LECTURE NOTES

Subject code:CS6402 **Subject Name:Design and Analysis of Algorithms**

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.

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 or study of Algorithm:

- How to device or design an algorithm → creating and algorithm.
- How to express an algorithm → definiteness.
- How to analysis an algorithm → time and space complexity.
- How to validate an algorithm → fitness.
- Testing the algorithm → checking for error.

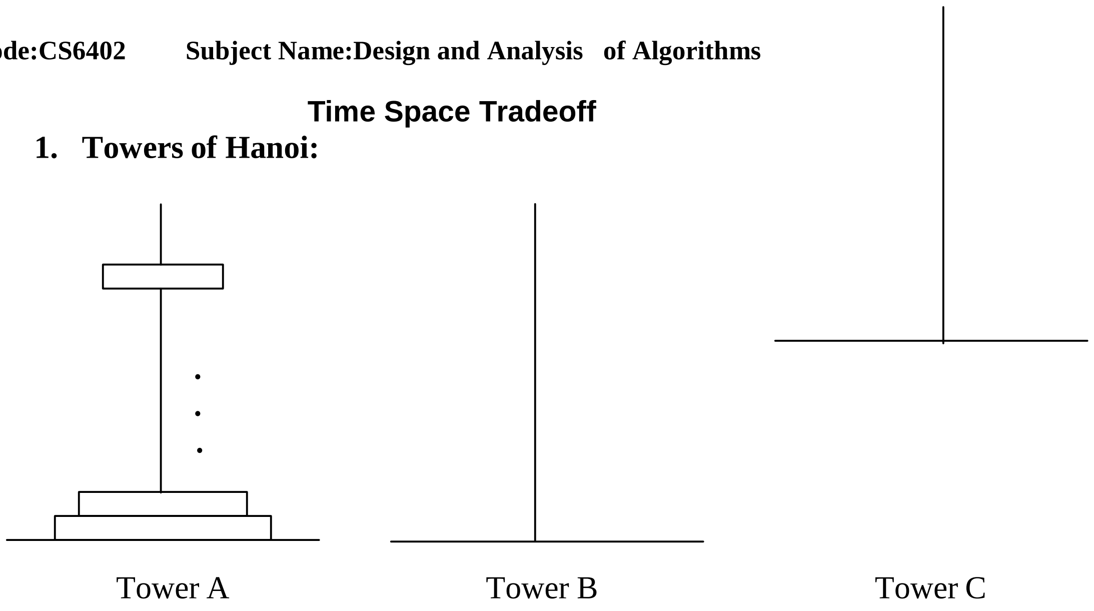
Recursive Algorithms:

- 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.

Time Space Tradeoff

1. Towers of Hanoi:



- 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.
- As the disks are very heavy, they can be moved only one at a time.
- In addition, at no time can a disk be on top of a smaller disk.
- According to legend, the world will come to an end when the priest have completed this task.
- A very elegant solution results from the use of recursion.
- Assume that the number of disks is ‘n’.
- To get the largest disk to the bottom of tower B, we move the remaining ‘n-1’ disks to tower C and then move the largest to tower B.
- Now we are left with the tasks of moving the disks from tower C to B.
- To do this, we have tower A and B available.
- The fact, that towers B has a disk on it can be ignored as the disks larger than the disks being moved from tower C and so any disk scan be placed on top of it.

Asymptotic Notations

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 such that $f(n)\leq c*g(n)$ for all n, $n \geq no$.

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

Theta: the function $f(n)=\theta(g(n))$ iff there exist positive constants c1,c2 and no such that $c1\ g(n) \leq f(n) \leq c2\ g(n)$ for all n, $n \geq no$.

Properties of big-Oh notation

1. Fastest growing function dominates a sum

- *O*(f(*n*)+g(*n*)) is *O*(max{f(*n*), g(*n*)})

2. Product of upper bounds is upper bound for the product

- If *f* is *O(g)* and *h* is *O(r)* then *fh* is *O(gr)*

***3. f* is *O(g)* is transitive**

- If *f* is *O(g)* and *g* is *O(h)* then *f* is *O(h)*

4. Hierarchy of functions

- *O*(1), *O*(log*n*), *O*(n1/2), *O*(*n*log*n*), *O*(n2), *O*(2n), *O*(*n*!)

Some Big-Oh's are not reasonable

Polynomial Time algorithms

- **An algorithm is said to be polynomial if it is *O(nc), c* >1**
- **Polynomial algorithms are said to be <u>reasonable</u>**
  - **They solve problems in reasonable times!**
  - **Coefficients, constants or low-order terms are ignored**

  e.g. if f(n) = 2n2 then f(n) = O(n2)

Exponential Time algorithms

- **An algorithm is said to be exponential if it is *O(rn), r* > 1**
- **Exponential algorithms are said to be <u>unreasonable</u>**

Classifying Algorithms based on Big-Oh

- **A function f(n) is said to be of at most logarithmic growth if f(n) = O(log n)**
- **A function f(n) is said to be of at most quadratic growth if f(n) = O(n2)**
- **A function f(n) is said to be of at most polynomial growth if f(n) = O(nk), for some natural number k > 1**
- **A function f(n) is said to be of at most exponential growth if there is a constant c, such that f(n) = O(cn), and c > 1**
- **A function f(n) is said to be of at most factorial growth if f(n) = O(n!).**
- **A function f(n) is said to have constant running time if the size of the input n has no effect on the running time of the algorithm (e.g., assignment of a value to a variable). The equation for this algorithm is f(n) = c**
- **Other logarithmic classifications: f(n) = O(n log n)**

  f(n) = O(log log n)

Conditional asymptotic notation

When we start the analysis of algorithms it becomes easy for us if we restrict our initial attentions to instances whose size is satisfied in certain conditions (like being a power of 2). Consider n as the size of integers to be multiplied. The algorithm moves forward if n=1, that needs microseconds for a suitable constant 'a'. If n>1, the algorithm proceeds by multiplying four pairs of integers of size n/2 (or three if we use the better algorithm). To accomplish additional tasks it takes some linear amount of time.

This algorithm takes a worst case time which is given by the function
$T: N \rightarrow R^{\geq 0}$ recursively defined where R is a set of non negative integers.
$T(1) = a$

$T(n) = 4T([n/2]) + bn$ for $n > 1$

Conditional asymptotic notation is a simple notational convenience. Its main interest is that we can eliminate it after using it for analyzing the algorithm. A function $F: N \rightarrow R^{\geq 0}$ is non decreasing function if there is an integer threshold n_0 such that $F(n) \leq F(n+1)$ for all $n \geq n_0$. This implies that mathematical induction is $F(n) \leq F(m)$ whenever $m \geq n \geq n_0$.

Consider $b \geq 2$ as an integer. Function F is b-smooth if it is non-decreasing and satisfies the condition $F(bn) \in O(F(n))$. Other wise, there should be a constant C (depending on b) such that $F(bn) \leq C\,F(n)$ for all $n \geq n_0$. A function is said to be smooth if it is b-smooth for every integer $b \geq 2$.

Most expected functions in the analysis of algorithms will be smooth, such as log n, nlogn, n^2, or any polynomial whose leading coefficient will be positive. However, functions those grow too fast, such as nlogn, 2n or n! will not be smooth because the ratio F(2n)/F(n) is unbounded.
Which shows that $(2n)^{\log(2n)}$ is not approximate of $O(n^{\log n})$ because a constant never bounds $2n^2$. Functions that are bounded above by a polynomial are usually smooth only if they are eventually non-decreasing. If they are not eventually non-decreasing then there is a probability for the function to be in the exact order of some other function which is smooth. For instance, let b(n) represent the number of bits equal to 1 in the binary expansion of n, for instance b(13) = 3 because 13 is written as 1101 in binary. Consider F(n)=b(n)+log n. It is easy to see that F(n) is not eventually non-decreasing and therefore it is not smooth because $b(2^k-1)=k$ whereas $b(2^k)=1$ for all k. However $F(n) \in \Theta(\log n)$ is a smooth function.

Removing condition from the conditional asymptotic notation

A constructive property of smoothness is that if we assume f is b-smooth for any specific integer $b \geq 2$, then it is actually smooth. To prove this, consider any two integers a and b (not smaller than 2). Assume that f is b-smooth. It is important to show that f is a-smooth as well.

Consider C and n_0 as constants such that F(bn)≤C F(n) and F(n)≤F(n+1) for all $n \geq n_0$. Let i=[$\log_b a$]. By definition of the logarithm $a = b^{\log_b a} \leq b^{\lceil \log_b a \rceil} = b^{i}$.

Consider $n \geq n_0$. It is obvious to show by mathematical induction from b-smoothness of F that $F(b^i n) \leq C^i F(n)$. But $F(an) \leq F(b^i n)$ because F is eventually non-decreasing and approximate to $b^i n \geq an \geq n_0$. It implies that $F(an) \leq \hat{C} F(n)$ for $\hat{C} = C^i$, and therefore F is a-smooth.

Smoothness rule

Smooth functions seem to be interesting because of the smoothness rule. Consider F: $N \rightarrow R^{\geq 0}$ (where $R^{\geq 0}$ is non negative integers) as a smooth function and T: $N \rightarrow R^{\geq 0}$ as an eventually non-decreasing function. Consider an integer where b≥2. The smoothness rule states that T(n)∈Θ(F(n)) whenever T(n) ∈Θ(F(n) | n is a power of b). We apply this rule equally to O and Ω notation. The smoothness rule assumes directly that T(n) ∈Θ(n^2) if n^2 is a smooth function and T(n) is eventually non-decreasing. The first condition is immediate since the function is approximate to n^2 (which is obviously nondecreasing) and $(2n)^2=4n^2$. We can demonstrate the second function from the recurrence relation using mathematical induction. Therefore conditional asymptotic notation is a stepping stone that generates the final result unconditionally. i.e. T(n)=Θ(n^2).

Some properties of asymptotic order of growth

- If $F1(n) \in O(h1(n))$ and $F2(n) \in O(h2(n))$, then $F1(n)+F2(n) \in O(\max\{h1(n),h2(n)\})$
- Implication: We determine the overall efficiency of the algorithm with a larger order of growth.
  - For example, $-6n^2+2n\log n \in O(n^2)$

Asymptotic growth rate

· O(h(n)): Class of function F(n) that grows no faster than h(n)

· Ω(h(n)): Class of function F(n) that grows at least as fast as h(n)

· Θ(h(n)): Class of function F(n) that grows at same rate as h(n)

Recurrence equations

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) * n
if end
algorithm ends.
```

Solving recurrence equations

A Recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs
Special techniques are required to analyze the space and time required

RECURRENCE RELATIONS EXAMPLE
EXAMPLE 1: QUICK SORT

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

In the above case the presence of function of T on both sides of the equation signifies the presence of recurrence relation

(*SUBSTITUTION MEATHOD used*) The equations are simplified to produce the final result:

$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \\ &= 2(2(n/22) + (n/2)) + n \\ &= 22\ T(n/22) + n + n \\ &= 22\ (T(n/23)+ (n/22)) + n + n \\ &= 23\ T(n/23) + \underline{n + n + n} \\ &= \textit{n log n} \end{aligned}$$

EXAMPLE 2: BINARY SEARCH

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

Above is another example of recurrence relation and the way to solve it is by Substitution.

$$\begin{aligned} T(n)&=T(n/2) +1 \\ &= T(n/22)+1+1 \\ &= T(n/23)+1+1+1 \\ &= \text{logn} \end{aligned}$$
$$T(n)= O(logn)$$

Analysis of linear search

Let us assume that we have a sequential file and we wish to retrieve an element matching with key ‘k’, then, we have to search the entire file from the beginning till the end to check whether the element matching k is present in the file or not.
There are a number of complex searching algorithms to serve the purpose of searching. The linear search and binary search methods are relatively straight forward methods of searching.

Sequential search:

In this method, we start to search from the beginning of the list and examine each element till the end of the list. If the desired element is found we stop the search and return the index of that element. If the item is not found and the list is exhausted the search returns a zero value.

In the worst case the item is not found or the search item is the last (n^{th}) element. For both situations we must examine all n elements of the array so the order of magnitude or complexity of the sequential search is n. i.e., O(n). The execution time for this algorithm is proportional to n that is the algorithm executes in linear time.

The algorithm for sequential search is as follows,

```
Algorithm : sequential search
Input : A, vector of n elements
K, search element
Output : j –index of k
Method : i=1
While(i<=n)
{
if(A[i]=k)
{
write("search successful")
write(k is at location i)
exit();
}
else
i++
if end
while end
write (search unsuccessful);
algorithm ends.
```

TUTORIAL 1

Algorithm Specification:

Algorithm can be described in three ways.

1. Natural language like English:
   When this way is choused 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:

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 items of 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:

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

In 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.

TUTORIAL 2

Selection Sort:

- Suppose we Must devise an algorithm that sorts a collection of n>=1 elements of arbitrary type.
- A Simple solution given by the following.
- (From those elements that are currently unsorted ,find the smallest & place it next in the sorted list.)

Algorithm:

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

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

- We can solve the latter problem using the code,

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

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

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

- Putting all these observations together, we get the algorithm Selection sort.

Theorem:

Algorithm selection sort(a,n) correctly sorts a set of n>=1 elements .The result 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 to 9,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 4 can 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.      {
        j:=I;
        for k:=i+1 to n do
            if (a[k]<a[j])
            t:=a[I];
            a[I]:=a[j];
            a[j]:=t;
    }
}
```

TUTORIAL 3

Performance Analysis:

1. **Space Complexity:**

The space complexity of an algorithm is the amount of money 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 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.

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.

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

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

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 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.

1. 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. So each invocation of sum execute a total of 2n+3 steps.

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 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)	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 or less 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 be sorted 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 divide by 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 sub array 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 element in
- Sub array T[1…I-1].

LECTURE PLAN

Subject code & Subject Name: 141401Design and Analysis of Algorithms
Unit Number: II

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 of the 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 size is 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 solutions to 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.

LECTURE PLAN

Subject code & Subject Name: 141401Design and Analysis of Algorithms
Unit Number: II **Period:2**

BINARY SEARCH

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

- 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 to check.
- 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.

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 is much higher than the cost of the other operations.

- Hence, the time is determined mainly by the total cost of the element comparison.

```
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];
         }
 }
```

MERGE SORT

- As another example divide-and-conquer, we investigate a sorting algorithm that 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 imagine then 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:

```
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 subproblems
        //find where to split the set
        mid = [(low+high)/2];
        //solve the subproblems.
        mergesort (low,mid);
        mergesort(mid+1,high);
        //combine the solutions .
        merge(low,mid,high);
    }
}
```

QUICK SORT

- The divide-and-conquer approach can be used to arrive at an efficient sorting method different from merge sort.
- In merge sort, the file a[1:n] was divided at its midpoint into sub arrays which were independently sorted & later merged.
- In Quick sort, the division into 2 sub arrays is made so that the sorted sub arrays 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 all elements 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 the set are better in practice.

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
```

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’ inputs
3. {
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 is assigned 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.

KNAPSACK PROBLEM

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

- The problem is we have to fill the bag with the help of N objects and the resulting profit 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 it according 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 weight of 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.Algorityhm Greedy knapsack (m,n)
2//P[1:n] and the w[1:n]contain the profit
3.// & 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***
½	1/3	¼	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 see this solution is optimal for the given problem instance.

TUTORIAL 1

BINARY SEARCH

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

- 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 to check.
- 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:

1) 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 Binsearch goes through as it searches for different values of 'x'.
→ Only the variables, low, high & mid need to be traced as we simulate the algorithm.
→ 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 elements to 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 the search.
- If low becomes > than high, then 'x' is not present & hence the loop is exited.

TUTORIAL 2

Algorithm: Merging 2 sorted subarrays using auxiliary storage.

```
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];
}
```

- 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)

→ 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 produces the 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)

TUTORIAL 3

MINIMUM 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 the nodes 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 that are 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.

Minimum 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 increasing order 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;
}
```

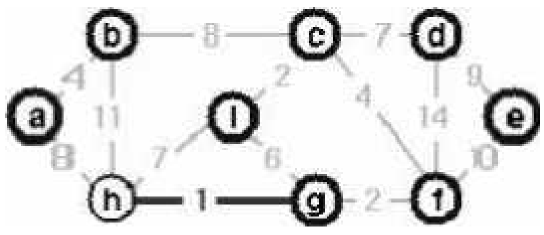
Analysis

- The time complexity of minimum cost spanning tree algorithm in worst case is O(|E|log|E|),

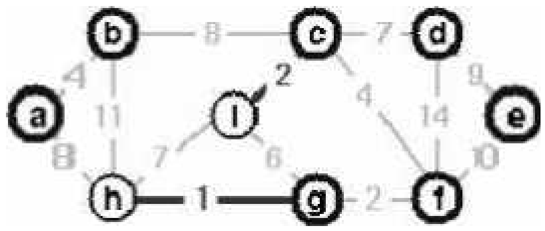
→where E is the edge set of G.

Example: Step by Step operation of Kurskal algorithm.

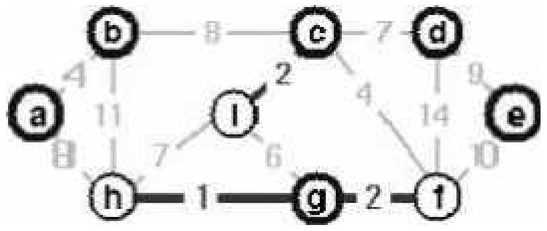
Step 1. In the graph, the Edge(g, h) is shortest. Either vertex g or vertex h could be representative. Lets choose vertex g arbitrarily.



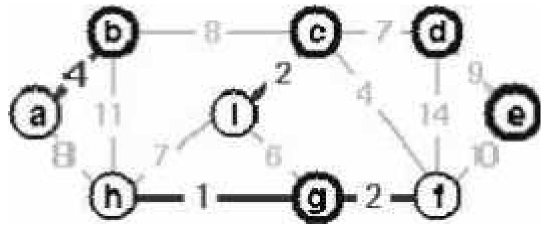
Step 2. The edge (c, i) creates the second tree. Choose vertex c as representative for second tree.



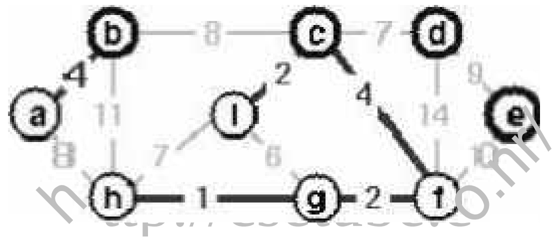
Step 3. Edge (g, g) is the next shortest edge. Add this edge and choose vertex g as representative.



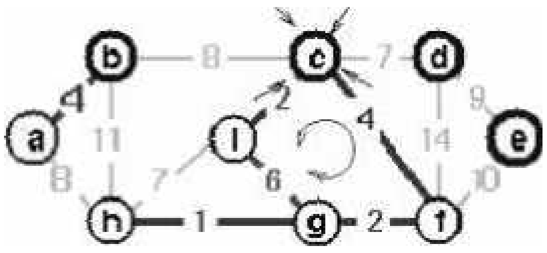
Step 4. Edge (a, b) creates a third tree.



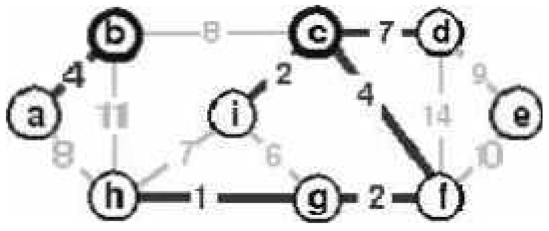
Step 5. Add edge (c, f) and merge two trees. Vertex c is chosen as the representative.



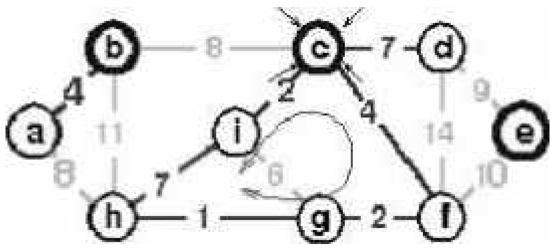
Step 6. Edge (g, i) is the next next cheapest, but if we add this edge a cycle would be created. Vertex c is the representative of both.



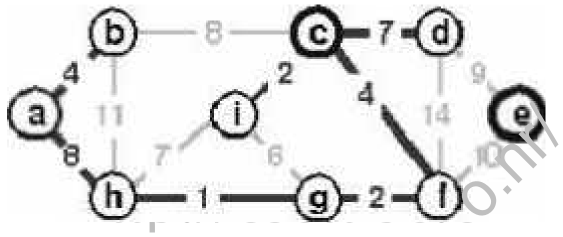
Step 7. Instead, add edge (c, d).



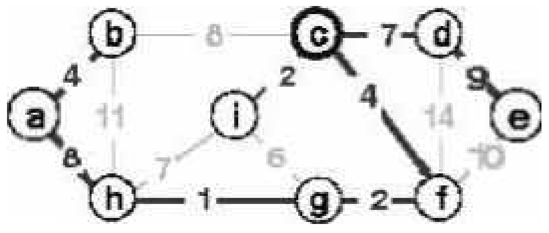
Step 8. If we add edge (h, i), edge(h, i) would make a cycle.



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.



DYNAMIC PROGRAMMING

General method-multistage graphs-all pair shortest path algorithm-0/1 knapsack and traveling salesman problem-chained matrix multiplication-approaches using recursion-memory functions

- The idea of dynamic programming is thus quit simple: avoid calculating the same thing twice, usually by keeping a table of known result that fills up a sub instances are solved.
- Divide and conquer is a top-down method.
- 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.
- Dynamic programming on the other hand is a bottom-up technique.
- We usually start with the smallest and hence the simplest sub- instances.
- 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.
- The essential difference between the greedy method and dynamic programming is that the greedy method only one decision sequence is ever generated.
- 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.

MULTISTAGE GRAPH

1. 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.
2. In addition, if < u,v > is an edge in E, then u < = Vi and V ∑ Vi+1 for some i, 1<= i < k.
3. If there will be only one vertex, then the sets Vi and Vk are such that [Vi]=[Vk] = 1.
4. Let ‘s’ and ‘t’ be the source and destination respectively.
5. The cost of a path from source (s) to destination (t) is the sum of the costs of the edger on the path.
6. The *MULTISTAGE GRAPH* problem is to find a minimum cost path from ‘s’ to ‘t’.
7. 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.
8. This *MULISTAGE GRAPH* problem can be solved in 2 ways.

   a) Forward Method.
   b) Backward Method.

FORWARD METHOD

1. Assume that there are ‘k’ stages in a graph.
2. 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.
3. We will find out the path (i.e.) minimum cost path from source to the destination (ie) [Stage-1 to Stage-k].

PROCEDURE:

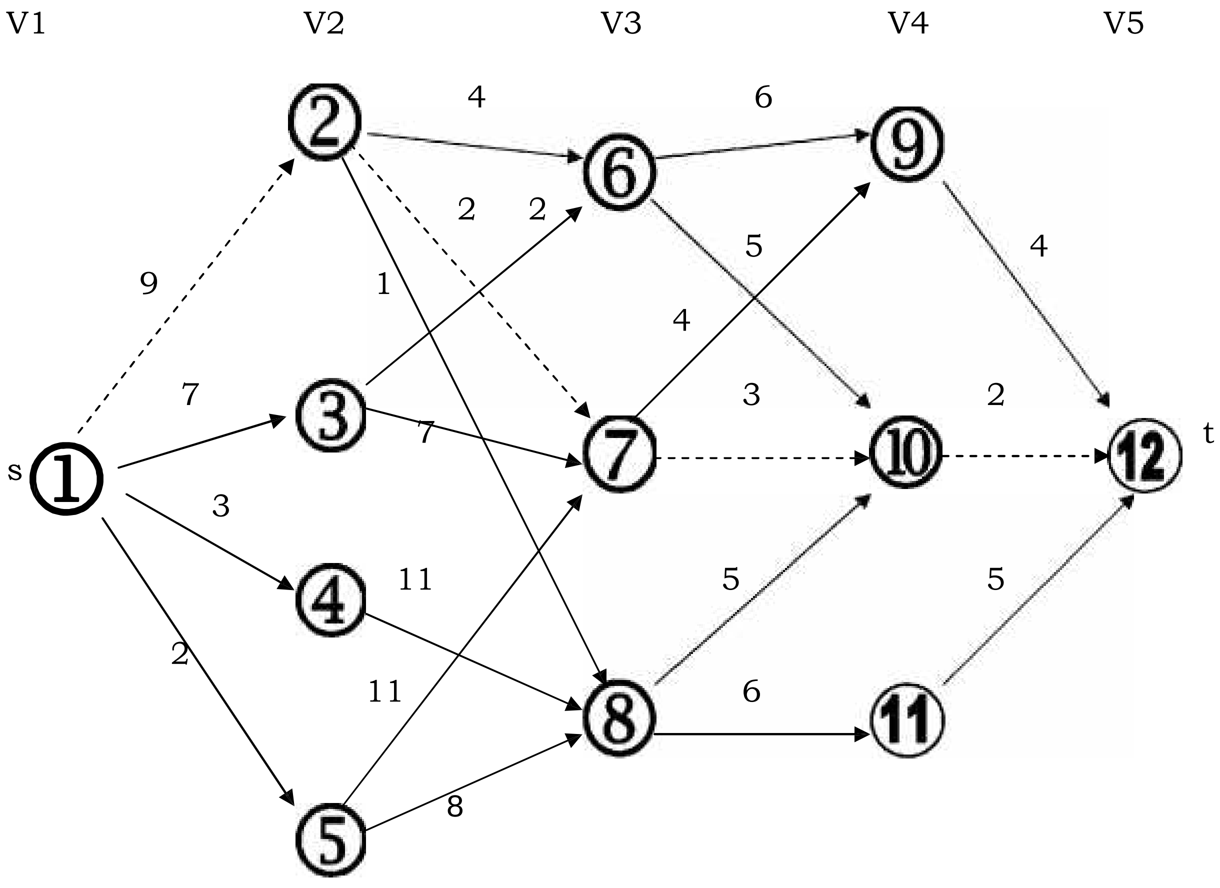
- ❖ Maintain a cost matrix cost (n) which stores the distance from any vertex to the destination.
- ❖ If a vertex is having more than one path, then we have to choose the minimum distance path and the intermediate vertex, which gives the minimum distance path, will be stored in the distance array ‘D’.
- ❖ In this way we will find out the minimum cost path from each and every vertex.
- ❖ Finally cost(1) will give the shortest distance from source to destination.
- ❖ 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.
- ❖ For a ‘k’ stage graph, there will be ‘k’ vertex in the path.
- ❖ In the above graph V1…V5 represent the stages. This 5 stage graph can be solved by using forward approach as follows,

BACKWARD METHOD

- if there one 'K' stages in a graph using back ward approach. we will find out 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)[from stage k to stage 1]

PROCEDURE:

1. It is similar to forward approach, but differs only in two or three ways.
2. Maintain a cost matrix to store the cost of every vertices and a distance matrix to store the minimum distance vertex.
3. Find out the cost of each and every vertex starting from vertex 1 up to vertex k.
4. 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.



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.
- Each edge has an associated non-negative length.
- We want to calculate the length of the shortest path between each pair of nodes.
- 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.
- The principle of optimality applies: if k is the node on the shortest path from i to j then the part of the path from i to k and the part from k to j must also be optimal, that is shorter.
- First, create a cost adjacency matrix for the given graph.
- Copy the above matrix-to-matrix D, which will give the direct distance between nodes.
- 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.
- At the iteration k, we have to check for each pair of nodes (i,j) whether or not there exists a path from i to j passing through node k.

0/1 KNAPSACK PROBLEM:

- This problem is similar to ordinary knapsack problem but we may not take a fraction of an object.
- We are given ' N ' object with weight W_i and profits P_i where I varies from l to N and also a knapsack with capacity ' M '.
- The problem is, we have to fill the bag with the help of ' N ' objects and the resulting profit has to be maximum.
- Formally, the problem can be started as, maximize $\sum_{i=l}^{n} X_i P_i$

  subject to $\sum_{i=l}^{n} X_i W_i$ L M

- Where X_i are constraints on the solution $X_i \in \{0,1\}$. (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 the unit is 0. That is why it is called as 0/1, knapsack problem.
- 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'.
- In the table T[i,j] will be the maximum valve of the objects i varies from 1 to n and j varies from O to M.

RULES TO FILL THE TABLE:-

- If i=l and j < w(i) then T(i,j) =o, (ic) o pre is filled in the table.
- If i=l and j ≥ w (i) then T (i,j) = p(i), the cell is filled with the profit p[i], since only one object can be selected to the maximum.
- If i>l and j < w(i) then T(i,l) = T (i-l,j) the cell is filled the profit of previous object since it is not possible with the current object.
- If i>l and j ≥ w(i) then T (i,j) = {f(i) +T(i-l,j-w(i)),. since only 'l' 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.
- After the table is generated, it will give details the profit.

TRAVELLING SALESMAN PROBLEM

- Let G(V,E) be a directed graph with edge cost c_{ij} is defined such that $c_{ij} > 0$ for all i and j and $c_{ij} = \infty$,if $<i,j> \notin E$.
  Let $|V| = n$ and assume $n>1$.
- The traveling salesman problem is to find a tour of minimum cost.
- A tour 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 :

1. Suppose we have to route a postal van to pick up mail from the mail boxes located at ‘n’ different sites.
2. An n+1 vertex graph can be used to represent the situation.
3. One vertex represent the post office from which the postal van starts and return.
4. Edge <i,j> is assigned a cost equal to the distance from site ‘i’ to site ‘j’.
5. the route taken by the postal van is a tour and we are finding a tour of minimum length.
6. every tour consists of an edge <1,k> for some $k \in V-\{\}$ and a path from vertex k to vertex 1.
7. the path from vertex k to vertex 1 goes through each vertex in V-{1,k} exactly once.
8. the function which is used to find the path is

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

9. 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.
10. the function g(1,v-{1}) is the length of an optimal tour.

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,

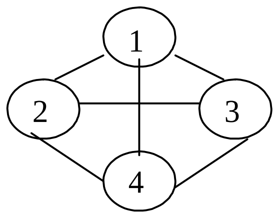


FIG: Graph G

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:

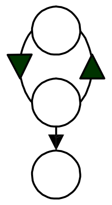
UNDIRECTED GRAPH:

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 said to 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 at vertex ‘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.
  1. Breadth first Search and Traversal.
  2. Depth first Search and Traversal.

BREADTH FIRST SEARCH AND TRAVERSAL:

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 algorithm has 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 continues until no unexplored vertex is left.
- The list of unexplored vertices operates as a queue and can be represented using 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

BACKTRACKING

- 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 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 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 ≥ 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 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.

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 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|.

STEPS TO GENERATE THE SOLUTION:

- Initialize x array to zero and start by placing the first queen in k=1 in the first 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. Here we have to check whether there is any queen in the same column or diagonal.
- For this considers the previous position, which had already, been found out. 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 th queen can’t be placed in position X (k).
- For not possible condition increment X (k) value by one and precede d until the 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 the matrix 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 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)

```
{
    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)    ;
          }
      }
}
```

- 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:

- 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.
- 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.

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.

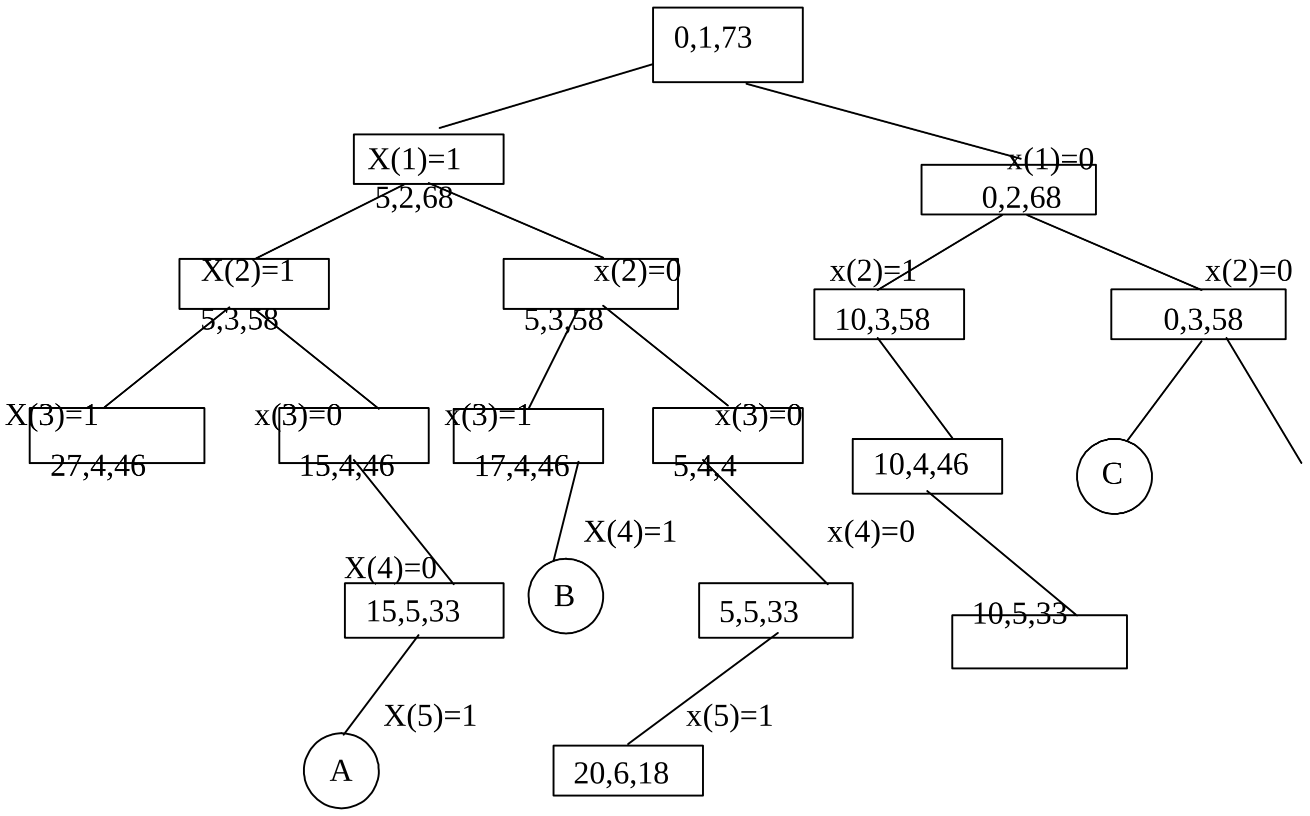
- If the above condition is not satisfied then we have to check S+X(k)+W(k+1)<=M. If so, we have to generate the left sub tree. It means W(t) can be included so the sum will be incremented and we have to check for the next k.

- 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.

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,



Ist

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.

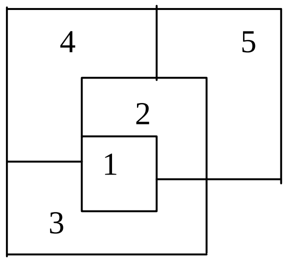
- The left sub tree of the root defines all subsets containing Wi.

- The right subtree of the root defines all subsets, which does not include Wi.

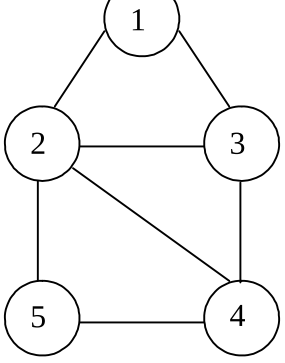
GRAPH COLORING:

- 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 same color. Yet only 'M' colors are used. So it's called M-color ability decision problem.
- 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.



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

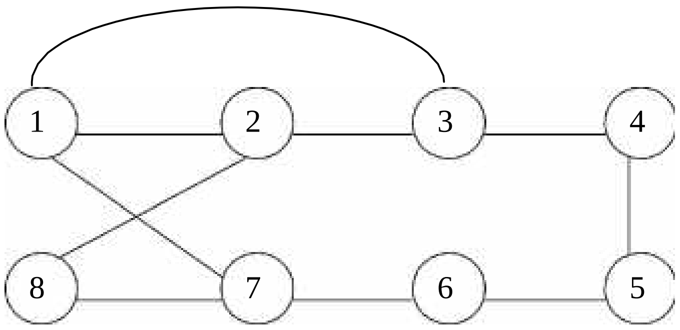


HAMILTONIAN CYCLES:

- ❖ 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 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.

- ❖ The backtracking algorithm helps to find Hamiltonian cycle for any type of graph.

Procedure:

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 these two 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:(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 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);
}
```

Knapsack Problem using Backtracking:

- The problem is similar to the zero-one (0/1) knapsack optimization problem is 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 for choosing a subset of the weights such that,

$$\sum_{1 \le i \le n} WiXi \le \text{m} \quad \text{and} \quad \sum_{1 \le i \le n} PiXi \text{ is Maximized.}$$

Xi $\rightarrow$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 by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node.

- The profits and weights are assigned in descending order depend upon the ratio.

(i.e.) Pi/Wi $\ge$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, if so then we include that object (i.e.) the unit is 1.(i.e.) K$\rightarrow$ 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 an upper bound on the best solution obtainable at level K+1.

- Repeat the process until we reach the optimal solution.

Graph Traversals

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,

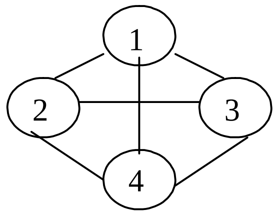


FIG: Graph G

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:

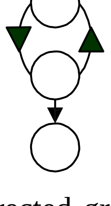
UNDIRECTED GRAPH:

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 said to 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 at vertex '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.

i. Breadth first Search and Traversal.
ii. Depth first Search and Traversal.