LECTURE NOTES ON

DESIGN & ANALYSIS OF ALGORITHMS (15A05604)

III B.TECH II SEMESTER

(JNTUA-R15)

NARAYANA
ENGINEERING COLLEGE
(AUTONOMOUS)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NARAYANA ENGINEERING COLLEGE :: GUDUR
(AUTONOMOUS)
Approved by AICTE and Permanently Affiliated to JNTUA, Ananthapuramu, An ISO 9001:2015 Certified Institution, Recognized by UGC U/S 2(f) & 12(B)
Dhoorjati Nagar, Gudur, SPSR Nellore Dist.- 524101

NARAYANA ENGINEERING COLLEGE :: GUDUR

Prepared By Mr.P.Muthyalu

<u>UNIT – I</u>

1. WHAT IS AN ALGORITHM:

Informal Definition:

An Algorithm is any well-defined computational procedure that takes some value or set of values as Input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the i/p into the o/p.

Formal Definition:

An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms should satisfy the following criteria.

INPUT Zero or more quantities are externally supplied.
OUTPUT At least one quantity is produced.
DEFINITENESS Each instruction is clear and unambiguous.
FINITENESS If we trace out the instructions of an algorithm, then for all cases,the algorithm terminates after a finite number of steps.
EFFECTIVENESS Every instruction must very basic so that it can be carriedout, in principle, by a person using only pencil & paper.

Issues or study of Algorithm:

1. How to device or design an algorithm creating and algorithm.
2. How to express an algorithm definiteness.
3. How to analysis an algorithm time and space complexity.
4. How to validate an algorithm fitness.
5. Testing the algorithm checking for error.

ALGORITHM SPECIFICATION:

Algorithm can be described in three ways.

1. Natural language like English:
When this way is chooses care should be taken, we should ensure thateach & every statement is definite.

2. Graphic representation called flowchart:
This method will work well when the algorithm is small& simple.

1. Pseudo-code Method:
In this method, we should typically describe algorithms as program, which resembles language like Pascal & algol.

Pseudo-Code Conventions:

1. Comments begin with // and continue until the end of line.

2. Blocks are indicated with matching braces {and}.

3. An identifier begins with a letter. The data types of variables are not explicitly declared.

4. Compound data types can be formed with records. Here is an example,

```
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 itemsof a record can be accessed with and period.

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

```
<Variable>:= <expression>;
```

6. There are two Boolean values TRUE and 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 step do

{
  <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 & write.

10. There is only one type of procedure:
Algorithm, the heading takes the form,

Algorithm Name (Parameter lists)

As an example, the following algorithm fields & returns the maximum of‘n’ given numbers:

```
3.  algorithm Max(A,n)
4.  // A is an array of size n
5. {
6.  Result := A[1];
7.  for I:= 2 to n do
8.      if A[I] > Result then
9.            Result :=A[I];
10.   return Result;
11.}
```

In this algorithm (named Max), A & n are procedure parameters.
Result & I are Local variables.

Next we present 2 examples to illustrate the process of translation problem into an algorithm.

Selection Sort:

1. Suppose we Must devise an algorithm that sorts a collection of n>=1 elements of arbitrary type.
2. A Simple solution given by the following.
3. (From those elements that are currently unsorted ,find the smallest &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 isat
a[j];
4.                   Interchange a[I] and a[j];5.
}
```

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

1. We can solve the latter problem using the code,

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

2. 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 the current minimum.

3. Putting all these observations together, we get the algorithm Selectionsort.

Theorem:

Algorithm selection sort(a,n) correctly sorts a set of n>=1 elements .Theresult remains is a a[1:n] such that a[1] <= a[2] ….<=a[n].

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.

Example:

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

Proof:

We first note that any I, say I=q, following the execution of lines 6 to9,it is the case that a[q] Þ a[r],q<r<=n.
Also observe that when 'i' becomes greater than q, a[1:q] is unchanged.
Hence, following the last execution of these lines (i.e.I=n).We have a[1] <= a[2] <=……a[n].
We observe this point that the upper limit of the for loop in the line 4can be changed to n-1 without damaging the correctness of the algorithm.

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[I];
10.                     a[I]:=a[j];
11.                     a[j]:=t;
12.     }
13.  } 1.2.2.Recursive
```

Algorithms:

A Recursive function is a function that is defined in terms of itself.

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

An algorithm that calls itself is Direct Recursive.

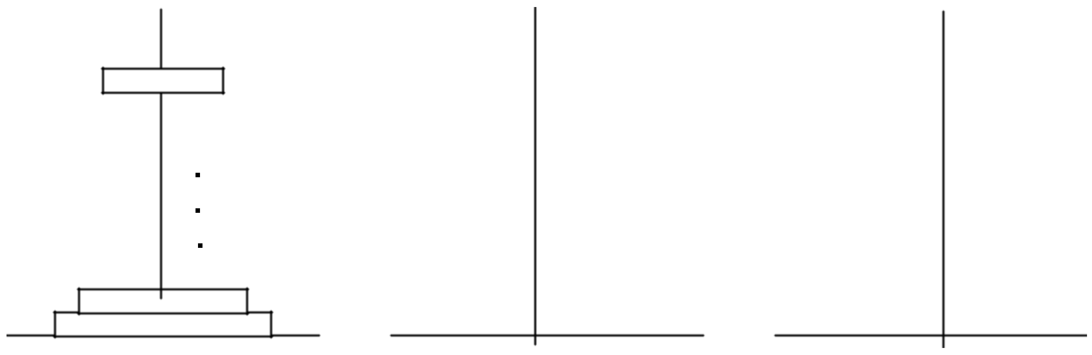
Algorithm 'A' is said to be Indirect Recursive if it calls another algorithm which in turns calls 'A'.

The Recursive mechanism, are externally powerful, but even more importantly, many times they can express an otherwise complex process very clearly. Or these reasons we introduce recursion here.

The following 2 examples show how to develop a recursive algorithms.

In the first, we consider the Towers of Hanoi problem, and in the second, we generate all possible permutations of a list of characters.

Towers of Hanoi:



Tower A Tower B
Tower C

It is Fashioned after the ancient tower of Brahma ritual.
According to legend, at the time the world was created, there was a diamond tower (labeled A) with 64 golden disks.
The disks were of decreasing size and were stacked on the tower in decreasing order of size bottom to top.
Besides these tower there were two other diamond towers(labeled B& C)
Since the time of creation, Brehman priests have been attempting to move the disks from tower A to tower B using tower C,for intermediate storage.

- o As the disks are very heavy, they can be moved only one at a time.
- o 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 priest have completed this task.

A very elegant solution results from the use of recursion.

- o Assume that the number of disks is 'n'.
- o To get the largest disk to the bottom of tower B, we move the remaining 'n-1' disks to tower C and then move the largest to tower B.
- o Now we are left with the tasks of moving the disks from tower C to B.
- o To do this, we have tower A and B available.
- o The fact, that towers B has a disk on it can be ignored as the disks largerthan the disks being moved from tower C and so any disk scan be placed on top of it.

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);
      Towersofhanoi(n-1,z,y,x);
           }
           }
```

2. Permutation Generator:

Given a set of n>=1elements, the problem is to print all possiblepermutations of this set.

For example, if the set is {a,b,c} ,then the set of permutation is,

{ (a,b,c),(a,c,b),(b,a,c),(b,c,a),(c,a,b),(c,b,a)}

1. It is easy to see that given ‘n’ elements there are n! different permutations.

8. A simple algorithm can be obtained by looking at the case of 4 statement(a,b,c,d)

9. The Answer can be constructed by writing

- a followed by all the permutations of (b,c,d)
- b followed by all the permutations of(a,c,d)
- c followed by all the permutations of (a,b,d)
- d followed by all the permutations of (a,b,c)

Algorithm:

```
Algorithm perm(a,k,n)
{
if(k=n) then write (a[1:n]); // output permutation else
//a[k:n] ahs more than one permutation
        // Generate this recursively.for
I:=k to n do
{
t:=a[k];
a[k]:=a[I];
a[I]:=t;
perm(a,k+1,n);
//all permutation of a[k+1:n]
t:=a[k];
a[k]:=a[I];
a[I]:=t;
}
}
```

PERFORMANCE ANALYSIS:

Space Complexity:

The space complexity of an algorithm is the amount of money itneeds to run to compilation.

Time Complexity:

The time complexity of an algorithm is the amount of computertime it needs to run to compilation.

Space Complexity:

Space Complexity Example:

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

The Space needed by each of these algorithms is seen to be the sum of the following component.

1.A fixed part that is independent of the characteristics (eg:number,size)of the inputs and outputs.

The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.

A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that is depends oninstance characteristics), and the recursion stack space.

- The space requirement s(p) of any algorithm p may therefore be written as,

  S(P) = c+ Sp(Instance characteristics)

Where ‘c’ is a constant.

Example 2:

```
Algorithm sum(a,n)
{
    s=0.0;
    for I=1 to n do
    s= s+a[I];
    return s;
}
```

1. The problem instances for this algorithm are characterized by n,the number of elements to be summed. The space needed d by ‘n’ is one word, since it is of type integer.
2. The space needed by ‘a’a is the space needed by variables of tyepe array of floating point numbers.
3. This is atleast ‘n’ words, since ‘a’ must be large enough to hold the ‘n’ elements to be summed.
4. So,we obtain Ssum(n)>=(n+s)

   [n for a[],one each for n,I a& s]

1.3.2.Time Complexity:

The time T(p) taken by a program P is the sum of the compile time and therun time(execution time)

The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation .This rum time is denoted by tp(instance characteristics).

The number of steps any problem statemn t is assigned depends on the kind of statement.

For example, comments	0 steps.
Assignment statements	1 steps.

[Which does not involve any calls to other algorithms]

Interactive statement such as for, while & repeat-until Control part of the statement.

We introduce a variable, count into the program statement to increment count with initial value 0.Statement to increment count by the appropriate amount are introduced into the program.

This is done so that each time a statement in the original program is executes count is incremented by the step count of that statement.

Algorithm: Algorithm

```
sum(a,n)
{
     s= 0.0;
     count = count+1;
     for I=1 to n do
     {
      count =count+1;
     s=s+a[I];
     count=count+1;
     }
     count=count+1;
     count=count+1;
     return s;
     }
```

If the count is zero to start with, then it will be 2n+3 on termination.

2. The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.

First determine the number of steps per execution (s/e) of the statementand the total number of times (ie., frequency) each statement is executed.

By combining these two quantities, the total contribution of allstatements, the step count for the entire algorithm is obtained.

Statement		S/e	Frequency	Total
1. Algorithm Sum(a,n)		0	-	0
2.{		0	-	0
3.	S=0.0;	1	1	1
4.	for I=1 to n do	1	n+1	n+1
5.	s=s+a[I];	1	n	n
6.	return s;	1	1	1
7.	}	0	-	0
Total				2n+3

Average –Case Analysis

Most of the time, average-case analysis are performed under the more orless realistic assumption that all instances of any given size are equally likely.

For sorting problems, it is simple to assume also that all the elements to besorted are distinct.

Suppose we have ‘n’ distinct elements to sort by insertion and all n! permutation of these elements are equally likely.

To determine the time taken on a average by the algorithm ,we could add the times required to sort each of the possible permutations ,and then divideby n! the answer thus obtained.

An alternative approach, easier in this case is to analyze directly the time required by the algorithm, reasoning probabilistically as we proceed.

For any I,2 I n, consider the sub array, T[1….i].

The partial rank of T[I] is defined as the position it would occupy if the subarray were sorted.

For Example, the partial rank of T[4] in [3,6,2,5,1,7,4] in 3 because T[1….4]once sorted is [2,3,5,6].

Clearly the partial rank of T[I] does not depend on the order of the elementin Sub array T[1…I-1].

Analysis

Best case:

This analysis constrains on the input, other than size. Resulting in the fasters possible run time

Worst case:

This analysis constrains on the input, other than size. Resulting in the fasters possible run time

Average case:

This type of analysis results in average running time over every type of input.

Complexity:

Complexity refers to the rate at which the storage time grows as a function of the problem size

Asymptotic analysis:

Expressing the complexity in term of its relationship to know function. This type analysis is called asymptotic analysis.

ASYMPTOTIC NOTATION:

Big ‘oh’: the function f(n)=O(g(n)) iff there exist positive constants c and no suchthat f(n)≤c*g(n) for all n, n ≥ no.

Omega: the function f(n)=Ω(g(n)) iff there exist positive constants c and no suchthat f(n) ≥ c*g(n) for all n, n ≥ no.

Theta: the function f(n)=ɵ(g(n)) iff there exist positive constants c1,c2 and no suchthat c1 g(n) ≤ f(n) ≤ c2 g(n) for all n, n ≥ no.

Recursion:

Recursion may have the following definitions:
-The nested repetition of identical algorithm is recursion.
-It is a technique of defining an object/process by itself.
-Recursion is a process by which a function calls itself repeatedly until some specified condition has been satisfied.

When to use recursion:

Recursion can be used for repetitive computations in which each action is stated in terms of previous result. There are two conditions that must be satisfied by any recursive procedure.

1 .Each time a function calls itself it should get nearer to the solution.
2 .There must be a decision criterion for stopping the process.

In making the decision about whether to write an algorithm in recursive or non- recursive form, it is always advisable to consider a tree structure for the problem. If the structure is simple then use non- recursive form. If the tree appears quite bushy, with little duplication of tasks, then recursion is suitable.

The recursion algorithm for finding the factorial of a number is given below,

```
Algorithm : factorial-recursion
Input : n, the number whose factorial is to be found.Output : f,
the factorial of n
Method : if(n=0)
f=1
else
f=factorial(n-1) * nif
end
algorithm ends.
```

The general procedure for any recursive algorithm is as follows,

1. Save the parameters, local variables and return addresses.

2. If the termination criterion is reached perform final computation andgoto step 3 otherwise perform final computations and goto step 1

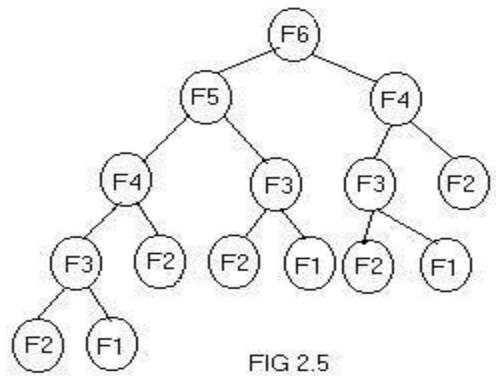


FIG 2.5

3. Restore the most recently saved parameters, local variable and return address and goto the latest return address.

Iteration v/s Recursion:

Demerits of recursive algorithms:

1. Many programming languages do not support recursion; hence, recursive mathematical function is implemented using iterative methods.

2. Even though mathematical functions can be easily implemented using recursion it is always at the cost of execution time and memory space. For example, the recursion tree for generating 6 numbers in a Fibonacci series generation is given in fig 2.5. A Fibonacci series is of the form 0,1,1,2,3,5,8,13,…etc, where the third number is the sum of preceding two numbers and so on. It can be noticed from the fig 2.5 that, f(n-2) is computed twice, f(n-3) is computed thrice, f(n-4) is computed 5 times.

3. A recursive procedure can be called from within or outside itself and to ensure its proper functioning it has to save in some order the return addresses so that, a return to the proper location will result when the return to a calling statement is made.
4. The recursive programs needs considerably more storage and will takemore time.

Demerits of iterative methods :

1. Mathematical functions such as factorial and Fibonacci series generationcan be easily implemented using recursion than iteration.
2. In iterative techniques looping of statement is very much necessary. Recursion

is a top down approach to problem solving. It divides the problem intopieces or selects out one key step, postponing the rest.
Iteration is more of a bottom up approach. It begins with what is known and fromthis constructs the solution step by step. The iterative function

obviously uses time that is O(n) where as recursive function has an exponentialtime complexity.
It is always true that recursion can be replaced by iteration and stacks. It is alsotrue that stack can be replaced by a recursive program with no stack.

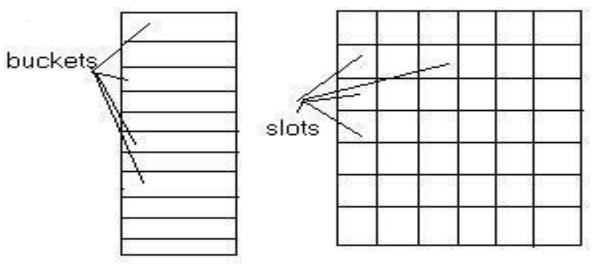


Fig 2.6

SOLVING RECURRENCES :-(Happen again (or) repeatedly)

1. The indispensable last step when analyzing an algorithm is often to solve a recurrence equation.
2. With a little experience and intention, most recurrence can be solved by intelligent guesswork.
3. However, there exists a powerful technique that can be used to solve certain classes of recurrence almost automatically.
4. This is a main topic of this section the technique of the characteristic equation.

Intelligent guess work:

This approach generally proceeds in 4 stages.

Calculate the first few values of the recurrence
Look for regularity.
Guess a suitable general form.
And finally prove by mathematical induction(perhaps constructive induction).

Then this form is correct.
Consider the following recurrence,

$$T(n) = \begin{cases} 0 & \text{if } n=0 \\ 3T(n \div 2)+n & \text{otherwise} \end{cases}$$

5. First step is to replace n ÷ 2 by n/2

6. It is tempting to restrict 'n' to being ever since in that case n÷2 = n/2, but recursively dividing an even no. by 2, may produce an odd no. larger than1.

7. Therefore, it is a better idea to restrict 'n' to being an exact power of 2.

8. First, we tabulate the value of the recurrence on the first few powers of 2.

n	1	2	4	8	16	32
T(n)	1	5	19	65	211	665

* For instance, T(16) = 3 * T(8) +16
= 3*65+16
= 211.

* Instead of writing T(2) = 5, it is more useful to write T(2)=3*1+2.

Then,

T(A) = 3 * T(2) +4
= 3*(3*1+2)+4
= $(3^2*1)+(3*2)+4$

* We continue in this way, writing 'n' as an explicit power of 2.

n	T(n)
o	1
o	3*1+2
2^2	$3^2*1+3*2+2^2$
2^3	$3^3*1+3^2*2+3*2^2+2^3$
2^4	$3^4*1+3^3*2+3^2*2^2+3*2^3+2^4$
2^5	$3^5*1+3^4*2+3^3*2^2+3^2*2^3+3*2^4+2^5$

9. The pattern is now obvious.

$$T(2^k) = 3^k2^0 + 3^{k-1}2^1 + 3^{k-2}2^2+\ldots+3^12^{k-1} + 3^02^k.$$

$$= \sum 3^{k-i}\, 2^i$$

$$= 3^k \sum (2/3)^i$$

$$= 3^k * [(1-(2/3)^{k+1}) / (1-(2/3)]$$

$$= 3^{k+1} - 2^{k+1}$$

Proposition: (Geometric Series)

Let S_n be the sum of the first n terms of the geometric series a, ar, ar^2….Then

$S_n = a(1-r^n)/(1-r)$, except in the special case when r = 1; when $S_n = a_n$.

$= 3^k * \quad [(1 – (2/3)^{k+1}) / (1 – (2/3))]$

$= 3^k * [((3^{k+1} – 2^{k+1})/ 3^{k+1}) / ((3 – 2) / 3)]$

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

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

$= 3^{k+1} – 2^{k+1}$

* It is easy to check this formula against our earlier tabulation

EG:2

$$t_n = \begin{cases} 0 & n=0 \\ 5 & n=1 \\ 3t_{n-1} + 4t_{n-2}, & \text{otherwise} \end{cases}$$

$t_n = 3t_{n-1} – 4t_{n-2} = 0$General function

Characteristics Polynomial, $x^2 – 3x – 4 = 0$

$(x – 4)(x + 1) = 0$

Roots $r_1 = 4$, $r_2 = -1$

General Solution, $f_n = C_1r_1^n + C_2 r_2^n$ (A)

n=0 $C_1 + C_2 = 0$ (1)

n=1 $C_1r_1 + C_2r_2 = 5$ (2)

Eqn 1 $C_1 = -C_2$

sub C_1 value in Eqn (2)

$-C_2r_1 + C_2r_2 = 5$

$C_2(r_2 – r_1) = 5$

$$C_2 = \frac{5}{r_2 - r_1}$$

$$= \frac{5}{-1+4}$$

$= 5/(-5) = -1$

$C_2=-1$, $C_1=1$

Sub C_1, C_2, r_1 & r_2 value in equation (A)

$f_n = 1.\ 4^n + (-1)\ .\ (-1)^n$
$f_n = 4^n + 1^n$

DIVIDE AND CONQUER

GENERAL METHOD:

Given a function to compute on 'n' inputs the divide-and-conquer strategy suggests splitting the inputs into 'k' distinct subsets, 1<k<=n, yielding 'k' sub problems.

These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.

If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.

Often the sub problems resulting from a divide-and-conquer design are ofthe same type as the original problem.

For those cases the re application of the divide-and-conquer principle is naturally expressed by a recursive algorithm.

D And C(Algorithm) is initially invoked as D and C(P), where 'p' is the problem to be solved.

Small(P) is a Boolean-valued function that determines whether the i/p sizeis small enough that the answer can be computed without splitting.

If this so, the function 'S' is invoked.

Otherwise, the problem P is divided into smaller sub problems.

These sub problems P1, P2 …Pk are solved by recursive application of D And C.

Combine is a function that determines the solution to p using the solutionsto the 'k' sub problems.

If the size of ‘p’ is n and the sizes of the ‘k’ sub problems are n1, n2 ….nk, respectively, then the computing time of D And C is described by the recurrence relation.

T(n)= { g(n)   n small
T(n1)+T(n2)+… .... +T(nk)+f(n);   otherwise.

Where T(n) is the time for D And C on any I/p of size ‘n’.
g(n) is the time of compute the answer directly for small I/ps.
f(n) is the time for dividing P & combining the solution tosub problems.

```
1)  Algorithm D And C(P)
2)  {
3)  if small(P) then return S(P);
4)  else
5)  {
6)  divide P into smaller instances
        P1, P2… Pk, k>=1;
7)   Apply D And C to each of these sub problems;
8)  return combine (D And C(P1), D And C(P2),…….,D And C(Pk));9)
    }
10) }
```

The complexity of many divide-and-conquer algorithms is given by recurrences of the form

T(n) = { T(1)   n=1
AT(n/b)+f(n)   n>1

Where a & b are known constants.
We assume that T(1) is known & ‘n’ is a power of b(i.e., n=b^k)

One of the methods for solving any such recurrence relation is called the substitution method.

This method repeatedly makes substitution for each occurrence of the function. T is the Right-hand side until all such occurrences disappear.

Example:

11) Consider the case in which a=2 and b=2. Let T(1)=2 & f(n)=n. We have,

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

*
*
*

In general, we see that T(n)=2^iT(n/2^i)+in., for any log n >=I>=1. T(n)

=2^log n T(n/2^log n) + n log n

Corresponding to the choice of i=logn

Thus, T(n) = 2^log n T(n/2^log n) + n log n

= n. T(n/n) + n log n
= n. T(1) + n log n [since, log 1=0, 2^0=1] = 2n + n log n

BINARY SEARCH:

```
1.    Algorithm Bin search(a,n,x)
2.    // Given an array a[1:n] of elements in non-decreasing
3.    //order, n>=0,determine whether 'x' is present and
4.    // if so, return 'j' such that x=a[j]; else return 0.5. {
6.    low:=1; high:=n;
7.    while (low<=high) do
8.    {
9.        mid:=[(low+high)/2];
10.       if (x<a[mid]) then high;
11.       else if(x>a[mid]) then
              low=mid+1;
12.     else return mid;
13.    }
14.    return 0;
15.   }
```

Algorithm, describes this binary search method, where Binsrch has 4I/ps a[], I , l &x.
It is initially invoked as Binsrch (a,1,n,x)
A non-recursive version of Binsrch is given below.This
Binsearch has 3 i/ps a,n, & x.
The while loop continues processing as long as there are more elements left tocheck.
At the conclusion of the procedure 0 is returned if x is not present, or 'j' is returned, such that a[j]=x.
We observe that low & high are integer Variables such that each time through the loop either x is found or low is increased by at least one or high is decreased at least one.

Thus we have 2 sequences of integers approaching each other and eventually low becomes > than high & causes termination in a finite no. of steps if 'x' is not present.

Example: Let us select the 14 entries.

-15,-6,0,7,9,23,54,82,101,112,125,131,142,151.

Place them in a[1:14], and simulate the steps Binary search goes through as it searches for different values of 'x'.

Only the variables, low, high & mid need to be traced as we simulate thealgorithm. We try the following values for x: 151, -14 and 9.for

2 successful searches &

1 unsuccessful search.

Table. Shows the traces of Bin search on these 3 steps.

X=151	low	high	mid
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			Found

x=-14	low	high	mid
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	Not found

x=9	low	high	mid
	1	14	7
	1	6	3
	4	6	5
			Found

Theorem: Algorithm Binsearch(a,n,x) works correctly.

Proof:
We assume that all statements work as expected and that comparisons such as x>a[mid] are appropriately carried out.

Initially low =1, high= n,n>=0, and a[1]<=a[2]<= .... <=a[n].
If n=0, the while loop is not entered and is returned.

Otherwise we observe that each time thro' the loop the possible elementsto be checked of or equality with x and a[low], a[low+1],……..,a[mid],……a[high].

If x=a[mid], then the algorithm terminates successfully.

Otherwise, the range is narrowed by either increasing low to (mid+1) or decreasing high to (mid-1).

Clearly, this narrowing of the range does not affect the outcome of thesearch.

If low becomes > than high, then ‘x’ is not present & hence the loop is exited.

FINDING THE MAXIMUM AND MINIMUM:

Let us consider another simple problem that can be solved by the divide-and-conquer technique.

The problem is to find the maximum and minimum items in a set of ‘n’elements.

In analyzing the time complexity of this algorithm, we once again concentrate on the no. of element comparisons.

More importantly, when the elements in a[1:n] are polynomials, vectors, very large numbers, or strings of character, the cost of an element comparison ismuch higher than the cost of the other operations.

Hence, the time is determined mainly by the total cost of the elementcomparison.

```
Algorithm straight MaxMin(a,n,max,min)
// set max to the maximum & min to the minimum of a[1:n]
{
max:=min:=a[1];
for I:=2 to n do
{
if(a[I]>max) then max:=a[I];
if(a[I]<min) then min:=a[I];
}
}
```

Algorithm: Straight forward Maximum & Minimum

Straight MaxMin requires 2(n-1) element comparison in the best, average & worst cases.

An immediate improvement is possible by realizing that the comparison a[I]<min is necessary only when a[I]>max is false.

Hence we can replace the contents of the for loop by,

```
    If(a[I]>max) then max:=a[I];
    Else if (a[I]<min) then min:=a[I];
```

Now the best case occurs when the elements are in increasing order.
The no. of element comparison is (n-1).

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

The average no. of element comparison is < than 2(n-1)

On the average a[I] is > than max half the time, and so, the avg. no. of comparison is3n/2-1.

A divide- and conquer algorithm for this problem would proceed as follows:Let

P=(n, a[I] ,……,a[j]) denote an arbitrary instance of the problem.
Here 'n' is the no. of elements in the list (a[I],….,a[j]) and we are interested in finding the maximum and minimum of the list.

If the list has more than 2 elements, P has to be divided into smaller instances. For

example , we might divide 'P' into the 2 instances, P1=([n/2],a[1],........................a[n/2]) & P2= (n-[n/2],a[[n/2]+1],....... ,a[n])

After having divided 'P' into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm.

Algorithm: Recursively Finding the Maximum & Minimum

```
Algorithm MaxMin (I,j,max,min)
//a[1:n] is a global array, parameters I & j
//are integers, 1<=I<=j<=n.The effect is to
//set max & min to the largest & smallest value
//in a[I:j], respectively.
{
    if(I=j) then max:= min:= a[I];
    else if (I=j-1) then // Another case of small(p)
    {
        if (a[I]<a[j]) then
        {
            max:=a[j];
            min:=a[I];
        }
        else
        {
            max:=a[I];
            min:=a[j];
        }
    }
else
{
// if P is not small, divide P into subproblems.
// find where to split the set mid:=[(I+j)/2];
//solve the subproblems
MaxMin(I,mid,max.min);
MaxMin(mid+1,j,max1,min1);
//combine the solution
if (max<max1) then max=max1;
if(min>min1) then min = min1;
}
}
```

The procedure is initially invoked by the statement,

MaxMin(1,n,x,y)

Suppose we simulate MaxMin on the following 9 elements

A:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	-5	-8	15	60	17	31	47

A good way of keeping track of recursive calls is to build a tree by adding a nodeeach time a new call is made.

For this Algorithm, each node has 4 items of information: I, j, max & imin.

Examining fig: we see that the root node contains 1 & 9 as the values of I &j corresponding to the initial call to MaxMin.

This execution produces 2 new calls to MaxMin, where I & j have the values 1, 5 & 6, 9 respectively & thus split the set into 2 subsets of approximately the same size.

From the tree, we can immediately see the maximum depth of recursion is 4. (including the 1st call)

The include no.s in the upper left corner of each node represent the order in whichmax & min are assigned values.

No. of element Comparison:

If T(n) represents this no., then the resulting recurrence relations is

T(n)={ T([n/2]+T[n/2]+2 n>2

n=2

n=1

When ‘n’ is a power of 2, n=2^k for some +ve integer ‘k’, then T(n)

= 2T(n/2) +2

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

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

*

*

= 2^k-1T(2)+

= 2^k-1+2^k-2

= 2^k/2+2^k-2

= n/2+n-2
= (n+2n)/2)-2
T(N)=(3N/2)-2

*Note that (3n/3)-3 is the best-average, and worst-case no. of comparisons when‘n’ is a power of 2.

MERGE SORT

As another example divide-and-conquer, we investigate a sorting algorithmthat has the nice property that is the worst case its complexity is O(n log n)

This algorithm is called merge sort

We assume throughout that the elements are to be sorted in non-decreasing order.

Given a sequence of ‘n’ elements a[1],…,a[n] the general idea is to imaginethen split into 2 sets a[1],…..,a[n/2] and a[[n/2]+1],….a[n].

Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of ‘n’ elements.

Thus, we have another ideal example of the divide-and-conquer strategy in which the splitting is into 2 equal-sized sets & the combining operation is the merging of 2 sorted sets into one.

Algorithm For Merge Sort:

```
1.    Algorithm MergeSort(low,high)
2.    //a[low:high] is a global array to be sorted
3.    //Small(P) is true if there is only one element
4.    //to sort. In this case the list is already sorted.5. {
6.    if (low<high) then //if there are more than one element7.
      {
8.    //Divide P into subproblems
9.    //find where to split the set
10.   mid = [(low+high)/2];
11.   //solve the subproblems.
12.   mergesort (low,mid);
13.   mergesort(mid+1,high);
14.   //combine the solutions .
15.   merge(low,mid,high);
16.   }
17.   }
```

Algorithm: Merging 2 sorted subarrays using auxiliary storage.

```
1.    Algorithm merge(low,mid,high)
2.    //a[low:high] is a global array containing
3.    //two sorted subsets in a[low:mid]
```

```
//and in a[mid+1:high].The goal is to merge these 2 sets into
//a single set residing in a[low:high].b[] is an auxiliary global array.6.    {
h=low; I=low; j=mid+1;
while ((h<=mid) and (j<=high)) do9.
{
if (a[h]<=a[j]) then
{
    b[I]=a[h];
    h = h+1;
}
else
{
    b[I]= a[j];
     j=j+1;
}
I=I+1;
}
if (h>mid) then
   for k=j to high do
    {
         b[I]=a[k];
         I=I+1;
     }
 else
     for k=h to mid do
     {
          b[I]=a[k];
          I=I+1;
      }
     for k=low to high do a[k] = b[k];
}
```

Consider the array of 10 elements a[1:10] =(310, 285, 179, 652, 351, 423, 861, 254, 450, 520)

Algorithm Mergesort begins by splitting a[] into 2 sub arrays each of sizefive (a[1:5] and a[6:10]).

The elements in a[1:5] are then split into 2 sub arrays of size 3 (a[1:3]) and 2(a[4:5])

Then the items in a a[1:3] are split into sub arrays of size 2 a[1:2] & one(a[3:3])

The 2 values in a[1:2} are split to find time into one-element sub arrays, andnow the merging begins.

(310| 285| 179| 652, 351| 423, 861, 254, 450, 520)

Where vertical bars indicate the boundaries of sub arrays.

Elements a[I] and a[2] are merged to yield,
(285, 310|179|652, 351| 423, 861, 254, 450, 520)

Then a[3] is merged with a[1:2] and
(179, 285, 310| 652, 351| 423, 861, 254, 450, 520)

Next, elements a[4] & a[5] are merged.
(179, 285, 310| 351, 652 | 423, 861, 254, 450, 520)

And then a[1:3] & a[4:5]
(179, 285, 310, 351, 652| 423, 861, 254, 450, 520)

Repeated recursive calls are invoked producing the following sub arrays.(179, 285, 310, 351, 652| 423| 861| 254| 450, 520)

Elements a[6] &a[7] are merged.

Then a[8] is merged with a[6:7]
(179, 285, 310, 351, 652| 254,423, 861| 450, 520)

Next a[9] &a[10] are merged, and then a[6:8] & a[9:10] (179, 285, 310, 351, 652| 254, 423, 450, 520, 861)

At this point there are 2 sorted sub arrays & the final merge producesthe fully sorted result.
(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

IF THE TIME FOR THE MERGING OPERATIONS IS PROPORTIONAL TO 'N', THEN THE COMPUTING TIME FOR MERGE SORT IS DESCRIBED BY THE RECURRENCE RELATION.

T(N)={A N=1,'A' A CONSTANT

2T(N/2)+CN N>1,'C' A CONSTANT.

When 'n' is a power of 2, n= 2^k, we can solve this equation by successive substitution.

T(n) =2(2T(n/4) +cn/2) +cn
= 4T(n/4)+2cn
= 4(2T(n/8)+cn/4)+2cn
*
*
= 2^k T(1)+kCn.
= an + cn log n.

It is easy to see that if s^k<n<=2^k+1, then T(n)<=T(2^k+1). Therefore,

T(n)=O(n log n)

QUICK SORT

The divide-and-conquer approach can be used to arrive at an efficientsorting method different from merge sort.

In merge sort, the file a[1:n] was divided at its midpoint into sub arrayswhich were independently sorted & later merged.

In Quick sort, the division into 2 sub arrays is made so that the sorted subarrays do not need to be merged later.

This is accomplished by rearranging the elements in a[1:n] such that a[I]<=a[j] for all I between 1 & n and all j between (m+1) & n for some m, 1<=m<=n.

Thus the elements in a[1:m] & a[m+1:n] can be independently sorted.No

merge is needed. This rearranging is referred to as partitioning.

Function partition of Algorithm accomplishes an in-place partitioning of the elements of a[m:p-1]

It is assumed that a[p]>=a[m] and that a[m] is the partitioning element. If m=1 & p-1=n, then a[n+1] must be defined and must be greater than or equal to allelements in a[1:n]

The assumption that a[m] is the partition element is merely for convenience, other choices for the partitioning element than the first item in theset are better in practice.

The function interchange (a,I,j) exchanges a[I] with a[j].

Algorithm: Partition the array a[m:p-1] about a[m]

```
1.    Algorithm Partition(a,m,p)
2.    //within a[m],a[m+1],…..,a[p-1] the elements
3.    // are rearranged in such a manner that if
4.    //initially t=a[m],then after completion
5.    //a[q]=t for some q between m and
6.    //p-1,a[k]<=t for m<=k<q, and
7.    //a[k]>=t for q<k<p. q is returned
8.    //Set a[p]=infinite.
```

```
{
v=a[m];I=m;j=p;
repeat
{
repeat
    I=I+1;
 until(a[I]>=v);
repeat
j=j-1;
until(a[j]<=v);
if (I<j) then interchange(a,i.j);
}until(I>=j);
a[m]=a[j]; a[j]=v;
retun j;
}
```

```
Algorithm Interchange(a,I,j)
//Exchange a[I] with a[j]
{
p=a[I];
a[I]=a[j];
a[j]=p;
}
```

Algorithm: Sorting by Partitioning

```
Algorithm Quicksort(p,q)
//Sort the elements a[p],….a[q] which resides
//is the global array a[1:n] into ascending
//order; a[n+1] is considered to be defined
// and must be >= all the elements in a[1:n]6.
{
if(p<q) then // If there are more than one element8.
{
// divide p into 2 subproblems
j=partition(a,p,q+1);
//'j' is the position of the partitioning element.
//solve the subproblems.
quicksort(p,j-1);
quicksort(j+1,q);
//There is no need for combining solution.
}
}
```

Record Program: Quick Sort

```
#include <stdio.h>
#include <conio.h>
```

```
int a[20];
main()
{
      int n,I;
      clrscr();
      printf("QUICK SORT");
      printf("\n Enter the no. of elements ");
      scanf("%d",&n);
       printf("\nEnter the array elements");
        for(I=0;I<n;I++)
            scanf("%d",&a[I]);
        quicksort(0,n-1);
       printf("\nThe array elements are");
       for(I=0;I<n;I++)
       printf("\n%d",a[I]);
       getch();
}
quicksort(int p, int q)
{
     int j;
     if(p,q)
     {
          j=partition(p,q+1);
          quicksort(p,j-1);
          quicksort(j+1,q);
      }
}

Partition(int m, int p)
{
      int v,I,j;
      v=a[m];
      i=m;
      j=p;
      do
      {
            do
                 i=i+1;
                 while(a[i]<v);
           if (i<j)
                  interchange(I,j);
      } while (I<j);
a[m]=a[j]; a[j]=v;

return j;
}

Interchange(int I, int j)
```

```
{
    int  p; p=
    a[I];
    a[I]=a[j];
    a[j]=p;
}
```

```
Output:
Enter the no. of elements 5
Enter the array elements
3
8
1
5
2
The sorted elements are,1
2
3
5
8
```

STRASSEN'S MATRIX MULTIPLICAION

1. Let A and B be the 2 n*n Matrix. The product matrix C=AB is calculated by using the formula,

   C (i ,j)= A(i,k) B(k,j) for all 'i' and and j between 1 and n.

2. The time complexity for the matrix Multiplication is O(n^3).

3. Divide and conquer method suggest another way to compute the product of n*n matrix.

4. We assume that N is a power of 2 .In the case N is not a power of 2 ,then enough rows and columns of zero can be added to both A and B .SO that the resulting dimension are the powers of two.

5. If n=2 then the following formula as a computed using a matrix multiplication operation for the elements of A & B.

6. If n>2,Then the elements are partitioned into sub matrix n/2*n/2..since 'n' is a power of 2 these product can be recursively computed using the same formula .This Algorithm will continue applying itself to smaller sub matrix until 'N" become suitable small(n=2) so that the product is computed directly .

7. The formula are

$$\begin{pmatrix} A11 & A12 \\ A21 & A21 \end{pmatrix} * \begin{pmatrix} B11 & B12 \\ B21 & B22 \end{pmatrix} = \begin{pmatrix} \mathbf{C11} & C12 \\ \mathbf{C21} & C22 \end{pmatrix}$$

C11 = A11 B11 + A12 B21
C12 = A11 B12 + A12 B22
C21 = A21 B11 + A22 B21
C22 = A21 B12 + A22 B22

For EX:

$$4*4= \begin{pmatrix} 2222 \\ 2222 \\ 2222 \\ 2222 \end{pmatrix} * \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

The Divide and conquer method

$$\begin{pmatrix} \begin{vmatrix} \mathbf{2} & \mathbf{2} \\ \mathbf{2} & \mathbf{2} \end{vmatrix} & \begin{vmatrix} \mathbf{2} & \mathbf{2} \\ \mathbf{2} & \mathbf{2} \end{vmatrix} \\ \begin{vmatrix} \mathbf{2} & \mathbf{2} \\ \mathbf{2} & \mathbf{2} \end{vmatrix} & \begin{vmatrix} \mathbf{2} & \mathbf{2} \\ \mathbf{2} & \mathbf{2} \end{vmatrix} \end{pmatrix} * \begin{pmatrix} \begin{vmatrix} \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{1} \end{vmatrix} & \begin{vmatrix} \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{1} \end{vmatrix} \\ \begin{vmatrix} \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{1} \end{vmatrix} & \begin{vmatrix} \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{1} \end{vmatrix} \end{pmatrix} = \begin{pmatrix} \begin{vmatrix} \mathbf{4} & \mathbf{4} \\ \mathbf{4} & \mathbf{4} \end{vmatrix} & \begin{vmatrix} \mathbf{4} & \mathbf{4} \\ \mathbf{4} & \mathbf{4} \end{vmatrix} \\ \begin{vmatrix} \mathbf{4} & \mathbf{4} \\ \mathbf{4} & \mathbf{4} \end{vmatrix} & \begin{vmatrix} \mathbf{4} & \mathbf{4} \\ \mathbf{4} & \mathbf{4} \end{vmatrix} \end{pmatrix}$$

8. 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.
9. Ci,j are computed using the formula in equation 4
10. As can be sum P, Q, R, S, T, U, and V can be computed using 7 Matrix multiplication and 10 addition or subtraction.
11. The Cij are required addition 8 addition or subtraction.

$$T(n)= \begin{cases} b & n<=2 \text{ a \&b are} \\ 7T(n/2)+an^2 & n>2 \text{ constant} \end{cases}$$

Finally we get T(n) =O(n ^log27)

Example

$$\begin{vmatrix} 4 \\ 4 \end{vmatrix} * \begin{vmatrix} 4 \\ 4 \end{vmatrix} \quad \begin{matrix} 4 & 4 \\ 4 & 4 \end{matrix}$$

P=(4+4) * (4+4)=64
Q=(4+4)4=32
R=4(4-4)=0
S=4(4-4)=0
T=(4+4)4=32
U=(4-4)(4+4)=0
V=(4-4)(4+4)=0
C11=(64+0-32+0)=32
C12=0+32=32
C21=32+0=32
C22=64+0-32+0=32

So the answer c(i,j) is 32 | | 32

| 32 32|

since n/2n/2 &matrix can be can be added in Cn for some constant C, The overall computing time T(n) of the resulting divide and conquer algorithm is given by the sequence.

$$T(n)= \begin{cases} b & n<=2 \text{ a \&b are} \\ 8T(n/2)+cn^2 & n>2 \text{ constant} \end{cases}$$

That is T(n)=O(n^3)

* Matrix multiplication are more expensive then the matrix addition O(n^3).We can attempt to reformulate the equation for Cij so as to have fewer multiplication and possibly more addition .

12. Stressen has discovered a way to compute the Cij of equation (2) usingonly 7 multiplication and 18 addition or subtraction.

13. Strassen's formula are

P= (A11+A12)(B11+B22)
Q= (A12+A22)B11
R= A11(B12-B22)
S= A22(B21-B11)
T= (A11+A12)B22
U= (A21-A11)(B11+B12)
V= (A12-A22)(B21+B22)

C11=P+S-T+V
C!2=R+t
C21=Q+T
C22=P+R-Q+V

UNIT - II

GREEDY METHOD

Greedy method is the most straightforward designed technique.

As the name suggest they are short sighted in their approach taking decision on the basis of the information immediately at the hand without worrying about the effect these decision may have in the future.

DEFINITION:

A problem with N inputs will have some constraints .any subsets that satisfy these constraints are called a feasible solution.

A feasible solution that either maximize can minimize a given objectives function is called an optimal solution.

Control algorithm for Greedy Method:

```
1.Algorithm Greedy (a,n)
2.//a[1:n] contain the 'n' inputs3.
{
4.solution =0;//Initialise the solution.
5.For i=1 to n do
6.{
7.x=select(a);
8.if(feasible(solution,x))then
9.solution=union(solution,x);
10.}
11.return solution;
12.}
```

* The function select an input from a[] and removes it. The select input value isassigned to X.

Feasible is a Boolean value function that determines whether X can be included into the solution vector.

The function Union combines X with The solution and updates the objective function.

The function Greedy describes the essential way that a greedy algorithm will once a particular problem is chosen ands the function subset, feasible & union are properly implemented.

Example

Suppose we have in a country the following coins are available :Dollars(100

cents)

Quarters(25 cents)

Dimes(10 cents)

Nickel(5 Cents)

Pennies(1 cent)

Our aim is paying a given amount to a customer using the smallest possible number of coins.

For example if we must pay 276 cents possible solution then,1

doll+7 q+ 1 pen 9 coins 2

doll +3Q +1 pen 6 coins

2 doll+7dim+1 nic +1 pen 11 coins.

KNAPSACK PROBLEM

we are given n objects and knapsack or bag with capacity M object I has a weightWi where I varies from 1 to N.

The problem is we have to fill the bag with the help of N objects and the resultingprofit has to be maximum.

Formally the problem can be stated as

Maximize xipi subject to XiWi<=M

Where Xi is the fraction of object and it lies between 0 to 1.

There are so many ways to solve this problem, which will give many feasible solution 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.

First, we find the profit & weight rates of each and every object and sort itaccording to the descending order of the ratios.

Select an object with highest p/w ratio and check whether its height is lesser than the capacity of the bag.

If so place 1 unit of the first object and decrement .the capacity of the bag by the weight of the object you have placed.

Repeat the above steps until the capacity of the bag becomes less than the weightof the object you have selected .in this case place a fraction of the object and come out of the loop.

Whenever you selected.

ALGORITHM:

```
1.Algorithm Greedy knapsack (m,n) 2//P[1:n]
and the w[1:n]contain the profit3.// & weight
res'.of the n object ordered.4.//such that
p[i]/w[i] >=p[i+1]/W[i+1]
5.//n is the Knapsack size and x[1:n] is the solution vertex.6.{
7.for I=1 to n do a[I]=0.0;
8.U=n;
9.For I=1 to n do
10.{
11.if (w[i]>u)then break;
13.x[i]=1.0;U=U-w[i]
14.}
15.if(i<=n)then x[i]=U/w[i];
16.}
```

Example:

Capacity=20

N=3 ,M=20

Wi=18,15,10

Pi=25,24,15

Pi/Wi=25/18=1.36,24/15=1.6,15/10=1.5

Descending Order	Pi/Wi	1.6	1.5	1.36
Pi	=	24	15	25
Wi	=	15	10	18
Xi	=	1	5/10	0

PiXi=1*24+0.5*15 31.5

The optimal solution is 31.5

X1 X2 X3	WiXi	PiXi
½1/3¼	16.6	24.25
1 2/5 0	20	18.2
0 2/3 1	20	31
0 1 ½	20	31.5

Of these feasible solution Solution 4 yield the Max profit .As we shall soon seethis solution is optimal for the given problem instance.

JOB SCHEDULING WITH DEAD LINES

The problem is the number of jobs, their profit and deadlines will be given and we have to find a sequence of job, which will be completed within its deadlines, and itshould yield a maximum profit.

Points To remember:

To complete a job, one has to process the job or a action for one unit of time.Only

one machine is available for processing jobs.

A feasible solution for this problem is a subset of j of jobs such that each job inthis subject can be completed by this deadline.

If we select a job at that time ,

Since one job can be processed in a single m/c. The other job has to be in its waiting state until the job is completed and the machine becomes free.

So the waiting time and the processing time should be less than or equal to thedead line of the job.

ALGORITHM:

```
Algorithm JS(d,j,n)

//The job are ordered such that p[1]>p[2]…>p[n] //j[i]is

the ith job in the optimal solution

//  Also at terminal d [ J[ i]<=d[ J {i+1],1<i<k

 {

  d[0]= J[0]=0;




{ // consider jobs in non increasing order of P[I];find the position for I and checkfeasibility
insertion

r=k;

while((d[J[r]]>d[i] )and

  (d[J[r]] ≠ r)do r =r-1;

if (d[J[r]]<d[I])and (d[I]>r))then

{

for q=k to (r+1) step –1 do J [q+1]=j[q]

J[r+1]=i;


}

}

return k;

}
```

1. n=5 (P1,P2,…P5)=(20,15,10,5,1)

   (d1,d2….d3)=(2,2,1,3,3)

Feasible solution	Processing Sequence	Value
(1)	(1)	20
(2)	(2)	15
(3)	(3)	10
(4)	(4)	5
(5)	(5)	1
1,2)	(2,1)	35
(1,3)	(3,1)	30
(1,4)	(1,4)	25
(1,5)	(1,5)	21
(2,3)	(3,2)	25
(2,4)	(2,4)	20
(2,5)	(2,5)	16
(1,2,3)	(3,2,1)	45
(1,2,4)	(1,2,4)	40

The Solution 13 is optimal

n=4 (P1,P2,…P4)=(100,10,15,27)

(d1,d2….d4)=(2,1,2,1)

Feasible solution	Processing Sequence	Value
(1,2)	(2,1)	110
(1,3)	(1,3)	115
(1,4)	(4,1)	127
(2,3)	(9,3)	25
(2,4)	(4,2)	37
(3,4)	(4,3)	42
(1)	(1)	100

(2)	(2)	10
(3)	(3)	15
(4)	(4)	27

The solution 3 is optimal. 2.4.MINIMUM

COST SPANNING TREE

Let G(V,E) be an undirected connected graph with vertices 'v' and edge 'E'.A

sub-graph t=(V,E') of the G is a Spanning tree of G iff 't' is a tree.3

The problem is to generate a graph G'= (V,E) where 'E' is the subset of E,G' is a Minimum spanning tree.

Each and every edge will contain the given non-negative length .connect all thenodes with edge present in set E' and weight has to be minimum.

NOTE:

We have to visit all the nodes.

The subset tree (i.e) any connected graph with 'N' vertices must have at least N-1 edges and also it does not form a cycle.

Definition:

A spanning tree of a graph is an undirected tree consisting of only those edge thatare necessary to connect all the vertices in the original graph.

A Spanning tree has a property that for any pair of vertices there exist only one path between them and the insertion of an edge to a spanning tree form a unique cycle.

Application of the spanning tree:

1. Analysis of electrical circuit.
2. Shortest route problems.

METHODS OFMINIMUM COST SPANNING TREE:

The cost of a spanning tree is the sum of cost of the edges in that trees.There are

2 method to determine a minimum cost spanning tree are

1. Kruskal's Algorithm
2. Prom's Algorithm.

KRUSKAL’S ALGORITHM:

In kruskal's algorithm the selection function chooses edges in increasing order of length without worrying too much about their connection to previously chosen edges, except that never to form a cycle. The result is a forest of trees that grows until all the trees in a forest (all the components) merge in a single tree.

In this algorithm, a minimum cost-spanning tree ‘T’ is built edge by edge.Edge

are considered for inclusion in ‘T’ in increasing order of their cost.

An edge is included in ‘T’ if it doesn’t form a cycle with edge already in T.

To find the minimum cost spanning tree the edge are inserted to tree in increasingorder of their cost

Algorithm:

```
Algorithm kruskal(E,cost,n,t)
//E    set of edges in G has ‘n’ vertices.
//cost[u,v]      cost of edge (u,v).t        set of edge in minimum cost spanning tree
// the first cost is returned.
{
for i=1 to n do parent[I]=-1;
I=0;mincost=0.0; While((I<n-1)and (heap
not empty)) do
{
j=find(n);
k=find(v);
if(j not equal k) than
{
i=i+1
t[i,1]=u;
t[i,2]=v;
mincost=mincost+cost[u,v];
union(j,k);
        }
    }
if(i notequal n-1) then write("No spanning tree")else
return minimum cost;
}
```

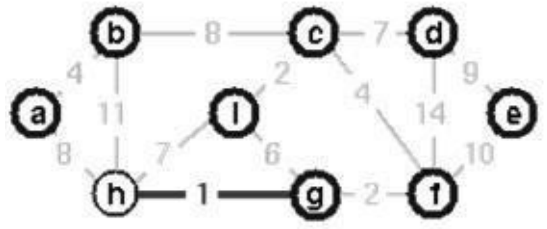
Analysis

14. The time complexity of minimum cost spanning tree algorithm in worstcase is O(|E|log|E|),

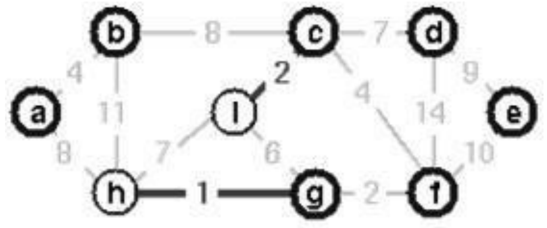
where E is the edge set of G.

<u>Example: Step by Step operation of Kurskal algorithm.</u>

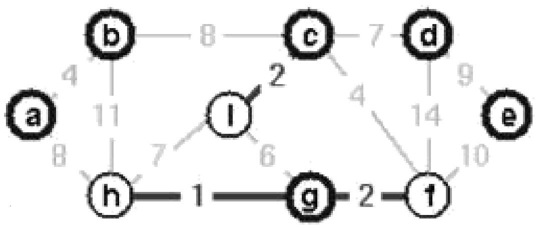
Step 1. In the graph, the Edge(g, h) is shortest. Either vertex g or vertex h could be representative. Lets choose vertex g arbitrarily.



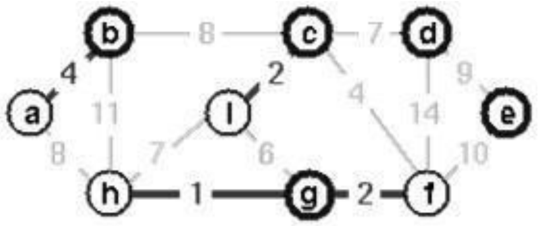
Step 2. The edge (c, i) creates the second tree. Choose vertex c as representativefor second tree.



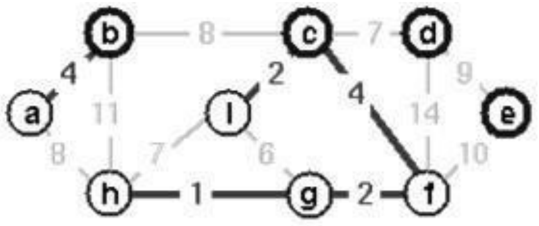
Step 3. Edge (g, g) is the next shortest edge. Add this edge and choose vertex gas representative.



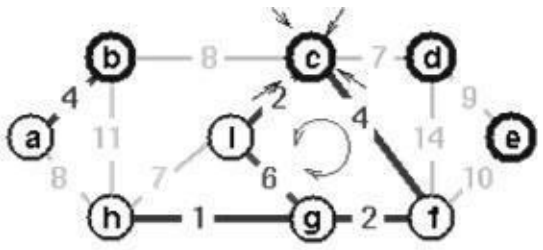
Step 4. Edge (a, b) creates a third tree.



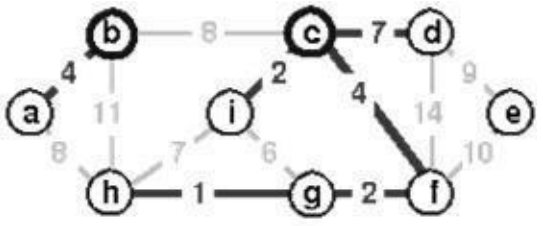
Step 5. Add edge (c, f) and merge two trees. Vertex c is chosen as the representative.



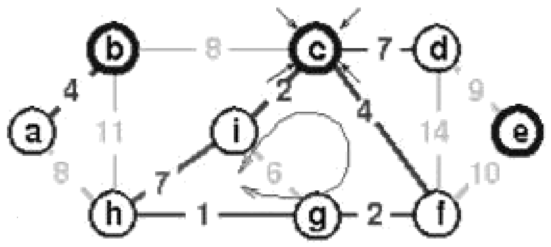
Step 6. Edge (g, i) is the next next cheapest, but if we add this edge a cycle wouldbe created. Vertex c is the representative of both.



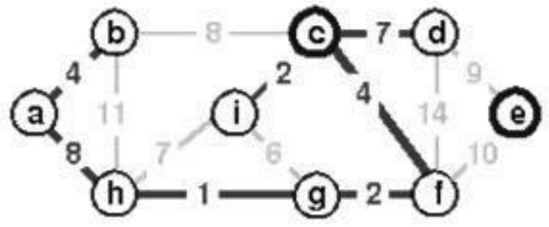
Step 7. Instead, add edge (c, d).



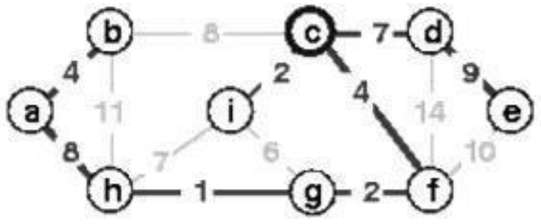
Step 8. If we add edge (h, i), edge(h, i) would make a cycle.



Step 9. Instead of adding edge (h, i) add edge (a, h).

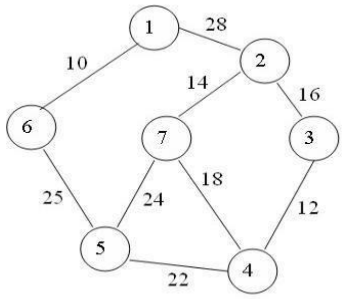


Step 10. Again, if we add edge (b, c), it would create a cycle. Add edge (d, e) instead to complete the spanning tree. In this spanning tree all trees joined and vertex c is a sole representative.



PRIM'S ALGORITHM

Start from an arbitrary vertex (root). At each stage, add a new branch (edge) to the tree already constructed; the algorithm halts when all the vertices in the graph have been reached.



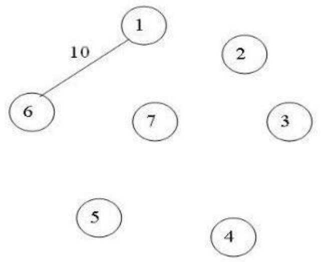
```
Algorithm prims(e,cost,n,t)
{
  Let (k,l) be an edge of minimum cost in E;
  Mincost :=cost[k,l];
  T[1,1]:=k; t[1,2]:=l;
  For I:=1 to n do
```

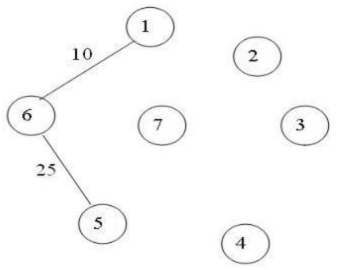
```
        If (cost[i,l]<cost[i,k]) then near[i]:=l;
        Else near[i]:=k;
    Near[k]:=near[l]:=0;
    For i:=2 to n-1 do
    {
      Let j be an index such that near[j]≠0 and
      Cost[j,near[j]] is minimum;
      T[i,1]:=j; t[i,2]:=near[j];
      Mincost:=mincost+ Cost[j,near[j]];
      Near[j]:=0;
      For k:=0 to n do
          If near((near[k]≠0) and (Cost[k,near[k]]>cost[k,j])) then
         Near[k]:=j;
    }
    Return mincost;
}
```

15. The prims algorithm will start with a tree that includes only a minimum cost edge of G.
16. Then, edges are added to the tree one by one. the next edge (i,j) to be added in such that I is a vertex included in the tree, j is a vertex not yet included, and cost of (i,j), cost[i,j] is minimum among all the edges.

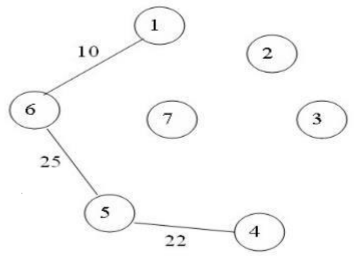
17. The working of prims will be explained by following diagramStep

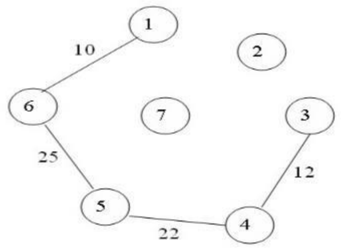
1: Step 2:



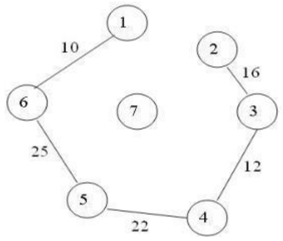


Step 3: Step 4:

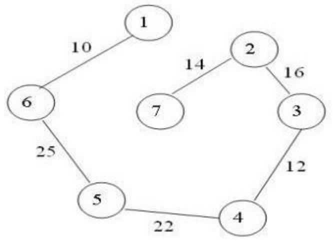




Step 5:



Step 6:



SINGLE-SOURCE SHORTEST PATH:

Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges can then be assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway. A motorist wishing to drive from city A to B would be interested in answers to the following questions:

- o Is there a path from A to B?
- o If there is more than one path from A to B? Which is the shortest path?

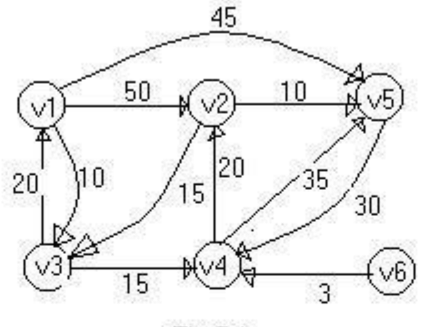


Fig 7.1

The problems defined by these questions are special case of the path problem we study in this section. The length of a path is now defined to be the sum of the weights of the edges on that path. The starting vertex of the path is referred to as the source and the last vertex the destination. The graphs are digraphs representing streets. Consider a digraph G=(V,E), with the distance to be traveled as weights on the edges. The problem is to determine the shortest path from v0 to all the remaining vertices of G. It is assumed that all the weights associated with the edges are positive. The shortest path between v0 and some other node v is an ordering among a subset of the edges. Hence this problem fits the ordering paradigm.

Example:

Consider the digraph of fig 7- 1. Let the numbers on the edges be the costs of travelling along that route. If a person is interested travel from v1 to v2, then he encounters many paths. Some of them are

- o v1 v2 = 50 units
- o v1 v3 v4 v2 = 10+15+20=45 units
- o v1 v5 v4 v2 = 45+30+20= 95 units
- o v1 v3 v4 v5 v4 v2 = 10+15+35+30+20=110 units

The cheapest path among these is the path along v1 v3 v4 v2. The cost of the path is 10+15+20 = 45 units. Even though there are three edges on this path, it is cheaper than travelling along the path connecting v1 and v2 directly i.e., the path v1 v2 that costs 50 units. One can also notice that, it is not possible to travel to v6 from any other node.

To formulate a greedy based algorithm to generate the cheapest paths, we must conceive a multistage solution to the problem and also of an optimization measure. One possibility is to build the shortest paths one by one. As an optimization measure we can use the sum of the lengths of all paths so far generated. For this measure to be minimized, each individual path must be of minimum length. If we have already constructed i shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path. The greedy way 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.

A much simpler method would be to solve it using matrix representation. The steps that should be followed is as follows,

Step 1: find the adjacency matrix for the given graph. The adjacency matrix for fig is given below

	V1	V2	V3	V4	V5	V6
V1	-	50	10	Inf	45	Inf
V2	Inf	-	15	Inf	10	Inf
V3	20	Inf	-	15	inf	Inf
V4	Inf	20	Inf	-	35	Inf
V5	Inf	Inf	Inf	30	-	Inf
V6	Inf	Inf	Inf	3	Inf	-

Step 2: consider v1 to be the source and choose the minimum entry in the row v1.In the above table the minimum in row v1 is 10.

Step 3: find out the column in which the minimum is present, for the above example it is column v3. Hence, this is the node that has to be next visited.

Step 4: compute a matrix by eliminating v1 and v3 columns. Initially retain onlyrow v1. The second row is computed by adding 10 to all values of row v3.

The resulting matrix is

	V2	V4	V5	V6
V1 Vw	50	Inf	45	Inf
V1 V3 Vw	10+inf	10+15	10+inf	10+inf
Minimum	50	25	45	inf

Step 5: find the minimum in each column. Now select the minimum from the resulting row. In the above example the minimum is 25. Repeat step 3 followed by step 4 till all vertices are covered or single column is left.

The solution for the fig 7.1 can be continued as follows

	V2	V5	V6
V1 Vw	50	45	Inf
V1 V3 V4 25+20 Vw		25+35	25+inf
Minimum	45	45	inf

	V5	V6
V1 Vw	45	Inf
V1 V3 V4 V2 45+10 Vw	45+inf	
Minimum	45	Inf

	V6
V1 Vw	Inf
V1 V3 V4 V2 V5 Vw	45+inf
Minimum	inf

Finally the cheapest path from v1 to all other vertices is given by V1 V3 V4 V2 V5.

DYNAMIC PROGRAMMING

DYNAMIC PROGRAMING GENERAL METHOD:

1. The idea of dynamic programming is thus quit simple: avoid calculating thesame thing twice, usually by keeping a table of known result that fills up a sub instances are solved.
2. Divide and conquer is a top-down method.
3. When a problem is solved by divide and conquer, we immediately attack the complete instance, which we then divide into smaller and smaller sub- instances as the algorithm progresses.
4. Dynamic programming on the other hand is a bottom-up technique.
5. We usually start with the smallest and hence the simplest sub-instances.
6. By combining their solutions, we obtain the answers to sub-instances of increasing size, until finally we arrive at the solution of the original instances.
7. The essential difference between the greedy method and dynamic programming is that the greedy method only one decision sequence is evergenerated.
8. In dynamic programming, many decision sequences may be generated. However, sequences containing sub-optimal sub-sequences can not be optimal and so will not be generated.

ALL PAIR SHORTEST PATH

Let G=<N,A> be a directed graph 'N' is a set of nodes and 'A' is the set of edges.

1. Each edge has an associated non-negative length.
2. We want to calculate the length of the shortest path between each pair of nodes.
3. Suppose the nodes of G are numbered from 1 to n, so N={1,2,...N},and suppose G matrix L gives the length of each edge, with L(i,j)=0 for i=1,2...n,L(i,j)>=for all i& j, and L(i,j)=infinity, if the edge (i,j) does not exist.
4. The principle of optimality applies: if k is the node on the shortest path from i toj then the part of the path from i to k and the part from k to j must also be optimal, that is shorter.
5. First, create a cost adjacency matrix for the given graph.

6. Copy the above matrix-to-matrix D, which will give the direct distance between nodes.

7. We have to perform N iteration after iteration k.the matrix D will give you the distance between nodes with only (1,2...,k)as intermediate nodes.

8. At the iteration k, we have to check for each pair of nodes (i,j) whether or notthere exists a path from i to j passing through node k.

COST ADJACENCY MATRIX:

D0=L= 05

500 15 5

30 0 15

15 5 0

1 7 5 1112 - -

2 72 21--2

3 3 - 32 - -

4 4 1 41 – 43 -

vertex 1:

7 5 11 12 - -

7 12 2 21 212 - 24

3 - 32 - -

4 9 1 41 412 43 –

vertex 2:

$$\left|\begin{array}{cccc} 7 & 5 & & 7 \\ 7\ 12 & & & 2 \\ 10\ 3 & & & 5 \\ 4 & 9 & 1 & 11 \end{array}\right| \quad \left|\begin{array}{cccc} 11 & 12 & - & 124 \\ 21 & 212 & - & 24 \\ 321\ 32 & & - & 324 \\ 41 & 412 & 43 & 4124 \end{array}\right|$$

vertex 3:

$$\left|\begin{array}{cccc} 7 & 5 & & 7 \\ 7\ 12 & & & 2 \\ 10 & 3 & & 5 \\ 4 & 4 & 1 & 6 \end{array}\right| \quad \left|\begin{array}{cccc} 11 & 12 & - & 124 \\ 21 & 212 & - & 24 \\ 321 & 32 & - & 324 \\ 41 & 432 & 43 & 4324 \end{array}\right|$$

vertex 4:

$$\left|\begin{array}{cccc} 7 & 5 & 8 & 7 \\ 6 & 6 & 3 & 2 \\ 9\ 3 & & 6 & 5 \\ 4 & 4 & 1 & 6 \end{array}\right| \quad \left|\begin{array}{cccc} 11 & 12 & 1243 & 124 \\ 241 & 2432 & 243 & 24 \\ 3241 & 32 & 3243 & 324 \\ 41 & 432 & 43 & 4324 \end{array}\right|$$

1. At 0th iteration it nil give you the direct distances between any 2 nodes

$$D0=0\left|\begin{array}{ccc} & 5 & \\ 50015 & & 5 \\ 30 & 015 & \\ 15 & 5 & 0 \end{array}\right|$$

At 1st iteration we have to check the each pair(i,j) whether there is a path throughnode 1.if so we have to check whether it is minimum than the previous value andif I is so than the distance through 1 is the value of d1(i,j).at the same time we have to solve the intermediate node in the matrix position p(i,j).

```
              0    5
              50   0   15   5      p[3,2]= 1
        D1=   3035 015              p[4,2]= 1
               1520 5       0
```

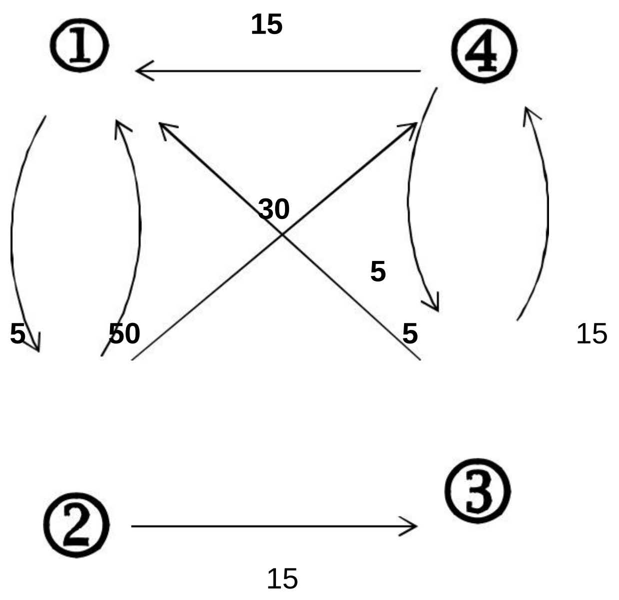


Fig: floyd’s algorithm and work

3. likewise we have to find the value for N iteration (ie) for N nodes.

$$D2 = \begin{vmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{vmatrix} \quad \begin{matrix} P[1,3] = 2 \\ P[1,4] = 2 \end{matrix}$$

$$D3 = \begin{vmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{vmatrix} \quad P[2,1]=3$$

$$D4 = \begin{vmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{vmatrix} \quad \begin{matrix} P[1,3]=4 \\ P[2,3]=4 \end{matrix}$$

4. D4 will give the shortest distance between any pair of nodes.

5. If you want the exact path then we have to refer the matrix p.The matrix willbe,

$$P = \begin{vmatrix} 0042 \\ 3040 \\ 0100 \\ 0100 \end{vmatrix} \qquad 0 \longrightarrow \text{direct path}$$

Since,p[1,3]=4,the shortest path from 1 to3 passes through 4.

Looking now at p[1,4]&p[4,3] we discover that between 1 & 4, we have to goto node 2 but that from 4 to 3 we proceed directly.

Finally we see the trips from 1 to 2, & from 2 to 4, are also direct. The

shortest path from 1 to 3 is 1,2,4,3.

ALGORITHM :

```
Function Floyd (L[1..r,1..r]):array[1..n,1..n]
array D[1..n,1..n]
D = L
For k = 1 to n do
For i = 1 to n do
For j = 1 to n do
D [ i , j ] = min (D[ i, j ], D[ i, k ] + D[ k, j ]
Return D
```

ANALYSIS:

This algorithm takes a time of (n^3)

MULTISTAGE GRAPH

A multistage graph G = (V,E) is a directed graph in which the vertices are portioned into K > = 2 disjoint sets Vi, 1 <= i<= k.

In addition, if < u,v > is an edge in E, then u < = Vi and V Vi+1 for some i, 1<= i < k.

If there will be only one vertex, then the sets Vi and Vk are such that [Vi]=[Vk] = 1.

Let ‘s’ and ‘t’ be the source and destination respectively.

The cost of a path from source (s) to destination (t) is the sum of the costs ofthe edger on the path.

The MULTISTAGE GRAPH problem is to find a minimum cost path from ‘s’ to‘t’.

Each set Vi defines a stage in the graph. Every path from ‘s’ to ‘t’ starts in stage-1, goes to stage-2 then to stage-3, then to stage-4, and so on, and terminates in stage-k.

This MULISTAGE GRAPH problem can be solved in 2 ways. o

Forward Method.

o Backward Method.

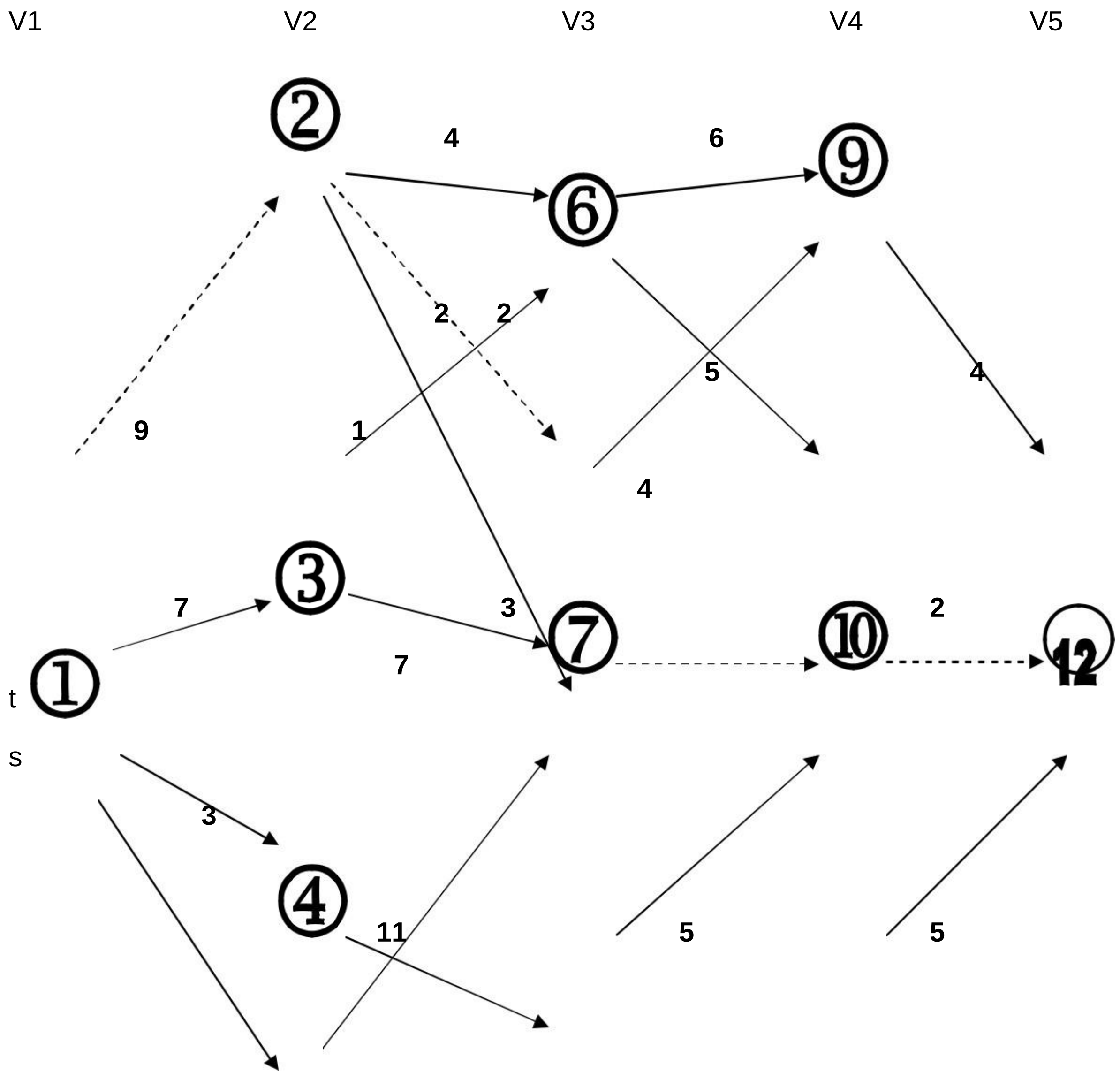
FORWARD METHOD

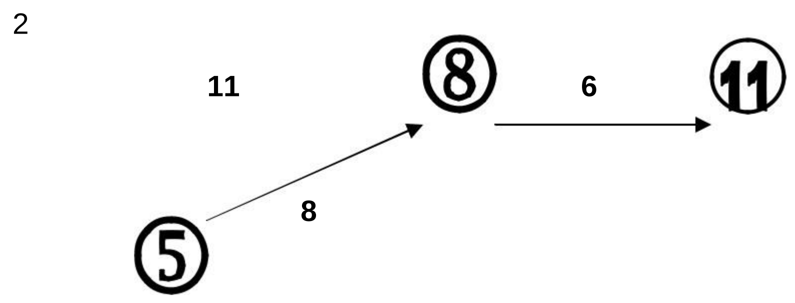
Assume that there are ‘k’ stages in a graph.

In this FORWARD approach, we will find out the cost of each and every node starling from the ‘k’ th stage to the 1st stage.

We will find out the path (i.e.) minimum cost path from source to the destination (ie) [Stage-1 to Stage-k].

PROCEDURE:





1. Maintain a cost matrix cost (n) which stores the distance from any vertex to the destination.
2. If a vertex is having more than one path, then we have to choose the minimum distance path and the intermediate vertex, which gives the minimum distance path, will be stored in the distance array 'D'.
3. In this way we will find out the minimum cost path from each and every vertex.
4. Finally cost(1) will give the shortest distance from source to destination.
5. For finding the path, start from vertex-1 then the distance array D(1) will give the minimum cost neighbour vertex which in turn give the next nearest vertex and proceed in this way till we reach the Destination.
6. For a 'k' stage graph, there will be 'k' vertex in the path.
7. In the above graph V1…V5 represent the stages. This 5 stage graph can be solved by using forward approach as follows,

STEPS: -		DESTINATION, D
Cost (12)=0	$\longrightarrow$	D (12)=0
Cost (11)=5	$\longrightarrow$	D (11)=12
Cost (10)=2	$\longrightarrow$	D (10)=12
Cost (9)=4	$\longrightarrow$	D (9)=12

1. For forward approach,

Cost (i,j) = min {C (j,l) + Cost (i+1,l) }

l Vi + 1

(j,l) E

Cost(8) = min {C (8,10) + Cost (10), C (8,11) + Cost (11) }

= min (5 + 2, 6 + 5)

= min (7,11)

= 7

cost(8) =7 =>D(8)=10

cost(7) = min(c (7,9)+ cost(9),c (7,10)+ cost(10))

(4+4,3+2)

= min(8,5)

= 5

cost(7) = 5 =>D(7) = 10

cost(6) = min (c (6,9) + cost(9),c (6,10) +cost(10))

= min(6+4 , 5 +2)

= min(10,7)

= 7

cost(6) = 7 =>D(6) = 10

cost(5) = min (c (5,7) + cost(7),c (5,8) +cost(8))

= min(11+5 , 8 +7)

= min(16,15)

= 15

cost(5) = 15 =>D(5) = 18

cost(4) = min (c (4,8) + cost(8))

= min(11+7)

= 18

cost(4) = 18 =>D(4) = 8

cost(3) = min (c (3,6) + cost(6),c (3,7) +cost(7))

= min(2+7 , 7 +5)

= min(9,12)

= 9

cost(3) = 9 =>D(3) = 6

cost(2) = min (c (2,6) + cost(6),c (2,7) +cost(7) ,c (2,8) +cost(8))

= min(4+7 , 2+5 , 1+7)

= min(11,7,8)

= 7

cost(2) = 7 =>D(2) = 7

cost(1) = min (c (1,2)+cost(2) ,c (1,3)+cost(3) ,c (1,4)+cost(4) ,c(1,5)+cost(5))

= min(9+7 , 7 +9 , 3+18 , 2+15)

= min(16,16,21,17)

= 16

cost(1) = 16 =>D(1) = 2

.... $\longrightarrow$ The path through which you have to find the shortest distance.

(i.e.)

$$\begin{tikzcd} \textcircled{1} \arrow[r] & \textcircled{2} \arrow[r] & \textcircled{7} \arrow[r] & \textcircled{10} \arrow[r] & \textcircled{12} \end{tikzcd}$$

Start from vertex - 2

D(1) = 2

D(2) = 7

D(7) =10

D (10) = 12

So, the minimum –cost path is,

$$\begin{tikzcd} \textcircled{1} \arrow[r, "9"] & \textcircled{2} \arrow[r, "2"] & \textcircled{7} \arrow[r, "3"] & \textcircled{10} \arrow[r, "2"] & \textcircled{12} \end{tikzcd}$$

The cost is 9+2+3+2+=16

ALGORITHM: FORWARD METHOD

Algorithm FGraph (G,k,n,p)

// The I/p is a k-stage graph G=(V,E) with 'n' vertex.

// Indexed in order of stages E is a set of edges.

```
// and c[i,J] is the cost of<i,j>,p[1:k] is a minimum cost path.
{
    cost[n]=0.0;
    for j=n-1 to 1 step-1 do
    {
        //compute cost[j],
        // let 'r' be the vertex such that <j,r> is an edge of 'G' &
        // c[j,r]+cost[r] is minimum.

        cost[j] = c[j+r] + cost[r];
        d[j] =r;
    }
    // find a minimum cost path.

        P[1]=1;
        P[k]=n;
        For j=2 to k-1 do
        P[j]=d[p[j-1]];
}
```

ANALYSIS:

The time complexity of this forward method is O($|V| + |E|$)

2.8.2.BACKWARD METHOD

if there one 'K' stages in a graph using back ward approach. we will findout the cost of each & every vertex starting from 1st stage to the kth stage.

We will find out the minimum cost path from destination to source (ie)[fromstage k to stage 1]

PROCEDURE:

It is similar to forward approach, but differs only in two or three ways.

Maintain a cost matrix to store the cost of every vertices and a distance matrix to store the minimum distance vertex.

Find out the cost of each and every vertex starting from vertex 1 up to vertex k.

To find out the path star from vertex 'k', then the distance array D (k) will give the minimum cost neighbor vertex which in turn gives the next nearest neighbor vertex and proceed till we reach the destination.

STEP:

Cost(1) = 0 => D(1)=0

Cost(2) = 9 => D(2)=1

Cost(3) = 7 => D(3)=1

Cost(4) = 3 => D(4)=1

Cost(5) = 2 => D(5)=1

Cost(6) =min(c (2,6) + cost(2),c (3,6) + cost(3))

=min(13,9)

cost(6) = 9 =>D(6)=3

Cost(7) =min(c (3,7) + cost(3),c (5,7) + cost(5) ,c (2,7) + cost(2))

=min(14,13,11)

cost(7) = 11 =>D(7)=2

Cost(8) =min(c (2,8) + cost(2),c (4,8) + cost(4) ,c (5,8) +cost(5))

=min(10,14,10)

cost(8) = 10 =>D(8)=2

Cost(9) =min(c (6,9) + cost(6),c (7,9) + cost(7))

=min(15,15)

cost(9) = 15 =>D(9)=6

Cost(10)=min(c(6,10)+cost(6),c(7,10)+cost(7)),c (8,10)+cost(8)) =min(14,14,15)

cost(10)= 14 =>D(10)=6

Cost(11) =min(c (8,11) + cost(8))

cost(11) = 16 =>D(11)=8

cost(12)=min(c(9,12)+cost(9),c(10,12)+cost(10),c(11,12)+cost(11))

=min(19,16,21)

cost(12) = 16 =>D(12)=10

PATH:

Start from vertex-12

D(12) = 10

D(10) = 6

D(6) = 3

D(3) = 1

So the minimum cost path is,

$\mathbf{1^{7} \rightarrow 3^{2} \rightarrow 6^{5} \rightarrow 10^{2} \rightarrow 12}$

The cost is 16.

ALGORITHM : BACKWARD METHOD

```
Algorithm BGraph (G,k,n,p)
//   The I/p is a k-stage graph G=(V,E) with 'n' vertex.
//   Indexed in order of stages E is a set of edges.
// and c[i,J] is the cost of<i,j>,p[1:k] is a minimum cost path.
{
      bcost[1]=0.0; for
       j=2 to n do
   {
       //compute bcost[j],
      //    let 'r' be the vertex such that <r,j> is an edge of 'G' &
     //    bcost[r]+c[r,j] is minimum.
```

```
  bcost[j] = bcost[r] + c[r,j];
   d[j] =r;
 }
// find a minimum cost path.


     P[1]=1;
     P[k]=n;
     For j= k-1 to 2 do
     P[j]=d[p[j+1]];
}
```

TRAVELLING SALESMAN PROBLEM

Let G(V,E) be a directed graph with edge cost c_{ij} is defined such that c_{ij} >0 forall i and j and c_{ij} = ,if <i,j> E.

Let |V| =n and assume n>1.

The traveling salesman problem is to find a tour of minimum cost. Atour of G

is a directed cycle that include every vertex in V.

The cost of the tour is the sum of cost of the edges on the tour.

The tour is the shortest path that starts and ends at the same vertex (ie) 1.

APPLICATION :

Suppose we have to route a postal van to pick up mail from the mail boxes located at ‘n’ different sites.

An n+1 vertex graph can be used to represent the situation.

One vertex represent the post office from which the postal van starts and return.

Edge <i,j> 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 ofminimum length.

every tour consists of an edge <1,k> for some k V-{} and a path from vertexk 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

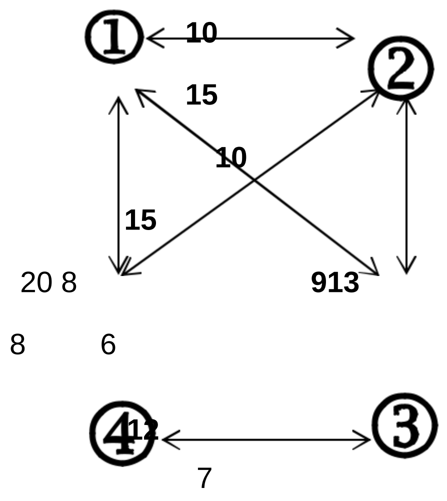
$g(1,V-\{1\}) = \min\{ c_{ij} + g(j,s-\{j\})\}$

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.

the function g(1,v-{1}) is the length of an optimal tour.

1. Find $g(i,) = c_{i1}$, 1<=i<n, hence we can use equation(2) to obtain g(i,s) for all s to size 1.
2. That we have to start with s=1,(ie) there will be only one vertex in set ‘s’.
3. Then s=2, and we have to proceed until |s| <n-1.
4. for example consider the graph.



Cost matrix

0101520

5 0 910

6 13012

8890

$$\begin{tikzcd}g(i,s) \arrow[r] \arrow[d] & \text{set of nodes/vertex have to visited.} \\ \text{starting position} & \end{tikzcd}$$

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

STEP 1:

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

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

$\min\{35,35,43\}$

$=35$

STEP 2:

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

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

$\min\{29,25\}$

$=25$

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

$\min\{13+18,12+13\}$

$\min\{31,25\}$

$=25$

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

$\min\{8+15,9+18\}$

$\min\{23,27\}$

$=23$

STEP 3:

1. $g(3,\{4\}) = \min\{c_{34} + g\{4, \}\}$ 12+8

$=20$

2. g(4,{3}) = min{c_{43} +g{3, }} 9+6

   =15

3. g(2,{4}) = min{c_{24} +g{4, }} 10+8

   =18

4. g(4,{2}) = min{c_{42} +g{2, }} 8+5

   =13

5. g(2,{3}) = min{c_{23} +g{3, }}

   9+6=15

6. g(3,{2}) = min{c_{32} +g{2, }}

   13+5=18

STEP 4:

g{4, } =c_{41} = 8

g{3, } =c_{31} = 6

g{2, } =c_{21} = 5

$|s| = 0.$

i =1 to n.

g(1,) = c_{11} => 0

g(2,) = c_{21} => 5

g(3,) = c_{31} => 6

g(4,) = c_{41} => 8

$|s| = 1$

i =2 to 4

g(2,{3}) = c_{23} + g(3,)

= 9+6 =15

$g(2,\{4\}) = c_{24} + g(4,)$

$= 10+8 =18$

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

$= 13+5 =18$

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

$= 12+8 =20$

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

$= 8+5 =13$

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

$= 9+6 =15$

$| \quad s| \quad = 2$

i 1, 1 s and i s.

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

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

$\min\{29,25\}$

$=25$

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

$\min\{13+18,12+13\}$

$\min\{31,25\}$

$=25$

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

$\min\{8+15,9+18\}$

$\min\{23,27\}$

$=23$

$|s| = 3$

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

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

$\min\{35,35,43\}$

$=35$

optimal cost is 35

the shortest path is,

$g(1,\{2,3,4\}) = c_{12} + g(2,\{3,4\})$ => 1->2

$g(2,\{3,4\}) = c_{24} + g(4,\{3\})$ => 1->2->4

$g(4,\{3\}) = c_{43} + g(3\{\ \})$ => 1->2->4->3->1

so the optimal tour is 1 2 4 3 1

0/1 KNAPSACK PROBLEM:

1. This problem is similar to ordinary knapsack problem but we may not takea fraction of an object.

2. We are given ‘ N ‘ object with weight W_i and profits P_i where I varies from lto N and also a knapsack with capacity ‘ M ‘.

3. The problem is, we have to fill the bag with the help of ‘ N ‘ objects and the resulting profit has to be maximum.

4. Formally, the problem can be started as, maximize $X_i\ P_i$i=l-

   n

   subject to $X_i\ W_i$ L M

   i=l- n

5. Where X_i are constraints on the solution X_i {0,1}. (u) X_i is required to be 0 or **1.** if the object is selected then the unit in 1. if the object is rejected than theunit is 0. That is why it is called as 0/1, knapsack problem.

6. To solve the problem by dynamic programming we up a table T[1…N, 0…M] (ic) the size is N. where ‘N’ is the no. of objects and column starts with ‘O’ to capacity (ic) ‘M’.

7. In the table T[i,j] will be the maximum valve of the objects i varies from 1 ton and j varies from O to M.

RULES TO FILL THE TABLE:-

1. If i=l and j < w(i) then T(i,j) =o, (ic) o pre is filled in the table.

2. If i=l and j w (i) then T (i,j) = p(i), the cell is filled with the profit p[i], sinceonly one object can be selected to the maximum.

3. If i>I and j < w(i) then T(i,I) = T (i-I,j) the cell is filled the profit of previous object since it is not possible with the current object.

4. If i>I and j w(i) then T (i,j) = {f(i) +T(i-I,j-w(i)),. since only 'I' unit can be selected to the maximum. If is the current profit + profit of the previous object to fill the remaining capacity of the bag.

5. After the table is generated, it will give details the profit.

ES TO GET THE COMBINATION OF OBJECT:

Start with the last position of i and j, T[i,j], if T[i,j] = T[i-I,j] then no object of'i' is required so move up to T[i-I,j].

After moved, we have to check if, T[i,j]=T[i-I,j-w(i)]+ p[I], if it is equal thenone unit of object 'i' is selected and move up to the position T[i-I,j-w(i)]

Repeat the same process until we reach T[i,o], then there will be nothing tofill the bag stop the process.

Time is 0(nw) is necessary to construct the table T.

Consider a Example,

M=6,

N = 3

W_1=2,W_2=3,W_3=4

P_1=1,P_2=2,P_3=5

$i \longrightarrow$ 1 to N

$j \longrightarrow$ 0 to 6

i=l, j=o (ic) i=l & j < w(i)

o<2 $\longrightarrow$ T_1,o =0

i=l, j=l (ic) i=l & j < w(i)

l<2 $\longrightarrow$ T_1,1 =0 (Here j is equal to w(i) $\longrightarrow$ P(i)

i=l, j=2

2 o,= T_1,2 = l.

i=l, j=3

3>2,= T_1,3 = l.

i=l, j=4

4>2,= T_1,4 = l.

i=l, j=5

5>2,= T_1,5 = l.

i=l, j=6

6>2,= T_1,6 = l.

=> i=2, j=o (ic) i>l,j<w(i)

o<3= T(2,0) = T(i-l,j) = T(2)

T 2,0 =0

i=2, j=1

I<3= T(2,1) = T(i-I)

T 2,1 =0

UNIT-III

BASIC SEARCH AND TRAVERSAL TECHNIQUE:

3.1.DEFINING GRAPH:

A graphs g consists of a set V of vertices (nodes) and a set E of edges (arcs) . We write G=(V,E). V is a finite and non-empty set of vertices. E is a set of pair of vertices; these pairs are called as edges . Therefore,

V(G).read as V of G, is a set of vertices and E(G),read as E of G is a set of edges.An

edge e=(v, w) is a pair of vertices v and w, and to be incident with v and w.

A graph can be pictorially represented as follows,

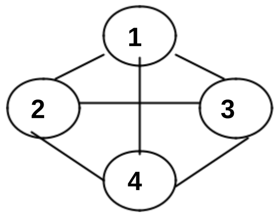


FIG: Graph G

We have numbered the graph as 1,2,3,4. Therefore, V(G)=(1,2,3,4) and E(G) =

{(1,2),(1,3),(1,4),(2,3),(2,4)}.

BASIC TERMINOLGIES OF GRAPH:

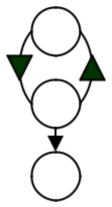
UNDIRECTED GRAPH:

An undirected graph is that in which, the pair of vertices representing the edges is unordered.

DIRECTED GRAPH:

An directed graph is that in which, each edge is an ordered pair of vertices, (i.e.) each edge is represented by a directed pair. It is also referred to as digraph.

DIRECTED GRAPH



COMPLETE GRAPH:

An n vertex undirected graph with exactly n(n-1)/2 edges is saidto be complete graph. The graph G is said to be complete graph .

TECHNIQUES FOR GRAPHS:

The fundamental problem concerning graphs is the reach-ability problem.

In it simplest from it requires us to determine whether there exist a path in the given graph, G +(V,E) such that this path starts at vertex ‘v’ and ends atvertex ‘u’.

A more general form is to determine for a given starting vertex v6 V all vertex ‘u’ such that there is a path from if it u.

This problem can be solved by starting at vertex ‘v’ and systematically searching the graph ‘G’ for vertex that can be reached from ‘v’.

We describe 2 search methods for this.

- o Breadth first Search and Traversal.
- o Depth first Search and Traversal.

BREADTH FIRST SEARCH AND TRAVERSAL:

Breadth first search:

In Breadth first search we start at vertex v and mark it as having been reached. The vertex v at this time is said to be unexplored. A vertex is said to have been explored by an algorithm when the algorithm has visited all vertices adjacent from it. All unvisited vertices adjacent from v are visited next. There are new unexplored vertices. Vertex v has now been explored. The newly visited vertices have not been explored and are put onto the end of the list of unexplored vertices.The first vertex on this list is the next to be explored. Exploration continues until no unexplored vertex is left. The list of unexplored vertices acts as a queue and can be represented using any of the standard queue representations.

In Breadth First Search we start at a vertex ‘v’ and mark it as having been reached (visited).

The vertex ‘v’ is at this time said to be unexplored.

A vertex is said to have been explored by an algorithm when the algorithmhas visited all vertices adjust from it.

All unvisited vertices adjust from ‘v’ are visited next. These are new unexplored vertices.

Vertex ‘v’ has now been explored. The newly visit vertices have not been explored and are put on the end of a list of unexplored vertices.

The first vertex on this list in the next to be explored. Exploration continuesuntil no unexplored vertex is left.

The list of unexplored vertices operates as a queue and can be representedusing any of the start queue representation.

ALGORITHM:

Algorithm BPS (v)

```
// A breadth first search of 'G' is carried out.
//  beginning at vertex-v; For any node i, visit.
//  if 'i' has already been visited. The graph 'v'
// and array visited [] are global; visited []
// initialized to zero.
{ y=v; // q is a queue of unexplored 1visited (v)= 1
repeat
{ for all vertices 'w' adjacent from u do {if
     (visited[w]=0) then
        {Add w to q;
          visited[w]=1
            }
  }
if q is empty then return;// No delete u from q; }
       until (false)
}
```

```
algrothim : breadth first traversal
        algorithm BFT(G,n)

  {

        for i= 1 to n do
            visited[i] =0;
        for i =1 to n do
       if (visited[i]=0)then BFS(i)
   }
```

here the time and space required by BFT on an n-vertex e-edge graph one O(n+e) and O(n) resp if adjacency list is used.if adjancey matrix is used then the bounds are O(n^2) and O(n) resp

Depth first search

A depth first search of a graph differs from a breadth first search in that the exploration of a vertex v is suspended as soon as a new vertex is reached. At this time the exploration of the new vertex u begins. When this new vertex has been explored, the exploration of u continues. The search terminates when all reached vertices have been fully explored. This search process is best-described recursively.

```
Algorithm DFS(v)
{
visited[v]=1
for each vertex w adjacent from v do
{
If (visited[w]=0)then
DFS(w);
}
}
```

BACKTRACKING

BACKTRACKING

It is one of the most general algorithm design techniques.

Many problems which deal with searching for a set of solutions or for aoptimal solution satisfying some constraints can be solved using the backtracking formulation.

To apply backtracking method, tne desired solution must be expressible asan n-tuple (x1…xn) where xi is chosen from some finite set Si.

The problem is to find a vector, which maximizes or minimizes a criterion function P(x1….xn).

The major advantage of this method is, once we know that a partial vector (x1,…xi) will not lead to an optimal solution that (m_{i+1} .... m_n) possible test vectors may be ignored entirely.

Many problems solved using backtracking require that all the solutionssatisfy a complex set of constraints.

These constraints are classified as:

i) Explicit constraints.

ii) Implicit constraints.

Explicit constraints:

Explicit constraints are rules that restrict each Xi to take values onlyfrom a given set.

Some examples are, Xi

0 or Si = {all non-negative real nos.}

Xi =0 or 1 or Si={0,1}.

Li Xi Ui or Si= {a: Li a Ui}

All tupules that satisfy the explicit constraint define a possible solutionspace for I.

Implicit constraints:

The implicit constraint determines which of the tuples in the solution space I can actually satisfy the criterion functions.

Algorithm:

```
Algorithm IBacktracking (n)
//    This schema describes the backtracking procedure .All solutions are
generated in X[1:n]
//and printed as soon as they are determined.
 {
    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;                        //consider the next step.
     }
  else k=k-1;                              //consider backtracking to the previous set.
 }
}
```

All solutions are generated in X[1:n] and printed as soon as they are determined.

T(X[1]…..X[k-1]) is all possible values of X[k] gives that X[1],................ X[k-1] have already been chosen.

B_k(X[1]........... X[k]) is a boundary function which determines the elements of X[k] which satisfies the implicit constraint.

Certain problems which are solved using backtracking method are,

1. Sum of subsets.
2. Graph coloring.
3. Hamiltonian cycle.
4. N-Queens problem.

SUM OF SUBSETS:

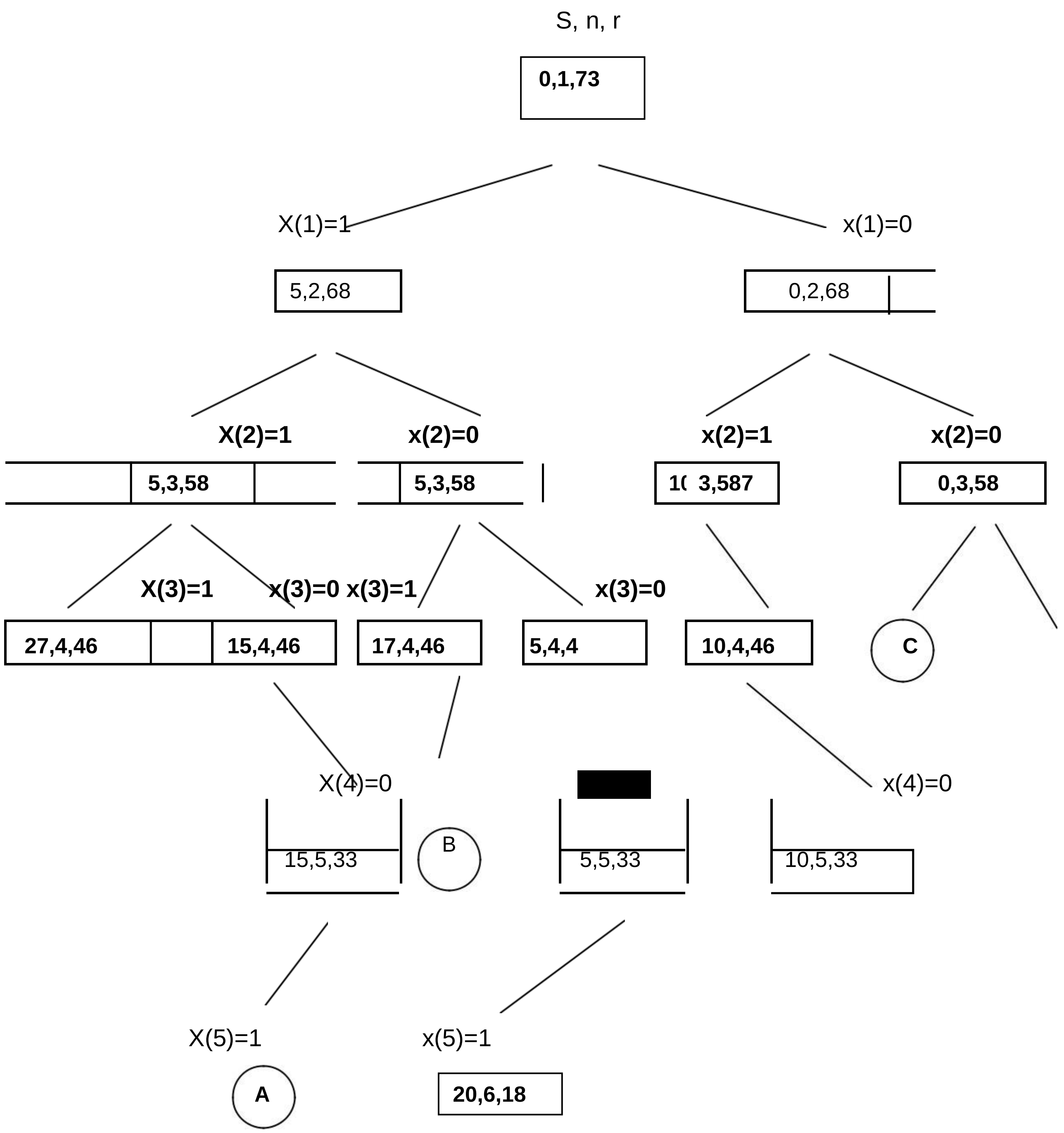
1) We are given 'n' positive numbers called weights and we have to find all combinations of these numbers whose sum is M. this is called sum of subsets problem.
2) If we consider backtracking procedure using fixed tuple strategy , the elements X(i) of the solution vector is either 1 or 0 depending on if the weight W(i) is included or not.
3) If the state space tree of the solution, for a node at level I, the left child corresponds to X(i)=1 and right to X(i)=0.

Example:

a. Given n=6,M=30 and W(1…6)=(5,10,12,13,15,18).We have to generate all possible combinations of subsets whose sum is equal to the given value M=30.

b. In state space tree of the solution the rectangular node lists the values of s, k, r, where s is the sum of subsets,’k’ is the iteration and‘r’ is the sum of elements after ‘k’ in the original set.

c. The state space tree for the given problem is,



Ist	solution is	A ->1	1 0 0 1 0
IInd	solution is	B ->1	0 1 1 0 0
III rd	solution is	C -> 0	0 1 0 0 1

In the state space tree, edges from level 'i' nodes to 'i+1' nodes are labeledwith the values of Xi, which is either 0 or 1.

The left sub tree of the root defines all subsets containing Wi.

The right subtree of the root defines all subsets, which does not includeWi.

GENERATION OF STATE SPACE TREE:

Maintain an array X to represent all elements in the set.

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

Sum is initialized to 0 i.e., s=0.

We have to check starting from the first node.

Assign X(k)<- 1.

If S+X(k)=M then we print the subset b'coz the sum is the required output.

If the above condition is not satisfied then we have to check S+X(k)+W(k+1)<=M. If so, we have to generate the left sub tree. It means W(t) can be included so the sum will be incremented and we have to checkfor the next k.

After generating the left sub tree we have to generate the right sub tree, forthis we have to check S+W(k+1)<=M.B'coz W(k) is omitted and W(k+1) has to be selected.

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

Algorithm:

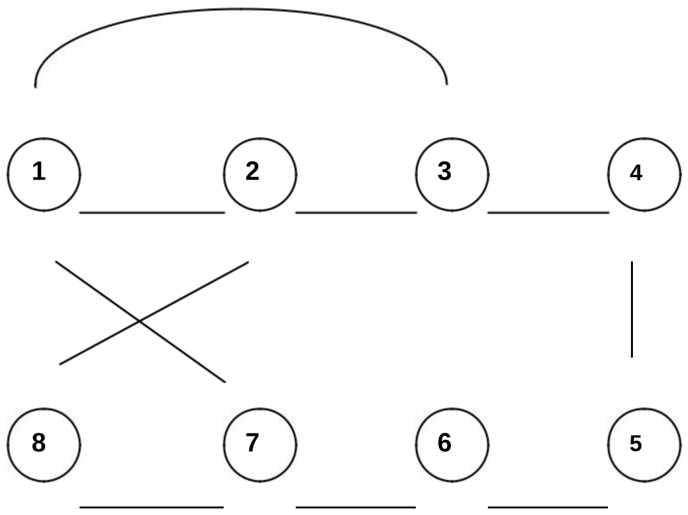
```
Algorithm sumofsubset(s,k,r)
{
//generate the left child. note s+w(k)<=M since Bk-1 is true.
X{k]=1;
If (S+W[k]=m) then write(X[1:k]); // there is no recursive call here as
W[j]>0,1<=j<=n.
Else if (S+W[k]+W[k+1]<=m) then sum of sub (S+W[k], k+1,r- W[k]);
//generate right child and evaluate Bk.
If ((S+ r- W[k]>=m)and(S+ W[k+1]<=m)) then
{
   X{k]=0;
   sum of sub (S, k+1, r- W[k]);
}
}
```

HAMILTONIAN CYCLES:

Let G=(V,E) be a connected graph with ‘n’ vertices. A HAMILTONIAN CYCLE is a round trip path along ‘n’ edges of G which every vertex onceand returns to its starting position.

If the Hamiltonian cycle begins at some vertex V1 belongs to G and thevertex are visited in the order of V1,V2…….Vn+1,then the edges are in E,1<=I<=n and the Vi are distinct except V1 and Vn+1 which are equal.

Consider an example graph G1.



The graph G1 has Hamiltonian cycles:

->1,3,4,5,6,7,8,2,1 and

->1,2,8,7,6,5,4,3,1.

The backtracking algorithm helps to find Hamiltonian cycle for any type ofgraph.

Procedure:

Define a solution vector X(Xi ....Xn) where Xi represents the I th visited vertex of the proposed cycle.

Create a cost adjacency matrix for the given graph.

The solution array initialized to all zeros except X(1)=1,b'coz the cycleshould start at vertex '1'.

Now we have to find the second vertex to be visited in the cycle.

The vertex from 1 to n are included in the cycle one by one by checking 2 conditions,

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

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

6. When these two conditions are satisfied the current vertex is included inthe cycle, else the next vertex is tried.

7. When the nth vertex is visited we have to check, is there any path from nth vertex to first 8vertex. if no path, the go back one step and after the previous visited node.

8. Repeat the above steps to generate possible Hamiltonian cycle.

Algorithm:(Finding all 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); //next vertexIf
  (X [k]=0) then return;
  If (G [X [k-1], X [k]]          0) then
{
  For j=1 to k-1 do if (X [j]=X [k]) then break; //
  Check for distinction.
  If (j=k) then             //if true then the vertex is distinct.
    If ((k<n) or ((k=n) and G [X [n], X [1]]               0)) then return;
```

}

} Until (false);

}

3.7.8-QUEENS PROBLEM:

This 8 queens problem is to place n-queens in an ‘N*N’ matrix in such a way that no two queens attack each otherwise no two queens should be in the same row, column, diagonal.

Solution:

The solution vector X (X1…Xn) represents a solution in which Xi is the column of the th row where I th queen is placed.

First, we have to check no two queens are in same row.

Second, we have to check no two queens are in same column.

The function, which is used to check these two conditions, is [I, X (j)], which gives position of the I th queen, where I represents the row and X (j)represents the column position.

Third, we have to check no two queens are in it diagonal.

Consider two dimensional array A[1:n,1:n] in which we observe that every element on the same diagonal that runs from upper left to lower right has the same value.

Also, every element on the same diagonal that runs from lower right toupper left has the same value.

Suppose two queens are in same position (i,j) and (k,l) then two queens lieon the same diagonal , if and only if |j-l|=|I-k|.

3.7.1.STEPS TO GENERATE THE SOLUTION:

Initialize x array to zero and start by placing the first queen in k=1 in thefirst row.

To find the column position start from value 1 to n, where 'n' is the no. Of columns or no. Of queens.

If k=1 then x (k)=1.so (k,x(k)) will give the position of the k th queen. Herewe have to check whether there is any queen in the same column or diagonal.

For this considers the previous position, which had already, been foundout. Check whether

X (I)=X(k) for column |X(i)-X(k)|=(I-k) for the same diagonal.

If any one of the conditions is true then return false indicating that k thqueen can't be placed in position X (k).

For not possible condition increment X (k) value by one and precede d untilthe position is found.

If the position X (k) n and k=n then the solution is generated completely.

If k<n, then increment the 'k' value and find position of the next queen.

If the position X (k)>n then k th queen cannot be placed as the size of thematrix is 'N*N'.

So decrement the 'k' value by one i.e. we have to back track and after the position of the previous queen.

Algorithm:

Algorithm place (k,I)

//return true if a queen can be placed in k th row and I th column. otherwise itreturns //

```
//false .X[] is a global array whose first k-1 values have been set. Abs® returns the
//absolute value of r.
{
  For j=1 to k-1 do
     If ((X [j]=I)                    //two in same column.
     Or (abs (X [j]-I)=Abs (j-k)))
Then return false;
Return true;
}
```

```
Algorithm Nqueen (k,n)
//using backtracking it prints all possible positions of n queens in 'n*n'chessboard. So
 //that they are non-tracking.
{
     For I=1 to n do
         {
            If place (k,I) then
              {
                 X [k]=I;
                  If (k=n) then write (X [1:n]);
                   Else nquenns(k+1,n)        ;
           }
      }
}
```

Example: 4 queens.

Two possible solutions are

	Q		
			Q
Q			
		Q	

		Q	
Q			
			Q
	Q		

Solutin-1 Solution 2

(2413) (3142)

GRAPH COLORING:

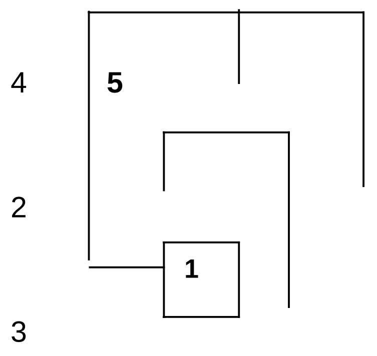
Let ‘G’ be a graph and ‘m’ be a given positive integer. If the nodes of ‘G’can be colored in such a way that no two adjacent nodes have the samecolor. Yet only ‘M’ colors are used. So it’s called M-color ability decisionproblem.

The graph G can be colored using the smallest integer ‘m’. This integer is referred to as chromatic number of the graph.

A graph is said to be planar iff it can be drawn on plane in such a way thatno two edges cross each other.

Suppose we are given a map then, we have to convert it into planar. Consider each and every region as a node. If two regions are adjacent thenthe corresponding nodes are joined by an edge.

Consider a map with five regions and its graph.



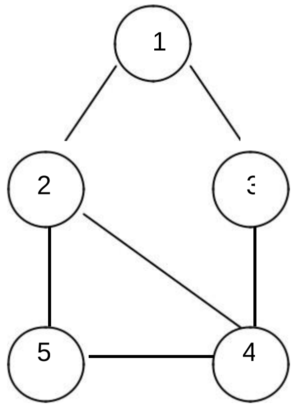
1 is adjacent to 2, 3, 4.

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 the Graph:

1. First create the adjacency matrix graph(1:m,1:n) for a graph, if there is anedge between i,j then C(i,j) = 1 otherwise C(i,j) =0.

2. The Colors will be represented by the integers 1,2,…..m and the solutionswill be stored in the array X(1),X(2),.......................,X(n) ,X(index) is the color, index is the node.

3. He formula is used to set the color is,

   X(k) = (X(k)+1) % (m+1)

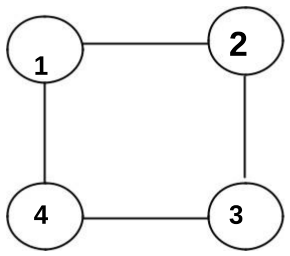
4. First one chromatic number is assigned ,after assigning a number for ‘k’ node, we have to check whether the adjacent nodes has got the same values if so then we have to assign the next value.

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

6. The function which is used to check the adjacent nodes and same color is,If((

Graph (k,j) == 1) and X(k) = X(j))

Example:



N= 4

M= 3

Adjacency Matrix:

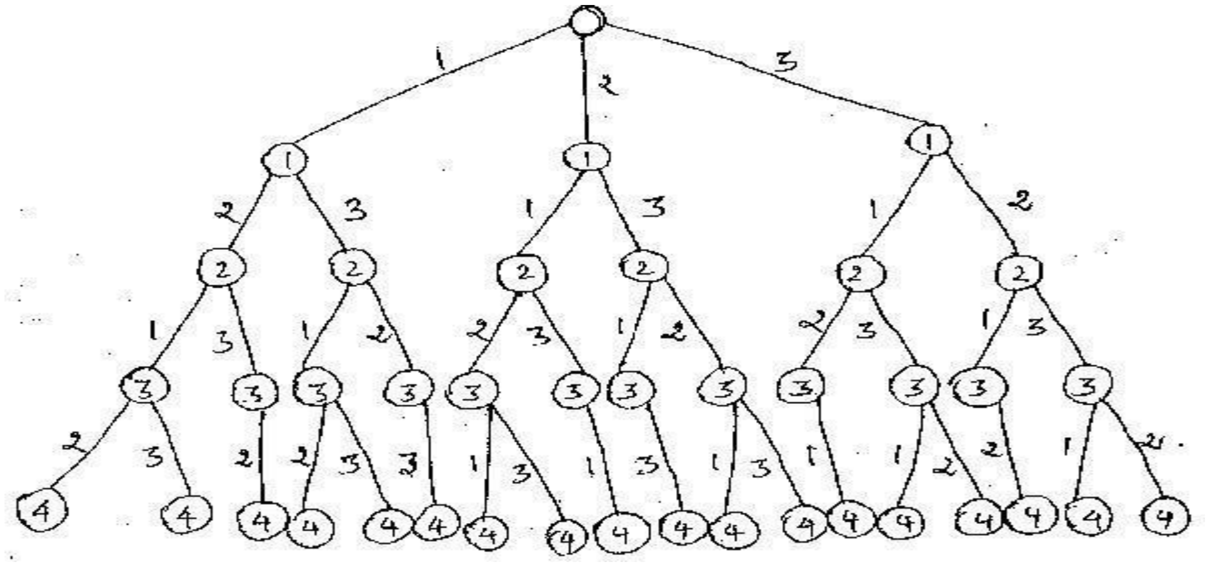
$$\begin{vmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{vmatrix}$$

Problem is to color the given graph of 4 nodes using 3 colors.

Node-1 can take the given graph of 4 nodes using 3 colors.

The state space tree will give all possible colors in that ,the numbers which are inside the circles are nodes ,and the branch with a number is the colors of the nodes.

State Space Tree:



Algorithm:

Algorithm mColoring(k)

// the graph is represented by its Boolean adjacency matrix G[1:n,1:n] .All assignments //of 1,2,……….,m to the vertices of the graph such that adjacent vertices are assigned //distinct integers are printed. 'k' is the index of the next vertex to color.

```
{
repeat
{
    // generate all legal assignment for X[k].
  Nextvalue(k);        // Assign to X[k] a legal color.
         If (X[k]=0) then return;                    // No new color possible.
        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);
}
```

Algorithm Nextvalue(k)

// X[1],……X[k-1] have been assigned integer values in the range[1,m] such that //adjacent values have distinct integers. A value for X[k] is determined in the //range[0,m].X[k] is assigned the next highest numbers color while maintaining //distinctness form the adjacent vertices of vertex K. If no such color exists, thenX[k] is 0.

```
{

  repeat
  {
          X[k] = (X[k]+1)mod(m+1);            // next highest color.
          If(X[k]=0) then return;              //All colors have been used.
           For j=1 to n do
          {
             // Check if this color is distinct from adjacent color.
          If((G[k,j] 0)and(X[k] = X[j]))
             // If (k,j) is an edge and if adjacent vertices have the same color.
          Then break;
          }

        if(j=n+1) then return;         //new color found.
    } until(false);       //otherwise try to find another color.
}
```

The time spent by Nextvalue to determine the children is (mn)

Total time is = $(m^n n)$.

KNAPSACK PROBLEM USING BACKTRACKING:

The problem is similar to the zero-one (0/1) knapsack optimization problemis dynamic programming algorithm.

We are given ‘n’ positive weights Wi and ’n’ positive profits Pi, and a positive number ‘m’ that is the knapsack capacity, the is problem calls forchoosing a subset of the weights such that,

$\underset{1\,i\,n}{WiXi}$ m and $\underset{1\,i\,n}{PiXi}$ is Maximized.

Xi Constitute Zero-one valued Vector.

The Solution space is the same as that for the sum of subset’s problem.

Bounding functions are needed to help kill some live nodes without expanding them. A good bounding function for this problem is obtained byusing an upper bound on the value of the best feasible solution obtainableby expanding the given live node.

The profits and weights are assigned in descending order depend upon theratio.

(i.e.) Pi/Wi P(I+1) / W(I+1)

Solution :

After assigning the profit and weights ,we have to take the first object weights and check if the first weight is less than or equal to the capacity, ifso then we include that object (i.e.) the unit is 1.(i.e.) K 1.

Then We are going to the next object, if the object weight is exceeded that object does not fit. So unit of that object is ‘0’.(i.e.) K=0.

Then We are going to the bounding function ,this function determines anupper bound on the best solution obtainable at level K+1.

Repeat the process until we reach the optimal solution.

Algorithm:

```
Algorithm Bknap(k,cp,cw)

//  ‘m’ is the size of the knapsack; ‘n’ no.of weights & profits. W[]&P[] are the
//weights & weights. P[I]/W[I] P[I+1]/W[I+1].
//fw     Final weights of knapsack.
//fp      final max.profit.
//x[k] = 0 if W[k] is not the knapsack,else X[k]=1.

{
        //  Generate left child.
         If((W+W[k]  m) then
         {
                 Y[k] =1;
                  If(k<n) then Bnap(k+1,cp+P[k],Cw +W[k])
                     If((Cp + p[w] > fp) and (k=n)) then
```

```
                    {
                       fp = cp + P[k];
                       fw = Cw+W[k];
                      for j=1 to k do X[j] = Y[j];
                  }
    }

  if(Bound(cp,cw,k)        fp) then
  {
         y[k] = 0;
       if(k<n) then Bnap (K+1,cp,cw);
     if((cp>fp) and (k=n)) then
        {
              fp = cp;
               fw = cw;
                 for j=1 to k do X[j] = Y[j];
           }
    }
}
```

Algorithm for Bounding function:

Algorithm Bound(cp,cw,k)

// cp current profit total. //cw

current weight total.

//k the index of the last removed item. //m

the knapsack size.

```
{
    b=cp;
    c=cw;
    for I =- k+1 to n do
  {
        c= c+w[I];
      if (c<m) then b=b+p[I];
          else return b+ (1-(c-m)/W[I]) * P[I];
}
return b;
}
```

Example:

M= 6 Wi = 2,3,4   4 2 2

N= 3 Pi = 1,2,5   Pi/Wi (i.e.)   5 2 1

Xi = 1 0 1

The maximum weight is 6

The Maximum profit is (1*5) + (0*2) + (1*1)

5+1

6.

Fp = (-1)

1 3&0+4 6

cw = 4,cp = 5,y(1) =1

k = k+2

2 3 but 7>6 so

y(2) = 0

So bound(5,4,2,6)

B=5

C=4

I=3 to 3

C=6

6 6

So return 5+(1-(6-6))/(2*1)

5.5 is not less than fp.

So, k=k+1 (i.e.) 3.

3=3 & 4+2 6

cw= 6,cp = 6, y(3)=1.

K=4.

If 4> 3 then

Fp =6,fw=6,k=3 ,x(1) 1 0 1

The solution Xi 1 0 1

Profit 6

Weight 6.

UNIT – IV

BRANCH AND BOUND -- THE METHOD

The design technique known as branch and bound is very similar to backtracking (seen in unit 4) in that it searches a tree model of the solution space and is applicable to a wide variety of discrete combinatorial problems.

Each node in the combinatorial tree generated in the last Unit defines a problem state. All paths from the root to other nodes define the state space of the problem.

Solution states are those problem states 's' for which the path from the root to 's' defines a tuple in the solution space. The leaf nodes in the combinatorial tree are the solution states.

Answer states are those solution states 's' for which the path from the root to 's' defines a tuple that is a member of the set of solutions (i.e.,it satisfies the implicit constraints) of the problem.

The tree organization of the solution space is referred to as the state space tree.

A node which has been generated and all of whose children have not yet been generated is called a live node.

The live node whose children are currently being generated is called the E- node (node being expanded).

A dead node is a generated node, which is not to be expanded further or all of whose children have been generated.

Bounding functions are used to kill live nodes without generating all their children.

Depth first node generation with bounding function is called backtracking. State generation methods in which the E-node remains the E-node until it is dead lead to branch-and-bound method.

The term branch-and-bound refers to all state space search methods in which all children of the E-node are generated before any other live node can become the E-node.

In branch-and-bound terminology breadth first search(BFS)- like state space search will be called FIFO (First In First Output) search as the list of live nodes is a first -in-first -out list(or queue).

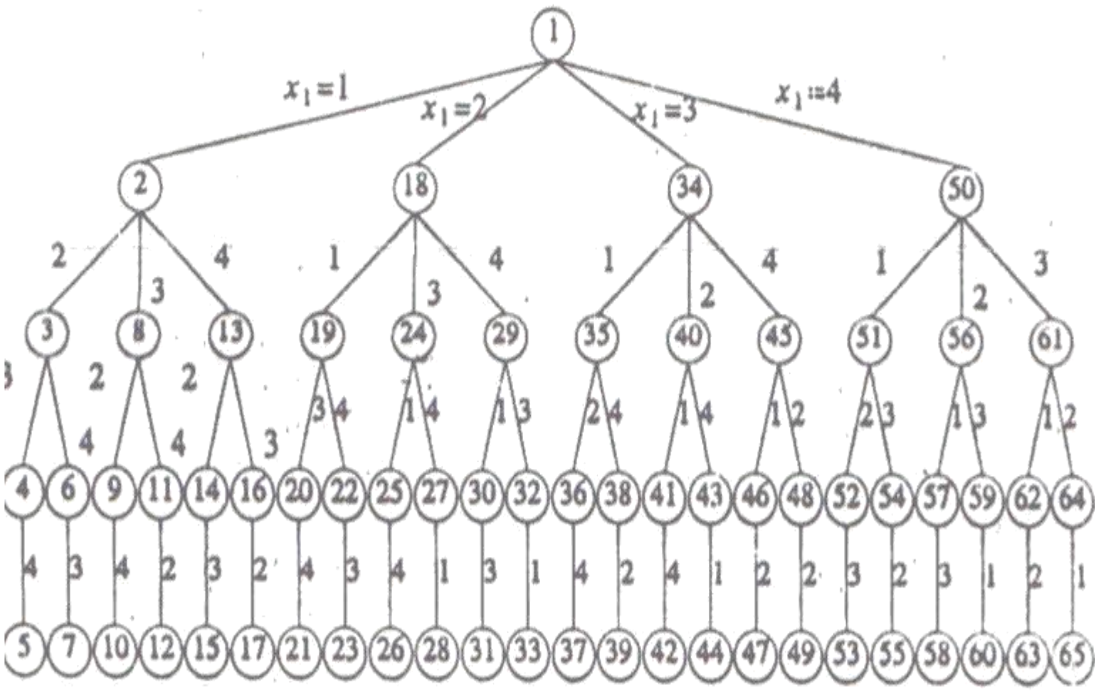
A D-search (depth search) state space search will be called LIFO (Last In First Out) search, as the list of live nodes is a list-in-first-out list (or stack).

Bounding functions are used to help avoid the generation of sub trees that do not contain an answer node.

The branch-and-bound algorithms search a tree model of the solution space to get the solution. However, this type of algorithms is oriented more toward optimization. An algorithm of this type specifies a real -valued cost function for each of the nodes that appear in the search tree.

Usually, the goal here is to find a configuration for which the cost function is minimized. The branch-and-bound algorithms are rarely simple. They tend to be quite complicated in many cases.

Example 8.1[4-queens] Let us see how a FIFO branch-and-bound algorithm would search the state space tree (figure 7.2) for the 4-queens problem.



Initially, there is only one live node, node1. This represents the case in which no queen has been placed on the chessboard. This node becomes the E-node.

It is expanded and its children, nodes2, 18, 34 and 50 are generated.

These nodes represent a chessboard with queen1 in row 1and columns 1, 2, 3, and 4 respectively.

The only live nodes 2, 18, 34, and 50.If the nodes are generated in this order, then the next E-node are node 2.

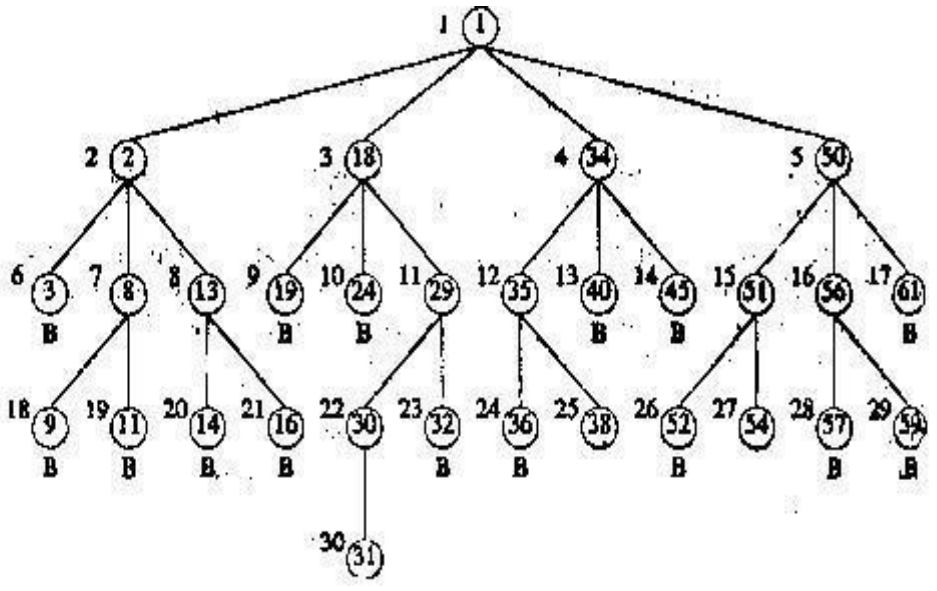
It is expanded and the nodes 3, 8, and 13 are generated. Node 3 is immediately killed using the bounding function. Nodes 8 and 13 are added to the queue of live nodes.

Node 18 becomes the next E -node. Nodes 19, 24, and 29 are generated. Nodes19 and 24 are killed as a result of the bounding functions. Node 29 is added to the queue of live nodes.

Now the E- node is node 34.Figure 8.1 shows the portion of the tree of Figure that is generated by a FIFO branch -and-bound search. Nodes that are killed asa result of the bounding functions are a "B" under them.

Numbers inside the nodes correspond to the numbers in Figure 7.2.Numbers outside the nodes give the order in which the nodes are generated by FIFObranch-and-bound.

At the time the answer node, node 31, is reached, the only live nodes remaining are nodes 38 and 54.



Least Cost (LC) Search:

In both LIFO and FIFO branch-and-bound the selection rule for the next E-node is rather rigid and in a sense blind. The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.

Thus, in Example 8.1, when node 30 is generated, it should have become obvious to the search algorithm that this node will lead to answer node in one move. However, the rigid FIFO rule first requires the expansion of all live nodes generated before node 30 was expanded.

The search for an answer node can often be speeded by using an "intelligent" ranking function $\hat{c}$ (.) for live nodes. The next E-node is selected on the basis of this ranking function.

If in the 4-queens example we use a ranking function that assigns node 30 a better rank than all other live nodes, then node 30 will become E -node, following node 29.The remaining live nodes will never become E-nodes as the expansion of node 30 results in the generation of an answer node (node 31).

The ideal way to assign ranks would be on the basis of the additional computational effort (or cost) needed to reach an answer node from the live node.
For any node x, this cost could be

(1) The number of nodes on the sub-tree x that need to be generated before any answer node is generated or, more simply,

(2) The number of levels the nearest answer node (in the sub-tree x) isfrom x

Using cost measure (2), the cost of the root of the tree of Figure 8.1 is 4 (node 31 is four levels from node 1).The costs of nodes 18 and 34,29 and 35,and 30 and 38 are respectively 3, 2, and 1.The costs of all remaining nodes on levels 2, 3, and 4 are respectively greater than 3, 2, and 1.

Using these costs as a basis to select the next E-node, the E-nodes are nodes 1,18, 29, and 30 (in that order).The only other nodes to get generated are nodes 2,34, 50, 19, 24, 32, and 31.

The difficulty of using the ideal cost function is that computing the cost of a node usually involves a search of the sub-tree x for an answer node. Hence, bythe time the cost of a node is determined, that sub-tree has been searched and there is no need to explore x again. For this reason, search
algorithms usually rank nodes only based on an estimate $\hat{g}$ (.) of their cost.

Let $\hat{g}$ (x) be an estimate of the additional effort needed to reach an answer node from x. node x is assigned a rank using a function (.) such that $\hat{c}$ (x) =f
(h(x)) + $\hat{g}$ (x), where h(x) is the cost of reaching x from the root and f(.) is any non-decreasing function.

A search strategy that uses a cost function $\hat{c}$(x) =f (h(x)) + $\hat{g}$ (x), to select the next e-node would always choose for its next e-node a live node with least (.).Hence, such a strategy is called an LC-search (least cost search).

Cost function c (.) is defined as, if x is an answer node, then c(x) is the cost (level, computational difficulty, etc.) of reaching x from the root of the state space tree. If x is not an answer node, then c(x) =infinity, providing the sub-tree x contains no answer node; otherwise c(x) is equals the cost of a minimum cost answer node in the sub-tree x.

It should be easy to see that $\hat{c}$ (.) with f (h(x)) =h(x) is an approximation to c (.).
From now on (x) is referred to as the cost of x.

Bounding:

A branch -and-bound searches the state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node.

We assume that each answer node x has a cost c(x) associated with it and that a minimum-cost answer node is to be found. Three common search strategiesare FIFO, LIFO, and LC.

A cost function $\hat{c}$ (.) such that $\hat{c}$ (x) <=c(x) is used to provide lower bounds on solutions obtainable from any node x. If upper is an upper bound on the cost of a minimum-cost solution, then all live nodes x with $\hat{c}$ (x)>upper may be killed as all answer nodes reachable from x have cost c(x)>= $\hat{c}$ (x)>upper. The starting value for upper can be set to infinity.

Clearly, so long as the initial value for upper is no less than the cost of a minimum-cost answer node, the above rule to kill live nodes will not result in the killing of a live node that can reach a minimum-cost answer node
.Each time a new answer is found ,the value of upper can be updated.

As an example optimization problem, consider the problem of job scheduling with deadlines. We generalize this problem to allow jobs with different processing times. We are given n jobs and one processor. Each job i has associated with it a three tuple

(p_i d_i t_i).job i requires t_i units of processing time .if its processing is not completed by the deadline d_i , and then a penalty p_i is incurred.

The objective is to select a subset j of the n jobs such that all jobs in j can be completed by their deadlines. Hence, a penalty can be incurred only on those jobs not in j. The subset j should be such that the penalty incurred is minimum among all possible subsets j. such a j is optimal.

Consider the following instances: n=4,(, d_1, t_1)=(5,1,1),(p_2 , d_2 , t_2)=(10,3,2),(p_3 , d_3 , t_3)=(6,2,1),and(p_4, d_4, t_4)=(3,1,1).The solution space for this instances consists of all possible subsets of the job index set{1,2,3,4}. The solution space can be organized into a tree by means of either of the two formulations used for the sum of subsets problem.

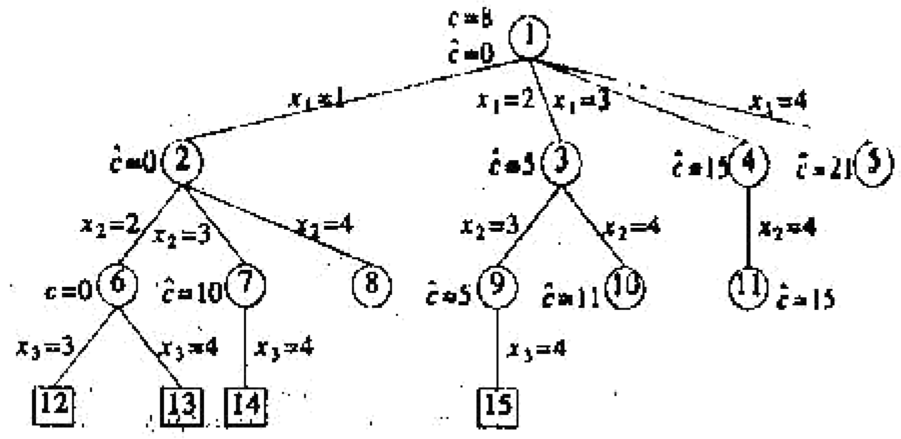


Figure 8.6 State space tree corresponding to variable tuple size formulation

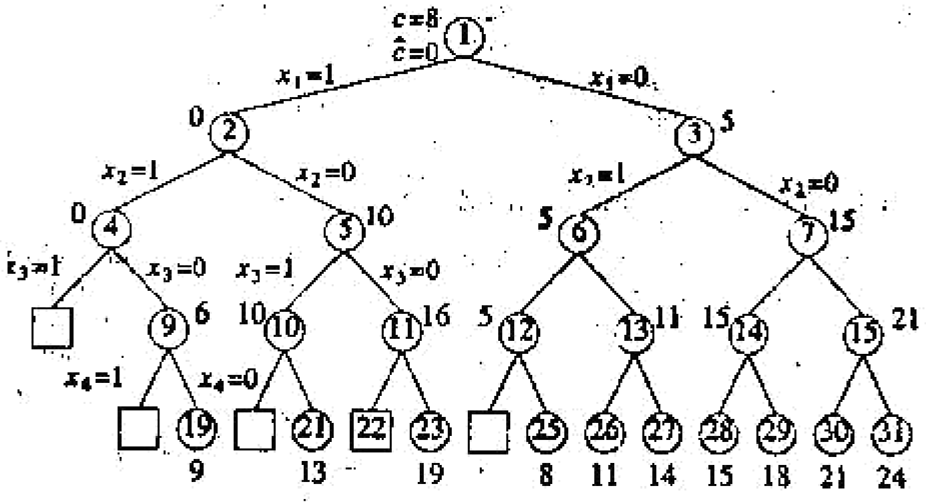


Figure 8.7

Figure 8.6 corresponds to the variable tuple size formulations while figure 8.7 corresponds to the fixed tuple size formulation. In both figures square nodes represent infeasible subsets. In figure 8.6 all non-square nodes are answer nodes. Node 9 represents an optimal solution and is the only minimum-cost answer node
.For this node j= {2,3} and the penalty (cost) is
8. In figure 8.7 only non-square leaf nodes are answer nodes. Node 25 represents the optimal solution and is also a minimum-cost answer node.

This node corresponds to j={2,3} and a penalty of 8. The costs of the answer nodes of figure 8.7 are given below the nodes.

TRAVELLING SALESMAN PROBLEM

It is algorithmic procedures similar to backtracking in which a new branch is chosen and is there (bound there) until new branch is choosing for advancing.

This technique is implemented in the traveling salesman problem [TSP] which are asymmetric (Cij <>Cij) where this technique is an effective procedure.

STEPS INVOLVED IN THIS PROCEDURE ARE AS FOLLOWS:

STEP 0: Generate cost matrix C [for the given graph g]

STEP 1: [ROW REDUCTION]

For all rows do step 2

STEP: Find least cost in a row and negate it with rest of the elements.

STEP 3: [COLUMN REDUCTION]

Use cost matrix- Row reduced one for all columns do STEP 4.

STEP 4: Find least cost in a column and negate it with rest of the elements.

STEP 5: Preserve cost matrix C [which row reduced first and then column reduced] for the i th time.

STEP 6: Enlist all edges (i, j) having cost = 0.

STEP 7: Calculate effective cost of the edges. (i, j)=least cost in the i th rowexcluding (i, j) + least cost in the j th column excluding (i, j).

STEP 8: Compare all effective cost and pick up the largest l. If two or more havesame cost then arbitrarily choose any one among them.

STEP 9: Delete (i, j) means delete ith row and jth column change (j, i) value to infinity. (Used to avoid infinite loop formation) If (i,j) not present, leave it.

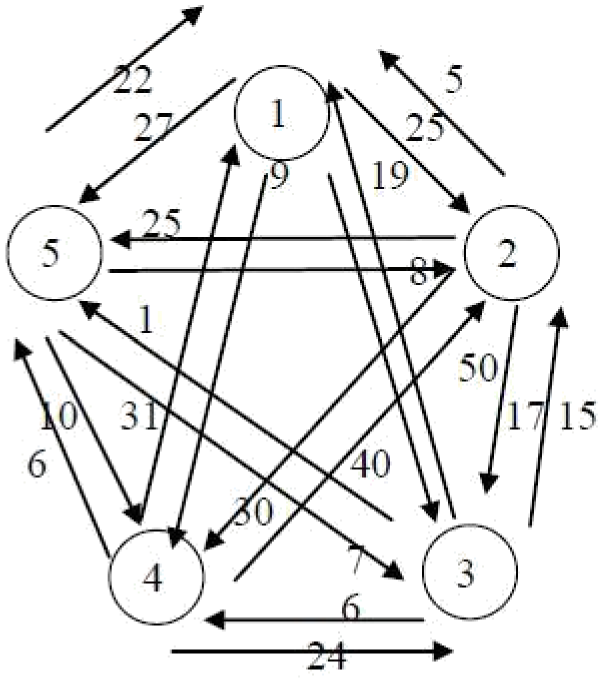
STEP 10: Repeat step 1 to step 9 until the resultant cost matrix having order of 2*2and reduce it. (Both R.R and C.C)

STEP 11: Use preserved cost matrix Cn, Cn-1… C1

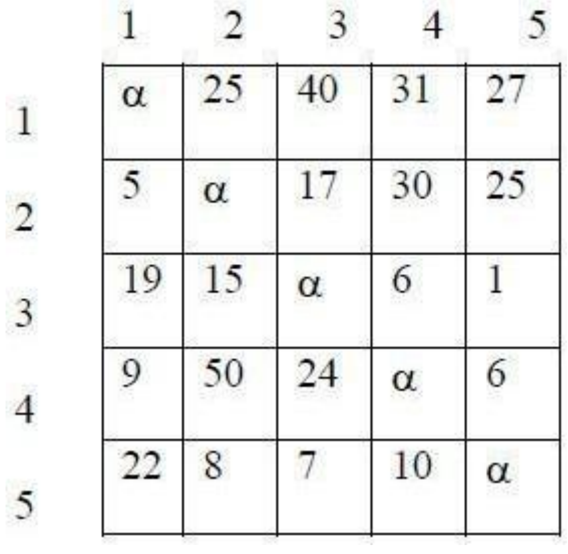
Choose an edge [i, j] having value =0, at the first time for a preserved matrix andleave that matrix.

STEP 12: Use result obtained in Step 11 to generate a complete tour.

EXAMPLE: Given graph G



MATRIX:

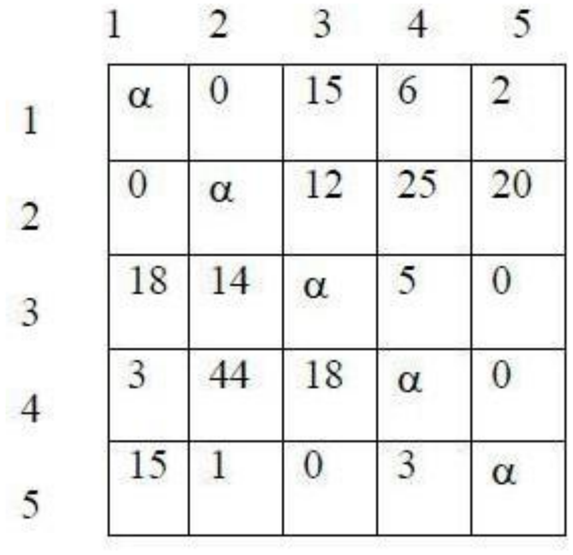


	1	2	3	4	5
1	α	25	40	31	27
2	5	α	17	30	25
3	19	15	α	6	1
4	9	50	24	α	6
5	22	8	7	10	α

PHASE I

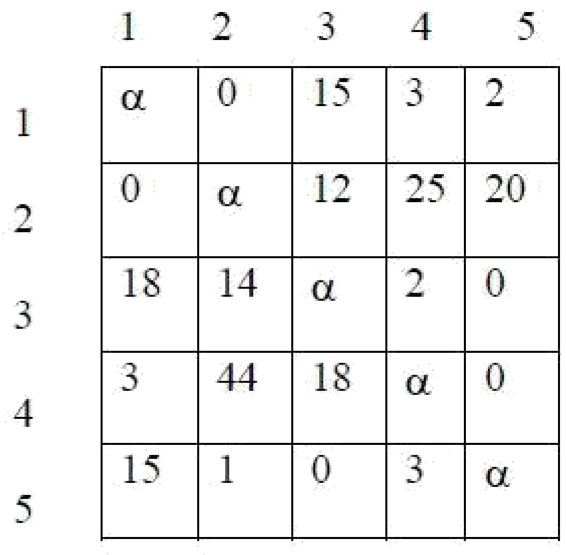
STEP 1: Row Reduction C

C1 [ROW REDUCTION:



	1	2	3	4	5
1	α	0	15	6	2
2	0	α	12	25	20
3	18	14	α	5	0
4	3	44	18	α	0
5	15	1	0	3	α

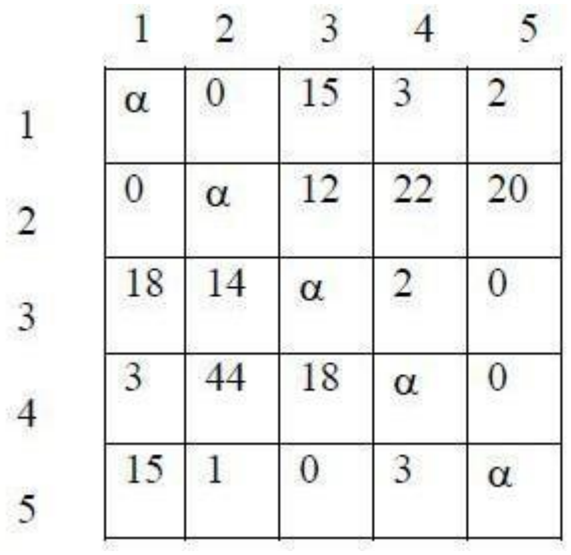
STEP 3: C1 [Column Reduction]



	1	2	3	4	5
1	α	0	15	3	2
2	0	α	12	25	20
3	18	14	α	2	0
4	3	44	18	α	0
5	15	1	0	3	α

STEP 5:

Preserve the above in C1,



	1	2	3	4	5
1	α	0	15	3	2
2	0	α	12	22	20
3	18	14	α	2	0
4	3	44	18	α	0
5	15	1	0	3	α

STEP 6:

L= (1,2), (2,1), (3,5), (4,5), (5,3), (5,4)

STEP 7:

Calculation of effective cost [E.C]

(1,2) = 2+1 =3

(2,1) = 12+3 = 15

(3,5) = 2+0 =2

(4,5) = 3+0 = 3

(5,3) = 0+12 = 12

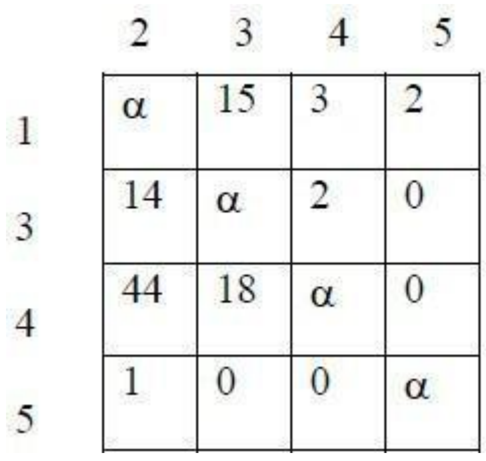
(5,4) = 0+2 = 2

STEP 8:

L having edge (2,1) is the largest.

STEP 9: Delete (2,1) from C1 and make change in it as (1,2) if exists.

Now Cost Matrix =



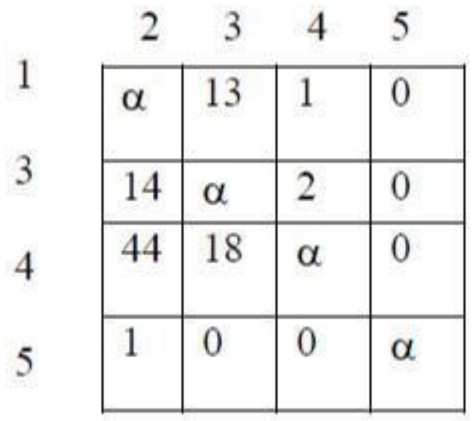
	2	3	4	5
1	α	15	3	2
3	14	α	2	0
4	44	18	α	0
5	1	0	0	α

STEP 10: The Cost matrix 2 x 2.

Therefore, go to step 1.

PHASE II:

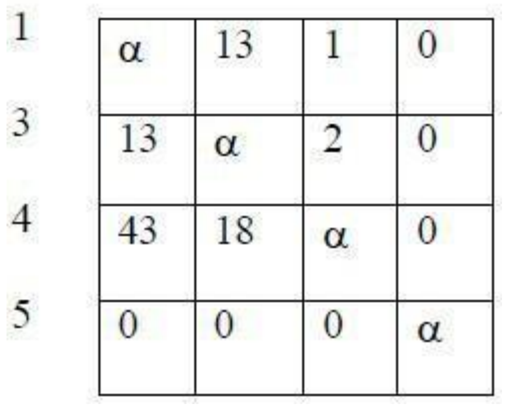
STEP1: C2(R, R)



	2	3	4	5
1	α	13	1	0
3	14	α	2	0
4	44	18	α	0
5	1	0	0	α

STEP 3: C2 (C, R)

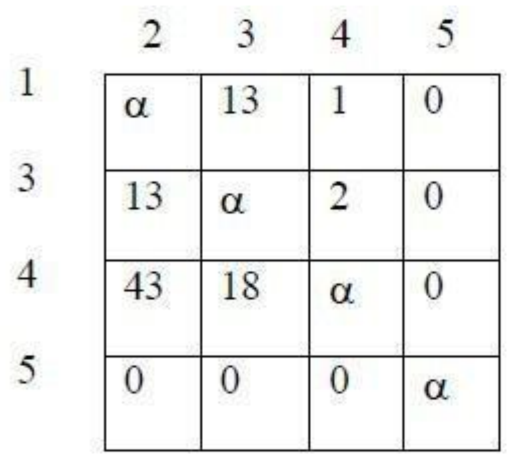
2 3 4 5



1	α	13	1	0
3	13	α	2	0
4	43	18	α	0
5	0	0	0	α

STEP 5: Preserve the above in C2

C2 =



	2	3	4	5
1	α	13	1	0
3	13	α	2	0
4	43	18	α	0
5	0	0	0	α

STEP 6:

L= {(1,5), (3,5), (4,5), (5,2), (5,3), (5,4)}

STEP 7: calculation of E.C.

(1,5) = 1+0 =1

(3,5) = 2+0 =2

(4,5) = 18+0 =18

(5,2) = 0+13 =13

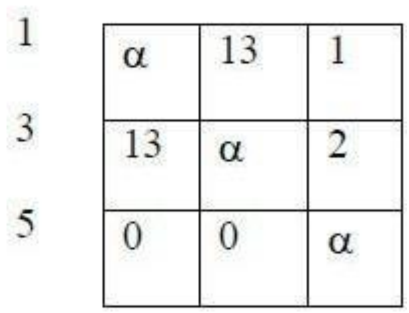
(5,3) = 0+13 =13

(5,4) = 0+1 =1

STEP 8: L having an edge (4,5) is the largest.

<u>STEP 9:</u> Delete (4,5) from C2 and make change in it as (5,4) = if exists.

Now, cost matrix

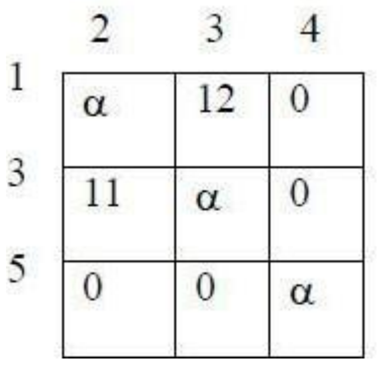


$$\begin{array}{c|ccc} & 2 & 3 & 4 \\ \hline 1 & \alpha & 13 & 1 \\ 3 & 13 & \alpha & 2 \\ 5 & 0 & 0 & \alpha \end{array}$$

<u>STEP 10:</u> THE cost matrix 2x2 hence go to step 1

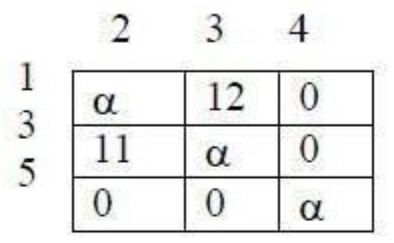
<u>PHASE III:</u>

<u>STEP 1: C3 (R, R)</u>



$$\begin{array}{c|ccc} & 2 & 3 & 4 \\ \hline 1 & \alpha & 12 & 0 \\ 3 & 11 & \alpha & 0 \\ 5 & 0 & 0 & \alpha \end{array}$$

STEP 3: C3 (C, R)



$$\begin{array}{c|ccc} & 2 & 3 & 4 \\ \hline 1 & \alpha & 12 & 0 \\ 3 & 11 & \alpha & 0 \\ 5 & 0 & 0 & \alpha \end{array}$$

STEP 5: preserve the above in C2

STEP 6: L={(1,4), (3,4), (5,2), (5,3)}

STEP 7: calculation of E.C

(1,4)=12+0=12

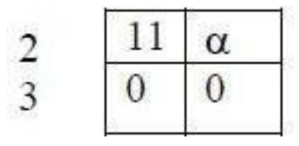
(3,4)=11+0=11

(5,2)=0+11=11

(5,3)=0+12=12

STEP 8: Here we are having two edges (1,4) and (5,3) with cost = 12. Hencearbitrarily choose (1,4)

STEP 9: Delete (i,j) (1,4) and make change in it (4,1) = if exists. Now costmatrix is



$$\begin{array}{c|cc} & 2 & 3 \\ \hline 2 & 11 & \alpha \\ 3 & 0 & 0 \end{array}$$

<u>STEP 10</u>: We have got 2x2 matrix

C4 (RR)=

$$\begin{array}{c|cc} & 2 & 3 \\ \hline 3 & 0 & \\ 5 & 0 & 0 \end{array}$$

C4 (C, R) =

$$\begin{array}{c|cc} & 2 & 3 \\ \hline 3 & 0 & \\ 5 & 0 & 0 \end{array}$$

Therefore,C4 =

$$\begin{array}{c|cc} & 2 & 3 \\ \hline 3 & 0 & \\ 5 & 0 & 0 \end{array}$$

<u>STEP 11:</u> LIST C1, C2, C3 AND C4

C4

	2	3
3	**0**	
5	**0**	**0**

C3	2	3	4
1	**12 0**		
3	11		0
5	0	0	

C2 =

1		13	1	0
3	13		2	0
4	43	18		0
5	**0**	**0**	**0**	

C1 =

	1	2	3	4	5
1		0	15	3	2
2	0		12	22	20
3	18	14		2	0
4	3	44	18		0
5	**15**	**1**	**0**	**0**	

<u>STEP 12:</u>

Use C4 =

	2	3
3	0	
5	**0**	**0**

Pick up an edge (I, j) =0 having least index

Here (3,2) =0

Hence, T (3,2)

Use C3 =

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

Pick up an edge (i, j) =0 having least index

Here (1,4) =0

Hence, T (3,2), (1,4)

Use C2=

	2	3	4	5
		13	1	0
1	13		2	0
3	43	18		0
4				
5	**0 0**	**0**		

Pick up an edge (i, j) with least cost index.

Here (1,5) not possible because already chosen index i (i=j)

(3,5) not possible as already chosen index.

(4,5) 0

Hence, T (3,2), (1,4), (4,5)

Use C1 =

	1	2	3	4	5
1		0	15	3	2
2	0		12	22	20
3	18	14		2	0
4	3	44	18		0
5	**15**	**1**	**0**	**0**	

Pick up an edge (i, j) with least index

(1,2) Not possible

(2,1)   Choose it

HENCE T   (3,2), (1,4), (4,5), (2,1)

SOLUTION:

From the above list

3—2—1—4—5

This result now, we have to return to the same city where we started (Here 3).

Final result:

3—2—1—4—5—3

Cost is 15+15+31+6+7=64

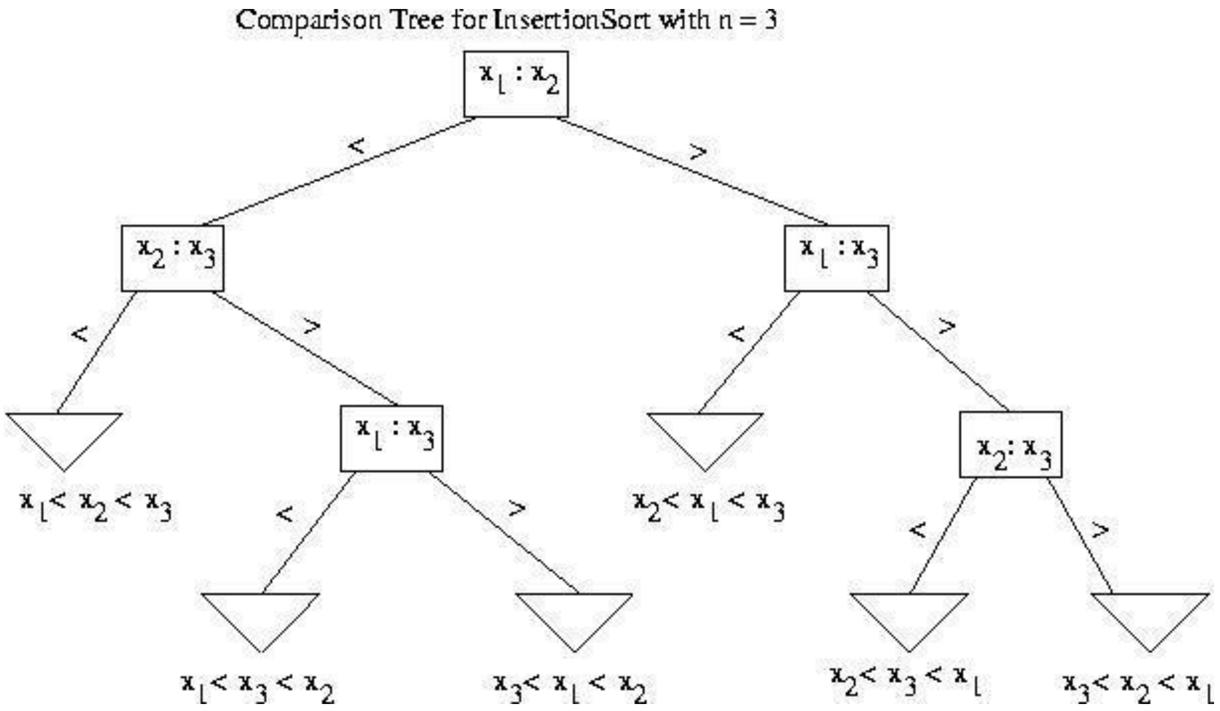
LOWER BOUND THEORY

COMPARISON TREES

We now show that MergeSort is also optimal on average, since nlog n is also a lower bound (again, up to a constant) for the average behavior of comparison-based sorting. This latter result will be established using a comparison tree argument. Given any comparison-based algorithm with input list
L[0:n-1] = {x_1, x_2, ..., x_n}, (internal) nodes in the comparison tree T associated with the algorithm correspond to comparisons performed by the algorithm between list elements. For specificity, our convention will be that if the comparison is made between x_i and x_j, and i < j, then we will label the corresponding node x_i:x_j. If $x_i < x_j$, then a left child will be the node corresponding to the next comparison made next by the algorithm, or this left child will be a leaf node if the algorithm terminates. Similarly, if $x_i > x_j$ (we can assume distinct list elements for the purpose of establishing lower bounds), then the right child will be the nodecorresponding to the next comparison made by the algorithm, or will be a leaf node if the algorithm terminates.

NOTE: our labeling of the nodes refers to the list elements, not to their positionsat the time the comparison corresponding to a given node is made.

The following figure illustrates the comparison-tree associated with InsertionSort for a list L[0:2] of 3 distinct elements x_1, x_2, x_3.



Key Fact. The comparson tree associated with any comparison-based sorting algorithm has n! leaf nodes.

The Key Fact follows from the fact that there are n! factorial permutations of n symbols, and different permutations must end up at different leaf nodes of the comparison tree when input to the algorithm. Since the comparison tree associated with a comparison-based sorting algorithm is a binary tree, lowerbounds for both worst-case and average complexity can be obtained from lower bounds for the depth and leaf path length (= sum of the lengths of all paths from the root to a leaf), respectively, of a binary tree having L leaf nodes.

Proposition 1. Let T be any binary tree with L leaf nodes. Then

$$\text{Depth}(T) \geq \text{ceil}(\log_2 L)$$

Proposition 1 is clearly true for complete binary trees (verify this!), so it is intuitively evident that it holds for arbitrary binary trees since the complete binary tree has the smallest depth for a given number L of leaf nodes. The formal proof ofProposition 1 can be found on p. 120 in the text. It follows immediately from Proposition 1 that

$$W(n) \geq \text{ceil}(\log_2 L)$$

for any comparison-based sorting algorithm. Now, ceil($\log_2 L$) = ceil($\log_2 n!$) $\in$ $\Omega(n\log n)$, so that we have established another proof of the fact that nlog n is a lower bound for the worst-case complexity of comparison-based sorting.

The following Proposition will give us a lower bound of nlog n for the averagecase as well.

Proposition 2. Let T be any binary tree having L leaf nodes. Then the leaf path length LPL of T satisfies:

$$LPT(T) \geq L\text{floor}(\log_2 L) \in \Omega(L\log L)$$

Again, Proposition 2 is clearly true for complete binary trees (verify this!), so it is evidently true for arbitrary trees. A formal proof of Proposition 2 can be found on p. 124 in the text. Now if T is the comparison tree associated with any comparison-based sorting algorithm, we see that A(n) = LPT(T)/L, so that Proposition 2 shows that nlog n is a lower bound for the average behavior of any comparison-based algorithm. Again, our old friends MergeSort, QuickSort, and TreeSort are all optimal average behavior comparison-based sorting algorithms.

We now illustrate our third technique for establishing lower bounds, namely adversary arguments. This technique establishes lower bounds by creating an input instance, based on the performance of the algorithm, which guarantees that the algorithm must do a determined amount of work on this input in order to be correct for this input. This amount of work then gives a lower bound for the worst-case complexity of the algorithm. Another adversary-type technique is to construct an input to an algorithm which contradicts the correctness of the algorithm if the algorithm performs less than some given number of basic operations. We start with an example of this type of adversary argument.

FINDING THE MAXIMUM IN A LIST

The usual linear scan for finding the maximum element in a list L[0:n-1] of size n turns out to be optimal, since ANY comparison-based algorithm for solving this problem must make n - 1 comparsions between list elements. This might seem obvious, since certainly every element must participate in at least one comparison. However, only n/2 comparisons are required to ensure that each element is involved in a comparison. Just pair the elements up into disjoint pairs amake a comparison to the two elements in each pair. Of course, this doesn't yet determine the maximum, but it shows the need for further justification that n - 1 comparisons will eventually be required. Again, throughout we will assume distinct list elements.

To get the lower bound of n - 1 comparisons, we consider a comparison between list elements x and y to declare as the loser the smaller of the two elements. Thus, each comparison results in exactly one loser. We now note that there must be n -1 losers if the algorithm is to determine the maximum

element in a list L[0:n-1] correctly. Indeed, assume that there are two elements x and y who never lost a comparison, and, for definiteness, assume that x > y. Now we may suppose that the algorithm has declared that x is the maximum element (otherwise it is clearly incorrect). Now construct a new list L' which agrees with L except that y is replaced by an element y' > x
> y. Note that the algorithm will perform exactly the same action on L' as it did with L, since y' will win every comparison that involved y (and the outcome of all the other comparisons will also be the same). Hence, the algorithm will again declare x to be the maximum element, which is a contradiction. We state this result as a proposition.

Proposition 3. Any comparison-based algorithm must make (at least) n - 1 comparisons of list elements in order to correctly determine the maximum.

Proposition 3 shows that the familiar linear scan algorithm for finding the maximum is an (exactly) optimal algorithm. It is interesting that there is another algorithm also performing n - 1 comparisons to find the maximum, but this time it is based on the familiar single elimination tournament model so familiar from the sporting world. For simplicity, we assume that $n = 2^k$. Divide up the list into disjoint pairs, and determine the n/2 pair-wise winners (1st round of the tournament). Then divide up the n/2 first-round winners into pairs and determine
the n/4 second round winners. After precisely $\log_2 n$ rounds the winner (maximum) will be determined. But how many comparisons (matches) were made? Easy, we get, for $n = 2^k$: $2^{k-1} + 2^{k-2} + ... + 1 = 2^k - 1 = n - 1$.

FINDING THE MAXIMUM AND THE MINIMUM

The most naive method MaxMin1 for finding the maximum and minimum elements in a list is to make two linear scans (or run winner and loser tournaments), resulting in 2n - 2 comparisons. However, one imagines that this can be improved, since information about both elements involved in a comparison might be utilized. In fact, the following slightly less naive algorithm certainly improves MaxMin1, at least on average.

```
function MaxMin2(L[0:n-1]) Input:
L[0:n-1] (a list of size n) Output:
the maximum value in L
  Max = Min = L[0] for
  i = 1 to n-1 do
     if L[i] > Max then Max = L[i]
     else
        if L[i] < Min then Min = L[i]
        endif
     endif
  enfor
end MaxMin1
```

Note that in the best case of a strictly increasing list, MaxMin2 only makes n - 1 comparisons, whereas in the worst case W(n) of an decreasing list, MaxMin2 makes 2n - 2 comparisons, i.e., is just as bad as MaxMin1. The question is, how good is MaxMin2 on average? Well, it turns out that it is disappointing, since its average behavior, while improved slightly over W(n), is nevertheless stronglyasymptotic to W(n). This is because the average number of times that Max is updated in MaxMin2 is (guess what!) logarithmic in n, so that the average complexity A(n) of MaxMin2 is of the form A(n) = W(n) - f(n), where f(n) ∈ O(log n), i.e., A(n) ∈ Θ(W(n)) (actually, A(n) ~ W(n)).

In order to determine the average behavior A(n) of MaxMin2, we assume, as usual, that the inputs are all permutations π: {1,2, ..., n} → {1,2, ..., n}, and that each permutation is equally likely. Now if m(π) denotes the the number of times Max is updated for input permutation π, then it is clear that

$$A(n) = 2n - 2 - E[m].$$

Let A*(n) = E[m]. Note that π(n) is equally likely to be 1, 2, ..., n. Hence,

$$A^*(n) = 1/n(E[m|\ \pi(n) = 1] + E[m|\ \pi(n) = 2] + \ldots + E[m|\ \pi(n) = n]).$$

Now it is clear that E[m| π(n) = n] = A*(n-1) + 1, whereas E[m| π(n) = i ≠ n] = A*(n-1). Hence, we have the following recurrence for A*(n):

$$\begin{aligned} A^*(n) &= A^*(n - 1) + 1/n \\ &= A^*(n - 2) + 1/n + 1/(n - 1) \\ = \\ &\ldots \\ &= A^*(1) + 1/n + 1/(n - 1) + \ldots + 1/2 \\ &= \sim \ln n. \end{aligned}$$

Thus, we see that A(n) ~ W(n) ~ 2n.

UNIT-V

NP-HARD AND NP-COMPLETE PROBLEMS:

Basic concepts:

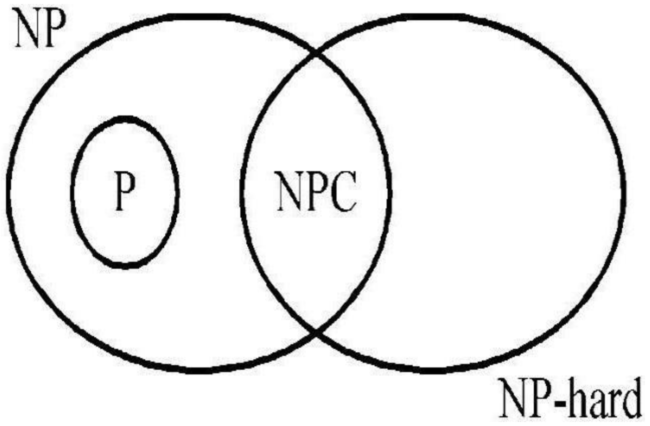
Tractability: Some problems are tractable: that is the problems are solvable in reasonable amount of time called polynomial time. Some problems are intractable: that is as problem grow large, we are unable to solve them in reasonable amount of time called polynomial time.

Polynomial Time Complexity: An algorithm is of Polynomial Complexity, if there exists a polynomial p() such that the computing time is O(p(n)) for every input size of 'n'. Polynomial time is the worst-case running time required to an algorithm to process an input of size n the is $O(n^k)$ for some constant k

Polynomial time: $O(n^2)$, $O(n^3)$, O(n log n)

Not in polynomial time: $O(2^n)$, $O(n^n)$, O(n!) Exponential Time

Most problems that do not yield polynomial-time algorithms are either optimization or decision problems.



Decision Problems: Computational problem with produces output of "yes" or "no", 1 or 0 are decision problems.

Examples: 1. Path in a graph

2. Minimum Spanning Tree whose cost is less than some value w.

Optimization Problems: Computational problem where we try to maximize or minimize some value that is identifying optimal solution to problem

Examples: 1. Shortest-path in a graph.
2. Minimum Spanning Tree

CLASS P PROBLEMS:

Class P problems are the set of decision problems solvable by deterministic algorithms in polynomial-time.

A deterministic algorithm is (essentially) one that always computes the correct answer

Examples: Fractional Knapsack, MST, Single-source shortest path

CLASS NP PROBLEMS:

NP problems are set of decision problems solvable by non-deterministic algorithms in polynomial-time.

A nondeterministic algorithm is one that can “guess” the right answer or solution Examples:

Hamiltonian Cycle (Traveling Sales Person), Conjunctive Normal Form (CNF)

NP-Complete Problems:

A problem ‘x’ is a NP class problem and also NP-Complete if and only if every other problem in NPcan be reducible (solvable) using non-deterministic algorithm in polynomial time.

The class of problems which are NP-hard and belong to NP.

The NP-Complete problems are always decision problems only.

Example : TSP, Vertex covering problem

Packing problems: SET-PACKING, INDEPENDENT-SET.

Covering problems: SET-COVER, VERTEX-COVER.

Sequencing problems: HAMILTONIAN-CYCLE, TSP.

Partitioning problems: 3-COLOR, CLIQUE.

Constraint satisfaction problems: SAT, 3-SAT. Numerical

problems: SUBSET-SUM, PARTITION, KNAPSACK

NP-Hard Problems:

A problem ‘x’ is a NP class problem and also NP-Hard if and only if every other problem in NP canbe reducible (solvable) using non-deterministic algorithm in exponential time.

The class of problems to which every NP problem reduces.

The NP-Hard problems are decision problems and sometimes may be optimization problems.

Example : Integer Linear Programming.

<u>Nondeterministic Algorithms:</u>

Deterministic Algorithms:

- Let A be an algorithm to solve problem P. A is called deterministic if it has only one choice in each step throughout its execution. Even if we run algorithm A again and again, there is no change in output.
- Deterministic algorithms are identified with uniquely defined results in terms of output for a certaininput.

Nondeterministic Algorithms:

- Let A be a nondeterministic algorithm for a problem P. We say that algorithm A accepts an instanceof P if and only if, there exists a guess that leads to a yes answer.
- In non deterministic algorithms, there is no uniquely defined result in terms of output for a certaininput.
- Nondeterministic algorithms are allowed to contain operations whose outcomes are limited to agiven set of instances of P, instead of being uniquely defined results.
- A Non-deterministic algorithm A on input x consists of two phases:
  - Guessing: An arbitrary “string of characters” is generated in polynomial time. It may

Correspond to a solution Not correspond to a solution

Not be in proper format of a solution Differ from one run to another

  - Verification: A deterministic algorithm verifiesThe

generated “string of characters” is in proper format Whether it

is a solution in polynomial time

- The Nondeterministic algorithm uses three basic procedures, whose time complexity is O(1).

1. CHOICE(1,n) or CHOICE(S) : This procedure chooses and returns an arbitrary element,

in favor of the algorithm, from the closed interval [1,n] or from the set S.

2. SUCCESS : This procedure declares a successful completion of the algorithm.

3. FAILURE : This procedure declares an unsuccessful termination of the algorithm.

- Non deterministic algorithm terminates unsuccessfully if and only if there is no set of choicesleading to successful completion of algorithm
- Non deterministic algorithm terminates successfully if and only if there exists set of choices leadingto successful completion of algorithm

Nondeterministic Search Algorithm: The following algorithm enables nondeterministic search of x in an unordered array A with n elements. It determines an index j such that A[j] = x or j = −1 if x does not belongs to A.

```
Algorithm nd_search ( A, n, x )
{
int j = choice ( 0, n-1 );if
( A[j] == x )
{
cout << j;
success();
}
cout << -1;
failure();
}
```

By the definition of nondeterministic algorithm, the output is -1 iff there is no j such that A[j] = x . Since A is not ordered, every deterministic search algorithm is of complexity O(n), whereas the nondeterministic algorithm has the complexity as O(1).

Nondeterministic Sort Algorithm: The following algorithm sorts ‘n’ positive integers in non-decreasing order and produces output in sorted order. The array B[] is an auxiliary array initialized to 0 and is used for convenience.

```
Algorithm nd_sort ( A, n )
{
for ( i = 0; i < n; B[i++] = 0; );
for ( i = 0; i < n; i++ )
{
j = choice ( 0, n - 1 ); if
( B[j] != 0 ) failure();B[j]
= A[i];
}

// Verify order
for ( i = 0; i < n-1; i++ )
if ( B[i] > B[i+1] ) failure();
write ( B );
success();
}
```

The time complexity of nd_sort is O(n). Best-known deterministic sorting algorithm like binary search has a complexity of (n log n).

Satisfiability: (SAT Problem)

Let x1, x2 . . . denote a set of Boolean variables and xi denote the complement of $\bar{x}$ i.A

variable or its complement is called a literal

A formula in propositional calculus is an expression that is constructed by connecting literalsusing the operations and ( ) & or ( )

Examples of formulas in propositional calculus$(x_1$ ^

$x_2)$ V $(x_3$ ^ $\bar{x}_4)$

$(x_3$ V $\bar{x}_4)$ ^ $(x_1$ V $\bar{x}_2)$

Conjunctive normal form (CNF): A Boolean formula is said to be in conjunctive normal form(CNF) if it is the conjunction of formulas.

Example: $(x_1 \quad \bar{x}_2) \quad (\bar{x}_1 \quad x_5)$

Disjunctive normal form (DNF) : A Boolean formula is said to be in disjunctive normal form(CNF) if it is the disjunction of formulas.

Example: $(x_1 \quad \bar{x}_2) \quad (x_1 \quad \bar{x}_5)$

Satisfiability problem is to determine whether a formula is true for some assignment of truthvalues to the variables

CNF--satisfiability is the satisfiability problem for CNF formulasDNF-

-satisfiability is the satisfiability problem for DNF formulas

Polynomial time nondeterministic algorithm that terminates successfully iff a given

propositional formula E(x1, . . . , xn) is satisfiable

Non deterministically choose one of the 2^n possible assignments of truth values to (x1, . . . , xn) and verify that E(x1, . . . , xn) is true for that assignment

Algorithm eval (E, n)

{

// Determine whether the propositional formula E is satisfiable.Here

variable are x1, x2, ..., xn

```
 for ( i = 1; i <= n; i++ )
x(i) = choice ( true, false );if
( E ( x1, ..., xn ) ) success();
else
failure();
}
```

The nondeterministic time to choose the truth value is O(n)

The deterministic evaluation of the assignment is also done in O(n) time

<u>Decision Problem Vs Optimization Problem:</u>

<u>Decision Problem and Algorithm</u> : Any problem for which the answer is either zero or one is called a decision problem. An algorithm for a decision problem is termed a decision algorithm.

A decision algorithm will output 0 or 1 Implicit in

the signals success() and failure()

Output from a decision algorithm is uniquely defined by input parameters and algorithm specification.

<u>Optimization Problem and Algorithm:</u> Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as an optimization problem. An optimization algorithm is used to solve an optimization problem.

An optimization problem may have many feasible solutions

The problem is to find out the feasible solution with the best associated value

NP-completeness applies directly not to optimization problems but to decision problems.

<u>Casting (Conversion) of Optimization Problem into Decision Problem</u>:

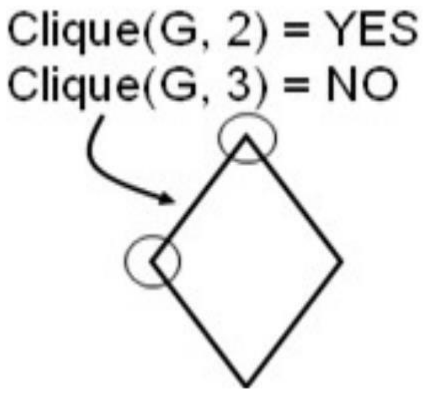
Optimization problems can be cast into decision problems by imposing a bound on output or solution. Decision problem is assumed to be easier (or no harder) to solve compared to the optimization problem. Decision problem can be solved in polynomial time if and only if the corresponding optimization problem can be solved in polynomial time. If the decision problem cannot be solved in polynomial time, the optimization problem cannot be solved in polynomial time either.

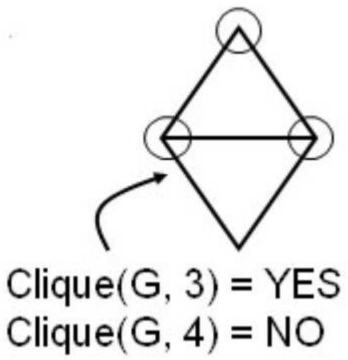
For example consider, shortest path problem. Optimization problem is to find a shortest path between two vertices in an undirected weighted graph, so that shortest path consists least number of edges. Whereas the decision problem is to determine that given an integer k, whether a path exists between two specified nodes consisting of at most k edges.

Maximal Clique:

Clique is a maximal complete sub-graph of a graph G = (V,E), that is a subset of vertices in V all connected to each other by edges in E (i.e., forming a complete graph).

Example:





The Size of a clique is the number of vertices in it. The Maximal clique problem is an optimization problem that has to determine the size of a largest clique in G. A decision problem is to determine whether G has a clique of size at least 'k'.

Input for Maximal clique problem: Input can be provided as a sequence of edges. Each edge in E(G) is a pair of vertices (i, j) .The size of input for each edge (i, j) in binary representation is

$$\lfloor \log_2 i \rfloor + \lfloor \log_2 j \rfloor + 2$$

And input size of any instance is given by

$$n = \sum_{\substack{(i,j) \in E(G) \\ i<j}} (\lfloor \log_2 i \rfloor + \lfloor \log_2 j \rfloor + 2) + \lfloor \log_2 k \rfloor + 1$$

Where 'k' is the number to indicate the clique size

Maximal clique problem as Decision Problem:

Let us denote the deterministic decision algorithm for the clique decision problem as dclique(G,k). If |V | = n, then the size of a maximal clique can be found by

for (k = n; dclique (G, k) != 1; k--);

If time complexity of dclique is f(n), size of maximal clique can be found in time g(n) <=n.f(n). Therefore, the decision problem can be solved in time g(n)

Note that Maximal clique problem can be solved in polynomial time if and only if the clique decision problem can be solved in polynomial time.

Non deterministic Clique Algorithm:

```
    Algorithm DCK(G, n, k)
{
    S=0; // Empty set.
    for i=1 to k do
    {
        t = Choice(1,n);
        if t ε S then Failure();
        S= SU{ t };
    }
    for all pairs (i,j) such that i ε S, j ε S and i!=j doif (i,
        j) is not an edge of G then Failure();
    Success();
}
```

Non-deterministic knapsack problem:

It is a non-deterministic polynomial time complexity algorithm.

The for loop selects or discards each of the n items It also re-computes the total weightand profit corresponding to the selection

The if statement checks to see the feasibility of assignment and whether the profit isabove a lower bound r

The time complexity of the algorithm is O(n) . If the input length is q in binary, then O(q).

```
algorithm nd_knapsack ( p, w, n, m, r, x )

{


for ( i = 1; i <= n; i++ )

{

x[i] = choice ( 0, 1 );

W += x[i] * w[i];


}

if ( ( W > m ) || ( P < r ) )

failure();

else

success();

}
```

SUM OF SUBSETS PROBLEM:

Bits are numbered from 0 to m from right to left

Bit i will be 0 if and only if no subsets of A[j], 1 _ j _ n sums to i

Bit 0 is always 1 and bits are numbered 0, 1, 2, . . . ,m right to left

Number of steps for this algorithm is O(n)

Each step moves m + 1 bits of data and would take O(m) time on a conventional computer

Assuming one unit of time for each basic operation for a fixed word size, the complexityof deterministic algorithm is O(nm)

Consider the deterministic decision algorithm to get sum of subsets

```
algorithm sum_of_subsets ( A, n, m )
{
// A[n] is an array of integerss
= 1 // s is an m+1 bit word
// bit 0 is always 1for
i = 1 to n
s |= ( s << A[i] ) // shift s left by A[i] bits if bitm
in s is 1
write ( "A subset sums to m" );
else
write ( "No subset sums to m" );
}
```

COOK'S THEOREM:

We know that, Class P problems are the set of all decision problems solvable by deterministic algorithms in polynomial time. Similarly Class NP problems are set of alldecision problems solvable by nondeterministic algorithms in polynomial time.

Since deterministic algorithms are a special case of nondeterministic algorithms, P NP

Cook formulated the following question: Is there any single problem in NP such that if we showed it to be in P, then that would imply that P = NP? This led to Cook's theorem as :

Satisfiability is in P if and only if P = NP.

Cook's Theorem Proof:

Consider Z - denotes a deterministic polynomial algorithmA -

denotes a non- deterministic polynomial algorithm

I - denotes input instance of algorithmn -

denotes length of input instance

Q - denotes a formulae

m - denotes length of formulae

Now, the formula 'Q' is satisfiable if and only if the non-deterministic algorithm 'A' has a successful termination with input 'I'.

If the time complexity of 'A' is p(n) for some polynomial p(), then the time needed to construct the formula 'Q' by algorithm 'A' is given by $O(p^3(n)\log n)$.

Therefore complexity of non-deterministic algorithm 'A' is $O(p^3(n)\log n)$. (NP)

Similarly, the formula 'Q' is satisfiable if and only if the deterministic algorithm 'Z' has a successful termination with input 'I'.

If the time complexity of 'Z' is q(m) for some polynomial q(), then the time needed to construct the formula 'Q' by algorithm 'Z' is given by $O(p^3(n)\log n + q(p^3(n)\log n))$.

Therefore complexity of deterministic algorithm 'Z' is $O(p^3(n)\log n + q(p^3(n)\log n))$. (P)

If satisfiability is in P, then q(m) is a polynomial function and the complexity of 'Z' becomesO(r(n)) for some polynomial r(n).

Hence, P is satisfiable, then for every non-deterministic algorithm 'A' in NP can obtain a deterministic algorithm 'Z' in P.

So, the above construction shows that "if satisfiability is in P, then P=NP"