UNIT-1

Algorithm: The word algorithm comes from the name of a Persian author '***Abu Jafar Mohammed ibn Musa al Khowarizmi***', who wrote a text book on mathematics. According to him an algorithm is a set of rules used to perform some calculations either by hand (or) more usually on a machine.

(or)

An algorithm is a finite set of instructions to accomplish a particular task.

Properties (or) Characteristics of Algorithm:

- **Input:** Zero or more quantities are externally supplied.
- **Output:** Atleast one quantity is produced.
- **Definiteness:** Each instruction is clear and unambiguous
- **Finiteness:** If we trace out the instructions of an algorithm, then for all cases the algorithm terminates after a finite number of steps.
- **Effectiveness:** Every instruction must be very basic so that it can be carried out in principle by a person using only pencil and paper.

Design of Algorithm: The study of algorithm includes many important and active areas of researches. They are:

- **Understanding the problem:** This is the very first step in designing of an algorithm. In this step first of all need to understand the problem statement completely by reading the problem description carefully. After that, find out what are the necessary inputs for solving that problem. The input to the algorithm is called **instance** of the problem. It is very important to **decide the range of inputs** so that the boundary values of algorithm get fixed. The algorithm should work correctly for all **valid inputs.**

- **Decision Making:** After finding the required input set for the given problem we have to analyze the input and need to decide certain issues such as
  - *Capabilities of computational devices:* It is necessary to know the computational capabilities of devices on which the algorithm will be running ie sequential or parallel algorithm. Complex problems require huge amount of memory and more execution time. For solving such problems it is essential to have proper choice of a **computational device** which is **space and time efficient**.
  - *Choice for either exact or approximate problem solving method:* The next important decision is to decide whether the problem is to be solved exactly or approximately. If the problem needs to be solved correctly then we need **exact algorithm.** Otherwise, if the problem is so complex then we need approximation algorithm.(Example: Salesperson Problem)

- *Data Structures:* Data Structure and algorithm work together and these are interdependent. Hence choice of proper data structure is required before designing the actual algorithm.
- *Algorithmic Strategies:* It is a general approach by which many problems can be solved algorithmically. Algorithmic strategies are also called as algorithmic techniques or algorithmic paradigm.

  Algorithm Design Techniques:
  - *Brute Force*: This is straight forward technique with naïve approach.
  - *Divide-and-Conquer*: The problem is divided into smaller instances.
  - *Dynamic Programming*: The results of smaller, reoccurring instances are obtained to solve the problem.
  - *Greedy Technique*: To solve the problem locally optimal decisions are made.
  - *Back tracking*: This method is based on the trial and error.

♦ **Specification of Algorithm:** There are various ways to specify an algorithm

- *Using natural language:* It is very simple to specify an algorithm using natural language

  For Example: Write an algorithm to perform addition of two numbers.

  Step1: Read the first number say **a**.

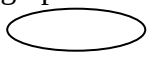
  Step2: Read the second number say **b**.

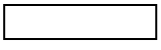
  Step3: Add the two numbers and store the result in a variable **c**.

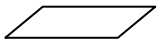
  Step 4: Display the result.
- *Pseudo code:* It is a combination of natural language and programming language

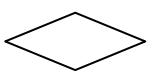
```
Algorithm sum(a,b):
// Addition of two numbers
//Input: Two numbers a and b
//Output: sum of numbers
{
  c=a+b;
  Write(c)
}
```

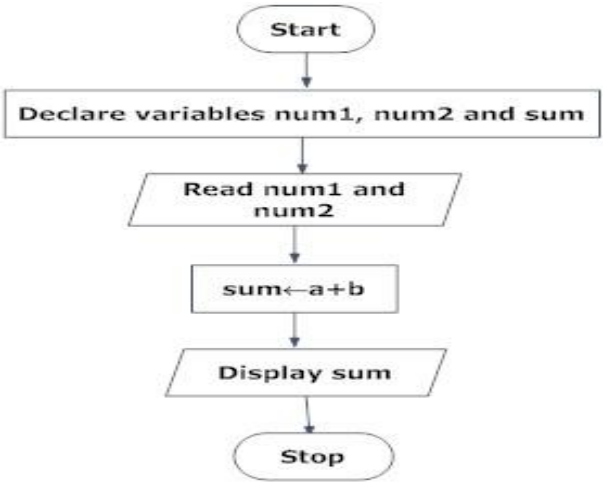
- *Flowchart:* Flowchart is a graphical representation of an algorithm

  Start/ Stop state: 

  Transition: 

  Processing or assignment statement: 

  Conditional Statement: 



- **Algorithm Verification:** Algorithm verification means checking correctness of an algorithm that is to check whether the algorithm gives correct output in finite amount of time for a valid set of input.

- **Analysis of algorithm:** The following factors should consider while analyzing an algorithm
  - *Time Complexity:* The amount of time taken by an algorithm to run.
  - *Space Complexity:* The amount of space taken by an algorithm to store variable.
  - *Range of Input:* The design of an algorithm should be such that it should handle the range of input.
  - *Simplicity:* Simplicity of an algorithm means generating sequence of instructions which are easy to understand.
  - *Generality:* Generality shows that sometimes it becomes easier to design an algorithm in more general way rather than designing it for particular set of input.(Example: GCD)
- **Implementation of Algorithm:** The implementation of an algorithm is done by suitable programming language.
- **Testing a Program:** Testing a program is an activity carried out to expose as many errors as possible and to correct them.

  There are two phases for testing a program:
  - Debugging
  - Profiling

  - **Debugging** is a technique in which a sample set of data is tested to see whether faulty results occur or not. If any faulty result occurs then those results are corrected.
  - But in Debugging technique only presence of error is pointed out. Any hidden error cannot be identified.
  - So, we cannot verify correctness of output on sample data. Hence, **Profiling** Concept is introduced.

- **Profiling or Performance Measurement** is the process of executing a correct program on a sample set of data. Then the time and space required by the program to execute is measured.

Pseudo Code for Expressing Algorithms: In computational theory, we distinguish between an algorithm and a program conventions used in writing a pseudo code. Pseudo code is a combination of natural language and programming language.

Pseudo code is divided into two parts

- ***Algorithm Heading*** which contains the keyword algorithm, name of algorithm, Parameters, Problem Description, Input and Output.

  Example: Algorithm name(p1,p2..pn)
  // problem description
  // Input
  //Output

  - Comments begin with // and continuous until the end of line.
- ***Algorithm Body*** which consists of logical body of the algorithm by making use of various programming constructs and assignment statement
  - Block of statements that are enclosed within '{ }'
  - Each statement is ended with delimiter ' ; '
  - An identifier begins with a letter not by digit and can contain combination of letters and numbers
  - Data types are not declared but we assume simple data types like int, float, char etc.
  - The compound data types are represented with node. If an instance of 'P' is created to a node then the values of node can be accessed by using the operator (.) or →

    ```
    Node = record
    {
        Datatype1 Variable;
        Datatype2 Variable;

    }
    ```

  - Assignment of value to a variable is done using assignment statement

    Variable := Expression;
    Variable←Expression;
  - Logical operators such as AND, OR, NOT and Relational operators such as <,>,<=,>=,=≠,= are used to get a Boolean values true or false.

- The array indices are stored within square brackets "[ ]". The multidimensional arrays can also be used in algorithm.

  Example: A[i]- Single Dimensional Array, A[i,j]-2D Array

- Loop statements like for, while, repeat until are represented as follows

```
for i:=1 to n
{
        Statement 1;
        Statement 2;
}
```

```
while (condition) do
{
        Statement 1;
        Statement 2;
}
```

```
repeat
{
        Statement 1;
        Statement 2;
} until (condition)
```

- Conditional Statements such as if-then or if-then-else are represented as follows:

```
if(condition) then
{
        Statement 1;
        Statement 2;
}
```

```
if(condition) then
{
        Statement 1;
        Statement 2;
}

else

{
        Statement 1;
        Statement 2;
}
```

- Input and output are represented by using read and write

```
read (val);
write (stmt);
```

Example 1:

```
Algorithm Max(A,n)
// A is an array of size n
{
        Result:=A[1];
        for i:=2 to n do
                if A[i]>Result then Result:=A[i];
        return Result;
}
```

Example 2:

```
Algorithm even or odd (n)
// Finding whether given number is even or odd
{
        if(n%2==0)
                write(even);
        else
                write(odd);
}
```

Example 3:

```
Algorithm Selection Sort(a,n)
// Sort the array in Ascending order using selection sort
{
        for i:=1 to n do
        {
                j:=i;
                for k:=i+1 to n do
                {
                        if(a[k]<a[j]) then j:=k;
                }
                t:=a[i];
                a[i]:=a[j];
                a[j]:=a[t];
        }
}
```

Example 4:

```
Algorithm Sort(a,n)
// Sort elements in ascending order
{
        for i:=1 to n do
        for j:=i+1 to n-1 do
        {
                if(a[i]>a[j]) then
                {
                        temp:=a[i];
                        a[i]:=a[j];
                        a[j]:=temp;
                }
        }
}
```

More Examples do it : Multiplication of two matrices, addition of two matrices, Fibonacci, factorial, GCD, reverse of a number etc discussed in running notes

Recursive Algorithm: A recursive function is a function that is defined in terms of itself. Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body. An algorithm that calls itself is **Direct Recursive**. If it calls another algorithm then **indirect recursive.**

Example:

```
Algorithm Towers of Hanoi (n,x,y,z)
// Move top n disks from tower x to tower y
{
        if(n>=1) then
        {
                Towers of Hanoi(n-1,x,z,y);
                write("Top disk x to y")
                Towers of Hanoi (n-1,z,y,x);
        }
}
```

Example:

```
Algorithm factorial(n)
// Factorial using recursion
{
        if(n:=1) then
                return 1;
else
        return n*factorial(n-1);
}
```

Example:

```
Algorithm Perm (a,k,n)
{
        if(k==n) then
                write(a[1:n]); //output permutation
        else    //a[k:n] has more than one permutation
                //generate this recursively
      for i:=k to n do
      {
       t:=a[k];
       a[k]:=a[i];
       a[i]:=t;
       Perm(a,k+1,n);
       t:=a[k];
       a[k]:=a[i];
       a[i]:=t;
      }
}
```

Difference between Algorithm and Pseudo code:

- Algorithm is a well defined sequence of steps that provide a solution for a given problem.
- Pseudo code is one of the methods that can be used to represent an algorithm.
- Algorithms can be written in natural Language.
- Pseudo code is written in a format that is closely related to high level programming language structures
- Pseudo code does not use specific programming language syntax and therefore could be understood by programmer's who are familiar with different programming language.
- Transforming an algorithm presented in pseudo code to programming code could be much easier than converting an algorithm written in natural language.

Performance Analysis: Algorithm evaluation can be done in two ways either before execution of a program or after execution of a program

- Priori Estimate (Performance Analysis) - [before execution of a program]
- Posteriori Estimate (Performance Measurement) - [after execution of a program]

Efficiency of an algorithm can be done by measuring the performance of algorithm. Performance Analysis mainly deals with two factors

- **Space Complexity:** The space complexity of an algorithm is the amount of memory it needs to run to completion.
- **Time Complexity:** The Time complexity of an algorithm is the amount of computer time it needs to run to completion.

Performance Analysis also deals with other factors like

- Measuring Input Range
- Measuring run time
- Computing order of growth of an algorithm
- Computing best case, worst case and average case.

- **Space Complexity:** It is defined as amount of memory required by an algorithm to run. Two factors are used to compute space complexity

  - *Fixed Part:* Independent of input and output characteristics. It includes instruction space, Space for variables, Space for constants and so on.

  - *Variable Part:* It is also called as dynamic part that consists of space needed by variable whose size is dependent on problem instance at runtime. It includes Space needed by reference variables, Recursion stack space and so on.

Let P be an algorithm, then total space required for algorithm is **S(P)=C+Sp**

Where C is constant which is fixed space and Sp is variable space which varies depend on the problem

When we analyze space complexity of an algorithm, we concentrate on estimating Sp(Variable Space)

Example:

```
Algorithm sum(a,n)
//adding elements in array
{
        s:=0;
        for i:=1 to n
                s:=s+a[i];
        return s;
}
```

Space needed for this algorithm as follows:

Sum variable's' – 1 word
Loop Variable 'i' - 1 word
Size variable 'n' – 1 word
Array 'a' values – n words

} total size = n+3 words

Example:

```
Algorithm Rsum(a,n):
// Addition of elements using recursion
{
        if(n<=0) then return 0;
        else
                return Rsum(a,n)+a[n];
}
```

Space needed for this algorithm as follows:

Return address (Rsum) - 1 word
Pointer to 'a' - 1 word
Local variable 'n' - 1 word

} total size=3 words

Depth of Recursion - n+1 words

} total space=3(n+1)

- **Time Complexity:** The amount of time required by an algorithm to complete its execution. Two factors are used to compute time complexity.
  - *Compile Time:* Does not depend on instance characteristics
  - *Run Time:* Depend on particular problem instance

  Let P be an algorithm, then total time required for algorithm is **T(P)=C+Tp,** Where c is compile time and Tp is runtime

Time depends on several other factors like

- System Load
- Number of programs that are running
- Instruction set used
- Speed of underlying hardware

Because of these reasons the time complexity is calculated by using frequency count that is number of times each instruction is executed.

One of the easiest methods to calculate time complexity is counting the number of steps.

We can determine the number of steps needed by a program to solve a particular problem instance in one of two ways

- ♦ In the First method, introduce a new variable **count** into the program. This is a global variable with initial value '0'.
- ♦ The Second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement.

Example: First Method

```
Algorithm sum(a,n)
{
        sum:=0;
        count:=count+1; // Count is global, initially zero
        for i:=1 to n  do
        {
                count:=count+1; // for for i:=1 to n(true condition)
                sum:=sum+a[i];
                count:=count+1; // for assignment
        } count:=count+1; // last time for (false condition)
        return sum;
        count:=count+1; // for return stmt
 }
```

Time complexity: inner loop – count=2n, Remaining – count=3 } total 2n+3 **frequency count**: Θ(n)

Second Method

StatementNum	Statement	Steps per execution	Frequency	Total steps
1	Algorithm sum(a,n)	0	-	0
2	{	0	-	0
3	sum:=0;	1	1	1
4	for i:=1 to n	1	n+1	n+1
5	sum:=sum+a[i]	1	n	n
6	return sum;	1	1	1
7	}	0	-	0
Total Steps for algorithm				2n+3

Time complexity frequency count: Θ(n)

Examples: Matrix Addition: $2n^2+2n+1$ $O(n^2)$, Matrix multiplication: $2n^3+3n^2+2n+1$ $O(n^3)$

```
Algorithm Fibonacci (a,b,c,n)
{
        a:=0;
        b:=1;
        write(a,b);
        for i:=2 to n step 1 do
        {
                c:=a+b;
                a:=b;
                b:=c;
                write(c);
        }
}
```

Time complexity: 5n-1 **Frequency Count**: O(n)

First Method:

```
Algorithm Rsum(a,n):
// Addition of elements using recursion
{
        count:=count+1; // for if condition
        if(n<=0) then
        count:=count+1; // for return stmt
         return 0;
        else
        return Rsum(a,n)+a[n]; // for addition, function invocation and return
}
```

Time Complexity: 2(for n=0)+ TRsum(n-1)

2+TRsum(n-1) => 2+2+TRsum(n-2) …….. n(2)+TRsum(0) => 2n+2 n>0

Second Method:

StatementNum	Statement	Steps per execution	Frequency		Total steps	
			n=0	n>0	n=0	n>0
1	Algorithm Rsum(a,n):	0	-	-	0	0
2	**{**	0	-	-	0	0
3	if(n<=0) then	1	1	1	1	1
4	return 0;	1	1	0	1	0
5	else	0	-	-	0	0
6	return Rsum(a,n)+a[n];	1+x	0	1	0	1+x
7	}	0	-	-	0	0
Total Steps for algorithm					2	2+x

T(n)= 2 for n=0

T(n)=2+T(n-1) for n>0 => 2n+2 for (n>0) **Frequency count**: O(n)

Asymptotic Notations: To choose the best algorithm, we need to check the efficiency of each algorithm. The efficiency can be measured by computing the space and time complexities. Commonly used asymptotic notations are:

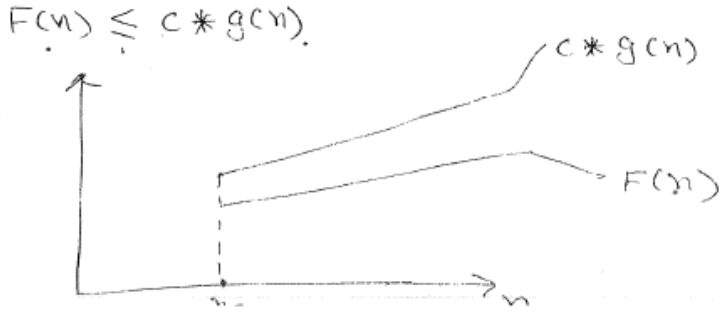
- Big O notation (Upper Bound)
- Omega Notation (Lower Bound)
- Theta Notation (Tight Bound)

By using these notations we can give time complexities as

- Big O – Worst Case
- Omega – Best Case
- Theta – Average Case

- **Big O Notation:** It is denoted by 'O'. It is a method of representing the upper bound of an algorithm that is **worst case** time complexity of an algorithm.
  Definition: A function Let f(n) and g(n) be two non-negative functions and Let n_0,n_c are two integers such that the value of input 'n' is greater than n_0($n>n_0$). Similarly 'c' is a constant which must be greater than zero($c>0$), then we define the function as follows

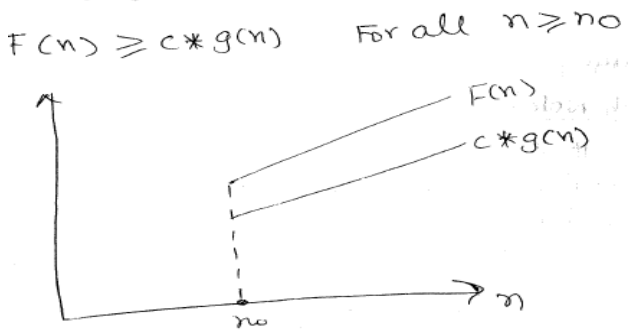
  If f(n) ≤ c*g(n) then f(n)=O(g(n)))



- Consider f(n)=2n+2, g(n)=n^2 so that f(n) ≤ c*g(n) by using mathematical induction
- Show that the time complexity of $3n^2+4n-2$ is $O(n^2)$
- Remaining problems see running notes

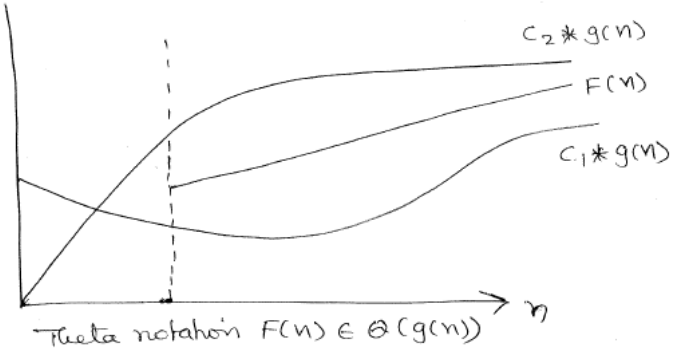
- **Big Omega Notation:** It is denoted by Ω. This notation is used to represent the lower bound of algorithm runtime that is **best case** time complexity.
  Definition: A function f(n) is said to be Ω(g(n)) if f(n) is bounded below by some positive constant multiple of g(n) such that **f(n)≥ c*g(n)**

  If f(n)≥ c*g(n) then f(n)= Ω(g(n))



Consider f(n)=$2n^2$+5, g(n)=7n show that **f(n)= Ω(g(n))**

- **Theta Notation:** It is denoted by Θ. By this method the running time is between upper bound and lower bound.
  Definition: Let f(n) and g(n) be two non-negative functions and consider two positive numbers c1 and c2 such that **c_1*g(n)≤f(n)≤ c_2*g(n) then f(n)=Θ(g(n))**



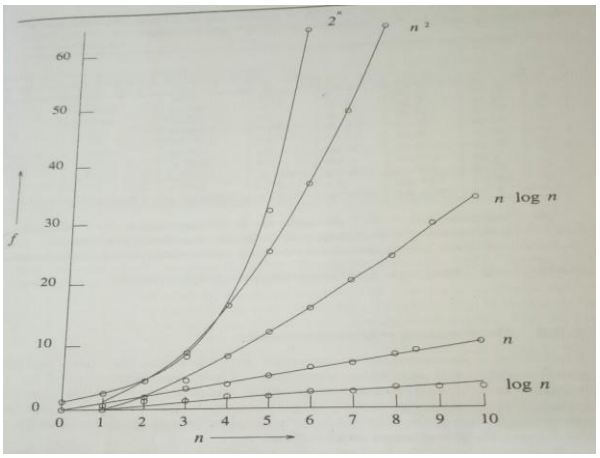
- Consider two functions f(n)=2n+2, g(n)=n find the value of c1 and c2
- Find the theta notation for following function f(n)=$3n^2$+5n+2 , g(n)=n^2 **c_1*g(n)≤f(n)≤ c_2*g(n)**

- **Little o notation: if f(n)=O(g(n))) and f(n)≠Θ(g(n)) then f(n)=o(g(n))**
- **Little Omega notation: if f(n)= Ω(g(n))and f(n)≠Θ(g(n)) then f(n)=ω(g(n))**

Practical Complexities:

- Time complexity of an algorithm is generally some function of the instance characteristics.
- This function is very useful in determining how the time requirements vary as the instance characteristics change.
- The complexity function can also be used to compare two algorithms P and Q that perform the same task.
- Assume that algorithm P has complexity Θ(n) and algorithm Q has complexity Θ(n^2). We can assert that algorithm P is faster than algorithm Q for sufficiently large n by seeing polynomial of high degree.
- How the various functions grow with 'n'. See the below table and figure.

log n	n	n log n	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4,096	65,536
5	32	160	1,024	32,768	4,294,967,296



- By seeing the above figure we can say that the function 2^n grows very rapidly with n.

Amortized Analysis:

- Amortized Analysis means finding average running time per operation over a worst case sequence of operations.
- An Amortized analysis indicates that average cost of a single operation is small if average of sequence of operations is obtained.
- There is a difference between amortized and average case analysis. In average case analysis, averaging over all possible inputs but in amortized analysis, averaging over a sequence of operations.
- Suppose that a sequence of operations I1, I2, D1, I3, I4, I5, I6, D2, I7 of insert and delete operations are performed on set. Assume the actual cost of each seven inserts takes 1 unit of time and delete operations D1 and D2 takes 8 and 10 respectively. Total Actual Cost=7+8+10=25
- In amortization scheme, charge some actual cost of operations to other operations. This reduces cost of one operation and reduces cost of other operations.
- If we charge cost of one unit for each insertion (I1 to I6) from delete operation then D1=6, D2=6.
- The two units of D1 is transferred to I1, I2 that is I1=2, I2=2 and 4 units of D2 is transferred that is I3=2, I4=2, I5=2, I6=2, lastly I7=1.
- Total Amortized Cost of I1, I2, D1, I3, I4, I5, I6, D2, I7 = 2+2+6+2+2+2+2++6+1=25 that is amortized cost equal to actual cost. In general amortized cost greater than or equal to sum of their actual cost
- There are three commonly used techniques used in amortized analysis:
  - Aggregate Analysis
  - Accounting Method
  - Potential Method

➢ **Aggregate Analysis:** It is similar to average case analysis but we consider the average for worst case sequence.

In aggregate analysis, if sequence of 'n' operations takes worst case time T(n) in total. In that worst case, the average cost or amortized cost per operation is **T(n)/n**

$$\text{Amortized cost} \geqslant \text{Actual cost}$$

Potential function p(i)=Amortization cost (i) – Actual cost(i) + p(i-1)

$P(n) - P(0) \geqslant 0$

Example 1: In Jan, you buy a new car from a dealer who offers you the following maintenance contract $50 each month other than March, June, September and December, $100 every March, June and September and $200 every December.

Worst Case Method: $50- Jan,Feb,Apr,May,Jul,Aug,Oct,Nov
$100- Mar,Jun,Sep
$200- Dec

Aggregate Method: To use aggregate method for amortized complexity, we first determine an upper bound on the sum of the costs for the first 'n' months. As tight a bound as is possible is desired.

The sum of the actual monthly costs of the contract for the first 'n' months is (50*8)+(100*3)+(200)=400+300+200=900/12=75

MONTH	1	2	3	4	5	6	7	8	9	10	11	12
Actual Cost	50	50	100	50	50	100	50	50	100	50	50	200
Amortized Cost	75	75	75	75	75	75	75	75	75	75	75	75
P(i)	25	50	25	50	75	50	75	100	75	100	125	0

Example 2: *Implementing Stacks on Array*

Two fundamental operations takes O(1) time – (push, pop) Since each of these operations runs in O(1) time. The total cost of a sequence 'n' push and pop operation is **O(n).**

Now, we add other operation MULTIPOP(S, K) which remove K top objects of stack S.

```
MULTIPOP(S, K)
While not STACK = EMPTY(S) and K>0
 pop(S)
K=K-1
```

Average Cost of an operation is O(n)/n = O(1). Therefore, all three stack operations have an **amortized cost O(1).**

- **<u>Accounting Method:</u>** Accounting method is performed based on the charges that we are assigning to the operation. The idea of accounting method as follows:
  - Assign different charges to different operations.
  - Amount of charge is called amortized cost.
  - There will be actual cost which will define the nature of the algorithm.
  - Amortization cost can be less than or greater than actual cost.
  - When it is greater than actual cost, the difference is saved in an object called **credit**
  - If it is less than actual cost, the **stored credits** are used.
  - In Accounting method, **amortized cost = actual cost + credit.**

Example 1:

MONTH	1	2	3	4	5	6	7	8	9	10	11	12
Actual Cost	50	50	100	50	50	100	50	50	100	50	50	200
Amortized Cost	70	70	70	70	70	70	70	70	70	70	70	70
P(i)	20	40	10	30	50	20	40	60	30	50	70	-60

p(i)=Amortization cost (i) – Actual cost(i) + p(i-1)

$P(n) - P(0) \geqslant 0$

P(i)=-60<0 , condition failed

Invalid Amortization Cost (70). Again assume amortized cost as 80 and do the calculation

Example 2: *Implementing Stacks on Array*

Consider Push and pop operation

Push: top=max-1
top++
a[top]=item

pop: item=a[top]
top--

Let the actual cost required to perform either push or pop operation is 1. Let us assume amortization cost push=2 and pop=0 then

For push: Credit= Amortization cost – Actual cost
=2-1 = 1
For pop: Credit= Amortization cost – Actual cost
=0-1 = -1

- **<u>Potential Method:</u>** This method is similar to accounting method in which the concept the prepay is used. In this method there will be no credit but there will be some potential difference or energy which can be used to pay for future operations. Instead of associating potential with specific object, it is associated with whole data structure.
Working of Potential Method: Let D_0 be the initial data structure for 'n' operations. We have data structure D_0 to D_n and then actual cost has $C_1 \ldots C_n$. Potential function is denoted by Ø then

Amortization cost = actual cost + potential difference

$$\sum_{i=1}^{n} C_i = \sum_{i=1}^{n} C_i + [Ø(D_n) - Ø(D_0)]$$

Example: Consider a stack of size 'S' then initial charge be 'S'.

- if a push operation is performed on stack then the potential function $Ø(D_n) = S+1$
- If a pop operation is perform on stack then potential function $Ø(D_n) = S-1$
- Potential difference for push operation is $Ø(D_n) - Ø(D_0) = S+1-S = 1$
- Potential difference for pop operation is $Ø(D_n) - Ø(D_0) = S-1-S = -1$
- Amortization cost for push and pop is

Amortization cost = actual cost + potential difference
= 1+1 = 2
Amortization cost = actual cost + potential difference
= 1-1 = 0

<u>Example:</u> Amortized cost = actual cost + P(1)-P(0) = 50+25-0=75
= actual cost + P(2)-P(1) = 50+50-25=75
= actual cost + P(3)-P(2) = 100+25-50=75
:
:
:

Calculate all costs (Maintenance car)

- **<u>Note:</u>** Amortized analysis is used for algorithms where an occasional operation is very slow but most of the other operations are faster.

In Amortized analysis, we analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation.

Performance Measurement:

- Performance evaluation can be done in two ways. Before the execution of a program called Performance Analysis and after the execution of a program called Performance Measurement.
- Performance Measurement is concerned with obtaining the space and time requirements of a particular algorithm.
- These quantities depend on the compiler and options used as well as on a computer on which the algorithm is run.
- To obtain computing or run time of a program we need clocking procedure that returns the current time in milliseconds.
- Time complexity can be calculated with the help of any programming language like C, C++, Java and Python.
- In C language, the time events are stores in standard library #include<time.h> and use GetTime() to get current time in milliseconds.
- In Java language, the time events are stores in package java.util.Date and use getTime() to get current time in milliseconds.
- In Python language, the time events are stores in package time and use time.time() to get current time in milliseconds.
- Performance Measurement can be calculated by subtracting start time from current time that is **performance measurement=current time - start time**
- Suppose we wish to measure the worst case performance of the sequential search algorithm. First, we need to decide the values of n for which the times are to be obtained and then determine for each of the above values of n, the data that exhibit the worst case behavior.

```
Algorithm SeqSearch(a,x,n)
{
        i:=n;
        a[0]:=x;
        while(a[i]!=x) do
                i:=i-1;
        return i;
}
```

- In the worst case, to search for x, given the size n of a. An asymptotic analysis reveals that this time is Θ(n). So, we expect a plot of the times to be a straight line.
- To measure the computing time of this algorithm we need to write a TimeSeqSearch algorithm, in that GetTime() is used to get the current time and start time of algorithm in milliseconds.
- Finally, performance measurement of an algorithm can be done in milliseconds.

```
Algorithm TimeSeqSearch(a,x,n)
{
        h:=GetTime();
        i:=n;
        a[0]:=x;
```

```
        while(a[i]!=x) do
                i:=i-1;
        return i;
        h1:=GetTime();
        t:=h1-h;
        write(t);
}
```

- Now we are writing an algorithm to measure the time in milliseconds for ‘n’ different values.

```
Algorithm TimeSearch()
{
        for j:=1 to 1000 do
        {
                a[j]:=j;
        }
        for j:=1 to 10 do
        {
                n[j]:=10*(j-1);
                n[j+10]:=100*j;
        }
        for j:=1 to 20 do
        {
                h:=GetTime();
                k:=SeqSearch(a,0,n[j]);
                h1:=GetTime();
                t:=h1-h;
                write(n[j],t);
        }
}
```

- Timing results of above algorithm in milliseconds are

n	time	n	time
0	0	100	0
10	0	200	0
20	0	300	1
30	0	400	0
40	0	500	1
50	0	600	0
60	0	700	0
70	0	800	1
80	0	900	0
90	0	1000	0

- The times obtained are too small to be of any use to us. Most of the times are zero, this indicates that precision of our clock is inadequate. The nonzero times are just noise and are not representative of the time taken.

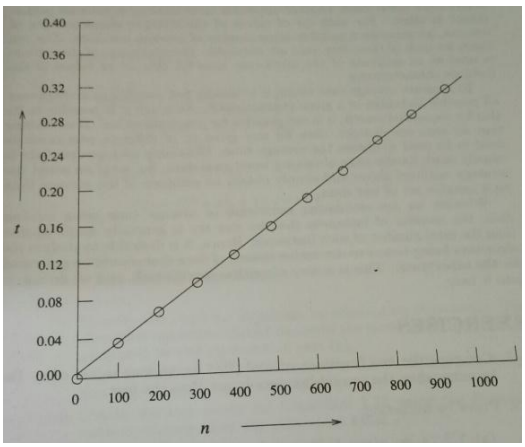
- To time a short event, it is necessary to repeat it several times and divide the total time for the event by the number of repetitions

```
Algorithm TimeSearch()
{
    // Repetition factors
    r[21]:={0, 200000, 200000, 150000, 100000, 100000, 100000, 50000,
    50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000,
    25000, 25000, 25000, 25000}
    for j:=1 to 1000 do
    {
        a[j]:=j;
    }
    for j:=1 to 10 do
    {
        n[j]:=10*(j-1);
        n[j+10]:=100*j;
    }
    for j:=1 to 20 do
    {
        h:=GetTime();
        for i:=1 to r[j]do
        {
            k:=SeqSearch(a,0,n[j]);
        }
        h1:=GetTime();
        t1:=h1-h;
        t:=t1;
        t:=t/r[j];
        write(n[j],t1,t);
    }
}
```

- Timing results and graph of above algorithm in milliseconds are

n	$t1$	t	n	$t1$	t
0	308	0.002	100	1683	0.034
10	923	0.005	200	3359	0.067
20	1181	0.008	300	4693	0.094
30	1087	0.011	400	6323	0.126
40	1384	0.014	500	7799	0.156
50	1691	0.017	600	9310	0.186
60	999	0.020	700	5419	0.217
70	1156	0.023	800	6201	0.248
80	1306	0.026	900	6994	0.280
90	1460	0.029	1000	7725	0.309

Times are in milliseconds



Unit-II

Divide & Conquer

/control abstractions

→ General method of divide & conquer :-

In divide & conquer method, a given problem is

→ divided into smaller sub problems.

→ These sub problems are solved independently.

→ combining all the solutions of sub problems into a single soln.

→ If the sub-problems are the large enough then D&C is reapplied.

→ The generated sub-problems are usually of same type as the original prblm hence recursive algorithms are used.

E.g:- finding a defective coin from a group of 16 coins.

→ If we use normal method, i.e. picking one coin and comparing it with each of remaining coins require maximum of 15 comparisions.

→ If we perform d & c, 16 coins can be divided into two-halves, after compairing two-halves it is again divided into 4-halves, this method will continue upto we recognise defective coin.

→ By using d & c, we require maximum of 4 comparisons to find defective coin.

Control abstraction of D & C :-

```
Algorithm DC(P)
{
    if P is too small then
        return P;
    else
        P is divided into sub.problms P1 P2 -- Pn where
                                                n ≥ 1
        Apply DC for these problems
        return Combine (DC(P1), DC(P2) -- DC(Pn));
}
```

→ The computing time for above procedure 'd&c' is given by the recurrence relation is

$$T(n) = \begin{cases} g(n) & n \text{ is small} \\ T(n_1) + \cdots T(n_k) + f(n) & \text{if } n \text{ is large} \end{cases}$$

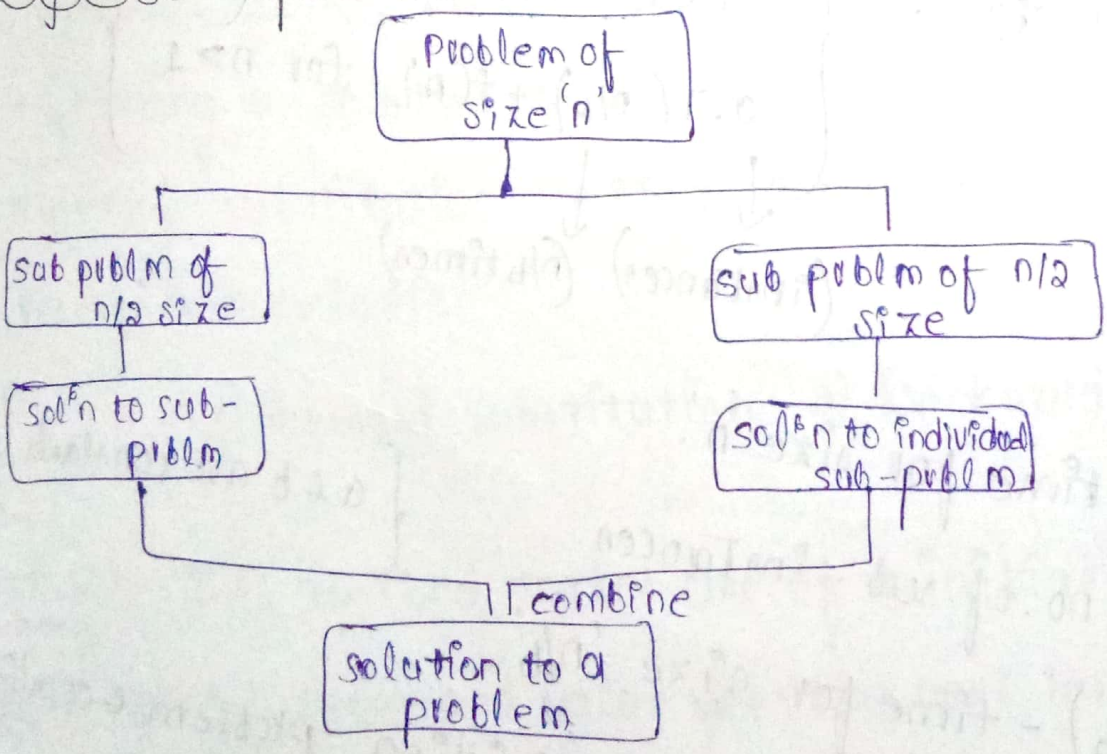
$T(n)$ - time for 'd&c' of size 'n'.

$g(n)$ - computing time required to solve small inputs.

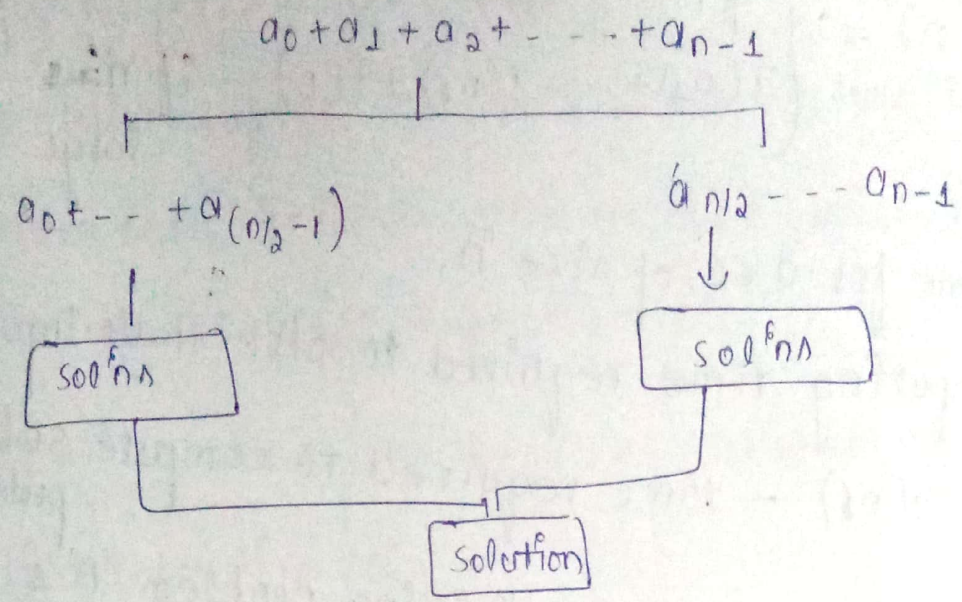
$T(n_1) + \cdots T(n_k)$ - time required to compute sub problems

$f(n)$ - time required dividing problem 'P' & combine the solutions of sub-problems.

Diagramatic Representation:-



→ If you want to compute, sum of 'n' no's then by divide & conquer we can solve the problem.



→ Time complexity of d&c algorithm is given by the form $T(n) = \begin{cases} T(1) & \text{for } n=1 \\ a.T(n/b)+f(n) & \text{for } n>1 \end{cases}$

(a instances) (n/b times)

→ $T(n)$ - time for size 'n'.

a - no. of sub-instances { a & b are constants }

$T(n/b)$ - time for size 'n/b'

$f(n)$ - time required for dividing problem & combine the solutions of sub-problems.

Recurrence Relation:-

→ The recurrance equ^{n} is an equation that defines sequence recursively.

→ The normal form of recurrence relation is

$T(n) = T(n-1) + n$ — (1) [recurrence relation equ^n]

$T(n) = 0$ — (2) [initial condition]

→ Solving Recurrence Relation :- The R.R. can be solved by following methods.

1) substitution method [guessing the sol^n for prob^m]

2) Master's method.

1) Substitution method :-

→ Guessing a solution to recurrence relation. Two types of substitution methods are used to solve recurrence relation.

1) forward substitution 2) Backward substitution.

1) F.S. :- This method makes use of an initial condition in the initial term and value for the next term is generated.

→ This process is continued until there some formula is guessed.

E.g:- Consider a recurrence relation with

$T(n) = T(n-1) + n$ with initial condition $T(0) = 0$.

Forward :-

$n=0$, $T(0) = 0$

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

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

$n=3$, $T(3) = T(3-1) + 3 = 6$

$n=4$, $T(4) = T(3) + 4 = 10$

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

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

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

Backward substitution :- [in this method, backward values are substituted recursively in order to derive some formula]

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

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

$T(n-3) = T(n-4) + n-3 \rightarrow (4)$

substitute equn^n (2), (3), (4) in eq. (1)

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

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

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

Consider $k = 4$

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

$n-k = 0 \Rightarrow \boxed{n = k}$

$= T(0) + (n-n+1) + (n-n+2) + (n-n+3) + n$

$= 0 + 1 + 2 + 3 + \cdots + n$

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

→ Solve the following R.R. $T(n) = T(n-1) + 1$ with $T(0) = 0$, find bigoh notation.

R.S:-

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

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

$T(n-3) = T(n-4) + 1 \rightarrow (4)$

substituting,

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

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

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

$= T(n-4) + 4$

$K = 4,$

$= T(n-K) + K$

$n - K = 0 \Rightarrow n = K$

$\Rightarrow T(n-n) + n$

$T(0) + n$

$0 + n$

$= n$

$\Rightarrow O(n)$

F.S:-

$n = 0, T(0) = 0$

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

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

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

$n = 4, T(4) = 3 + 1 = 4$

$n = 5, T(5) = 4 + 1 = 5$

$T(n) = O(n)$

(ii) Master's method :- (master's theory)

consider recurrence relation $T(n) = a \cdot T(n/b) + F(n)$ where $(n \geq d)$

where d is constant

then the masters theorem can be stated for efficiency analysis an

if $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ in the recursive relation then

(i) $T(n) = \Theta(n^d)$ if $a < b^d$

(ii) $T(n) = \Theta(n^d \log n)$ if $a = b^d$

(iii) $T(n) = \Theta(n \log_b a)$ if $a > b^d$

1. $T(n) = 2T(n/2) + n$ with initial condition $T(1) = 1$.

2. $T(n) = T(n/2) + 1$ with initial condition $T(1) = 1$.

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

4. Solve the recurrence relation $T(n)$ where

$$T(n) = \begin{cases} T(1) \text{ for } n = 1 \\ a \cdot T(n/b) + f(n) \end{cases}$$

where $a = 5, b = 4, f(n) = cn^2$

1) Masters method :-

$a = 2$

$b = 2$

$d = 1$

$b^d = 2$

$\Rightarrow a = b^d$

$O(n^d \log n) \Rightarrow O(n^1 \log n)$

$= O(n \log n)$

Substitution method :-

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

$T(1) = 1$

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

$T(n/4) = 4[2T(n/8) + n/4] + 2n = 8T(n/8) + 3n$

$T(n/8) = 8[2T(n/16) + n/8] + 3n = 16T(n/16) + 4n$

$= 2^4 T(n/24) + 4n$

$k = 4$

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

$2^k = n \Rightarrow \log 2^k = \log n$

$k \log_2 2 = \log n$

$k = \log n$

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

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

$= n + n \log n$

$= O(n \log n)$

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

$T(1) = 1$

$T(n/2) = T(n/4) + 1 \rightarrow (2)$

$T(n/4) = T(n/8) + 1 \rightarrow (3)$

$= T(n/8) + 3$

$T(n/8) = T(n/16) + 4 \rightarrow (4)$

$a = 1$
$b = 2$
$d = 1$,
$b^d = 2$
$\theta(n^d) = \theta(n^1)$
$= \theta(n)$

Sub. (2), (3), (4) in (1)

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

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

$= T(n/8) + 3$

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

$k = 4$

$= T(n/\log 4) + k$

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

$2^k = n \Rightarrow \log 2^k = \log n$

$k \log 2 = \log n$

$k^2 = \log n$

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

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

$a = 4 ; b = 2 ; d = 1 ; b^d = 2$

$a > b^d$

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

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

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

iv) $T(n) = \begin{cases} T(1) \text{ for } n=1 \\ a \cdot T(n/b) + f(n) \end{cases}$ where $a=5; b=4; f(n)=cn^2$

$T(n) = 5T(n/4) + cn^2 \rightarrow (1)$

$a = 5; b = 4; d = 2; b^d = 16$

$a < b^d$ then

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

Substitution method :-

$T(n/4) = 5T(n/16) + c(n/4)^2 \rightarrow (2)$

$T(n/16) = 5T(n/64) + c(n/16)^2 \rightarrow (3)$

Sub. (2) & (3) in (1)

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

$= 5\left[5T(n/16) + \frac{cn^2}{16}\right] + cn^2$

$= 25T(n/16) + 5\frac{cn^2}{16} + cn^2$

$= 25T(n/16) + n^2\underbrace{\left(\frac{5c}{16} + 1\right)}_{c}$

$= 25T(n/16) + n^2(c)$

consider $k=16$

$= (k+9)T(n/k) + n^2c$

$k=n$

$(n+9)T(n/n) + n^2c \Rightarrow n^2c + 1 + n + 9$

$cn^2 + n + 9 + 1$

$\Rightarrow cn^2 + n + 10$

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

Another variation of master's theorem is :-

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

1. if $f(n) = O(n \log_b^{a-\varepsilon})$ then

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

2. if $f(n) = \theta(n^{\log_b a} \log^k n)$ then

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

3. if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ then

$$T(n) = \theta(F(n))$$

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

(ii) $T(n) = 8T(n/2) + n^2$

(iii) $T(n) = 9T(n/3) + n^3$

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

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

$a = 2, b = 2;$

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

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

ii) $T(n) = 8T(n/2) + n^2$

$a = 8, b = 2; d = 2$

$\log_2^8 = \log_2 2^3 = 3$

$\Rightarrow \boxed{k = 3}$

$f(n) = \theta(n \log_2 2^3)$

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

iii) $T(n) = 9T(n/3) + n^3$

$a = 9, b = 3; d = 3$

$\log_b^a \Rightarrow \log_3^9 \Rightarrow \log_3 3^2 \Rightarrow 2$

$f(n) = \theta(n \log_3 3^2)$

$T(n) = \theta(n^3)$ → Adding θ(case iii)

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

$a = 1, b = 2; d = 1$

$\log_b^a \Rightarrow \log_2^1 = 0$

$f(n) = \theta(n^{\log_b a} \log^k n)$

$= \theta(n^0 \log^{0+1} n)$

$T(n) = \theta(\log n)$ [case (iii)]

- Applications of divide & conquer :-

→ Defective chessboard

→ finding max & min.

→ Binary search

→ Quick sort

→ Merge sort

Binary search :-

→ it is an efficient searching method while searching the elements using this method the essential thing is that the elements in that array should be sorted order.

→ An element which is to be search from the list of elements sorted in an array A[0-(n-1)] is called key elements.

procedure :-

Let A[m] be the mid element of an array 'A' then there are 3 conditions that needs to be tested while searching the array using this method.

i) if key = = A[m] then
   desired element is present in the list.

ii) otherwise if key < A[m] then
   search the left sub list (high = mid - 1)

iii) otherwise if key > A[M] then
   search the right sub list (low = mid + 1)

E.g:- Consider a list of elements 10, 20, 30, 40, 50, 60, 70.

0	1	2	3	4	5	6
10	20	30	40	50	60	70

key = 60

low = 0

high = 6

$$\text{middle} = \frac{\text{low} + \text{high}}{2} = \frac{0+6}{2} = 3 = M$$

mid = A[3] = A[M]

here A[M] = 3 = 40

40 not equal to 60.

but 60 > 40 that is key > A[M]

Search right side list

so, 50 60 70
↓ ↓ ↓
0 1 2

$m = \frac{0+2}{2} = 1$

∴ A[M] = A[1] = 60

∴ key = 60 = A[M]

∴ So the desired element is occurred in 5 position.

Binary Search Algorithm:-

```
Algorithm BinarySearch (A, key, n)
// Searching an element from a list
// input: = A[n], key
// output: If key found print the key.
{
    low: = 0
    high: = n-1
    m: = (low+high)/2
    if (key == A[M]) then
        return key;
```

```
    else if (key < A[M]) then
    {
        high = M-1;
    }
    else
    {
        low = m+1;
    }
}
```

Iterative Method:-

```
Algorithm BinarySearch (A, key, n)
// searching an element from a list.
// Input: A(n), key
// output: If key found print the key otherwise
                                    return -1
{
    low := 0
    high := n-1
    while (low <= high)
    {
        M := high+low / 2
        if (key == A[M]) then
            return m;
        else if (key < A[M])
        {
            high := m-1;
        }
        else
        {   low := m+1;
        }return -1
} }
```

Recursive Algorithm :-

```
Algorithm BinarySearch (A, x, i, j, n)   // i → low, j → high
// Searching an element from al.
// input : A[n], x, n ≥ j ≥ i
// output : if key found then return m, otherwise
                                        return -1
{
    if (i == j) then
    {
        if (x == A[i]) then
        {
            return i;
        }
        else
        {
            return -1;
        }
    }
else
{
    m = ((i+j)/2);
    if (x == A[M]) then {
            return m;
    }
    else if (x < A[M]) then {
        return BinarySearch (A, x, i, m-1, n);
    }
```

```
else {
    return BinarySearch(A, x, m+1, j, n);
  }
}
}
```

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

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

$T(1) = 1$

Analysis of Binary Search Algorithm:-

→ The basic operation in binary search is comparison of search key with the array elements.

→ To analyse efficiency of binary search we must count the no. of times the search key get compared with the array elements.

→ The comparison is also called a three-way comparison bcz algorithm makes the comparison to determine whether key is smaller, equal or greater than A[M].

→ In the algorithm after one comparison the list of n elements is divided into n/2 sublists.

→ The worst case efficiency is that all algorithm compares all the array elements for searching the desired element.

→ Hence, the worst case time complexity is given by $T(n) = T(n/2) + 1$, $T(1) = 1$

$T(n/2)$ = Time required to compare left/right sublist.

1 := One comparison made with middle element.

Backward substitution :-

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

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

$T(n/4) = T(n/8) + 1$

$T(n/8) = T(n/16) + 1$

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

$= T(n/4) + 2$

$= T(n/8) + 3$

$= T(n/16) + 4$

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

$k = 4,$

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

$2^k = n$

$k = \log n$

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

$= 1 + \log n$

T.C = $O(\log n)$

Advantages of Binary Search :-

→ It is an optimal searching algorithm which we can search the desired element very efficiently.

Dis-adv of B.S :- This algorithm requires the list to be sorted then only binary search method is applicable.

Applications of Binary Search :-

1) The Binary search is an efficient searching method & it is used to search desired record from database.

2) For solving non-linear eqn's with one unknown, Binary search method is used.

Time complexity of Binary Search :

Best case : O(1)

Average case : O(log n)

Worst case : O(log n)

Quick Sort :-

→ It is a sorting algorithm that uses the

divide & conquer strategy.

The 3 steps of quicksort are as follows.

i) divide :- Divide the elements of an array into 2 subarrays based on pivot element. All the elements that are less than pivot should be left subarray & all the elements that are greater than pivot should be in right subarray.

ii) Conquer :- Recursively sort the 2 subarrays.

iii) Combine :- Combine all the sorted elements in a group to form a list of sorted elements.

→ In merge sort the division of array is based on the positions of array elements, but in quick sort division is based on actual value of the element.

[Quick Sort]

E X A M P L E

↓ ↓ ↓

P i j

$A[i] \leq p$ $x \leq E$ x

$A[j] \geq p$ $E \geq E$ ✓ $(j--)$

- $i < j$
- $A[i] \leq p$ then $i++$
- $A[j] \geq p$ then $j--$
- $i < j$ ↓ swap $[A[i], A[j]]$
- else swap $(pivot, A[j])$

E X A M P L E

↓ ↓ ↓

P i j

$A[j] \geq p \Rightarrow L \geq E$ (✓) $(j--)$

E X A M P L E

↓ ↓ ↓

P i j

$A[j] \geq p \Rightarrow P \geq E$ $(j--)$

E X A M P L E

P i j

$A[j] \geq p \Rightarrow M \geq E$ ✓ $(j--)$

E X A M P L E

P i j

$A[j] \geq p \Rightarrow A \geq E$ (x)

$i < j$ then swap $(A[i], A[j])$

E A X M P L E

P i j

A[i] ≤ p ⇒ A ≤ E ∨ (i++)

E A X M P L E
↓ ↓
P i j

A[i] ≤ p ⇒ X ≤ E[x]

A[j] ≥ p ⇒ X ≥ E(∨)

[j--]

E A X M P L E
↓ ↓ ↓
P j i

A[j] ≥ p ⇒ A ≥ E (x)

swap [A[j], p]

A | E | X M P L E

X M P L E
↓ ↓ ↓
P i j

i < j

A[i] ≤ p , M ≤ X (i++)

X M P L E

↓ ↓ ↓

P i j

A[i] ≤ p , P ≤ X ✓ (i++)

X M P L E

↓ ↓ ↓

P i j

A[i] ≤ p , L ≤ X[v]

(i++)

X M P L E

↓ ↓

P ij

A[i] ≤ p , E ≤ X (v)

(i++)

A[j] ≥ p , E ≥ X (x)

i < j (x)

swap (A[j], p)

E M P L (x)

E M P L

↓ ↓ ↓

P i j

A[i] ≤ p , M ≤ E x

E M P L

↓ ↓ ↓

P i j

$A[j] \geq p$, $L \geq E \vee (j--)$

E M P L

↓ ↓ ↓

P i j

$A[j] \geq p$, $p \underset{(\vee)}{\geq} E \; (j--)$

E M P L

↓ ↓

P ij

$A[j] \geq p \Rightarrow M \geq E \vee (j--)$

E M P L

↓ ↓

Pj i $\quad A[j] \geq p$, $E \geq E \vee (j--)$

$i < j \; x \; [swap(A[j], p)$

E | M P L

M P L $\quad \Rightarrow A[i] \leq p$, $p \leq M(x)$

↓ ↓ ↓

P i j $\quad A[j] \geq p$, $L \geq M(x)$

$i < j$

↓

swap $(A[i], A[j])$

M L P
↓ ↓ ↓
p i j

$i < j$

$A[i] \leq p$, $L \leq M \vee (i++)$

M L P
↓ ↓
p ij

$A[i] \leq p$, $P \leq M$ (x)

$A[j] \geq p$, $P \geq M \vee (j--)$

M L P
↓ ↓ ↓
p j i

$A[j] \geq P$ $(L \geq M)$ x

$i < j$ x

swap(A[j], p)

L | M | P

∴ The final sorted list

A E E L M P X

Quick Sort :-

```
Algorithm Quicksort (A[0..n-1], low, high)
// Sort the elements in an array A[0..n-1] in
ascending order.
// low-left most element in the list, high-Right
most ele. in the list, unsorted array A[0,..n-1];
// sorted in ascending order
{
    if (low < high) then
        m := partition (A[low .. high]) // m is mid of
                                           the array
        Quicksort (A[low .. m-1])
        Quicksort (A[m+1 .. high])
}
```

Recursive Quicksort Alg :-

It has 2 algorithms. One is Quicksort, another one is partition.

1) Quick Sort :-

→ Partition performs arrangement of the elements in ascendingorder. The recursive Quick routine is for dividing the lists in two sub lists.

3| Partition:-

```
Algorithm Partition (A[low.. high])
// Partition array using first element as pivot
   element
// A subarray low as left most index of array & high
as right most index of array.
// Partition of array 'A' is done & pivot occupies its
   proper position & correct index of it is return
{
   pivot := A[low];
   i := low+1, j := high;
   while (i<j) do
   {
      while (A[i] <= pivot) do
      {
         i := i+1;
      }
      while (A[j] >= pivot) do
      {
         j := j-1;
      }
      if (i<j) then
      {  swap (A[i], A[j]);
      }
}
```

```
        swap (pivot, A[j]);
        return j;
}
```

→ The partition algorithm is called to arrange the elements such that all the elements that are less than "pivot" are at left side of pivot & all the elements that are greater than pivot are at right of the pivot.

→ In other words pivot is occupying its proper position & the partition list is obtained in an order.

Analysis of Quick Sort Algorithm:-

In Best case, the given list is divided into 2 equal halfs. The recurrence relation for best case is

T(n) = Time Complexity to sort sublists + Time required for partitioning & combining

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

Backward substitution method:-

$$T(n/2) = 2T(n/4) + n/2 \quad -(2)$$

$$T(n/4) = 2T(n/8) + n/4 \quad -(3)$$

$$T(n/8) = 2T(n/16) + n/8 \quad -(4)$$

sub. ②, ③ & ④ in ①

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

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

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

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

$= 8\left[2T(n/16) + n/8\right] + 3n$

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

$k = 4 \Rightarrow 2^k T(n/2^k) + kn$

If $n <= 1$
$T(0) = 0$
$T(1) = 0$

let $n/2^k = 1 \Rightarrow n = 2^k$

$\log n = k \log 2$

$\log n = k$

$= 2^{\log n} T\left(n/2^{\log n}\right) + n \log n$

$= nT(n/n) + n \log n$

$= nT(1) + n \log n \quad [\because T(1) = 0]$

$= n \log n$

$T(n) = O(n \log n)$ — Best case, Average case

Worst case:- The worst case for Quick sort occurs when the pivot is a minimum or maximum of all the elements in the list.

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

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

Randomized Quicksort:- (Las Vegas Algorithm)

→ The worstcase for quicksort depends upon how we select our partition or pivot element.

→ If an i/p array is already sorted & if we select 1st or last element as pivot for applying the quicksort method, then it results in worst case.

→ To bring improvement over the quicksort, choice of pivot is key factor. So, we can use following choices.

i) Use the middle element of array as pivot.

ii) Use random element of array as pivot.

iii) Take the median of three elements (1st, last & middle).

→ The quicksort works better in average case but its performance is very poor in worst case.

→ Hence from above mentioned three choices randomization is the better choice to improve the performance of quicksort in worst case.

Algorithm RQuicksort (A, low, high)

{

```
if (low < high) then
{
    if ( (high-low)>5) then
        swap (A,( random() mod (high-(low+1))+
                                low), low);

    k:= Partition (A, low, high);
    RQuicksort (A, low, k-1);
    RQuicksort (A, k+1, high);
}
}
```

→ Average time complexity of Randomized Quicksort algorithm is : O(nlogn)

Merge Sort :-

→ Merge sort is a sorting algorithm that uses divide & conquer strategy. In this method, division is dynamically carried out.

→ Let 'S' be an array.

Divide :-

→ Partition array in 'S' into two sub-arrays S1 & S2 with n/2 elements each.

Conquer :-

→ Then sort sub-arrays S1 & S2.

Combine :-

Merge S1 & S2 into a unique sorted group.

E.g :- 70, 20, 30, 40, 10, 50, 60

1 2 3 4 5 6 7 [index]

$$mid = \frac{low + high}{2} = \frac{1+7}{2} = 4$$

Left subtree - low to mid

R.S. - [mid+1 - high]

0 ... n-1

mid = 3.5 = 4

1 ... n

mid = 3.5 = 3

70 20 30 40 | 10 50 60

$mid = \frac{1+4}{2} = 2$ | m = 2

70 20 | 30 40 | 10 50 | 60

m = 1 | mid = 3/2 = 1.5 = 1 | m = 1

70 | 20 | 30 | 40 | 10 | 50 | 60

i j

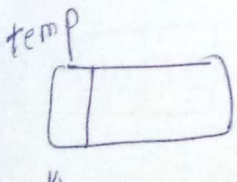
i = low

70 | 20 | 30 | 40 | 10 | 50 | 60

j = mid+1 (1st element in R.S.T)

k := low (temp)

↓ (temporary)



if (i < j)

if (A[i] < A[j]) then i = i + 1, K := K + 1,

temp[K] = A[i]

if(A[i] ≥ A[j]) then temp[K] := A[j],

j := j + 1, K := K + 1

Merge Sort :-

Eg :-

Combine :-

70	20	30	40
i	j	i	j

A[i] > A[j]

temp[k] = A[j]

temp = 20

j = j+1

k = k+1

20	70
i	

A[i] < A[j]

i = i+1

k = k+1

temp[k] = A[i]

30	40
j	

20	70
↓ i	

30	40
j	

i < j => A[i] < A[j]

temp[k] = A[i] => temp = 20

i := i+1, k := k+1.

20	

70	30	40
i	j	

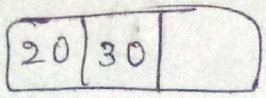
=> A[i] > A[j] => temp[k] = A[i],

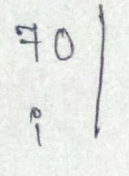
j = j+1, k = k+1

temp = 30

Continued...

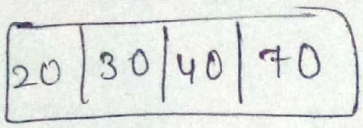
→



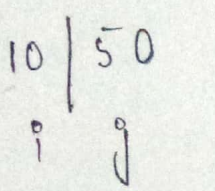


A[i] > A[j] ⇒ temp = A[j] ⇒ j = j+1;
k = k+1

temp = 40



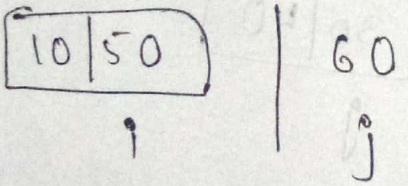
Eg:



i < j ⇒ temp[k] := A[i]

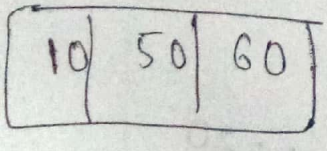
temp = 10

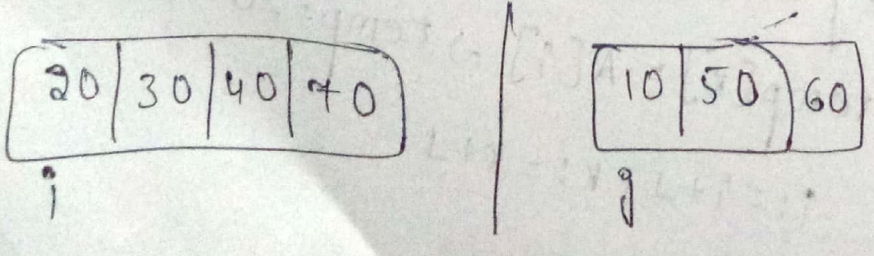
i := i+1; k := k+1



A[i] < A[j] ⇒ temp = 50

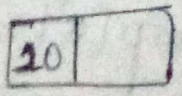
i := i+1, k := k+1

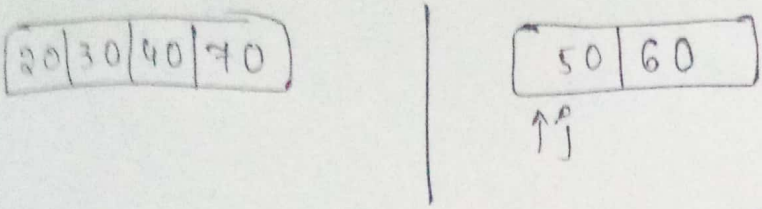




A[i] > A[j] ⇒ temp[k] = A[j] = 10

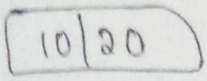
k := k+1; j = j+1

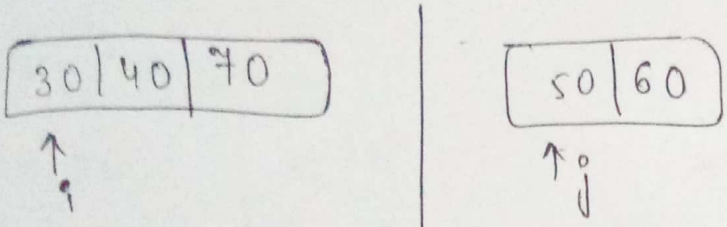




$A[i] < A[j] \Rightarrow temp = A[i] = 20;$

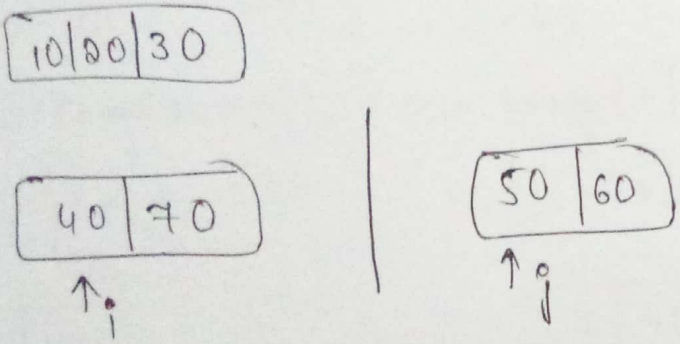
$i := i+1, k := k+1$



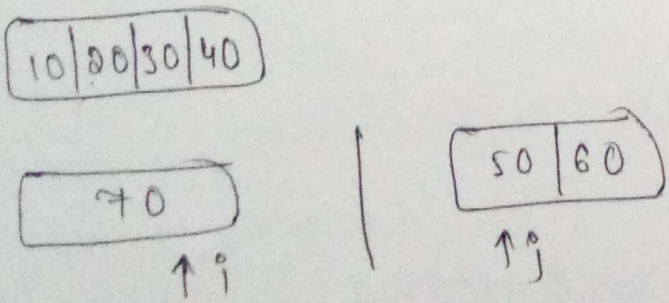


$A[i] < A[j] \Rightarrow temp = A[i] = 30;$

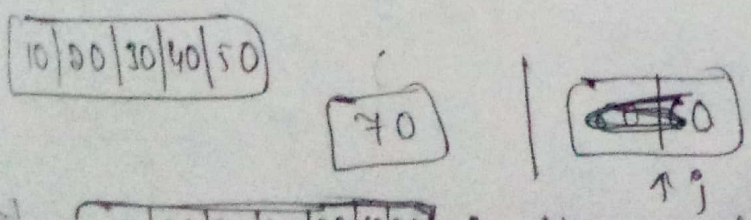
$i := i+1, k := k+1$



$A[i] < A[j] \Rightarrow temp = 40$



$A[i] > A[j] \Rightarrow temp = 50$



$\Rightarrow$ 10 20 30 40 50 60 70 is the final one.

Recursive Algorithm for Merge Sort :-

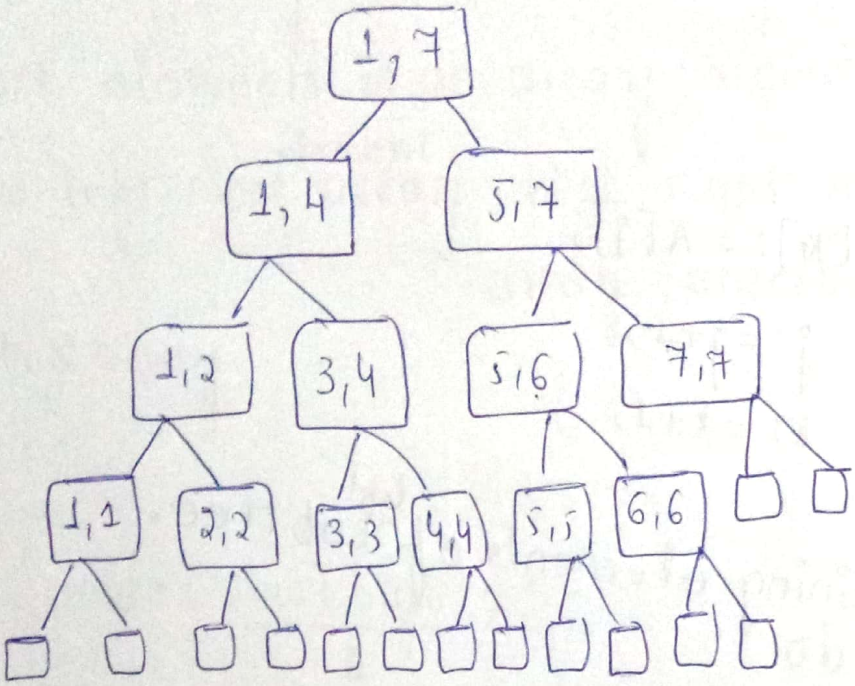
```
Algorithm MergeSort (A, low, high)
// sort elements in an array (ascending order)
// low - left most element in array, high - right most element in
                                    array, unsorted array.
// sorted array
{
    mid := (low + high) / 2 ;
    MergeSort (A, low, mid);
    MergeSort (A, mid+1, high);
    Merge (A, low, mid, high);
}
```

Algorithm for merge :-

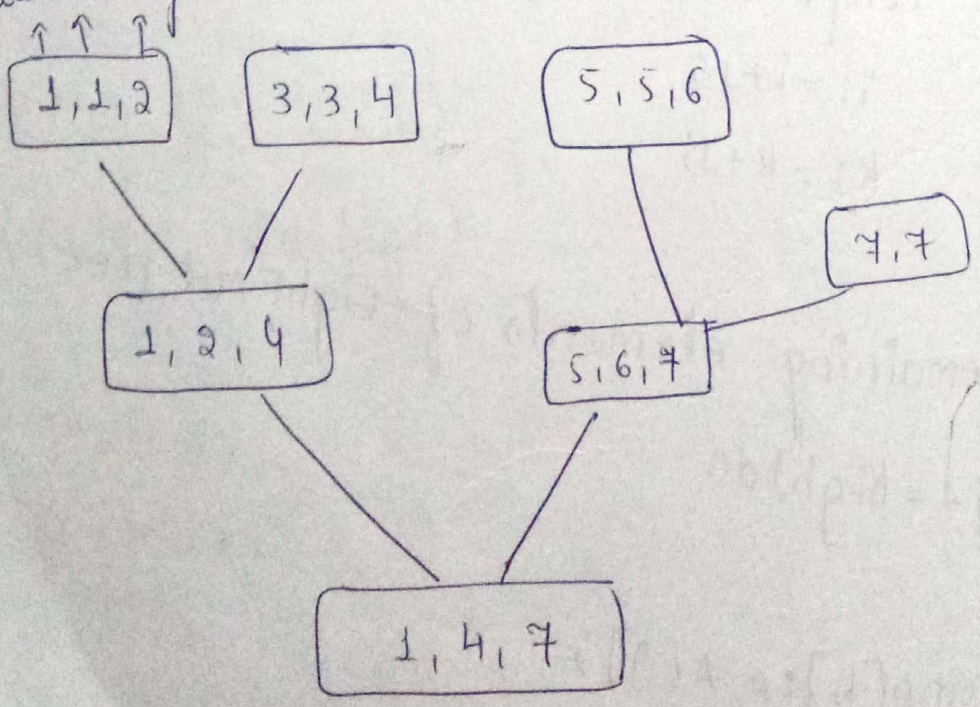
```
Algorithm
Merge (A, low, mid, high)
{
    i := low;
    j := mid + 1;
    k := low;
    while (i <= mid and j <= high) do
    {
        if (A[i] <= A[j]) then
        {
```

```
        temp[k] = A[i];
            i := i+1;
            k := k+1;
    }
    else
    {
        temp[k] := A[j];
            j := j+1;
            k := k+1;
    }
}
// copy remaining elements of left subtree.
while (i<=mid) do
{
    if (A[i]<=A[j]) then
    {
        temp[k] := A[i];
        i := i+1;
        k := k+1;
    }
}
// copy remaining elements of right sub tree.
while (j<=high) do
{
    temp[k] := A[j];
        j := j+1;
        k := k+1;
}
}
```

→ Draw the tree of Recursive calls of Mergesort, & Merge functions.



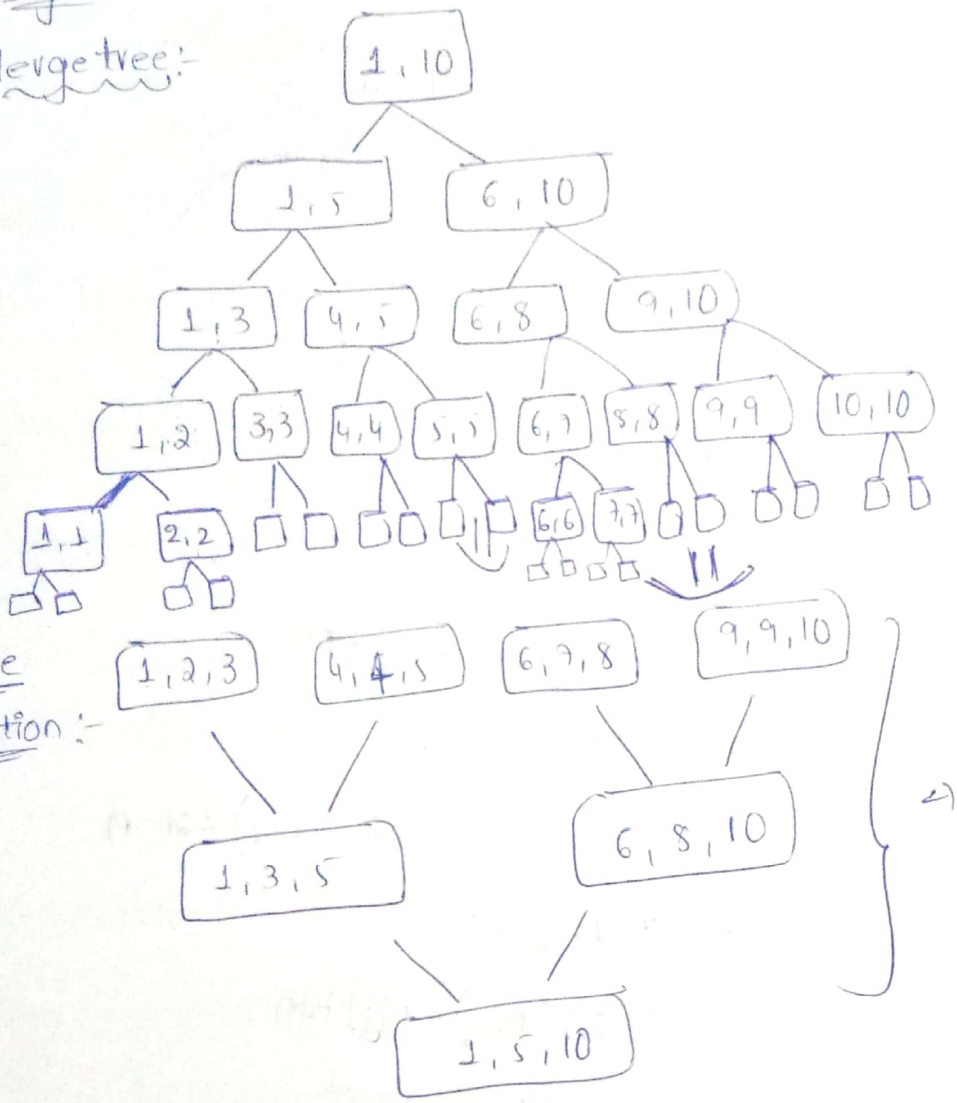
starts from leaf node ⇒ (low, mid, high)



Eg: 31, 28, 17, 65, 35, 42, 86, 25, 45, 52
1 2 3 4 5 6 7 8 9 10

Mergesort:-

Mergetree:-



Merge function:-

→ 8 mappea

Analysis of Mergesort:-

→ In mergesort the given list is divided into two sub lists each containing $n/2$ elements. So the recurrence relation is given form of

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

time complexity is n.

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

time complexity = $O(n \log n)$

Backward Substitution:-

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

$T(n/4) = 2T(n/8) + n/4 \rightarrow (1)$

$T(n/8) = 2T(n/16) + n/8 \rightarrow (2)$

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

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

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

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

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

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

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

$= 8T(n/8) + 3n$

$= 8(2T(n/16) + n/8) + 3n$

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

$k = 4$

$\Rightarrow = 2^k T(n/2^k) + kn$

$\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$ $\quad (T(1) = 0)$

$= 2\log_2 n\, T(\frac{n}{n}) + \log n$

$= 2\log n\, T(1) + (\log n)\, n$

$= 2\log n\, T(1) + (\log n)\, n$

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

$= n + \log n \cdot n$

$= n + n\log n$

$= O(n \log n)$

$T(n) = 2\log n\, T(1) + n\log n = n\log n = O(n\log n)$

The time complexity of mergesort:-

Best case - O(n log n)

Average case - O(n log n).

worst case - O(n log n)

→ Before the execution - performance Analysis

After the execution - performance measurement.

Performance measurement:-

→ Quicksort & mergesort were evaluated on a workstation 10/30 in both cases the recursive version is used.

→ for the Quicksort the partition fn. was altered to compute the median of three rules.

→ each data set consisted of random integers in the range (0, 1000), for each n 50 random datasets were use.

→ Table 1 displays the average computing times for two sorting algorithms.

→ Table 2 displays the worst case computing times for two sorting algorithms.

n	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
merge sort	72.8	167.2	275.1	378.5	500.6	607.6	723.4	811.5	949.2	1073.6
Quick sort	36.6	85.1	138.9	205.7	269.0	339.4	411.0	487.7	556.3	645.2

Average computing time for two sorting algorithms on random inputs.

n	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
merge sort	103.7	206.4	335.2	422.1	589.9	691.3	794.8	889.5	1067.2	1167.6
Quick sort	41.6	97.1	158.6	244.9	296.9	383.8	497.3	569.5	616.3	738.1

Worst case computing times for 2 sorting algorithms:

→ By computing the two algorithms Quicksort is better than mergesort in time complexity.

→ By scanning above two tables, we can say that quick sort is faster than merge sort for all values.

→ Even though both algorithms require O(nlogn) time on the average, quick sort usually performance on practise.

→ Is mergesort stable sorting algorithm?

Yes. Mergesort is stable sorting algorithm.

↳ A sorting alg. is said to be stable if it preserves the ordering of similar elements after

applying sorting methods & merge sort is a method which preserves this kind of ordering. Hence merge sort alg. is stable sorting algorithm.

→ Explain Merge sort { ↓ after that }

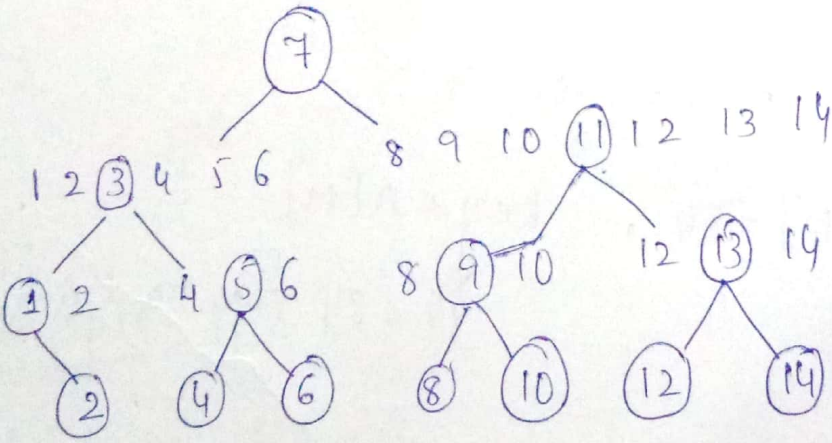
8) Is quicksort is stable sorting algorithm?

No, quicksort is not a stable sorting algo. Because it swaps non-adjacent elements.

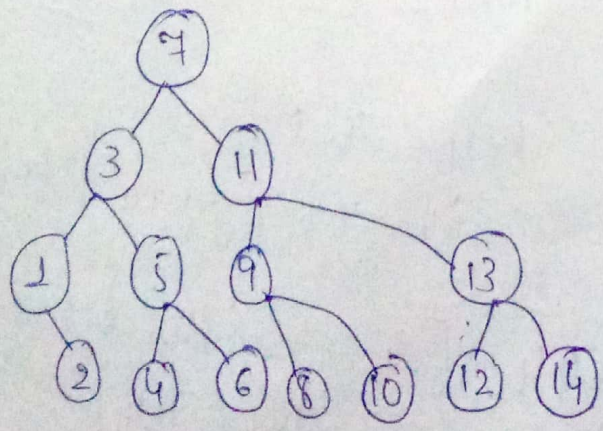
9) Draw a binary decision tree for a binary search n=14.

↳ we draw based on mid-value of 'n'.

n = 14



⇒



→ Determine the no. of passes required to search the element 44 in the following list of elements.

5, 13, 17, 33, 38, 44, 77, 84, 90

0 1 2 3 4 5 6 7 8

key - 44 ,

$$mid = \frac{0+8}{2} = 4 \checkmark$$

A[M] = 38

key > A[M]

44 > 38 [go to R.S.L]

44 77 84 90 } mid+1 to high

↓5 ↓6 ↓7 ↓8

$$mid = \frac{5+8}{2} = 7 \checkmark$$

A[M] = 84 , key < A[M]

44 < 84 [go to L.S.L]

44 77

↓5 ↓6

$$mid = \frac{5+6}{2} = 6 \checkmark$$

key < A[M]

44 < 77 [go to L.S.L]

A[M] = 44 [at this high & low element]

mid = 5+5/2 = 5 [44]

→ [key found].

Defective Chess Board :-

↓

n x n - n is the power of 2.

$2^k \times 2^k = 2^{2k}$ squares

↓ rows ↓ columns

with atleast one defective square then,

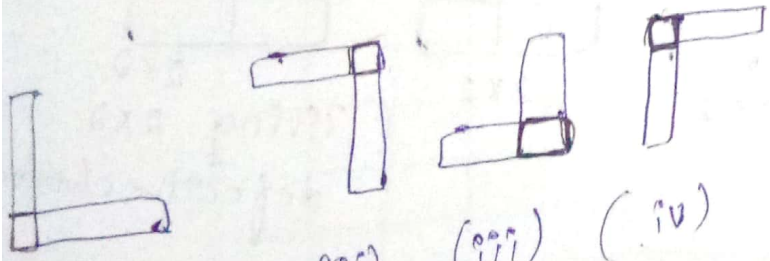
non - " " $= 2^{2k} - 1$

Cover non-defective squares with L-shaped tiles

↓

triominoes (or)

trominoes



(i) (ii) (iii) (iv)

conditions :-

1) it should be in the form of $(2^{2k} - 1)/3$

for example :-

k	$2^{2k} - 1$
0	0
1	3
2	15
3	63

2) No two triominoes should overlap each other.

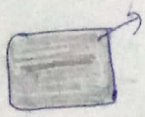
It is not applicable for defective square.

$k=0 \Rightarrow 2^{2k}$ - 1 square - 1×1

$k=1 \Rightarrow 2^{2k}$ - 4 squares - 2×2

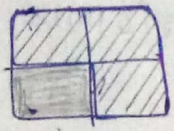
$k=2 \Rightarrow 2^{2k}$ - 16 squares - 4×4

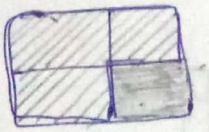
$k=3 \Rightarrow 2^{2k}$ - 64 squares - 8×8

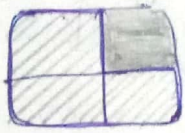


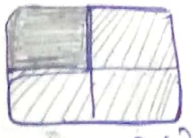
$k=0 \Rightarrow 1 \times 1$

$k=1 \Rightarrow$



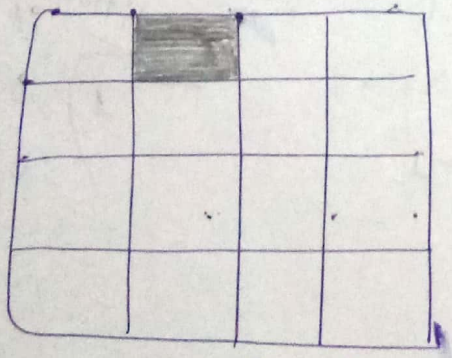






[Tiling 2x2 defective chessboard

$k=2 \Rightarrow$



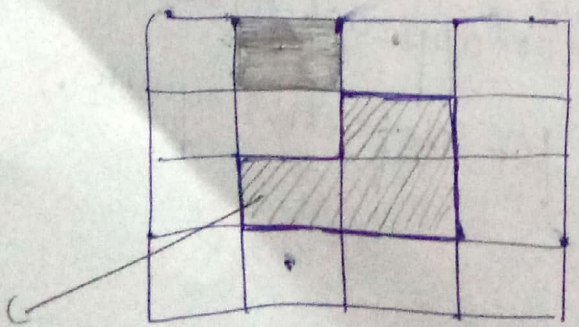
(4x4)

[tiling 4x4 d.c.b]

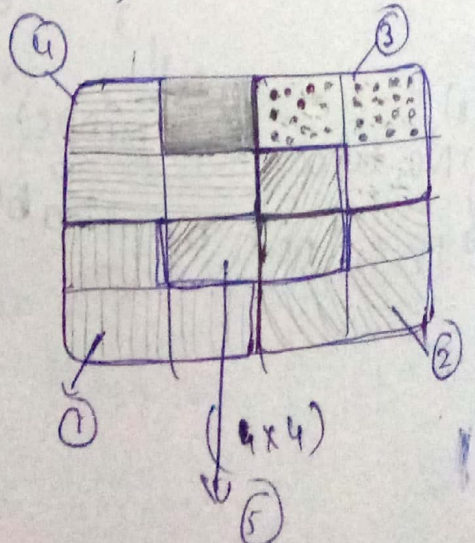
4x4 → 2x2

layer

smaller

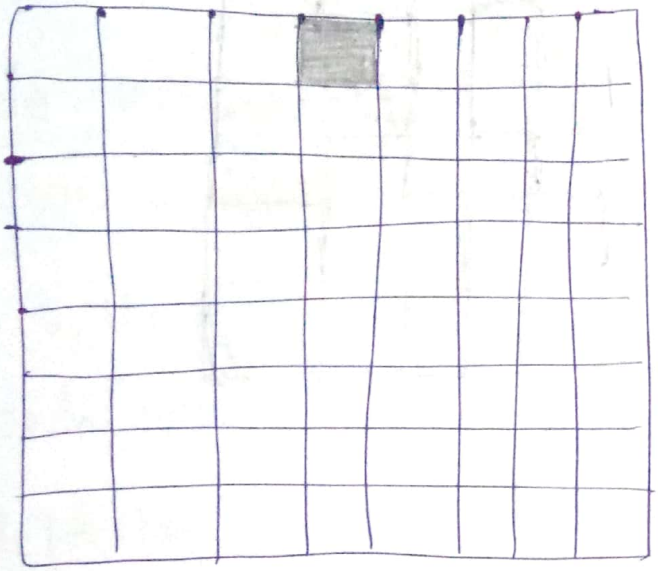


(4x4)

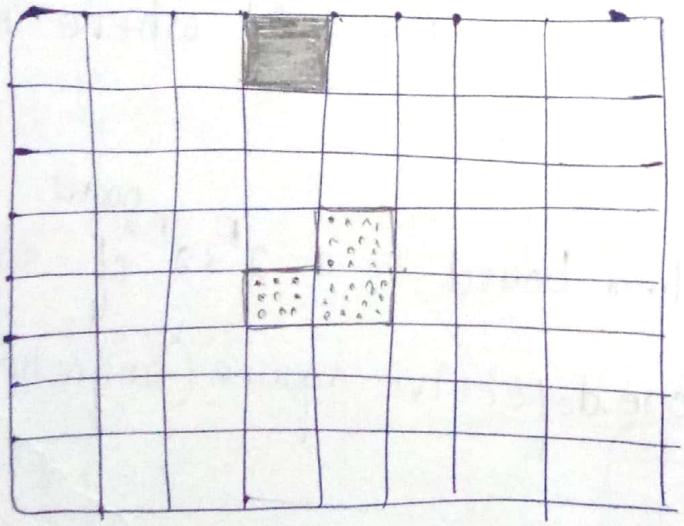


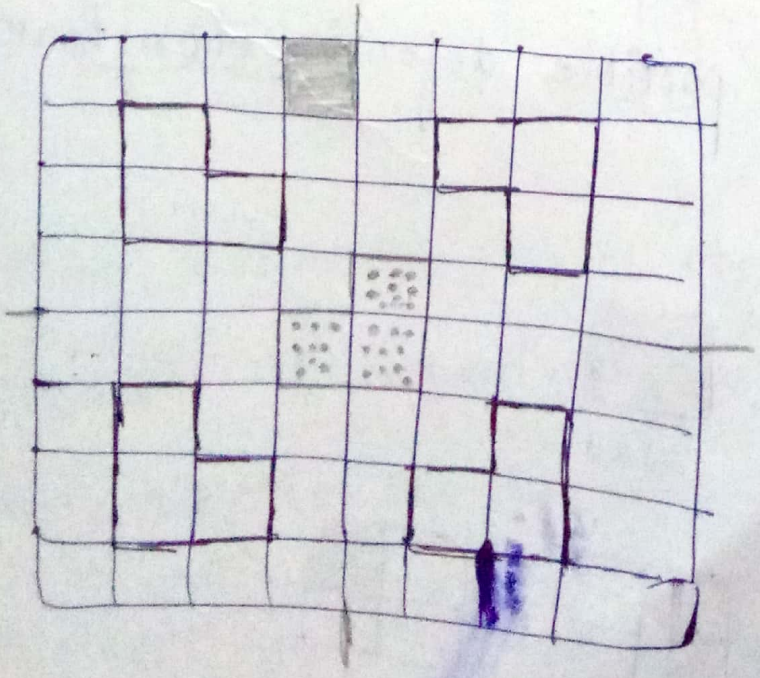
(4x4)

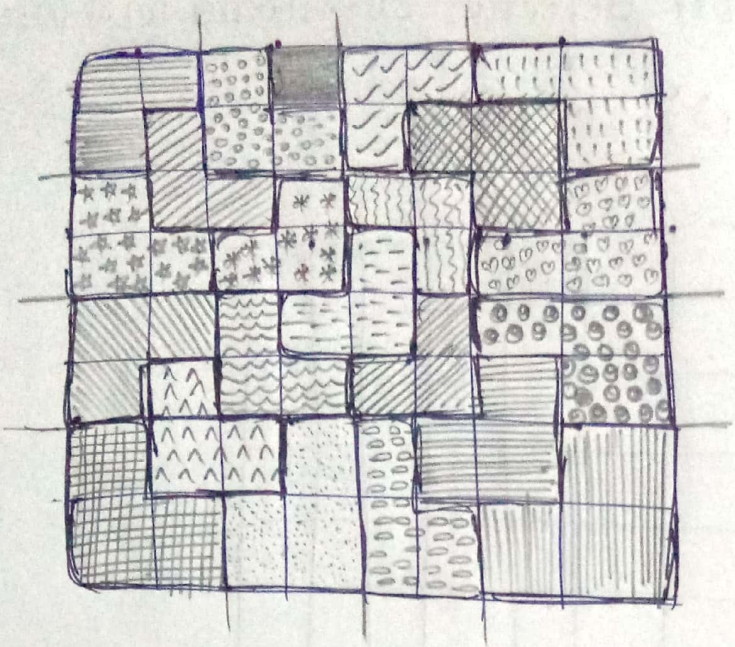
→ Consider a 8x8 defective chessboard with a defective position (0,3). s.t. tiling.



8 x 8
↓
4 x 4
↓
2 x 2







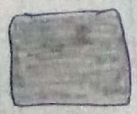
Defective chess board :-

→ A chessboard is an 'n x n grid where 'n' is a power of 2.

→ A defective chess board is a $2^k \times 2^k$ board of squares with exactly one defective square (anywhere on the chess board.

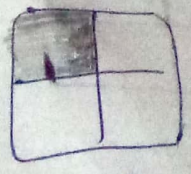
→ Some of the possible defective chess board for $k \leq 2$.

$k = 0$

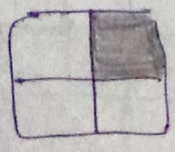


1 x 1

$k = 1$

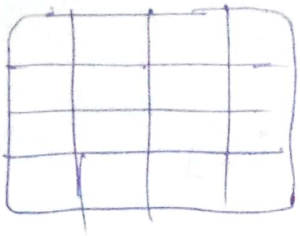


2 x 2



2 x 2

$k = 2$

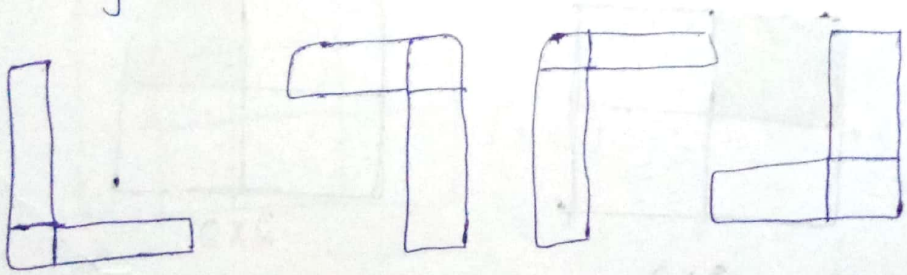


→ for $k = 0$, there is only one possible defective chessboard.

→ for any k, there are exactly 2^{2k} different defective chessboards (squares)

→ in the defective chessboard prblm, we are required to tile a defective chessboard using triominoes or trominoes. (L-shapled tiles)

→ L-shaped tiles are made up of 3-squares (or) 3 tiles & they are



→ In this tiling, 2 trimonies may not overlapp, trimonies should not cover the defective squares & trimonies must cover all other non-defective square.

→ With this constraint, the number of trimonies to be ~~be~~ be $2^{2k} - 1/3$.

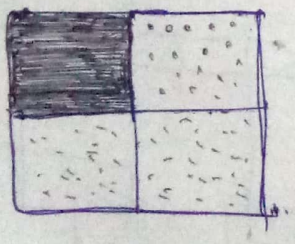
→ Defective square - 1

Non- " " $= 2^{2k} - 1$.

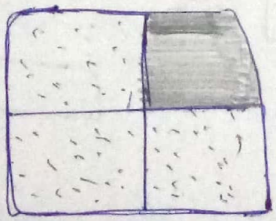
→ A defective chessboard with $k=0$ is easily tiled as it has no non-defective squares. So, the no. of triominoes used in the tiling is zero.

→ When $k=1$, there are exactly 3 non-defective squares & these squares are covered using triminoes in one of the orientation (L-shaped tiles) as shown in the above figure.

$k=1$



2 x 2



2 x 2

→ The divide & conquer method leads to an elegant solution to the defective chessboard problm.

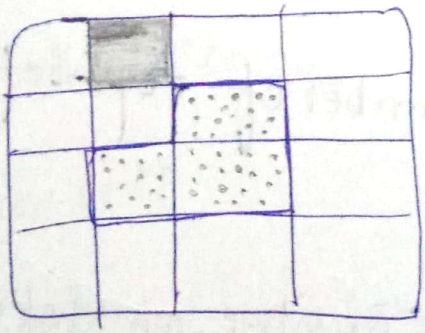
→ The method suggests reducing the problm of tiling of $2^k \times 2^k$ defective chessboard to that of tiling smaller defective chessboards.

→ A natural partioting of a $2^k \times 2^k$ chessboard would be into 4 $(2^{k-1} \times 2^{k-1})$ as shown in figure.

$2^{k-1} \times 2^{k-1}$	$2^{k-1} \times 2^{k-1}$
$2^{k-1} \times 2^{k-1}$	$2^{k-1} \times 2^{k-1}$

→ Such a partitioning is done, only one of the four chessboard has it defect & we can tile that defective when board.

→ to convert the remaining 3 boards to defective boards, We place a triomino at the corner formed by 3 tiles.



(4×4)

→ The above figure shows the placement for the stage when the defect in the original $2^k \times 2^k$ chessboard falls into upper left $2^{k-1} \times 2^{k-1}$ chessboard.

→ We can use this partition technique recursively to tile the enter $2^k \times 2^k$ defective chessboard.

→ The recursion terminates when the chess board size has been reduced to 1×1.

Recursive algorithm for defective chessboard:-

```
Algorithm TileBoard (topRow, topcolumn, defectrow,
                     defectcolumn, size)

// topRow is row number of top-left corner of
   board
// topcolumn is col number of top-left corner of
   board

// defect Row is row number of defective square
// defect colomn is col number of defective square
// size is length of one side of chessboard.
{
    if (size = 1) return;
    tile ToUse := tile++;
    quadrant size := size/2;
    // top-left Quadrant
```

```
if (defectRow < topRow + Quadrantsize && defectColumn
    < topcolumn + Quadrantsize) then

//defect is in this quadrant
TileBoard (topRow, topcolumn, defectRow, defectColu
                                              -mn,
        Quadrantsize);

else
{
  //no defect in this Quadrant
  //place a tile in bottom right corner.
  board [topRow + Quadrantsize - 1] [topcolumn +
        Quadant size - 1] := tileTouse;

  //tile the rest
  TileBoard (topRow, topColumn, topRow + Quadrantsize
                                              - 1,
      topcolumn + Quadrantsize - 1, Quadrantsize);

// code for remaining 3 quadrants is similar.
```

→ Above algorithm given the pseudo code for divide & conquer strategy.

→ The pseudocode consists of 2 global variables board & tile. board - top left corner i.e. [0][0]
tile - is always 1 initially & increement representing the next square.

→ Board is a 2-dimensional array that represents the chess board & board [0][0] represents top-left corner square of the chess board.

→ Tile → global variable with initial value '1' given the index of the next tile to use.

→ Initial values for the algorithm is TileBoard(0, 0, drow, dcolumn, size)

where size = 2^k,
drow & dcolumn are the row & column index of the defective square.

→ The no. of tiles needed to tile the defective chess board is $\frac{(size)^2 - 1}{3}$.

Analysis of defective chessboard algorithm:-

Let $T(k)$ denotes the time taken by tile board to tile a $2^k \times 2^k$ defective chess board.

→ when $k=0$, size $=1$ & constant 'd' amount of time is spent.

→ when $k>0$, 4 recursive calls are made. This calls takes $4T(k-1)$ time. In addition to this time, we have to calculate the time spent for if-conditionals, tiling remaining 3 non-defective squares & it is denoted by c.

→ The following recurrence relation is

$$T(k) = \begin{cases} d & k=0 \\ 4T(k-1)+c & k>0 \end{cases}$$

$$T(k) = 4T(k-1)+c$$

$$T(k-1) = 4T(k-2)+c$$

$$T(k-2) = 4T(k-3)+c$$

$$T(k) = 4\left[4T(k-2)+c\right]+c$$

$$= 4^2 T(k-2)+4c+c$$

$$= 4^2(4T(k-3)+c)+4c+c$$

$= 4^3 T(k-3) + 4^2 c + 4c + c$

$k=3,$

$= 4^k T(0) + 4^{k-1} c + 4^{k-2} c + \cdots + c$

$= 4^k d + c\left(4^{k-1} + 4^{k-2} + \cdots + 1\right)$

$a^n - 1 = (a-1)\left(a^{n-1} + a^{n-2} + \cdots + a + 1\right)$

$a = 4, n = k$

$4^k - 1 = (4-1)\left(4^{k-1} + 4^{k-2} + \cdots + 4 + 1\right)$

$4^{k-1} + 4^{k-2} + \cdots + 1 = \frac{4^k - 1}{3}$

$\Rightarrow T(k) = 4^k d + c\left(\frac{4^k - 1}{3}\right)$

Time complexity = $O(4^k)$

↑ (No. of tiles needed)

→ Time complexity of defective chessboard is $O(4^k)$ [Best case, Worst case & Average case]

→ We must spend atleast $O(1)$ time placing each tile.

→ So, we cannot obtain asymptotically faster algorithm than divide & conquer.

Finding the maximum & minimum :-

→ The max, min prblm in algorithm analysis is finding the maximum & minimum value in an array.

→ To find the maximum & minimum no's in a given array 'a' of size 'n' straight forward algorithm is 'a[]' used.

Algorithm :-

```
Algorithm straightMaxMin (a, n, max, min)
// set max as maximum & min as minimum
{
    max := min := a[1];
    for i := 2 to n do
    {
        if (a[i] > max) then
            max := a[i];
        if (a[i] < min) then
            min := a[i];
    }
}
```

for ele

W.C
-30, 20, 10
-2(n-1)

10, 20, 30

B.C - (n-1)

→ (n-1)

→ (n-1)

2(n-1)

Best case, Worst case & Average case.

→ In analysing the time complexity of algorithm we concentrate on no. of elements comparisions so straight maxmin requires 2(n-1) elements comparision in the b.c., worst & Average case.
case

(1)

Algorithm StraightMaxMin (a, n, max, min)

// set max as maximum & min as minimum

```
{
    max := min := a[1];
    for i := 2 to n
    {
        if (a[i] > max) then
            max := a[i];
        else
            min := a[i];
    }
}
```

→ Best case occurs when the elements are in increasing order, the no. of elements comparision is (n-1).

10, 20, 30
(n-1)

→ The worst case occurs when the elements are in decreasing order, the no. of elements comparision is 2(n-1).

30, 20, 10
2(n-1)

→ A divide & conquer algorithm for this problm is

Let P = (n, a[i] ... a[j]) denotes an arbitary instance of the problm where 'n' is no. of elements in the list a[i] .. a[j]. We have to find maximum & minimum no. from the list.

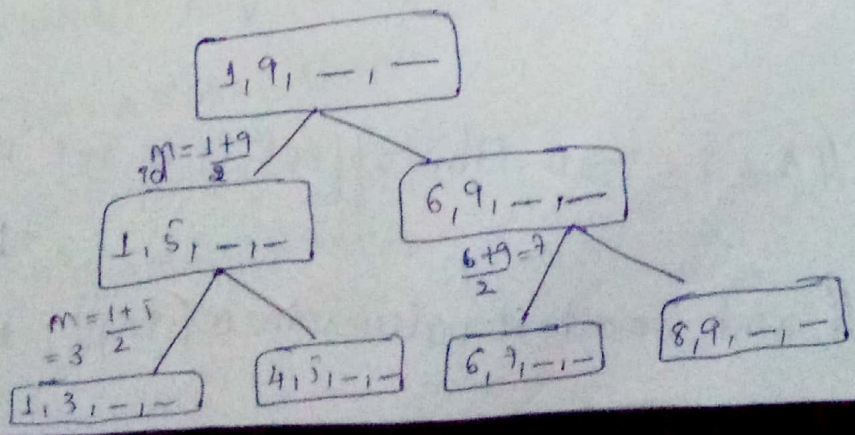
→ for n ≤ 2, the problem can be solved by making one comapision.

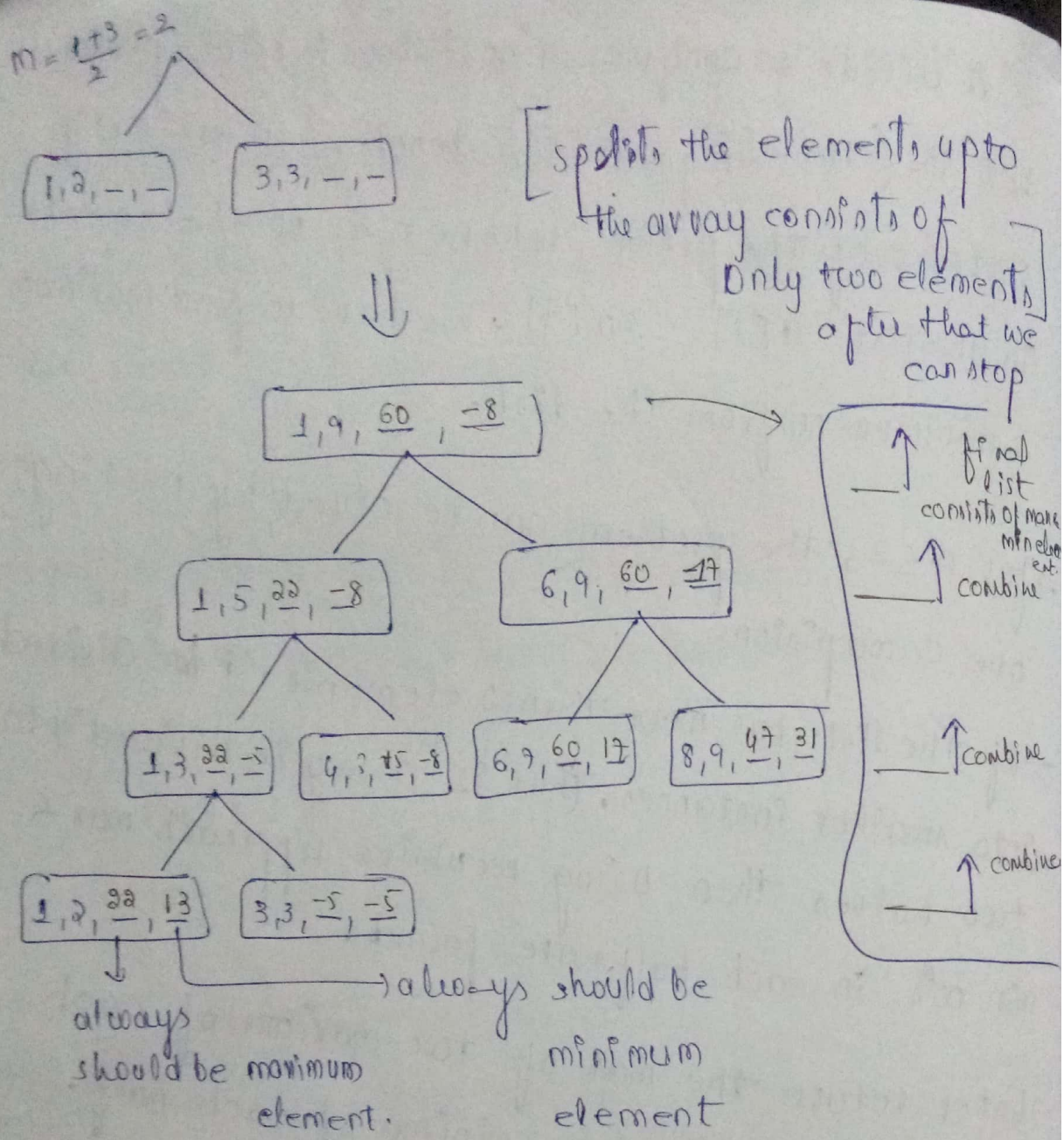
→ If the list has more than 2 elements, P has to be divided into smaller instances, that is the array is divided into two halves then using recursive approach max & min no's in each halfs are found.

→ later return the max of two maxima of each half & the minimum of two minima of each half.

→ E.g:- 22, 13, -5, -8, 15, 60, 17, 31, 47

(1) (2) (3) (4) (5) (6) (7) (8) (9) [indexes]





Algorithm:-

Algorithm MaxMin(i, j, max, min)

// a[1:n] is a global array parameters i & j are integers.

// $1 \le i \le j \le n$. The effect is to set max & min to the largest.

// and smallest values in a[i:j], respectively.

```
if (i == j) then
    max := min = a[i]; // small(p)
else if (i == j-1) then // Another case of small(p)
{
    if (a[i] < a[j]) then
    {
        max := a[j];
        min := a[i];
    }
    else
    {
        max := a[i];
        min := a[j];
    }
}
else
{
    // If P is not small, divide p into subproblems
    // find where to split the set
    mid := (i+j)/2;
    // solve the subproblems
    MaxMin (i, mid, max, min)
    MaxMin (mid+1, j, max1, min1);
    // combine the solutions
    if (max < max1) then
        max := max1;
    if (min > min1) then
        min := min1;
}
return
```

Analysis:-

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

$\{n = 2^k\}$

Solution:-

$$T(n) = 2T(n/2) + 2 \quad -(1)$$
$$T(n/2) = 2T(n/4) + 2 \quad -(2)$$
$$T(n/4) = 2T(n/8) + 2 \quad -(3)$$

} sub.

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

$(k = 4)$

$$= 2^{k-1} T\left(n/2^{k-1}\right) + 2^{k-1} + 2^{1-2} + \cdots + 2$$

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

$(n = 2^k)$

$$= n/2\, T(2) + \sum_{i=1}^{x-1} 2^i$$

Formulae:-

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\sum_{i=1}^{n} 2^i = 2^{n+1} - 2$$

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

$$= \frac{n}{2} + 2^k - 2$$

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

(which is better than other algorithms).

→ $\left(\frac{3n}{2} - 2\right)$ is best, average & worst case no of comparisions when 'n' is a power of 2.

⇒ Compare with the $(2n-2)$ comparisions of straight MaxMin ~~algo~~ is a saving of 25% in comparision. So, maxmin is optimal.

→ In terms of storage, maxMin is worse than straight MaxMin algorithm bcz it requires more stack space i.e., i, j, max, min, max1, min1, return address.

→ We have to find the count when element comparisions have the same cost as comparisions b/n i & j.

[If we merge/consider if elif else → one then the recurrence relation is

$$C(n) = \begin{cases} 2 & n \le 2 \\ 2\,C(n/2) + 3 & n \ge 2 \end{cases}$$

$C(n) = 2\,C(n/2) + 3$ — (1)

$C(n/2) = 2\,C(n/4) + 3$ — (2)

$C(n/4) = 2\,C(n/8) + 3$ — (3)

$C(n) = 2\left[2\,C(n/4) + 3\right] + 3$

$= 4\,C(n/4) + 6 + 3$

$= 4\left[2\,C(n/8) + 3\right] + 6 + 3$

$= 8\,C(n/8) + 12 + 6 + 3$

$k = 4$

$= 2^{k-1}\,C\left(n/2^{k-1}\right) + 3\left[4 + 2 + 1\right]$

$= 2^{k-1}\,C\left(n/2^{k-1}\right) + 3\left[2^{k-2} + 2^{k-3} + 2^{k-4} + \cdots\right]$

$= \frac{2^k}{2}\,C(2) + 3\left[\sum_{i=0}^{k-2} 2^i\right]$

$= n/2\,(2) + 3\left[\frac{2^k}{2} - 1\right] = n/2\,(2) + 3\left(\frac{n}{2} - 1\right)$

$= n + \frac{3n}{2} - 3 = \frac{5n}{2} - 3$

→ straight MaxMin algorithm consists of $3(n-1)$ comparisions (including for loop) in worst than $\left(\frac{5n}{2}-3\right)$ comparisions.

→ compared to storage again, This algorithm consists of more space than straight MaxMin algorithm.

→ By seeing There we can say both MaxMin & straight MaxMin are $\Theta[n]$.

→ By seeing the storage capacity, we can say that divide and conquer concept is used to divide the problem 'p' & combine soln's to sub-problms without defining small problms.

→ To solve the small problms. i.e, $n=1$ & $n=2$, we can use straightMaxMin algorithm.

UNIT-3

Greedy method

→ In greedy method the decision of solution is taken based on information available.

→ The greedy method is straight point method. This method is popular for obtaining feasible solution.

→ In greedy technique the solutions is constructed through a sequence of steps each expanding a partially constructed solution obtained so far until a complete solution to problem is reached.

→ At each step the choice made should be

(1) feasible :- It should satisfy the problem constraints

(2) Optimal :- Among all the feasible solutions the best choice is to be maid.

→ we need to find a feasible solution that either minimizes or maximizes a given objective function.

→ In greedy method following activities are performed.

(a) first we select some solution from input domain

(b) Then we check whether the solution is feasible or not.

(c) from the set of feasible solutions, particular sdution that satisfies or nearly satisfies the objective of function such a solution called Optimal solution.

(3) As greedy method works in stages, at each stage only one input is considered at each time. Based on this input it is decided whether the particular input gives the optimal solution or not.

Control Abstraction for Greedy method (in 3 page)

Applications of Greedy method

1. Job sequencing with deadlines
2. knapsack problem
3. Minimum spanning tree
4. Single source shortest path algorithm
5. Optimal merge patterns

Difference Between Divide and Conquer And Greedy Algorithm

Divide and Conquer	Greedy Algorithm
• Divide and Conquer is used to obtain a solution to given problem.	• It is used to obtain optimum solution.
• In this technique the problem is divided into small sub problems These subproblems are solved independently. finally all the solutions of subproblems are collected together to get the solution to given problem.	• In greedy method a set of feasible solution is generated and optimum solution is picked up.

• In this method duplications on subsolutions are neglected that means duplicate solutions may be obtained.	• In greedy method the optimum selection is without revising the previously generated solutions.
• divide and conquer is less efficient because of rework on solutions.	• Greedy method is comparatively efficient but there is no such guarantee of getting optimum solution.
• Examples for divide & conquer are - Quicksort, merge sort, binary search.	• Eg :- knapsack problem, finding minimum spanning tree.

Control Abstraction for Greedy method

```
Algorithm Greedy(a,n)
// a[1:n] contains the n inputs {solution := ∅
{
    solution := ∅; //Initialise the solution
    for i:= 1 to n do
    {
        x := select(a);
        if (feasible(solution, x)) then
            solution := Union(solution, x);
    }
    return solution;
}
```

Job sequencing with Deadlines

→ Consider that there are n jobs that are to be executed at any time t = 1 2 3 .... Only one job is to be executed.

→ The profits Pi are given. these profits are gained by corresponding jobs.

→ for obtaining feasible solutions we should take care that the jobs are completed within their deadline.

Eg :- Consider 4 jobs with profits and deadlines.

n	Pi	di
1	70	2
2	12	1
3	18	2
4	35	1

The following rules to obtain feasible solution are

1) Each job takes 1-unit of time.

2) If job starts before or at its deadline, profit is obtained. otherwise no profit.

3) Goal is to shedule jobs to maximize the total profit

4) Consider all possible schedules and compute minimum total time in system

step(1) :- Consider single job at a time

Sequence	profit
1	70
2	12
3	18
4	35

step(2) :- Consider 2 jobs at a time

Sequence	profit
1, 2	x
1, 3	88
1, 4	x
2, 1	82
2, 3	30
2, 4	x
3, 1	88
3, 2	x
3, 4	x
4, 1	105
4, 2	x
4, 3	53

Feasible solution :-

Sequence	profit
1	70
2	12
3	18
4	35
1, 3	88
2, 1	82
2, 3	30
3, 1	88
4, 1	105
4, 3	53

therefore optimal solution is (4, 1) with max profit 105.

Feasible sequence is a sequence that allows all jobs in a sequence to be executed within their deadlines and highest profit can be gained.

We have chosen job1 first then we have chosen 4 the solution (4,1) is feasible as the order of execution is 4 then 1 Also $d_4 < d_1$. If we try {1,3,4} then it is not a feasible solution hence reject 3 then set.

Examples :-

① Consider 4 jobs A,B,C,D with $(P_1 \cdots P_4) = (20,10,40,30)$ and $(d_1 \cdots d_4) = (4,1,2,1)$

② Let n=4 $(P_1,P_2,P_3,P_4) = (100,10,15,27)$ and $(d_1,d_2,d_3,d_4) = (2,1,2,1)$

③ Consider 5 jobs $(P_1 \cdots P_5) = (20,15,10,5,1)$ and deadline $(d_1 \cdots d_5) = (2,2,1,3,3)$

Algorithm for job sequencing

```
Algorithm JS(d,j,n)
//d[i]≥1, 1≤i≤n are the deadlines, n≥1. The jobs
//are ordered such that P[1]≥P[2]≥ .... ≥P[n]. J[i]
//is the ith job in the optimal solution, 1≤i≤k.
//Also, at termination d[J[i]]≤ d[J[i+1]], 1≤i≤k.
{
    d[0] := J[0] := 0; //Initialize
    J[1] := 1; //Include job 1
    k := 1;
    for i := 2 to n do
    {
        //Consider jobs in nonincreasing order of P[i].
        //Find position for i and check feasibility of
```

```
//insertion.
r := k;
while (d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do
        r := r - 1;
if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
{
    //insert i into J[].
    for q := k to (r+1) step -1 do J[q+1] := J[q];
    J[r+1] := i; k := k+1;
}
}
return k;
}
```

The sequence of J will be inserted if and only if $d[J[r]] \neq r$. This also means that the job 'J' will be processed if it is in within the deadline.

The computing time taken by above JobSequence Algorithm is $O(n^2)$, because the basic operation of computing sequencing in array 'J' is within two nested for loops.

Knapsack Problem

Suppose there are 'n' objects from $i = 1, 2, \ldots n$ each object (i) has some positive weight (w_i) and some profit value (P_i) is associated with each object which is denoted as P_i. These knapsack carry out at the most weight 'W'.

2) When we try to solve this problem using greedy approach our goal is to choose those objects which give maximum profit.

3) The total weight of selected objects should less than or equal to the maximum weight.

$$\text{Maximized} \sum_{i=1}^{n} P_i x_i \text{ subjected to } \sum_{i=1}^{n} w_i x_i \leq W$$

$$1 \leq i \leq n, \quad 0 \leq x_i \leq 1,$$

~~$0 \leq 1 \leq n$~~

Eg:- Consider there are 3 items with weight and profit values of each item is $(P_1 \ldots P_3) = (30, 21, 18)$ and $(w_1 \ldots w_3) = (18, 15, 10)$ and $W = 20$. Obtain the solution for the given knapsack problem.

Sol:-

Case ①:- Consider items that give maximum profit

$$\sum_{i=1}^{n} w_i x_i \leq W, \quad W = 20$$

$$x_i = (1, 2/15, 0)$$

$$\sum w_i x_i = 18 \times 1 + 15 \times \frac{2}{15} + 0$$

$$= 20$$

$$\sum P_i x_i = 30 \times 1 + 21 \times \frac{2}{15} + 18 \times 0$$

$$= 30 + 2.8$$

$$= 32.8$$

Case ②:- Consider items that contains minimum weight.

$$\sum_{i=1}^{n} w_i x_i \leq W, \quad W = 20$$

$x_i = (0, 10/15, 1)$

$\sum w_i x_i = 18 \times 0 + 15 \times \frac{10}{15} + 10$

$= 20$

$\sum P_i x_i = 30 \times 0 + 21 \times \frac{10}{15} + 18 \times 1$

$= 14 + 18$

$= 32$

case (3) :- Consider items with maximum P/w ratio

1st item — P/w = 30/18 = 1.6

2nd item — P/w = 21/15 = 1.4

3rd item — P/w = 18/10 = 1.8

$x_i = (10/18, 0, 1)$ $\sum_{i=1}^{n} w_i x_i \leq W, W = 20$

$\sum w_i x_i = 18 + \frac{10}{18} + 15 \times 0 + 10 \times 1$

$= 20$

$\sum P_i x_i = 30 \times \frac{10}{18} + 21 \times 0 + 18 \times 1$

$= 16.6 + 18$

$= 34.6$

Feasible solution :-

x_i	$\sum w_i x_i$	$\sum P_i x_i$
$(1, 2/15, 0)$	20	32.8
$(0, 10/15, 1)$	20	32
$(10/18, 0, 1)$	20	34.6

Therefore, Optimal solution is $x_i = (10/18, 0, 1)$ with profit 34.6.

Examples :-

① let $n=4$ $(P_1 \cdots P_4) = (5, 10, 9, 5)$ & $(w_1 \cdots w_4) = (1, 5, 5, 4)$ with $m = 8$.

② let $n=7$ and $m=15$ $(P_1 \cdots P_7) = (10, 5, 15, 9, 6, 18, 3)$ and $(w_1 \cdots w_7) = (2, 3, 5, 7, 1, 4, 1)$

Algorithm for knapsack :

```
Algorithm GreedyKnapsack(m,n)
// P[1:n] and w[1:n] contain the profits and weights
// respectively of n-objects ordered such that
// P[i]/w[i] ≥ P[i+1]/w[i+1].
// m is the knapsack size and x[1:n] is the
// solution vector.
{
    for i:=1 to n do
        x[i] := 0.0; //initialize x
    W := m;
    for i:=1 to n do
    {
        if(w[i]>W) then break;
        x[i] := 1.0;
        W := W-w[i];
    }
    if(i≤n) then
        x[i] := W/w[i];
}
```

Time complexity of knapsack algorithm is O(n).

Spanning Tree

→ let G=(V,E) be an undirected connected graph. A subgraph T=(V,E') of G is a spanning tree of 'G', if and only if T is a tree which has n-1 edges and no cycles.

→ A spanning tree is a subset of graph 'G' which has all the vertices covered with minimum possible number of edges.

→ By this definition we can say that every connected and undirected graph 'G' has atleast one spanning tree.

Minimum cost Spanning tree

→ A spanning tree with minimum or smallest weight is called minimum cost spanning tree. Algorithm to find min-cost spanning tree are

1) prim's Algorithm
2) kruskal's Algorithm

1) Prim's Algorithm:

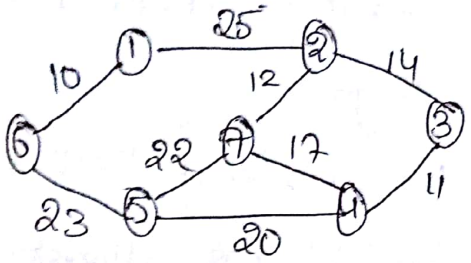
→ A greedy method to obtain a min-cost spanning tree for a weighted undirected graph.

→ This means it finds a subset of the edges

that form a tree that includes every vertex where the total weight of all the edges in the tree is minimised.

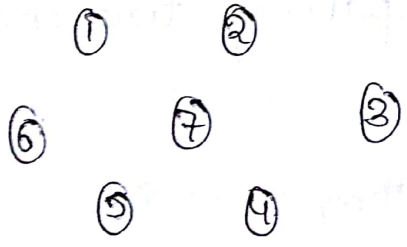
→ The algorithm operates by building this tree one vertex at a time from an ~~algorithm~~ arbitary starting vertex at each step adding the cheapest possible connection from tree to another possible vertex.

Eg:-



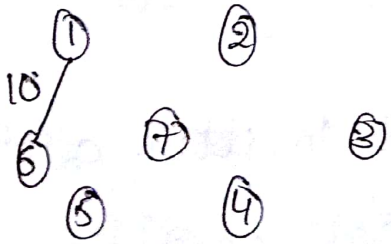
Now we consider all the vertices ~~first~~ then we will select an edge with minimum weight. The algorithm proceed by selecting the adjacent edges with minimum weight. ~~care~~ should be taken for not formatting circuit or cycle.

step ① :-



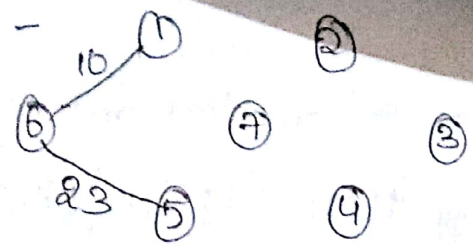
$U = \{\ \}$
$V = \{1,2,3,4,5,6,7\}$
$w = 0$

step ② :-



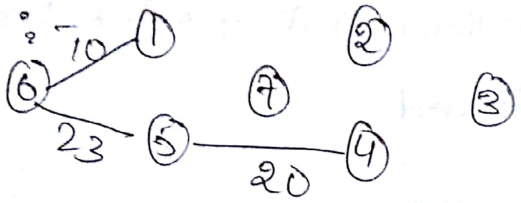
$U = \{1,6\}$
$V = \{2,3,4,5,7\}$
$w = 10$

step ③ :-



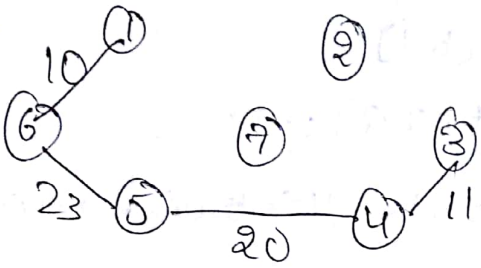
U = {1, 6, 5}
V = {2, 3, 4, 7}
ω = 10 + 23
= 33

step ④ :-



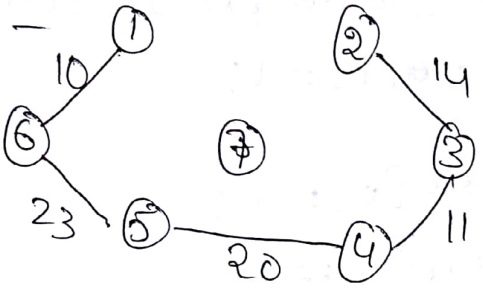
U = {1, 6, 5, 4}
V = {2, 3, 7}
ω = 33 + 20
= 53

Step ⑤ :-



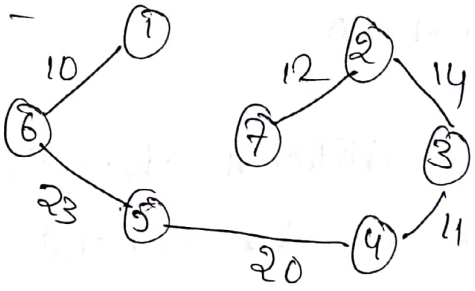
U = {1, 6, 5, 4, 3}
V = {2, 7}
ω = 53 + 11
= 64

step ⑥ :-



U = {1, 6, 5, 4, 3, 2}
V = {7}
ω = 64 + 14
= 78

Step ⑦ :-



U = {1, 6, 5, 4, 3, 2, 7}
V = { }
ω = 78 + 12
= 90

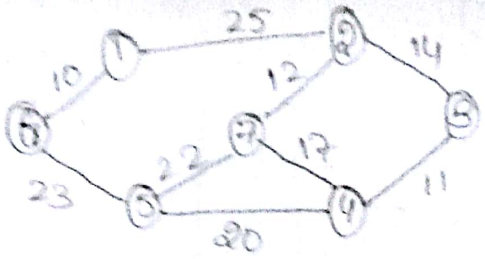
∴ minimum cost of spanning tree = 90

Prims Algorithm (or) Control Abstraction of Prims Algorithm:

```
Algorithm Prim (E, cost, n, t)
// E is the set of edges in G, cost [1:n, 1:n] is the cost
// adjacency matrix of an n vertex graph such that
// cost [i, j] is either a positive real number or ∞
```

```
//if no edge (l,j) exists.
//A minimum spanning tree is computed and stored as a
//set of edges in the array t[1:n-1,1:2]. (t[i,1],t[i,2])
// is an edge in the minimum cost spanning tree.
// The final cost is returned.
{
    Let (k,l) be an edge of minimum cost in E;
    mincost := cost[k,l];
    t[1,1] := k; t[1,2] := l;
    for i := 1 to n do //initialize near
        if (cost[i,l] < cost[i,k]) then
                near[i] := l;
        else
                near[i] := k;
    near[k] := near[l] := 0;
    for i := 2 to n-1 do
    {
      //find n-2 additional edges for t
       Let j be an index such that near[j] ≠ 0 and
       cost[j, near[j]] is minimum;
       t[i,1] := j;
       t[i,2] := near[j];
       near[j] := 0;
       for k := 1 to n do //update near[]
           if ((near[k] ≠ 0) and (cost[k, near[k]] >
                                        cost[k,j]))
               then near[k] := j;
    }
    return mincost;
}
```

Eg :-



Adjacency matrix :-

	1	2	3	4	5	6	7
1	0	25	∞	∞	∞	10	∞
2	25	0	14	∞	∞	∞	12
3	∞	14	0	11	∞	∞	∞
4	∞	∞	11	0	20	∞	17
5	∞	∞	∞	20	0	23	22
6	10	∞	∞	∞	23	0	∞
7	∞	12	∞	17	22	∞	0

(tree) t :-

	1	2
1		
2		
3		
4		
5		
6		

near :-

1	
2	
3	
4	
5	
6	
7	

step ① :-

let (k, l) be an edge of mincost

k = 1 , l = 6

mincost := cost (k, l) = 10

```
for i := 1 to n
{
    if (cost[i, 1] < cost[i, k]) then
        near[i] := l
    else
        near(i] := k
}
t[1, 1] = k    t[1, 2] = l
near(k] := near[l] := 0
```

cost(1,6) cost(1,1)

10 > 0

cost(2,6) cost(2,1)

∞ > 25

cost(3,6) cost(3,1)

∞ = ∞

cost(4,6) cost(4,1)

∞ = ∞

cost(5,6) cost(5,1)

23 < ∞

cost(6,6) cost(6,1)

0 < 10

near:-

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

t:-

	1	2
1	1	6
2		
3		
4		
5		
6		

t[1,1] = k = 1

t[1,2] = L = 6

near(k] := near[1] := 0

near[1] := near[6] := 0

So index ~~values~~ 1 and 6 values of

near:-

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

Step2: In near table remaining non zero values are considered as near[j] and index is 'j'.

→ for i := 2 to n-1

{

j - index array

near[j] ≠ 0

mincost = cost[j, near[j]]

from these value minimum cost is 23 so j=5 and near[j]=6

cost(2,1) = 25
cost(3,1) = ∞
cost(4,1) = ∞
cost(5,6) = 23
cost(7,1) = ∞

t[i,1] = t[2,1] = 5

t[i,2] = t[2,2] = 6

minimum cost = minimum cost + Cost[j, near[j])

= 10 + 23

= 33

updated t :-

	1	2
1	5	6
2		6
3		
4		
5		
6		

Step 3: for k := 1 to n

q

if (near[k] ≠ 0 and cost[k, near[k]] > cost[k, j])

then near[k] := j) {.

cost(2,1) cost(2,5)

25 < ∞

cost(3,1) cost(3,5)

2 = 2

cost(4,1) cost(4,5)

2 > 20

cost(7,1) cost(7,5)

2 > 22

updated near :-

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

Input

This steps 2 and 3 is repeated for q := 3 to 6.

Updated near

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

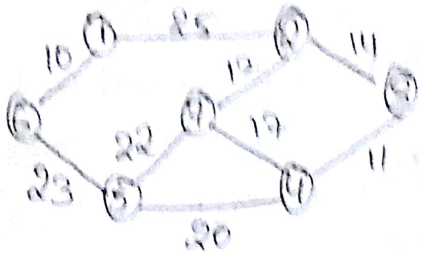
Time complexity of prim's Algorithm is $O(n^2)$

2) Kruskal's Algorithm :

→ Kruskal's Algorithm is another algorithm of obtaining minimum spanning tree. this algorithm is discovered by a second year graduate student Joseph kruska

→ In this algorithm, always the minimum cost edge has to be selected. But it is not necessary that selected optimum edge is adjacent.

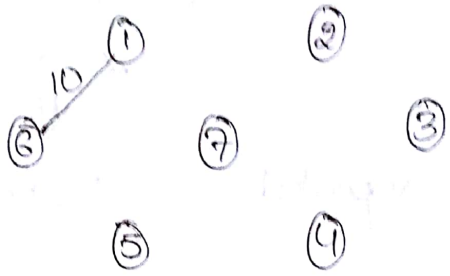
Eg :-



Step (1) :-

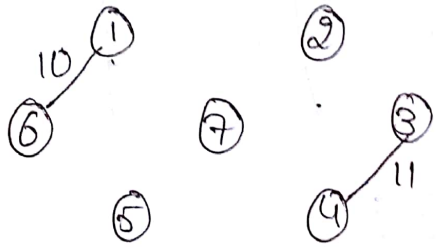
$\omega = 0$

Step (2) :-



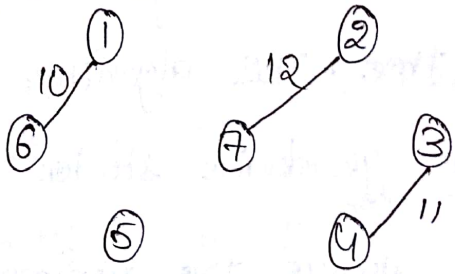
$\omega = 10$

step (3) :-



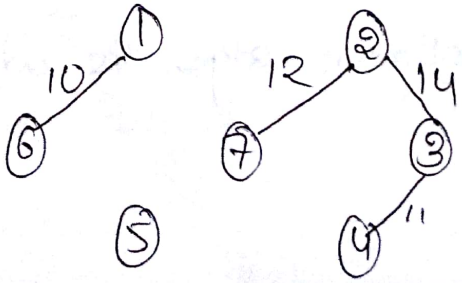
$\omega = 10 + 11$
$= 21$

Step (4) :-



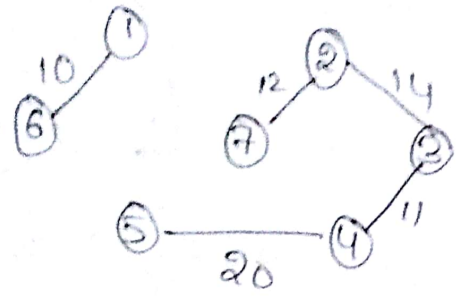
$\omega = 21 + 12$
$= 33$

step (5) :-



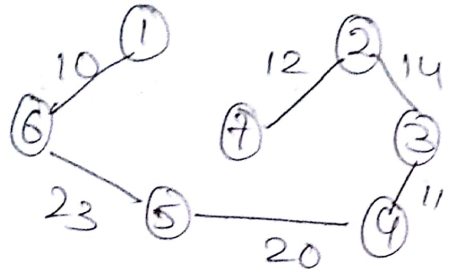
$\omega = 33 + 14$
$= 47$

step (6) :-



$\omega = 47 + 20$
$= 67$

step (7) :-



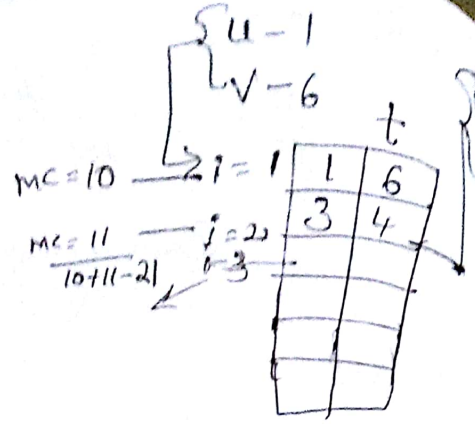
$\omega = 67 + 23$
$= 90$

∴ minimum cost spanning tree = 90

Control Abstraction of Kruskal's Algorithm :

```
Algorithm kruska (E, cost, n, t)
// E is the set of edges in G, G has n vertices.
// cost [U,V] is the cost of edge (U,V). t is the
// set of edges in the minimum-cost spanning
// tree. The final cost is returned
{
    Construct a heap out of the edge costs using
                                        Heapify;
    for i := 1 to n do { parent [i] := -1; }
    // Each vertex is in a different set.
    i := 0;
    mincost := 0.0;
    while ((i < n-1) and (heap not empty)) do
    {
        Delete a minimum cost edge (U,V) from
        the heap and reheapify using Adjust;
```

```
j := Find(u); k := Find(v);
if (j ≠ k) then
{
    i := i+1;
    t[i,1] := u;
    t[i,2] := v;
    mincost := mincost + cost[u,v];
    Union(j,k);
}
}
if (i ≠ n-1) then write("No spanning tree");
else return mincost;
}
```



Time complexity of Kruskal's Algorithm is $O(E \log E)$ or $O(E \log V)$. [for creating heap tree and taking each edge]

An optimal Randomized Algorithm :

→ Any algorithm for finding the minimum cost spanning tree of a given graph $G[V, E)$ will have to spend $\Omega(|V|+|E|)$ time in the worst case.

→ By using randomized algorithm we can reduce the number of vertices and edges on each step.

→ It reduces number of vertices by Boruvka steps

The following steps are :

1) Apply two Boruvka steps. At the end, the number of nodes will have decreased by a factor

at least 4. let the resultant graph be $\bar{G}(\bar{V}, \bar{E})$.

2) form a subgraph $G'(V', E')$ of $\bar{G}$, where each edge of $\bar{G}$ is chosen randomly to be in E' with probability $\frac{1}{2}$. The expected number of edges in E' is $\frac{|\bar{E}|}{2}$.

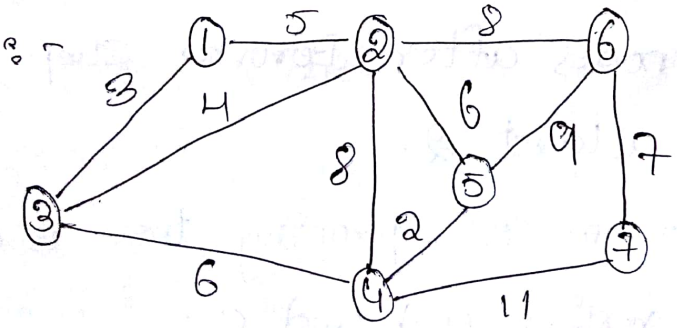
3) Recursively find a minimum-cost spanning forest F for G'.

4) Eliminate all the F-heavy edges from $\bar{G}$. with high probability, at least a constant fraction of the edges of $\bar{G}$ will be eliminated. let G'' be the resultant graph.

5) compute a minimum-cost spanning tree (call it T'') for G'' recursively. The tree T'' will also be a minimum-cost spanning tree for $\bar{G}$.

6) Return the edges of T'' together with the edges chosen in the Boruvka steps of step 1. These are the edges of a minimum-cost spanning tree for G.

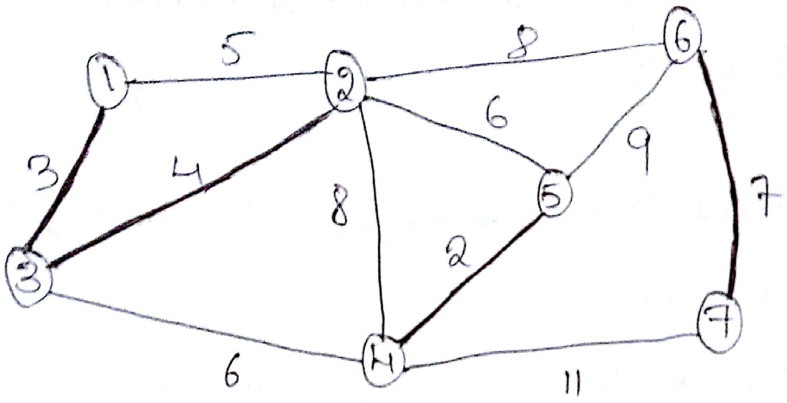
Eg :-



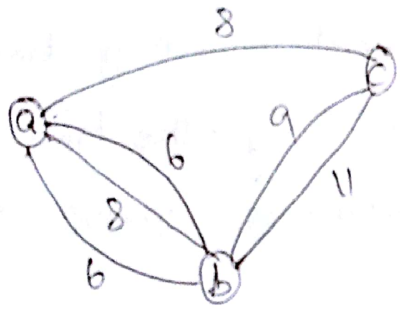
⇒ In a Boruvka step, for each node an incident edge with minimum weight is chosen.

) The edge (1,3) is chosen for node 1, the edge (2,1) is chosen for node 7 and so on.

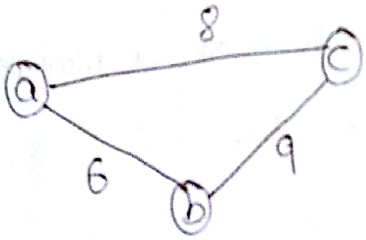
) All the chosen edges are shown with their lines.



→ The nodes 1, 2 and 3 form 1 component (a), nodes 4 and 5 form 2nd component (b), nodes 6 and 7 form another component (c)



→ Keep only edge of minimum weight between any 2 nodes.



→ The number of nodes after Boruvka step reduces by a factor of atleast 2.

→ Now the minimum cost spanning tree & calculate for graph of vertices a, b and c.

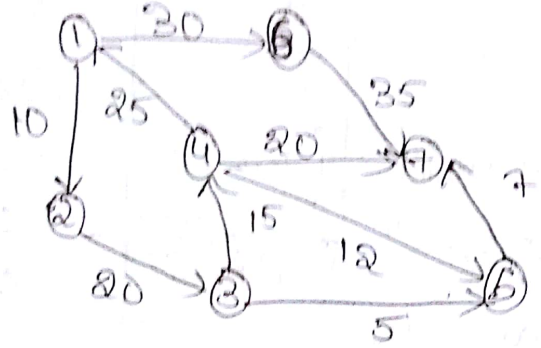
→ minimum cost spanning tree for the graph of (a)

will have the edges (1,3), (3,5), (4,5), (6,7), (3,4) and (2,6)

Single Source shortest path

Let G(V,E) be a graph then in single source shortest path from vertex V_0 to all remaining vertex is obtained. The vertex V_0 is called as source and the last vertex is called destination.

Eg:- Consider the following directed graph.



Sol:- here source vertex is 1 and destination vertex is 7

Initial vertex = 1

{1,2} = 10
{1,3} = 45
{1,4} = ∞
{1,5} = ∞
{1,6} = 30
{1,7} = ∞

Initialvertex = {1,2,3}

{1,2,3,4} = 45
{1,2,3,5} = 35
{1,2,3,6} = ∞
{1,2,3,7} = ∞

Initialvertex = {1,2}

{1,2,3} = 30
{1,2,4} = ∞
{1,2,5} = ∞
{1,2,6} = ∞
{1,2,7} = ∞

Initialvertex = {1,2,3,5}

{1,2,3,5,4} = ∞
{1,2,3,5,6} = ∞
{1,2,3,5,7} = 42

Initial vertex = {1,2,3,4}

{1,2,3,4,5} = 57

{1,2,3,4,6} = ∞

{1,2,3,4,7} = 65

Initial vertex = {1,2,3,4,5}

{1,2,3,4,5,6} = ∞

{1,2,3,4,5,7} = 64

Initial vertex = {1,6}

{1,6,2} = ∞

{1,6,3} = ∞

{1,6,4} = ∞

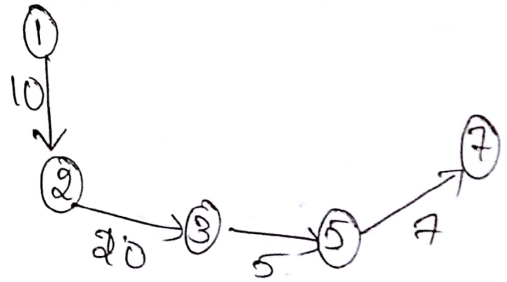
{1,6,5} = ∞

{1,6,7} = 65

Feasible solution :-

Path	distance
{1,2}	10
{1,2,3}	30
{1,2,3,5}	35
{1,2,3,5,7}	42
{1,2,3,4,5}	57
{1,2,3,4,5,7}	64
{1,6,3}	80
{1,6,7}	65

∴ from these feasible solution → optimal solution is {1,2,3,5,7} with minimum distance 42.



Control Abstraction of single source shortest path:

Algorithm ShortestPaths (v, cost, dist, n)

//dist[j], 1≤j≤n, is set to the length of the shortest

//path from vertex v to vertex j in a digraph G with n

//vertices. dist[v] is set to zero. G is represented

//by its cost adjacency matrix cost[1:n, 1:n].

{

for i:=1 to n do

{ //initialize S.

```
    S[i] := false;
    dist[i] := cost[v,i];
}
S[v] := true;
dist[v] := 0.0; //put v in S
for num := 2 to n do
{
    //Determine n-1 paths from v.
    choose u from among those vertices not
    in S such that dist[u] is minimum;
    S[u] := true; //put u in S.
    for (each w adjacent to u with S[w] = false) do
        //update distances.
        if(dist[w] > dist[u] + cost[u,w]) then
            dist[w] := dist[u] + cost[u,w];
}
```

Time complexity of single source shortest path is $O(n^2)$.

UNIT-4

Dynamic Programming

Dynamic programming

The idea of dynamic programming is very simple, if you have solved a problem with the given input, then save the result for future reference. So to avoid solving same problem again

If the problem is divided in to subproblems and these subproblems are in turn divided in to still smaller ones, and in this process, if you observe some overlapping subproblems, then it is a big hint for Dynamic programming

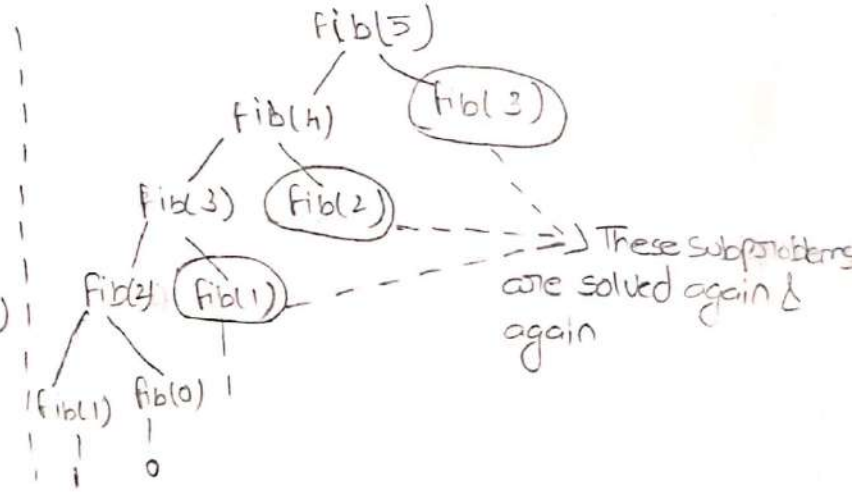
There are two ways of doing this

1) Top-down approach: Start solving problem by breaking it down. After solving each problem store its result in a table. If you see the problem has been solved already, then just return saved answer. This is referred as memorization

2) Bottom-up approach: Start solving trivial subproblems and solve them independently. It is guaranteed that these subproblems are solved before solving the problem This is referred as Dynamic programming. Result of each subproblem is again stored in a seperate table and used to solve bigger problem.

For example consider fibonnaci number calculation using recursion

```
int fib(int n)
{ if (n<=1)
    return n
  else
    return fib(n-1)+fib(n-2)
}
```



Time complexity is $O(2^n)$, because same subproblems are solved again & again

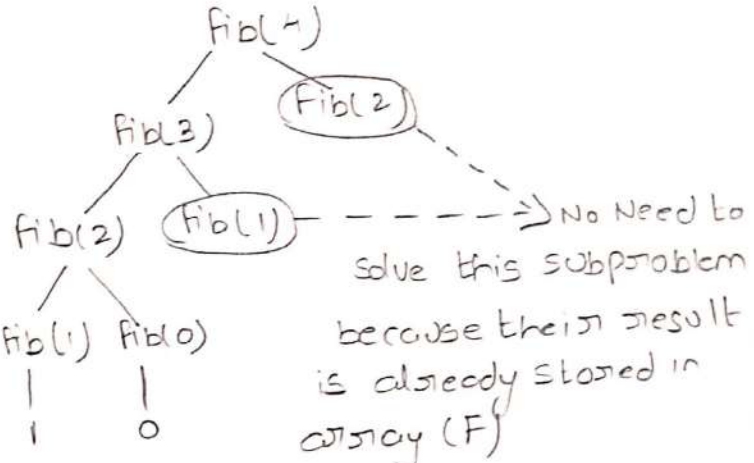
-> we can solve above problem by removing all recursive calls by using top down (or) bottom-up approach (Dyamic programming)

by Memorization (top-down approach)

let us consider array (F) to store result of each subproblem

```
int fib(int n){
   if(n<=1) then
      return n
   else if(F[n]!=-1)
      return F[n]
   else
      F[n]=fib(n-1)+fib(n-2)
   return F[n]
}
```

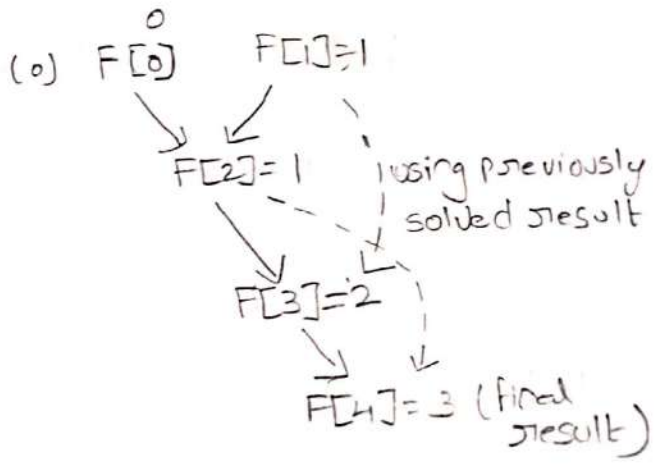
	0	1	2	3	4	
F	-1	-1	-1	-1	-1	(initially)
F	0	1	1	2	3	(after solving)



-> Time complexity is O(n)

by using Dyamic programming (Bottom-up approach)

```
int fib(int n)
{  F[0]=0
   F[1]=1
   for(i=2;i<=n;i++)
      F[i]=F[i-1]+F[i-2]
}
```



Time complexity is O(n)

By observing, above two algorithms we can state that, if we have Fi-1 and Fi-2, we can easily solve Fi

Key elements of Dynamic programming

Dynamic programming is typically applied to optimize problem. DP has two key elements

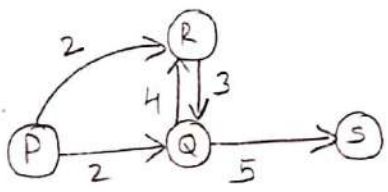
1) Optimal substructure
2) Overlapping subproblem

1) Optimal substructure : Here we build an optimization to problem from optimal solutions of subproblem. Dynamic programming uses optimal substructure in bottom-up fashion

2) Overlapping subproblems : DP solves distinct subproblems only once and result is stored in a table. Whenever this subproblem occurs again, it's result is taken from table (where as recursive algorithm solve same problem again & again)

Dynamic programming is mainly based on principle of optimality*

→ principle of optimality : In short we can say principle of optimality is satisfied when an optimal solution is found for a problem, if and only if there is an optimal solutions for it's subproblem as well. For example consider finding longest simple path in a graph below



Note : Path should not contain a cycle

Suppose we have to find longest simple path from P to S, then we have two possible path from P to S

P → Q → S i.e 2+5 = 7
P → R → Q → S i.e 2+3+5 = 10

we can see that P→R→Q→S, it is taken as optimal solution. According to principle of optimality, it should applicable to it's subpath also. If we observe carefully the subpath PR has length 2, but it is not longest possible path. The longest possible Path is P→Q→R is 2+4=6

Therefore principle of optimality is not satisfied for this Problem because as per definition of "principle of optimality", optimal solution must also be found for subproblem as well

Comparision between Dynamic programming and greedy method

Dynamic Programming	Greedy method
1) In this method we can solve solve some complex problem which can't be solved by greedy method	1) There are some problems which cannot be solved by greedy method. Therefore first try greedy, if it fails then try Dynamic programming
2) Principle of optimality in DP hold and optimal solution is guaranteed	2) Solution obtained is not guaranteed as optimal
3) DP considers all possible sequences in order to obtain optimal solution	3) In this method only one decision is made
4) DP solves the subproblems in bottom up fashion. The problem cannot be solved until we find solution to subproblems.	4) Greedy method solves subproblems from top down fashion with set of feasible solution and optimal solution

Difference between Divide and conquer & Dynamic programming

Dynamic Programming	Divide & conquer
1) In DP, Problem is divided into number of overlapping subproblems dependently	1) In divide & conquer, we divide problem in to non overlapping sub problem & solve them independently
2) In this method duplications in subsolution are neglected. Every subproblem is solved only once	2) Divide and conquer solves problem by solving same subproblem again and again
3) DP is efficient then divide & conquer	3) Divide & conquer is not that efficient because of duplication in problem solving
4) DP uses bottom up approach to solve problem	4) DP uses top-down approach to solve problem

General Method & application of Dynamic programming

General method of DP works the same way as divided conquer by combining solution to subproblems. Here partitioning of subproblem is done in overlapped manner (i.e dependent). The subproblem is solved only once and answer is saved in table

DP problem is divided in to number of stages, where optimal decision must to be made at each stage. The decision made at each stage influences the decisions taken in entire subsequent stages

Dynamic programming involves following steps

1) Given a problem, it is divided in to overlapping subproblem

2) Try to obtain optimal solution of each subproblem and save it in a table for future purpose. Each subproblem is solved only once

3) Solve all subproblems to make optimal solution

4) Construct optimal solution of a problem from computed information.

Applications of Dynamic Programming

1) Matrix chain multiplication
2) All pairs shortest path problem
3) Travelling salesman problem
4) Reliability design
5) 0/1 knapsack problem
6) Optimal binary search tree
7) Single source shortest path general weights
8) String Edition

2) All pair shortest path problem

All pairs shortest path problem find the shortest path b/w every pair of vertices i, j in G

let $G=(V,E)$ be a directed weighted graph with n vertices. let cost adjacency matrix for G is $cost(i,j)$, such that

$$cost(i,j) = \begin{cases} 0 & \text{if } i==j \\ \infty & \text{if } (i,j) \notin E(G), \text{otherwise} \\ \text{length of edge} & \text{if } (i,j) \in E(G) \end{cases}$$

→ The all pair shortest-path problem is to determine matrix A. Such that $A(i,j)$ is the length of a shortest path from i to j

→ The matrix (A) can be obtained (or) determined by solving 'n' single-source shortest path problem. since each single source shortest path takes $O(n^2)$ time, the matrix (A) is obtained in $O(n^3)$ time

→ We have an alternate solution by floyd warshall to compute all pairs shortest path problem is $O(n^3)$ time

→ Let us examine a shortest path between i to j. This path originates at vertex i and goes through some intermediate vertices (Possibly None) and terminates at vertex j (assume path does not containing cycles)

→ If k is an intermediate vertex on this shortest path, then subpath i to k and from k to j must be shortest path, so principle of optimality holds.

→ If k is the intermediate vertex with highest index, then i to k path is shortest i to k in G going through no vertex with index greater than k-1. similarly k to j no vertex is greater than k-1

→ Using $A^k(i,j)$ to represent length of a shortest path from i to j going through no vertex greater than k, than

$$A^k(i,j) = A^{k-1}(i,k) + A^{k-1}(k,j)$$

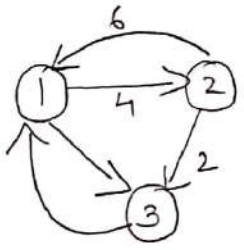
if shortest path from i to j going through vertices not greater than k and not goes trough k then

$$A^k(i,j) = A^{k-1}(i,j)$$

combining we get

$$A^k(i,j) = \min\{A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j)\} \quad k \geq 1$$

Example



Possible large k = 3

A^0 :

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

Now find

A^1	1	2	3
1			
2			
3			

→ i=1 j=1 k=1

$$A^k(i,j) = \min\{A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j)\}$$

$$A^1(1,1) = \min\{A^0(1,1), A^0(1,1) + A^0(1,1)\} = \min\{0, (0+0)\} = 0$$

i=1 j=2 k=1

$$A^1(1,2) = \min\{A^0(1,2), A^0(1,1) + A^0(1,2)\} = \{4, 0+4\} = 4$$

i=1 j=3 k=1

$$A^1(1,3) = \min\{A^0(1,3), A^0(1,1) + A^0(1,3)\} = \{11, 0+11\} = 11$$

i=2 j=1 k=1

$$A^1(2,1) = \min\{6, 6+4\} = 6$$

i=2 j=2 k=1

$$A^1(2,2) = 0$$

i=2 j=3 k=1

$$A^1(2,3) = \min\{2, 6+11\} = 2$$

i=3 j=1 k=1

$$A^1(3,1) = \min\{3, 3+0\} = 3$$

i=3 j=2 k=1

$$A^1(3,2) = \min\{\infty, 3+4\} = 7$$

i=3 j=3 k=1

$$A^1(3,3) = 0$$

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

ic = k=1

let us find A2 i.e K = 2

i=1 j=2 K=2 $\Rightarrow \min\{4, 4+0\} = 4$

i=1 j=3 K=2 $\Rightarrow \min\{11, 4+2\} = 6$

i=2 j=1 K=2 $\Rightarrow \min\{6, 0+6\} = 6$

i=2 j=3 K=2 $\Rightarrow \min\{2, 0+2\} = 2$

i=3 j=1 K=2 $\Rightarrow \min\{3, 7+6\} = 3$

i=3 j=2 K=2 $\Rightarrow \min\{7, 7+0\} = 7$

A2(1,1) = A2(2,2) = A3(3,3) = 0

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

→ Now find A3 i.e K = 3

A3(1,1) = A3(2,2) = A3(3,3) = 0

i=1 j=2 K=3 = $\min\{4, 6+7\} = 4$

i=1 j=3 K=3 = $\min\{6, 6+0\} = 6$

i=2 j=1 K=3 = $\min\{6, 2+3\} = 5$

i=2 j=3 K=3 = $\min\{2, 2+0\} = 2$

i=3 j=1 K=3 = $\min\{3, 0+3\} = 3$

i=3 j=2 K=3 = $\min\{7, 0+7\} = 7$

A3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

A3 matrix represent shortest path between all pairs in graph(G)

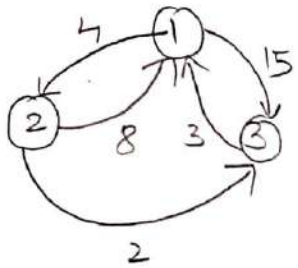
→ It's algorithm is given as

```
Algorithm Allpaths(cost, A, n)
// cost[1:n, 1:n] is the cost adjacency matrix of graph(G)
// A[i,j] is the shortest path from vertex i to vertex j
// cost[i,i] = 0.0, for 1≤i≤n
{
  for i:=1 to n do
    for j:=1 to n do
      A[i,j] = cost[i,j]  // copy cost in to A
  For k:=1 to n do
    for i:=1 to n do
      for j:=1 to n do
        A[i,j] = min(A[i,j], A[i,k]+A[k,j]);
}
```

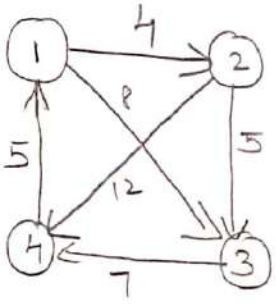
Time complexity of above algorithm is $O(n^3)$

Solve the all pairs shortest path problem for following graphs

(1) =



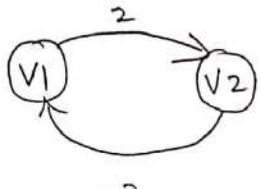
(2) =



Limitation of Floyd warshall algorithm

→ If we allow (G) to contain a cycle of negative length, then the shortest path between any two vertices on this cycle is $-\infty$

for example



D^0	1	2
1	0	2
2	-3	0

$\Rightarrow$

D^1	1	2
1	0	2
2	-3	-1

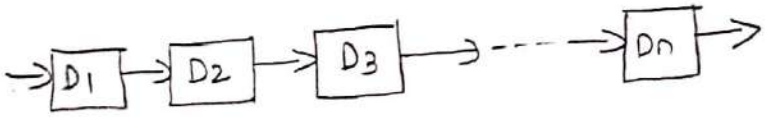
$\Rightarrow$

D^2	1	2
1	-1	1
2	-4	-4

The last matrix does not give correct value. In fact we get $-\infty$. This shows diagraph with negative weights with floydIs warshall algorithm does not yield correct result.

4) Reliability design

The problem is to design a system that is composed of several devices connected in series as shown below



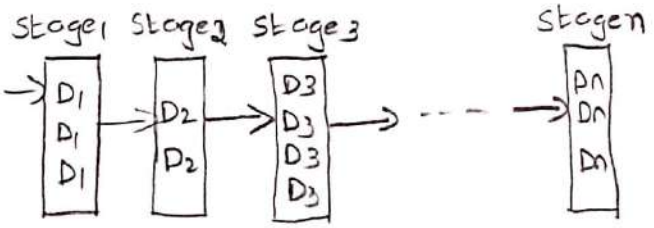
let r_i be the reliability of device (D_i), where $1 \le i \le n$ (i.e r_i's is the probability that device D_i will function properly. The reliability of system is Πr_i

Even if the individual devices are very reliable (then r_i is is very close to one), the reliability of entire system may not be very good for eg

if $n=10$ and $r_i = 0.99$ $1 \le i \le 10$ [Π is used to represent product of bunch of terms]

then $\Pi r_i = (0.99)^{10} = 0.90438$

Here $0.90438 < 0.99$ (indivial device reliability). To improve entire system performance iit is desirable to duplicate devices. Multiple copies of same device type are connected in parallel as shown below as a single stage through the use of switching circuts



m_1 (number of copies of D_1) = 3

m_2 (number of copies of D_2) = 2

m_3 (number of copies of D_3) = 4

-> if stage 'i' contain m_i copies of device D_i, then the probability that all m_i have mulfunction $(1-r_i)^{m_i}$. Hence reliability of stagei becomes $1-(1-r_i)^{m_i}$

Thus
If $r_i = 0.99$ and $m_i = 2$ then reliability becomes

$$1-(1-0.99)^2 = 0.999$$

previously it is 0.90 when $m_i = 1$

In any practical situation, the stage reliability is little less than $1-(1-r_i)^{m_i}$ because the switching circuits themselves are not reliable. Also failures of copies of the same device may not fully independent

Our goal is to use device duplication to maximize reliability. If a stage contains m_i copies of device D_i then reliability of stage i is given by fortion $\phi_i(m_i)$. The reliability of entire system of all stages $\prod_{1 \le i \le n} \phi_i(m_i)$, Here n is number of stages

let c_i be the cost of each unit of device i and let c be the maximum allowable cost of the system being designed, then we have to solve reliability design problem such that

$$\text{maximize } \prod_{1 \le i \le n} \phi_i(m_i)$$

$$\text{subject to } \sum_{1 \le i \le n} c_i m_i \le c$$

$$m_i \ge 1 \text{ where, } 1 \le i \le n$$

Definitely $c_i > 0$, each m_i must be in range $1 \le m_i \le u_i$, Here u_i is maximum allowable numbers devices in a stage i

$$u_i = \left\lfloor \left(c + c_i - \sum_{1}^{n} c_j\right) / c_i \right\rfloor \quad (u_i \text{ means upper bound})$$

-> Reliability design problem is solved using an approach similar to 0/1 knapsack problem

let us assume S^i represent design of stage i, we can obtain each s^i from s^{i-1}. The s^i_j is represented as

$$S^i_j = \max \left\{ \prod_{1 \le i \le n} \phi_i(m_i), \sum_{1 \le i \le n} c_i m_i \right\}$$

Here i = stage number

j = number of duplicate devices

Example: Design a three stage system with device types D_1, D_2, D_3. The costs are \$30, \$15, \$20 respectively. The cost of system is to be no more than \$105. The reliability of each device type is 0.9, 0.8, 0.5 respectively

Sol: Given data

$c_1 = 30 \quad c_2 = \$15, \quad c_3 = \$20 \quad c = \$105$

$r_1 = 0.9 \quad r_2 = 0.8 \quad r_3 = 0.5 \quad n = 3$ (no of stages

At first calculate maximum allowable number of duplicate devices in a stage (i.e u_i)

$$u_1 = \lfloor (105 + 30 - (30 + 15 + 20))/30 \rfloor = \lfloor (135 - 65)/30 \rfloor = \lfloor 2.333 \rfloor = 2$$

$$u_2 = \lfloor (105 + 15 - (30 + 15 + 20))/15 \rfloor = \lfloor (120 - 65)/15 \rfloor = \lfloor 3.666 \rfloor = 3$$

$$u_3 = \lfloor (105 + 20 - (30 + 15 + 20))/20 \rfloor = \lfloor (125 - 65)/20 \rfloor = \lfloor 3 \rfloor = 3$$

$u_1 = 2, u_2 = 3, u_3 = 3$

Beginning with $S^0 = \{(1, 0)\}$, calculate S^1 i.e stage 1

Here $i = 1 \quad u_1 = 2$ each m_i must be in range $1 \le m_i \le u_i$

so $1 \le m_i \le 2$ i.e $m_1 = 1$ (or) 2, it means $j = 1$ (or) 2

i.e S^1 is combination of S^1_1 and S^1_2

* Calculate S^1_1 i=1 j=1 i.e $m_1=1$

$\phi_1(m_1) = 1-(1-r_1)^{m_1} = 1-(1-0.9) = 1-0.1 = 0.9$

cost if we have single copy of D_1 in stage S_1 is 30 (C_1)

$S^1_1 = \{(0.9, 30)\}$

* Calculate S^1_2 i=1 j=2 i.e $m_1=2$

$\phi_1(m_1) = 1-(1-r_1)^{m_1} = (1-0.9)^2 = 1-(0.01) = 0.99$

cost if we have two copies of D_1 in stage S_1 is $2C_1$ i.e 60

$S^1_2 = \{(0.99, 60)\}$

since S^1 can be obtained by merging sets S^1_1 & S^1_2

$S^1 = \{(0.9, 30), (0.99, 60)\}$

→ we can't calculate S^3_1, since upper bound is 2 for stage 1 i.e $U_1=2$

→ Similarly S^2 is calculated from S^2_1, S^2_2, S^2_3, They can be determined using S^1

* calculate S_1^2

i=2 j=1 i.e $m_2=1$

$\phi_2(m_2) = 1-(1-r_2)^{m_2} = 1-(1-0.8)^1 = 1-0.2 = 0.8$

$S^2_1 = \{(0.9 \times 0.8, 30+15), (0.99 \times 0.8, 60+15)\} = \{(0.72, 45), (0.792, 75)\}$

* calculate S^2_2 i=2 j=2 i.e $m_2=2$

$\phi_2(m_2) = 1-(1-r_2)^{m_2} = 1-(1-0.8)^2 = 1-(0.2)^2 = 0.96$

since j=2, $^{1,2}C_1 = 2 \times 15 = 30$ is added to S^1 pairs

$S_2^2 = \{(0.9 \times 0.96, 30+30), (0.99 \times 0.96, 60+30)\}$

$= \{(0.864, 60), (0.9504, 90)\}$

* calculate S_3^2 $i=2$ $j=3$ i.e $m_2=3$

$\Phi_2(m_2) = 1-(1-r_2)^{m_2} = 1-(1-0.8)^3 = 1-(0.2)^3 = 0.992$

Since $j=3$, $3C_2 = 3 \times 15 = 45$ add to S^1 pairs

$S_3^2 = \{(0.9 \times 0.992, 30+45), (0.99 \times 0.992, 60+45)\}$

$= \{(0.8928, 75), (0.98208, 105)\}$

Totally

$S^2 = \{(0.72, 45), (0.792, 75), (0.864, 60), (0.9504, 90), (0.8928, 75), (0.98208, 105)\}$

The tuples (0.9504, 90) & (0.98208, 105) are removed from S^2 because 1st one least cost $105-90=15$ & 2nd tuple leaves cost $105-105=0$, since both costs are not enough for stage 3 they are eliminated

→Now $S^2 = \{(0.72, 45), (0.792, 75), (0.864, 60), (0.8928, 75)\}$

Here cost must be in increasing order

$45 < 75 < 60 < 75$ here $75 < 60$ condition fails i.e tuple (0.792, 75) dominates (0.864, 60) so eliminate (0.792, 75)

→Now $S^2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

similarly S^3 is calculated from S_1^3, S_2^3, S_3^3

* Calculate S_1^3 i.e $i=3$ $j=1$ i.e $m_3=1$

$\phi_3(m_3) = 1-(1-r_3)^{m_3} = 1-(1-0.5)^1 = 0.5$ $c_3=20$ is added to S^2

$S_1^3 = \{(0.72\times0.5, 45+20), (0.864\times0.5, 60+20), (0.8928\times0.5, 75+20)\}$

$S_1^3 = \{(0.36, 65), (0.432, 80), (0.4464, 95)\}$

* Calculate S_2^3 $i=3$ $j=2$ i.e $m_3=2$

$\phi_3(m_3) = 1-(1-r_3)^{m_3} = 1-(1-0.5)^2 = 1-0.25 = 0.75$

since $j=2$, $2\times c_3 = 40$

$S_2^3 = \{(0.72\times0.75, 45+40), (0.864\times0.75, 60+40), (0.8928\times0.75, 75+40)\}$

$= \{(0.54, 85), (0.648, 100)\}$

* Calculate S_3^3 i.e $i=3$ $j=3$ i.e $m_3=3$

since $j=3$, $3\times c_3 = 60$

$\phi_3(m_3) = 1-(1-r_3)^{m_3} = 1-(1-0.5)^3 = 1-0.125 = 0.875$

$S_3^3 = \{(0.63, 105)\}$

combining, we get $S^3 = \{(0.36, 65), (0.432, 80), (0.54, 85), (0.648, 100), (0.63, 105)\}$

eliminating $(0.63, 105)$

$S^3 = \{(0.36, 65), (0.432, 80), (0.54, 85), (0.648, 100)\}$

The best design has a reliability is 0.648 and a cost 100. Tracing back S^i's we determine $m_1=1$, $m_2=2$, and $m_3=2$

i.e $\overbrace{(0.648, 100)}^{S_2^3}$ is obtained from $(0.864, 60)$ for $m_3=2$

$\overbrace{(0.864, 60)}^{S_2^2}$ is obtained from $(0.9, 30)$ for $m_2=2$

$\overbrace{(0.9, 30)}^{S_1^1}$ is obtained when $m_1=1$

$m_1=1$ $m_2=2$ $m_3=2$

-> 5) 0/1 knapsack problem

The terminology and notations used in this section is the same as previous greedy method. let us assume a theif went to a shop with empty knapsack (bag). He has to pick ornaments and should place in his bag where each ornament has some profit(P) and weight(w) and his knapsack capacity is (M). Ornaments are placed in bag such that it maximizes the profit(P) with total weight less than (or) equal to knapsack capacity.

->let x denotes the status of object placed in knapsack. Here x has only two values

$x_i = 0$ i.e ith object is not placed at all in knapsack

$x_i = 1$ i.e ith object is completely placed in knapsack

Here we do not consider fraction of objects like we seen in greedy method. profit has accrued only when $x_i = 1$ then remaining capacity of knapsack is $\boxed{m = m - w_i}$ and P_i is profit accured, otherwise

the remaining capacity is m and profit(P_i) = 0 i.e no profit has accured. let f be the function representing knapsack problem

$$f_i(m) = \max\{f_{i-1}(m), f_{i-1}(m - w_i) + P_i\}$$

let us consider ordered set (S) to represent above function. S^1 means placing object, and S^2 and S^3 means placing object 2 & object 3 in one bag. The value of S^i can be obtained such that

$$S^i = S^{i-1} + S^i_1$$

where $S^i_1 = S^{i-1} + \{P_i, w_i\}$

->consider a problem to keep 3 objects in a bag with capacity (6) and weight $(w_1, w_2, w_3) = (2, 3, 4)$ and their respective profits $(P_1, P_2, P_3) = (1, 2, 5)$

solution: let initial value => $S^0_0 = \{(P_i, w_i)\} = \{0, 0\}$

$S_1^i = S^{i-1} + \{P_i, w_i\}$ → addition

$S^i = S^{i-1} + S_1^i$ → Merging

$i = 1, 2, 3$

* For $i = 1$

$S_1^1 = S^0 + \{(P_1, w_1)\}$

$= S^0 + \{(1,2)\}$

$= \{(0,0)\} + \{(1,2)\}$

$= \{(1,2)\}$

$S^1 = S^0 + S_1^1$

$= \{(0,0)\} + \{(1,2)\}$

$= \{(0,0), (1,2)\}$

* For $i = 2$

$S_1^2 = S^1 + \{(P_2, w_2)\}$

$= S^1 + \{(2,3)\}$

$= \{(0,0), (1,2)\} + \{(2,3)\}$

$= \{(2,3), (3,5)\}$

$S^2 = S^{2-1} + S_1^2$

$= S^1 + S_1^2$

$= \{(0,0), (1,2)\} + \{(2,3), (3,5)\}$

$= \{(0,0), (1,2), (2,3), (3,5)\}$

* For i = 3

$S_1^3 = S^2 + \{P_3, W_3\}$

$= S^2 + \{(5,4)\}$

$= \{(0,0), (1,2), (2,3), (3,5)\} + \{(5,4)\}$

$= \{(5,4), (6,6), (7,7), (8,9)\}$

$S^3 = S^{3-1} + S^3$

$= S^2 + \{(5,4), (6,6), (7,7), (8,9)\}$

$= \{(0,0), (1,2), (2,3), (3,5)\} + \{(5,4), (6,6), (7,7), (8,9)\}$

$= \{(0,0), (1,2), (2,3), (3,5), (5,4), (6,6)\}$

Here pairs (7,7) & (8,9) are eliminated because their weight is greater than bag capacity(M) i.e 6. Here consider pair (6,6) as (P_K, W_K) to get optimal solution using the notation

Step 1:- if $(P_K, W_K) \in S^n$ and $(P_K, W_K) \notin S^{n-1}$ then

$x_i = 1$

$(P_K, W_K) = (P_K, W_K) - (P_j, W_j)$ repeat step 1

else

$x_i = 0$

$(P_K, W_K) = (P_K, W_K) - (P_j, W_j)$ repeat step 1

For example, consider initial $(P_K, W_K) = (6,6)$

* $(6,6) \in S^3$ and $(6,6) \notin S^2$

$\therefore x_3 = 1$

$(P_K, W_K) = (6,6) - (5,4) = (1,2)$

* Now $(P_K, W_K) = (1,2)$

$(1,2) \in S^2$ and $(1,2) \in S^1$

So, $x_2 = 0$

$(P_K, W_K) = (1,2) - (0,0) = (1,2)$

* Now $(P_K, W_K) = (1,2)$

$(1,2) \in S^1$ and $(1,2) \notin S^0$

So $x_1 = 1$

∴ Totally $(x_1, x_2, x_3) = (1,0,1)$

To check the maximum profit

$$\sum_{i=1}^{n} P_i x_i \qquad (i = 1,2,3)$$

$= P_1x_1 + P_2x_2 + P_3x_3$

$= 1(1) + 2(0) + 5(1)$

$= 1 + 0 + 5 = 6$

Maximum profit is 6

Algorithm of 0/1 Knapsack problem

Algorithm DKP(P, W, n, M)

$S^0 \leftarrow \{(0,0)\}$

for $i = 1$ to n do

$S_1^i = S^{i-1} + (P_i, W_i)$ // addition

$S^i = S^{i-1} + S_1^i$ // Merging

//eliminate all the pairs whose weight $(W) > M$ (knapsack capacity)

$(P_K, W_K) = (P, W)$ where satisfying maximum profit & weight

for $i = 1$ to n do

```
    if (pk,wk) ∈ s^n and (pk,wk) ∈ s^(n-1) then
            xi=1
        (pk,wk) = (pk,wk) - (pj,wj)
    else
        xi=0
        (pk,wk) = (pk,wk) - (pj,wj)
  end for
  // Total profit
  for i=1 to n
     P= pi x xi
  return (p,x,w)
end DKP
```

Time complexity of above algorithm is $O(mn)$ where n is the number of items and m is the capacity of knapsack

⇒ STRING EDITING :-

- Consider two strings $X = x_1 x_2, \ldots x_n$ &

$Y = y_1 y_2 \ldots y_m$ where x_i is $1 \leq i \leq n$

and y_j is $1 \leq j \leq m$, are the members

of finite set of symbols known as alphabet.

- Transform x into y using a sequence of editing operations on x. The edit operations are insert, delete & change. And there is a cost associated with performing each.
- The cost of a sequence of operation is the sum of the costs of individuals operations in the sequence.
- The problem of string editing is identifying the minimum cost sequence of edit operations that will transform x into y.
- A dynamic programming solution for this problem can be obtained as follows:-

(i) Let $D(x_i)$ be the cost of deleting symbol x_i from x.

(ii) Let $I(y_j)$ be the cost of inserting the symbol y_j into x.

(iii) $c(x_i, y_j)$ be the cost of changing the symbol x_i of x and y_j of y.

- If $i=0$, $j=0$ then cost of $c(i,j)=0$.
- If $i>0$, $j=0$ then deletion i.e., $c(i,0)=$ $c(i-1,0)+D(x_i)$
- If $i=0$, $j>0$ then we can transform x into y by a sequence of insertion i.e.,

$$c(0,j)=c(0,j-1)+I(y_j)$$

- If $i\neq 0$, $j\neq 0$ then we can transform x into y in one of the 3 ways :
  (i) Transform x into y using a minimum cost edit sequence and then delete x_i i.e.,

$$cost(i-1,j)+D(x_j)$$

  (ii) Transform x into y ~~by~~ using a minimum cost edit sequence and then change symbol x_i to y_j i.e.,

$$c(i-1,j-1)+c(x_i,y_j)$$

Note :-

if $(x_i == y_j)$ then

$$c(x_i,y_j)=0$$

else

$c(x_i, y_j)$ = cost of change operations.

(iii) Transform x into y using a minimum cost edit sequence and then insert y_j

i.e., $c(i, j-1) + I(y_j)$

- According to the principle of optimality, the minimum cost of any edit sequence that transform x to y is minimum of above three costs.

$$cost[i,j] = \begin{cases} 0 & i=0, j=0 \\ cost[i-1,0] + D(x_i) & i>0, j=0 \\ cost[0,j-1] + I(y_j) & i=0, j>0 \\ cost'[i,j] & i>0, j>0. \end{cases}$$

Here,

$$cost'[i,j] = \min \begin{cases} cost[i-1,j] + D(x_i), \; cost[i-1,j-1] \\ + c(x_i, y_j), \; cost[i,j-1] + I(y_j) \end{cases}$$

// if $x[i] == y[j]$ then $c(x_i, y_j) = 0$

else $c(x_i, y_j)$ = cost of change operation

- We have to compute cost[i,j] for all possible values of $0 \le i \le j$: There are

(n+1)(m+1) values and these values are computed in the form of a table.

- The table 'T' corresponds to a row i and column j.
  - T(i,j) stores the value of cost(i,j)
  - Zeroth row and zeroth column are computed first by using insertion and deletion operation.
- The remaining values are computed using insertion, deletion and change.
- The value cost of n,m is the final answer and to compute minimum edit sequence we use backward trace for the cost(n,m)

∴ The algorithm is given by :-

```
Algorithm StringEditing (n, m, i, j, I(yj), D(xi))
{
    for i := 0 to n+1 // n size of string x
    {
        for j := 0 to m+1 // m size of string y
        {
```

```
if (i==0 and j==0) then
        c[i][j] := 0;
else if (i==0 and j>0) then
        c[i][j] = c[i, j-1] + I(yj)
                // I(yj) is cost of
                   insertion.
else if (i>0 and j==0) then
        c[i][j] = c[i-1, j] + D(xi) // D(xi)
                  is cost of deletion
else
        c[i,j] = min{ c[i-1, j] + D(xi),
                 c[i-1, j-1] + c(xi, yj),
                 c[i, j-1] + I(yj) }
        // if x[i] = y[j] . then
                c(xi, yj) = 0
           else
                c(xi, yj) = cost of
                   change operation
    }
}
}
```

∴ Time complexity = $O(n^2)$

⇒ STRING EDITING :-

$$X = x_1 x_2 x_3 \ldots\ldots x_n$$
$$Y = y_1 y_2 y_3 \ldots\ldots y_n$$
} alphabets.

$$\text{cost}(i,j) = \begin{cases} 0 & i=0, j=0 \\ \text{cost}(i-1,0) + D(x_i) & i>0, j=0 \\ \text{cost}(0,j-1) + I(y_j) & i=0, j>0 \\ \text{cost}'(i,j) & i>0, j>0. \end{cases}$$

Operations :

Insertion, Deletion, change

change	Deletion
insertion →	↓

$$\text{Cost}'(i,j) = \min \{ \text{cost}(i-1,j) + D(x_i), \text{cost}(i-1,j-1) + c(x_i, y_i), \text{cost}(i,j-1) + I(y_j) \}$$

if $x_i == y_j$ then $c(x_i, y_j) = 0$
else $c(x_i, y_j) = c$.

Problem :

(1) Consider a string editing problem $x = \{a, a, b, a, b\}$ and $y = \{b, a, b, b\}$ each insertion & deletion has a unit cost and a change cost = 2 units.

Sol:- Given,

X = aabab $\Rightarrow x_1 = a, x_2 = a, x_3 = b, x_4 = a, x_5 = b$

Y = babb $\Rightarrow y_1 = b, y_2 = a, y_3 = b, y_4 = b$

$I = 1, D = 1, C = 2$.

i \ j →		b	a	b	b
	0	1	2	3	4
0	0	1	2	3	4
a 1	1	2	1	2	3
a 2	2	3	2	3	4
b 3	3	2	3	2	3
a 4	4	3	2	3	4
b 5	5	4	3	2	3

$Cost(0,0) = 0$

$Cost(0,1) = Cost(0,j-1) + I(y_j)$

$i=0, j>0$ $= Cost(0,0) + I(y_1)$

$= 0 + 1$

$= 1$

$Cost(0,2) = Cost(0,1) + I(y_2)$

$= 1 + 1$

$= 2$

$Cost(0,3) = Cost(0,2) + I(y_3)$

$= 2 + 1$

$= 3$

$Cost(0,4) = Cost(0,3) + I(y_4)$

$= 3 + 1$

$= 4$

$Cost(1,0) = Cost(i-1,0) + D(x_i)$

$i>1, j=0$ $= Cost(0,0) + D(x_1)$

$= 0 + 1$

$= 1$

$cost(2,0) = cost(1,0) + D(x_2)$

$= 1 + 1$

$= 2.$

$cost(3,0) = cost(2,0) + D(x_3)$

$= 2 + 1$

$= 3.$

$cost(4,0) = cost(3,0) + D(x_4)$

$= 3 + 1$

$= 4.$

$cost(5,0) = cost(4,0) + D(x_5)$

$= 4 + 1$

$= 5.$

$i > 0, j > 0.$

$$cost(1,1) = \min \begin{cases} cost(i-1, j) + D(x_i), \\ cost(i-1, j-1) + c(x_i, y_j) \\ cost(i, j-1) + I(y_j) \end{cases}$$

$i=1, j=1$

if $x_i == y_j$ then $c(x_i, y_j) = 0$

else $c(x_i, y_j) = c$

$$= \min \begin{cases} cost(0,1) + D(x_1), \\ cost(0,0) + c(x_1, y_1) \\ cost(1,0) + I(y_1) \end{cases}$$

$= \min \{ 1+1, 0+2, 1+1 \}$

$= 2.$

$cost(1,2) = \min \{ cost(0,2) + D(x_1), cost(0,1) + c(x_1, y_2)$

$+ cost(1,1) + I(y_2) \}$

$= \min \{ 2+1, 1+0, 2+1 \} = 1$

$$cost(1,3) = \min\{ cost(0,3) + D(x_1),\ cost(0,2) + c(x_1, y_3),\ cost(1,2) + I(y_3) \}$$

$$= \min\{ 3+1,\ 2+2,\ 1+1 \} = 2.$$

$$cost(1,4) = \min\{ cost(0,4) + D(x_1),\ cost(0,3) + c(x_1, y_4),\ cost(1,3) + I(y_4) \}$$

$$= \min\{ 4+1,\ 3+2,\ 2+1 \} = 3$$

$$cost(2,1) = \min\{ cost(1,1) + D(x_2),\ cost(1,0) + c(x_2, y_1),\ cost(2,0) + I(y_1) \}$$

$$= \min\{ 2+1,\ 1+2,\ 2+1 \}$$

$$= 3.$$

$$cost(2,2) = \min\{ cost(1,2) + D(x_2),\ cost(1,1) + c(x_2, y_2),\ cost(2,1) + I(y_2) \}$$

$$= \min\{ 1+1,\ 2+0,\ 3+1 \} = 2.$$

$$cost(2,3) = \min\{ cost(1,3) + D(x_2),\ cost(1,2) + c(x_2, y_3),\ cost(2,2) + I(y_3) \}$$

$$= \min\{ 2+1,\ 1+2,\ 2+1 \}$$

$$cost(2,4) = \min\{ cost(1,4) + D(x_2),\ cost(1,3) + c(x_2, y_4),\ cost(2,3) + I(y_4) \}$$

$$= \min\{ 3+1,\ 2+2,\ 3+1 \}$$

$$= 4.$$

$$cost(3,1) = \min\{ cost(2,1) + D(x_3) + cost(2,0) + c(x_3, y_1),\ cost(3,0) + I(y_1) \}$$

$$= \min\{ 3+1,\ 2+0,\ 3+1 \} = 2.$$

$$cost(3,2) = \min\{ cost(2,2) + D(x_3),\ cost(2,1) + c(x_3, y_2),\ cost(3,1) + I(y_2) \}$$

$$= \min\{ 2+1,\ 3+2,\ 2+1 \} = 3.$$

$$cost(3,3) = \min\{ cost(2,3) + D(x_3),\ cost(2,2) + c(x_3, y_3),\ cost(3,2) + I(y_3) \}$$

$$= \min\{ 3+1,\ 2+0,\ 3+1 \} = 2.$$

$$cost(3,4) = \min\{ cost(2,4) + D(x_3),\ cost(2,3) + c(x_3, y_4),\ cost(3,3) + I(y_4) \}$$

$$= \min\{ 4+1,\ 3+0,\ 2+1 \} = 3$$

$$cost(4,1) = \min\{ cost(3,1) + D(x_4),\ cost(3,0) + c(x_4, y_1),\ cost(4,0) + I(y_1) \}$$

$$= \min\{ 2+1,\ 3+2,\ 4+1 \} = 3$$

$$cost(4,2) = \min\{ cost(3,2) + D(x_4),\ cost(3,1) + c(x_4, y_2),\ cost(4,1) + I(y_2)\}$$

$$= \min\{3+1,\ 2+0,\ 3+1\}$$

$$= 2.$$

$$cost(4,3) = \min\{ cost(3,3) + D(x_4),\ cost(3,2) + c(x_4, y_3),\ cost(4,2) + I(y_3)\}$$

$$= \min\{2+1,\ 3+2,\ 2+1\} = 3$$

$$cost(4,4) = \min\{ cost(3,4) + D(x_4),\ cost(3,3) + c(x_4, y_4),\ cost(4,3) + I(y_4)\}$$

$$= \min\{3+1,\ 2+2,\ 3+1\}$$

$$= 4.$$

$$cost(5,1) = \min\{ cost(4,1) + D(x_5),\ cost(4,0) + c(x_5, y_1),\ cost(5,0) + I(y_1)\}$$

$$= \min\{3+1,\ 4+0,\ 5+1\} = 4.$$

$$cost(5,2) = \min\{ cost(4,2) + D(x_5),\ cost(4,1) + c(x_5, y_2),\ cost(5,1) + I(y_2)\}$$

$$= \min\{2+1,\ 3+2,\ 4+1\} = 3$$

$$cost(5,3) = \min\{ cost(4,3) + D(x_5),\ cost(4,2) + C(x_5, y_3),\ cost(5,2) + I(y_3) \}$$

$$= \min\{ 3+1,\ 2+0,\ 3+1 \} = 2.$$

$$cost(5,4) = \min\{ cost(4,4) + D(x_5),\ cost(4,3) + C(x_5, y_4),\ cost(5,3) + I(y_4) \}$$

$$= \min\{ 4+1,\ 3+0,\ 2+1 \} = 3.$$

Solution:

Therefore, there are maximum three operations we have to perform.

In this problem, there are 2 possibilities of minimum cost edit sequence.

(1.) Delete x_1, x_2 Insert y_3

(2) Replace x_1, y_1 and Delete x_4.

⇒ SINGLE SOURCE SHORTEST PATH IN DYNAMIC PROGRAMMING USING BELLMAN FORD ALGORITHM:

Note :

* Single source shortest path using greedy method. — Dijkstra's
* Single source shortest path using dynamic programming — Bellman Ford alg
* Find the shortest path using Floyd's algorithm is called All Pairs shortest path using dynamic programming.

- Dijkstra's algorithm doesn't work if the given graph contains -ve edge weights. i.e., this algorithm may (or) may not produce the shortest path correctly.
- Bellman ford algorithm will produce correct result. By using this algorithm, we try out all the possible solutions then we take best solution as optimal solution.
- In this algorithm, we apply relaxation property to all the edges upto (N-1) times.

Relaxation property:

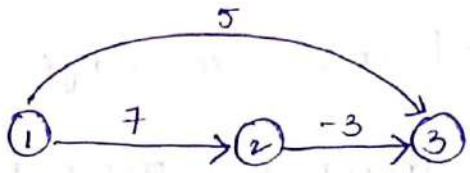
Let $dist^k[u]$ be the length of the shortest path from source vertex 'v' to destination vertex 'u' under the constraint that the shortest path contains atmost 'k' edges.

Here, v - source vertex

u - destination vertex

k - Edges wher k = 1 to (N-1)

- A directed graph with -ve edge is



If $k = 1$ then relaxation property is.

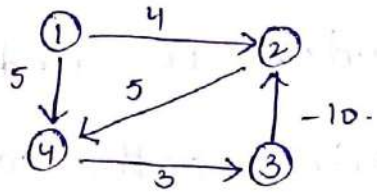
$$dist^1[u] = cost[v, u]$$

If $k > 1$ then

$$dist^k[u] = \min\{ dist^{k-1}[u], \min\{ dist^{k-1}[i] + cost[i,u] \} \}$$

- If the graph contains a -ve weight cycle i.e., the total weight of the cycle is -ve, then bellman ford algorithm doesn't work.

For example,



[∵ we need to show that it doesn't work. and show relaxation property upto 'N' times].

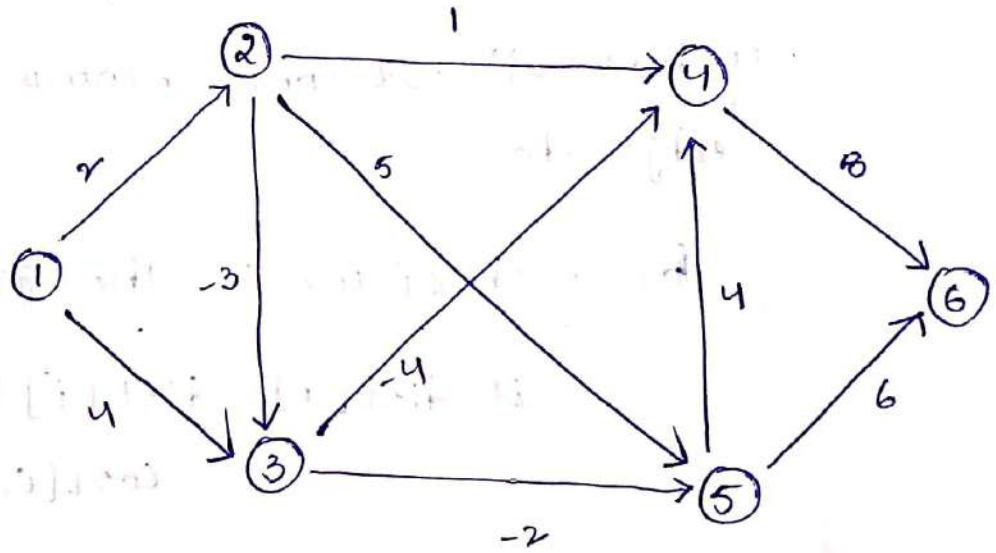
Bellman Ford Algorithm:

```
Algorithm BellmanFord (v, cost, dist, n)
// single-source / all destinations shortest
// paths with negative edge costs
{
    for i := 1 to n do // Initialize dist
        dist[i] := cost[v, i];
    for k := 2 to n-1 do
        for each u such that u ≠ v and
            u has atleast one incoming
            edge do
            for each <i, u> in the graph do
                if dist[u] > dist[i] +
                    cost[i,u] then
                dist[u] := dist[i] +
                    cost[i,u];
}
```

- If adjacency matrices are used then and $O(E)$ time, if adjacency lists are used then the time complexity is $O(n^2)$.

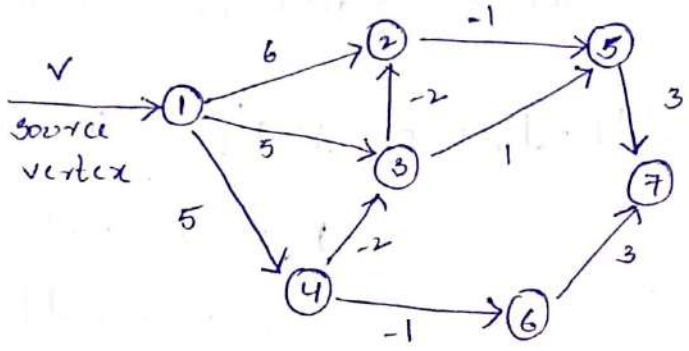
  Here, E is the number of edges in the graph.

- The overall complexity is $O(n^3)$ when adjacency matrices are used and $O(ne)$ adjacency lists are used.



Problems :

(1)



N = 7

N - 1 = 6 thus we have to apply relaxation property

K	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	∞ 7
4	0	1	3	5	0	4.	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

K = 1 ⇒ dist'[u] = cost[v, u]

dist'[1] = 0

dist'[2] = 6

dist'[3] = 5

dist'[4] = 5

dist'[5] = ∞

dist'[6] = ∞

dist'[7] = ∞

k = 2

$dist^k[u] = \min\{ dist^{k-1}[u], \min\{dist^{k-1}[i] + cost[i,u]\}\}$

$dist^2[2] = \min\{ dist^1[2], \min\{ dist^1[3] + cost[3,2]\}$

$dist^1[4] + cost[4,2]$

$dist^1[5] + cost[5,2]$

$dist^1[6] + cost[6,2]$

$dist^1[7] + cost[7,2]$

$= \min\{ 6, \min\{5-2, 5+\infty, \infty+\infty, \infty+\infty, \infty+\infty\}\}$.

$= \min\{6, 3\} = \underline{3}$.

$dist^2[3] = \min\{dist^1[3], \min\{ dist^1[2] + cost[2,3]\}\}$

$dist^1[4] + cost[4,3]$

$dist^1[5] + cost[5,3]$

$dist^1[6] + cost[6,3]$

$dist^1[7] + cost[7,3]$.

$= \min\{5, \min\{6+\infty, 5+(-2), \infty+\infty, \infty+\infty, \infty+\infty\}\}$

$= \min\{5, 3\}$

$= \underline{3}$

$dist^2[4] = \min\{ dist^1[4], \min\{ dist^1[2] + cost[2,4]\}\}$

$dist^1[3] + cost[3,4]$

$dist^1[5] + cost[5,4]$

dist'[6] + cost[6,4]

dist'[7] + cost[7,4]

$= \min\{5, \min\{6+\infty, 5+\infty, \infty+\infty+\infty+\infty, \infty+\infty\}\}$

$= \min\{5, \infty\}$

$= 5$

$dist^{\nu}[5] = \min\{dist'[5], \min\{dist'[2] + cost[2,5]\}\}$

dist'[3] + cost[3,5]

dist'[4] + cost[4,5]

dist'[6] + cost[6,5]

dist'[7] + cost[7,5]

$= \min\{\infty, \min\{6-1, 5+1, 5+\infty, \infty+\infty, \infty+\infty\}\}$

$= \min\{\infty, 5\} = 5$

$dist^{\nu}[6] = \min\{dist'[6], \min\{dist'[2] + cost[2,6]\}\}$

dist'[3] + cost[3,6]

dist'[4] + cost[4,6]

dist'[5] + cost[5,6]

dist'[7] + cost[7,6]

$= \min\{\infty, \min\{6+\infty, 5+\infty, 5+(-1), \infty, \infty\}\}$

$= \min\{\infty, 4\} = 4.$

$dist^2[7] = \min\{dist^1[7], \min\{dist^1[2] + cost[2,7]\}\}$

$dist^1[3] + cost[3,7]$

$dist^1[4] + cost[4,7]$

$dist^1[5] + cost[5,7]$

$dist^1[6] + cost[6,7]$

$= \min\{\infty, \min\{6+\infty, 5+\infty + 5+\infty, \infty+3, \infty+3\}\}$

$= \min\{\infty, \infty\} = \infty$

$k = 3$

$dist^3[2] = \min\{dist^2[2], \min\{dist^2[3] + cost[3,2]\}\}$

$dist^2[4] + cost[4,2]$

$dist^2[5] + cost[5,2]$

$dist^2[6] + cost[6,2]$

$dist^2[7] + cost[7,2]$

$= \min\{3, \min\{3+(-2), 5+\infty, 5+\infty, 4+\infty, \infty+\infty\}\}$

$= \min\{3, \min\{1, \infty, \infty, \infty, \infty\}\}$

$= \min\{3, 1\}$

$= 1$

$dist^3[3] = \min\{dist^2[3], \min\{dist^2[2] + cost[2,3]\}$

$dist^2[4] + cost[4,3]$

$dist^2[5] + cost[5,3]$

$dist^r[6] + cost[6,3]$

$dist^r[7] + cost[7,3]$

$= \min\{4, \min\{3+\infty, 5+(-2)+5+\infty, 5+\infty, \infty+\infty\}\}$

$= \min\{4, \min\{\infty, 3, \infty, \infty, \infty\}\}$

$= \underline{3}$

$dist^3[4] = \min\{dist^r[4]; \min\{dist^r[2] + cost[2,4]\}\}$

$dist^r[3] + cost[3,4]$

$dist^r[5] + cost[5,4]$

$dist^r[6] + cost[6,4]$

$dist^r[7] + cost[7,4]$

$= \min\{5, \min\{3+\infty, 3+\infty, 5+\infty, 4+\infty, \infty+\infty\}\}$

$= 5$

$dist^3[5] = \min\{dist^r[5], \min\{dist^r[2] + cost[2,5]\}\}$

$dist^r[3] + cost[3,5]$

$dist^r[4] + cost[4,5]$

$dist^r[6] + cost[6,5]$

$dist^r[7] + cost[7,5]$

$= \min\{5, \min\{3-1, 3+1, 5+\infty, 4+\infty, \infty+\infty\}\}$

$= \min\{5, 2\} = 2$

$dist^3[6] = \min\{4, \min\{3+\infty, 3+\infty, 5-1, 5+\infty, \infty+\infty\}\}$

$= \min\{4, 4\}$

$= 4$

$dist^3[7] = \min\{ dist^2[7], \min\{ dist^2[2] + cost[2,7] \}$

$dist^2[3] + cost[3,7]$

$dist^2[4] + cost[4,7]$

$dist^2[5] + cost[5,7]$

$dist^2[6] + cost[6,7]$

$= \min\{ \infty, \min\{ 3+\infty, 3+\infty, 5+\infty, 5+\infty, 4+\infty \} \}$

$= \min\{ \infty, \infty \} = \infty.$

K = 4

$dist^4[2] = \min\{ dist^3[2], \min\{ dist^3[3] + cost[3,2] \}$

4

5

6

7

$= \min\{ 1, \min\{ 3-2, 5+\infty, 2+\infty, 4+\infty, \infty+\infty \} \}$

$= \min\{1, 1\} = 1$

$dist^4[3] = \min\{ dist^3[3], \min\{ dist^3[2] + cost[2,3] \} \}$

4

5

6

7

$= \min\{ 3, \min\{ 1+\infty, 5-2, 2+\infty, 4+\infty, \infty, \infty \} \}$

$= \min\{3, 3\}$

$dist^4[4] = \min\{dist^3[4], \min\{dist^3[2] + cost[2,4]\}\}$

3
5
6
7

$= \min\{5, \min\{1+\infty, 3+\infty, 2+\infty, 4+\infty, \infty+\infty\}\}$

$= \min\{5, \infty\} = 5$

$dist^4[5] = \min\{dist^3[5], \min\{dist^3[2] + cost[2,5]\}\}$

3
4
6
7

$= \min\{2, \min\{1-1, 3+1, 5+\infty, 4+\infty, \infty+\infty\}\}$

$= \min\{2, 0\} = 0$

$dist^4[6] = \min\{dist^3[6], \min\{dist^3[2] + cost[2,6]\}\}$

3
4
5
7

$= \min\{4, \min\{1+\infty, 3+\infty, 5-1, 2+\infty, \infty+\infty\}\}$

$= \min\{4, 4\} = 4$

$dist^4[7] = \min\{dist^3[7], \min\{dist^3[2] + cost[2,7]\}\}$

3
4
5
6

$= \min\{\infty, \min\{1+\infty, 3+\infty, 5+\infty, 2+3, 4+3\}\}$

$= \min\{\infty, 5\} = 5$

$k = 5$

$dist^5[2] = \min\{dist^4[2], \min\{dist^4[3] + cost[3,2]\}$

4
5
6
7

$= \min\{1, \min\{3-2, 5+\infty, 0+\infty, 4+\infty, 5+\infty\}\}$

$= \min\{1, \min\{1, \infty, \infty, \infty, \infty\}\}$

$= 1$

$dist^5[3] = \min\{dist^4[3], \min\{dist^4[2] + cost[2,3]\}$

4
5
6
7

$= \min\{3, \min\{1+\infty, 5-2, 0+\infty, 4+\infty, 5+\infty\}\}$

$= \min\{3, 3\} = 3$

$dist^5[4] = \min\{dist^4[4], \min\{dist^4[2] + cost[2,4]\}$

3
5
6
7

$= \min\{5, \min\{1+\infty, 3+\infty, 0+\infty, 4+\infty, 5+\infty\}\}$

$= \min\{5, \infty\} = 5$

$dist^5[5] = \min\{dist^4[5], \min\{dist^4[2] + cost[2,5]\}$

3
4
6
7

$= \infty$

$= \min\{0, \min\{1-1, 3+1, 5+\infty, 4+\infty, 5+\infty\}\}$

$= 0$

$dist^5[6] = \min\{dist^4[6], \min\{dist^4[2] + cost[2,6]\}\}$

3
4
5
7

$= \min\{4, \min\{1+\infty, 3+\infty, 7-1, 0+\infty, 5+\infty\}\}$

$= 4$

$dist^5[7] = \min\{dist^4[7], \min\{dist^4[2] + cost[2,7]\}\}$

3
4
5
6

$= \min\{\infty, \min\{1+\infty, 3+\infty, 5+\infty, 0+3, 4+3\}\}$

$= \min\{\infty, 3\} = 3$

dist^5

K = 6

$dist^6[2] = \min\{dist^5[2], \min\{dist^5[3] + cost[3,2]\}\}$

4
5
6
7

$= \min\{1, \min\{3-2, 5+\infty, 0+\infty, 4+\infty, 3+\infty\}\}$

$= 1$

$dist^6[3] = \min\{dist^5[3], \min\{dist^5[2] + cost[2,3]\}\}$

4
5
6
7

$= \min\{3, \min\{1+\infty, 5-2, 0+\infty, 4+\infty, 3+\infty\}\}$

$= 3$

$dist^6[4] = \min\{ dist^5[4], \min\{ dist^5[2] + cost[2,4] \} \}$

3
5
6
7

$= \min\{ 5, \min\{ \infty, \infty, \infty, \infty, \infty \} \}$

$= 5$

$dist^6[5] = \min\{ dist^5[5], \min\{ dist^5[2] + cost[2,5] \} \}$

3
4
6
7

$= \min\{ 0, \min\{ 1-1, 3+1, \infty, \infty, \infty \} \}$

$= 0$

$dist^6[6] = \min\{ dist^5[6], \min\{ dist^5[2] + cost[2,6] \} \}$

3
4
5
7

$= \min\{ 4, \min\{ \infty, \infty, 5-1, \infty, \infty \} \}$

$= 4.$

$dist^6[7] = \min\{ dist^5[7], \min\{ dist^5[2] + cost[2,7] \} \}$

3
4
5
6

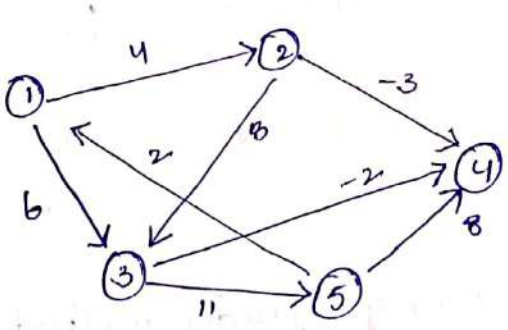
$= \min\{ 3, \min\{ \infty, \infty, \infty, 0+3, 4+3 \} \}$

$= 3$

Therefore, the final solution is

Source Vertex	Destination Vertex	Path	Cost
1	2	1 → 4 → 3 → 2	1
1	3	1 → 4 → 3	3
1	4	1 → 4	5
1	5	1 → 4 → 3 → 2 → 5	0
1	6	1 → 4 → 6	4
1	7	1 → 4 → 3 → 2 → 5 → 7	3

(2.)
=



k	1	2	3	4	5
1	0	4	6	∞	∞
2	0	4	6	1	17
3	0	4	6	1	17
4	0	4	6	1	17

Branch and Bound

Introduction: Branch and Bound

The Branch and Bound represent a set of solutions in a tree and performs a search called state space search. Here tree contains number of branches called candidate solutions

The Branch and Bound algorithm explores branches of this tree, that represent subset of solutions, each candidate solution is checked against upper and lower estimated bounds to find optimal solution, and is discarded if it cannot produces a better solution

Branch and bound is a systematic method for solving optimization problem. B&B is applied where greedy method & Dynamic programming fail. However it is much slower, it often leads to exponential time in worst case

Bounding functions are used to avoid the generation of subtrees the do not contain answer node

Solution states: A set of tuples or states that represent a possible solutions in solution space is called solution states

Answer States: A set of tuples or states that represent a possible solution in solution space that satisfies the implicit and explicit constraints

Tree organization of the solution space is referred as state space tree

Live node: A node which has been generated and all of whose children have not yet been generated is called a live node

E-node: The live node whose children are currently being generated is called the E-Node (node being expanded)

Dead node: A node which is not expanded further is called Dead node

Bounding functions: They are constraints used to kill live node without generating all their children

In Branch and bound all children of "a particular E-node are generated before any other live node become the E-node" We can have two graph Strategies, BFS and D-search to implement above statement i.e "A new node cannot begin until the node currently being explored is fully explored"

Both of these generalize to branch and bound strategies:

* BFS -like state space search will be called FIFO (First in First out) search as the list of live nodes in First-in First out (using queues)

* D-search (D-search) like state space search will be called LIFO (Last in First out) where list of live nodes are visited in Last in First out using stacks

->let us see how FIFO branch-and-bound algorithm would search the state space tree for the four queens problem given below

→ The state space tree for 4-Queens problem is given as

State space tree for 4 Queen's problem

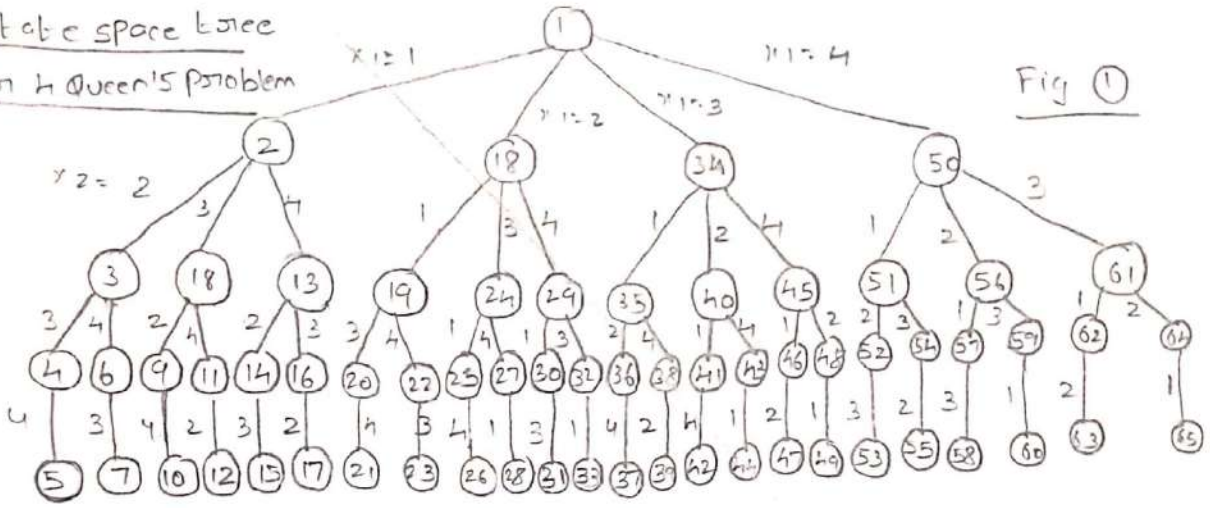
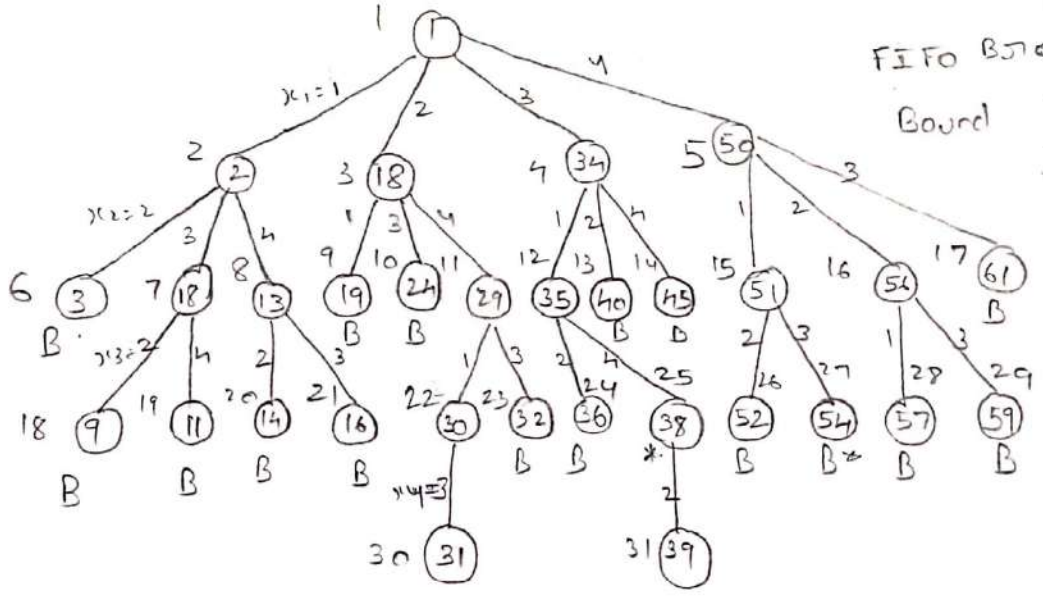


Fig (1)

→ In N-Queens problem we use a bounding functions: "No 2-Queens are placed on same row, same column, same diagonal

→ Portion of 4-queens state space tree generated by FIFO branch and bound



FIFO Branch and Bound Fig (2)

Initially, there is only one live node, node 1. This represents the case in which no queen has been placed on the chess board. Now Node 1 becomes E-node. It is expanded and it's children, nodes 2, 18, 34, 50 are generated and placed in a queue. The only live nodes are 2, 18, 34, 50 then the next E-node is 2. It is expanded and

nodes 3, 8, 13 are generated

Here node is 3 which is immediately killed using the bounding function (3) and no further expanded. Nodes 8 and 13 are added to the queue of live nodes

18	34	50	8	13

Queue of live nodes

Node 18 becomes next E node. Node 19, 24, 29 are generated. Node 19 and 24 are killed as a result of bounding functions. Node 29 is added to the queue

34	50	8	13	29

-> Now E node is 34. This process continues by making enqueues and dequeues. At the time the answer node, node 31 reached the only live nodes remaining are nodes 38 and 54. Node 38 expanded Node 39 becomes answer nodes. whereas Node 54 is killed due to bounding function

-> As by seeing above example. The two Basic Stages of a general Branch and Bound method is

* Branching: Splitting the problems in to sub problems
* Bounding: calculating lower and/or upper bounds for the objective function

Comparison between Backtracking and Branch & Bound

Backtracking Method	Branch & Bound Method
1) In this technique, the solution is obtained using depth first search	1) In this technique any of search methods among DFS & BFS (or) least cost search is used
2) There is a posibility of obtaining the bad solutions	2) No bad solutions are obtained
3) Backtracking is used in Problems like graph coloring, Sum of subsets, N-queens etc	3) B&B is applied to the problems like Travelling Sales person, job sequencing, 0/1 knapsack etc
4) A state space tree is not searched completely instead the searching terminates as soon as the solution is obtained	4) Here state space tree is searched completely - optimum solution can be obtained at any point of time
5) Backtracking approach provides solution to decision Problems	5) Branch & Bound technique is used to solve optimizations Problem

General method of branch and bound

In branch and bound method searches a state space tree using any search mechanism in which all the children of E-node are generated before any another node become

E node . we assume that each answer node x has a cost $c(x)$ associated with it and that is minimum cost of answer node $c(x)$ is to be found

The three types of search strategies in branch and bound for finding answer node is

1) FIFO search (Previously seen)
2) LIFO search
3) Least cost search (LC) search

Least cost search (LC)

In both LIFO and FIFO branch and bound the selection rule for the next E-node is some times blind i.e selection rule for next E-node does not give any preference to a node that has very good chance of getting the search to an answer node quickly.

Thus when node 30 is generated. It will lead to an answer node in one move. However, the rigid FIFO & LIFO rule first requires the expansion of all live nodes generated before node 30 was expanded. This results slowness of getting answers

The search for an answer node can be speeded by using intelligent ranking function for live nodes. The next E-node among these live nodes is selected on the basis of the ranking function. If in the 4-queens example we use ranking function that assigns node 30 a better rank than all other live nodes

then node 30 will become the E-node following node 29. The remaining live nodes will never becomes E-nodes as the expansion of node 30 results answer node (31).

[The ideal way to assign ranks to nodes would be on basis of the additional computational effort (or) cost needed to reach an answer node from the live node

For any node x, cost can be guessed in two measures

* ① The number of nodes in subtree x that need to be generated before an answer node is generated becomes a cost

② The number of levels nearest to answer node from any node x becomes a cost]

* If we use measure 2, the cost of root node in Figure ② is 4 (i.e node 31 is four levels from 'node 1). Similarly cost of node 18 and 34 is 3, 29 and 35 is 2, 30 and 38 is 1. The E-nodes generated using measure 2 are 1, 18, 29, 30 (in order) and other are 2, 34, 50, 19, 24, 32 & 31

* If we use measure 1. The number of node in subtree must be 4 before an answer node. It would always result minimum number of nodes

The disadvantage of using both cost measures is

* They explore subtree x and answer node again & again

For this reason search algorithms usually rank nodes only on the basis of estimate $\hat{g}()$

Each node x in search tree is associated with cost $c(x)$. Then $c(x)$ = cost of reaching the current node x (E-node) from the root + the cost of reaching an answer node from node x

$$c(x) = h(x) + g(x)$$

Get an approximation of $c(x)$, $\hat{c}(x)$ such that

$$\hat{c}(x) = f(h(x)) + \hat{g}(x)$$

$\hat{g}(x)$ is same as $g(x)$, where $h(x)$ is the cost of reaching x from the root and $f()$ is any non decreasing function that does not consider the effort already expanded in reaching the live node and all we concerned is minimizing the additional effort we spend to find an answer node

A search strategy that uses cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ to select the next E-node, would always choose from collection live nodes with least $\hat{c}()$. Hence, such search strategy is is called an LC search (Least cost search). The following observation can be made from above cost function $\hat{c}()$

1) If $g=0$, $f=1$, this LC-search is a BFS algorithm which generate node by level

2) If $f=0$, we would normally except $\hat{g}(y) \leq \hat{g}(x)$ where y is child of x. Hence, following x, y will become the E-node, then one of the y's children will become E-node, next one of y's grandchildren

will become the E-node, and so-on - This LC search is a D-search

3) if x is a solution node then cost function can be viewed as

$$\hat{c}(x) < c(x) \text{ and}$$

$$\hat{c}(x) = c(x) \quad \text{if } x \text{ is a solution-node}$$

4) otherwise, $c(x) = \infty$

-> As an simple example consider 8-Puzzle problem

Initial state ⟶ Final state

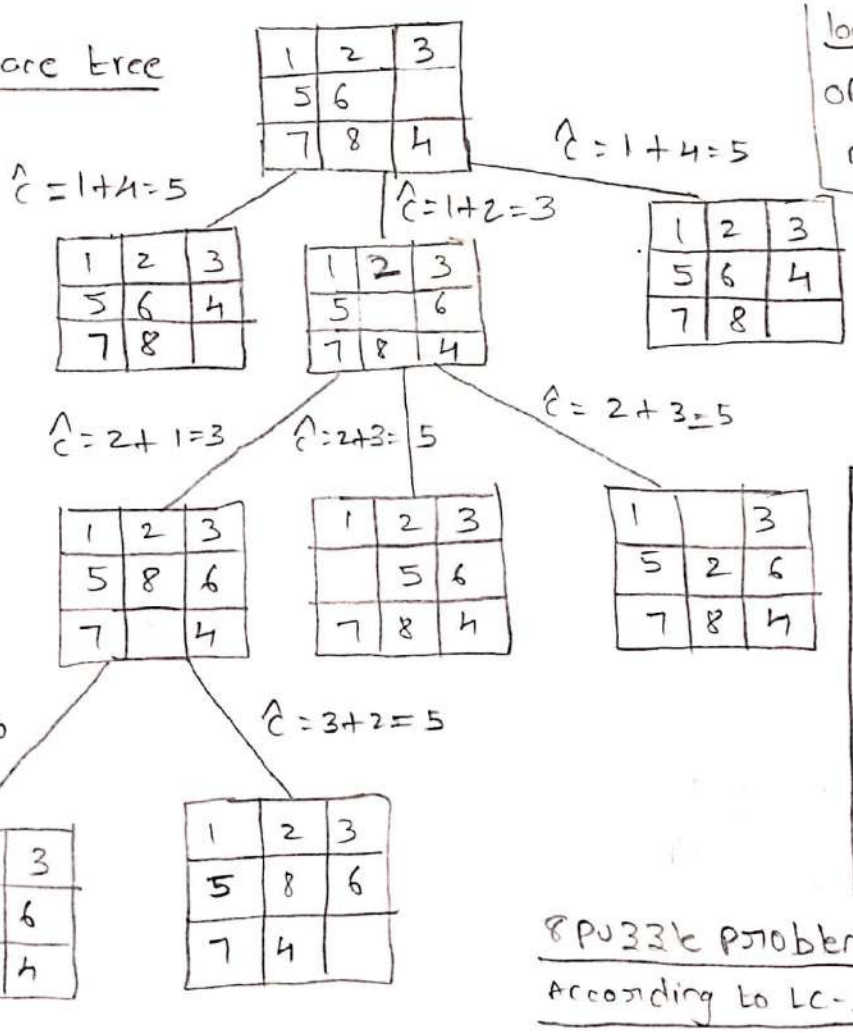
1	2	3
5	6	
7	8	4

1	2	3
5	8	6
	7	4

(goal arrangement)

lower bound: Number of moves should be minimum

State space tree



Note: If any node x has many changes when compared with goal arrangement, it requires more number of moves to reach Final state

8 Puzzle problem

According to LC-Search B&B

Note: In case of tie left most node is selected

Control Abstraction of LC-search

```
Algorithm LCSearch(t)
{
  if (t is an answer node) then
  { write (t)
    return;
  }
  E ← t   // E-node
repeat
{
  for each child x of E do
    {
      if (x is answer node) then
      {
        output the path from x to t
        return
      }
      Add(x); // x is new live node
      (x->parent) := E //Pointer for path to root
    if (no more live nodes) then
      {
        write ("No answer node");
        return;
      }
    E := Least(); //selecting next E node from live node with
                  // minimum cost (ĉ())
}until(false)
}
```

The 15 Puzzle problem

The 15-Puzzle Problem is invented by Sam Loyd in 1878 consists of 15 tiles on a square with capacity of 16 tiles. As shown below

ES- Empty Slot

1	3	4	15
2	ES	5	12
7	6	11	14
8	9	10	13

Initial arrangement

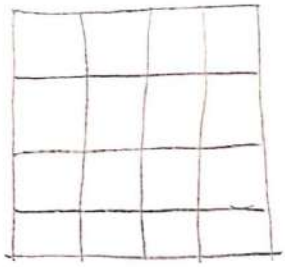


fig (3)

16 tiles

We are given an initial arrangement of the tiles, and the objective is to transform this arrangement in to the goal arrangement as shown below through the series of legal moves

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	→ ES

Fig (4)

The only legal moves are ones in which tile adjacent to empty slot(ES) is moved to ES. In fig (3) we can move any one of tiles numbered 2,3,5 (or) 6 to empty slot. Each move creates a new arrangement of the tiles. These arrangements are called the States of the Puzzle

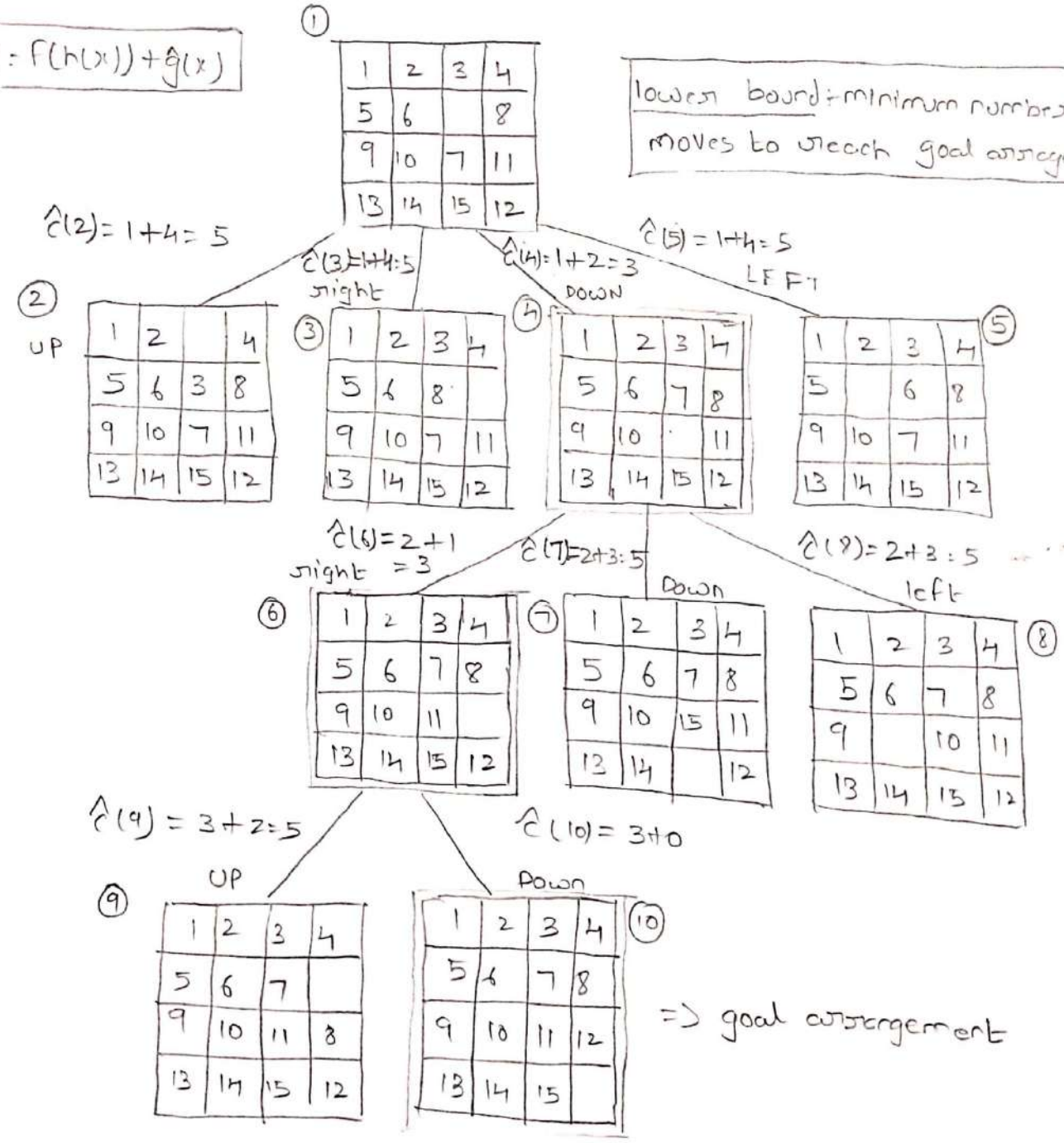
The initial & goal arrangements are called the initial and goal states. Empty slots can be moved in UP, DOWN, RIGHT, LEFT

The most straightforward way to solve the puzzle would be to search the state space for the goal state with contain 16! different arrangements of tiles. only one-half are reachable from any given initial state

→ If you use D-search then we will have lots of branching before goal arrangement. Therefore we will go with LC-search to generate goal arrangement. The state space of 15-puzzle problem is given as

$\hat{c}(x) = f(h(x)) + \hat{g}(x)$

lower bound = minimum number of moves to reach goal arrangement



Applications of Branch & Bound method

Some of major applications of branch and bound are

1) Travelling sales person problem
2) 0/1 knapsack problem

Travelling sales person problem

If there are n cities and cost of travelling from one city to another city is given. A salesman has to start from any one of the city and has to visit all cities exactly ones and has to return to the starting place with shortest distance (or) minimum cost.

let $G=(V,E)$ be a directed graph defining an instance of travelling sales problem let c_{ij} be the cost of the edge (i,j) & $c_{ij}=\infty$ if $(i,j) \notin E(G)$. Assume that every tour starts at vertex 1

We assume each answer node x has a cost $c(x)$ associated with it. A cost function $\hat{c}(\cdot)$ such that $\hat{c}(x) \leq c(x)$ is used to provide lower bound from any node x. If upper$(\hat{U}(x))$ is an upper bound on the cost of a minimum cost, then all live nodes x with $\hat{c}(x) >$ upper$(\hat{U}(x))$ will be killed

we must define cost function $c(x)$ such that

$$\hat{c}(x) \leq c(x) \leq \hat{U}(x)$$

Reduced cost matrix - A row (or) column is said to be reduced if it contains atleast one zero and all remaining entries are non-negative. A matrix is reduced if every row and column is reduced

1) If element 'x' is chosen to be minimum in row i, then subtract it from all other entries in row i including x. Then matrix is said to be row reduced

2) If element 'y' is chosen to be minimum in column j, then subtract it from all other entries in column j including y. Then matrix is said to be column reduced

3) The total amount subtracted from column's and row's is lower bound on the length of a minimum cost tour and can be used as the $\hat{c}(x)$ value for the root of state space tree

→ With every node in state space tree, we associate a reduced cost matrix

let A be the reduced cost matrix for node R. let S be the child of R. such that the edge (R,S) corresponds to including edge(i,j). If S is not a leaf node then the reduced cost matrix for node S can be obtained as follows

1) change all entries in row i & column j of A to ∞

2) set A(j,1) to ∞

3) Apply row reduction & column reduction except for rows & columns containing ∞

4) Total cost of node last node S can be calculated as

$$\hat{c}(s) = \hat{c}(R) + A[i,j] + r$$

Problem

Solve the following instance of travelling sales person using LCBB

	1	2	3	4	5
1	∞	20	30	10	11
2	15	∞	16	4	2
3	3	5	∞	2	4
4	19	6	18	∞	3
5	16	4	7	16	∞

Solution:

Row reduction

minimum in row

	1	2	3	4	5	
1	∞	20	30	10	11	10
2	15	∞	16	4	2	2
3	3	5	∞	2	4	2
4	19	6	18	∞	3	3
5	16	4	7	16	∞	4

Subtract them with remaining row element

21 total

$$\Rightarrow \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

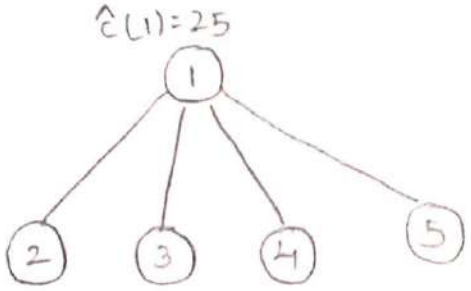
column reduction

$$\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

1 0 3 0 0 = 4 (total)

$$\Rightarrow \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

Total amount subtracted, $r = 21 + 4 = 5$ it is taken as lower bound $\hat{c}(x)$ value for state space tree



Part of state space tree

* consider path (1,2) : change all entries of first row and second column to ∞ & set A(2,1) to ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & 0 \end{bmatrix} \begin{matrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{matrix}$$

Apply row reductions & column reduction Here r=0

$$\hat{c}(2) = \hat{c}(1) + 10 + 0$$
$$= 25 + 10 + 0 = 35$$

* consider Path (1,3) : Change all entries of first row & third column to ∞ and set A(3,1) to ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{bmatrix} \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

Applying row and column reduction

r = 11

11

$$\hat{c}(3) = \hat{c}(1) + A(1,3) + r = 25 + 17 + 11 = 53$$

* consider path (1,4) : Change all entries of row 1 & column 1 to ∞ set A(4,1) to ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

Apply row & column reduction

$\Rightarrow r = 0$

$\hat{c}(4) = \hat{c}(1) + A(1,4) + r = 25 + 0 + 0 = 25$

*Consider the path (1,5) : change the all entries of first row and fifth column to ∞ and set $A(5,1) = \infty$

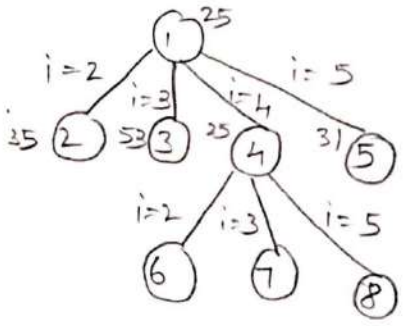
$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 12 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix} \begin{matrix} \\ 2 \\ \\ 3 \\ \\ \end{matrix} \quad \Rightarrow \quad \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

Total 5

$\hat{c}(5) = \hat{c}(1) + A(1,5) + r = 25 + 1 + 5 = 31$

since the minimum cost is 25, select node 4

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$



consider path (4,2) : change all entries of fourth row and 2nd to infinite and set A(2,1) to ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

Apply row & column reduction

$\Rightarrow r = 0 \qquad \hat{c}(2) = \hat{c}(4) + A(4,2) + r$

$\hat{c}(2) = 25 + 3 + 0 = 28$

consider path (4,3) : change all entries of 4th row & 3rd column to ∞ and A(3,1) to ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix} 2 \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & \infty \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

11

$r = 2 + 11 = 13$

$\hat{c}(3) = \hat{c}(4) + A(4,3) + r$

$= 25 + 12 + 13 = 50$

consider path (4,5) : change all entries of 4th row & 5th column to ∞ and A(5,1) to ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix} 11$$

After applying row & column reduction

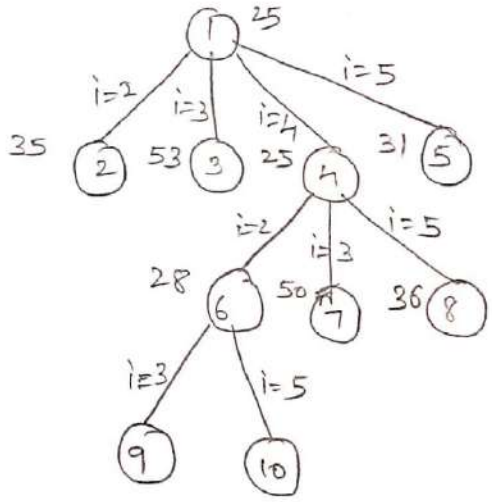
$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

$\pi = 11 + 0 = 11$

$\hat{c}(5) = \hat{c}(4) + A(4,5) + \pi$

$= 25 + 0 + 11 = 36$

Since minimum cost is 28, select node 2



The matrix obtained for path (4,2) is considered as reduced cost matrix

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

Consider the path (2,3) : Change all entries of second row & third column is ∞ and set A(3,1) to ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix} 2$$

11

Apply row & column reduction

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

$\pi = 2 + 11 = 13$

$\hat{c}(3) = \hat{c}(2) + A(2,3) + r = 28 + 11 + 13 = 52$

consider the path (2,5): change all entries 2nd row 5th column to ∞ and set A(5,1) to ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

Apply row reduction and column reduction

$\Rightarrow r = 0$

$\hat{c}(3) = \hat{c}(5) + A(5,1) + r = 28 + 0 + 0 = 28$

Since the minimum cost is 28, select node 5. The matrix obtained for (2,5) is considered as reduced cost matrix

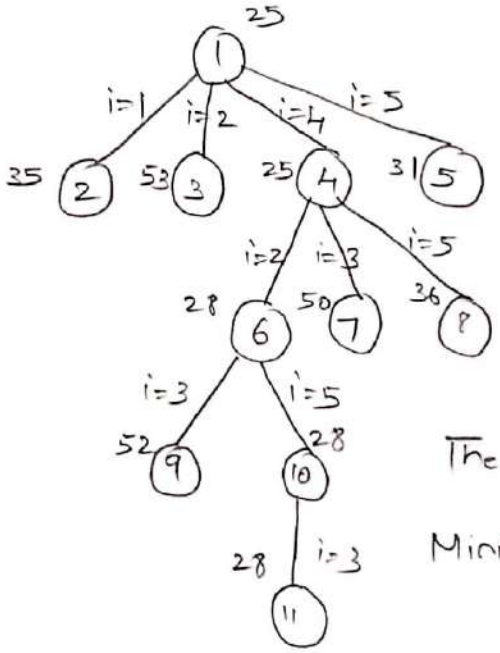
consider path (5,3): change all elements in 5th row and 3rd column to ∞ and set A(3,1) to ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

Apply row reduction and column reduction

$\Rightarrow r = 0$

$\hat{c}(3) = \hat{c}(5) + A(5,3) + r = 28 + 0 + 0 = 28$



The path is 1->4->2->5->3->1 and

Minimum cost = 10+6+2+7+3 = 28

e.g.2 Solve the following instance of travelling salesperson problem using LCBB

$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

2) 0/1 Knapsack problem

0/1 Knapsack problem can be solved using

a) LC Branch and Bound

b) FIFO Branch and Bound

a) 0/1 Knapsack problem using LC Branch and Bound

The 0/1 knapsack problem states that, there are n objects given and capacity of knapsack is m. Then select some objects to fill the knapsack in such a way that it should not exceed the capacity of knapsack and maximum profit can be earned

$$\text{maximize} \quad \sum_{i=1}^{n} p_i x_i$$

subject to $w_1x_1 + w_2x_2 + \cdots + w_nx_n \leq m$, $x_i = 0$ or 1

i.e $\sum_{i=1}^{n} w_i x_i \leq m$

A branch and bound technique cannot directly apply to knapsack

Problem. Because the branch and bound deals deals only the minimization problems. we modify the knapsack problem to the minimization problem. The problem is modified as

$$\text{minimumum} \quad -P_1x_1 - P_2x_2 \cdots -P_nx_n$$

$$\text{i.e} \quad -\sum_{i=1}^{n} P_ix_i \quad \text{where} \quad x_i = 0 \text{ (or) } 1$$

let $\hat{c}(x)$ and $\hat{u}(x)$ are two cost functions, such that $\hat{c}(x) \le c(x) \le \hat{u}(x)$. where $c(x)$ is the cost for answer node x which lies between two functions called upper and lower bounds $\hat{c}(x)$ and $\hat{u}(x)$

Initially we compute upper($\hat{u}(x)$) & lower ($\hat{c}(x)$) bounds at root node $\hat{c}(1)$ and $\hat{u}(1)$. consider the first variable x_1 (i.e selecting object 1 to take decision. The x_1 takes 0 (or) 1. compute lower bounds and upper bounds on each case variables like ($x_2, x_3 \ldots x_n$). select the node whose cost is minimum i.e,

$$\hat{c}(x) = \min\{c(\text{lchild}), c(\text{rchild})\}$$

$$\hat{c}(1) = \min\{\hat{c}(2), \hat{c}(3)\}$$

Problem

1) Draw a portion of state space tree generated by LCBB by the following knapsack problem $n = 4$ Profits (P_1, P_2, P_3, P_4) $(10, 10, 12, 18)$

and weights (w_1, w_2, w_3, w_4) $(2, 4, 6, 9)$; where knapsack capacity is $m = 15$ using LCBB

Solution: Convert the profits to negative (P_1, P_2, P_3, P_4)

$(-10, -10, -12, 18)$

Now calculate the lower bound & upper bound for each node

Place the first item in bag i.e it's weight 2, remaining weight = 15 - 2 = 13

Place second item in bag i.e it's weight 4, remaining weight = 13 - 4 = 9

Place third item in bag i.e it's weight 6, remaining weight = 9 - 6 = 3

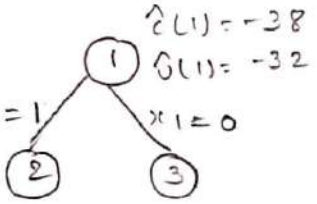
Here fractions are not allowed in calculation of upper bound, so we cannot place the fourth item in bag because it weight is 9

∴ profit earned $-10 - 10 - 12 = -32$

Now calculate the lower bound, where fractions are allowed, so we now can place 4th item too

$$\therefore \text{Lower bound} = -10 - 10 - 12 - \frac{3}{\cancel{6}^{2}} \times \cancel{18}^{3} = -38$$

$\therefore \hat{c}(1) = -38$ & $\hat{u}(1) = -32$, Node 1 is E-node and expanded further



->Now calculate $\hat{c}(\cdot)$, $\hat{u}(\cdot)$ values for node 2 & 3

* For node 2 (ie $x_1 = 1$)

$$\hat{c}(2) = -10 - 10 - 12 - \frac{3}{6} \times 18 = -38$$

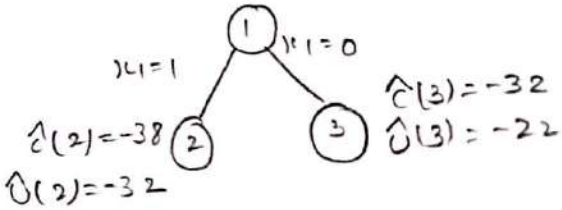
$$\hat{u}(2) = -10 - 10 - 12 = -32$$

* For node 3 (ie $x_1 = 0$)

$$\hat{c}(3) = -10 - 12 - \frac{5}{\cancel{6}^{1}} \times \cancel{18}^{3} = -32$$

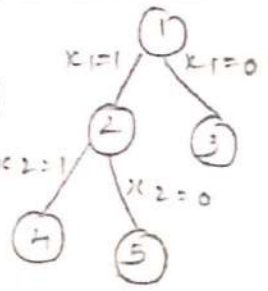
$$\hat{u}(3) = -10 - 12 = -22$$

$15 - 4 = 11$
$11 - 6 = 5$



Select node with minimum lower bounds i.e $\min\{\hat{c}(2), \hat{c}(3)\}$ =

$= \min\{-38, -32\} = -38$



∴ Node 2 become next E-node. It is expanded and Node 4 & 5 generated

→ Now calculate $\hat{c}(\cdot)$ & $\hat{u}(\cdot)$ for node 4 & node 5

* For node 4 (i.e $x_2 = 1$)

$$\hat{c}(4) = -10 - 10 - 12 - \frac{3}{9} \times 18 = -38$$

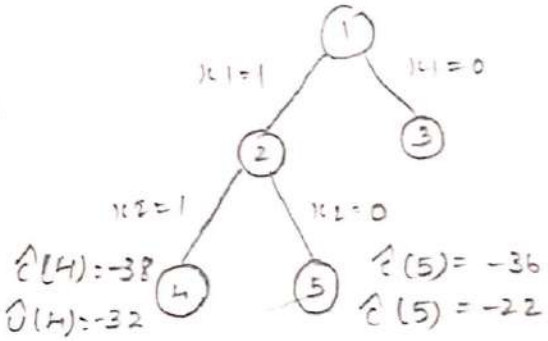
$$\hat{u}(4) = -10 - 10 - 12 = -32$$

15 - 2 = 13
13 - 6 = 7

* For node 5

$$\hat{c}(5) = -10 - 12 - \frac{7}{9} \times 18^{2} = -36$$

$$\hat{u}(5) = -10 - 12 = -22$$



select minimum lower bound

$$\min\{\hat{c}(4), \hat{c}(5)\} = \min\{-38, -36\} = -38$$

∴ Node 4 is next E-node, it is expanded to (6) & (7)

→ Now calculate $\hat{c}(\cdot)$ & $\hat{u}(x)$ for node 6 & 7

* For node 6 (i.e $x_3 = 1$)

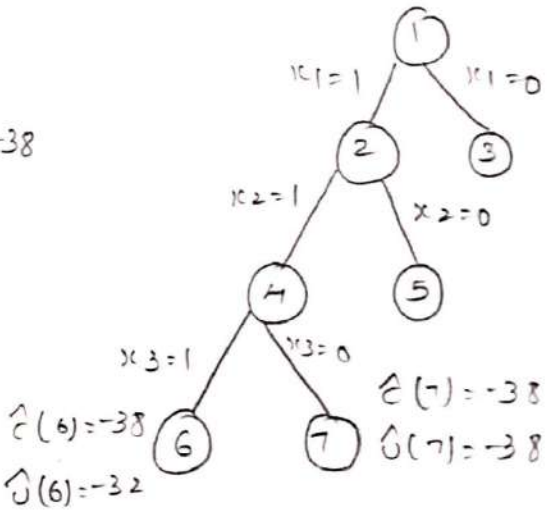
$$\hat{c}(6) = -10 - 10 - 12 - \frac{3}{9} \times 18^{2} = -38$$

$$\hat{u}(6) = -10 - 10 - 12 = -32$$

* For node 7 (ie $x_3 = 0$)

$$\hat{c}(7) = -10 - 10 - 18 = -38$$

$$\hat{u}(8) = -10 - 10 - 18 = -38$$



Select minimum lower bound

$$\min\{\hat{c}(6), \hat{c}(7)\} = \min\{-38, -38\} = -38$$

since lower bounds are same, select node with minimum upper bound i.e $\min\{\hat{u}(6), \hat{u}(7)\} = \min\{-32, -38\} = -38$

∴ Node 7 is selected as next E node and it is further expanded to 8 & 9

-> Now calculate $\hat{c}(\cdot)$, $\hat{u}(\cdot)$ for node 8 & 9

* For node 8 ($x4 = 1$)

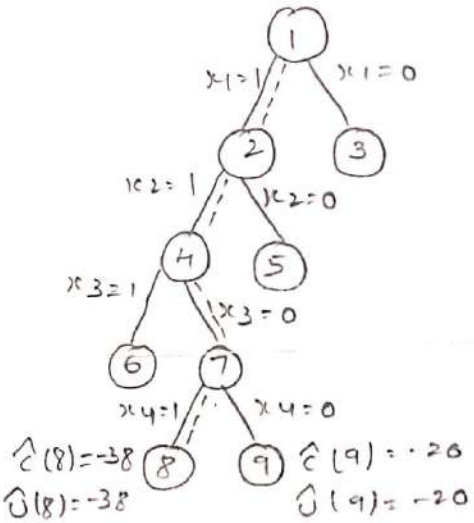
$\hat{c}(8) = -10 - 10 - 18 = -38$

$\hat{u}(8) = -10 - 10 - 18 = -38$

* for node 9 ($x4 = 0$)

$\hat{c}(9) = -10 - 10 = -20$

$\hat{u}(9) = -10 - 10 = -20$



select node with minimum lower bound

$\min\{\hat{c}(8), \hat{c}(9)\} = \min\{-38, -20\} = -38$

∴ Node 8 ($x4 = 1$) is selected

-> The path which we extended determines $x1, x2, x3, x4$ values

i.e $x1 = 1$ $x2 = 1$ $x3 = 0$ $x4 = 1$ with profit = 38

2) 0/1 knapsack problem using FIFO Branch & Bound

We already know FIFO Branch & Bound uses queue to solve problems in First in First out manner. Now let us know through FIFOBB algorithm using same knap sack problem

instance

1) Draw a portion of state space tree generated by FIFOBB by the following knapsack problem n=4 Profit $(P_1, P_2, P_3, P_4) = (10, 10, 12, 18)$ and weights $(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$, where knapsack capacity is $\boxed{m = 15}$

Sol: Now calculate the lower bound & upper bound for each node in FIFO fashion using queue. Initially the root node is node ① becomes E-node and queue of live nodes is empty

queue

Place the first item in bag i.e it's weight 2, remaining weight = 15-2 = 13
Place the second item in bag i.e it's weight 4, remaining weight = 13-4 = 9
Place the third item in bag i.e it's weight 6, remaining weight = 9-6 = 3

∴ Profit earned is -10-10-12 = -32 (Note: profits converted in to negative values)
$\hat{u}(1)$

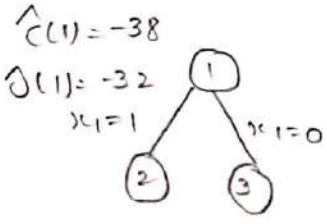
Now calculate the lower bound, where fractions are followed, so we now can place 4th item too

∴ Lower bound = $-10-10-12-\frac{3}{9}\times 18 = -38$

$\hat{c}(1) = -38$
$\hat{u}(1) = -32$ ①

∴ $\hat{c}(1) = -38$ & $\hat{u}(1) = -32$, since node ① is E-node it is expanded and Nodes 2 and 3 are generated and added to queue (Q)

Q | 2 | 3 |



-> Now Node 2 becomes E-node because it is First-in to the queue. Now calculate $\hat{c}(\cdot)$ & $\hat{u}(\cdot)$ values for node 2. For node 2 (i.e $x_1 = 1$)

$\hat{c}(2) = -10-10-12-\frac{3}{9}\times 18 = -38$

$\hat{u}(2) = -10-10-12 = -32$

Now Node-2 childerns are gerenated i.e 4 & 5 and added to queue

Q | 3 | 4 | 5 |

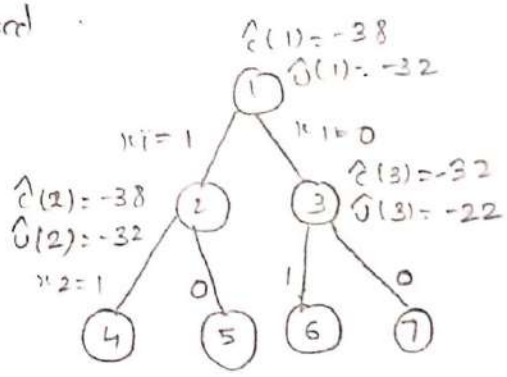
-> Node 3, becomes next E-node, it's $\hat{c}(\cdot)$, $\hat{u}(\cdot)$ value is given as

$$\hat{c}(3) = -10-12-\frac{5}{8}\times 18^{2} = -32$$

$$\hat{u}(3) = -10-12 = -22$$

It childern nodes are gerenated, nodes 6 and 7 and added to queue

Q | 4 | 5 | 6 | 7 |



-> Now node 4, becomes next E-node, it $\hat{c}(\cdot)$, $\hat{u}(\cdot)$ value is given as

$$\hat{c}(4) = -10-10-12-\frac{3}{8}\times 18^{2} = -38$$

$$\hat{u}(4) = -10-10-12 = -32$$

It childern nodes 8 & 9 are gerenated and added to queue

Q | 5 | 6 | 7 | 8 | 9 |

-> Next E-node is 5, it's $\hat{c}(\cdot)$, $\hat{u}(\cdot)$ values is given as

$$\hat{c}(5) = -10-12-\frac{7}{8}\times 18^{2} = -36$$

$$\hat{u}(5) = -10-12 = -22$$

Node 5 is killed because $\hat{c}(5) > \hat{c}(2)$ (Parents upper) $-22 > -32$

Q | 6 | 7 | 8 | 9 |

-> Next E-node is 6, it's $\hat{c}(\cdot)$, $\hat{u}(\cdot)$ values is given as

$$\hat{c}(6) = -10-12-\frac{5}{8}\times 18^{2} = -32$$

$$\hat{u}(6) = -10-12 = -22$$

Node 6 is killed

Q | 7 | 8 | 9 |

->Next Enode is 7, it's $\hat{c}(\cdot)$ & $\hat{u}(\cdot)$ is given as

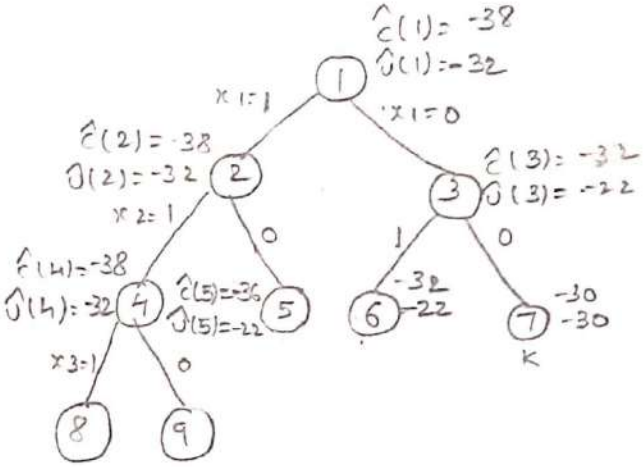
$\hat{c}(7) = -12-18 = -30$

$\hat{u}(7) = -12-18 = -30$

Node 7 is killed because $\hat{c}(7) > \hat{c}(2)$

$-30 > -32$

	8	9



K-means killed

->Next Enode is 8 it's $\hat{c}(\cdot)$ & $\hat{u}(\cdot)$ is given as

$\hat{c}(8) = -10-10-12-\frac{3}{8}\times 16 = -38$

$\hat{u}(8) = -10-10-12 = -32$

Here $\hat{c}(8) \leq \hat{c}(4)$, E-node is further generated to 10 & 11 and added to queue

	9	10	11

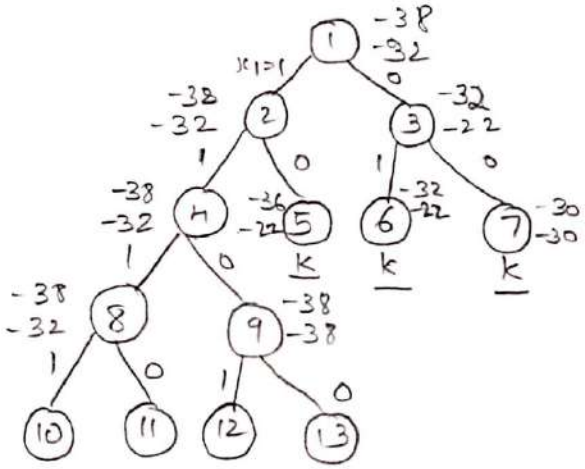
=>Next Enode is 9, it's $\hat{c}(\cdot)$ & $\hat{u}(\cdot)$ is given as

$\hat{c}(9) = -10-10-18 = -38$

$\hat{u}(9) = -10-10-18 = -38$

Here $\hat{c}(9) \leq \hat{c}(4)$, E-node 9 is further generated to 12 & 13 and added to queue

10	11	12	13



->Next E-node is 10 (i.e $x_4=1$)

it is infeasible and so killed because having $x_1=1, x_2=1, x_3=1, x_4=1$

it is not possible to calculate $\hat{U}(10)$

11	12	13

Q

->Next E-node is 11, it's $\hat{c}(\cdot)$, $\hat{U}(\cdot)$ values is given as $(x_4=0)$

$$\hat{c}(11) = -10-10-12 = -32$$
$$\hat{U}(11) = -10-10-12 = -32$$

Node is 11 is also killed because $\hat{c}(11) > \hat{c}(8)$
$-32 > -38$

12	13

Q

-> Next E-node is 12, it's $\hat{c}(\cdot)$, $\hat{U}(\cdot)$ values is given as $(x_4=1)$

$$\hat{c}(12) = -10-10-18 = -38$$
$$\hat{U}(12) = -10-10-18 = -38$$

Node 12 becomes answer node because $\hat{c}(12) \leq \hat{c}(8)$

13

Q

->Next E-node is 13, it's $\hat{c}(\cdot), \hat{U}(\cdot)$ values is given as

$$\hat{c}(13) = -10-10 = -20$$
$$\hat{U}(12) = -10-10 = -20$$

Node 13 is killed because $\hat{c}(13) > \hat{c}(8)$
$-20 > -30$

Q

Now Queue is empty and no more live nodes in queue(Q), then search terminates. The final result is $x_1=1$, $x_2=1$, $x_3=0$, $x_4=1$