DESIGN AND ANALYSIS OF ALGORITHM

UNIT I

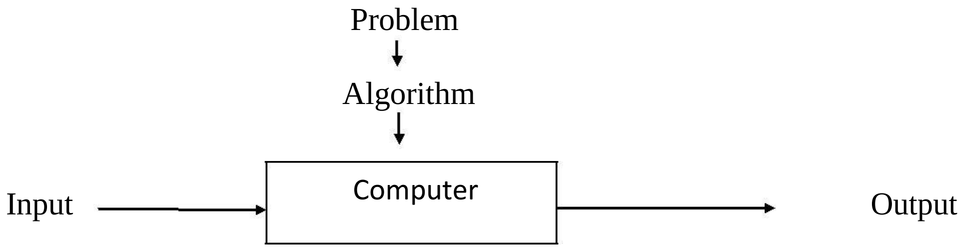
Algorithms: Definitions and notations: standard notations - asymptotic notations – worst case, best case and average case analysis; big oh, small oh, omega and theta notations; Analysis of Sorting and Searching: Heap, shell, radix, insertion, selection and bubble sort; sequential, binary and Fibonacci search. Recursive algorithms, analysis of non-recursive and recursive algorithms, solving recurrence equations, analyzing control structures.

2 Marks

1. What is an algorithm? (UQ April 2012 & APRIL 2013)

An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in finite amount of time.

2. Give the diagram representation of Notion of algorithm.



3. Why is the need of studying algorithms?

From a practical standpoint,

- ✓ A standard set of algorithms from different areas of computing must be known
- ✓ In addition to be able to design them
- ✓ Analyze the algorithm efficiencies.

From a theoretical standpoint, the study of algorithms is the basis of computer science.

4. What is algorithmic?

The study of algorithms is called algorithmic. It is more than a branch of computer science. It is the core of computer science and is said to be relevant to most of science, business and technology.

5. What is the formula used in Euclid's algorithm for finding the greatest common divisor of two numbers?

Euclid‟s algorithm is based on repeatedly applying the equality

Gcd(m,n)=gcd(n,m mod n) until m mod n is equal to 0, since gcd(m,0)=m.

6. What are the three diffe rent algorithms used to find the gcd of two numbe rs?

The three algorithms used to find the gcd of two numbers are .

- Euclid‟s algorithm
- Consecutive integer checking algorithm
- Middle school procedure

7. What are the fundamental steps involved in algorithmic proble m solving?

The fundamental steps are

1. Understanding the problem.
2. Ascertain the capabilities of computational device
3. Choose between exact and approximate problem solving.
4. Decide on appropriate data structures.
5. Algorithm design techniques
6. Methods for specifying the algorithm
7. Proving an algorithms correctness
8. Analyzing an algorithm.
9. Coding an algorithm

8. What is an algorithm design technique?

An algorithm design technique is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

9. What is pseudo code?

A pseudo code is a mixture of a natural language and programming language constructs to specify an algorithm. A pseudo code is more precise than a natural language and its usage often yields more concise algorithm descriptions.

10. What are the types of algorithm efficiencies?

The two types of algorithm efficiencies are

- Time efficiency: indicates how fast the algorithm runs.
- Space efficiency: indicates how much extra memory the algorithm needs

11. Mention some of the important problem types?

Some of the important problem types are as follows.

1. Sorting.
2. Searching
3. String processing.
4. Graph problems
5. Combinatorial problems
6. Geometric problems.
7. Numerical problems

12. What are the classical geometric problems?

The two classic geometric problems are ,

1. The closest pair problem: given n points in a plane find the closest pair among them
2. The convex hull problem: find the smallest convex polygon that would include all the points of a given set.

13. What are the steps involved in the analysis frame work?

The various steps are as follows

- Measuring the input‟s size.
- Units for measuring running time.
- Orders of growth.
- Worst case, best case and average case efficiencies

14. What is the basic operation of an algorithm and how is it identified?

The most important operation of the algorithm is called the basic operation of the algorithm, the operation that contributes the most to the total running time. It can be identified easily because it is usually the most time consuming operation in the algorithms innermost loop.

15. What is the running time of a program implementing the algorithm?

The running time T(n) is given by the following formula

$$T(n) \approx c_{op}C(n)$$

Cop is the time of execution of an algorithm‟s basic operation on a particular computer and C(n) is the number of times this operation needs to be executed for the particular algorithm.

16. What are exponential growth functions?

The functions 2n and n! are exponential growth functions, because these two functions grow so fast that their values become astronomically large even for rather smaller values of n.

17. What is worst-case efficiency?

The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input or inputs of size n for which the algorithm runs the longest among all possible inputs of that size.

18. What is best-case efficiency? (UQ APRIL 2013)

The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input or inputs for which the algorithm runs the fastest among all possible inputs of that size.

19. What is average case efficiency?

The average case efficiency of an algorithm is its efficiency for an average case input of size n. I t provides information about an algorithm behavior on a “typical” or “random” input.

20. What is amortized efficiency?

In some situations a single operation can be expensive, but the total time for the entire sequence of n such operations is always significantly better that the worst case efficiency of that single operation multiplied by n. this is called amortized efficiency.

21. Define O-notation?

A function t(n) is said to be in O(g(n)), denoted by t(n) O(g(n)), if t(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exists some positive constant c and some nonnegative integer n0 such that

T (n) <= cg (n) for all n >=. n0

22. Define Ω-notation?

A function t(n) is said to be in (g(n)), denoted by t(n) (g(n)), if t(n) is bounded below by some constant multiple of g(n) for all large n, i.e., if there exists some positive constant c and some nonnegative integer n0 such that

T(n) >= cg(n) for all n >=. n0

23. Define Θnotation?

A function t(n) is said to be in (g(n)), denoted by t(n) (g(n)), if t(n) is bounded both above & below by some constant multiple of g(n) for all large n, i.e., if there exists some positive constants c1 & c2 and some nonnegative integer n0 such that

c_2g (n) <= t(n) <= c_1g(n) for all n >= n0

24. What are the basic asymptotic efficiency classes?

The various basic efficiency classes are.

- Constant : 1
- Logarithmic: log n.
- Linear : n
- N-log- n : nlog n
- Quadratic : n2
- Cubic: n3.
- Exponential : 2n
- Factorial: n!

25. Give an non-recursive algorithm to find out the largest element in a list of n numbers.

```
ALGORITHM MaxElement(A[0..n-1])
//Determines the value of the largest element in a given
array //Input:An array A[0..n-1] of real numbers
//Output: The value of the largest element in A
          maxval ←   a[0]
          for I ← 1 to n-1 do
  if A[I] > maxval
          maxval ←   A[I]
  return maxval
```

26. Write the general plan for analyzing the efficiency for non-recursive algorithms.

The various steps include.

- Decide on a parameter indicating input‟s size.
- Identify the algorithms basic operation. .
- Check whether the number of times the basic operation is executed depends on size of input. If it depends on some additional property the worst, average and best-case efficiencies have to be investigated separately.
- Set up a sum expressing the number of times the algorithm‟s basic operation is executed.

Using standard formulas and rules of manipulation find a closed- form formula for the count or at least establish its order of growth.

27. Give a non-recursive algorithm for element uniqueness problem.

```
ALGORITHM UniqueElements(A[0..n-1])
//Checks whether all the elements in a given array are
distinct //Input: An array A[0..n-1]
//Output Returns „true‟ if all elements in A are distinct and „false‟
//otherwise
          for I ← to n-2 do
          for j ← I+1 to n-1 do
          if A[I] = A[j] return
          false return true
```

28. Mention the non-recursive algorithm for matrix multiplication?

```
ALGORITHM MatrixMultiplication(A[0..n-1,0..n-1], B[0..n-1,0..n-1])
//Multiplies two square matrices of order n by the definition based
//algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix C = AB
          for I ← 0 to n-1 do
          for j ← 0 to n-1 do
          C[I,j] ← 0.0
          for k ← 0 to n-1 do
       C[I,j]    ← C[I,j] + A[I,k]*B[k,j]
          return C
```

29. Write a non-recursive algorithm for finding the numbe r of binary digits for a positive decimal integer.

```
     ALGORITHM Binary(n)
     // Input A positive decimal integer n
     // Output The number of binary digits in n‟s binary representation
  // count ← 1 while n>1 do
  //  count ←count + 1 n ← n/2
  //  return count
```

30. Write a recursive algorithm to find the n-th factorial number.

```
     ALGORITHM F(n)
     // Computes n! Recursively
     // Input A non- negative integer n
     // Output The value of n!
               if n=0 return 1
               else return F(n-1) * n
```

31. What is the recurrence relation to find out the numbe r of multiplications and the initial condition for finding the n-th factorial number?

The recurrence relation and initial condition for the number of multiplications

is M(n)=M(n-1)+1 for n>0
M(0)=0

32. Write the general plan for analyzing the efficiency for recursive algorithms.

The various steps include

- Decide on a parameter indicating input‟s size.
- Identify the algorithms basic operation. .
- Check whether the number of times the basic operation is executed
- Depends on size of input. If it depends on some additional property the worst, average and best-case efficiencies have to be investigated separately.
- Set up a recurrence relation with the appropriate initial condition, for the number of times the basic operation is executed.
- Solve the recurrence or at least ascertain the orders of growth of its solution.

33. Write a recursive algorithm for solving Towe r of Hanoi proble m.

ALGORITHM

- To move n>1 disks from peg1 to peg3, with peg2 as auxiliary, first move recursively n-1 disks from peg1 to peg2 with peg3 as auxiliary.
- .Then move the largest disk directly from peg1 to peg3.
- Finally move recursively n-1 disks from peg2 to peg3 with peg1 as auxiliary.
- If n=1 simply move the single disk from source peg to destination peg.

34. Write a recursive algorithm to find the number of binary digits in the binary representation of an integer.

```
ALGORITHM BinRec(n)
// Input A positive decimal integer n
// Output The number of binary digits in n‟s binary representation
if n=1 return 1
else return BinRec(n/2)+1
```

35. What is selection sort?

Selection sort is started by scanning the entire list to find the smallest element and exchange it with the first element, putting the first element in the final position in the sorted list. Then the scan starts from the second element to find the smallest among n-1 elements and exchange it with the second element.

36. Mention the pseudo code for selection sort.

```
ALGORITHM SelectionSort(A[0..n-1])
//The algorithm sorts a given array by selection sort
//Input: An array A[0..n-1] of orderable elements
//Output: Array A[0..n-1] sorted in ascending order
for I ← 0 to n-2 do
min ← I
for j ← I+1 to n-1 do
if A[j] < A[min] min ← j
swap A[I] and A[min]
```

37. What is bubble sort?

Another brute force approach to sort a problem is to compare adjacent elements of the list and exchange them if they are out of order, so we end up “bubbling up” the largest element to the last position in the list. The next pass bubbles up the second largest element, and so on until n-1 passes, the list is sorted. Pass I can be represented as follows

```
For i = 1 to n
    For j = 1 to n
        If (a[i]>a[j])
        {
            t= a[i];
            a[i]=a[j];
            a[j]=t;
        }
```

38. Give an algorithm for bubble sort?

```
ALGORITHM BubbleSort(A[0..n-1])
//The algorithm sorts array A[0..n-1] by bubble sort
//Input: An array A[0..n-1] of orderable elements
//Output: Arrar A[0..n-1] sorted in ascending order
    For i = 1 to n
        For j = 1 to n
            If (a[i]>a[j])
            {
            t= a[i];

            a[i]=a[j];

             a[j]=t;

            }
```

39. Explain about the enhanced version of sequential search.

Sequential search simply compares successive elements of a list with a given search key until either a match is encountered or the list is exhausted without finding a match. The enhancement in this version is to append the search key to the end of the list , then the search for the key will have to be successful & so we can eliminate a check for the list‟s end on each iteration.

40. What is binary search? (UQ April'12)

If „q‟ is always chosen such that „aq‟ is the middle element(that is, q=[(n+1)/2), then the resulting search algorithm is known as binary search.

41. What is insertion sort?

Insertion sort in an application of decrease-by-one technique to sort an array A[0..n-1]. We assume that the smaller problem of sorting an array A[0..n-2] has already been solved to give us a sorted array of size n-1. Then an appropriate position for A[n-1] is found among the sorted element and then the element is inserted.

42. Give the algorithm for insertion sort.

```
    //Sorts a given array by insertion sort
    //Input: An array A[0..n-1] of n orderable elements
    //Output: Array A[0..n-1] sorted in non-decreasing order
        for I ← 1 to n-1 do
        v ← A[I]
        j ← I-1
        while j >= 0 and A[j] > v do
        A[j+1] ← A[j]
        j ← j – 1
        A[j+1] ← v
```

43. What is time complexity?

The time complexity of an algorithm is the amount of computer time it needs to run to completion.

44. What is space complexity?

The space complexity of an algorithm is the amount of memory it needs to run to completion.

45. Discuss about various asymptotic notation (UQ Nov’12)

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 ³n0.

Omega

The function f(n) =W (g(n)) iff there exist positive constant C and no such that f(n) C * g(n) for all n, n ³ n0.

Theta

The function f(n) =q (g(n)) iff there exist positive constant C1, C2, and no such that C1 g(n)£ f(n) £ C2 g(n) for all n, n ³ n0.

Little 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 ³n0.

Little Omega.

The function f(n) =W (g(n)) iff there exist positive constant C and no such that f(n) C * g(n) for all n, n ³ n0.

46. What is recursive algorithm? (UQ Nov’12 , Apr/May'14)

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 indeed recursive if it calls another algorit hm, which in turn calls A.

47. Define sorting (UQ Nov’12 , Apr/May'14)

Arrange given number in an order may be ascending or descending order

48. Define search (UQ: NOV’14)

A search is an algorithm for finding an item with specified properties among a collection of items.
There are 3 types of search

1 Linear search
2 Binary search
3 Fibonacci search

49.what is pe rformance measure ment? (UQ: NOV’14)

There are two aspects of algorithmic performance:

- Time
  - Instructions execution time.
  - How fast does the algorithm perform?
  - What affects its runtime?
- Space
  - Data structures take space
  - What kind of data structures can be used?
  - How does choice of data structure affect the runtime?

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.

11 Marks

1. Describe briefly about Algorithm. (or) Describe the rules for writing algorithm. (or) Describe Analysis of Algorithm (or) Describe Analysis of Control Structures. Explain Standard Notation (UQ APRIL'13)

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.

Algorithm Specification: Algorithm can be described in three ways.

1. Natural language like English:

   When this way is chosed 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:

```
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.

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

2.Explain Average case Analysis.

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,$2 \le I \le n$, 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].

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 $f(n) \le c*g(n)$ for all n, $n \ge no$.

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

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

3.Describe Asymptotic notation and basic Efficiency classes.

Asymptotic notations are methods used to estimate and represent the efficiency of an algorithm using simple formula. Another names

1. Asymptotic growth rate
2. Asymptotic order
3. The order of function

a) Big oh notation (0)

b) Big omega notation (Ω)

c) Big Theta notation (θ)

d) Little-oh notation (0)

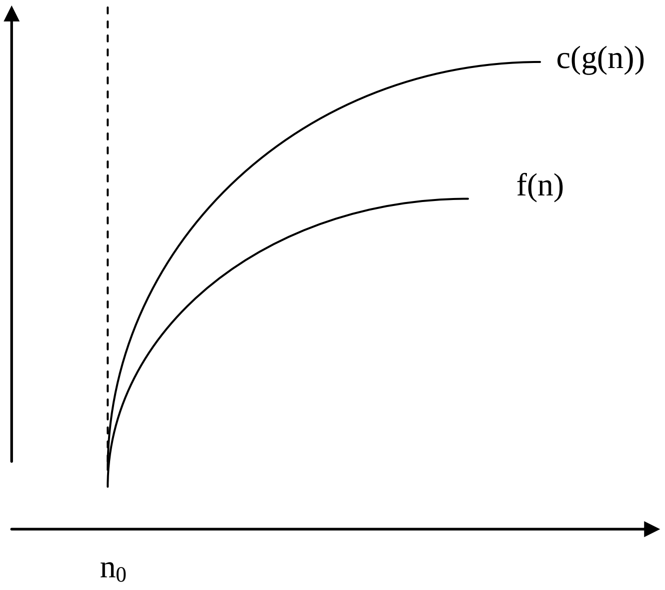
d) Little-omega notation (ω)

<u>Big oh notation (0)</u>

This is used to define the worst case running time of an algorithm and concerned with very large values of n. It defines the function.

$f(n)=0(g(n))$, if $f(n) \le (g(n)) \forall\ n \ge n_0$, and c is a positive constant and n_0 is break even point i.e f(n) grows no faster than g(n). The g(n) is the upper bound.

$0(g(n)) = \{ f(n)$: there exist +ve const c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$



Big – oh notation: f (n) ∈ O (g (n))

COMPUTING TIME	**NAME**
0(1)	Constant
0(log n)	Logarithmic function
0(n)	linear in n
$0(n^2)$	quadratic
$o(n^3)$	cubic
$0(2^n)$	exponential
0(n log n)	logarithmic

Example 4.1:

1. f(n)=3n+2
   $f(n) \leq cg(n)$

   - ⇨ $3n + 2 \leq c(n)$
   - ⇨ $3n + 2 \leq cn$ here c = 4
   - ⇨ if n=1 => $3 + 2 \leq 4$ (i.e) $5 \leq 4$
     if n=2 => $3 X 2 + 2 \leq 4 X 2$ (i.e) $8 \leq 8$

     if n=3 => $11 \leq 12$ Then f(n) = 3n + 2 = 0(n) as $3n + 2 \leq cn$;

   For all n >= 2 { (i.e) n>= n_0 }

2. $10n^2 + 4n + 2$
   $f(n) \leq cg(n)$

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

$= 11(n^2) \quad \{ c=11\}$

if n=1 then $16 \leq 11$;

if n=2 then $50 \leq 44$;

if n=3 then $104 \leq 99$;

if n=4 then $178 \leq 176$;

if n=5 then $272 \leq 275$;

if n=6 then $386 \leq 396$;

Then $f(n) = 10n^2 + 4n + 2= 0(n^2)$ as $10n^2 + 4n + 2 \leq 11n^2$ for all $n \geq 5$;

Big omega notation (Ω)

This is used to describe the best case running time of also and concerned with very large values of n. It define the fn $f(n)= \Omega\ g(n)$ iff $f(n) \geq cg(n)$ for all n, $(n \geq n_0)$

Where g(n) is only a lower bound of f(n) an c is a positive constant and n_0 is a break even point.

$\Omega(g(n)=\{f(n)$: There exist +ve constant c and n_0 such that $0\leq cg(n) \leq f(n)$ for all $n \geq n_0\}$

Example 4.2:

$f(n)=3n+2$

$f(n)\geq cg(n)$

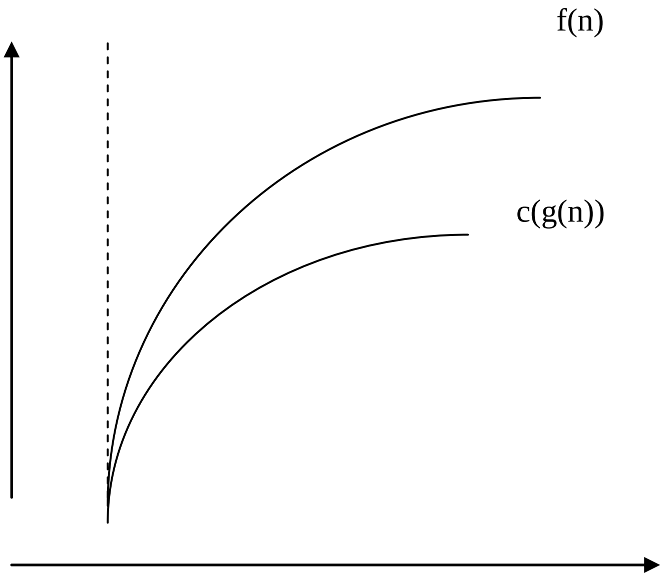
$3n+2\geq3(n)$;

if n=1 then $5\geq4$

then $f(n)=3n+2=\Omega(n)$ as $3n+2\geq3n$ for all $n\geq1$

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

$\Rightarrow \Omega(1)$



n_0 Big – omega notation: f (n) $\in \Omega$ (g (n))

<u>Big Theta notation (θ)</u>

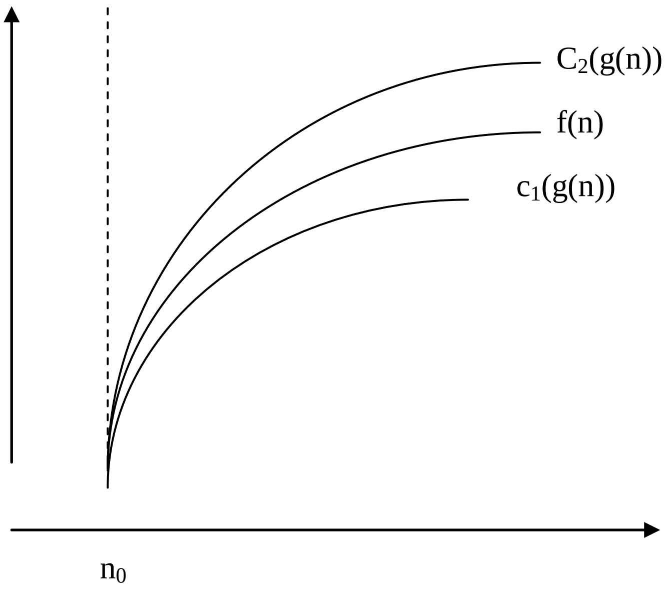
This is used to describe the average case running time of algorithm and concerned with very large values of n. It defines function $f(n)=\theta g(n)$

iff $c1g(n) \leq c2g(n)$ for all $n(n \geq n_0)$ C1,c2 positive constant and n_0 is break even point.

θ (g(n))={ f(n): there exist +ve constant c1,c2 and n0 such that $0 \leq c1g(n) \leq c2g(n)$ for all

i.e. $f(n)=\theta(g(n))$ if $f(n)=\theta(g(n))$ & $f(n)= \Omega g(n))$ $n \geq n_0$ }

Here g(n) is both upper and lower bound on f(n).



Big – theta notation: f (n) $\in \Theta$ (g (n))

<u>Example 4.3:</u>

$f(n)=3n+2$

$c1g(n) \leq f(n) \leq c2g(n)$

$3n \leq 3n+2 \leq 4n$ for all $n \geq 2$;

so $c1=3$, $c2=4$, $n_0 = 2$

then $3n+2=\theta(n)$ as $3n \leq 3n+2 \leq 4$

<u>Little-oh notation (0)</u>

This is used to determine the worst-case analysis of algorithms and concerned with small values of n.

0(g(n)={f(n) : for any +ve const c>0, there exist a const $n_0>0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$}

The function f(n)=0(g(n))

iff $$\lim_{n \to \alpha} \frac{f(n)}{g(n)} == 0.$$

Little-omega notation (ω)

This is used to describe the best-case analysis of algorithms and concerned with small values of n.

$f(n)=\omega(g(n))$

$$\text{iff} \quad \lim_{n\to\alpha} \frac{f(n)}{g(n)} \;==\; 0.$$

$\omega(g(n))=\{f(n)$: for any +ve const $c>0$ there exist a const $n_0 > 0$ such that $0\leq cg(n)<f(n)$ for all $n\geq n_0\}$

a. $100n + 6 \Rightarrow n$
b. $6 \text{ x } 2^n + n^2 \Rightarrow 2^n$
c. $3n^3 + 2n^2 + n + 1 \Rightarrow n^3$

Difference between big-oh to small-oh

1. In $f(n)=0(g(n))$, the bound $0\leq f(n)\leq cg(n)$ holds for some constant $c>0$.
2. In $f(n)=0(g(n))$, the bound $0\leq f(n)<cg(n)$ holds for some constant $c>0$.

Useful property involving the asymptotic notations

Theorem:

If $t_1(n) \in 0(g(n))$ and $t_2(n)\in 0(g_2(n))$, then

$t_1(n) +t_2(n) \in(\max\{g,(n),g_2(n)\})$

Proof:

Real numbers a_1,b_1,a_2,and b_2

If $a_1\leq b_1$ and $a_2\leq b_2$ the $a_1+a_2\leq 2 \max\{b_1,b_2\}$

Since $t_1(n)\in 0(g(n))$

$t_1(n)\leq c_1$g1(n) for all n≥n1

Since $t_2(n) \in 0(g(n))$

$t_2(n)\leq c_2g_2(n)$ for all $n\geq n_2$

$C_3=\max\{c_1,c_2\}$

Consider $n\geq \max\{n_1,n_2\}$

$t_1(n)+t_2(n)\leq C_1g_1(n)+c_2g_2(n)$

$\leq \quad C_3g_1(n)+c_3g_2(n)=c_3[g_1(n)+g_2(n)]$

$\leq \quad C_3\ 2\max\{g_1(n),g_2(n)\}$

hence, $t_1(n)+t_2(n)\in 0(\max\{g_1(n),g_2(n)\})$

$$\left.\begin{array}{c} t_1(n)\in 0(g_1(n)) \\ \hline t_2(n)\in 0(g_2(n)) \end{array}\right\} \; t_1(n)+t_2(n)\in 0(\max\{g_1(n),g_2(n)\})$$

Basic efficiency classes

Basic asymptotic efficiency classes

Class	Name	Comments
1	Constant	Short of best case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
log n	Logarithmic	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm . Note that a logarithmic algorithm cannot take into account all its input, any algorithm that does so will have at least linear running time.
n	Linear	Algorithms that running scan a list of size n belong to this class. (e.g. sequential search).
nlog n	n-log-n	Many divide-and-conquer algorithms including merge sort and quick sort in the average case fall into this category.

Class	Name	Comments
n^2	Quadratic	Typically characterizes efficiency of algorithms with two embedded loops. Elementary sorting algorithms and certain operations on n-by-n matrices are standard examples.

n^3	Cubic	Typically characterizes efficiency of algorithms with three embedded loops. Several nontrivial algorithms from linear algebra fall into this class.
2^n	Exponential	Typically for algorithms that generate all subsets of an n- element set. Often, the term “exponential” is used in a broader sense to include this and faster orders of growth as well.
n!	Factorial	Typical for algorithms that generate all permutations of an n- element set.

4. What is recursion and when to use recursion?

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.
```

The general procedure for any recursive algorithm is as follows,

1. Save the parameters, local variables and return addresses.
2. If the termination criterion is reached perform final computation and goto step 3 otherwise perform final computations and goto step 1

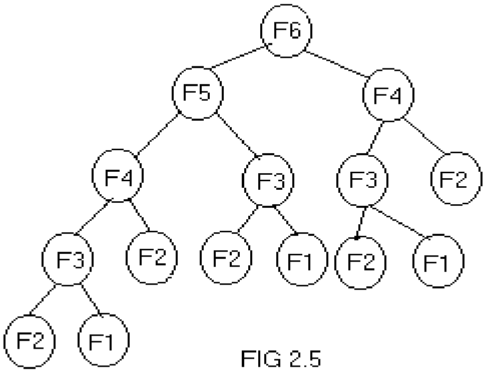


FIG 2.5

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

Iteration v/s Recursion:

Demerits of recursive algorithms:

1. Many programming languages do not support recursion; hence, recursive mathematical function is implemented using iterative methods.
2. Even though mathematical functions can be easily implemented using recursion it is always at the cost of execution time and memory space. For example, the recursion tree for generating 6 numbers in a Fibonacci series generation is given in fig 2.5. A Fibonacci series is of the form 0,1,1,2,3,5,8,13,…etc, where the third number is the sum of preceding two numbers and so on. It can be noticed from the fig 2.5 that, f(n-2) is computed twice, f(n-3) is computed thrice, f(n-4) is computed 5 times.

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

Demerits of iterative methods :

- Mathematical functions such as factorial and Fibonacci series generation can be easily implemented using recursion than iteration.
- In iterative techniques looping of statement is very much necessary.

Recursion is a top down approach to problem solving. It divides the problem into pieces or selects out one key step, postponing the rest.

Iteration is more of a bottom up approach. It begins with what is known and from this constructs the solution step by step. The iterative function obviously uses time that is O(n) where as recursive function has an exponential time complexity.

It is always true that recursion can be replaced by iteration and stacks. It is also true that stack can be replaced by a recursive program with no stack.

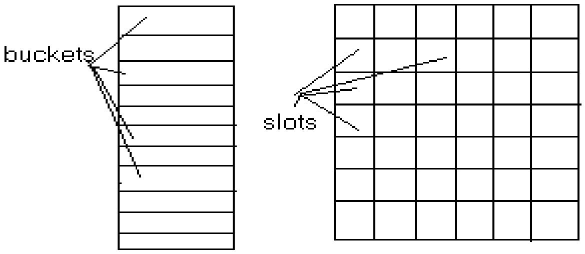


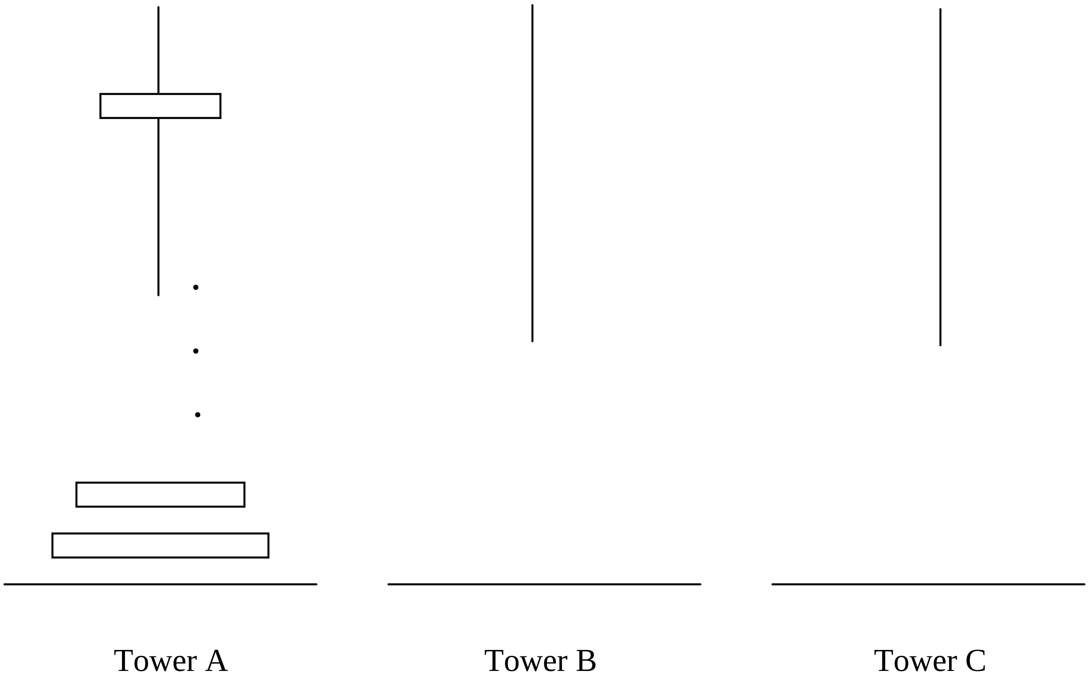
Fig 2.6

5. Describe 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.

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.

Algorithm:

```
Algorithm TowersofHanoi(n,x,y,z)
//Move the top ‘n’ disks from tower x to tower y.
{
        .
        .
        .
if(n>=1) then
{
        TowersofHanoi(n-1,x,z,y);
        Write(“move top disk from tower “ X ,”to top of tower “ ,Y);
Towersofhanoi(n-1,z,y,x);
}
}
```

2. Permutation Generator:

- Given a set of n>=1elements, the problem is to print all possible permutations of this set.
- For example, if the set is {a,b,c} ,then the set of permutation is,

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

- It is easy to see that given ‘n’ elements there are n! different permutations.
- A simple algorithm can be obtained by looking at the case of 4 statement(a,b,c,d)
- The Answer can be constructed by writing

1. a followed by all the permutations of (b,c,d)
2. b followed by all the permutations of(a,c,d)
3. c followed by all the permutations of (a,b,d)
4. d followed by all the permutations of (a,b,c)

Algorithm:

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

Performance Analysis:

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 statement 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

6. EXPLAIN Mathematical Analysis of Non recursive Algorithms.

Consider the problem of finding the value of the largest element in a list of n numbers.

ALGORITHM: Max element (A [0...n-1])

```
//Determines the value of the largest element in a given array
// Input: an array A [0...n-1] of real numbers
//Output: the value of the largest element in A
maxval   =     A[0]
for i  =  1 to n-1 do
        if A[i]  > maxval
                maxval  =   A[i]
                return maxval
```

There are two operations in the loop's body: the comparison A[i]>maxval and the assignment maxval = A[i]

Let us denote C(n) the number of times this comparison is executed. Which is repeated per each value of the loop's variable i within the bounds between 1 and n-1.

$$C(n) = \sum_{i=1}^{n-1} 1$$

$$C(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n)$$

Generals plan for analyzing efficiency of non-recursive algorithms:

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed depends only on the size of an input, if is also depends on some additional property, the worst case, average case, and best case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation, either fined a closed-form formula for the counter, of the very least, establish its order of growth.

Example

Consider the element uniqueness problem. Check whether all the elements in a given array are distinct. The following straightforward algorithm can solve this problem.

```
ALGORITHM:    Unique elements (A [0...n-1])
//check whether all the elements in a given array are distinct.
//input: an array A [0…n-1]
//output: returns "true" if all the elements in A are
//distinct and "false" otherwise
  for i  =   0 to n-2 do
        for j  =  i+1 to n-1 do
             if A[i]=A[j]
                  return false
return true.
```

For each value of the loop's variable j between its limits i+1 and n-1; and this is repeated for each value of the outer loop, for each value of the loop's variable i between its limits 0 and n-2. Accordingly, we get

$$C_{worst}(n)=\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1)-(i+1)+1] = \sum_{i=0}^{n-2} (n-1-I)$$

$$=\sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-2)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)^n}{2} \approx 1/2 \quad n^2 \in \Theta\,(n)^2$$

7. Explain Mathematical Analysis of Recursive Algorithms.

Computing the factorial function f (n)=n! For an arbitrary non-negative integer n. Since

$n!=1\ldots\ldots(n-1).n=(n-1)!.n$ for $n\geq1$

and $0!=1$ by definition.

We can compute f (n) =f (n-1). n with the following recursive algorithm

```
ALGORITHM  F (n)
//computes n! Recursively
//input: a nonnegative integer n
//output: the value of n!
if n=0          return 1
else            return F(n-1)*n
```

The function F (n) is computed according to the formula

$$F(n) = F(n-1).\, n \qquad \text{for } n>0,$$

A general plan for analyzing efficiency of recursive algorithms:

1. Decide on a parameter indicating an inputs size.
2. Identify the algorithm's basic operation
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst case, average case, and best case efficiencies must be investigated separately.
4. Setup a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or at least as certain the order of growth of its solution.

Algorithms for computing fibonacci numbers

Recursive algorithm for computing f (n)

```
ALGORITHM F (n):
 // Computers the nth Fibonacci number recursively by using its definition
//input: a non-negative integer n
// Output: the nth Fibonacci number
 if n≤ 1          return n
 else             return F (n-1) + F (n-2)
```

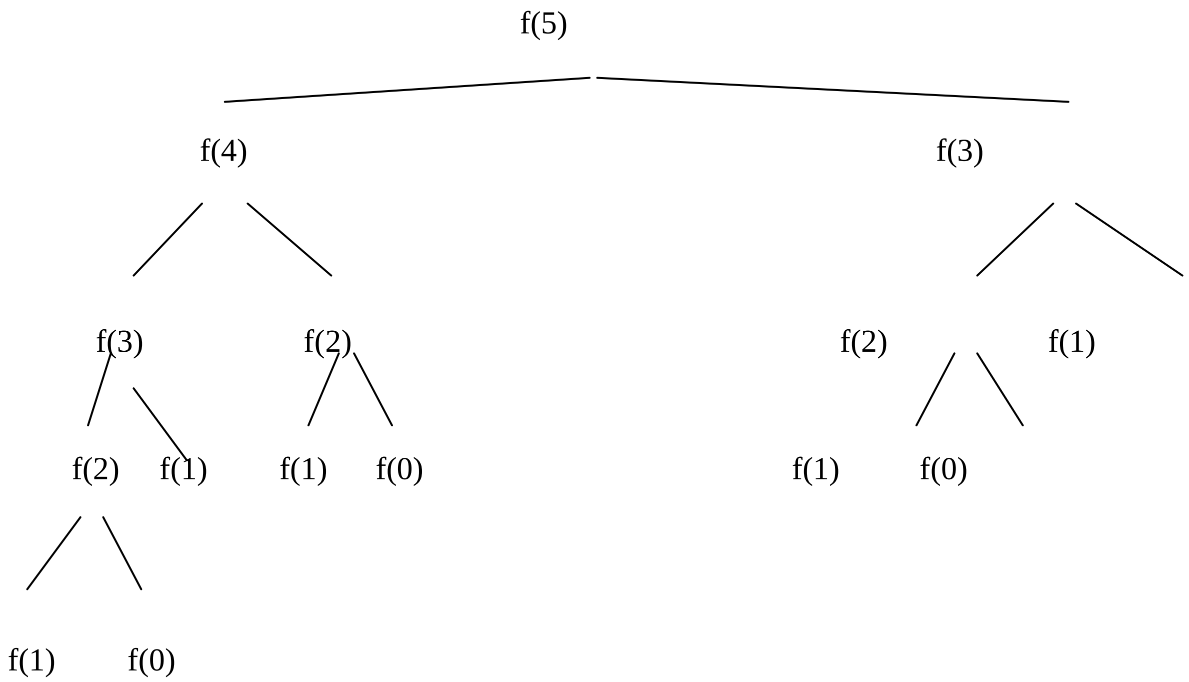


Figure 6.1: Tree of recursive calls for computing the Fibonacci number for n=5

Fibonacci sequence iteratively as is done in the following algorithm.

```
ALGORITHM: Fib(n)

//computes the nth Fibonacci number iteratively by using its definition

//input: a nonnegative integer n

//Output: the nth Fibonacci number

    F(0) = 0;

    F(1)  = 1;

    for i = 2 to n do

        F [i] = F[i-1] + F[i-2]

    return F[n]
```

8.Write about Solving recurrence equations.(UQ:APRIL 12) (UQ:Apr/May'14)

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

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

1. Intelligent guess work:

This approach generally proceeds in 4 stages.

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

Then this form is correct.

Consider the following recurrence,

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

- First step is to replace n ÷ 2 by n/2
- It is tempting to restrict ‘n’ to being ever since in that case n÷2 = n/2, but recursively dividing an even no. by 2, may produce an odd no. larger than 1.
- Therefore, it is a better idea to restrict ‘n’ to being an exact power of 2.
- First, we tabulate the value of the recurrence on the first few powers of 2.

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

* For instance, $T(16) = 3 * T(8) + 16$

$= 3 * 65 + 16$

$= 211.$

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

Then,

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

$= 3 * (3 * 1 + 2) + 4$

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

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

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

- The pattern is now obvious.

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

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

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

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

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

Proposition: (Geometric Series)

Let S_n be the sum of the first n terms of the geometric series a, ar, ar^2….Then $S_n = a(1-r^n)/(1-r)$, except in the special case when r = 1; when $S_n = a_n$.

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

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

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

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

$= 3^{k+1} – 2^{k+1}$

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

Eg : 2

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

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

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

$(x – 4)(x + 1) = 0$

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

General Solution, $f_n = C_1 r_1^n + C_2 r_2^n$ → (A)

$n=0$ ➔ $C_1 + C_2 = 0$ → (1)

$n=1$ ➔ $C_1 r_1 + C_2 r_2 = 5$ → (2)

Eqn 1 ➔ $C_1 = -C_2$

sub C_1 value in Eqn (2)

$-C_2 r_1 + C_2 r_2 = 5$

$C_2(r_2 – r_1) = 5$

$$C_2 = \frac{5}{r_2 – r_1}$$

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

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

$C_2 = -1 \quad , \quad C_1 = 1$

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

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

$\mathbf{f_n = 4^n + 1^n}$

Homogenous Recurrences :

* We begin our study of the technique of the characteristic equation with the resolution of homogenous linear recurrences with constant co-efficient, i.e the recurrences of the form,

$a_0t_n + a_1t_{n-1} + \ldots.. + a_kt_{n-k} = 0$

where the t_i are the values we are looking for.

* The values of t_i on ‘K’ values of i (Usually $0 \leq i \leq k-1$ (or) $0 \leq i \leq k$) are needed to determine the sequence.

* The initial condition will be considered later.

* The equation typically has infinitely many solution.

* The recurrence is,

→ linear because it does not contain terms of the form t_{n-i}, t_{n-j}, $t^2{}_{n-i}$, and soon.

→ homogeneous because the linear combination of the t_{n-i} is equal to zero.

→ With constant co-efficient because the a_i are constants

* Consider for instance our non familiar recurrence for the Fibonacci sequence,

$f_n = f_{n-1} + f_{n-2}$

* This recurrence easily fits the mould of equation after obvious rewriting.

$$f_n - f_{n-1} - f_{n-2} = 0$$

* Therefore, the fibonacci sequence corresponds to a homogenous linear recurrence with constant co-efficient with k=2,a_0=1&a_1=a_2 = -1.

1) (Fibonacci) Consider the recurrence.

$$f_n = \begin{cases} n & \text{if } n=0 \text{ or } n=1 \\ f_{n-1} + f_{n-2} & \text{otherwise} \end{cases}$$

We rewrite the recurrence as,

$$f_n - f_{n-1} - f_{n-2} = 0.$$

The characteristic polynomial is,

$$x^2 - x - 1 = 0.$$

The roots are,

$$x = \frac{-(-1) \pm \sqrt{((-1)2 + 4)}}{2}$$

$$= \frac{1 \pm \sqrt{(1 + 4)}}{2}$$

$$= \frac{1 \pm \sqrt{5}}{2}$$

$$r_1 = \frac{1 + \sqrt{5}}{2} \quad \text{and} \quad r_2 = \frac{1 - \sqrt{5}}{2}$$

The general solution is,

$$f_n = C_1 r_1^n + C_2 r_2^n$$

when n=0, $f_0 = C_1 + C_2 = 0$

when n=1, $f_1 = C_1 r_1 + C_2 r_2 = 1$

$$C_1 + C_2 = 0 \quad \rightarrow (1)$$

$$C_1 r_1 + C_2 r_2 = 1 \quad \rightarrow (2)$$

From equation (1)

$$C_1 = -C_2$$

Substitute C_1 in equation(2)

$$-C_2 r_1 + C_2 r_2 = 1$$

$$C_2[r_2 – r_1] = 1$$

Substitute r_1 and r_2 values

$$C_2 \quad \frac{1 - \sqrt{5}}{2} \quad - \quad \frac{1 - \sqrt{5}}{2} \quad = 1$$

$$C_2 \quad \frac{1 – \sqrt{5} – 1 – \sqrt{5}}{2} \quad = 1$$

$$\frac{-C_2 * 2\sqrt{5}}{2} = 1$$

$– \sqrt{5}C_2 = 1$

$C_1 = 1/\sqrt{5}$ $\qquad C_2 = -1/\sqrt{5}$

Thus,

$$f_n = \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2}\right)^n + \frac{-1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2}\right)^n$$

$$= \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2}\right)^n - \left(\frac{1 - \sqrt{5}}{2}\right)^n$$

3. Inhomogeneous recurrence :

* The solution of a linear recurrence with constant co-efficient becomes more difficult when the recurrence is not homogeneous, that is when the linear combination is not equal to zero.

* Consider the following recurrence

$a_0t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n p(n)$

* The left hand side is the same as before,(homogeneous) but on the right-hand side we have $b^n p(n)$, where,

→ b is a constant

→ p(n) is a polynomial in ‘n’ of degree ‘d’.

Example(1) :

Consider the recurrence,

$t_n – 2t_{n-1} = 3^n$ →(A)

In this case, b=3, p(n) = 1, degree = 0.

The characteristic polynomial is,

$(x – 2)(x – 3) = 0$

The roots are, $r_1 = 2$, $r_2 = 3$

The general solution,

$t_n = C_1 r_1^n + C_2 r_2^n$

$t_n = C_1 2^n + C_2 3^n$ → (1)

when n=0, $C_1 + C_2 = t0$ → (2)

when n=1, $2C_1 + 3C_2 = t1$ → (3)

sub n=1 in eqn (A)

$t_1 – 2t_0 = 3$

$t_1 = 3 + 2t_0$

substitute t_1 in eqn(3),

(2) * 2 → $2C_1 + 2C_2 = 2t_0$

$2C_1 + 3C_2 = (3 + 2t_0)$

$-C_2 = -3$

$C_2 = 3$

Sub $C_2 = 3$ in eqn (2)

$C_1 + C_2 = t_0$

$C_1 + 3 = t_0$

$C_1 = t_0 – 3$

Therefore $t_n = (t_0-3)2^n + 3. 3^n$

$= Max[O[(t_0 – 3) 2^n], O[3.3^n]]$

$= Max[O(2^n), O(3^n)]$ constants

$= O[3^n]$

Example :(2)

$t_n - 2t_{n-1} = (n + 5)3^n, n \geq 1$ →(A)

This is Inhomogeneous

In this case, b=3, p(n) = n+5, degree = 1

So, the characteristic polynomial is,

$(x-2)(x-3)^2 = 0$

The roots are,

$r_1 = 2, r_2 = 3, r_3 = 3$

The general equation,

$t_n = C_1r_1^n + C_2r_2^n + C_3nr_3^n$ → (1)

when n=0, $t_0 = C_1 + C_2$ →(2)

when n=1, $t_1 = 2C_1 + 3C_2 + 3C_3$ →(3)

substituting n=1 in eqn(A),

$t_1 - 2t_0 = 6 . 3$

$t_1 - 2t_0 = 18$

$t_1 = 18 + 2t_0$

substituting t_1 value in eqn(3)

$2C_1 + 3C_2 + 3C_3 = 18 + 2t_0$ →(4)

$C_1 + \quad C_2 + \qquad = t_0$ →(2)

Sub. n=2 in eqn(1)

$4C_1 + 9C_2 + 18C_3 = t_2$ → (5)

sub n=2 in eqn (A)

$t_2 - 2t_1 = 7. 9$

$t_2 = 63 + 2t_1$

$= 63 \ + 2[18 + 2t_0]$

$t_2 = 63 + 36 +4t_0$

$t_2 = 99 +4t_0$

sub. t_2 value in eqn(3),

$4C_1 +9C_2 + 18C_3 = 99 + 4t_0$ →(5)

solve eqn (2),(4) & (5)

n=0, $C_1 + \ C_2 = t_0$ →(2)

$n=1,\ 2C_1 + 3C_2 + 3C_3 = 18 + 2t_0 \rightarrow (4)$

$n=2,\ 4C_1 + 9C_2 + 18C_3 = 99 + 4t_0 \rightarrow (5)$

$(4) * 6 \Rightarrow 12C_1 + 18C_2 + 18C_3 = 108 + 2t_0 \rightarrow (4)$

$(5) \Rightarrow 4C_1 + 9C_2 + 18C_3 = 99 + 4t0 \rightarrow (5)$

$8C_1 + 9C_2 = 9 + 8t_0 \rightarrow (6)$

$(2) * 8 \Rightarrow 8C_1 + 8C_2 = 8t_0 \rightarrow (2)$

$(6) \Rightarrow 8C_1 + 9C_2 = 9 + 8t_0 \rightarrow (6)$

$-C_2 = -9$

$C_2 = 9$

Sub, $C_2 = 9$ in eqn(2)

$C_1 + C_2 = t_0$

$C_1 + 9 = t_0$

$C_1 = t_0 - 9$

Sub C_1 & C_2 in eqn (4)

$2C_1 + 3C_2 + 3C_3 = 18 + 2t_0$

$2(t_0-9) + 3(9) + 3C_3 = 18 + 2t_0$

$2t_0 - 18 + 27 + 3C_3 = 18 + 2t_0$

$2t_0 + 9 + 3C_3 = 18 + 2t_0$

$3C_3 = 18 - 9 + 2t_0 - 2t_0$

$3C_3 = 9$

$C_3 = 9/3$

$C_3 = 3$

Sub. $C_1, C_2, C_3, r_1, r_2, r_3$ values in eqn (1)

$t_n = C_1 2^n + C_2 3^n + C_3.n.3^n$

$= (t_0 - 9)2^n + 9.3^n + 3.n.3^n$

$= \text{Max}[O[(t_0-9),2^n], O[9.3^n], O[3.n.3^n]]$

$= \text{Max}[O(2^n), O(3^n), O(n3^n)]$

$tn = O[n3^n]$

Example: (3)

Consider the recurrence,

$$t_n = \begin{cases} 1 & \text{if } n=0 \\ 4t_{n-1} - 2^n & \text{otherwise} \end{cases}$$

$t_n - 4t_{n-1} = -2_n \rightarrow (A)$

In this case , c=2, p(n) = -1, degree =0

$(x-4)(x-2) = 0$

The roots are, $r_1 = 4$, $r_2 = 2$

The general solution ,

$t_n = C_1 r_1^n + C_2 r_2^n$

$t_n = C_1 4^n + C_2 2^n \rightarrow (1)$

when n=0, in (1) ➔ $C_1 + C_2 = 1 \rightarrow (2)$

when n=1, in (1) ➔ $4C_1 + 2C_2 = t1 \rightarrow (3)$

sub n=1 in (A),

$t_n - 4t_{n-}1 = -2^n$

$t_1 - 4t_0 = -2$

$t_1 = 4t_0 - 2$ [since $t_0 = 1$]

$t_1 = 2$

sub t1 value in eqn (3)

$4C_1 + 2C_2 = 4t_0 - 2 \rightarrow (3)$

(2) * 4 ➔ $4C_1 + 4C_2 = 4$

----------------------------

$-2C_2 = 4t_0 - 6$

$= 4(1) - 6$

$= -2$

$C_2 = 1$

$-2C_2 = 4t_0 - 6$

$2C_2 = 6 - 4t_0$

$C_2 = 3 - 2t_0$

$3 - 2(1) = 1$

$C_2 = 1$

Sub. C_2 value in eqn(2),

$C_1 + C_2 = 1$

$C_1 + (3-2t_0) = 1$

$C_1 + 3 - 2t_0 = 1$

$C_1 = 1 - 3 + 2t_0$

$C_1 = 2t_0 - 2$

$= 2(1) - 2 = 0$

$C_1 = 0$

Sub C_1 & C_2 value in eqn (1)

$t_n = C_1 4^n + C_2 2^n$

$= \text{Max}[O(2t_0 - 2).4^n, O(3 - 2t_0).2^n]$

$= \text{Max}[O(2^n)]$

$\mathbf{t_n = O(2^n)}$

Example : (4)

$$t_n = \begin{cases} 0 & \text{if } n=0 \\ 2t_{n-1} + n + 2^n & \text{otherwise} \end{cases}$$

$t_n - 2t_{n-1} = n + 2^n \rightarrow (A)$

There are two polynomials.

For n; b=1, p(n), degree = 1

For 2n; b=2, p(n) = 1, degree =0

The characteristic polynomial is,

$(x-2)(x-1)^2(x-2) = 0$

The roots are, $r_1 = 2$, $r_2 = 2$, $r_3 = 1$, $r_4 = 1$.

So, the general solution,

$$t_n = C_1 r_1^n + C_2 n r_2^n + C_3 r_3^n + C_4\, n\, r_4^n$$

sub r_1, r_2, r_3 in the above eqn

$$t_n = 2^n C_1 + 2^n C_2\, n + C_3 \,.\, 1^n + C_4 .n.1^n \quad \rightarrow (1)$$

sub. $n=0 \Rightarrow C_1 + C_3 = 0 \qquad \rightarrow (2)$

sub. $n=1 \Rightarrow 2C_1 + 2C_2 + C_3 + C_4 = t_1 \qquad \rightarrow (3)$

sub. n=1 in eqn (A)

$t_n - 2t_{n-1} = n + 2^n$

$t_1 – 2t_0 = 1 + 2$

$t_1 – 2t_0 = 3$

$t_1 = 3$ [since $t_0 = 0$]

sub. n=2 in eqn (1)

$2^2 C_1 + 2.\, 2^2.C_2 + C_3 + 2.C_4 = t_2$

$4C_1 + 8C_2 + C_3 + 2C_4 = t_2$

sub n=2 in eqn (A)

$t_2 – 2t_1 = 2 + 2^2$

$t_2 – 2t_1 = 2 + 4$

$t_2 – 2t_1 = 6$

$t_2 = 6 + 2t_1$

$t_2 = 6 + 2.3$

$t_2 = 6 + 6$

$t_2 = 12$

$\Rightarrow 4C_1 + 8C_2 + C_3 + 2C_4 = 12 \quad \rightarrow (4)$

sub n=3 in eqn (!)

$2^3 C_1 + 3.2^3.C_2 + C_3 + 3C_4 = t_3$

$3C_1 + 24C_2 + C_3 + 3C_4 = t_3$

sub n=3 in eqn (A)

$t_3 – 2t_2 = 3 + 2^3$

$t_3 – 2t_2 = 3 + 8$

$t_3 – 2(12) = 11$

$t_3 – 2_4 \;\; = 11$

$t_3 = 11 + 24$

$t_3 = 35$

➔ $8C_1 + 24C_2 + C_3 + 3C_4 = 35$ →(5)

n=0, solve; $C_1 + C_3 = 0$ →(2)

n=1,(2), (3), (4)&(5) $2C_1 + 2C_2 + C_3 + C_4 = 3$ →(3)

n=2, $4C_1 + 8C_2 + C_3 + 2C_4 = 12$ →(4)

n=3, $8C_1 + 24C_2 + C_3 + 3C_4 = 35$ →(5)

---------------------------------------

$-4C_1 – 16C_2 – C_4 = -23$ →(6)

solve: (2) & (3)

(2) ➔ $C_1 + C_3 = 0$

(3) ➔ $2C_1 + C_3 + 2C_2 + C_4 = 3$

---------------------------------

$-C_1 – 2C_2 – C_4 = -3$ →(7)

solve(6) & (7)

(6) ➔ $-4C_1 – 16C_2 – C_4 = -23$

(7) ➔ $-C_1 – \;\; 2C_2 – C_4 = -3$

---------------------------------

$-3C_1 - 14C_2 = 20 \quad \rightarrow (8)$

4. Change of variables:

- It is sometimes possible to solve more complicated recurrences by making a change of variable.
- In the following example, we write T(n) for the term of a general recurrences, and t_i for the term of a new recurrence obtained from the first by a change of variable.

Example: (1)

Consider the recurrence,

$$T(n) = \begin{cases} 1 & \text{, if } n=1 \\ 3T(n/2) + n & \text{, if 'n' is a power of 2, } n>1 \end{cases}$$

➔ Reconsider the recurrence we solved by intelligent guesswork in the previous section, but only for the case when 'n' is a power of 2

$$T(n) = \begin{cases} 1 \\ 3T(n/2) + n \end{cases}$$

* We replace 'n' by 2^i.

* This is achieved by introducing new recurrence t_i, define by $t_i = T(2^i)$

* This transformation is useful because n/2 becomes $(2^i)/2 = 2^{i-1}$

* In other words, our original recurrence in which T(n) is defined as a function of T(n/2) given way to one in which t_i is defined as a function of t_{i-1}, precisely the type of recurrence we have learned to solve.

$$t_i = T(2^i) = 3T(2^{i-1}) + 2^i$$

$$t_i = 3t_{i-1} + 2^i$$

$$t_i - 3t_{i-1} = 2^i \quad \rightarrow (A)$$

In this case,

$b = 2, p(n) = 1$, degree = 0

So, the characteristic equation,

$(x - 3)(x - 2) = 0$

The roots are, r1 = 3, r2 = 2.

The general equation,

$t_n = C_1 r_1^i + C_2 r_2^i$

sub. r_1 & r_2: $t_n = 3^n C_1 + C_2 2^n$

$t_n = C_1 3^i + C_2 2^i$

We use the fact that, $T(2^i) = t_i$ & thus T(n) = tlogn when $n= 2^i$ to obtain,

$T(n) = C_1 . 3^{\log_2 n} + C_2 . 2^{\log_2 n}$

$T(n) = C_1 . n^{\log_2 3} + C_2.n$ [i = logn]

When 'n' is a power of 2, which is sufficient to conclude that,

T(n) = O($n^{\log 3}$) 'n' is a power of 2

Example: (2)

Consider the recurrence,

$T(n) = 4T(n/2) + n^2 \rightarrow (A)$

Where 'n' is a power of 2, $n \geq 2$.

$t_i = T(2^i) = 4T(2^{i-1}) + (2^i)^2$

$t_i = 4t_{i-1} + 4^i$

$\Rightarrow t_i - 4t_{i-1} = 4^i$

In this eqn,

$b = 4, P(n) = 1$, degree = 0

The characteristic polynomial,

$(x - 4)(x - 4) = 0$

The roots are, $r_1 = 4$, $r_2 = 4$.

So, the general equation,

$t_i = C_1 4^i + C_2 4^i.i$ [since i = logn]

$= C_1\ 4^{\log n} + C_2.\ 4^{\log n}\ .\ \log n$ [since $2^i = n$]

$= C_1\ .\ n^{\log 4} + C_2.\ n^{\log 4}.n^{\log 1}$

$T(n) =\ O(n^{\log 4})$ 'n' is the power of 2.

EXAMPLE : 3

$T(n) = 2T(n/2) + n \log n$

When 'n' is a power of 2, $n \geq 2$

$t_i = T(2^i) = 2T(2^i/2) + 2^i\ .i$ [since $2^i = n$; $i = \log n$]

$t_i - 2t_{i-1} = i.\ 2^i$

In this case,

$b = 2$, $P(n) = i$, degree = 1

$(x - 2)(x - 2)^2 = 0$

The roots are, $r_1 = 2$, $r_2 = 2$, $r_3 = 2$

The general solution is,

$t_n = C_1 2^i + C_2.\ 2^i\ .\ i + C_3.\ i^2.\ 2^i$

$= nC_1 + nC_2 + nC_3(\log n^2{}_2 n)$

$t_n = O(n.\log^2{}_2 n)$

Example: 4

$$T(n) = \begin{cases} 2 & ,n=1 \\ 5T(n/4) + Cn^2 & , n>1 \end{cases}$$

$t_i = T(4^i) = 5T(4^i/4) + C(4^i)^2$

$= 5T\ 4^{\ i-1} + C.\ 16^i$

$= 5t_{\ i-1} + C.16^i$

$t_i - 5t_{\ i-1} = C.\ 16^i$

In this case,

$b = 16$, $P(n) = 1$, degree $= 0$

The characteristic eqn,

$(x - 5)(x - 16) = 0$

The roots are, $r_1 = 5$, $r_2 = 16$

The general solution,

$t_i = C_1.5^i + C_2.16^i$

$= C_1.5^i + C_2.(4^2)^i$

$t_n = O(n^2)$

EXAMPLE: 5

$$T(n) = \begin{cases} 2 & , n = 1 \\ T(n/2) + Cn & , n > 1 \end{cases}$$

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

$= T(2^i/2) + C.\ 2^i$

$= T(2^{i-1}) + C. 2^i$

$t_i = t_{i-1} + C. 2^i$

$t_i – t_{i-1} = C. 2^i$

In this case, b=2, P(n) =1, degree =0

So, the characteristic polynomial,

$(x -1)(x – 2) = 0$

The roots are, $r_1 = 1$, $r_2 = 2$

$t_i = C_1. 1^i + c_2. 2^i$

$= C_1. 1^{\log_2 n} + C_2.n$

$= C_1 . n^{\log_2 1} + C_2.n$

$t_n = O(n)$

EXAMPLE: 6

$$T(n) = \begin{cases} 1 & , n =1 \\ 3T(n/2) + n; & n \text{ is a power of } 2 \end{cases}$$

$t_i = T(2^i) = 3T(2^i/2) + 2^i$

$= 3T(2^{i-1}) + 2^i$

$t_i = 3t_{i-1} + 2^i$

So, b = 2, P(n) =1, degree = 0

$(x – 3)(x – 2) = 0$

The roots are, $r_1 = 3$, $r_2 = 2$

$t_i = C_1. 3^i + C_2. 2^i$

$= C_1. 3^{\log_2 n} + C_2. 2^{\log_2 n}$

$= C_1. n^{\log_2 3} + C_2. n^{\log_2 2} = 1$

$= C_1. n^{\log_2 3} + C_2.n$

$\mathbf{t_n = O(n^{\log_2 3})}$

EXAMPLE: 7

$T(n) = 2T(n/2) + n. \log n$

$t_i = T(2^i) = 2T(2^i/2) + 2^i . i$

$= 2T(2^{i-1}) + i. 2^i$

$= 2t_{i-1} + i.2^i$

$t_i - 2t_{i-1} = i. 2i$

➔ b=2, P(n) = I, degree = 1

The roots is $(x - 2)(x - 2)2 = 0$

$x = 2,2,2$

General solution,

$t_n = C_1.r_1^i + C_2.i.r_2^i + C_3. i2. r_3^i$

$= C_1.2^i + C_2. i.2^i + C_3. i^2.2^i$

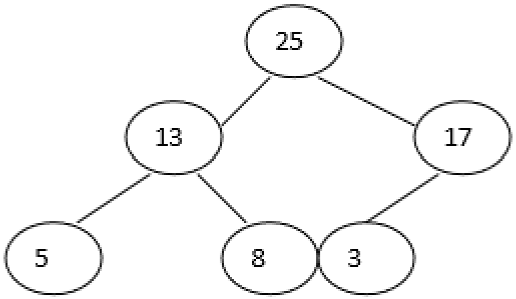
$= C_1. n+C_2.n.^{\log_2 n} + C_3.i^2.n$

$= C_1.n + C_2.n.^{\log_2 n} + C_3(2^{\log_2 n}).n$

$\mathbf{t_n = O(n. 2^{\log_2 n})}$

9. Write and analyze about heap sort (UQ Nov'12)

A heap is a complete binary tree with the property that the value at each node is at least as large as the value at its children.



An array [25, 13, 17, 5, 8, 3] is represented as heap above. The maximum element in an array is always a root node.

- The indices of its parent, left child and right child can be computed as
  PARENT (i) = floor (i/2)
  LEFT (i) = 2i
  RIGHT (i) = 2i + 1

- All the tree levels are completely filled except possibly for the lowest level, which is filled from the left up to a point.
- The levels above the lowest level from a complete binary tree of height h-1 contain 2^h –1 node.
- A heap of height h has
  - Maximum no. of elements =2^{h+1} nodes.(When lowest level is completely filled)
  - Minimum no. of nodes=2^h nodes.

Height of a node:

It is defined as no. of edges on a simple downward path from a node to leaf.

Height of a tree:

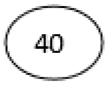
It is defined as no. of edges on a simple downward path from a root node to leaf.

To insert an element into the heap, one adds it "at the bottom" of the heap and then compares it with its parent, grandparent, great grandparent and so on, until it is less than or equal to one of these values.

Example:

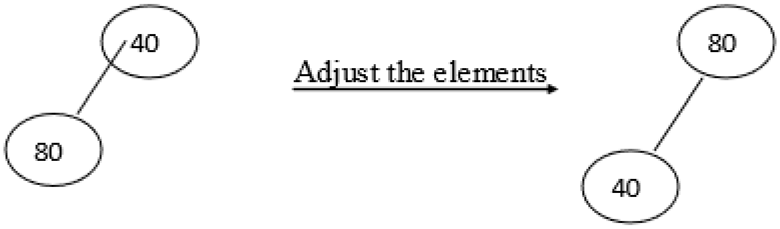
a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
40	80	35	90	45	50	70

Step1: Take the first element of an array.a[1]=40.

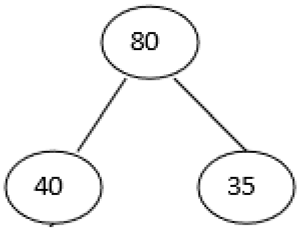


Step 2: Take the second element of an array.a[2]=80.

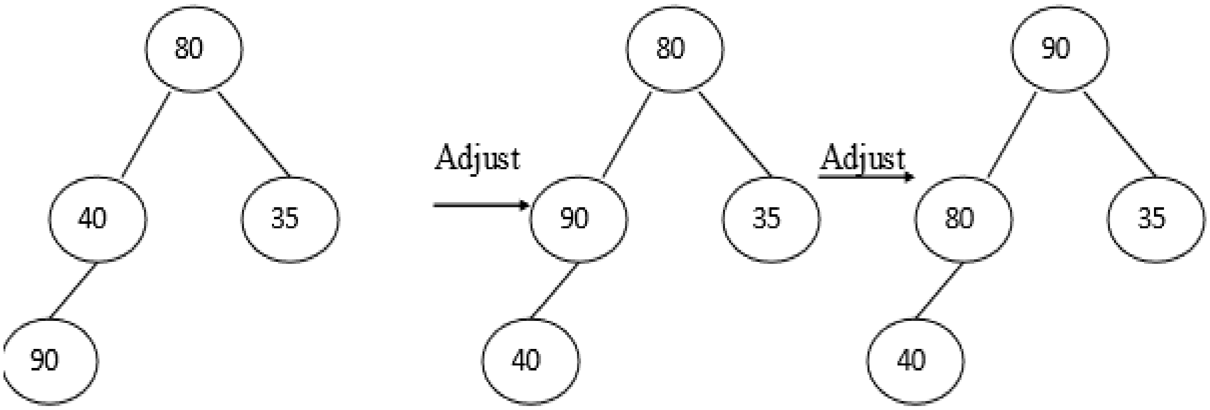
Insert a[2] to the left side of the first element.



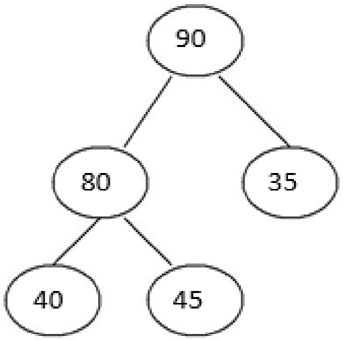
Step 3: Take the 3rd element .a[3]=35.Add this element to the right side of the root node.



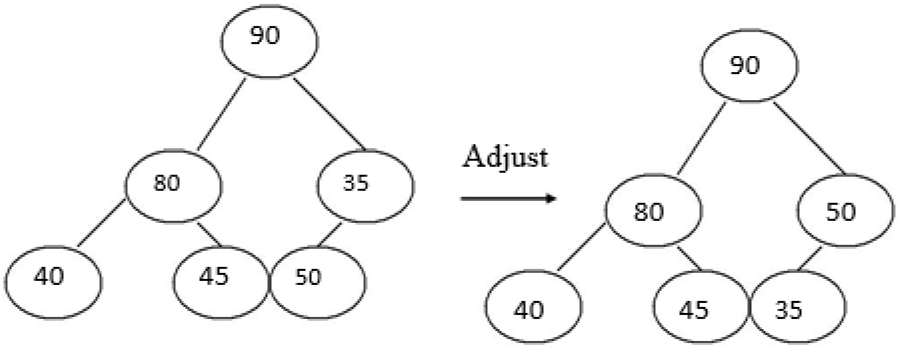
Step 4: Take the 4th element .a[4]=90.Add this element to the left side of the node 40



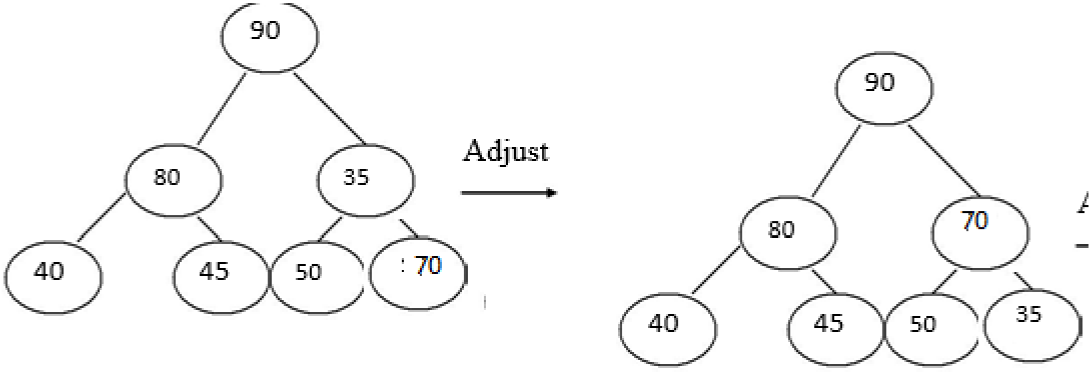
Step 5: Take the 5th element .a[5]=45.Add this element to the right side of the node 80.



Step 6: Take the 6th element .a[6]=50.Add this element to the left side of the node 35.



Step 7: Take the 7th element .a[7]=70.Add this element to the right side of the node 50.



The figure shows one example of how insert ( ) would insert a new value into an existing heap. To do heap sort, first add the first element in the tree. Then add the second element to the tree. If the tree does not satisfy the heap property then adjust the nodes in the tree. Then insert the next element and adjust the nodes. Repeat the above steps till the insertion of last element. Finally we obtain the sorted elements in heap tree representation.

Procedures for Heap Sort:

1. Build a Heap(Heapify)
2. Adjust

Procedure Heapify( ):

- Create a heap from n elements.
- Maintain the heap property. That‟s why we are using adjust ( ) procedure to arrange the elements.

Procedure Adjust ( );

Adjust takes as input the array a[ ] and integer I and n. It regards a[1..n] as a complete binary tree. If the sub trees rooted at 2I and 2I+1 are max heaps, then adjust will rearrange elements of a[ ] such that the tree rooted at I is also a max heap. The maximum elements from the max heap a[1..n] can be deleted by deleting the root of the corresponding complete binary tree. The last element of the array, i.e. a[n], is copied to the root, and finally we call Adjust(a,1,n-1).

```
Algorithm Heapsort(a,n)
{
    Heapify(a,n);
    //Transform the array into heap.
    //Interc hange the new ma with the element at the end of the
    array. for i=n to 2 step –1 do
    {
        t:=a[i];
        a[i]:=a[1];
        a[1]:=t;
        Adjust(a,1,i-1);
    }
}
```

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

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

Analysis:

- Heapify() requires n operations.->o(n)

- Adjust() requires o(log n) operations for each invocation.Adjust is called n times by Heapsort( ).So time is o(n log n).
- Heapsort = O(n) +O(n log n) =O(n).

Algorithm HeapSort( ):
    T(n)=O(n)

Algorithm Heapify( ):
    T(n)=O(n)

Algorithm Adjust( ):
    T(n)=O(n log n)

10. Write and analyze about shell sort

- Shell sort, named after its inventor, Donald Shell, was one of the first algorithms to break the quadratic time barrier.
- It works by comparing elements that are distant; the distance between comparisons decreases as the algorithm runs until the last phase, in which adjacent elements are compared. For this reason, Shell sort is sometimes referred to as ***diminishing increment*** **sort.**
- Shell sort uses a sequence, h_1, h_2, . . . , h_t, called the *increment sequence*. Any increment sequence will do as long as h_1 = 1, but obviously some choices are better than others.
- All elements spaced h_k apart are sorted. The file is then said to be *h_k-sorted.*
- For example, Figure shows an array after several phases of Shell sort.
- Shellsort routine using Shell's increments (better increments are possible)

PROGRAM

```
void shell(int a[10],int n)
{
int k,i,j,temp;
for(k=n/2;k>0;k=k/2)
for(i=k;i<n;i++)
{
temp=a[i];
for(j=i;(j>=k)&&(a[j-k]>temp);j=j-k)
{
a[j]=a[j-k];
}
a[j]=temp;
}
}
```

40 80 35 75 60 57 34 90 70 45.

__STEP 1__ : Find Gap value. [Gap value = Total no. of elements/2].

Gap. value = $10/2$ = 5.

__STEP 2__ : $\underset{\uparrow}{40} \quad \underbrace{80 \quad 35 \quad 75 \quad 60 \quad 57}_{5^{th}\ element} \quad 34 \quad 90 \quad 70 \quad 45. \rightarrow (1)$

Initially 57th is the 5th element to 40. [Gap. value = 5].

__STEP 3__ : Increment Pointer and find 5th element of its respective Pointer. [Eg: For 80, 5th element is 34]. Continue till last. Because Gap. value is equal to 5.

40	57
80	34
35	90
75	70
60	45

→ (A).

__STEP 4__ : Sort the elements in an row manner.

40	57
34	80
35	90
70	75
45	60.

→ (B).

STEP 5: CONDITION.

Sort each set of element. If you made any change of position in Ⓑ from Ⓐ. Then Same Position have to be changed in ①.

40 34 35 70 45 57 80 90 75 60. → ②.

STEP 6: Again Find Gap. value. Now In, Gap value, the total no. of elements should be Considered as previous gap value.

$$\text{Gap} \cdot \text{value} = 5/2 = 2.5 = 2.$$

STEP 7: Continue from Step 2. [Now Gap·value = 2]. But apply the Steps in ② not ①.

40 34 35 70 45 57 80 90 75 60.

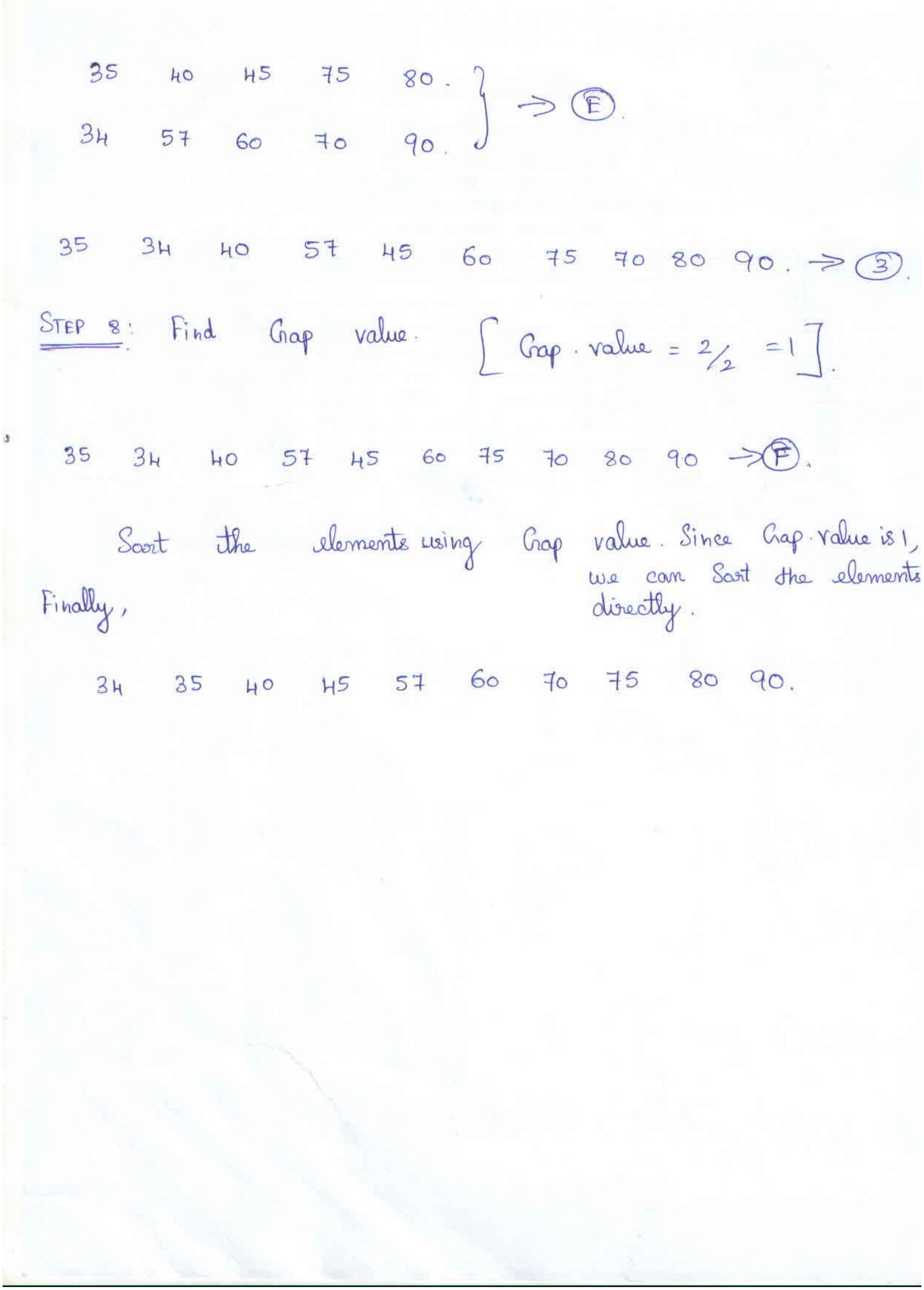
↑

40 35 45 80 75.
34 70 57 90 60. } → Ⓒ Sort the elements.

35 40 45 75 80.
34 57 70 60 90. } → Ⓓ

35 40 45 75 80
34 57 60 60 90.

35 34 40 57 45 70 75 60 80 90.



35 40 45 75 80.

34 57 60 70 90. } → Ⓔ.

35 34 40 57 45 60 75 70 80 90. → ③.

STEP 8: Find Gap value. [Gap . value = $2/2$ = 1].

35 34 40 57 45 60 75 70 80 90 → Ⓕ.

Sort the elements using Gap value. Since Gap . value is 1, we can Sort the elements directly.

Finally,

34 35 40 45 57 60 70 75 80 90.

TIME & SPACE COMPLEXITY

Worst Case : O(n^2) , Best Case : O(n log n), Average case: O($n^{1.5}$)

Advantages:

- Fastest algorithm for sorting smaller elements.
- Requires less memory

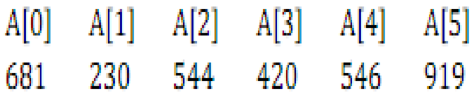
11. Write and analyze about radix or bucket sort (UQ APRIL12)

- Radix sort is otherwise called as bucket sort
- Let A be the array of 'n' nos.
- Our aim is to sort the nos in ascending order

IMPLEMENTATION:

1. Initialize 10 queues, Q[0],Q[1]………..Q[9]. Set FRONT[i]=REAR[i]=-1 where i=1,2………9.
2. Scan the array from left to right and find the least significant digit for all array elements.
3. If the digit is 0, then push Q[0]. If it is 1, then push Q[1] and if it is 2, then push Q[2]…….. up to if it is 9, hen push Q[9].
4. After pushing the elements, then pop the elements from Q[0] to A[9] and restore in an array.
5. Again scan the array from left to right and find second least significant digit for all elements.
6. Repeat step 3 and step 4.
7. No. of scanning process depends upon the width of the elements. If the width is 3, then we need 3 can for finding first least digit, second least significant digit and third significant digit.

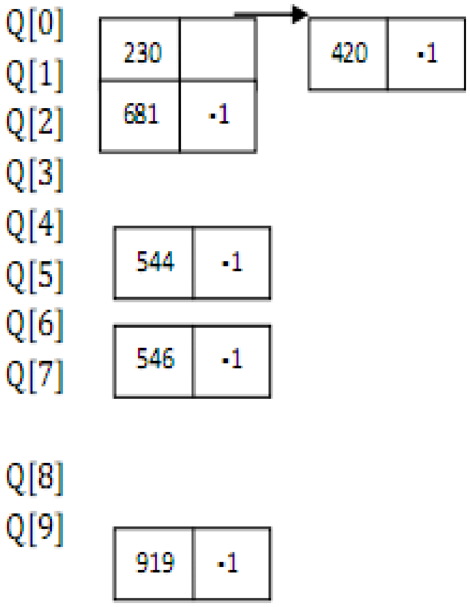
Consider the elements,



A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
681	230	544	420	546	919

FRONT[0] = REAR[0] = -1
FRONT[9] = REAR[9] = -1

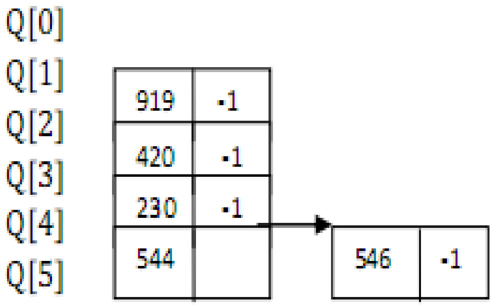
SCAN -1:

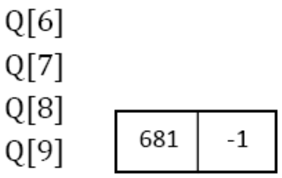


Delete the elements from the queue and store it in an array.

230 420 681 544 546 919

SCAN-2:

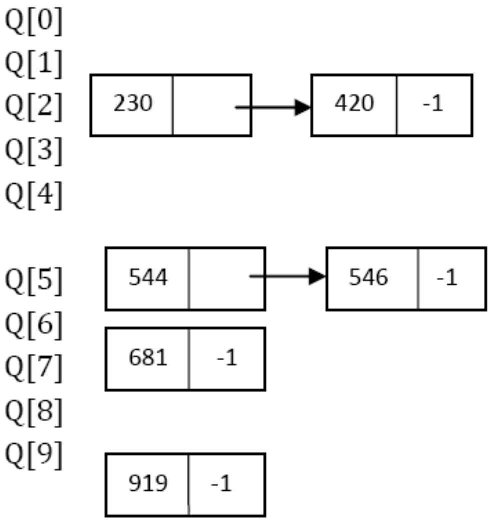




Delete the elements from the queue and store it in an array.

919 420 230 544 546 681

SCAN-3:



Delete the elements from the queue and store it in an array.

Delete the elements from the queue and store it in an array.
 230 420 544 546 681 919

PROGRAM:

```
 for (k = 0; k < 3; k++)

 {

    for (i = 0; i < count; i++)

    {

       min = array[i] % 10;        /* To find minimum lsd */

       t = i;

       for (j = i + 1; j < count; j++)

       {

          if (min > (array[j] % 10))

          {

             min = array[j] % 10;

             t = j;

          }
    }
       temp = array1[t];
       array1[t] = array1[i];
       array1[i] = temp;
       temp = array[t];
       array[t] = array[i];
       array[i] = temp;
     }
    for (j = 0; j < count; j++)       /*to find MSB */
       array[j] = array[j] / 10;
  }
```

TIME AND SPACE COMPLEXITY OF RADIX SORT:

The average case for radix sort is O(m+n), the worst case is also O(m+n).

Analysis:

- Running time depends on the stable sort.
- Each digit is in the range of 1 to k.
- Each pass over n takes θ (n+k) time.
- There are d no. of passes.
- Total time for radix sort is θ(dn+kd) where d is constant and k is θ (n)
- The radix sort is linear in time.

12.Explain about insertion sort with example.

Sorting techniques:

- ○ File is a collection of data items.
- ○ Each data item in a file is called record.
- ○ The records are numbered by R[1],R[2],R[3]……… and so on up to R[n]
- ○ The key is associated with each record, using the key only we can identify the particular record.
- ○ Sorting a group of nos or sequences of nos or data items means rearranging them in either ascending rder or descending order.

INSERTION SORTING:

□ Let A be the array of ‘n’ nos.

□ Our aim is to sort the numbers in ascending order.

□ Scan the array from A[1] to A[n-1] and find the smallest element A[R] where R=1,2,3……(N-1) and insert into the proper position previously sorted sub-array, A[1],A[2]…..A[R-1]

□ If R=1, the sorted sub-array is empty, so A[1] is sorted itself.

□ If R=2, A[2] is inserted into the previously sorted sub-array A[1], i.e., A[2] is inserted either before A[1] or after A[1]

□ If R=3, A[3] is inserted into the previously sorted sub-array A[1],A[2] i.e., A[3] is inserted either before A[1] or after A[2] or in between A[1] and A[2].

□ We can repeat the process for(n-1) times, and finally we get the sorted array.

Eg.,

A[1]	**A[2]**	**A[3]**	**A[4]**	**A[5]**	**A[6]**
[56]	**91**	35	42	68	78
[56	**91]**	35	42	68	78
[35	**56**	**91]**	42	68	78
[35	**42**	**56**	**91]**	68	78
[35	**42**	**56**	**68**	**91]**	78
[35	**42**	**56**	**68**	**78**	**91]**

PROGRAM

```
void insertion(int a[10])
{
int i,j,temp;
for(i=0;i<=n-1;i++)
{
temp=a[i];
j=i-1;
while ((j>=0)&&(a[j]>temp))
{
a[j+1]=a[j];
j=j-1;
}
a[j+1]=temp;
}
```

TIME AND SPACE COMPLEXITY OF SELECTION SORTING:

Worst Case : $O(n^2)$, Best Case : O(n), Average case: $O(n^2)$

Limitations:

- Efficient for smaller list(sorted list)
- Expensive

Analysis:

Best-Case:

The while- loop in line 5 executed only once for each *j*. This happens if given array A is already sorted.

T(n) = an + b =
O(n) T(n)=O(n)

It is a linear function of
n. Worst-Case:

The worst-case occurs, when line 5 executed *j* times for each *j*. This can happens if array A starts out in reverse order .

$T(n) = an^2 + bc + c = O(n^2)$
$T(n)=O(n^2)$

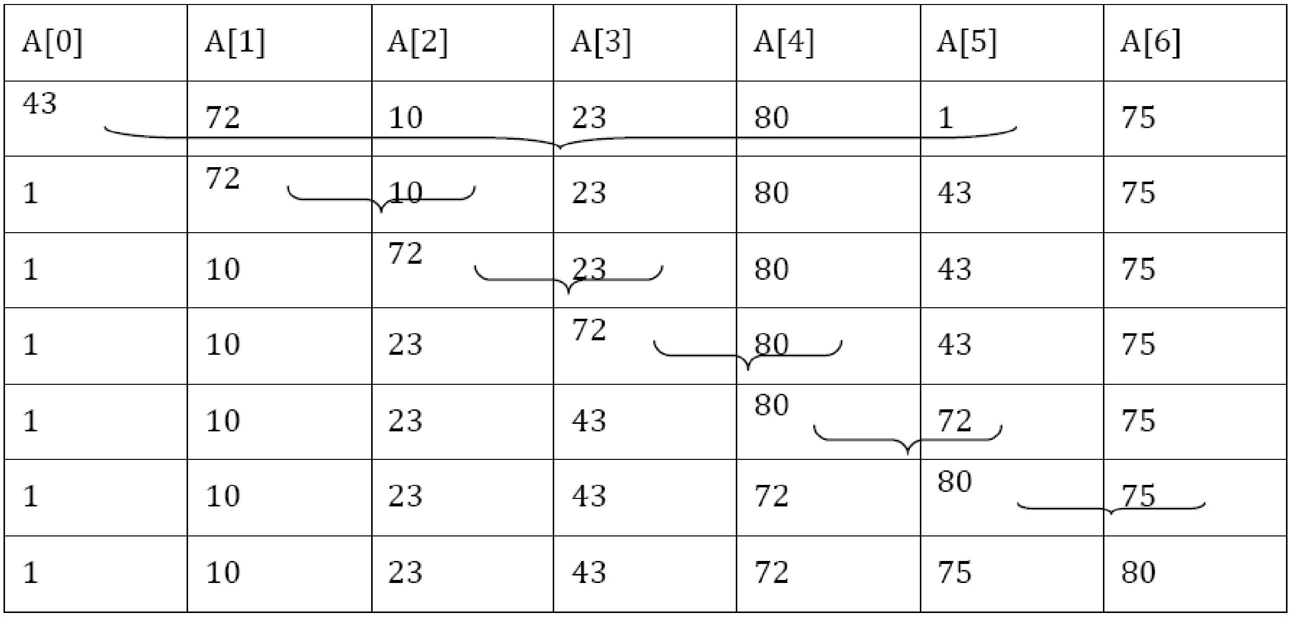
It is a quadratic function of *n*.

Extra Memory

This algorithm does not require extra memory.

11. Explain about selection sort? **(UQ APRIL13 & Nov’12)**

- Let A be the array of n numbers.
- Our aim is to sort the nos in ascending order.
- First find the smallest element position in the array(say i) then interchange zeroth position and ith position element.
- Find the second smallest element position in the array(say j),then interchange first position and jth position element.
- We can repeat the process up to (n-1) times. Finally we will be getting sorted array



A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
43	72	10	23	80	1	75
1	72	10	23	80	43	75
1	10	72	23	80	43	75
1	10	23	72	80	43	75
1	10	23	43	80	72	75
1	10	23	43	72	80	75
1	10	23	43	72	75	80

PROGRAM:

```
for(i=1;i<=n-1;i++)
        {
                min=a[i];
                k=i;
                for(j=i+1;j<=n;j++)
                {
                        if(a[j]<min)
                        {
                                min=a[j];
                                k=j;
                        }
                        a[k]=a[i];
                }
                a[i]=min;
        }
```

TIME AND SPACE COMPLEXITY OF SELECTION SORTING:

The algorithm, the search for the record with the next smallest key is called a pass. There are n-1 such passes required in order to perform the sort. This is because each pass places one record into its proper location.

During the first pass, in which the record with the smallest key is found, n-1 records are compared. In general, for the ith pass of the sort, n-i comparisons are required. The total number of comparisons is herefore, the sum

$$\sum_{i-1}^{n-1}(n-i)=\frac{1}{2}n(n-1)$$

Therefore, the no. of comparisons is proportional to n^2. i.e., $O(n^2)$.

Analysis:

The number of moves with this technique is always of the order O(n).So the time complexity is O(n).

13.Explain in detail bubble sorting. (UQ APRIL 13,NOV’14)

Bubble sorting is otherwise called sinking sorts

- The idea of bubble sorting is to move the highest element to nth position.
- The principle of bubble sorting is scan or read the array (n-1) times.
- For each scan, do the following steps.

Scan-1:

1. Take the first no and compare with the second no. if it is small, don’t interchange, otherwise interchange the elements.
2. Take the second no and compare with the third no. if it is small, don’t interchange, otherwise interchange the elements.
3. This process is continued till (n-1)th element is compared with nth element.
4. At end of the scan-1, the highest element is placed on the nth position.

Scan-2:

1. Take the first no and compare with the second no. if it is small, don’t interchange, otherwise interchange the elements.
2. Take the second no and compare with the third no. if it is small, don’t interchange, otherwise interchange the elements.
3. This process is continued till (n-2)th element is compared with (n-1)th element.
4. At end of the scan-2, the second highest element is placed on the (n-1)th position.

Scan-(n-1):

1. Take the first no and compare with the second no. if it is small, don’t interchange, otherwise interchange the elements.

PROGRAM:

```
void bubble(int a[10])
{
int i,j,temp;
for(i=0;i<=n-2;i++)
{
for(j=0;j<=n-2-i;j++)
{
if(a[j]>a[j+1])
{
temp=a[j];
a[j]=a[j+1];
a[j+1]=temp;
}
}
}
```

Eg.,

Consider the nos 43 72 10 23 1

Scan-0

A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**	**J**	**J+1**	**I**
43	72	10	23	1	0	1	0
43	72	10	23	1	1	2	0
43	10	72	23	1	2	3	0
43	10	23	72	1	3	4	0
43	10	23	1	72 □	4	5	0

Scan-1 :

A[0]	A[1]	A[2]	A[3]	A[4]	J	J+1	I
43	10	23	1	72	0	1	1
10	43	23	1	72	1	2	1
10	23	43	1	72	2	3	1
10	23	1	43 □	72	3	4	1

Scan-2 :

A[0]	A[1]	A[2]	A[3]	A[4]	J	J+1	I
10	23	1	43	72	0	1	2
10	23	1	43	72	1	2	2
10	1	23 □	43	72	2	3	2

Scan-3 :

A[0]	A[1]	A[2]	A[3]	A[4]	J	J+1	I
10	1	23	43	72	0	1	3
1	10 □	23	43	72	1	2	3

TIME AND SPACE COMPLEXITY OF BUBBLE SORTING:

The best case involves performing one pass which requires n-1 comparisons. Consequently, the best

case is O(n). The worst case performance of the bubble sort is $\frac{n(n-1)}{2}$ comparisons and $\frac{n(n-1)}{2}$ exchanges. the average case is more difficult to analyze than the other cases. It can be shown that the average case analysis is O(n^2). the average no. of passes is approximately $n - 1.25\sqrt{n}$. for n=10 the average number of casses is 6. The average no. of comparisons and exchanges are both O(n^2).

Analysis:

- The number of comparisons made

  $1 + 2 + 3 + . . . + (n - 1) = n(n - 1)/2 = O(n^2)$
- In this algorithm the number of comparison is irrespective of data set i.e., input whether best or worst.
- Time Complexity $T(n) = O(n^2)$
- It does not require extra memory.

14. Write in detail about searching or Explain about Sequential(Linear), Binary and Fibonacci search (UQ APRIL13 & Nov’10) (UQ: Apr/May'14) ((11)

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 or Linear Search

The simplest search technique is the linear or sequential search. In this technique, we start at a beginning of a list or table and search for the required record until the desired record is found or the list is exhausted. This technique is suitable for a table or a linked list or an array and it can be applied to an nordered list but the efficiency is increased if it is an ordered list.

For any search the total work is reflected by the number of comparisons of keys that makes in the earch.

The number of comparisons depends on where the target (value to be searched) key appears.

If the desired target key is at the first position of the list, only one comparison is required.

If the record is at second position, two comparisons are required. If it is the last position of the list, n comparisons are compulsory.

If the search is unsuccessful, it makes n comparisons as the target will be compared with all the entries of the list.

ALGORITHM

```
Alg linear search (a, n, x)
// a - array of integers

// n -number of elements
// x -search element

{

 for i=1 to n do

  {

       if(x==a[i])
       return i;

  }
```

```
        return 0;

}
```

For example:

Let us take the following example.

12 7 3 8 5

The target element is 8.

Comparisons 1:

A[0]	12	8 checks the target key with the array element A[0].
A[1]	7	
A[2]	3	
A[3]	8	
A[4]	5	

The two elements are not equal so the target checks with the next value.

Comparisons 2:

A[0]	12	
A[1]	7	8 checks the target key with the array element A[1].
A[2]	3	
A[3]	8	
A[4]	5	

The two elements are not equal so the target checks with the next value.

Comparisons 3:

A[0]	12	
A[1]	7	
A[2]	3	8 checks the target key with the array element A[2].
A[3]	8	
A[4]	5	

The two elements are not equal so the target checks with the next value.

Comparisons 4:

A[0]	12	
A[1]	7	
A[2]	3	
A[3]	8	8 checks the target key with the array element A[3].
A[4]	5	

The two elements are equal. So the checking is finished. The result is displayed.

ANALYSIS OF LINEAR /SEQUENTIAL SEARCH

Worst Case:

The worst case occurs when an item is not found or the search item is the last (nth) element. For both situations we must examine all n elements of the array.So the complexity of the sequential search is n. i.e., O(n). The execution time is proportional to n.i.e) the algorithm is exec uted in linear time.

Best Case:

The best case occurs when the search item is at the first location itself. So the time complexity is O(1).

Average Case:

The average case occurs if the search element is at the middle location. So we must examine n/2 elements of the array. So the time complexity is O(n/2).

ii.BINARY SEARCH

The main objective of binary search is to search a particular element which is present in the list of elements. The list of elements should be in a increasing or non-decreasing or sorted order.

For searching lists with more values linear search is insufficient, binary search helps in searching larger lists. To search a particular item the approximate middle entry of the table is located, and its key values examined.

1. If the search value is higher than the middle value, then the search is made with the elements after the middle element.

2. If the search value is smaller than the middle element, then the search is made with the elements before the middle value.

3. This process continues till the required target is found

ALGORITHM

```
Alg Binary search (a, n,x)

//   a - array of integers

//  n - number of elements

//   x - search element

{

        low=1;

         high=n;
        while(low<=high)

        { mid=(low+high)/2;
                if(a[mid]==x)
                return x;

                else if(x<a[mid])
                low=1;
                 high=mid-1;
                else if(x >a[mid])
                low=mid+1;
                 high=n;        }
                return 0;
}
```

EXAMPLE:

Let us apply the algorithm with an example. Suppose array A [ ] contains elements the following elements.

8 10 15 20 28 30 33 Let us search for the element 15.

Is low > high? NO

Mid= (0+6)/2 = 3.

8	10	15	20	28	30	33

[0] [1] [2] [3] [4] [5] [6]

Low Mid High

Is 15 = = A[3] ? No

15 < A[3], repeat the steps with low =0 and high = mid-1=2

Is low > high? NO

Mid=(0+2)/2 = 1.

8	10	15

[0] [1] [2]

Low Mid High

Is 15 = = A [1]? No

15 > A [1], repeat the steps with **low = mid +1 =2 and high = 2**

Is low > high ? NO

Mid=(2+2)/2 = 2.

8	10	15

[0] [1] [2]

Mid=high=2

Is 15 = = A[2] ? Yes

Return (2).

Element 15 is present in the the position 2.

Let us search for an element that is not in the list. Search Element=9

Is low > high? NO

Mid=(0+6)/2 = 3.

8	10	15	20	28	30	33
[0]	[1]	[2]	[3]	[4]	[5]	[6]

low **mid** **high**

Is 9 = = A[3] ? No

9 < A[3], repeat the steps with low =0 and high = mid-1=2

Is low > high ? NO

Mid=(0+2)/2 = 1.

8	10	15
[0]	[1]	[2]

Low **Mid** **High**

Is 9 = = A[1] ? No

9 < A[1], repeat the steps with low = 0 and high = mid -1 = 0

Is low > high ? NO

Mid=(0+0)/2 = 0.

8
[0]

Low **= mid = high =0**

Is 9 = = A[0]? No

Repeat the steps with low = mid + 1=1 and high = 0

Is low > high? Yes

When low value becomes greater than high value algorithm returns that the element 9 is not present n the list.

TIME COMPLEXITY:	**BEST**	**WORST**	**AVERAGE**
	O (1)	**O(log n)**	**O(log n)**

.iii)Fibonacci search

The **Fibonacci search technique** is a method of searching a sorted array using a divide and conquer algorithm that narrows down possible locations with the aid of Fibonacci numbers. This method is faster than raditional binary search algorithms, with a complexity of *O*(log(*x*)).

ALGORITHM:

```
Step 1. Start
Step 2. Read the value of n
Step 3. for i=1 to n increment in steps of 1
            Read the value of ith element into array
Step 4. Read the element(x) to be searched
Step 5. search<--fibonacci(a,n,p,q,r,key)
Step 6. if search equal to 0 goto step 7 otherwise goto step 8
Step 7.print unsuccessful search
Step 8. print successful search
Step 9. stop
```

PROGRAM

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int n,a[50],i,key,loc,p,q,r,m,fk;
   clrscr();
   printf("\nenter the value of n");
   scanf("%d",&n);
   printf("enter the number");
   for(i=1;i<=n;i++)
    scanf("%d",&a[i]);
   printf("enter the number to be searched");
   scanf("%d",&key);
   fk=fib(n+1);
   p=fib(fk);
   q=fib(p);
   r=fib(q) ;
   m=(n+1)-(p+q);
   if(key>a[p])
    p=p+m;
   loc=rfibsearch(a,n,p,q,r,key);
   if(loc==0)
    printf("key is not found");
   else
    printf("The number is found in %d position",loc);
   getch();
}
int fib(int m)
{
   int a,b,c;
   a=0;
   b=1;
   c=a+b;
   while(c<m)
   {
    a=b;
    b=c;
    c=a+b;
   }
   return b;
}
int rfibsearch(int a[],int n,int p,int q,int r,int key)
{
   int t;
   if(p<1||p>n)
    return 0;
   else if(key==a[p])
    return p;
   else if(key<a[p])
  {
    if(r==0)
     return 0;
    else
    {
     p=p-r;
     t=q;
     q=r;
     r=t-r;
     return rfibsearch(a,n,p,q,r,key);
    }
   }
   else
   {
    if(q==1)
      return 0;
    else
    {
     p=p+r;
     q=q-r;
     r=r-q;
     return rfibsearch(a,n,p,q,r,key);
    }
   }
}
```

<u>OUTPUT:</u>

```
Enter the value of n : 6
Enter the number :
20
15
28
31
56
18
Enter the number to be searched : 15
The number is found in 2 position

Enter the number to be searched: 12
The number is not found
```

WORST CASE PERFORMANCE

Beginning with an array containing F*j* – 1 elements, the length of the subset is boundedto F*j*-1-1 elements. To earch through a range with a length of F*n*-1 at the beginning we have to make *n* comparisons in the worst ase. F*n* = (1/sqrt(5))*((1+sqrt(5))/2)*n*, that’s approximately c*1,618*n* (with a constant c). For N+1 = *1,618*n* elements we need *n* comparisons, *i.e.*, the maximum number of comparisons is O (ld N).

Pondicherry University Questions

2 Marks

1. What is an algorithm? (UQ April 2012 & APRIL 2013)
2. What is best-case efficiency? (UQ APRIL 2013)
3. What is binary search?(UQ April‟12)
4. Discuss about various asymptotic notation(UQ Nov‟12)
5. What is recursive algorithm?(UQ Nov‟12,Apr/May‟14)
6. Define sorting (UQ Nov‟12 ,Apr/May‟14)
7. Define search (UQ: NOV‟14)
8. What is performance measurement?

11 Marks

1. Explain about analysis of algorithms & analysis of control structures. (UQ APRIL‟13)
2. Write about solving recurrences or Discuss about recursive and non recursive algorithms.(UQ APRIL12) (UQ:Apr/May'14)
3. Write and analyze about heap sort (UQ Nov‟12)
4. Write in detail about radix sort (UQ APRIL12)
5. Write and analyze selection sort(UQ APRIL13 & Nov‟12)
6. Write and analyze about bubble sort (UQ APRIL 13,NOV‟14))
7. Write in detail about searching or Explain about Sequential and Fibonacci search (UQ APRIL13 & Nov‟10) (UQ: Apr/May'14)

UNIT II

Divide and Conquer Method: General Method –binary search –maximum and minimum –merge sort - quick sort –Strassen‟s Matrix multiplication–knapsack. problem –Greedyminimum spanning tree algorithms – single source shortest path algorithm –scheduling, optimal storage on tapes, optimal merge patterns.

2 Marks

1.Define the divide and conquer method. (UQ APRIL'13 & APRIL'12)

Given a function to compute-and-conquer strategyon suggests„n‟ splitting inputs the inputs including k‟ sub problems. The sub problems mu 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.

2. Define control abstraction.

A control abstraction we mean a procedure whose flow of control is clear but whose primary operations are by other procedures whose precise meanings are left undefined.

3. Write the Control abstraction for Divide -and conquer.

```
Algorithm DandC (k)
{
If small(p) then return(s) else
{
Apply D and C to each of these subproblems
Return combine (DAnd C(_1) DAnd C(_2),----, DAnd ((_k));
}
}
```

4. What is the substitution method?

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

5. What is the binary search? (UQ APRIL'12)

If „q‟aysischosenalw such that „aq‟ is the middl search algorithm is known as binary search.

6. Give computing time for Binary search?

In conclusion we are now able completely describe the computing time of binar y search by giving formulas that describe the best, average and worst cases.

O(1) O(logn) O(Logn) best average worst
Unsuccessful searches

O(logn)

best, average, worst

7. What is the maximum and minimum problem?

The problem is to find the maximum and minim may look so simple as to be contrived, it allows us to demonstrate divide and- conquer in simple setting.

8. What is the Quick sort? (UQ Nov'10& APRIL'12)

The n numbers in quick sort, the division into sub arrays is made so that the sorted sub arrays do not need to be merged later.

9. Write the Analysis for the Quick sot.

In analyzing QUICKSORT, we can only make the number of element comparisons c(n). It is easy to see that the frequency count of other operations is of the same order as C(n).

10. Is insertion sort better than the me rge sort? (UQ APRIL'13)

Insertion sort works exceedingly fast on ar computing time is O(n2).

11. Write a algorithm for straightforward maximum and minimum>

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

12. Give the recurrence relation of divide -and-conquer?

The recurrence relation is T(n) = g(n)

T(n1) + T(n2) + ----+ T(nk) + f(n)

13. Write the algorithm for Iterative binary search?

```
Algorithm BinSearch(a,n,x)
//Given an array a[1:n] of elements in nondecreasing // order, n>0, determine whether x is present
{
low : = 1; high : = n;
while (low < high) do
{
mid : = [(low+high)/2]; then high:= mid-1; else return mid;
}
return 0;
}
```

14. What is meant by feasible solution? (UQ:NOV'14)

Given n inputs and we are required to form a subset such that it satisfies some given constraints then such a subset is called feasible solution.

15. Write any two characte ristics of Greedy Algorithm?

* To solve a problem in an optimal way construct the solution from given set of candidates.

* As the algorithm proceeds, two other sets get accumulated among this one set contains the candidates that have been already considered and chosen while the other set contains the candidates that have been considered but rejected.

16. Define optimal solution?

A feasible solution either maximizes or minimizes the given objective function is called as optimal solution

17. What is Knapsack problem? (UQ Nov'12)

A bag or sack is given capacity n and n objects are given. Each object has weight wi and profit pi .Fraction of object is considered as xi (i.e) 0<=xi<=1 .If fraction is 1 then entire object is put into sack. When we place this fraction into the sack we get wixi and pixi.

18. What is the Greedy choice property?

* The first component is greedy choice property (i.e.) a globally optimal solution can arrive at by making a locally optimal choice.
* The choice made by greedy algorithm depends on choices made so far but it cannot depend on any future choices or on solution to the sub problem.
* It progresses in top down fashion.

19. What is greedy method? (UQ:Apr/May'14)

Greedy method is the most important design technique, which makes a choice that looks best at that moment. A given „n" inputs are required us to o
solution. A greedy method suggests that one can device an algorithm that works in stages considering one input at a time.

20. What are the steps required to develop a greedy algorithm?

* Determine the optimal substructure of the problem.
* Develop a recursive solution.
* Prove that at any stage of recursion one of the optimal choices is greedy choice.
Thus it is always safe to make greedy choice.
* Show that all but one of the sub problems induced by having made the greedy choice are empty.
* Develop a recursive algorithm and convert into iterative algorithm.

21. What is activity selection problem?

The „n" task will be given with starting tim
should not overlap and optimal solution is that the task should be completed in minimum number of machine set.

22. What does Job sequencing with deadlines mean?

* Given a set of „n" jobs each job „i" has a de
* For job „i" profit pi is earned iff it is com
* Processing the job on the machine is for 1unit of time. Only one machine is available.

23. When is a task said to be late in a schedule?

A task is said to be late in any schedule if it finishes after its deadline else a task is early in a schedule.

24. Write the general algorithm for Greedy method control abstraction.

```
Algorithm Greedy (a, n)
{
Solution = 0;
For i=1 to n do
{
X=select(a);
If fesible(solution,x) then
Solution=union(Solution,x);
}
Return solution;
}
```

25. Define optimal solution for Job sequencing with deadlines.

Feasible solution with maximum profit is optimal solution for Job sequencing with deadlines.

26. Define minimum spanning tree (UQ Nov’12)

Let G(V,E) be an undirected connected„E".-Agraphsubgrapht=(V the G is a Spanning tree of G iff „t‟ is a tree subset of E,G‟ is a Minimum spanning tree.

27. Write about optimal merge pattern proble m.

Two files x1 and x2 containing m & n records could be merged together to obtain one merged file. When more than 2 files are to be merged together. The merge can be accomplished by repeatedly merging the files in pairs. An optimal merge pattern tells which pair of files should be merged at each step. An optimal sequence is a least cost sequence.

28. Name the best sorting method .Give its time c

Quick sort is the best sorting method.The n numbers in quick sort, the division into sub arrays is made so that the sorted sub arrays do not need to be merged later. The time complexity of Quick sort is

Worst case performance :$O(n^2)$
Best case performance: O(n log n)
Average case performance : O(n log n)

<u>11 Marks</u>

1. **Explain divide and conquer general method**

General method:

- Given a function to compute-and-conquerstrategyon suggests„n‟splittinginputsthe th into „k‟ distinct subsets, 1<k<=n, yielding
- 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 t
- 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
- Combine is a function that determines the so
- If the sizetheof sizes„p‟isofn theand „k‟ sub problem computing time of D And C is described by the recurrence relation.

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

Where T(n) → is the time for D And Con any I/p o g(n) → is the time of compute the answer directly for small I/ps.

f(n) → is the time for dividing P & combining the solution to sub problems.

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

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

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

→ Where a & b are known constants.

→ We assume that T(1) is known & „n‟ is a powe

- One of the methods for solving any such recurrence relation is called the substitution method.
- This method repeatedly makes substitution for each occurrence of the function. T is the Right-hand side until all such occurrences disappear.

Example:

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

$$\begin{aligned} T(n) &= 2T(n/2)+n \\ &= 2[2T(n/2/2)+n/2]+n \\ &= [4T(n/4)+n]+n \\ &= 4T(n/4)+2n \\ &= 4[2T(n/4/2)+n/4]+2n \\ &= 4[2T(n/8)+n/4]+2n \\ &= 8T(n/8)+n+2n \\ &= 8T(n/8)+3n \\ &\quad * \\ &\quad * \\ &\quad * \end{aligned}$$

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

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

→ Corresponding to the choice of i=log n

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

= n. T(n/n) + n log n

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

= 2n + n log n

2. Write algorithm and explain binary search (UQ APRIL'13& APRIL'12)

The main objective of binary search is to search a particular element which is present in the list of elements. The list of elements should be in a increasing or non-decreasing or sorted order.

For searching lists with more values linear search is insufficient, binary search helps in searching larger lists. To search a particular item the approximate middle entry of the table is located, and its key values examined.

1. If the search value is higher than the middle value, then the search is made with the elements after the middle element.

2. If the search value is smaller than the middle element, then the search is made with the elements before the middle value.

3. This process continues till the required target is found

ALGORITHM

```
Alg Binary search (a, n,x)
//   a - array of integers
//  n - number of elements
//   x - search element
{
          low=1;
           high=n;
          while(low<=high)
          { mid=(low+high)/2; if(a[mid]==x)
                    return x;

                    else if(x<a[mid]) low=1;
                    high=mid-1; else if(x
                    >a[mid]) low=mid+1;

                     high=n;        }

return 0;
}
```

EXAMPLE:

Let us apply the algorithm with an example. Suppose array A [ ] contains elements the following elements.

8 10 15 20 28 30 33 Let us search for the element 15.

Is low > high? NO

Mid= (0+6)/2 = 3.

8	10	15	20	28	30	33

[0] [1] [2] [3] [4] [5] [6]

Low Mid High

Is 15 = = A[3] ? No

15 < A[3], repeat the steps with low =0 and high = mid-1=2

Is low > high? NO

Mid=(0+2)/2 = 1.

8	10	15
[0]	[1]	[2]
Low	**Mid**	**High**

Is 15 = = A [1]? No

15 > A [1], repeat the steps with **low = mid +1 =2 and high = 2**

Is low > high ? NO

Mid=(2+2)/2 = 2.

8	10	15
[0]	**[1]**	**[2]**

Mid=high=2

Is 15 = = A[2] ? Yes

Return (2).

Element 15 is present in the the position 2.

Let us search for an element that is not in the list. Search Element=9

Is low > high? NO

Mid=(0+6)/2 = 3.

8	10	15	20	28	30	33
[0]	[1]	[2]	[3]	[4]	[5]	[6]
low			**mid**			**high**

Is 9 = = A[3] ? No

9 < A[3], repeat the steps with low =0 and high = mid-1=2

Is low > high ? NO

Mid=(0+2)/2 = 1.

8	10	15
[0]	[1]	[2]
Low	**Mid**	**High**

Is 9 = = A[1] ? No

9 < A[1], repeat the steps with low = 0 and high = mid -1 = 0

Is low > high ? NO

Mid=(0+0)/2 = 0.

8
[0]

Low **= mid = high =0**

Is 9 = = A[0]? No

Repeat the steps with low = mid + 1=1 and high = 0

Is low > high? Yes

When low value becomes greater than high value algorithm returns that the element 9 is not present n the list.

TIME COMPLEXITY:	BEST	WORST	AVERAGE
	O (1)	**O(log n)**	**O(log n)**

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‟ th with x and a[low], a[low+1],……..,a[mid],……a[
- 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

3. Explain Maximum and Minimum (UQ APRIL'13,NOV'14)

- 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];
         }
  }
```

Algorithm: Straight forward Maximum & Minimum

- Straight MaxMin requires 2(n-1) element comparison in the best, average & worst cases.
- An immediate improvement is possible by realizing that the comparison a[I]<min is necessary only when a[I]>max is false.

- Hence we can replace the contents of the for loop by,
    If(a[I]>max) then max:=a[I];
    Else if (a[I]<min) then min:=a[I];

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

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

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

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

  → Let P=(n, a[I] ,……,a[j]) denote an arbitra → Here „n‟ is the no. of elements in the list (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 [n/2],a[[n/2]+1],…..,a[n])

- After having divided „P‟ into 2 smallerivelyinvoking thesub 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];
{
        if (a[I]<a[j]) then
        {
            max:=a[j];
            min:=a[I];
        }
        else
        {
            max:=a[I];
            min:=a[j];
        }
    }
    else
    {
        // if P is not small, divide P into subproblems.
        // find where to split the set mid:=[(I+j)/2];
        //solve the subproblems
        MaxMin(I,mid,max.min);
        MaxMin(mid+1,j,max1,min1);
        //combine the solution
        if (max<max1) then max=max1;
        if(min>min1) then min = min1;
    }
}
```

- The procedure is initially invoked by the statement, MaxMin(1,n,x,y)
- Suppose we simulate MaxMin on the following 9 elements

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

- A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made.
- For this Algorithm, each node has 4 items of information: I, j, max & imin.
- Examining fig: we see that the root node contains 1 & 9 as the values of I &j corresponding to the initial call to MaxMin.
- This execution produces 2 new calls to MaxMin, where I & j have the values 1, 5 & 6, 9 respectively & thus split the set into 2 subsets of approximately the same size.
- From the tree, we can immediately see the maximum depth of recursion is 4. (including the 1st call)
- The include no.s in the upper left corner of each node represent the order in which max & min are assigned values.

No. of element Comparison:

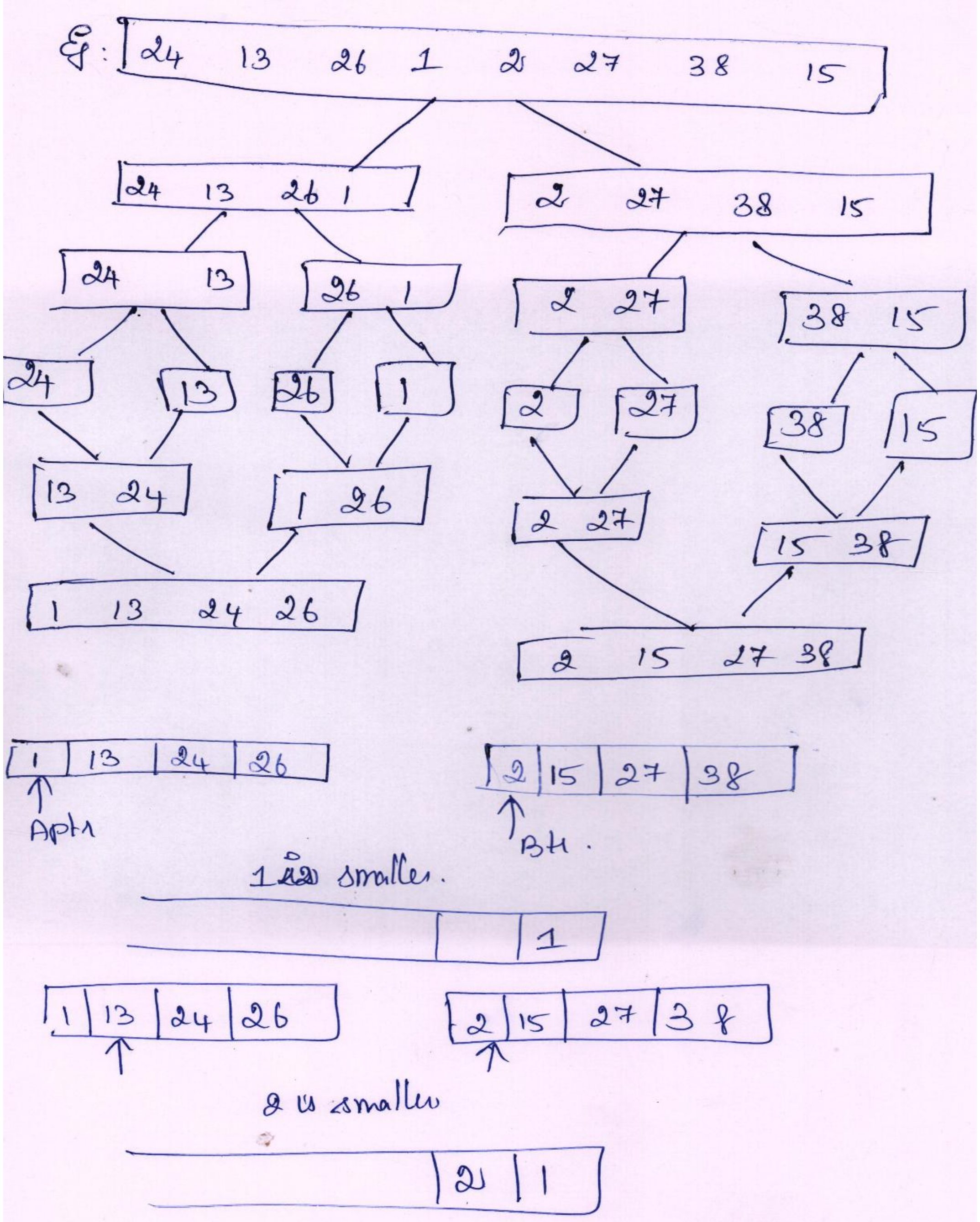
- If T(n) represents this no., then the resulting recurrence relations is

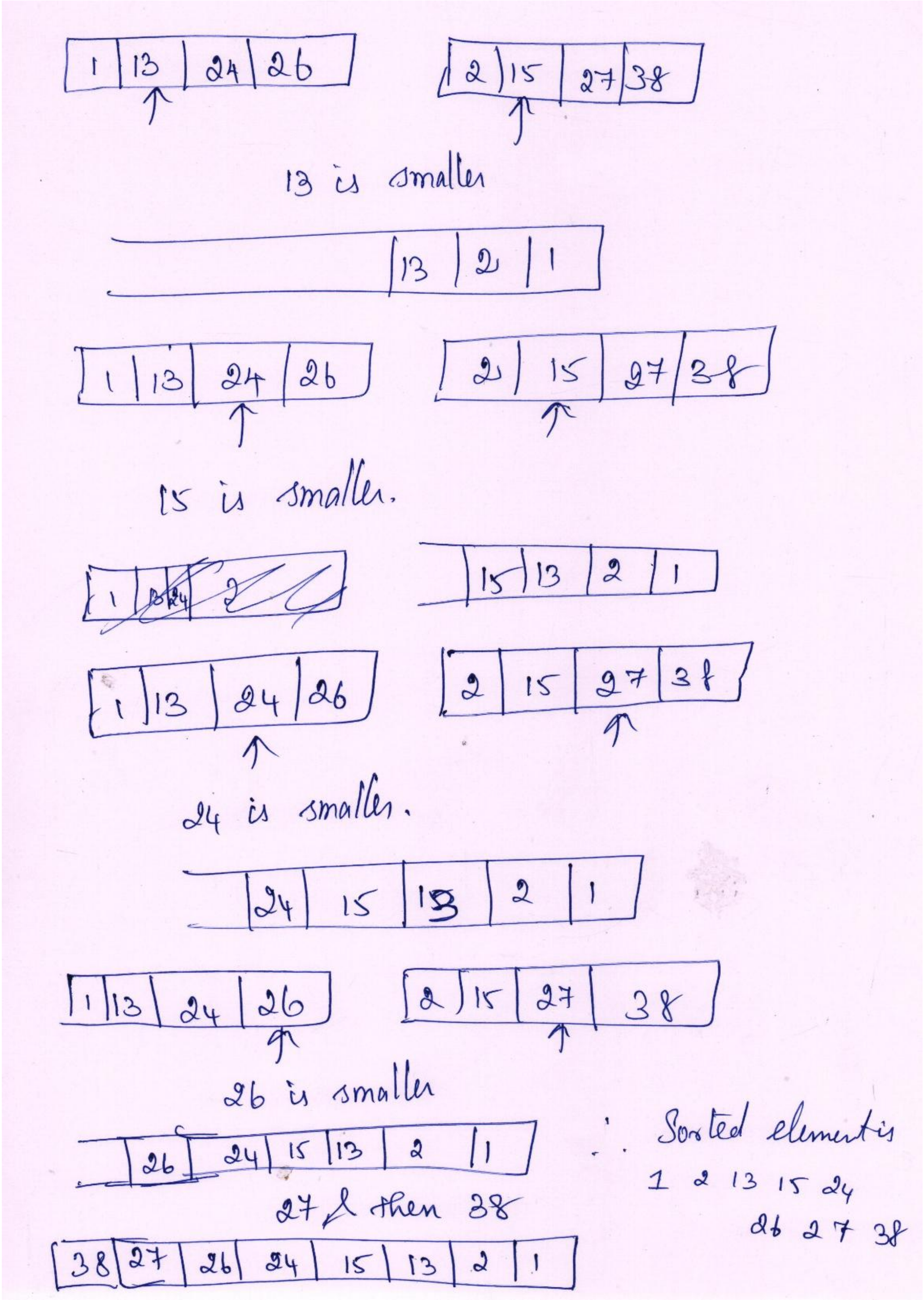
$$T(n)=\begin{cases} T([n/2]+T[n/2]+2 & n>2 \\ 1 & n=2 \\ 0 & n=1 \end{cases}$$

4. Explain merge sort (UQ APRIL'12,APRIL/MAY'14)

- 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
  - □ It uses divide and conquer rule for its operation.
  - □ Merging is a process of combining 2 sorted sub tables and produce a single sorted table.
  - □ This can be achieved by finding the record with the smallest key occurring either of the table and lace

into a new table.





IMPLEMENTATION:

1. Merge 2 sorted sub tables into a single table and store into a temporary array TEMP

2. Copy the content of TEMP array into original array K.

PROGRAM:

```
void merge(int a[30],int low,int high)
{
int mid;
if(low<high)
{
mid=(low+high)/2;
merge(a,low,mid);
merge(a,mid+1,high);
combine(a,low,mid,high);
}
}
int combine(int a[30],int low,int mid,int high)
{
int k=low,i=low,j=mid+1;
int temp[30];
while((i<=mid) && (j<=high))
{
if(a[i]<=a[j])
{
temp[k]=a[i];
i++;
k++;
}
else
 {
temp[k]=a[j];
j++;
k++;
}
}
while(i<=mid)
{
temp[k]=a[i];
i++;
k++;
}
while(j<=high)
{
temp[k]=a[j];
j++;
k++;
}
for(i=low;i<=high;i++)
a[i]=temp[i];
}
```

TIME AND SPACE COMPLEXITY OF MERGE SORT:

Worst Case : O(nlog n) , Best Case : O(n log n), Average case: O(n log n)

Advantages:

- Better cache performance
- It is stable sort
- Simple to understand than heap sort

Limitations:

- Require extra memory space

- If the time for the merging operations is proportional t described by the recurrence relation.
  T(n)={an=1,‟

→

When „n‟ is a power of 2, n= 2^k, we can solve t

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,

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

5. Explain quick sort (UQ:APRIL/MAY’14)

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

Quick sort is otherwise called as partition exchange sorting.

□ It uses divide and conquer method.

□ Let A be the array of ‘n’ nos.

□ Our aim is to sort the nos by ascending order.

IMPLEMENTATION:

1.The value of I is incremenr=ted till a[i]<=pivot and the value of j is decremented till a[j]>pivot, this process is repeated until i<j

2.If a[i]>pivot and a[j]<pivot, and also if i<j then swap a[i] and a[j]

3.If i>j then swap a[j] and a[pivot]

`

Eg 40 20 70 14 60 61 97 30

Pivot i j

Compare i with pivot => True continue

=> False => Compare j with pivot

↓ True continue

↓ false → swap & Compare i with pivot.

① compare i with pivot.

40 >= 20 => true.

40 >= 70 => false.

40 20 70 14 60 61 97 30

i j

Compare j with pivot

40 < 30 => false so swap.

40 20 30 14 60 61 97 70

i j

Compare i with pivot

40 >= 30

40 >= 14 => 40 20 30 14 60 61 97 70

i j

40 >= 60 => false

Compare j with pivot.

~~90~~ 40 < 97 => true 40 20 30 14 60 61 97 70

i j

40 < 61 => true 40 20 30 14 60 61 97 70

i j

40 < 60 => true 40 20 30 14 60 61 97 70

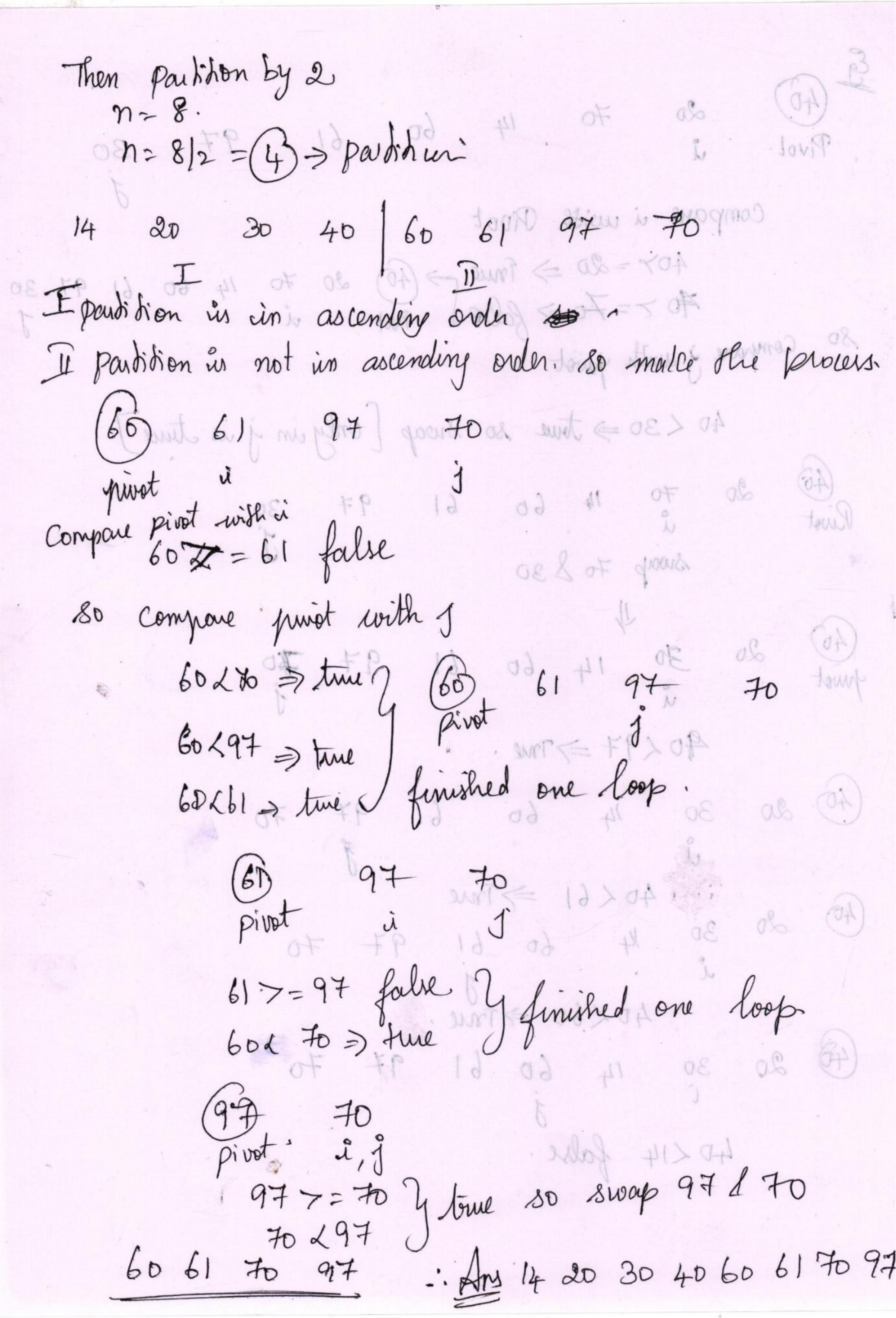
i j

40 < 14 => false. (i, j, nearer.)

so swap pivot & i

Swap 40 & 14

~~40~~ 14 20 30 40 60 61 97 70



Then partition by 2

n = 8.

n = 8/2 = (4) ⇒ partition

14 20 30 40 | 60 61 97 70

I II

I partition is in ascending order

II partition is not in ascending order. so make the process.

(60) 61 97 70

pivot i j

Compare pivot with i

60 ~~>~~ = 61 false

so compare pivot with j

60 < 70 ⇒ true

60 < 97 ⇒ true

60 < 61 ⇒ true } finished one loop.

(60) 61 97 70

pivot j

(61) 97 70

pivot i j

61 >= 97 false

60 < 70 ⇒ true } finished one loop

(97) 70

pivot i, j

97 >= 70

70 < 97 } true so swap 97 & 70

60 61 70 97

∴ Ans 14 20 30 40 60 61 70 97

PROGRAM

```
void quick(int a[10],int low,int high)
{
int m,i;
if(low<high)
{
m=partition(a,low,high);
quick(a,low,m-1);
quick(a,m+1,high);
}
}
int partition(int a[10],int low,int high)
{
int pivot=a[low],i=low,j=high;
while(i<=j)
{
while(a[i]<=pivot)
i++;
while(a[j]>pivot)
{
j--;
}
if(i<=j)
swap(a,&i,&j);
}
swap(a,&low,&j);
return j;
}
void swap(int a[10],int*i,int*j)
{
int temp;
temp=a[*i];
a[*i]=a[*j];
a[*j]=temp;
}
```

TIME AND SPACE COMPLEXITY OF QUICK SORTING:

Worst Case : O(n^2) , Best Case : O(n log n), Average case: O(nlog n)

Advantages:

- o Faster than other algorithms.
- o Better cache performance & high speed.

Limitations

□ Requires more memory space

6. **Explainstrasson'smultiplicationmatrix(UQ Nov'12&APRIL'12)**

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

C (i ,j)= A(i,k) B(k,j) for all „i‟ and

- The time complexity for the matrix Multiplication is O(n^3).
- Divide and conquer method suggest another way to compute the product of n*n matrix.
- We assume that N is a power of 2 .In the case N is not a power of 2 ,then enough rows and columns of zero can be added to both A and B .SO that the resulting dimension are the powers of two.
- If n=2 then the following formula as a computed using a matrix multiplication operation for the elements of A & B.
- If n>2,Then the elements are partitioned in product can be recursively computed using the same formula .This Algorithm will continue applying itself to smaller sub matrix until „N" beco directly .
- The formula are

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

C11=A11B11+A12B21
C12=A11B12+A12B22
C21=A21B11+A22B21
C22=A21B12+A22B22

For EX:

$$4 * 4 = \begin{matrix} 2\ 2\ 2\ 2 \\ 2\ 2\ 2\ 2 \\ 2\ 2\ 2\ 2 \\ 2\ 2\ 2\ 2 \end{matrix} * \begin{matrix} 1\ 1\ 1\ 1 \\ 1\ 1\ 1\ 1 \\ 1\ 1\ 1\ 1 \\ 1\ 1\ 1\ 1 \end{matrix}$$

The Divide and conquer method

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

- To compute AB using the equation we need to perform 8 multiplication of n/2*n/2 matrix and from 4 addition of n/2*n/2 matrix.
- Ci,j are computed using the formula in equation----- $\xrightarrow{}$ 4
- As can be sum P, q, R, S, T, U, and V can be computed using 7 Matrix multiplication and 10 addition or subtraction.
- The Cij are required addition 8 addition or subtraction.

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

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

Example

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

P=(4*4)+(4+4)=64
Q=(4+4)4=32
R=4(4-4)=0
S=4(4-4)=0
T=(4+4)4=32
U=(4-4)(4+4)=0

V=(4-4)(4+4)=0
C11=(64+0-32+0)=32
C12=0+32=32
C21=32+0=32
C22=64+0-32+0=32

So the answer c(i,j) is $\begin{vmatrix} 32 & 32 \\ 32 & 32 \end{vmatrix}$

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

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

□

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

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

- Stressen has discovered a way to compute the Cij of equation (2) using only 7 multiplication and 18 addition or subtraction.
- Strassen‟s formula are

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

C11=P+S-T+V
C12=R+t
C21=Q+T
C22=P+R-Q+V

7. Explain 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. {
```

```
solution =0;//Initialise the solution.
For i=1 to n do
{
    x=select(a);
    if(feasible(solution,x))then
        solution=union(solution,x);
}
return solution;
}
```

* The function select 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 algor ithm will once a particular problem is chosen ends the function subset, feasible & union are properly implemented.

Example

- Suppose we have in a country the following coins are available :

Dollars(100 cents)
Quarters(25 cents)
Dimes(10 cents)
Nickel(5 Cents)
Pennies(1 cent)

- Our aim is paying a given amount to a customer using the smallest possible number of coins.
- For example if we must pay 276 cents possible solution then,

→ 1 doll+7 q+ 1 pen → 9 coins

→ 2 doll +3Q +1 pen → 6 coins

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

8. Explain knapsack problem (UQ:NOV'14)

- 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 solut ion 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
8.{
9.if (w[i]>u)then break;
10.x[i]=1.0;U=U-w[i] 11.}
12.}
```

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/Wi	1.6	1.5	1.36
Pi	= 24	15	25
Wi	= 15	10	18
Xi =	1	0.5	0

PiXi=1*24+0.5*15 → 31.5

The optimal solution is → 31.5

9. Explain job scheduling with dead lines

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

Points To re member:

- To complete a job, one has to process the job or a action for one unit of time.
- Only one machine is available for processing jobs.
- A feasible solution for this problem is a subset of j of jobs such that each job in this subject can be completed by this deadline.
- If we select a job at that time ,

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

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

ALGORITHM:

```
Algorithm JS(d,j,n)
//The job are ordered such that p[1]>p[2]…>p[n]
//j[i]  is  the ith job in the optimal solution
// Also at terminal d [ J[ i]<=d[ J {i+1],1<i<k
 {
  d[0]= J[0]=0;
J[1]=1;
K=1;
For I =1 to n do
if (d[J[r]]<d[I]) then
{
for q=k to (r+1) step –1 do J temp=I;
i=k;
else
exit;
}
return k;
}
```

Example :

1. n=5 (P1,P2,P5)=(20,15,10,5,1)…
   (d1,d2….d3)=(2,2,1,3,3)

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

………etc to find all solutions
The Solution 13 is optimal

2. n=4 (P1,P2,…P4)=(100,10,15,27) (d1,d2….2,1)d4)=(2,1,

Feasible solution	***Processing Sequence***	***Value***
(1,2)	(2,1)	110
(1,3)	(1,3)	115
(1,4)	(4,1)	127

(2,3)	(9,3)	25
(2,4)	(4,2)	37
(3,4)	(4,3)	42
(1)	(1)	100
(2)	(2)	10
(3)	(3)	15
(4)	(4)	27

The solution 3 is optimal.

10. Explain minimum spanning tree (UQ Nov'12,April/May'14)

SPANNING TREE:

A spanning tree of a connected graph is its connected a cyclic subgraph that contains all the vertices of graph. A minimum spanning tree of a weighted connected graph G is its spanning tree of smallest weight, where the weight of a tree is defined as the sum of weights on all its edges.

There are two Algorithms in spanning tree

a. Prims Algorithm
b. kruskal Algorithm

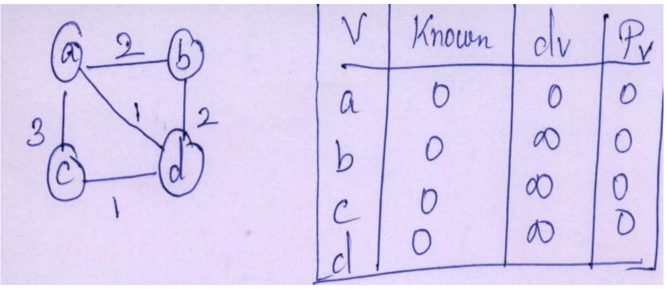
PRIMS ALGORITHM:Explain about Prims method for creating a spanning tree

Prims Algorithm is one of the way to compute a minimum spanning tree which uses a greedy technique.This algorithm begins with a set U initialized to { 1 }. It then grows a spanning tree, one edge at a time. At each step it finds a shortest edge (u,v) such that cost of(u,v) is the smallest among all the edges, whwre u is in minimum spanning tree and V is not in Minimum Spanning Tree.

ALGOTITHM:

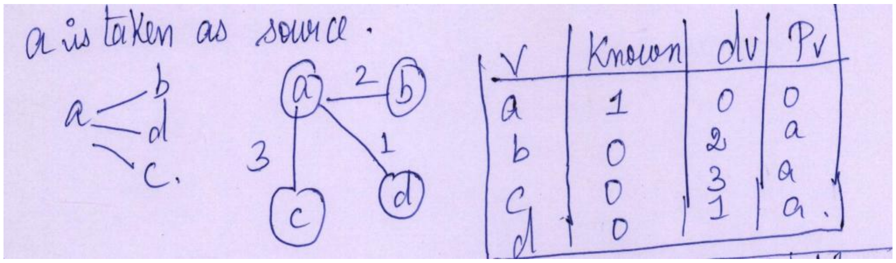
```
void Prims(Table T)
{
vertex V,W;
/* Table initialization */
for(i=0;i<NumVertex;i++)
{
T[i].known=False;
T[i].dist=Infinity;
T[i].path=0;
}
for(; ;)
{
Let V be the start vertex with smallest distance
T[V].dist=0;
T[v].known=Treu;
for each W adjacent to V
if(!T[w].known)
{
T[w].dist=Min(T[W].dist,C_VW);
T[W].path=V;
}
}
```

EXAMPLE:



V	Known	dv	Pv
a	0	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

Here a is taken as source vertex.
Then the distance of its adjacent vertex is as follows:T[b].dist=Min(T[b].dist,C_{ab})
= Min(∞,2)
=2
T[d].dist=Min(T[d].dist,C_{ad})
= Min(∞,1)
=1
T[c].dist=Min(T[c].dist,C_{ac})
= Min(∞,3)
=3

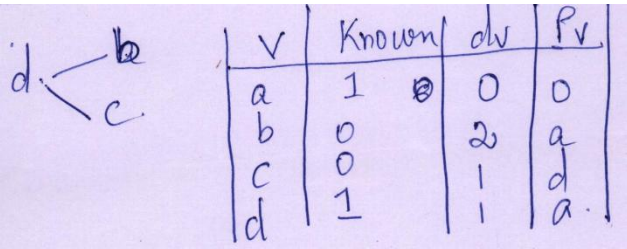


a is taken as source.

a
b
d
c.
a
b
d
c
2
1
3

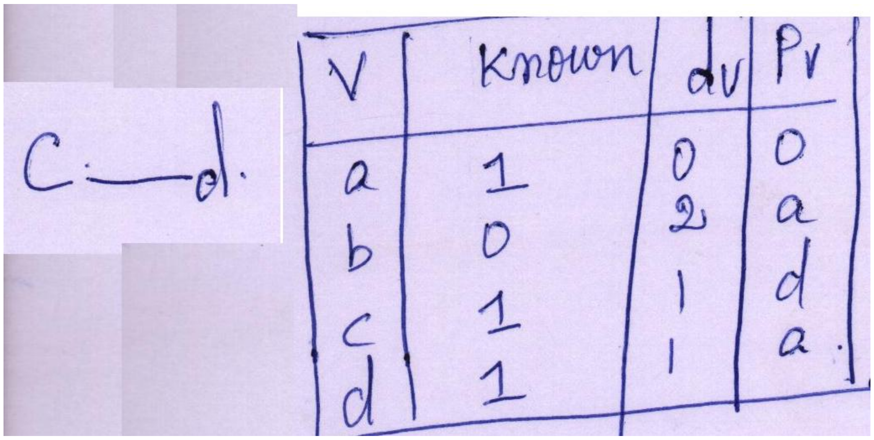
V	Known	dv	Pv
a	1	0	0
b	0	2	a
c	0	3	a
d	0	1	a.

Next vertex d with its minimum distance is marked as visited and the distance of its unknown adjacent vertex is updated.
T[b].dist=Min(T[b].dist,C_{db})
= Min(2,2)
=2
T[c].dist=Min(T[c].dist,C_{dc})
= Min(3,1)
=1



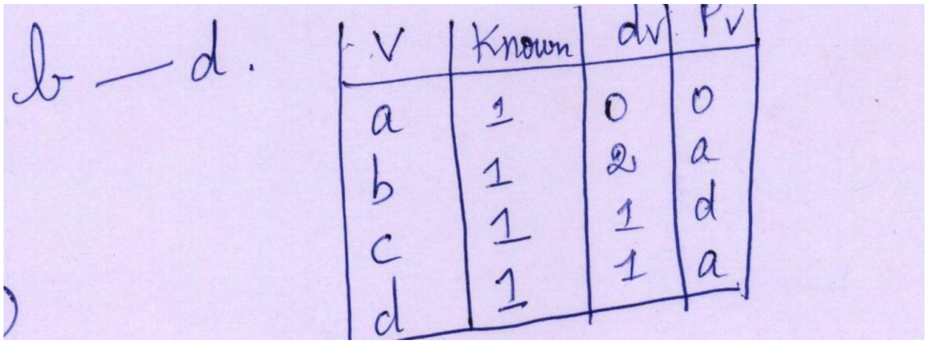
V	Known	dv	Pv
a	1	0	0
b	0	2	a
c	0	1	d
d	1	1	a.

Next vertex c with its minimum distance is marked as visited and the distance of its unknown adjacent vertex is updated.



V	Known	dv	Pv
a	1	0	0
b	0	2	a
c	1	1	d
d	1	1	a.

Finally vertex b which is not visited is marked



b — d.

V	Known	dv	Pv
a	1	0	0
b	1	2	a
c	1	1	d
d	1	1	a

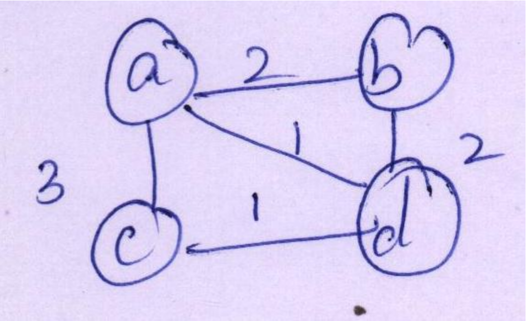
KRUSKAL ALGORITHM: Explain about kruskal's method for creating a spanning tree

Kruskal algorithm ueses a greedy technique to compute a minimum spanning tree. This algorithm selects the edges in the order of smallest weight and accept an edge if it does not cause a cycle.The algorithm terminates if enough edges are accepted.

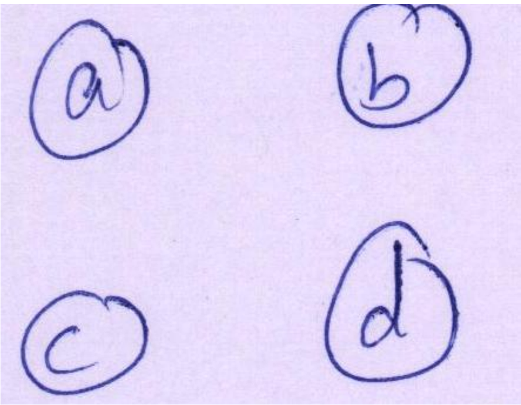
ALGORITHM

```
Voidd Kruskal(graph G)
{
int EdgesAccepted=0;
while(EdgesAccepted<Numvertx -1)
{
USet=Find(U,S);
VSet=Find(V,S);
if(USet!=VSet)
{
EdgesAccepted++;
SetUnion(S,Uset,VSet);
}
}
}
```

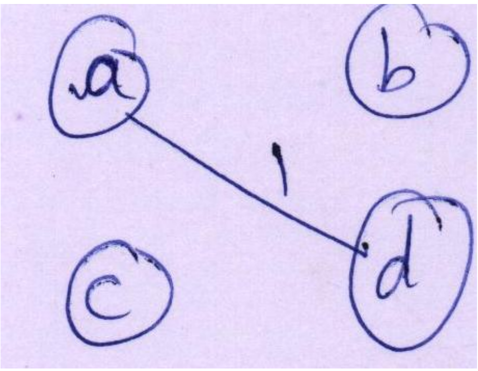
EXAMPLE:



Step 1: Initially each vertex is in its own set

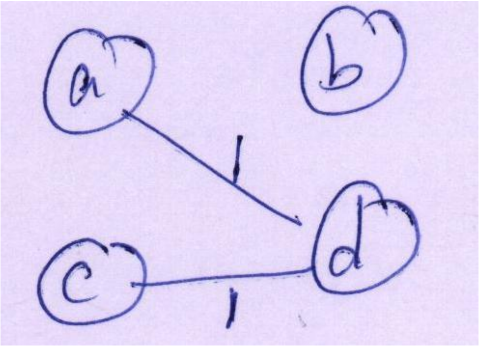


Step 2: Now select the edge with minimum weight



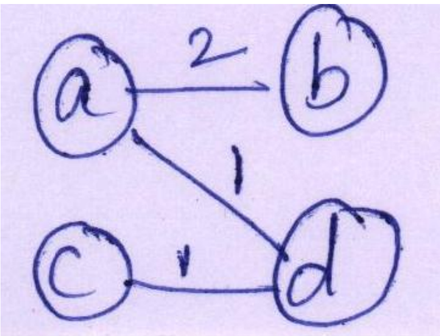
Edge(a,b) is added to minimum spanning tree

Step 3: Select the next edge , with minimum weight and check whether it forms a cycle. If it forms a cycle then that edge is rejected. else add the edge to minimum spanning tree.

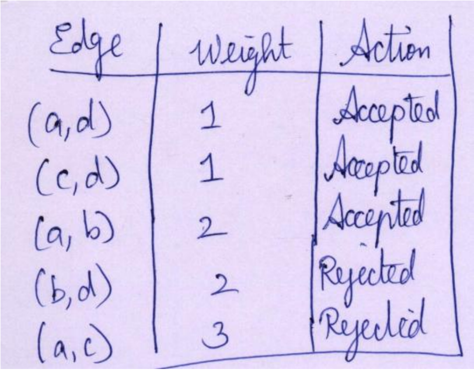


(c,d) is next minimum weighted edge.

Step 4: Repeat stpe 3 until all vertices included in minimum spanning tree.



The minimum cost of spanning tree is 4 (2 + 1 + 1 = 4)



Edge	Weight	Action
(a,d)	1	Accepted
(c,d)	1	Accepted
(a,b)	2	Accepted
(b,d)	2	Rejected
(a,c)	3	Rejected

Application of the spanning tree:

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

11. Explain Single-source shortest path

The shortest path algorithm determines the minimum cost of the path from source to every other vertex. Two types of shortest path problem, exist namely,

1. The single source shortest path problems.
2. The all pair shortest path problem.

Two methods of Shortest path algorithm:

1. unweighted Shortest path
2. Dijiktra's Algorithm

UNWEIGHTED SHORTEST PATH (Explain Unweighted shortest path Algorithm)

In unweighted shortest path all the edges are assigned a weight of " 1 " for each vertex. The following three information is maintained.

Known:

Specifies whether the vertex is processed or not. It is set to 1 after it is processed, otherwise 0. Initially all vertices are marked as 0

dv:

Specifies the distance from souce s , initiallyall vertices are unreachable except for s, whose path length is 0.

pv:

Specifies the bookkeeping variable which will allow us to print. The actual path (i.e) The vertex which makes the changes in dv.

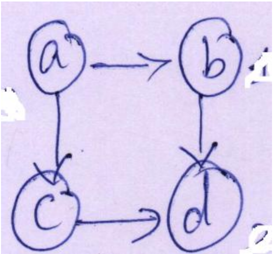
To implement the unweighted shortest path, perform the following steps:

1. **Assign the source node as s and Enqueue s.**
2. **Dequeue the vertex s from queue and assign the value of that vertex to be known and the find its adjacency vertices.**
3. **If the distance of the adjacency vertices is equal to infinity the change the distance of that vertex as the distance of its source vertex incremented by 1 and enqueue the vertex**
4. **Repeat from step 2 until the become empty**

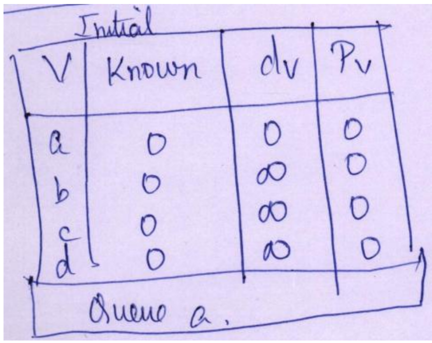
ALGORITHM:

```
void Unweighted(table T)
{
Queue Q;
vertex V,W;
Q=CreateQueue(NumVertex)
MakeEmpty(Q);
Enqueue(s,Q);
{
V=Dequeue(Q);
T[V].known=true;
for each adjacent to V
if(T[W].dist=T[V].dist+1;
                T[W].path=V;
                Enqueue(W,Q);
}
}
                DisposeQueue(Q);
}
```

EXAMPLE:



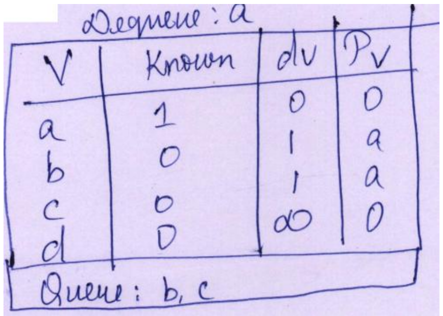
Source Vertex a is initially assigned a path length 0



Initial

V	Known	dv	Pv
a	0	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0
Queue a.			

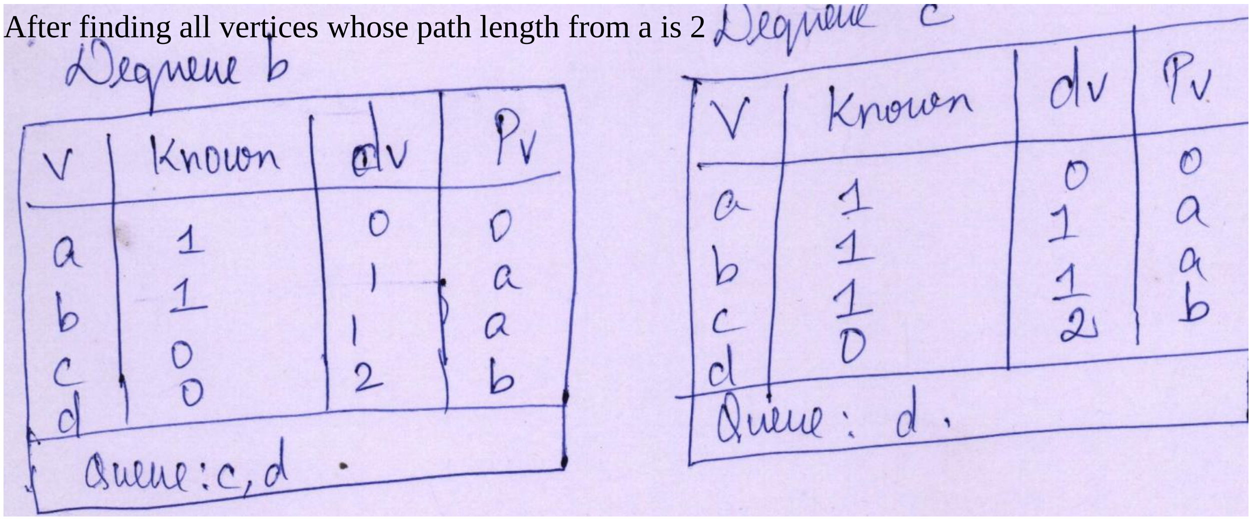
After finding all vertices whose path length from a is 1



Dequeue : a

V	Known	dv	Pv
a	1	0	0
b	0	1	a
c	0	1	a
d	0	∞	0
Queue : b, c			

After finding all vertices whose path length from a is 2

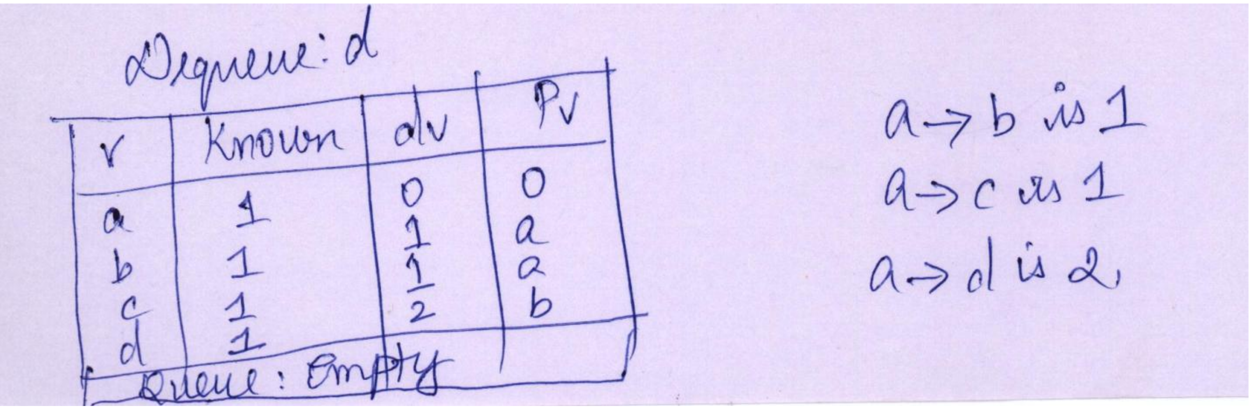


Dequeue b

V	Known	dv	Pv
a	1	0	0
b	1	1	a
c	0	1	a
d	0	2	b
Queue:c, d .			

Dequeue c

V	Known	dv	Pv
a	1	0	0
b	1	1	a
c	1	1	a
d	0	2	b
Queue : d .			



Dequeue: d

V	Known	dv	Pv
a	1	0	0
b	1	1	a
c	1	1	a
d	1	2	b
Queue : Empty			

a→b is 1

a→c is 1

a→d is 2,

DIJIKSTRA’S ALGORITHM: Explain about Dijikstra's algorithm: (Apr 12, Nov 12, Nov 13)

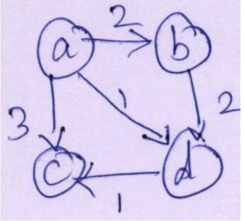
The general method to solve the single source shortest path problem is known as Dijikstra's algorithm. This is applied to weighted graph

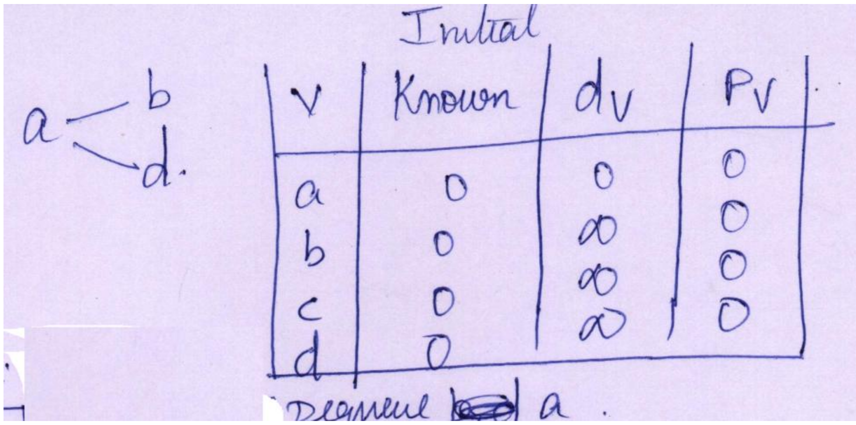
This algorithm is the prime example of Greedy technique, which generally solve a problem in stages by doing what appears to be the best thing in each stage. This algorithm proceeds in stages, just like the unweighted shortest path algorithm. At each stage, it selects a vertex V, which has the smallest dv among all the unknown vertices, and declares that as the shortest path from S to V and mark it to be known. We should set dw = dv + C_{vw}, if the new value for dw would be an improvement.

ALGORITHM:

```
void Dijikstra( Graph G, Table T)
{
    vertex V,W;
    for(i=0;iNNumvertex;i++)
    {
        T[i].known=false;
        T[i].dist=Infinity;
        T[i].path = Not aVetrex;
    }
    T[start].dist=0;
    for( ; ; )
    {
        V = smallest unknown distance vertes;
        if( V = = Not A vertex)
            break;
        T[V].known=true;
        for each W adjacent to V
            if(!T[W].known)
            {
                T[W].dist = Min(T[W].dist,T[V].dist+$C_{vw}$)
                T[W].path=V;
            }
    }
}
```

EXAMPLE:



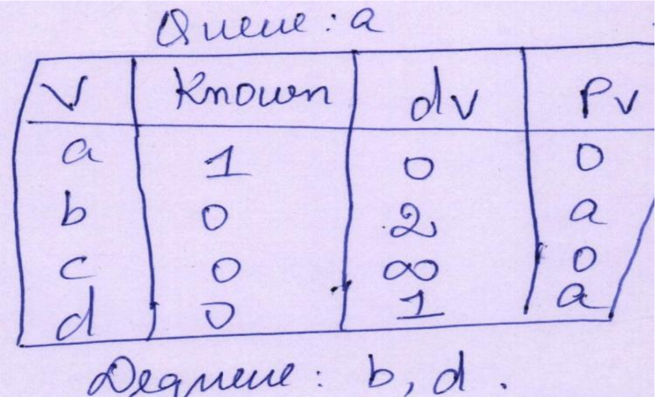


Initial

V	Known	dv	Pv
a	0	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

Dequeue a .

Vertex a is choosen as source and is declared as known vertex.

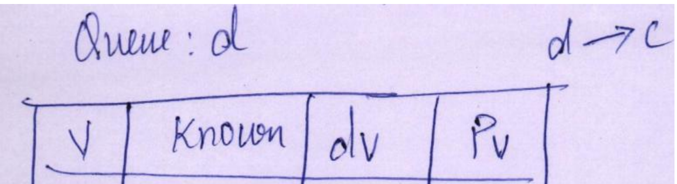


Queue : a

V	Known	dv	Pv
a	1	0	0
b	0	2	a
c	0	∞	0
d	0	1	a

Dequeue : b, d .

Here d is the next minimum distance vertex . The adjacent vertex to d is c, therefore the distance of c is updated as follows:

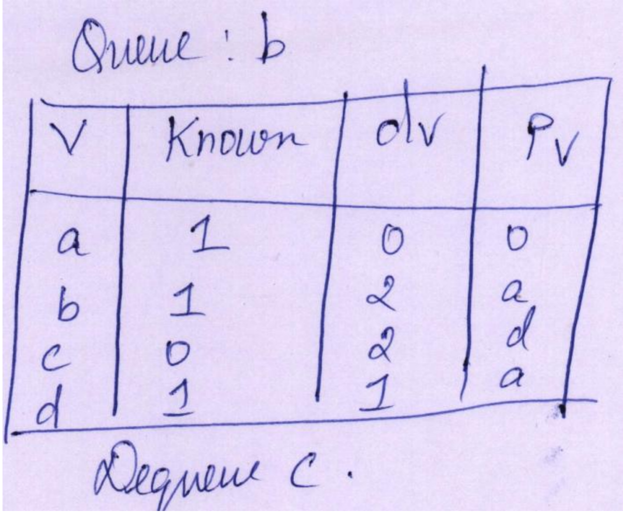


Queue : d

d → c

V	Known	dv	Pv

Next vertex is b.

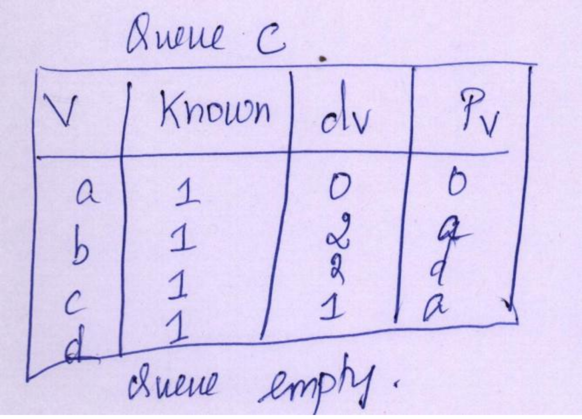


Queue : b

V	Known	dv	Pv
a	1	0	0
b	1	2	a
c	0	2	d
d	1	1	a

Dequeue c .

Since the the adjacent vertex d is already visited , select the next minimum vertex c and marked it as visited.



Queue c .

V	Known	dv	Pv
a	1	0	0
b	1	2	a
c	1	2	d
d	1	1	a

Queue empty.

12. Explain optimal storage on tapes (UQ APRIL'12)

- We have n no of programs. The length of the
- The objective of the problem is to store 'n' no of programs on a tape with the constraint: Sum of the length of all the programs should not exceed the length of the tape l.(i.e)All the programs can be stored on the tape iff the sum of the length of the program is atmost l.
- Assume whenever a program is retrieved from the tape, tape is initially positioned at the front.
- Time needed (tj)to retrieve the program lj i
- Program P1 having length l1 takes t1 time to store it.
- Program P2 having length l2 takes t2 time to store it.
- Program P3 having length l3 takes t3 time to store it.
- We have to store l1,l2,l3 in l;
- If all the programs are equally retrieved, then
  Mean retrieval time $d = (1/n) \sum t_j \ ,1 \le j \le n$
  $d = (t_1+t_2+.....+t_n)/n$
- Our aim is to minimize the mean retrieval time. To minimize MRT we have to find the permutation for n programs.(i.e) We must consider the order in which programs are saved.

Example:

P1---l1---> 5

P2---l2--->10
P3---l3---> 3
Possible orderings=n!=3!=6

I	d(i)	MRT
1, 2 , 3	5+15+!8	38
1, 3 , 2	5+8+18	31
2, 3 , 1	10+13+18	41
2, 1 , 3	10+15+18	43
3, 1 , 2	3+8+18	29(min)
3, 2 ,1	3+13+18	34

The 5th order 3,2,1 takes minimum MRT 29.So that if the programs are stored in the sequence 3,2,1 then it is an optimal solution.

Analysis:

The greedy method simply requires to store the programs in non decreasing order of their lengths. This ordering can be carried out in O(n log n) time using an efficient sorting algorithm(heap sort).

13. Explain optical merge patterns

- We have 2 sorted files containing n and m records. The file f1 contains n records. The file f2 contains m records.
- These 2 files could be merged together and resultant file is in also sorted order.
- Time taken for merging these two files is O(m+n).
- If we have more than 2 files,then we can merge the files in some order.
- Let us take 4 files say x1,x2,x3&x4. We can merge these 4 files in any order.
- For example:
  1.Merge x1 & x2 => y1
  Merge y1 & x3 => y2
  Merge y2 & x4 => y3
  2.Merge x1 & x2 => y1
  Merge x3 & x4 => y2
  Merge y1 & y2 => y3
- Given n sorted files. We have to sort the n sorted files & store it in a single file in sorted order.
- We can merge the files in any order, but it should take minimum amount of computing time.

Example 1:

n=3
File x1 length=30
File x2 length=20
File x3 length=10

Way 1:

Merge x1 & x2	- 50 record moves
Merge(x1 & x2)&x3	- 60 record moves
Total no of record moves	-110 record moves

Way 2:

Merge x2 & x3	- 30 record moves
Merge(x2 & x3)&x1	- 60 record moves
Total no of record moves	- 90 record moves

Minimum record moves is in way2(90).So the second merge pattern is faster than first.

By greedy method, optimal merge pattern suggests:At each step me rge the 2 smallest files together.

Example 2 :

x1	x2	x3	x4	x5
20	30	10	5	30

Merge x3+x4 → 10+5=15=>z1 Merge z1+x1 → 15+20=35=>z2 Merge x2+x5 → 30+30=60=>z3 Merge z2+z3 → 35+60=95

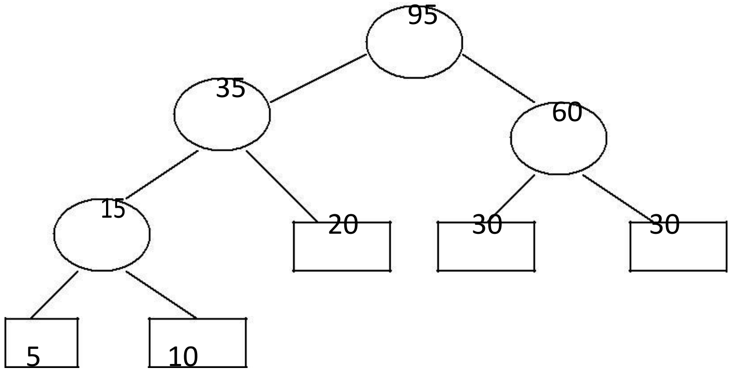
Total no. of record moves = 205moves

2 Way Merge Pattern:

1.It is represented by binary merge trees. 2.Leaf nodes are drawn as squares. 3.Nodes are called as external nodes.

4.Remaining nodes are drawn as circles.(ie) internal nodes

5. Each internal node has exactly 2 children. Internal node represents the file obtained by its 2 children.



In the above tree, square represents a single sorted file and circle represents the merging of 2 files.

Pondicherry University Questions

2 Marks

1. Define the divide and conquer method. (UQ APRIL‟13APRIL‟12)&
2. What is the binary search? (UQ APRIL‟12)
3. What is the Quick sort? (UQ Nov‟10&APRIL‟12)
4. Is insertion sort better than the merge sort? (UQ APRIL‟13)
5. What is Knapsack problem? (UQ Nov‟12)
6. Define minimum spanning tree (UQ Nov‟12)
7. What is greedy method? (UQ:Apr/May'14)
8. What is meant by feasible(Ref.Pg.No.2solution? (UQ:NOV‟14
9. Name the best sorting method .Give its time complexity (UQ:NOV‟14

11 Marks

1. Write algorithm and explain binary search (UQ APRIL‟13 & APRIL‟12)
2. Explain Maximum and Minimum (UQ APRIL‟13,NOV‟14)
3. Explain merge sort (UQ APRIL‟12,APRIL/MAY‟14)
4. Explain strasson‟s matrix
5. Explain minimum spanning,Apr/May‟14)
6. Explain optimal storage on tapes (UQ APRIL‟12)
7. Explain quick sort (UQ:APRIL/MAY‟14)
8. ExplaiN knapsacK problem(UQ:NOV‟14)

.

UNIT III

Dynamic Programming: General method – multi- stage graphs – all pair shortest path algorithm – 0/1 Knapsack and Traveling salesman problem – chained matrix multiplication. Basic Search and Traversal technique: Techniques for binary trees and graphs – AND/OR graphs – biconnected components – topological sorting.

2 Marks

1. Write the difference between the Greedy method and Dynamic programming. Greedy method Dynamic programming

1. Only one sequence of decision is generated.
1. Many number of decisions are generated.
2. It does not guarantee to give an optimal solution always.
2. It definitely gives an optimal solution always.

2. Define dynamic programming. (UQ APRIL'12)

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

3. What are the features of dynamic programming?

Optimal solutions to sub problems are retained so as to avoid re-computing their values. Decision sequences containing subsequences that are sub optimal are not considered. It definitely gives the optimal solution always.

4. What are the drawbacks of dynamic programming?

Time and space requirements are high, since storage is needed for all level. Optimality should be checked at all levels.

5. Write the general procedure of dynamic programming. (UQ APRIL'13)

The development of dynamic programming algorithm can be broken into a sequence of 4 steps.

1. Characterize the structure of an optimal solution.
2. Recursively defines the value of the optimal solution.
3. Compute the value of an optimal solution in the bottom- up fashion.
4. Construct an optimal solution from the computed information.

6. Define principle of optimality. (UQ:NOV'14)

It states that an optimal sequence of decisions has the property that whenever the initial stage or decisions must constitute an optimal sequence with regard to stage resulting from the first decision.

7. Give an example of dynamic programming and explain.

An example of dynamic programming is knapsack problem. The solution to the knapsack problem can be viewed as a result of sequence of decisions. We have to decide the value of xi for 1<i<n. First we make a decision on x1 and then on x2 and so on. An optimal sequence of decisions maximizes the object function

8. Explain any one method of finding the shortest path.

One way of finding a shortest path from vertex i to j in a directed graph G is to decide which vertex should be the second, which is the third, which is the fourth, and so on, until vertex j is reached. An optimal sequence of decisions is one that results in a path of least length.

9. Define 0/1 knapsack proble m.

The solution to the knapsack problem can be viewed as a result of sequence of decisions. We have to decide the value of xi. xi is restricted to have the value 0 or 1 and by using the function knap(l, j, y) we can represent the problem as maximum pi xi subject to wi xi < y where l - iteration, j - number of objects, y – capacity.

10. What is the formula to calculate optimal solution in 0/1 knapsack proble m?

The formula to calculate optimal solution is g0(m)=max{g1, g1(m-w1)+p1}.

11. Write about traveling salesperson proble m.

Let g = (V, E) be a directed. The tour of G is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The traveling salesperson problem to find a tour of minimum cost.

12. Write some applications of traveling salesperson problem.

Routing a postal van to pick up mail from boxes located at n different sites.

- Using a robot arm to tighten the nuts on some piece of machinery on an assembly line. Production environment in which several commodities are manufactured on the same set of machines.

13. Give the time complexity and space complexity of traveling salesperson proble m.

Time complexity is O (n2 2n).
Space complexity is O (n 2n).

14. List different traversal technique in graph (UQ APRIL'12)

1. DFS and
2. BFS.

15. What is biconnected component? (UQ Nov'12)

Graph with no articulation point then it is called as biconnected component

16. What is articulation point? (UQ:NOV'14)

When a node in a graph is deleted, the graph divided into two or more graph then the node is called articulation point.

17. Write about multistage graph with example (UQ APRIL'13 & Nov'12,APRIL/MAY'14)

A multistage graph G = (V,E) is a directed graph in which the vertices are portioned into K > = 2 disjoint sets Vi, 1 <= i<= k. In addition, if <u,v> is an edge in E, then u < = Vi and V ∑ Vi+1 for some i, 1<= i< k.

18. What is all pair shortest path problem?(APRIL/MAY'14)

Let G(V,E) be a directed graph with n vertices , „E" is the set of edges& V is the set of n vertices. Each edge has an associated non-negative length. Calculate the length of the shortest path between each pair of nodes i.eShortest path between every vertex to all other vertices. 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.

The principle of optimality: If k is the node on the shortest path from i to j then the part of the path from i to k and the part from k to j must also be optimal, that is shorter.

11 Marks

1. **Explain dynamic program general method**

- It is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.
- 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 t hat 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 cannot be optimal and so will not be generated.
- Because of principle of optimality, decision sequences containing subsequences that are suboptimal are not considered. Although the total number of different decision sequences is exponential in the number of decisions(if there are d choices for each of the n decisions to be made then there are d^n possible decision sequences),Dynamic programming algorithms often have a polynomial complexity.

2. Explain multistage graph with example (UQ :APRIL/MAY'14)

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

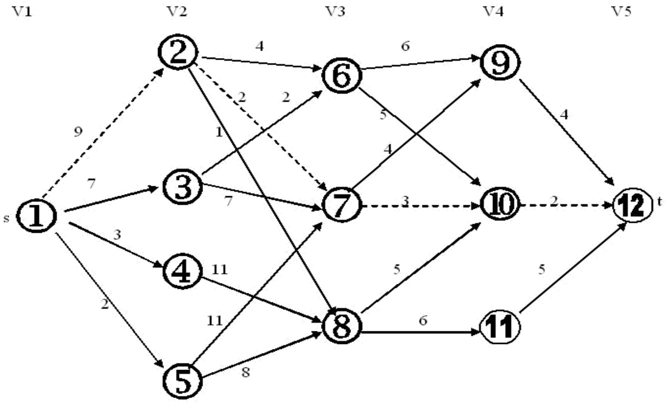
Assume that there are „k‟ stages in a graph. In this *FORWARD* approach, we will find out the cost of each and every node starting from the „k‟ th stage to the 1 st stage. We will find out the path (i.e.) minimum cost path from source to the destination (i.e.) [Stage-1 to Stage-k].

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 neighbor 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,

ALGORITHM:

```
Algorithm FGraph (G,k,n,p)
// Thei/p is a k-stage graph G=(V,E) with „n‟ vertex indexed in order of stages.
// E is a set of edges.
// c[i,j] is the cost of <i,j>
// p[1:k] is a minimum cost path.
{
    cost[n]=0.0;
    for j=n-1 to 1 step-1 do
   {
         //compute cost[j],
         Let „r" be the vertex such that <j,r> is an edge of „G"
        & c[j,r]+cost[r] is minimum.
        cost[j] = c[j+r] + cost[r];
        d[j] =r;
    }
    // find a minimum cost path.
    p[1]=1;
    p[k]=n;
    for j=2 to k-1 do
          p[j]=d[p[j-1]];
}
```



TRACE OUT:

n=12k=5

cost (12)=0 d (12)=0

for j = 11 to 1

When j=11

cost (11)=5 d (11)=12

When j=10

cost (10)=2 d (10)=12

When j=9

cost (9)=4 d (9)=12

When j=8

Note:

If there is more than one path from vertex j to next stage vertices,then we find the closest vertex by the following formula:

- For forward approach,

$$\boxed{\text{Cost }(i,j) = \min_{\substack{l \in V_{i+1} \\ (j,l) \in E}} \{C(j,l) + \text{Cost}(i+1,l)\}}$$

cost(8) = min {C (8,10) + Cost (10), C (8,11) + Cost (11) }
= min (5 + 2, 6 + 5)
= min (7,11)
= 7

cost(8) =7 d(8)=10

When j=7

cost(7) = min(c (7,9)+ cost(9),c (7,10)+ cost(10))
= min(4+4,3+2)
= min(8,5)
= 5

cost(7) = 5 d(7) = 10

When j=6
cost(6) = min (c (6,9) + cost(9),c (6,10) +cost(10))
= min(6+4 , 5 +2)
= min(10,7)
= 7
cost(6) = 7 d(6) = 10

When j=5
cost(5) = min (c (5,7) + cost(7),c (5,8) +cost(8))
= min(11+5 , 8 +7)
= min(16,15)
= 15
cost(5) = 15 d(5) = 18

When j=4
cost(4) = min (c (4,8) + cost(8))
= min(11+7)
= 18
cost(4) = 18 d(4) = 8

When j=3
cost(3) = min (c (3,6) + cost(6),c (3,7) +cost(7))
= min(2+7 , 7 +5)
= min(9,12)
= 9
cost(3) = 9 d(3) = 6

When j=2
cost(2) = min (c (2,6) + cost(6),c (2,7) +cost(7) ,c (2,8) +cost(8))
= min(4+7 , 2+5 , 1+7)
= min(11,7,8)
= 7
cost(2) = 7 d(2) = 7

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

= min(9+7 , 7 +9 , 3+18 , 2+15)
= min(16,16,21,17)
= 16
cost(1) = 16 d(1) = 2

p[1]=1 p[5]=12
for j=2 to 4
When j=2 p[2]=d[p[1]]=d[1]=2
When j=3 p[3]=d[p[2]]=d[2]=7
When j=4 p[4]=d[p[3]]=d[7]=10

The path through which you have to find the shortest distance from vertex 1 to vertex 12.

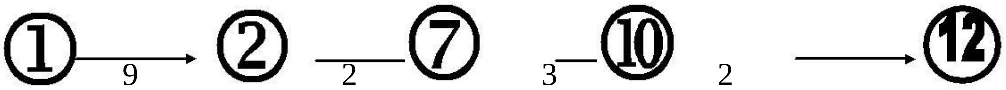
(ie) → ① → ② → ⑦ → ⑩ → ⑫

p[1] p[2] p[3] p[4] p[5]

Start from vertex - 1

D (1) = 2
D (2) = 7
D (7) = 10
D (10) = 12

So, the minimum –cost path is,



∴ The cost is 9+2+3+2+=16

ANALYSIS:

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

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 1^{st} stage to the k^{th} stage.
- We will find out the minimum cost path from destination to source (i.e.)[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.

ALGORITHM :

Algorithm BGraph (G,k,n,p)

```
// The i/p is a k-stage graph G=(V,E) with „n‟ vertex indexed in order of stages
// E is a set of edges.
// c[i,j] is the cost of<i,j>
// p[1:k] is a minimum cost path.
{
    bcost[1]=0.0;
     for j=2 to n do
    {
            //compute bcost[j],
           Let „r‟ be the vertex such that <r,j> is an edge of „G‟ &bcost[r]+c[r,j]
           is minimum.
           bcost[j] = bcost[r] + c[r,j];
          d[j] =r;
   }
  // find a minimum cost path.
  p[1]=1;
  p[k]=n;
```

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

TRACE OUT:

```
        n=12k=5
                bcost (1)=0                     d (1)=0
        for j = 2 to 12
        When j=2
                bcost(2) = 9                   d(2)=1
        When j=3


                bcost(3) = 7                   d(3)=1
        When j=4
                bcost(4) = 3                   d(4)=1
        When j=5
                bcost(5) = 2                   d(5)=1
        When j=6
                bcost(6) =min(c (2,6) + bcost(2),c (3,6) + bcost(3))
                        =min(13,9)
                bcost(6) = 9                     d(6)=3
        When j=7
                bcost(7) =min(c (3,7) + bcost(3),c (5,7) + bcost(5) ,c (2,7) + bcost(2))
                        =min(14,13,11)
                bcost(7) = 11                    d(7)=2

        When j=8
                bcost(8) =min(c (2,8) + bcost(2),c (4,8) + bcost(4) ,c (5,8) +bcost(5))
                        =min(10,14,10)
                bcost(8) = 10                    d(8)=2
        When j=9
                bcost(9) =min(c (6,9) + bcost(6),c (7,9) + bcost(7))
                        =min(15,15)
                bcost(9) = 15                    d(9)=6
        When j=10
                bcost(10)=min(c(6,10)+bcost(6),c(7,10)+bcost(7)),c(8,10)+bcost(8))
                         =min(14,14,15)
                bcost(10)= 14                    d(10)=6
        When j=11
                bcost(11) =c (8,11) + bcost(8))
                bcost(11) = 16                           d(11)=8
        When j=12
                bcost(12)=min(c(9,12)+bcost(9),c(10,12)+bcost(10),c(11,12)+bcost(11))
                         =min(19,16,21)
                bcost(12) = 16                           d(12)=10


        p[1]=1          p[5]=12
        for j=4 to 2
                When j=4        p[4]=d[p[5]]=d[12]=10
                When j=3        p[3]=d[p[4]]=d[10]=6
                When j=2        p[2]=d[p[3]]=d[6]=3
```

The path through which you have to find the shortest distance from vertex 1 to vertex 12.
p[5]=12p[4]= =10 p[3]= =6 p[2]= =3 p[1]=1

So the minimum cost path is,

$$1 \xrightarrow{7} 3 \xrightarrow{2} 6 \xrightarrow{5} 10 \xrightarrow{2} 12$$

The cost is 16.

3. Explain all pair shortest path (UQ:NOV'14)

The shortest path algorithm determines the minimum cost of the path from source to every other vertex. Two types of shortest path problem, exist namely,

1. The single source shortest path problems.
2. The all pair shortest path problem.

Two methods of Shortest path algorithm:

1. unweighted Shortest path
2. Dijiktra's Algorithm

UNWEIGHTED SHORTEST PATH (Explain Unweighted shortest path Algorithm)

In unweighted shortest path all the edges are assigned a weight of " 1 " for each vertex. The following three information is maintained.

Known:

Specifies whether the vertex is processed or not. It is set to 1 after it is processed, otherwise 0. Initially all vertices are marked as 0

dv:

Specifies the distance from souces ,initiallyall vertices are unreachable except for s, whose path length is 0.

pv:

Specifies the bookkeeping variable which will allow us to print. The actual path (i.e) The vertex which makes the changes in dv.

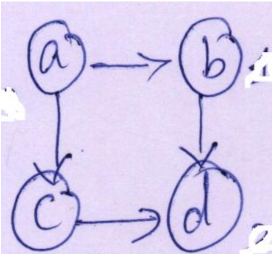
To implement the unweighted shortest path, perform the following steps:

1. **Assign the source node as s and Enqueue s.**
2. **Dequeue the vertex s from queue and assign the value of that vertex to be known and the find its adjacency vertices.**
3. **If the distance of the adjacency vertices is equal to infinity the change the distance of that vertex as the distance of its source vertex incremented by 1 and enqueue the vertex**
4. **Repeat from step 2 until the become empty**

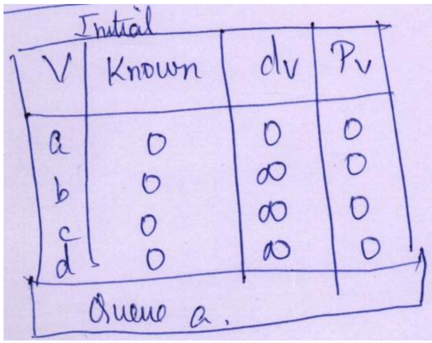
ALGORITHM:

```
void Unweighted(table T)
{
Queue Q;
vertex V,W;
Q=CreateQueue(NumVertex)
MakeEmpty(Q);
Enqueue(s,Q);
{
V=Dequeue(Q);
T[V].known=true;
for each adjacent to V
if(T[W].dist=T[V].dist+1;
T[W].path=V;
Enqueue(W,Q);
}
}
DisposeQueue(Q);
}
```

EXAMPLE:



Source Vertex a is initially assigned a path length 0

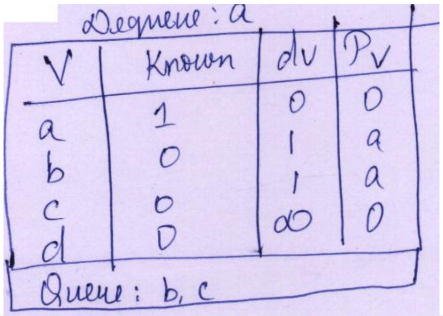


Initial

V	Known	dv	Pv
a	0	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

Queue a.

After finding all vertices whose path length from a is 1

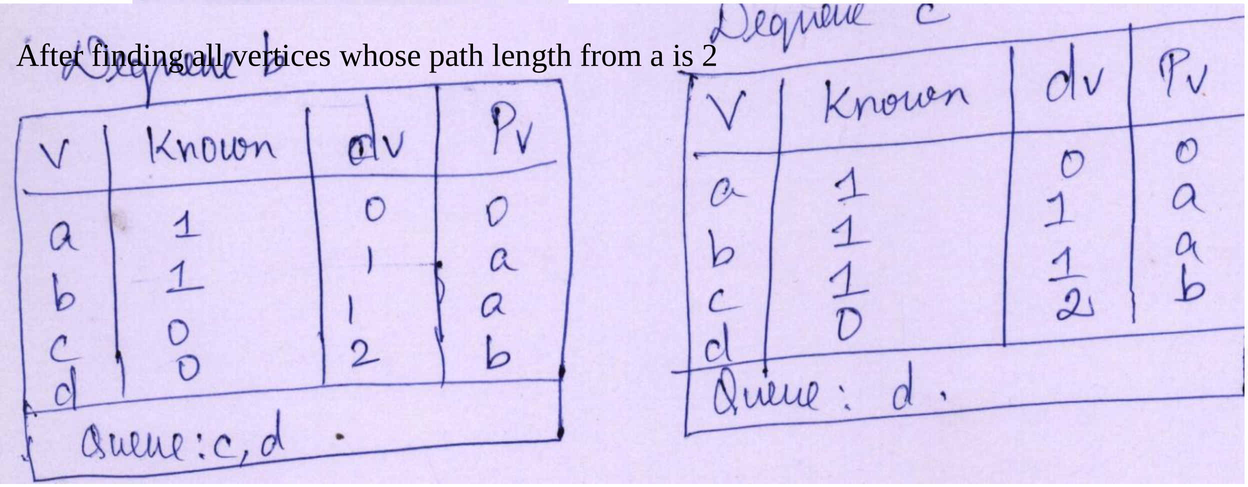


Dequeue : a

V	Known	dv	Pv
a	1	0	0
b	0	1	a
c	0	1	a
d	0	∞	0

Queue : b, c

After finding all vertices whose path length from a is 2



Dequeue b

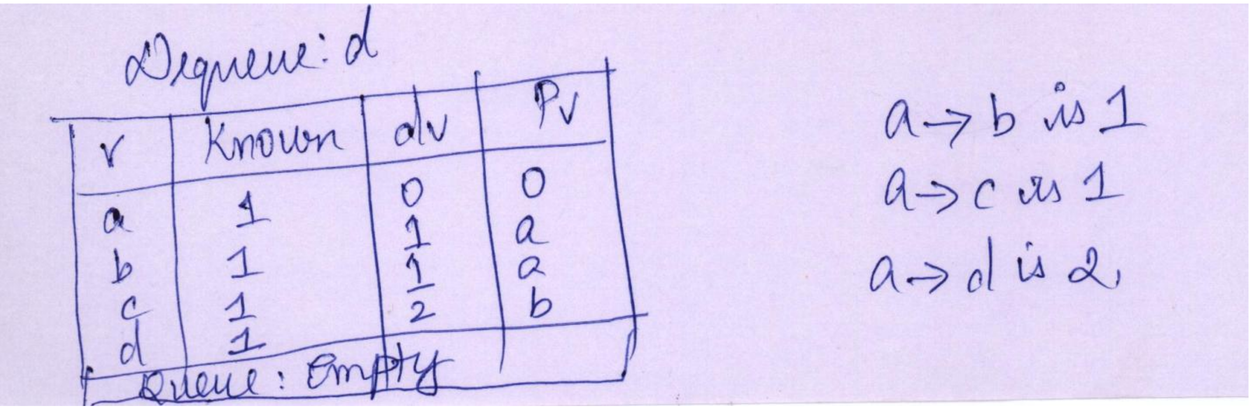
V	Known	dv	Pv
a	1	0	0
b	1	1	a
c	0	1	a
d	0	2	b

Queue : c, d .

Dequeue c

V	Known	dv	Pv
a	1	0	0
b	1	1	a
c	1	1	a
d	0	2	b

Queue : d .



Dequeue : d

V	Known	dv	Pv
a	1	0	0
b	1	1	a
c	1	1	a
d	1	2	b

Queue : Empty

a→b is 1

a→c is 1

a→d is 2,

DIJIKSTRA’S ALGORITHM: Explain about Dijikstra's algorithm: (Apr 12, Nov 12, Nov 13)

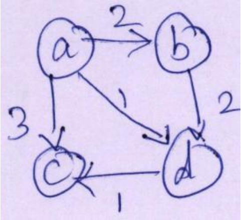
The general method to solve the single source shortest path problem is known as Dijikstra’s algorithm. This is applied to weighted graph

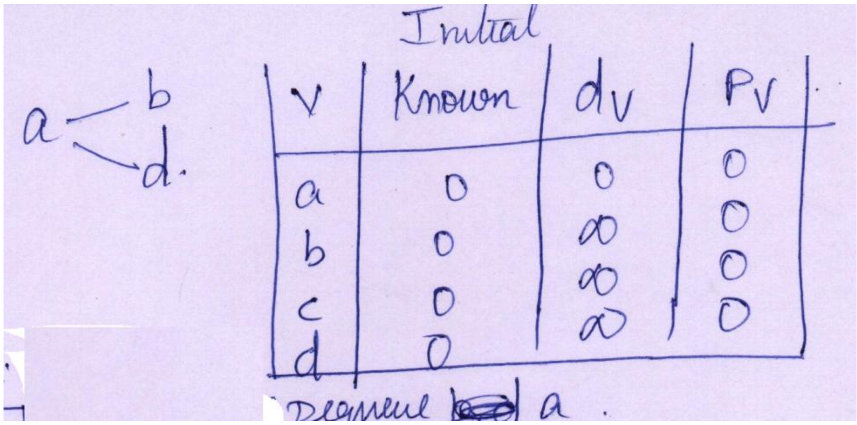
This algorithm is the prime example of Greedy technique, which generally solve a problem in stages by doing what appears to be the best thing in each stage. This algorithm proceeds in stages, just like the unweighted shortest path algorithm. At each stage, it selects a vertex V, which has the smallest dv among all the unknown vertices, and declares that as the shortest path from S to V and mark it to be known. We should set dw = dv + C_{vw}, if the new value for dw would be an improvement.

ALGORITHM:

```
voidDijikstra( Graph G, Table T)
{
vertex V,W;
for(i=0;iNNumvertex;i++)
{
T[i].known=false;
T[i].dist=Infinity;
T[i].path = Not aVetrex;
}
T[start].dist=0;
for( ; ; )
{
V = smallest unknown distance vertes;
if( V = = Not A vertex)
break;
T[V].known=true;
for each W adjacent to V
if(!T[W].known)
{
T[W].dist = Min(T[W].dist,T[V].dist+Cvw)
T[W].path=V;
}
}
}
}
```

EXAMPLE:



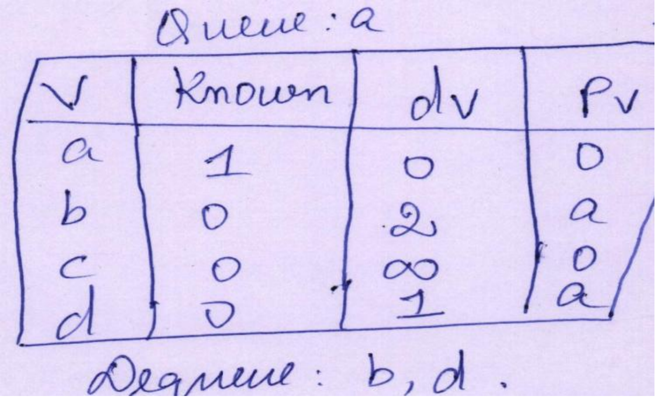


Initial

V	Known	dv	Pv
a	0	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

Dequeue a.

Vertex a ischoosen as source and is declared as known vertex.

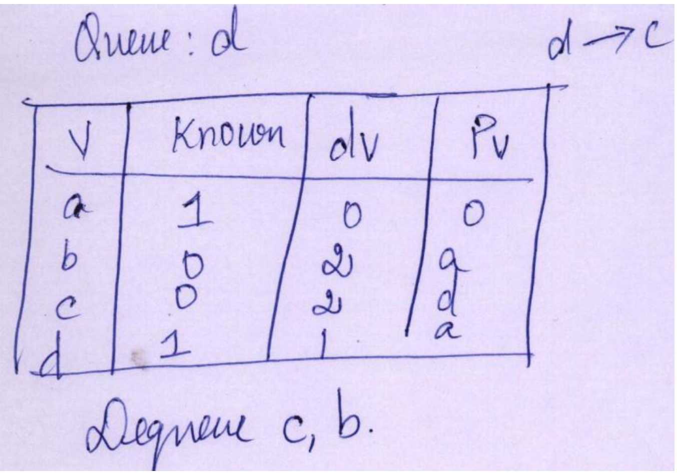


Queue : a

V	Known	dv	Pv
a	1	0	0
b	0	2	a
c	0	∞	0
d	0	1	a

Dequeue : b, d.

Here d is the next minimum distance vertex . The adjacent vertex to d is c, therefore the distance of c is updated as follows:



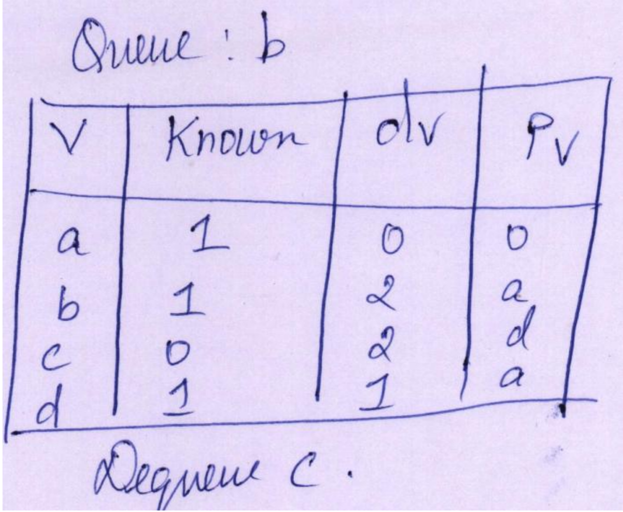
Queue : d

d → c

V	Known	dv	Pv
a	1	0	0
b	0	2	a
c	0	2	d
d	1	1	a

Dequeue c, b.

Next vertex is b.

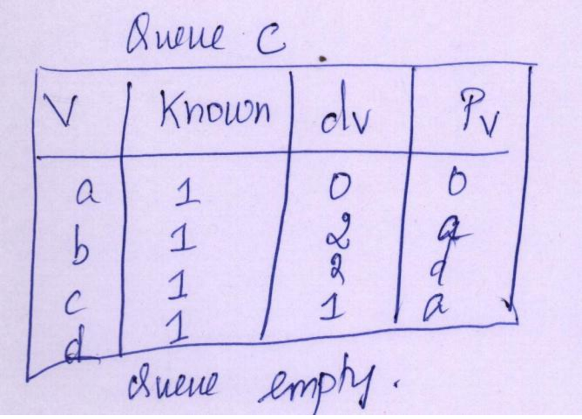


Queue : b

V	Known	dv	Pv
a	1	0	0
b	1	2	a
c	0	2	d
d	1	1	a

Dequeue c.

Since the the adjacent vertex d is already visited , select the next minimum vertex c and marked it as visited.



Queue c.

V	Known	dv	Pv
a	1	0	0
b	1	2	a
c	1	2	d
d	1	1	a

Queue empty.

4. Explain Travelling salesman proble m (UQ APRIL'13 & NOV'12)

- 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 $c_{ij} =\propto$,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:

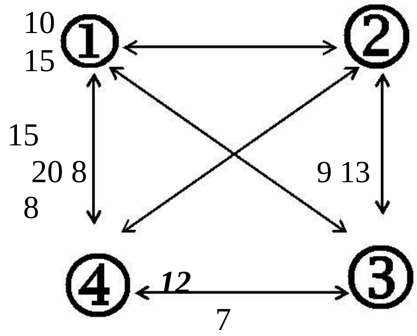
1. Suppose we have to route a postal va n to pick up mail from the mail boxes located at „n‟ different sites.
2. An n+1 vertex graph can be used to represent the situation.
3. One vertex represents the post office from which the postal van starts and return.
4. Edge <i,j> is assigned a cost equal to 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 ∈ 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.

STEPS TO FIND THE PATH:

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



Cost matrix

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

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

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

STEP 1:

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

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

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

$=35$

STEP 2:

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

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

$\min\{29,25\}$

$=25$

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

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

$\min\{31,25\}$

$=25$

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

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

$\min\{23,27\}$

$=23$

STEP 3:

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

   $12+8 = 20$

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

9+6 =15

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

10+8 =18

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

8+5 =13

5. $g(2,\{3\}) = \min\{c_{23} + g\{3,\Phi\}\}$ 9+6=15

5. $g(3,\{2\}) = \min\{c_{32} + g\{2,\Phi\}\}$

13+5=18

STEP 4:

$g\{4,\Phi\} = c_{41} = 8$

$g\{3,\Phi\} = c_{31} = 6$

$g\{2,\Phi\} = c_{21} = 5$

a

s = 0.

i =1 to n.

$g(1,\Phi) = c_{11} \Rightarrow 0$

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

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

$g(4,\Phi) = c_{41} \Rightarrow 8$

$|s| = 1$

i =2 to 4

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

= 9+6 =15

$g(2,\{4\})$

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

= 10+8 =18 $g(3,\{2\})$

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

$= 13+5 = 18$

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

$= 12+8 = 20$

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

$= 8+5 = 13$

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

$= 9+6 = 15$

$s = |2|$

$i \neq 1$, $1 \in s$ and $i \in s$.

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

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

$\min\{29,25\}$

$=25$

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

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

$\min\{31,25\}$

$=25$

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

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

$\min\{23,27\}$

$=23$

$|s| = 3$

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

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

min{35,35,43}

=35 optimal cost is 35

the shortest path is,

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

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

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

So the optimal tour is **1** → **2** → **4** → **3** → **1**

5. Explain chained matrix multiplication (UQ NOV'12 & NOV'10)

If we have a matrix A of size p×q and B matrix of size q×r. the product of these two matrixes C is given by,

$c_{ij} = \Sigma a_{ik} b_{kj}$, 1≤i≤ p, 1≤ j ≤ r , 1= k =q.

it requires a total of pqr scalar multiplication.

Matrix multiplication is associative, so if we want to calculate the product of more than 2 matrixes m= m1m2……. mn.

For example,

A = 13 × 5
B = 5 × 89
C = 89 × 3
D = 3 × 34

Now, we will see some of the sequence,

M = (((A.B).C).D)
A.B C
= (13 * 5 * 89) * (89 * 3)
A.B.C. D
= (13 * 89 * 3) * (3 * 34) A.B.C.D
= 13 * 3 * 34

(ic) = 13 * 5 * 89 + 13 * 89 * 3 + 13 * 3 * 34
= 10,582 no. of multiplications one required for that sequence.

2^{nd} Sequence,

M = (A * B) * (C * D)
= 13 * 5 * 89 + 89 * 3 * 34 + 13 * 89 * 34
= 54201 no. of Multiplication

3^{rd} Sequence,

M = (A.(BC)) . D
= 5 * 89 * 3 + 13 * 5 * 3 + 13 * 3 *34
= 2856

For comparing all these sequence, (A(BC)).D sequences less no. of multiplication.

For finding the no. of multiplication directly we are going to the Dynamic programming method.

STRAIGHT FORWARD METHOD:

Our aim is to find the total no. of scalar multiplication required to compute the matrix product.

Let (M1,M2,……Mi) and Mi+1,Mi+2,……Mn be the chain of matrix to be calculated using the dynamic programming method.

In dynamic programming we always start with the smallest instances and continue till we reach the required size.

We maintain a table mij, 1≤i≤ j≤ n,
Where mij gives the optimal solution.

Sizes of all the matrixes are stored in the array d[0..n]

We build the table diagonal by diagonal; diagonal s contains the elements mij such that j-1 =s.

RULES TO FILL THE TABLE Mij:

S =0,1,……n-1

If s=0 =>m(i,i) =0 ,i =1,2,……n
If s=1 =>m(i,i+1) = d(i-1)*di *d(i+1)

i=1,2,……n-1.

If 1< s <n =>$m_{i,i+s}$ =min(m_{ik}+$m_{k+1,i+s}$+$d_{i-1}d_k d_{i+s}$) i≤ k < i+si = 1,2,……n-s

Apply this to the example,

A=>13×5
B=>5×89
C=>89×3
D=>3×34

Single dimension array is used to store the sizes.

d[0]=13
d[1]=5
d[2]=89
d[3]=3
d[4]=34

if s=0,

m(1,1)=0
m(2,2)=0
m(3,3)=0

m(4,4)=0

if s=1,

mi,i+1 = d(i-1)*di *d(i+1)

m12 = d0*d1*d2 =13*5*89=5785
m23 = d1*d2*d3 =5*89*3=1335
m34 = d2*d3*d4 =89*3*34=9078

if s=2,

$m_{i,i+s} = \min(m_{ik}+m_{k+1,i+s}+d_{i-1}d_kd_{i+s})$

$m_{13} = \min(m_{11}+m_{23}+d_0d_1d_3 \text{ , } m_{12}+m_{33}+d_0d_2d_3)$

=(0+1335+(13*5*3),5785+0+(13*89*3))

=min(1530,9256)

=1530

$m_{13} = \min(m_{22}+m_{34}+d_1d_2d_4 \text{ , } m_{23}+m_{44}+d_1d_3d_4)$

=(0+9078+(5*89*34),1335+0+(5*3*34))

=min(24208,1845)

=1845

$M_{24} = \min(m_{11}+m_{24}+d_0d_1d_4 \text{ , } m_{12}+m_{34}+d_0d_2d_4, \; m_{13}+m_{44}+d_0d_3d_4)$

=min(4055,54201,2856)

=2856

the matrix table mij will look like,

	1	2	3	4	
1	0	5785	1530	2856	s=3
2		0	1335	1845	s=2
3			0	9078	s=1
4				0	s=0

Here there is no need to fill the lower diagonal.

ALGORITHM:

Procedure cmatrix(n,d[0..n])

```
For s=0 to n-1 do
   {

      if(s==0)

      for I=1 ton do

         m(i,j) =0

   if(s==1)
    fori=1 to n-1 do
      m(i,i+1) =d(i-1)*d(i)*d(i+1)

    else
   {
      m=∝
       fori=1 to n-s do
        for k=i to i+s do

      {


             min=[m(i,k) +m(k+1,i+s)+d(i-1)*d(k)*d(i+s)]

         }

           m(i,i+s) =min

         }

      }
```

6. Explain 0/1 knapsack proble m (UQ APRIL’12)

- 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‟s 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) = max T(i-1,j), Vi +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.

ES TO GET THE COMBINATION OF OBJECT:

- Start with the last position of i and j, T[i,j], if T[i,j] = T[i- l,j] then no object of „i‟ is required so move up to T[i- l,j].

- After moved, we have to check if, T[i,j]=T[i- l,j-w(i)]+ p[I], if it is equal then one unit of object „i‟ is selected and move up to the position T[i- l,j-w(i)]

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

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

- Consider a Example,

M = 6,
N = 3
$W_1 = 2, W_2 = 3, W_3 = 4$
$P_1 = 1, P_2 =2, P_3 = 5$
i $\longrightarrow$ 1 to N
j $\longrightarrow$ 0 to 6

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

o<2 T $\longrightarrow$=0

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

l<2 T $\longrightarrow$=0 (Here j is equal to w(i) P(i) ⟶

i=l, j=2
2 o,= T_1,2 = l.

i=l, j=3
3>2,= T_1,3 = l.

i=l, j=4
4>2,= T_1,4 = l.

i=l, j=5
5>2,= T_1,5 = l.

i=l, j=6
6>2,= T_1,6 = l.

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

i=2, j=1
l<3= T(2,1)
= T(i- l) T 2,1 =0

8. Explain basic search and traversal technique (UQ APRIL'13,NOV'14)

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.

GRAPH TRAVERSAL

Given an undirected graph G = (V.E) and a vertex v in V(G) we are interested in visiting all vertices in G that are reachable from v (that is all vertices connected to v). We have two ways to do the traversal. They are

- DEPTH FIRST SEARCH
- BREADTH FIRST SEARCH.

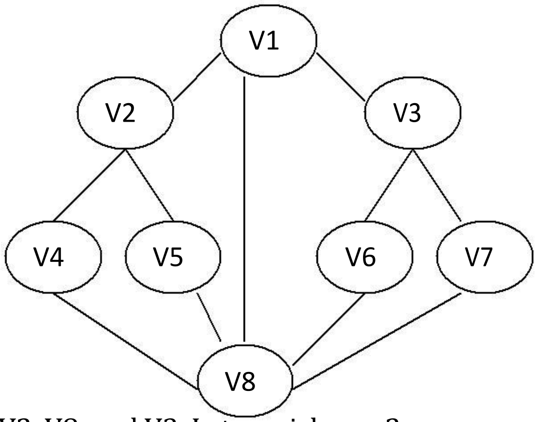
DEPTH FIRST SEARCH

In graphs, we do not have any start vertex or any special vertex singled out to start traversal from. Therefore the traversal may start from any arbitrary vertex.

We start with say, vertex v. An adjacent vertex is selected and a Depth First search is intimated from it, i.e. let V_1, V_2..V_k are adjacent vertices to vertex v. We may select any vertex from this list. Say, we select V_1. Now all the adjacent vertices to v_1 are identified and all of those are visited; next V_2 is selected and all its adjacent vertices visited and so on.

This process continues till are the vertices are visited. It is very much possible that we reach a traversed vertex second time. Therefore we have to set a flag somewhere to check if the vertex is already visited.

Let us see an example, consider the following graph.



Let us start with V_1,

1. Its adjacent vertices are V2, V8, and V3. Let us pick on v2.
2. Its adjacent vertices are V1, V4, V5, V1 is already visited.
3. Let us pick on V4.
4. Its adjacent vertices are V2, V8.
5. V2 is already visited .let us visit V8.
6. Its adjacent vertices are V4, V5, V1, V6, V7.
7. V4 and V1 are visited. Let us traverse V5.
8. Its adjacent vertices are V2, V8. Both are already visited therefore, we back track.
9. We had V6 and V7 unvisited in the list of V8. We may visit any. We may visit any. We visit V6.
10. Its adjacent is V8 and V3. Obviously the choice is V3.
11. Its adjacent vertices are V1, V7. We visit V7.
12. All the adjacent vertices of V7 are already visited, we back track and find that we have visited all the vertices.

Therefore the sequence of traversal is V1,

V2, V4, V5, V6, V3, V7.

This is not a unique or the only sequence possible using this traversal method.

We may implement the Depth First search by using a stack, pushing all unvisited vertices to the one just visited and popping the stack to find the next vertex to visit.

This procedure is best described recursively as in,

```
Procedure DFS(v)
// Given an undirected graph G = (V.E) with n vertices and an array visited (n) initially set to zero . This algorithm
visits all vertices reachable from v .G and VISITED are global > //VISITED (v) ← 1
for each vertex w adjacent to v do if
VISITED (w) =0 then call DFS (w)end
end DFS
```

COMPUTING TIME

1. In case G is represented by adjacency lists then the vertices w adjacent to v can be determined by following a chain of links. Since the algorithm DFS would examine each node in the adjacency lists at most once and there are 2e list nodes. The time to complete the search is O (e).
2. If G is represented by its adjacency matrix, then the time to determine all vertices adjacent to v is O(n).

   Since at most n vertices are visited. The total time is $O(n^2)$.

BREADTH FIRST SEARCH

- In DFS we pick on one of the adjacent vertices; visit all of its adjacent vertices and back track to visit the unvisited adjacent vertices.

- In BFS , we first visit all the adjacent vertices of the start vertex and then visit all the unvisited vertices adjacent to these and so on.

- Let us consider the same example, given in figure. We start say, with V_1. Its adjacent vertices are V_2, V_8 , V_3.

- We visit all one by one. We pick on one of these, say V_2. The unvisited adjacent vertices to V_2 are V_4, V_5 .we visit both.

- We go back to the remaining visited vertices of V_1 and pick on one of this, say V_3. T The unvisited adjacent vertices to V_3 are V_6,V_7. There are no more unvisited adjacent vertices of V_8, V_4, V_5, V_6 and V_7.

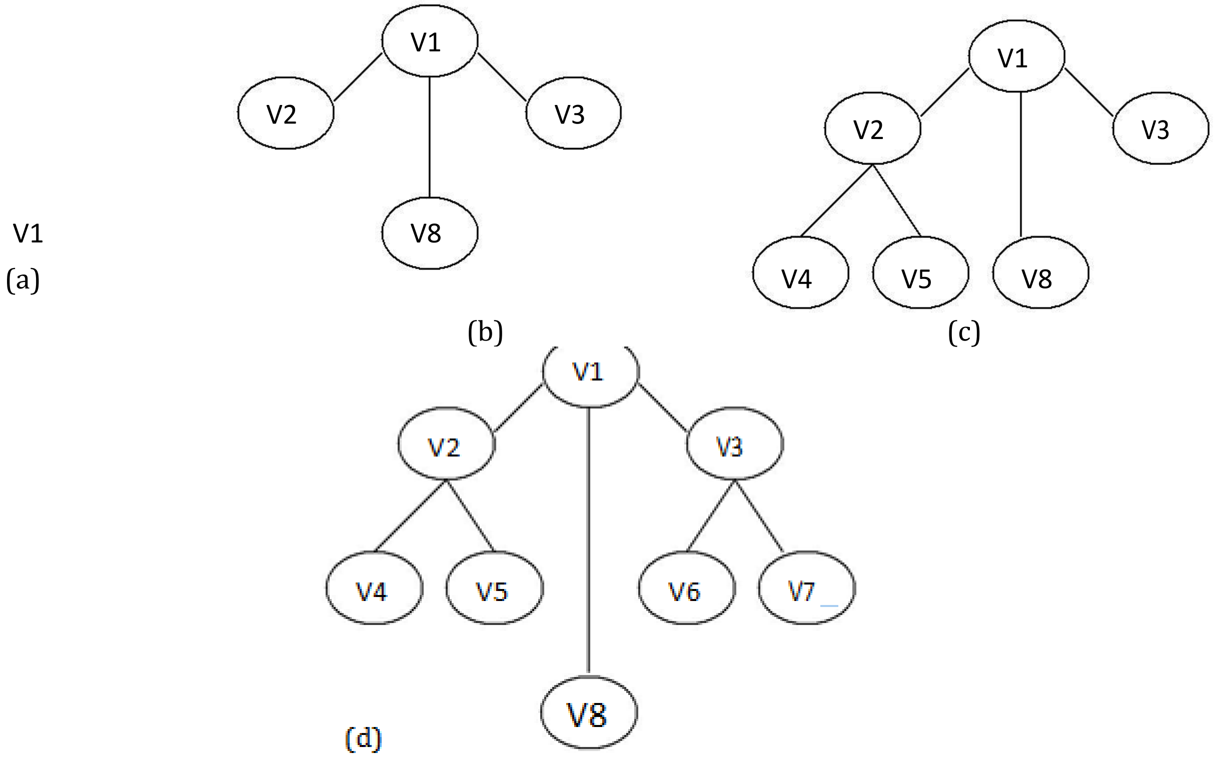


FIG: Breadth First Search

- Thus the sequence so generated is V_1,V_2, V_8, V_3,V_4, V_5,V_6, V_7. Here we need a queue instead of a stack to implement it.

- We add unvisited vertices adjacent to the one just visited at the rear and read at from to find the next vertex to visit.

Algorithm BFS gives the details.

Procedure BFS(v)

//A breadth first search of G is carried out beginning at vertex v. All vertices visited are marked as VISITED(I) = 1. The graph G and array VISITED are global and VISITED is initialised to 0.//

```
VISITED(v) $\leftarrow$ 1
Initialise Q to be empty //Q is a queue//
loop
for all vertices w adjacent to v do
if VISITED(w) = 0 //add w to queue//
then [call ADDQ(w, Q); VISITED(w) $\leftarrow$ 1] //mark w as VISITED//
end
if Q is empty then return
call DELETEQ(v,Q)
forever
end BFS
```

COMPUTING TIME

1. Each vertex visited gets into the queue exactly once, so the loop forever is iterated at most n times.
2. If an adjacency matrix is used, then the for loop takes O(n) time for each vertex visited. The total time is, therefore, O(n^2).
3. In case adjacency lists are used the for loop as a total cost of d_1+……..+d_n = O(e) where d_i = degree(v_i). Again, all vertices visited. Together with all edges incident to from a connected component of G.

9. Explain biconnected components (UQ:APRIL/MAY’14) Articulation Point:

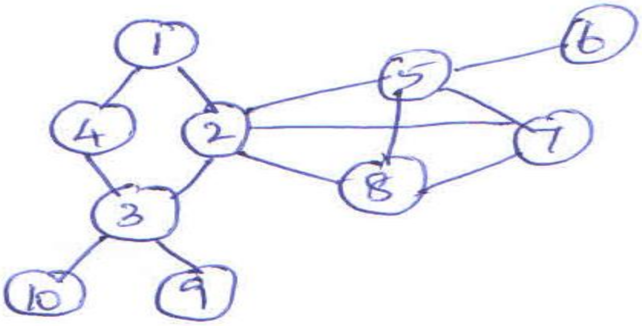
A vertex „v‟ in a connected graph „G‟ is an articulation point, iff the deletion of vertex v together with all vertices incident to v disconnects the graph into 2 or more non empty components.

Biconnected Graph:

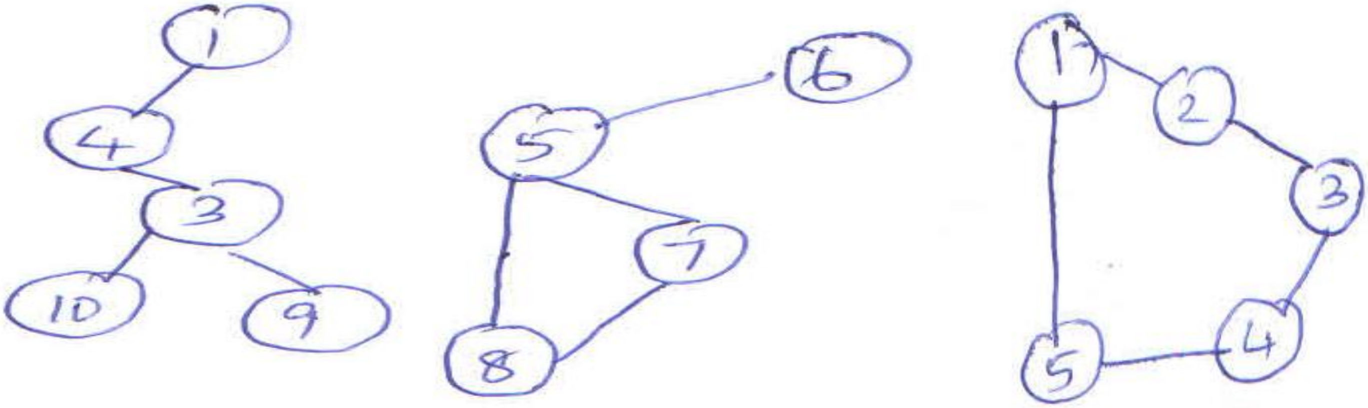
A graph „G‟ is biconnected, iff it contains no articulation points. Real Time Example:

„G‟ is a graph representing a communication network, vertices represent communication station, edges represent communication lines.

Example for articulation point in a graph



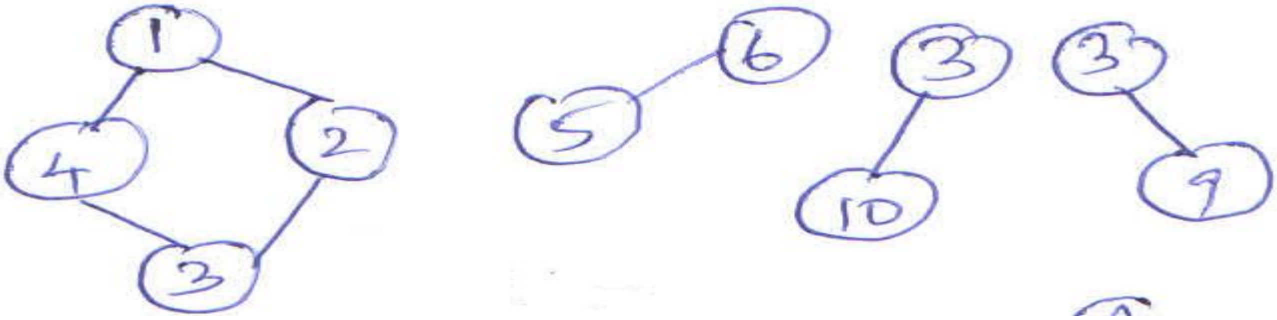
- In this Graph Vertex 2 is an articulation point.
- Deletion of 2 produces 2 disconnected non empty components.
- It is not a biconnected graph

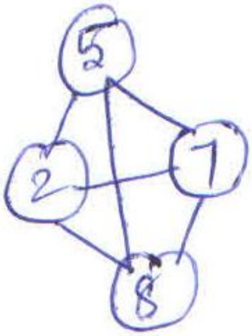


- Deletion of anyone of the vertex does not produce any biconnected component
- The graph han no articulation point

IDENTIFICATION OF ARTICULATION POINT USING COMMON VERTEX IN BICONNECTED COMPONENT:

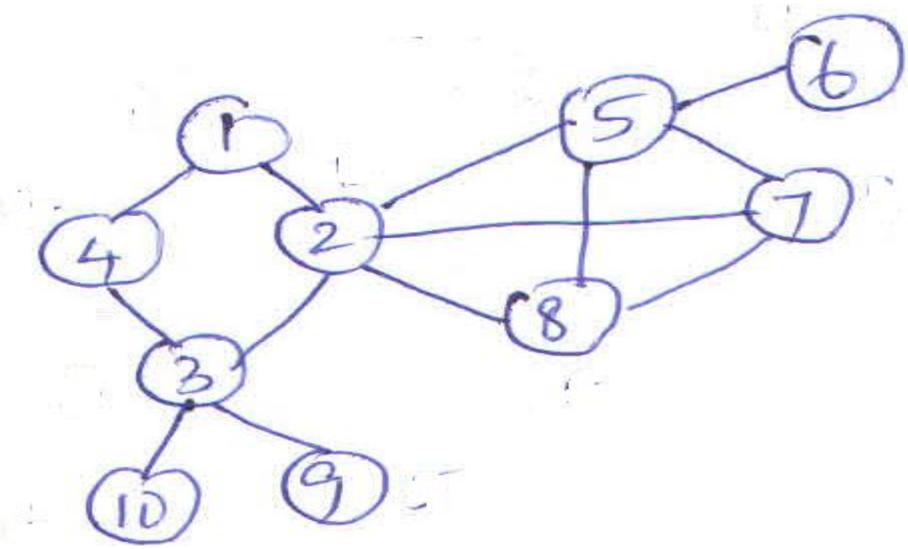
- Two biconnected component can have one vertex in common and this vertex is an articulation point
- If G has P articulation point and b biconnected components, the graph contains (b-p) new edges in G.





- This graph has 5 biconnected components and 2 articulation point.

- Combining of all biconnected it requires addition (5-2) of 3 edges.



Algorithm for connecting biconnected components using articulation points for each articulation point a do

{

Let B1,B2,…..Bk be the bi connected components containing vertex a

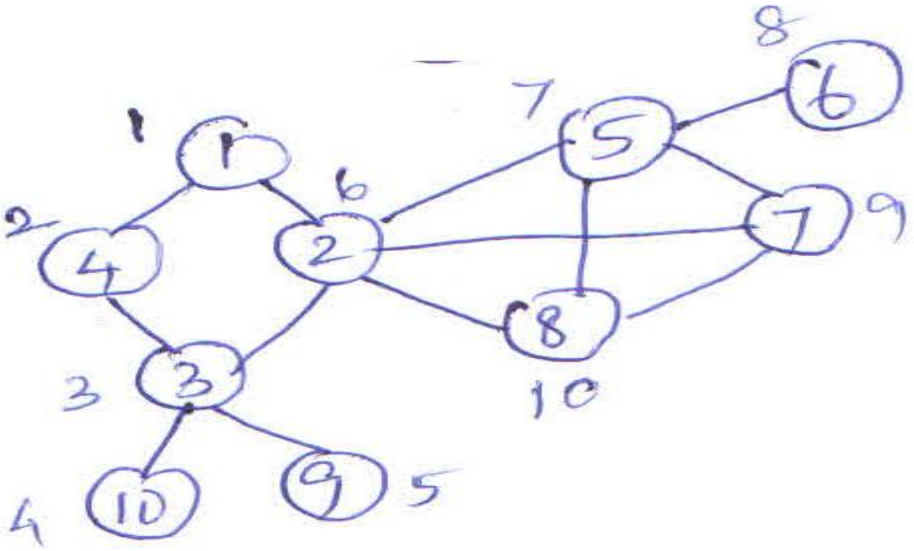
Let Vi,Vi is not equal to a be a vertex in Bi, 1<=i<=k;

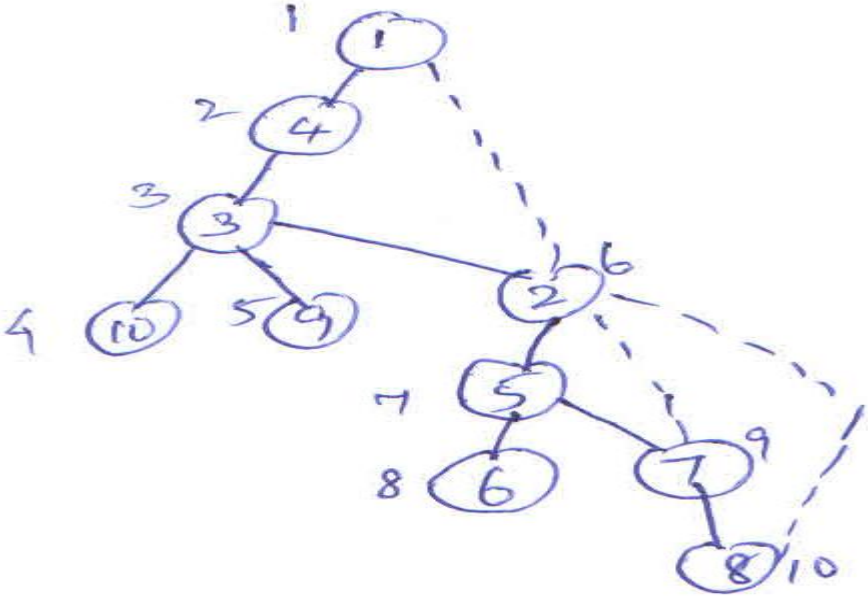
Add to G the edges (Vi,Vi+1), 1<=i<=k;

}

IDENTIFICATION OF ARTICULATION POINT USING DEPTH FIRST SPANNING TREE

- Depth First search provides a linear time algorithm, to find all articulation pointers in a connected graph.
- First, starting at any vertex , we perform a depth first search and number the nodes as they are visited.

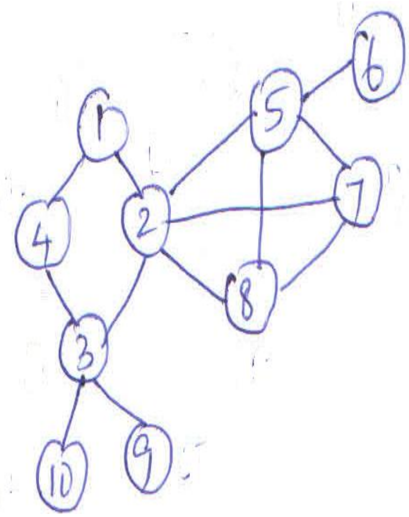


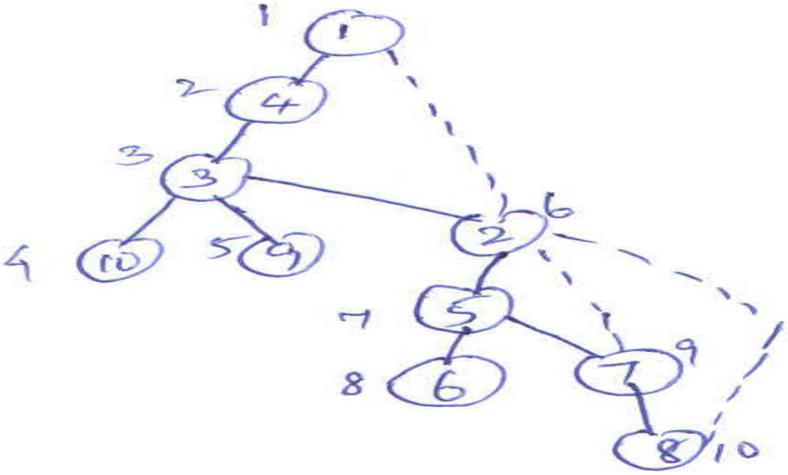


IDENTIFICATION OF ARTICULATION POINT USING LOWEST DEPTH FIRST NUMBER L9u)

Definition of lowest depth number L(u)”

Before every vertex V in the depth first spanning tree we compute the L(U) that is reachable from V by taking Zero or more tree edges and then possibly one back edge.





Find L(u) for all the vertices

L(1) = dfn(1)=1

L(2) = min{dfn(2),back edge of(2,1)}

=1

.
.
.
.

.

Refer Class notes for this Problem

```
Algorithm Bicomp(u,v)
{
If(dfn[w]<dfn[u]) then
Add[u,w] to the top of the stack S;
If(L(W)>=dfn(u))then
{
Write ("new Bicomponents");
Repeat;
{
Delete an edge from top of stack S;
Let the edge be(x,y)'
Write(x,y);
}
Until false;
}
Bicomp(w,u)
L(u)=min(L(u),L(w));
}
```

10. Explain topological sorting (UQ NOV'10, APRIL/MAY"14)

A topological sort is a linear ordering of vertices in adirected acyclic graph such that if there is a path from Vi to V j appears after Vi in the linear ordering

Topological sort is not possible. if the graph has a cycle, since for two vertices v and w on the cycle , v proceeds w and w proceeds v

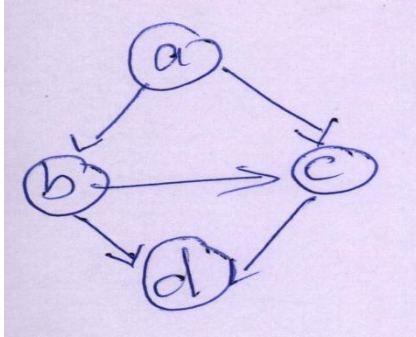
To implement the toplogicalsort , performs the following steps:

1. Find the indegree for every vertex
2. Place the vertices whose indegree is 0 on empty queue.
3. Dequeue the vertex V and decrement the indegree's of all its adjacent vertices.
4. Enqueue the vertex on the queue, if its indegree falls to zero.
5. Repeat from step 3 until queue become empty
6. The topological ordering is the order in which the vertices dequeued

ALGORITHM:

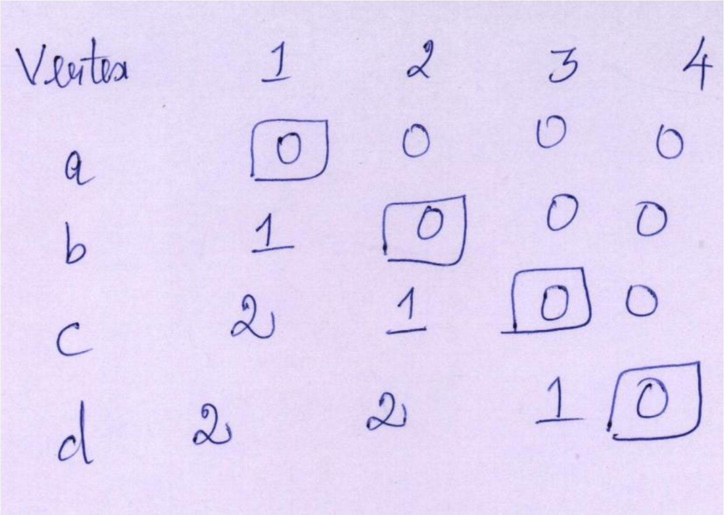
```
Void Toposort(Graph G)
{
Queue Q;
Vertex V,W;
Q=Create Queue(NumVErtex);
Makeempty(Q);
for each vertex V
if(indegree[v]== 0)
Enqueue(V,Q);
while(!ISEmpty(Q))
{
V=dequeue(Q);
TopNum[V]=++counter;
for each W adjacent to V
if(--Indegree[W]==0)
Enqueue(W,Q);
}
DisposeQueue(Q);
}
```

EXAMPLE:



Step1: Find indegree of all vertices:
indegree[a]=0 indegree[b]=1
indegree[c]=2 indegree[d]=2

Step2: Enqueue the vertex whose indegree is 0
Vertex a is 0 so placed in on Queue
Step3: Dequeue the vertex a from the queue and decrement the indegree of its adjacent vertex b and c
Hence indegree[b]=0 and indegree[c]=1
Now, enqueue the vertex b as its indegree becomes zero
Step4: Dequeue the vertex b from Q and decrement the indegress's of its adjacent vertex c and d
Hence indegree[c]=0 and indegree[d]=1
Now enqueue the vertex c as its indegree falls to zero
Step5: Dequeue the vertex c from Q and decrement the indegree's of its adjacent vertex d
Hence indegree[d]=0
Now enqueuue the vertex d as its indegree fall to zero
Step6: Dequeue the vertex d
Step7: As the Queue becomes empty , the topological ordering is performedwhich is nothing but the order in which the vertices are dequeued.



Vertex	1	2	3	4
a	$\boxed{0}$	0	0	0
b	1	$\boxed{0}$	0	0
c	2	1	$\boxed{0}$	0
d	2	2	1	$\boxed{0}$

11. Explain Binary Tree Search and Traversal.

- A binary tree traversal requires that each node of the tree be processed once and only once in a predetermined sequence.
- The two general approaches to the traversal sequence are,
  - ***Depth first traversal***
  - ***Breadth first traversal***

- In depth first traversal, the processing proceeds along a path from the root through one child to the most distant descendent of that first child before processing a second child. *In other words, in the depth first*

  traversal, all the descendants of a child are processed before going to the next child.
- In a breadth-first traversal, the processing proceeds horizontally form the root to all its children, then to its children's children, and so forth until all nodes have been processed. *In other words, in breadth*

  traversal, each level is completely processed before the next level is started.

<u>Depth-First Traversal</u>

There are basically three ways of binary tree traversals. They are :

1. **Pre Order Traversal**
2. **In Order Traversal**
3. **Post Order Traversal**

In C, each node is defined as a structure of the following form: struct node

```
                {
                int info;
                struct node *lchild;
                struct  node *rchild;
                }
        typedefstruct node NODE;
```

<u>Binary Tree Traversals</u>(Recursive procedure) <u>1. Inorder</u>

<u>Traversal</u>

Steps 1.Traverse left subtree in inorder
2.Process root node

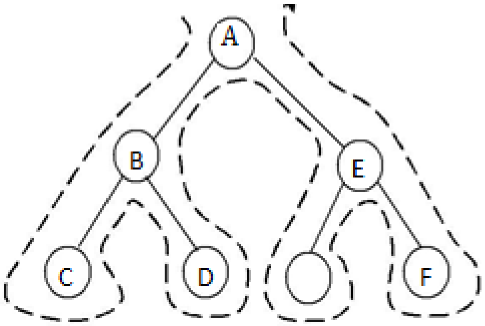
3.Traverse right subtree in inorder

Algorithm

```
Algorithm inoder traversal (Bin-Tree T)

    Begin
    If ( not empty (T) ) then
    Begin
        Inorder_traversal ( left subtree ( T ) )
        Print ( info ( T ) )            / * process node */
        Inorder_traversal ( right subtree ( T ) )
    End
    End
```

C Coding

```
    void inorder_traversal ( NODE * T)
      {
      if( T ! = NULL)
      {
    inorder_traversal(T->lchild);
printf("%d \t ", T->info);
    inorder_traversal(T->rchild);
    }
    }
```



The Output is C -> B -> D -> A -> E -> F

2. Preorder Traversal

Steps : 1. Process root node

2. Traverse left subtree in preorder
3. Traverse right subtree in preorder

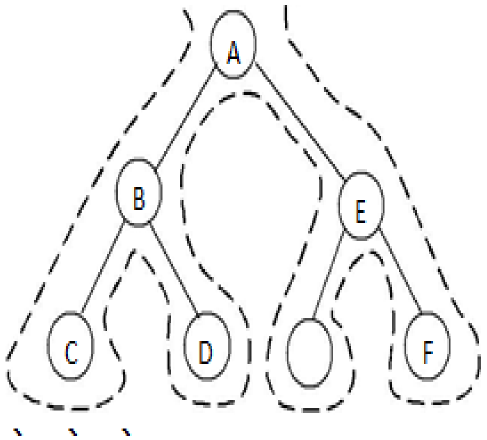
Algorithm

```
Algorithm preorder traversal (Bin-Tree T)
Begin
If ( not empty (T) ) then
Begin
    Print ( info ( T ) ) / * process node * /
    Preoder traversal (left subtree ( T ) )
```

```
                    Inorder traversal ( right subtree ( T ) )
          End
End
```

C function

```
       void preorder_traversal ( NODE * T)
       if( T ! = NULL)
       {
         printf("%d \t ", T->info);
         inorder_traversal(T->lchild);
inorder_traversal(T->rchild);
}
```



The Output is A -> B-. C -> D -> E -> F

3. Postorder Traversal

Steps : 1. Traverse left subtree in postorder
            2. Traverse right subtree in postorder
            3. process root node

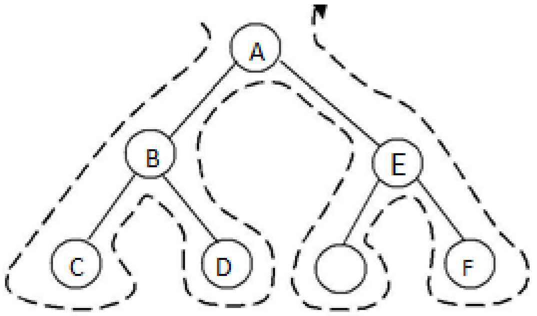
Algorithm

```
Postorder Traversal

Algorithm postorder traversal (Bin-Tree T)

Begin

If ( not empty (T) ) then

Begin

            Postorder_traversal ( left subtree ( T ) )

            Postorder_traversal ( right subtree( T))

            Print ( Info ( T ) ) / * process node */

End

End
```

C function

```
    void postorder_traversal ( NODE * T)

          {

        if( T ! = NULL)

                    {

                    postorder_traversal(T->lchild);
```

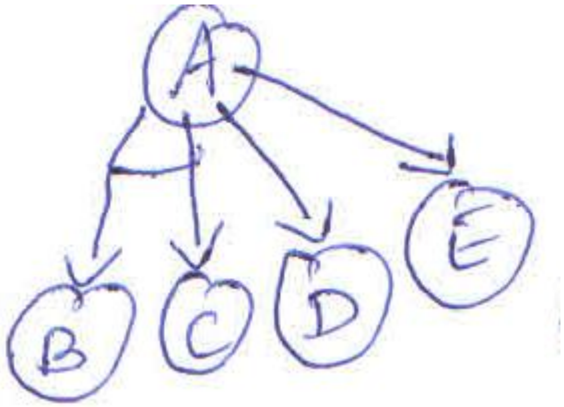
```
            postorder_traversal(T->rchild);
            printf("%d \t", T->info);
        }
    }
```

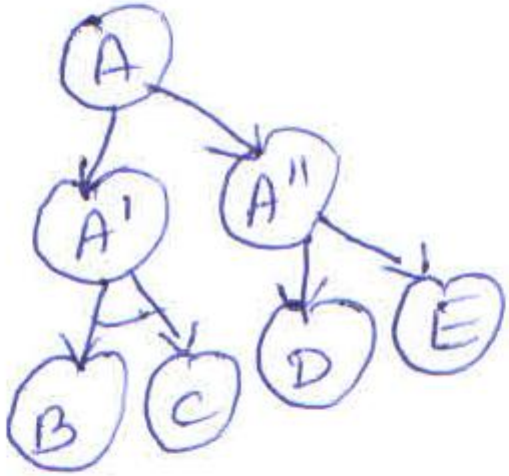


The Output is C -> D -> B -> F -> E -> A

12. Explain AND / OR Graph.(AOG)

Here the problem is solved into many sub problems. Descendent of node is sub problem.





The graph A represents problem A that can be solved by solving either both sub node B amd C or D or E.

Arc represents AND .

We prodeces the dummy variable A and $A^{II}$are OR

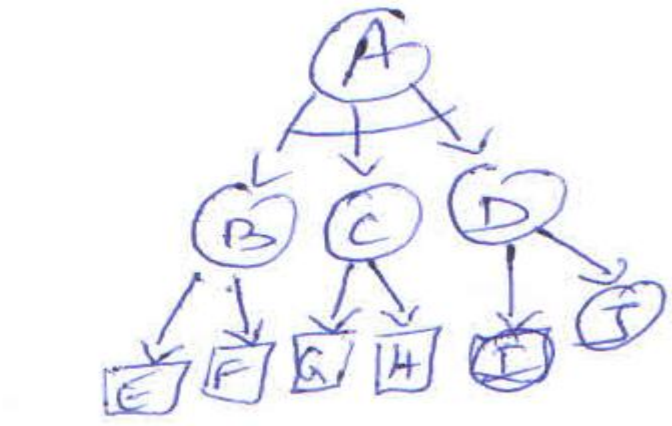
A and $A^{I}$are AND (arc)

Nodes with no descendents is terminal.

Terminals can be divided into 2 :

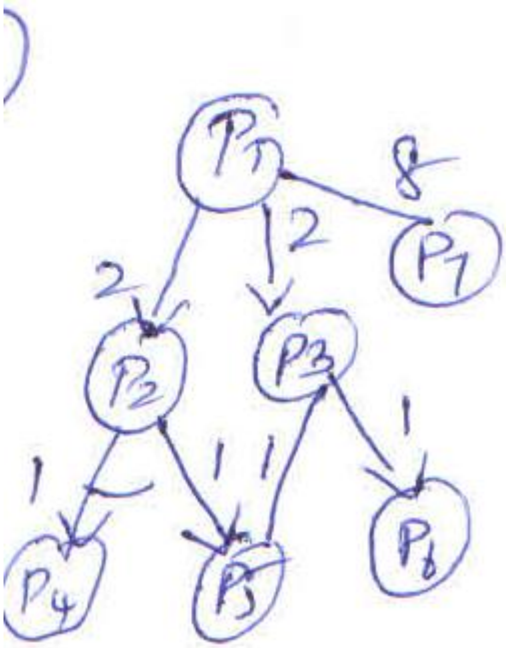
- Solvables (represent by square symbol)
- Non solvables (represent by circle symbol)

Breaking down a problem into many sub problem is problem reduction. Two different problems may generate a same problem. Graph is not a tree



Problem:

Consider a directed graph . The problem to be solved is P1. To do this, one can solve node P2,P3,P7 as P1 in an OR node. The cost incurred is either 2,2,8(P2,P3,P7). To solve P2, both P4 and P5 have to solved as P2 in AND node. Total cost to do this is 2. To solve P3 we can solve either P3 or P6 . The minimum cost is 1. Node P7 is set free.



Therefore optimal way is to solve P1, is tio solve P6 first and then P3 and finally P1

Cost = 3

Pondicherry University Questions

2 Marks

1 Define dynamic programming. (UQ APRIL‟12) (Qn.No.2)

2. Write the general procedure of dynamic programming. (UQ APRIL‟13) (Qn.No.5)

3. List different traversal technique in graph (UQ APRIL‟12) (Qn.No.2)

4. What is biconnected component? (UQ Nov‟12) (Qn.No.15)

5. Write about multistage graph with example (UQ APRIL‟13 & Nov‟12,Apr/May‟14 Qn.No.17)

6. What is all pair shortest path problem?(APRIL/MAY‟14) (Qn.No.18)

7. Define principle of optimality. (UQ:NOV‟14) (Qn.No.6)

8. What is articulation point? (UQ:NOV‟14) (Qn.No.16)

11 Marks

1. Explain Travelling salesman problem (UQ APRIL‟13 & NOV‟12) (Qn.No.4)

2. Explain chained matrix multiplication (UQ NOV‟12 & NOV‟10) (Qn.No.5)

3. Explain 0/1 knapsack problem (UQ APRIL‟12) (Qn.No.6)

4. Explain basic search and traversal technique (UQ APRIL‟13,NOV‟14) (Qn.No.8)

5. Explain topological sorting (UQ NOV‟10, APRIL/MAY‟14) (Qn.No.10)

6. Explain multistage graph with example (UQ :APRIL/MAY‟14) (Qn.No.2)

7.ExplainBiconnected components (UQ:APRIL/MAY‟14)) Qn.No.9)

8.Explain all pair shortest path (UQ:NOV‟14) (Qn.No.3)

UNIT IV

Backtracking: The general method – 8-queens problem – sum of subsets – graph coloring – Hamiltonian cycle – Knapsack problem.

2 Marks

1. What are the require ments that are needed for performing Backtracking? (UQ NOV'10)

To solve any problem using backtracking, it requires that all the solutions satisfy a complex set of constraints. They are:

i. Explicit constraints.
ii. Implicit constraints.

2. Define explicit constraint.

They are rules that restrict each xi to take on values only from a give set. They depend on the particular instance I of the problem being solved. All tuples that satisfy the explicit constraints define a possible solution space.

3. Define implicit constraint.

They are rules that determine which of the tuples in the solution space of I satisfy the criteria function. It describes the way in which the xi must relate to each other.

4. Define state space tree. (UQ:NOV'14)

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

5. Define state space of the problem.

All the paths from the root of the organization tree to all the nodes is called as state space of the problem

6. Define answer states.

Answer states are the solution states s for which the path from the root to s defines a tuple that is a member of the set of solutions of the problem.

7. What are static trees?

The tree organizations that are independent of the problem instance being solved are called as static tree.

8. What are dynamic trees?

The tree organizations those are independent of the problem instance being solved are called as static tree.

9. Define a live node.

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

10. Define an E – node.

E – Node (or) node being expanded. Any live node whose children are currently being generated is called as an E – node.

11. Define a dead node.

Dead node is defined as a generated node, which is to be expanded further all of whose children have been generated.

12. What are the factors that influence the efficiency of the backtracking algorithm?

The efficiency of the backtracking algorithm depends on the following four factors. They are:
i. The time needed to generate the next xk
ii. The number of xk satisfying the explicit constraints.
iii. The time for the bounding functions Bk
iv. The number of xk satisfying the Bk.

13. State 8 – Queens's proble m.

The problem is to place eight queens on a 8 x 8 chessboard so that no two queen "attack" each other that is, so that no two of them are on the same row, column or on the diagonal.

14. State Sum of Subsets proble m. (UQ APRIL'13 & APRIL'12,NOV'14)

Given n distinct positive numbers usually called as weights, the problem calls for finding all the combinations of these numbers whose sums are m.

15. State m – colorability decision problem. (UQ NOV'10)

Let G be a graph and m be a given positive integer. We want to discover whether 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.

16. Define chromatic number of the graph. (UQ APRIL'13)

The m – colorability optimization problem asks for the smallest integer m for which the graph G can be colored. This integer is referred to as the chromatic number of the graph

17. Define a planar graph.

A graph is said to be planar iff it can be drawn in such a way that no two edges cross each other.

18. Explain Hamiltonian cycles? (UQ NOV'12 & APRIL'12,APRIL/MAY'14)

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.

19. Define Backtracking (UQ APRIL'12,APRIL/MAY'14)

Depth first node generation with bounding function is called backtracking

11 Marks

1. **Explain general Backtracking method (UQ NOV'12)**

- 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.

Algorithm:

```
Algorithm IBacktracking (n)
// This schema describes the backtracking procedure .All solutions are generated in
X[1:n] //and printed as soon as they are determined.
 {
   k=1;
   While (k ≠ 0) do
   {
     if (there remains all untried
     X[k] ∈ T (X[1],[2],…..X[k-1]) and Bk (X[1],…..X[k])) is true ) then
    {
      if(X[1],……X[k] )is the path to the answer node)
     Then write(X[1:k]);
     k=k+1;              //consider the next step.
   }
  else k=k-1;                    //consider backtracking to the previous set.
 }
}
```

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

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

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

- Certain problems which are solved using backtracking method are,

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

2. Explain 8-Queens Problem?(UQ APRIL'13 & APRIL '12,APRIL/MAY'14)

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 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|.

Refer Example 4 queen in class notes

STEPS TO GENERATE THE SOLUTION:

- We start with the empty board and the placed queen 1 in the first poosible position i.e(1,1) row 1 , column 1.
- Queen 2 cannot be placed in (2,1) and (2,2) so acceptable position is(2,3) that is row 3 , column 3
- This position proves to be dead end, because there is no acceptable position for queen 3 . So the algorithm backtracks and put queen 2 in next possible position at (2,4) that is row 2, column 4.
- The queen 3 is placed at (3,2) that is row 3 , column 2 this lead dead end.
- the algorithm backtracks queen 1 and move it to (1,2) that is row 1, column 2.
- Queen 2 cannot be placed in (2,1), (2,2), (2,3) so acceptable position is (2,4) that is row 2, column 4.
- Queen 3 is placed at (3,1) that is row 3, column 1.
- Queen 4 cannot be placed in (4,1),(4,2) . So acceptable position is (4,3)

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)

```
//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) ;
          }
      }
}
```

Solving 8 queens problem:

Placing Queen on chess board, we have to check 3 conditions.

1. No 2 Queens on same row
2. No 2 Queens on same column
3. No 2 Queens on same diagonal.

Refer Example 8 queen in class notes

Complexity of 8 queen problem:

t(n)=def n!

3. Explain sum of subset

- 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.

Algorithm:

```
Algorithm sumofsubset(s,k,r)
{
        //generate the left child. notes+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]);
        }
}
```

Refer Example in class notes

Complexity:

$o(2^{N/2}N)$

4. Explain Hamiltonian cycles? (UQNOV'10,NOV'14)

- 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.

Refer Example in class notes

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 8 vertex. 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);
}
```

6. Explain graph coloring (UQ APRIL’12,APRIL/MAY’14,NOV’14)

- 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.

Procedure for Graph coloring:

- The basic idea behind the solution is that once a vertex is assigned a color then all the vertices which are connected to that are refrained from using the same color.
- We have some set of color C, then initially all colors are available to all the vertices.
- We start assigning color o vertices, the number of available colors to the remaining vertices would also start reducing depending on the existence of edges between vertices.

Refer Example in class notes

Algorithm:

Algorithm mColoring(k)

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

```
{
repeat
{
   // generate all legal assignment for X[k].
  Nextvalue(k); // Assign to X[k] a legal color.
        If (X[k]=0) then return;          // No new color possible.
       If (k=n) then                  // Almost „m‟ colors have been used to color the „n‟ vertices
                Write(x[1:n]);
     Else mcoloring(k+1);
}until(false);
}
```

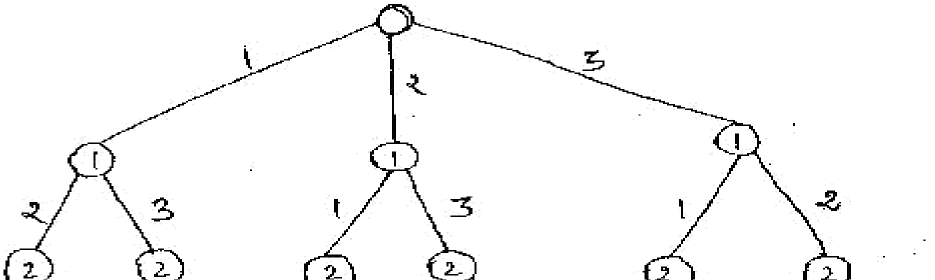
Algorithm Nextvalue(k)

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

State Space Tree:



7. Explain Knapsack Proble m using Backtracking (UQ APRIL’13 & NOV’12)

- 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,

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

Xi → Constitute Zero-one valued Vector.

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

- Bounding functions are needed to help kill some live nodes without expanding them. A good bounding function for this problem is obtained 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 ≥P(I+1) / W(I+1)

Algorithm:

Algorithm Bknap(k,cp,cw)

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

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

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

```
   }

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

Algorithm for Bounding function:

```
Algorithm Bound(cp,cw,k) //
cp → current profit total. //cw →
current weight total.
//k → the index of the last removed item.
//m → the knapsack size.
{
    b=cp;
    c=cw;
    for I =- k+1 to n do
  {
        c= c+w[I];
      if (c<m) then b=b+p[I];
          else return b+ (1-(c-m)/W[I]) * P[I];
}
return b;
}
```

Refer Example in class notes

Pondicherry University Questions

2 Marks

1 What are the requirements that are needed for performing Backtracking? (UQ NOV‟10) (Qn.No.1)
2. State Sum of Subsets problem. (UQ APRIL‟13 & APRIL‟12,NOV‟14) (Qn.No.14)
3. State m – colorability decision problem (UQ NOV‟10) (Qn.No.15)
4. Define chromatic number of the graph. (UQ APRIL‟13) (Qn.No.16)
5. Explain Hamiltonian cycles? (UQ NOV‟12 & APRIL‟12,APRIL/MAY‟14) (Qn.No.18)
6. Define Backtracking (UQ APRIL‟12,APRIL/MAY‟14) (Qn.No.19)
7. Define state space tree. (UQ:NOV‟14) (Qn.No.4)

11 Marks

1. Explain general Backtracking method (UQ NOV‟12) (Qn.No.1)
2. Explain 8-Queens Problem? (UQ APRIL‟13 & APRIL „12,APRIL/MAY‟14) (Qn.No.2)
3. Explain Hamiltonian cycles? (UQ NOV‟10,NOV‟14) (Qn.No.4)
4. Explain graph coloring (UQ APRIL‟12,APRIL/MAY‟14,NOV‟14) (Qn.No.6)
5. Explain Knapsack Problem using Backtracking (UQ APRIL‟13 & NOV‟12) (Ref.Pg.No.12 Qn.No.7)

UNIT V

Branch and Bound Method: Least Cost (LC) search–the 15-puzzle problem–control abstractions for LC-Search – Bounding – FIFO Branch-and-Bound - 0/1 Knapsack problem – traveling salesman problem. Introduction to NP-Hard and NP-Completeness.

2 Marks

1. Define Branch-and-Bound method. (UQ APRIL'13 & APRIL'12)

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

2. What are the searching techniques that are commonly used in Branch-and-Bound Method?

The searching techniques that are commonly used in Branch-and-Bound method are:

i. FIFO
ii. LIFO
iii. LC
iv. Heuristic search

3. What are NP- hard and Np-complete problems? (UQ NOV'12)

The problems whose solutions have computing times are bounded by polynomials of small degree.

4. What is a decision proble m?

Any problem for which the answer is either zero or one is called decision problem.

5. What is maxclique problem?

A maxclique problem is the optimization problem that has to determine the size of a largest clique in Graph G where clique is the maximal sub graph of a graph.

6. What is approximate solution?

A feasible solution with value close to the value of an optimal solution is called approximate solution.

7. Explain Least Cost (LC) Search (UQ APRIL'13)

In LIFO Method, the selection (generation) of next E-node is a rigid one. The next E-node is selected in terms of level by level.

8. FIFO Branch and Bound (UQ NOV'12)

This problem begins with upper = $\propto$ as an upper bound on the cost of minimum-cost answer node.,whenever the node is generated in the tree, C'(x) and u(x) are computed. Each time a new answer node is found, the value of upper can be updated by the value of u(x)

9. Differentiate backtracking and branch-and-bound. Techniques (UQ APR'11)

Backtracking Branch-and-bound

State-space tree is constructed using depth- first search State-space tree is constructed using best- first search Finds solutions for combinatorial non-optimization problems Finds solutions for combinatorial optimization problems.No bounds are associated with the nodes in the state-space tree Bounds are associated with the each and every node in the state-space tree

10. Compare the FIFO and LIFO search.

- BFS is an FIFO search in terms of live nodes
- List of live nodes is a queue
- DFS is an LIFO search in terms of live nodes
- List of live nodes is a stack

11. When can a search path be terminated in a branch-and-bound algorithm?

A search path at the current node in a state-space tree of a branch-and-bound algorithm can be terminated if

➢ The value of the node‘s bound is not better than the value of the best solution seen so far

The node represents no feasible solution because the constraints of the problem are already violated

➢ The subset of feasible solutions represented by the node consists of a single point in this case compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

12. What is best-first branch-and-bound?

It is sensible to consider a node with the best bound as the most promising, although this does not preclude the possibility that an optimal solution will ultimately belong to a different branch of the state-space tree. This strategy is called best-first branch-and-bound.

13. What is knapsack proble m?

Given n items of known weights wi and values vi, i=1,2,…,n, and a knapsack of capacit y W, find the most valuable subset of the items that fit the knapsack. It is convenient to order the items of a given instance in descending order by their value-to-weight ratios. Then the first item gives the best payoff per weight unit and the last one gives the worst payoff per weight unit

14. What are the strengths of backtracking and branch-and-bound?

The strengths are as follows

✓ It is typically applied to difficult combinatorial problems for which no efficient algorithm for finding exact solution possibly exist

✓ It holds hope for solving some instances of nontrivial sizes in an acceptable amount of time

✓ Even if it does not eliminate any elements of a problem‘s state space and ends up generating all its elements, it provides a specific technique for doing so, which can be of some value.

15 Define SATISFIABILITY.

Let x1, x2, x3….,xn denotes Boolean variables.

Let xi denotes the relation of xi.

A literal is either a variable or its negation.

A formula in the prepositional calculus is an expression that can be constructed using literals and the operators and Λ or v.

A clause is a formula with at least one positive literal. The satiability problem is to determine if a formula is true for some assignment of truth values to the variables.

16. What are the two classes of non polynomial time problems?

- NP? hard
- NP?complete

17. Define COOK'S theore m.

The satiability problem is to determine if a formula is true for some assignment of truth values to the variables.Satisfiability is in P if and only if P = NP.

18. Give the mathematical equation for 0/1knapsack problem. (UQ APR'11)

A simple way to find the upper bound _ub' is to add _v', the total value of the items already selected, the product of the remaining capacity of the knapsack W-w and the best per unit payoff among the remaining items, which is vi+1/wi+1 ub = v + (W-w)(vi+1/wi+1)

19.What is bounding function? (UQ:APRIL/MAY'14)

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

20.what is non deterministic algorithm (UQ:APRIL/MAY'14,NOV'14)

A conceptual *algorithm* with more than one allowed step at certain times and which always takes the right or best step. It is not random, as in *randomized algorithm*, or indeterminate. Rather it has the supercomputational characteristic of choosing the optimal behavior.

21.What is the use of ranking function (UQ:NOV'14)

The basis of Branch and Bound algorithms is a ranking function . The ranking function assigns a value to each node in the graph. At each step, a branch and bound algorithm uses the ranking function to decide which node to expand next

11 Marks

1. Explain Branch and bound method (UQ APRIL'12 & APRIL'13,APRIL/MAY'14)

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

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

Solution states are those problemstates‘s‘for which the path from the rootto‘s‘defines a tuple in thesolution space. The leaf nodes in the combinatorial tree are the solution states.

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

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

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

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

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

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

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

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

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

A ***D-search*** (depth search) state space search will be called LIFO (Last In First Out) search, as the list of live nodes is a list- in-first-out list (or stack). Bounding functions are used to help avoid the generation of sub trees that do not contain an answer node. The branch-and-bound algorithms search a tree model of the solution space to get the solution.

2. Explain Least Cost (LC) Search with example(UQ APRIL'13)

- The next E – node is selected on the basis of an intelligent ranking function c^(.) for live node.
- The ideal way to assign ranks would be on the basis of the additional computational effect to reach an answer node from the live node.
- We use ranking function to identify answer node speedily.
- Assigning rank to some live node require extra cost.
- Extra cost for node x is denoted by no. of nodes competing number of level the nearest answer node in the sub tree.

- Rank the live nodes by using a heuristic C^(.)
- The enxt E – node is selected on the basis of this ranking function.
- For any node x , the cost would be given by
    - Number of nodes in a subtree x that need to be generated before an answer node reached.Search will always generate the minimum no. of nodes.
    - Number of levels to the nearest answer node in subtree x.
        - C^(root) for 4 queen problem is 4
        - The only nodes on the path from root to nearest answer node.
- Let g^(x) be an estimate of the additional effect needed to reach an answer node from node x.
- X is assigned a rank using a function C^(.) such that
- C^(.)=f(n(x))+g^(x)
  Where n(x)-cost of reaching x from root
  F(.) – non decreasing function
- If x is an answer node, c(x) is cost of reaching x from root of space tree
- If x is not answer node c(x)=α provide the subtree x contains no answer node.
- If subtree x contains answer node c(x) is the cost of a minimum cost answer node in subtree x.

EXAMPLE:

Refer Class notes.

3. Explain Control Abstraction of Least Cost (LC)

Control abstraction for LC search

Let
t - state space tree
c() - cost function for the nodes in t
if x is a node in t, then c(x) is the minimum cost of any answer node in the sub tree with root x
Thus, c(t) – cost of a minimum cost answer node in t

We cannot find c(.) easily, so heuristic C‘() is used

```
listnode = record
{
listnode *next, *parent; Float cost;
}
```

```
Algorithm LC Search(t)
\\ Search t for an answe r node
{
If *t is an answer node then output *t and return; E
= t ; \\ E- node
Initialize the list of live nodes to be empty;
Repeat
{
For each child x of E do
{
                    If x is an answer node the output the
                            path from x to t and return
                            Add(x) ; \\ x is a new live node.
                            (x->parent):=E;  \\Pointer for path to root.
            }
            If there are no more live nodes then
            {
                    Write(―No answer  node‖); return;
```

```
                }
           E=Least(); } until (false);
}
```

- The LC Search algorithm uses c‘ to find an answer node
- Two function Add(x) and Least() are used
  - Add(x) -> Adds new live node x to the list of live nodes
  - Least() -> finds a live node with least c‘(). This node is deleted form the list of live nodes and returned.
- It outputs the path from the answer node it finds to the root node t
  - If each x becomes live node , then associate a field parent which gives the parent of node x
  - When answer node g is found, the path from g to t can be determined.

    g - answer node
    x - live node
    t - root node

Algorithm explanation:-

1) Variable E point the current E- node
2) Initially root node t is the first E-node ie E = t ;
3) Initially the list of live nodes are empty
4) For loop examines all the children of the E-node
   a. If x (one of the children) is answer node then output the path from x to t and exit
   b. If x is not a answer node then x becomes a live node and add it to the list of live nodes. Its parent pointer is set to E
5) When all the children are generated, E-node becomes a dead node
6) If there are no more live nodes then print no answer node and exit else call Least() to choose the next E-node
7) Repeat the steps 4 and 6 till if any answer node is found or if entire spacetree has been searched

Example

15 Puzzle Problems

- the 15 puzzle consists of 15 numbered tiles on a square frame with a capacity of 16 tiles
- Initial arrangement of the tiles are given, the objective of the problem is to transform this arrangement into the goal arrangement through a series of legal moves.
- The legal moves are ones in which a tile adjacent to the empty spot is moved to Empty Spot
- From the initial arrangement, four moves are possible
- We can move any one of the tiles numbered 2,3,5,6 to the empty spot
- These arrangements are called states of the puzzle
- To speed up the search, we can associate the cost c(x) with each node x in the state space tree.
- C‘(x)=f(x)+g‘(x) where f(x) – length of the path from root to node x

  g‘(x) – an estimate of the length of a shortest path from x to a goal node in the subtree with root x

  (or)

  g‘(x) – number of nonblank tiles not in their goal position

REFER CLASS NOTES FOR EXAMPLE

REFER CLASS NOTES FOR ALGORITHM

3. Explain Bounding function

- A branch -and-bound searches the state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node.
- We assume that each answer node x has a cost c(x) associated with it and that a minimum-cost answer

node is to be found. Three common search strategies are FIFO, LIFO, and LC.

- A cost function $\hat{c}$ (.) such that $\hat{c}$ (x) <=c(x) is used to provide lower bounds on solutions obtainable from any node x.
- If **upper** is an upper bound, then all live nodes x with $\hat{c}$ (x)>upper may be killed as all answer nodes reachable from x have cost c(x)>=c'(x)>upper6
- The starting value for upper can be set to infinity.
- Each time a new answer node is found, the value of upper can be updated
- C'(x) and upper and c(x) can be computed using formulas

Example

Job Sequencing Proble m

Given n jobs and one processor
Each job i is defined as 3 tuples(P_i,d_i,t_i)
t_i - processing time in units
d_i - deadline
P_i - penalty, if the job I is not completed within the deadline, then penalty P_i is incurred

Objective of the problem

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

Let us assume that n=4 (P1,d1,t1) = (5,1,1)
(P2,d2,t2) = (10,3,2)
(P3,d3,t3) = (6,2,1)
(P4,d5,t4) = (3,1,1)

- solution space for this instance consist of all possible subsets of the job index set (1,2,3,4)

  job index : 1,2,3,4
  solution space : all possible subset of job index ie {1,2,3} {2,3} {1,4} {1,3,4} ,……..

- solution space can be organized into a tree by means of two formulations
  1. variable tuple size formulation
  2. fixes tuple size formulation

Variable tuple size formulation

○ Node – answer node

□ Node - infeasible subsets

Compute c(x) using the following rule if x is a leaf node,

c(x) – summation of penalty of the jobs which are not in the path from root node to any node x

if x is a root node

c(x) - minimum penalty corresponding to any node in the subtree within root x
(or)
c(x)=min{c(child-1),c(child-2),….c(child-n) ,where x is node in subtree

c(x) - ∝ for □ node

1. FIFO Branch and Bound

- FIFO branch and bound is a state space tree in which list of live node is a first in first out list. The unexpected nodes are stored in queue.

- ➢ This problem begins with upper = ∝ as an upper bound on the cost of minimum-cost answer node.
- Whenever the node is generated in the tree, C‘(x) and u(x) are computed.
- Each time a new answer node is found, the value of upper can be updated by the value of u(x)
- For any node, if c‘(x)>upper then the node will be killed

REFER CLASS NOTES FOR EXAMPLE

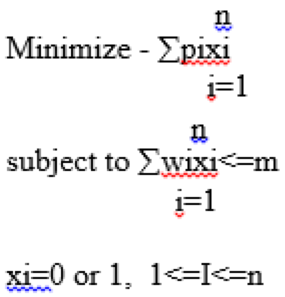
2.LIFO Brach and Bound

The LIFO branch and bound is a DFS like state space tree in which the list of live node is last in first out list.The unexpected node are stored in stack.

REFER CLASS NOTES FOR EXAMPLE

3. Explain Knapsack proble m (UQ NOV’10,NOV’14)

- To use branch and bound technique to solve any problem, it is first necessary to conceive of a state space tree for the problem.
- Branch and bound technique is used to solve minimization problem(eg 15 puzzle, 4 queen, node with minimum cost is selection criteria)
- The branch and bound technique can not be applied directly in the knapsack problem , because the knapsack problem is a maximization problem(eg maximum profit is the selection criteria)
- This difficulty is easily overcome by replacing the objective function ∑PiXi by the function -∑PiXi
- The modified knapsack problem is stated as



$$\text{Minimize } -\sum_{i=1}^{n} pixi$$

$$\text{subject to } \sum_{i=1}^{n} wixi<=m$$

$$xi=0 \text{ or } 1,\ 1<=I<=n$$

Two types of formulation

Fixed tuple size formulation
Variable tuple size formulation

Fixed tuple size formulation

- Every leaf node in the state space tree representing an assignment for which

$$\sum_{1<=I<=n} wixi<=m \text{ is an answer node}$$

- all other leaf nodes are infeasible
- for minimum cost answer node to correspond to any optimul solution, we need to define c(x)=
- $-\sum_{1<=I<=n}$ pixi for every answer node x
- the cost c(x)= α for infeasible leaf nodes
- there is methods to solve the knapsack problem

LC Branch And Bound

REFER CLASS NOTES FOR EXAMPLES

4. Explain travelling salesman problem (UQ NOV’12, NOV’10 & APRIL’12)

INTRODUCTION:

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

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

STEPS INVOLVED IN THIS PROCEDURE ARE AS FOLLOWS:

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

STEP 1: [*ROW REDUCTION]*
For all rows do step 2

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

STEP 3: [COLUMN REDUCTION]
Use cost matrix- Row reduced one for all columns do STEP 4.

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

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

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

STEP 7: Calculate effective cost of the edges. $\sum$ (i, j)=least cost in the ith row excluding (i, j) + least cost in the j th column excluding (i, j).

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

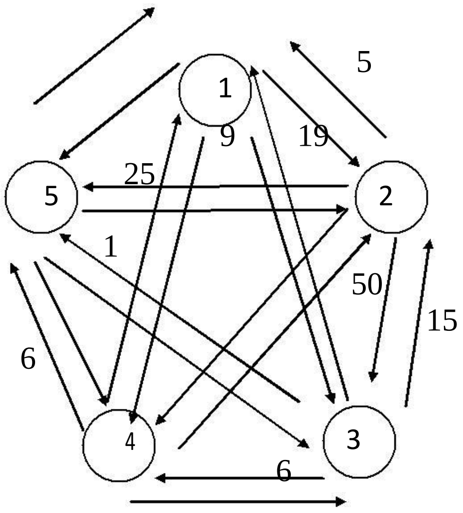
STEP 9: Delete (i, j) means delete ith row and jth column change (j, i) value to infinity. (Used to avoid infinite loop formation) If (i,j) not present, leave it.

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

STEP 11: Use preserved cost matrix Cn, Cn-1… C1
Choose an edge [i, j] having value =0, at the first time for a preserved matrix and leave that matrix.

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

EXAMPLE: *Given graph G*



MATRIX:

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

PHASE I

STEP 1: Row Reduction C

C1 [ROW REDUCTION:

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

STEP 3: C1 [Column Reduction]

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

STEP 5:

Preserve the above in C1,

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

STEP 6:

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

STEP 7:

Calculation of effective cost
[E.C] (1,2) = 2+1 =3 (2,1) = 12+3
= 15 (3,5) = 2+0 =2 (4,5) = 3+0 =
3 (5,3) = 0+12 = 12 (5,4) = 0+2 =
2

STEP 8:

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

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

Now Cost Matrix =

	2	3	4	5
1	α	15	3	2
3	14	α	2	0

4	44	18	α	0
5	1	0	0	α

STEP 10: The Cost matrix ≠ 2 x 2.
Therefore, go to step 1.

PHASE II:

STEP1:C2(R, R)

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

STEP 3: C2 (C, R)

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

STEP 5: Preserve the above in C2

C2 =

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

STEP 6:

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

STEP 7: calculation of E.C.

(1,5) = 1+0 =1

(3,5) = 2+0 =2
(4,5) = 18+0 =18
(5,2) = 0+13 =13
(5,3) = 0+13 =13
(5,4) = 0+1 =1

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

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

Now, cost matrix

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

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

<u>PHASE III:</u>

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

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

<u>STEP 3: C3 (C, R)</u>

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

STEP 5: preserve the above in C2

<u>STEP 6:</u> L={(1,4), (3,4), (5,2), (5,3)}

STEP 7: calculation of E.C
(1,4)=12+0=12
(3,4)=11+0=11
(5,2)=0+11=11
(5,3)=0+12=12

<u>STEP 8:</u> Here we are having two edges (1,4) and (5,3) with cost = 12. Hence arbitrarily choose (1,4)

<u>STEP 9:</u> Delete (i,j) → (1,4) and make change in it (4,1) = α if exists.

Now cost matrix is

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

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

C4 (RR)=

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

C4 (C, R) =

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

Therefore,C4 =

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

STEP 11: LIST C1, C2, C3 AND C4

C4

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

C3

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

C2 =

$$\begin{array}{c|c|c|c|c|} & 2 & 3 & 4 & 5 \\ \hline 1 & \alpha & 13 & 1 & 0 \\ \hline 3 & 13 & \alpha & 2 & 0 \\ \hline 4 & 43 & 18 & \alpha & 0 \\ \hline 5 & 0 & 0 & 0 & \alpha \\ \end{array}$$

C1 =

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

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

STEP 12:

i) Use C4 =

	2	3
3	0	α
5	0	0

Pick up an edge (I, j) =0 having least index

Here (3,2) =0

Hence, T ← (3,2)

Use C3 =

	2	3	4
1	α	12	0
3	11	α	0
5	0	0	α

Pick up an edge (i, j) =0 having least index

Here (1,4) =0

Hence, T ← (3,2), (1,4)

Use C2=

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

Pick up an edge (i, j) with least cost index.

Here (1,5) → not possible because already chosen index i (i=j) (3,5) → not possible as already chosen index.

(4,5) → 0

Hence, T ← (3,2), (1,4), (4,5)

Use C1 =

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \alpha & 0 & 15 & 3 & 2 \\ 2 & 0 & \alpha & 12 & 22 & 20 \\ 3 & 18 & 14 & \alpha & 2 & 0 \\ 4 & 3 & 44 & 18 & \alpha & 0 \\ 5 & 15 & 1 & 0 & 0 & \alpha \end{array}$$

Pick up an edge (i, j) with least index

(1,2) → Not possible

(2,1) → Choose it

HENCE T ← (3,2), (1,4), (4,5), (2,1)

SOLUTION:

From the above list

3—2—1—4—
5

This result now, we have to return to the same city where we started (Here 3).

Final result:3—2—1—
4—5—3

Cost is 15+15+31+6+7=64

6. Explain NP hard and NP-completeness (UQ APRIL'13, NOV'12 & APRIL'12 ,APRIL/MAY'14,NOV'14)

The field of complexity theory deals with how fast can one solve a certain type of problem.We assume problem in solvable.

Basic Concepts:

Algorithm contains operations whose outcome are uniquely defined.

Definition of non deterministic algorithm:

Algorithm contains operations whose outcomes are not uniquely defined but are limited to specified set of possibilities.

To specify non deterministic algorithm., 3 functions are needed.

1. Choice
2. Failure
3. Success

Choice(S)

Arbitrarily chosen one of the elements of set S

Failuer()

It signals an unsuccessful operation

Success()

It signals a successful completion.

Decision Problem:

Any problem for which the answer is either 0 or 1 is called a decision problem.

Example of 0/1 knapsack Problem:

The Knapsack decision problem is to determine whether there is 0/1 assignment of values to x_i, 1<=i<=n such that £p_i x_i>=r and £W_i x_i<= m
Where r is a given number
P_i W_i are non negative numbers

Optimization Problem:

Any problem that involves the identification of an optimal (either maximum or minimum) value of a given cost function is known as optimization problem.

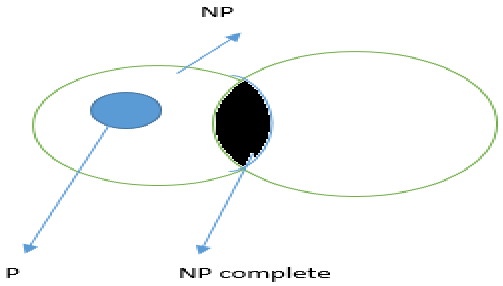
Polynomial Transformation

Problem Transformation: some algorithms which take a decision problem X (or rather ANY instance of theproblem of type X), and output a corresponding instance of the decision problem of type Y, in such a way that if the input has answer True, then the output (of type Y) is also True and vice versa. [REMINDER: Problem
≡ Input; & Solution ≡ Algorithm that takes any input and produces correct output.]
For example, you can write a problem transformation algorithm from 3-SAT problem to 3D-Matching problem
Note that the problem transformations are directed.
When a problem transformation algorithm is polynomial- time we call it a **polynomial transformation.**
Existence of a polynomial transformation algorithm has a great **significance** for the complexity issues.

Suppose you have (1) a poly- transformation A_{xy} exists from a (source) problem X to another (target) problem Y, and (2) Y has a poly algorithm P_y , then you can solve any instance of the source problem X polynomially, by the following method.
Just transform any instance of X into another instance of Y first using A_{xy}, and then use Y's poly-algorithm P_y. Both of these steps are polynomial, and the output (T/F) from Y's algorithm is valid for the source instance (of X) as well. Hence, the True/False answer for the original instance of P_y (A_{xy} (X)) will be obtained in poly- time. This constitutes an indirect poly-algorithm for X, thus making X also belonging to the P-class.
Note, $|A_{xy}(X)|$ is polynomial with respect to |X|.

Classes of NP hard and NP completeness:
Class P-definition:
P is the set of all decision problems solvable by deterministic algorithm in polynomial time.
Class NP-Definition:
NP is the set of all decision problems solvable by non deterministic algorithms in polynomial time.
Definition of reducibility:
Let L1 and L2 are problems.Problem L1 reduces to L2 if and if only if there is a way to solve L1 by a deterministic polynomial algorithm using a deterministic algorithm that solves L2 in polynomial time.
NP Hard definition:
A problem L is NP hard if and if only if atisfiability reduces to L.
NP-Complete definition

A problem L is complete if and if only if L is NP hard and L ε NP
Common relationship among P, NP, NP complete and NP hard Problems



It is easy to see that there are NP hard problems that are not NP complete
NP complete is a subset of NP , the set of all decision problems, whose solution can be verified in polynomial time.

<u>**Cook's theorem.**</u>

Cook modeled all NP-problems (an infinite set) to an abstract Turing machine. Then he developed a poly-transformation from this machine (i.e., all NP-class problems) to a particular decision problem, namely, the Boolean Satisfiability (SAT) problem.

Significance of Cook's theorem: if one can find a poly-algorithm for SAT, then by using Cook's poly-transformation one can solve all NP-class problems in poly-time (consequently, P-class = NP-class would be proved).

SAT is the historically first identified NP-hard problem!

Further significance of Cook's theorem: if you find a poly-transformation from SAT to another problem Z,then Z becomes another NP-hard problem. That is, if anyone finds a poly algorithm for Z, then by using your poly-transformation from SAT-to-Z, anyone will be able to solve any SAT problem- instance in poly-time, and hence would be able to solve all NP-class problems in poly- time (by Cook‵s theorem).

These problems, which have a chain of poly-transformation from SAT, are called **NP-hard problems**.

If an NP-hard problem also belongs to the NP-class it is called an NP-complete problem, and the group of such problems are called **NP-complete problems**.

Circuit Satisfiability

Given a Boolean circuit, if there is an assignment of Boolean values to the input make the output true.

Satisfiability

Given a Boolean formula, does these exixt a truth assignment to the variable to make the expression true.

- The goal of verification algorithm is verify a "yes" answer to a decision problem's input
- A verification algorithm takes a problem instances x and answer "yes". If there exists a certificate y such that answer for x with certificate y is "yes"

Example for NP complete problem:

1. **Travelsalesman Problem:**
   For each 2 cities an integer cost is given to travel from one of the cities to the another.
   The salesman wants to make a minimum cost circuit visiting each city exactly once.

2.Circuit Satisfiability:

Take a Boolean circuit with a single output node an ask whether there is an assignment of values to the circuit, input . So that the output is 1.

Logic Gates are examples

3.Class Scheduling problem:

N teachers with certain hour restrictions M classes to be scheduled.

We can Schedule all classes

Make sure that no 2 teachers teach the same class at same time

No teacher is scheduled to teach 2 classes at once.

Backtracking:

Effective for decision problem

Systematically traverse through possible paths to locate solution or dead ends.

A the end of the path algorithm left with(x,y) pair, x is remaining subproblem and y is set of choices to get x.

Branch and Bound

Effective for optimization problem.

Extended to backtracking

Instead of stopping once a single solution is found, continuous searching until the best solution is found.

Pondicherry University Questions

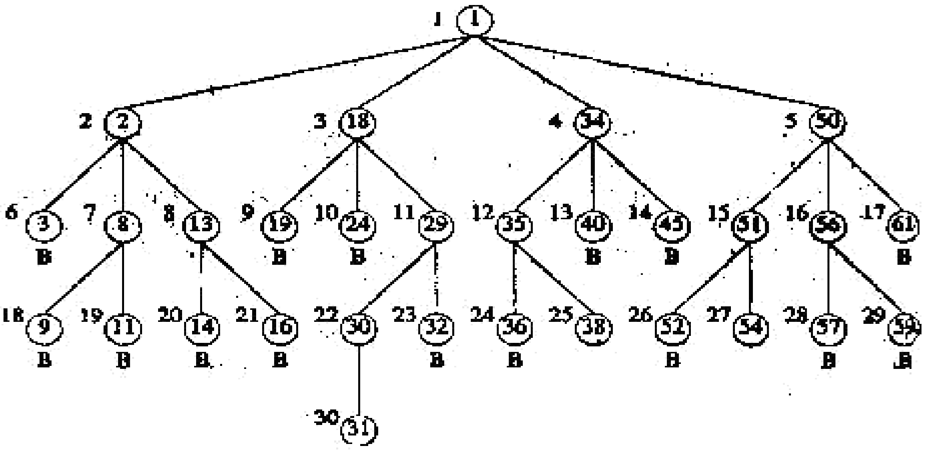
2 Marks

1 Define Branch-and-Bound method. (UQ APRIL‘13 & APRIL‘12) (Qn.No.1)
2. What are NP- hard and Np-complete problems? (UQ NOV‘12) (Qn.No.3)
3. Explain Least Cost (LC) Search (UQ APRIL‘13) (Qn.No.7) 4 .FIFO
Branch and Bound (UQ NOV‘12) (Qn.No.8)
5. Differentiate backtracking and branch-and-bound. Techniques (UQ APR‘11) (Qn.No.9)
6. Give the mathematical equation for 0/1knapsack problem. (UQ APR‘11) (Qn.No.18)
7. What is bounding function? (UQ:APRIL/MAY‘14) (Qn.No.19)
8. what is non deterministic algorithm (UQ:APRIL/MAY‘14,NOV‘14)) (Qn.No.20)
9. What is the use of ranking function (UQ:NOV‘14) (Qn.No.21)

11 Marks

1. Explain Branch and bound method (UQ APRIL‘12 & APRIL‘13,APRIL/MAY‘14) (Qn.No.1)
2. Explain Least Cost (LC) Search with 15 puzzle example(UQ APRIL‘13) (Qn.No.2)
3. Explain Knapsack problem (UQ NOV‘10,NOV‘14) (Qn.No.4)
4. Explain travelling salesman problem (UQ NOV‘12, NOV‘10 & APRIL‘12) (Qn.No.5)
5. Explain NP hard and NP-completeness (UQ APRIL‘13, NOV‘12 & APRIL‘12 ,APRIL/MAY‘14,NOV‘14) (Qn.No.6)

15 PUZZLE GRAPH


1
1
2
2
3
18
4
34
5
50
6
3
B
7
8
8
13
9
19
B
10
24
B
11
29
12
35
13
40
B
14
45
B
15
51
16
56
17
61
B
18
9
B
19
11
B
20
14
B
21
16
B
22
30
23
32
B
24
36
B
25
38
26
52
B
27
54
28
57
B
29
59
B
30
31