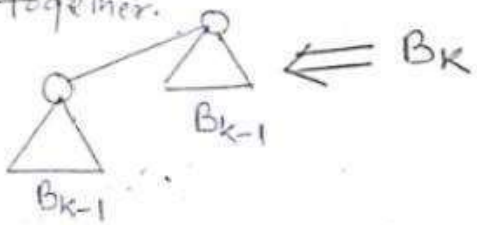
Step 11: The next key is 90, which cannot be inserted as one node can have max^m 4 keys only according to the question, so split it up.



Resultant B-tree with minimum degree 3 i.e max^{m} 5.

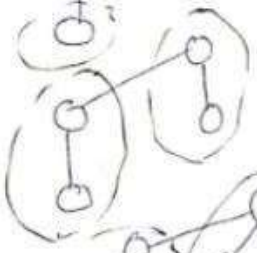
Q7. Define Binomial Tree and mention its properties.

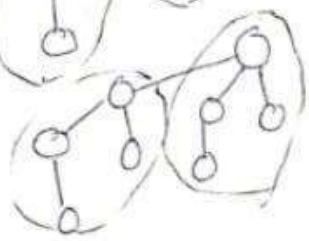
* The Binomial tree "B_k" is an ordered tree defined recursively.
* The Binomial tree B_0 consists of a single node.
* The Binomial tree B_k consists of two Binomial trees B_{k-1} that are linked together.



$B_0 \Rightarrow$ ○ (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)

.
.
.

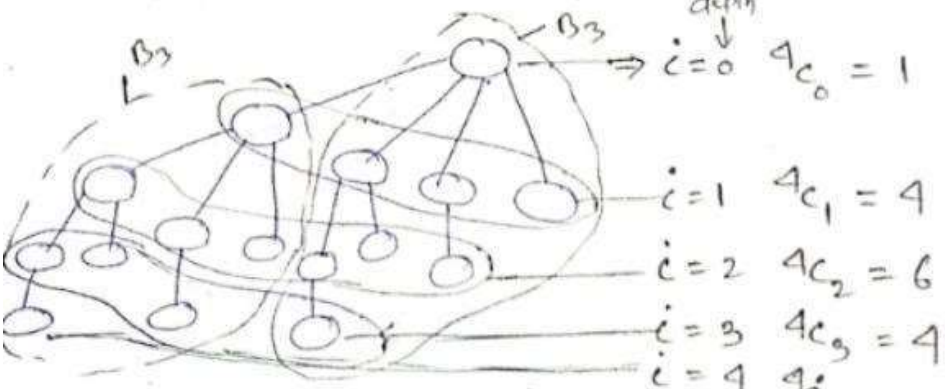
$B_k \Rightarrow$ (Consists of two B_{k-1} Binomial trees)

Properties of Binomial tree (B_k) :

1. There are 2^k nodes.
2. The height of tree is k.
3. There are exactly ^kC_i nodes at depth i for i = 0, 1, 2, ..., 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, ... 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 ⇒ It consists of two B_3 Binomial trees:



1. 2^k nodes = 2^4 = 16
2. Height = 4
3. ^kC_i nodes at depth i for i = 0, 1, 2, k

$$^kC_i \rightarrow \frac{k!}{i!(k-i)!}$$

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 subtree, 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

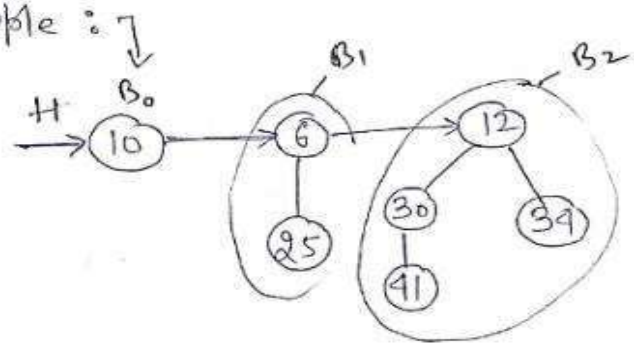
A Binomial heap "H" is a set of Binomial trees that satisfies the following Binomial heap properties:

(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:

$B_0 = 0$
$B_1 = 1$
$B_2 = 2$

⇒ increasing order

12:04 PM

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.

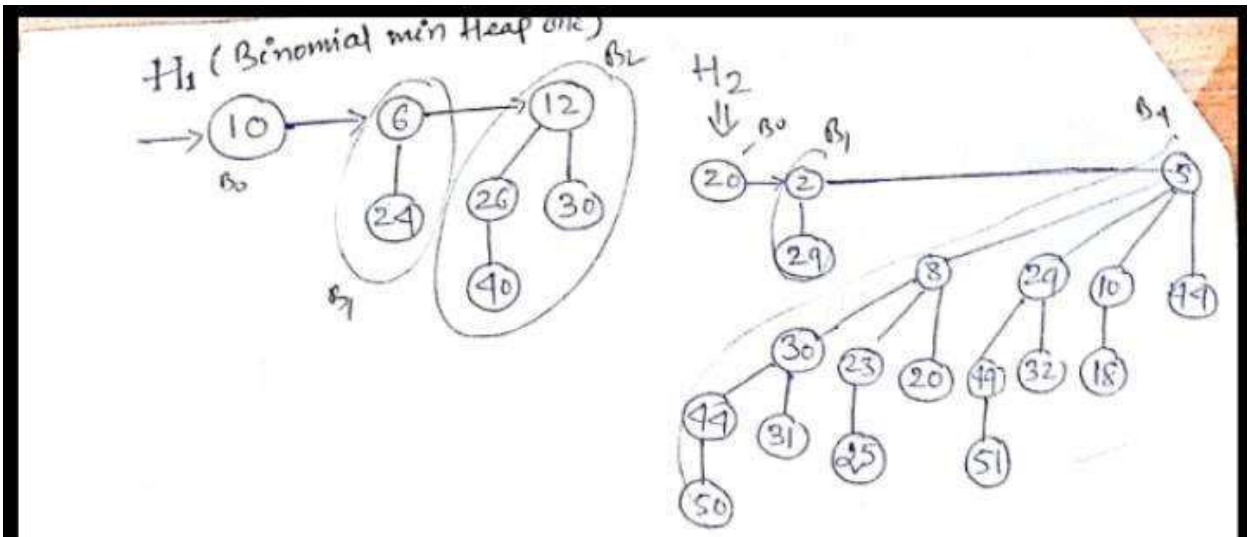
   (a) If (degree[x] ≠ degree[next-x] or
   degree[x] = degree[next-x] = degree[sibling[next-x]])
   , then move the pointers one position right.

   (b) If (degree[x] = degree[next-x] ≠ degree[sibling[next-x]])

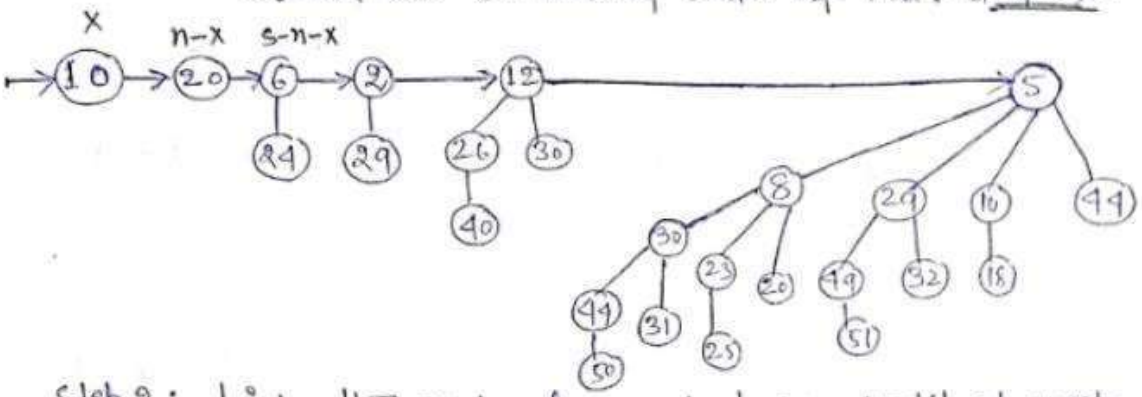
   (i) If key(x) < key(next-x), then make (next-x) as left most child of x.

   (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.

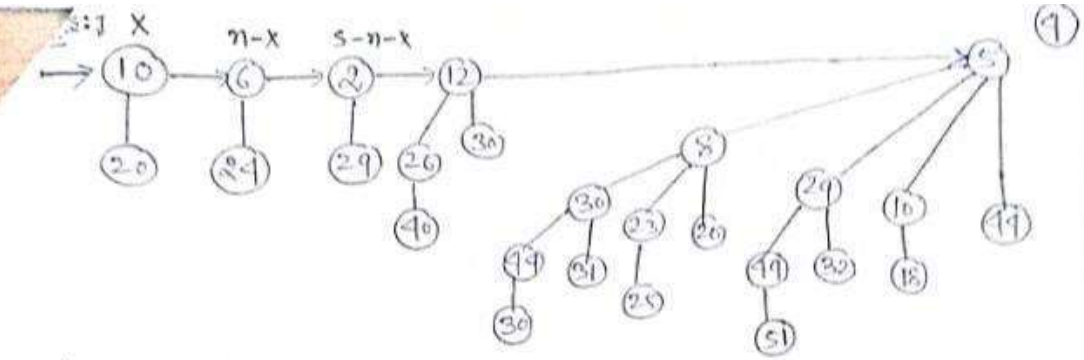
degree of x = 0
degree of next of x = 0 $\quad 0 = 0 \neq 1$
degree of sibling of next of x = 1

(b) If (degree [x] = degree [next-x] $\neq$ degree [sibling[next-x]])

$\Downarrow$ True.

(i) If key(x) < key(next-x) True
10 20

make (next-x) as left most child of x.



Degree of x = 1
Degree of n-x = 1
Degree of s-n-x = 1

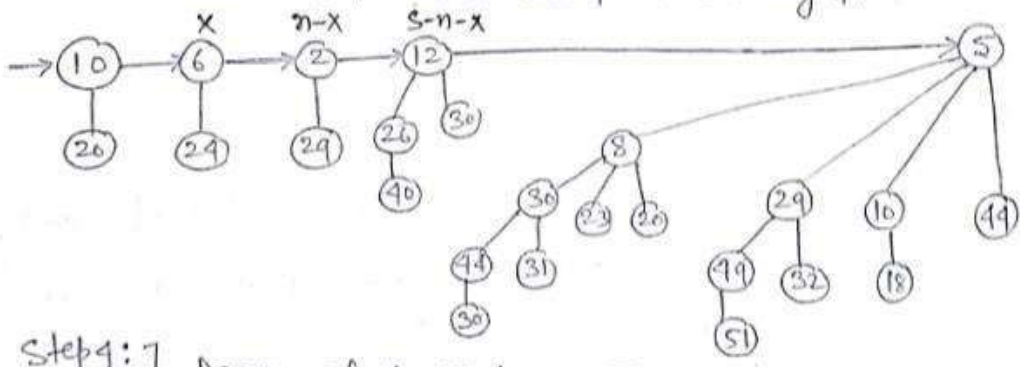
1 = 1 = 1

↳ Degree of these three Binomial heaps are same.

(2)(a) if (degree[x] ≠ degree[next-x] or
degree[x] = degree[next-x] = degree[sibling[next-x]])

⇓ True.

move the pointers one position right.

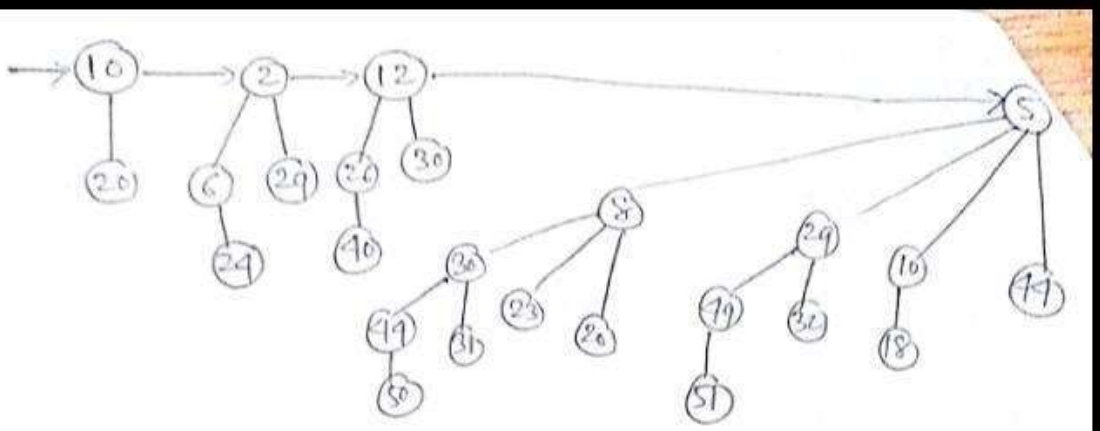


Step4: ↓ Degree of x = 1
Degree of n-x = 1
Degree of s-n-x = 2

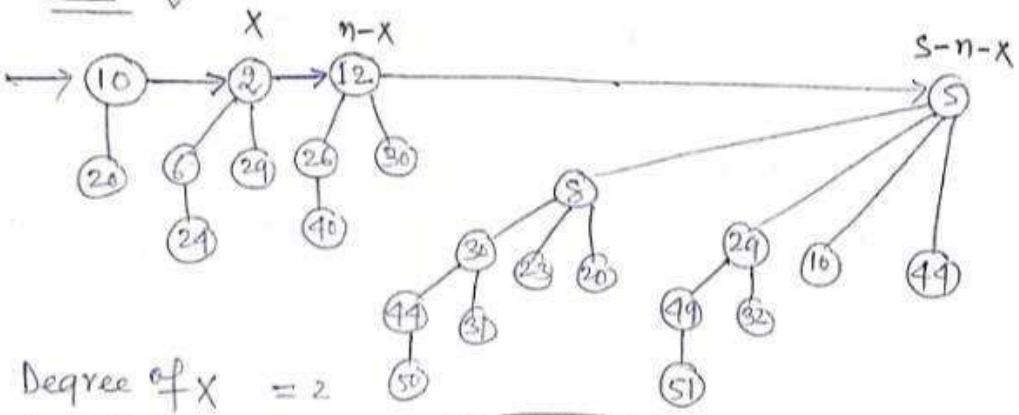
1 = 1 ≠ 2

(2)(b) 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 :↓

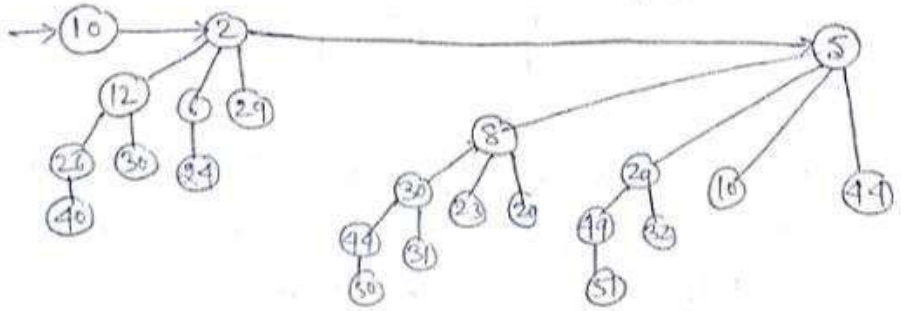


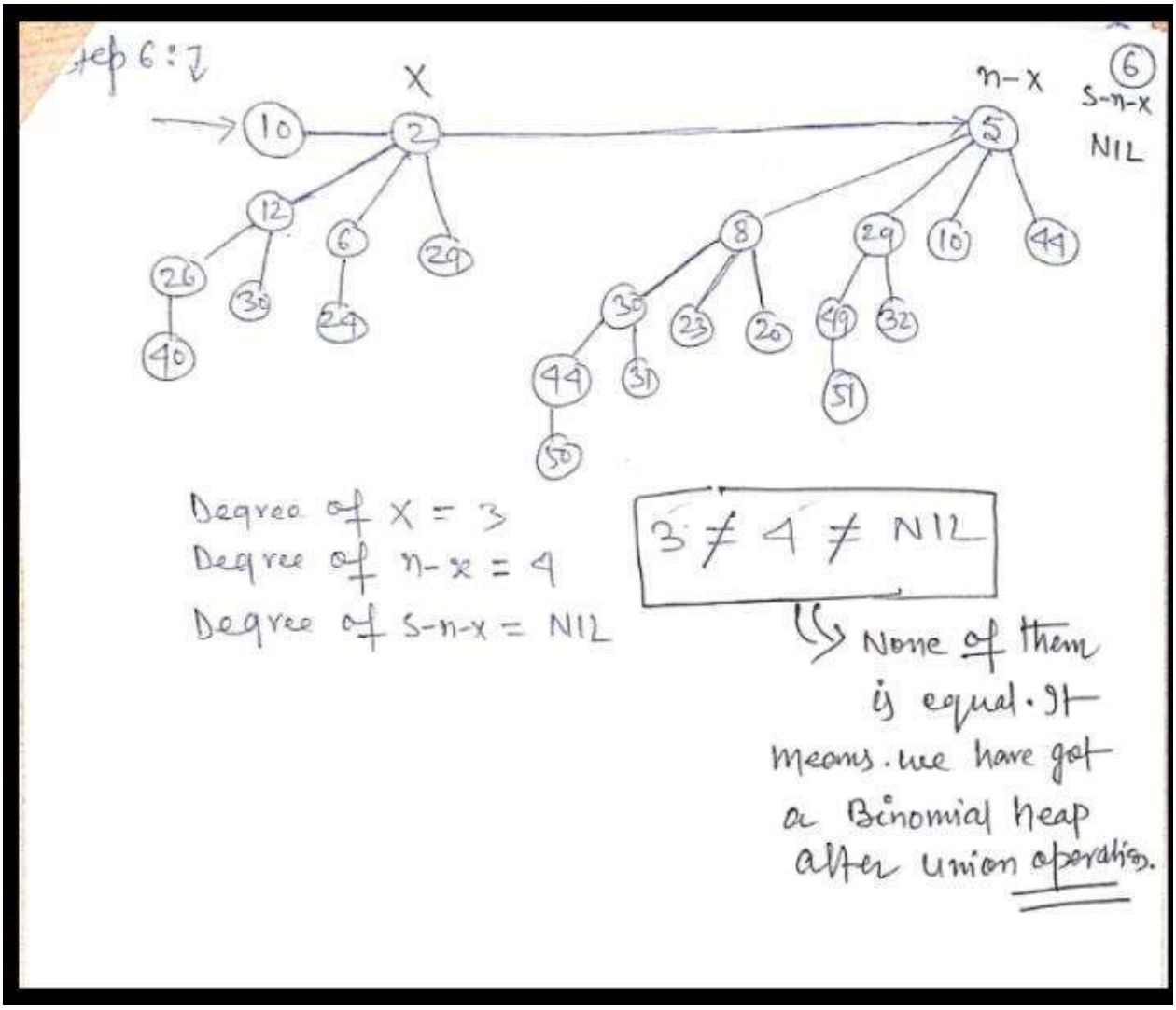
Degree of $x = 2$
Degree of $n-x = 2$
Degree of $s-n-x = 4$

$2 = 2 \neq 4$

(2)(b) 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. Draw the 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'

H

(7)

⇓ 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)

H B_0 (7) H1 B_0 (2)

⇓ Perform union

Merge the root lists of Binomial min heaps

(7) B_0 → (2) B_0

As degrees are same, make it min heap.

(2) → (7)

Binomial heap having one Binomial tree of degree one (B_1)

Now, insert 4.

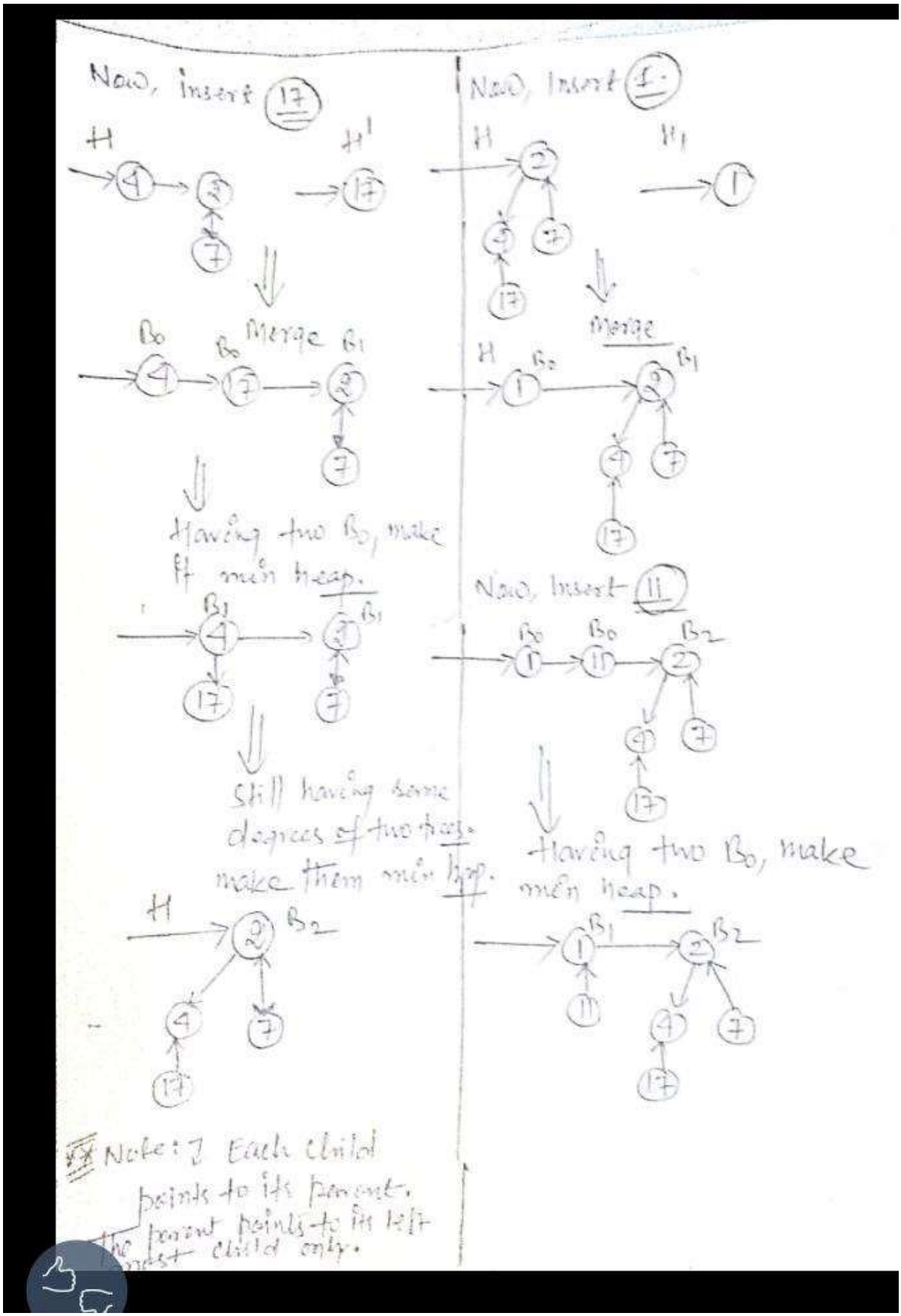
H: (2) → (7) H1: (4)

⇓

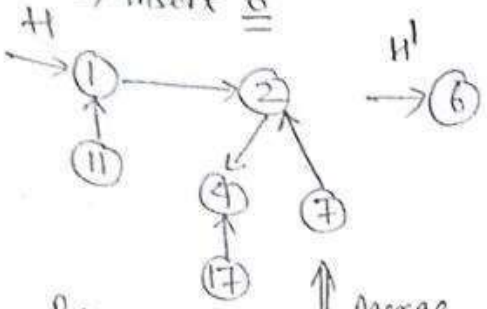
Merge the root lists of Binomial min heaps in the increasing order of their degrees.

(4) B_0 → (2) B_1, (7)

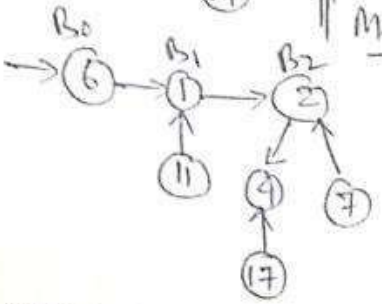
Binomial heap Containg 2 Binomial trees B_0 & B_1


Now, Insert 17
H
H'
Merge
B0
B1
Having two B0, make it min heap.
Still having some degrees of two trees. make them min hpp.
B2
Now, Insert 1.
H1
Now, Insert 11
Having two B0, make min heap.
Note: 1 Each child points to its parent.
the parent points to its left most child only.

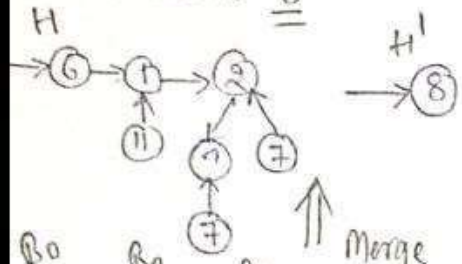
Now, Insert 6



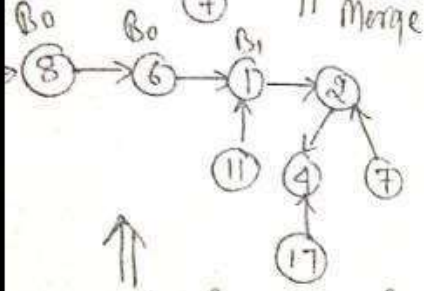
Merge



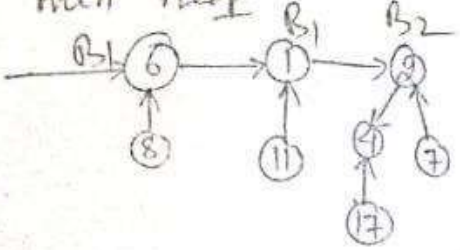
Now, Insert 8



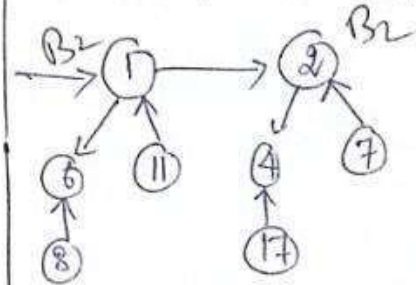
Merge



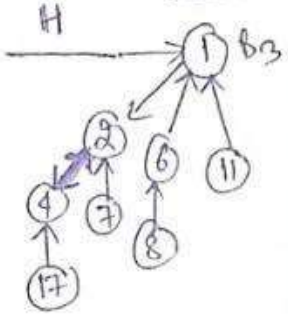
Having two B_0s, make min heap.



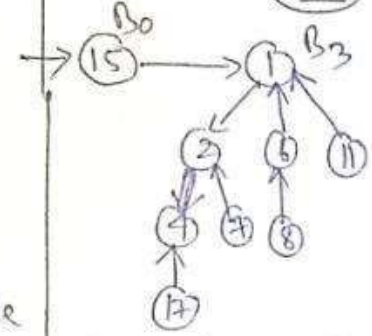
Still having two B_1, make Binomial min heap.



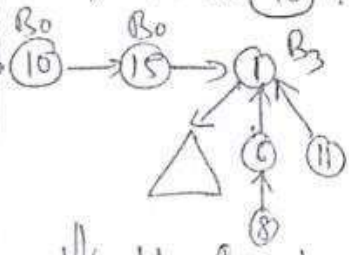
Still having two B_2, make ~~he~~ min heap.



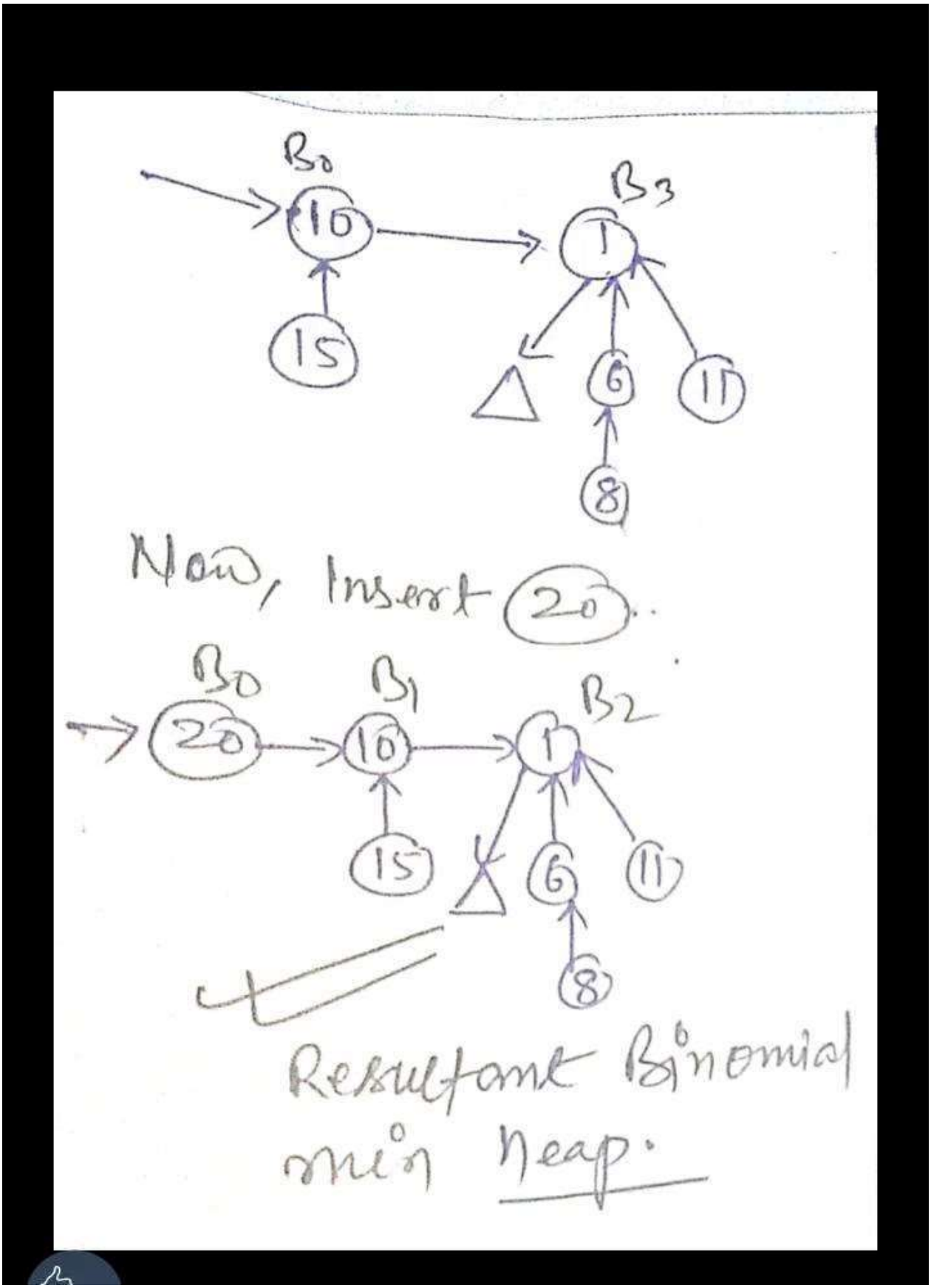
Now, Insert 15.



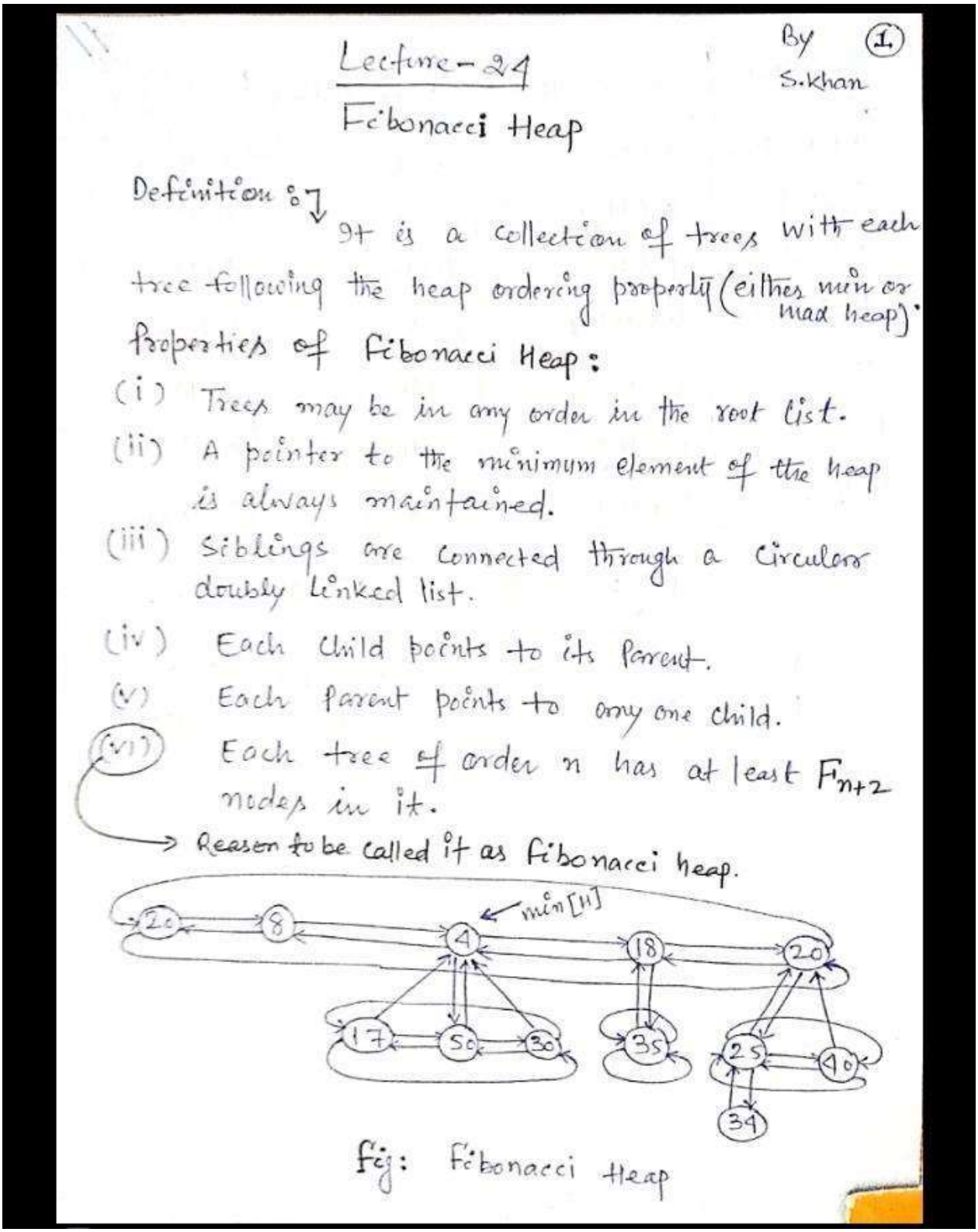
Now, Insert 10.



Having two B.


B0
10
15
B3
1
6
11
8
Now, Insert 20..
B0
B1
B2
20
10
1
15
6
11
8
Resultant Binomial min heap.

Q10. Define Fibonacci heap and also compare the complexities of Binary heap, Binomial heap and Fibonacci heap.



Lecture – 24

By (1) S. Khan

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.

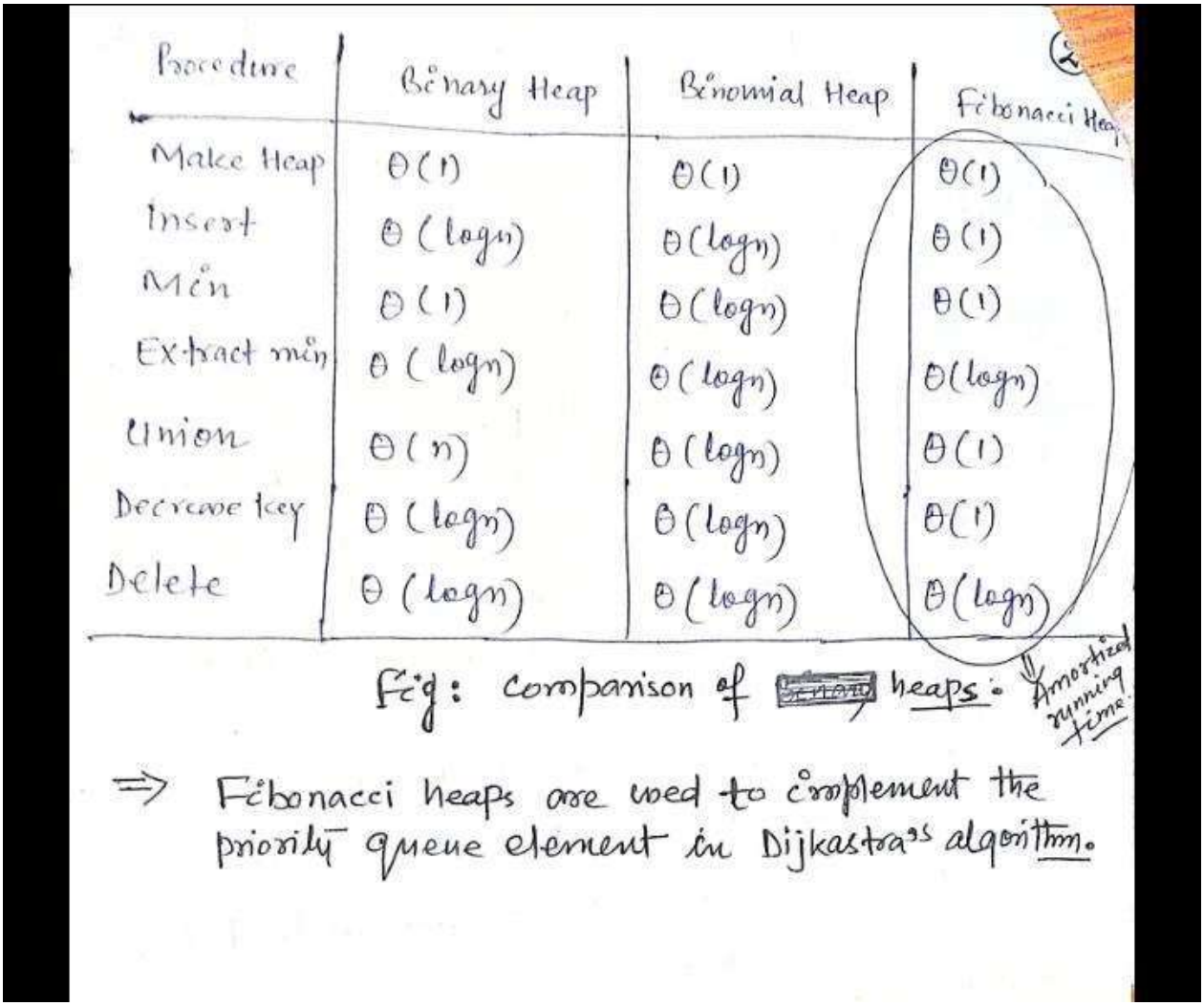
(iv) Each child points to its Parent.

(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



Q11. Explain the extracting minimum node operation of Fibonacci heap with example.

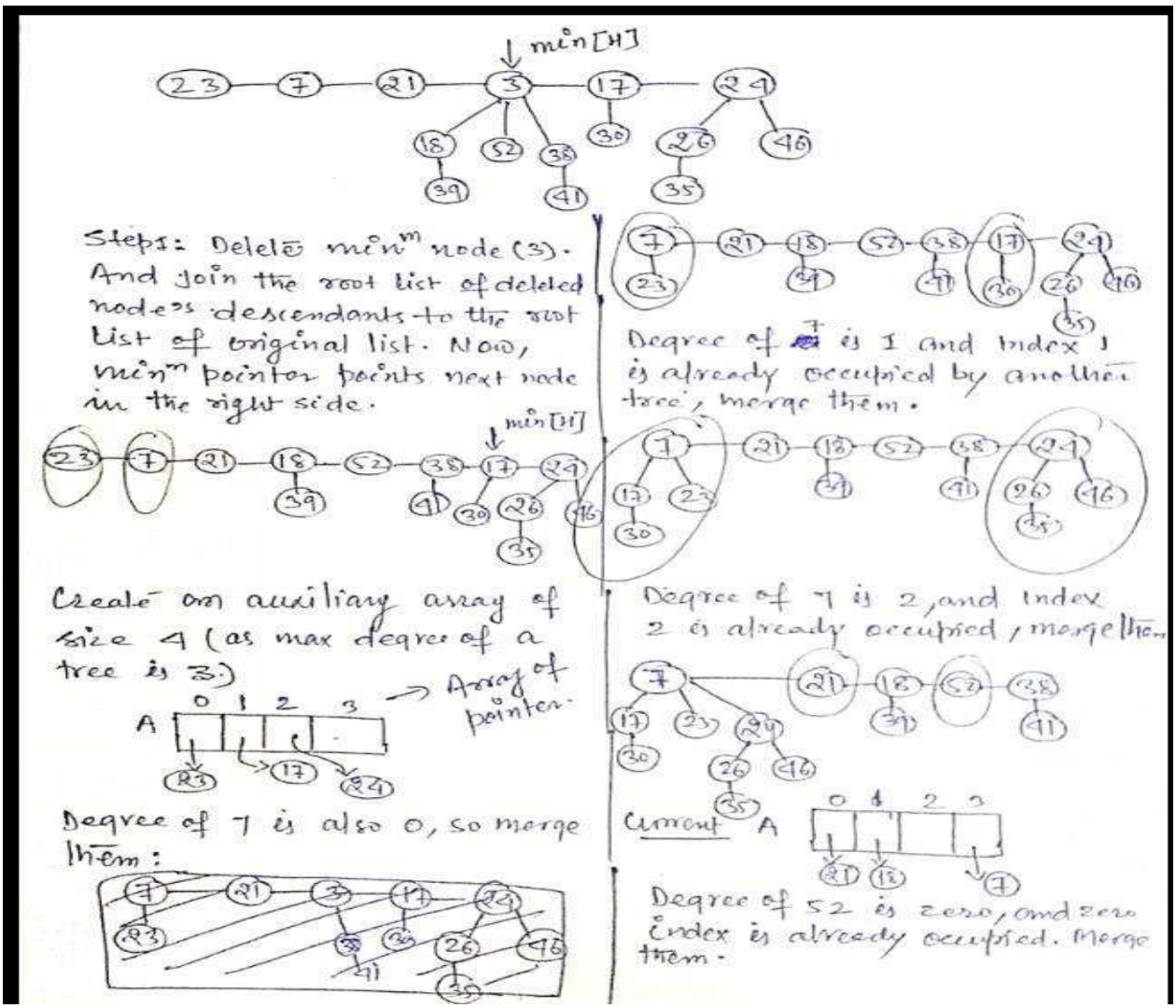
This operation is accomplished by deleting the minimum key node and then moving all its children to the root list. It uses the process called "consolidate" to merge the trees having same degree.

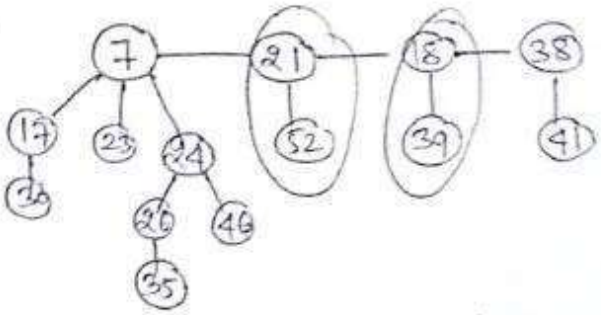
Steps:

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.

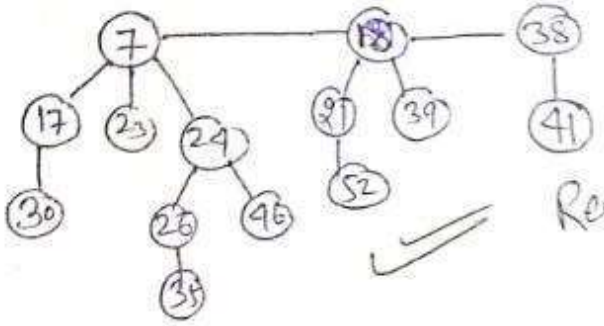
4. Stop after having every tree with unique degree.

Example,





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

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.

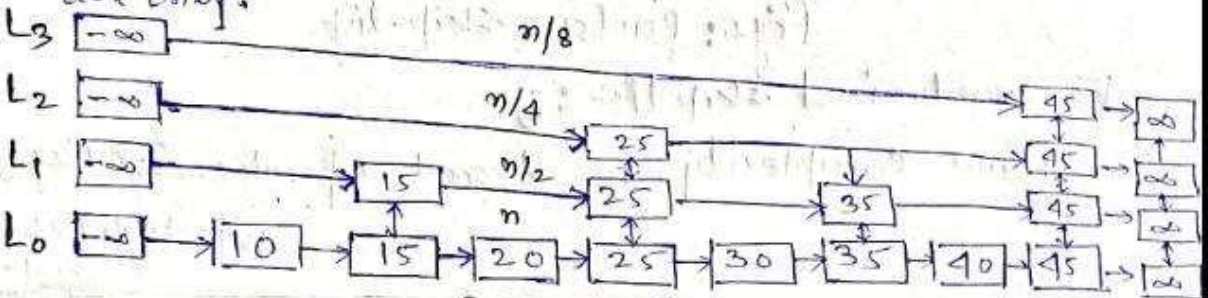


Fig1: A Perfect skip list

Number of levels in skip list ≈ $\log_2 n$

Time complexity for the search operation = $O(\log n)$

(2 * no of levels)

Perfect skip list : A skip list where we promote alternate elements and has $\log_2 n$ levels is called a perfect skip list. Fig 1 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.

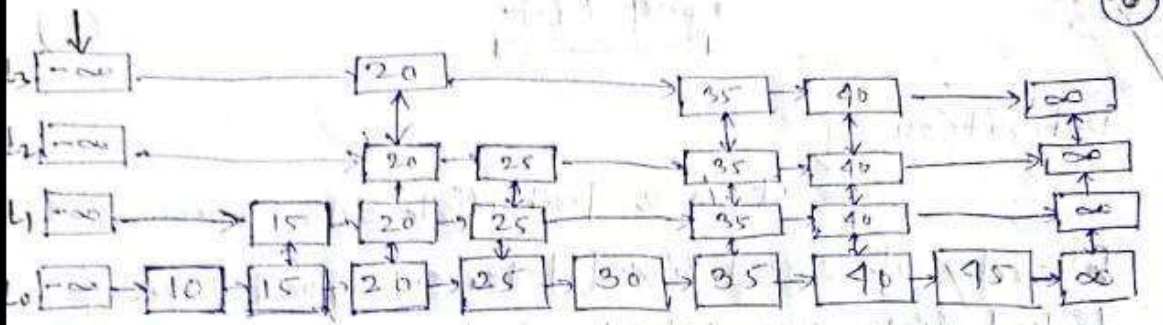


Fig 2: Random skip-list.

For randomized skip list: ↓

Time complexity for search operation ≅ O(log n) with high probability.

Time complexity to insert an element in an 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)

Trie
(Digital tree / Prefix tree)

By S. Khan (2)

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

$\Rightarrow O(L)$ where 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>.

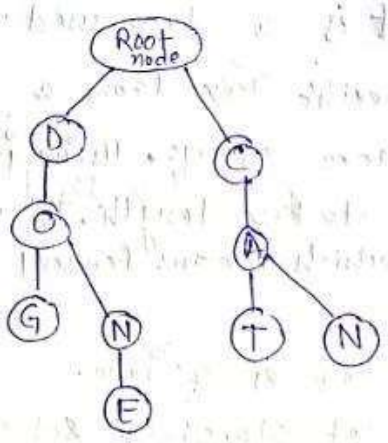


fig: Trie for the given strings.

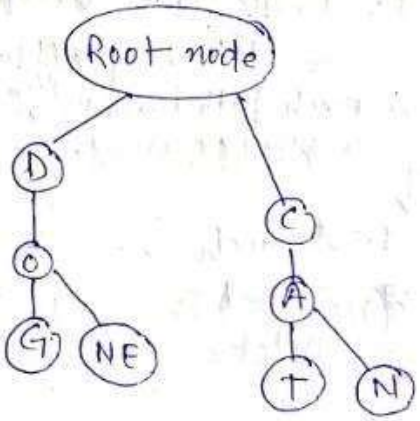


fig: compressed Trie

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:

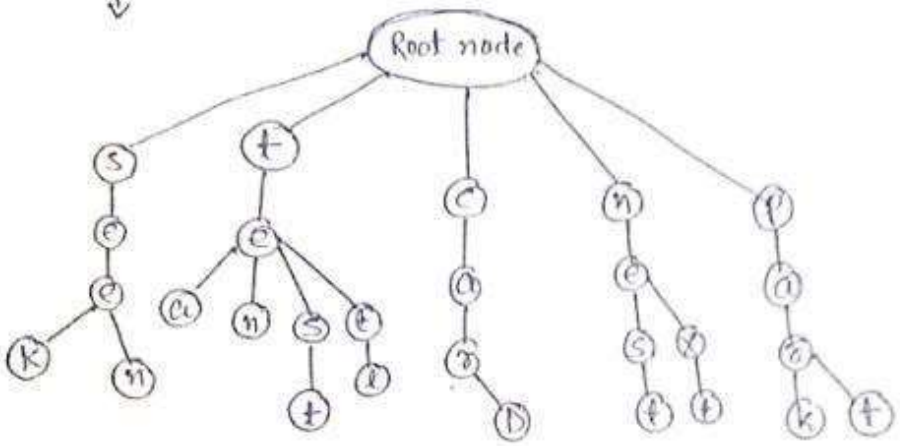


fig 1. Trie

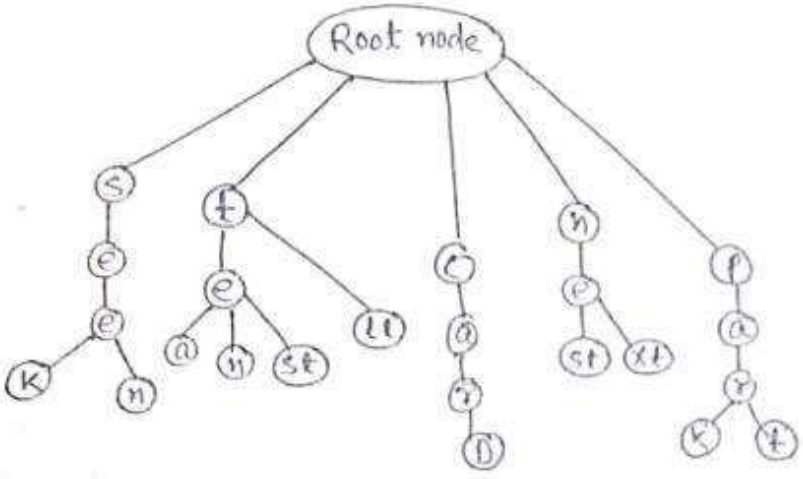


Fig 2: Compressed Trie

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.

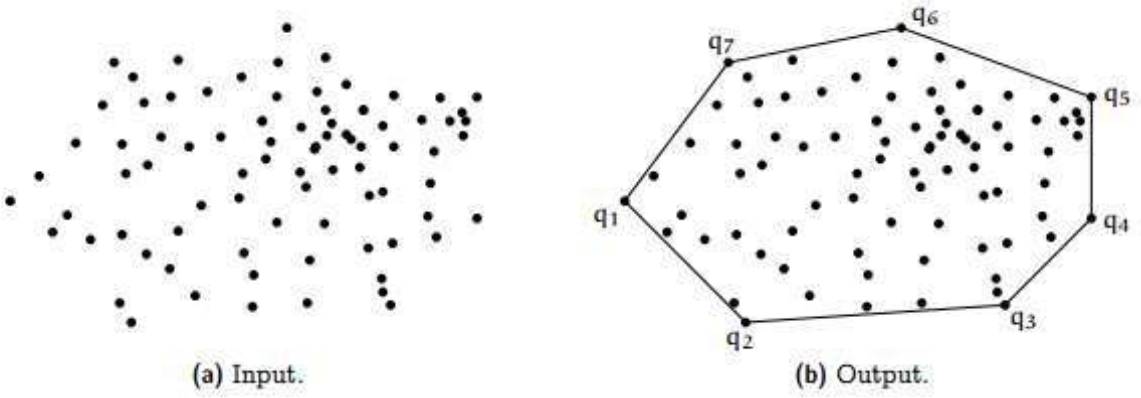
Combine: When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution to the original problem.

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. Closest pair of points
5. Convex Hull algorithm

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.



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 on the next page.

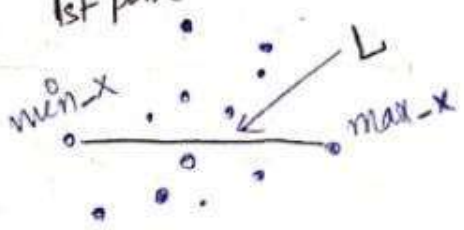
Example :↓ Find a convex hull from a set of points given below:

Solⁿ :↓

Step 1 : Find the point with minimum x-coordinate (min_x) and similarly the points with maximum x-coordinate (max_x)

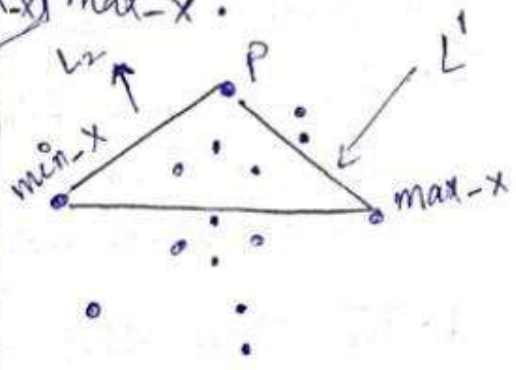
Step 2 : Make a line (L) joining these two points.

1st part

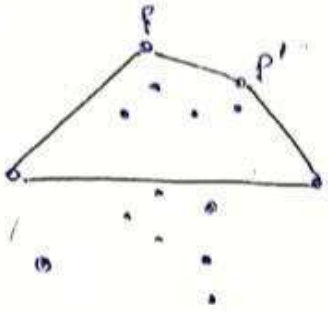


2nd part :

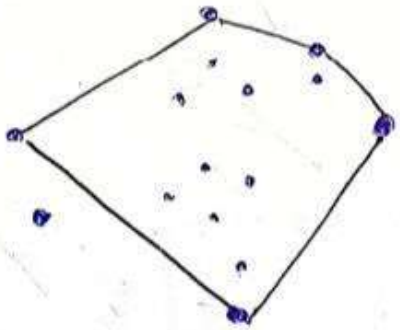
Step 3 : For the 1st part, find the point P with maximum distance from the Line L. P form a △ with min_x and max_x.

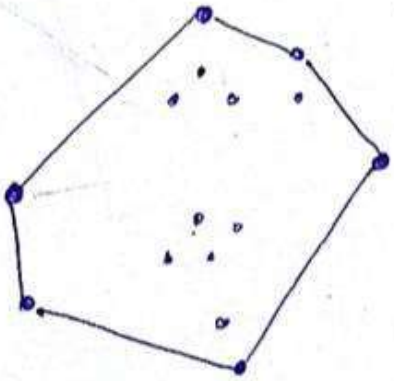


Step 4: Now, we have two new Lines : L_1 and L_2.
Now repeat the Line-3.



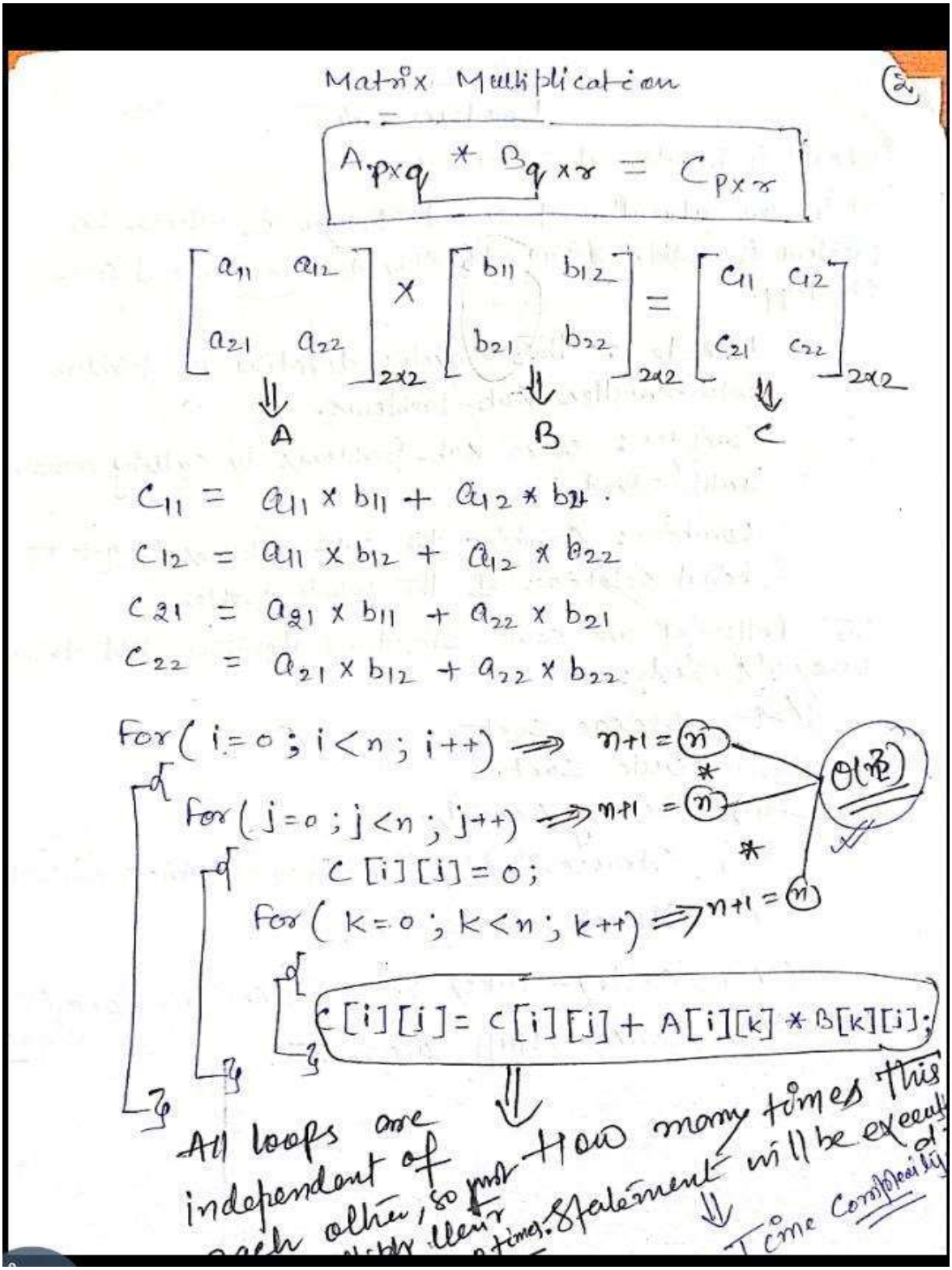
Step 5: Do the same with the second part.



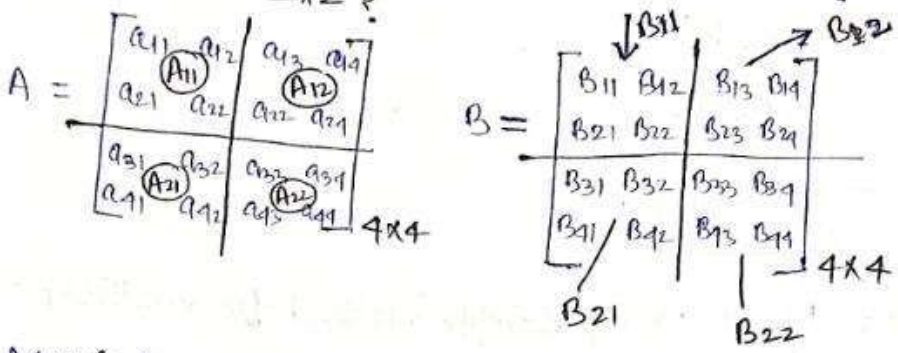


↳ convex hull

Matrix Multiplication



How can we apply Divide and Conquer technique to solve two matrices' multiplication of order more than 2x2 ?




```
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)
    }
}
```

$$T(n) = \begin{cases} 8T(n/2) + n^2 & n > 2 \\ 1 & n \leq 2 \end{cases}$$

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 Dec, we get $\Theta(n^3)$ running time.

/* Strassen's Matrix Multiplication */

$P = (A_{11} + A_{22}) * (B_{11} + B_{22})$

$Q = (A_{21} + A_{22}) * B_{11}$

$R = A_{11} * (B_{12} - B_{22})$

$S = A_{22} * (B_{21} - B_{11})$

$T = (A_{11} + A_{12}) * B_{22}$

$U = (A_{21} - A_{11}) * (B_{11} + B_{12})$

$V = (A_{12} - A_{22}) * (B_{21} + B_{22})$

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$

$$T(n) = \begin{cases} 7T(n/2) + n^2 & n > 2 \\ 1 & n \leq 2 \end{cases}$$

$\Theta(n^{2.81})$

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:

$A_{11} = 1$, $A_{12} = 3$, $A_{21} = 7$, $A_{22} = 5$

$B_{11} = 6$, $B_{12} = 7$, $B_{21} = 3$, $B_{22} = 8$

$P = (A_{11} + A_{22}) * (B_{11} + B_{22})$

$\Rightarrow 84$

$Q = (A_{21} + A_{22}) * B_{11}$

$\Rightarrow 72$

$R = A_{11} * (B_{12} - B_{22})$

$\Rightarrow -1$

$S = A_{22} * (B_{21} - B_{11})$

$\Rightarrow -15$

$T = (A_{11} + A_{12}) * B_{22}$

$\Rightarrow 32$

$U = (A_{21} - A_{11}) * (B_{11} + B_{12})$

$\Rightarrow 78$

$V = (A_{12} - A_{22}) * (B_{21} + B_{22})$

$\Rightarrow 32$

$C_{11} = P + S - T + V$

$\Rightarrow 150$

$C_{12} = R + T$

$\Rightarrow 31$

$C_{21} = Q + S$

$\Rightarrow 57$

$C_{22} = P + R - Q + U$

$\Rightarrow 89$

$$C = \begin{bmatrix} 150 & 31 \\ 57 & 89 \end{bmatrix}$$

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.

Let's understand it with a suitable real-life example.

Suppose we want to travel from Delhi to Mumbai shown as 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.

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.

Q3. Describe optimization problem, feasible solution and optimal solution.

Optimization problem – An optimization problem is the problem of finding the best solution (either minimum or maximum) from all feasible solutions. 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 the problem's constraints."

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 an optimal solution to the fractional knapsack instances n=7 and knapsack capacity m=15 where profits and weights 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 increasing 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 2kg out of 3kg, 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

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.

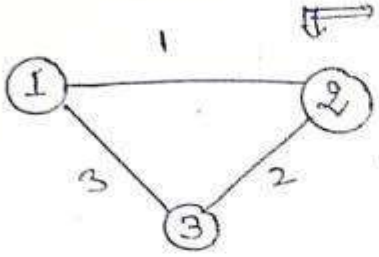
Minimum Spanning Tree (MST) – An MST for a weighted, connected, undirected graph is a spanning tree having a weight less than or equal to the weight of every other possible spanning tree.

Example:

⟹ Connected graph.

G:

Undirected ⟹ weighted graph



⟸ This is a labeled graph.

Fig 1: A complete graph with 3 vertices (K_3).

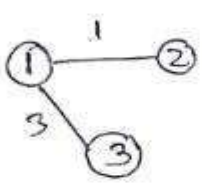
Vertices = {1, 2, 3}

Edges = {(1,2), (1,3), (3,2)}

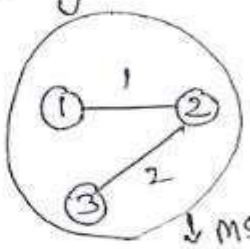
⟹ All possible spanning trees out of G.

As G has three vertices, every possible ST will have only two edges.

(A)



(B)



↓ MST

(C)

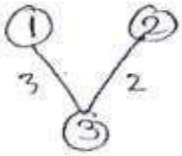


Fig 2: Three possible spanning trees out of G.

Note: For a Complete graph, if it has "n" vertices then we have n^{n-2} spanning trees.

(B) is the minimum spanning tree for G because its cost is least(3) among other STs.

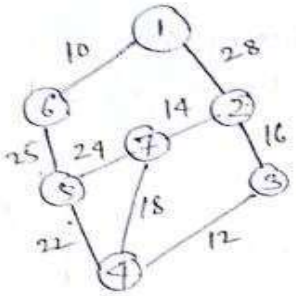
Prim's Algorithm for finding an MST

```
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,1] < 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.

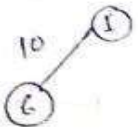
Example : (MST using Prim's algo)

$O(n^2)$ # vertices

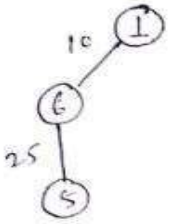


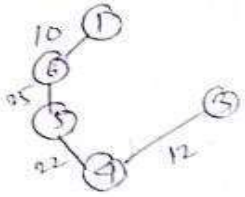
⟸ 7 vertices

Step 1 : Select minimum cost edge

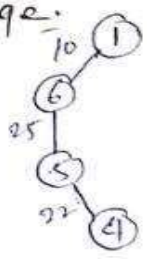


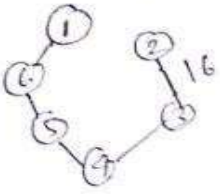
Step 2 : Select another minimum cost edge which is ~~already~~ connected to already selected vertices.

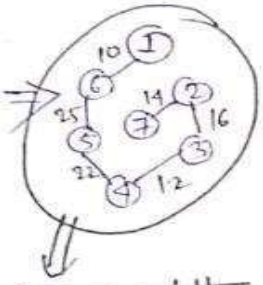




Follow Step 2 to select the next minimum cost edge.







Cost 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)

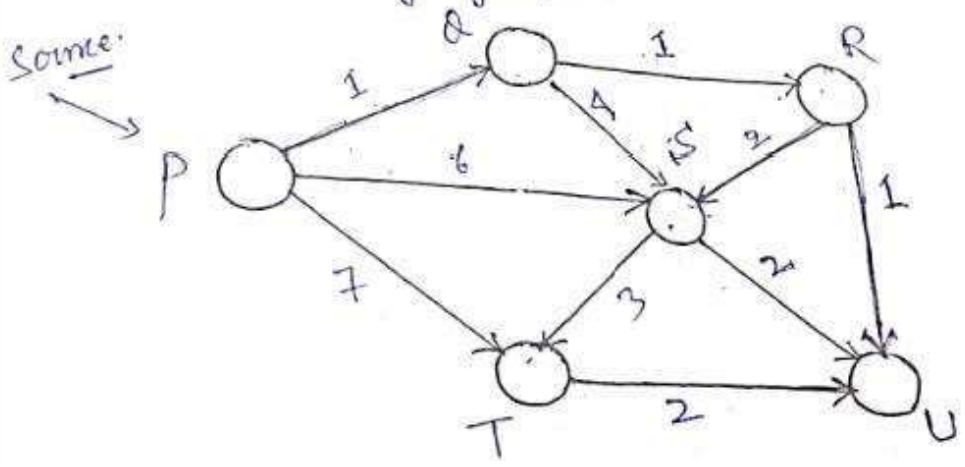
Given a graph G = (V, E), we want to find a shortest path from a given source vertex s ϵ V to each vertex v ϵ V.

1. Dijkstra's Algorithm for Single Source Shortest Paths

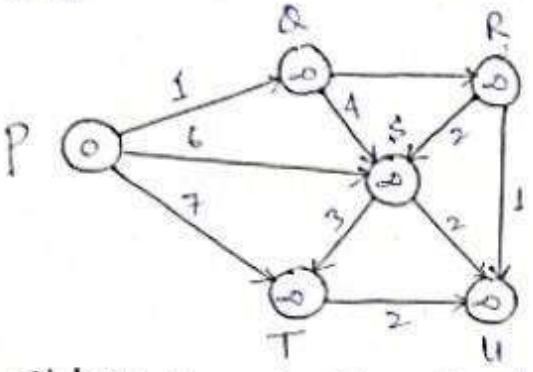
Dijkstra's algorithm solves the single source shortest-paths problem on a weighted graph G = (V , E) for the case in which all edge weights are nonnegative. It works for directed as well as undirected graph.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Apply the greedy single source shortest path algorithm on the following graph:



Step 1 : The source vertex is P, and distance of P to P is zero. We don't know the distance between P to all other vertices, so we mark them with ∞.



Step 2 : Now, Consider all outgoing edges from the vertex P and Preform "relaxation".

Relaxation :

```
if (distance[u] + cost(u,v) < distance[v])
    distance[v] = distance[u] + cost(u,v)
```

⇒ For P to Q edge :↓

u = 0, cost(u,v) = 1 and v = ∞

$0+1<\infty$ True

Update the value of Q with 1.

⇒ For P to S edge :↓

$0+6<\infty$ True

Update the value of S with 6.

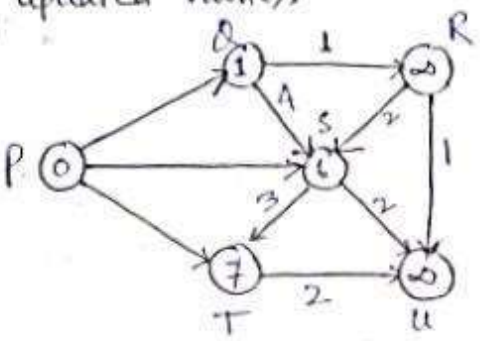
(6)

For P to T edge:

$0+7<\infty$ True

Update the distance value of T with 7.

We have got the graph with updated values.



Having.

After relaxing Q, S and T, we have to pick the vertex with the least value from (Q, S, T), which is Q.

Step 3 : Now, Consider all outgoing edges from the vertex Q and Perform "relaxation".

⇒ For Q to R edge :↓

$1+1<\infty$ True

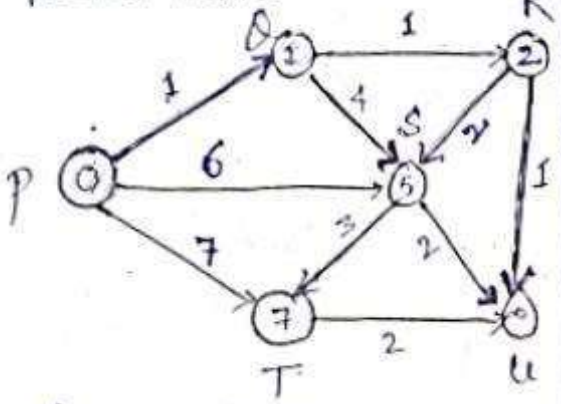
Update the value of R with 2.

⇒ For Q to S edge :↓

$1+4<6$ True

Update the value of S with 5.

we have got the graph with updated values.



After relaxing R and S, we have to pick the vertex with the least value from R, S and T which is 2(R).

Step 3. Now, consider all outgoing edges from the vertex R and perform relaxation.

⇒ For R to S edge :↓

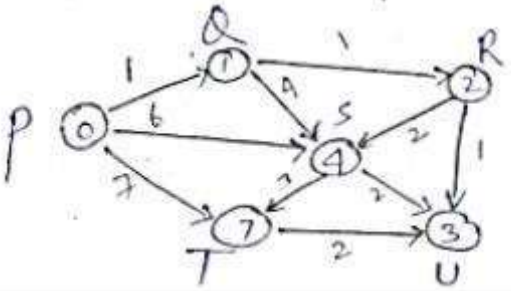
$2 + 2 < 5$ True

Update the value of S with 4

For R to U edge :↓

$2 + 1 < \infty$ True

Update the value of U with 3.



Tabulation Method

(Source)	(Destination) Q	R	S	T	U
P	∞	∞	∞	∞	∞
P	1	∞	6	7	∞
P to Q	1	2	5	7	∞
P to Q to R	1	2	4	7	3
P, Q, R, U	1	2	4	7	3
P, Q, R, U, S	1	2	4	7	3
P, Q, R, U, S, T	1	2	4	7	3

After relaxing S and U, we have to pick the vertex with the least value from R, S, T and U, which is U.

"U" does not have any outgoing edge, so we consider the next smallest value vertex, which is S.

"S" has two outgoing edges, but both of them are already the least distance from P.

P to Q = 1
P to R = 2
P to T = 7
P to S = 4
P to U = 3

Answer

2. Bellman-Ford Algorithm (Dynamic)

```
BELLMAN-FORD(G, w, s)
1.  INITIALIZE-SINGLE-SOURCE(G, s)
2.  for i = 1 to |G.V| - 1  ⇒ (V-1)        } O(VE)
3.      for each edge (u,v) ∈ G.E ⇒ (E)    }
4.          RELAX(u, v, w) - O(1)          }
5.  For each edge (u,v) ∈ G.E ⇒ O(E)
6.      if v.d > u.d + w(u,v)
7.          return FALSE
8.  return TRUE
```

O(VE)

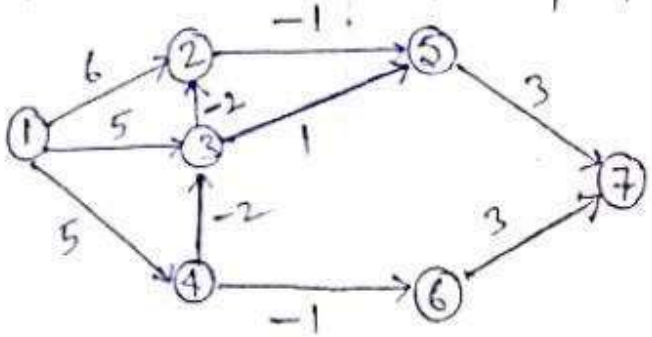
Note:

(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:

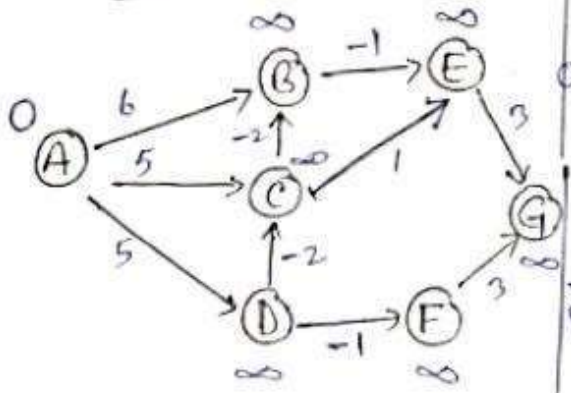


EdgeList → [(1,2), (1,3), (1,4), (2,5), (3,2), (3,5), (4,3), (4,6), (5,7), (6,7)]

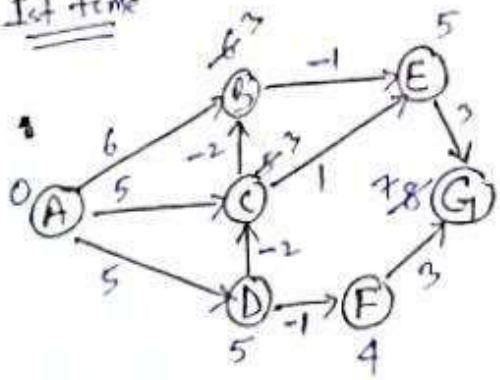
⇓

We need to relax all edges 6 times as the graph has 7 vertices.

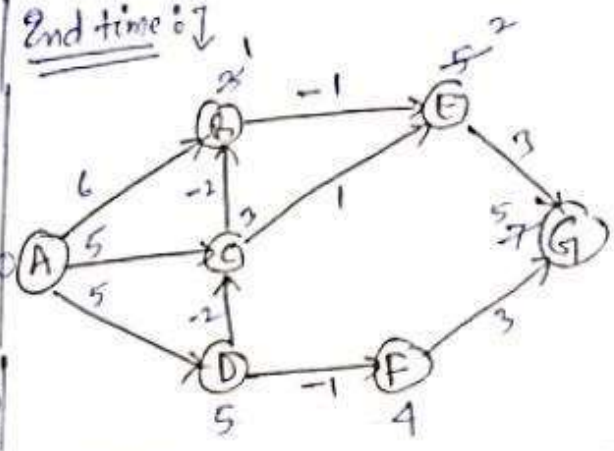
We have to relax all edges V−1 times. Here, V = 7, so we need to relax all edges 6 times.



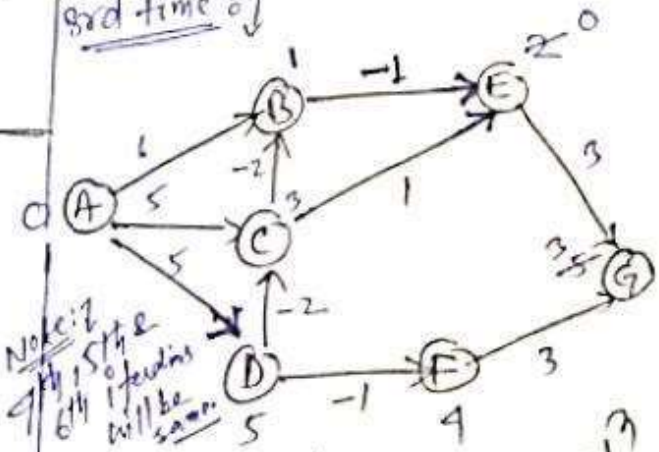
1st time :



2nd time :



3rd time :



Note: The 5th & 6th iteration will be same

A to B = 1
A to C = 3
A to D = 5
A to E = 0
A to F = 4
A to G = 3

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 terms using plain recursion (recursive program).

Fibonacci series : 0 1 1 2 3 5 . . .
Fn: 0 1 2 3 4 5 . . . (Fibonacci terms starting from zero (F_0))
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.

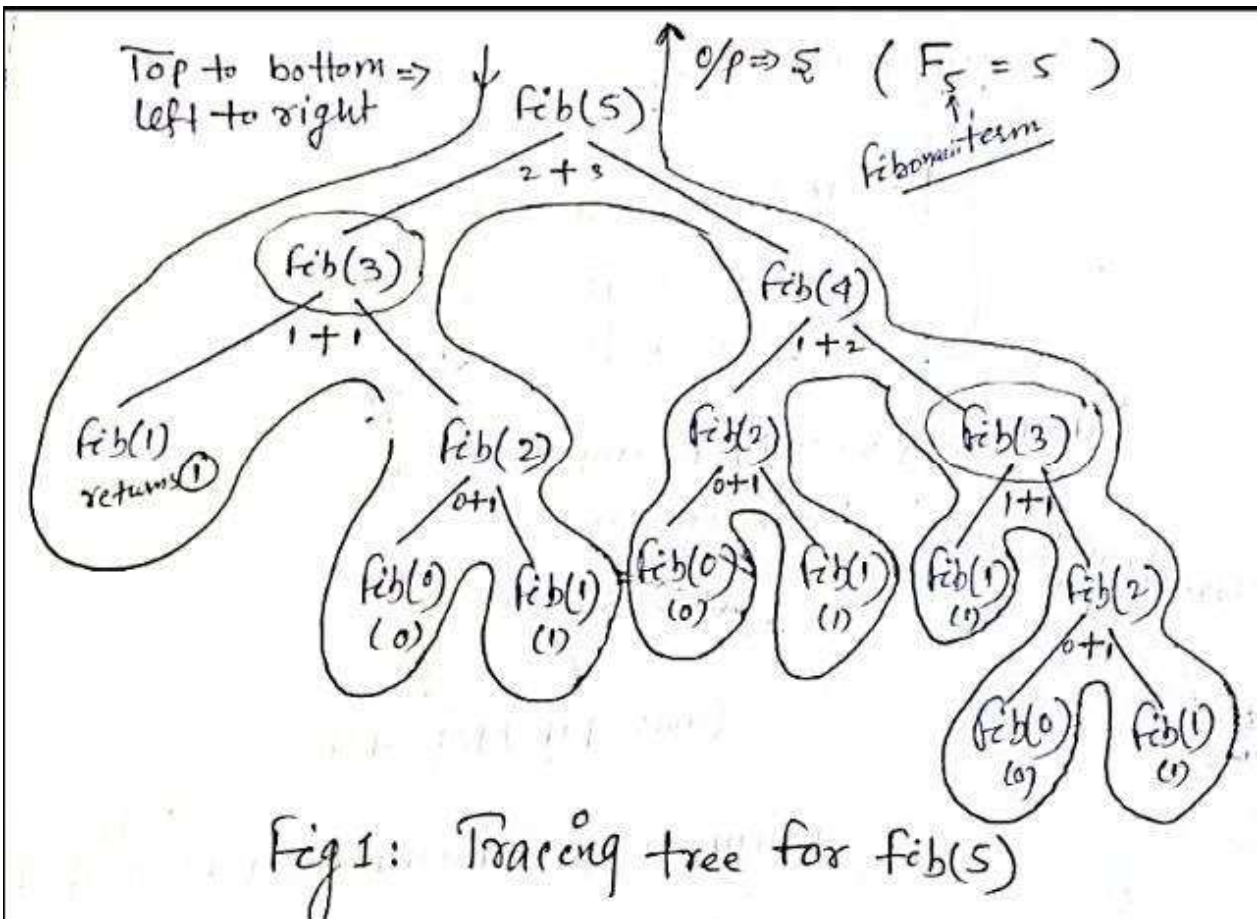


Fig 1: Tracing tree for fib(5)

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 terms 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/ Objects (i)

Capacity(w)

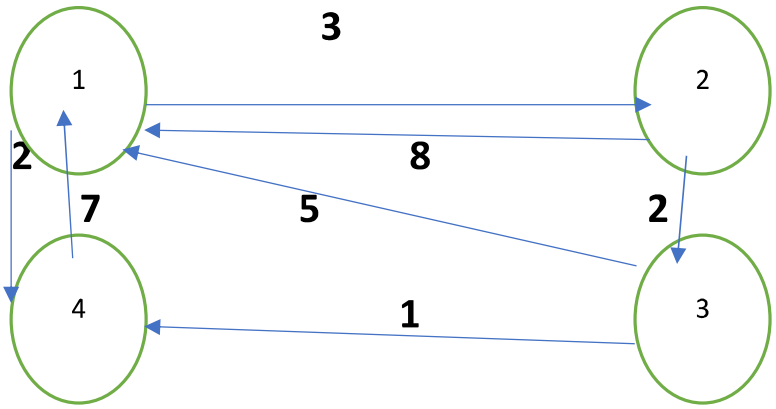
P_i	W_i	0	0	1	2	3	4	5	6	7	8
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5	6	6	7	8

2. Single Source Shortest Path using Bellman-Ford Algorithm (Dynamic Programming)

Kindly refer unit-03

3. All Pairs Shortest Path (Floyd-Warshall Algorithm)

Apply Floyd-Warshall algorithm on the below graph:



(1)

cost matrix

$A^0 =$

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

cost matrix

Note: ↓ use "∞" for not having a direct path.

** Note – As we have 4 vertices, we need 4 matrices to solve this problem.

$\Rightarrow A^1 =$

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

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.

$$A^k[i,j] = \min\{A^{k-1}[i,j],\ A^{k-1}[i,k] + A^{k-1}[k,j]\}$$

(k)

for $A^1[2,3]$

$A^1[2,3] = \min(A^0[2,3],\ A^0[2,1] + A^0[1,3])$

$= \min(2,\ 8+\infty)$

$A^1[2,3] = 2$ — update in A^1 matrix

for $A^1[2,4]$

$\min(A^0[2,4],\ A^0[2,1] + A^0[1,4])$

$(\infty,\ 8+7)$

$A^1[2,4] = 15$ — update in A^1

for $A^1[3,2]$

$\min(A^0[3,2] \to \infty,\ A^0[3,1] \to 5 + A^0[1,2] \to 3)$

$A^1[3,2] = 8$ → update in A^1

$A^1[3,4] = ?$

$\min(A^0[3,4], A^0[3,1] + A^0[1,4])$

$\min(1, 5+7)$

$A^1[3,4] = 1$ — update

$A^1[4,2] = ?$

$\min(A^0[4,2], A^0[4,1] + A^0[1,2])$

$\min(\infty, 2+3)$

$A^1[4,2] = 5$ — update

$A^1[4,3] = ?$

$\min(A^0[4,3], A^0[4,1] + A^0[1,3])$

$\infty \qquad 2+\infty$

$A^1[4,3] = \infty$ — update.

We get A^1 :

A^1	1	2	3	4
1	0	3	∞	7
2	8	0	2	15
3	5	8	0	1
4	2	5	∞	0

A^2	1	2	3	4
1	0	3		
2	8	0	2	15
3		8	0	
4		5		0

For the second matrix A^2, we copy values of the second row and the second column and also the left diagonal.

Again we the same formula to update the second matrix we get the second matrix:

A^2	1	2	3	4
1	0	3	5	7
2	8	0	2	15
3	5	8	0	1
4	2	5	7	0

$\Rightarrow$

A^3	1	2	3	4
1	0		5	
2		0	2	
3	5	8	0	1
4			7	0

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

$\Rightarrow A_4 =$

	1	2	3	4
1	0			6
2		0		3
3			0	1
4	2	3	7	0

copy the 4th row and 4th column from A^3 and also the left diagonal.

Again use the same formula to update the matrix A_4.

$$A^K[i,j] = \min\left(A^{K-1}[i,j],\ A^{K-1}_{[i,k]} + A^{K-1}_{[k,j]}\right)$$

we get the resultant matrix like this ↓

$A^4 =$

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

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 4.

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

$1/3 \times {}^{4}C_{2}$
$1/3 \times 4! / 2! * (4-2)!$
$1/3 \times 4 \times 3 \times 2! / 2! * 2!$
$1/3 \times 4 \times 3 / 2!$
$= 2$ (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?

Consider the following four matrices. Find out optimal parenthesization of matrix chain multiplication.

$A_1 * A_2 * A_3 * A_4$
5×4 4×6 6×2 2×7

Sol^n: Here

As $n = 4$, we can parenthesize these matrices in 5 different ways. We get it using the below formula:

$$\frac{1}{n} \times {}^{2(n-1)}C_{n-1} = \frac{1}{4} \times {}^{6}C_{3} = \frac{1}{4} \times \frac{6 \times 5 \times 4 \times 3 \times 2 \times 1}{3 \times 2 \times 1 \times 3 \times 2 \times 1} = 5$$

All 5 Solutions:

1. $\underset{5\times4}{(A_1} * \underset{4\times6}{A_2)} \cdot \underset{6\times2}{(A_3} * \underset{2\times7}{A_4)}$

2. $\underset{5\times4}{A_1} \underset{4\times2}{(A_2 * A_3)} \underset{2\times7}{A_4}$

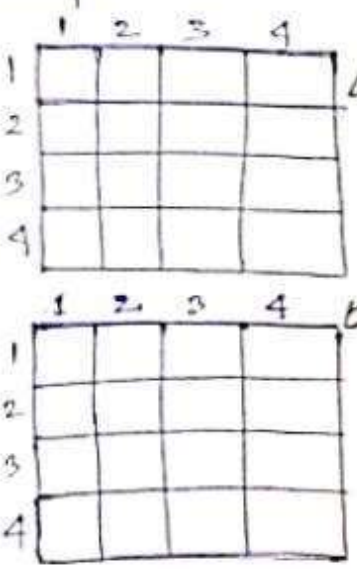
3. $\underset{5\times2}{(A_1 * A_2 * A_3)} \cdot \underset{2\times7}{A_4}$

4. $\underset{5\times4}{A_1} \underset{4\times7}{(A_2 * A_3 * A_4)}$

5. $\underset{5\times4}{A_1} \left(\underset{4\times6}{A_2} \underset{6\times7}{(A_3 \times A_4)} \right)$ $\underset{4\times7}{}$

Out of these 5, we have to find out which one takes the least number of multiplications to derive the resultant matrix using dynamic programming.

Step 01: Draw two matrices having dimension 4×4.



Step 02: order of matrix chain multiplication.

order $P = \langle 5, 4, 6, 2, 7 \rangle$
$p_0\ p_1\ p_2\ p_3\ p_4$

Step 3: 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 4: 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 and equal to 2.

$$m[2,3] = \min_{k=2} \{ m[2,2] + m[3,3] + p_1 p_2 p_3 \}$$

$$= \{ 0 + 0 + 4 \times 6 \times 2 \}$$

$$= 48$$

For the "s" matrix:

$$s[2,3] = 2$$

For $m[3,4]$: $i=3, j=4$ ...

$$m[3,4] = \min \{ m[3,3] + m[4,4] + p_2 p_3 p_4 \}$$

$$= \{ 0 + 0 + 6 \times 2 \times 7 \}$$

$$= 84$$

For the "s" matrix:

$$s[3,4] = 3$$

For $m[1,3]$, $i=1$, $j=3$ and $k=1, 2$ because $i \le k < j$.

$$m[1,3] = \min_{k=1 \text{ and } 2} \begin{cases} m[1,1] + m[2,3] + p_0 p_1 p_3 \\ \quad 0 + 48 + 5 \times 4 \times 2 = 88 \text{ (min)} \\ \text{and} \\ m[1,2] + m[3,3] + p_0 p_2 p_3 \\ \quad 120 + 0 + 5 \times 6 \times 2 = 180 \end{cases}$$

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.

$$m[2,4] = \min_{i \le k < j} \begin{cases} m[2,2] + m[3,4] + p_1 p_2 p_4 = 152 \\ \quad 0 \quad 84 \quad 4\ 6\ 7 \\ m[2,3] + m[4,4] + p_1 p_3 p_4 = 104 \checkmark \\ \quad 48 \quad 0 \quad 4\ 2\ 7 \end{cases}$$

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.

$$m[1,4] = \min_{i \le k < j} \begin{cases} m[1,1] + m[2,4] + p_0 p_1 p_4 = 319 \\ \quad 0 \quad 104 \quad 5\ 4\ 7 \\ m[1,2] + m[3,4] + p_0 p_2 p_4 = 414 \\ \quad 120 \quad 84 \quad 5\ 6\ 7 \\ m[1,3] + m[4,4] + p_0 p_3 p_4 \\ \quad 88 \quad 0 \quad 5 \times 2 \times 7 \\ \quad 88 + 70 = 158 \text{ (min)} \end{cases}$$

For the "s" matrix.

$s[1,4] = 3$ because at $k=3$, we get min value.

Steps: Follow the matrix "s" to get optimal parenthesization.

$$(A_1 * A_2 * A_3) * A_4$$

$$((A_1) * (A_2 * A_3)) * A_4$$

→ optimal soln.

Backtracking

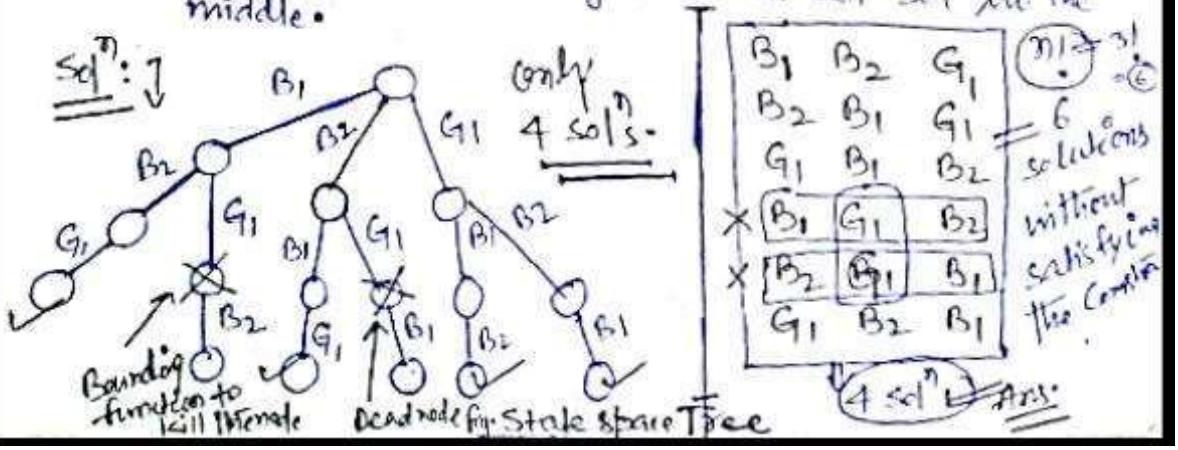
By S.Khan (1)

Definition: It is one of the algorithm design techniques. 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.



V.V.1

Sum of Subsets Problem (using Backtracking)

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.

Sum = 50

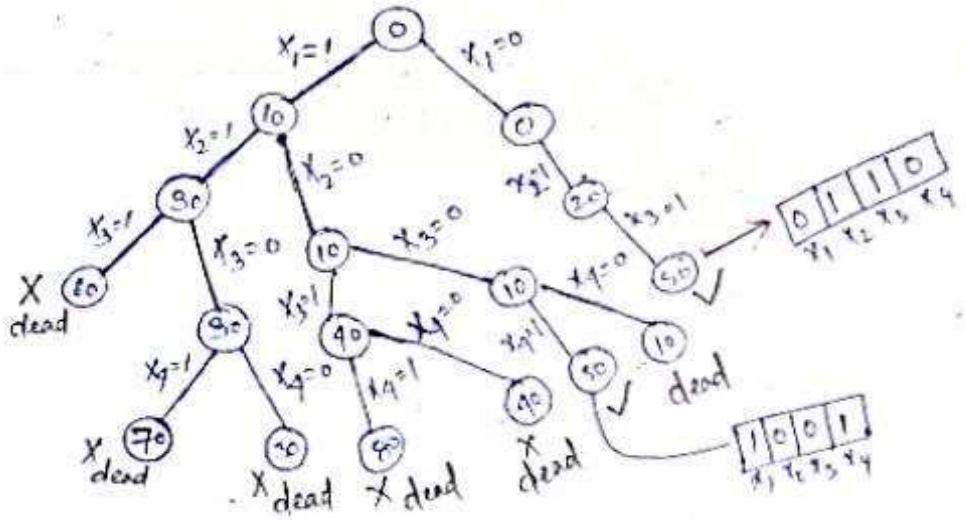
Soln:

$\{10, 20, 30, 40\}$ Sum = 50

$x_1 \; x_2 \; x_3 \; x_4$

Soln: $\{20, 30\}$ & $\{10, 40\}$

State Space Tree:



V.V.I

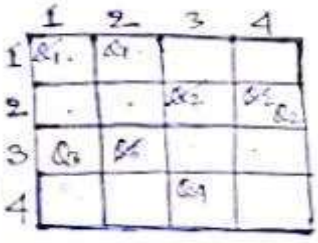
N- Queens problem (Back-tracking) (3)

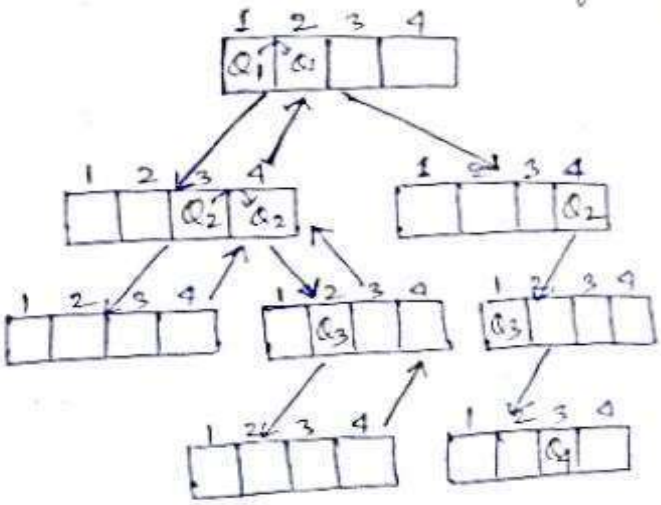
N-Queens problem is to place n-queens in such a manner on an nxn chessboard that no queens attacks each other by being in same row, column or diagonal.

Example :- 4 queens problem

Bounding function :- (Condition) No two queens should be in the 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.

16/11/2021

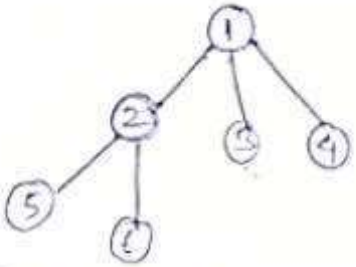
Branch and Bound

By Shan (1)

It is an algorithm design technique used to solve optimization problems. Like Backtracking, it also generates a state-space tree, but it uses BFS approach to generate the SST.

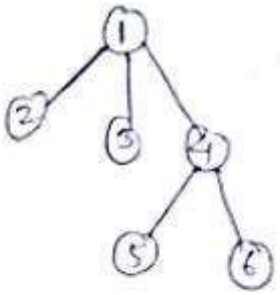
Classification of Branch and Bound Problems:

(1) FIFO Branch and Bound :↓



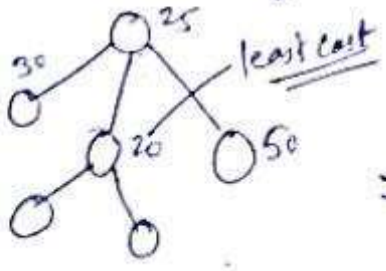
⇒ first in first out BB

(2) LIFO BB :↓



⇒ last-in-first-out BB

(3) Least cost BB :↓



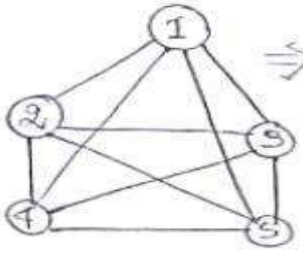
⇒ least cost BB

⇒ 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	∞	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 ✓

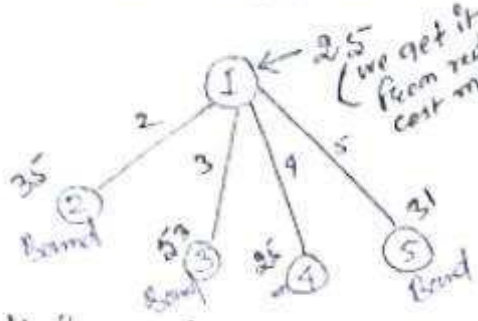
Matrix after column reduction:

	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	9	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:

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	11	2	0
3	∞	∞	∞	0	2
4	15	∞	12	∞	0
5	11	∞	0	12	∞

Make all entries for row 1 and col 2 ∞. Also for [2,1], make it ∞.

reduced cost = 0 → As all rows & col have at least 0.

Cost(2) = 25 + 10 + 0
= 35

[1,2] ↑ check in its reduced cost matrix.

Path 1 to 3 at node 3:

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	∞	2	0
3	∞	3	∞	0	2
4	15	3	∞	∞	0
5	11	0	∞	12	∞

11

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.

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	1	∞	∞	2	0
3	∞	9	∞	0	2
4	3	3	∞	∞	0
5	0	0	∞	12	∞

Path 1 to 4 at node 4:

Make 1st row and 4th col ∞ and also [4,1] as ∞. Rest entries are done from reduced cost matrix only.

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	∞	0
3	0	3	∞	∞	2
4	∞	3	12	∞	0
5	11	0	0	∞	∞

reduced cost = 0

Cost(4): 25 + 0 + 0 = 25

[1,4] ↑

Path 1 to 5 at node 5:

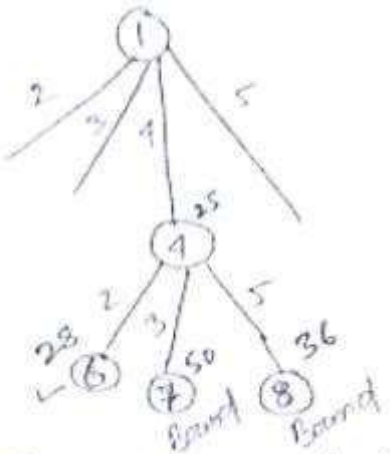
	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	2	∞
3	0	3	∞	0	∞
4	15	3	12	∞	∞
5	∞	0	0	12	∞

2, 3, 2

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 3 ∞ and also [2,1] as ∞.

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	11	∞	0
3	0	∞	∞	∞	2
4	∞	∞	∞	∞	∞
5	11	∞	0	∞	∞

reduced cost of this matrix is zero

Cost(6): 25 + 3 + 0 = 28

(3 ↑ [4,3], 0 ↑ [3,1])

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)

	1	2	3	4	5	
1	∞	∞	∞	∞	∞	
2	12	∞	∞	∞	0	
3	∞	3	∞	∞	2	2
4	∞	∞	∞	∞	∞	
5	11	0	∞	∞	∞	
			11			

reduced cost = 11 + 2 = 13

Cost(7): 25 + 12 + 13 = 50

([4,3])

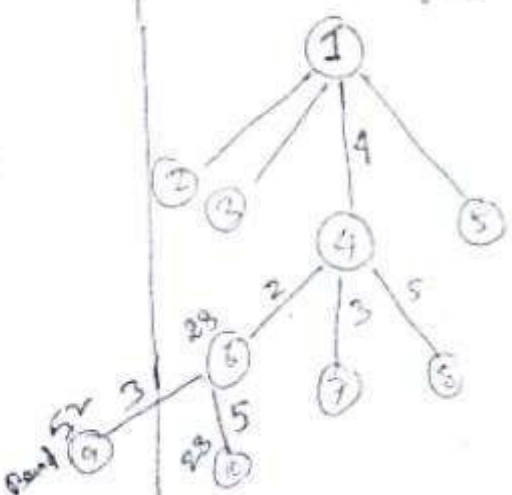
Path 1,4,5 at node 8

	1	2	3	4	5	
1	∞	∞	∞	∞	∞	
2	12	∞	11	∞	∞	11
3	0	3	∞	∞	∞	
4	∞	∞	∞	∞	∞	
5	∞	0	0	∞	∞	

reduced cost = 11

Cost(8): 25 + 0 + 11 = 36

([1,5])



Path 1, 4, 2, 3 at node 9.

Conside matrix having paths from 1 to 4, to 2 at node 6 as this matrix produces cost 28 for the node 6.

	1	2	3	4	5	
1	∞	∞	∞	∞	∞	
2	∞	∞	∞		∞	
3	∞	∞	∞		0	2
4	∞	∞	∞	∞	∞	
5	11	∞	∞	∞		
						11
						13

reduced cost = 13

Cost(9) : 28 + 11 + 13 = 52

[2,3]

Path 1, 4, 2, 5 at node 10.

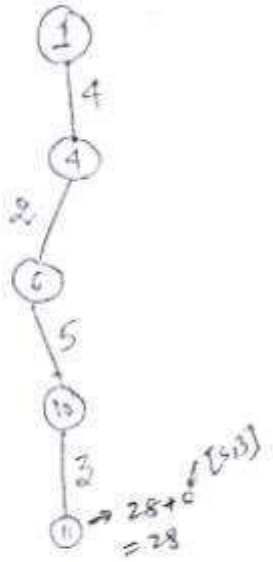
	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	0	∞	∞	∞	∞
4	∞	∞	∞	∞	∞
5	∞	∞	0	∞	∞

reduced cost = 0

Cost(10) : 28 + 0 + 0 = 28

[2,5]

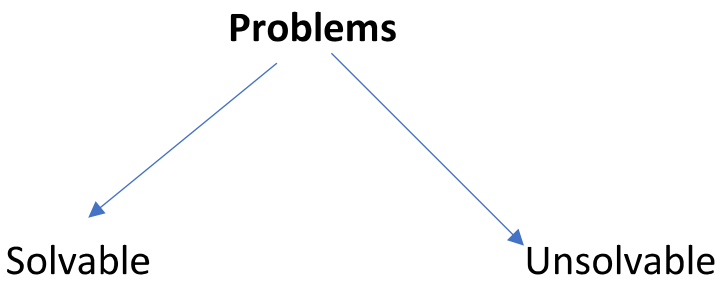
(9)



Paths → 1 → 4 → 2 → 5 → 3

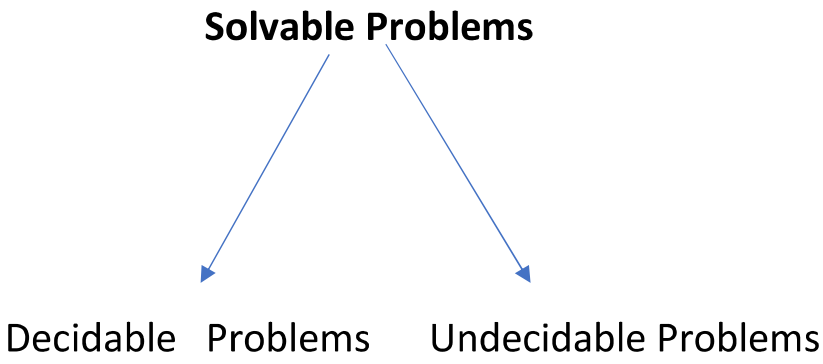
Soln

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?

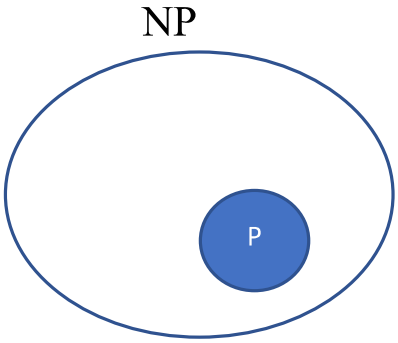
P class – P stands for polynomial. It is a set of problems which can be solved as well as verified in polynomial time. Linear Search O(n) , Binary Search O(logn), Merge Sort O(nlogn), Heap Sort O(nlog), etc., are the examples of algorithms which solve the problem in polynomial time.

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.

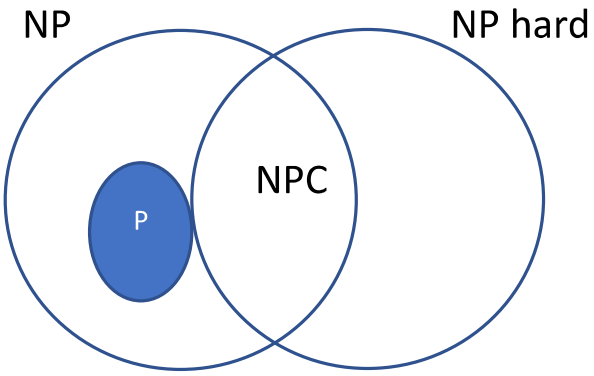


Fig - How P, NP, NP hard and NP complete are related to each other.

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 problems solved by 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.

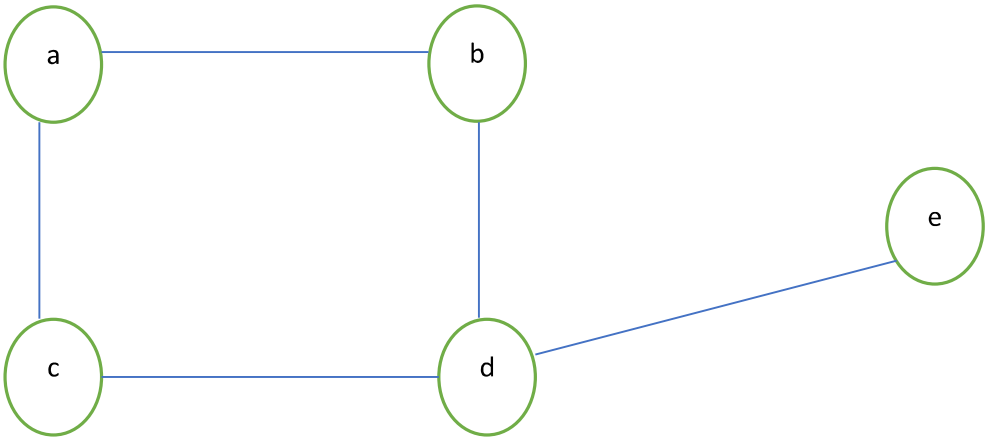
Vertex Cover Problem – Given an undirected graph, the vertex cover problem is to find minimum size vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. It is a minimization problem because we have to find a set of vertices

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
```

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].

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 length

Example.

Text → T = { bababbabaa } = 10

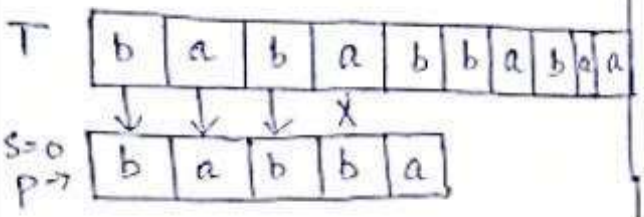
Pattern → P = { babba } = 5

Soln:

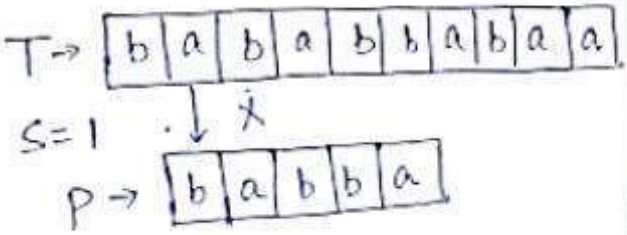
S (shift) = T − P = 10 − 5 = 5

We can shift till 5.

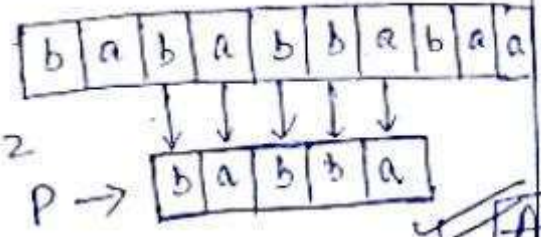
Step 1:



Step 2:

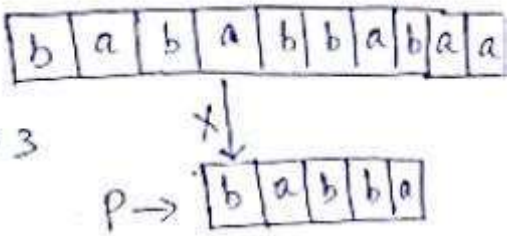


Step 3:

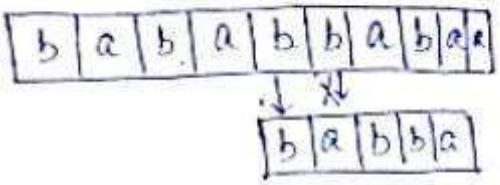


At S = 2, we have got a match.

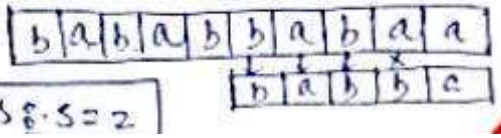
Step 4:



Step 5: Now, S = 4.



Step 6: Now last valid iteration S = 5



Ans: S = 2

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)
 1  n = T.length
 2  m = P.length
 3  h = d^(m-1) mod q
 4  p = 0
 5  t_0 = 0
 6  for i = 1 to m                    // preprocessing
 7      p = (dp + P[i]) mod q
 8      t_0 = (dt_0 + T[i]) mod q
 9  for s = 0 to n - m                // matching
10      if p == t_s
11          if P[1..m] == T[s + 1..s + m]
12              print "Pattern occurs with shift" s
13      if s < n - m
14          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

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)
1  n = T.length
2  q = 0
3  for i = 1 to n
4       q = δ(q, T[i])
5       if q == m
6            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 = { $q_0, \ldots, q_7$ }
∑ = { 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 $\delta : Q \times \sum \rightarrow Q$

$\delta(q_0, a)$ = σ(a) = 1 (since “a” is a prefix in P and a’s length is 1)
$\delta(q_0, b)$ = σ(b) = 0 (No transition as “b” is not a prefix in P)
$\delta(q_0, c)$ = σ(c) = 0 (No transition as “c” is not a prefix in P}

$\delta(q_1, a)$ = σ(aa) = 1 (only single “a” is a prefix in P)
$\delta(q_1, b)$ = σ(ab) = 2 (“ab” is a prefix in P and its length is 2)
$\delta(q_1, c)$ = σ(ac) = 0 (No transition as “ac” is not a prefix in P)

$\delta(q_2, a)$ = σ(aba) = 3 (“aba” is a prefix in P and its length is 3)
$\delta(q_2, b)$ = σ(abb) = 0 (None of its suffixes (b, bb and abb) is in P)
$\delta(q_2, c)$ = 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)

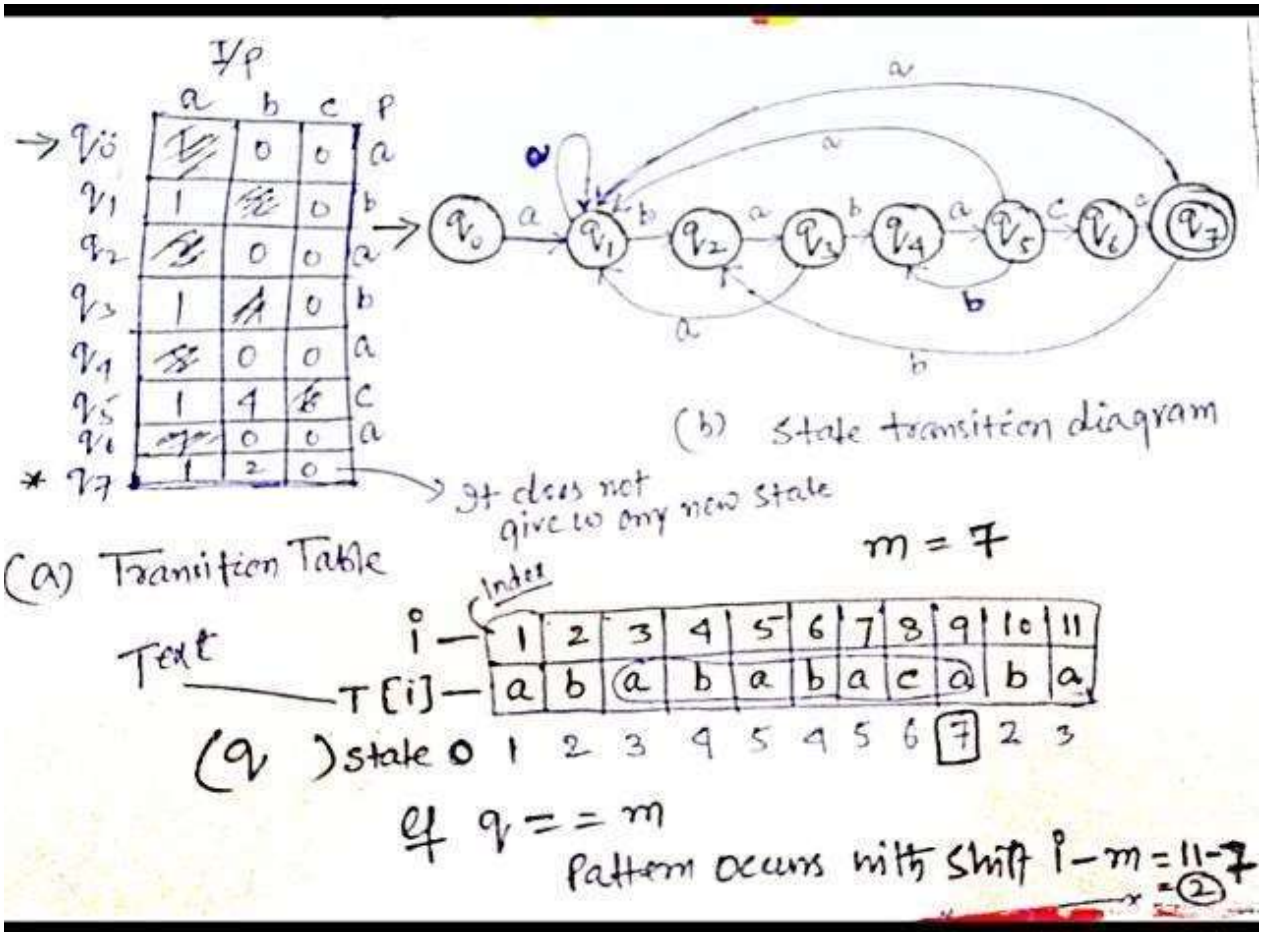
$\delta(q_6 , b) = \sigma(ababacb) = 0$

$\delta(q_6 , c) = \sigma(ababacc) = 0$

$\delta(q_7 , a) = \sigma(ababacaa) = 1$(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
→ q0 1 0 0 a
q1 1 0 b
q2 0 0 a
q3 1 0 b
q4 0 0 a
q5 1 c
q6 0 0 a
* q7 1 2 0
It does not give to any new state
(a) Transition Table
(b) State transition diagram
m = 7
Index
i
Text
T[i]
1 2 3 4 5 6 7 8 9 10 11
a b a b a b a c a b a
(q) State 0 1 2 3 4 5 4 5 6 7 2 3
If q == m
Pattern occurs with shift i − m = 11 − 7 = 4

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)

```
1. n ← length [T]
2. m ← length [P]
3. Π← COMPUTE-PREFIX-FUNCTION (P)
4. q ← 0          // numbers of characters matched
5. for i ← 1 to n// scan S from left to right
6. do while q > 0 and P [q + 1] ≠ T [i]
7. do q ← Π [q]         // next character does not match
8. If P [q + 1] = T [i]
9. then q ← q + 1       // next character matches
10. If q = m                              // is all of p matched?
11. then print "Pattern occurs with shift" i - m
12. q ← Π [q]                          // look for the next match
```

COMPUTE- PREFIX- FUNCTION (P)

```
1. m ←length [P]        //'p' pattern to be matched
2. Π [1] ← 0
3. k ← 0
4. for q ← 2 to m
5. do while k > 0 and P [k + 1] ≠ P [q]
6. do k ← Π [k]
7. If P [k + 1] = P [q]
8.      then k← k + 1
9. Π [q] ← k
10. Return Π
```

Q 4. Compute the prefix function π for the pattern ababbabbabbababbabb when the alphabet is $\sum$ = { 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 the 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).