CHAPTER - I.

Introduction : Algorithm Definition, Algorithm Specification, Performance Analysis - Space Complexity, Time Complexity, Randomized Algorithms.

Algorithm :.

Informal Definition :.

A step by step procedure to solve a problem is called algorithm.

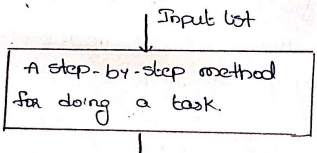


Fig :. Algorithm.

Formal Definition :.

A finite set of instructions to do a particular task is called algorithm.

To evaluate an algorithm we have to satisfy the following criteria.

① Input :.

The algorithm should be given zero (or) more inputs explicitly.

(2) Output :-

One or more quantities are produced as an outcome.

(3) Definiteness :-

Each instruction is clear and free from ambiguous.

(4) Finiteness :-

The algorithm must halt after finite number of steps for all the cases.

(5) Effectiveness :-

Every step in the algorithm should be easy to understand and must be implementable through any of the programming languages.

* Each step of algorithm must be subjected to one or more operations.

* For a computer to operate on the algorithm, certain constraints must be imposed on the operations performed on it.

* The time required to terminate an algorithm must be shortly sensible.

Issues of Algorithm :-

* How to design an algorithm → creating an algorithm
* How to express an algorithm → Definiteness
* How to analyse an algorithm → time and space complexity
* How to validate an algorithm → finiteness.

⇒ Algorithm Specification:-

Algorithm can be described in the following three ways.

① Natural Language like English:-

In this method the instructions must be finite and effective.

② Graphic Representation called Flowchart:-

This method will work well when the algorithm is small and simple.

③ Pseudo code method:-

In this method, we should typically describe algorithms as program, which resembles language like pascal and algol.

⇒ Pseudo code Conventions:-

① Comments are denoted by '//'

② Block of statements are enclosed with '{ }'

③ No need of explicit declaration of variables data type. An Identifier begins with a letter.

④ Compound data types can be formed with records

Ex:- Node. Record.

```
{
    data type-1 data 1;
    :
    data type-n data n;
    node *link;
}
```

Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.

⑤ Assignment of ~~variables~~ values to variables is done using the assignment statement.

<Variable>:= <expression>;

⑥ There are two boolean values TRUE & FALSE

→ Logical operators AND, OR, NOT

→ Relational operators <, <=, >, >=, =, !

⑦ The following looping statements are employed.

for, while and repeat-until.

While Loop:

```
While <condition> do
{
    <statement-1>
      :
    <statement-n>
}
```

For Loop:

```
For Variable:= value-1 to value-2 step stepdo
{
    <statement-1>
      :
    <statement-n>
}
```

repeat-until:

```
repeat
    <statement-1>
      :
    <statement-n>
until <condition>
```

⑧ A conditional statement has the following forms.

→ If <condition> then <statement>

→ If <condition> then <statement-1>
Else <statement-1>

Case statement:

```
Case
{
    <condition-1>: <statement-1>
                 :
    : <condition-n> : <statement-n>
    : else : <statement n+1>
}
```

⑨ Input and output are done using the instructions read and write.

⑩ There is only one type of procedure exist is of the form

Algorithm, the algorithm takes the form,

Algorithm Name(Parameter list)

Ex:- Return maximum of given n numbers.

```
1. Algorithm Max(A, n)
2. // A is an array of size n
3. {
4. Result := A[1];
5. for i:=2 to n do
6. if A[i] > Result then
7. Result := A[i];
8. return Result;
9. }
```

In the above algorithm A,n are parameters Result & i are

local variables.

Ex:② selection sort:

* Suppose we devise an algorithm that sorts a collection of n>=1 elements of arbitrary type.

* A simple solution given by the following

* From those elements that are currently unsorted, find the smallest and place it next in the sorted list.

Algorithm:

1. For i:=1 to n do
2. {
3. Examine a[i] to a[n] and suppose the smallest element is at a[j];
4. Interchange a[i] and a[j];
5. }

→ Finding the smallest element (a[j] and interchanging it with a[i])

→ swap can be solved using the following code.

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

* The first subtask can be solved by assuming the minimum is a[i]; checking a[i] with a[i+1], a[i+2] ...., and whenever a smaller element is found, regarding it as the new minimum. a[n] is compared with current minimum.

* Putting all these observations together, we get the algorithm selection sort.

Selection sort:

Selection sort begins by finding the least element in the list. This element is moved to the front. Then the least element among the remaining element is found out and put into second position. This procedure is repeated till the entire list has been studied.

Ex:. List L: 3, 5, 4, 1, 2

1 is selected : 1, 5, 4, 3, 2
2 is selected : 1, 2, 4, 3, 5
3 is selected : 1, 2, 3, 4, 5
4 is selected : 1, 2, 3, 4, 5

Algorithm:

```
1. Algorithm selection sort (A, n)
2. sort the array a[1:n] into non-decreasing order.
3. {
4. for i:=1 to n do
5. {
6.    j=i;
7.    for k=i+1 to n do
8.        if(a[k]<a[j])
9.        t:a[j];
10.       a[j]:=a[i];
11.       a[i]:=t;
12. }
```

⇒ Recursive Algorithms:

* A Recursive function is a function that is defined interms of itself.

* Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body.

* An algorithm that calls itself is direct recursi

* Algorithm A is said to be Indirect Recursive if it calls another algorithm which inturns calls 'A'

Ex:- ① Towers of Hanoi:

* It is fashioned after the ancient towers of Brahma situl.

* According to the legend, at the time the worl was created, there was a diamond tower (labled A) with 64 golden disks.

* The disks were of decreasing size and were stacked on the tower in decreasing order of type bottom to top.

* Besides these towers there were two other diamond towers (labeled B & C)

* Since the time of creation, Brahman pri has been attempting to move the disks from towe to tower B using tower C, as intermediate storage.

* As the disks are very high, they can be moved only one at a time.

* In addition, at no time can a disk be on top of a smaller disk.

* According to legend, the world will come to an end when the pesist have completed the task.

Solution using Recursion:

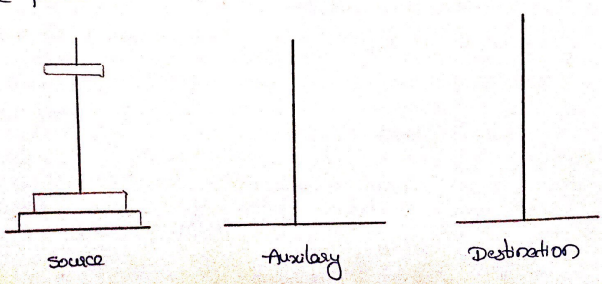
* Assume that the number of disks is 'n'

* To get the largest disk to the bottom of tower B, we move the remaining 'n-1' disk to tower C and move the largest to tower B

* Now we are left with tasks of moving the disks from tower C to B.

* To do this we have towers A & B available

* The fact, that towerss B has a disk on it can be ignored as the disks larger than the disks being moved from towers C and so and disk can be placed on the top of it.



step 1: move disk 1 from source to Destination
Auxilary

step 2: move disk 2 from source to Destination

step 3: move disk 1 from Auxilary to Destination

step 4: move disk 3 from source to Auxilary

step 5: move disk 1 from Destination to source.

step 6: move disk 2 from Destination to Auxilary

step 7: move disk 1 from source to Auxilary

step 8: move disk 4 from source to destination

step 9: move disk 1 from Auxilary to destination

step 10: move disk 2 from Auxilary to source

step 11: move disk 1 from Destination to source

step 12: move disk 3 from Auxilary to Destination

step 13: move disk 1 from source to Auxilary

step 14: move disk 2 from source to Destination

step 15: move disk 1 from Auxilary to Destination.

Algorithm:-

1. Algorithm TowersofHanoi(n, x, y, z)
2. // move the top 'n' disks from tower x to tower y
3. {
   :
   :
4. if(n>=1) then
5. {
6. TowersofHanoi(n-1, x, z, y);
7. write("move top disk from tower "x" to top of tower ",y);
8. TowersofHanoi(n-1, z, y, x);
9. }
10. }

⇒ Performance Analysis:.

There are many criteria upon which we can judge the algorithm. For instance:

① Does it do what we want it do?

② Does it work correctly to the original specifications of task?

③ Is there documentation that describes how to use it and how it works?

④ Are procedures created in such a way that they perform logical sub-functions?

⑤ Is the code readable?

Analysis of algorithms is two types

* Priori Analysis
* Posteri Analysis

Priori Analysis

Analysis of algorithm is done before running the algorithm on any computer machine i.e, before executing algorithm, we will study the behaviour of the algorithm.

It gives an estimate about the running time of the algorithm. It is also known as performance Analysis

Posteriori Analysis:

* It is also called performance measurement
* During the stage of analysis, the target must be identified.
* Algorithm is converted to a program and run on a machine. While algorithm is executed then the information

Kaaaa collected regarding execution time and primary memory requirements is called as posteriri analysis

⇒ Complexity :-

Complexity can be classified into two types

① Space Complexity

② Time Complexity

Space Complexity :-

The complexity can be defined as the amount of space an algorithm requires. The space needed by each of the algorithm is the sum of the following components

① Fixed Part

② Variable Part

① Fixed Part :-

It depends on the characteristics of input and output. That consists of space needed by component variable whose size is dependent on the particular problem instance being involved.

② Variable part :-

The space requirement denoted by S(P) of any algorithm 'P' can be given by as S(P) = C+Sp where 'C' is constant.

S(P) : instance characteristics.

Generally when a program is under execution it uses the computer memory for three reasons. They are

(1) Instruction Space:

It is the amount of memory used to store compiled version of instructions.

(2) Environmental Stack:

It is the amount of memory used to store information of partially executed functions at the time of function call.

(3) Data Space:

It is the amount of memory used to store all the variables and constants.

When we want to perform analysis of an algorithm based on its space complexity, we consider only Data space and ignore instruction space as well as Environmental stack.

That means we calculate only the memory required to store variables, constants, structures etc.

Ex(1):

```
int square (int a)
{
    return a*a;
}
```

In the above piece of code, it requires 2 bytes of memory to store variable 'a' another 2 bytes of memory is used for return value.

total: 4 bytes required.

Eg: ②
```
int sum(int A[], int n)
{
    int sum=0, i;
    for(i=0; i<n; i++)
    sum = sum + A[i];
    return sum;
}
```

n*2 bytes of memory req to store array variable 'a[]'
2 bytes of memory for integer parameter 'n';
4 bytes of memory for local integer variable sum & i
2 bytes of memory for return value.

total = 2n+8
the amount depends on input value of 'n'

③ Time Complexity :-

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Time complexity of a program 'p' can be defined as the sum of compile time and execution time (run time)

T(P) = compile time + runtime

We can consider time complexity in 3 cases

① Best case
② Worst case
③ Average case.

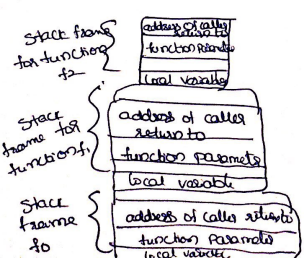
Space complexity for recursive algorithms:-

The space complexity for recursive algorithms depends upon 3 factors

① Formal Parameters
② Local variables
③ Return Address.

Here we need to understand how the stack frames are generated in memory for recursive calls. Sequence When a function 'f1' is called from function 'f0', stack frame corresponding to this function 'f1' is created. This stack frame is kept in the memory until call to function f1 is not terminated.

This stack frame is responsible for saving the parameters of 'f1', and return address of called function f0. Now function f1 calls another function 'f2' stack frame corresponding to 'f2' is also generated and is kept in the memory until call to f2 is made



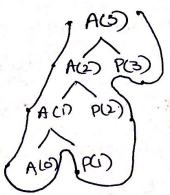
Now when call to function f_2 returns, the stack frame corresponding to 'f_2' is deleted from memory since it is no longer is use required. Same incase of function f_1 and function 'f_0'.

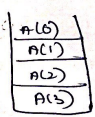
Using this analogy for recursive call sequence, it should follow that maximum number of stack frames that could be present at any point of time is equal to maximum depth of recursion tree.

In the recursion tree, when the call corresponding to leaf node state is getting executed, the call sequence could be represented by the path from root node in recursion tree to that leaf node.

Ex:- ①

```
A(n)
{
  if(n≥1)
  {
    A(n-1);
    P(n);
  }
}
```





$= (n+1)K$ → constant

$= O(n)$

① Best Case:-

If an algorithm takes minimum amount of time to run to completion for a specific set of input, then it is called best case time complexity.

E.g:-

While searching a particular element by using sequential search we get the desired element at first place itself then it is called best case time complexity.

② Worst Case:-

If an algorithm takes maximum amount of time to run to completion for a specific set of input, then it is called worst case time complexity.

(or)

It is maximum amount of time taken by an algorithm to given an output is called as worst case

Eg:-

While searching an element by using linear searching method if desired element we get the desired element at first place else

while searching an element by using linear searching method if desired element is placed at the end of the list then we get worst case time complexity.

③ Average Case:-

The average time taken by an algorithm to run to completion is called average case.

This classification gives the complexity information about the behaviour of the algorithm at a particular instance and of an algorithm on specific in random algorithm.

Time Complexity of an algorithm can be found in three ways:-

(i) Brute force method.

(ii) Step count method

(iii) Asymptotic Notation.

① Brute Force Method:-

Time complexity of a program 'p' can be defined as the sum of compile time and execution time. Let us suppose that we compile a program for once and we run it several times.

Then runtime T_p can be calculated as

T_p = time taken to perform all the operations present in the program (addition, substraction etc.)

As T_p influenced by many other factors and these factors are unknown at the time of conceiving of program, only estimation of T_p is possible.

Let us assume that we know the characteristics of the compiler to be used. Let us determine the number of additions, substractions etc. that are performed by code P

$t_p(x) = c_a Add(x) + c_s Sub(x) + c_m Mul(x) + \dots$

x denotes instance characteristics and c_a, c_s, c_m etc ... represent addition, substraction.

In case of multi user system the execution time depends upon numerous factors such as system load, number of other programs being run on computer at particular instance.

(a) Step Count method:

* A program step is defined as a meaningfull statement of a program that has an execution time that is independent of the instance characteristics

* It is assumed that all the statements have same cost

* We can determine the total number of steps needed by the program by counting the number of steps.

Order of magnitude of an algorithm:

* Each algorithm contains finite number of statements

* Each statement occurs one or more times

* The sum of number of occurances of all statements contained in an algorithm is called order of magnitude of an algorithm.

Ex:

```
for (i=0 ; i<n ; i++)
{
    ..... } 'k' statements.
}
```

Let us assume there are 'k' statements enclosed in the for loop and statements takes one unit of time for execution.

If these k statements are executed 'n' times the execution time is n*k units

Ex: Algorithm sum()

```
{
    read (a,b,c,d);        1 unit
    x=a+b+c+d;             1 unit
    write (x);             1 unit
}                          ------
                           3 units.
```

Ex:- Algorithm sum(a, n)

```
{
    sum := 0.0;          ← 1 unit
    for i := 1 to n do   ← n+1 unit
    sum := sum + a[i];   ← n units
    write (sum);         ← 1 unit
}                        ---------
                         2n+3 units.
```

③ Algorithm Avg()

```
{
    sum := 0.0           ← 1
    Read n;              ← 1
    for i := 1 to n do   ← n+1
    {
        Read num;        ← n
        sum = sum + num; ← n
    }
    Avg = sum/n          ← 1
    write (sum, avg);    ← 1
}                        ---------
                         3n+5 units
```

④ Algorithm mat(a, b, c, m, n)

```
{
    for i := 1 to m do ←m+1
    for j := 1 to n do ←m(n+1)
    c[i, j] = a[i, j] + b[i, j] ← mn
}                        ---------
                         2mn+2m+1
```

⑤ Algorithm Recsum(a, n)

```
                                  ←T(n)
{
    if (n<=0) then                ← 1 unit
    {
        return 0;
    }
    else
    {
        return Recsum(a, n-1)+a[n]; ← T(n-1)+b
    }
}
```

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

3) Asymptotic Notation :-

Asymptotic notation contain 3 types they are as follows

1. Big-oh Notation (O-Notation)
2. omega Notation (Ω-Notation)
3. Theta Notation (Θ-Notation)

1) Big-oh Notation :-

Big-oh notation denoted by 'O' is a method of representing the upper bound or worst case of algorithm's run time.

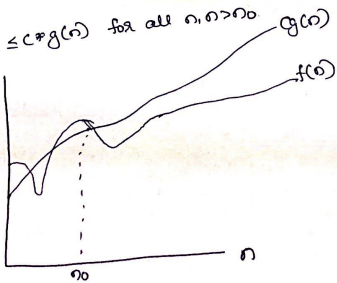
using big-oh notation, we can give longest amount of time taken by the algorithm to complete.

Definition :-

Let f(n) and g(n) are two non-negative functions

The function $f(n) = O(g(n))$ if there exist positive constants n_0 and c such that

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



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

$0 \leq f(n) \leq cg(n)$,

g(n) is an asymptotic upper bound for f(n)

① $f(n) = 2n^2 + 3n + 1$

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

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

$n \geq n_0$

$2n^2+3n+1 <= 1$ false

$2n^2+3n+1 <= 3n$ false

$2n^2+3n+1 <= n^2$ false

$2n^2+3n+1 <= 2n^2$ false

$2n^2+3n+1 <= 3n^2$ true for $n \geq 4$

$f(n) = 3n^2$

$f(n) = O(n^2)$ where $c = 3$ and $n_0 = 4$

Omega Notation:

Omega notation denoted as 'Ω' is a method of representing the bound of algorithm running time using omega notation.

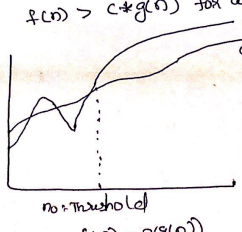
We can denote short amount of time taken by algorithm to complete.

Definition:

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

$f(n) = \Omega(g(n))$ if there exists positive constant c and n_0

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



Q2).

Consider $f(n) = 2n+5$ & $g(n) = 2n$ then $2n+5 \geq 2n$ for $n \geq 1$

$2n+5 = \Omega(n)$ $5n+2 = \Omega(n)$ as $5n+2 >= cn$ for $n >= 1$

$10gn+6 = \Omega(n)$ as $10gn+6 >= 10gn >= 1$

$13n^2+6n+2 = \Omega(n^2)$ as $13n^2+6n+2 >= 13n^2$ for $n >= 1$

$4*2^n+n^2 = \Omega(2^n)$ as $4*2^n+n^2 >= 4*2^n$ for $n >= 1$

observe also that

$5n+2 = \Omega(1)$

$13n^2+6n+2 = \Omega(n)$

$13n^2+6n+2 = \Omega(1)$

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

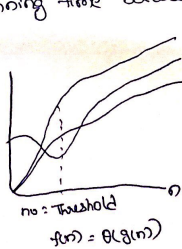
$4*2^n+n^2 = \Omega(2^n)$

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

$4*2^n+n^2 = \Omega(1)$

③ Theta notation :-

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



Let $f(n)$ and $g(n)$ be two non negative functions The function $f(n) = \theta(g(n))$ if there exists positive constants c_1, c_2, and n_0 such that $c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n, n \geq n_0$

g(n) is an asymptotically tight bound for f(n)

Eg:- if $f(n) = 2n+8 > 5n$ where $n \geq 2$; $2n+8 \geq 6n$ where $n \geq 2$ and $2n+8 < 7n$ when $n \geq 2$ Hence $2n+8 = \Theta(n)$ such that constants $c_1 = 5$, $c_2 = 7$ and $n_0 = 2$

Ex:-

① for i:=1 to n
  s;
Time complexity: O(n)

② for i:=1 to n
  for j:=1 to n
  s;
Time complexity: $O(n^2)$

③ i=1, k=1;
while (k<=n)
{
  i++;
  k=k+i;
}
Time complexity: O(√n)

④ for (i=1; i*i<=n; i++)
  s;
Time complexity: O(√n)

⑤ j=1;
while (j<=n)
{
  j=j*2;
}
Time complexity: O(log n)

⑥ for i:=1 to n/2
  for j:=1 to n/3
  for k:=1 to n/4
  s;
Time complexity: $O(n^3)$

⑦ for i:=1 to n ——— n
  for(j=1; j<=n; j=j*2) ——— log n
  s;
Time complexity: O(n log n)

⑧

i=	1	2	3	4	n
k=	1	3	6	10	$\frac{i(i+1)}{2} \leq n$
	1+2	1+2+3			

for i:=1 i<=n i=i+1

4 i=1 i<=n i=i*

Big-oh notation example

Consider function $f(n) = 2n+2$ and $g(n) = n^2$. Then we have to find some constant c, so that $f(n) \leq c * g(n)$. As $f(n) = 2n+2$ and $g(n) = n^2$ then we find c for $n=1$ then

$$f(n) = 2n+2 = 2(1)+2 = 4$$
$$g(n) = n^2 = (1)^2 = 1$$
$$\Big\} \; f(n) > g(n)$$

if $n=2$ then

$$f(n) = 2(2)+2 = 6$$
$$g(n) = 4$$
$$\Big\} \; f(n) > g(n)$$

if $n=3$ then

$$f(n) = 2(3)+2 = 8$$
$$g(n) = (3)^2 = 9$$
$$\Big\} \; f(n) < g(n) \text{ is true.}$$

Hence we conclude that for $n>2$, we obtain

$$f(n) < g(n).$$

Omega notation example :-

consider $f(n) = 2n^2+5$ and $g(n) = 7n$

Then if $n=0$

$$f(n) = 2(0)^2+5 = 5$$
$$g(n) = 7(0) = 0$$
$$\Big\} \; f(n) > g(n)$$

But if $n=1$

$$f(n) = 2(1)^2+5 = 7$$
$$g(n) = 7(1) = 7$$
$$\Big\} \; f(n) = g(n)$$

if $n=3$

$$f(n) = 2(3)^2+5 = 23$$
$$g(n) = 7(3) = 21$$
$$\Big\} \; f(n) > g(n)$$

For $n \geq 3$ we get $f(n) > c * g(n)$.

$2n^2+5 \in \Omega(n)$

$(n^3 \in \Omega(n^2))$

Θ notation example

$f(n) = 2n+8$
$g(n) = 7n$
where $n \geq 2$

$f(n) = 2n+8$
$g(n) = 7n$
$5n < 2n+8 < 7n$
for $n \geq 2$

$c_1 = 5 \quad c_2 = 7$
with $n_0 = 2$

④ Little "oh" Notation (o-Notation)

Little oh notation is denoted as o. It is used to denote proper upper bound that is not asymptotically tight.

Let $f(n)$ and $g(n)$ are two non-negative functions. The function $f(n) = o(g(n))$ if there exist positive constants n_0 and c such that $f(n) < c * g(n)$ for all $n, n > n_0$

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

Eg: The function $3n+2 = o(n^2)$ since $\lim_{n \to \infty} \frac{3n+2}{n^2} = 0$

$f(n) = 3n+2$

$f(n) < cg(n)$

$3n+2 < n^2 \quad n >= 4$

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

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

⑤ Little omega Notation (ω-Notation):

Little omega notation is used to denote proper lower bound that is not asymptotically tight.

Let $f(n)$ and $g(n)$ are two non-negative functions. The function $f(n) = \omega(g(n))$ if there exists constants n_0 and c such that $f(n) > c * g(n)$ for all $n, n > n_0$

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

Eg: The function n^2+6 since $\lim_{n \to \infty} \frac{n}{n^2+6} = 0$

$f(n) = n^2+6$

$f(n) > cg(n)$

$n^2+6 > n \quad n >= 1$

$\lim_{n \to \infty} \frac{n}{n^2+6} = 0 \quad n^2+6 > \omega(n)$

⇒ Recurrence Relations:-

Recurrence Relation is an equation that recursively defines a sequence, one or more initial terms are given. Each further terms of sequence is defined as a function of the preceding terms.

A recurrence relation is the arrangement of a series of values in terms of previous values in the sequence and base values.

Solving recurrence relations:-

Solution to recurrence relation can be obtained using two methods. ① Substitution method

② Master's method.

① Substitution method:-

In this method we consistently guess an asymptotic bound on the solution, and try to prove it by induction. There are two types of substitution methods.

① Forward substitution

② Backward substitution.

① Forward substitution:-

Steps:-

① Take the Recurrence Equation and Initial condition

② Put the initial condition in equation & look for pattern

③ Guess the pattern

④ Prove that guess pattern is correct using induction

Ex:- $T(n) = T(n-1) + n \quad - n>1$

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

$T(1) = 1$

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

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

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

$T(5) = T(5-1) + 5 = 1+2+3+4+5$

↓
sum of n natural numbers

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

Proof : $\frac{n(n+1)}{2}$

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

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

② Backward Substitution method:

Ex:-
```
T(n)
{
  if(n>1)
  {
    return A(n-1)
  }
}
```

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

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

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

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

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

(2) in (1)

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

(3) in (4)

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

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

$\vdots$

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

$T(n-k) = T(1)$ when loop terminates

so $n-k = 1$

$k = n-1$

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

$= T(1)+n-1$

$= O(n)$

③ $T(n-1)+n$ when $n>1$

$T(1) = 1$

$T(n) = T(n-1)+n$ — ①

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

$= T(n-2)+(n-1)$ — ②

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

$= T(n-3)+(n-2)$. ③

② in ①

$T(n) = T(n-2)+(n-1)+n$ - ④

③ in ④

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

$\vdots$

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

$n-(k+1) = 1$

$k = n-2$.

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

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

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

⇒ Master Theorem:

$T(n) = aT(n/b) + \theta(n^k \log^p n)$

a ≥ 1, b > 1, k ≥ 0 and p is real number.

① If $a > b^k$ then $T(n) = \theta(n^{\log_b a})$

② If $a = b^k$

ⓐ If $p > -1$ then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$

ⓑ If $p = -1$ then $T(n) = \theta(n^{\log_b a} \log\log n)$

ⓒ If $p < -1$ then $T(n) = \theta(n^{\log_b a})$

③ If $a < b^k$

ⓐ If $p \geq 0$ then $T(n) = \theta(n^k \log^p n)$

ⓑ If $p < 0$ then $T(n) = O(n^k)$

eg ① $T(n) = 3T(n/2) + n^2$

$a = 3 \quad b = 2 \quad k = 2 \quad p = 0$

$a < b^k \quad 3 < 2^2 = 3 < 4$

$p \geq 0$

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

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

$= \theta(n^2)$

② $T(n) = 2T(n/2) + n\log n$

$= 2T(n/2) + \theta(n^1 \log^1 n)$

$a = 2 \quad b = 2 \quad k = 1 \quad p = 1$

$a = b^k \quad 2 = 2^1 \quad p = 1$

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

$= \theta(n^{\log_2 2} \log^{1+1} n)$

$= \theta(n^1 \log^2 n)$

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

Master's Method:-

$T(n) = aT(n/b) + \theta(n^k \log^p n)$.

$a \geq 1$, $b > 1$, $k \geq 0$ & p is real number.

1) if $a > b^k$, then $T(n) = \theta(n^{\log_b^a})$

2) :

Eg:-

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

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

$a < b^k \Rightarrow 3 < 2^2 = 4$ (True)

$p = 0$

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

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

$= \theta(n^2)$.

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

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

$a = b^k \Rightarrow 2 = 2^1 \Rightarrow$ True

$p = -1$

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

$= \theta(n \log \log n)$ $[\because \log_2^2 = 1]$.

3) $T(n) = 16T(\frac{n}{4}) + n$

$a = 16, b = 4, k = 1$

$a > b^k \Rightarrow 16 > 4^1$ (True)

$T(n) = \theta(n^{\log_4^{16}})$

$= \theta(n^{\log_4^{4^2}})$

$= \theta(n^2 \log_4^4)$

$= \theta(n^2)$.

⇒ Randomized Algorithms:

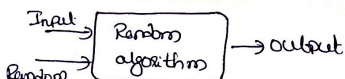
An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm.

Ex: Randomized Quick Sort

we use random number to pick the next pivot element.

To analyze the running time of an algorithm we use theory of probability. probabilistic analysis is used to find out average running over all possible inputs.

Input distribution should be known to apply probabilistic analysis. Some part of algorithm behaviour is randomized, for analysis and design of algorithms randomness is very frequently used and these algorithms are called randomized algorithms.



In these algorithms a random bits generated through pseudo random generator are used as an auxiliary input to get the good performance in average case. The performance of an algorithm will be determined by the random input and it is called expected runtime. The worst case is ignored as probability of its occurrence is very less.

CHAPTER - II

Divide and Conquer :-

General method, applications,- Binary Search, Merge Sort, Quick sort, Strassen's matrix multiplication.

⇒ General method :-

Divide and conquer is a technique which divides the problem into smaller units. After dividing the problems, the technique tries to solve the smaller units.

Once the smaller units are solved, the solutions of smaller units are combined to get the final solution of the problem.

If the input size to a problem is complex, then we say that the problem is large or complex. small problem is a problem which can be solved with one or two operations.

The divide and conquer principle has three steps

Step 1 : Divide :-

Dividing the given problem into smaller sub problems. Also each of these subproblems is almost of the same size

Step 2 : Conquer :-

These sub problems are solved independently in a successive manner.

Step 3 : Combine :-

Combining all the solution of sub problems to get the solution for the original problem.

* If the sub problems are large enough then divide and conquer is reapplied.

* The generated problems are usually of same type of the original problem. Hence recursive algorithms are used in divide and conquer strategy.

⇒ control abstraction for divide and conquer

```
Algorithm DC(P)
{
    if P is very small then return solution of P
    else
    {
        Divide(P) and obtain P1, P2, ... Pk, k>1
        Apply DC to each sub problem;
        return combine ( DC(P1), DC(P2) ... DC(Pk));
    }
}
```

The computing time of above procedure of divide and conquer is given by the recurrence relation

$T(n) = b(n)$ n is small

$T(n) = T(n_1) + T(n_2) + \dots + T(n_k) + f(n)$ where n is sufficiently large where

$b(n)$ is the time taken by the algorithm to give solution when n is small and

$T(n)$ is the time taken by the algorithm DC on any inputs of size n.

$f(n)$ is the time taken by the algorithm for dividing the problem P into smaller instances and combining the solutions to sub problems.

The following recurrence relation given the time complexity of problems that come under divide and conquer algorithms.

$$T(n) = \begin{cases} T(1) \cdots \cdots n=1 \\ aT(n/b) + f(n) \cdots n>1 \end{cases}$$

Applications:

1) Binary Search:

* The pre requisite for Binary search is, elements must be sorted either in non-decreasing or non-increasing order.
* Let the elements are sorted in non decreasing order, if we want to search for the element say x,
* Then we first divide the list at mid (low + high/2), x is compare with mid element.

→ If x is equals to mid element then search terminate, otherwise two sublists get created.
→ If x is greater than mid element then right sublist is considered and x is searched in the right sublist.
→ Otherwise x is less than the mid element, x is searched in the left sublist.

To search an element using binary search method, divide and conquer strategy is used. If we consider P is the number of elements and if there is only one element then return 1. that means problem P is small enough & can't be split.

If P i.e, the no of elements are more than one then we divide P into 2 sublists, and try to solve each list seperately As soon as desired element is found in the list

the search terminates successfully.

Non recursive algorithm:.

```
BinarySearch (A, N, key)
low ← 1
high ← N
Found ← False.
Repeat while (low ≤ high) and (Not Found)
mid ← (low+high)/2
If (key = A[mid])
Then
        Found ← True
location ← mid
else
If (key < A[mid])
Then    high ← mid-1
else
        low ← mid+1
Return.
```

Recursive Algorithm for Binary Search:.

```
Algorithm RBinarySearch (a, low, high, x)
{
    If (low = high) then
    {
      If (x = a[low]) then return 1;
      else
         return -1;
    }
    else
    {
        mid := (low+high)/2;
        If (x = a[mid]) then return mid;
        else If (x < a[mid]) then
```

return BinarySearch(a, low, mid-1, x);

else return Binary Search (a, mid+1, high, x);

}

}

The recurrence relation for Binary Search is

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

$T(n) = a \quad n=1$

For n=1 we have only one comparision i.e., constant time $T(n) = a, \ n=1$

Time complexity for this algorithm is

	Best	Average	Worst
Successfull search	O(1)	O(logn)	O(logn)
unsuccessfull search	O(logn)	O(logn)	O(logn)

Eg: element to be searched is 32

18	32	35	48	62	75
0	1	2	3	4	5

low = 0 high = 5 mid = (0+5)/2 = 2

32 < 35

low = 0 high = mid-1 = 1

mid = (0+1)/2 = 0

18 > 32

low = mid+1 = low = 1 high = 1 (1+1)/2 = 1

32 = 32 successful.

Analysis

$T(n) = 2T[\frac{n}{2}] + 1$ ①

$T(n/2) = 2T[\frac{n}{4}] + 1$ ②

$T[\frac{n}{4}] = T[\frac{n}{8}] + 1$ ③

② in ①

$T[n] = T[\frac{n}{4}] + 1 + 1$

$T[n] = T[\frac{n}{4}] + 2$ ④

③ in ④

$T[n] = T[\frac{n}{8}] + 3$

$= T[\frac{n}{2^3}] + 3$

$T(n) = T[\frac{n}{2^k}] + k$

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

$n = 2^k$

$\log n = k \log 2$

$\log n = \log + T[\frac{2^k}{2^k}]$

Merge Sort :-

The merge sort is a sorting algorithm that uses the divide and conquer strategy. In this method division is dynamically carried out.

merge sort on an input array with n elements consists of 3 steps.

Divide :- Partition array into two sublists S_1 and S_2 with n/2 elements each.

Conquer :- Recursively sort S_1 and S_2

Combine :- merge S_1 and S_2 into a unique sorted group.

merge sort has two algorithms merge sort and merge. merge sort divide the array into two equal halfs irrespective of elements. This procedure continues until sub array is of size 1. Initially there are n elements in an array which are not sorted After division we have having n sorted sub arrays of size 1, because array of size 1 is a sorted array.

Algorithm merge will merge two sorted sub arrays into one sorted array. In merge sort we are not sorting elements, by the virtue of division of elements sub array becomes sorted. Here sorting takes place when combining the arrays

Algorithm :-

① i ← low
② j ← mid + 1
③ k ← high
④ Repeat while (i ≤ mid) and (j ≤ high)
⑤ if (c[i] ≤ c[j])

⇒ Recursive merge sort Algorithm

① Read N

② Repeat for I = 1, 2 ... N

③ Read [x[I]]

④ Call mergesort (x, 1, N)

⑤ Exit

Procedure mergeSort (x, low, high)

① If (low < high)

Then

mid = [(low+high)/2]

call mergeSort (x, low, mid).

Call mergeSort (x, mid+1, high).

call merge (x, low, mid, high).

② Return

procedure merge (x, low, mid, high)

① i ← low

② j ← mid+1

③ k ← high

④ Repeat while (i ≤ mid) and (j ≤ high)

⑤ If (x[i] ≤ x[j]

then

B[k] ← x[i]

i ← i+1

k ← k+1

else

B[k] ← x[j]

j ← j+1

k ← k+1

⑥ Repeat while (i ≤ mid)

B[k] ← x[i]

i ← i+1

k ← k+1

⑦ Repeat while (j ≤ high)

B[k] ← x[j]

j ← j+1

k ← k+1

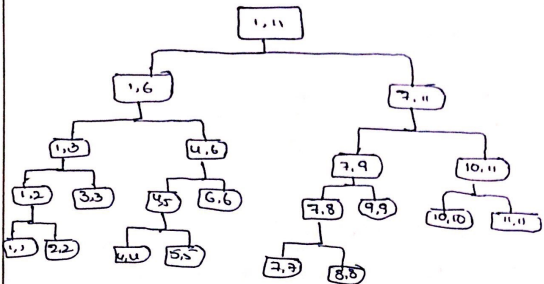
⑧ Repeat for k ← low, low+1, ... high

x[k] ← B[k]

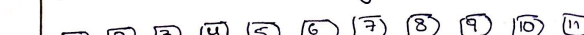
⑨ Return.

Eg:- Let us assume there are 11 elements to be sorted using merge sort technique where low = 1 and high = 11. Now the array is divided into two equal halfs at mid values, left subarray has low = 1, high = mid = 6 and right subarray has low = mid + 1 = 7, high = 11.

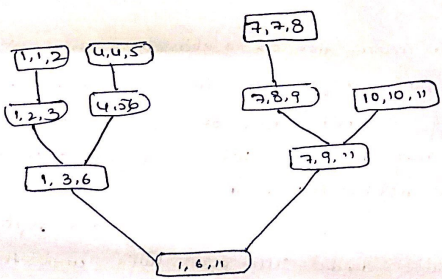
The left sub array and right subarray are further divided into equal halfs until the sub array of size 1. This position is showed in the following fig:



Initially we have an array of size n after partition we have n sub arrays of size as shown below



Now all sorted subarrays are merged using merge algorithm The tree calls of merge algorithm is shown below

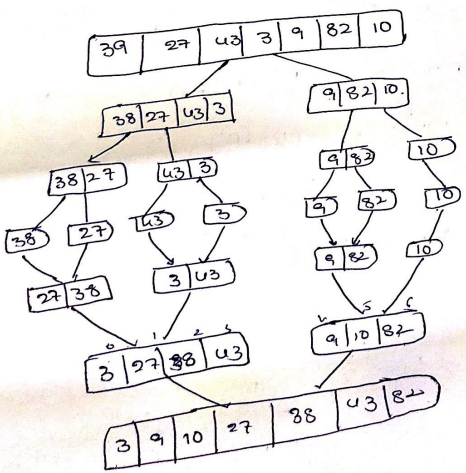


First sub array 1 and sub array 2 are merged to form a sorted sub array represented by (1,1,2) where the first parameter is low, second parameter is mid

and third parameter is high. next this array is merged with sub array 3 to form a sorted sub array by (1,2,3). next subarray 4 and sub array 5 are merged to form a sorted sub array represented by (4,4,5). next this subarray merged with sub array 6 to form a sorted sub array represented by (4,5,6). Now sorted sub array (1,2,3) and sorted subarray (4,5,6) are merged to form sorted subarray (1,3,6)

Similarly sorted sub arrays 7,8,9,10,11 are merged to form sorted sub array (7,9,11). Finally two sub arrays (1,3,6) and (7,9,11) are merged to form sorted array (1,6,11)

Ex:



⇒ Time Complexity of merge sort :

Here array is divided into two equal half's which is independent of elements. Because of the above reason Best case, worst case, and Average case are same.

Recurrence Relation for merge sort is

$T(n) = a \quad n=1$

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

By using substitution procedure $n = \frac{n}{2}$

$T(n) = 2T(n/2) + n$ — ①

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

$T(\frac{n}{4}) = 2T(\frac{n}{8}) + \frac{n}{4}$ — ③

sub ② in ①

$T(n) = 2[2T(\frac{n}{4}) + \frac{n}{2}] + n$

$= 2^2 T(\frac{n}{2^2}) + n + n$ — ④

sub ③ in ④

$T(n) = 2^2[2T(\frac{n}{8}) + \frac{n}{4}] + 2n$

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

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

$= 2^k T(\frac{n}{2^k}) + kn$

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

$n = 2^k$

$\log n = k$

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

$= O(n \log n)$

Space complexity:-

merge sort requires 2n locations. The additional n locations are required for merging two sorted sub lists. Stack space required by the merge sort for the use of recursion. since merge sort splits each array into equal sized sub arrays. The maximum depth of stack is 'log n'. The additional space required is 'n+ log n', n for auxiliary array and log n stack space.

⇒ Quick Sort :-

Quick sort is a sorting algorithm that uses the divide and conquer strategy. The three steps of quick sort as follows.

Divide :- Rearrange the elements and split the array into two subarrays based on pivot element. Each element in the left sub array is less than or equal to the pivot element and each element in the right subarray is greater than or equal to the element.

conquer :-

Recursively sort the two sub arrays. They are further divided until the sub array is of size 1

combine :-

combine all the sorted elements in a group to form a sorted list.

⇒ Algorithm :-

1. Read N
2. Repeat for i:=1,2...,N
3. Read x[i];
4. call Quick(x,1,N)
5. Repeat for i:=1,2,... N
6. write(x[i])
7. Exit.

```
Procedure Quick (x, low, high)
if (low < high)
then
        call partition (x, low, high, position)
        call Quick (x, low, position-1);
        call Quick (x, position+1, high);
    }
}
procedure partition (x, low, high, pos)
1. pivot = x[low];
2. i ← low+1;
3. j = high;
4. Repeat while (i < j)
   {
5. Repeat while (x[i] ≤ pivot) and (i<low)
   i = i+1
6. Repeat while (x[j] > pivot)
   j ← j-1
7. if (i<j) then
   {
       t ← x[i]
       x[i] ← x[j]
       x[j] ← t.
   }
8. x[low] ← x[j]
9. x[j] ← pivot
10. position ← j
11. Return.
```

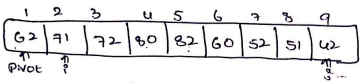
The partition function is called to arrange the elements such that all the elements that are less than pivot are the left side of pivot and all the elements that are greater than pivot are all at the right of pivot.

Example:

62, 71, 80, 82, 60, 52, 51, 42

Pivot = x[low] = 62

low=1, high=9, i=low+1=2 j=high=9



x[i] <=pivot x[71]<=62 F

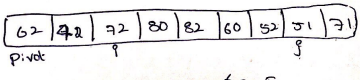
x[i]>pivot 42>62 F

swap x[i] and x[j]

x[i]<=pivot 42<=62 T

x[i]<pivot 71<62 F

x[j]>pivot 71>62 T



x[i] <=pivot 72<62 F

x[j]>pivot 51>62 F

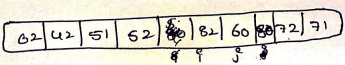
swap x[i], x[j]

x[i] <=pivot 51<62 T i++

x[i] <=pivot 80<=62 F

x[j]>pivot 52>62 F

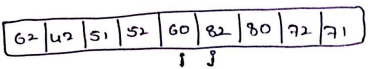
swap x[i] x[j]



x[i] <=pivot 82<62 F

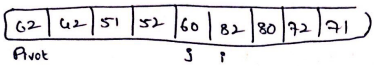
x[j]>pivot 60>62 F

swap x[i] x[j]



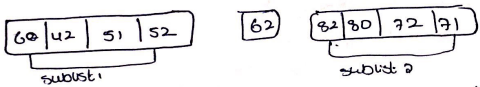
x[i] < pivot 60 < 62 T

x[j] > pivot 82 > 62 T

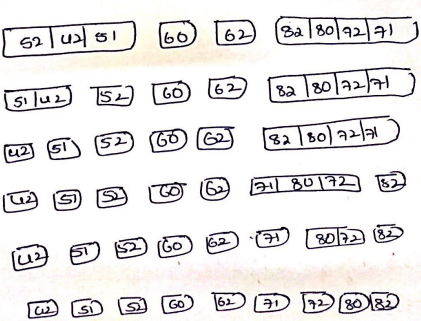


i < j F

x[j] and pivot are swapped. Now the given list is divided at pivot into two sublists.



Now we will consider 60 as pivot for sublist1 and repeat above steps to place pivot at proper position.



Analysis of quick sort algorithm:-

The running time of quick sort depends on whether partition is balanced or unbalanced, which inturn depends on which elements on an array to be sorted are used for partitioning.

A very good ~~array~~ partition splits an array up into two equal sized arrays. A bad partition on the other hand splits an array up into two arrays of very different sizes

The worst partion puts only one element in one array and all elements in the other array.

If partition is balanced quicksort runs faster otherwise quicksort runs very slow.

Best case time complexity:-

The best case of quick sort will happen if each partitioning stage divides the array exactly in half. If the procedure partition produces 2 subarrays of size $n/2$.

The recurrence relation is then

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

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

$$= 1 \quad n = 1$$

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

$$T(n/2) = 2T\left(\frac{n}{4}\right) + \left(\frac{n}{2} - 1\right) \quad \text{--- (2)}$$

$$T(n/4) = 2T\left(\frac{n}{8}\right) + \left(\frac{n}{4} - 1\right) \quad \text{--- (3)}$$

② in ①

$$T(n) = 2\left[2T\left(\frac{n}{4}\right) + \left(\frac{n}{2}-1\right)\right] + (n-1)$$

$$2\left(\frac{n-2}{2}\right) = n-2$$

$$T(n) = 4T\left(\frac{n}{4}\right) + (n-2) + (n-1)$$

$$= 4T\left(\frac{n}{4}\right) + (2n-3) \quad \text{--- ④}$$

③ in ④

$$T(n) = 4\left[2T\left[\frac{n}{8}\right] + \left(\frac{n}{4}-1\right)\right] + (2n-3)$$

$$= 8T\left(\frac{n}{8}\right) + \left(\frac{n-4}{4}\times 4\right) + 2n-3$$

$$= 8T\left(\frac{n}{8}\right) + (n-4) + (2n-3)$$

$$= 8T\left(\frac{n}{8}\right) + 3n - 7$$

$$T(n) = 2^3 T\left(\frac{n}{2^3}\right) + 3n - 7$$

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

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

$$n = 2^k$$

$$k = \log n.$$

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

$$= n + n\log n - n + 1$$

$$= n + n\log n - n + 1$$

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

//.

Ex:-

10 20 30 40 50 60 70
pivot i j

10 20 30 40 50 60 70
Pivot i j

10 20 30 40 50 60 70
Pivot i j

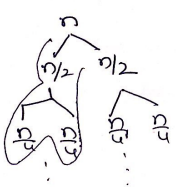
10 20 30 40 50 60 70
Pivot i j

10 20 30 40 50 60 70
pivot i j

10 20 30 40 50 60 70

Space complexity of quick sort:

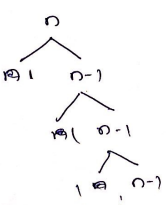
Best case



$= O(n \log n)$

every time half part of the elements recursive calls are stored in stack.

worst case



$= O(n^2)$

⇒Strassen's Matrix multiplication:

matrix multiplication:

Let A and B be two $n \times n$ matrix. The product matrix C = AB is calculated by using the formula

$$C(i,j) = \sum_{k} A(i,k)\, B(k,j)$$ for all i and j between 1 & n

The time complexity for the matrix multiplication is $O(n^3)$. Divide and conquer method suggests another way to compute the product of $n \times n$ matrix.

To apply the divide and conquer n must be the power of two. otherwise enough rows and columns of zeros can be added to both A & B. so that resulting dimension is power of two. This process is called padding of zeros.

If n=2 then the following algorithm is used to compute matrix multiplication operation for the elements of A&B.

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

If n>2, then the elements are partitioned into sub matrix $n/2 * n/2$ - since n is the power of 2 these product can be recursively computed using same formula.

This algorithm will continually applying itself to smaller sub matrix until n become suitable small (n=2) so that product is compute directly

The formula are

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

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

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

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}.$$

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

To compute AB using the equation we need to perform 8 multiplication of $n/2 * n/2$ matrix and from 4 addition of $n/2 * n/2$ matrix.

since $n/2 \times n/2$ matrix can be added in cn^2 for some constant c. The overall computing time $T(n)$ of the resulting divide and conquer algorithm is given by the sequence

$$T(n) = b \quad n <= 2$$

$$T(n) = 8T(n/2) + cn^2 \quad n > 2$$

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

```
Algorithm mm(A, B, n)
{
    if (n<=2)
    {
        C11 = A11 B11 + A12 B21
        C12 = A11 B12 + A12 B22
        C21 = A21 B11 + A22 B21
        C22 = A21 B12 + A22 B22
    }
    else
    {
        mid = n/2
        mm(A11, B11, n/2) + mm(A12 B21, n/2)
        mm(A11, B12, n/2) + mm(A12 B22, n/2)
        mm(A21, B11, n/2) + mm(A22, B11, n/2)
        mm(A21, B12 n/2) + mm(A22, B21, n/2)
    }
}
```

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

multiply matrix A and B using divide and conquer

$$A = \begin{bmatrix} 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$C = A \times B$

matrix A and B are partitioned into sub matrices of size $n/2 \times n/2$

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

matrix A dimension is 4×4 which is divided into 4 sub matrix of size 2×2

matrix B dimension is 4×4 which is divided into 4 sub matrices of size 2×2

$$A_{11} = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} \quad A_{12} = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} \quad A_{21} = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} \quad A_{22} = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$$

$$B_{11} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad B_{12} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad B_{21} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad B_{22} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$A = \begin{bmatrix} \begin{vmatrix} 2 & 2 \\ 2 & 2 \end{vmatrix} & \begin{vmatrix} 2 & 2 \\ 2 & 2 \end{vmatrix} \\ \begin{vmatrix} 2 & 2 \\ 2 & 2 \end{vmatrix} & \begin{vmatrix} 2 & 2 \\ 2 & 2 \end{vmatrix} \end{bmatrix} \quad B = \begin{bmatrix} \begin{vmatrix} 1 & 1 \\ 1 & 1 \end{vmatrix} & \begin{vmatrix} 1 & 1 \\ 1 & 1 \end{vmatrix} \\ \begin{vmatrix} 1 & 1 \\ 1 & 1 \end{vmatrix} & \begin{vmatrix} 1 & 1 \\ 1 & 1 \end{vmatrix} \end{bmatrix}$$

$C = A \times B$

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

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

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

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

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

$$c_{11} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} + \begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} i & j \\ k & l \end{bmatrix}$$

$$c = \begin{bmatrix} \begin{bmatrix} s & s \\ s & s \end{bmatrix} & \begin{bmatrix} s & s \\ s & s \end{bmatrix} \\ \begin{bmatrix} s & s \\ s & s \end{bmatrix} & \begin{bmatrix} s & s \\ s & s \end{bmatrix} \end{bmatrix}$$

Time complexity :-

$$T(n) = 1 \quad n \leq 2$$

$$8T\left(\frac{n}{2}\right) + n^2 \quad n > 2$$

using master theorem.

$$a = 8 \quad b = 2 \quad k = 2$$

$$a > b^k \quad : 8 > 4$$

$$T(n) = O(n^{\log_b a}) = O(n^{\log_2 8}) = O(n^3)$$

⇒ Strassen's matrix multiplication :-

matrix multiplication are more expensive than the matrix addition. we can attempt to reformulate the equation for c_{ij} so as to have fewer multiplication and possibly more addition.

Strassen showed that 2x2 matrix multiplication can be accomplished in **7 multiplication and 18 addition** or subtractions.

* Divide :- Divide matrices into sub matrices : A_0, A_1, A_2 etc
* Conquer :- use a group of matrix multiply equations
* Combine :- Recursively multiply submatrices and get the final result of multiplication after performing required additions or substraction

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

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

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

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

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

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

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

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

$C_{11} = S_1 + S_4 - S_5 + S_7$

$C_{12} = S_3 + S_5$

$C_{21} = S_2 + S_4$

$C_{22} = S_1 + S_3 - S_2 + S_6$

$C_{11} = S_1 + S_4$

$= A_{11}(B_{11} - B_{12}) + (A_{11} + A_{12}) \times B_{22}$

$= A_{11}B_{11} - A_{11}B_{12} + A_{11}B_{22} + A_{12}B_{22}$

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

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

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

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

$C_{22} = A_{21}B_{12} + A_{22}B_{22}$

Time complexity for Strassen's matrix multiplication

Recurrence Relation

$$T(n) = 7T\left(\frac{n}{2}\right) + cn^2 \quad \text{---} (1)$$

$$T\left(\frac{n}{2}\right) = 7T\left(\frac{n}{4}\right) + c\left(\frac{n}{2}\right)^2 \quad \text{---} (2)$$

$$T\left(\frac{n}{4}\right) = 7T\left(\frac{n}{8}\right) + c\left(\frac{n}{4}\right)^2 \quad \text{---} (3)$$

$$T(n) = 7\left[7T\left(\frac{n}{4}\right) + c\left(\frac{n}{2}\right)^2\right] + cn^2 \quad \text{---} (4)$$

(1) in (4)

$$T(n) = 7\left[7\left[7T\left(\frac{n}{8}\right) + c\left(\frac{n}{4}\right)^2\right] + c\left(\frac{n}{2}\right)^2\right] + cn^2 \quad \text{---} (5)$$

(3) in (5)

$$T(n) = 7^3 T\left(\frac{n}{8}\right) + 7c\left(\frac{n}{4}\right)^2 + 7c\left(\frac{n}{2}\right)^2 + cn^2$$

$$T(n) = 7^3 T\left(\frac{n}{2^3}\right) + cn^2\left(\frac{7}{4}\right) + cn^2\left(\frac{7}{4}\right)^2 + cn^2$$

$$T(n) = 7^3 T\left(\frac{n}{2^3}\right) + cn^2\left[\left(\frac{7}{4}\right)^2 + \left(\frac{7}{4}\right) + 1\right]$$

$$T(n) = 7^k T\left(\frac{n}{2^k}\right) + cn^2\left[\left(\frac{7}{4}\right)^{k-1} + \cdots + \left(\frac{7}{4}\right)^2 + \left(\frac{7}{4}\right) + 1\right]$$

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

$$n = 2^k$$

$$T(n) = 7^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + cn^2\left[\left(\frac{7}{4}\right)^{\log_2 n - 1} + \cdots + \left(\frac{7}{4}\right)^2 + \left(\frac{7}{4}\right) + 1\right]$$

$$T(n) = 7^{\log_2 n} T(1) + cn^2\left[\left(\frac{7}{4}\right)^{\log_2 n - 1} + \cdots + \left(\frac{7}{4}\right)^2 + \left(\frac{7}{4}\right) + 1\right]$$

$$T(n) = n^{\log_2 7} + cn^2\left[\left(\frac{7}{4}\right)^{\log_2 n - 1} + \cdots + \left(\frac{7}{4}\right)^2 + \left(\frac{7}{4}\right) + 1\right]$$

$$T(n) = n^{\log_2 7} + cn^2\left[\frac{\left(\frac{7}{4}\right)^{\log_2 n} - 1}{\frac{7}{4} - 1}\right]$$

$$T(n) = n^{\log_2 7} + cn^2\left[\frac{\left(\frac{7}{4}\right)^{\log_2 n} - 1}{\frac{3}{4}}\right]$$

$$T(n) = n^{\log_2 7} + cn^2\left[\frac{4}{3}\left(\frac{7}{4}\right)^{\log_2 n} - \frac{4}{3}\right]$$

$$T(n) = n^{\log_2 7} + \frac{4}{3}cn^2\left(\frac{7}{4}\right)^{\log_2 n} - \frac{4}{3}cn^2$$

$$T(n) = n^{\log_2 7} + \frac{4}{3}cn^2\left(\frac{7^{\log_2 n}}{4^{\log_2 n}}\right) - \frac{4}{3}cn^2$$

$$T(n) = n^{\log_2 7} + \frac{4}{3}cn^2\left(\frac{7^{\log_2 n}}{n^2}\right) - \frac{4}{3}cn^2$$

$$T(n) = n^{\log_2 7} + \frac{4}{3}c\left(7^{\log_2 n}\right) - \frac{4}{3}cn^2$$

$$T(n) = n^{\log_2 7} + \frac{4}{3}c\left(n^{\log_2 7}\right) - \frac{4}{3}cn^2$$

$$T(n) = n^{\log_2 7}\left(1 + \frac{4}{3}c\right) - \frac{4}{3}cn^2$$

$$T(n) = O\left(n^{\log_2 7}\right)$$

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

$$= n^{\log_2 7} + n^2 \left[n^{\log_2 4 - \log_2 n} \right]$$

$$= n^{\log_2 7} + n^2 \left[n^{2 - \log_2 n} \right]$$

$$= n^{\log_2 7} + n^{2 + 2 - \log_2 n}$$

$$= n^{\log_2 7} + n^{4 - \log_2 n}$$

$$= n^{\log_2 7} + n^{\log_2 n^4 - \log_2 n}$$

$$= n^{\log_2 7} + n^{\log_2 \frac{n^4}{n}}$$

$$= n^{\log_2 7} + n^{\log_2 n^3}$$

$$= n^{\log_2 7} + n^{3 \log_2 n}$$

$$= n^{\log_2 7} + n^{\log_2 8}$$

$$= 2n^{\log_2 7}$$

$$= 2n^{2.81}$$

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

using masters theorem

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

$$a = 7 \quad b = 2 \quad k = 2$$

$$a > b^k \Rightarrow 7 > 4$$

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

$$= n^{\log_2 7}$$

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

UNIT - II

Disjoint set operations, union and find algorithms, AND/OR graphs, connected components and spanning trees, Bi-connected components.

Backtracking:

General method, applications - The 8-queen problem, sum of subsets problem, graph coloring, Hamiltonian cycles

⇒ Disjoint set Operations:

A set is a collection of ordered elements. It is assumed that the elements that are present in the sets are numbers. The represented sets are assumed to be pairwise disjoint.

If S_i and S_j are two sets then these sets must not have any common elements ($S_i \cap S_j = \phi$). A partition is a collection of disjoint sets.

Suppose there are 14 elements, the elements are partitioned into 4 disjoint sets.

$S_1 = \{2, 8, 10, 12\}$ $S_2 = \{1, 3, 5, 7\}$ $S_3 = \{11, 13, 14\}$ $S_4 = \{4, 6, 9\}$

Partition $P = \{S_1, S_2, S_3, S_4\}$

S_1 S_2 S_3 S_4

Fig: Tree Representation of sets

Disjoint set union:-

When we perform union operation on two disjoint sets S_i and S_j, it results in with a new set $S_i \cup S_j$ which contains all the elements that are represented in S_i and S_j. Since all sets are disjoint, there are no elements common in both sets. S_i and S_j are replaced with set $S_i \cup S_j$

Union(i,j):-

We pass in the two trees with roots i and j adopting the convention that the first tree becomes the sub-tree of second. The statement P[i]:=j or P[j]:=i accomplish the union

Suppose that we wish to obtain the union of S_1 and S_2. Since we have linked the nodes from children to parent, we simply make one of the trees a subtree of the other. Thus $S_1 \cup S_2 = \{1,2,3,5,7,8,10,12\}$. $S_1 \cup S_2$ could then have one of the representations of the following

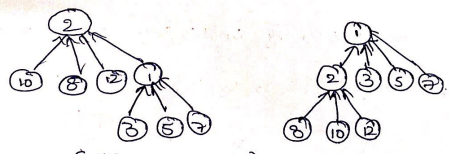


Fig: Union of sets

Find(i):- Given the element i, find the set containing i. Thus, 13 is in set S_3 and 12 is in set S_1.

⇒ Representation of sets:-

Sets can be represented in three ways.

(1) Tree representation

(2) Data representation

(3) Array representation

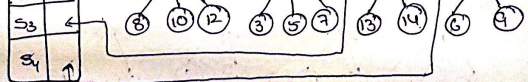
(1) Tree Representation of sets:-

In trees links are given from the parent to the children, but in the implementation of set operation nodes are linked from the children to the parent.

(2) Data Representation of sets:-

The data representation for S_1, S_2, S_3 and S_4 may be of the form as shown in fig. Union operation on two sets is accomplished by assigning the parent field of one of the roots to another root.

In this two pointers are used, one pointer is with root which points to the set name and another pointer is kept along with each sets name which points to the root of the tree.



Roots of the trees are used to identifying sets in union and find algorithms.

Simple Union Algorithm:-

union (i,j) with roots i and j are joined by the function union(i,j). After union operation root of second tree becomes the root and the first tree becomes its subtree. The statement x[i] := j accomplishes above task.

Algorithm Simple union (i,j):-

```
Algorithm Simple union (i,j)
{
    x[i]:=j;
}
```

Algorithm Simple find(i):-

```
Algorithm Simplefind (i)
{
    while(x[i]>0) do i:=x[i];
    return i;
}
```

If we have to perform operations like

union(1,2), union(2,3), union(3,4) . . . . union(n-1,n)

union(1,2) 

union(2,3) 

union(3,4) 

Similarly after union(n-1, n) results in

This sequence will give you degenerate tree as shown in above fig.

Analysis:-

Since the time taken for union is constant.

The n-1 unions can be processed in time O(n).

Time complexity = O(n).

Time taken for find operation is O(n)

The sequence of find operations find(1), find(2), find(3), ..., find(n)

The n finds can be processed in time $O(\sum_{i=1}^{n} i) = O(n^2)$

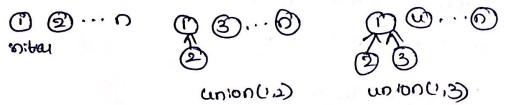
Drawback:-

Depth is more to findout the element in degenerate tree.

To improve the performance of our union and find algorithms by avoiding the creation of the degenerate trees, for this we can use a weighting rule for union(i,j).

Weighting rule for union(i,j):

If the number of nodes on the tree with root i is less than the tree with root 'j' then make j the parent of i otherwise make i the parent of j



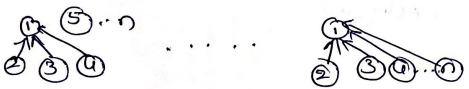


Fig:- Tree obtained using weighted rule

Algorithm for weighted union(i,j):

```
Algorithm WeightedUnion(i,j)
{
                    // x[i] := count[i], x[j] := count[j].
    temp := x[i] + x[j];
    if (x[i] > x[j]) then
    {
        x[i] := j;
        x[j] := temp
    }
    else
    {
        x[j] := i;
        x[i] := temp;
    }
}
```

*For implementing the weighting rule, we need to know how many nodes are there in every tree

*For this we maintain a count field in the root of every tree

ie root node

count[i] → number of nodes in the tree.

Time required for the above algorithm is O(1) + time for remaining unchange.

$= O(1) + \log_2 m$

$= O(\log_2 m)$

Collapsing rule:

If j is a node on the path from 'i' to its root and p[j] ≠ root[i], then set p[j] to root[i].

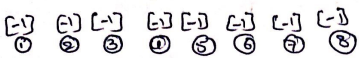
Algorithm for Find with collapsing rule:

```
CollapsingFind(i)
{
    r := i;
    while(p[r] > 0) do r := p[r];
    while(i ≠ r) do
    {
        s = p[i];  p[i] := r;  i := s;
    }
    return r;
}
```

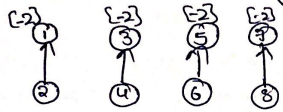
The time complexity becomes O(1)

Example of Find Algorithm with collapsing rules.

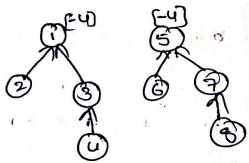
(a) Initial height -1 trees



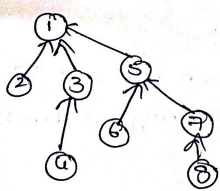
(b) Height -2 trees following union(1,2) (3,4) (5,6) (7,8)

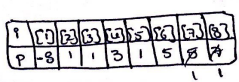


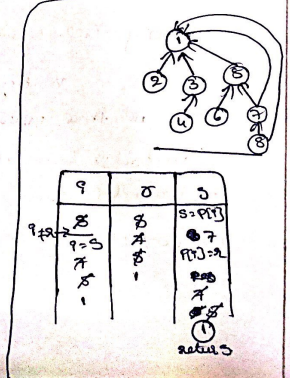
(c) height -3 trees following union(1,3) & (6,7)



(d) height :4 trees following union [1:5] ---







i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
P	+8	1	1	3	1	5	5	7

while [P[x] > 0] do x = P[x]

x = ?

Find (8)

x = 8

P[8] > 0 do x = 7

P[7] > 0 do x = 5

P[5] > 0 do x = 1

P[1] > 0 ✗

while (? ≠ x) do j = P[x] x = P[j] i = j

i = 8 ≠ 1 do j = 7 x = 5 i = 7

while (7 ≠ x) do j = 5 x = 1 i = 5

while (5 ≠ 1) do j = 1 x = 1 i = 1

1 = 1 ✓

So the root of 8 = [1]

⇒ Binary Tree traversal Techniques.

Visiting each node once is called binary tree. There are three types of binary tree traversal algorithms.

1. In-order traversal
2. Pre-order traversal
3. Post-order traversal.

⇒ Graph :-

Graph is defined as set of vertices and edges i.e., G=(V,E). There are two types of graph.

① Directed Graph

② Undirected Graph

There are two types

If the edges of the graph are not having direction then it is called undirected graph as shown in fig.

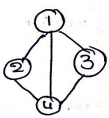
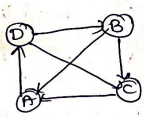


fig: undirected graph.

If the edges of the graph have direction then the graph is called directed graph as shown in fig



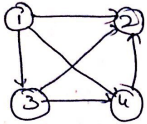
Graphs can be represented in memory by using arrays and linked list. Array representation is called adjacency matrix. & linked list representation is called adjacency list.

Adjacency ~~list~~ Matrix :-

A two-dimensional square matrix is used to represent a graph, whose size is nxn where n is the number of vertices in the graph.

The contents of adjacency matrix are 0 & 1, if an edge is present between two vertices then its value is 1 and other contents are zero.

Ex:-



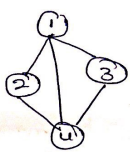
	1	2	3	4
1	0	1	1	1
2	0	0	0	0
3	0	1	0	1
4	0	1	0	0

Fig:- Adjacency matrix.

⇒ Adjacency Lists:

Linked lists are used to represent a graph. The number of linked lists required are n to represent a graph having n vertices.

Ex:-



→ Four linked lists are required

→ For vertex 1, the adjacency vertex are 2, 3, 4

→ For vertex 2, the adjacency vertex are 1, 4, 3 and so on

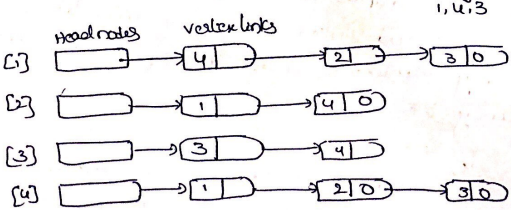


Fig:- Adjacency list.

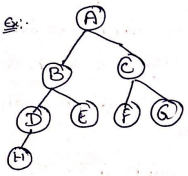
⇒ Graph Traversal Algorithms:

① Breadth First Traversal

② Depth First Traversal.

① Breadth First Traversal:

Ex:



BFS(A) = A — 1

BC — 2

DEFG — 3

H — 4

⇒ ABCDEFG

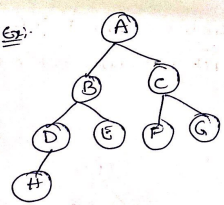
Time & space complexity when there are n vertices & e edges are

$T(n,e) = \theta(n+e)$

$S(n,e) = \theta(n)$ //

② Depth First Traversal:

Ex:



DFS(A) will give

A B D H

E

C F G

⇒ A BDHECFG

The time complexity when there are n vertices and e edges are

$T(n,e) = \theta(n+e)$

$S(n,e) = \theta(n)$ //

⇒ Spanning Tree:

A tree is a connected acyclic Graph. All the trees are graph but all the graphs are not trees. A tree is a connected graph without cycles.

Assume there is an undirected graph G, a spanning tree is a subgraph of G which contains all the vertices of G and subset of edges of Graph G.

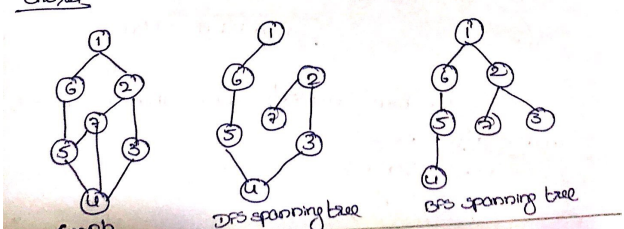
If there are 'n' vertices in Graph G then spanning tree contains n-1 edges.

A spanning tree of a graph 'G' is any tree that includes every vertex in the graph, it is a subgraph of 'G' containing no circuit

If spanning tree is constructed using DFS then it is called DFS spanning tree.

If it is constructed using BFS then it is called BFS spanning trees.

⇒ An edge in a spanning tree is called 'tree edge'. An edge in the graph that is not in spanning tree is called 'chord'



Applications of spanning tree:

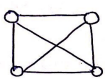
* spanning trees are very important in designing efficient routing algorithms.

* spanning trees have wide applications in many areas, such as network design.

→ The time complexity of this algorithm is clearly $O(n)$, where 'n' is the number of edges.

→ The number of spanning trees in the complete graph K_n is $\boxed{n^{n-2}}$

Eg:



$= 4^{4-2}$

$= 16$

Fig: complete graph.

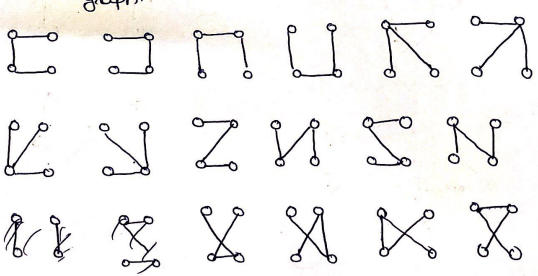


Fig: spanning trees for the above graph.

There are two basic algorithms for finding minimum-cost spanning trees, and both are greedy algorithms.

→ Prim's Algorithm.

→ Kruskal's Algorithm.

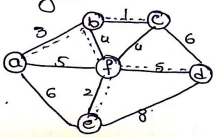
⇒ Prim's Algorithm:-

Start with any one node in the spanning tree, and repeatedly add the cheapest edge, and the node it leads to, or for which is not already in the spanning tree.

→ Kruskal's Algorithm:-

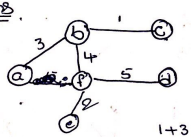
Start with no nodes, or edges in the spanning tree, and repeatedly add the cheapest edge that does not create a cycle.

Ex:-



Starts from b

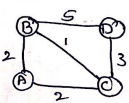
Prims

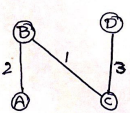


1+3+4+2+5
= (15)

Kruskals example

Ex:-





Sum of edges = 1+3+2 = 6

⇒ Connected Components:-

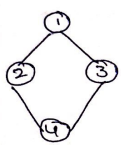
Connected Graph: A Graph is connected if for any two vertices there is a path between them.

If a graph G is not connected, its maximal connected subgraphs are called connected components of G. There are two types of connected components namely

(i) Strongly connected components

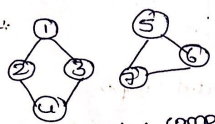
(ii) Bi-connected components.

Ex:-



Connected Graph

Ex:-



connected components.

(1) Strongly connected Graphs:- Components:-

There is a connectivity issue for a directed graph. Two nodes u and v of a directed graph are connected i.e., $u \to v$, $v \to u$. This property or relation partitions vertex set V into disjoint sets known as strongly connected components.

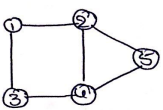
Ex: Strongly connected components.

⇒ Bi-connected Components:-

Maximal sub graph of the given graph which is bi-connected is said to be bi-connected component of the graph.

A graph is said to be bi-connected if it does not contain articulation point and bridge.

Ex:-



(a) Biconnected Graph.

⇒ Identification of Articulation Point:-

(i) The easiest method is to remove a vertex and its corresponding edges one by one from graph G and test whether the resulting graph is still disconnected or not. The time complexity of this will be $O(V(V+E))$

(ii) Another method is to use depth first search in order to find the articulation point after performing depth first search on the given graph we get a 'DFS' tree.

While buiding the DFS tree we number outside each vertex. These numbers indicate the order in which a depth first search visits the vertics. These numbers are called as depth first search numbers.

while buding the DFS tree we can classify the graph into 4 catagories:

(i) Tree Edge:: It is an edge in depth first search tree.

(ii) Back Edge:: It is an edge (u,v) which is not in DFS tree and v is an ancestor of u, it basically indicates a loop.

(iii) Forward Edge:: An edge (u,v) which is not in search tree and u is an ancestor of v.

(iv) Cross Edge:: An edge (u,v) not in search tree and v is neither an ancestor nor a descendents of u

$$low(u) = \min\{depthfirstnumber(u),$$
$$\min\{low(w) \mid w \text{ is a child of } u\},$$
$$\min\{depthfirstnumber[w] \mid (u,w) \text{ is a back edge}\}$$

where low(u) is the lowest depth first number that can be reached from u using a path of descendents followed by almost one back edge.

The vertex u is an articulation point of u is child of w such that.

$$L(w) \geq \text{depth-first number}[u]$$

⇒ Identification of Bi-connected components:.

(i) A bi-connected graph G=(V,E) is a connected graph which has no articulation points.

(ii) A bi-connected component of a graph G is a maximal bi-connected subgraph that means it is not contained in

any largest biconnected sub graph of G.

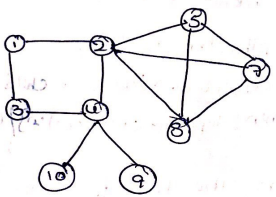
(3) Some key observation can be made in regard to biconnected components of graph.

(i) Two different bi-connected components should not have any common edges.

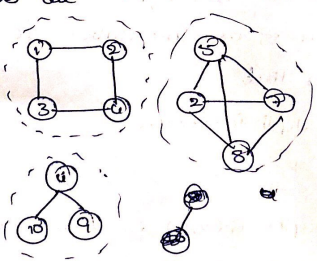
(ii) Two different bi-connected components can have common vertex.

(iii) The common vertex which is attaching two or more biconnected components must be an articulation point of G.

Ex:-



The articulation points are 2, 4, 5 Hence biconnected components are



Biconnected components of G.

⇒ And/OR Graphs:

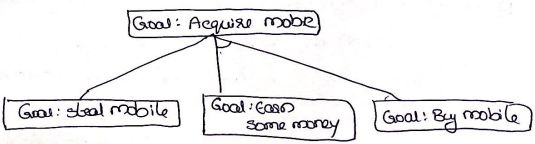
The AND-OR Graph is one also known as AND-OR trees. A solution to problem can be achieved by decomposing the problem into a set of smaller problems and solving the subproblems.

And Node:- All of its successor nodes has to be solved.

OR Node:- one of its successor node has to be solved.

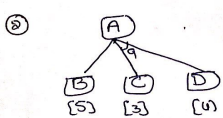
* And arcs are indicated with a line connecting all the components.
* In the process of decomposition AND arcs are generated.
* many successor nodes can be pointed by AND arc.
* To get a solution all successor nodes must be solved.

Ex:-



In fig. the parent node Goal: Acquire mobile can be achived in two ways ① steal mobile

② Earn some money and buy mobile.



The above problem has two solutions: (i) B

(ii) C&D

The cost incurred to solve a node is represented by its 'f' value.

f value of node 'B' is 5

node 'c' is 3

node 'D' is 4

f value will be these for only terminal nodes combination of these terminal nodes gives 'f' value of the parent node.

Every operation has a cost, for simplicity it is assumed that all operations have unique cost. Each edge with a single successor has a cost 1.

Nodes having multiple successor have cost 1 for each one components.

In the above example node 'C' has lowest cost if we select that we must select: c+d+2

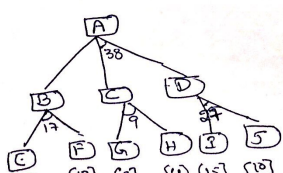
$= 3+4+2 = 9$

we have added 2 because of operating cost of 2

If we select B the total cost : (B+1)

$= 5+1 = 6.$

so we have to choose B.



G = 3 GH is pair ✓ G+H+2 = 9

B is best solution

EF pair E+F+2 = 17

G has minimum but we expand E+F

IJ pair I+J+2 = 27

B = 17

C+D = 38+2 = 38

Back Tracking :- General method, applications - n queen problem, sum of subset problem, graph colouring, hamiltonian problem.

Back Tracking

General method:-

* Back Tracking is one of the most general algorithm design techniques.

* This algorithms are used for obtaining an optimal solution satisfying some constraints.

* The desired solution must be expressible as an n-tuple $(x_1, \ldots x_n)$ where x_i is choosen from some finite set S_i

* It is used to find a vector which minimizes or maximizes the criterion function $P(x_1, \ldots x_n)$

Advantage:-

The advantage of this algorithm is once we know that a partial vector $(x_1, \ldots x_i)$ will lead to an optimal solution that $(m_{i+1} \ldots m_n)$ possible test vectors may be ignored entirely.

Back Tracking is an application of recursion. Implementation of backtracking becomes easy by using recursion. because stack is maintained by recursion.

The solution of backtracking problem can be represented by a tree on which a modified depth first search is performed to provide one or all posible solutions to given the problem.

The solutions to the problems that are to be solved by the backtracking technique may require to satisfy the set of constraints.

The constraints are classified into two types

they are

(a) implicit constraints

(b) explicit constraints

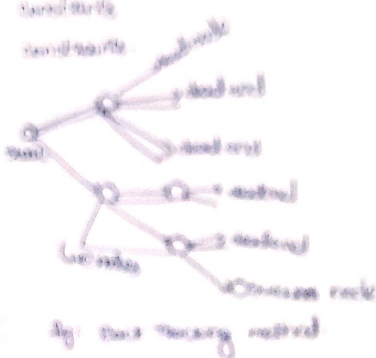


fig: Data mining method

(a) Implicit constraints

The rules that determine which of the tuples in the solution space i can actually satisfy the criterion function are called as implicit constraints

(b) explicit constraints:

The rules that restrict each x_i to take values only from a given set are called as explicit constraints

Ex:- $x_i \geq 0$ OR $x_i = \{$all non-negative real no's$\}$

$x_i = 0$ or 1 OR $x_i \geq 0\}$

$l_i \leq x_i \leq u_i$ OR $x_i = \{a : l_i \leq a \leq u_i\}$

Algorithm:-

```
Algorithm Backtracking(n)
{
    k=1;
    while (k ≠ 0) do
    {
        if (there remains all untried
            x[k] ∈ T(x[1],[2] .... x[k-1] and Bk(x[1]..x[k])
                                    is true) then
        {
        if (x[1]..x[k] is the path to the answer node)
          Then write (x[1:k]);
            k=k+1;
        }
        else k=k-1;
    }
}
```

T(x[1]...x[k-1]) is all possible values of x[k] gives that x[1]...x[k-1] have already be choosen. All solutions are generated in x[1:n] and printed as soon as they are determined.

B[x[1]...x[k]] is a boundary function which determines the elements of x[k] which satisfy the implicit constraint

The efficiency of the above backtracking algorithm depends on 4 factors.

(i) the time to generate the next x_k

(ii) the number of x_k satisfing the explicit constraints

(iii) The time for the bounding functions B_k

(iv) The number of x_k satisfying the B_k

The problems which are solved using this method are (a) N-queens problem

(b) Sum of subsets

(c) Graph coloring

(d) Hamiltonian cycle.

⇒ N-Queens Problems :-

The n-queens problems is to place n-queens on an nxn chessboard in such a way that no two queens attack each other i.e., no 2 queens to be placed on same row, same column and same diagonal.

For writing an algorithm to solve n-queens problem the input is 'n' & output is n-tuple.

Solution :-

* The solution vector $x(x_1, \cdots x_n)$ represents a solution in which x_i is the column of the i^{th} row where i^{th} queen is placed.

* First, check if any two queens are in same row next check that no two queens are in same column.

* The function, which is used to check these two conditions, is $[i, x[i]]$, which gives position of the i^{th} queen, where i represents the row and $x(i)$ represents the column position.

* We have to check no two queen are in it diagonal If two queens are in positions (i,j) and (k,l) then two queen lie on the same diagonal, if and only if $|j-l| = |i-k|$.

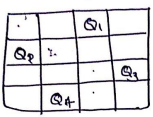
→ For example when n=4 then the problem becomes 4-Queens problem.

→ When n=8 then the problem becomes the 8-queens problem

⇒ 4-Queens Problem:-

* The 4-queens problem is to place 4-queens on an 4x4 chessboard in a such a way that no two queen attack each other.

The solution of 4-queen problem is shown in fig.



The output is x=(2,4,1,3) and this vector x indicates the positions of 4-queens problem.

Brute Force Approach:-

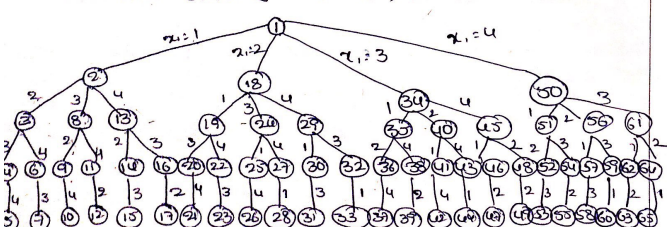
* To solve n-queens problem using brute force approach requires the testing of n!, n-tuples.

* For 4 queen problem where n=4, 24, 4-tuples are to be tested.

* Therefore this approach has a time complexity of O(n!) and this time complexity is not partially acceptable.

* We solve this problem by systematically searching a solution in a solution space.

n=4 {1,2,3,4} 4! (4-tuples)



Backtracking approach:

The above tree has two types of nodes they are

(i) promising nodes.

(ii) non-promising nodes.

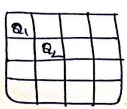
* promising node is a node that indicates there is a chance of finding an answer by proceeding further.

* non promising node is a node from which when we proceed there is no chance of reaching an answer node.

ex:- node 3 in the above tree is a non promising node, we reach node 3 from root via node 2.

Reaching from node 1 to node 2 is represented in a partial solution vector as $X = (1, - - -)$. then move node 3 updates the partial vector as $X = (1, 2 - -)$

This partial vector indicates that first queen at cell (1,1) & second queen at cell (2,2) as shown in fig.



Irrespective of the elements x_3, x_4 the partial vector can't lead to an answer.

Back Track approach:-

By considering the above example in backtracking approach we can back track from node 3

Backtracking from node 3 has an advantage of skipping two permutations effectively.

* For backtracking the bounding function is used.

* The above tree represents the solution space for 4queen problem we have 24, 4-tuple vectors.

* By searching this tree using DFS we can construct vector 'x' element by element.

* When one element is added to vector x, we test whether proceeding in this direction has a chance of leading to the solution.

* If we find that there is no chance of leading to this solution we back track from that node.

⇒ Steps to generate the solution:-

1. Initialize x array to zero
2. place first queen in k=1 in the first row
3. To find the column position start from value 1 to n
4. If k=1 then x(k)=1. so (k, x(k)) will give the position of the k^{th} queen. Here we have to check, whether there is any queen in the same column or diagonal.

$$x(i) = x(k) \text{ for column}$$

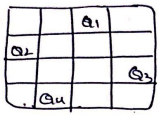
$$|x(i) - x(k)| = (i-k) \text{ for the same diagonal.}$$

5. If any one of the conditions is true then return false indicating that k^{th} queen can't be placed in the position x(k)
6. If the condition fails, increment x(k) value by one and proceed until the position is found.
7. If the position x(k) ≤ n and k=n then solution is generated completely.
8. If x<n, then increment the 'k' value and find position of the next queen.

a. If the position x(k) > n then kth queen cannot be placed as the size of the matrix

b. so decrement the 'k' value by 1 i.e, we have to backtrack and after the position of the previous queen.

4queens : Two possible solutions are



solution: 1

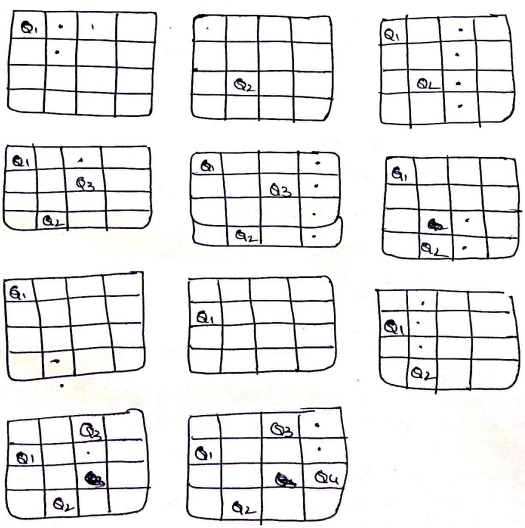
solution: 2

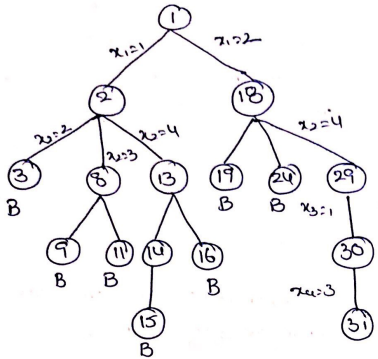
```
Algorithm place(k,i)
{
    for j = 1 to k-1 do
        if ((x[j] = i)
        or (abs (x[j]-i) = abs (j-k)))
        then return false;
    return true;
}
Algorithm Nqueen (k,n)
{
    for i=1 to n do
    {   if place (k,i) then
        {
            x[k] = i;
            if (k=n) then write (x[1:n]);
            else nqueens(k+1,n);
        }
    }
}
```

Tracing of solution to 4-Queen problem:-

The problem is to place 4 queens in a non attacking mode on a 4x4 chess board. So we have to place 4 queen in 4 different columns.

Each column should accomodate one queen, therfore we place queen1 (Q_1) in first column; Q_2 in second column Q_3 is third column Q_4 in fourth column.





Terminology of tree organization:

A problem state is defined by each node of the tree organization. The state space of the problem is defined by all paths from the root to other nodes.

Any node s for which the path from root to s defines a tuple in the solution space is called solution state. If this solution state satisfies implicit constraints of the problem then they are called answer states.

State space tree is the tree organization of the solution space.

Static Trees: Tree organization that are independent of problem instances are called static tree.

Dynamic Trees: Tree organization that are dependent of problem instances they are called dynamic trees.

A generated node, whose children are not yet generated is called live node. The live node whose children

are currently generated is called e-node. A generated node, all of those children are already generated is called dead node.

In back tracking method nodes are generated using depth first search with bounding functions.

⇒ 8-Queen Problem:

The 8-queens problem is to place 8-queens on an 8×8 chessboard in such a way that no two queens attack each other.

Monte carlo method is used to estimate the number of nodes that are going to be generated by a backtracking.

In this process of estimation, a random path Y is generated in the state space tree. The number of children n_i that do not get bounded are determined by the bounding function.

one of these n_i children is randomly selected and is identical as the next node of the path. once we reach terminal node or there is no child node that do not get bounded at that node the path generation terminates.

Using these n_i one can estimate the total number of nodes m in the state space tree that will not get bounded.

If the solution of a given problem instance 'I' the estimated number of nodes is m and is given by

$$m = 1 + n_1 + n_1*n_2 + n_1*n_2*n_3 + \cdots$$

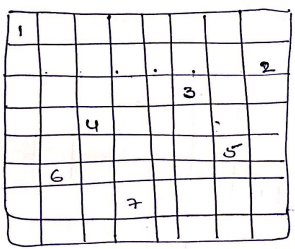
Q5 can be placed in two positions (3,8) among these two positions randomly we have chosen column 5 and Q5 is placed in column 5.

There is no place for Q6. so algorithm terminates so the number of nodes generated are

$= 1+8*5+8*4*5+8*5*4*3+8*5*4*3*2$

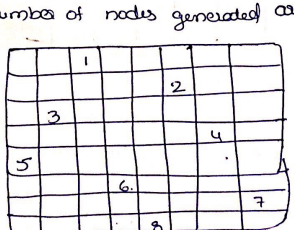
$= 1+40+160+480+960$

$= 1649.$



(8, 6, 4, 2, 1, 1, 1) = 1401

Number of nodes generated are 1401.



(8,5,3,2,2,1,1,1) = 2329

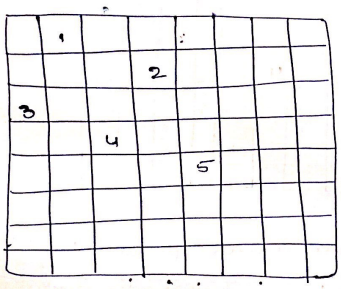
Number of nodes generated are 2329.

The above function estimate can be used to estimate the number of generated by the 8-queen problem.

The first queen can be placed

The placement of each queen on the chessboard was choosen randomly. The first queen can be placed in 8 different ways in the first row.

Let us assume Q_1 is placed in 2nd column. Now Q_2 can be placed any of 5 positions (4,5,6,7,8) in second row. Among these randomly we have selected with column as shown in fig below.



(8, 5, 4, 3, 2) : 1649

Q_3 can be placed in any of four positions (1,6,7,8) among these randomly we have choosen column 1 and Q_3 is placed in column 1.

Q_4 can be placed in three positions (3,7,8) among these randomly we column 3. and Q_4 is placed in column 3

⇒ SUM OF SUBSETS :-

*In sum of subsets problem, n positive numbers called weights are given and we need to find all combinations of these numbers whose sum is m.

*The explicit constraint requires $x_i \in \{j \mid j$ is an integer and $1 \le j \le i\}$. The implicit constraints require that no two solutions be the same and the sum of corresponding weights must be equal to m.

*The solution of sum of subset problem can be expressed in two ways. They are fixed tuple size solution and variable tuple sized solution.

Ex:- if $n=4$, $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$ and $m=31$

solutions are (11, 13, 7) and (24, 7)

→ using variable tuple size the two solutions are

a) (1, 2, 4) indicating 1st, 2nd & 4th objects are selected.

b) (3, 4) indicating 3rd & 4th objects are selected.

→ using fixed tuple size solution two solutions are

a) (1, 1, 0, 1)

b) (0, 0, 1, 1)

objects selected are represented by 1 and not selected are represented by 0

The solution for a given problem instance are determined by the back tracking algorithm by systematically searching a solution space. This search is done by using a tree organization for the solution space

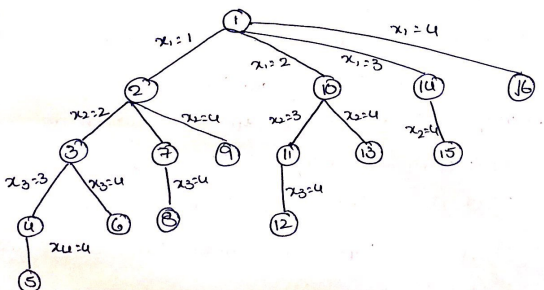
The variable tuple size formulation for $n=4$ and nodes are numbered as depth first search.

There are 16 possible solutions i.e. 2^4 one among this is ( ), this corresponds to empty path the root itself.

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

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

The number of nodes generated by the tree equal to number of possible solution i.e., 16 as



The fixed tuple size formulation for $n=4$ is shown in below figure. The node at depth 1 are for item 1, nodes at depth 2 are for item 2 and so on.

The tree is a full binary tree each node has two children. The left child of the node includes corresponding item and right child of the root excludes the item.

All paths from the root to a terminal node

In fixed tuple strategy, the elements x(i) of the solution vector is either 1 or 0 depending on if the weight w(i) is included or not.

Generation of state space tree:-

① An array X is maintained.

② The value of Xi indicates whether the weight wi is included or not.

③ Initialize sum to zero i.e., s=0

④ Check each node starting from the first node.

⑤ Assign x(k) ← 1

⑥ If s+x(k) = m then we print the subset because the sum is the required output.

⑦ If the above condition is satisfied then check s+x(k)+w(k+1) ≤ m. If, so, we have to generate the left subtree.

It means w(k) can be included so the sum will be incremented and we have, to check for next k.

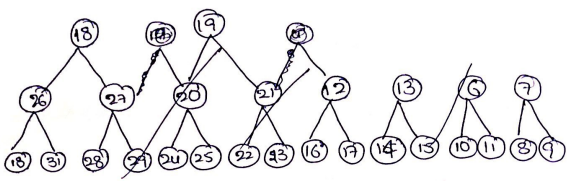
⑧ Now right subtree is to be generated. For this the condition is s+w(k+1) ≤ m. Because w(k) is omitted and w(k+1) has to be selected.

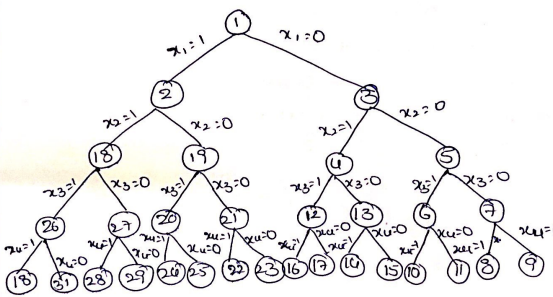
⑨ Repeat the process and find all the possible combinations of the subset.

Algorithm:-

```
Algorithm sumofsubset (s,k,r)
{
    x[k] = 1;
    if (s+w[k] = m) then write (x[1:k]);
    else if (s+w[k]+w[k+1] <= m) then
    sumofsubset (s+w[k], k+1, r-w[k]);
```

defines the solution space. There are 16 possible solutions i.e. 2^4 leaf nodes. Total number of nodes are 31





A backtracking solution for the sum of subset problem using fixed tuple size strategy is given below.

In the solution vector element x_i is one if weight w_i is included otherwise x_i is zero. Initially all w_i's are arranged in non decreasing order.

The bounding function is

$$B_k(x_1, \cdots x_k) = \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

and $\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$

⇒ GRAPH COLORING:-

* If all vertices of graph G can be colored using m colors in such a way that no two adjacent vertices will have the same color, then graph G is said to be m-colorable.

* The problem of assigning colors to every vertex is termed as "m-colorability decision problem".

* A graph G containing n vertices can be colored using n colors, but, yet there may be a smaller m(m<n) such that using m, G can be colored.

* The smallest possible integer m is called the chromatic number of G. The problem of finding the chromatic number of a graph is termed as "m-colorability optimization problem"

Ex:-

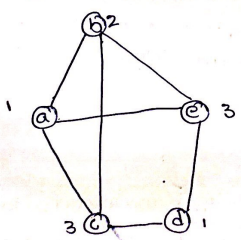


fig: An example of a graph and its coloring

The above graph can be colored using the colors 1, 2, 3. one can also observe that it can not be colored using less than 3 colors.

Hence chromatic number of that graph is 3.

Problem:-

The problem is to determine all the different ways in a given graph can be colored using almost m colors

Solution:-

Let G[1:n, 1:n] be the adjacency matrix of the graph G where G[i,j] = 1 if (i,j) is an edge of G, G

G[i,j] = 0 otherwise.

The colors are represented by the integers 1,2,3,4 ... m. and the solutions are given by n-tuple $(x_1, x_2, \ldots x_n)$ where x_i is the color of node i.

Algorithm for finding all m-colorings of graph with n vertices:-

The following recursive backtracking algorithm uses boolean adjacency matrix G[1:n, 1:n] to represent a Graph, the vertices of the graph are numbered such that adjacency vertices are assigned distinct integers are printed.

```
Algorithm mcoloring(k)
{
    repeat
    {
        //Generate all legal assignments for x[k].
        NextValue(k); // Assign to x[k] a legal color.
        if (x[k]=0) then return;
        if (k=n) then // Almost m colors have been used
                      to color the n vertices.
        write(x[1:n]);
          else mcoloring(k+1);
    } until(false);
}
```

So the following algorithm x[1], ... x[k-1] have been assigned integer values in the range [1, m] such that adjacency vertices have distinct integers.

A values for x[k] is determined in the range [0,m] x[k] is assigned the next highest numbered color while maintaining distinctness from the adjacent vertices of vertex k.

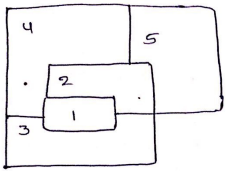
If no such color exist, then x[k] is 0

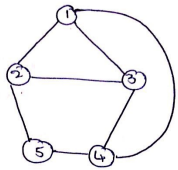
```
Algorithm NextValue(k):
{
    repeat
    {
        x[k] := (x[k+1]) mod (m+1);
        if (x[k]=0) then return;
        for j := 1 to n do
        {
            //check if this color is distinct from adjacent
                                                    colors
            if ((G[k,j] != 0) and (x[k] = x[j])) then
                break;
        }
        if (j = n+1) then return; //new color found
    } until (false);
}
```

Function m-coloring is begun by as first assigning the graph to its adjacency matrix, setting the array x[] to zero, and then invoking the statement NextValue(k)

The time complexity of graph coloring is $O(n^m)$

Consider the map with five regions & its graph





1 is adjacent to 2, 3, 4

2 is adjacent to 1, 3, 4, 5

3 is adjacent to 1, 2, 4

4 is adjacent to 1, 2, 3, 5

5 is adjacent to 2, 4

Steps to color:-

① Create the adjacency matrix graph(1:m, 1:n) for a graph.

② If there is an edge between i, j then C(i,j) = 1 otherwise C(i,j) = 0.

③ The colors will be represented by the integers 1, 2 ... m and the solution will be stored in the array x(1), x(2) ... x(n), x(index) is the color, index is the node.

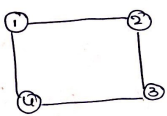
④ The formula used to set the color is x(k) = (x(k)+1) % (m+1)

⑤ First one chromatic number is assigned, after assigning a number for 'k' node, we have to check whether the adjacent nodes has got the same values if so then we have to assign the next value.

⑥ Repeat the procedure until all possible combinations of colors are found.

(7) The function which is used to check the adjacent nodes and same color is, if((Graph(k,s) == 1) and x(k) = x(j))

Ex:



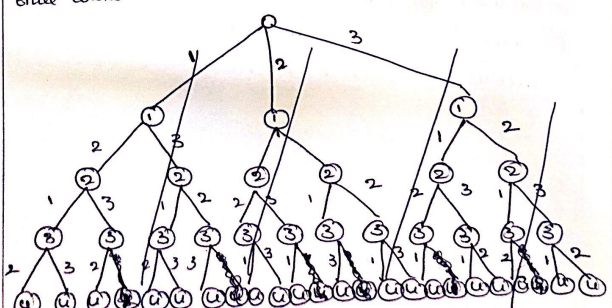
N=4
m=3

The problem is to color the given graph of 4 nodes using 3 colors. The above graph can be colored using two colors also.

State space tree:

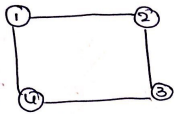
In the state space tree numbers in the nodes are the vertices of the graph where as the weights of the edges are the colors.

In the tree we have three different weights. they are 1,2,3. Node 1 can be colored with any of the three colors 1,2,3 as show in the fig.



(7) The function which is used to check the adjacent nodes and same color is, if((Graph[k,s]==1) and x[k] = x[j])

Ex:-



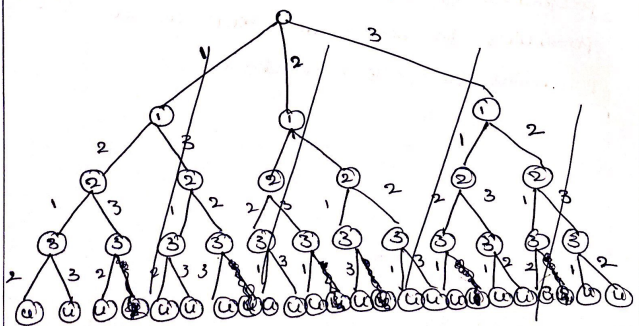
N=4
m=3

The problem is to color the given graph of 4 nodes using 3 colors. The above graph can be colored using two colors also.

State space tree:-

In the state space tree numbers in the nodes are the vertices of the graph where as the weights of the edges are the colors.

In the tree we have three different weights. they are 1,2,3. Node 1 can be colored with any of the three colors 1,2,3 as show in the fig.

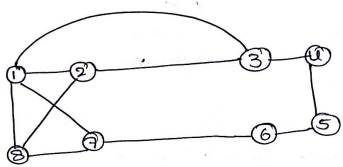


⇒ Hamiltonian Cycles:

A Hamiltonian cycle is a closed path that includes every vertex exactly once.

Let $G = (V, E)$ be a connected graph with 'n' vertices

Ex:



The above graph has two hamiltonian cycles:

1, 3, 4, 5, 8, 6, 7, 8, 2, 1 &

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

The backtracking algorithm can be used to find the Hamiltonian cycle of any graph.

Procedure:

(1) Define a solution vector $x(x_1 \cdots x_n)$ where x_i represents the i^{th} visited vertex of the proposed cycle.

(2) Create a cost adjacency matrix for the given graph.

(3) The solution array should be initialized to all zeros except $x(1) = 1$, because the cycle should start at vertex 1.

(4) Now second vertex to be visited in the cycle should be found.

(5) The vertex from 1 to n are included in the cycle.

one by one by checking the following two conditions.

6. There should be a path from previous visited vertex to current vertex.

7. The current vertex must be distinct and should not have been visited earlier.

8. When the above two conditions are satisfied the current vertex is included in the cycle, else the next vertex is tried.

9. When the n^{th} vertex is visited we have to check, is there any path from n^{th} vertex to first 8 vertices.

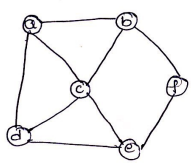
If no path, then go back one step and after the previous visited node, repeat the above steps to generate possible Hamiltonian cycle.

Algorithm Hamiltonian(k):

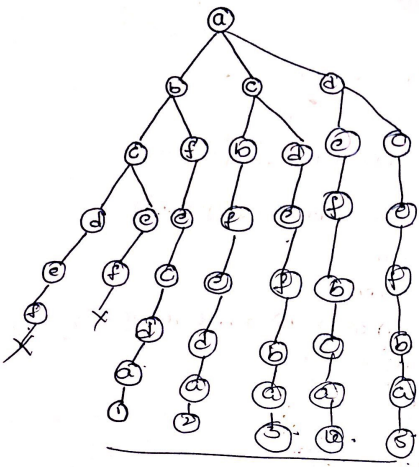
```
{
    loop
      next value(k)
      if (x(k)=0) then return;
      {
        if k=n then
            print(x)
        else
        Hamiltonian(k+1);
      end if
      }
      repeat
    }
  Algorithm nextvalue(k)
  {
    repeat
```

```
{
    x[k] = (x[k]+1) mod (n+1);
    if(x[k]=0) then return;
    if(a[x[k-1], x[k]] ≠0) then
    {
        for j=1 to k-1 do if (x[j]=x[k])
        then break;
        if(j=k) then
        if((k<n) or((k=n) and G[x[n], x[1]] ≠0))
        then
        return;
    }
} until (false);
}
```

Ex:-



Hamiltonian cycle :-
① a b f e d a
② a c e f b a
③ a d c e f b a
④ a d e f b c a



Space Ex:- State space tree for the above example.

UNIT- III

Syllabus:-

Greedy method - General method, applications - knapsack problem, Job sequencing with deadlines, minimum cost spanning trees, Single source shortest path problem.

GREEDY METHOD:-

* The Greedy method is the most simple design technique. It is an optimization problem which has numerous applications.

* optimization problem is required to find the solution which has optimal value. i.e which minizes or maximizes the value by satisfing the given constraints.

* most of the problems has n inputs and require us to obtain a subset that satisfies some constraints which is called a feasible solution.

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

* A problem may have any numbers of feasible solutions. but there will be only one optimal solution.

* optimal solution is one of the feasible solution for which either the cost is minimum or profit is maximum.

* The Greedy method algorithm generally works in stages. In the Greedy approach each stage is choosen has to satisfy the constraints given in the problem. and it forms feasible solution, and it will be included in the optimal solution.

Functions of Greedy Algorithm:

To construct the solution in an optimal way, algorithm maintains two sets. one contains choosen items and other contains rejected items

The following 4 functions are available in the greedy algorithm:

① A function that checks whether the choosen set of items provide a solution.

② A function that checks the feasibility of a set.

③ The selection function tells which of the candidates is the most promising.

④ An objective function, which does not appear explicitly, gives the value of a solution.

The "greedy-choice property" and "optimal substructure" gives the value of a solution.

Greedy choice Property:

A globally optimal solution can be arrived at by making a locally optimal choice.

Optimal Substructure:

A problem exhibits optimal substructure if an optimum solution to the problem contains optimal solutions to sub problems. within it

An input 'a' is selected from the array X by the function SELECT and the function FEASIBLE determines if 'a' can be included into the optimum solution vector.

The function UNION actually combines 'a' with partial solution and updates the objective function. procedure

GREEDY describes that, once a particular problem is chosen and the procedures SELECT, FEASIBLE, and UNION are properly implemented.

Greedy method control abstraction for the subset paradigm:

```
Procedure GREEDY(x, n)
    optimum-solution = NULL
    for i = 1 to n do
      a ← = SELECT(x)
        if FEASIBLE(optimum-solution, a)
          then optimum-solution = UNION(optimum-solution, a)
        end if
    repeat
    return (solution)
    end GREEDY.
```

The applications of greedy method are

(1) Knapsack problem
(2) Job sequencing with deadlines
(3) minimum cost spanning trees
    (a) prim's Algorithm
    (b) kruskal's Algorithm
(4) single source shortest paths.

① Knapsack Problem:-

Given a set of objects, each with a weight and profit, it is to be determined the number of each object to include in a collection so that the total weight is less than or equal to a given limit and total profit is as large as possible.

* Given n objects an a knapsack or bag with a capacity m, the problem is to place 'n' objects in knapsack or bag.

* A weight and profit are associated with each object, Object 'i' has a weight w_i and profit p_i.

* The profit $p_i x_i$ is earned if a fraction x_i, $0 <= x_i <= 1$, of object 'i' is placed into the knapsack and the object with weight $w_i x_i$ is stored in the knapsack.

* The total weight of all choosen objects. can be at most m which is the knapsack capacity.

* The main objective of the knapsack problem is to maximize the total profit earned by filling the knapsack with the objects.

* Greedy method is used to solve the knapsack problem.

Formally, the problem can be stated as

maximize $\Sigma_{1 \le i \le n} p_i x_i$

subject to $\Sigma_{1 \le i \le n} w_i x_i \le m$ and

$0 \le x_i \le 1, 1 \le i \le n$

$\Sigma_{(1 \le i \le n)} w_i x_i \le m$ is the objective function

All inputs are not considered for the solution in a knapsack problem, a subset of inputs is selected hence this problem will fall under subset paradigm.

Further if a selected object doesn't fit into the knapsack, then fraction of it is added to fill the knapsack.

Ex:

Consider the following instance of the knapsack problem

$n=3 \quad m=20 \quad (P_1, P_2, P_3) = (25, 24, 15)$

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

4 feasible solutions are.

Profits(P_i) :	25	24	15
weights(w_i) :	18	15	10

Any subset that satisfies the objective function is called feasible solution.

$\sum_{(1 \le i \le n)} w_i x_i \le m$

Four feasible solutions which satisfy the objective function are considered. In the first solution we randomly choose the fractions $\frac{1}{2}, \frac{1}{3}, \frac{1}{4}$ & their corresponding weights are calculated.

In second feasible solution objects are choosen with decreasing order of profits & their corresponding weights and profits are calculated.

In third feasible solution objects are choosen with increasing order of weights and corresponig weights and profits are calculated.

The above solution doesn't give optimal profit. Optimal profit obtained if the objects are placed with decreasing order of P/w ratio, which is the fourth feasible solution.

NO	x_1, x_2, x_3	$\sum w_i x_i$	$\sum p_i x_i$
①	$(\frac{1}{2}, \frac{1}{3}, \frac{1}{4})$	$18(\frac{1}{2})+15(\frac{1}{3})+10(\frac{1}{4})$ $= 9+5+2.5$ $= 16$	$25(\frac{1}{2})+24(\frac{1}{3})+15(\frac{1}{4})$ $= 12.5+8+3.75$ $= 24.25$
②	$(1, 2/15, 0)$	$18(1)+15(\frac{2}{15})+0$ $= 18+2 = 20$	$25(1)+24(\frac{2}{15})+0$ $= 28.2$
③	$(0, \frac{2}{3}, 1)$	$0+15(\frac{2}{3})+10(1)$ $= 20$	$0+24(\frac{2}{3})+15(1)$ $= 31$
④	$(0, 1, \frac{1}{2})$	$0+15(1)+10(\frac{1}{2})$ $= 20$	$0+24(1)+15(\frac{1}{2})$ $= 31.5$

→ ① ② ③ ④ feasible solution & ④ is called optimal solution

maximal / optimal profit : 31.5

p_i = 25 24 15

w_i = 18 15 10

$\frac{p_i}{w_i}$ = 1.3 1.6 1.5

↓ max ↓ 2 maximum $\frac{5}{10} = \frac{1}{2}$

x_i | 0 | 1 | 1/2 |

n = 3

bag capacity = 20

20 − 15 − 5

5 − 5 = 0 → bag

```
Algorithm Greedy knapsack (m, n)
x[1:n] → 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];
}
```

There are so many ways to solve this problem, which will give many feasible solutions for which we have to find the optimal solution.

But in this algorithm it will generate only one solution which is going to be feasible as well as optimal.

An optimal solution is obtained by selecting a solution having optimum value among all possible feasible solutions.

Greedy approach directly produces an optimal solution, in knapsack problem one can get optimal solution directly by selecting the objects in the decreasing order of p/w ratio

Time complexity : Time complexity to find P/W vector +
Time complexity to sort objects such that
$P[i]/w[i] >= P[i+1]/w[i+1]$ + Time complexity of Greedy knapsack. $= c_1*n + c_2*n\log n + c_3*n = O(n\log n)$

Time complexity of knapsack problem

$O(n \log n)$

⇒ JOB SEQUENCING WITH DEADLINES :-

* Given 'n' jobs, associated with each job a deadline d and a profit p.

* For any job 'i' the profit P_i is earned if and only if the job is completed by its deadline d_i

* 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

* Since one job can be processed in a machine the other jobs has to be in its waiting state until the job is completed and the machine becomes free.

* The feasible solution for the above problem is to find a subset of jobs S such that all the jobs in the subset S has to be completed before their respective deadlines.

* The sum of the profit of the jobs in S is the value of feasible solution S

* An optimum solution is a feasible solution with maximum value.

* All inputs are not considered for the solution set in this problem, a subset of inputs is selected hence this will fall under subset paradigm.

These jobs must be processed in the order job4 followed by job1. Thus the processing of job4 begins at 0 and that of job1 is completed at time 2

An optimal solution is obtained by selecting a solution having optimum value among all possible feasible solutions

Greedy Approach:-

It directly produces an optimal solution. In this problem one can get optimal solution directly by selecting the jobs in the non-decreasing order and checking whether they can be completed before deadline.

P_1 P_2 P_3 P_4 100 27 15 10

d_1 d_2 d_3 d_4 2 1 2 1

max 2

0 J_2 1 J_1 2

100 + 27 = 127

General Algorithm for job sequencing with deadline:-

```
Algorithm GreedyJob(d, J, n)
{
    J := {1};
    for i := 2 to n do
    {
        If (all jobs in J∪{i} can be completed by their
deadlines) then J := J∪{i};
    }
}
```

* The problem is given number of jobs, their profits and deadlines, it is required to find a sequence of jobs, which will be completed before its deadlines, and it should yield maximum profit.

So the waiting time and the processing time should be less than or equal to the dead line of job.

Example :-

Let n=4, Profits $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$

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

Maximum deadline of the job is 2. It means that atmost we can do two jobs only among 4 jobs. A feasible solution contains all possibilities of execution of two jobs and one job.

The feasible solutions and their values are

S.NO	Feasible solution	Solution sequence	value (profit)
1	(1,2)	(2,1)	110
2	(1,3)	1,3 or 3,1	115
3	(1,4)	4,1	127
4	(2,3)	2,3	25
5	(3,4)	4,3	42
6	(1)	1	100
7	(2)	2	10
8	(3)	3	15
9	(4)	4	27

Solution 3 is optimal. In this solution only jobs 1 and 4 are processed and the value is 127

Greedy Algorithm for job sequencing with deadlines and profits:-

```
Algorithm JS(d,J,n)
{
    d[0] := J[0] := 0;
    J[1] := 1;
    k := 1;
    for i := 2 to n do
    {
        r := k;
        while(d[J[r]] > d[i]) and d[J[r]] != r))
        do
        r := r-1;
        if (d[J[r]] <= d[i]) and (d[i] > r)) then
        {
        for q := k to (r+1) step -1 do
            J[q+1] := J[q];
            J[r+1] := i;
            k := k+1;
    }
    }
    return k;
}
```

* There are two possible parameters for JS Algorithm in terms of which its complexity can be measured.
    n - the no of jobs
    s - the no of jobs included in the solution J.
* Each iteration takes O(1) time
* The computing time of JS can be reduced from $O(n^2)$ to nearly O(n).

⇒ Minimum cost Spanning Tree :.

The cost of a spanning tree is the sum of cost of the edges in that tree. There are two methods to determine minimum cost spanning trees are

① Prim's Algorithm
② Kruskals Algorithm.

Advantages :.

* spanning trees used in designing efficient routing algorithms
* spanning trees have wide applications in many areas, such as network design.

Time complexity of this algorithm is O(n), where 'n' is the number of edges.

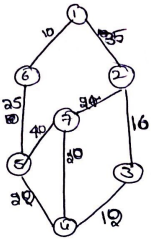
The number of spanning trees in the complete graph K_n is n^{k-2}

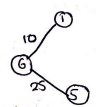
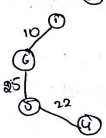
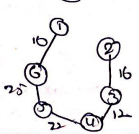
① Prim's Algorithm:.

Ex:* We have to consider all the vertices first.

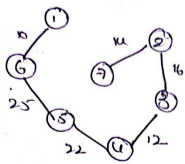
* Then we will select an edge with min weights.
* The algorithm proceeds by selecting adjacent edges with minimum weight.
* We should note be taken for not performing circuit.

Given Graph:



Edge	Cost	Spanning Tree
(1,6)	10	
(6,5)	25	
(5,4)	22	
(4,3)	12	
(3,2)	16	

(2,3) 14



Total cost = 99

No of vertices = 9

No of edges = 6

Time complexity $O(n^2)$

n = number of vertices.

Prim's Algorithm:

```
Algorithm prim-mst(G, cost, n, tree)
{
    min := cost[p,q]; // initially we store starting
                         in array tree.
    tree[1,1] := p;
    tree[1,2] := q;
    for i := 1 to n do
    // finding min cost from the neighbouring vertices
    if (cost[i,q] < cost[i,p])
        optimum[i] := q;
    else
        optimum[i] := p;
    // after visiting vertex p or q put 0 in an array
    optimum[p] := optimum[q] := 0;
    for i := 2 to n-1
```

```
{
    Tree[i,j] := 5;
    Tree[1,2] := optimum[5];
    min := min + cost[5], optimum[5];
    optimum[5] := 0;
    for k:=1 to n do
    if (optimum[k] != 0) AND (cost(k, optimum(k))
            >cost[k,5] then
}
return min;
```

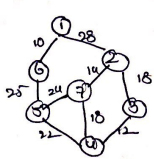
Analysis:

The algorithm spends most of its time in finding the smallest edge. So, time of the algorithm basically depends on how do we search this edge. Therefore prims algorithm runs in $O(n^2)$ time.

Kruskal's Algorithm:

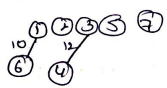
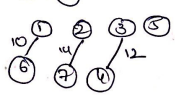
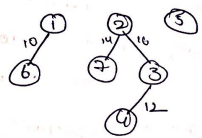
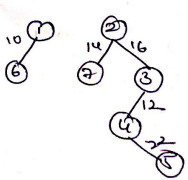
In kruskal's algorithm always the minimum cost edge has to be selected. It is not necessary that selected optimum edge is adjacent.

Ex: Consider graph

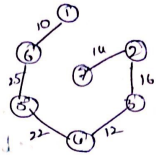


* First we select all the vertices then an edge with optimum weight is selected from heap.

* Even though it is not adjacent to previously selected edge, care should be taken for not forming circuit.

Edge	cost	spanning tree.
(1,6)	10	
(4,3)	12	
(2,7)	14	
(2,3)	16	
(7,4)	18	reject because it forms a circuit.
(5,4)	22	

(5,6) 25



Algorithm:

```
Algorithm kruskal(g, cost, n, tree)
{
    delete minimum cost edge (p,q) from heap and
    reheap /
    s: = find(p);
    t: = find(q): // finds minimum cost edge
    if (s != t) then //check whether it creates cycles
    {
        i = i++;
        tree(i,1): = p;
        tree(i,2): = q;
        min: = min + cost[p,q];
        union(s,t);
    }
    Return min;
}.
```

Time complexity of kruskal's algorithm

Average performance $O(|E| \log |V|)$

worst-case space complexity $O(|E| + |V|)$

Single Source shortest paths :-

Given a directed graph $G=(V, E)$ whose V is the set of vertices and E is the set of edges, a weighting function cost for the edges of G, and a source vertex V_0

The problem of single source shortest path is to determine the shortest path from V_0 to all the remaining vertices of G.

It is assumed that all weights are positive. Since the shortest path between V_0 and some other node v is an ordering among a subset of the edges.

Hence this problem is in the ordering paradigm

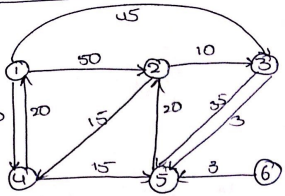
* To formulate greedy - based algorithm one possibly is to build the shortest paths one by one.

* As an optimization measure we can use the sum of lengths of all paths so far generated.

The greedy way to generate the shortest paths from V_0 to remaining vertices is to generate these paths in non-decreasing order of path length.

First, a shortest path to the nearest vertex is generated. Then a shortest path to the second nearest vertex is generated and so on.

Ex:



v_0 : source = 1

(1→2)
Feasible solutions are
(1,2) = 50
(1,4,5,2) = 45

(1→3)
Feasible solutions are
(1,3) = 45
(1,4,5,3) = 58

(1→4)
(1,4) = 10
(1,2,4) = 65

(1→5)
(1,4,5) = 25

1→6
no path.

Greedy solutions

① 1 → 4 → 5 → 2 → 3
10 15 20
45

② 1 → 3
45

③ 1 → 4 → 5 → 3
45

If source is 1 & destination is 2 then 45 is the optimal solution.

If source is 1 & destination is 3 then 45 is optimal solution etc.

Greedy Algorithm to generate shortest path

```
Algorithm shortestpath(v, cost, dist, n)
{
    for i := 1 to n do
    {
        s[i] := false; dist[i] = cost[v,i];
    }
    s[v] = true; dist[v] = 0.0;
    for num = 2 to n-1 do
    {
        Choose u from among those vertices
        not in s such that dist[u] is minimum;
        s[u] = true;
        for (each w adjacent to u with s[w] =
                                    false) do
            if { dist[w] > dist[u] + cost[u,w] } then
            dist[w] = dist[u] + cost[u,w];
    }
}
```

Analysis :-

Any shortest path algorithm must examine each edge in the graph at least once since any of the edges could be in the shortest path

Hence minimum possible time for such such an algorithm would be $\Omega(|E|)$. Since cost adjacency matrix were used to represent the graph, it takes $O(n^2)$ time. overall running time = $O((n+|E|)\log n)$

UNIT - IV

Syllabus :-

Dynamic Programming :- General method, Applications - Chained matrix multiplication, All pairs shortest path problem, Optimal binary search trees, 0/1 knapsack problem, Reliability design, Travelling sales person problem.

Dynamic Programming :-

Dynamic Programming is technique for solving problems with overlapping subproblems.

In this method each subproblem is solved by only once. The result of each subproblem is recorded in a table from which we can obtain a solution to the original problem.

General method :-

* An algorithm design method that has a sequence of decisions in its solution to a problem is called Dynamic Programming

* Decisions are made one at a time leading to optimal sequence of decisions. In taking each individual time leading to optimal sequence of decisions.

* In taking each individual decision "The principle of optimality" should be followed.

Principle of optimality :- It states that an optimal sequence of decisions has the property that whatever the initial state and decisions are, the remaining decisions

must constitute an optimal decision sequence with regard to the state resulting from the first decision.

Difference between Divide and Conquer and Dynamic programming

Divide and Conquer	Dynamic Programming
1. The problem is divided into small pseudo problems. These subproblems are solved independently. Finally all the solutions of sub problems are collected together to get the solution to the given problem.	1. In dynamic programming many decision sequences are generated and all the overlapping sub instances are considered.
2. In this method duplications in subsolutions are neglected, i.e. duplicate subsolutions may be obtained.	2. In dynamic computing duplications in solutions is avoided totally.
3. Divide and conquer is less efficient because of rework on solutions.	3. Dynamic programming is efficient than divide and conquer strategy.
4. The divide and conquer uses top down approach of problem solving.	4. Dynamic programming uses bottom up approach

⇒ Difference between Greedy Algorithm and Dynamic Programing

Greedy method	Dynamic Programming
1. Greedy method is used for obtaining optimal solution	1. Dynamic Programming also used for obtaining optimal solution
2. In greedy method a set of feasible solutions and picks up the optimum solution	2. There is no special set of feasible solutions in this method.
3. In Greedy method the optimum selection is without revising previously generated solutions	3. Dynamic programming considers all possible sequences inorder to obtain optimum solution
4. In Greedy method there is no such guarantee of getting optimum solution	4. It is guaranteed that the dynamic programming will generate the optimum solution.

Steps of Dynamic Programming :-

Dynamic Programming design involves 4 major steps

1. Characterize the structure of optimal solution. That means develop a mathematical notation that can express any solution and subsolution for the given problem.
2. Recursively define the value of an optimal solution
3. By using bottom up technique compute value of optimal solution. For that you have to develop a recurrence relation that relates a solution to its subsolutions, using mathematical notation of step1

4. compute an optimum solution from computed information.

⇒ Principle of optimality:

The dynamic programming algorithm obtains the solution using principle of optimality.

The principle of optimality states that "in an optimal sequence of decisions or choices, each subsequence must also be optimal."

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

Applications of Dynamic Programming

1. 0/1 knapsack problem
2. matrix chain multiplication
3. All pairs shortest path problem
4. Travelling sales person problem
5. Optimal Binary Search Tree
6. Reliability Design.

⇒ 0/1 knapsack Problem:-

In 0/1 knapsack problem we have a knapsack of capacity "C". This knapsack is to be filled with given objects such that we get a maximum profit.

Associated with the "n" objects profits $\{P_1, P_2, P_3 \cdots P_n\}$ and weights $\{w_1, w_2, w_3, \cdots w_n\}$ are given. The selection of the objects should meet the following constraints.

(i) $x_i \in \{0, 1\}$, where x_i is 'ith' selected object

(ii) $\sum x_i w_i \leq C$

knapsack problem and 0/1 knapsack problem are different. Greedy method is used to solve knapsack problem

* Dynamic programming is used to solve 0/1 knapsack problem.

The additional constraint that in 0/1 knapsack problem is $x_i \in \{0, 1\}$ i.e. part of an object can't be put in the knapsack.

→ let $f_j(y)$ is the profit on including the objects between 1 and j subject to the weight y i.e KNAP(1, j, y)

By using principle of optimality we can write that:

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

In general for $i > 0$

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

where $f_0(y) = 0$ for any weight y.

*. The solution set is represented by a tuple called (P, W) where 'P' is the profit 'W' is the weight.

* In the case where w's are can be real numbers $s^i = \{(P, W) | P = f_i(y_i) \text{ and } W = y_i\}$.

* Let s^i be the solution set by including only the object from 1 to i. $s^0 = \{0, 0\}$

Step1:- * When generating the s^i's, we can also purge all pairs (P, W) with $W > m$, as these pairs determine the value of $f_n(x)$ only for $x > m$.

* Since the knapsack capacity is m, we are not interested in the behaviour of f_n for $x > m$. When all pairs (P_j, W_j) with $W_j > m$ are purged from the s^i's, $f_n(m)$ is given by the P values of the last pair in s^n.

Step2:-

When computing s^n, we can find the solutions to all the knapsack problems KNAP(1, n, x), $0 \le x \le m$, and not just KNAP(1, n, x). Since, we want only a solution to KNAP(1, n, m), we can dispense with the computation of s^n.

The last pair in s^n is the either the last one in s^{n-1} or it is $(P_j + P_n, W_j + W_n)$, where $(P_j, W_j) \in s^{n-1}$ such that $W_j + W_n \le m$ and W_j is maximum.

Step3:-

If (P_1, W_1) is the last tuple in s^n, a set of

Ex:- Consider the knapsack instance n: 3, (w_1, w_2, w_3): $(2,3,4)$, $(P_1, P_2, P_3) = (1,2,5)$ and m: 6

$(P_1, w_1) = (1,2)$

$(P_2, w_2) = (2,3)$

$(P_3, w_3) = (5,4)$

$s_0 = \{(0,0)\}$ empty knapsack.

S_1^0

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

$S^1 = \{(0,0), (1,2)\}$; $S_1^1 = \{(2,3), (3,5)\}$

$S^2 = \{(0,0)(1,2)(2,3)(3,5)\}$; $S_1^2 = \{(5,4), (6,6), (8,7), (8,9)\}$

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

on empty knapsack we place one object

Hence $S_1^0 = \{1,2\}$

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

with one object we have two alternatives i.e., object is selected or not selected.

$S_1^1 \rightarrow$ on S^1 object 2 is placed.

$S_1^1 \{S^1 + (2,3)\}$

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

$S_1^1 = \{(2,3), (3,5)\}$

$S^2 = S^1 \cup S_1^1$

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

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

S^2 → on S^2 object 3 is placed.

$S_1^2 = \{S^2 + (5,4)\}$

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

$S_1^2 = \{(5,4), (6,6), (7,9), (8,9)\}$

$S^3 = S^2 \cup S_1^2$

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

From this solution set some pairs are removed. Pairs (7,9), (8,9) are removed as the weight of above pair is more than knapsack capacity.

By using dominance rule/ purging rule pair (3,5) is compared with (5,4). In this pair (3,5) is dominated by pair (5,4) because profit (3,5) is less than (5,4) and weight is more.

ep. (3,5) has been dominated from S^3

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

From above solution maximum profit is (6,6)

Hence maximum profit obtained = 6

Pair $(6,6) \in S^3$
$(6,6) \notin S^2$ } Hence object 3 is selected. $x_3 = 1$

$(6,6) - (5,4) = (1,2)$

$(1,2) \in S^2$
$(1,2) \in S^1$ } x_2 is 0 which is not included.

For one object we have two alternatives:

1. Object is not included in the knapsack then profit:0, weight = 0. (P,W) pair = (0,0).

2. Object is included in knapsack then profit = P_1, weight = w_1. (P,W) pair = (P_1, w_1)

Hence $S^1 = \{(0,0)(P_1,w_1)\}$

Now on S^1 we place second object (P_2, w_2). It is represented by S^1.

$S^1 = \{(0+P_2, 0+w_2), (P_1+P_2, w_1+w_2)\}$

$S^1 = \{(P_2,w_2), (P_1+P_2, w_1+w_2)\}$

$S^2 = S^1+S^1 = \{(0,0)(P_1,w_1)(P_2,w_2)(P_1+P_2, w_1+w_2)\}$.

For two objects we have four alternatives:

1. Two objects are not selected $P=0, W=0, (P,W)=(0,0)$

2. First object is selected and second object is not selected

$P=P_1, W=w_1 (P,W)=(P_1,w_1)$

3. First object is not selected and second object is selected.

$P=P_{i+1}$

$P=P_2, W=w_2 (P,W)=(P_2,w_2)$

4. Both objects are selected.

$P=P_1+P_2 \quad W=w_1+w_2, (P,W) = (P_1+P_2, w_1+w_2)$

o/1 values for the x_i's such that $\Sigma P_i x_i = P$, and $\Sigma w_i x_i = W$, can be determined by carrying out a search through the s^i's.

We can set $s_n = 0$ if $(P, W) \in S^{n-1}$. If $(P, W) \notin S^{n-1}$, then $(P - P_n, W - W_n) \in S^{n-1}$ and we can set $x_n = 1$. This leaves us to determine how either (P, W) or $(P - P_n, W - W_n)$ was obtained in S^{n-1}. This can be done recursively.

Dominance Rule:

* S^{i+1} contains two pairs (P_1, W_1) and (P_2, W_2) with the property that $P_1 \leq P_2$ and $W_1 \geq W_2$ then the pair (P_1, W_1) can be discarded because containing with (P_1, W_1) will not lead to optimal solution.

S^i represents the possible states resulting from the i^{th} decision sequence for $x_1, x_2 \cdots x_i$

A state refers to a pair (P_j, W_j), W_j the total weight of objects included in the knapsack and P_j is the corresponding profit.

$S^0 = \{(0,0)\}$; Empty knapsack profit pair is .

Empty knapsack is represented by $S^0 = \{(0,0)\}$

By adding the first tuple to the solution set so, the solution set becomes

$S_1^0 = \{(P_1, W_1)\}$

$S^{i+1} = S^i \cup S_1^i$

$S^1 = S^0 \cup S_1^0 = \{(0,0), (P_1, W_1)\}$

S^i is a solution set by including ... the objects from 1 to i. S^i is the solution set by including one object

$(1,2) \in S^1$ } $x_1 = 1$ $(1,2)-(1,2) = (0,0)$

$(1,2) \notin S^0$

object 1 is selected

$x_1 = 1$

object 3 is selected

solution is (1,0,1)

Algorithm:-

```
DKP(P, w, n, m)
{
    S^0 := {(0,0)};
    for i := 1 to n-1 do
    {
        S_1^{i-1} := {(P,w) | (P-P_i, w-w_i) ∈ S^{i-1} and
                                        w_i ≤ m};
        S^i := mergepurge(S^{i-1}, S_1^{i-1});
    }
    (Px, wx) = last pair in S^{n-1};
    (Py, wy) = (P+P_n, w+w_n) where w is the largest
        w in any pair in S^{n-1} such that w+w_n ≤ m;
    if (Px > Py) then x_n := 0;
    else
        x_n := 1;
    TraceBackFor (x_{n-1} ... x_1);
}
```

Time complexity:- Time complexity of 0/1 knapsack problem is $O(nw)$ where, n is the number of items w is the capacity of knapsack.

⇒ All Pairs Shortest Path Problem:-

All pairs shortest path problem is to find the shortest distances between every pair of nodes of the given graph.

Let G=(V,E) be a directed graph where 'V' is a set of vertices or nodes and 'E' is the set of edges. Each edge has an associated non-negative length.

The problem is to calculate the length of shortest path between each pair of nodes.

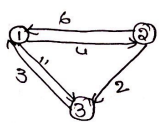
Suppose the nodes of G are numbered from 1 to n, so V = {1, 2, ... n}. Let cost be an adjacency matrix which stores weight of edges of graph G.

Graph G has n vertices, cost(i,i) = 0, 1<=i<=n. Vertices that are not adjacent i.e., vertices which are not connected with an edge. If the edge (i,j) does not exist then for them cost(i,j) = ∞

Principle of optimality:-

If k is the node on the shortest path from i to j then the part of the path from i to k and part from k to j must also be optimal, that is shorter.

Ex:-



First create cost adjacency matrix for the given graph.

consider the example graph, its adjacency matrix is

$$A = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

To solve this problem we introduce vertex -1 as an intermediate vertex to find shortest distances between every pair of vertices.

(v_i, v_j) where $1 \le i \le n$; $1 \le j \le n$

By following this method the shortest distances between every pair of vertices are

$$A_1 = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

In the above matrix $A_1(3,2) = \min(A(3,2), A(3,1)+A(1,2))$

$= \min(\infty, 3+4) = 7$

In the second step the intermediate vertex is 2, the shortest distances matrix A_2 is given below

$$A_2 = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

In the above matrix $A_2(2,3) =$

$\min(A_1(3,1), A_1(3,2) + A_1(2,1))$

$\min(3, 7+6)$

$= 3$

In the third step the intermediate vertex is 3, the shorter distance matrix A_3 is

$$A_3 \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 8 & 7 & 0 \end{bmatrix}$$

In the above matrix $A_3(2,1)$:

$= \min(A_2(2,1), A_2(2,3) + A_2(3,1))$

$= \min(6, 2+3)$

$= 5$

$A_3(1,2) = \min(A_2(1,2), A_2(1,3) + A_2(3,2))$

$= \min(4, 6+7)$

$= 4.$

The above matrix gives the shortest distances between every pair of vertices.

Algorithm All-paths(cost, x, n)

```
for i:=1 to n do
{
    for j:=1 to n do
    {
        x[i,j] = cost[i,j];
    }
}
for i:=1 to n do
{
    for j:=1 to n do
    {
        for k:=1 to n do
        {
            x[i,j] = min[x[i,j], x[i,k] + x[k,j]);
        }
    }
}
```

Parameters :

cost → Adjacency matrix of the graph. this is input parameter.

x → It is an output parameter at the termination of the algorithm. this matrix holds the shortest distance matrix.

n → number of vertices in the graph

The time complexity of above algorithm is equals to $O(n^3)$. because the statement $x[i,j] = \min\{x[i,j], x[i,k] + x[k,j]\}$; gets executed n^3 times as the other nested loop statements has less complexity of n^2.

Time complexity is $O(n^3)$.

TRAVELLING SALES PERSON PROBLEM :

* Let $G = (V, E)$ be a directed graph with n vertices. Let c_{ij} is the cost of the edge between i and j.

* $c_{ij} > 0$ and $c_{ij} = \infty$ if $\langle i, j \rangle \notin E$. Let $|V| = n$ and assume $n > 1$. A tour of G is a directed simple cycle that includes every vertex in V.

* The problem is to find out a tour of minimum cost. The cost of the tour is is the sum of cost of edges on the tour.

* The tour is the shortest path that starts and ends at the same vertex i.e. vertex 1.

Applications :- water pipelines, mail delivery, transport etc.

* Suppose we have to route a postal van to pick up mail from one mail box located at 'n' different sites.

* A n+1 vertex graph can be used to represent the situation.

* one vertex represent the post office from which the post office from which the postal van starts and return.

* Edge $\langle i, j \rangle$ is assigned a cost equal to the distance from site 'i' to site 'j'. the route taken by the postal van is a tour and we are finding a tour of minimum length.

* Every tour consists of an edge $\langle 1, k \rangle$ for some $k \in V - \{1\}$ and a path from vertex k to vertex 1.

* The path from vertex k to vertex 1 goes through each vertex in $V - \{1, k\}$ exactly once.

* The function which is used to find the path is $g(1, V - \{1\}) = \min \{C_{1s} + g(s, s - \{s\})\}$ $g(i, s)$ be the length of shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1.

* The function $g(1, V - \{1\})$ is the length of an optimal tour.

Solution:-

Let $g(i, s)$ be the length of a shortest path starting at vertex i, going through all vertices in S and terminating at vertex 1.

Then $g(1, V - \{1\})$ is the length of the optimal sales person tour.

Using Principle of optimality:

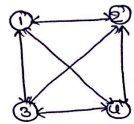
$$g(i, V-\{j\}) = \min_{k}\{c_{ik} + g(k, V-\{j,k\})\} \quad 2 \le k \le n$$

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

We know that $g(i,\phi) = c_{i1}$, $1 \le i \le n$. From this we can compute for $g(i,S)$ for $|S| = 1, 2, \cdots n-1$

When $|S| < n-1$, the values of i,j are $i \neq 1$, $i \notin S$ & $j \in S$

Example:-



$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

Let $g(i,S)$ be the length of the shortest path in which we start at vertex i, visit all vertex in S and reach the vertex 1.

Therefore, the required solution is $g(1, V-\{1\})$

Let us find the optimal solutions for $|S| = 0, 1, 2 \cdots (n-1)$

When $|S| = 0$,

The optimal solution: $g(2,\phi) = c_{21} = 5$

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

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

When $|S| = 1$,

The optimal solution: $g(2,\{3\}) = c_{23} + c_{31} = 9+6 = 15$

$g(2,\{4\}) = c_{24} + c_{41} = 10+8 = 18$

$g(3,\{4\}) = c_{34} + c_{41} = 12+8 = 20$

$g(3,\{2\}) = c_{32} + c_{21} = 13+5 = 18$

$g(4,\{2\}) = c_{42} + c_{21} = 8+5 = 13$

$g(4,\{3\}) = c_{43} + c_{31} = 9+6 = 15$

when |s| = 2.

optimal solution: $g(2,\{3,4\}) = \min(c_{23}+g(3,\{4\}), c_{24}+g(4,\{3\}))$

$= \min = \{29, 25\} = 25$

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

$= \min = \{31, 25\} = 25$

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

$= \min = \{23, 27\} = 23$

when |s| = 3.

optimal solution is given by

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

$c_{14}+g(4,\{2,3\})\}$

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

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

Having calculated all posible g terms, we are at a position to take decisions on solving the given problem. The shortest distance of traveling sales person problem. in example is $g(1,\{2,3,4\})$

① Therefore, at this stage we can take a decision to reach vertex 2 because reaching vertex 2 gives min distance

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

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

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

Having taken this decision to reach vertex 2 we are in position to taken the next decision. The next is on visiting the next vertex after v_2.

$\Rightarrow \{c_{12}+g(2,\{3,4\})$ is minimum, we need to follow $g(2,\{3,4\})$ [not $c_{13}+g(3,\{2,4\}$ and $c_{14}+g(4,\{2,3\})$]

⑤ By following $g(2,\{3,4\})$ we need to visit vertex 4 after vertex 2.

As $g(2,\{3,4\}) = \min\{c_{23}+g(3,\{4\}), c_{24}+g(4,\{3\})\}$

$\min\{29,25\} = 25$

③ And the next decision is to visit vertex 3 because that is the only possibility and the last decision is to reach 1.

An optimal tour of the graph has a length of 35. This tour can be retained by retaining with each $g(i,s)$ the value of j that minimizes the right hand side of the equation.

Let $J(i,s)$ be this value then

$J(1,\{2,3,4\}) = 2$ tour starts from 1 and goes to 2

$J(2,\{3,4\}) = 4$ &

$J(4,\{3\}) = 3$

Hence optimal tour is 1,2,4,3,1

Analysis:-

Time complexity : $\theta(n^2 2^n)$

space complexity : $\theta(n2^n)$.

Drawback of this algorithm is its space complexity.

⇒ Matrix Chain Multiplication:

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

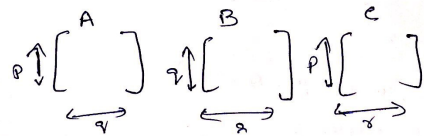
Input: n matrices $A_1, A_2, \ldots A_n$ of dimensions $P_1 \times P_2$, $P_2 \times P_3 \ldots P_n \times P_{n+1}$ respectively.

Goal: To compute matrix product $A_1 A_2 \ldots A_n$

Problem: In what order should $A_1, A_2 \ldots A_n$ be multiplied so that it could take the minimum number of computations to derive the product.

For performing matrix multiplication the cost:

⇒ Let A and B be two matrices of dimensions $P \times q$ and $q \times R$



then C = AB C is of dimension $P \times R$

→ Thus C takes n scalar multiplications and (n-1) scalar additions.

→ Consider an example of the best way of multiplies 3 matrices:

Let A_1 of dimension 5×4, A_2 of dimension 4×6, and A_3 of dimension 6×2

$(A_1A_2)A_3$ takes $(5\times4\times6)+(5\times6\times2) = 180$

$A_2(A_2A_3)$ takes $(5\times4\times2)+(4\times6\times2) = 88$

Thus $A_1(A_2A_3)$ is much cheaper to compute than $(A_1A_2)A_3$ although both lead to the same final answer. Hence optimal is 88.

⇒ To solve this problem using dynamic programming method we will perform following steps

Step 1:- Let m_{ij} denote the cost of multiplying $A_i \cdots A_j$ where the cost of measured in the number of scalar multiplications.

Here $m[i,i] = 0$ for all i, and $m[1,n]$ is required solution.

Step 2:- The sequence of decisions can be made using the principle of optimality.

We will apply the following formula for computing each sequence.

$$m[i,j] = \begin{cases} \text{not defined} & \text{if } i=j \\ 0 & \text{if } i=j \\ \min_{i\le k<j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} \end{cases}$$

Prob :- $A_1 \times A_2 \times A_3 \times A_4 \times A_5$

4×10 10×3 3×12 12×20 20×7

$P_0\ P_1\ P_1 P_2\ P_2 P_3\ P_3 P_4\ P_4 P_5$

$m[1,2] = \min_{1 \le k < 2} \{ m[1,1] + m[1+1,2] + P_0 P_1 P_2 \}$

$m[1,3] = \min_{1 \le k < 2} \{ 0 + 0 + 4 \times 10 \times 3 \}$

$m[1,2] = 120$

$m[2,3] = \min_{2 \le k < 3} \{ m[2,2] + m[2+1,3] + P_1 P_2 P_3 \}$

$m[2,3] = \min_{2 \le k < 3} \{ 0 + 0 + 10 \times 3 \times 12 \}$

$= 360$

$m[3,4] = \min_{3 \le k < 4} \{ m[3,3] + m[3+1,4] + P_2 P_3 P_4 \}$

$= m[3,4] = \min_{3 \le k < 4} \{ 0 + 0 + 3 \times 12 \times 20 \}$

$= 720$

$m[4,5] = \min_{4 \le k < 5} \{ m[4,4] + m[4+1,5] + P_3 P_4 P_5 \}$

$= m[4,5] = \min_{4 \le k < 5} \{ 0 + 0 + 12 \times 20 \times 7 \}$

$= 1680$

$m[1,3] = \min_{1 \le k < 3} \{ m[1,1] + m[1+1,3] + P_0 P_1 P_3 \}$

$k = 1 \ : \ 360 + 4 \times 10 \times 12 = 840$

$k = 2 \ : \ 120 + 0 + 4 \times 3 \times 12 = 264$

	1	2	3	4	5
1	0	120	264	1080	1344
2	X	0	360	1320	1350
3	X	X	0	720	1140
4	X	X	X	0	1680
5	X	X	X	X	0

$m[2,4]$: min
$2 \le k < 4$

$k=2$: $m[2,2] + m[2+1,4] + P_1 P_2 P_4 = 0 + 720 + 10 \times 3 \times 20$
$= 1320$

$k=3$: $m[2,3] + m[3+1,4] + P_1 P_3 P_4 = 360 + 0 + 10 \times 2 \times 20 = 2760$

$m[3,6]$: min
$3 \le k < 5$

$k=3$ $m[3,3] + m[3+1,5] + P_2 P_3 P_5 = 0 + 1680 + 3 \times 12 \times 7 = 1932$

$k=4$ $m[3,4] + m[4+1,5] + P_2 P_4 P_5 = 720 + 0 + 3 \times 20 \times 7 = 1140$

$m[1,4]$: min
$1 \le k < 4$

$k=1$: $m[1,1] = m[1+1,4] + P_0 P_1 P_4 = 0 + 1320 + 4 \times 10 \times 20 = 2120$

$k=2$: $m[1,2] = m[2+1,4] + P_0 P_2 P_4 = 120 + 720 + 4 \times 3 \times 20 = 1080$

$k=3$: $m[1,3] + m[3+1,4] + P_0 P_3 P_4 = 264 + 0 + 4 \times 12 \times 20 = 1224$

$m[2,5]$: min
$2 \le k < 4$

$k=2$ $m[2,2] + m[2+1,5] + P_1 P_2 P_5 = 0 + 1140 + 10 \times 3 \times 7 = 1350$

$k=3$ $m[2,3] + m[3+1,5] + P_1 P_3 P_5 = 360 + 1680 + 10 \times 2 \times 7 = 2880$

$k=4$ $m[2,4] + m[4+1,5] + P_1 P_4 P_5 = 1320 + 0 + 10 \times 20 \times 7 = 2720$

$m[1,5]$: min
$1 \le k < 5$

$m=1$ $m[1,1] + m[1+1,5] + P_0 P_1 P_5 = 0 + 1350 + 4 \times 10 \times 7 = 1630$

$m=2$ $m[1,2] + m[2+1,5] + P_0 P_2 P_5 = 120 + 1140 + 4 \times 3 \times 7 = 1344$

$m=3$ $m[1,3] + m[3+1,5] + P_0 P_3 P_5 = 264 + 1680 + 4 \times 12 \times 7 = 2280$

$m=4$ $m[1,4] + m[4+1,5] + P_0 P_4 P_5 = 1080 + 0 + 4 \times 20 \times 7 = 1680$

$k=4$
$m[1,4] + m[5,5]$

$(A_1 A_2 A_3 A_4) A_5$
$4 \times 10 \quad 10 \times 3 \quad 3 \times 12 \quad 12 \times 20 \quad 20 \times 7$
$(120) \quad (720) \quad 420$
120

$120 + 720 + 420 + 84 = 1344$

84

$A_1\ A_2\ A_3\ A_4\ A_5$

$k = 2$ $\quad (A_1 \times A_2)\ (A_3 \times A_4 \times A_5)$

$m[1,5]$ $\quad 4\times10\ \ 10\times3\ \ \ 3\times12\ \ 12\times20\ \ 20\times7$

$m[1,2] + m[3,5]$

$m = 4\ (m[3,5]) \quad (A_1 \times A_2)\ ((A_3 \times A_4)\ A_5)$

$4\times10\ \ 10\times3\ \ 3\times12\ \ 12\times20\ \ 20\times7$

$(120)\ \ (720\ \ A_5)$

$4\times3 \qquad 3\times20 \quad 20\times7$

$120 \qquad 420$

$4\times3 \qquad 3\times7$

84

4×7

120
720
420
84
1344

Algorithm:-

1. Algorithm matrix-chain-mul (array $p[1\ldots n]$)
2. / we can maintain a parallel array $s[i,j]$ in which
3. we will store the value of providing the $s[1\ldots n-1, 2\ldots n]$
4. // k is optimal split such that for $A_{i\ldots s}$ for multiply the subchain A_{ik} and then multiply subchain $A_{k+1,j}$

for $i:=1$ to n do

$m[i,i] = 0;$

for $l := 2$ to n do

{ for $i:=1$ to $(n - len + 1)$ do

```
}
    for i
    j = i + len - 1;
    m[i,j] = ∞;
    for k = i to j-1 do
    q = m[i,k] + m[k+1,j] + p[i-1] * p[k] * p[j]
    if (q < m[i,j])
    {
        m[i,j] = q;
        s[i,j] = k;
    }
}
}
}
return m[1,n]
}
```

Analysis :-

Time complexity : $O(n^3)$

Space complexity : $O(n^2)$

Optimal Binary Search Trees:

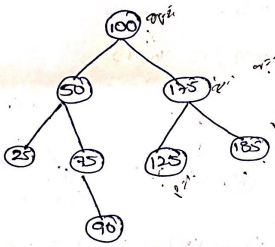
Binary Search tree:

A Binary search Tree is a binary tree satisfying the following conditions.

(i) At every node its left child is smaller and right child is greater

(ii) All elements in left subtree of root must be smaller than root and all the elements in Right subtree must be greater than the root.

ex:



Search Operation:

To search for an element X in a binary search tree, initially a pointer (p) is set to the node that pointer moves down the tree either to right or left depends upon if the element X is less than p or greater.

The problem of optimal binary search tree is to construct an optimal binary search tree given the following data.

(i) $a_1, a_2, a_3 \cdots a_n$

(ii) $P_1, P_2, P_3 \cdots P_n$

(iii) $q_0, q_1, q_2 \cdots q_n$

a_1, a_2, a_n are given set of identifiers

$P[i]$ is probability of successful search

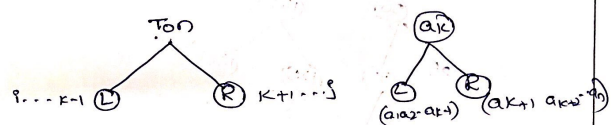
$q[i]$ is probability of unsuccessful search

objective is to construct tree with minimum cost

$\sum P[i] + q[i] \quad 1 \le k \le n \quad 0 \le k \le n$

obtain tree with minimum cost is set to optimal binary tree.

let root of OBST is $T_{0,n}$



$c[T_{0,n}] = c[L] + c[R] + \sum_{1 \le i \le n} P[i] + \sum_{0 \le i \le n} q[i]$

$w[T_{0,n}]$: weight of each $T_{0,n}$ then expected cost of search tree would be

$c[L] + c[R] + w[0, k-1] + w[k, n]$

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

$$w(1,j) = w(i, j-1) + P[j] + q[j]$$

where $w(i,j) = \sum_{1 \le i \le j} P(j) + \sum_{0 \le i \le j} q[i]$

$= P(i) + P(2) + \cdots P_{j-1} + P_j + q_0 + q_1 + \cdots$

$r[i,j]$ = k root of T_{ij}

Problem:- consider n=4 and a_1, a_2, a_3, a_4 = (do, if, int, while) $P[1:4] = (3,3,1,1)$ and $q[0:4] = (2,3,1,1,1)$ construct optimal binary search tree

Solution:-

Initially $w[i,i] = q[i]$ // unsuccessfull search when i=0

$c[i,i] = 0$

$r[i,i] = 0 \quad 0 \le i \le 4$

i=0	i=1	i=2	i=3	i=4
w[0,0]=2	w[1,1]=3	w[2,2]=1	w[3,3]=1	w[4,4]=1
c[0,0]=0	c[1,1]=0	c[2,2]=0	c[3,3]=0	c[4,4]=0
r[0,0]=0	r[1,1]=0	r[2,2]=0	r[3,3]=0	r[4,4]=0

when OBST with one element:

i=0 & j=1

$w[i, i+1] = P[i+1] + q[i+1] + w[i, i] \Rightarrow p_j + q_j + w[i, j-1]$

$w[0,1] = P_1 + q_1 + w[0,0]$

$= 3 + 3 + 2 = 8$

$c[i, i+1] = w[i, i+1] \Rightarrow \min_{i<k\le j}\{c[i,k-1] + c[k,j]\} + w[i,j]$

$c[0,1] = \min_{k=1}\{c[0,1-1] + c[1,1] + w(0,0)\} = 0+0+8 = 8$

$r[i, i+1] = i+1 = 1$

(iii) when $i=2$ & $j=4$ $k=3,4$

$w(2,4) = P_4 + q_4 + w(2,3)$

$= 1+1+3 = 5$

$c(2,4) = \min\{c(i,k-1) + c(k,j)\} + w(i,j)$

$k=3$ $c(2,4) = c(2,2) + c(3,4) + w(2,4)$

$= 0+3+5 = 8$

$k=4$ $c(2,4) = c(2,3) + c(4,4) + w(2,4)$

$= 3+0+5 = 8$

for $k=3,4$ $c(2,4)$ is min

$r(2,4) = 3+5 = 8$

$r(2,4) = 3$

→ when OBST with 3 elements.

when $i=0$ $j=3$ $k=1,2,3$

$w[0,3] = P_3 + q_3 + w[0,2]$

$= 1+1+12 = 14$

$k \neq i$ $c_{03} = \min\{c[i,k-1] + c[k,j]\} + w(i,j)$

$k=1$ $c[0,3] = c_{00} + c_{13} + w[0,3]$

$= 0+12+14 = 26$

$k=2$ $c[0,3] = c_{01} + c_{23} + w[0,3]$

$= 8+3+14 = 25$

$k=3$ $c[0,3] = c_{02} + c_{33} + w(0,3]$

$= 19+14 = 33$

for $k=2$ c_{03} is min $c_{03} = 25$

$r_{03} = 2$

(ii) when $i=1$ & $j=4$ $k=2,3,4$

$w[1,4] = P_4 + q_4 + w[1,3]$

$= 1+1+9 = 11$

$c[1,4] = \min_k \{ c[1,k-1] + c[k,j] \} + w[i,j]$

$k=2$ $c_{14} = c_{11} + c_{24} + w[1,4]$

$= 0 + 8 + 11 = 19$

$k=3$ $c_{14} = c_{12} + c_{34} + w_{14}$

$= 7 + 3 + 11 = 21$

$k=4$ $c_{14} = c_{13} + c_{44} + w_{14}$

$= 12 + 0 + 11 = 23$

for $k=2$ $c[1,3]$ is minimum

$R[1,4] = 2$

OBST with Four elements.

(i) when $i=0$ $j=4$ $k=1,2,3,4$

$w[0,4] = P_4 + q_4 + w[0,3]$

$= 1+1+16 = 16$

$c[0,4] = \min_k \{ c[i,k-1] + c[k,j] \} + w[i,j]$

$k=1$ $c[0,4] = c_{00} + c_{14} + w_{04}$

$= 0 + 19 + 16 = 35$

$k=2$ $c[0,4] = c_{01} + c_{24} + w_{04}$

$= 8 + 8 + 16 = 32$

$k=3$ $c[0,4] = c_{02} + c_{34} + w_{04}$

$= 19 + 3 + 16 = 38$

K=4 $c[0,4] = c_{03} + c_{44} + w_{04}$

$= 25 + 0 + 16$

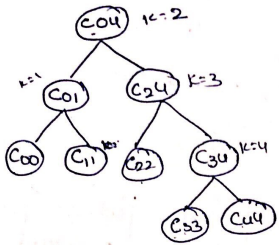
$= 41$

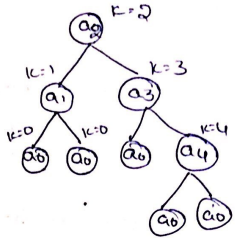
for K= 2 $c[1,3]$ is minimum

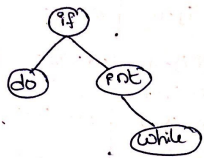
$c[0,4] = 32$

$r[0,4] = 2$

$w[0,0]=2$ $c[0,0]=0$ $r[0,0]=0$	$w[1,1]=3$ $c[1,1]=0$ $r[1,1]=0$	$w[2,2]=1$ $c[2,2]=0$ $r[2,2]=0$	$w[3,3]=1$ $c[3,3]=0$ $r[3,3]=0$	$w[4,4]=1$ $c[4,4]=0$ $r[4,4]=0$
$w[0,1]=8$ $c[0,1]=8$ $r[0,1]=1$	$w[1,2]=7$ $c[1,2]=7$ $r[1,2]=2$	$w[2,3]=3$ $c[2,3]=3$ $r[2,3]=3$	$w[3,4]=3$ $c[3,4]=3$ $r[3,4]=4$	
$w[0,2]=12$ $c[0,2]=19$ $r[0,2]=1$	$w[1,3]=9$ $c[1,3]=12$ $r[1,3]=2$	$w[2,4]=5$ $c[2,4]=8$ $r[2,4]=3$		
$w[0,3]=14$ $c[0,3]=25$ $r[0,3]=2$	$w[1,4]=11$ $c[1,4]=19$ $r[1,4]=2$			
$w[0,4]=16$ $c[0,4]=32$ $r[0,4]=2$				







Complexity Analysis:-

Time complexity = $O(n^2)$

space complexity = $O(n^2)$

⇒ Reliability Design:

* In this problem a system is to be designed using multiplicative optimization function.

* System consists of several devices. Let r_i be the reliability of device D_i.

* Let us suppose there are 3 devices D_1, D_2 & D_3 having reliability 0.9 each. then the reliability of entire system is $0.9 * 0.9 * 0.9 = 0.729$. ie the system will work when all 3 devices are working and system will fail even one device fails.

*Though each device reliability is high the reliability of entire system is less. Our problem is to increase the reliability of the system by duplicating devices.

* multiple copies of same devices are connected in parallel by switching the circuits, they determine which device in a group are functioning properly. At each stage they make use of one such device.

* Let us suppose stage i has m_i copies of device D_i, the probability that all m_i copies fail is $(1-r_i)^{m_i}$. The reliability of stage i is $1-(1-r_i)^{m_i}$.

*Hence the reliability improve with m_i. If $r_i = 0.9$, and $m_i = 2$ the stage reliability becomes

$$1-(1-0.9)^2 = 0.99$$

*Let us assume that the reliability of stage i is given by a function $\phi_i(m_i)$

The reliability of entire system is $\prod_{1 \le i \le n} \phi_i(m_i)$.

Let c_i be cost of each unit of device i and let c be the maximum allowable cost of system being designed.

The maximization problem is

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

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

$m_i \ge 1$ and integer, $1 \le i \le n$

We use S^i to represent the set of all ordered tuples (r, c) where r is the reliability & c is the cost.

Problem:

The problem is to construct a system of n types of devices D_i, $1 \le i \le n$ with some multiplicity of each device $m_i \ge 1$, $1 \le i \le n$ such that to maximize the reliability subjected to total cost of the system $\le c$.

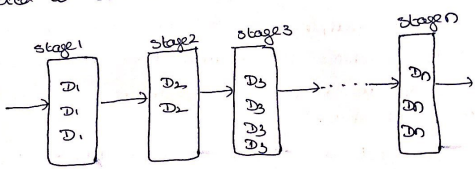


Fig: multiple devices connected in parallel in each stage.

Let r_i be the reliability of device D_i. If the multiplicity of D_i is m_i then the reliability of stage i is $1-(1-r_i)^{m_i}$. Hence reliability will improve rapidly with m_i.

Let the reliability of stage i is $\phi(m_i)$, $1 \le i \le n$.

$\Rightarrow$ Reliability of total system $= \prod_{1 \le i \le n} \phi_i(m_i)$

Formally the problem is to maximize $\prod_{1 \le i \le n} \phi_i(m_i)$ subjected to $\sum_{1 \le i \le n} c_i(m_i)$ $m_i \ge 1$ & integer, $1 \le i \le n$ where c_i is cost of D_i

Solution:- This can be solved by using dynamic programming technique similar to knapsack problem. Here each $c_i > 0$. We have to find each m_i

$$1 \le m_i \le u_i$$

where u_i (upper limit to m_i) $= \lfloor (c + c_1 - \sum_1^n c_i) / c_i \rfloor$

An optimal solution $m_1, m_2, m_3 \cdots m_n$ is the result of a sequence of decisions, one decision for each m_i

Let $f_i(x)$ represent the maximum value of $\sum_{1 \le j \le i} \phi_j(m_j)$ subject to $\sum_{1 \le j \le i} c_j m_j \le x$ and $1 \le m_j \le u_j$, $1 \le j \le i \Rightarrow$ the value of an optimal solution is $f_n(c)$.

using the principle of optimality:-

$$f_n(c) = \max \{ \phi_n(m_n) f_{n-1}(c - c_n m_n) \} \quad 1 \le m_n \le u_n$$

for any $f_i(x)$, $i \ge 1$, this generalizes to

$$f_i(x) = \max \{ \phi_i(m_i) f_{i-1}(x - c_i m_i) \quad 1 \le m_i \le u_i$$

Clearly $f_0(x) = 1$ for all x, $0 \le x \le c$

Let S^i consist of tuples of the form (f, x) where $f = f_i(x)$

Let S^i is the set of all tuples obtained from S^{i-1} by choosing $m_i = j$

$\rightarrow S^i = \cup \{ j \le u_i \} S^i_j$

Dominance rule: (f_1, x_1) dominates (f_2, x_2) if $f_1 \ge f_2$, $x_1 \le x_2$.

Example: Devices $D_1, D_2, D_3 \rightarrow n = 3$ $c_1 = 30, c_2 = 15, c_3 = 20$

$C = 105$; $r_1 = 0.9$, $r_2 = 0.8$, $r_3 = 0.5$

$u_1 = \lfloor (C + c_1 - \sum_1^3 c_3) / c_1 \rfloor$

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

$= \lfloor (135 - 65) / 30 \rfloor = \lfloor 75/30 \rfloor = 2$

$u_2 = \lfloor (C + c_2 - \sum_1^3 c_3) / c_2 \rfloor$

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

$= \lfloor (105 - 50) / 15 \rfloor = \lfloor 55/15 \rfloor = 3$

$u_3 = \lfloor (C + c_3 - \sum_1^3 c_3) / c_3 \rfloor$

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

$= \lfloor (125 - 65) / 20 \rfloor = \lfloor 60/20 \rfloor = 3$

	n	r	u
c_1	30	0.9	2
c_2	15	0.8	3
c_3	20	0.5	3

$C = 105$

$S^0 = (r, c) = (1, 0)$

consider D_1 $S_1^1 = \{(0.9, 30)\}$

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

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

$1-(1-r_1)^2 = 1-(1-0.9)^2$
$= 1-(0.1)^2$
$= 1-0.01$
$= 0.99$

consider D_2 $S_1^2 = \{(0.8, 15$

$= \{(0.8 \times 0.9, 15+30), (0.8 \times 0.99, 15+60)\}$

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

$S_2^2 = \{(0.9 \times 0.96, 30+30), (0.99 \times 0.96, 60+30)\}$

$= \{(0.864, 60), (0.9504, 90)\}$ → infeasible

$S_3^2 = \{(0.9 \times 0.992, 45+30), (0.$ 60+45)

$S^2 = \{(0.8928, 75)\}$

$1-(1-r_3)^3$
$1-(1-0.5)^3$
$1-(0.2)^3$
$1-0.008$
$r_3 = (0.992, 45)$

$1-(1-r_2)^2$
$1-(1-0.8)^2$
$1-(0.2)^2$
$1-0.04$
$r_2 = 0.96$ 30
(r c)

$S^2 = \{(0.72, 45), (0.792, 75), (0.864, 60), (0.8928, 75)\}$

note:- As the tuple (0.792, 75) has been dominated by (0.864, 60), (0.792, 75) has been eliminated.

$S^3 = S_3^3 = \{(0.5 \times 0.9, 45+20), (0.864 \times 0.5, 60+35), (0.8928 \times 0.5, 75+20)\}$

$= \{(0.36, 65), (0.432, 80), (0.4464, 95)\}$

$S_2^3 = \{(0.54, 85), (0.648, 100)\}$ (115) by exceeded interest rate

$1-(1-0.5)^2$
$1-(1+0.5^2)$

$S_3^3 = \{(0.63, 105)\}$ (120) by interest rate (135)

$1-(0.5)^2$
$1-0.25$
$= 0.75, 40$

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

$1-(1-0.3)^3$
$1-(1-0.5)^3$
$1-(0.5)^3$
$1-0.125$
$= 0.875$

$= \{(0, 36, 65), (0.432, 80), (0.54, 85), (0.648, 100)\}$

'back track for $m_1 = 1$ - S_1 get answer at S_1^1

$m_2 = 2$ - S_2 get answer at S_2^2

$m_3 = 2$ - S_3 get answer at S_3^3

D_1	D_2	D_3
1	2	2

$\pi r_i = 0.648$

$\sum c_i = 100$

Algorithm:-

```
REL-DESIGN (C, n, c, R)
{
    S^0 = {(1, 0)}
    for i := 1 to n do {find u_i}
    for i := 1 to n do
    {
        S_i^0 = U_{1<=j<=u_i} S_j^{i-1} + (r_i, c_i)
    }   S^i = merge purge (S_i^0)
```

```
for (i=n ; i>1 ; i--)
{
    trace - back mi
}
}
```

Syllabus:-

Branch and Bound: General method, applications - 0/1 knapsack problem, LC Branch and Bound solution, FIFO Branch and Bound solution, Non-deterministic algorithms, NP-Hard, NP-complete classes, cooks theorem. Travelling sales person problem.

NP Hard NP complete Problems:

Basic concepts. Non-deterministic algorithms, NP-hard and NP-complete classes, cook's theorem.

The branch and bound is a systematic method for handling optimization problems. This method is applied where greedy method and dynamic method fails.

The functionality of branch and bound is like BFS for searching optimal solution and it uses bounding function to determine which node and when to expand and there by provides solution.

collection of every possible path from root to the each node is called the statespace.

The problem states from which the tuple is defined the solution space are called solution states. A live node is one which is generated and all children of 'n' are not completly generated.

A dead node is generated one which cannot be expanded further or all all its children are generated. The live node whose children are currently generated called e-node.

Breadth first node generation with bounding function is called branch and bound.

Difference b/w backtracking and branch and bound

back tracking	branch and bound
(1) It is used to find all possible solutions available to the problem	1. It is used to solve optimization problem.
(2) It traverse tree by DFS	2. It may traverse tree in any manner. DFS or BFS.
(3) It realizes that it has made a bad choice and undoes the last choice by backup	3. It realizes that it is already has a better optimal solution and the pre-solution lead to so it uses that solution.
(4) It searches the statespace tree until it found a solution	4. It completely searches the state space tree to get optimal solution
(5) It involves feasibility function	(5) It involves bounding function.

These are 3 common search strategies of branch and bound technique they are.

① FIFO Branch and Bound
② LIFO Branch and Bound
③ Least cost Branch and Bound.

BFS will be called FIFO (uses queue)
DFS will be called LIFO (uses stack).

FIFO Branch and Bound:

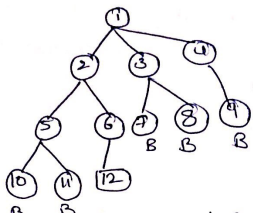
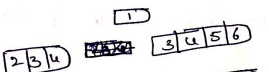


fig: state space tree.

Assume the node 12 is an answer node To begin with node 1 is E-node, next children node, are generated and these are placed in queue.



Queue storing live nodes.

Now we will delete an element from queue node 2, node 2 becomes E-node generate childrens of node 2

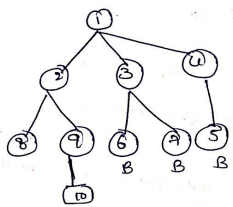
3 is deleted from queue and it becomes E-node. childrens of node 3 are generated. and these live nodes are killed by bounding function. so we will not include in the queue.

4 is deleted from queue child of node 4 are generated. and these lives nodes are killed by

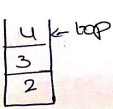
bounding function.

Next delete 5 and 10 & 11 are added. and killed by bounding function. The child node of 6 is 12 satisfies condition. which is answer node so terminates.

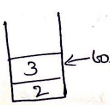
LIFO BRANCH AND BOUND



Initially stack is empty. The operations of stack are



Children of 4 are killed by bounding function



Children of node 3 are 6 & 7 those are killed by bounding function

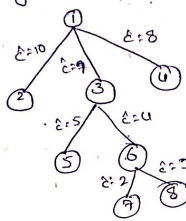
children of node 2 are added

children of node 9 is 10

Answer node find and all conditions are satisfied so terminates

Least cost branch and bound :-

Here the node with minimum cost should explore further. Here every node has some cost.



Note:- If any two nodes have some cost then left most node is always choosen.

To solve the least cost we choose method called LC-search. It is a kind of search in which a least cost bound for reaching the answer node.

LC is find out by ranking function denoted by

$$\hat{c}(x) = f(h(x)) + \hat{g}(x)$$

where $f(h(x))$ denotes level of node in BFS

$g(x)$ = estimate additional effort needed to reach a node.

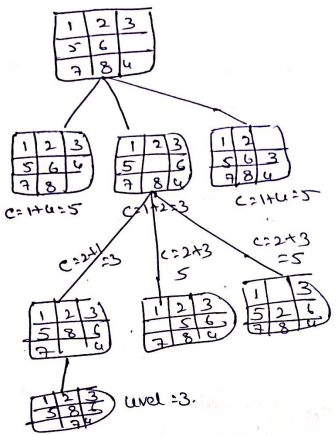
Ex:- 8-puzzle

Initial state

1	2	3
5	6	
7	8	

Final state

1	2	3
5	8	6
	7	4



Algorithm for BB:

```
1. Algorithm Branch-Bound [ ] {
2. // E-node is node pointer.
3. E ← new nodes //
4. while (true)
5. {
6. If (E is a final leaf) then
7. {
8. write (path from E to the root)
9. return;
10. expand (E);
11. if (heap is empty.) then {
12. write ("there no solution")
13. return; }.
14. E ← delete - top (H); }.
```

⇒ 0/1 knapsack Problem:-

Problem Statement:-

The 0/1 knapsack problem states that, there are n objects given and capacity of knapsack is 'm'. Then select some objects to fill the knapsack in such a way that it should not exceed the capacity of knapsack and maximum profit can be earned.

Branch and bound technique is used to find solution to knapsack problem.

Branch and bound technique cannot be directly applied to the knapsack problem, because the branch and bound deals with minimization problems.

The knapsack problem is modified and converted into the minimization problem.

The modified problem is

minimize profit $= -\sum_{i=1}^{n} P_i x_i$

subject to $\sum_{i=1}^{n} w_i x_i$

such that $\sum w_i x_i \leq m$ and

$x_i = 0$ or 1, where $1 \leq i \leq n$

LC Branch and Bound solution:

The LC Branch and Bound solution can be obtained using fixed tuple size formulation.

The steps to be followed for LC BB solution are

(1) Draw state space tree

(2) compute $\hat{c}(\cdot)$ and $u(\cdot)$ for each node.

(3) if $\hat{c}(x) >$ upper kill node x

(4) otherwise the minimum cost $\hat{c}(x)$ becomes E-node. Generate children for E-node.

(5) Repeat step 3 and step 4 until all the nodes get covered.

(6) The minimum cost $\hat{c}(x)$ becomes the answer node. trace the path in backward direction from x to root for solution subset.

example: consider knapsack instance $n=4$ with capacity $m=15$ such that:

object i	P_i	w_i
1	10	2
2	10	4
3	12	6
4	18	9

solution: Let us design state space tree using fixed tuple size formulation. The computation of $\hat{c}(x)$ and $u(x)$ for each node x is done.

Step:-1 $U(i)$ can be computed as

for $i = 1, 2$ and 3

$-\sum P_i = -(10+10+12)$

$U(i) = -32$

$u = \sum_{i=1}^{2} P_i x_i$ without fraction

$c = \sum_{i=1}^{2} P_i x_i$ with fraction

* If we select $i = u$, then it will exceed capacity of knapsack.

The computation of $\hat{c}(i)$ can be done as

$$\hat{c}(i) = U(i) - \left[\frac{m - \text{current total weight}}{\text{actual weight of remaining object}}\right] * \left[\text{Actual profit of remaining object}\right]$$

$$\hat{c}(i) = \left[-32 - \left[15 - \left[\frac{(2+4+6)}{9}\right] * 18\right]\right]$$

$$= \left[-32 - \left[\frac{3}{9} * 18\right]\right]$$

$$= \hat{c}(i) = -38$$

$U(i) = -32$

$\hat{c}(i) = -38$

with x_1

$U = -32$

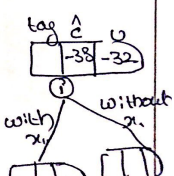
$\hat{c} = -38$

without x_1

we can place objects x_2 and x_3 into knapsack then

$U = -[\text{Profit of } x_2 + \text{Profit of } x_3]$

$U = -(10+12) = -22$



$$\hat{c} : U(i) - \left[\frac{m - \text{current total weight}}{\text{Actual weight of remaining object}}\right] * \left[\text{actual profit of remaining object}\right]$$

$$\hat{c} = -22 - \left[\frac{15-[4+6]}{9}\right] * 18$$

$$= -22 - \left[\frac{15-10}{9}\right] * 18$$

$$= -22 - \left(\frac{5}{9} * 18\right)$$

$$= -22 - 10 = -32$$

without x_1

$U = -22$

$\hat{c} = -32$

From with x_1 and without x_1, with x_1 is min

cost $= -38$ [-38 is less than -32]

↳ $\hat{c} \leq u \times$

Branches are expanded from node with LC "-38"

Step 3

with x_2

$U = -32$

$C(x) = -38$

without x_2

We can place objects x_1 and x_3 into knapsack then

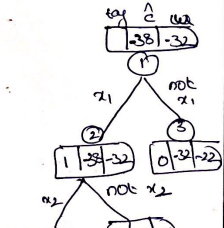
$$U = \left[\text{profit}(x_1) + \text{profit}(x_3)\right]$$

$$= U = -[10+12]$$

$$= -22$$

$$\hat{c} = U(2) - \left[\frac{m - \text{current total weight}}{\text{Actual weight of remaining object}}\right] * \text{actual profit of remaining object}$$

$$= -22 - \left[\frac{15 - [2+6]}{9}\right] * 18$$



$= -22 - \left[\frac{15-8}{9}\right] * 18$

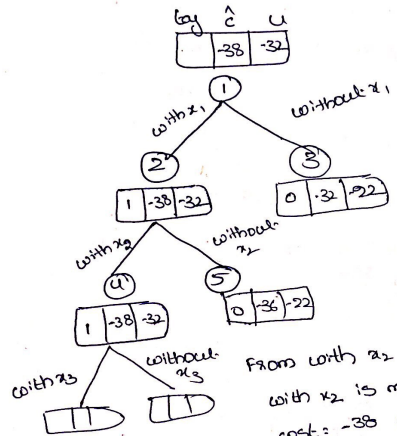
$= -22 - \left(\frac{7}{9}\right) * 18^{2}$

$= -22 - 14 = -36$

without x_2

$u = -22$

$\hat{c} = -36$



From with x_2 and without x_2

with x_2 is min $(u > \hat{c})$

cost = -38 (-38 less than -36)

expand from node LC = -38

Step 4

with x_3

here $u = -32$

$c(x) = -38$

without x_3

we can place object x_1, x_2, x_4 into the knapsack bag. then

$u = -\left[p(x_1 + x_2 + x_4)\right]$

$= -\left[10 + 10 + 18\right] = -38$

$\hat{c} = u(3) - \left[\frac{m - \text{current total weight}}{\text{Actual weight of remaining object}}\right] * \text{actual profit of remaining object}$

$$= -38\left[\frac{15-[15]}{0}\right]*0$$

$$= -38$$

without x_3

$U = -38$

$\hat{c} = -38$

from with x_3 and without x_3

without x_3 is minimum

$U = -38$ is less than -32

so expand from without x_3

with x_4

$U = -38$

$C = -38$

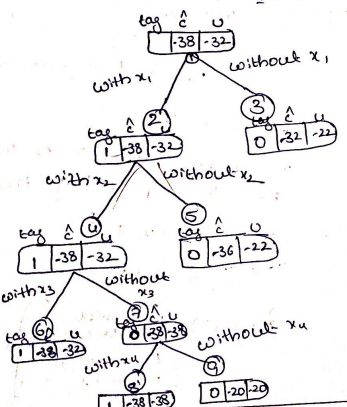
without x_4

$U = -[P_1(x_1 + x_2)]$

$= -20$

$\hat{c}(x) = -20 + [0]$

$= -20$



(7)

∴ Therefore the path is

$1 \to 2 \to 4 \to 7 \to 8$

(x_1, x_2, x_3, x_4)

$(1 \quad 1 \quad 0 \quad 1)$

maximum profit =

$10 + 10 + 0 + 18$

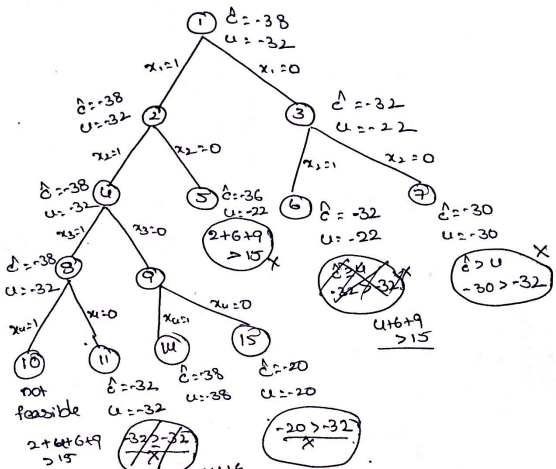
$= \boxed{38}$

FIFO BRANCH AND BOUND SOLUTION

The state space tree with variable tuple size formulation can be drawn and $\hat{c}(\cdot)$ and $u(\cdot)$ is computed

$n = 4$ $(P_1, P_2, P_3, P_4) = (10, 10, 12, 18)$ $m = 15$

$(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$



Initially upper $u(1) = -32$ then child of node 1 are generated. node 2 becomes E-node and hence 4, 5 are generated. node 4 are added in the list of the nodes. node 4 and 5 are added in the list of the nodes. next node 3 becomes enode and children 6, 7 are generated. As $\hat{c}(7) >$ upper we will kill node 7

→ Hence node 6 will be added in the list of live nodes. Node 4 is e-node and children 8, 9 are generated.

→ The upper is updated and it is now $u(9) = -38$ nodes 8 and 9 are added in the list of live nodes.

→ Nodes 5 & 6 becomes the next e-node but those values are exceed than m

node 5 : $2+6+9 > 15$ } so kill nodes 5 & 6
node 6 : $4+6+9 > 15$

→ Node 8 becomes next e node and children 10 & 11 are generated. As node 10 is infeasible do not consider.

→ As $\hat{c}(11) \geq$ upper Hence kill node.

→ Node 9 becomes next e-node and upper $= -38$ children 12 and 13 are generated.

→ But $\hat{c}(13) >$ upper so kill node 13

→ Finally node 12 becomes an answer node.

solution is $x_1 = 1 \quad x_2 = 1 \quad x_3 = 0 \quad x_4 = 1$

Object i	p_i	w_i
1	10	2
2	10	4
3	12	6
4	18	9

Solution :- $u(i) : i = 1,2,3$

$u(x) = -(10+10+12)$

$= -32$

$\hat{c}(x) = u - \left[\frac{m - \text{current total weight}}{\text{weight of remaining object}}\right] * \text{Profit of actual remaining object}$

$\hat{c}[x] = -32\left[\frac{-15-12}{9}\right] * 18$

$= -32-6 = -38$

Step 1

with x_1

$\hat{c} = -38$

$u = -32$

without x_1

we can place x_2 & x_3

$u = -(x_2+x_3)$

$= (-10+12) - 22$

$\hat{c} = -22\left[\frac{15-(0+6)}{9}\right] * 18$

$= -22-\left[\frac{9}{9} \times 18\right] = -32$

$u = -22$

$\hat{c} = -32$

Step 2

with x_2

$u = -32$

$\hat{c} = -38$

without x_2

same as above LCBB

$u = -22$

$c = -36$

Step 3

with x_2

$u = (P(x_2+x_3))$

$= -(10+12)$

$u = -22$

$c = -22\left[\frac{15-10}{9}\right] * 18$

$c = -32$

without x_2

$u = P(x_3+x_4)$

$u = -[12+18] = -30$

$c = -30\left[\frac{-15-15}{0}\right]$

$c = -30$

Step 3:- from node 4

with x_3

$C = -38$
$U = -32$ — same as above LCBB

without x_3

$C = -38$
$U = -38$

from node 5

with x_3

$= C = -36$
$U = -22$

without x_3

$U = 10+18 = 28$
$\hat{C} = -28$

Step 4:- from node 8

with x_4
not feasible.

without x_4

$U = -(x_1+x_2+x_3)$

$U = -[10+10+12]$

$U = -32$

$\hat{C} = -32\ [15-15]$

$\hat{C} = -32$

Step 5: from node 9

with x_4

U = with x_1, with x_2, with x_3, with x_4

$U = -[10+10+18]$

$U = -38$

$C = -38\ [15-(2+4+9)]$

$= -38$

$\hat{C} = -38$

without x_4

$U = -[10+10]$

$= -20$

$C = -20-0$

$C = 20$

$2+4+0+9$
$10+10+0+18$

x_1	x_2	x_3	x_4
1	1	0	1

Algorithm:. For computing $\hat{c}(x)$

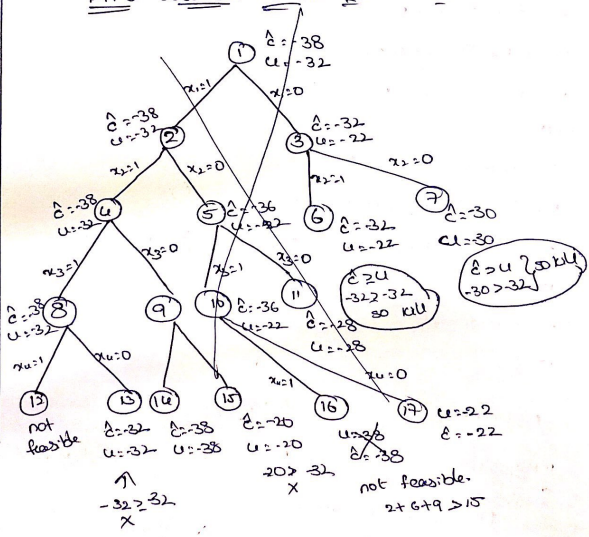
```
1. Algorithm Bound( total profit, total wt, k)
2. // Total profits denotes current total profit
3. // Total weight denotes current total weight
4. // k is index of last removed object
5// wt[] represents weight of object;
6 // p[] represents profit of object;
7. // m is weight capacity of knapsack.
8. {
9. pt ← total - profit;
10. wt ← total - wt;
11. for (i ← k+1 to n) do
12. {
13. wt ← wt + wt[i];
14. if (wt < m) then pt ← pt + p(i);
15. else return ( pt + (1-wt-m)/wi * P(i));
16. }
17. return pt;
18. }
```

Algorithm for compute $\hat{u}(x)$

```
1. Algorithm U-Bound (total profit, total wt, k, m)
{
    pt ← total - Profit;
    wt ← total - wt;
```

```
for ( i ← k+1 to n) do
{
    if (wt + w(i) <= m) then
    {
        pt ← pt - p[i];
        wt ← wt + w[i];
    }
    return pt;
}
```

FIFO solution for the above problem.



TRAVELLING SALES PERSON PROBLEM:

If there are 'n' cities and cost of travelling from one city to other city is given. A salesman start from one city and has to visit all the cities exactly once and has to return to the starting place with shortest distance or minimum cost.

Let $G = (V, E)$ be a directed graph defining an instance of the travelling sales person problem. Let c_{ij} be the cost of edge (i, j) and $c_{ij} = \infty$ if $(i, j) \notin E(G)$ and let $|V| = n$.

The following fig shows the state space tree organization for the case of a complete graph with $n = 4$. Each leaf node is a solution node.

The tree is defined by the path from root to the leaf node L.

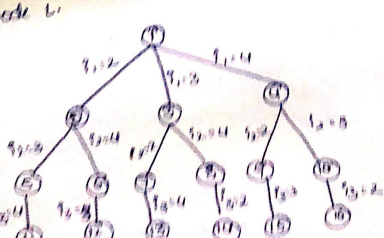


Fig: state space tree for the travelling sales person problem with $n = 4$ and i_0 to i_4.

Robust cost matrix:

A row or column is said to be reduced if it contains at least one zero and all remaining entries are

non-negative. A matrix is reduced if every row and column is reduced. If a constant 't' is choosen to be minimum entry in row 'i' or column 'j' then subtract it from all entries in row 'i' will introduce a zero into a row 'i'.

The total amount subtracted from the columns and rows is lower bound on the length of a minimum cost tour and can be used as the $\hat{c}(x)$ value for the root of state space tree.

With every node in state space tree, we associate a reduced cost matrix. To find a reduced cost matrix we have to following these steps:

(1) Change all entries in row i column j of 'A' to ∞

(2) Set A(j,i) to ∞

(3) Apply row reduction and column reduction except for rows and columns containing ∞

(4) Total cost for node 'S' can be calculated as

$$\hat{c}(s) = \hat{c}(R) + A(i,j) + x$$

where 'x' is the total amount subtracted in step 3.

Problem:- solve the following instance of travelling sales person cost matrix as shown below using LCBB

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

fig: cost matrix

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \begin{matrix} 10 \\ 2 \\ 2 \\ 3 \\ 4 \end{matrix} \quad \overline{21}$$

Reduce first row by 10
Reduce second row by 2
Reduce third row by 2
Reduce Fourth row by 3
Reduce Fifth row by 4

$$\begin{matrix} R_1-10 \\ R_2-2 \\ R_3-2 \\ R_4-3 \\ R_5-4 \end{matrix} \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix} \begin{matrix} & & & & \\ & & & & \\ & & & & \\ & & & & \\ \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\ 1 & 0 & 3 & \text{ignore} & \text{ignore} \end{matrix} = \begin{matrix} C_1-1 \\ C_3-3 \end{matrix} \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

ignore

$1+3=4$

Finally, the reduced cost matrix is

$$\begin{bmatrix} \infty & 10 & 19 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix} \quad 21+4$$

Reduced cost matrix $L=25$

Consider the path $(1,2)$: The first row and second column of reduced matrix are made infinity (∞) and element $(2,1)$ is also made ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix} \begin{matrix} \leftarrow \text{ignore} \\ \leftarrow \text{ignore} \\ \leftarrow \text{ignore} \\ \leftarrow \text{ignore} \\ \swarrow \text{path } 1,2 : \text{node } 2 \end{matrix}$$

Apply row and column reduction $\alpha = 0$

$$\hat{c}(2) = \hat{c}(1) + A(1,2) + \alpha$$

node $= 25 + 10 + 0 = 35$

consider path $(1,3)$ The first row third column of reduced matrix are made (∞) and element 3 is also made ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{bmatrix} \begin{matrix} \leftarrow \text{ignore} \\ \leftarrow \text{ignore} \\ \leftarrow \text{ignore} \\ \leftarrow \text{ignore} \\ \leftarrow \text{ignore} \end{matrix}$$

$\downarrow$ $\downarrow$ $\downarrow$
11 ignore ignore

Path $(1,3)$ node 3.

Above matrix is not reduced. Subtract 11 from 1st column then matrix is reduced.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

path 1,3 node 3

Apply row and column reduction $\alpha = 11$

$$\hat{c}(3) = \hat{c}(1) + A(1,3) + \alpha$$

node $= 25 + 17 + 11 = 53$

consider the path $(1,4)$: The first row and fourth column of reduced matrix are made ∞ and element 4,1 also made ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

fig: path 1,4 ; node 4

Apply row and column reduction $\alpha = 0$

$$\hat{c}(4) = \hat{c}(1) + A(1,4) + \alpha$$
$$= 25 + 0 + 0 = 25$$

Consider the path (1,5) : The first row and fifth column of reduced matrix are made ∞ and element (5,1) is also made ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 12 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix} \leftarrow 9,\ \leftarrow 2,\ \leftarrow 9,\ \leftarrow 3,\ \leftarrow 9$$

Path(1,5) node 5

This matrix is not reduced. subtract 2 from 2nd row 3 from 4th row.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

Apply row reduction and column reduction $r = 5+0 = 5$

$\hat{c}(u) = \hat{c}(i) + A(1,5) + r$

$= 25 + 1 + 5 = 31$

fig : path(1,5) : node 5

Since the minimum cost is 25. so select nodes. The matrix obtained for path (1,4) is considered as reduced cost matrix. and apart of space tree generated by procedure LCBB

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

Reduced cost matrix

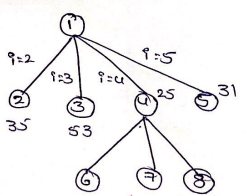


fig: Part of state space tree generated by procedure LCBB

Consider the path (1, 4, 2): The fourth row and second column of reduced matrix are made ∞ and element (2,1) also made ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 10 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix} \begin{matrix} \leftarrow 9 \\ \leftarrow 9 \\ \leftarrow 9 \\ \leftarrow 9 \\ \leftarrow 9 \end{matrix}$$

$\uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow$

Path 1,4,2 : node 6

Apply row and column reduction $r = 0$

$\hat{c}(6) = \hat{c}(4) + A(4,2) + r$

$= 25 + 3 + 0$

$= 28$

Consider the path (1, 4, 3): The fourth row and third column of reduced matrix are ∞ and element (3,1) is also made ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix} \begin{matrix} \leftarrow 9 \\ \leftarrow 9 \\ \leftarrow 2 \\ \leftarrow 9 \\ \leftarrow 9 \end{matrix}$$

$\uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow$

Path 1,4,3 : node 7.

This matrix is not reduced, substract 11 from 1st column 2 from 3rd row.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

Path 1,2,3 : node 7

Apply row and column reduction $r = 2 + 11 = 13$

$\hat{c}(7) = \hat{c}(4) + A(4,3) + r$

$= 25 + 12 + 13 = 50$

Consider the path (1,4,3) :- The fourth row and fifth column of reduced matrix and made ∞ and element 5,1 is also made ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix} \begin{matrix} \leftarrow 9 \\ \leftarrow 11 \\ \leftarrow 9 \\ \leftarrow 9 \\ \leftarrow 9 \end{matrix}$$

$\downarrow 9 \quad \downarrow 0 \quad \downarrow 0 \quad \downarrow \infty \quad \downarrow \infty$

path 1,4,3 node 8

This matrix is not reduced substract 11 from 2nd row and then this matrix is reduced.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

path 1,4,3 : node 8

Apply row reduction and column reduction.

$$r = 11+0 = 11$$

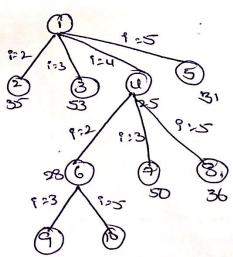
$$\hat{c}(8) = \hat{c}(4) + A(4,3) + r$$

$$= 25+0+11 = 36$$

since the minimum cost is 28, so select node 2 The matrix obtained for path (1,4,2) is considered as reduced cost matrix.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

Reduced cost matrix



Consider path (1,4,2,3) :- The 2nd row and 3rd column of reduced matrix are ∞. and element (3,1) also made ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix} \leftarrow 2$$

$\uparrow$ 11 $\qquad\qquad \uparrow_2$

This matrix is not reduced matrix. subtract 11 from 1st column and 2 from last column

path 1,4,2,3 node 9

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

Apply row and column reduction

$r = 2+11 = 13$

$\hat{c}(9) = \hat{c}(6) + A(2,3) + r$.

$= 28 + 11 + 13 = 52$

Consider path (1,4,2,5) :- The 2nd row and 5th column of reduced matrix are made ∞ and (5,1) also ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

path 1,4,2,5 node 10

Apply row and column reduction $r = 0$

$\hat{c}(10) = \hat{c}(2) + A(2,5) + r$

$= 28 + 0 + 0 = 28$

since minimum cost is 28, so select node 5. The matrix obtained for path (1,4,2,5) is considered as reduced cost matrix.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

Reduced cost matrix.

Consider the path (1,4,2,5,3):- The 5th row and 3rd column of reduced matrix are made ∞ and element (3,1) also made ∞.

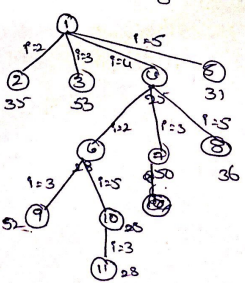
$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

path (1,4,2,5,3) node 11

Apply row reduction and column reduction r=0

$\hat{c}(11) = \hat{c}(5) + A(5,3) + r = 28+0+0 = 28$

complete state space tree generated by procedure LCBB



Path is
1→4→2→5→3→1
min cost :
10+6+2+7+3
= 28.

fig: state space tree generated by LCBB

NP-HARD AND NP-COMPLETE PROBLEMS:

Introduction:

The problems that we intend to solve, using a computer, can broadly be divided into two types P-class and NP-class problems.

The problems that can be solved in polynomial time are said to be in P-class.

Ex:- The problem of searching an element from the list. It can be solved in O(log n) time.

The problems that can be solved in the polynomial time, by non-deterministic machine, are called NP-class problems.

Ex:- knapsack problem whose time complexity is $(O(2^{n/2}))$

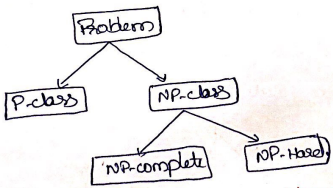


Fig:- classification of problems.

Tracable Problems:

They are problems which can be solved in a finite duration of time. The problems that can be solved in polynomial time are called tracable problems.

Polynomial time Algorithms:

An algorithm for a given problem size(n) is said to be a polynomial time algorithm if it's worst case complexity belongs to $O(n^k)$ for a fixed size small integer value k and an input size of n.

INTRACABLE PROBLEMS:

The problems that cannot be solved in the polynomial time are called intracable problems. In real time applications, there are some problems whose solution may take large amount of time or there might not be any known solution.

This might be due to the nature of the input, complexity of the algorithm used for solving etc. They are called intracable problems.

Ex:- Algorithms whose worst case complexity belongs to $O(2^n)$, $O(n^n)$ etc.

Problems can be divided into

* Decision problems
* optimization problems.

Decision problem is a problem for which the output is binary i.e., true/ false /no.

Optimization problem is a problem that finds an optimal value of a cost function.

NP-PROBLEMS:-

They are also called Deterministic polynomial Problems. NP problems can be solved by non-deterministic Algorithms.

NP class is a set of all decision problems solvable by non-deterministic algorithm in polynomial time.

Ex:- consider the problem of searching a particular element in a binary tree. Let the depth of binary tree be 'd'. suppose depth is 0, we have only one element in the tree

If the depth is 1, we have 3 elements. In general no of elements in the tree having depth 'd' is given by $(2^{d+1})-1$

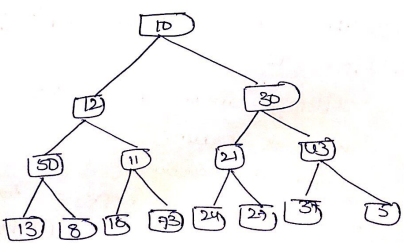


Fig:- Binary search problem representation.

A deterministic algorithm for this problem would be as follows

Search from the root node (10) and keep traversing in preorder comparing each element. Because search is sequential the worst case is $O(2^{h+1})$ comparisons.

A non deterministic algorithm for searching problem could be as follows:

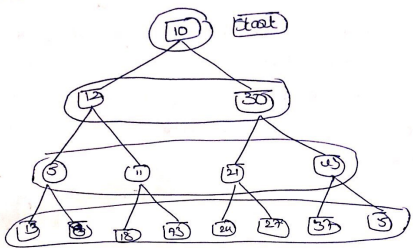


Fig: Binary search for a non deterministic algorithm.

For a non deterministic algorithm which can compare the search element with all the elements at every level in one unit of time, the search will have a worst case complexity of $O(d) \geq O(n)$: which is polynomial time.

The key point to note here is that search is not conducted in a serial fashion. Machines of such capability are non deterministic machines.

A non-deterministic algorithm is an algorithm which can be executed on a non-deterministic machine.

NP HARD AND NP-COMPLETE CLASSES

NP-complete problem:

There are large number of problems in real time applications belonging to NP-class.

It is possible to identify a small set of NP problems such that if any one problem in this set has a polynomial time solution on a deterministic machine, then the rest of all NP-problems will also have a polynomial time solution on a deterministic machine

This subset of NP problems having the above mentioned special property is called NP-complete problems.

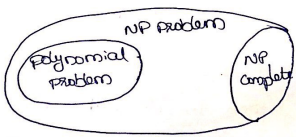
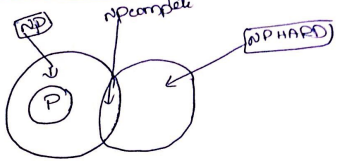


fig: NP complete problem representation

NP-HARD PROBLEMS:

A problem L is hard if and only if the satisfiability is reduced to L. If an NP-HARD problem can be solved in polynomial time then all NP-complete problems are solved in polynomial time.

A problem L is NP-complete if and only if L is NP-HARD and $L \in NP$



Satisfiability Problem:

We formulate satisfiability problem to determine whether a formulae is true for same assignment of truth values to the variables.

COOK's Theorem:

COOK's theorem states that satisfiability is in P if P=NP. We already know that satisfiability is in NP. Hence if P=NP, then satisfiability is in P. Then it remains to be shown that if satisfiability is in P then P=NP.

In order to prove this statement, we show how to obtain any polynomial time non deterministic decision algorithm A and input I, a formulae Q(A,I) such that Q is satisfiability if A has successful termination with input I.

If the length of I is 'n' and the time complexity of A is P(n) for some polynomial P() then the length of Q will be $O(P^3(n) \log n) = O(P^4(n))$. The time needed to construct Q will also be same.