DAA

Internal 1 Notes

Credit : Krishna Keerthi Mam

(exp : 12 years +)

OMSAIRAM

Design And Analysis of Algorithms

UNIT-I

Algorithm:-

- It is a step by step procedure to solve particular problem or task.
- Algorithm is a finite set of instructions which are used to solve a particular problem or a task
- Algorithm can be defined as an ordered sequence of well defined and effective operations that when executed will always produce a result and eventually terminate in a finite amount of time.
- All algorithms must satisfy the following properties

(1) Input:-
Each algorithm must take zero (DONE) or more quantities are externally supplied (as input)

(2) Output:-
Atleast one quantity is produced

(3) Definiteness:-
Each instruction is clear & unambiguous
Eg add 6 or 7 & 2 & compute 5/0 are not definite

(4) Finiteness:-
If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps (operations)

(5) Effectiveness :-

Every instruction must be effective i.e., every operation should be done roughly by pen & paper. i.e tracing of each step should be possible.

- Algorithms that are definite & effective are also called computational procedures

Eg operating system of a digital computer.

This procedure is designed to control the execution of jobs if no jobs are available, it doesn't terminate but continues in waiting state until a new job is entered.

Algorithm Specification

Pseudocode Conventions :-

- Comments //
- Blocks are indicated with matching braces { & }
- Statements are delimited by ;
- Assignment of values to variable is done using the assignment operator

  <variable> := <expression>;
- Boolean values true & false
  logical operators and, or & not
  relational operators <, ≤, =, ≠, ≥ & >
- Elements of multidimensional arrays [&].

  Array A[i, j]. Array indices start at zero

- while loop

```
while <condition> do
{
    <statement 1>
    :
    <statement n>
}
```

```
for variable := value1 to value2 step stepdo
{
    stmt1
    :
    stmtn
}
```

step → +ve (incr) -ve

value1, value2, step are arithmeticExp.

A repeat-until stmt is

```
repeat
    <stmt 1>
    :
    <stmt n>
until <condition>
```

The statements are executed as long as condition is false.

- Break → results in the exit of the innermost loop
- Return → results in the exit of the function itself

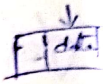
- If <condition> then <stmt>
- If <condition> then <stmt1> else <stmt2>
- Case statement
- case
{
    : <condition 1> : <statement 1>
    :
    ; <condition n> : <statement n>
    : else : <stmt n+1>
}

⇒ Algorithm Selection Sort (a,n)

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

    t := a[i]; a[i] := a[j]; a[j] := t;
  }
}
```

⇒ Routine to insert an element into a Binary Search Tree.



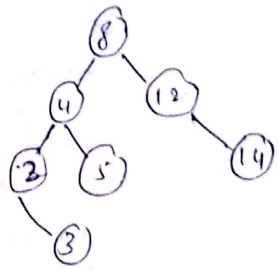

```
insert (struct btree node *T, int x)
{
  if (T == NULL)
  {
    T = malloc (sizeof (structure btree node));
    T -> data = x
    T -> left = T -> right = NULL
  }
  else
  if (x < T -> data)
    T -> left = insert (T -> left, x);
  else
  if (x > T -> data)
    T -> right = insert (T -> right, x);
  else
  printf ("Insertion is not possible");

  return T;
}
```

```
struct btree no...
{
  int data;
  struct btree...
  struct btree...
};
```

Routine to deletion an element into B.S.T

```
delete (struct btree *T, int x)
{
  if(T == NULL)
  {
   Printf ("Deletion is not possible");
  }
  else
   if (x < T -> data)
   T -> left = delete (T->left, x);
   else
    if(x > T -> data)
     T -> right = delete (T -> right, x);
    else
     if( T -> left and T -> right != NULL)
      {
        temp = findmin(T -> right);
         T -> data = temp -> data;
         T -> right = delete (T -> data, T -> right);
      }
     else
      { temp = T;
       if(T -> left == NULL)
        child = T -> right;
         if(T -> right == NULL)
         child = T -> left;
          free(temp);
          return T;
       }
}
/* Routine for finding min. element */
 findmin(struct btree *T)
 { if(T != NULL)
    while (T -> left != NULL)
      T = T -> left;
    return T;
 }
```



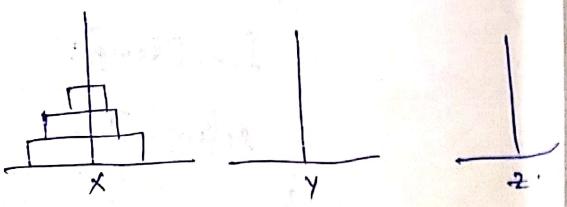
Towers of Hanoi

```
//gobal variable, Tower[1:3] are the three towers
array Stack<int> Tower[4];
void moveAndShow(int, int, int, int);

void TowersOfHanoi(int n)
{
   for(int d = n; d>0; d--)
      Tower[1].push(d); // add disk d to to..
   //move n disks from tower 1 to 3 using as i
   moveAndShow(n, 1, 2, 3);
}

void moveAndShow(int n, int x, int y, int z)
{ //move top n disks from tower x to tower y showi
   if(n>0)
   {
      moveAndShow(n-1, x, z, y)
      int d = Tower[x].top()  //move a disk from top
      tower[x].pop()          // tower x to top of
      tower[y].push(d)        // tower y
      showState();            //show state of 3 towers
      moveAndShow(n-1, z, y, x);
   }
}
```

Recursive




```
TowersOfHanoi(n, x, y, z)
{ //move top n disks from tower x to tower y.
   if(n>=1) then
   { TowersOfHanoi(n-1, x, z, y);
      write("move top disk from tower ", x, " to top of tower
      TowersOfHanoi(n-1, z, y, x);
   }
}
```

Performance Analysis

- Space complexity of an algorithm is the amount of memory it needs to run to completion.
- Time complexity of an algorithm is the amount of computer time it needs to run to completion.
- Algorithm analysis refers to the task of determining time & space requirements of an algorithm. It is also called performance analysis which leads us to select an efficient algorithm.
- If an algorithm to be analyse the first task is which operations are implied & determine what their relative costs.
  Second task is to determine the sufficient no. of data sets which causes an algorithm to exhibit all possible patterns of behaviour.

Space Complexity :-

- The space needed by algorithms is seen by the sum of following components.

(1) A fixed part that is independent of the characteristics of the inputs & outputs. This part includes the instruction space (i.e space for code), space for simple variables and fixed-size components variables, space for constants, & soon.

(2) A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved.

The space requirement $S(P)$ of any algorithm P may therefore be written as $S(P) = c + Sp$,

where c is a constant.

Time Complexity

The time $T(P)$ taken by a program P is the sum of the compile time and the run time

The compile time does not depend on the instance characteristics

Runtime is denoted by tp

$$tp(n) = Ca\ ADD(n) + Cs\ SUB(n) + Cm\ MUL(n) + Cd\ DIV(n)$$

$n \rightarrow$ instance characteristics

Ca, Cs, Cm, Cd indicates time require to perform addition, substraction, mult, div.

Asymptotic Notations

- It is a step by step process for solving a task in finite amount of time

Let f & g are non negative functions

(1) Big 'oh' notation (O):-

The function $f(n) = O(g(n))$ (read as f of n is big oh of) iff there exist positive constants c and n_0 such that

$f(n) \leq c * g(n)$ for all n, $n \geq n_0$

Notation provides $g(n)$ is upper bound on value of the function $f(n)$ $\forall\ n, n \geq n_0$

Eg (1) $3n+2 = O(n)$ as $3n+2 \leq 4n \quad \forall \quad n \geq 2$

$n=1 \quad 5 \leq 4$

$n=2 \quad 6+2 \leq 8$

$n=3 \quad 11 \leq 12$

$f(n) \leq cg(n)$

$c=4 \quad g(n) = n$

$f(n) = 3n+2$

(2) $10n^2 + 4n + 2 \leq 11n^2 \quad \forall \quad n \geq 5$

$272 \leq 275$

time complexity $= O(n^2)$

$10n^2 + 4n + n = O(n^2)$

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

- $O(1) \rightarrow$ computing time that is constant
- $O(n) \rightarrow$ linear
- $O(n^3) \rightarrow$ cubic
- $O(2^n) \rightarrow$ exponential
- If algorithm takes time $O(\log n)$, it is faster for sufficiently large n, than if it had taken $O(n)$
- $O(n \log n)$ is better than $O(n^2)$ but not as good as $O(n)$

(2) Big Omega (Ω)

The function $f(n) = \Omega(g(n))$ iff there exist positive constants c & n_0 such that $f(n) \geq c * g(n)$

$\forall\ n, n \geq n_0$

Eg $3n+2 \geq 3n$ for $n \geq 1$

$n=1 \quad 5 \geq 3$

$n=2 \quad 8 > 6$

$3n+2 = \Omega(n)$

$10n^2+4n+2 \geq n^2$, n

$10+4+2 \geq 1$

$10n^2+4n+2 = \Omega(n^2)$

$6 * 2^n + n^2 \geq 2^n$ for $n \geq 1$

$12+1 \geq 2$

$6 * 2^n + n^2 = \Omega(2^n)$

The function $g(n)$ is only a lower bound on $f(n)$

(3) Theta (Big-Theta notation Θ)

The function $f(n) = \Theta(g(n))$ iff there exist positive constants c_1, c_2 and n_0 such that

$c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall\ n, n \geq n_0$

Eg $3n+2 \geq 3n \quad \forall\ n \geq 2$ & $3n+2 \leq 4n \quad \forall\ n \geq$

$f(n) = 3n+2$

$c_1 = 3 \quad c_2 = 4 \quad n_0 = 2$

$g(n) = n$

$3n \leq 3n+2 \leq 4n$

$n=2 \quad 6 \leq 8 \leq 8$,

$\therefore 3n+3 = \Theta(n)$

The function $f(n) = \Theta(g(n))$ iff $g(n)$ is both an upper & lower bound on $f(n)$

(4) Little "oh" :- (o)

The function $f(n) = o(g(n))$ iff $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

(5) Little omega notation (ω):-

The function $f(n) = \omega(g(n))$ iff $\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$

Eg :- Magic Square :-

A magic square is an $n \times n$ matrix of the integers 1 to n^2 such that the sum of every row, column & diagonal is same.

Eg case $n = 5$, sum $= 65$,

	0	1	2	3	4
0	15	8	1	24	17
1	16	14	7	5	23
2	22	20	13	6	4
3	3	21	19	12	10
4	9	2	25	18	11

- Start with 1 in the middle of the top row, then go up & left, assigning numbers in increasing order to empty squares

Algorithm Add (a, b, c, m, n)

```
{
    for i := 1 to m do
    {   count := count+1; // for for i
        for j := 1 to n do
        {   count := count+1; // for for j
            c[i,j] := a[i,j] + b[i,j];
            count := count+1; // for assignment
        }
        count := count+1; // last time of for j
                          (closing of loop alwoperation)
    }
    count := count+1; // last time of for i
}
```

$m + mn + mn + m + 1$

$2mn + 2m + 1$

```
Algorithm Sum(a,n)
{
    s:=0.0;
    count:=count+1;
    for i:=1 to n do
    {
        count:=count+1;   // fa for loop
        s:=s+a[i];
        count:=count+1;   // fa assignment
    }
    count:=count+1;   // last time of for
    return s;  count:=count+1;   // fa return
}
```

$1+n+n+1+1 = \boxed{2n+3}$

```
Algorithm Rsum(a,n)
{
    if (n<=0) then
    {
        return 0.0;
    }
    else
        return Rsum(a,n-1)+a[n];
}
```

Count Method:

```
Algorithm Rsum(a,n)
{
    if(n<=0) then
    {
        count:=count+1; // for if        Counts as 1
        return 0.0;
        count:=count+1; // for return    Counts as 1
                                         1+1=2
    }
    else
    {
        return Rsum(a,n-1)+a[n];
        count:=count+1; // for addition
    }
}
```

for else condition also it has to check if condition once so '1' {+a[n] is '1'}

$$t_{RSum}(n) = \begin{cases} 2 & \text{if } n=0 \\ 2+t_{RSum}(n-1) & \text{if } n>0 \end{cases}$$

These recursive formulas are referred as recurrence relations. One way of solving any such recurrence relation is to make repeated substitutions for each occurrence of the function t_{RSum}, until all such occurrences disappear.

$$\begin{aligned} t_{RSum}(n) &= 2 + t_{RSum}(n-1) \\ &= 2 + [2 + t_{RSum}(n-2)] \\ &= 2+2 + [2 + t_{RSum}(n-3)] \\ &= n(2) + t_{RSum}(0) \\ &= n(2) + 2 \\ &= \boxed{2n+2} \end{aligned}$$

When the performance is measured for both the iterative & recursive algorithms for sum of n-numbers, iterative algorithm is better than recursive algorithm, since recursive algorithm uses an extra data structure stack.

The time complexities of both the algorithms are almost same.

// Compute the nth Fibonacci number

Algorithm Fibonacci(n)

```
{
    if(n<=1) then  count := count+1; // for if
        write(n);  count := count+1; // for write
    else
    {
        fnm2 := 0;  fnm1 := 1;  count := count+1;
        count := count+1;
        for i := 2 to n do
                    count := count+1; // for i loop
        {   fn := fnm1 + fnm2;
            count := count+1;
            fnm2 := fnm1;
            count := count+1;
            fnm1 := fn;
            count := count+1;
        }
        count := count+1; // last time of for
        write(fn);
        count := count+1;
    }
}
```

when $n \leq 1$, $T_{fibo}(n) = \boxed{2}$

when $n \geq 2$ ~~$T_{fibo}(n)$~~

$T_{Fibo}(n) = 1+1+1+n-1+n-1+n-1+n-1+1+1$

$= \boxed{4n+1}$

Statement	s/e	Statement executed frequency n=1	n>1	Total steps n=1	n>1
Algorithm Fibonacci(n)	0	—	—	0	0
{	0	—	—	0	0
if(n≤1) then	1	1	1	1	1
write (n);	1	1	0	1	0
else	0	0	1	0	1
{	0	0	1	0	1
fnm2 := 0;	1	0	1	0	1
fnm1 := 1;	1	0	1	0	1
for i := 2 to n do	1	0	n (n-1+1) (total no. of executions)	0	n
{	0	0	n-1	0	n-1
fn := fnm1 + fnm2;	1	0	n-1	0	n-1
fnm2 := fnm1;	1	0	n-1	0	n-1
fnm1 := fn;	1	0	n-1	0	n-1
}	0	0	1	0	1
write(fn);	1	0	1	0	1
}	0	—	—	0	0
		—	—	0	0
				2	4n+1

}

Tabular Method

Statement	s/e	Frequency	Total steps
Algorithm Sum(a,n)	0	0	0
{	0	0	0
s := 0.0	1	1	1
for i := 1 to n do	1	n+1 (total), n (tail car)	n+1
s := s+i	1	n	n
return s;	1	1	1
}	0	0	0
			2n+3

Statement	s/e	Frequency	Total steps
Algorithm Add(a,b,c,m,n)	0	0	0
{	0	0	0
for i := 1 to m do	1	m+1	m+1
for j := 1 to n do	1	m(n+1)	m(n+1)
c[i,j] := a[i,j] + b[i,j]	1	mn	mn
}	0	0	0
			2mn+2m+1

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

$x = t_{RSum}(n-1)$

```
{
    return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

Sp = 0

Algorithm Sum(a, n)

```
{
    s := 0.0;
    for i := 1 to n do
        s := s+a[i];
    return s;
}
```

$S_{Sum}(n) \geq (n+3)$ [n for a[], one each for n, s and i]

Algorithm RSum(a, n)

```
{   if (n≤0) then return 0.0;
    else
    Return RSum(a, n-1)+a[n];
}
```

$S_{Sum}(n) \geq 3(n+1)$

[Space for n, return address, pointer to a[]]

Asymptotic Notation: Used to make meaningful statement about the time (complexity) of an algorithm

f & g are non-negative functions

Big 'oh' (O): The function $f(n) = O(g(n))$ iff there exists positive constants c and no such that $f(n) \leq c * g(n)$ for all n, $n \geq n_0$

Eg: 1

$3n+2 \leq 4n$

when $n=1$

$3(1)+2 \leq 4(1)$

$5 \leq 4$ false

when $n=2$

$3(2)+2 \leq 4(2)$

$8 \leq 8$ true

$\therefore 3n+2 = O(n)$ $\forall$ $n \geq 2$

Eg: 2

$3n+3 \leq 4n$

when $n \geq 3$

$3(3)+3 \leq 4(3)$

$12 \leq 12$

$\therefore 3n+3 = O(n)$ $\forall$ $n \geq 3$

Eg: 3. $100n + 6 \leq 101n$

when $n = 6$

$100(6) + 6 \leq 101(6)$

$606 \leq 606$

$\therefore \quad 100n + 6 = O(n) \quad \forall \quad n \geq 6$

Eg. 4 $10n^2 + 4n + 2 \leq 11n^2$

when $n \geq 5$

$10(5)^2 + 4(5) + 2 \leq 11(5)^2$

$250 + 20 + 2 \leq 275$

$272 \leq 275$

$\therefore \quad 10n^2 + 4n + 2 = O(n^2) \quad \forall \quad n \geq 5$

Eg. 5 $6 * 2^n + n^2 \leq 7 * 2^n$

when $n = 4$

$6 * 2^4 + 4^2 \leq 7 * 2^4$

$6 * 16 + 16 \leq 7 * 16$

$112 \leq 112$

$\therefore \quad 6 * 2^n + n^2 = O(2^n)$

$O(1)$ means computing time is constant

$O(n)$ is called linear

$O(n^2)$ is quadratic

$O(n^3)$ is cubic

$O(2^n)$ is exponential

$O(n \log n)$ is faster for sufficiently large n, than $O(n)$.

$O(n \log n)$ is better than $O(n^2)$ but not as good as $O(n)$.

$f(n) = O(g(n))$ states that $g(n)$ is an upper bound on the value of $f(n)$ $\forall$ n, $n \geq n_0$.

there exists positive constants c and n_0 such that

$f(n) \geq c * g(n)$ for all n; $n \geq n_0$.

Eg: 1 $3n+2 \geq 3n$

when $n=1$

$3(1)+2 \geq 3(1)$

$5 \geq 3$

$\therefore 3n+2 = \Omega(n) \quad \forall \quad n \geq 1$

Eg: 2 $10n^2+4n+2 \geq 10n^2$

when $n=1$

$10(1)^2+4(1)+2 \geq 10(1)^2$

$16 \geq 10$

$\therefore 10n^2+4n+2 \geq \Omega(n^2) \quad \forall \quad n \geq 1$

Eg 3: $6 * 2^n + n^2 \geq 6 * 2^n$

when $n=1$

$6 * 2^1 + (1)^2 \geq 6 * 2^1$

$13 \geq 12$

$\therefore 6 * 2^n + n^2 \geq \Omega(2^n) \quad \forall \quad n \geq 1$

The function $g(n)$ is a lower bound on $f(n)$

Theta (Θ): The function $f(n) = \Theta(g(n))$ iff there exist positive constants c_1, c_2 and n_0 such that

$c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n, n \geq n_0$

$3n+2 = \Theta(n)$ as

$3n+2 \leq 4n$

$n=2$

$3(2)+2 \leq 4(2)$

$8 \leq 8$

$3n+2 \geq 3n$

$n=2$

$3(2)+2 \geq 3(2)$

$8 \geq 6$

$\therefore 3n+2 = \Theta(n)$ for $n \geq 2$

$c_1 = 3$, $c_2 = 4$ and $n_0 = 2$

The function $f(n) = \Theta(g(n))$ is both an upper and lower bound on $f(n)$.

Little oh (o): The function $f(n) = o(g(n))$ iff

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

Eg: $3n+2 = o(n^2)$

$$\lim_{n \to \infty} \frac{3n+2}{n^2} = 0$$

Little omega (ω): The function $f(n) = \omega(g(n))$ iff

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

Randomized Algorithms

Basics of probability Theory :- probability theory has the goal of characterizing the outcomes of natural (or) conceptual experiments. Each possible outcome of an experiment is called a sample point & the set of all possible outcomes is known as the sample space S.

An event E is a subset of the sample space S.

If the sample space consists of n sample points, there are 2^n possible events.

The probability of sample space S is 1.

Eg 1: Tossing a coin: When a coin is tossed, there are two possible outcomes head (H) and tail (T).

Consider the experiment of tossing three coins. There are eight possible outcomes.

{ HHH, HTT, HTH, TTT, HHT, THT, TTH, THH }

For this experiment there are 2^8 possible events.

The probability of an event E is defined to be

$\frac{|E|}{|S|}$, where S is the sample space

Eg 2 The probability of the event { HHT, HTT, TTT } is

$$\frac{3}{2^3} = \frac{3}{8}.$$

Rolling two dice, there are 6^2 i.e 36 possible outcomes

What is the probability that the sum of two faces is 10.

$E = \{(4,6), (5,5), (6,4)\}$

$$\frac{|E|}{|S|} = \frac{3}{36} = \frac{1}{12}$$

Mutual Exclusion: Two events E_1 and E_2 are said to be mutually exclusive if they do not have any common sample points, i.e; $E_1 \cap E_2 = \phi$.

Eg: When we toss three coins, let E_1 be the event that there are two H's and let E_2 be the event that there are at least two T's. These two events are mutually exclusive since there are no common sample points.

Theorem:

1. $\text{Prob}[\bar{E}] = 1 - \text{prob}[E]$
2. $\text{Prob.}[E_1 \cup E_2] = \text{Prob}[E_1] + \text{Prob}[E_2] - \text{Prob}[E_1 \cap E_2]$
   $\leq \text{Prob}[E_1] + \text{Prob}[E_2]$

Conditional Probability: Let E_1 and E_2 [illegible] events of an experiment. The conditional probability of E_1 given E_2, denoted by

$$\text{Prob}[E_1 | E_2] = \frac{\text{Prob}[E_1 \cap E_2]}{\text{Prob}[E_2]}$$

Independence: Two events E_1 and E_2 are said to be independent if $\text{Prob}[E_1 \cap E_2] = \text{Prob}[E_1] * \text{Prob}[E_2]$

Random variable: Let S be the sample space of an experiment. A random variable on S is a function that maps the elements of S to the set of real numbers. For any sample point $s \in S$, $X(s)$ denotes the image of s under this mapping. If the range of X, i.e; the set of values X can take is finite, we say X is discrete.

Eg: We flip a coin four times. The sample space consists of 2^4 sample points. We can define a random variable X on S as the number of heads in the coin flips

Then $X(HHHH) = 4$, $X(HHHT) = 3$ & so on.

The possible values that X can take are $0, 1, 2, 3, 4$.

Thus X is discrete.

$$\text{Prob}[X = 0] = \frac{1}{2^4} = \frac{1}{16}$$

the expected value (or) mean of any random variable

$$X = \sum_{i=1}^{n} \text{Prob}[s_i] * X[s_i] = \frac{1}{n} \sum_{i=1}^{n} X(s_i)$$

Eg: The Sample space corresponding to the experiment of tossing three coins is $S = \{HHH, HHT, HTH, THH, THT, TTH, HTT, TTT\}$ If X is the no. of heads in the coin flips, then the expected value of X is $\frac{1}{8}(3+2+2+2+1+1+1+0)$

$$= \frac{12}{8} = \frac{3}{2} = 1.5$$

Probability distribution: Let X be a discrete random variable defined over the sample space S. Let $\{r_1, r_2, \ldots r_m\}$ be its range. Then the probability distribution of X is the sequence $\text{Prob}[X=r_1], \text{Prob}[X=r_2], \ldots \text{Prob}[X=r_m]$. $\sum_{i=1}^{m} \text{Prob}[X=r_i] = 1$.

Eg: If three coin are flipped and X is the number of heads, then X can take on four values 0, 1, 2 and 3. The probability distribution of X is given by $\text{Prob}[X=0] = \frac{1}{8}$, $\text{Prob}[X=1] = \frac{3}{8}$, $\text{Prob}[X=2] = \frac{3}{8}$, and $\text{Prob}[X=3] = \frac{1}{8}$.

Binomial distribution:- Let X be a random variable on S defined to be the number of successes in the n trials. The variable X is said to have a binomial distribution with parameters (n, p). The expected value of X is np. Also,

$$\text{Prob.}[X=i] = (^nC_i)\, p^i (1-p)^{n-i}$$

Randomized Algorithms:-

A randomized algorithm is one that makes use of a randomizer (eg. random number generator) and the decisions made in the algorithm depend upon the output of the randomizer. The output of a randomized algorithm could differ from run to run for the same input. The execution could also vary from run to run for the same input.

Randomized algorithms can be categorized into two classes.

(i) Algorithms that always produce the same output for the same input. These are called Las Vegas algorithms. The execution time of a Las Vegas algorithm depends on the o/p of the randomizer.

run. These are called Monte Carlo algorithms. For a fixed input, there is not much variation in execution time.

Primality Testing

Given an integer n, the problem of deciding whether n is prime is known as primality testing. It has a number of applications including cryptology.

If a number n is composite, it must have a divisor $\leq \sqrt{n}$.

```
Algorithm Prime(n)
{
    for i := 2 to √n
        if (n mod i = 0) then
        { write 'n is not prime';
          return;
        }
    write 'n is prime';
}
```

The naive primality testing algorithm has a run time of $O(\sqrt{n})$.

A Monte Carlo algorithm can be devised for primality testing that runs in time $O((\log n)^2)$. If the input is prime, the algorithm never gives an incorrect answer. However, if the input number is composite, then there is a small probability that the answer may be incorrect. Algorithms of this kind are said to have one-sided error.

PRIORITY QUEUES

Any data structure that supports the operations of search min (& max), insert and delete min (& max) is called a priority queue.

Example: Suppose we are selling the services of a machine. Each user pays a fixed amount per use. However, the time needed by each user is different. We wish to maximize the returns from this machine assuming the machine is not to be kept idle unless no user is available. For this we maintain a priority queue of all persons waiting to use the machine. Whenever the machine becomes available, the user with smallest time requirement is selected. Hence, a priority queue with delete min is selected.

If each user needs the same amount of time on the machine but are willing to pay different amounts for the service, then a priority queue based on the amount of payment can be maintained. This requires a delete max operation.

Max Heap: A max heap is a complete binary tree with the property that the value at each node from left and right subtrees are smaller than the root element. Left and right subtrees are again max heap.

max heap sorts the elements in descending order.

Min Heap: A min heap is a complete binary tree with the property that the value at each node is greater than the root element and left and right subtrees are again min heap.

Min heap sorts the elements in ascending order.

Heap and Sets

Insertion into a Heap

```
Algorithm Insert(a,n)
{
    // Inserts a[n] into the heap which is stored in a[1 : n − 1].
    i := n; item := a[n];
    while ((i > 1) and (a[⌊i/2⌋] < item)) do
    {
        a[i] := a[⌊i/2⌋]; i := ⌊i/2⌋;
    }
    a[i] := item; return true;
}
```

Deletion from a Heap

```
Algorithm Adjust(a, i, n)
// The complete binary trees with roots 2i and 2i + 1 are
// combined with node i to form a heap rooted at i. No
// node has an address greater than n or less than 1.
{
    j := 2i; item := a[i];
    while (j ≤ n) do
    {
        if ((j < n) and (a[j] < a[j + 1])) then j := j + 1;
            // Compare left and right child
            // and let j be the larger child.
        if (item ≥ a[j]) then break;
            // A position for item is found.
        a[⌊j/2⌋] := a[j]; j := 2j;
    }
    a[⌊j/2⌋] := item;
}
```

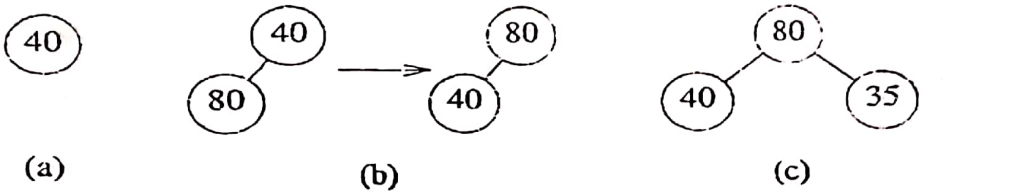

Deletion from a Heap

```
Algorithm DelMax(a, n, x)
// Delete the maximum from the heap a[1 : n] and store it in x.
{
    if (n = 0) then
    {
        write ("heap is empty"); return false;
    }
    x := a[1]; a[1] := a[n];
    Adjust(a, 1, n − 1); return true;
}
```

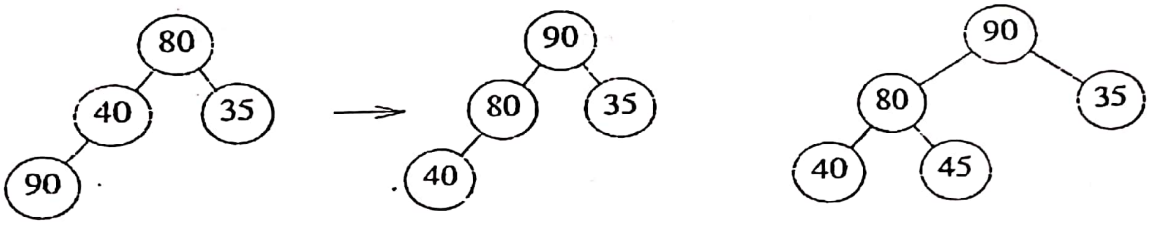
Sorting Algorithm

```
Algorithm Sort(a, n)
// Sort the elements a[1 : n].
{
     for i := 1 to n do Insert(a, i);
     for i := n to 1 step -1 do
     {
          DelMax(a, i, x); a[i] := x;
     }
}
```

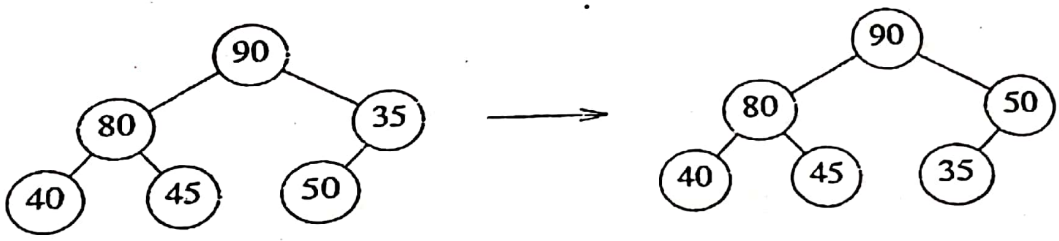
Forming a heap from the set {40, 80, 35, 90, 45, 50, 70}



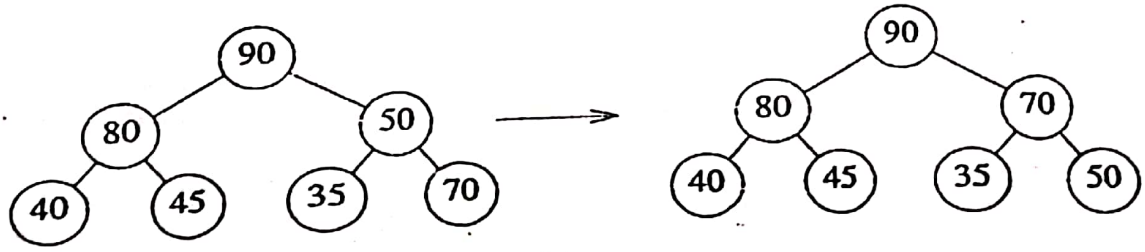
(a) (b) (c)



(d) (e)



(f)

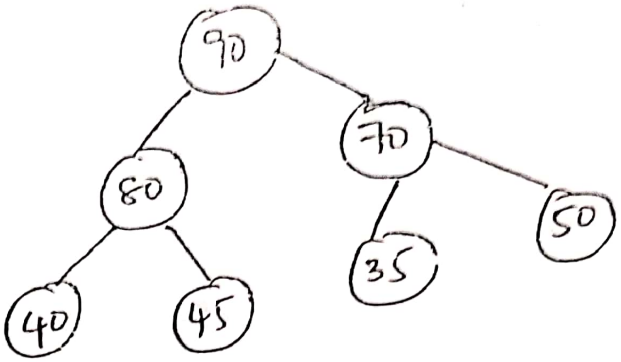


(g)

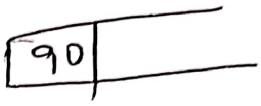
Munawwara Tahaseen

Deletion:

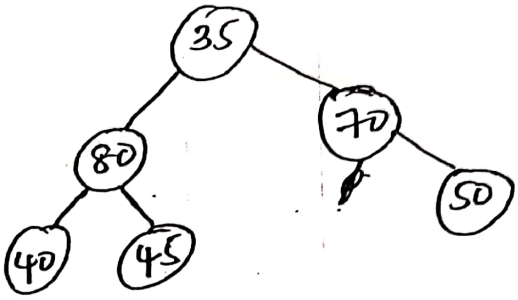
(1)

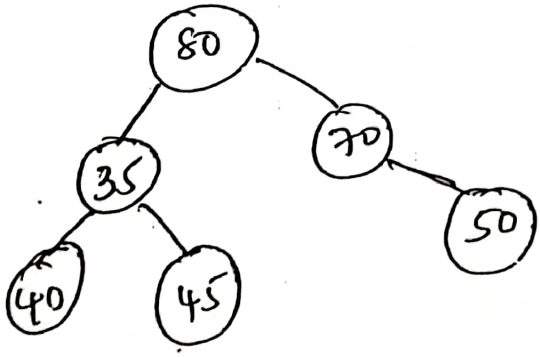


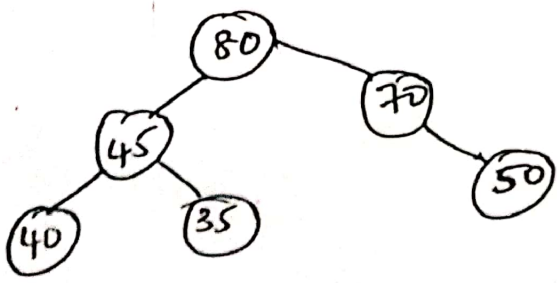
Delete 90



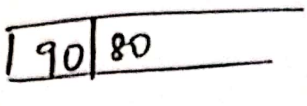
(2)



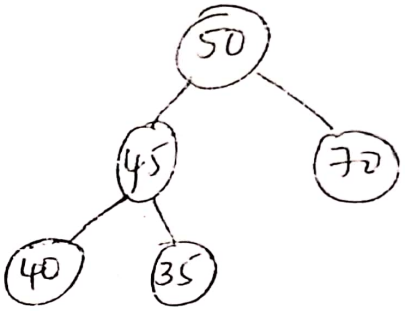


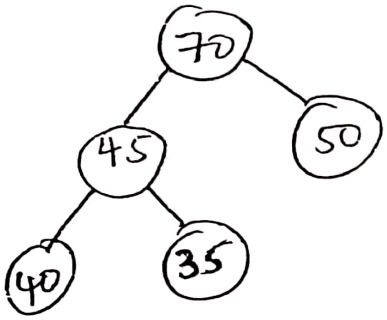


Delete 80



(3)

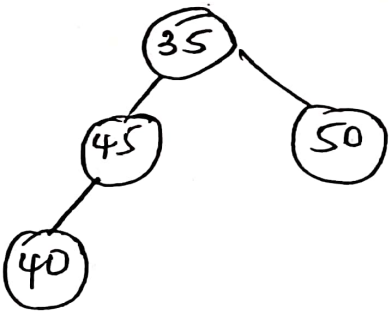


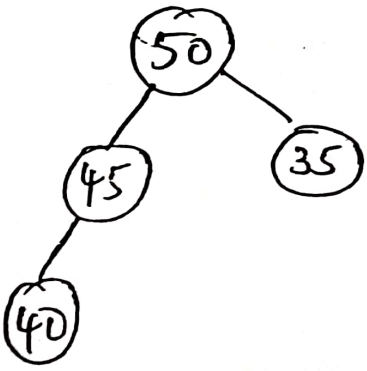


Delete 70

90	80	70

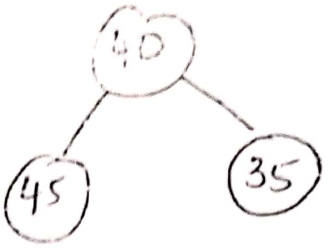
(4)

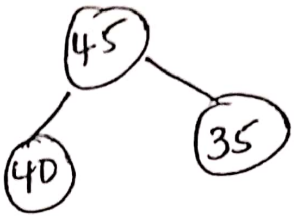




Delete 50

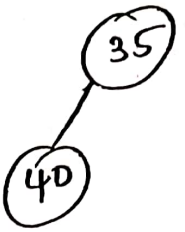
90	80	70	50

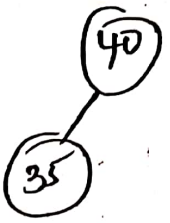




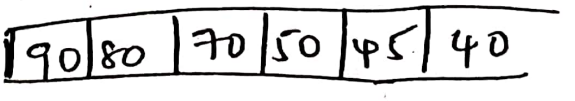
Delete 45

(6)

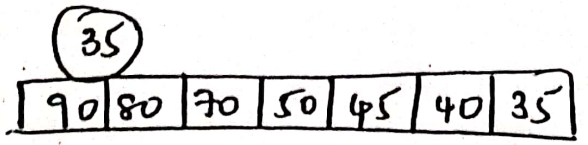




Delete 40



(7)



Time Complexity

- To sort n elements it takes n insertions followed by n deletions from a heap.
- The worst case time complexity of insertion is O(log n)
- The worst case time complexity of deletion is O(log n)
- Hence the worst case time complexity of this sorting algorithm is O(n log n)

Heapify

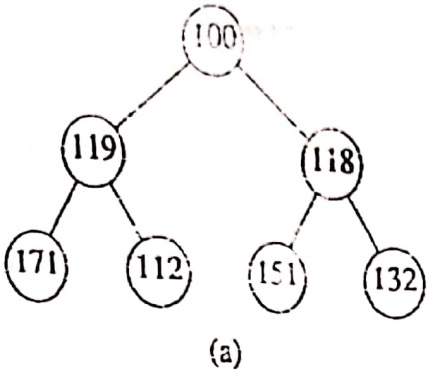
```
Algorithm Heapify(a, n)
// Readjust the elements in a[1 : n] to form a heap.
{
    for i := ⌊n/2⌋ to 1 step −1 do Adjust(a, i, n);
}
```

Heap Sort using Heapify

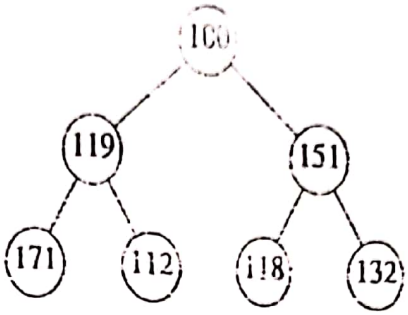
```
Algorithm HeapSort(a, n)
// a[1 : n] contains n elements to be sorted. HeapSort
// rearranges them inplace into nondecreasing order.
{
    Heapify(a, n); // Transform the array into a heap.
    // Interchange the new maximum with the element
    // at the end of the array.
    for i := n to 2 step −1 do
    {
        t := a[i]; a[i] := a[1]; a[1] := t;
        Adjust(a, 1, i − 1);
    }
}
```

Munawwara Tabassum

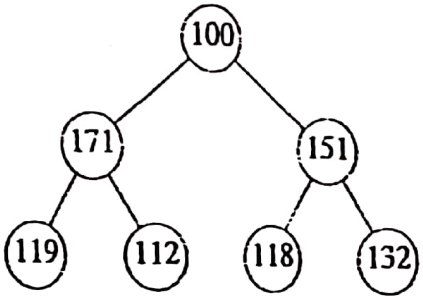
Heapify (100,119,118,171,112,151,132)



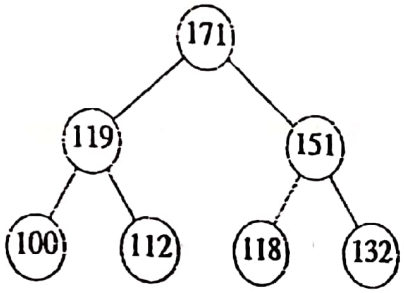
(a)



(b)



(c)



(d)

Munavvara Tahaseen

Heap Sort

Using Heapify only n/2 iterations are required.
Therefore the time complexity of Heapify is O(n) which is better than O(n log n).

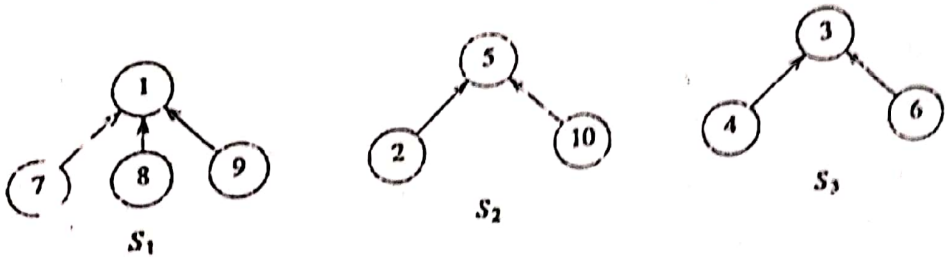
Munavvara Tahaseen

Set Representation

There are three ways of representing sets.

- Tree representation
- Array representation
- Data representation

Tree representation

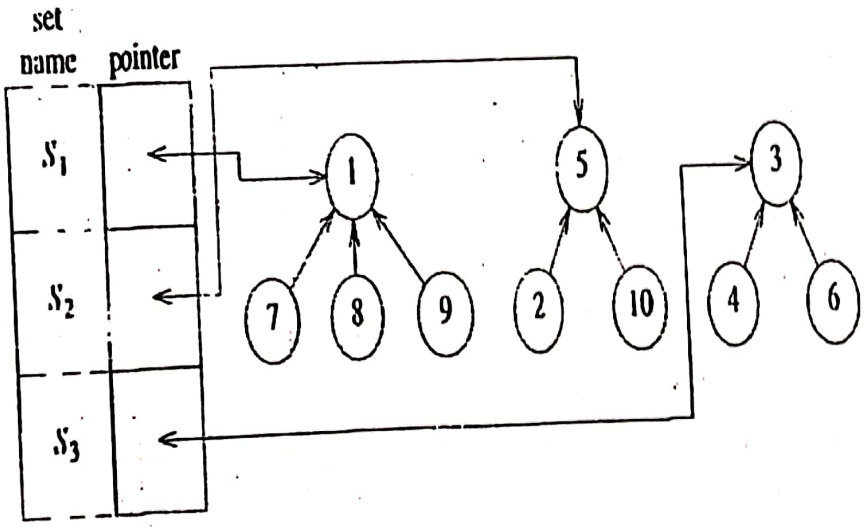


Array representation

i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	−1	5	−1	3	−1	3	1	1	1	5

Root nodes have a parent of -1.

Data Representation



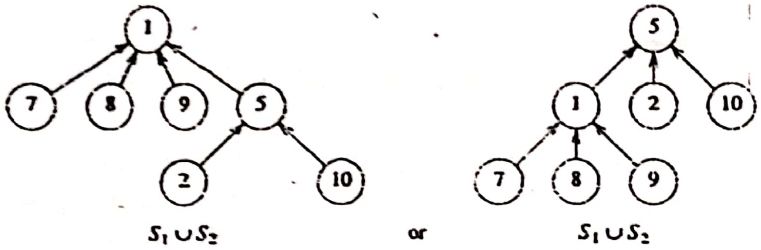
Operations on Sets

There are two operations performed on sets

1. Union
   - Simple union
   - Weighted union
2. Find
   - Simple find
   - Collapsing find

Union operation

Simple Union



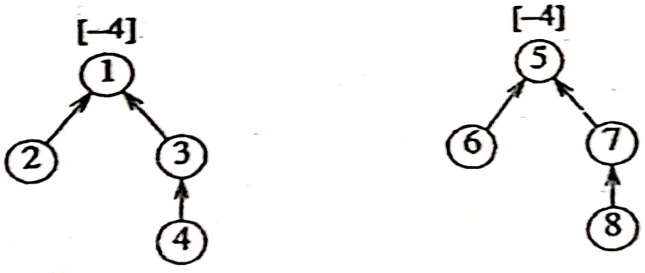
Simple Union

```
Algorithm SimpleUnion(i, j)
{
    p[i] := j;
}
```

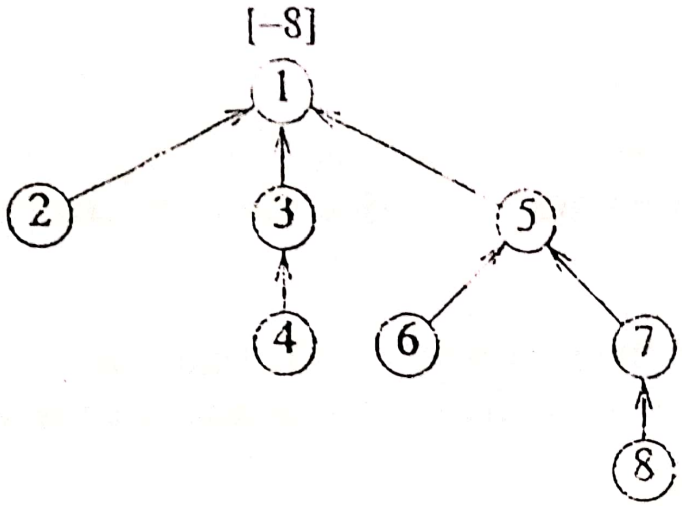
Weighted Union

```
Algorithm WeightedUnion(i, j)
// Union sets with roots i and j, i ≠ j, using the
// weighting rule. p[i] = -count[i] and p[j] = -count[j].
{
    temp := p[i] + p[j];
    if (p[i] > p[j]) then
    { // i has fewer nodes.
        p[i] := j; p[j] := temp;
    }
    else
    { // j has fewer or equal nodes.
        p[j] := i; p[i] := temp;
    }
}
```

Weighted Union



Weighted Union



Find Operation

```
Algorithm SimpleFind(i)
{
    while (p[i] ≥ 0) do i := p[i];
    return i;
}
```

Collapsing Find

```
Algorithm CollapsingFind(i)
// Find the root of the tree containing element i. Use the
// collapsing rule to collapse all nodes from i to the root.
{
    r := i;
    while (p[r] > 0) do r := p[r]; // Find the root.
    while (i ≠ r) do  // Collapse nodes from i to root r.
    {
        s := p[i]; p[i] := r; i := s;
    }
    return r;
}
```

Greedy Method

General Method

The greedy method suggests that one can devise 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

- A feasible solution that either maximizes or minimizes a given objective function is called an optimal solution.

```
Algorithm Grady(a,n)
// a[1:n] contains the n inputs
{
    solution := φ; // Initialize the solution
    for i:=1 to n do
    {
        x := Select(a);
        if Feasible(solution, x) then
            solution := Union(solution, x);
    }
    return solution;
}
```

The function Select selects an input from a[] removes it. The selected input's value is assigned to x.
Feasible is a Boolean-valued function that determines whether x can be included into the solution vector.
The function Union combines x with the solution and updates the objective function

Job Sequencing with deadlines

A set of n jobs. Associated with job i is an integer deadline $d_i \geq 0$ & a profit $p_i > 0$.

For any job i the profit p_i is earned iff the job is completed by its deadline.

To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs.

- A feasible solution for this problem is a subset of jobs such that each job in this subset can be completed by its deadline.
- The value of a feasible solution J is the sum of the profits of the jobs in J, or $\sum_{i \in J} p_i$.

An optimal solution is a feasible solution with maximum value.

(1) Eg Let n = 4 $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$

$(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

Maximum jobs can be executed is 2

	Feasible solution	Processing sequence	value	
1.	(1,2)	2,1	110	1,2,5
2.	(1,3)	1,3 or 3,1	115	1,3,4
3.	(1,4)	4,1	127 ✓	1,3,5
4.	(2,3)	2,3	25	
5.	(3,4)	(4,3)	42	
6.	(1)	1	100	
7.	(2)	(2)	10	
8.	(3)	3	15	
9.	(4)	4	27	

2,4 have (1,1) not possible to complete in same time

Both jobs must complete by 1 dead line.

```
Algorithm GreedyJob(d,J,n)
// J is a set of jobs that can be completed by their deadlines
{
    J := {1};
    for i := 2 to n do
    {
        if (all jobs in J ∪ {i} can be completed
                by their deadlines) Then J := J ∪ {i};
    }
}
```

```
Algorithm JS(d,j,n) // d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1
{  // The jobs are ordered such that p[1] ≥ p[2] ≥ ... ≥ p[n].
    d[0] := J[0] := 0 // Initialize
    J[1] := 1; // Include Job 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
        {  // Insert i into J[]
            for q := k to (r+1) step -1 do
                J[q+1] := J[q];
            J[r+1] := i;
            k := k+1;
        }
    }
    return k;
}
```

Complexity is $O(n^2)$.

(2) Sol: Let $n=5$, $(P_1, P_2, P_3, P_4, P_5) = (20, 15, 10, 5, 1)$
$(d_1, d_2, d_3, d_4, d_5) = (2, 2, 1, 3, 3)$.

Feasible Sol	processing Sequence	prof.t-
1	1	20
2	2	15
3	3	10
4	4	5
5	5	1
(1,2,3)	Not possible	-
1,2,4	1,2,4 (or) 2,1,4	40
1,2,5	1,2,5 (or) 2,1,5	31
2,3,4	3,2,4	30
2,3,5	3,2,5	26
2,4,5	2,4,5 (or) 2,5,4 (or) 5,2,4 (or) 4,2,5	21
1,3,4	3,1,4	35
1,3,5	3,1,5	31
1,4,5	1,4,5 (or) 1,5,4 (or) 5,1,4 (or) 4,1,5	26
3,4,5	3,4,5 (or) 3,5,4	16

Feasible solution (1,2,4) with profit 40

Algo GreedyKnapsack(m, n)
// P[1:n] contains profits
// w[1:n] contains weights of n object
// m is the size of knapsack x[1:n] is the solution vector
{
  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];

Knapsack Problem

In olden days, there was a store, contains different types of gold powders. Let these powders be $n1, n2, n3$ types, cost and weight of these powders are $c1, c2, c3$ dollors, $w1, w2, w3$ pounds respectively.

Now thief wants to rob the store, for that he brought a empty bag (knapsack means empty bag) of size M.

Now his problem was, in what way he has to place gold powder in the bag that he should get maximum profit? This problem is called as knapsack problem

- Knapsack is an empty bag. Consider knapsack size is M.

It contains n items with weights $w_1, w_2 \ldots w_n$ resp.

The profits of each weight is $P_1, P_2 \ldots P_n$.

Let $x_1, x_2, \ldots x_n$ be fractions of items.

The problem is $\boxed{\text{maximize} \sum_{1 \le i \le n} P_i x_i}$ Subject to $\sum_{1 \le i \le n} w_i x_i \le M$

The conditions are $0 \le x_i \le 1$ and $1 \le i \le n$

Example:-

Consider the following instance of knapsack problem

$n = 3$, $M = 20$ $(P_1, P_2, P_3) = (25, 24, 15)$

$(w_1, w_2, w_3) = (18, 15, 10)$

(1) Maximum profit =

(2) Minimum weight =

(3) Maximum profit per unit weight

Case 1:

In this case place an item in the bag whose profit maximum.

$x_1 = 1$ complete item w_1 is kept in the bag i.e 18 is kept

only 2 units place is left ($m = 20 - 18 = 2$)

x_2 = left out space / weight of item to be placed = 2/15

when w_2 is place no space is left in bag. So $x_3 = 0$.

we cannot place 3rd item

$$\sum_{1 \le i \le n} p_i x_i = p_1 x_1 + p_2 x_2 + p_3 x_3$$

$$= 25 \times 1 + 2/15 \times 24 + 15 \times 0$$

$$= 25 + 3.2 + 0$$

$$= 28.2$$

Case 2:

place an ~~at~~ item in bag, whose weight is minimum.

$x_3 = 1$

$x_2 = \frac{10}{15} = 2/3$

$x_1 = 0$

$$\sum p_i x_i = p_1 x_1 + p_2 x_2 + p_3 x_3$$

$$= 25 \times 0 + 24 \times 2/3 + 15 \times 1$$

$$= 0 + 16 + 15 = 31$$

Case 3:

Maximum profit per unit weight:

place an item in the bag, whose p/w ratio is max.

$$\frac{p_1}{w_1} = \frac{25}{18} = 1.4$$

$$\frac{p_2}{w_2} = \frac{24}{15} = 1.6$$

$$\frac{p_3}{w_3} = \frac{15}{10} = 1.5$$

P_2/w_2 is maximum so $x_2 = 1$. Now remaining weight of bag is $20 - 15 = 5$. Next maximum is P_3/w_3. No we have to place third item in bag but the space is not sufficient. So we have to place 1/2 of third item.

Therefore $x_3 = 1/2$. The bag is already filled, so $x_1 = 0$

$x_2 = 1$

$x_3 = \frac{5}{10} = 0.5$

$x_1 = 0$

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

$= 0 + 24 \times 1 + 15 \times 0.5 = 31.5$

$= 24 + 7.5 =$

$= 31.5$

Example : 2

$n = 7 \quad M = 15$

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

$(w_1, w_2, w_3, w_4, w_5, w_6, w_7) = 2, 3, 5, 7, 1, 4, 1$

Case 1 : - maximum profit

$x_6 = 1 \quad x_3 = 1 \quad x_1 = 1 \quad x_4 = 4/7$

$\sum P_i x_i = P_1 x_1 + P_2 x_2 + P_3 x_3 + P_4 x_4$

$= 1 \times 18 + 15 \times 1 + 10 \times 1 + \frac{4}{7} \times 7$

$= 18 + 15 + 10 + 4 = 47$

Case 2 :

$x_7 = 1 \quad x_5 = 1 \quad x_1 = 1 \quad x_2 = 1 \quad x_6 = 1 \quad x_3 = 4/5$

$\sum P_i x_i = P_1 x_1 + P_2 x_2 + P_3 x_3 + P_5 x_5 + P_6 x_6 + P_7 x_7$

$= 1 \times 10 + 1 \times 5 + 4/5 \times 15 + 1 \times 6 + 1 \times 18 + 1 \times 3$

$= 10 + 5 + 12 + 6 + 18 + 3 = 54$

Case 3 :

$\frac{P_1}{w_1} = \frac{10}{2} = 5 \quad \frac{P_2}{w_2} = \frac{15}{5} = 3 \quad \frac{P_5}{w_5} = \frac{6}{1} = 6 \quad \frac{P_7}{w_7} = \frac{3}{1} = 3$

$\frac{P_3}{w_3} = \frac{5}{5} = 1.6 \quad \frac{P_4}{P_4} = \frac{7}{7} = 1 \quad \frac{P_6}{w_6} = \frac{18}{4} = 4.5$

$\sum p_i x_i = p_5 x_5 + p_1 x_1 + p_6 x_6 + p_3 x_3 + p_7 x_7 + p_2 x_2$

$= 6 \times 1 + 10 \times 1 + 18 \times 1 + 15 \times 1 + 3 \times 1 + 15 \times 2/3$

$= 55.3$

Algorithm Greedy-knapsack (P, w, M, x, n)

```
p[1:n]  w[1:n]  x[1:n], m, n
{
  for i := 1 to n do
      x[i] := 0.0 ; // Initialize x.
      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] ;
}
```

Time complexity for knapsack is O(n)

Optimal Storage on Tapes

There are n programs that are to be stored on a computer tape of length L. Associated with each program is a length l_i, $1 \le i \le n$.

Clearly all programs can be stored on a tape iff the sum of the length of the program is almost

(i.e. $\sum_{i=1}^{n} l_i \le L$). $\exists$ $= i_1, i_2, \ldots, i_n$ Time t_j needed to retrieve prog i_j

$$t_j = \sum_{k=1}^{j} l_{i_k}$$

we shall assume that whenever a program is to be retrieved from this tape, the tape is initially positioned at the front.

In optimal storage on tape problem, we are required to find a permutation for the n programs so that they are stored on a tape in this order the MRT (mean retrieval time) is minimized

$$MRT = \frac{1}{n} \sum_{j=1}^{n} t_j$$

This problem fits the ordering paradigm. Minimizing the MRT is equalent to minimizing

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

Example

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

$3! = 6$.

1 2 3	$\longrightarrow$ 5 + 15 + 18 = 38
1 3 2	$\longrightarrow$ 5 + 8 + 18 = 31
2 1 3	$\longrightarrow$ 10 + 15 + 18 = 43
2 3 1	$\longrightarrow$ 10 + 13 + 18 = 41
3 1 2	$\longrightarrow$ 3 + 8 + 18 = (29)
3 2 1	$\longrightarrow$ 3 + 13 + 18 = 34

3, 1, 2 permutation is used since it has minimum MRT

Algorithm STORES (n, m)

```
{
    j ← 0;
    for i ← 1 to n do
    printf ("append program 'i' to permutation for tape, j");
        j ← (j + 1) mod m
    repeat
}
```

Time Complexity O(n log n)

Optimal Storage On multiple Tapes:- It requires the program to be stored on multiple tapes in circular queue fashion i.e, after storing the program in last tape the next program is stored in first tape.

Example

Find an optimal placement for 13 programs on 3 tapes where T_0, T_1, T_2 where the programs of the lengths 12, 5, 8, 32, 7, 5, 18, 26, 43, 11, 10, 6.

- Arrange all programs in ascending order.

3, 4, 5, 5, 6, 7, 8, 10, 11, 12, 18, 26, 32

T_0	3	5	8	12	32
T_1	4	6	10	18	
T_2	5	7	11	26	

} optimal placement

Find an optimal placement for 10 programs on T_0, T_1, T_2, T_3 where lengths of programs are

55, 60, 26, 1, 3, 7, 9, 11, 14, 85

- Ascending order 1, 3, 7, 9, 11, 14, 26, 55, 60, 85

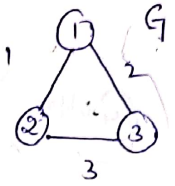
T_0	1	11	60
T_1	3	14	85
T_2	7	26	
T_3	9	55	

Minimum - Cost Spanning Trees

Definition:- Let $G = (V, E)$ be an undirected graph.

A subgraph $t = (V, E')$ of G is a spanning tree of G iff t is a tree.

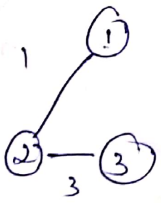
spanning tree:- A tree which includes all vertices in a graph G is called spanning tree.



If the graph consist of 'n' vertices then the possible spanning trees are n^{n-2}.

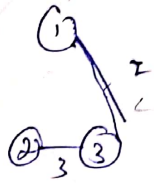
So $n = 3$ i.e $3^{3-2} = 3^1 = 3$ spanning Trees.

(a)



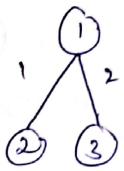
Cost = 1+3 = 4

(b)



Cost = 2+3 = 5

(c)



Cost = 1+2 = 3

Among these spanning Trees, fig (c) has minimum cost, so it is called as minimum spanning tree: is a tree because it is acyclic & it is spanning because it covers every vertex.

- A tree which includes all vertices of G with minimum cost is called minimum spanning tree.

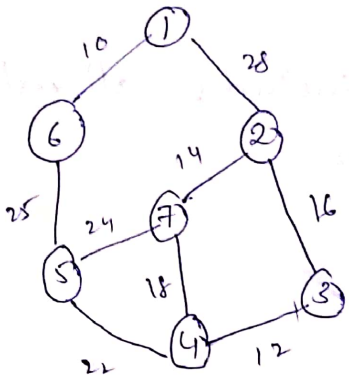
- If no. of vertices increased then it is difficult to find the cost of all possible spanning tree.

It is a time consuming and difficult process as n value increases.

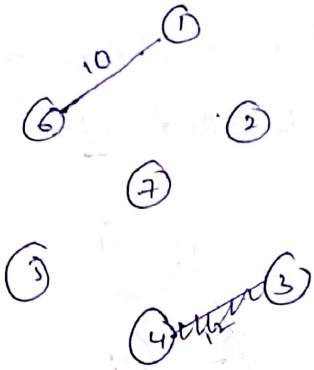
To avoid this difficulty, there are two standard algo

(1) Kruskal's algorithm (2) prim's algorithm.

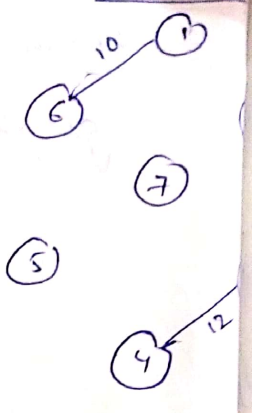
Kruskal's algorithm



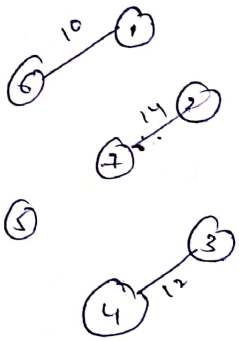
stage 1



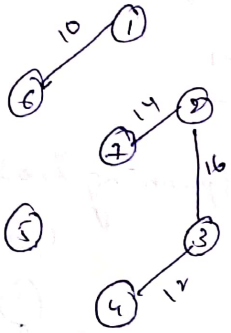
stage



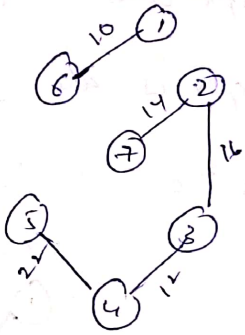
stage 3



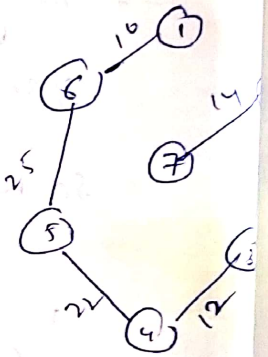
stage 4



stage 5



stage



Method:-

(1) Arrange all the edges in the increasing order of cost

(2) Include the edge with the minimum cost

(3) Add the next edge with minimum cost and delete edge from edge set until n-1 edges are added or edge set is exhausted.

(4) If the inclusion of any edge results in a cycle reject it and move on to next edge

(5) Go to step 3.

Arranging all edges in increasing order.

Edge	cost
(1,6)	10
(3,4)	12
(2,7)	14
(2,3)	16
(4,7)	18
(4,5)	22
(5,7)	24
(5,6)	25
(1,2)	28

Edge	Action	Added edges
(1,6)	add	{1,6}
(3,4)	add	{1,6} {3,4}
(2,7)	add	{1,6} {3,4} {2,7}
(2,3)	add	{1,6} {3,4} {2,7} {2,3}
(4,7)	reject	{1,6} {2,3,4,7}
(4,5)	add	{1,6} {2,3,4,5,7}
(5,7)	reject	{1,6} {2,3,4,5,7}
(5,6)	add	{1,6} {2,3,4,5,6,7}
(1,2)	reject	{1,2,3,4,5,6,7}

Selected edges = {(1,6) (3,4) (2,7) (2,3) (4,5) (5,6)}

= 10 + 12 + 14 + 16 + 22 + 25

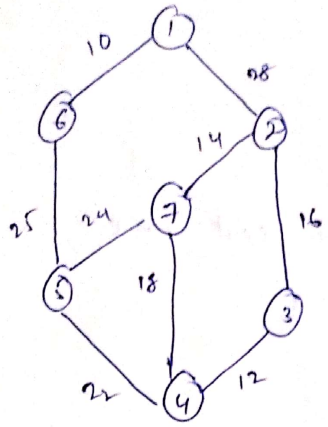
= 99

```
Kruskal ( E , cost, n, t) mincost )
{
real mincost, COST (1:n, 1:n)
int PARENT (1:n), t (1:n-1, 1:2), n;
Construct a heap out of edge costs using Heapify.

for i:=1 to n do
  parent [i] := -1; // Each vertex is assined to a distinct set
    i:=0;
    mincost:=0.0;          // edges are removed from heap one by one in non
                              order of cost
  while ( (i < n-1) and (heap not empty)) do
  {
    delete a mincost edge (u,v) from heap & reheapify using
      j := find(u);
      k := find(v);
      if (j ≠ k) then
      {
          i := i+1;
          t[i,1] := u;
          t[i,2] := v;
          mincost := mincost + cost(u,v);
              union(j,k); // Sets contains u & v are combine
      }
  }
  if ( i ≠ n-1) then    // if graph G is not connected
    write ("no spanning tree);
    else
     return mincost;
  }
```

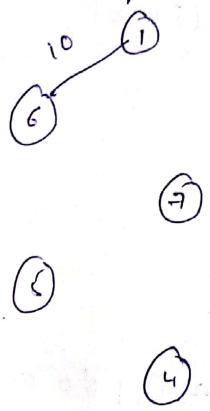
Time Complexity :- The time taken to delete an edge from heap is $O(\log_2 e)$, where 'e' is num of edges in the graph (G). while loop executes (n-1) times, approximately 'e' times. So the time complexity is $O(e \log_2 e)$ (or) $O(n \log n)$.

prims algorithm :-

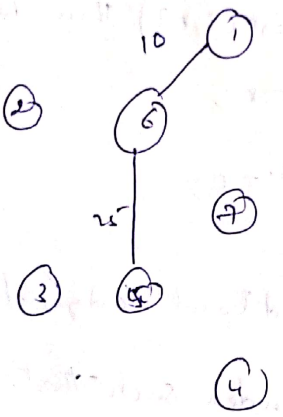


Edge	action	added edges
(1,6)	add	{(1,6)}
(6,5)	add	{(1,6)(6,5)}
(5,4)	add	{(1,6)(6,5)(5,4)}
(4,3)	add	{(1,6)(6,5)(5,4)(4,3)}
(3,2)	add	{(1,6)(6,5)(5,4)(4,3) (3,2)}
(2,7)	add	{(1,6)(6,5)(5,4)(4,3) (3,2)(2,7)}

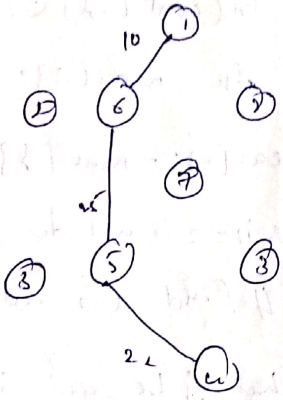
Stage 1



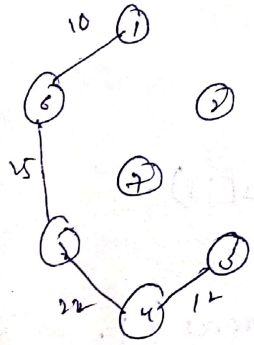
stage 2



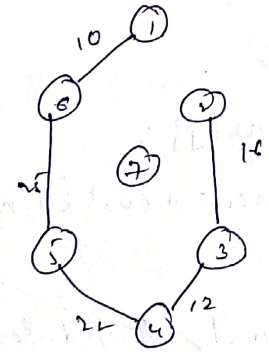
stage 3



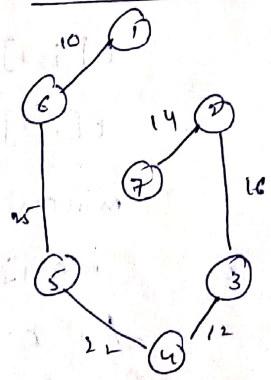
step 4



stage 5



stage 6



minimum cost = 99.

Method :-

(1) Include the edge with minimum cost

(2) Next include that edge which is connected from the vertices that has been included so far in the tree and with minimum cost

(3) Go to step2 if number of edges included is not equal

to n-1 edges and if edges are present in the graph otherwise stop.

```
Algorithm prims(E, cost, n, t)
{
    let (k, l) be an edge minimum cost in E;
    mincost := cost[k,l];
    t[1,1] = k;
    t[1,2] := l;
    for i:=1 to n do
        if(cost[i,l] < cost[i,k]) then near[i]:=l;
        else near[i]:=k;
    near[k]:= near[l]:=0;
    for i:=2 to n-1 do
    { // Find n-2 additional edges for t
        let j be an index such that near[j] ≠0 and
            cost[j, near[j]] is minimum;
        t[i,1] := j;
        t[i,2] := near[j];
        mincost := mincost + cost[j, near[j]];
        near[j]=0;
        for k:=1 to n do // updated near[].
            if((near[k] ≠0) and (cost[k,near[k]] > cost[k,
                then near[k]:=j;
    }
    return mincost;
}
```

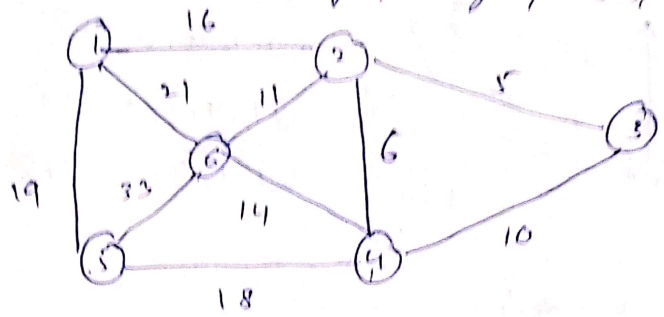
Time complexity :- The outer for loop executes n-1 t
and inner for loop executes n times, so

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

$\therefore O(n^2)$.

Example :- Determine the minimum cost spanning tree of the graph by prims & kruskals algo's.

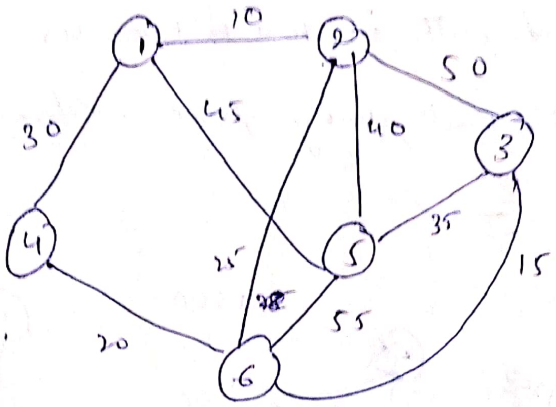
(1)



Kr's = 51

prim = 78

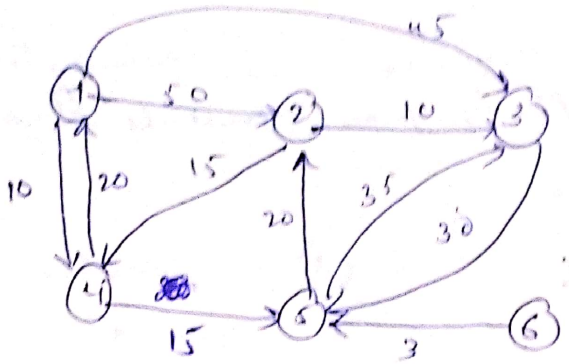
(2)



Single Source Shortest paths:- (or Dijkstra's Algo

The length of a path is now defined to be the sum of the weights of the edges on that path. The starting vertex of the path is referred to as the source, and the last vertex the destination.

A directed graph $G = (V, E)$, a weighting function cost for the edges of G, and a source vertex v_0. The problem is to determine the shortest paths form v_0 to all the remaining vertices of G.



	path	length
1)	1, 4	10
2)	1, 4, 5	25
3)	1, 4, 5, 2	45
4)	1, 3	45

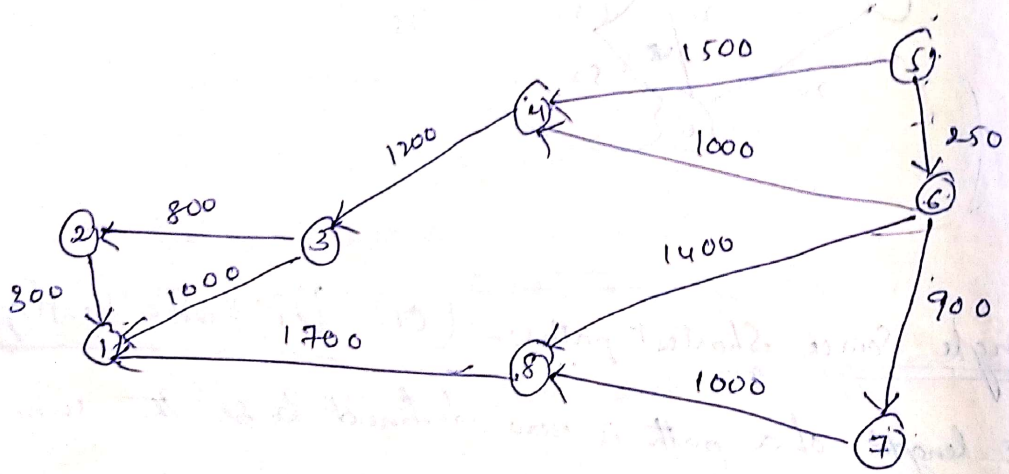
Source vertex is node '1'

shortest path from 1 to 2 is 1 → 4 → 5 → 2

the length of path is 10 + 15 + 20 = 45

from 1 → 2 is 50 but going from 3 edges is shorter

=



	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	1000	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

length - adjacency matrix.

Iteration	S	vertex selected	Distance [1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Initial	—	—	∞	∞	∞	1500	0	250	∞	∞
1	{5}	6	∞	∞	∞	1250	0	250	1150	1650
2	{5,6}	7	∞	∞	∞	1250	0	250	1150	1650
3	{5,6,7}	4	∞	∞	2450	1250	0	250	1150	1650
4	{5,6,7,4}	8	3350	∞	2450	1250	0	250	1150	1650
5	{5,6,7,4,8}	3	3350	3250	2450	1250	0	250	1150	1650
6	{5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
	{5,6,7,4,8,3,2}									

Action of shortest path

Algorithm Shortest paths (v, cost, dist, n)

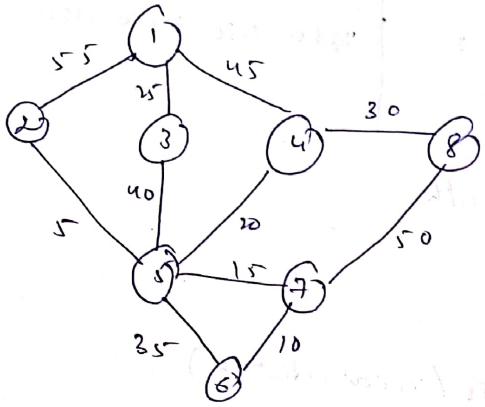
```
// n - vertices
// vertex v to j in d-agraph.
// dist[v] is set to zero
// cost[1:n, 1:n].
{ for i := 1 to n do
  { // Initialize S
    S[i] := false;
    dist[i] := cost[v,i];
  }
  S[v] := true;
  dist[v] := 0.0; // put v in S.
  for num := 2 to n-1 do
  { // Determine n-1 paths from v.
    choose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

Time Complexity

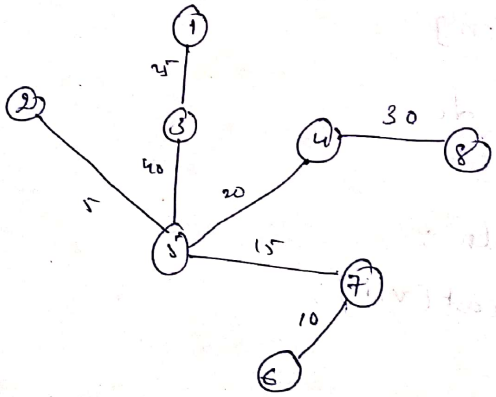
Time taken by the algorithm on a graph with 'n' vertices is $O(n^2)$

Overall time is $O((n+|E|)\log n)$.

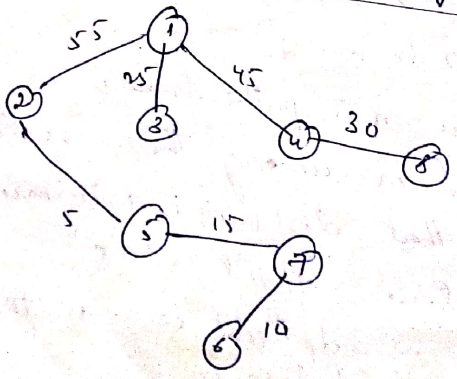
Eg :-



Graph.



minimum cost spanning tree



shortest path spanning tree from vertex '1'.

Dynamic Programming

General Method :-

- The drawback of greedy method is, we will make one decision at a time, this can be overcome in dynamic programming. In this, we will make more than one decision at a time.

- Dynamic programming is an algorithm design method, that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.

Dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share subsubproblems.

A dynamic-programming algorithm solves every subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subsubproblem is encountered.

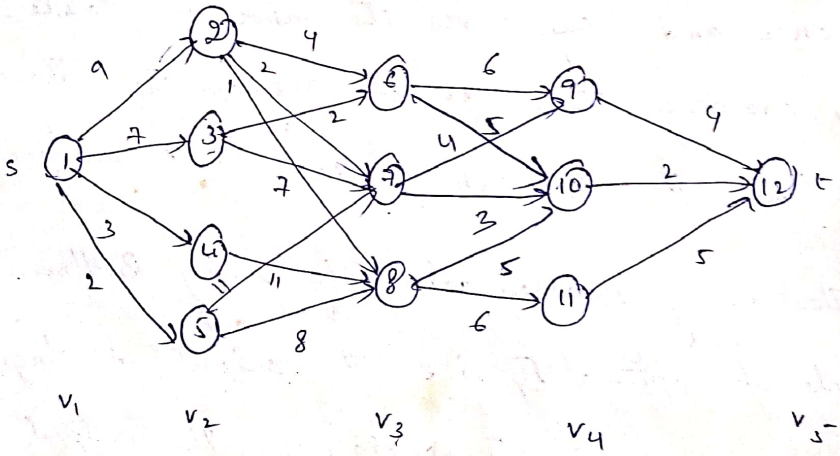
In this every problem solved by using Bellman's principle of optimality, i.e., the output of stage 1 will be given as input to stage 2, the output of stage 2 will be given as input to stage 3 and so on.

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

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 where $1 \leq i \leq k$. The sets V_1 & V_k are such that $|V_1| = |V_k| = 1$.
- Let s & t be the vertices in V_1 & V_k. The vertex s is source, and t the sink (destination)
- Let cost(i,j) be the cost of edge $\langle i,j \rangle$.
- 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.

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



- Each V_i defines a stage in the graph

cost(1,1)
cost(k-2,j) where j is a vertex belongs to the stage.
cost(k-3,j)
cost(k-4,j)

stage 5
cost(5,12) = 0

stage 4
cost(4,9) = 4 cost(4,11) = 5
cost(4,10) = 2

Step 3

First we have to make k-2 decisions.

Here k = 5

k - 2 = 5 - 2 = 3.

cost(3,6) , cost(3,7) , cost(3,8)

$\Rightarrow cost(\overset{i}{3},\overset{j}{6}) = \min_{l \in V_{i+1}} \{ c(6,9) + cost(4,9), \; c(6,10) + cos(4,10) \}$

$= \min\{6+4, \; 5+2\}$

$= \min\{10, 7\}$

$= 7.$

$d(3,6) = 10$

$\Rightarrow cost(\overset{i}{3},\overset{j}{7}) = \min\{ c(7,9) + cost(4,9), \; c(7,10) + cost(4,10) \}$

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

$= \min\{8, 5\}$

$= (5).$

$d(3,7) = 10$

$\Rightarrow cost(3,8) = \min\{ c(8,10) + cost(4,10), \; c(8,11) + cost(4,11) \}$

$= \min\{5+2, \; 6+5\}$

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

$= 7$

$d(3,8) = 10$

Stage 3 Now k-3 decisions $k-3 = 5-3 = 2$.

cost(2,2) cost(2,3) cost(2,4) cost(2,5)

$\Rightarrow$ cost(2,2) = min{ c(2,6) + cost(3,6), c(2,7) + cost(3,7), c(2,8) + cost(3,8) }

= min{ 4 + 7, 2 + 5, 1 + 7 }

= (7) $d(2,2) = 7$

$\Rightarrow$ cost(2,3) = min{ c(3,6) + cost(9,6), c(3,7) + cost(3,7) }

= min{ 2 + 7, 7 + 7 }

= 9 $d(2,3) = 6$

$\Rightarrow$ cost(2,4) = min{ c(4,8) + cost(3,8) }

= min{ 11 + 7 } = 18 $d(2,4) = 8$.

$\Rightarrow$ cost(2,5) = min{ c(5,7) + cost(3,7), c(5,8) + cost(3,8) }

= min{ 11 + 5, 8 + 7 } = 15

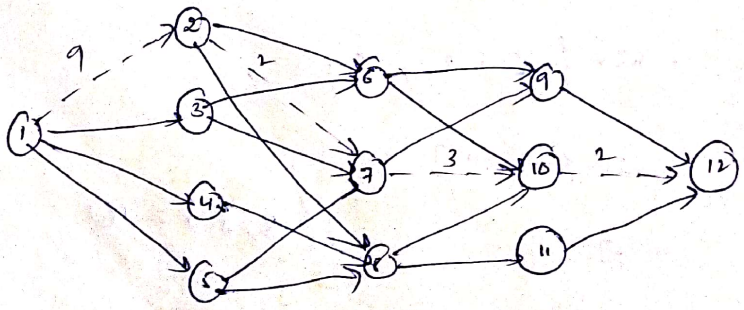
$d(2,5) = 8$

Stage 1 Now k-4 decisions $5-4 = 1$

$\Rightarrow$ cost(1,1) = min{ c(1,2) + cost(2,2), cost(1,3) + cost(2,3), c(1,4) + cost(2,4), c(1,5) + cost(2,5) }

= min{ 9+7, 7+9, 3+6, 2+15 } = 16.

$d(1,1) = 2$ (or) 3.



Backward

A minimum cost 5 to 1 path is indicated by broken edg

9 + 2 + 3 + 2 = 16.

- Let us suppose d(i,j) be the value of l (where l is a node) that minimizes c(j,l) + cost(i+1,l)

d(3,6) = 10 d(3,7) = 10 d(3,8) = 10

d(2,2) = 7 d(2,3) = 6 d(2,4) = 8 d(2,5) 7

d(1,1) = 2

minimum cost path be s = 1, v2, v3 ... vk-1, t

$v_2 = d(1,1) = 2$

$v_3 = d(2, d(1,1)) = 7$

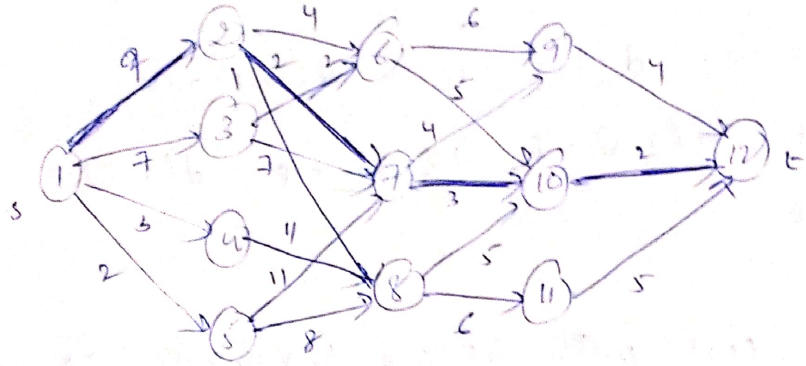
$v_4 = d(3, d(2, d(1,1))) = d(3,7) = 10$.

1 → 2 → 7 → 10 → 12

/* Routine for multi graph using Forward approach */

```
FGraph(G, k, n, p)
{
  cost = 0.0;
  for j := n-1 to 1 step -1 do
  {
    Let r be the vertex such that <j,r> is an edge
    of graph { cost(r) + c(j,r) is minimum;
        cost[j] := c[j,r) + cost(r);
        d[j] := r;
    }
    p[1] = 1;
    p[k] := n;
    for j := 2 to k-1 do
        p[j] := d[p[j-1]];
}
```

Backward approach



$$bcost[i,j] = \min_{l \in V_0,\ \langle l,j\rangle \in E} \{ c(l,j) + bcost(i-1,l) \}$$

$bcost(2,j) = c(1,j)$ if $\langle 1,j\rangle \in E$

$bcost(2,j) = \infty$

$i = 3, 4, \ldots$

for $i = 3$

$bcost[\overset{i}{3},\overset{j}{6}] = \min\{ c(2,6) + bcost(2,2),\ c(3,6) + bcost(2,3) \}$

$= \min\{4+9,\ 2+7\} = 9$

$bcost[3,7] = \min\{4+2,\ 7+7,\ 2+11\} = 11$

$bcost[3,8] = \min[1+9,\ 11+3,\ 8+2\} = 10$

for $i = 4$

$bcost[4,9] = \min\{ c(6,9) + bcost(3,6),\ c(7,9) + bcost$

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

$bcost[4,10] = \min\{5+9,\ 3+11,\ 5+10\} = 14$

$bcost[4,11] = \min\{6+10\} = 6.$

for i = 5

bcost[5,12] = min{ 4+15, 2+14, 5+16 } = 16

d[j,j] = ?

d[3,6] = 3 d[3,7] = 2 d[3,8] = 5

d[4,9] = 6 or 7 d[4,10] = 6 or 7 d[4,11] = 8

d[5,12] = 10 = v_4 v_3 = d(4, d(5,12))

= d(4,10) = 7

v_2 = d(3(4, d(5,12)))

v_1 = 1

= d(3, d(4,10))

= d(3,7) = 2

Using Backward

12 → 10 → 7 → 2 → 1 (or) 12 → 10 → 6 → 3 → 1

/* Routine for Backward approach */

```
Bgraph(G, k, n, p)
{ cost[i] := 0.0;
for j := 2 to n do
{ Let 'r' be a vertex such that <r,j>
  be an edge of G } c[r,j] + cost[r] is minimum;
  cost[j] := c[r,j] + cost[r];
    d[j] := r;
}
p[1] = 1;
p[k] = n;
for (j = k-1) to 2 do
  p[j] := d[p[j+1]];
}
```

Optimal Binary Search Trees

Binary Search Tree

A Binary Search tree T is a binary tree, either it is empty or each node in the tree contains an identifier and

(i) all identifiers in the left subtree of T are less than the identifier in the root node T

(ii) all identifiers in the right subtree are greater than the identifier in the root node T

(iii) The left and right subtree of T are also binary Search tree.

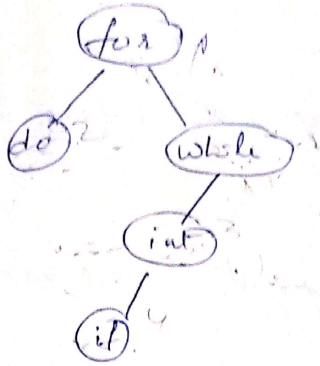
/* For searching a binary search tree */

```
Algorithm SEARCH(T, X, i)
// Search the binary search tree T for X.
// Each node of the tree has fields LCHILD,
// IDENT, RCHILD, if X is not in T then
// set i = 0, IDENT(i) = X
    i ← NIL;
    while i ≠ 0 do
      case
        : X < IDENT(i) : i ← LCHILD(i)   // Search left subtree
        : X = IDENT(i) : return
        : X > IDENT(i) : i ← RCHILD(i)   // Search right subtree
      endcase
      repeat
end SEARCH
```

- To search an element in binary search tree, First that element compared with root node. If element is less than root node then search continue in left subtree. If element is greater than root node then search continue in right subtree. If element

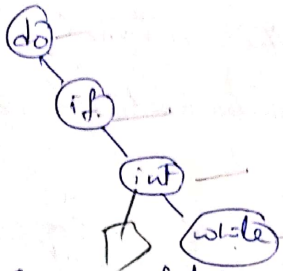
Case 1:

- Each and every node has equal probability and no unsuccessful searches are made.



on average it takes 1, 2, 2, 3, 4 comparisons respectively to find the identifiers for, do, while, int, if.

Avg num of comparisons is $\frac{1+2+2+3+4}{5} = \frac{12}{5}$



$\frac{1+2+3+4}{4} = \frac{10}{4}$

A successful search is terminated at an internal node is denoted by circle ○ & unsuccessful node at external is denoted by square □

Case 2:

Every node has different probability & unsuccessful searches.

Set of identifiers is $\{a_1, a_2, \ldots a_n\}$ with $a_1 < a_2 < \ldots a_n$.

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 $a_i < x < a_{i+1}$, $0 \le i \le n$.

$\sum_{0 \le i \le n} q(i)$ is the probability of an unsuccessful search

$\sum_{1 \le i \le n} p(i) + \sum_{0 \le i \le n} q(i) = 1$.

- To obtain binary search trees, it is useful to add a external node in place of every empty subtree in the search tree. Such nodes are called external nodes. All other nodes are internal nodes.

If a binary search tree represents n identifiers, then there will be exactly n internal nodes & n+1 external nodes.

Every internal node represents a point where a successful search may terminate.

Every external node represents a point where an unsuccessful search may terminate.

The expected cost contribution from the internal node for a_i is $\sum_{i=1}^{n} p(i) * level(a_i)$ [probability of successful search

probability of unsuccessful search node is $\sum_{i=0}^{n} q(i) * (level(E_i))$

The Expected cost of a binary search tree:

$$cost(t) = \sum_{1 \le i \le n} p(i) * level(a_i) + \sum_{0 \le i \le n} q(i) * (level(E_i))$$

Eg 1:-

Consider an identifier set $(a_1, a_2, a_3) = (do, if, while)$. The probabilities of successful & unsuccessful search is given by $p[i] = q[i] = 1/7$. Find an optimal B.S.T

$n = 3$

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

$= \frac{1}{4} {}^{6}C_3$

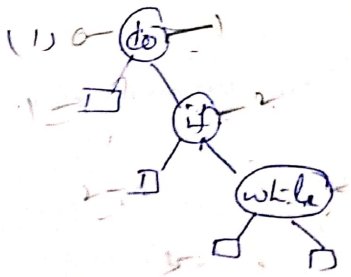
$= \frac{1}{4} \frac{6!}{3!3!}$

$= \frac{1}{4} \frac{6 \times 5 \times 4 \times 3!}{3!3!}$

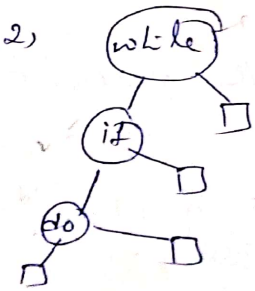
$= \frac{1}{4} \frac{6 \times 5 \times 4}{3 \times 2 \times 1}$

$= 5$

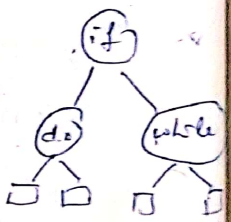
(1)



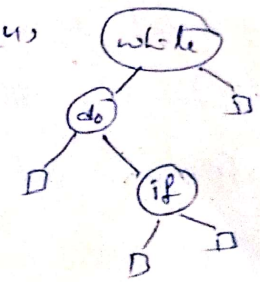
(2)



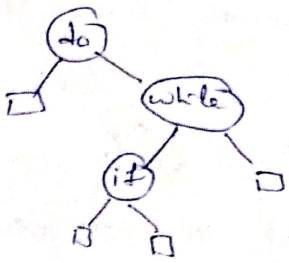
(3)



(4)



(5)



(1) probability of successful search $= \left(\sum_{1 \le i \le n} p(i) * level(a_i)\right)$

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

probability of unsuccessful search

$= \left(\sum_{0 \le i \le n} q(i) * (level(E_i) - 1)\right)$

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

$cost(T) = \frac{6}{7} + \frac{9}{7} = \frac{15}{7}$

(2) $Succes = 1 \times \frac{1}{7} + 2 \times \frac{1}{7} + 3 \times \frac{1}{7} = \frac{6}{7}$

$unsuccess = 1 \times \frac{1}{7} + 2 \times \frac{1}{7} + 3 \times \frac{1}{7} + 3 \times \frac{1}{7} = \frac{9}{7}$

$cost(T) = \frac{6}{7} + \frac{9}{7} = \frac{15}{7}$

(3) $Succes = 1 \times \frac{1}{7} + 2 \times \frac{1}{7} + 2 \times \frac{1}{7} = \frac{5}{7}$

$unsuccess = 2 \times \frac{1}{7} + 2 \times \frac{1}{7} + 2 \times \frac{1}{7} + 2 \times \frac{1}{7} = \frac{8}{7}$

$cost(T) = \frac{5}{7} + \frac{8}{7} = \frac{13}{7}$

(4) $Succes = 1 \times \frac{1}{7} + 2 \times \frac{1}{7} + 3 \times \frac{1}{7} = \frac{6}{7}$

$unsuccess = 1 \times \frac{1}{7} + 2 \times \frac{1}{7} + 3 \times \frac{1}{7} + 3 \times \frac{1}{7} = \frac{9}{7}$

$Cost(T) = \frac{6}{7} + \frac{9}{7} = \frac{15}{7}$

(5) $Succes = 1 \times \frac{1}{7} + 2 \times \frac{1}{7} + 3 \times \frac{1}{7} = \frac{6}{7}$

$unsuccess = 1 \times \frac{1}{7} + 2 \times \frac{1}{7} + 3 \times \frac{1}{7} + 3 \times \frac{1}{7} = \frac{9}{7}$

$cost(T) = \frac{6}{7} + \frac{9}{7} = \frac{15}{7}$

So Third B.S.T is the optimal B.S.T

Eg 2:- Consider an identifier set $(a_1, a_2, a_3) = (do, if, \ldots$ Eg

$p[i] = q[i] = 1/7$. Find O.B.S.T .

By using dynamic programming you can select root node for optimal binary search tree.

Let internal nodes be $a_1, a_2, \ldots a_n$ {

external nodes is $E_0, E_1 \ldots E_n$.

If a_k as root for optimal Binary Search tree then a_0 to a_{k-1} will be the left subtree { a_{k+1} to a_n will be in the right subtrees.

a_0 to a_{k-1} (E_0 to E_{k-1})

a_{k+1} to a_n (E_{k+1} to E_n)

If we consider left subtree as l { right subtree as r

$$cost(l) = \sum_{i=1}^{k} p(i) * level(a_i) + \sum_{i=0}^{k} q(i) * level(E_i)$$

$$cost(r) = \sum_{i=k+1}^{n} p(i) * level(a_i) + \sum_{i=k}^{n} q(i) * level(E_i)$$

Ingenad

To find O.B.S.T the following equation is used

$$c(i,j) = \min_{i<k\leq j} \{ c(i,k-1) + c(k,j) \} + w(i,j)$$

where, $w(i,j) = p(j) + q(j) + w(i,j-1)$

$w(i,i) = q(i)$

$c(i,i) = 0$

To find O.B.S.T the following equation should be

$$c(0,n) = \min_{i<k\leq n} \{ c(0,k-1) + c(k,n) \} + w(0,n).$$

Eg Let $n = 4$ and (q_1, q_2, q_3, q_4) (do, if, int, while).

$P(1:4) = (3, 3, 1, 1)$ $\quad q_0(0,4) = (2, 3, 1, 1)$

Initially $w(i,i) = q(i)$ $\quad c(i,i) = 0$ $\quad r(i,i) = 0$, $0 \le i \le 4$.

- No. of possible binary search trees $= \frac{1}{n+1} {}^{2n}C_n$

$= \frac{1}{5} {}^{8}C_4 = 14$

Draw one table by taking i corresponds to row & j corresponds to columns.

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $R_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $R_{11} = 0$	$w_{12} = 1$ $c_{22} = 0$ $R_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $R_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $R_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $R_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $R_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $R_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $R_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $R_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $R_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $R_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $R_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $R_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $R_{04} = 2$				

$$w(i,j) = p(j) + q(j) + w(i,j-1)$$

$$c(i,j) = \min_{i<k\le j} \{ c(i,k-1) + c(k,j) \} + w(i,j)$$

$$w(i,i) = q(i)$$

$$c(i,i) = r(i,i) = 0$$

So $w(0,0) = q(0) = 2$ $\quad c(0,0) = R(0,0) = 0$

$w(1,1) = q(1) = 3$ $\quad c(1,1) = R(1,1) = 0$

$w(2,2) = q(2) = 1$ $\quad c(2,2) = R(2,2) = 0$

$w(3,3) = q(3) = 1$ $\quad c(3,3) = R(3,3) = 0$

$w(4,4) = q(4) = 1$ $\quad c(4,4) = R(4,4) = 0$

$w(0,1) = p(1) + q(1) + w(0,0)$

$= 3 + 3 + 2 = 8$

$w(1,2) = p(2) + q(2) + w(1,1) = 3 + 1 + 3 = 7$

$w(2,3) = p(3) + q(3) + w(2,2) = 1 + 1 + 1 = 3$

$w(3,4) = p(4) + q(4) + w(3,3) = 1 + 1 + 1 = 3$

$w(0,2) = p(2) + q(2) + w(0,1) = 8 + 1 + 8 = 12$

$w(1,3) = p(3) + q(3) + w(1,2) = 1 + 1 + 7 = 9$

$w(2,4) = p(4) + q(4) + w(2,3) = 1 + 1 + 3 = 5$

$w(0,3) = p(3) + q(3) + w(0,2) = 1 + 1 + 12 = 14$

$w(1,4) = p(4) + q(4) + w(1,3) = 1 + 1 + 9 = 11$

$w(0,4) = p(4) + q(4) + w(0,3) = 1 + 1 + 14 = 16$

$$c(i,j) = \min_{i<k\leq j} \{ c(i,k-1) + c(k,j) \} + w(i,j)$$

for $j - i = 1$

$c(0,1), c(1,2), c(2,3), c(3,4)$

for $(j - i) = 2$

$c(0,2), c(1,3), c(2,4)$

for $(j - i) = 3$

$c(0,3), c(1,4)$

for $(j - i) = 4$

$c(0,4)$

⇒ for j-i = 1

$$c(0,1) = \min_{0<k\le 1} \{ c(0,0) + c(1,1) \} + w(0,1) \qquad (k=1)$$

$$= \min_{0<k\le 1} (0+0) + w(0,1) = 0+8 = 8$$

min of k value is 1 so $r(0,1) = k$

$r(0,1) = 1$

$$c(1,2) = \min_{1<k\le 2} \{ c(1,1) + c(2,2) \} + w(1,2) \qquad (k=2)$$

$$= w(1,2) = 7$$

$r(1,2) = 2$

$$c(2,3) = \min_{2<k\le 3} \{ c(2,2) + c(3,3) \} + w(2,3) = 0+3 = 3 \qquad (k=3)$$

$r(2,3) = 3$

$$c(3,4) = \min_{3<k\le 4} \{ c(3,3) + c(4,4) \} + w(3,4) \qquad (k=4)$$

$$= 0+3 = 3$$

$r(3,4) = 4$

⇒ for j-i = 2.

$$c(0,2) = \min_{0<k\le 2} \{ \underbrace{c(0,0) + c(1,2)}_{k=1}, \underbrace{c(0,1) + c(2,2)}_{k=2} \} + w(0,2)$$

$$0 \quad 7 \qquad\qquad 8 \quad 0$$

$$= \min\{7, 8\} + w(0,2)$$

$$= 7 + w(0,2) = 7 + 12 = 19$$

↳ so k is 1

$r(0,2) = 1$

$c(1,3) = \min_{1<k\leq 3} \{ \underbrace{c(1,1) + c(1,3)}_{k=1},\ \underbrace{c(1,2) + c(2,3)}_{k=2} \} + w(1)$

$= 3 + w(1,3) = 3 + 9 = 12$

$R(1,3) =$ min among k is (1, 3) is k value is 2.

$= 2.$

$c(2,4) = \min_{2<k\leq 4} \{ \underbrace{c(2,2) + c(3,4)}_{k=3},\ \underbrace{c(2,3) + c(4,4)}_{k=4} \} + w$

$= 3 + 5 = 8$

$R(2,4) = 3$

$\Rightarrow$ for $j - i = 3$

$c(0,3) = \min_{0<k\leq 3} \{ \underbrace{c(0,0) + c(1,3)}_{k=1},\ \underbrace{c(0,1) + c(1,3)}_{k=2},$

$\underbrace{c(0,2) + c(3,3)}_{k=3} \} + w(0,3)$

$= 11 + 14 = 25$

$R(0,3) = 2$

$c(1,4) = \min_{1<k\leq 4} \{ \underbrace{c(1,1) + c(2,4)}_{k=2},\ \underbrace{c(1,2) + c(3,4)}_{k=3},$

$\underbrace{c(1,3) + c(4,4)}_{k=4} \} + w(1,4)$

$= 8 + 11 = 19$

$R(1,4) = 2$

⇒ for j-i = 4

$$c(0,4) = \min_{0<k\le 4} \{ \overbrace{c(0,0) + c(1,4)}^{k=1}, \overbrace{c(0,1) + c(2,4)}^{k=2}, \overbrace{c(0,2) + c(3,4)}^{k=3}, \overbrace{c(0,3) + c(4,4)}^{k=4} \} + w(0,4)$$

$$= 16 + 16 = 32$$

$$R(0,4) = 2$$

R(i,j) = represents root tij it is the value of k that minimises a equation

Now Observe last cell i.e., 4th row & 0th column.

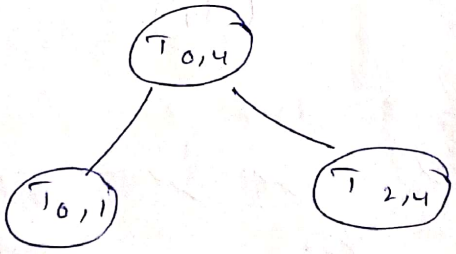
Contains R(0,4) = 2 i.e a_2 (2 corresponds to second node a_2)

R(ou) = k Then k = 2

Let T be the optimal binary Search tree

$T_{i,j} = T_{i,k-1}, T_{k,j}$

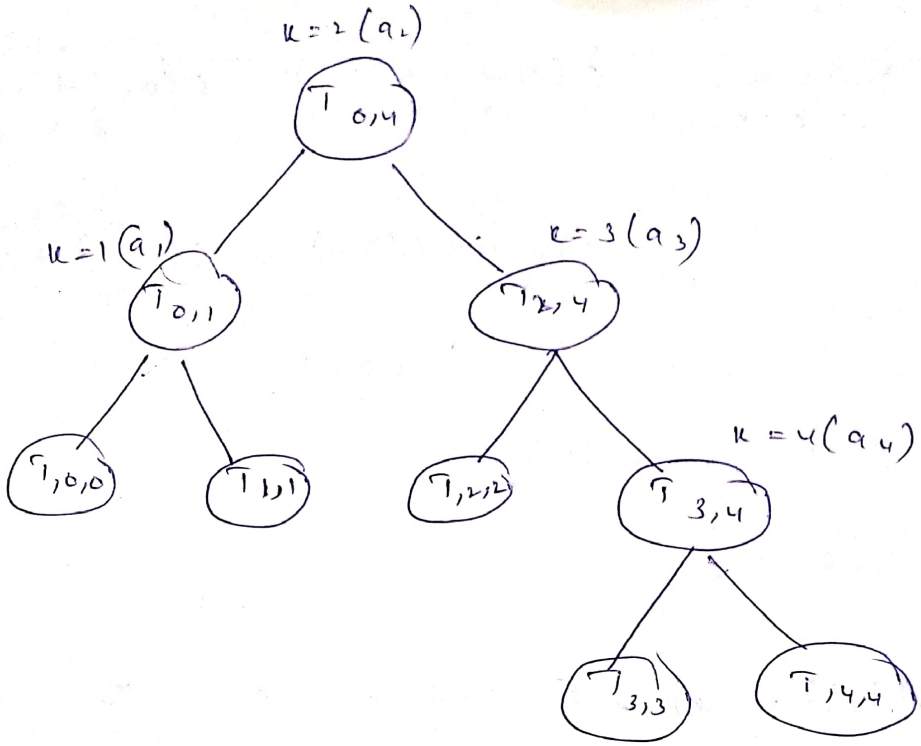
$T_{0,4}$ is divided into $T_{0,1}$ & $T_{2,4}$ (∵ k = 2)



$T_{0,1}$ is divided into $T_{0,0}$ & $T_{1,1}$ (∵ k = 1)

$T_{2,4}$ is divided into $T_{2,2}$ & $T_{3,4}$ (∵ k = 3)

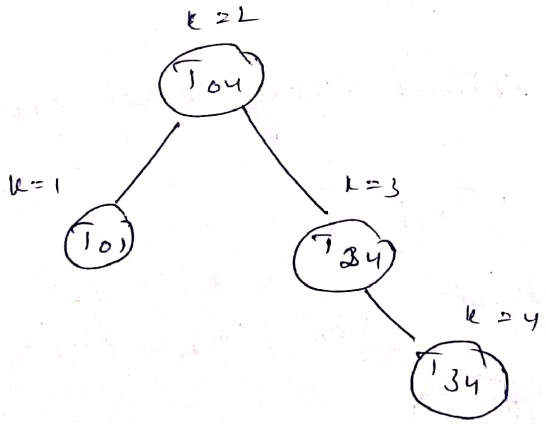
$T_{3,4}$ is " $T_{3,3}$ & $T_{4,4}$ (∵ k = 4)

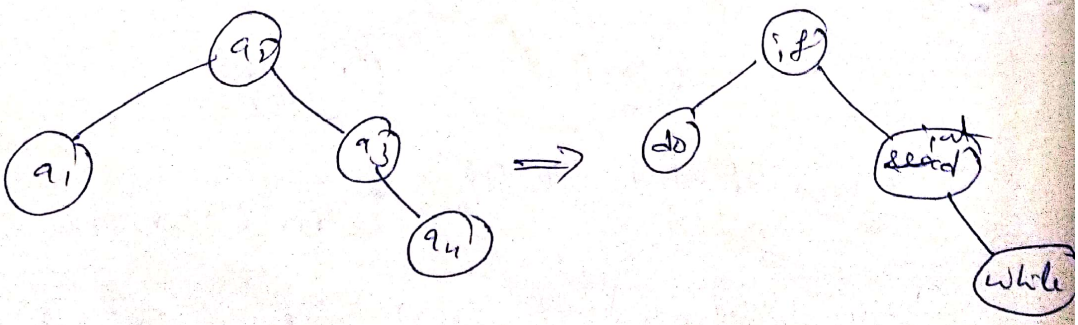


∴ R(0,0) R(1,1) R(2,2) R(3,3) and R(4,4) is 0

These are external nodes, we can neglect these nodes

The cost of optimal binary search tree is 32.





optimal Binary Search tree.

```
Algorithm OBST (p, q, n)
// w[i,j] is the weight of t_ij
// c[i,j] cost of OBST
// 'n' is q_1 < q_2 < ... < q_n {probabilities p[i], 1≤i≤n} q[i], 0≤i≤n.
// r[i,j], the root of t_ij
{
  for i := 0 to n-1 do
  {
      // Initialize
      w[i,i] = q[i]; r[i,i] = 0; c[i,i] = 0.0;
      // optimal trees with one node
      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 // Find optimal trees with m nodes.
    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);
            // A value of l in the range r[i,j-1] ≤ l
            // ≤ r[i+1,j] that minimize c[i,l-1]+c[l,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]);
}
Algo Find(c,r,i,j)
{
  min := ∞;
  for m := r[i,j-1] to r[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;
```

Time Complexity :-

Each $c(i,j)$ can be computed in time $O(m)$.

The total time for all $c(i,j)$'s is $O(n-m) \cdot O(m)$

$= O(nm - m^2)$

$= n-1 + 2n-4 + 3n-9 + \ldots + n(n-1) - (n-1)^2 + n^2$

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

$= 3n^2 - 2n^2$

$T(n) = \frac{n(n+1)(2n+1)}{6} - \frac{n(n+1)(2n+1)}{6}$

$= n^3 - n^3 =$

$\therefore O(n^3)$

(OR)

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

$$= \frac{n(m+1)}{2} m - \frac{m(m+1)(2m+1)}{6}$$

$$= \frac{nm^2}{2} - \frac{m^3}{3}$$

So Time complexity is $O(m^3)$ or $O(n^3)$

0/1 knapsack problem

- Now compare knapsack problem with general problem.

In store, contains different types of ornaments, with gold. Let n_1, n_2, n_3 be ornaments, cost and weight of these ornaments are c_1, c_2, c_3 dollors, w_1, w_2, w_3 pounds.

Now a thief wants to rob the ornaments should get maximum profit.

The thief can't place fraction of ornament in the bag, i,e either place an ornament completely in the bag or he can't place ornament.

So $x_i = 0$ or 1

$x_i = 0$, means we cannot place ornament in the bag.

$x_i = 1$, means we can place ornament completely in the bag.

- This problem contains either 0 or 1, that's why this problem is called as 0/1 knapsack problem.

Fraction item can't be placed in the bags.

Consider weights $w_1, w_2, \ldots w_n$ and fraction of weights to be put in the bag should be

$x_1, x_2 \ldots x_n, x_i = 0$ or 1.

- Let $f_i(y)$ be the value of an optimal solution to KNAP(1,j,y)

Since the principle of optimality holds,

$$f_n(m) = \max\{f_{n-1}(m), f_{n-1}(m - w_n) + p_n\}$$

$f_i(y), i > 0$ then

$$f_i(y) = \max\{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\}$$

$f_n(m)$ by beginning with $f_0(y) = 0 \quad \forall y$ $\{ f_i(y) = -\infty, y < 0$.

Order set $S^i = \{(f(y_j), y_j) \mid 1 \le j \le k\}$

Each member of S^i is a pair (P, w)

where $P = f_i(y_j)$ $\quad w = y_j$, Notice that $S^0 = \{(0,$

So we can compute S^{i+1} from S^i by first computing

$$S_1^i = \{(P, w) \mid (P - P_i, w - w_i) \in S^i\}$$

- S^{i+1} is computed by merging the pair in S^i & S_1^i

If S^{i+1} contains two pairs (P_j, w_j) & (P_k, w_k) then check if $P_j \le P_k$ & $w_j \ge w_k$.

If so (P_j, w_j) can be discarded

This is known as dominance rule

$S_0^0 = \{(0,0)$
$S_1^i = S^{i-1} + (P_i$
$S^i = S^{i-1} + S_1^i$ m

Eg :-

Consider the knapsack instance $n = 3$, $(w_1, w_2, w_3) = ($
$(P_1, P_2, P_3) = (1, 2, 5)$ & $m = 6$.

2, 3, 4

~~$S^0 = \{(0,0)\}$~~ ; ~~$S_1^i = S^{i-1} + (P_i, w_i)$~~ ~~$S_1^0 = S^0 +$~~

~~$S_1^0 = \{(1,2)\}$~~

~~$S^1 = \{(0,0)(1,2)\}$~~ ~~$S^1 = \{(2,3),$~~

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

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

$S_1^1 = S^0 + (P_1, w_1)$

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

$= \{1,2\}$

$S^1 = S^0 + S_1^1$ (merging)

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

$S_1^2 = S^1 + (P_1, w_1)$ (add a tuple)

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

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

$S^2 = S^1 + S_1^2$ (merge)

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

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

$S_1^3 = S^2 + (P_3, w_3)$

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

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

$S^3 = S^2 + S_1^2$

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

Purging Rule (Dominance rule)

if one of S^{i-1} & S^i has a pair (P_j, w_j) & other has a pair (P_k, w_k) & $P_j < P_k$ while $w_j \geq w_k$ then the pair (P_j, w_j) is discarded. $3 < 5$ & $5 \geq 4$ so $(3,5)$ is discarded.

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

After applying purge rule, check condition

if $(P_i, w_i) \in S^n$

Remove &

$(P_i, w_i) \notin S^{n-1}$ then $x_n = 1$ otherwise $x_n = 0$

In example $n=6$ so take $(6,6)$ from S^3

$(6,6) \in S^3$

$(6,6) \notin S^2$

$x_3 = 1$ so $(P_1 - P_3, w_1 - w_3) = (6-3, 6-3)$

$(6,6) - (5,4) = (1,2) \in S^2$ $= (3,3)$

$(1,2) \in S^1$ → False → do condition fail ($P_i w_i \in S^n$ $P_i w_i \notin S^{n-1}$)

$x_2 = 0$

$(1,2) \in S^1$

$(1,1) \notin S^0 \rightarrow$ true

$x_1 = 1$

Maximum profit is $\sum p_i x_i = p_1 x_1 + p_2 x_2 + p_3 x_3$

$= 1 \times 1 + 2 \times 0 + 5 \times 1$

$= 1 + 5 = 6.$

0/1 knapsack Algorithm

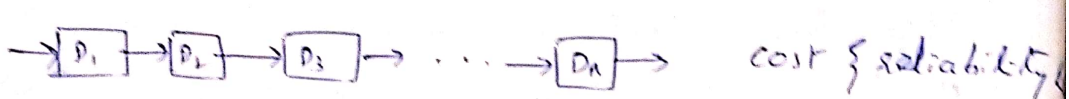
```
DKP(p, w, n, m)
S^0 ← {(0,0)}
for i ← 1 to n-1 do
    S^i_1 ← {(p_i, w_i)(p - p_i, w - w_i) ∈ S^{i-1} | w_i ≤ m}
    S^i ← MERGE-PURGE(S^{i-1}, S^i_1)
repeat
(P_y, w_x) ← last tuple in S^{n-1}
(P_y, w_y) ← (P_i + P_n, w_i + w_n) where w_i is the largest
        in any tuple in S^{n-1} such that w + w_n ≤ m
if P_x > P_y then x_n ← 0
        else x_n ← 1
end if
end DKP.
```

Time complexity for knapsack is $O(n)$

Time complexity for 0/1 knapsack problem is $O(2^{n/2})$

Reliability Design :-

- The problem is to design a system that is composed of several devices connected in series.



n Devices D_i, $1 \le i \le n$, connected in series.

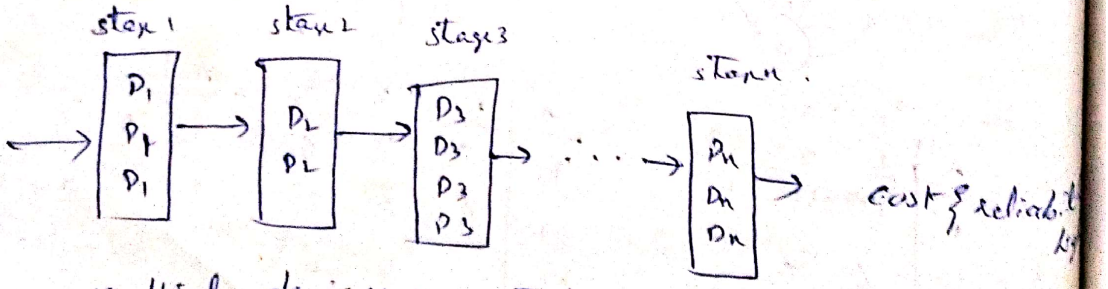
- Let r_i be the reliability of device D_i (r_i is the probability that device i will function properly) Then the reliability of the entire system is $\prod r_i$

Even if the individual devices are very reliable, the reliability of the system may not be very good.

for Eg if $n = 10$, $r_i = .99$, $1 \le i \le 10$, then $\prod r_i = .904$

~~Multiple copies of the~~

- Hence, it is desirable to duplicate devices.
- Multiple copies of the same device type are connected in parallel through the use of switching circuits. The switching circuits determine which devices in any given group are functioning properly. They then make use of one such device at each stage.



Multiple devices connected in parallel in each stage.

- If stage i contains m_i copies of device D_i, then the probability that all m_i have a malfunction is $(1-r_i)^{m_i}$. Hence the reliability of stage i becomes $1-(1-r_i)^{m_i}$. If $r_i = .99$ & $m_i = 2$, the stage reliability becomes .9999.

- Failures of copies of same device may not be fully independent. Let us assume that the reliability of stage i is given by a function $\phi_i(m_i), 1 \le n$. The reliability of the system of stage is $\prod_{1 \le i \le n} \phi_i(m_i)$

$$\prod_{1 \le i \le n} \phi_i(m_i)$$

Our problem is to use device duplication to maximize reliability. The maximization is to be carried out under a cost constraint. Let c_i be the cost of each unit of device i & let c be the maximum allowable cost of the system being designed.

- To solve the following maximization problem:

$$\text{maximize} \prod_{1 \le i \le n} \phi_i(m_i)$$

$$\text{subject to} \sum_{1 \le i \le n} c_i m_i \le c$$

$$m_i \ge 1 \quad \& \quad 1 \le i \le n$$

- A dynamic programming solution can be obtained in manner similar to that used for the knapsack problem.

we may assume that each $c_i > 0$, each m_i must be in the range $1 \le m_i \le u_i$

$$u_i = \left\lfloor \left(c + c_i - \sum_{j=1}^{n} c_j \right) / c_i \right\rfloor .$$

Eg:-

To design a three stage system with device types D_1, D_2, D_3. The costs are \$30, \$15, \$20. The cost of the system is to be no more than \$105. The reliability of each device type is 0.9, 0.8, 0.5

$c_1 = 30 \quad c_2 = 15 \quad c_3 = 20 \quad c = 105$

$r_1 = 0.9 \quad r_2 = 0.8 \quad r_3 = 0.5$

u_i = upper bound $= \left\lfloor \left(c + c_i - \sum_{j=1}^{n} c_j \right) / c_i \right\rfloor$

$u_1 = \lfloor (105 + 30 - (30 + 15 + 20)) / 30 \rfloor = 2.33 = 2.$

$u_2 = \lfloor (105 + 15 - (30 + 15 + 20)) / 15 \rfloor = 3.66 = 3.$

$u_3 = \lfloor (105 + 20 - (30 + 15 + 20)) / 20 \rfloor = 3.$

Reliability function

$\phi_i(m_i) = 1 - (1 - r_i)^{m_i}$

Reliability

$C_1 = 30 \quad C_2 = 15 \quad C_3 = 20$ Cost = Rs 105

$r_1 = 0.9 \quad r_2 = 0.8 \quad r_3 = 0.5$

$$v_i = \left(\left(C + c_i - \sum_{j=1}^{n} c_j \right) / c_i \right)$$

$$v_1 = \frac{(105 + 30 - (30+15+20))}{30} = 2.33 \cong 2$$

$$v_2 = \frac{(105 + 15 - (30+15+20))}{15} = 3$$

$$v_3 = \frac{(105 + 20 - (30+15+20))}{20} = 3$$

Initially $\phi_i(m_i) = 1 - (1 - r_i)^{m_i}$, S_j^i $j = m_i$

Each pair is (r, c) r - reliability, c - cost.

$m_i = j$

Initially $S^0 = \{(1, 0)\}$ $i = 1, j = 1, m_1 = 1$.

$\phi_1(m_1) = 1 - (1 - r_1)^1 = 1 - (1 - 0.9) = 0.9$

$S_1^1 = \{(0.9, 30)\}$ (0.9 and 0+30)

Here reliability (0.9) is multiplied with 1 & $c_1 = 30$ is added to 0

$i = 1, \; j = 2, \; m_1 = 2$

$\phi_1(m_1) = 1 - (1 - r_1)^2 = 1 - (1 - 0.9)^2 = 1 - (0.01) = 0.99$

0.99×1 & $\sum c_i m_i = 60$ $(\because j = 2)$ is added to 0

$S_2^1 = \{(0.99, 60)\}$

S^1 can be obtained by merging S_1^1, S_2^1

$S^1 = \{(0.9, 30), (0.99, 60)\}$.

We can't calculate S_3^1 $\because v_1 = 2$

Illy S_1^2, S_2^2, S_3^2 can be calculated from S^1.

Sol:

$j=1$, $m_2=1$, $i=2$

$\phi_2(m_L) = 1-(1-R_L)^{m_L} = 1-(1-0.8)^1 = 0.8$

$S_1^2 = \{(0.9\times0.8, 30+15), (0.99\times0.8, 60+15)\}$

$= \{(0.72, 45), (0.792, 75)\}$.

for S_2^2, $j=2$, $i=2$, $m_L=2$

$\phi_2(m_L) = 1-(1-R_L)^2 = 1-(1-0.8)^2 = 1-(0.2)^2$

$= 0.96$.

$\therefore j=2$, $2C_2 = 2\times15 = 30$ is added to S^1 pairs

$S_2^2 = \{(0.9\times0.96, 30+30), (0.99\times0.96, 60+30)\}$

$S_2^2 = \{(0.864, 60), (0.9504, 90)\}$

S_3^2, here $i=2$, $j=3$, $m_L=3$

$\phi_2(m_L) = 1-(1-R_L)^{m_L} = 1-(1-0.8)^3 = 0.992$

$\therefore j=3$, $3C_2 = 3\times15 = 45$

$S_3^2 = \{(0.9\times0.992, 30+45), (0.99\times0.992, 60+45)$

$= \{(0.8928, 75), (0.98208, 105)\}$.

$S^2 = \{(0.72, 45)\ (0.792, 75)\ (0.864, 60)$

$(0.9504, 90)\ (0.8928, 75)\ (0.98208, 105)\}$

By purge rule $(0.792, 75)\{(0.9504, 90)$ Cost is more and $(0.98208, 105)$ is removed from S^2

$S^2 = \{(0.72, 45)\ (0.864, 60)\ (0.8928, 75)$

$\}$.

Illy S_1^3, S_2^3, S_3^3 can be calculated from S^2

S_1^3, $i=3, j=1$

$\phi_3(m_3) = 1-(1-\lambda_3)^{m_3} = 1-(1-0.5)^1 = 0.5$

$C_3 = 20$. is added to S^2

$S_1^3 = \{(0.72\times0.5, 45+20), (0.864\times0.5, 60+100),$
$(0.8928\times0.5, 75+20), (0.98208\times0.5, 105+20)\}$

Last tuple is removed from S_1^3, ∵ cost is 125 exceeding the given cost 105.

update as per $S_1^3 = \{(0.36, 64) (0.432, 80), (0.4464, 95)\}$.

S_2^3, $i=3$, $j=2$, i.e $m_3=2$

∵ $j=2$, $2C_3 = 2\times20 = 40$

$\phi_3(m_3) = 1-(1-\lambda_3)^{m_3} = 1-(1-0.5)^2 = 0.75$

$S_2^3 = \{(0.72\times0.75, 45+40) (0.864\times0.75, 60+40)\}$

$= \{(0.54, 85) (0.648, 100)\}$

S_3^3 $i=3$, $j=3$, i.e $m_3=3$

∵ $j=3$, $3C_3 = 3\times20 = 60$

$\phi_3(m_3) = 1-(1-\lambda_3)^{m_3} = 1-(1-0.5)^3 = 0.875$.

$S_3^3 = \{(0.72\times0.875, 45+60)\} = \{(0.63, 105)\}$.

$S^3 = \{(0.36, 65) (0.432, 80) (0.4464, 95) (0.54, 85)$
$(0.648, 100) (0.63, 105)\}$.

Apply pruning rule

$S^3 = \{(0.36, 65)\ (0.432, 80)\ (0.54, 85)\ (0.648, 100)\}$

The Best design has reliability of 0.648 & a cost of 100.

$(0.648, 100)$ pair present in S_2^3, $j=2$, so $m_3 = 2$

The $(0.648, 100)$ pair obtained from $(0.864, 80)$ which is present in S_2^2, $j=2$, so $m_2 = 2$.

$(0.864, 80)$ obtained from $(0.9, 30)$ when S_1^1, $j=1$ so $m_1 = 1$.

$\therefore m_1 = 1,\ m_2 = 2,\ m_3 = 2.$

Travelling Sales problem :-

The sales man should start at a point and travels all the places and come back to starting point. The problem is to minimize the travelling cost.

The main requirement is there should be communication between nodes.

Ex:-

(1) Choose an arbitrary point
(2) Find the next nearest point
(3) Calculate distance between the two nodes
(4) Repeat the steps Till you visit all the cities
(5) Find the length of tour
(6) make all possible tours and take minimum length as the solution of Travelling Sales problem.

Eg

Suppose we have to route a postal van to pickup mail from mail boxes located at n different sites. A 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 is a tour and it should have minimum length (minimum cost).

- Consider a tour to be a simple path that starts and ends at vertex '1'. Every tour consists of an edge 'k' from some $k \in V-1$ & a path from vertex 'k' to vertex '1'.

The path from vertex 'k' to vertex 'i' goes through the each vertex in $V-\{1,k\}$ exactly ones.

If '2' is optimal then the path from k to 1 going through all vertices $V-\{1,k\}$.

Let $g(i,s)$ be the length of a shortest path starting at vertex 'i', $g(1, V-\{1\})$ is the length of an optimal sales person tour.

From the principle of optimality follows that

$$g(1, V-\{1\}) = \min_{2 \le k \le n} \{ c_{1k} + g(k, V-\{1,k\}) \}$$

where c_{1k} is the edge constant from 1 to k.

$g(k, \{V-\{1,k\}\})$ represents the shortest path starting at 'k' visiting all vertices $V-\{1,k\}$ exactly ones & terminating at vertex '1'.

Generalizing.

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

clearly

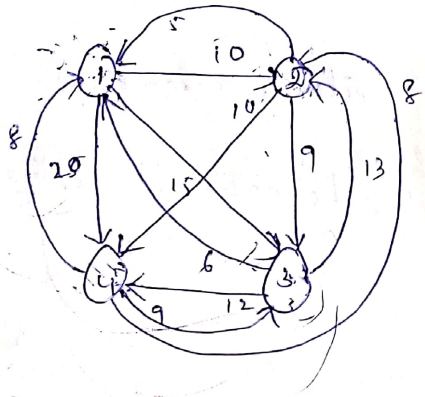
$$g(i,\phi) = c_{i1} \quad , \quad 1 \le i \le n$$

To obtain $g(i,s)$ for all s of size 1.

To obtain $g(i,s)$ for s with $|s|=2$, & so...on.

when $|s| < n-1$, the values of i and s for which $g(i,s)$ is needed are such that $i \neq 1$, $1 \notin s$, & $i \notin s$.

Eg:- Consider the directed graph



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

① $i \neq 1$ ② $1 \notin s$ ③ $i \notin s$

Now compute $i \neq 1$, $1 \notin s$, $i \notin s$.

$g(i,\phi) = c_{i1}$, $1 \leq i \leq n$

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

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

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

Then compute $|s|=1$, $i \neq 1$, $1 \notin s$, $i \notin s$.

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

$|s|=1$ it means set contains only one element.

Before solving this problem we make an assumption that the sales person starts at vertex 1, from that he can move to vertex 2, If he visits vertex 2, from that vertex, he can visit either 3 or 4 vertex

$|s| = 1$ (i) $g(2,3)$

(ii) $g(2,4)$

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

(iii) $g(3,2)$

(iv) $g(3,4)$

From vertex 1, he can visit vertex 4 instead of vertex 3.

In this case from vertex 4, he can visit either 2 or 3.

(v) $g(4,2)$

(vi) $g(4,3)$

$\{3\} - \{3\}$

$$g(2,3) = \min_{j \in 3} \{ c_{23} + g(3,\phi) \}$$

$$= \min\{9+6\} = 15$$

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

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

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

$$g(4,\{2,3\}) = \min_{j \in 2} \{ c_{42} + g(2,\phi) \} = 13.$$

$$g(4,\{3\}) = \min_{j \in 2} \{ c_{43} + g(3,4) \} = \min\{9+6\}$$

Now compute $|y|=2$, $j \neq 1$, $j \notin S$, $1 \notin S$.

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

$= \{ 9+20,\ 10+15 \}$

Now compute $|S|=2$, here set contains two 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 3 or 4 vertex. So determine

$g(2,\{3,4\})$, ($S=\{3,4\}$, since $|S|=2$),

$g(3,\{2,4\})$ & $g(4,\{2,3\})$.

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

$= \min \{ 9+20,\ 10+15 \} = 25$

$g(3,\{2,4\}) = \min \{ c_{32} + g(2,4),\ c_{34} + g(4,2) \}$

$= \min \{ 13+18,\ 12+13 \} = 25$.

$g(4,\{2,3\}) = \min \{ c_{42} + g(2,3),\ c_{43} + g(3,2) \}$

$= \min \{ 8+15,\ 9+18 \} = 23$.

$|S|=3$, here S contains 3 elements. After starting from vertex 1, next he can visit 2 or 3 or 4 vertices.

So determine $g(1,\{2,3,4\})$

$g(1,\{2,3,4\}) = \min \{ c_{12} + g(2,3,4),\ c_{13} + g(3,2,4),\ c_{14} + g(4,2,3) \}$

$= \min \{ 10+25,\ 15+25,\ 20+23 \}$

$= \min \{ 35, 40, 43 \} = 35$.

$1 \to 2 \to 4 \to 3 \to 1$

$10+10+9+6 = 35$

$1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$

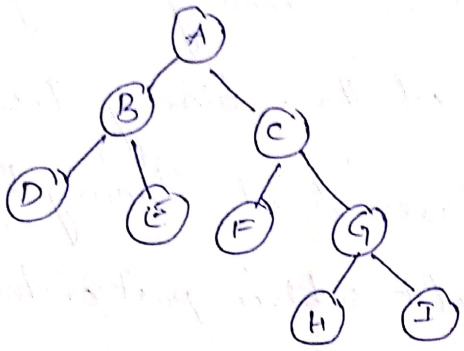
$10 + 10 + 9 + 6 = 35$

Time complexity $= O(n^2 2^n)$

Tree Traversal Techniques

- There are 3 # traversal methods are
(a) preorder (b) Inorder (c) post order

Tree for Traversal :-



preorder Traversal :-

- pre order traversal of a binary tree yields the prefix expression of a binary tree.
- preorder traversal of binary tree
(1) Visits the roots (R)
(2) Traverse the left subtree preorder (L)
(3) Traverses the right subtree preorder (R)
- preorder Traversal of tree yield the following lot

A B D E C F G H I.

Inorder Traversal :-

- Inorder Traversal of a binary tree yields the infix expression of a binary tree.
- The Inorder Traversal of binary tree
(1) Traverses the left subtree (L)
(2) Visits the root (R)
(3) Traverse the right subtree (R)

- The inorder traversal of the Tree yields the following listing of the nodes.

D B E A F C H G I.

postorder Traversal

- Post order traversal of a binary tree yields the postfix expression of the binary tree.
- The post order traversal of binary tree.
  (1) Traverses the left subtree postorder (L)
  (2) Traverse the right subtree postorder (R)
  (3) Visits the root (R)
- The postorder traversal of the tree yields the following listing of the nodes

D E B F H I G C A.

Algorithms

```
treenode = record
{
   Type data;
   tree node *lchild; treenode *rightchild;
}

Algorithm PreOrder(t) binary tree,
{                                  each node of t has lchild, data, rchild
   if t ≠ 0 then
   {
      Visit(t);
      PreOrder(t → lchild); lchild(t)
      PreOrder(t → rchild);
   }
}
```

```
Algorithm InOrder(t)
{
  if t ≠ 0 then
  {
    InOrder(t → lchild);
    Visit(t)
    InOrder(t → rchild);
  }
}

Algorithm PostOrder(t)
{
  if t ≠ 0 then
  {
    PostOrder(t → lchild);
    PostOrder(t → rchild);
    Visit(t);
  }
}
```

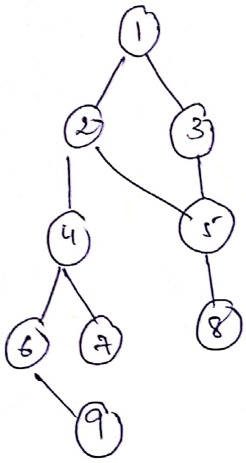
Technique for Graphs :-

- A fundamental problem concerning graphs is the reachability problem. To determine a path in graph $G = (V, E)$ such that, the path starts at vertex v & end at vertex u.

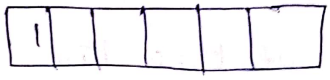
Generally is to determine for a starting vertex $v \in V$ all vertices u such that there is a path from v to u. problem can be solved by starting at vertex v & systematically search the graph G for vertices that can be reached from v. There are two search vertex for this.

Breadth first Search & Traversal

- Breadth first Search will use a queue as a auxiliary structure to hold nodes for future processing.
- In BFS the search proceeds level by level. The algorithm generates possible success of initial node tested then & add them at the end of list.
- BFS will give a gaurentee to find a solution.

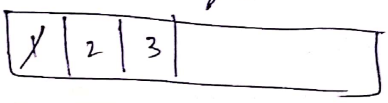


Step 1:-

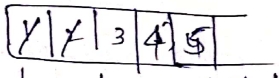


step 2: q : 2,3

The starting node 1 have two child 2,3 the nodes 2 & 3 are added to the queue VISIT 1

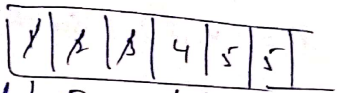


step 3: q : 3,4,5



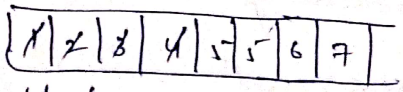
Delete the node 2 & node 2 have child 5, 4 so add 5,4 to queue VISIT 2

step 4: q : 4,5,5



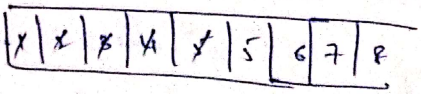
pop 3, the node have child 5, add 5 to the queue VISIT 3

step 5:- q : 5, 5, 6, 7



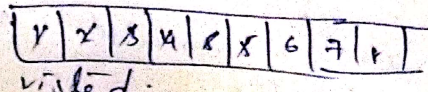
pop 4, the nodes 6, 7 are added to queue VISIT 4.

step 6: q : 5, 6, 7, 8.



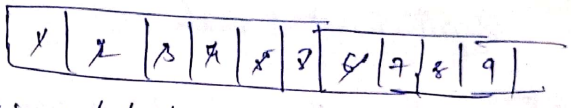
pop 5, the node 8 is added VISIT 5

step 7: q : 6, 7, 8.



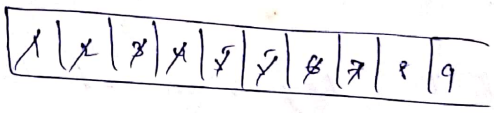
pop 5, the node already visited.

Step 8 : q : 7, 8, 9



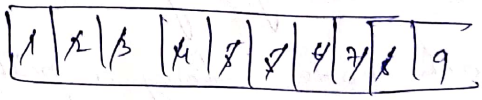
Delete 6, the node 9 is added VISIT 6.

Step 9 : q : 8, 9



pop 7, VISIT 7.

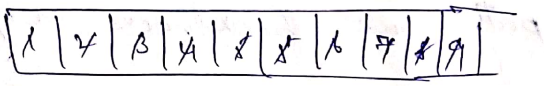
Step 10 : q : 9



pop 8, VISIT 8.

Step 11 :

q : 9



pop 9

VISIT 9

Result : 1, 2, 3, 4, 5, 6, 7, 8, 9

Advantage :- BFS will not get trapped, explored, blind alley.

Disadvantage :- The space complexity will be more,

The time complexity is more.

Algorithm BFS(v)

// Beginning at vertex v. for any node i, visited[i] = 1

// if i has already been visited. visited[] is initialized to zero

{

u := v

visited[v] := 1;

repeat

{ for all vertices w adjacent from u do

{ if (visited[w] = 0) then

{

Add w to q; // w is unexplored

visited[w] := 1;

} }

if q is empty then return; // No unexplored vertex

Delete u from q; //

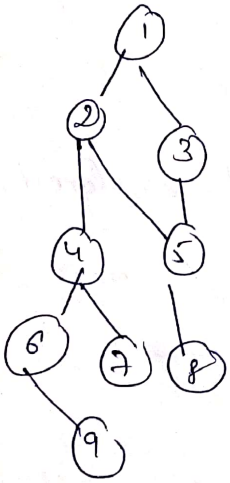
} until (false);

Depth First Search

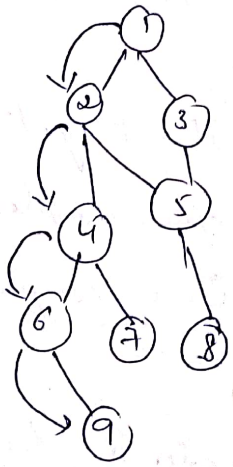
* { In DFS the search proceeds by expanding the initial node, generate possible successors tested them } add them at the beginning of the stack } x

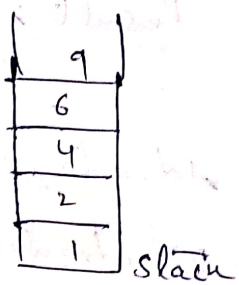
- DFS is begining at a starting node, and examining each node along a path ~~...~~.

After coming to the 'dead end' ; e the end of the path, we backtrack on path until we can continue along another path } so on.



step 1

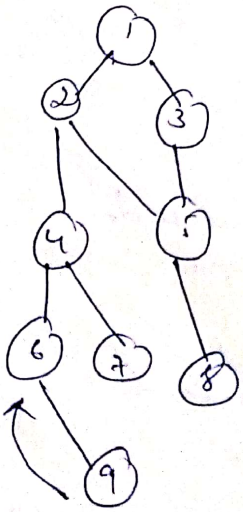


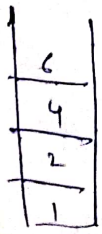


push 1, 2, 4, 6, 9 to stack

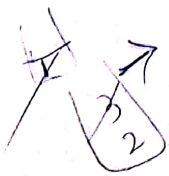
visit 1, 2, 4, 6, 9

step 2



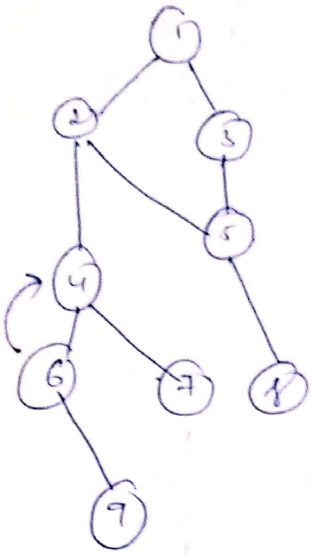


pop 9



For the node 9, there is no unprocessed adjacent nodes.

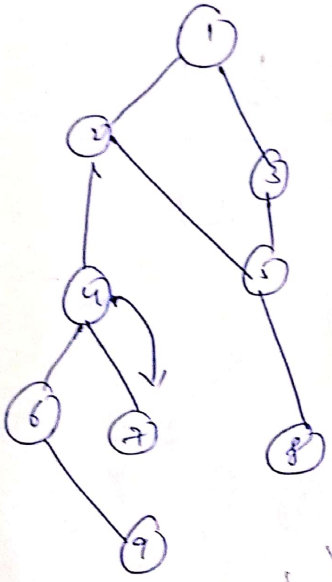
step 3

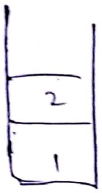


pop 6

For the node 6 no unprocessed adjacent nodes

step 4

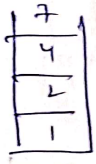




pop 4

The node 4 has adjacent node 7

push 4 & 7 visit 7



step 5

pop 7

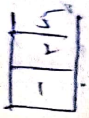
The node 7 has no unprocessed adjacent nodes

step 6

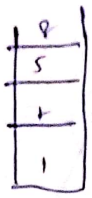
pop 4 , Nodes are already processed

step 7

pop 2 , The node 2 have adjacent node 5 so 2 & 5 are pushed into the stack visits



Step 8



pop 5

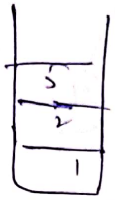
The node 5 have adjacent nodes 8

So 5, 8 are pushed

VISIT 8.

Step 9

pop 8



There is no unprocessed adjacent node

Step 10

pop 5

push 5 & 3 to stack VISIT 3

Step 11

pop 3, 5, 2, 1

No unprocessed adjacent node

RESULT :- 1, 2, 4, 6, 9, 7, 5, 8, 3.

(1) One visiting the node, push each & every node in the stack for future processing

(2) Visit upto end of the path, & pop each and every node & check if there are adjacent nodes. If the nodes is present. push the nodes into the stack for further processing.

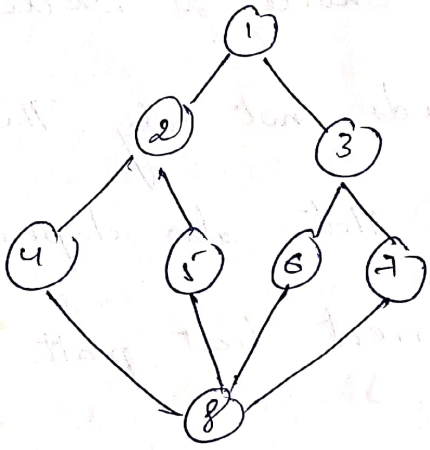
Advantage :- TC & SC is less

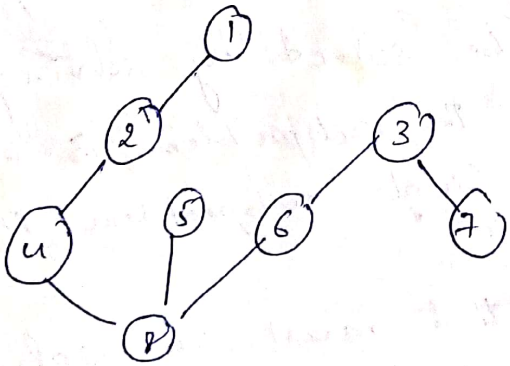
Disadvantage :- possibility of having dead

If end of search is large it may be a time complexity

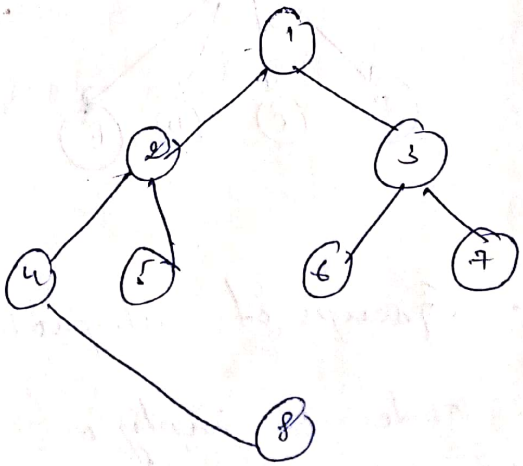
```
Algorithm DFS(v)
{ // initially visited[] = 0
  visited[v] := 1
  for all vertices w adjacent from u do
  {
    if(visited[w]=0) then
      DFS(w);
  }
}
```

Connected Components and spanning trees





DFS



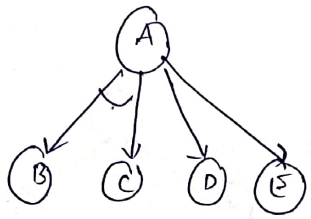
BFS

AND/OR Graph (AOG)

- AND-OR-graph is useful for representing solutions of the problems by decomposing them into smaller independent set of sub problems. Then combine sub solutions to get the solutions of the original problem. This decomposition generates arcs called AND-arcs & OR-arcs.

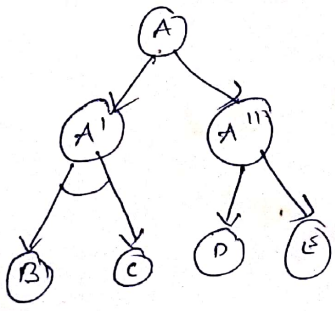
In AND-OR graph the choice of which node to be expended next depends not only the value of most promising node that also depends whether it is the part of current best path from the initial node.

Eg:-



The graph represents a problem can be solved by solving either both the subproblems B and C or the single subproblem D or E.

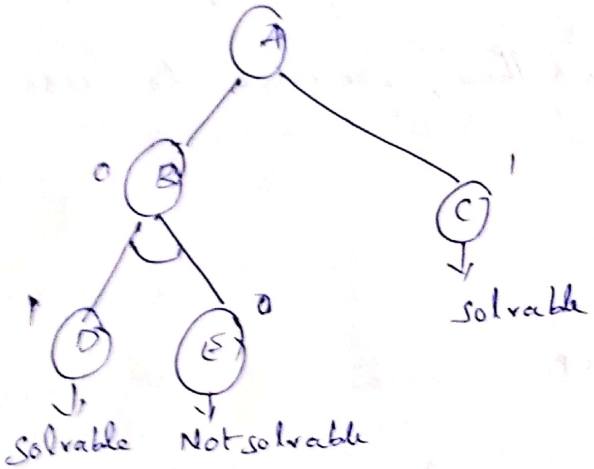
- Groups of sub problems that must be solved in order to imply a solution to the parent node are joined together by an arc going across the respective edges.



By introducing dummy nodes, all nodes can be made to be such that their solution requires either all descendents to be solved or only one decendent to be solved

- Nodes of the first type are called AND nodes & those of the latter type OR nodes.
Nodes A & A'' are OR nodes whereas node A' is an AND node. The AND nodes are drawn with an arc across all edges leaving the node.

Eg



- 'A' is given problem assume that D, C are solvable but E is not solvable. Node B is an AND node, it is not solvable since node E returns 0. Node A is a OR node, so it is solvable since node C returns 1. Thus given problem A is solved

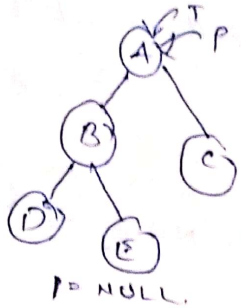
Non Recursive Binary Traversals Algorithms

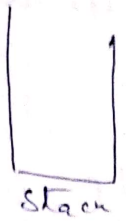
- The procedure for traversals is same for both recursive & non recursive algorithms.

In case of recursive algorithms the stacks is automatically implemented by compiler but in case of non recursive algorithms, we have to use stack explicitly.

Inorder:-

```
Algo Inorder(T)
{ int stack(m), m, TOP
 if T=0 then return
  P ← T;
 TOP ← 0;
 while(P ≠ NULL) do
 {
   TOP ← TOP + 1
   if (TOP > m) then printf("stack is full")
         return;
         endif;
   stack (TOP) ← P;
    P ← LCHILD (P),
  repeat
 }
 if(TOP ≠ 0)
 {
   P ← stack (TOP);
    TOP ← TOP - 1;
  call VISIT(P)
  P ← RCHILD (P);
 }
 else
    exit
 repeat
}// inorder.
```





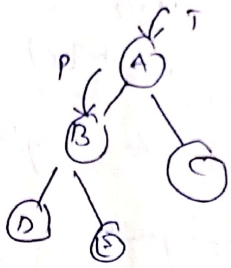
Initially P, T pointing to root node A. Stack is initially empty, start in while associated

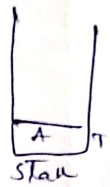
$\Rightarrow$ P = NULL.

TOP ← 0 + 1 = 1 , as is size of stack

stack (TOP) ← P, i.e stack (1) ← A

next P ← left child of (P), so

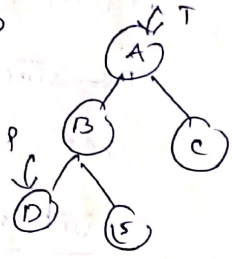


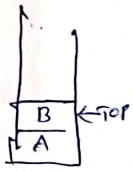


TOP ← 1+1 = 2

stack(2) ← P, stack(1) ← B

next P ← left child of(P), so now P = D





TOP ← 2+1 = 3

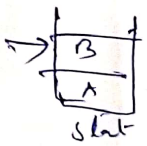
stack(3) ← P, stack(3) ← D.

P ← left child of P. Here P = NULL, while loop terminated

if (TOP ≠ 0), which is True, so

P ← stack (TOP) ; i.e P = stack(3)

TOP = 3 - 1 = 2.



VISIT(P), i.e node D will be printed

P ← right child of (P)

$\Rightarrow$ P = NULL, while loop terminated

P ← stack(2) i.e P pointing to node B.

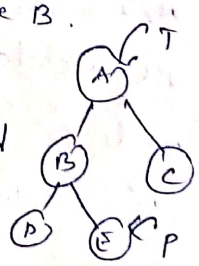
TOP = 2 - 1 = 1

VISIT(P) i.e node B is visited

Next

P ← Right child of (P)

P = E, P pointing to node E

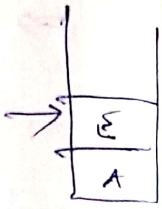


Here $p \neq NULL$, so statements with in while loop are executed

TOP = 1+1 = 2

stack(2) ← p

p ← left child of p i.e p = NULL while loop is terminated



if (TOP ≠ 0) so

p ← stack(2) p = E,

TOP = 2-1 = 1

visit(p) i.e node E is visited

p ← right child of p i.e p = NULL

Again

if (TOP ≠ 0) condition, it is true

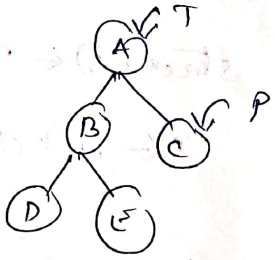
p ← stack(1) i.e p = A,

TOP = 1-1 = 0

visit node A

p ← right child of A i.e p = C

p is pointing to node C



while (p ≠ NULL) so true

TOP = 0+1 = 1

stack(1) ← p, stack(1) = C

p ← LCHILD(p) p = NULL

p ← stack(1) p = C

TOP = 1-1 = 0

node C is visited

Next

p ← RChild(p) i.e p = NULL

while is terminated

if (TOP ≠ 0) false so exit is executed

D, B, E, A, C

Pre order :-

```
Algo PreOrder(T)
{ stack(m), TOP;
 if (T=0) then return // T is empty
 p ← T ; TOP ← 0;                    p = 0
 while (p ≠ NULL) do                 p ≠ NULL
 {
    TOP ← TOP+1                      TOP = 1
  Call visit (p);
  if (Top>m) then print("stack is full") return
  stack (Top) ← RCHILD(p);
     p ← LCHILD(p)
  repeat
 }
 p ← stack(Top);
  Top ← Top - 1
 if (TOP = 0) exit
 repeat
}
```

Post Order :-

```
Algorithm Post-Order(T)
{ int stack(m), TOP;
 if (T=0) then return // T is empty
  p ← T ; TOP ← 0;
 while ( p ≠ NULL)
 {
   TOP ← TOP +1
   call visit(p);
   if (TOP>M) then print("stack is full") return
   stack (Top) ← p;
   p ← LCHILD (p)
 repeat
}
```

```
while(stack(TOP).check = 0)
{
    p ← stack(TOP);
    TOP ← TOP - 1;
    call visit(p);
    repeat
}
    p ← stack(TOP);
    p ← RCHILD(p);
    stack(TOP).check ← 0;
    if (TOP = 0) exit
    repeat
}
```