Unit-I:

Introduction: What is an Algorithm?, Algorithm specification, Performance analysis

Divide and Conquer: General method, Binary search, Finding the maximum and Minimum, Merge sort, Selection sort, Stressen's Matrix multiplication.

Unit-II

Greedy Method: General method, knapsack problem, Job scheduling with Deadlines, Minimum cost spanning Trees, Optimal storage on tapes, Single-source shortest paths.

Dynamic programming: General method, Multistage graphs, All-pairs shortest paths, Optimal binary search trees, 0/1 knapsack, The traveling sales person problem.

Unit-III:

Basic Traversal and Search Techniques: Techniques for binary trees, Techniques for Graphs, Connected Components and Spanning trees, Bi-connected components and DFS

Backtracking: General method, 8-queens problem, Sum of Subsets problem, Graph coloring and Hamiltonian cycles, Knapsack Problem.

Unit-IV:

Branch and Bound: The method, Travelling Salesperson, 0/1 knapsack problem, Efficiency considerations.

Lower Bound Theory: Comparison trees, Lower bounds through reductions - Multiplying triangular matrices, inverting a lower triangular matrix, Computing the transitive closure.

Scanned with CamScanner

Unit-I

Np What is an algorithm?

Algorithm:- The word algorithm comes from the name of persian author Abu Jafar Mohammed ibn musa al Khowarizmi

ions Def: A finite set of step by step instructions to solve (or) to accomplish a particular task.

textbook on mathematics

→ Mainly an algorithm should satisfy five criteria's (or) properties 2M

1) Input
2) output
3) Definiteness
4) Finiteness
5) Effectiveness

Def:-

1) Input: 0 (or) more quantities that are supplied externally

2) Output: result of algorithm must produce atleast one quantity as Output criteria must produce a result.

3) Definiteness: The instruction must be clear and unambiguous.

4) Finiteness: The algorithm should execute some set of statements in each case and terminate at one point

5) Effectiveness: The instructions must be feasible. (modified according to the algorithm) Instruction must be simple and easy. We must be able to write algorithm using pencil and paper.

The study of algorithms is used in many areas. the four important (or) major areas to study the algorithm are:

Scanned with CamScanner

① How to devise an algorithm
② How to validate algorithms
③ How to analyse algorithms
④ How to test a program.

1) How we write an algorithm, gather the requirements, produce the solution, implement the solution using appropriate technique

2) Validate - checking, by giving inputs to the algorithm we check the correctness - whether the algorithm is giving correct output or not.

3) analysis - analyse (based on operations) the memory space (to store algo in the system memory), how much time it takes to execute

4) test a program — Debugging / Profilling
is done by 2 techniques

Debugging: traces the errors, corrects the error but will not check correctness of the result

Profilling: traces the errors, corrects the error and also checks the correctness of result.

Algorithm: general english → no specifications
pseudo code: general code + algorithm
Present programmers usually prefer pseudo code technique. — Algorithm specifications

→ "Algorithm" is derived from the Persian author "Abu Jafar Mohammed ibn musa al khowarizmi" who wrote books on Mathematics

Scanned with CamScanner

Algorithm Specifications:- (steps to write algorithm using pseudo code)

Algorithm is divided into two sections

1) Algorithm heading: It consists of name of algorithm with parameters, problem description, input, output

2) Algorithm body: It consists of logical instructions (Code)

The rules for writing an algorithm:

1) Algorithm is a procedure consisting of head and body. The head consists of name of the algorithm and parameter list

Syntax: Algorithm Name $(P_1, P_2, \ldots P_n)$

Eg: factorial(n, a)

2) Head section Consists the following things:

// problem description: finding the factorial of n numbers

// Input: (n, a)

// output: factorial of number

3) The body of an algorithm written in which various programming constructs like for loop, while condition or some assignment statements ("Read" input, "write" output)

4) The Compound statements should be enclosed within { }

5) Single line of comments are written using "//" as beginning of comment

6) The identifier should begin by letter and not by digit. An identifier can be a combination of alpha numeric string.

Scanned with CamScanner

7) Using assignment operator (← (or) (:=)), an assignment statements be given.

Eg: Variable ← expression

8) There are other types of operators such as boolean operators, arithmetic operators, logical operators and relational operators

9) The array indices are stored within [ ] square brackets. The index of array usually starts at '0'. The multi-dimensional arrays can also be used in algorithm

10) The Inputs and outputs can be done by using read and write statements.

The conditional statements such as if-then or if-else

```
if condition              if (condition) then
then                         statements
    statements            else
                              statements
```

While statement can be written as :

```
while (condition) do
{
statement 1
statement 2
   :
statement n
}
```

The general form for writing for loop :

```
for Variable ← Val 1 to Val n do (or) step-1 do
{
statement1
```

(Val 1 → initialisation cond; Val n → termination condition)

⋮

statement n

}

do-while

The repeat-until statements can be written as

repeat

{

statement 1

⋮

statement n

}

until (condition)

The break statement is used to exit from inner loop. The return statement is used to return control from one point to another

Examples:

1) Addition of two numbers

Name of the algorithm : add (a, b)

//problem description : Algorithm for addition of two numbers

//Input: a, b

//output : c

Read a, b

c = a+b

print c

Scanned with CamScanner

2) Factorial of a number:

Name of the algorithm: Factorial(n)

// problem description: Algorithm for finding the factorial of a given number

// Input: n

// Output: factorial of n

```
if (n=1) then
    return(1)
else
return n*factorial(n-1)
```

3) Algorithm to check whether the given number is even or odd

Name of the algorithm: Even or odd(n)

// problem description: Algorithm to check whether the given number is even or odd

// Input: n

// output: ~~even or odd~~ n is even or odd

```
{
Read n
if (n%2==0) then
write ("Given number is even")
else
write ("Given number is Odd")
}
```

4) Write an algorithm for sorting the 'n' elements

Name of the algorithm: sort(n[ ])

// problem description: algorithm for sorting the elements

// Input: n[ ]

// output:

Scanned with CamScanner

```
{
for i ← 1 to n do
for j ← 1 to n-1 do
 { if (a[i] > a[j]) then
   {
   temp ← a[i];
    a[i] ← a[j];
    a[j] ← temp;
   }
 }
Write("The sorted array")
}
```

5)-Algorithm for finding the multiplication of two matrices

Name of the algorithm: multiplication(a, b)

//Problem description: Algorithm for finding the multiplication of two matrices

// Input: a,b a[ ][ ], b[ ][ ]

//output: c[ ][ ]

```
{
Read a[ ][ ], b[ ][ ]
for i ← 1 to n do
for j ← 1 to n do
   c[i][j] ← 0
for k := 1 to n do
 c[i,j] ← c[i,j] + A[i,k] + B[k,j]
 Print c
```

Scanned with CamScanner

6) Algorithm to count the sum of n elements

Name of the algorithm : sum(n)

// problem description : Algorithm to count the sum of n elements

// Input: n

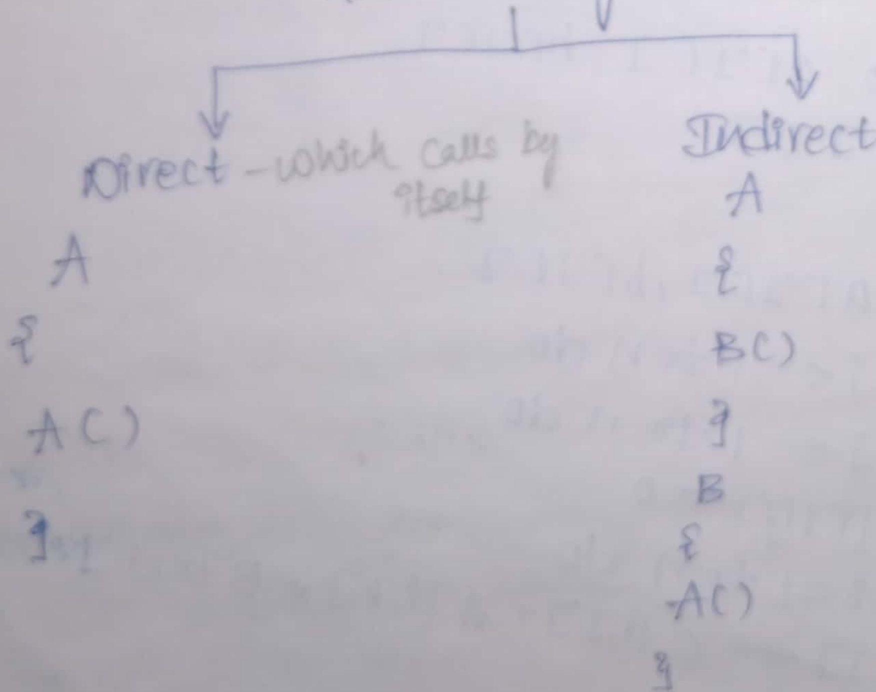
// output: sum of n

{
1. Read n
2. i=1, sum=0
3. if (i>n) goto 7
4. S=S+i
5. i=i+1
6. goto 3
7. print S
}

Recursive Algorithms:-

An algorithm call by itself is called recursive algorithm

→ Recursion reduces the no. of instruction in the program thereby reducing the complexity



Towers of Hanoi (n, a, b, c)

// problem Description : An algorithm to solve towers of Hanoi

// Input : n no. of disks in peg A (source)

// output : n no. of disks in peg C (destination)

```
for i:=1 to n do
{ j:=i;
for k:=1+i to n do
if (a[k] < a[j]) then
  j:=k
  t:=a[i];
  a[i]:=a[j];
  a[j]:=t;
  }
}
```

```
{
  if (n≥1) then
  {
  TowersofHanoi(n-1, x, y, z);
  write ("move top disk from tower", x, "to top of tower", y);
  TowersofHanoi(n-1, z, y, x);
  }
}
```

Performance analysis:-

To measure the performance of an algorithm, there are two techniques

a) space complexity

b) time complexity

1. Space Complexity: amount of storage i.e required to execute the algorithm.

The amount of storage is calculated by

~~Space(s) = c + S(p)~~

$S(P) = C + S_P$

(or)

$S = C + S(P)$

C = Constant

P = instant characteristics

For example:

Name of the Algorithm : add (a, b, c)

// problem description: addition of numbers

// Input : a, b, c are of float type
// Output : returns addition

return a+b+c

Space S = C + S(p) (→ 3 variables (a,b,c))
= 3+0
= 3

Algorithm Add (x,n)
{
sum ← 0.0
for i ← 1 to n do
sum ← sum + x[i]
return sum
}

Space required : S = C+S(p)
= 3+n → array of n
(↓ sum, i, x[ ])

Analysis of Space Complexity:

1. Instruction space
2. Data space
3. stack space

1. Instruction space: The compiler stores the machine code in instruction space
→ It is an fixed part
→ Compiler uses some optimisations techniques to convert

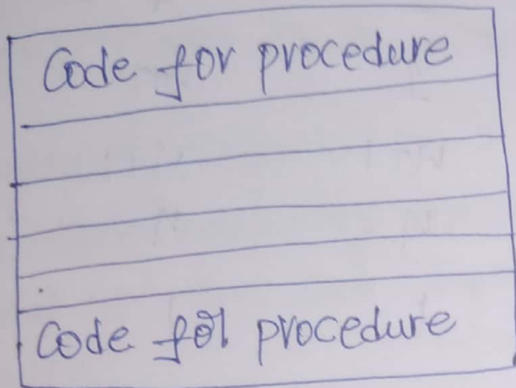
2. Data space: This is occupied by the variables in program.

3. Stack space: Depending on the execution of program are allocated dynamically.

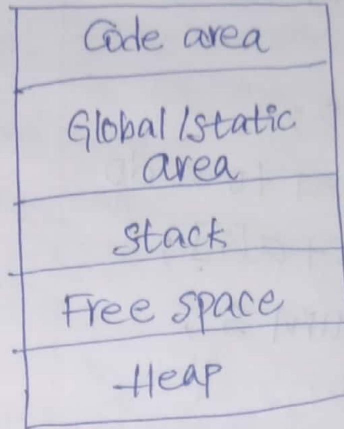
Scanned with CamScanner

→ Mainly used in functions

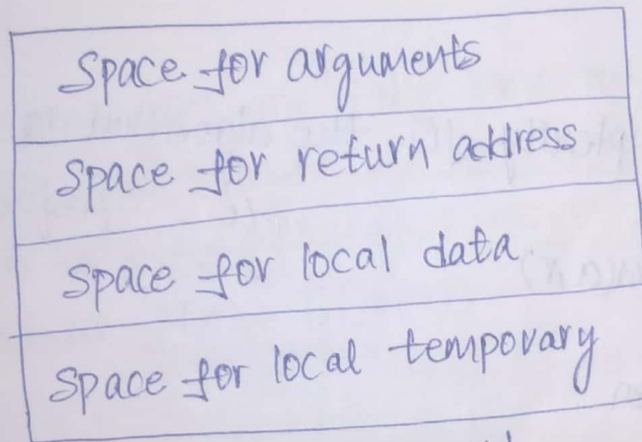
Stack frame work



Core memory



runtime storage



activation record

2) Time Complexity:
The total amount of time that is required to Complete the execution of an algorithm is called the time complexity

$S(P) = C + S_p$ → changes based on the no of i/p's

$T(P) = \text{Compile time} + \text{Run time}$ (Compile time → fixed)

To measure the time complexity, scientists introduced "frequency count"

→ Frequency count is nothing but the number of times an instruction is executed

Statement	s/e (steps per execution)	frequency	total steps
Algorithm sum(a,n)	0	—	—
{	0	—	—
s:=0.0;	1	1	1
for i:=1 to n do	1	n+1	n+1
s:=s+a[i];	1	n	n
return s;	1	1	1
}	0	—	—
total			2n+3

The time complexity for the algorithm is 2n+3

Statement	s/e	frequency n=0	frequency n>0	total steps n=0	total steps n>0
Algorithm Rsum(a,n)	0		—	0	0
{	0		—	0	0
if (n≤0) then	1	1	1	1	1+x
return 0.0;	1	1	0	1	0
else	—	—	—	—	—
return Rsum(a,n-1)+a[n];	1+x	0	1+x	0	1+x
}	—	—	—	—	—
				2	2+2x

Asymptotic Notations:

These are the short hand way representation to represent time complexity

→ We prefer time complexity when compared to space complexity because it is somewhat complex to calculate space complex, as there is a chance to lose data

Bigoh - O → represents upper bound - Max time to execute algorithm

Omega - Ω

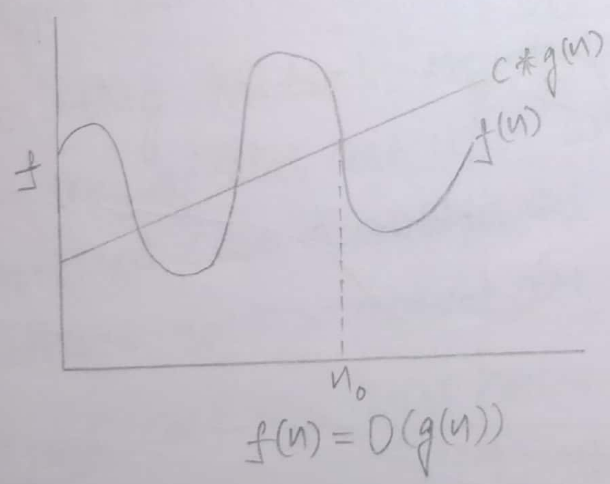
Scanned with CamScanner

Theta - Θ
Small oh - o
Small omega - ω
→ We prefer Bigoh - O to represent time Complexity.

Asymptotic Notation is a short hand way to represent the time Complexity. In asymptotic notations we have several Notations like Bigoh, omega, Theta, small omega and smalloh

1) Bigoh: Bigoh notation denoted by 'O' is a method for representing the upper bound of an algorithm running time

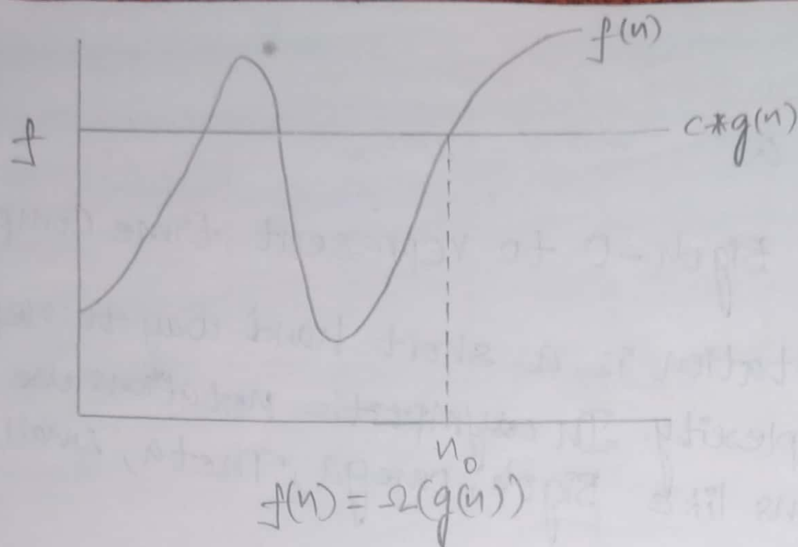
Def: Let $f(n)$ and $g(n)$ are two non-negative functions and if there exists an integer n_0 and a constant 'c' such that $c>0$ and for all integers $n>n_0$ $f(n)<c*g(n)$ then it is denoted as $f(n)=O(g(n))$



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

Omega Notation:
Omega notation is denoted as 'Ω' is a Method of representing the lower bound of algorithm running time

Def: Let $f(n)$ and $g(n)$ are two non-negative functions and there exists a Constant 'c' and an integer 'n_0' such that $c>0$ & $n>n_0$ then $f(n)>c*g(n)$ i.e $f(n)=\Omega(g(n))$

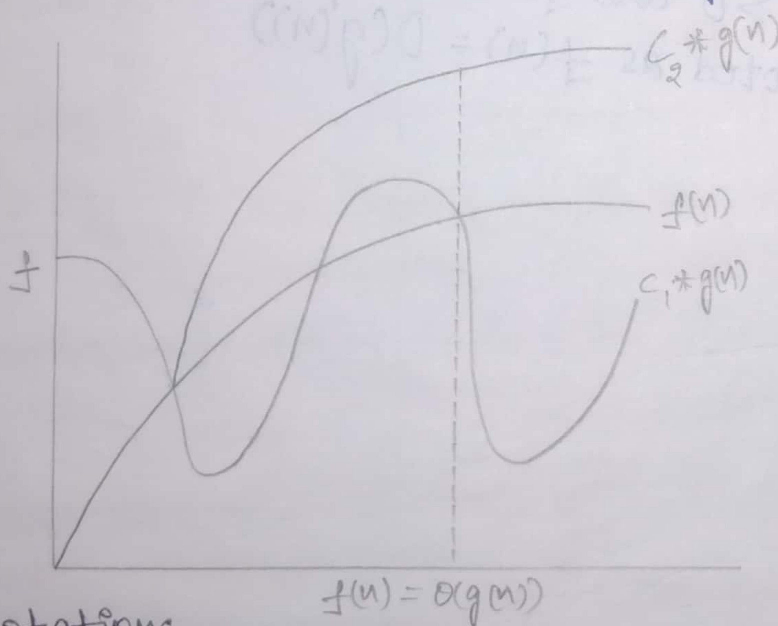


$f(n) = \Omega(g(n))$

Theta notation:

Theta notation denoted as 'Θ' is a method of representing running time between upper bound and lower bound

Def: Let $f(n)$ and $g(n)$ be two non-negative functions there exists two +ve constants c_1 and c_2 such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ and it is denoted as $f(n) = \Theta(g(n))$.



$f(n) = \Theta(g(n))$

Littleoh Notation:

The littleoh is denoted as 'o'. It is defined as:

Let $f(n)$ and $g(n)$ be two non-negative functions such that $f(n) = o(g(n))$

$f(n) = o(g(n))$ iff $f(n) = O(g(n))$ & $f(n) \neq \Theta(g(n))$

Scanned with CamScanner

Little omega Notation:

The little Omega is denoted as ω. It is defined as:

Let $f(n)$ and $g(n)$ be two non-negative functions then

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

such that $f(n) = \omega(g(n))$

$f(n) = \omega(g(n))$ iff $f(n) = \Omega(g(n))$ & $f(n) \neq \Theta(g(n))$

Amortized Analysis:

finding the average running time per operation over a sequence of operations

Amortized cost is calculated by using three techniques

Amortized cost is calculated by using three techniques

1) Aggregate Analysis: Here we calculate amortized cost for sequence of operations

n - no. of operations

T(n) - time required to run sequence of operations

$$\text{Amortized Cost/operation} = \frac{T(n)}{n}$$

2) Accounting Method: we calculate amortized cost per operation

We compare actual cost with amortized cost

if amortised cost < actual cost - then credits are used

if amortised cost > actual cost - then credits are stored

amortized cost = actual cost + credits (may be used or stored)

3) ~~Potential Method~~: Conditions in calculating amortized cost:

1) $\sum_{i=1}^{n} c_i' \geq \sum_{i=1}^{n} c_i$

(c_i' → amortized cost, c_i → actual cost)

2) Total credits $= \sum_{i=1}^{n} c_i' - \sum_{i=1}^{n} c_i$

Total credits must non-negative then it is effective

if it is -ve, it is not effective.

Scanned with CamScanner

3) Potential Method:

We need to calculate potential energy / potential. Potential is stored in data structure

If there are 'n' data structures $D_0, D_1, D_2, \ldots D_n$ then

actual cost $= n \rightarrow c_1, c_2, c_3, \ldots c_n$

for i^{th} operation

Amortized Cost $= c_i + \phi(D_i) - \phi(D_{i-1})$

↓ Actual cost; Potential charge

$$\sum_{i=1}^{n} c_i' = \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_r) > 0$$

Divide and Conquer Method:-

General Method:-

In divide and conquer method a given problem is divided into subproblems. These subproblems are solved independently combining the solution of all subproblems into a single solution.

If the subproblem is large then divide and conquer method is reapplied.

In this method recursive algorithms are used

Control abstraction:

```
Algorithm DandC(p)
{
if Small(P) then
   return S(P);
else
{
divide P into smaller instances P1,P2,P3,---Pk; k>=1;
Apply DandC to each of these subproblems;
return Combine (DandC(P1), DandC(P2),----DandC(Pk));
```

Scanned with CamScanner

}

}

→ By using recurren relation we calculate the computing time of Divide and Conquer method

$$T(n) = \begin{cases} g(n) & \text{if } n \text{ is small} \\ T(n_1) + T(n_2) + \cdots + T(n_r) + F(n) & \text{if } n \text{ is large} \end{cases}$$

$T(n_1)$ = total time required to compute 'n_1' subproblem

$T(n_2)$ = total time required to execute 'n_2' subproblem

$F(n)$ = total time required to the problem into subproblems and Combine the solution of subproblems into single solution

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

a = no. of subproblems

n/b =

Recurrence relation is a relation which defines some sequence of equation recursively

Conditions:

$T(n) = T(n-1) + r \rightarrow$ ① General form

$T(0) = 0 \rightarrow$ ② initial term

The computing time of Divide and conquer method is given by the recurrence relation

$$T(n) = \begin{cases} g(n) & \text{if } n \text{ is small} \\ T(n_1) + T(n_2) + \cdots + T(n_r) + F(n) & \text{if } n \text{ is large} \end{cases}$$

$T(n)$ = the total ~~time~~ Computing time for divide and conquer method of size 'n'

$g(n)$ = the Computing time required to solve small inputs

$F(n)$ = the Computing time required in dividing problem P

into subproblems and combining the solutions into a single solution.

If we want to divide a problem of size 'n' into a size of 'n/b' taking F(n) computing time to divide and combine the subproblems and solutions, then the recurren relation for obtaining the computing time of size 'n' is

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

Recurrence Relation:

The Recurrence Relation is an equation that defines a sequence recursively. The general form of recurrence relation is $T(n) = T(n-1) + r \longrightarrow (1)$

$T(0) = 0 \longrightarrow (2)$

eq (1) is called recurrance relation

eq (2) is called initial condition

The recurrance relation have infinite no. of sequences.

→ The recurrance relation can be solved by using 2 methods

1. Substitution method
2. Master's method.

(1) Substitution Method:

The substitution method is a method in which a guess is made for the solution. There are 2 types of substitution methods

a) forward substitution

b) backward substitution

a) Forward substitution:

This method makes use of initial condition to generate the initial term and the next term is generated based on the initial term. This process is continued until some formula is guessed.

Scanned with CamScanner

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

Sol: Let $T(n) = T(n-1) + n \longrightarrow ①$

$T(0) = 0 \longrightarrow ②$

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

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

if $n=3$ then $T(n) = T(3-1) + 3 = T(2) + 3 = 3 + 3 = 6$

By observing above generated equations we can derive a formula $T(n) = \frac{n(n+1)}{2}$

We can also denote $T(n)$ in terms of Bigoh notation as

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

b) Backward Substitution:

In this method backward values are substituted recursively to derive formula

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

Sol: Let $T(n) = T(n-1) + n \longrightarrow ①$

Let $n = n-1$

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

Substitute eq ② in eq ①

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

Let $n = n-2$ in eq ①

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

Substitute ④ in ③ we get

$T(n) = T(n-3) + n-2 + n-1 + n \longrightarrow ⑤$

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

if $k = n$ then

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

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

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

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

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

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

Eg 2) $T(n) = T(n-1) + 1$ with initial condition $T(0) = 0$

a) forward substitution:

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

$T(0) = 0$

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

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

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

b) backward substitution:

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

Let $n = n-1$

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

Substitute eq (2) in eq (1)

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

Let $n = n-2$ in eq (1)

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

Substitute eq (4) in eq (3) we get

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

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

Scanned with CamScanner

(2) Master's Method:-

In this Method the basic recurrance relation equation is

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

where $a \geq 1$, $b > 1$ be Constants

Let $F(n)$ be a function and $T(n)$ define non-negative integers

$T(n)$ can be bounded as follows:

Case i: if $F(n) = O(n^{\log_b^a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b^a})$ — $f(n) < n^{\log_b^a}$

Case ii: if $F(n) = \Theta(n^{\log_b^a})$ then — $f(n) = n^{\log_b^a}$

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

Case iii: if $F(n) = \Omega(n^{\log_b^a + \epsilon})$ for some Constant $\epsilon > 0$ — $f(n) > n^{\log_b^a}$

and if $a \cdot F(n/b) \leq c\, F(n) \{c < 1\}$ (→ regulatory condition)

then $T(n) = \Theta(F(n))$

(1) Eg: $T(n) = 9T(n/3) + n$

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

Here $a = 9$, $b = 3$, $f(n) = n$

$n^{\log_b^a} = n^{\log_3^9} = n^{\log_3^{3^2}} = n^{2 \log_3^3} = n^{2 \times 1} = n^2$

Scanned with CamScanner

$n^{\log_b^a} = n^2$

$f(n) = n$

$f(n) < n^{\log_b^a}$ (case i is applied)

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

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

(2)

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

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

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

$n^{\log_b^a} = n^{\log_2^2} = n^{\log 1} = n^1 = n$

$f(n) > n^{\log_b^a}$ (Case iii is applied)

Then $T(n) = \Theta(F(n)) = \Theta(n^3)$

top down approach

Binary search:-

It is an efficient searching method, while searching the elements using this method the elements in the array should be sorted. An element which is to be searched from the list of elements stored in array and the searched element is called key element.

low - starting index

$\frac{low+high}{2}$

In this technique first find out the middle element $A[m]$ with the key element then 3 conditions need to be tested

then the searched element is in the list

1) if $key = A[m]$ then the searched element is in the list
2) if $key < A[m]$ then search the left sublist
3) if $key > A[m]$ then search the right sublist

Algorithm:

Name of the algorithm: Algorithm Binsearch($A[0..n]$, key)

//problem description: This algorithm is for searching the element by using binary search method

//Input: An array 'A' where the key element is searched
//output: It returns the index of an array element if it is equal to key, otherwise it returns -1

```
low ← 0
high ← n-1
while (low < high) do
{
m ← (low + high) / 2;
if (key == A[m]) then
  return m;  → location of the value
Else if (key < A[m]) then
  high ← m-1;
  [} insearch (A [0 -- m+1], key); ]  → optional
else
  low ← m+1;
[} insearch (A[m+1 ---n], key); ]  → optional
}
return -1;
```

The basic operation in binary operation is comparison of search key with the array elements. To analyze efficiency of binary search we must count the number of times the key gets compared the array elements

→ In the algorithm after one comparison the list of 'n' elements are divided into $\frac{n}{2}$ sublists. The worst case efficiency is that the algorithm compares all the array elements for searching the desired element. In this, one comparison is made and based on this comparison array is divided each time into '$\frac{n}{2}$' sublists. Hence the worst case time complexity is given by

$C_{worst}(n) = C_{worst}\left(\frac{n}{2}\right) + 1$ for $n > 1$ → ①

$C_{worst}(n/2)$ = Computing time required to compare left sublist or right sublist

1 = one comparison is made with middle element

But as we consider the rounded down value when array gets divided the above situation can be written as

$C_{worst}(1) = 1 \longrightarrow$ ② (initial condition) middle value = key value

Assume $n = 2^k$

$C_{worst}(2^k) = C_{worst}(2^{k/2}) + 1$

$C_{worst}(2^k) = C_{worst}(2^{k-1}) + 1 \longrightarrow$ ③

using backward substitution method we can substitute

1. $C_{worst}(2^{k-1}) = C_{worst}(2^{k-2}) + 1$

Then eq③ becomes

$C_{worst}(2^k) = C_{worst}(2^{k-2}) + 1 + 1$

$= C_{worst}(2^{k-2}) + 2$

Then $C_{worst}(2^k) = [C_{worst}(2^{k-3}) + 1] + 2$

$= C_{worst}(2^{k-3}) + 3$

⋮

$C_{worst}(2^k) = C_{worst}(2^{k-k}) + k$

$= C_{worst}(2^0) + k$

$= C_{worst}(1) + k$

$= 1 + k$

$\therefore C_{worst}(1) = 1$

Scanned with CamScanner

$C_{worst}(n) = 1 + \log_2^n \quad n = 2^k$

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

$\log_2^n = k \log_2^2$

~~The binary search~~ $\log_2^n = k.1$

$\therefore k = \log_2^n$

The binary search time complesity is $O(\log_2^n)$

average case

Advantages of Binary Search:

This method is an optimal searching algorithm which can search the desired element very efficiently

Disadvantages of Binary Search:

This algorithm requires the sorted list, then only this method is applicable.

Applications:

1) This Method is an efficient method to search the desired records from the databases.

2) For solving non-linear equations with one unknown value

Merge sort: bits bottom up approach

The Merge sort is a sorting algorithm that uses divide and conquer strategy. In this method, the division is done dynamically.

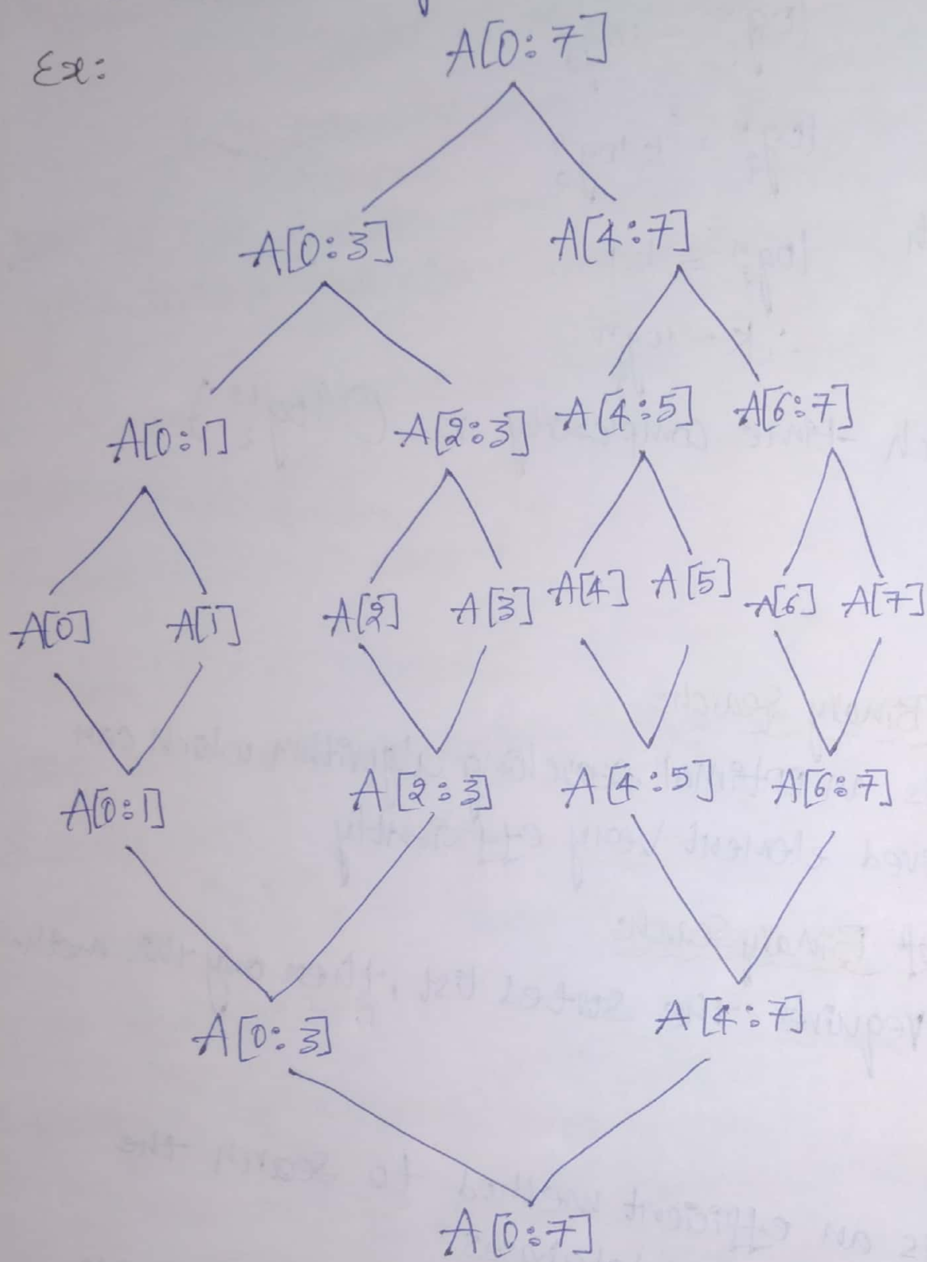
Merge sort consists of 3 steps:

1. Divide: partition the array into 2 sublists
2. Conquer: sort the each sublist

Scanned with CamScanner

3. Combine: merge solutions of sublist into a single list

Ex:



1) A =

E	X	A	M	P	L	E

A[0:6]

$$Mid = \frac{0+6}{2} = 3$$

a[1:10]: {310, 285, 179, 652, 351, 423, 861, 254, 450, 520}

| E | X | A | M | P | L | E |

| E | X | A | M | | P | L | E |

| E | X | | A | M | P | L | E |

E   X   A   M   P   L   E

| E | X | | A | M | | E | L |

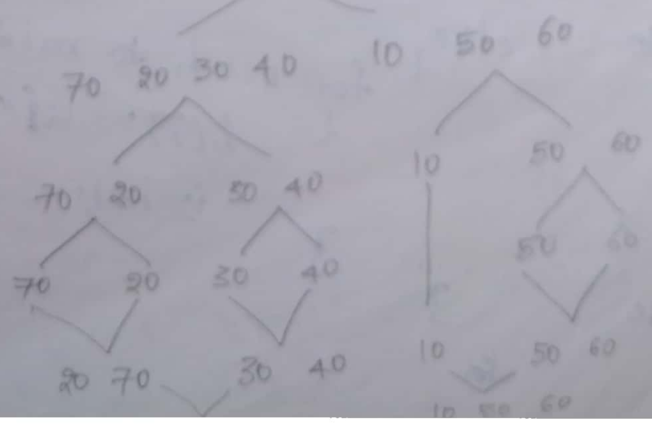
| A | E | M | X | | E | L | P |

| A | E | E | L | M | P | X |

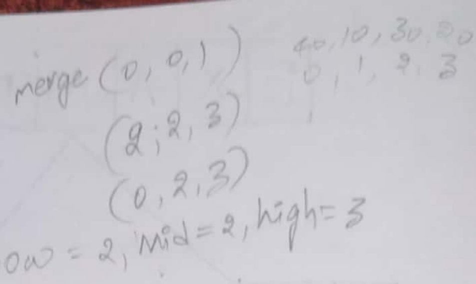
2) 70, 20, 30, 40, 10, 50, 60

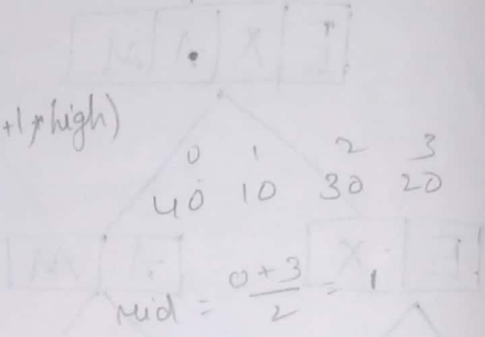
A[0:6]

$Mid = \frac{0+6}{2} = 3$

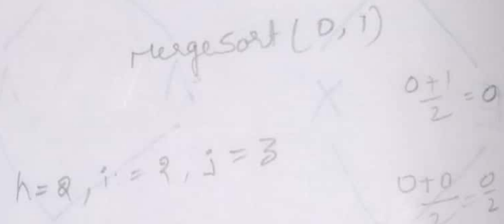


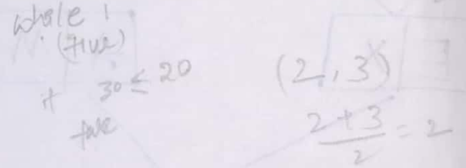
Algorithm:



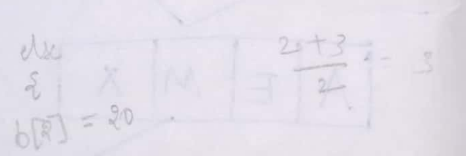


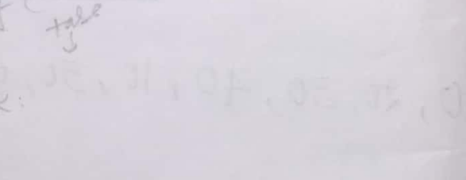
```
Algorithm mergesort (low, high)
{
if (low < high) then
  {
  mid = [(low + high) / 2];
  mergesort (low, mid); -> mergesort (mid+1, high)
  merge (low, mid, high);
  }
}
```



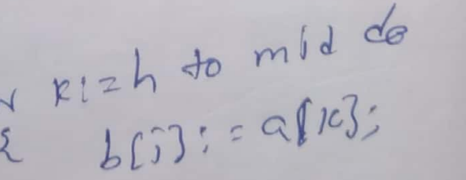


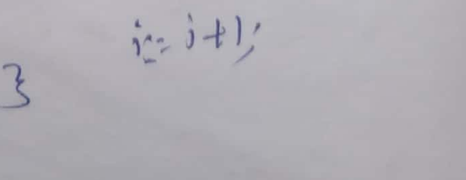
```
Algorithm merge (low, mid, high)
  {
h = low, i := low, j := mid+1;
while((h <= mid) and (j <= high)) do
{
  if (a[h] <= a[j]) then
    {
     b[i] := a[h];
      h := h+1;
    }
  else
    {
     b[i] := a[j];
      j := j+1;
    }
    i := i+1;
}
if (h > mid) then
  {
for k := j to high do
    {
      b[i] := a[k]
      i := i+1;
    }
  }
for k := low to high do
      a[k] := b[k];
  }
```

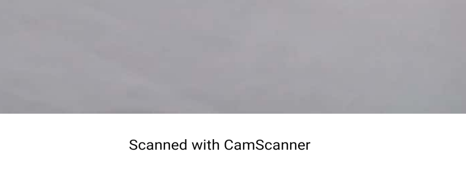




```
else {  for k := h to mid do
        {  b[i] := a[k];
           i := i+1;
        }
     }
```







Scanned with CamScanner

Analysis:-

In merge sort algorithm 2 recursive calls are made. Each Recursive call focusses on 'n/2' elements of the list and 1 call is made to combine two sublists i.e to merge all 'n' elements. Hence the recurrance relation is

$C(n) = C(n/2) + C(n/2) + Cn$ for $n>1$

$= 2C(n/2) + Cn$

$C(1) = 0$ for $n=0$

By using Master's theorem

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

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

$n^{\log_b^a} = n^{\log_2^2} = n^1 = n$

Here case (ii) condition is satisfied

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

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

$n = n$ ✓

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

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

$= \theta(n \log n)$

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

Using substitution method, let the recurrance relation be

$C(n) = C(n/2) + C(n/2) + C_n$ for $n>1$ → (1)

$T(1) = 0$ for $n=0$ → (2)

$= 2C(n/2) + C_n$ → (3)

Apply Backward substitution method

Assume $n = 2^k$

$$C(2^k) = 2\,C\left(\frac{2^k}{2}\right) + C\,2^k.$$

$$= C(2^{k-1}) + C\,2^k \rightarrow (4)$$

Let $k = k-1$

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

$$= 2C(2^{k-2}) + C\,2^{k-1} \rightarrow (5)$$

Substitute eq (5) in eq (4)

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

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

$$= 2^2\,C(2^{k-2}) + 2\,C\,\frac{2^k}{2} + C\,2^k$$

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

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

We can write

$$C(2^k) = 2^3\,C(2^{k-3}) + 3\,C\,2^k$$

$$= 2^4\,C(2^{k-4}) + 4C\,2^k$$

$$\vdots \qquad \vdots$$

$$C(2^k) = 2^k C(2^{k-k}) + kC\,2^k$$

$$= 2^k C(2^0) + kC\,2^k$$

$$= 2^k C(1) + kC\,2^k$$

$$= 2^k \cdot 0 + kC\,2^k$$

constant

$$= kC\,2^k$$

$$C(n) = \log n \cdot n$$

$$\therefore C(n) = n \log n$$

Scanned with CamScanner

$n = 2^k$

$\log_2 n = \log_2 2^k$

$\log_2 n = k \log_2 2$

$\log_2 n = k \cdot 1$

$\therefore \boxed{k = \log_2 n}$

Quick sort :-

This technique was invented by Hoare and he is considered that this method to be a fast method to sort the elements. Here the division into 2 subarrays is made so that the sorted subarrays do not need to be merge later. This is accomplished by rearranging the elements in an array ~~A1 [A1 - - An]~~ A [1-n]

In this method, the list is divided into 2 based on the pivot element. Usually the first element is considered as pivot element now move the pivot into its correct position in the list. The elements to the left of pivot are less than the pivot and the elements to the right of the pivot are greater than the pivot

$i < j$: swap i^{th} element & j^{th} element

$i > j$: swap pivot element & j^{th} element

Scanned with CamScanner

Eg:

pivot	2	3	4	5	6	7	8	9	10	i	j	
(65)	70 (i)	75	80	85	60	55	50	45 (j)	+∞	2	9	2<9
65	45	75 (i)	80	85	60	55	50 (j)	70	+∞	3	8	3<8
65	45	50	80 (i)	85	60	55 (j)	75	70	+∞	4	7	4<7
65	45	50	55	85 (i)	60 (j)	80	75	70	+∞	5	6	5<6
65	45	50	55	60 (j)	85 (i)	80	75	70	+∞	6	5	6>5

[60 45 50 55] [65] [85 80 75 70 +∞]

left array sublist — right array sublist

2) 65, 45, 50, ~~65~~, ~~45~~, 80, 48, 78, 63, 90

pivot	2	3	4	5	6	7	8	9	i	j	
(65)	45	50	80 (i)	48	78	63 (j)	90	+∞	4	7	4<7
65	45	50	63	48 (j)	78 (i)	80	90	+∞	6	5	6>5

48 45 50 63 [65] 78 80 90 +∞

pivot element

Algorithm Quick sort (left, right)

```
{
  ~~if(p ≠ left)~~ p := left, q = right+1;
  if (p<q) then
  {
    j = Partition(a, p, q);
    Quicksort (p, j-1);
    Quicksort (j+1, q);
  }
}
```

Algorithm Partition(a, left, right);

85, 80, 75, 70, 72

$P = a[1]$

$P = 85$, $i = 1$, $j = 5$

$i = i+1 = 1+1 = 2$

$a[i] \geq P$

$a[i] \geq$

```
{
 P = a[left], i := left, j := right;
 repeat
 {
  repeat
   i := i+1;
  until (a[i] >= P);
  repeat
   j := j-1;
  until (a[j] <= P);
  if (i<j) then
  Interchange (a,i,j);
 } until (i >= j)
  a[left] := a[j]
   a[j] := P;
   return j;
 }
Algorithm Interchange (a,i,j)
 {
 t := a[i].
  a[i] := a[j];
  a[j] := t;
 }
```

```
repeat
   i := i+1;
until (a[i] >= P);
repeat
   j := j-1;
until (a[j] <= P);
```

1) 60, 45, 50, 55 (positions 1, 2, 3, 4)

60 ↓ left, 55 ↓ right

$P = a[left] = a[1] = 60$

$i = left = 1$

$j = right+1 = 4+1$

$j = 5$

$i = i+1$

$i = 1+1$

$i = 2$

$a[i] \geq P$

$a[2] \geq 60$

$45 \geq 60$ ✗

$a[3] \geq P$

$50 \geq 60$ ✗

$a[4] \geq P$

$55 \geq 60$ ✗

$j = j-1 \longrightarrow a[j] \leq P$

$j = 5-1$

$j = 4$

$a[4] \leq 60$

$55 \leq 60$ ✓ $\rightarrow$ if $i < j$

$5 < 4$ ✗

$i \geq j$

$5 \geq 4$ ✓

$a[1] = a[4]$

$a[1] = 55$

$a[4] = 60$

Analysis:

If the array is always partitioned at the mid then it brings the best case efficiency of an algorithm. The recurrance relation of Quick Sort for obtaining best case is

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

$$T(1) = 0$$

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

Using Masters theorem

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

$$T(1) = 0 \longrightarrow (2)$$

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

$\Rightarrow a = 2, b = 2, f(n) = n$

$n^{\log_b^a} = n^{\log_2^2} = n^1 = n$

$\therefore \boxed{n^{\log_b^a} = n}$

$f(n) = n^{\log_b^a} \Rightarrow \boxed{n = n}$

Case ii is applied

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

$= \Theta(n \log n)$

By using Big oh Notation $T(n) = O(n \log n)$

Substitution Method:-

We can obtain the best case time complexity of Quicksort using backward substitution method

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

$T(1) = 0$ for $n = 0$ → (2)

$T(n) = 2T(n/2) + Tn$ → (3)

Apply backward substitution method

Assume $n = 2^k$

$T(2^k) = 2T\left(\frac{2^k}{2}\right) + T2^k$

$= T(2^{k-1}) + T2^k$ → (4)

Let $k = k-1$

$T(2^{k-1}) = 2T(2^{k-1-1}) + Tc2^{k-1}$

$= 2T\left(\frac{k-2}{2}\right) + Tc^{2k-1}$ → (5)

substitute eq (5) in eq (4)

$T(2^k) = 2[2T(2^{k-2}) + T2^{k-1}] + T2^k$

$= 2^2 T(2^{k-2}) + 2T(2^{k-1}) + T2^k$

$= 2^2 T(2^{k-2}) + 2T\frac{2^k}{2} + T2^k$

$= 2^2 T(2^{k-2}) + T2^k + T2^k$

$= 2^2 T(2^{k-2}) + 2T2^k$

We can write

$T(2^k) = 2^3 T(2^{k-3}) + 3T2^k$

$= 2^4 T(2^{k-3}) + 4T2^k$

⋮ ⋮

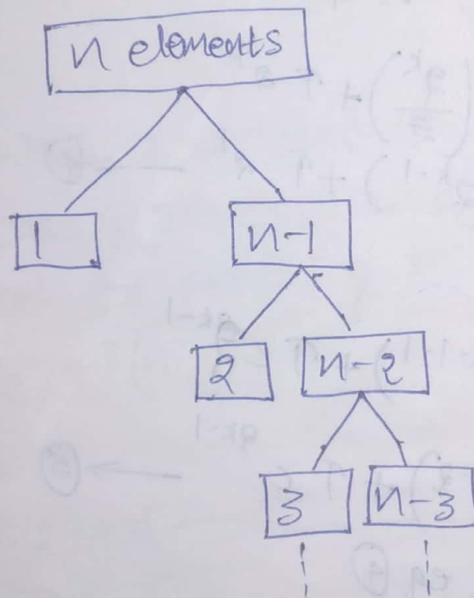
Scanned with CamScanner

$$T(2^k) = 2^k T(2^{k-k}) + kT2^k$$
$$= 2^k T(2^0) + kT2^k$$
$$= 2^k T(1) + kT2^k$$
$$= 0 + kT2^k$$
$$= kT2^k$$

Similar to merge sort

$$T(n) = \log n . n$$
$$\therefore \boxed{T(n) = \log n . n}$$

The worst case time complexity for quick sort occurs when the pivot element is minimum or maximum element of all the elements in the list. This can be graphically represented as



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

We know that $1 + 2 + 3 + \ldots + (n-1) + n = \frac{n(n+1)}{2}$

$$T(n) = 1 + 2 + 3 + \ldots + n$$
$$= \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$
$$T(n) = n^2$$

By using Bigoh notation $T(n) = O(n^2)$

~~Selection Sort~~

Applications of Divide and Conquer:

Binary search, Merge sort, Quick sort, selection sort, Stressen's matrix multiplication, finding maximum and minimum elements

Selection sort:

In this problem, we have 'n' elements A[1-n] and are required to determine the kth smallest element if the partitioning element 'v' is positioned at A[j] then 'j-1' elements are less than or equal to A[j] and 'j+1' elements are greater than the A[j]. If k<j then the kth smallest element is in A[1:j-1]. If k=j then A[j] is the kth smallest element if k>j then k^{th} smallest element is in A[j+1:n]. The select function places the kth smallest element into positions A[k] and partitions the remaining elements so that A[i] ≤ A[k], 1 ≤ i < k and A[i] ≥ A[k], k < i ≤ n

Algorithm:

```
Algorithm Select1(a,n,k)
{
  low:= 1, up:=n+1;              low=1, up=6+1
    a[n+1]:= ∞                          up=7
  repeat
  {
   j:= partition(a, low, up);
             (→smallest element)
    if(k=j) then
    return;                          1<18
  else if (k<j) then                 up:=18
       up:=j
  else
```

```
low := j+1;
} until (false);
}
                              low   18  up
Algorithm partition (a, left, right);
  { P = a [low]; i := low , j := high;
repeat     P = 18        i = 1,  j = 7
 {
  repeat
     i := i+1;          i = 2
 until (a[i] ≥ P);   7 ≥ 18
 repeat
     j := j-1;          j = 6
 until (a [j] ≤ P);    1 ≤ 18
 if (i < j) then        1 < 7
Interchange (a, i, j);           1, 7, 15, 5, 3, 18
 } until (i ≥ j)
   a[left] := a[j];
   a [j] := P;                          pg 190 → 3.21 algorithm
                                        pg. 187 ×
   return j;
}
Algorithm Interchange (a, i, j)
   {
  t := a[i];
  a[i] := a [j];
  a [j] := t;
  }
repeat
     i := i+1;
until (a[i] ≥ P);
 repeat
     j := j-1;
 until (a[j] ≤ P);
```

① 18, 7, 15, 5, 3, 1

low := 1, up := 6+1

up := 7

a[7] = ∞

Analysis:

The worst case time complexity of selection sort is $O(n^2)$ when the input a[1:n] is such that the partitioning element on the ith call to partition is the ith smallest element and k=n. In this case 'm' increases by 1 following each call to partition and 'j' remains unchanged.

Therefore the recurrance relation is

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

$$\therefore T(n) = O(n^2)$$ → worst case

After applying divide conquer → time complexity changes to $O(n)$ for best & average case

Average time complexity:

The average time complexity

On the first call to partition, the partition element with probability $1/n$, 'v' is the ith smallest element $1 \le i \le n$.

The time required by partition and 'if' statement is $O(n)$. Hence there is a constant 'c', $c > 0$ such that the rec $T(n) \le c_n + \frac{1}{n} \max \left[\sum_{1 \le i < k} T^{k-1}(n-1) + \sum_{k < i \le n} T^k(i-1) \right], n \ge 2$

So $R(n) \le c_n + \frac{1}{n} \max \left\{ \sum_{i \le i \le k} R(n-1) + \sum_{k < i \le n} R(i-1) \right\}$

$$R(n) \le c_n + \frac{1}{n} \max \left\{ \sum_{1 \le i \le k} R(i) + \sum_{k < i \le n} R(i) \right\}, n \ge 2 \rightarrow ①$$

Scanned with CamScanner

$R(1) \leq C$

$R(n) \leq 4C_n$ by Induction on n

for $n=2, i=1$

$R(n) \leq 2C + \frac{1}{2} \max R(1), R(1)$

$\leq$ ~~2.5C~~ $2.5C$

$\therefore R(n) \leq 2C \leq 4C_n$

Assume $R(n) \leq 4C_n$ for all n,

$2 \leq n < m$

Similarly for $n=m$

$$R(m) \leq Cm + \frac{1}{m} \max \left\{ \sum_{m-k+1}^{m-1} R(i) + \sum_{k}^{m-1} R(i) \right\}$$

Since we know that $R(n)$ is a non-decreasing function of 'n', it follows that $\sum_{m-k+1}^{m-1} R(i) + \sum_{k}^{m-1} R(i)$ is

Maximised if $k = \frac{m}{2}$ when 'm' is even and $k = \frac{m+1}{2}$ when 'm' is odd

If m is even $R(m) \leq Cm + \frac{2}{m} \sum_{m/2}^{m-1} R(i)$

$$\leq Cm + \frac{8C}{m} \sum_{m/2}^{m-1} R(i)$$

$$< 4Cm$$

If m is odd

$$R(m) \leq Cm + \frac{2}{m} \sum_{\left(\frac{m+1}{2}\right)}^{m-1} R(i)$$

$$\leq Cm + \frac{8C}{m} \sum_{\left(\frac{m+1}{2}\right)}^{m-1} R(i)$$

$$\leq 4Cm$$

Scanned with CamScanner

Since $T(n) \leq R(n)$

It follows that $T(n) \leq 4cn$

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

Stressen's Matrix Multiplication:-

To multiply '2' matrices A and B each of size 'n'

$C = A * B$

suppose the matrix size is 2x2 then

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$

$C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$

$C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$

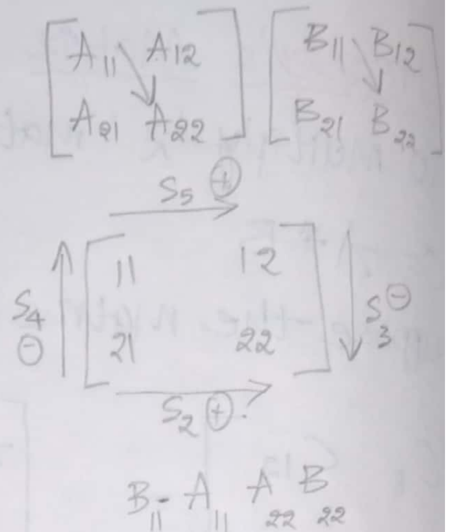
$C_{22} = A_{21} * B_{21} + A_{22} * B_{22}$

In this procedure, to accomplish 2x2 matrix multiplication we require 8 multiplications and 4 additions. We can write the following algorithm

```
Algorithm mul(A, B, C, n)
{
for i:=1 to n do
for j:=1 to n do
    C[i,j]:=0;
 for k:=1 to n do
   C[i,j] = C[i,j] + A[i,k]* B[j,k]
   }
```

The time Complexity of the above algorithm is $O(n^3)$.

Stressen showed that 2x2 Matrix multiplication accomplished by using 7 multiplications and 18 subractions (or) additions



$S_1 = (A_{11} + A_{22}) * (B_{11} + B_{22})$

$S_2 = (A_{21} + A_{22}) * B_{11}$

$S_3 = A_{11} * (B_{12} - B_{22})$

$S_4 = A_{22} * (B_{21} - B_{11})$

$S_5 = (A_{11} + A_{12}) * B_{22}$

$S_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$

$S_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$

change A to B

$C_{11} = S_1 + S_4 - S_5 + S_7$ P+S−T+V

$C_{12} = S_3 + S_5$ R+T

$C_{21} = S_2 + S_4$ Q+S

$C_{22} = S_1 + S_3 - S_2 + S_6$ P+R−Q+U

The divide & Conquer approach can be used for implementing

i) divide:

Divide the matrices into Submatrices

ii) Conquer

Use a group of matrix multiplication equations

iii) Combine

Recursively multiply submatrices and get the final result of multiplication

Algorithm:

```
Algorithm stmul(A,B,C,n)
{
if (n==1) then
```

```
{
C = C + (A*B)
}
else
{
stmul (A, B, C, n/4);
stmul (A, B+n/4, C+n/4);
stmul (A+2*n/4, B, C+2*n/4, n/4);
stmul (A+2*n/4, B+n/4, C+3*n/4, n/4);
stmul (A+n/4, B+2*n/4, C, n/4);
stmul (A+n/4, B+3*(n/4), C+n/4, n/4);
stmul (A+3*(n/4), B+2*(n/4), C+2*(n/4), n/4);
stmul (A+3*(n/4), B+3*(n/4), C+3*(n/4), n/4);
}
}
```

Analysis:-

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

$T(1) = 1$

Assume $n = 2^k$

$T(2^k) = 7T\left(\frac{2^k}{2}\right)$

$= 7T(2^{k-1}) \longrightarrow (1)$

Let $k = k-1$ in (1)

$T(2^{k-1}) = 7.T(2^{k-1-1})$

$= 7.T(2^{k-2}) \longrightarrow (2)$

Substitute eq(2) in eq(1)

$T(2^k) = 7[7T(2^{k-2})]$

$= 7^2 T(2^{k-2})$

Let $k = k-2$

$T(2^k) = 7^3 T(2^{k-3})$

$\vdots$

$T(2^k) = 7^k T(2^{k-k})$

$= 7^k . T(2^0)$

$= 7^k T(1)$

$= 7^k (1)$

$= 7^k$

$= \log_2^n$

$= n^{\log_2^7}$

$= n^{2.81}$

$T(n) = n^{2.81}$

$T(n) = O(n^{2.81})$

~~Substitution~~ Masters Method:

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

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

$a = 7, b = 2, F(n) = 0$

$n^{\log_b^a} = n^{\log_2^7} = n^{2.81}$

$F(n) = 0$

$n^{\log_b^a} = n^{2.81}$

$F(n) < n^{\log_b^a}$

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

$= \theta(n^{2.81})$

By using Big oh notation $T(n) = O(n^{2.81})$

Scanned with CamScanner

Finding Maximum and Minimum :-

Here the problem is to find the maximum and minimum items in a set of 'n' elements. This can be solved by using the divide and conquer method (or) straight forward algorithm.

Straight Forward Algorithm:

```
Algorithm straightminmax(a,n, max,min)
{
  max:= min = a[1];
for i:=2 to n do
{                                  8>7
if (a[i] > max) then            → n-1   } 2*(n-1)
   max:= a[i];                   max=8
 if (a[i] < min) then  8<7       → n-1
   min:= a[i];
 }
}
```

write after this algorithm

Modified Straight Forward Algorithm:

```
Algorithm straightminmax(a,n, max, min)
 { for i=2 to n do {
  max:= min= a[1];
  if (a[i] > max ) then        8>7
                               max=8
     max:= a[i];
  else if (a[i] < min) then
       min:= a[i];    } }
```

7, 8, 5, 4, 9, 6, 11, 12

{
i = 2 to 6 do
max = min = 7
i = 3
5 > 8
}

Here the no. of the comparisons are 2(n-1) and the time complexity is O(2(n-1)) so the above algorithm is modified. Now the comparisons required for modification method is (n-1) and the time complexity is O(n-1)

When the elements are in increasing order then the best case time complexity and the no. of comparisons required are n-1

When the elements are in decreasing order then we get the worst case time complexity and no. of comparisons required are $2(n-1)$.

Then the algorithm is Modified, then we require '$n-1$' element comparisons and the time complexity is $O(n-1)$ in best, average and worst cases.

Divide and Conquer method:

The number of Comparisons for finding maximum & Minimum elements can be reduced by using divide and Conquer strategy. Here the list is divided into 2 Sublists A_1, A_2 where $A_1 = (A[1] ---- A[n/2])$

$A_2 = (A[n/2+1] --- A[n])$

The two sublists are solved separately by calling Divide and Conquer method recursively. Finally $Max[A] = (Max(A_1), max(A_2))$

$Min(A) = min(min(A_1), min(A_2))$

Algorithm:

```
Algorithm MaxMin (first, last, Max, Min)
  {
  if (first = = last)
    {
   Max:= A[first];
   Min: = A[first];
    }
 else if (first+1 = = last)
   {
  if (A [first] < A[last]) then
   {
   Max:= A [last];
   Min: = A [first];
```

```
    }
    else
    {
    Max: = A[first];
    Min: = A[last];
    }
}
else
{
Mid = (first + last) / 2;
MaxMin (first, mid, Max, Min);          -> left sublist   (divides into left & right)
MaxMin (Mid+1, last, tMax, tMin);       -> right sublist
if (Max < tMax)                         -> to combine the elements.
  {
  Max: = tmax;
  }
if (Min > tMin)
  {
  Min: = tMin;
  }
}
}
```

① 53, 43, −3, −8, 48, 92, 68, 25, 75 (positions 1–9)

$Mid = \frac{1+9}{2} = 5$

53, 43, −3, −8, 48 (positions 1–5)

$Mid = \frac{1+5}{2} = 3$

92, 68, 25, 75 (positions 1–4)

$Mid = \frac{1+4}{2} = 2.5 = 3$

53, 43, −3 (positions 1–3) $Mid = \frac{1+3}{2} = 2$

−8, 48 (positions 1–2) $Mid = \frac{1+2}{2} = 1.5$

92, 68, 25 (positions 1–3) Mid = 2

75 (position 1)

53 43 → Max = 53

−3 → Min = −3

−8 48 → Min = −8, Max = 48

92 68 → Max = 92

25 → Min = 25

75 → Max = 75, Min = 75

Max = 92, Min = 25

Analysis: write this method if asked even substitution or Master.

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

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

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

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

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

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

$\vdots$

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

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

Put $\boxed{n = 2^k}$ we get

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

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

$= 2^{k-1}(1) + 2^k - 2 \quad [\because T(2) = 1]$

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

$T(n) = \frac{3n}{2} - 2$

$\therefore \boxed{T(n) = \frac{3n}{2} - 2}$

$\frac{a(r^{n-1}-1)}{r-1}$

optimal- best or most favourable optimum.

optimum- most conductive to a favourable outcome, best

feasible- Possible to do easily or conveniently (practical)

Unit-2

Greedy Method

General method:

The Greedy method is a straight forward and powerful technique to design algorithms. It is popular to obtain optimal solution, most of these problems have 'n' inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution.

A feasible solution which maximises and minimises the objective function. So when the required solution is obtained it is called as optimal solution. The Greedy method divides an algorithm that works in stages, considering one input at a time, at each stage a decision is made regarding whether a particular input is in an optimal solution. This is done by some selection procedure. The feasible solutions are added to the partially constructed optimal solution. If the input is infeasible then that will not be added to the partial solution. The selection process itself is based on some optimisation measure. This measure may be the objective function. This kind of selection process is called selection paradigm

Control abstraction:

```
Algorithm Greedy(a,n)
{
 Solution: = 0;
 for i:=1 to n do          → to n i/p's
 {
   x: = select(a);         → select the feasible solution
   if (feasible(solution, x)) then
   Solution: = union(solution, x);   → feasible, then it is added to the set
```

}
return solution;
}

Applications of Greedy method:

1) knapsack problem
2) Job Scheduling with deadlines
3) Minimum cost Spanning tree
4) Optimal storage on tapes
5) Single source shortest path problem

Advantages:

→ It is easy to write the algorithm
→ It is easy to write the code
→ It produces efficient results

Disadvantage:

It doesn't give guarantee for the optimal solution

Knapsack Problem:-

In this method 'n' objects are given and a knapsack (or) empty bag are given. object 'i' has a weight W_i and the knapsack has the capacity 'm'. Now the knapsack problem is that we should place the objects into the bag without exceeding the bag capacity.

If a fraction x_i, $0 \leq x_i \leq 1$, of object 'i' is placed into the knapsack then a profit of '$P_i x_i$' is earned.

The problem can be stated as Maximise $\sum_{1 \leq i \leq n} P_i x_i$

such that $\sum_{1 \leq i \leq n} W_i x_i$ such that $1 \leq i \leq n$. The profits and weights are +ve numbers.

Example: Consider the following instance of knapsack problem $n=3$, $m=20$

$$(P_1, P_2, P_3) = (25, 24, 15)$$

$$(W_1, W_2, W_3) = (18, 15, 10)$$

Case (i): Maximum Profit

Case (ii): Minimum weight

Case (iii): Maximum profit per unit weight

Case (iv): Maximum weight

Sol:

Case (i): Maximum Profit

Here we need to place an item in the bag whose profit is maximum

$$x_1 = 1, \ x_2 = 2/15, \ x_3 = 0$$

$$\Rightarrow \sum P_i x_i = P_1 x_1 + P_2 x_2 + P_3 x_3$$

$$= 25(1) + 24(2/15) + 15(0)$$

$$= 28.2$$

Case (ii): Minimum weight:

We place an item in the bag whose weight is minimum

$$x_1 = 0, \ x_2 = 10/15, \ x_3 = 1$$

$$\sum P_i x_i \Rightarrow P_1 x_1 + P_2 x_2 + P_3 x_3$$

$$= 25(0) + 24(10/20) + 15(1)$$

$$= 0 + 12 + 15$$

$$= 31$$

Case (iii): Maximum profit per unit weight

We place an item in the bag whose P/W ratio is maximum

$$\frac{P_1}{W_1} = \frac{25}{18} = 1.4 \qquad \frac{P_2}{W_2} = \frac{24}{15} = 1.6 \qquad \frac{P_3}{W_3} = \frac{15}{10} = 1.5$$

$$x_1 = 0, \ x_2 = 1, \ x_3 = 5/10 = 1/2$$

Scanned with CamScanner

$\sum P_i X_i \Rightarrow P_1 X_1 + P_2 X_2 + P_3 X_3$

$= 25(0) + 24(1) + 15(1/2)$

$= 24 + 7.5$

$= 31.5$

Case(iv): Maximum weight

$X_1 = 1;\ X_2 = 2/15;\ X_3 = 0$

$\sum W_i X_i = W_1 X_1 + W_2 X_2 + W_3 X_3$

$= 18(1) + 15(2/15) + 10(0)$

$= 18 + 2$

$= 20$

Algorithm:

```
Algorithm Greedyknapsack(m,n)
{
for i:=1 to n do
    x[i]:=0.0;
    U:=m;
for i:=1 to n do
 {
 if (w[i]>U) then
     break;
    x[i]:=1.0;
    U:=U-w[i];
 }
if (i<=n) then
   x[i]:=U/w[i];
}
```

① Ex

$n=7$

$m=15$

$\{P_1, P_2 \ldots P_7\} = \{10, 5, 15, 7, 6, 18, 3\}$

$\{W_1, W_2 \ldots W_7\} = \{2, 3, 5, 7, 1, 4, 1\}$

Initia

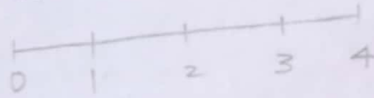
Scanned with CamScanner

Job scheduling with deadlines:

Given a set of 'n' jobs. Job 'i' is associated with an integer deadline $d_i \geq 0$ and a profit $P_i > 0$

For any job 'i' the profit earned if and only if the job is completed by its deadline. To complete a job one has to process the job on a machine for '1' unit of time. Only one machine is available for processing the jobs.

A feasible solution for this problem is a subset 'j' of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution 'j' is sum of the profits of the job 'j', And optimal solution with maximum value.



Ex:

n	P_i	d_i
1	70	2
2	12	1
3	18	2
4	35	1

(1,3)
(2,1)
(2,3)
(3,1)
(4,1)
(4,3)

n	P_i
(1,3)	88
(2,1)	82
(2,3)	30
(3,1)	88
(4,1)	105
(4,3)	53

Example

J1	J2	J3	J4	J5	J6
5	3	3	2	4	2
200	180	190	300	120	100

optimal → 990 units

J2, J4, J3, J5, J1

{4,1} is the optimal solution. Which produces maximum {4,1} profit

Find the optimal solution of the knapsack instances

$n = 7, M = 15$

$(P_1, P_2, P_3, P_4, P_5, P_6, P_7) = (10, 5, 15, 7, 6, 18, 3)$

$(W_1, W_2, W_3, W_4, W_5, W_6, W_7) = (2, 3, 5, 7, 1, 4, 1)$

Case(i): Maximum profit

$x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 4/7, x_5 = 0, x_6 = 1, x_7 = 0$

$\Sigma p_i x_i = 10(1) + 5(0) + 15(1) + 7(4/7) + 6(0) + 18(1) + 3(0)$

$= 10 + 15 + 4 + 18$

$= 47$

Case(ii): Minimum weight

$x_1 = 1, x_2 = 1, x_3 = \frac{4}{5}, x_4 = 0, x_5 = 1, x_6 = 1, x_7 = 1$

$\Sigma P_i x_i = 10(1) + 5(1) + 15(4/5) + 7(0) + 6(1) + 18(1) + 3(1)$

$= 10 + 5 + 12 + 6 + 18 + 3$

$= 54$

Case(iii): Maximum profit per unit weight:

$\frac{P_1}{W_1} = 5, \frac{P_2}{W_2} = 1.6, \frac{P_3}{W_3} = 3, \frac{P_4}{W_4} = 1, \frac{P_5}{W_5} = 6, \frac{P_6}{W_6} = 4.5, \frac{P_7}{W_7} = 3$

$x_1 = 1 \quad x_2 = \frac{1.5}{3} \quad x_3 = 1 \quad x_4 = 0 \quad x_5 = 1 \quad x_6 = 1 \quad x_7 = 1$

$\Sigma P_i x_i = 10(1) + 5(0.5) + 15(1) + 7(0) + 6(1) + 18(1) + 3(1)$

$= 10 + 2.5 + 15 + 0 + 6 + 18 + 3$

$= 54.5$

Case(iv): Maximum weight: $x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1, x_5 = 0, x_6 = 3/4, x_7 = 0$

$\Sigma P_i x_i = 10(0) + 5(0) + 15(1) + 7(1) + 6(0) + 18(3/4) + 3(0)$

$= 15 + 7 + 13.5$

$= 35.5$

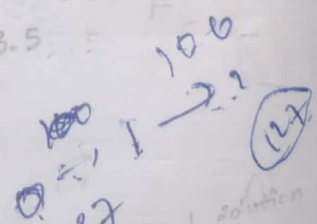
Ex: Job sequencing $n = 4$

i) $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$

$(D_1, D_2, D_3, D_4) = (2, 1, 2, 1)$

ii) $(P_1, P_2, P_3, P_4, P_5) = (20, 15, 10, 5, 1)$

$(D_1, D_2, D_3, D_4, D_5) = (2, 2, 1, 3, 3)$



i)

n	P_i	D_i
1	100	2
2	10	1
3	15	2
4	27	1

n	P_i
(1,3)	115
(2,1)	110
(2,3)	25
(3,1)	115
(4,1)	127
(4,3)	42

(4,1) is the optimal solution

ii)

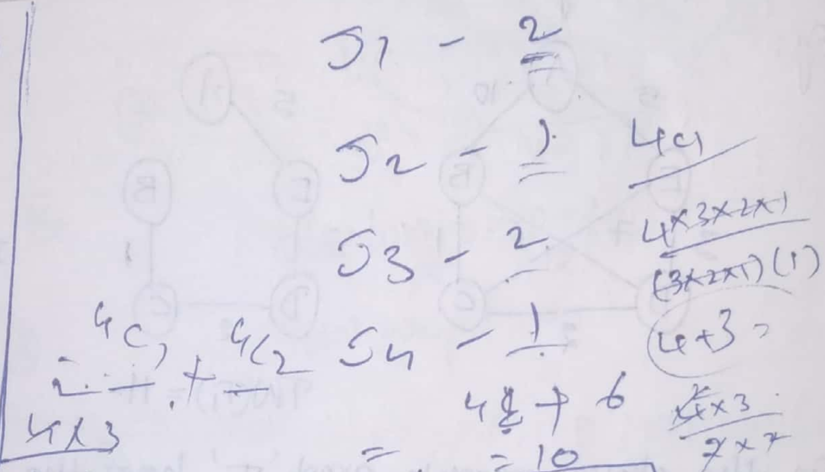
n	P_i	D_i
1	20	2
2	15	2
3	10	1
4	5	3
5	1	3

n	P_i
(1,2)	35
(1,4)	25
(1,5)	26
(2,1)	35
(2,4)	20
(2,5)	16
(3,4)	15
(3,5)	16
(4,1)	25
(4,2)	20
(4,5)	6
(5,1)	21
(5,4)	6
[illegible]	16

Scanned with CamScanner

Algorithm:

```
Algorithm Greedyjob(d,j,n)
{
j:={1}
for i:=2 to n do
{
if((all jobs in JU{i} can be completed by their deadlines)
then
j:=JU{i};  }
}
```



```
Algorithm JS(d,j,n)
{
d[0]:=J[0]:=0;
  J[1]:=1;
  k:=1;
for i:=2 to n do
{
  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
{
 for q:=k to (r+1) step -1 do
   J[q+1]:= J[q];
   J[r+1]:=i;
     k:=k+1;
  }
}
  return k;
}
```

$4 + 6$

$= \frac{4!}{3!\times1!} + \frac{4!}{2!\times2!}$

$nP_r = \frac{n!}{(n-r)!+r!}$

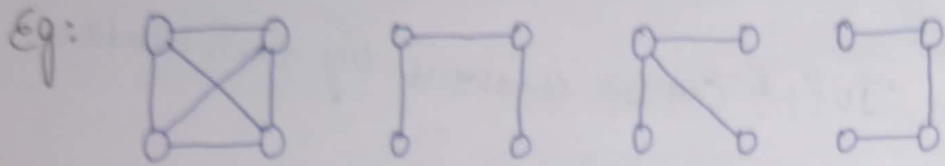
$= n_{c_1} = \frac{4!}{} + \frac{4!}{} = 4+12$

S.NO	Feasible sol	Process seq	value	Remarks

Minimum cost spanning tree:-

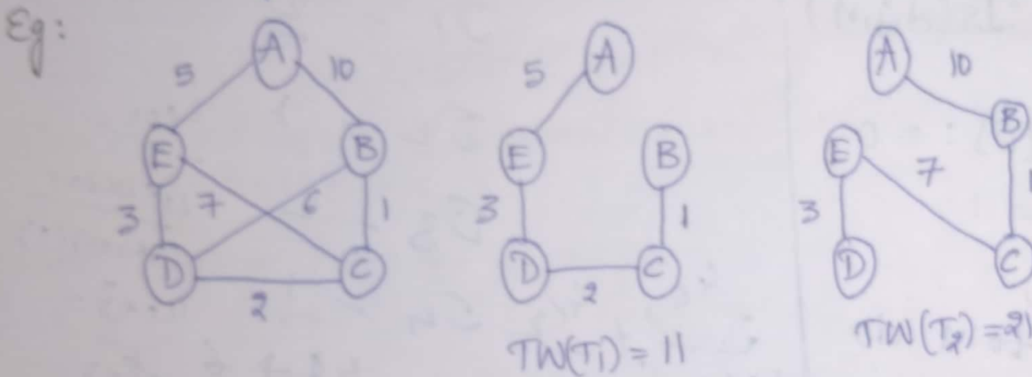
Let $G=(V,E)$ be an undirected connected graph. A subgraph $T=(V,E')$ of 'G' is a spanning tree if and only if 'T' is a tree.

Eg:



Minimum cost spanning trees of weighted connected graphs 'G' is a spanning tree with minimum (or) smallest weights.

Eg:



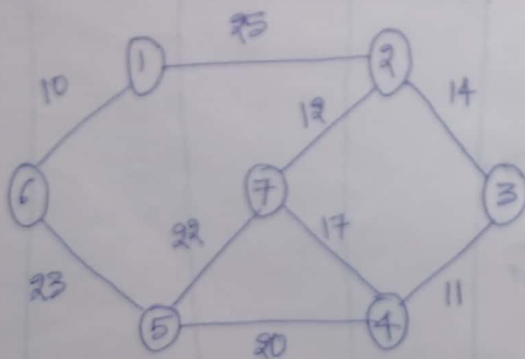
In the above example graph 'T_1' have the minimum weight. Therefore 'T_1' is considered as the minimum cost spanning tree.

Minimum cost spanning tree can be obtained by using the following two algorithms.

i) Prim's algorithm
ii) krushkal's algorithm

i) Prim's algorithm:

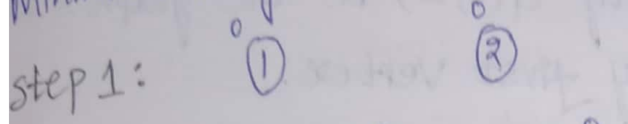
Consider the graph given below

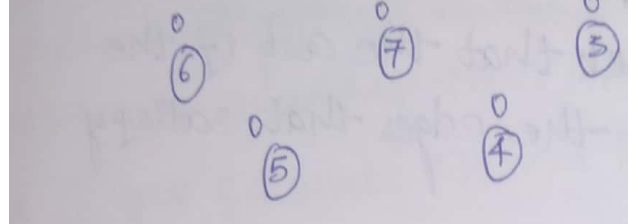


Now consider all the vertices as '0' then select one vertex as the first vertex, then select adjacent vertices of first vertex

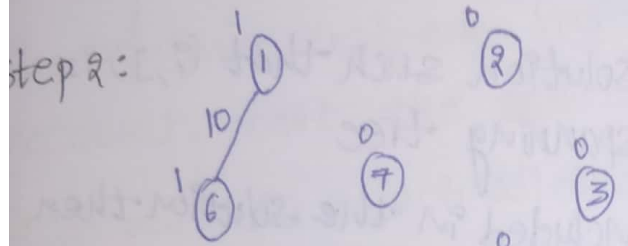
among the adjacent vertices, select one vertex with minimum edge and the algorithm proceeds by selecting adjacent edges with minimum weight, care should be taken for not forming circuit

step 1:

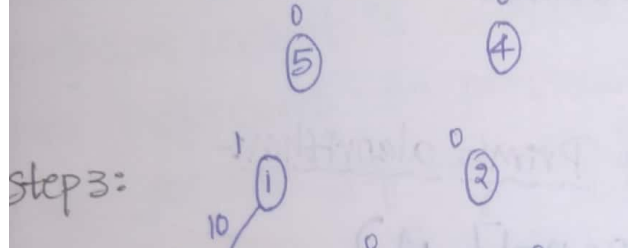




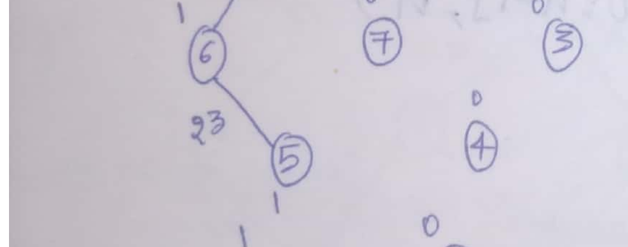
step 2:



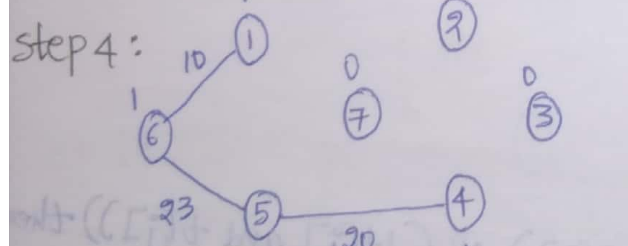
step 3:



step 4:

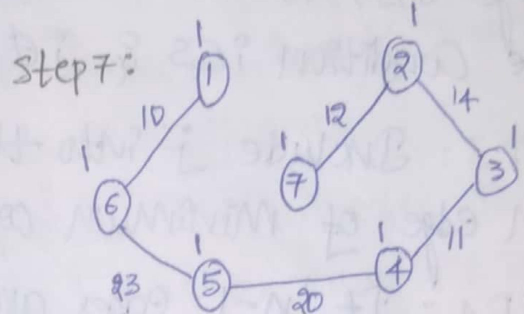


step 5:



step 6:

step 7:



Total weight = 10 + 23 + 20 + 11 + 14 + 12

Total weight = 90

Algorithm:

step1: Let 's' be the solution for minimum spanning tree which is initially empty $G(V,E)$ be the graph. Include any vertex i, say first vertex.

step2: Select a vertex 'j' such that the cost of the edge (i,j) is minimum along the edges that satisfy the condition $i \in S$ & $j \notin S$

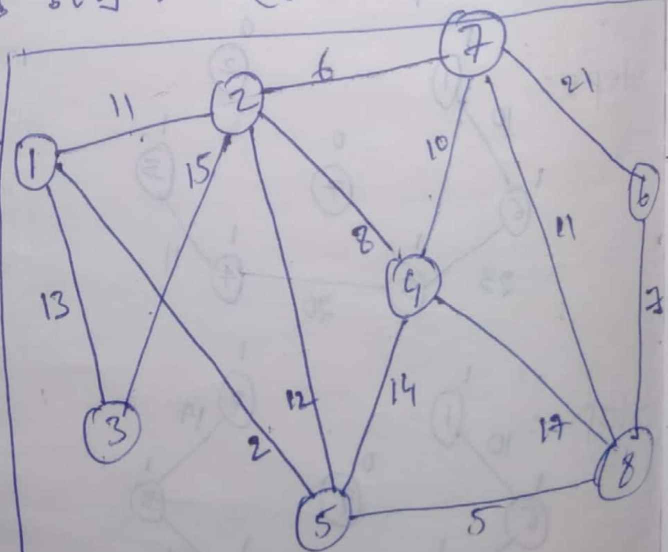
step3: Include 'j' into the solution such that (i,j) is an edge of minimum cost spanning tree.

step4: If 'n-1' edges are included in the solution then stop, else goto step2

Detailed Implementation of Prim's algorithm:-

```
Algorithm prim's (G[0:n-1,0:n-1], n)
for i <- 0 to n-1 do
              all the vertices init to '0'
   t[i] <- 0;   t[0] <- 1;
 mindist <- 0.0;   -> checking adjacent vertices
 for i <- 0 to n-1 do
   {
    for j <- 1 to n-1 do
   {
 if (G[i,j] and ((t[i] and !t[j]) or (!t[i] and t[j])) then
  {
 if (G[i,j] < mindist) then
   {
   mindist <- G[i,j];
     V1 <- i; V2 <- j;
    }
   }
  }
 write (V1, V2, mindist)
   t[V1] <- t[V2] <- 1;
```

any one should be in the solution not



total ← total + Mindist;

}

write("Total path cost", total);

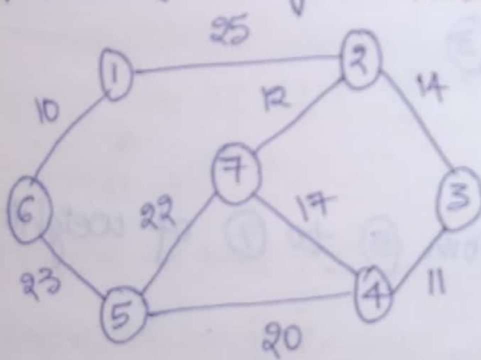
Kruskal's Algorithm:-

Kruskal's Algorithm is the another algorithm of obtaining minimum spanning tree. This algorithm was discovered by a II year graduate Joseph kruskals. In this algorithm always the minimum cost edge has to be selected but it is not necessary that the selected edge is adjacent.

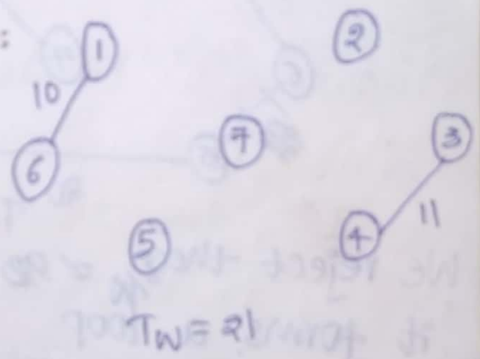
First we will select all the vertices then an edge with optimum weight is selected from heap even though it is not adjacent to previously selected edge. Care should be taken for not forming circuit.

Increasing Order: 10, 11, 12, 14, 17, 20, 22, 23, 25

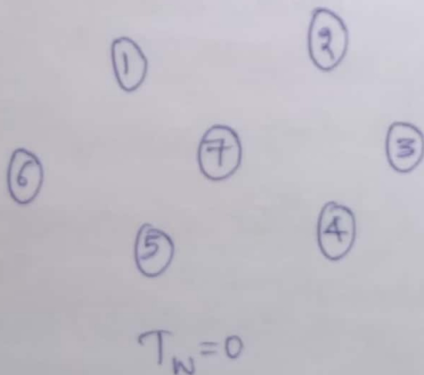
Eg:



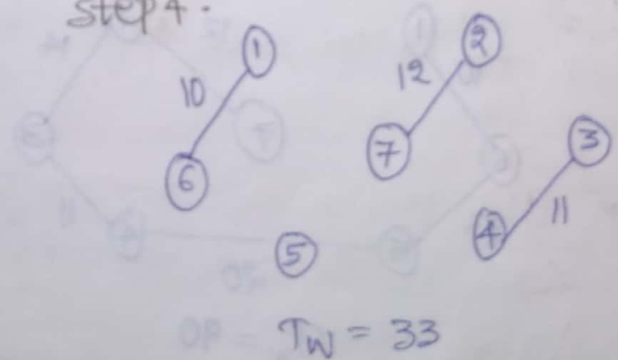
step3:



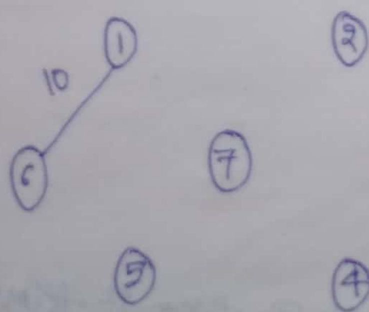
Step 1:



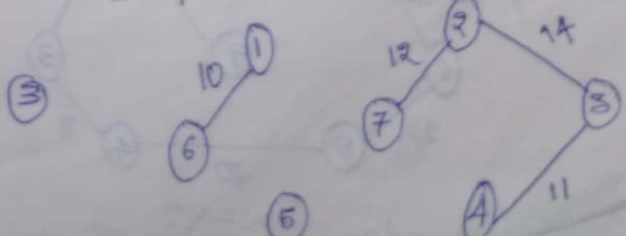
step4:



Step 2:

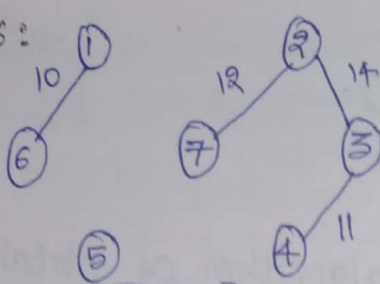


Step 5:



Scanned with CamScanner

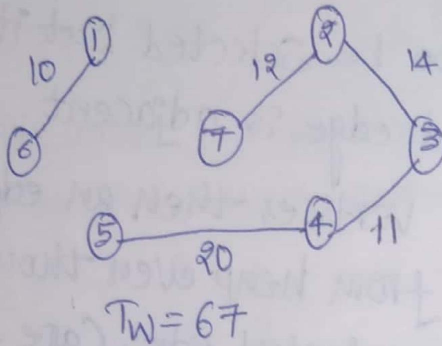
step 6:



$T_W = 47$

We reject the edge weight '17' i.e from ④ to ⑦ as it forms a loop

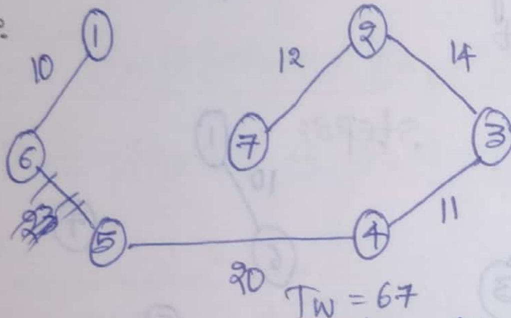
step 7:



$T_W = 67$

67
23
90

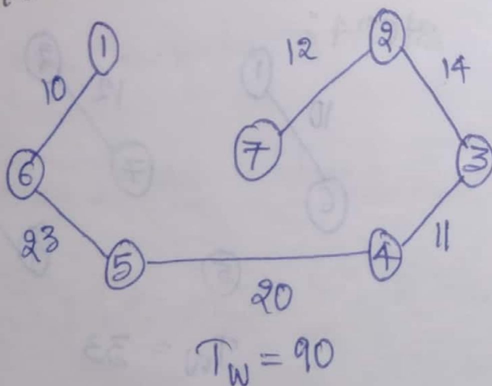
step 8:



$T_W = 67$

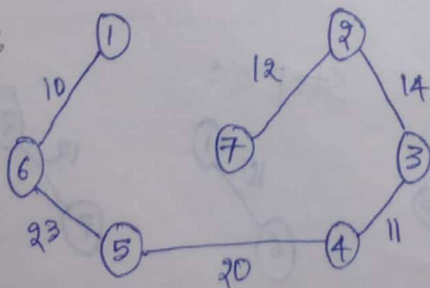
We reject the edge from ⑤ to ⑦ of weight '22' since it forms a loop.

step 9:



$T_W = 90$

step 10:



$T_W = 90$

Reject the edge from node ① to node ② since it forms a loop.

Scanned with CamScanner

Algorithm:

Step 1: Arrange all the edges in the increasing order of cost

Step 2: Include the edge with minimum cost in to the solution set.

Step 3: Add the next edge with minimum cost and delete edge from edge set until 'n-1' edges are added (or) the edge set become empty

Step 4: If the inclusion of any edge results in a cycle, reject it and move to the next edge.

Step 5: Goto step 3

Detailed Implementation of kruskal's Algorithm:-

```
Algorithm kruskals (E, cost, n, t)
{
Construct a heap out of the edge costs using heapify;
for i <- 1 to n do
      P[i] := -1;
i := 0 ; mincost := 0.0;
while((i < n-1) and (heap not empty)) do
{
Delete a minimum cost edge (u,v) from 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];
```

Scanned with CamScanner

```
    union[j,k];
  }
}
if (i ≠ n-1) then
write ("No spanning tree");
else
return mincost;
}
```

Optimal storage on Tapes:-

There are n- programs that are to be stored on a computer tape of length 'l' associated with each program 'i' is a length l_i, $1 \leq i \leq n$. All programs can be stored on the tape if & only if the sum of the length of the program is almost 'l'. We assume that whenever program is to be retrieved from this tape, the tape is initially positioned at the front end. Hence if the programs are stored in the order $I = I_1, I_2, --- I_n$. The time 't_j' needed to retrieve the program 'I_j' is proportional to $\sum_{1 \leq k \leq j} l_{i_k}$. If all the programs are retrieved equally often, then the expected/mean retrieval mean is $\boxed{\frac{1}{n} \sum_{1 \leq j \leq n} t_j}$

In this problem we require to find a permutation for 'n' programs so that when they are stored on a tape, Mean Retrieval Time is minimized.

Minimising the MRT is equivalent to minimising the

$$\boxed{d(I) = \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq j} l_{i_k}}$$

Eg: Let $n=3$, $(l_1, l_2, l_3) = (5, 10, 3)$

sol: $n! = 3! = 6$

Reite.

There are $n! = 3! = 3 \times 2 \times 1 = 6$ possible orderings. These orderings and their respective 'd' values are

orderings I	d(I)
1, 2, 3	5+5+10+5+10+3 = 38
1, 3, 2	5+5+3+5+3+10 = 31
2, 1, 3	10+10+5+10+5+3 = 43
2, 3, 1	10+10+3+10+3+5 = 41
3, 1, 2	3+3+5+3+5+10 = 29
3, 2, 1	3+3+10+3+10+5 = 34

The optimal ordering is (3,1,2)

Algorithm:

```
Algorithm store(n,m)
{
 j:=0;
for i:=1 to n do
{
write ("append program", i, "to permutation for tape", j);
  j:=(j+1) mod m;
  }
}
```

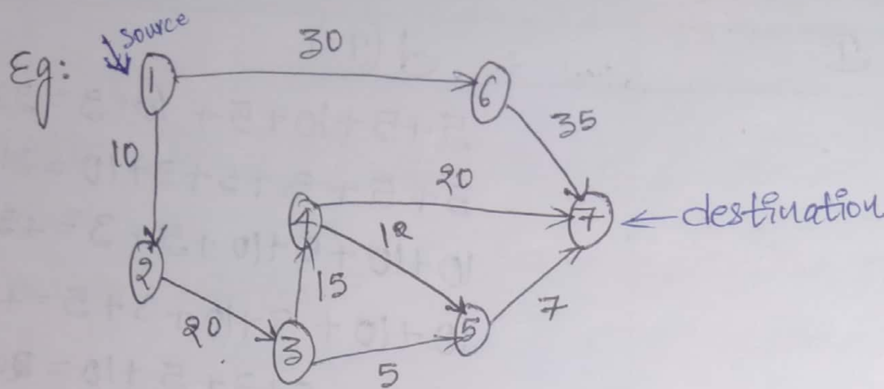
Single Source Shortest Path Problem:-

Many times graph is used to represent the distances b/n two cities. In single source shortest path problem the shortest distance from a single vertex called source is obtained.

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

last vertex is called destination. Assume that all the distances are +ve

Eg:



Start from source vertex, say 1st vertex S[1]=1. Now select next vertex from vertex '1'

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

{1,7} = ∞

~~Hence set S[2]=1~~

{1,2} produces shortest path. Hence set S[2]=1.
From vertex '2' select the next vertex.

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

{1,2,6} = ∞ {1,2,7} = ∞

{1,2,3} produces shortest path. Hence set S[3]=1
From vertex '3' select the next vertex.

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

{1,2,3,7} = ∞

{1,2,3,5} produces shortest path. Hence set S[5]=1
From vertex 5 select the next vertex

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

{1,2,3,5,7} produces shortest path. Hence set S[7]=1
From vertex '7', select the next vertex

{1,2,3,5,7,4} = ∞ {1,2,3,5,7,6} = ∞

Algorithm:

```
Algorithm shortestpath(v, cost, dist, n)   -> no. of nodes
{
for i:=1 to n do
  { s[i]:=false;
dist[i]:= cost[v,i];
}
s[v]= true, dist[v]= 0.0;     (v -> source vertex)
for num:=2 to n do
                // select u with minimum in dist
{
 s[u]:=true;
for (each w adjacent to u with s[w]:=false)
{
if (dist[w] > dist[u] + cost [u,w]) then
  dist[w]:= dist[u]+cost[u,w];
      }
   }
}
```

Dynamic programming:-

General method:-

Dynamic programming is applied to optimization problems This technique is invented by US Mathematician Richard Bellman in 1955. In the word dynamic programming, the word programming stands for planning. It is a technique for solving problems with overlapping subprograms. In this method, each subprogram is solved only once, the result of each subproblem is recorded in a table form, which can obtain a solution to the original problem.

For each given problem we may get a number of solutions

We seek for optimal solution, such an optimal solution becomes the solution to the given problem.

Steps of dynamic Programming:-

Dynamic Programming design involves 4 steps

Step 1: Characterize the structure of Optimal solution that means develop a mathematical solution that can express any solution and subsolution for the given problem.

Step 2:

Recursively define the value of an optimal solution.

Step 3:

By using Bottom-up technique, compute the value of optimal solution, for that you have to develop a recurrance ~~solut~~ relation that relates a solution to its subsolutions using mathematical solution of step 1

Step 4:

Compute an optimal solution from computed solution.

Principle of Optimality:-

The DP Algorithm obtained the solution using principle of optimality.

The principle of optimality states that "In an optimal ~~solution~~ sequences of decisions, (or) choices each sub sequence must also be optimal"

When it is not possible to apply the principle of optimality it is almost impossible to obtain the solution using dynamic programming approach

Applications of Dynamic Programming:-

1) Matrix chain multiplication
2) 0/1 knapsack problem

3) Optimal ~~sear~~ binary search trees
4) All-pairs shortest path problem
5) Travelling sales person problem
6) Reliability design

Differences b/n divide and Conquer method and dynamic programming:-

Divide and Conquer	Dynamic programming
1) The problem is divided into subproblems. These subproblems are solved independently finally all the solutions of subproblems are collected together to get the solution to the given problem.	1) In dynamic programming, many decision sequences are generated & all the overlapping sub-instances are considered.
2) In this method, duplications in subsolutions are neglected.	2) In dynamic computing, duplications are completely avoided.
3) Divide and Conquer method is less efficient because of rework on solutions.	3) DP is efficient than the divide & conquer strategy.
4) It uses top down approach of problem solving	4) DP uses bottom up approach of problem solving.
5) In this method, splits its input at specific deterministic points usually in the middle.	5) In this method, splits its input at every possible split points rather than at a particular point after trying all the split points it determine which split points is optimal

Greedy method	Dynamic programming
1) Greedy method is used for obtaining the optimum solution	1) DP is also used for obtaining the optimum solution.

2) In Greedy Method, a set of feasible solutions are obtained and picks up the optimum solution from feasible solutions	2) There is no special set of feasible solutions in this Method.
3) In greedy method, the optimum selection is without revising previously generated solution	3) In this method, consider all possible sequences in order to obtain the optimum solution
4) In Greedy method, there is no such ~~method of~~ guarantee of getting optimal solution.	4) It is guaranteed that the DP will generate optimal solution using principle of optimality.

Principle of optimality:- The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaing decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.

0/1 knapsack Problem:

In this problem we can't place fractional weights in the knapsack. Here either the item should be put into the bag completely or no item is to be put into the bag. Fraction of item can't be placed in the bag. Consider the weights $W_1, W_2, \ldots W_n$ and fraction of weights to be put into the bag should be $X_1, X_2, X_3, \ldots X_n$, $X_i = 0/1$. The dynamic programming solution for 0/1 knapsack problem is

$$F_n(m) = \text{Max}\left[F_{n-1}(M), F_{n-1}(M - W_n) + P_n\right]$$

when $X_n = 1$, then size of the bag is reduced by W_n which is the weight of nth item. As we are placing the nth item we should the profit for the last item. Similarly we find for $F_{n-1}(M)$ and so on upto $F_1(M)$

If we try to remove the ith item, then the profit of the ith item reduced from the total profit and weight of ith item from the total weight which belongs to profit and weight of ith item

$S^0 = \{0,0\}$ // Initial condition

$S_1^i = S^{i-1} + (P_i, W_i)$ // addition operation

$S^i = S^{i-1} + S_1^{i-1}$ // merging operation

$S^0 = \{(0,0)\}$

$S_1^j = S^j + (P_{j+1}, W_{j+1})$

$S^{j+1} = S^j + S_1^j$

Eg: $n=3$, $(W_1, W_2, W_3) = (2,3,4)$

$(P_1, P_2, P_3) = (1,2,5)$

Sol: $S^0 = \{0,0\}$

$S_1^1 = S^{1-1} + (P_1, W_1)$

$= S^0 + (1,2)$

$= (0,0) + (1,2)$

$= (1,2)$

$S^1 = S^{1-1} + S_1^1$

$= S^0 + (1,2)$

$= (0,0) + (1,2)$

$= \{(0,0),(1,2)\}$

i=2:

$S_1^2 = S^{2-1} + (P_2, W_2)$

$= S^1 + (2,3)$

$= \{(0,0),(1,2)\} + (2,3)$

$= \{(2,3),(3,5)\}$

$S^2 = S^{2-1} + S_1^2$

$= S^1 + S_1^2$

$= \{(0,0),(1,2)\} + \{(2,3),(3,5)\}$

$= \{(0,0),(1,2),(2,3),(3,5)\}$

i=3:

$S_1^3 = S^{3-1} + (P_3, W_3)$

$= S^2 + (5,4)$

$= \{(0,0),(1,2),(2,3),(3,5)\} + (5,4)$

$= \{(5,4),(6,6),(7,7),(8,9)\}$

$S^3 = S^{3-1} + S_1^3$

$= S^2 + \{(5,4),(6,6),(7,7),(8,9)\}$

$= \{(0,0),(1,2),(2,3),(3,5)\} + \{(5,4),(6,6),(7,7),(8,9)\}$

$= \{(0,0),(1,2),(2,3),(3,5),(5,4),(6,6),(7,7),(8,9)\}$

Purging Rule/Dominance Rule:

If one of S^{i-1} and S_1^i has a pair (P_j, W_j) and the other has a pair (P_k, W_k) and $P_j \leq P_k$ and $W_j \geq W_k$ then the pair (P_j, W_j) is discarded

(3,5) is discarded

After Applying the purging rule

$S^3 = \{(0,0),(1,2),(2,3),(5,4),(6,6),(7,7),(8,9)\}$

Scanned with CamScanner

Now we will check the following condition in order to find the solution

If $(P_i, W_i) \in S^n$ & $(P_i, W_i) \notin S^{n-1}$ then

$X_n = 1$

otherwise $X_n = 0$

If $(6,6) \in S^3$ &
$(6,6) \notin S^2 \rightarrow$ true

$X_3 = 1$

$(6,6) - (5,4) = (1,2) \in S^2$
$(1,2) \notin S^1 \rightarrow$ false

$X_2 = 0$

$(1,2) - (0,0) = (1,2) \in S^1$
$(1,2) \notin S^0 \rightarrow$ true

$X_1 = 1$

$\therefore X_1 = 1, X_2 = 0, X_3 = 1$

$P_i X_i = P_1 X_1 + P_2 X_2 + P_3 X_3$
$= 1*1 + 2*0 + 5*1$
$= 1 + 0 + 5$
$= 6$

Algorithm:

Algorithm DKP (P, W, n, M)

$S^0 \leftarrow \{(0,0)\}$

for $i := 1$ to $n-1$ do

$S_1^i \leftarrow \{(P_i, W_i) / (P - P_i), (W - W_i) \in S^{i-1}$ & $W_i \le M$

$S^i \leftarrow$ MERGE_PURGE (S^{i-1}, S_1^i)

repeat

$(P_x, W_x) \leftarrow$ last tuple in S^{n-1}

$(P_y, W_y) \leftarrow (P_i + P_n, W_i + W_n)$

Scanned with CamScanner

where w_i is the largest
w in any tuple in S^{n-1} such that
$w + w_n \leq m$
If $P_x > P_y$ then
$x_n \leftarrow 0$;
else
$x_n \leftarrow 1$
end if
end OKP

All pairs shortest path problem:-

When a weighted graph represented by its weight Matrix 'w' then the objective is to find the distance b/n every pair of node.

Step 1: We will decompose the given problem into subproblems. Let '$A^k_{(i,j)}$' be the length of shortest path from node 'i' to node 'j' such that the label for intermediary node will be $\leq k$. We will compute A^k for $k = 1, 2, \ldots n$ for 'n' nodes

Step 2: Any subpath of shortest path is a shortest path b/n the 'n' nodes. Divide the paths from i^{th} node to j^{th} node for every intermediary node, say 'k' then there arises 2 cases

Case (i): Path going from i to j through 'k'

Case (ii): Path which is not going through 'k'

Select only the shortest path from these two cases

Step 3: The shortest path can be computed using bottom-up Computation method, here follows recursive method

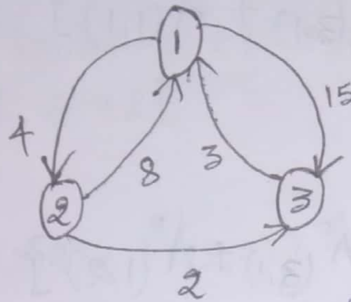
Initially

$A^0 = w[i, j]$

Next Computations

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

Eg: Calculate all pairs shortest path for the following graph



Sol: ① Intially the weight matrix 'W' is

$$A^0 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 15 \\ 2 & 8 & 0 & 2 \\ 3 & 3 & \infty & 0 \end{array}$$

To generate the next Matrices the equation is

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

Now $k=1$

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

$$A^1_{(1,2)} = \min\{A^0_{(1,2)}, A^0_{(2,1)} + A^0_{(1,2)}\}$$
$$= \min\{4, 8+4\}$$
$$= 4$$

$$A^1_{(1,3)} = \min\{A^0_{(1,3)}, A^0_{(1,1)} + A^0_{(1,3)}\}$$
$$= \min\{15, 0+15\}$$
$$= 15$$

$$A^1_{(2,1)} = \min\{A^0_{(2,1)}, A^0_{(2,1)} + A^0_{(1,1)}\}$$
$$= \min\{8, 8+0\}$$
$$= 8$$

$$A^1_{(2,2)} = \min\{A^0_{(2,2)}, A^0_{(2,1)} + A^0_{(1,2)}\} = \min\{0, 8+4\}$$
$$= 0$$

$A^1(2,3) = \min\{A^0(2,3),\ A^0(3,1) + A^0(1,3)\}$
$= \min\{2,\ 3+15\}$
$= 2$

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

$A^1(3,2) = \min\{A^0(3,2),\ A^0(3,1) + A^0(1,2)\}$
$= \min\{6,\ 3+4\}$
$= 7$

$A^1(3,3) = \min\{A^0(3,3),\ A^0(3,1) + A^0(1,3)\}$
$= \min\{0,\ 3+15\}$
$= 0$

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{matrix} 1 & 2 & 3 \end{matrix} \\ \begin{bmatrix} 0 & 4 & 15 \\ 8 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array}$$

<u>k=2 :</u>

$A^2(1,1) = \min\{A^1(1,1),\ A^1(1,2) + A^1(2,1)\}$
$= \min\{0,\ 4+8\}$
$= 0$

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

$A^2(1,3) = \min\{A^1(1,3),\ A^1(1,2) + A^1(2,3)\}$
$= \min\{15,\ 4+2\}$
$= \{6\}$

$A^2(2,1) = \min\{A^1(2,1),\ A^1(2,2) + A^1(2,1)\}$
$= \min\{8,\ 0+8\}$
$= 8$

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

Scanned with CamScanner

$A^2(2,3) = \min\{A^1(2,3), A^1(2,2) + A^1(2,3)\}$
$= \min\{2, 0+2\}$
$= 2$

$A^2(3,1) = \min\{A^1(3,1), A^1(3,2) + A^1(2,1)\}$
$= \min\{3, 7+8\}$
$= 3$

$A^2(3,2) = \min\{A^1(3,2), A^1(3,2) + A^1(2,2)\}$
$= \min\{7, 7+0\}$
$= 7$

$$A^2 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 6 \\ 2 & 8 & 0 & 2 \\ 3 & 3 & 7 & 0 \end{array}$$

<u>K=3:</u>

$A^3(2,1) = \min\{A^2(2,1), A^2(2,3) + A(3,1)\}$
$= \min\{8, 2+3\}$
$= 5$

$A^3(1,2) = \min\{A^2(1,2), A^2(1,3) + A^2(3,2)\}$
$= \min\{4, 6+7\}$
$= 4$

$A^3(1,3) = \min\{A^2(1,3), A^2(1,3) + A^2(3,3)\}$
$= \min\{6, 6+0\}$
$= 6$

$A^3(2,3) = \min\{A^2(2,3), A^2(2,3) + A^2(3,3)\}$
$= \min\{2, 2+0\}$
$= 2$

$A^3(3,1) = \min\{A^2(3,1), A^2(3,3) + A^2(3,1)\}$
$= \min\{3, 0+3\}$
$= 3$

$A^3(3,2) = \min\{A^2(3,2), A^2(3,3) + A(3,2)\}$
$= \min\{7, 0+7\} = 7$

Scanned with CamScanner

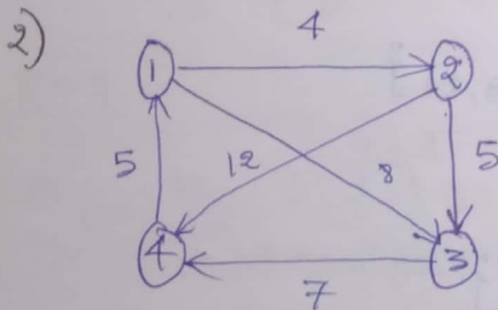
$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{matrix} 1 & 2 & 3 \end{matrix} \\ \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array}$$

A^3 gives shortest distance between any pair of vertices

Algorithm:

```
Algorithm Allpairs (Cost, A, n)
{
for i:=1 to n do
for j:=1 to n do
   A[i,j] = Cost[i,j];
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]);
}
```

2)



Initially the weight matrix is

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{bmatrix} 0 & 4 & 8 & \infty \\ \infty & 0 & 5 & 12 \\ \infty & \infty & 0 & 7 \\ 5 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

To generate the next matrices the equation is

$A^k_{(i,j)} = \min$

Multistage Graphs:

A multistage graph $G=(V,E)$ is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets V_i, $1 \leq i \leq k$. In addition, if <u,v> is an edge in E, then $u \in V$, and $v \in V_{i+1}$ for some 'i', $1 \leq i < k$.

Let the vertex 's' is the source and 't' the sink. Let $c(i,j)$ be the cost of edge <i,j>. The cost of a path from 's' to 't' is the sum of the costs of the edges on the path. The multistage graph problem is to find a minimum cost path from 's' to 't'. Each set 'V_i' defines a stage in the graph. Because of the constraints on E, every path from 's' to 't' starts in stage 1, goes to stage 2, then to stage 3, then to stage 4, and so on, and eventually terminates in stage k.

A dynamic programming formulation for k-stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of k-2 decisions. The ith decision involves determining which vertex in V_{i+1}, $1 \leq i \leq k-2$, is to be on the path. Let $c(i,j)$ be the cost of the path from source to destination.

To solve the multistage graph we follow 2 approaches
1) forward approach
2) Backward approach

By using forward approach we obtain

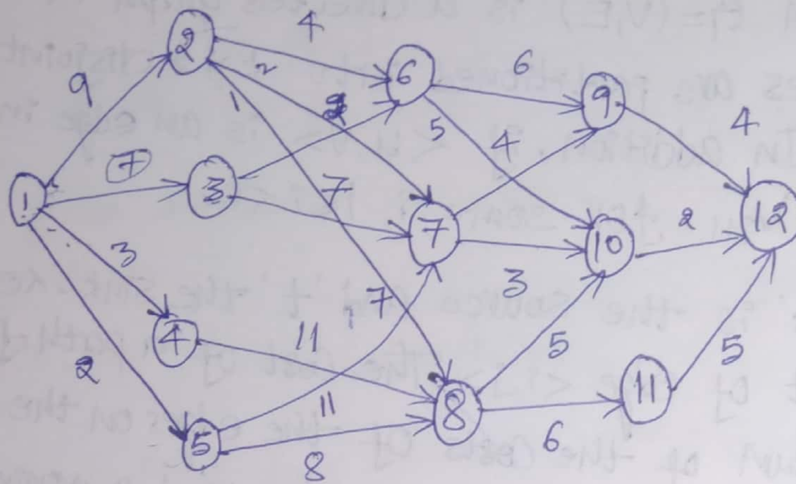
$$\text{cost}(i,j) = \min_{\substack{l \in V_{i+1} \\ \langle j,l \rangle \in E}} \{ c(j,l) + \text{cost}(i+1, l) \}$$

By using backward approach we obtain

$$\text{Bcost}(i,j) = \min_{l \in V_{i-1}} \{ \text{Bcost}(i-1, l) + c(l,j) \}$$

Scanned with CamScanner

Example:



Using Forward approach:-

We use the following equation to find the minimum cost from s to t

$$Cost(i,j) = \min_{\substack{l \in V_{i+1} \\ \langle j,l \rangle \in E}} \{ C(j,l) + Cost(i+1, l) \}$$

(i starts from 1 to 5)

Node 1: $i=1, j=1$

$Cost(1,1) = \min \{ C(1,2) + Cost(2,2),\ C(1,3) + Cost(2,3),\ C(1,4) + Cost(2,4),\ C(1,5) + Cost(2,5) \}$

$= \min \{ 9 + Cost(2,2),\ 7 + Cost(2,3),\ 3 + Cost(2,4),\ 2 + Cost(2,5) \}$

$Cost(2,2) = \min \{ C(2,6) + Cost(3,6),\ C(2,7) + Cost(3,7),\ C(2,8) + Cost(3,8) \}$

$= \min \{ 4 + Cost(3,6),\ 2 + Cost(3,7),\ 1 + Cost(3,8) \}$

$Cost(3,6) = \min \{ C(6,9) + Cost(4,9),\ C(6,10) + Cost(4,10) \}$

$= \min \{ 6 + Cost(4,9),\ 5 + Cost(4,10) \}$

$Cost(4,9) = \min \{ C(9,12) + Cost(5,12) \} = \min \{4+0\} = 4$

$Cost(4,10) = \min \{ C(10,12) + Cost(5,12) \} = \min \{2+0\} = 2$

Scanned with CamScanner

$Cost(3,6) = \min\{6+4, 5+2\} = 7$

$Cost(3,7) = \min\{C(7,9) + cost(4,9), C(7,10) + cost(4,10)\}$

$= \min\{4 + Cost(4,9), 3 + cost(4,10)\}$

$= \min\{4+4, 3+2\} = 5$

~~$Cost(4,7) = \min\{C(7,9) + Cost(5,9), C(7,10) + Cost(5,10)\}$~~

~~$= \min\ 4 + Cost(5,9)$~~

$Cost(3,8) = \min\{C(8,10) + cost(4,10), C(8,11) + cost(4,11)\}$

$= \min\{5 + Cost(4,10), 6 + Cost(4,11)\}$

$= \min\{5+2, 6 + Cost(4,11)\}$

$Cost(4,11) = \min\{C(11,12) + cost(5,12)\}$

$= \min\{5+0\}$

$= 5$

$Cost(3,8) = \min\{7, 6+5\}$

$= 7$

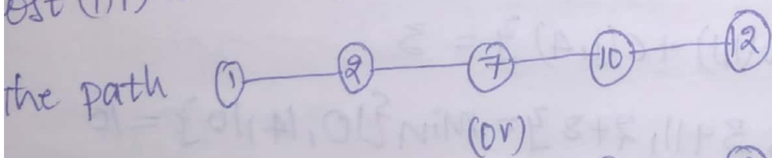
$Cost(2,2) = \min\{4+7, 2+5, 1+7\}$

$= \min\{11, 7, 8\}$

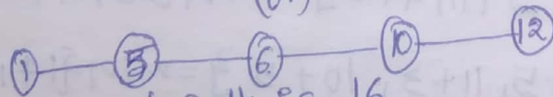
$= 7$

$Cost(1,1) = \min\{9+7, 2 + Cost(2,3), 3 + Cost(2,4), 2 + cost(2,5)\}$

The path



(or)



The minimum cost path is 16

Using Backward approach:

We use the following equation to find the minimum cost from s to t

$$Cost(i,j) = \min_{\substack{l \in V_{i-1} \\ \langle l,j \rangle \in E}} \{BCost(i-1, l) + C(l,j)\}$$

i starts from 5 to 1

Node 1: $i = 5, j = 12$

$Cost(5,12) = \min\{BCost(4,9) + C(9,12), BCost(4,10) + C(10,12),$
$BCost(4,11) + C(11,12)\}$

$= \min\{BCost(4,9)+4,\ BCost(4,10)+2,\ BCost(4,11)+5\}$

$BCost(4,9) = \min\{BCost(3,6)+C(6,9),\ BCost(3,7)+C(7,9)\}$
$= \min\{BCost(3,6)+6,\ BCost(3,7)+4\}$

$BCost(3,6) = \min\{BCost(2,2)+C(2,6),\ BCost(2,3)+C(3,6)\}$
$= \min\{BCost(2,2)+4,\ BCost(2,3)+2\}$

$BCost(2,2) = \min\{BCost(1,1)+C(1,2)\} = \min\{0+9\} = 9$

$BCost(2,3) = \min\{BCost(1,1)+C(1,3)\} = \min\{0+7\} = 7$

$BCost(3,6) = \min\{9+4,\ 7+2\} = \min\{13,9\} = 9$

$BCost(3,7) = \min\{BCost(2,2)+C(2,7),\ BCost(2,3)+C(3,7),$
$BCost(2,5)+C(5,7)\}$

$BCost(2,5) = \min\{BCost(1,1)+C(1,5)\} = 2$

$BCost(3,7) = \min\{9+2,\ 7+7,\ 2+11\} = \min\{11,14,13\} = 11$

$BCost(4,9) = \min\{9+6,\ 11+4\} = \min\{15,15\} = 15$

$BCost(4,10) = \min\{BCost(3,6)+C(6,10),\ BCost(3,7)+C(7,10),$
$BCost(3,8)+C(8,10)\}$

$BCost(3,8) = \min\{BCost(2,2)+C(2,8),\ BCost(2,4)+C(4,8),$
$BCost(2,5)+C(5,8)\}$

$BCost(2,4) = \min\{BCost(1,1)+C(1,4)\} = 3$

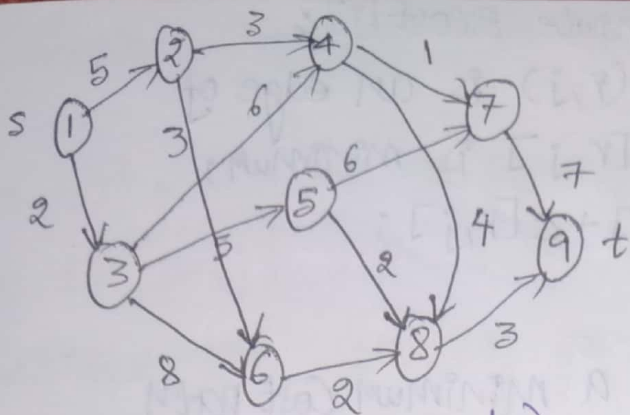
$BCost(3,8) = \min\{9+1,\ 3+11,\ 2+8\} = \min\{10,14,10\} = 10$

$BCost(4,10) = \min\{9+5,\ 11+3,\ 10+5\} = \min\{14,14,15\} = 14$

$BCost(4,11) = \min\{BCost(3,8)+C(8,11)\}$

$BCost(5,12) = \min\{15+4,\ 14+2,\ 16+5\} = \min\{19,16,21\} = 16$

Eg:2



Algorithm:- (Forward approach)

Algorithm Fgraph (G, k, n, p)

// The input is a k-stage graph G=(V, E) with n vertices

// indexed in order or stages, E is a set of edges and C[i, j]

// is the cost of (i, j). p[1:k] is a minimum cost path

```
{
  cost[n] := 0.0;
  for j := n-1 to 1 step -1 do           // compute cost[j]
  {
    let r be a vertex such that [j, r] is an edge
    of G and c[j, r] + cost[r] is minimum;
    cost[j] := c[j, r] + cost[r];
      d[j] := r;
  }
  p[1] := 1; p[k] := n;   // Find a minimum cost path
  for j := 2 to k-1 do p[j] := d[p[j-1]];
}
```

Algorithm for backward approach:-

Algorithm Bgraph (G, k, n, p)

```
// same function as Fgraph
{
BCost[1] := 0.0;
 for j := 2 to n do
```

{ //Compute Bcost[j]//
Let r be such that (r, j) is an edge of
G and Bcost[r] + c[r, j] is minimum;
Bcost[j] := Bcost[r] + c[r, j];
D[j] := r;
} //find a minimum cost path
p[1] := 1; p[k] := n;
for j := k-1 to 2 do p[j] := d[p[j+1]];
}

Complexity Analysis:
The complexity analysis of the algorithm is fairly straightforward. Here, if G has |E| edges, then the time for the first for loop is ϕ V E

Travelling sales Person Problem:-

In this problem the sales person should start at a starting point and travel all the places and come back to the starting point. In this problem we need to minimise the travelling cost.

Eg: Suppose we have to route a postal van to pick up the mails from the mail boxes located at 'n' different sites. n+1 vertex graph may be used to represent the situation, one vertex represents the post office from which the postal van starts and to which it must return. The route taken by the postal van should have minimum length (or) cost. To solve this problem the following equations are used.

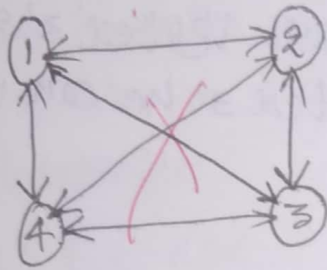
1) $g(i, \phi) = C_{i1},\ 1 \le i \le n$

2) $g(i, s) = \min_{j \in s} \{ C_{ij} + g(j, s - \{j\}) \}$

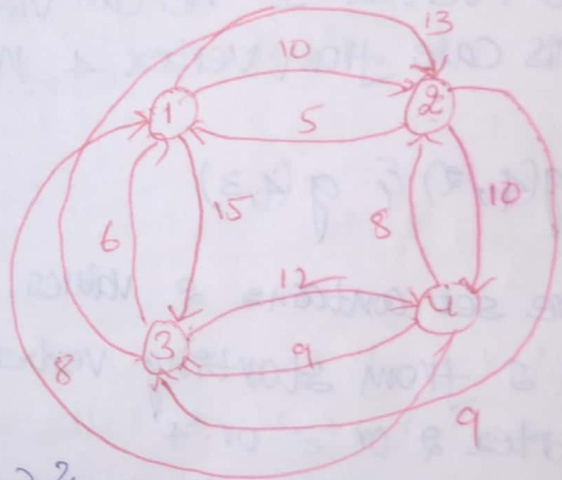
min is considered as the intermediate Node.
$j \in S$

$\{g(j, S-\{j\})\}$ means 'j' is already traversed so next we have to traverse $S-\{j\}$ with 'j' as starting point.

Eg: 1) Consider the following graph to solve travelling sales person problem



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0



sol: $g(i, \phi) = C_{i1}, \quad 1 \le i \le n$

$g(1, \phi) = C_{11} = 0$

$g(2, \phi) = C_{21} = 5$

$g(3, \phi) = C_{31} = 6$

$g(4, \phi) = C_{41} = 8$

$g(i, S) = \min_{j \in S} \{C_{ij} + g(j, S-\{j\})\}$

$|S| = 1$

set contains only 1 element and here we make an assumption that the sales person starts at vertex 1 from that vertex, he can move to vertex '2' if, he visits vertex '2', from that vertex he can visit next either vertex '3' or vertex '4' since $g(2,3)$ & $g(2,4)$

$g(2,3) = \min_{j \in S} \{C_{23} + g(3, \phi)\} = \min_{j \in S} \{9+6\} = 15$

$g(2,4) = \min_{j \in S} \{C_{24} + g(4, \phi)\} = \min_{j \in S} \{10+8\} = 18$

$g(3,2) = \min_{j \in S} \{C_{32} + g(2, \phi)\} = \min_{j \in S} \{13+5\} = 18$

$g(3,4) = \min_{j \in S} \{C_{34} + g(4,\phi)\} = \min_{j \in S} \{12+8\} = 20$

$g(4,2) = \min_{j \in S} \{C_{42} + g(2,\phi)\} = \min_{j \in S} \{8+5\} = 13$

$g(4,3) = \min_{j \in S} \{C_{43} + g(3,\phi)\} = \min_{j \in S} \{9+6\} = 15$

$|S| = 2$

From vertex '1' next he can visit vertex '3' instead of vertex '2' in this case from vertex '3' he can visit either vertex '2' or '4'

$\therefore g(3,2)$ & $g(3,4)$

From vertex '1' he can visit vertex '4' instead of '3'. In this case from vertex '4' next he can visit either '2' (or) 4

$\therefore g(4,2)$ & $g(4,3)$

Here set contains 2 values. So we can place two vertices in 'S' from starting vertex '1'. Next he can visit either vertex '2' or '3' or '4'

If he visits vertex '2' then from that vertex he can visit either vertex '3' or vertex '4'. So $g(2,\{3,4\})$

If he visits vertex '3' from that vertex he can visit either '2' or '4'. So $g(3,\{2,4\})$

If he visits vertex '4' from that vertex he can visit either vertex '2' (or) '3'. So $g(4,\{2,3\})$.

$g(2,\{3,4\}) = \min_{j \in S} \{C_{23} + g(3,4), C_{24} + g(4,3)\}$

$= \min_{j \in S} \{9+20, 10+15\} = 25$

$g(3,\{2,4\}) = \min_{j \in S} \{C_{32} + g(2,4), C_{34} + g(4,2)\}$

$= \min_{j \in S} \{13+18, 12+13\} = 25$

$g(4, \{2,3\}) = \min_{j \in S} \{c_{42} + g(2,3), c_{43} + g(3,2)\}$

$= \min_{j \in S} \{8+15, 9+18\}$

$= 23$

$|S| = 3$

Now the set contains 3 elements after starting from vertex 1, next he can visit either vertex '2' or '3' or '4'. so determine

$g\{1, \{2,3,4\}\} = \min_{j \in S} \{c_{12} + g\{2\{3,4\}\}, c_{13} + g\{3\{2,4\}\}, c_{14} + g\{4\{2,3\}\}\}$

$= \min_{j \in S} \{10+25, 15+25, 20+23\}$

$= \min_{j \in S} \{35, 40, 43\}$

$= 35$

$1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1 = 35$

↑ 2 or 4, 4 is visited here

Optimal binary search trees:-

Let us assume that the given set of identifiers is $\{a_1, --- a_n\}$ with $a_1 < a_2 < --- < a_n$. Let $p(i)$ be the probability with which we search for a_i. Let $q(i)$ be the probability that the identifier x being searched for is such that $a_i < x < a_{i+1}$, $0 \leq i \leq n$ (assume $a_0 = -\infty$ and $a_{n+1} = +\infty$). We have to arrange the identifiers in a binary search tree in a way that minimizes the expected total access time.

In a binary search tree, the number of comparisons

needed to access an element at depth 'd' is d+1, so if 'a_i' is placed at depth d_i, then we want to minimize

$\sum_{i=1}^{n} P_i (1+d_i)$

Let P(i) be the probability with which we shall be searching for 'a_i'. Let Q(i) be the probability of an un-successful search. Every internal node represents a point where a un-successful search may terminate.

The expected cost contribution for the internal node for 'a_i' is:

$P(i) * level(a_i)$

Unsuccessful search terminate with I=0. Hence the cost contribution for this node is:

$Q(i) * level((E_i)-1)$

The expected cost of binary search tree is:

$$\sum_{i=1}^{n} P(i) * level(a_i) + \sum_{i=0}^{n} Q(i) * level((E_i)-1)$$

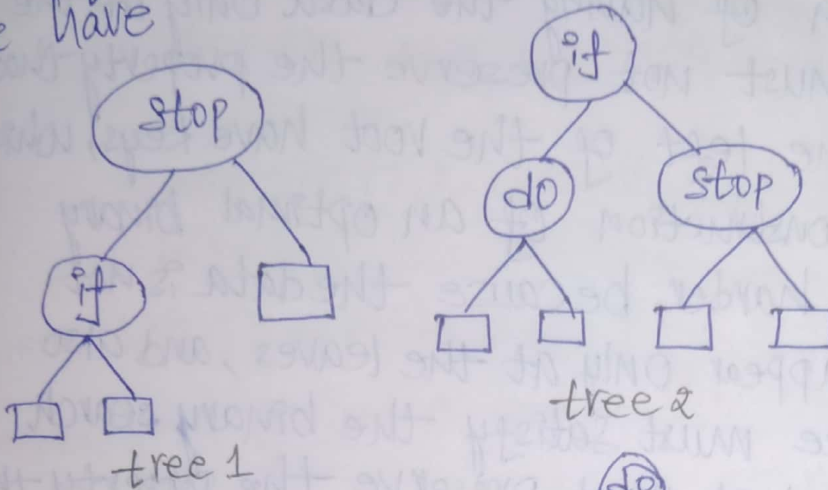
Given a fixed set of identifiers, we wish to create a binary search tree organization. We may expect different binary search trees for the same identifier set to have different performance characteristics.

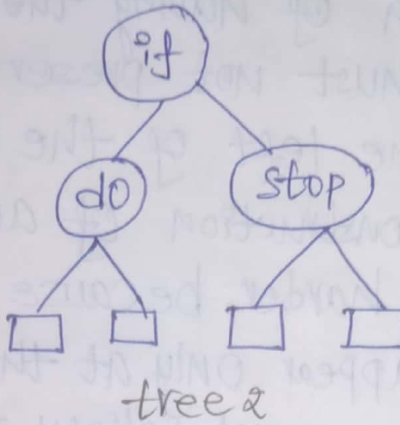
The computation of each of these C(i,j)'s requires us to find the minimum of 'm' quantities. Hence each such C(i,j) can be computed in time O(m). The total time for all C(i,j)'s with j-i=m is therefore $O(nm-m^2)$

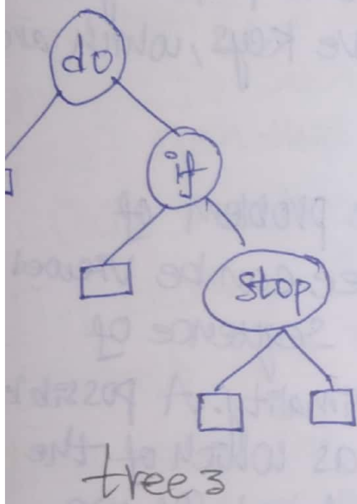
The total time to evaluate all the C(i,j)'s and r(i,j)'s is therefore:

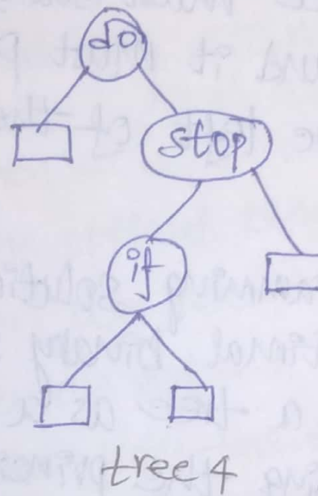
$$\sum_{1 \le m \le n} (nm-m^2) = O(n^3)$$

Eg-1) : The possible binary search tree for the identifier set $(a_1, a_2, a_3) = (d_0, if, stop)$ are as follows. Given the equal probabilities $P(i) = Q(i) = 1/7$ for all i, we have









$$Cost(tree\#1) = \left(\frac{1}{7}\times1 + \frac{1}{7}\times2 + \frac{1}{7}\times3\right) + \left(\frac{1}{7}\times1 + \frac{1}{7}\times2 + \frac{1}{7}\times3 + \frac{1}{7}\times3\right)$$

$$= \frac{1+2+3}{7} + \frac{1+2+3+3}{7} = \frac{6+9}{7} = \frac{15}{9}$$

$$Cost(tree\#2) = \left(\frac{1}{7}\times1 + \frac{1}{7}\times2 + \frac{1}{7}\times2\right) + \left(\frac{1}{7}\times2 + \frac{1}{7}\times2 + \frac{1}{7}\times2 + \frac{1}{7}\times2\right)$$

$$= \frac{1+2+2}{7} + \frac{2+2+2+2}{7} = \frac{5+8}{7} = \frac{13}{7}$$

$$Cost(tree\#3) = \left(\frac{1}{7}\times1 + \frac{1}{7}\times2 + \frac{1}{7}\times3\right) + \left(\frac{1}{7}\times1 + \frac{1}{7}\times2 + \frac{1}{7}\times3 + \frac{1}{7}\times3\right)$$

$$= \frac{1+2+3}{7} + \frac{1+2+3+3}{7} = \frac{6+9}{7} = \frac{15}{7}$$

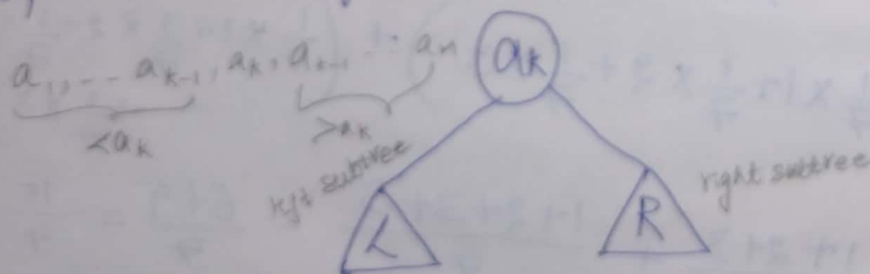
$$Cost(tree\#4) = \left(\frac{1}{7}\times1 + \frac{1}{7}\times2 + \frac{1}{7}\times3\right) + \left(\frac{1}{7}\times1 + \frac{1}{7}\times2 + \frac{1}{7}\times3 + \frac{1}{7}\times3\right)$$

Scanned with CamScanner

$$= \frac{1+2+3}{7} + \frac{1+2+3+3}{7} = \frac{15}{7}$$

Huffman Coding tree solved by a greedy algorithm has a limitation of having the data only at the leaves and it must not preserve the property that all nodes to the left of the root have keys, which are less etc. Construction of an optimal binary search tree is harder, because the data is not constrained to appear only at the leaves, and also because the tree must satisfy the binary search tree property and it must preserve the property that all nodes to the left of the root have keys, which are less

A dynamic programming solution to the problem of obtaining an optimal binary search tree can be viewed by constructing a tree as a result of sequence of decisions by holding the principle of optimality. A possible approach to this is to make a decision as which of the a_i's be arraigned to the root node at 'T'. If we choose a_k then is clear that the central internal nodes for $a_1, a_2 --- a_{k-1}$ as well as the external nodes for the classes $E_0, E_1, --- E_{k-1}$ will lie in the left subtree, L, of the root. The remaining nodes will be in the right subtree, R. The structure of an optimal binary search tree is:



$$Cost(K) = \sum_{i=1}^{k} P(i) * level(a_i) + \sum_{i=0}^{k} Q(i) * (level(E_i) - 1)$$

$cost(R) = \sum_{i=k}^{n} P(i) * level(a_i) + \sum_{i=k}^{n} Q(i) * (level(E_i) - 1)$

the $C(i,j)$ can be Computed as

$$C(i,j) = \min_{l<k\leq j} \{ C(i,k-1) + C(k,j) + P(k) + W(i,k-1) + W(k,j) \}$$

$$= \min_{l<k\leq j} \{ C(i,k-1) + C(k,j) \} + W(i,j) \longrightarrow ①$$

(→ column value)

where $W(i,j) = P(j) + Q(j) + W(i,j-1) \longrightarrow ②$

Initially $C(i,i) = 0$ and $W(i,i) = Q(i)$ for $0 \leq i \leq n$

Equation ① may be solved for $C(0,n)$ by first computing all $C(i,j)$ such that $j-i=1$. Next, we can compute all $C(i,j)$ such that $j-i=2$, then all $(C(i,j)$ with $j-i=3$, and so on. $C(i,j)$ is the cost of the optimal binary search tree 'T_{ij}' during Computation we record the root $R(i,j)$ of each tree 'T_{ij}'. Then an optimal binary search tree may be constructed from these $R(i,j)$. $R(i,j)$ is the value of 'k' that minimizes equation ①

We solve the problem by knowing $W(i,i+1)$, $C(i,i+1)$ and $R(i,i+1)$, $0 \leq i <$ knowing $W(i,i+2)$, $C(i,i+2)$ and $R(i,i+2)$, $0 \leq i < 3$ and repeating until $W(0,n)$, $C(0,n)$ and $R(0,n)$ are obtained.

The results are tabulated to recover the actual tree.

Example 1: Let $n=4$ and $(a_1, a_2, a_3, a_4) =$ (do, if, int, while). Let $P(1:4) = (3,3,1,1)$ and $Q(0:4) = (2,3,1,1,1)$

Solution:

Table for recording $W(i,j)$, $C(i,j)$ and $R(i,j)$

This computation is carried out row-wise from row 0 to row 4.

Initially, $W(i,i) = Q(i)$ and $C(i,i) = 0$ and $R(i,i) = 0$, $0 \le i < 4$.

Solving for $C(0,n)$:

First, computing all $C(i,j)$ such that $j-1 = 1$; $j = i+1$ and as $0 \le i < 4$; $i = 0,1,2,3$; $i < k \le j$. start with $i = 0$; so $j = 1$; as $i < k \le j$, so the possible value for $k = 1$.

$W(0,1) = P(1) + Q(1) + W(0,0) = 3+3+2 = 8$

$C(0,1) = W(0,1) + \min\{C(0,0) + C(1,1)\} = 8$

$R(0,1) = 1$ (value of 'k' that is minimum in the above equation)

Next with $i = 1$; so $j = 2$; as $i < k \le j$, so the possible value for $k = 2$

$W(1,2) = P(2) + Q(2) + W(1,1) = 3+1+3 = 7$

$C(1,2) = W(1,2) + \min\{C(1,1) + C(2,2)\} = 7$

$R(1,2) = 2$

Next with $i = 2$; so $j = 3$; as $i < k \le j$, so the possible value for $k = 3$

$W(2,3) = P(3) + Q(3) + W(2,2) = 1+1+1 = 3$

$C(2,3) = W(2,3) + \min\{C(2,2) + C(3,3)\} = 3 + [(0+0)] = 3$

Next with $i = 3$; so $j = 4$; as $i < k \le j$, so the possible value for $k = 4$

$W(3,4) = P(4) + Q(4) + W(3,3) = 1+1+1 = 3$

$C(3,4) = W(3,4) + \min\{[C(3,3) + C(4,4)]\} = 3 + [(0+0)] = 3$

$R(3,4) = 4$

Second, computing all $C(i,j)$ such that $j-1 = 2$; $j = i+2$ and as $0 \le i < 3$; $i = 0,1,2$; $i < k \le j$. start with $i = 0$; so $j = 2$; as $i < k \le j$, so the possible values for $k = 1$ and 2

$W(0,2) = P(2) + Q(2) + W(0,1) = 3+1+8 = 12$

$C(0,2) = W(0,2) + \min\{(C(0,0)+C(1,2), C(0,1)+C(2,2))\}$

$= 12 + \min\{(0+7, 8+0)\}$

$= 19$

$R(0,2) = 1$

Next, with $i=1$; so $j=3$; as $i<k\leq j$, so the possible value for $k=2$ and 3

$W(1,3) = P(3) + Q(3) + W(1,2) = 1+1+7 = 9$

$C(1,3) = W(1,3) + \min\{(C(1,1)+C(2,3)), [C(1,2)+C(3,3)]\}$

$= W(1,3) + \min\{(0+3), (7+0)\}$

$= 9+3$

$= 12$

$R(1,3) = 2$

Next, with $i=2$; so $j=4$; as $i<k\leq j$, so the possible values for $k=3$ and 4

$W(2,4) = P(4) + Q(4) + Q(2,3) = 1+1+3 = 5$

$C(2,4) = W(2,4) + \min\{[C(2,2)+C(3,4)], [C(2,3)+C(4,4)]\}$

$= 5 + \min\{(0+3), (3+0)\} = 5+3$

$= 8$

$R(2,4) = 3$

Third, Computing all $C(i,j)$ such that $j-i=3$; $j=i+3$ and as $0\leq i<2$; $i=0,1$; $i<k\leq j$. Start with $i=0$; so $j=3$; as $i<k\leq j$; so the possible values for $k=1,2$ and 3

$W(0,3) = P(3) + Q(3) + W(0,2) = 1+1+12 = 14$

$C(0,3) = W(0,3) + \min\{[C(0,0)+C(1,3)], [C(0,1)+C(2,3)], [C(0,2)+C(3,3)]\}$

$= 14 + \min\{(0+12), (8+3), (19+0)\} = 14+11 = 25$

$R(0,3) = 2$

Scanned with CamScanner

Start with $i=1$; so $j=4$; as $i<k\leq j$, so the possible values for $k = 2, 3$ and 4

$W(1,4) = P(4) + Q(4) + W(1,3) = 1+1+9 = 11$

$C(1,4) = W(1,4) + \min\{[C(1,1)+C(2,4)], [C(1,2)+C(3,4)], [C(1,3)+C(4,4)]\}$

$= 11 + \min\{(0+8), (7+3), (12+0)\} = 11+8 = 19$

$R(1,4) = 2$

Fourth, computing all $C(i,j)$ such that $j-i=4$; $j=i+4$ and as $0\leq i<1$, $i=0$; $i<k\leq j$

$W(0,4) = P(4) + Q(4) + W(0,3) = 1+1+14 = 16$

$C(0,4) = W(0,4) + \min\{[C(0,0)+C(1,4)], [C(0,1)+C(2,4)], [C(0,2), C(3,4)], [C(0,3)+C(4,4)]\}$

$= 16 + \min\{(0+19), (8+8), (19+3), (25+0)\}$

$= 16+16$

$= 32$

$R(0,4) = 2$

Column / Row	0	1	2	3	4
0	2,0,0	3,0,0	1,0,0	1,0,0	1,0,0
1	8,8,1	7,7,2	3,3,3	3,3,4	
2	12,19,1	19,12,2	5,8,3		
3	14,25,2	11,19,2			
4	16,32,2				

Scanned with CamScanner

Algorithm:

Algorithm OBST (p, q, n)

{

for i := 0 to n-1 do

{

w[i, 1] = q(i), r[i, 1] = 0, c[i, i] = 0.0

w[i, i+1] :=

From the table we see that $C(0,4) = 32$ is the minimum cost of a binary search tree for (a_1, a_2, a_3, a_4). The root of the tree 'T_{04}' is 'a_2'

Hence the left subtree is 'T_{01}' and right subtree is T_{24}. The root of 'T_{01}' is 'a_1' and the root of 'T_{24}' is a_3.

the left and right sub-trees for 'T_{01}' are 'T_{11}' respectively

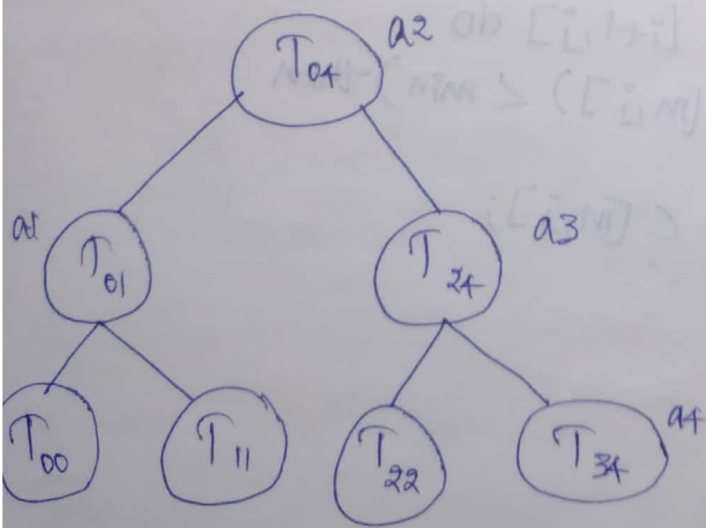
The root of T_{01} is 'a_1'

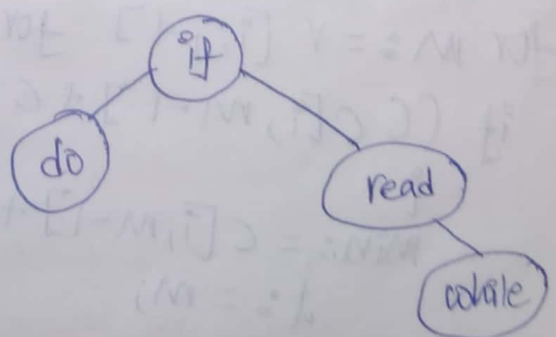
The left and right sub-trees for T_{24} are T_{22} and T_{34} respectively.

the root of 'T_{24}' is 'a_3'

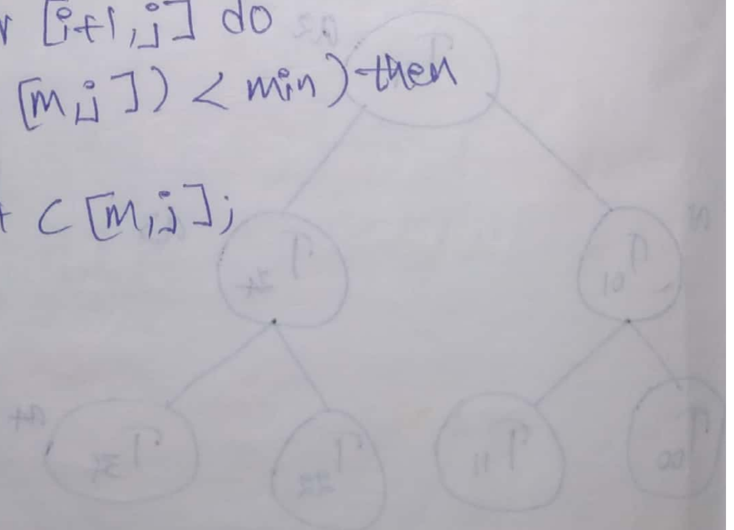
The root of T_{22} is null

the root of T_{34} is a_4





```
Algorithm OBST (P, q, n)
{
for i := 0 to n-1 do
 {
  w[i,i] = q(i), r[i,i] = 0, c[i,i] = 0.0;
  w[i,i+1] := q(i) + q[i+1] + p[i+1];
  r[i,i+1] = i+1;
  c[i,i+1] := q[i] + q[i+1] + p[i+1];
 }
w[n,n] := q[n]; r[n,n] := 0; c[n,n] := 0.0;
for m := 2 to n do
 for i := 0 to n-m do
   j := i+m;
  w[i,j] := w[i,j-1] + p[j] + q[j];
   k = Find (c, r, i, j);
  c[i,j] := w[i,j] + c[i,k-1] + c[k,j];
     r[i,j] := k;
 }
write (c[0,n], w[0,n], r[0,n]);
}
Algorithm Find (c, r, i, j)
 {
  min := ∞
for m := r [i,j-1] for [i+1,j] do
 if ((c[i,m-1] + c[m,j]) < min) then
   {
   min := c[i,m-1] + c[m,j];
      l := m;
      }
     return l;
  }
```



Unit-III

Techniques for binary trees:-

Binary tree:

Binary tree is a tree which has atmost 2 child nodes.

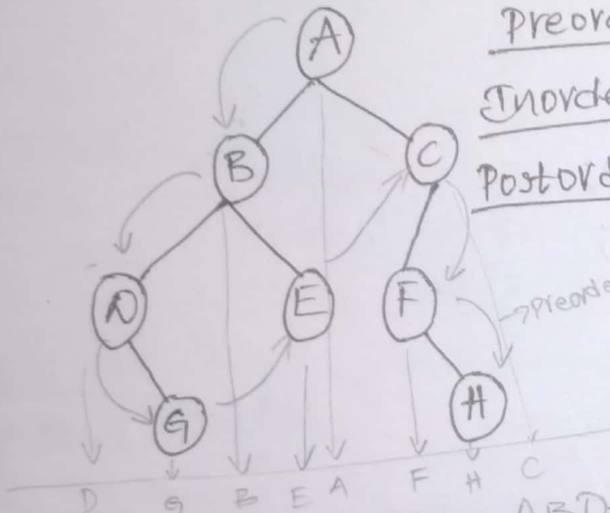
Traversal Techniques:

There are 3 types of traversal techniques in binary tree They are

1) Preorder : P, L, R (Parent, left, right)
2) Inorder : L, P, R (left, parent, right)
3) Postorder : L, R, P (left, right, parent)

10 or 2m

1) Find the traversal techniques for the following tree

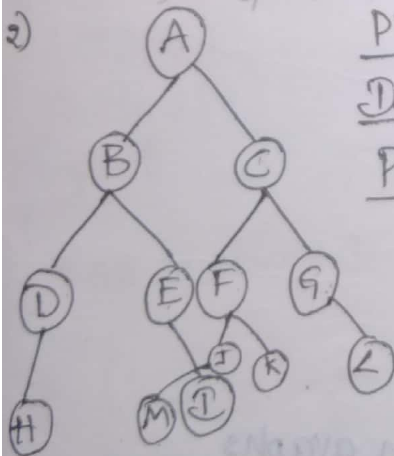


Preorder: ABDGECFH → first fall

Inorder: DGBEAFHC – Free fall

Postorder: GDEBHFCA → leaf break
first break left leaf

2)



Preorder: ABDHEICFJMKGL

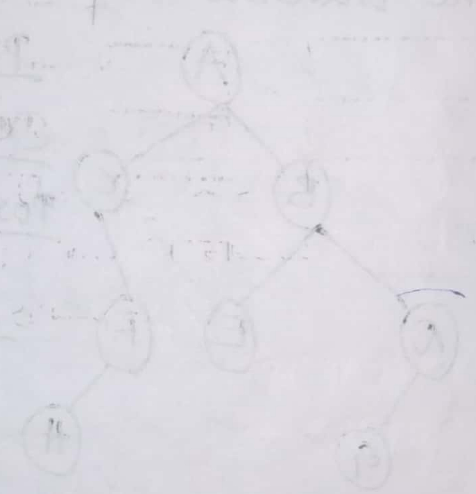
Inorder: HDBEIAMJFKCGL

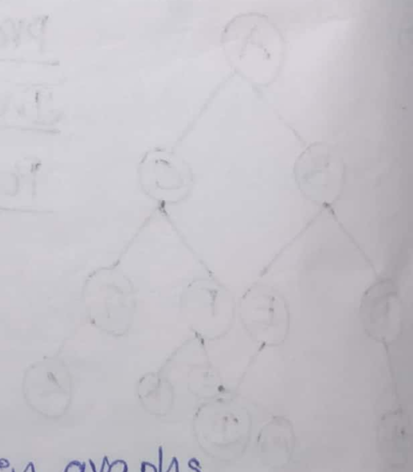
Postorder: HDIEBMJKFLGCA

Algorithm:

```
treenode = record;
{
type data;
treenode *lchild;
```

```
treenode *rchild;
}
Algorithm Inorder (t)
{                    -> tree is not empty
 if t≠0 then
 {
Inorder (t->lchild);
 Visit (t);
Inorder (t->rchild);
 }
}
Algorithm Preorder (t)
{
 if t≠0 then
{
  Visit (t);
  preorder (t->lchild);
  preorder (t->rchild);
   }
}
Algorithm postorder (t)
{
 if t≠0 then
{
 postorder (t->lchild);
 postorder (t->rchild);
 Visit (t)
  }
}
```





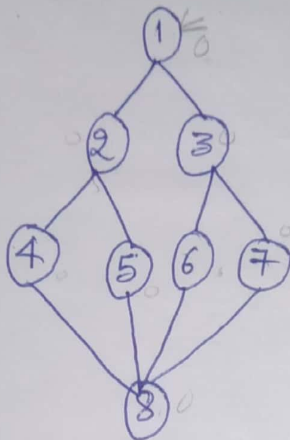
Techniques for Graphs:-

There are two traversal techniques in graphs

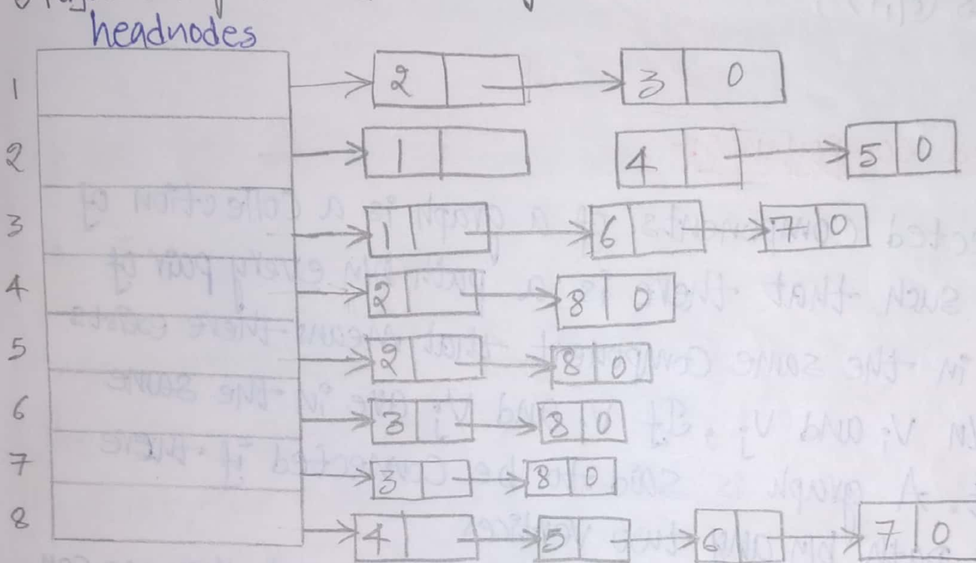
1) Breadth First search
2) Depth First search

Scanned with CamScanner

1) Breadth First search:

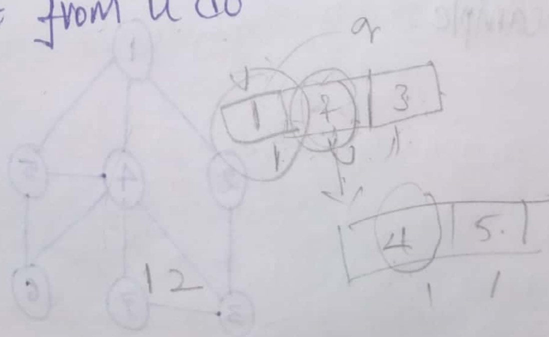


Adjacency list for the graph:



Algorithm:

```
Algorithm BFS(V)
{
  u:=V;
  Visited[u]:=1;
repeat
{
for all Vertices w adjacent from u do
  {
  if (Visited[w]=0) then
       Visited[w]:=1;
  }
}
if queue is empty then
     return
else
delete the next element u from queue;
```



Scanned with CamScanner

} until (false);
}

```
Algorithm BFS (G,n)
{
for i:=1 to n do
    Visited[i]:=0;
for i:=1 to n do
if (Visited[i]=0) then
    BFS (G,i);
}
```

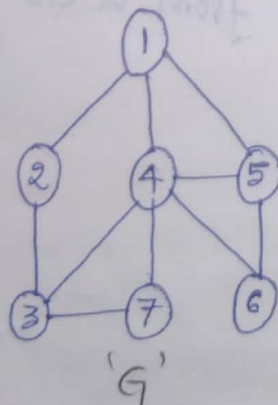
Connected components:-

The Connected Components of a graph is a collection of Vertices such that there is a path b/n every pair of Vertices in the same Component that means there exists a path b/n V_i and V_j, If V_i and V_j are in the same Component. A graph is said to be connected if there exists a path b/n any two vertices.

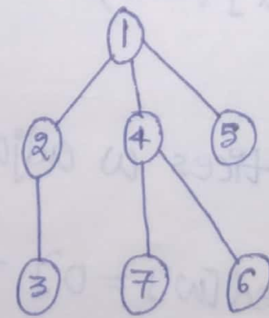
If the graph G is connected undirected graph then we can visit all the vertices of the graph in first call to BFS or DFS.

The subgraph which we obtain after traversing the graph using BFS or DFS represents the connected component of the graph.

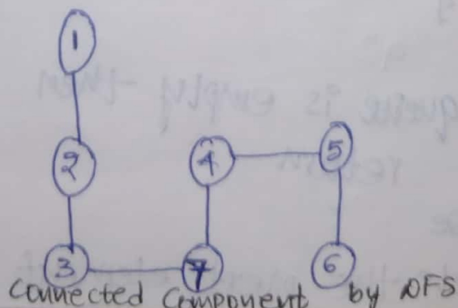
Example:



'G'



Connected Component by BFS



Connected Component by DFS

Algorithm:

```
Algorithm ConnComponents (G,n)
{                                         -> O(n+E)
 for (i<-1 to n) do
 {
   visited [i] <- 0;
 }
for (i<-1 to n) do
 {
 if (visited [i] = 0) then
     DFS (i);
Output the newly visited vertices with adjacent edges
   }
}
```

Time complexity:

The time complexity of the above algorithm is O(n+E). In that, the total time taken by DFS is O (E) and the for loop which DFS is called takes O(n) time.

Strongly Connected Components:

Finding strongly connected components of a directed graph is one of the application of DFS. In undirected graph, two vertices are connected if they have path connecting them. But in case of directed graph vertex A is strongly connected to B if there exists path from A to B & B to A. The relation b/n the vertices should satisfy the following properties:

i) Reflexive:

Any vertex is strongly connected to itself

ii) Symmetric:

If any vertex 'u' is connected to vertex 'v', then vertex v is connected to u then those two vertices satisfy the symmetric property.

iii) transitive:
For a strongly connected component if vertex 'U' is connected to vertex 'V', V is connected to 'W' with the path U-V, V-U, V-W, W-V. This holds the transitive property U=V=W.

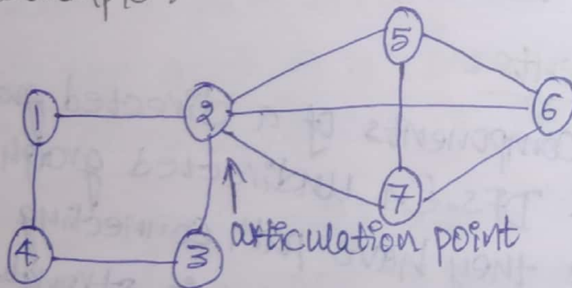
Biconnected Components and DFS:-

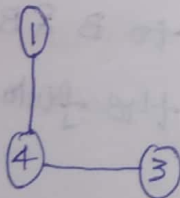
In this section, we study two concepts
1) Articulation point and the other is
2) Biconnected components

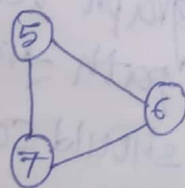
DFS technique is used to find articulation point and biconnected components.

Definition of articulation point:
Let G=(V,E') be a connected undirected graph then an articulation point of graph 'G' is a vertex whose removal disjoints the graph 'G'. The articulation point is also called cut vertex.

Example:

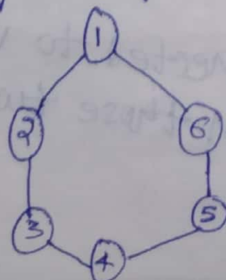






A graph 'G' is said to be biconnected if it contains no articulation point. That means if we remove any single vertex we do not get any disjoint graphs.

Example:



Scanned with CamScanner

Identification of articulation Points:-

i) The easiest Method is to remove a Vertex and its Corresponding edges one by one from graph 'G' and test whether the resulting graph is disconnected or not. The time complexity for this is O(V+E)

2) Another Method is using the DFS technique to find the articulation points.

Procedure:

First we need to apply the DFS technique to the graph 'G'. After performing DFS onto the given graph we get a DFS tree

2) While building the DFS tree, number is alloted for each Vertex. These numbers indicate the order in which a DFS visits the Vertices. These numbers are called Depth First Search Numbers (DFN)

3) While building the DFS tree we can classify the edges into 4 categories

i) tree edge - it is an edge in DFS tree

ii) Back edge - it is an edge U,V which is not in DFS-tree and V is ancestor of U. It indicates a loop

iii) Forward edge - An edge U,V which is not in tree and U is an ancestor of V

iv) Cross edge - An edge U,V is not in DFS tree and V is neither an ancestor nor a descendant of U

To identify the articulation point following observations Can be made

i) The root of the DFS tree is an articulation point if it has two (or) more children.

ii) A leaf node of DFS tree is not an articulation tree

iii) if u is an internal node then it is not an articulation

point if and only if from every child ŵ(u) is possible to reach an ancestor of 'u' using a path made up of descendence of w and back edge.

The equation to find the articulation point is as follows

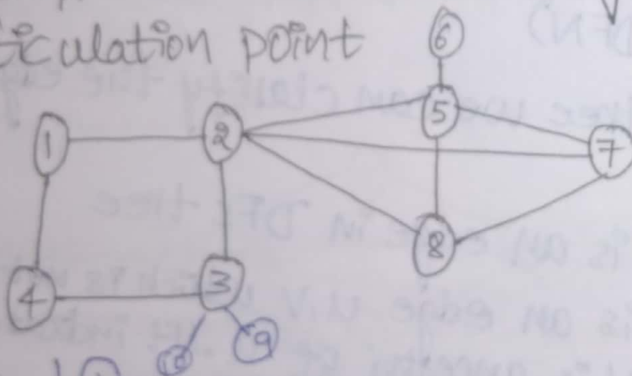
$$Low[u] = \min \{dfn[u], \min\{Low[w] / w \text{ is a child of } u\}, \min\{dfn[w] / (u,w) \text{ is a back edge}\}\}$$

where Low[u] is the lowest depth first number that can be reached from 'u' using a path of descendence followed by atmost one package.

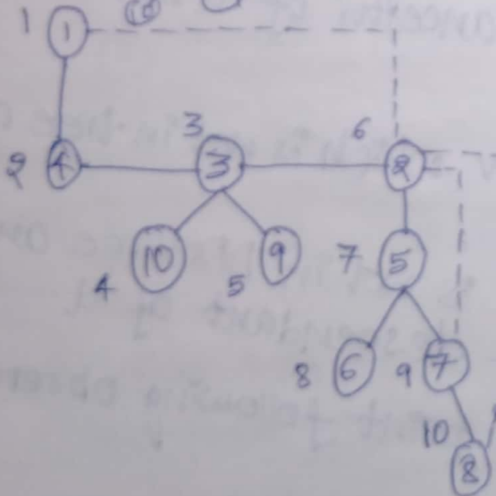
The vertex 'u' is an articulation point when the following condition is satisfied

$$Low[w] \geq dfn[u]$$

Example: Consider the below graph and identify the articulation point



Sol:



Let us compute Low[u] using the formula

$$Low[u] = \min \{dfn[u], \min\{Low[w] / w \text{ is a child of } u\}, \min\{dfn[w] / (u,w) \text{ is a back edge}\}\}$$

Scanned with CamScanner

Low[1] = min{dfn[1], min{Low[4]}, min{dfn[2]}}
= min{1, Low[4], 6}
= 1

Low[2] = min{dfn[2], min{Low[5]}, min{dfn[1], dfn[7], dfn[8]}}
= min{6, Low[5], min{1, 9, 10}}
= min{6, Low[5], 1}
= 1

Low[3] = min{dfn[3], min{Low[10], Low[9], Low[2]}, no back edge}
= min{3, min{Low[10], Low[9], 1}}
= 1

Low[4] = min{dfn[4], min{Low[3]}, No backedge}
= min{2, 1}
= 1

Low[5] = ~~min~~ {~~dfn~~[5], min{~~Low[7]~~}, ~~min{dfn[2]}~~}
= ~~min~~ {7, ~~min~~{Low[7], 6}
= min{dfn[5], min{Low[6], Low[7]}, no back edge}
~~= min 7~~

~~Low[6] = min{dfn[6]}, min no child, no back edge}~~
~~=~~

Low[5] will not be decided until calculating the Low[6] & Low[7]
∴ Low[5] keep it as it is.

Now Low[6] = min{dfn[6]}
= 8

Low[7] = min{dfn[7], min{Low[8]}, min{dfn[2]}}
= min{9, Low[8], 6} = min{9, 6, 6} = 6

Low[8] = min{dfn[8], –, min{dfn[2]}}
= min{10, 6}
= 6

Low[5] = min{7, min{8, 6}}
= min{7, 6}
= 6

$low[9] = min\{dfn[9]\} = 5$

$low[10] = min\{dfn[10]\} = 4$

The low values are $low[1:10] = \{1,1,1,1,6,8,6,6,5,4\}$

Now by checking the condition $low[w] \geq dfn[u]$. We need to identify the articulation point. → {1,2,3,5}

Vertex '2' is an articulation point because child of '2' is '5' $low[5] = 6$ and $dfn[2] = 6$.

Similarly vertex '3' is also an articulation point because children of '3' is 10 and 9

$low[10] = 4$

$low[9] = 5$

$dfn[3] = 3$

Vertex '5' is also an articulation point, the children of '5' are 6 & 7 $low[6] = 8$ and $low[7] = 6$, $dfn[5] = 7$

8>7 ✓ 6>7 ✗ If any one child satisfies the condition then it is A.P

∴ 2, 3 & 5 are the articulation points

Algorithm:

```
Algorithm DFS_Art(u,v)
{
 dfn[u] := dfn-num;
 low[u] := dfn-num;
 def-num := dfn-num+1;
 for (each vertex w adjacent to u) do
 {
  if (dfn[w] = 0) then
  {
   DFS_Art(w,u);
  low[u] := min {low[u], low[w]};
  }
 else
  if (w' = u) then
 low[u] := min{low[u], dfn[w]};
   }
 }
```

Scanned with CamScanner

Identification of Biconnected Components:

A Biconnected graph G=(V,E) is a connected graph which don't have articulation points. A biconnected component of a Graph G is a maximal biconnected subgraph that means it is not contained in any larger biconnected subgraph of G

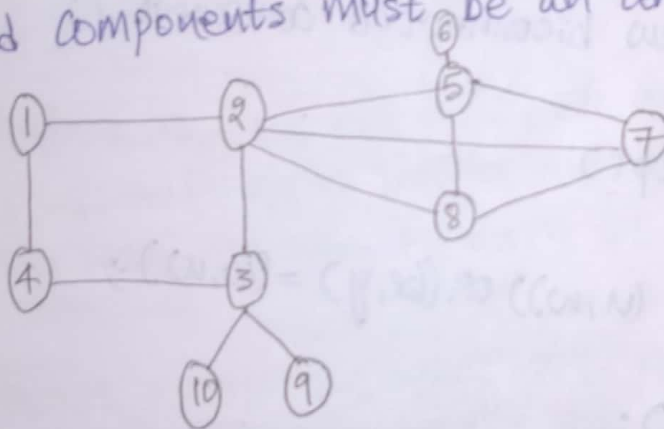
Some key observations can be made in regard to biconnected components of graph are

1) Two different biconnected components should not have any common edges

2) Two different biconnected components can have common vertex.

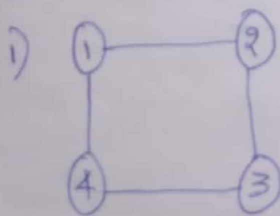
3) A common vertex which is attaching two (or) more biconnected components must be an articulation point.

Example:

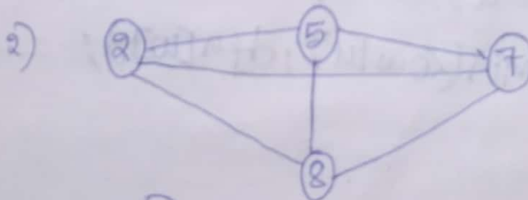


Here the articulation points are 2, 3 & 5. Based on these points the biconnected components are:

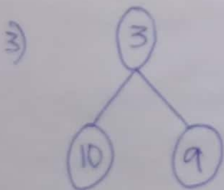
1)



2)



3)



4)

Scanned with CamScanner

```
[illegible]m Biconnect(u,v)

dfn[u] ← dfn-num;
low[u] ← dfn-num;
dfn-num ← dfn-num+1;
for (each vertex w adjacent to u) do
{
  if ((v!=w) and (dfn[w] < dfn[u])) then
    push (u,w) on to the stack st;
  if (dfn[w] = 0) then
  {
  if (low[w] ≥ dfn[u]) then
   {
write("obtained Articulation points");
write("The new biconnected component");
repeat
{
edge (x,y) ← pop()
write (x,y);
} until ((x,y) = (u,w)) OR ((x,y) = (w,u));
 }
 Biconnect(w,u);
low[u] ← min(low[u], low[w]);
 }
else if (w!=u) then
low[u] ← min(low[u], dfn[w]);
  }
}
```