Unit 1 ALGORITHM

Definition: Algorithm

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

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

Issues in study of Algorithm:

1. How to device or design an algorithm → creating an 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 chosen care should be taken, we should ensure that each & every statement is definite.

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

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

Pseudo-Code Conventions(Algoritm specification)

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;

}

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-2>

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:

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

```
        Result :=A[I];
        return Result;
    }
```

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

Recursive Algorithms:

A Recursive function is a function that calls itself in its body. Similarly an algorithm is said to be recursive if the same algorithm is called its 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'.

Performance Analysis of Algorithm:

Two criteria used in analyzing the performance of an algorithm are Space complexity and Time Complexity. They are discussed below.

1. Space Complexity: The space complexity of an algorithm is the amount of memory it needs to run to compilation.

2. Time Complexity: The time complexity of an algorithm is the amount of computer time it needs to run to compilation.

Space Complexity:

Space Complexity Example1:

```
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.

2. A variable part that consists of the space needed by component variables whose size is

dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that is depends on instance 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.

Space complexity Example 2:

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

- 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.
- The space needed by 'a' is the space needed by variables of type array of floating point numbers.
- This is at least 'n' words, since 'a' must be large enough to hold the 'n' elements to be summed.
- So, we obtain Ssum(n)>=(n+3)

Time Complexity:

The time T(p) taken by a program P is the sum of the compile time and the run 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 times a statement is executed is called step count. It is used to find the runtime of an algorithm. It is computed in two ways. The first way is given below.

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

For example, the step count for comments count as 0 steps , Assignment statements counted as 1 steps and iterative statement such as for, while & repeat-until are counted for the control part of the statement only

1. We introduce a variable, count ,into the program statement to increment . Count has 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. So each invocation of sum execute a total of 2n+3 steps.

The second way 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 statement and the total number of times (ie., frequency) each statement is executed.

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

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

Asymptotic notation: (O, Ω , Ø)

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

// Attention student : include examples for Big OH from class notes here//

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

// Attention student : include examples for Big Ω from class notes here//

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

// Attention student : include examples for Big **Ø** from class notes here//

Analyze the time complexity of following algorithms

1. Algorithm BSC( )
{
int i;
for i=0 to n do
write("Computer");
}
 	
Time Complexity is O(n)
	2. Algorithm BSC( )
	{
	int i,j;
	for i =1 to n do
	for j= 1 to n do
	write('Computer");
	}
	
	Time Complexity is: $O(n^2)$

3. Algorithm A( )
{
int i=1,s=1;
	4. Algorithm BSC( )
	{
	int i=1;

```
        while(s<=n)
        {
                i=i++;
                s=s+i;
                write("Android");
        }
}
```

i=1,2,3,4,5,6.....k;
s=1,3,4,10,15,21.....k(k+1)/2
so
(k(k+1))/2 >n
(k2 +k)/2>n

Time Complexity is : O(Squareroot(n))

```
        for (i=1 ; i^2<n;i++)
        write("DAA");
}
```

Time Complexity is : O(SR(n))

5. Algorithm BSW( )

```
{
        int i,j,k,n;
        for(i=1;i<=n;i++)
        for(j=1;j<=i;j++)
        {
                for(k=1;k<=100;k++)
                        write("Department");
        }
}
```

i=1	j=1	k=100
i=2	j=1,2	k=2*100
i=3	j=1,2,3	k=3*100
1=4	j=1,2,3,4	k=4*100
.		
.		
i=n	j=1,2,3.....n	k=n*100

Total Time k loop execute is:

=100+ 2*100 +3 *100 **+** 4*100+.....n*100
=100(1+2+3+.....n)
=100((n*(n+1))/2

Time Complexity is : O(n^2)

6. Algorithm BCA( )

```
{
        int i,j,k,n;
        for (i=1;i<=n;i++)
        for(j=1;j<=i2;j++)
        for(k=1;k<=n/2;k++)
                write("DAA");
}
```

i=1	j=1	k=1*n/2
i=2	j=4	k=4*n/2
i=3	j=9	k=9*n/2
i=n	j=n^2	k=n^2*n/2

Total Time: n/2(1+4+9+.....n^2)
=n/2(n(n+1)(2n+1))/6

Time Complexity is : O($n^{4)}$

7. Algortihm BSW( )

```
{
        int i,n;
        for (i=1;i<=n;i=i*2)
                write("Computer");
}
```

i=1,2,3,....n
$2^0,2^1,2^2,.....2^k$
2^k=n

8. Algorithm BCA( )

```
{
        int i,j,k;
        for (i=n/2;i<=n;i++)
        for(j=1;j<=n/2;j++)
        for(k=1:k<=n;k=k*2)
                write("Knapsack");
}
```

Time Complexity is: n/2*n/2*log2^n

$k=\log_2(n)$
Time Complexity is : $O(\log 2^n)$

$O(n^2 \log 2^n)$

9. Algorithm BSC( )

```
{
        int i,j,k;
        for(i=n/2;i<=n;i++)
        for(j=1;j<=n;j=2*j)
        for(k=1;k<=n;k=k*2)
                write("Computer");
}
```

Time Complexity: $n/2 * \log_2 n * \log_2 n$
$= n/2(\log_2 n)2$
$= O((n\log_2 n)^2)$

10. Algorithm BSC( )

```
{
        int i,j,n;
        for (i=1;i<=n;i++)
        for(j=1;j<=n;j=j+i)
                write("Computer");
}
```

i=1 j=1.....n so n times
i=2 j=1,3,5,.....n/2 times
i=3 j=1,4,7,10.....n/3 times
i=k j=.....n/k times
i=n j=.....n/n times

=n(1=1/2+1/3+.....1/n)
=nlogn
Time Complexity is : O(nlogn)

11. Algorithm BCA( )

```
{
        int n=(((2)2)k);
        for(i=1;i<=n;i++)
        {
                j=2;
                while(j<=n)
                {
                        j=j2;
                        write("Android");
                }
        }
}
```

k=1 n=4 j=2,4
n*2 times
k=2 n=16 j=2,4,16 n*3 times
k=3 n=(((2)2)3) $j=2^1,2^2,2^3,2^4,2^8$
.....n*(k+1) times
k=n.....n(k+1) times

But n= 2 power 2 power k
$\log 2^n = 2^k$
$\log\log_2 n$=;
so, $n(n+1)=n*(\log\log_2 n+1)$
Time Complexity is: $O(n\log\log_2 n)$

Randomized Alogrithm: An Informal Description

A randomized algorithm is one that makes use of a randomizer (such as a random number generator).Some of the decision made in the algorithm depend on the output of the randomizer. Since the output of any randomizer might differ in an unpredicted way from run to run, the output of a randomized algorithm could also differ from run to run for the same input. The execution time of a randomized algorithm could also vary from run to run for the same input.Randomized algorithms can be categorized into two classes namely Las Vegas algorithms and Monte Carlo algorithms. The Las Vegas algorithms are randomized algorithms that always produce same (correct) output for the same input. The execution time of a Las Vegas algorithm depends on the output of the randomizer. The algorithm might terminate fast, and if not, it might run for a longer period of time. The Monte Carlo algorithms are randomized algorithms that produce different output from run to run for the same input. Consider any problem for which there are only two possible answers say yes and no. If a Monte Carlo algorithm is employed to solve such a problem then the algorithm might give incorrect answers depending on the output of the randomizer.

Definition Õ(g(n)) of Las Vegas algorithm.

A Las Vegas algorithm has resource (time, space, and so on.) bound of Õ(g(n)) if there exist a constant c such that the amount of resource used by the algorithm (on any input size n) is no more than c α g(n) with probability >= 1- $1/n^{\alpha}$ We shall refer to these bounds as high probability bounds.

Las Vegas algorithm example: Identifying the Repeated Element

Problem description :

Consider an array α [] of numbers that n/2 has distinct elements and n/2 copies of another element. The problem is to identify the repeated element.

Method of solution:

The Las Vegas algorithm randomly picks two array elements and checks whether they come from two different cells and have the same value. If they do, the repeated element has been found. If not, this basic step of sampling is repeated as many times as it takes to identify the repeated element.

```
Repeated Element (α,n)
// Finds the repeated element from  α[1:].
{
   While (true) do
  {
    i  :=Random () mod  n + 1;
    J := Random() mod  n+1;
      //  and   are  random numbers in the range [1, ].
     if (( i ≠ j ) and ( α[  i] = α[  j])) then return  i ;
   }
}
```

Monte Carlo algorithm example: Primality Testing

Problem description: Given an integer n , the problem of deciding whether n is a prime is known as primality testing.

Method of solution:

If a number is composite (i.e., nonprime), it must have a divisor $\leq \sqrt{n}$. This observation leads to the following simple algorithm for primality testing : Consider each number l in the interval [2,√n] and check whether l divides n . If none of these numbers divides n , then n is prime; otherwise it is composite.

We can devise a Monte Carlo randomized algorithm for primality testing that runs in time $O((\log n)^2)$. The output of this algorithm is correct with high probability . If the input is prime , the algorithm never give an incorrect answer . However , if the input number is composite (i.e., nonprime), then there is a small probability that the answer may be incorrect. Algorithms of this kind are said to have one –sided error.

```
Prime (n,α)
// Returns true if n is prime and false otherwise.
// α is the probability parameter.
{
q:=n-1;
for i:=1 to large do // Specify large.
```

```
{
m:=q; y:=1;
α:=Random () mod q+1;
// Choose a random number in the range [1,n-1].
Z:=α;
//  Compute  α^(n-1) mod n
While (m>0) do
{
While (m mod 2=0) do
{
Z:=z^2 mod n; m:= m/2 ;
}
m:= m-1; y:=(y*z) mod n;
}
If (y≠1)then return false ;
// If  α^(n-1) mod n is not 1, n is not a prime.
}
Return true;
```

Advantages and Disadvantages of Randomized algorithms

Advantages: Two of the most important advantage of using randomize algorithms are their simplicity and efficiency. Randomize algorithms have also been shown to yield better complexity bounds.

Disadvantages: Randomized algorithms have some small Where other simple probability of error cannot be tolerated.

Unit 2 - Divide and Conquer Method.

General method:

This is one of the problem solving techniques. It can be used for certain kinds of problems like searching an element in a given list, finding the biggest and smallest of given numbers, sorting the given numbers, matrix multiplication etc,. The method of divide and conquer is as follows.

Given a function to compute on n-input, the divide and conquer technique split the input into k-distinct sub sets 1<=k<=n, yielding k-sub problems. These sub problems must be solved and the solution of these sub problem are combined into a solution of the given original problem. If the sub problem is still large, then reapply the divide and conquer technique on that sub problem. The sub problem and the original problem are of same kind and the characteristics. Smaller and smaller sub problems of the same kind are generated until eventually sub problems that are small enough to be solved without splitting are produced. The following is the control abstraction of the divide and conquer method

```
Algorithm DandC(p)
{
    If  Small(p) then return s(p);
    else
    {
        Divide  P into smaller instance P1,P2,…….Pk, k>=1 ;
        Apply DandC to each of these sub-problems;
        Return Combine (DandC(P1),DandC(P2)…….DandC(Pk));
    }
}
```

Here Small(p) is a Boolean function that determine whether the input size is small enough to compute the answer without splitting. If this is so, the function S(P) is involved, otherwise the problem P is divided into smaller sub problem. These sub problem P is divided into smaller sub problems. These sub problem P1,P2,……..,Pk are solved by recursive application of the function DandC . Combine is the function that determine the solution of the problem(p) using the solution of P1,P2,……..,Pk. If the size of problem P is n and sizes of k-sub problems are n1,n2,……,nk respectively then the computing time of the algorithm of DandC is determined by the recursive relation.

T(n) = { g(n) if n is small

{ T(n1)+T(n2)+T(n3)+……+T(nk)+ f(n).

The T(n) is the time taken by the DandC algorithm inputs of size n and g(n) is the time taken to compute the answer directly for small inputs and f(n) is the time for dividing the problem and combining the solutions of the problems. T(n1),T(n2),…….T(nk) are the time required to solve the sub problems.

The complexity of many divide and conquer algorithm is given by the recursive relation of the

T(n)= { T(1) if n=1

{ a T(n/b)+f(n) if n>1

Where a and b are constants.

Binary Search:

Problem Definition

Given a list of n-elements in an array a[1]..a[n] in ascending order, the binary search problem to find whether an element x is present in the list or not. If x is present, determine the index j such that a[j] is equal to x otherwise set j=0.

Method of Solution

Let p=(n, a[i],…….a[l], x) denote an arbitrary instance of this search problem. Here n is the number of element available in the array locations a[i]……a[l] and x is the element to be searched. Let small(p) be true if n=1. In this case s(p) will take the value of i if x=a[i] otherwise it will take the value 0. If p has more than one element ,it is divided into new sub problems as follows.

Pick an index q and compare x with possibilities

1. X=a[q], in this case the problem p immediately solved.
2. X<a[q], in this case x has to be searched in the list a[i],a[i+1]……,a[q-1]. Now the problem is reduced from (n, a[i],…….a[l],x) to (q-1, a[i],……, a[q-1], x).
3. X>a[q], in this case x has to be searched in sub list a[q+1],…….a[l]. now the problem p is reduced to (l-q,a[q+1],……..a[l], x).

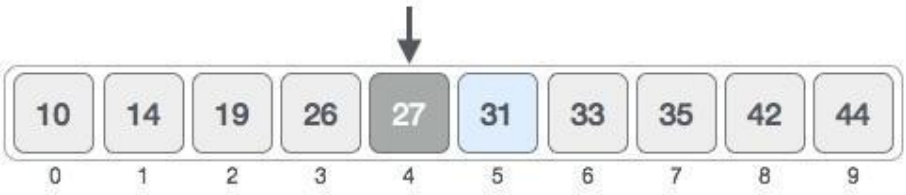
Algorithm:

```
Algorithm Binsrch(a,i,l,x)
//given an array a[i].....a[l] of elements in ascending orders, determine whether x is present. If present , return j such that a[j]=x else return j=0 //
{
    If (i=l) then  //if small(p) //
    {    if(x==a[i]) then return i
    else
        return 0;
    }
    Else
    {    //reduce p into smaller sub-programs //
        mid=(i+l)/2
        If x==a[mid] then return mid
        else if(x<a[mid]) then return Binsrch(a,i,mid-1,x)
        else return Binsrch(a,mid+1,l,x);
    }
}
```

Example:
Search the location of value 31 in the following array A using binary search.

mid = (low + high) / 2

Here it is, (0 + 9) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now a[4] = 27 which is less than 31, value 31 must be in the upper portion of the array.

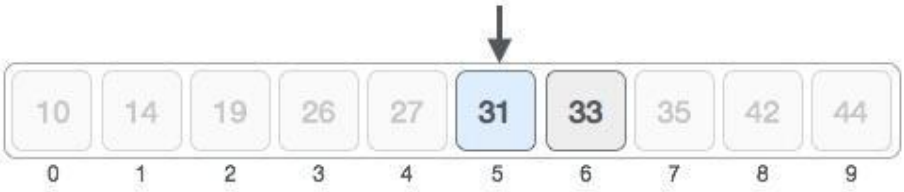
We change our low to mid + 1 and find the new mid value again.

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

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Maximum and Minimum

Problem definition :

Given a list of numbers in an array, the maxima and minima problem is to find the biggest and smallest among them using divide and conquer method

Method of solution:

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 sub problems //
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 node each time a new call is made. For this algorithm, each node has 4 items of information: i, j, max & min. Examining fig: we see that the root node contains 1 & 9 as the values of i &j corresponding to the initial call to MaxMin.

No. of element Comparison:

If T(n) represents number of element comparisons needed in this algorithm , then the resulting recurrence relations is

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

When 'n' is a power of two , n=2^k for some +ve integer 'k', then

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

= 2(2T(n/2)+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 two.

Merge sort

Problem Definition

Given a list of n number stored in an array in random order, arrange them in sorted order by dividing the array into two sets , sort each set and merge them .

Method of Solution

Let the given 'n' elements in random order be stored in array a[1],…,a[n] . Split the array 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. The following is the algorithm for merging two sorted sets and used in the subsequent algorithm for Mergesort.

```
Algorithm merge(low,mid,high)
    //a[low:high] is a global array containing 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. //
{
    h=low; i=low; j=mid+1;
    while ((h<=mid) and (j<=high)) do
     {
        if (a[h]<=a[j]) then
        {
            b[I]=a[h];
            h = h+1;
        }
        else
        {
            b[i]= a[j];
```

```
                    j=j+1;
                }
                 i=i+1;
            }
            if (h>mid) then
                for k=j to high do
                 {
                    b[I]=a[k];
                     i=i+1;
                }
             else
                 for k=h to mid do
                {
                     b[I]=a[k];
                     i=i+1;
                 }
                 for k=low to high do a[k] = b[k];
        }
```

```
        Algorithm MergeSort(low,high)
        //a[low:high] is a global array to be sorted Small(P) is true if there is only
one element to sort. In this case the list is already sorted. //
        {
            if (low<high) then //if there are more than one element //
            {
                 //Divide P into sub problems find where to split the set //
                mid = [(low+high)/2];
                //solve the sub problems.//
                 mergesort (low,mid);
                 mergesort(mid+1,high);
                 //combine the solutions .//
                 merge(low,mid,high);
            }
        }
```

Example :

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 size five (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, and now the merging begins.

(310| 285| 179| 652, 351| 423, 861, 254, 450, 520) ,here 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 produces the fully sorted result. (179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

Analysis of Merge sort

If the time for the merging operations is proportional to ‘n’, then the computing time for merge sort is described by the recurrence relation.

$$T(n) = \begin{cases} a & n=1,\text{‘a’ a constant} \\ 2T(n/2)+cn & n>1,\text{‘c’ a constant.} \end{cases}$$

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 2^k<n<=2^k+1, then T(n)<=T(2^k+1).
Therefore, **T(n)=O(n log n)**

QUICK SORT

Problem Definition:

Given a list of n number stored in an array in random order, arrange them in sorted order by recursively choosing a partition value from the given n numbers and separate the numbers into less than and greater than of partition value.

Method of solution:

For the given n number in the array, choose one value, called it as partition Value . Separate the numbers such that values less than the partition are pushed in front of partition value and the value greater than the partition value are pushed back of it.ie.,the given problem(P)is divided into two sub problems P1&P2.The elements of P1 are less than the partition value &The element of P2 are greater than the partition value is in the right location of the sorted sequence of numbers. The above procedure is repeated for numbers in problem. P1&P2(ie.,) choose another partition value from the elements of P1 and separate the elements P1 such that value less than new partition value are pushed in front of it and values greater than new partition values are pushed

back of it. Now the old partition value and new partition value are in their right location of the sorted sequence. By repeating the above process, in any sub problem, values less than the partition values are pushed in front and the values are greater than the partition value are pushed back and the partition value is placed in at the right location of the sorted sequence. At the end of the above process, all the location of array will have sorted sequence of numbers. . The following are the two algorithm, one for partition the values. Another for quick sort using for algorithm.

Algorithm Partition(a ,m, p)

// list of numbers in the array from a[m]to a[p]**//**

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

```
        a[m]=a[j]; a[j]=v;
        return j;
}
```

Algorithm Quick sort(p ,q)

// Sort the numbers from a[p]to a[q] **//**

{

```
if(p<q)then
        {
                J=partition(a, p,q+1);
                Quick sort (p,j-1);
                Quick sort(j+1,q);
        }
}
```

Analysis:-

Time analysis T(n)=O(n log n)

Space analysis s(n)=O(log n)

Example:-

a: 1 2 3 4 5 6 7 8 9 10

65 70 75 80 85 60 55 50 45 infinity

65 45 75 80 85 60 55 50 70 infinity

65 45 50 80 85 60 55 75 70 infinity

65 45 50 55 85 60 80 75 70 infinity

65 45 50 55 60 85 80 75 70 infinity

60 45 50 55 65 85 80 75 70 infinity

45 50 55 60 65 85 80 75 70

45 50 55 60 65 85 80 75 70

45 50 55 60 65 85 80 75 70

45 50 55 60 65 80 75 70 85

45 50 55 60 65 70 75 80 85

Selection sort

Problem definition:

Given a list of n number stored in an array A is random order, arrange them in a stored order by recursively finding the K^{th} smallest element.

Method of solution

This method uses the partition technique to find the K^{th} smallest element, the given number are partitioned if the partition 'v' is placed at a[j] the j-1 element are less than are equal to a[j] and n-j element are greater than are equal to a[j], if K is less than j the K^{th} smallest element is in a[1] to a[5] (or)a[j-1].if k=j then a[j] is the K^{th} smallest element . If k>j, the K^{th} smallest element is the (k-j)th is smallest element in a[j+1]to a[n].

The following is the algorithm that finds the K^{th} smallest element of the given elements and places in the position 'k' and remaining element such that a[i]<_a[k] for 1<_ 1<_k and a[i]>_ a[k] for k<i<_n. the following of two algorithm for partitioning the elements and selection sort.

Algorithm Partition (a, m, p).

// list of number in the array from a[m] to a[p]**//**

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

```
        a[m]=a[j];   a[j]=v;
return j;
}
```

Algorithm Selection Sort (a, n, k)

// n number are stored in array from a[1] to [n] and Kth smallest value has to be found among them**//**

```
{
low =1 ; up=n+1; a[n+1]=infinity;
repeat
        j=partition (a, low, up);
        if  (k==j) then return j;
        else
        if  (k<j)  then up=j;
        else
                low =j+1;
until(false);
}
```

Analysis:-

The computing time T(n)=O(n)

The space needed is O(1).

Problem:-

a : 1 2 3 4 5 6 7 8 9 10

Let k=7 ie., find 7^{th} smallest element

Let partition value v=65

65 45 75 80 85 60 55 50 70 infinity

65 45 50 80 85 60 55 75 70 infinity

65 45 50 55 85 60 80 75 70 infinity

65 45 50 55 60 85 80 75 70 infinity

60 45 50 55 65 85 80 75 70 infinity

J=5

60 45 50 55 65 70 80 75 85 infinity

So the 9^{th} smallest value is found out

Still 7^{th} smallest is not found.

STRASSENS MATRIX MULTIPLICAION

Problem definition:

Given two matrices A and B of dimension n * n where n is a power of 2, multiply the matrices using divide and conquer method such that lesser number of arithmetic operations are used than the regular multiplication method.

Method of solution:

Let A and B be the two n*n matrices . The product matrix C=A * B is calculated by using the regular formula, C (i ,j)= A(i,k) B(k,j) for all 'i' and and j between 1 and n. Divide and conquer method suggest another way to compute the product of n*n matrix where n is a power of 2.

Strassens matrix multiplication formula based on Divide and conquer method for a 2 * 2 matrix is

P= (A11+A22)(B11+B22)

Q= (A21+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+S
C22=P+R-Q+U

The above formula uses only 7 multiplication and 10 addition/subtraction operations which is less than regular method of matrix multiplication of 2 * 2 matrix.

The overall computing time T(n) of this technique is given by the recurrence relation

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

$= 7T(n/2)+an^2 \quad n>2$ where a &b are constant

Working further the above relation , it is found that **$T(n)=O(n^{2.81})$**

Unit 3. GREEDY METHOD

GENERAL METHOD:

Most of the problem solved using the greedy method has n input. Every problem has one objective function and some constraints. The aim is to find the subset of the given n input which satisfy the constraints and maximises or minimises to the given objective function. From the given inputs there could be more than one subset that satisfy the given constraints. All these subset are called the feasible solution . The optimal solution for the problem is one among the collection of feasible solutions. The Greedy method suggest that one can devise an algorithm that works in stages by considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution. This is done by considering the input in an order determined by some selecting procedures. If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution. Otherwise it is added. The selection procedure is itself is based on some optimisation measure. This measure may be the objective function. They are two versions of greedy method namely .Subset paradigm and ordering paradigm. The above said method is subset paradigm greedy method. The control abstraction for the subset paradigm greedy method is given below.

```
ALGORITHM GREEDY (a,n)
{
// a[1]…..a[n] contains n inputs//
solutions=ϕ; //Initialise the solutions
for (i=1;i<=n;i++)
{
        x=select (a);
        if feasible (solution .x)  then
                solution=Union (solution .x);
}
return solution;
}
```

The above algorithm the function Select selects an input from the array a[] and removes it. The selected input value is assigned to x . The function Feasible is a Boolean valued function that determines whether x can be included into the solution set. The function Union is combines x with the solution set and update the objective function.

For problem that do not call for the solution of an optimal subset. The decisions are made by considering the input in some order. Each decision is made using an optimisation criterion that can be computed using decisions already made. This kind of greedy method is called as ordering paradigm greedy method.

KNAPSACK:

Problem Description :

Given n objects and a knapsack with a weight capacity of m. Each of the n objects have a profit value Pi,$1 \le i \le n$,`and a weight Wi,$1 \le i \le n$. If a fraction xi,$0 \le xi \le 1$ of object i is placed into the bag then the profit PiXi is earned. The knapsack problem is to maximise

$\sum 1<i<n$ PiXi subject to $\sum 1 \le i \le n$ WiXi $\le m$.

EXAMPLE 1: Solve the following KNAPSACK problem.

N=3 m=20

(P1,P2,P3) = (25,24,15)

(W1,W2,W3) = (18,15,10)

X1	X2	X3	∑WiXi	∑PiXi
½	1/3	¼	9+5+2.5=16.5	12.5+8+3.8=24.3
(No particular selection method)				
1	2/15	0	18+2+0=20	25+3.2+0=28.2
(Decreasing order of profit)				
0	2/3	1	0+10+10=20	0+16+15=31
(Increasing order of weight)				
0	1	½	0+15+5=20	0+24+7.5=31.5

(Decreasing order of P/W)

Maximum profit is earned if the items are selected in the decreasing order of Pi/Wi. So the optimal solution is (0,1,1/2) of the given 3 items.

Example 2: Find the optimum solution for the knapsack problem

N=7 m=15

(P1,P2,P3,P4,P5,P6,P7) = (10,5,15,7,6,18,3)

(W1,W2,W3,W4,W5,W6,W7) = (2,3,5,7,1,4,1)

SOLUTION:

Select items in decreasing order of profit such that ∑WiXi = m

X1	X2	X3	X4	X5	X6	X7	∑WiRi∑PiXi	
1	0	1	4/7	0	1	0	4+5+2+4=15	10+15+7+18=50

Select items in increasing order of weight such that ∑WiXi = m

1	1	4/5	0	1	1	1	1+1+2+3+4+4	3+6+10+5+18+12=54

Select items in decreasing of profit value

1	2/3	1	0	1	1	0	1+2+4+5+1+2	6+1+18+15+3+3.3=55.3

To optimal solution is

X1	X2	X3	X4	X5	X6	X7
1	1	1	0	0	1/5	0

And the profit is 55.3

Example 3:

Find the optimal solution when

n=5 m=30 (p1,p2,p3,p4,p5)=(10,12,14,16,18) (w1,w2,w3,w4,w5) = (4,2,3,1,15)

(x1,x2,x3…….x5) = (1,1,1,1,1,)

Pn = 10+12+14+16+18 = 70

Lemma_1

For the given n elements, if the sum of weight is less than the capacity m of the knapsack i.e… $\sum wi \leq m$ then

Xi=1 for $1 \leq I \leq n$

Lemma_2

All the optimal solutions will fill the knapsack exactly.

Theorem

Items selected in the order

P1/w1>p2/w2>p3/w3…….>pn/wn will generate the optimal solution for Greek Knapsack Problem.

Proof

Let x=(x1,x2,x3…….xn) be the solution generated by the above statement. It all xi are equal to 1 then the above solution x=(x1,x2,x3…….xn) is optimal. If not all xi=1 let j be the least index such that $xj \neq 1$. From the above algorithm it follows that

Xi=1 for $1 \leq i \leq j$

Xi=0 for $j < i \leq n$

And $0 \leq xj \leq 1$

Let y=(y1,y2,y3……..yn)be an optimal solution

Since y is an optimal solution , $\sum wiyi=m$. Let k be least index such that $yk \neq xk$

If k<j then xk=1 but $yk \neq xk$. So yk<xk

If k=j then since $\sum wixi=m$ and yi=x1 for $1 \leq I \leq j$, it follows that either yk<xk or $\sum wiyi>m$

If k> j , $\sum wiyi > m$ which is not possible.

Since it is proved that yk<xk, increase yk to xk and decrease as many of (yk+1,……,yn)as necessary so that ∑wiyi=m is still satisfied. Let the modified y be called as Z where Z=(z1,z2,…….zn) with zi=xi, 1 ≤ i ≤ k and ∑k<i≤n wi(yi-zi)=wk(zk-yk).

Then for z.

$$\sum 1< i \le n\ pi\ zi = \sum 1<= i \le n\ pi\ yi+(zk-yk)wkpk/wk - \sum k< i \le n\ (yi-zi)wipi/wi$$

$$\geq \sum 1< =i \le n\ pi\ yi+[(zk-yk)wk-\sum k< i \le n\ (yi-zi)wi]\ pk/wk$$

$$= \sum 1< =i \le n\ pi\ yi$$

If ∑ pi zi ≥∑ pi yi, then y could not have been an optimal solution

If ∑ pi zi=∑ pi Yi, then either z=x and x is optimal, or z≠x

Repeated use of above argument will either show y is not optimal , or transform y info x and hence x is also optimal.

Job sequencing with dead lines

Problem Description

Given n jobs and one machine to process the job. Each job has a profit value and a dead line time. A job can be processed in the machine for only one unit of time. A profit from the job is earned only if the job is completed in its dead line time. The aim is to get a maximum profit by completing the jobs within their dead time.

Feasible Solution

A set j of the given jobs in which all the jobs are completed in their dead line.

Optimal Solution

A Feasible solution of given jobs giving maximum profit i.e.. ∑ i<j pi is maximum where J is a feasible solution.

Greedy Selection of Jobs

Select jobs in decreasing order of profit but still they are completed within their dead line.

Example:

n=4 (p1,p2,p3,p4)=(100,10,15,27)

(d1,d2,d3,d4)=(2,1,2,1) Find the optimal solution.

Solution:

Feasible Solution of Jobs	Processing Sequence	Profit Earned
(1,2)	2,1	10+100=110
(1,3)	1,3 or 3,1	100+15=115
(1,4)	4,1	27+100=127
(2,3)	2,3	10+15=25
(3,4)	4,3	27+15=42
(1)	1	100=100
(2)	2	10=10
(3)	3	15=15
(4)	4	27=27

The optimal solution J={1,4} as it gives the maximum profit 127

Example 2:

n=5 (p1,p2,p3,p4,p5)=(20,15,10,5,1)

(d1,d2,d3,d4,d5)=(2,2,1,3,3) Find the optimal solution giving maximum Profit.

Solution:

1→ 1→20

2→ 2→15

3→ 4→5

40 The optimal solution is (1,2,4) and profit is 20+15+5=40

Theorem

The Greedy selection of jobs sequencing problem always gives a optimal solution. i.e.. Prove that jobs selected in decreasing order of profit meeting that dead lines gives an optimal solution.

Proof

Let there be n jobs with profit pi,$1 \leq i \leq n$, and dead lines di,$1 \leq i \leq n$. Let I be the set of jobs selected by Greedy method. Let J be set of jobs with optimal solution. Now it need to be proved that both I and J have same profit value and hence I is also optimal. If I=J then nothing to prove. So let I≠J.

If JC I then I has more profit, so I cannot be optimal. Also due to greedy selections I≠J. so there exists job 1 and job 2 such that job 1 ∈ I, job ∈ J, job 2 ∈ J, job 2 ∈ I. let A be the highest profit job such that a∈ I and a∈ I and ∈ J. Then Pa>Pe for all jobs b∈ J,b∈ I. This is true because if Pb>Pa then the greedy selection would have considered b for inclusion I before a in I.

Let SI and SJ be feasible schedules of I and J respectively. Let I be a job such that

i ∈ I and i ∈ J. let I be schedule in [t,t+1] in SI and [^{tl},^{tl}+1] in SJ. It t<t^{l} then interchange any job scheduled [^{tl},^{tl}+1]in J with I. If no job is scheduled [^{tl},^{tl}+1] in I then I is moved to [^{tl},^{tl}+1].

The resulting schedules is also feasible. It t1<t then a similar transformation can be made in Sj. After transformation like the above, let Si1 and Sj1 are the new schedule of jobs in which jobs common to I and J are placed in the same time slot. Consider the time interval [ta, ta+1] in Si in which job, ‘a’ is placed. Let b be the job scheduled in Sj in the time interval. Then by the native of selection of job ‘a’, Pa ≥ pa. Scheduling job a in [ta,ta+1] in Sj and discarding job b in Sj1 gives a feasible solution for the job set J1=J-{b} U {a}. Clearly, the profit through J1 is better than J does.

By repeatedly doing the above transformation, J can be transformed into I with no less of profit value. So I must be optimal.

OPTIMAL STORAGE ON TAPES

PROBLEM DESCRIPTION:

Given n programs, each with length li, $1 \le i \le n$. Also given a magnetic tape of length l in which the program are stored in the order I=i1,i2,i3,…….. in. Let tj be the time taken to retrieve the program ij. The mean retrieval time (MRT) of the programs is $(1/n) \sum 1 \le j \le n$ tj. Identify the ordering of the program in the table which minimises the MRT.

Minimises $d(i) = \sum 1 \le j \le n \quad \sum 1 \le k \le j \quad lik$

GREEDY SELECTION:

Choose programs in ascending order of length

CONTROL ABSTRACTION:

```
// n numbers of program to stored in m number of tapes
{
    J=0; // Next tape to store on
    for(i=1; i ≤ n ; i++)
        j=(j+1) nod m;
}
```

EXAMPLE 1:

Identify the optimal ordering of n=3 program with length (l1,l2,l3) = (5,10,3) to be stored in a magnetic tape.

SOLUTION:

Since 3!=6 there are 6 possible ordering they are

ORDERING I	D(I)
1,2,3	5+5+10+5+10+3=38
1,3,2	5+5+3+5+3+10=31
2,1,3	10+10+5+10+5+3=43

2,3,1	10+10+3+10+3+5=41
3,1,2	3+3+5+3+5+10=29 → minimum
3,2,1	3+3+10+3+10+5=34

`

So that optimal ordering is (3,1,2)

EXAMPLE 2:

N=5 (l1,l2,……..l5)=(12,5,8,32,7)

Let some random order be as given aboue itself

i.e. I=12,5,8,32,7

d(I) = 12+12+5+12+5+8+12+5+8+32+12+5+8+32+7

= 12+17+25+57+64

=175

Consider the greedy order i.e program in increasing order of length I= 2,j,3,1,4

D(I) = 5+5+7+5+7+8+5+7+8+12+5+7+8+12+32

= 5+12+20+32+64

= 133

Optimum order is (2,5,3,1,4)

THEOREM:

If l1≤ l2 ≤ …≤ ln then the ordering ij = j, 1 ≤ j ≤ n minimises ∑ 1 <=k<= n ∑1<= j<=k1 lij over all permutations of ij.

PROOF:

Let I= (i1,i2,i3,…..in) be any permutation of the index set (1,2,.n).

Then d(I) = n∑ k=1 k∑ j=1 lij = n∑ k=1 (n-k+1) lik then swapping ia and ib gives a permutation I ‘ with d(I’) = [∑ k (n-k=1) lik

$k \neq n$

$k \neq b$] + (n-a+1) lib + (n-b+1) lia

Subtracting d(I') from d(I)

D(I) –D(I') = (n-a+1) (lia - lib) + (n-b+1) (lib-lia)

= (b-a) (lia - lib)

>0

Hence a permutation of program not in ascending order of length cannot give minimum d(I) value. Hence the gives permutation ordering in the statement given minimum d(I) value.

OPTIMAL MERGE PATTERN :

Problem Description:

Given n sorted files for merging into a single sorted file, identify the selection of files one by one such that number of record movement is minimised.

Greedy Method of Merging:

At each step merge two smallest files together.

Example 1:

Let x1,x2,x3 are three sorted files with 30,20,10 records respectively. Merge x2 and x3 (30 record movement) to get files 2, with 30 records.

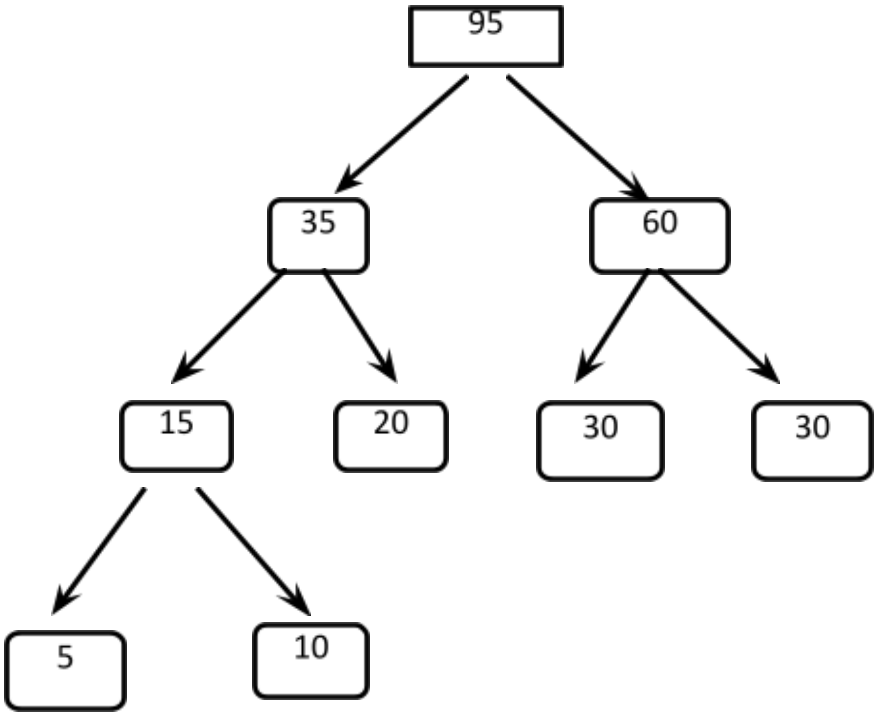
Merge z1 and x1 (60 records movement) to single merged file z2 with 60 records. Total record movement are 30+60=90

Example 2:

Let (x1,x2,x3,x4,x5)=(20,30,10,5,30) records respectively. Files x3 & x4 are with least records.

Merge x3 & x4 to get z1 (15 records,15 records movement). Files z1 and x1 are with least records.

Merge z1 and x1 to get z2(35 records, 35 record movement). Now merge z2 and z3 to get z1(95 records, 95 movements). So total minimum record movement are '15+35+60+95=205'. The binary tree for above solution is given below.



Optimal Merge Pattern Problem

Control Abstraction:

Struct treenode

{int weight;
Treenode *lchild, *rchild;
}
Algorithm Tree(n)
{
//n sorted files to be merged//

```
For(i=1;i<=n-1;i++)
        {
Pt=new treenode;
Pt   lchild=least(n);
Pt   rchild=least(n);
Pt   wieght=pt   lchild   weight + pt   rchild   weight,
Insert(n,pt);
        }
}
```

Single Source Shortest Path Problem

Problem Description

Given a directed, weighted graph G=(V,E) where v is a set of vertices of graph and E is a set of edges of the graph , the problem is to identify the shortest path from a given source vertex to remaining vertices of the graph.

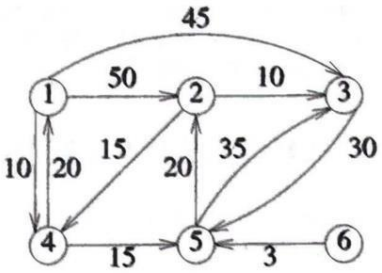
Greedy Selection:

Next vertex to be selected for the shortest path is the vertex gives minimum increment to the shortest path generated so far.

Example:

Single-source Shortest Paths

- Example



(a) Graph

Path	Length
1) 1, 4	10
2) 1, 4, 5	25
3) 1, 4, 5, 2	45
4) 1, 3	45

(b) Shortest paths from 1

Control Abstraction:

Algorithm shortest path(V,cost,dist,m)

{

//n is the number of vertices of grapfh G. V is the source vetex for which shortest path to be found to other vertices.

Cost is the e dimentional adjacency matrix cost[n][n] dist[j], 1<=j<=n is the length of the shortest path from vertex v to vertex j//

For(i=1;i<=n;i++)

{

S[i]=false; dist[i]=cost[v,i];

}

S[v]=true; dist[v]=0;

For (num=2;num<=n-1;num++)

{

Choose u such that u!ϵs and dist[u] is minimum

s[u]=true;

for(each w adjacent to u with s[w]=false)

do

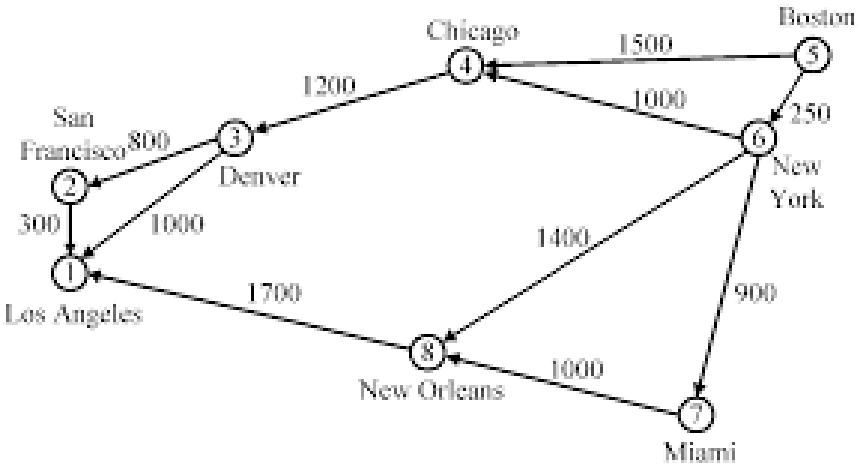
if(dist[w]>dist[u]+cost[u,w])

thendist[w]=dist[u]+cost[u,w]

}

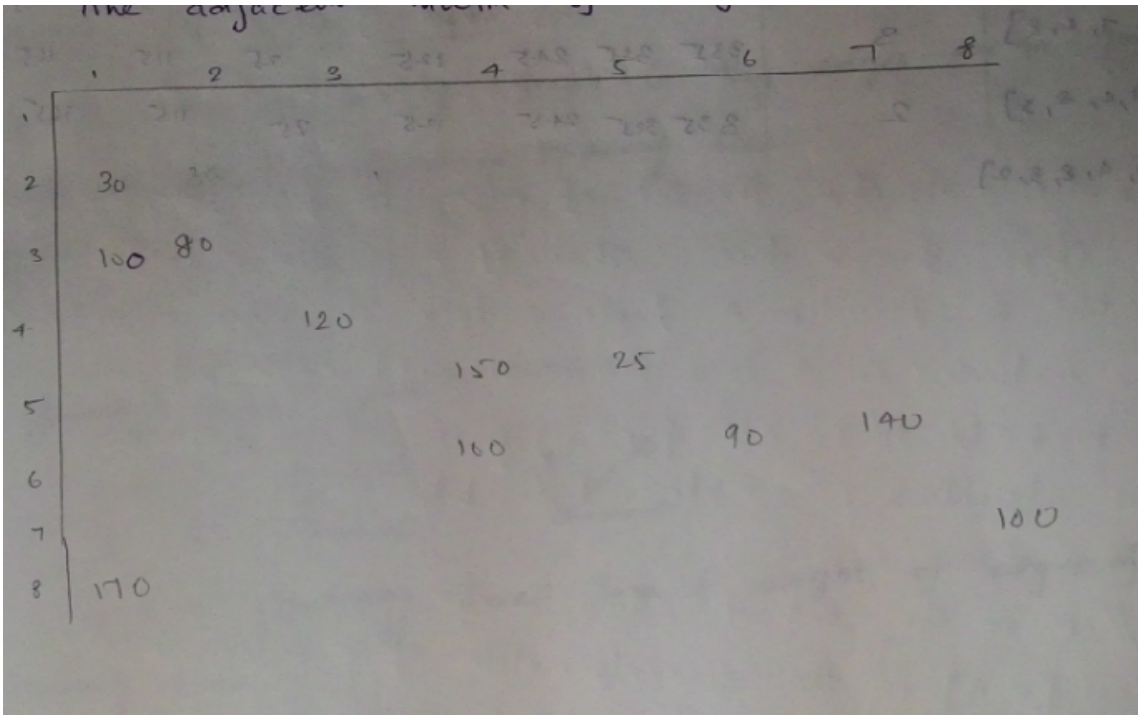
}

Example 2: Solution based on greedy Strategy:



Aim: To find shortest path from vertex 5 to remaining vertices.

The adjacent matrix of the graph is given below



Action of shortest path by greedy Strategy:

Solution (s)	Vertex Selected	1	2	3	4	5	6	7	8
	-----				150		250		
[5]	6				125		25	115	165
[5,6]	7				125		25	115	165
[5,6,7]	4			245	125		25	115	165
[5,6,7,4]	8	335		245	125		25	115	165
[5,6,7,4,8]	3	335	325	245	125		25	115	165
[5,6,7,4,8,3]	2	335	325	245	125		25	115	165
[5,6,7,4,8,3]									

Minimum Cost Spanning Tree

Problem Description:

Given a connected graph G=(V,E) where V is the set of vertices and E is a set of edges identify the Spanning tree of the graph G whose cost is minimum using Greedy selection of edges of the graph G.

Spanning Tree of a Graph:

1. If n vertices are in graph G then n-1 edges are in spanning tree.
2. No cycles are involved in spanning tree.

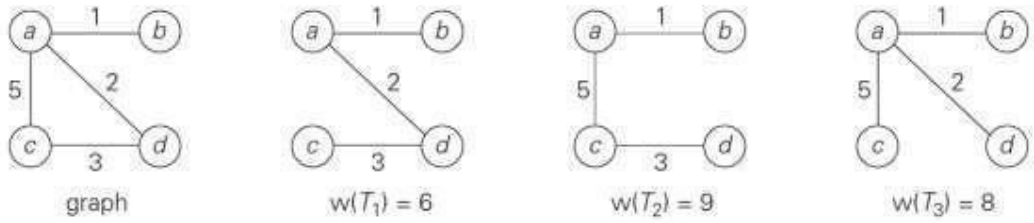


FIGURE 9.2 Graph and its spanning trees, with T_1 being the minimum spanning tree.

Cost of Spanning Tree: Sum of weight of edges of Spanning tree.

Application:

1.Used to get an independent set of circuit equations for an electric network.

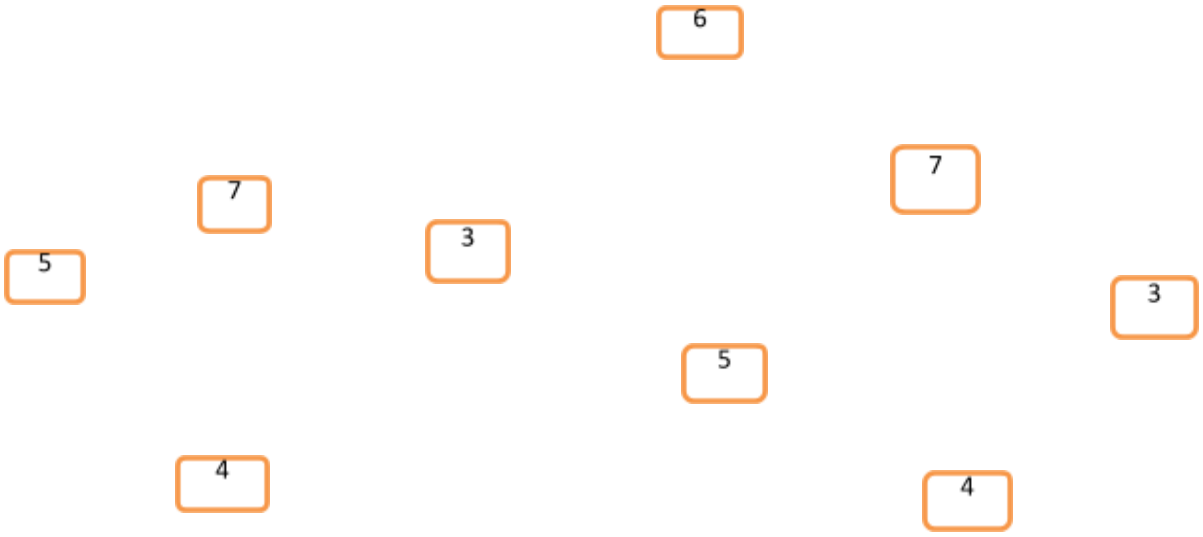
2.It vertices represent cities and edges represent communication link, then Spanning trees represent minimum number of communication links needed to connect all cities.

Greedy Selection of Edges:

Select an edge that result in a minimum increase in the sum of the cost of edges so far included.

Prims Algorithm:(Based on Greedy Selection of edges)

1 1 2 2 6



In algorithm, the set of edges so far selected at any stage form a tree. Thus if A is the set of edges selected so far, then A forms a tree. The next edge (u,v) to be included in A is a minimum cost edge not in A with the property that AU{ (u,v) } is also a tree. The princes algorithms is implemented below.

Algorithm Prims (E,cost,n,t)

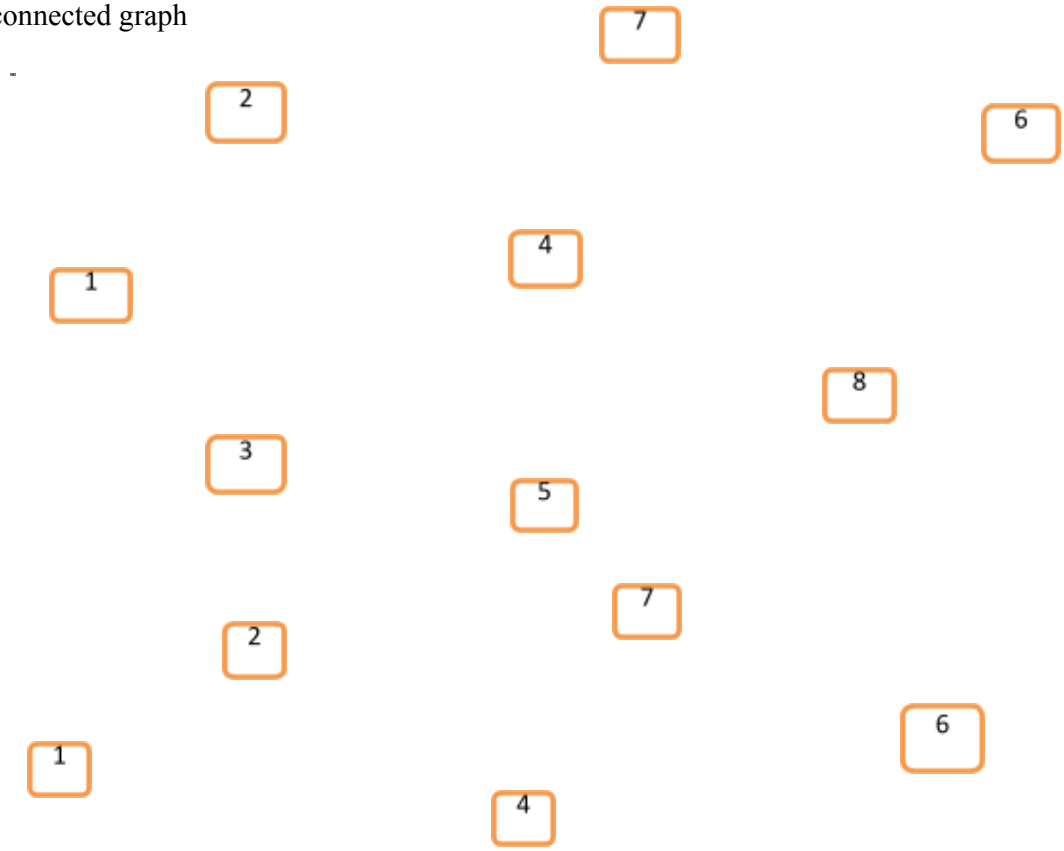
{ // E is the set of edges in G. Cost[n,n]is the adjacenty matrix of n vertex graph G.K[n-1,2]will contain the edges of minimum cost spanning tree // Let (k,l) be an edge of minimum cost in E

```
Mini cost=cost[k,l]
t[1,1]=k; t[1,2]=l;
for(i=1;i<=n;i++)
        if(cost [i,l] < cost [i,k] )
                near[i]=l
        else
                near[i]=k;
near[k]=near[l]=0;
for(i=2;i=n;i++)
{ //find n-2 additional edges for t //
  Let j be an index such that near [i]!=0 and cost [j,near[j] ] is minimum;
t[I,1] = j; t[I,2] =near[j;
mini cost=minicost +cost (j near[j]);
near [j]>=0
```

```
        for(k=1;k<=n;k++)
                if(near [k]!=0) and (cost[k] ,near[k])>(cost [k,j]));
                then near[k]=j;
        }
        returnminicost;
}
```

Problem :

Generate the minimum cost spanning tree and find its cost for the following connected graph



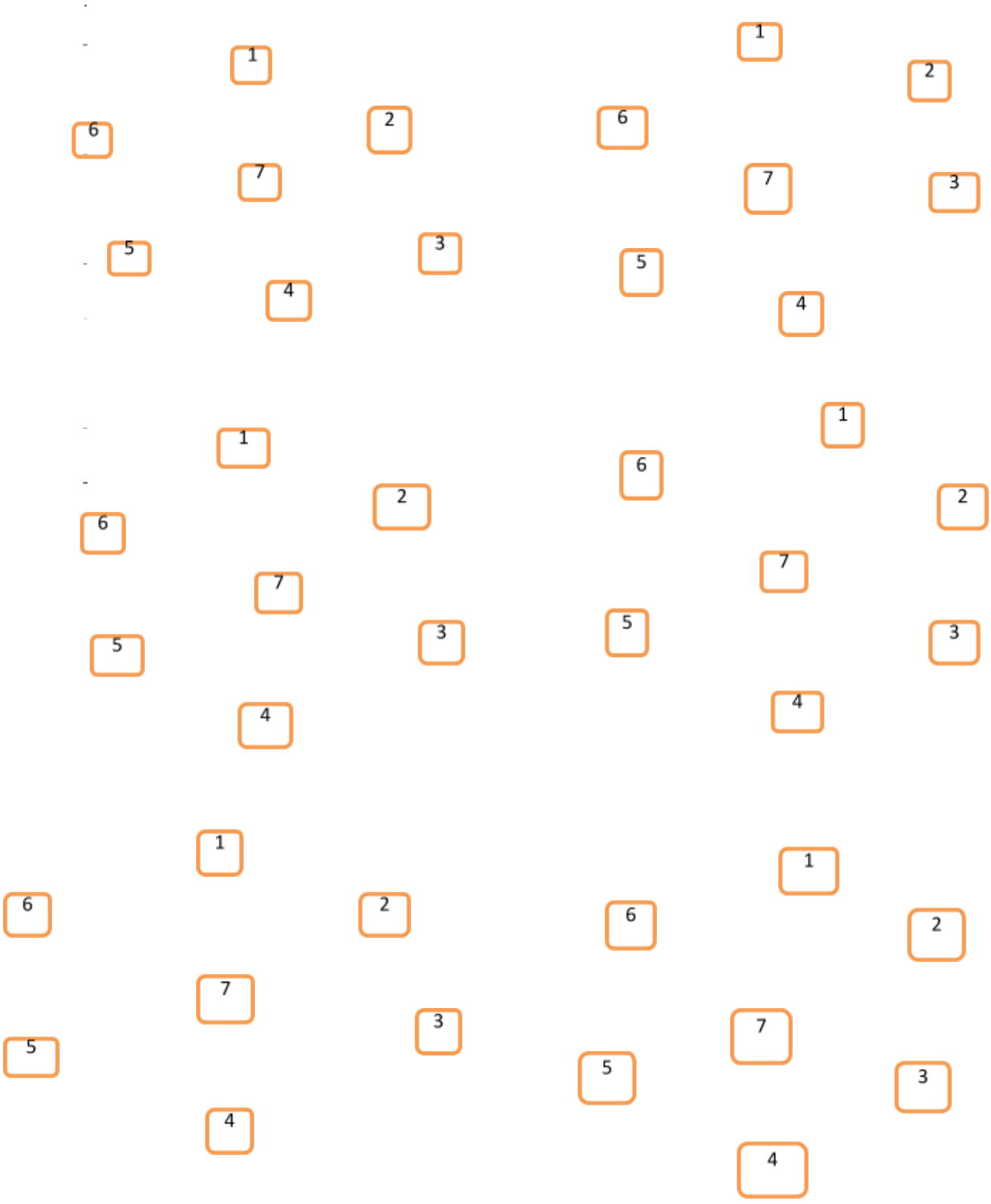
Cost of spanning tree is u+2+5+7+21+6+8=60//

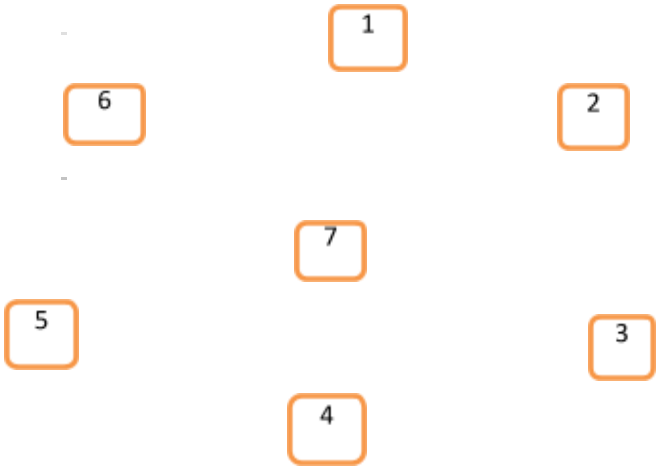
Kruskal algorithm for minimum cost spanning tree :

This algorithm suggest that "The set t of edge so for selected at any stage be such that it is possible to complete t into a tree".

Example : 1

Generate minimum cast spanning tree for the following graphs.



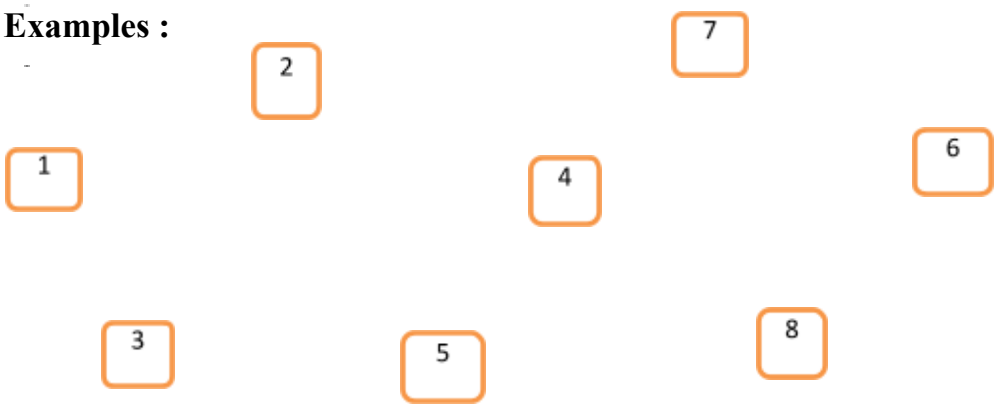


The kruskal algorithm is implement as follows.

```
Algorithm kruskal (E ,cost,n,t)
{       //E is a set of edges
Cost is adjacency matrix of g
N is number of vertices of  G
T is contain edges of spanning tree//
Contruct a heap out of edge costs
        For(i=1;i<=n;i++) parent [i]=-1;
        i=0; mlncost=0;
        while((i<n-1) and heap not empty)
        {       Delete a minimum cost edge (u,v) from the heap and do reheap.
        J=find (u) ; k=find(v);
        If(j!=k)
        {
        I=i+1;
        T[l,1]=n; t[I,2]=v;
        Mincost = mincost +cost [u,v];
        Union (j,k);
        }
        If(i!= n-1) write (“No spanning tree ”) ;
        Else
        Return mincost;
```

}

Examples :



Single Source Shortest Path Example

5->6>->7

	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	10	80	0					

4			120	0				
5				150	0	25		
6				100		0	90	140
7							0	100
8	170							0

S	Vertex selected	1	2	3	4	5	6	7	8	
---	----				150	0	25			
5	6				125	0	25	115	165	
5,6,	7				125	0	25	115	165	
5,6,7	4			245	125	0	25	115	165	
					125	0	25	115	165	
					125	0	25	115	165	
					125	0	25	115	165	
					125	0	25	115	165	
						0				

Unit 4 Dynamic Programming

General method of problem solving using Dynamic Programming

Dynamic programming method can be used when solution to a problem can be viewed as a result of a sequence of decisions. For example the solution to Kanapsack problem can be viewed as a sequence of decisions on items Xi. Dynamic programming avoids some decision sequences that cannot be an optimal solution. The difference between Greedy and Dynamic Programming method is that in Greedy method only one decision sequence is generated but in Dynamic Programming multiple decision sequences are generated.

Definition : Principle of Optimality

An Optimal Sequence of decision has the property that whatever the initial state and decision are the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision

MultiStage Graph :

Problem description:

- G(V,E) is a directed graph

- Vertices are partitioned into $k \geq 2$ disjoint sets $V_i, 1 \leq i \leq k$

- if $(u,v) \in$ then $u \in V_i$ and $v \in V_{i+1}$ for $1 <= i < k$

- $|V1|=1$ and $|Vk|=1$

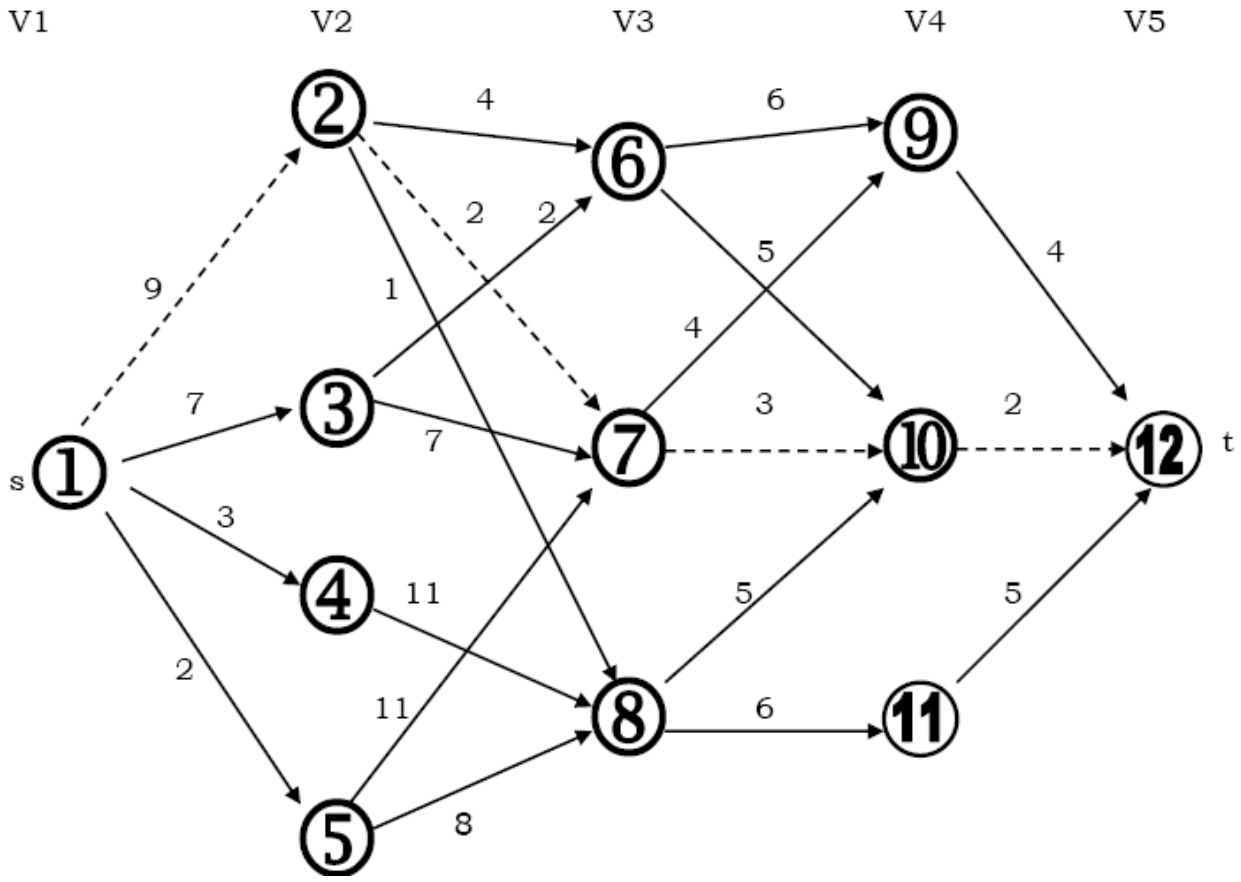
- the source vertex $s \in V_i$ and sink vertex $t \in V_k$

- C(i,j) is the cost of the edge (i,j)

- The multistage graph problem is to find minimum cost path from vertex s to t

Example

Find Minimum cost path from source s to sink t of the following graph



It is k=5 stage Graph it needs k-2 = 5-2 = 3 decisions.

the i_{th} decision includes determining which rates in

v_{i+1},1≤i≤k-2 is to be on the path the problem is reduced using the relation

COST(i. j) = min { *c(j, /*) + *COST(i* + 1, /)}
I€Vi+ 1
(j,I)€E

cost(1,1)= min(9+cost(2,2),7+cost(2,3),3+cost(2,4),2+cost(2,5))

cost(2,2)= min(4+cost(3,6),2+cost(3,7),1+cost())

cost(3,6)= min(6+cost(4,9),5+cost(4,10)

min(6+4,5+2)=min(10,7)=7

cost(3,7)= min(4+cost(4,4),3+cost(4,10)

min(8,5)=5

cost(3,8)= min(5+cost(4,10),6+cost(4,1)

min(5+2,6+5)=min(7,11)=7

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

min(11,7,8)=7

cost(2,3)= min(2+cost(3,6),7+cost(3,7)

min(2+7,7+5)=min(9,12)=9

cost(2,4)= min(11+cost(3,8))= 11+7=18

cost(2,5)= min(11+cost(3,9),8+cost(3,8)

min(11+5,8+7)=min(16,15)=15

cost(1,1)= min(9+7,7+9,3+18,2+15)

min(16,16,21,17)= 16

The Minimum Cost Path is 1→2→7→10→12

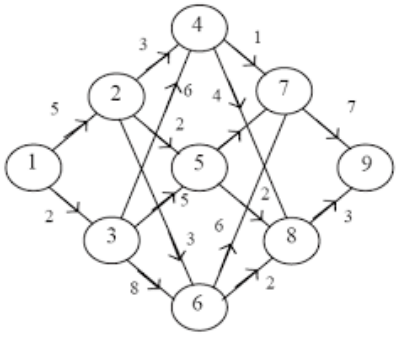
The algorithm for minimum cost path in M.S.G using forward approach is given below

```
Algorithm FGraph (G,K,n,p)
// G is a K stage Graph with n  vertices
C[i,j] is the cost of (i,j)
P[i]…P[k] is minimum cost path- //
 {
cost[n]=0.0;
```

```
        for (j=  n-1 to step-1 do
        {
        // compute cost[j] //
        let r be a vertex such the (j,r) ∈E and c[j-r] +cost[r] is minimum;
        cost[j]=cost [j,r]+cost[r];
        d[j]=r;
    }
    // Find a minimum cost path//
        p[i]=1;
        p[k]=n;
        for(j=2 to k-1 ) do p[j]=d[p[j-1]]
    }
```

Problem

Find the minimum cost and path of following graph using dynamic programming technique



cost(i,j)=min{c(j,l)+cost(i+1,l)} where l∈ v_{i+1} (j,l)∈E

Solution:

It is k=5 stage graph it needs k-2=5-2=3 decision the i^{th} decision involves determination which vertex is Vi+1 $1 \leq i \leq k-2$ is to be

The problem is solved using the recurence redation

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

cost(1,1)= min{5+ cost(2,2),2+cost(2,3)} = min { 5+7,2+7} = 9

cost(2,2)= min{3+ cost(3,4),3+cost(3,6), 2+cost(3,5)}

min{3+7,3+5,2+5}=min(10,8,7)=7

cost(3,4)=min{1+cost (4,7) 4+cost(4,8)}

min{1+7,4+3}=min(8,7)=7

cost(3,5)=min{6+ cost(4,7),2+cost(4,3)}

min{6+7,2+3}=min(13,5)=5

cost(3,6)= min{6+ cost(4,7),2+cost(4,3)}

min{6+7,2+3}=min(13,5)=5

cost(3,5)= min{16+ cost(4,7),2+cost(4,8)}

min{11+7,2+3}=min(23,5)=5

cost(2,3)= min(6+7,2+5,8+5)

min(13,7,13)=7

min{5+8,2+7}=min(13,9)=9

m.cost path= 1 3 5 89

ALL PAIRS SHORTEST PATHS

Problem description:

Let G = *(V,* E) be a directed graph with *n* vertices. Let *C* be a cost adjacency matrix for G such that *C(i,* i) = 0, 1<=i<-n. *C(i,* j) is the length (or cost) of edge (i,j) if (i,j) ∈ E(G) and *C(i,j)* = ∞ if *i* ≠ *j* and (i,j) ∈ E(G). The *all pairs shortest path problem* is to determine a matrix *A* such that *A (i,* j) is the length of a shortest path from *i* to *j.*

Method of solution :

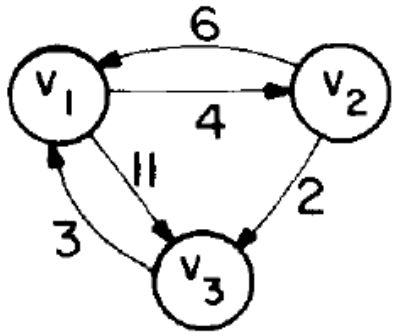
Let the graph G have no cycles with negative length. Let us examine a shortest *i* to *j* path in G, *i* ≠*. j.* This path originates at vertex *i* and goes through some intermediate vertices (possibly none) and terminates at vertex *j.* If *k* is an intermediate vertex on this shortest path then the subpaths from *i* to *k* and from *k* to j must be shortest paths from *i* to *k* and *k* to *j* respectively. Otherwise, the *i* to *j* path is not of minimum length. So, the principle of optimality holds.. If *k* is the intermediate vertex with highest index then the *i* to *k* path is a shortest *i* to *k* path in G going through no vertex with index greater than *k* - 1. Similarly the *k* to *j* path is a shortest *k* to *j* path in G going through no vertex of index greater than *k* - 1. We may regard the construction of a shortest *i* to *j* path as first requiring a decision as to which is the highest indexed intermediate vertex *k.* Once this decision has been made, we need to find two shortest paths. One from *i* to *k* and the other from *k* to j. Neither of these may go through a vertex with index greater than k - 1. Using $A^k(i,$ J) to represent the length of a shortest path from *i* to j going through no vertex of index greater than *k, we get*

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

The above Recurrence relation may be solved for A^n by first computing A^1, then A^2 , then A^3 , etc. Since there is no vertex in G with index greater then *n,* $A(i,$ j) = A^n *(i,* j). The following Procedure ALL_PATHS computes $A^n(i,$ j).

Algorithm *ALLPATHS(COST, A,* n)
//COST(n, n) is the cost adjacency matrix of a graph with *n ver*tices; *A(i,j)* is the cost of a shortest path from I to j
{
for *i* = 1 to *n* do
for *J* = 1 to *n* do
A(i. j) = *COST(i,* j) //copy COST into *A! I*
for k = 1 *to n* do *I !for* a path with highest vertex index *kl I*
for *i* = 1 to *n* do *I !for* all possible pairs of vertices/ *I*
for *J* = 1 to *n* do
A(i,j) = min{A(i,j), *A(i,* k) + *A(k,j)}*
}

Example : Find the shortest path between any two vertices of the following graph



(a) *G*

	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) Cost Matrix for *G*

Solution:

$A^{(0)}$	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

$A^{(1)}$	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

$A^{(2)}$	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

$A^{(3)}$	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

Single Source Shortest path

Problem Description:

Given a directed graph G with no negative edge cycle, the problem is to find shortest length between source vertex v to all remaining vertices

Method of Solution:

Let dist [u] be the shortest length between source vertex v and the vertex u with l edges then dist[u]=cost[v,u] 1≤u≤n

The relation that finds shortest length between vertex v and any other vertex with k edges between them is

$$dist^{k}[u] = mins\{dist^{k-1}[u],\ mins\{dist^{k-1}[i] + cost[i,u]\}\}$$

where i is the vertex other then u,

```
Algorithm bellmanford(v,cost,dist,n)
{
    for(i=1;i≤n;i++) dist[i]=cost[u,i];
    for(k=1;k≤n-1;k++)
```

for each is such that u≠v and u atleast one incoming edge do for each(i,u) in the graph do

if(list[u]>dist[i]+cost[i,u])then

dist[u]=dist[i]+cost[i,u];

}

String Editing

Problem description:

→Given two strings

X=X1,X2,X3,…..,Xn

Y=Y1,Y2,Y3,….,Ym

→Xi,1≤ $i \leq n$,and Yj 1≤ $i \leq m$ are members of a symbol set

→Editing operations permitted are insertion, deletion, change

→Each editing operation has a cost

→The problem is to convert the string X into Y using Insertion, Deletion ,Change such that the total cost of editing is minimum

Method of solution

Let D(xi) be the cost of deleting a symbol xi from X

I(yj) be the cost of inserting a symbol yj into X

C(xi,yj) be the cost of changing xi of X into yj

Let cost(i,j) be the minimum cost of any edit sequence for transforming x1,x2,..xi $1 \leq i \leq n$ into y1,y2,…ym , $1 \leq j \leq m$

Compute cost (i,j) for each of i and j . Then cost(n,m) is the cost of optimal edit sequence

If i=0 and j=0 then cost(i,j)=0 since the two sequences are identical or empty.

If j=0 and i>0 then string X is transformed into string Y by the sequence of deletion . So cost (i,0) = cost(i-1,0)+D(xi)

If i=0 and j>0 then cost(0,j) = cost (0,j-1)+ I(yj)

If i≠0 and j≠0 then x1,x2,…xi can be transformed into y1,y2,…yj in one of the following three ways

1. Transform x1,x2,…xi-1 into y1,y2,…yj using a min cost edit sequence and then delete xi. The corresponding cost is cost(i-1,j)+D(Xi)
2. Transform x1,x2,…xi-1 into y1,y2,…yj-1 using a min cost edit sequence and then change the symbol xi to yj . The corresponding cost is cost(i-1,j-1)+C(xi,yj)
3. Transform x1,x2,…xi into y1,y2,…yj-1 using a min cost edit sequence and then insert Yj. The corresponding cost is cost(i,j-1)+I(y,j)

The min cost of any edit sequence that transform X1,X2,…Xi ,i>0 into Y1,Y2,…Yt , j>0 is the min of above three costs

So the recurrence solution to solve the problem based on above three points is

cost (i,j) =

$\{O \qquad if\ i = 0, j = 0\ \ cost(i - 1, o) + D(xi)\ \ if\ j = 0, i > 0\ co$

where cost' (i,j) = min{cost(i-1,j)+D(Xi),cost(i-1,j-1)+C(Xi,Yj),cost (i,j-1)+I(Yj)}

compute cost(i,j) for $0 \leq i \leq n$ and $0 \leq j \leq m$. There are (n+1)(m+1) such a values. These value are computed in the form of a two dimensional array. Each row corresponds to a particular value of i and each column corresponds to a particular value of j. The 0^{th} row can be computed first since it corresponds to performing a series of insertion, similarly the 0^{th} column is computed . Then values for the first row, second row and so on are computed.

Example

X= $x_1, x_2, x_3, x_4 x_5$= a,a,b,a,b

Y=$y_1, y_2, y_3, y_4 y_5$ = b,a,b,b

Cost of insertion is 1,deletion is 1, change is 2

Find the minimum cost of edit sequence convert X into Y

Sol:

Cost(0,0)= 0 Cost(1,o) = 1

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

Cost(0,2)= 2 Cost(3,0) = 3

Cost(0,3)= 3 Cost(4,0) = 4

Cost(0,4)= 4 Cost(5,0) = 5

Cost(1,1) = min { cost(0,1) + D(x1), Cost(0,0)+C(x1,y1), cost(1,0)+I(y1)}

= min {2,2,2}

= 2

Similarly cost(1,2), cost(1,3).....cost(5,4) are computed.

Travelling Salesman Problem

Problem Description:

G(V,E) dynamic graph with c_{ij} is the weight of the edges vertex i and vertex j

n be the number of vertices in the graph

Tour of G is a cycle having all vertices of G

Cost of tour is sum of weight of edges in the tour

The travelling salesman problem is to find a tour of minimum length / cost starting from a vertex and come back to it

Method of solution:

Let the tour start at vertex 1 and end in vertex 1. Every tour consists of an edge (1,k) for some k ∈ $V\_\{1\}$ and a path from vertex 1 goes through each vertex in v-{1,k} exactly once. If the tour is optimal then the path from k to 1 must be a shortest k to 1 path going through al vertices in v-{1,k} . Let g(i,S) be the length of shortest path vertex i , going through all vertices in S and terminating vertex 1. Then the function g(1,v-{1}) is the length of an optimal sales person tour.

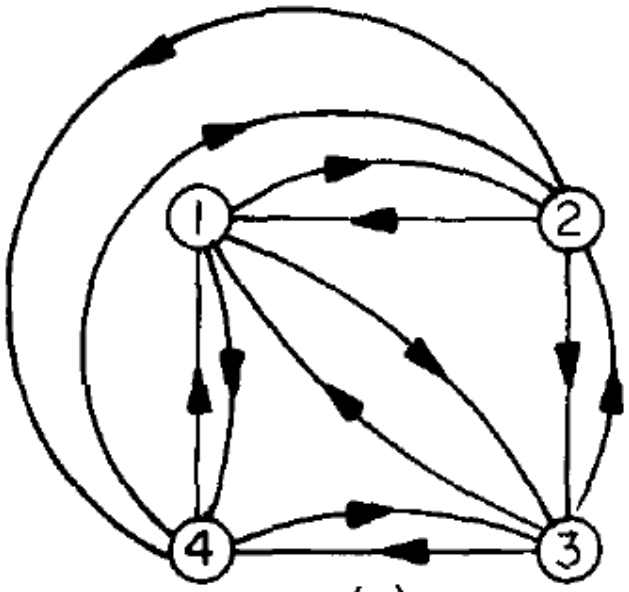
g(1,v-{1}) = $min_{2\leq k\leq n}\{c_{ik} + g(k, v - \{1, k\})\}$

generalising the above expression for all i not belonging to S

g(i,s)= min j€S{c_{ij}+g(j,s-{j})}

Example:

Find the minimum cost tour from vertex 1 of the following graph whose cost matrix is also given



(a)

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

(b)

Solution:

First compute g(i,q) for $1 \leq i \leq 4$

i.e g(1,Φ) = 0, g(2,Φ) = 5, g(3,Φ)=6, g(4,Φ)=8

i.e g(1,Φ) = c_{11}=0

g(2,Φ) = c_{21}=5

g(3,Φ) = c_{31}=6

g(4,Φ) = c_{41}=8

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

so

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

g(2,{3,4})= min(c_2,3+g(3,{4}),(c_{24} + g(4,{3})).

min(9+12+8,10+9+6)=25

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

min(13+10+8, 12+8+5)

min(31,25)=25

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

min(8+9+6, 9+13+5)

min(23,27)=23

now

g(1,{2,3,4}) = min(10+25,15+25,20+23)

min(35,40,43)=35

0/1 KNAPSACK PROBLEM:

This problem is similar to ordinary knapsack problem but a fraction of an object can not be selected. We are given ' N ' object with weight Wi and profits Pi where i varies from 1 to N and also a knapsack with capacity ' M '.
The 0/1 Knapsack problem is to fill the bag with the help of ' N ' objects and the resulting profit has to be maximum.

Unit 5- BASIC SEARCH AND TRAVERSAL TECHNIQUE *:*

BINARY TREE TRAVERSAL

Binary tee is a data structure with a unique node called root and every node has a maximum of two children and there exists a unique path from root to every other node. The nodes of a binary are displayed using inorder, preorder and post order methods. They are discussed below.

```
Treenode = record
{
        Type data;
        Treenpde *lchild,*rchild;
}
```

Inorder Traversal: Recursively print the left child , parent , right child.

```
procedure INORDER(T)
{
        if  (T !=0)  then
        {
                 INORDER(t->LCHILD)
                Print T
                INORDER(t->RCHILD)
        }
}
```

Preorder Traversal: Recursively print the parent, left child , right child.

```
procedure PreORDER(T)
{
        if  (T !=0)  then
        {
                Print T;
                 PreORDER(t->LCHILD)
                PreORDER(t->RCHILD)
        }
}
```

Postorder Traversal: Recursively print the left child , right child, parent

procedure *PostORDER(T)*

```
{
        if  (T !=0)  then
        {
                PostORDER(t->LCHILD)
                PostORDER(t->RCHILD)
                Print T
        }
}
```

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,

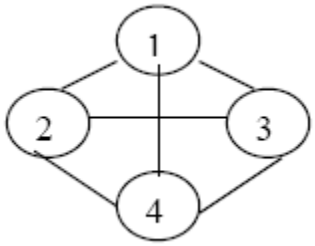


FIG: Graph G

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)}.

UNDIRECTED GRAPH: Graph in which the edges are not given arrow mark direction..

DIRECTED GRAPH: Graph in which the edges are given arrow mark direction

DIRECTED GRAPH COMPLETE GRAPH: Graph in which every node is connected with every other node.

Graph Traversal :

It means finding a path from a source vertex to a destination vertex . There are two methods for searching such a path namely Breadth First Search and Depth First Search

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.

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.
{
        u=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)
}
```

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

General method of problem solving using Backtracking method

It is one of the most general algorithm design techniques. Many problems which deal with searching for a set of solutions or for a optimal solution satisfying some constraints can be solved using the backtracking formulation. To apply backtracking method, tne desired solution must be expressible as an ntuple (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 (mi+1……..mn) possible test vectors may be ignored entirely. Many problems solved using backtracking require that all the solutions satisfy a complex set of constraints. These constraints are classified as:

i) Explicit constraints.

ii) Implicit constraints.

1) Explicit constraints:

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

Some examples are,

$Xi \geq 0$ or $Si = \{\text{all non-negative real nos.}\}$

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

$Li \leq Xi \leq Ui$ or $Si= \{a: Li \leq a \leq Ui\}$

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

2) Implicit constraints:

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

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. Bk(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.

THE 8-QUEENS PROBLEM

Problem definition

This 8 queens problem is to place n-queens in an 'N*N' matrix in such a way that no two queens attack each other.

Method Solution:

The solution vector X (X1…Xn) represents a solution in which Xi is the column of the I 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 to upper left has the same value. Suppose two queens are in same position (i,j) and (k,l) then two queens lie on the same diagonal , if and only if |j-l|=|I-k|.

```
Algorithm place (k,I)
//return true if a queen can be placed in k th row and I th column. otherwise it returns //
//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) ;
            }
        }
}
```

SUM OF SUBSETS Problem

Problem definition :

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.

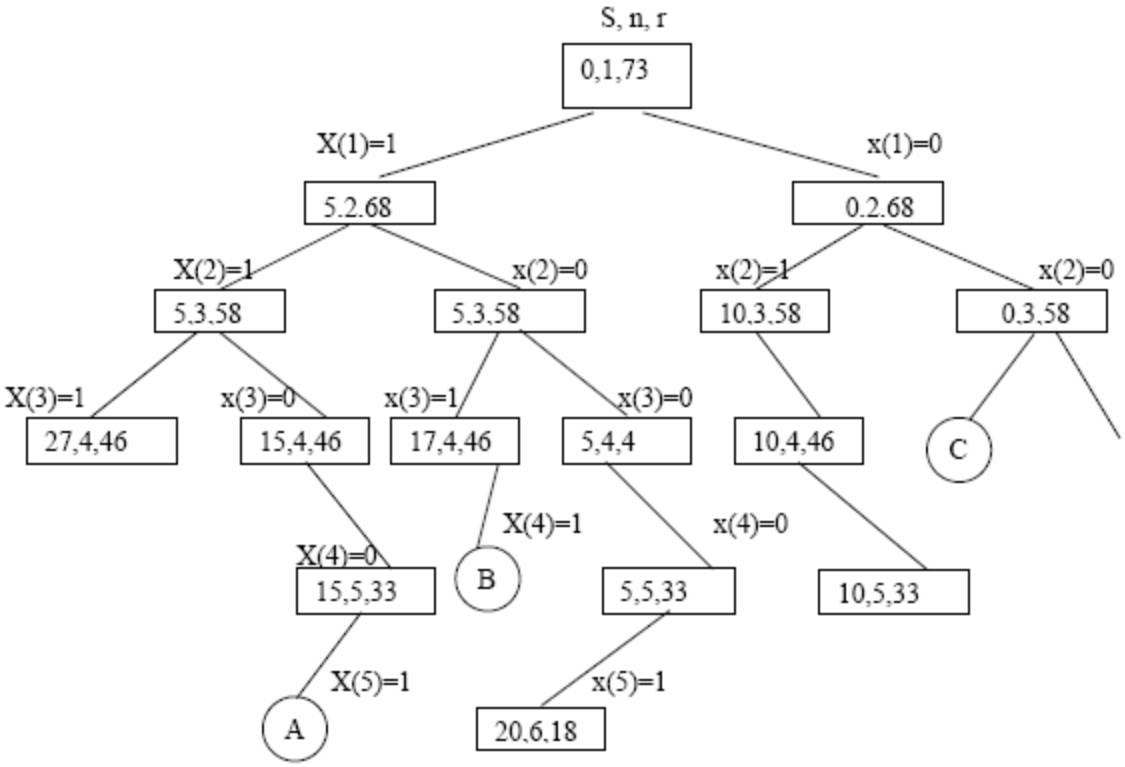
Method of solution:

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. 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:

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.

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. The state space tree for the given problem is,



I[st] solution is **A** -> 1 1 0 0 1 0
II[nd] 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 labeled with 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 include Wi.

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 check for the next k. After generating the left sub tree we have to generate the right sub tree, for this 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]);
                }
}
```

GRAPH COLORING:

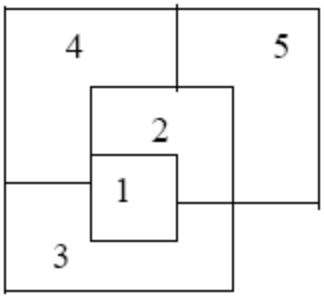
Problem definition:

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 same color yet only ‘M’ colors are used. So it’s called M-color ability decision problem.

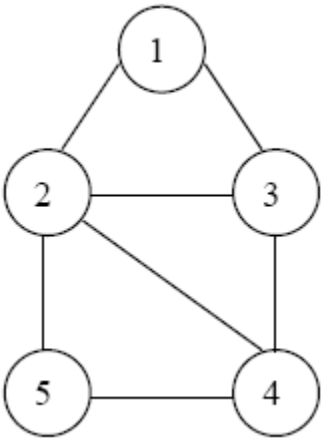
Method of solution:

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 that no 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 then the corresponding nodes are joined by an edge. Consider a map with five regions and its graph.

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



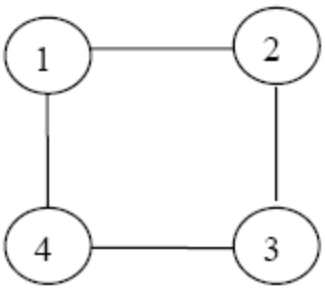
Steps to color the Graph:

First create the adjacency matrix graph(1:m,1:n) for a graph, if there is an edge between i,j then C(i,j) = 1 otherwise C(i,j) =0. The Colors will be represented by the integers 1,2,…..m and the solutions will be stored in the array X(1),X(2),……..,X(n) ,X(index) is the color, index is the node. The formula is used to set the color is,

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

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

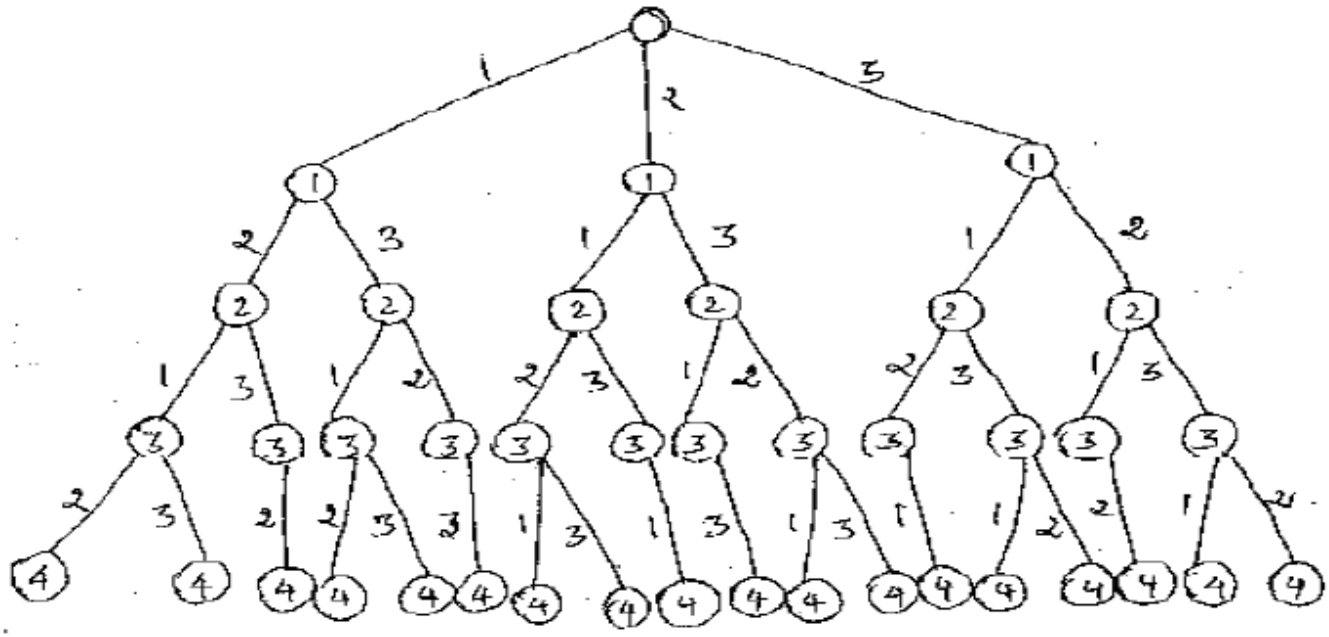
M= 3

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 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, then X[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.

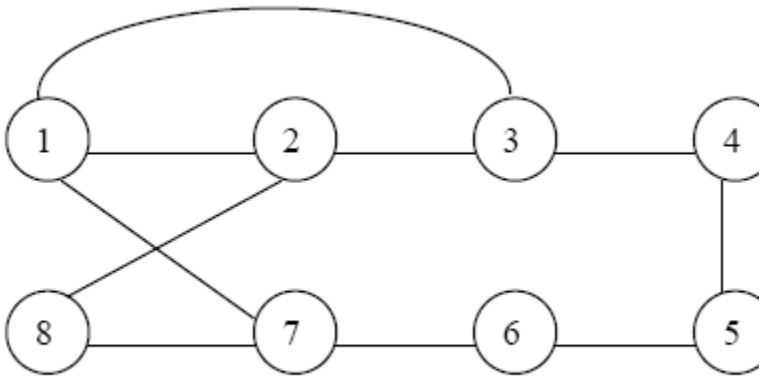
}

The time spent by Nextvalue to determine the children is θ (mn) .Total time is = θ(mn n).

HAMILTONIAN CYCLES:

Problem definition

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 once and returns to its starting position. If the Hamiltonian cycle begins at some vertex V1 belongs to G and the vertex 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.

Method of Solution

1. Define a solution vector X(Xi……..Xn) where Xi represents the I th visited vertex of the proposed cycle.
2. Create a cost adjacency matrix for the given graph.
3. The solution array initialized to all zeros except X(1)=1,b'coz the cycle should start at vertex '1'.
4. Now we have to find the second vertex to be visited in the cycle.
5. 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 above conditions are satisfied the current vertex is included in the 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 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 vertex
    If (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);
}
```