Semester-IV

Design and Analysis of Algorithms

B.E./B.TECH. DEGREE EXAMINATION, MAY/JUNE 2012

Fourth Semester

Computer Science and Engineering

CS 2251/141401/CS 41/CS 1251/10144 CS 402/080230013 – DESIGN AND ANALYSIS OF ALGORITHMS
(Common to Information Technology)
(Regulation 2008)

Time: Three hours **Maximum: 100 marks**

Answer All Questions

PART A – (10 × 2 = 20 marks)

1. Define Big 'Oh' notation.
2. What is meant by linear search?
3. What is the draw back of greedy algorithm?
4. What is the time complexity of binary search?
5. State principle of optimality.
6. Differentiate greedy method and dynamic programming.
7. What is the difference between explicit and implicit constraints?
8. Define the basic principles of Back tracking.
9. What is an articulation point in a graph?
10. What is the difference between BFS and DFS methods?

PART B – (5 × 16 = 80 marks)

11. (a) Discuss in detail all the asymptotic notations with examples.

Or

(b) With a suitable example, explain the method of solving recurrence equations.

12. (a) Explain divide – and – conquer method with merge sort algorithm. Give an example.

Or

(b) Explain how greedy method can be applied to solve the Knapsack problem.

13. (a) With a suitable example, explain all – pair shortest paths problem.

Or

(b) How is dynamic programming applied to solve the travelling sales-person problem? Explain in detail with an example.

14. (a) Explain the algorithm for finding all m-colorings of a graph.

Or

(b) Write down and explain the procedure for tackling the 8 – queens problem using a backtracking approach.

15. (a) Discuss in detail about the biconnected components of a graph.

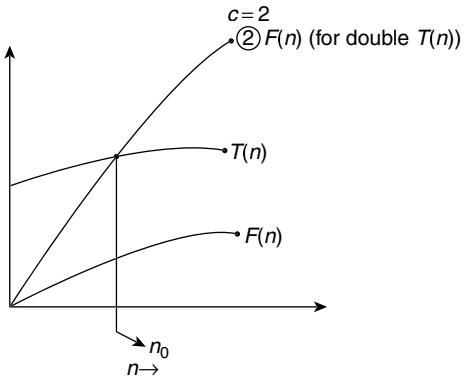
Or

(b) Compare and contrast FIFO and LC branch – and – bound search techniques.

Solutions

Part A

1. Big-Oh Notation:
Let $T(n)$ = function on n =1, 2, 3 … [usually, the worst case running time of an algorithm].
$T(n) = 0$ ($f(n)$) iff eventually (for all sufficient large n)
$T(n)$ is bounded above by a constant multiple of $f(n)$.



In the above diagram $T(n) = 0(f(n))$
Formal definition :- $T(n) = 0(f(n))$ if and only if the re exists constant c, $n_o > 0$ such that $T(n) \le c, f(n)$ for all $n \le n_0$[c, n_0 cannot depend on n].

2. Linear search or sequential search is a method for finding a particular value in a list, that consists of checking every one of its elements, one and at a time until the desired one is found.

For a list with n items, the best case is when the value is equal to the first element of the list, in which only one comparison is needed. The worst case is when the value is not in the list, in which case n comparisons are needed.

If the value being sought occurs k-times in the list, and all orderings of the list are equally likely, the expected number of comparison is

$$\begin{cases} n & \text{if } k = 0 \\ \dfrac{n+1}{k+1} & \text{if } 1 \le k \le n \end{cases}.$$

3. Draw back of greedy algorithm:
For many other problems, greedy algorithms fail to produce the optimal solution, and may even produce the unique worst possible solution. One ex-

ample is the traveling salesman problem. For each number of cities there is an assignment of distances of between the cities for which the nearest neighbor heuristic produces the unique worst possible tour.

For E.g.,:- Imagine the coin with 25-cent, 10-cent and 4-cent coins The greedy algorithm could not make changes for 41 cents, since after committing to use one 25-cent coin and one 10-cent coin it would be impossible to use 4-cent coins for the balance for 6-cents, whereas a person or a more sophisticated algorithm could make change for 41-cents one 25-cent coin and four 4-cent coins.

4. Time complexity of Binary search algorithm

Best case: Key is in the middle of the array

run time = 1 loop.

Worst case : key is not in the array.

Pare down array to size 1 by having the array m times

$$n, \frac{n}{2}, \frac{n}{2^2}, \ldots \frac{n}{2^m}$$

Worst case time complexity in binary search is 0 (log n).

Average time complexity.

Let $C \sum 2^i$

$$C = \log \frac{n}{2} = \left(\frac{l}{n}\right) * (\log n) / 2 * (n-1)$$
$$= 0(\log n)$$

5 Principle of Optimality:

To use dynamic programming, the problem must observe the principle of optimality, that whatever the initial state is, remaining decisions must be optimal with regard the state following from the first decision.

Combinatorial problems may have this property with too much of memory/time used to be efficient

e.g.,

Let $T(i, j_1, j_2, \ldots, j_k)$ be the cost of the optimal tour for i to 1 that goes through each of the other cities once

$$T(i, i_1, j_2 \ldots, j_i) = \min 1 \leq m \leq k [C(i, j_m) T(j_m, j_1, j_2 \ldots j_k)$$
$$T(i, j) = C(i, j) + C(j, i)$$

Here may subset for $j_{1,} j_2 \ldots j_k$ instead of any subinterval–hence exponential.

Still with other ideas it can be effective for combinatorial search.

6. Difference between dynamic programming & Greedy algorithm:

Dynamic Programming	Greedy Algorithm
• Dynamic programming will solve each of them once and their answers are stored in future reference	• Greedy algorithm is easier than dynamic programming
• Bottom up approach	• Top down approach
• Optimal solutions are guaranteed in dynamic programming	• It is not guaranteed in • greedy algorithm
• Dynamic programming provides solutions for some problem with a brute force approach would be very slow.	• But Greedy algorithm • finds tough to solve there • problems.
• Principal of optimality is used in the problems.	• Here optimal greedy or • greedy algorithms are used.

7. Difference between implicit and explicit constrains:

Implicit Constrains	Explicit Constrains
• Implicit constrains are constrains that will be satisfied by the manner in which the branch-and-bound algorithm, is constructed.	• Explicit constrains are constrains that will require procedures for regognition as a integral path of branch and bound
• In the resource, constrained project scheduling problems, the precedence constrains are example of implicit constrains.	• The resource constrains are clearly example for explicit constrain
• These constrain can easily satisfied by branch and bound solution method.	• But this is linear programming model, difficulty occurs when a branch & bound method is chosen.

8. Basic principles of back-tracking:

The problems backtracking helps with case of the form: If possible, find a solution of the following format that satisfies the following constraints. Example in the n-queen problem, the solution is a placement of queens on the chess board. One format for presenting such a solution is: an array S, indexed by rows, with values from 1 to n. $S(i)$ is the coloumn where there is a queen in the i^{th} row. The constrains would be: no two queens are in same coloumn. ($S(i) \neq S(i')$ for each $i \neq i'$) or in the same diagonal : $S(i) - S(i') \neq i - i'$, $i' - i$ for many $i \neq i$

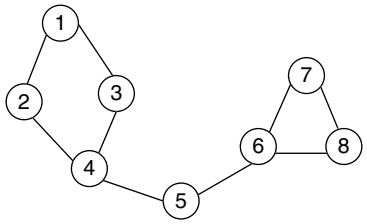
Basic steps:

- View finding a solution as making a series of choice. The format for candidate solution is a big clue as to how to do this .E.g., queens, the end are rows.

- For each row, must choose where to put the queen.
- Put a single choice and do analysis of option in queen example, for each row we have to choose a place to put the queen out of *n* possible places.
- In some order, try each option, if it succeeds proceed otherwise. "back-track" and try next option

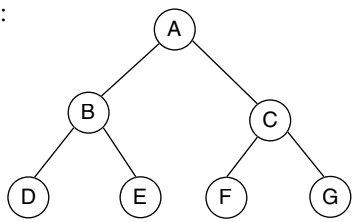
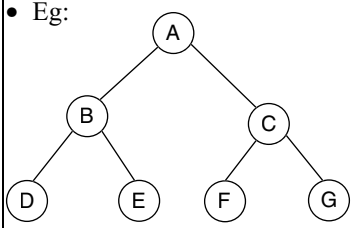
9. Aurticulation point :-

Any connected graph decomposes into a tree of biconnected components called the block tree of the graph. The blocks are attached to each other at shared vertices called cut vertices or aurticulation points. An aurticulation point is that any vertex that when removed increases the no: of connected components



5 is an aurticulation point the above graph is a connected graph if 5 is removed from the graph it gives 2 connected components.

10. Difference between Breath first and Depth first search:

Breath First	Depth First
• Breath First search is a carefull search where you equally progress in all possible directions.	• Depth first is a more aggressive and risk taking search where you choose one path and ignore all the other path until you reach the end of your chosen path. If there is no end to that path continuation is done.
• It is optimal and complete.	• DFS is neither optimal nor complete.
• Eg:  The search path is A-B-C-D-E-F-G	• Eg:  The search path is A-B-D-E-C-F-G

11. (a) Asymptotic Notation:

Asymptotic Notation is used for finding the best algorithm among a group a algorithm they performing some sort of comparison between them. It has following types. They are

- Big-O Notation.
- Big Omega Notation
- Theta Notation
- Little-O-Notation
- Little Omega Notation.

Big-O-Notation

Big-O - is the formal method of expressing the upper bound of an algorithm's running time.

It could be the maximum time taken to compute the algorithm.

Non negative functions $f(n)$, $g(n)$, if there exists an integer no and a constant $c > 0$ such that for all integer $n > n_0$, $f(n) \leq c\ g(n)$ is Big O of $g(n)$.This is denoted as $f(n) = O(g(n))$.

If it is graphed, $g(n)$ serves as an upper bound to the curve you are analysing $f(n)$.

E.g.,: $f(n) = 2n + 8$ $g(n) = n^2$

$\therefore f(n) \leq c\ g(n)$

$2n + 8c = n^2$

If $n = 4$ $16 \leq 16$. In this we can say that $f(n)$ is generally faster than $g(n)$. That is $f(n)$ is bound by and will always be less than it. $f(n)$ run in $O(n^2)$ time.

Big Omega Notation:

For non negative functions $f(n)$ and $g(n)$, if there exists an integer n_o and a constant $c > 0$ such that for all integer $n > n_0$ $f(n) \geq c\ g(n)$. Then $f(n)$ is omega of $g(n)$. denoted as "$f(n) = \Omega(g(n))$". This makes $g(n)$ a lower bound function.

$$f(n) = 3n + 2 \quad g(n) = 3n.$$

$$3n + 2 \geq 3n (n > 1).$$

$$\text{if } n = 1 \quad 5 \geq 3.$$

Theta Notation:

Non negative functions $f(n)$& $g(n)$, $f(n)$ is theta of g(n) denoted as $f(n) = O(g(n))$.

This is saying that the function $f(n)$ is bounded both from the top & bottom by same function $g(n)$ also denoted as $c_1 g(n) \leq f(n) \leq c_2 g(n)$.

c_1, c_2 are constant.

$$3n+2 \geq 3n \rightarrow n \geq 2$$
$$3n+2 \leq 4n \rightarrow n \geq 2$$

Where $c_1 = 3$ $c_2 = 4$.

Little O Notation:

Non negative fun $f(n)$ & $g(n)$, $f(n)$ is little O of $g(n)$ if and only if $f(n) = O(g(n))$ but $f(n) \neq O\, g(n))$ denoted as $f(n) = O\,(g(n))$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$
$$f(n) = 4n+2 \;\; g(n) = n^2.$$
$$\lim_{n\to\alpha} \frac{4n+2}{n^2} = 0$$

Little ω notation:

Non negative function $f(n)$ & $g(n)$, $f(n)$ is little omega of $g(n)$ if and only if $f(n) = \Omega(g(n))$. But $f(n) \neq \theta\, g(n)$ denoted as $f(n) = \omega$ $(g(n))$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$
$$f(n) = 4n+2 \;\; g(n) = n^2$$
$$\lim_{n\to\infty} \frac{n^2}{4n+2} = 0$$

Asymptotic Complexity is a way of expressing the main component of the cost of an algorithm. For example, the algorithm for sorting a deck of cards, which proceeds by repeatedly searching through the deck for the lowest card. The asymptotic complexity of this algorithm is the square of the number of cards in the deck.

The first scan would involve scanning 52 cards, the next would take 51 etc. so the cost formula is 52 + 51 +…. + 2 generally letting N be the numbers of cards, the formula is 2 + …..+ N which equals.

$$((N)*(N+1)/2)-1 = ((N^2+N)/2)-1 = (1/2)N^2 \;\&\; 1/2(N)-1.$$

11. (b) Recurrence relations in the analysis of algorithm:
The recurrence relation is a data structure of size n is processed by first processing one or more piece of size $k > n$. The resulting

algorithm is naturally expressed recursively, but may be converted to an iterative version without affecting the analysis.

Let t_n standard for the worst case time to execute a algorithm with i / p size n. The divide and conquer algorithms, there's usually a natural recurrence relation for t_n

E.g.,:

Sequential search $\Rightarrow t_n = t_n + 1$

sorting $\Rightarrow t_n = t_{n-1} + n$

binary search $\Rightarrow t_n = t_n/2 + 1$

merge sort $\Rightarrow t_n = 2t_n/2 + n$

Queue a recurrence relation, the possible values for t_n may be found by first finding a characteristics polynomial. The characteristics polynomials for the first 2 examples above are respectively.

Sequential search = $(E - 1)^2$

selection sort = $(E - 1)^2$

Methods:

A recurrence relation is a functions that tells how to find the n^{th} element of a sequence in terms of one or more of its predecessor. By solving a recurrence relation by computing the function a_{n-1}, a_{n-2}, etc.

Homogeneous Linear recurrences:

The easiest type of recurrence to solve is the linear homogenous recurrence with constant coefficients. In other words, the recurrence does not have any of the following forms,

$$a^n = a^{n-1},\ a^{n-2},\ a^n = a^{n-1} + 5 \text{ or } a^n = na^{n-1}.$$

E.g.: $a^n = 2a^{n-1} + 3a^{n-2}$ and $a^0 = a^1 = 1$.

Solution:

- substitute r^n for a^n(r^{n-1} for a^{n-1}etc) and simplify. The characteristics equation is $r_n = 2r_{n-1} + 3r_{n-2}$ simplifies to
$$r^2 = 2r + 3$$
- Find the roots of characteristics equation
$$r^2 - 2r - 3 = 0.$$
factor as $(r - 3)(r + 1)$ giving roots $r^1 = 3$, $r^2 = -1$.When then are 2 distinct roots, the general form of solution is $a^n = a^1.r^{1n} + a^2.r^{2n}$
In this case we have $a_n = \alpha_1 3n + \alpha_2(-1)n$.
- We have to find the constants α_1 & α_2.
$$a^0 = 1 = \alpha^1 + \alpha^2.$$
$$a^1 = 1 = 3\alpha^1 - \alpha^2.$$

So $\alpha^1 = 1/2$ & $\alpha^2 = 1/2$ and the final solution is

$$a_n = \left(\frac{1}{2}\right)3n + \left(\frac{1}{2}\right)(-1)n$$

If a characteristics equation has equal roots (i.e., $r^1 = r^2$) then the general solution has the forms.

$$a_n = \alpha_1.r^n + \alpha_2.n.r^n$$

In homogeneous recurrence relation:

We now turn to inhomogenous recurrence relations and look at 2 distinct methods of solving them. There recurrence relations in general have the form $a_n = c^{n-1}a^{n-1} + g(n)$ where $g(n)$ is function of n. [$g(n)$ could be a constant].

One of the first examples of these recurrences usually encountered is the tower of Hanoi recurrence relations. $H(n) = 2H(n-1) + 1$ which has as it solution $H(n) = 2^n - 1$.One way to solve this is by use of backward substitution as above.

E.g.: $a^n = 3a^{n-1} + 1\ 2^n$ with $a^1 = 5$.

The homogeneous part $a^n = 3a^{n-1}$ has a root of 3, so $f(n) = a.3^n$.the particular solution should have the form $a^n = \beta.2^n$.

Substituting this into the original recurrence gives $B.2^n = 3\beta.2^{n-1} + 2^n$.

Solving this equation for β gives us the particular solution $P(n) = -2^{n+1}$.

Thus the solution has the form, $a^n = a.3^{n-1} - 2^{n+1}$.

Using the initial condition that $a' = 5$ we get $5 = 3a - 4$ which gives $a = 3$. Thus, the final solution is $a^n = 3^{n+1} - 2^{n+1}$.

12. (a) By using the divide and conquer method the sorting algorithm has the nice property that in the worst case its complexity is $0(n \log n)$. This algorithm is called merge sort.

Let a problem of function to compute on n input size divide and conquer strategy y suggest splitting the input to k sub problems where $l > k > n$. These sub problems must be solved then a method must be found to combine sub solutions into a solution of whole

```
void mergesort. (int l, int n)
{
if (l < h)
{
   int m = [(l + h) / 2];
   mergesort (l, m);
```

```
    mergesort (m + 1, h);
    merge (l, m, h);
}
}
void merge (int l, int m, int n)
{
    int d [20], int e [20];
    int a, b, c;
    a = l;
    b = l;
    c = m + 1;
    do
    {
        if(d [h] ≤ d [c])
        {
            e[b] = d[h];
            h  = h +1;
        }
        else
        {
            e [b] = d [c];
            c = c + 1;
        }
        b = b + 1;
    }} while (ch ≤ m) and (b ≤ h)
    if (h > m) then
        for (int i = 0; i < h; i++)
        {
            e[b] = d[i];
            b = b + 1;
        }
    else
        for(j = h;  j < m; j++)
        {
            e[b] = d[i];
            b = b +1;
        }
        for (i = l; i < h; i++)
        {
            d [i] = e [i];
        }
```

Example:

Consider the array of ten elements = {450, 360, 510, 312, 220, 980, 460, 540, 800, 150}
Step 1:
Split the array into two subarrays each of size 5 450, 360, 510, 312, 220| 980, 460, 540, 800, 150.
Step 2:
Split the subarrays into size of two and three 450, 360, 510| 312, 220,|980, 460, 540, 800, 150.
Step3:
Split the subarrays into size two and one 450, 360,| 510, 312, 220,|980, 460, 540, 800, 150
Step 4:
Split the two values into one element 450| 360| 510| 312, 220| 980, 460, 540, 800, 150
Step 5:
Merge the values
360, 450| 510| 312, 220| 980, 460, 540, 800, 150
360, 450, 510| 312,220| 980, 460, 540, 800, 150
360, 450, 510|220, 312| 980, 460, 540, 800, 150
220, 312, 360, 450, 510| 980, 460, 540, 800, 150
Step 6:
Follow the same rules to split other set of sub array.
220, 312, 360, 450, 510| 980| 460, 540| 800, 150
220, 312, 360, 450, 510| 980, 460| 540| 800, 150
220, 312, 360, 450, 510|980| 460| 540| 800, 150
220, 312, 360, 450, 510| 460, 980| 540| 800, 150
220, 312, 360, 450, 510| 460, 540, 980| 800, 150
220, 312, 360, 450, 510|450, 460, 540, 800, 980
Step 7:
Then these two sub arrays are sorted and final result is 150, 220, 312, 360, 450, 460, 510, 540, 800, 980

12. (b) The Knapsack problem can be stated as follows.
Suppose there are n objects from $i = 1, 2,\ldots n$. Each object i has some positive weight W_i and some profit value is associated with each object which is denoted as P_i. This Knapsack carry at the most weight W.

While solving above mentioned Knapsack problem we have the capacity constraint. When we try to solve this problem using greedy approach our goal is

1. Choose only those objects that give maximum profit
2. The total weight of selected objects should be ≤ Wand we can obtain the set of feasible solutions. In other words maximized $\sum_{n}^{1} P_i x_i$ subject to $\sum_{n}^{1} W_i x_i \le W$. Where the Knapsack can carry the fraction X_i of an object i such that $0 \le x_i \le 1$ and $1 \le i \le n$

ALGORITHM:

```
void Greedy Knapsack (int m, int n)
{
   for (i = 1; i < n; i++)
   int a[i] = 0.0;
   int v = m;
   for (i = 1; i < n; i++)
   {
      if(x[i] > v)
         break;
      else;
      {
         a[i] = 1.0;
         v = v – x[i];
      }
   }
   if(i ≤ n)
   {
      a[i] = v/x[i];
   }
```

Example:

Find the optimal solution to the Knapsack instance.

$n = 6$, $m = 20$; $\{P_1, P_2, P_3, \ldots P_6\} = \{7, 8, 5, 11, 2, 8\}$

$\{W_1, W_2, W_3, \ldots W_6\} = \{5, 2, 4, 3, 5, 4\}$

Arrange $\frac{P_i}{w_i}$ in descending order such that $\frac{P_i}{W_i} >= \frac{P[i+1]}{W[i+1]}$

$$\frac{P_i}{w_i} = \left\{\frac{7}{5}, \frac{3}{2}, \frac{5}{4}, \frac{11}{3}, \frac{2}{5}, \frac{8}{4}\right\}$$

$$= \{1.4, 1.5, 1.25, 3.6, 0.4, 2\}$$

$$= \frac{P_i}{W_i} = \frac{P_4}{W_4} \ge \frac{P_6}{W_6} \ge \frac{P_2}{W_2} \ge \frac{P_1}{W_1} \ge \frac{P_3}{W_3} \ge \frac{P_5}{W_5}$$

Action	Remaining Weight in Knapsack	Profit Earned
$i = 4$	$20 - 3 = 17$	$0 + 11 = 11$
$i = 6$	$17 - 4 = 13$	$11 + 8 = 19$
$i = 2$	$13 - 2 = 11$	$19 + 3 = 22$
$i = 1$	$11 - 5 = 6$	$22 + 5 = 27$
$i = 3$	$6 - 4 = 2$	$27 + 5 = 32$
$i = 5$		$32 + 0.8 = 32.8$

In such a situation select fraction part of object

$$= \frac{\text{remaining space in knapsack}}{\text{Actual weight of selected object}} \times \text{profit of selected object}$$

$$= \frac{2}{5} \times 2$$

$$= \frac{4}{5}$$

$$= 0.8$$

13. (a) All Pair Shortest Path

We consider the problem of finding shortest pair between the all pairs of vertices in a graph. We are given a weighted, directed graph $G = (V, E)$ with a weighted function W: $E \rightarrow R$ that maps edges to real – value d weight.

Let $G = (V, E)$ be a graph is directed with n vertices. Let cost be a cost adjacency matrix for G such that cost $(i, j) = 0$, $1 < i \leq n$. Then cost (i, j) is the length of edge (i, j) if $(i, j) \in E(G)$ and cost $(i, j) = \infty$ if $i \neq j$ and $(i, j) \in E(G)$.

Definition:

The all pairs shortest path problem is to determine matrix A such that $A(i, j)$ is the length of a shortest path from i to j. The matrix A can be obtained by solving n single – source problems using the algorithms of shortest – paths.

Note:

If we allow G to contain a cycle of negative length, then the shortest path between any two vertices on this cycle has length $-\infty$

Using $A^k(i, j)$ to represent the length of a shortest path from i to j going through no vertex of index greater than k, we obtain.

$$A(i,j) = \min \begin{cases} \min \\ 1 \le k \le n \end{cases} \left\{ A^{k-1}(i,k) + A^{k-1}(k,j) \right\}, \cos t(i,j) \Bigg\}$$

We can obtain a recurrence for $A^k(i, j)$ using an argument similar to that used before. A shortest path from i to j going through no vertex higher than k either goes through vertex k or it does not.

If it does, $A^k(i,j) = A^{k-1}(i,k) + A^{k-1}(k,j)$.

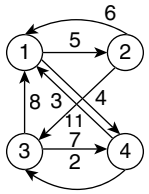
If it does not, then no intermediate vertex has index greater than $k-1$. Hence $A^k(i,j) = A^{k-1}(i,j)$ Combining, we get

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

```
Algorithm All paths (Cost, A , n)
{
  for i: = 1 to n do
  for j = 1 to n do
  A [i, j] = cost [i, j];//Copy cost into A
  for k: = 1 to n do
  for i: = 1 to n do
  for j: = 1 to n do
  A [i, j]: = min (A[i, j], A[i, k] + A[K + j]};
}
```

E.g.,:-

The graph of fig (a) has the cost matrix of fig.(b). The initial A matrix, $A^{(0)}$, plus its values after 4 iterations $A^{(1)}$, $A^{(2)}$, $A^{(3)}$ and $A^{(4)}$ are given.



Step 1:

$$A^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 5 & \infty & 4 \\ 2 & 6 & 0 & 11 & \infty \\ 3 & 8 & \infty & 0 & 7 \\ 4 & 3 & \infty & 2 & 0 \end{array}$$

Step 2:

$k = 1$

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

$i = 1, j = 1$

$A^1(1, 1) = \min \{A^0(1, 1), A^0(1, 1) + A^0(1, 1)$
$= \min \{0, 0 + 0\}$
$= 0.$

$i = 1, j = 2$

$A^1(1, 2) = \min \{A^0(1, 2), A^0(1, 2) + A^0(1, 2)\}$
$= \min \{5, 0 + 5\}$
$= 5.$

$i = 1, j = 3$

$A^1(1, 3) = \min \{A^0(1, 3), A^0(1, 1) + A^0(1, 3)\}$
$= \min \{\infty, 0 + \infty\}$
$= \infty$

$i = 1, j = 4$

$A^1(1,4) = \min \{A^0(1, 4), A^0(1, 1) + A^0(1, 4)\}$
$= \min \{4, 0 + 4\}$
$= 4.$

$i = 2, j = 1$

$A^1(2, 1) = \min \{A^0(2, 1), A^0(2, 1) + A^0(1, 1)\}$
$= \min \{6, 6 + 0\}$
$= 6.$

$A^1(2, 2) = \min \{A^0(2, 1), A^0(2, 1) + A^0(1, 2)\}$
$= \min \{0, 6 + 3\}$
$= 0.$

$A^1(2, 3) = \min \{A^0(2, 3), A^0(2, 1) + A^0(1, 3)\}$
$= \min \{11, 6 + \infty\}$
$= 11.$

$A^1(2, 4) = \min \{A^0(2, 4), A^0(2, 1) + A^0(1, 4)\}$
$= \min \{\infty, 6 + 4\}$
$= 10.$

$A^1(3, 1) = \min \{A^0(3, 1), A^0(3, 1) + A^0(1, 1)\}$
$= \min \{8, 8 + 0\}$
$= 8.$

$A^1(3, 2) = \min \{A^0(3, 2), A^0(3, 1) + A^0(1, 2)\}$
$= \min \{\infty, 8 + 5\}$
$= 13.$

$A^1(3, 3) = \min \{A^0(3, 3), A^0(3, 1) + A^0(1, 3)\}$
$= \min \{0, 8 + \infty\}$
$= 0.$

$A^1(3, 4) = \min \{A^0(3, 4), A^0(3, 1) + A^0(1, 4)\}$

$= \min \{7, 8 + 4\}$
$= 7.$

$A^1(4, 1) = \min \{A^0(4, 1), A^0(4, 1) + A^0(1, 1)\}$
$= \min \{3, 3 + 0\}$
$= 3.$

$A^1(4, 2) = \min \{A^0(4, 2), A^0(4, 1) + A^0(1, 2)\}$
$= \min \{\infty, 3 + 5\}$
$= 8.$

$A^1(4, 3) = \min \{A^0(4, 3), A^0(4, 1) + A^0(1, 3)\}$
$= \min \{2, 3 + \infty\}$
$= 2.$

$A^1(4, 4) = \min \{A^0(4, 4), A^0(4, 1) + A^0(1, 4)\}$
$= \min \{0, 3 + 4\}$
$= 0.$

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 5 & \infty & 4 \\ 6 & 0 & 11 & 10 \\ 8 & 13 & 0 & 7 \\ 3 & 8 & 2 & 0 \end{pmatrix} \end{array}$$

Step 3:

$K = 2$

$A^2(1, 1) = \min \{A^1(1, 1), A^1(1, 2) + A^1(2, 1)\}$
$= \min \{0, 5 + 6\} = 0$

$A^2(1, 2) = \min \{A^1(1, 2), A^1(1, 2) + A^1(2, 2)\}$
$= \min \{5, 5 + 0\} = 5$

$A^2(1, 3) = \min \{A^1(1, 3), A^1(1, 2) + A^1(2, 3)\}$
$= \min\{\infty, 5 + 11\} = 16$

$A^2(1, 4) = \min \{A^1(1, 4), A^1(1, 2) + A^1(2,4)\}$
$= \min \{4, 5 + 10\} = 4$

$A^2(2, 1) = \min \{A^1(2, 1), A^1(2, 2) + A^1(2, 1)\}$
$= \min \{6, 0 + 6\} = 6$

$A^2(2, 2) = \min \{A^1(2, 2), A^1(2, 2) + \mathrm{A}^1(2, 2)\}$
$= \min \{0, 0 + 0\} = 0$

$A^2(2, 3) = \min \{A^1(2, 3), A^1(2, 2) + A^1(2, 3)\}$
$= \min \{11, 0 + 11\} = 11$

$A^2(2, 4) = \min \{A^1(2, 4), A^1(2, 2) + A^1(2, 4)\}$
$= \min\{10, 0 + 10\} = 10$

$A^2(3, 1) = \min \{A^1(3, 1), A^1(3, 2) + A^1(2, 1)\}$
$= \min \{8, 13 + 6\} = 8$

Step 4:

$K = 3$

$A^3(1, 1) = \min \{A^2(1, 1), A^2(1, 3) + A^2(3, 1)\}$
$= \min \{0, 16 + 8\} = 0.$

$A^3(1, 2) = \min \{A^2(1, 2), A^2(1, 3) + A^2(3, 2)\}$
$= \min \{5, 16 + 13\} = 5.$

$A^3(1, 3) = \min \{A^2(1, 3), A^2(1, 3) + A^2(3, 3)\}$
$= \min \{16, 16 + 0\} = 16$

$A^3(1, 4) = \min \{A^2(1, 4), A^2(1, 3) + A^2(3, 4)\}$
$= \min \{4, 16 + 7\} = 4$

$A^3(2, 1) = \min \{A^2(2, 1), A^2(2, 3) + A^2(3, 1)\}$
$= \min \{6, 11 + 8\} = 6$

$A^3(2, 2) = \min \{A^2(2, 2), A^2(2, 3) + A^2(3, 2)\}$
$= \min \{0, 11 + 13\} = 0.$

$A^3(2, 3) = \min \{A^2(2, 3), A^2(2, 3) + A^2(3, 3)\}$
$= \min \{11, 11 + 0\} = 11$

$A^3(2, 4) = \min \{A^2(2, 4), A^2(2, 3) + A^2(3, 4)\}$
$= \min \{10, 11 + 7\} = 10.$

$A^2(3, 2) = \min \{A^1(3, 2), A^1(3, 2) + A^1(2, 2)\}$
$= \min \{13, 13 + 0\} = 13$

$A^2(3, 3) = \min \{A^1(3, 3), A^1(3, 2) + A^1(3, 3)\}$
$= \min \{0, 14 + 0\} = 0$

$A^2(3, 4) = \min \{A^1(3, 4), A^1(3, 2) + A^1(3,4)\}$
$= \min \{7, 13 + 7\} = 7$

$A^2(4, 1) = \min \{A^1(4, 1), A^1(4, 2) + A^1(2, 1)\}$
$= \min \{3, 8 + 6\} = 3$

$A^2(4, 2) = \min \{A^1(4, 2), A^1(4, 2) + A^1(2, 2)\}$
$= \min (8, 8 + 0) = 8$

$A^2(4, 3) = \min \{A^1(4, 3), A^1(4, 2) + A^1(2, 3)\}$
$= \min (2, 8 + 11) = 2$

$A^2(4, 4) = 0.$

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 5 & 16 & 4 \\ 6 & 0 & 11 & 10 \\ 8 & 13 & 0 & 7 \\ 3 & 8 & 2 & 0 \end{pmatrix} \end{array}$$

$A^3(3, 1) = \min \{A^2(3, 1), A^2(3,3) + A^3(3, 1)\}$
$= \min \{8, 0 + 8\} = 8.$

$A^3(3, 2) = \min \{A^2(3, 2), A^2(3, 3) + A^3(3, 2)\}$

$= \min \{13, 0 + 13\} = 13.$

$A^3(3, 3) = \min \{A^2(3, 3), A^2(3, 3) + A^3(3, 3)\}$
$= \min \{0, 0 + 0\} = 0.$

$A^3(3, 4) = \min \{A^2(3, 4), A^2(3, 3) + A^2(3, 4)\}$
$= \min \{7, 0 + 7\} = 7.$

$A^3(4, 1) = \min \{A^2(4, 1), A^2(4, 3) + A^2(3, 1)\}$
$= \min \{3, 2 + 3\} = 3.$

$A^3(4, 2) = \min \{A^2(4, 2), A^2(4, 3) + A^2(3, 2)\}$
$= \min \{8, 2 + 14\} = 8.$

$A^3(4, 3) = \min \{A^2(4, 3), A^2(4, 3) + A^2(3, 3)\}$
$= \min \{2, 2 + 0\} = 2.$

$A^3(4, 4) = \min \{A^2(4, 4), A^2(4, 3) + A^2(3, 4)\}$
$= \min \{0, 2 + 7\} = 0.$

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 5 & 16 & 4 \\ 6 & 0 & 11 & 10 \\ 8 & 13 & 0 & 7 \\ 3 & 8 & 2 & 0 \end{pmatrix} \end{array}$$

Step 5:

$K = 4$

$A^4(1, 1) = \min \{A^3(1, 1), A^3(1, 4) + A^3(4, 1)\}$
$= \min \{0, 4 + 3\} = 0.$

$A^4(1, 2) = \min \{A^3(1, 2), A^3(1, 4) + A^3(4, 2)\}$
$= \min \{5, 4 + 8\} = 5.$

$A^4(1, 3) = \min \{A^3(1, 3), A^3(1, 4) + A^3(4, 3)\}$
$= \min \{16, 4 + 2\} = 6.$

$A^4(1, 4) = \min \{A^3(1, 4), A^3(1, 4) + A^3(4, 4)\}$
$= \min \{4, 4 + 0\} = 4.$

$A^4(2, 1) = \min \{A^3(2, 1), A^3(2, 4) + A^3(4, 1)\}$
$= \min \{6, 10 + 3\} = 6.$

$A^4(2, 2) = \min \{A^3(2, 2), A^3(2, 4) + A^3(4, 2)\}$
$= \min \{0, 10 + 8\} = 0.$

$A^4(2, 3) = \min \{A^3(2, 3), A^3(2, 4) + A^3(4, 3)\}$
$= \min \{11, 10 + 2\} = 11$

$A^4(2, 4) = \min \{A^3(2, 4), A^3(2, 4) + A^3(4, 4)\}$
$= \min \{10, 10 + 0\} = 10.$

$A^4(3, 1) = \min \{A^3(3, 1), A^3(3, 4) + A^3(4, 1)\}$
$= \min \{8, 7 + 3\} = 8.$

$A^4(3, 2) = \min \{A^3(3, 2), A^3(3, 4) + A^3(4, 2)\}$

$= \min \{13, 7 + 8) = 13$

$A^4(3, 3) = \min \{A^3(3, 3), A^3(3, 4) + A^3(4, 3)\}$

$= \min \{0, 7 + 2\} = 0$

$A^4(3, 4) = \min \{A^3(3, 4), A^3(3, 4) + A^3(4, 4)\}$

$= \min \{7, 7 + 0\} = 7.$

$A^4(4, 1) = \min \{ A^3(4, 1), A^3(4, 4) + A^3(4, 1)\}$

$= \min\{3, 0 + 3\} = 3.$

$A^4(4, 2) = \min \{ A^3(4, 2), A^3(4, 4) + A^3(4, 2)\}$

$= \min \{8, 0 + 8\} = 8.$

$A^4(4, 3) = \min \{ A^3(4, 3), A^3(4, 4) + A^3(4, 3)\}$

$= \min \{2, 0 + 2\} = 2$

$A^4(4, 4) = \min \{ A^3(4, 4), A^3(4, 4) + A^3(4, 4)\}$

$= \min \{0, 0 + 0\} = 0.$

$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 5 & 6 & 4 \\ 6 & 0 & 11 & 10 \\ 8 & 13 & 0 & 7 \\ 3 & 8 & 2 & 0 \end{pmatrix} \end{array}$$

13. (b) Traveling sale person problem is solved by applying the dynamic programming since the solution is viewed as the result of sequence of decisions. Let us consider graph G with vertices V and edges E denoted as $G = (V, E)$ which is a directed graph with the cost specified for each edge. Cost is denoted as C_{ij} where i, j is vertices in graph. Value for C_{ij}, is greater than 0 if there is edge between i and j otherwise $C_{ij} = \infty$ i.e. $(i, j) \notin E$ and no of vertices is always greater than 1.

Traveling sales person problem is to find tour of least (i.e.,) minimum cost. Taking graph G tour is that every vertex in V should be included and cost for tour is sum of costs of edges. Which specified by $C_{ij.}$

Traveling salesperson may be applied in many situations. For example, let us consider courier service, courier man should collect courier from office and should deliver to customers and return to the office again. So here no of couriers he have to deliver is taken has vertices let i be $n + 1$ where n is number of couriers and one added is office. Each couriers have to delivered is located in different sites. So if i is one site and i is another distance is taken as cost and i to j edge with cost C_{ij}. So the path taken by courier man to deliver all couriers located in different sites should be of minimum length. This

minimum distance can be found by traveling sales person problem.

Let the tour be simple path where it starts and ends at vertex '*a*'.

Then every tour has edge (a, e) for some $e \varepsilon V - \{a\}$ V-vertices of graph. And it also has path from e to vertex a.

The path from e to a should go through all other vertices exactly once.

If path from e to a is the only shortest path of e to a through all vertices. Then the tour is optimal.

The function $t(a, V - \{a\})$ is the length of an optimal salesperson tour.

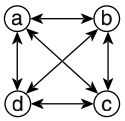
We obtain generalized function as.

$$t(i, G) = \min_{j \in G} \{C_{ij} + t(j, G - \{j\})\}$$

Where $i \notin G$

G is vertices other than $i(G = V - \{i\})$

Now let us consider the following example



The edge cost for the graph is given below.

$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 8 & 12 & 13 \\ 6 & 0 & 5 & 10 \\ 11 & 3 & 0 & 1 \\ 7 & 6 & 6 & 0 \end{bmatrix} \end{array}$$

Now the function is calculated for each vertices.

Case1:

Let intermediate node be ϕ (No intermediate node)

$$t(b, \phi) = 6 \min \{c_{ba}\} = 6$$
$$t(c, \phi) = \min \{c_{ca}\} = 11$$
$$t(d, \phi) = \min\{c_{ca}\} = 7$$

Now there is 1 intermediate node $|G|$

$$t(b, \{c\}) = c_{bc} + t(c, \phi) \Rightarrow \min_{C \in G} \{c_{bc} + t(c, G - \{c\})$$
$$= 5 + 11$$
$$= 16$$

$$t(b, \{d\}) = c_{bd} + t(d, \phi) \Rightarrow \min_{d \in G} \{c_{bd} + t(d, G - \{d\})$$
$$= 10 + 7$$
$$= 17$$

$$t(c, \{b\}) = C_{cb} + t(b, \phi) \Rightarrow \min_{b \in G} \{c_{cb} + t(b, G - \{b\})$$
$$= 3 + 6$$
$$= 9$$

$$t(c, \{d\}) = c_{cd} + t(d, \phi) \Rightarrow \min_{d \in G} \{c_{cd} + t(d, G - \{d\})$$
$$= 1 + 7$$
$$= 8$$

$$t(d, \{b\}) = C_{db} + t(b, \phi) \Rightarrow \min_{b \in G} \{c_{db} + t(b, G - \{b\})$$
$$= 6 + 6$$
$$= 12$$

$$t(d\{c\}) = c_{dc} + t(c, \phi) \Rightarrow \min_{c \in G} \{c_{dc} + t(c, G - \{c\})$$
$$= 6 + 11$$
$$= 17$$

Now there is 2 intermediate node $|G| = 2$.

$$t(b, \{c, d\}) = \min \{c_{bc} + t(c, G - \{c\}, c_{bd} + t(d, G - \{d\})$$
$$= \min \{c_{bc} + t(c, \{d\}, c_{bd} + t(d, \{c\}\}$$
$$= \min (5 + 8, 10 + 17)$$
$$= \min (13, 27)$$
$$= 13$$

$$t(c, \{b, d\}) = \min_{b,d \in G} \{c_{cb} + t(b, G - \{b\}, c_{cd} + t(d, G - \{d\})$$
$$= \min_{b,d \in G} \{c_{cb} + t(b, \{d\}); c_{cd} + t(d, \{b\})$$
$$= \min \{3 + 17, 1 + 12\}$$
$$= \min \{20, 13\}$$
$$= 13$$

$$t(d, \{b, c\}) = \min_{b,c \in G} \{c_{db} + t(b, G - \{b\}), c_{dc} + t(c, G - \{c\}$$

$$= \min_{b,c \in G} \{c_{db} + t(b,\{c\}; c_{dc} + t(c,\{b\})$$
$$= \min\{6+16,\ 6+9\}$$
$$= \min\{22,15\}$$
$$= 15$$

Now there is 3 intermediate node |G| = 3

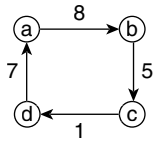
$$t(a,\{b,c,d,\}) = \min_{b,c,d \in G} \{c_{ab} + t(b, G - \{b\}),\ c_{ac} + t(c, G - \{c\},$$
$$c_{ad} + t(d, G - \{d\}\}$$

$$t(a,\{b,c,d\}) = \min_{b,\,c,\,d \in G} \{c_{ab} + t(b,\{c,d\}),\ c_{ac} + t(c,\{b,d\},$$
$$c_{ad} + t(d,\{b,c\}\}$$
$$= \min\{8+13,\ 12+13,\ 13+15\}$$
$$= \min\{21,\ 25,\ 28\}$$
$$= 21$$

So the length of optimal tour is 21
Let this value buy $T(i, G)$
Then $T(a, \{b, c, d\}) = b$ (gives the minimum), so the next tour starts from $a \to b$ then

$$T(b, \{c, d\}) = c$$

$a \to b \to c \to d \to a$ is minimal optimal hour
Thus the traveling sales person problem is solved by applying Dynamic programming Design method.



$$(8 + 5 + 1 + 7) = 21$$

14. (a) **Graph coloring:**
Let G be the graph where every node of G can be coloured in such a way that, no two adjacent vertices have the same color. Only m-colours are used. This is called graph coloring (or) m-colorability decision problem.

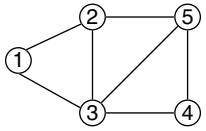
If d is the degree of a graph, then it can be colored with $(d + 1)$ colors.

Coding:

```
#include <stdio.h>
#include <conio.h>
int z, n, a[ ];
void m-colors (int z);
void main( )
{
   Printf("\n program for Graph coloring");
}
void m-colors (int z)
{
   while (!false)
   {
      Nxt val(z);//Assign a [z] a legal color.
      if (a[z] = = 0)
      {
         break; //no new color possible.
      }
      if(z = = n) //m-colors have been used to n-vertices.
      {
         for (int i = 1; i< = n; i++)
            {
               printf(x[i]); //print the assignments made
            }
      }
      else
      m-colors (z + 1);
   }
}
int Nxtval(int z)
{
// if no color exits, then a[z] = 0.
   while (!false)
   {
      a[z] = (a[z] + 1) % (m+1); //Next color;
      if(a[z] = = 0)
      {
         break;
```

```
        }
        for (int j =1; j <= n; j ++)
        {
            if((G[Z, J]! = 0) && (a[z] = a[j]))
            {
                break; //adjacent vertices having same color,
            }
            if (j = n + 1)
            {
                return z; //New color found
            }
        }
    }
```

Example:



c = Red, blue, green, black. Obtain the chromatic number.

Solution:

First find the adjacent vertices for each vertex in the given graph

Vertex	Adj. Vertices
1	2, 3
2	1, 3, 5
3	1, 2, 4, 5
4	3, 5
5	2, 3, 4

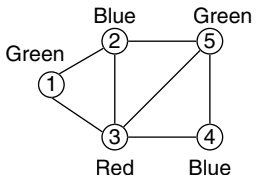
Now, Arrange the vertices that has highest adjacent vertices to lowest adjacent vertices.

sorted vertices = {3, 2, 5, 1, 4}

Construct a table that the given colors & vertices in the sorted order. Give a ( ) mark if that color is used else use (*X*).

Vertex	Red	Blue	Green	Black
3		X	X	X
2	X		X	X
5	X	X		X
1	X	X		X
4	X		X	X

Here only 3 colors are used to color the graph.
∴ chromatic number = 3.



14. (b) Write down and explain the procedure for tackling the 8-queens problem using a backtracking approach.

Backtracking:

Backtracking: is one of the most general technique. In backtracking method.

- The desired solution is expressed as an n-tuple (X_1, X_2, …, X_n) where X_i is choosen from some finite set S_i
- The solution maximizes (or) minimizes (or) satifies a criterion function C (X_1, X_2, X_2,…X_n).

8-Queens problem:

The 8-queens problem can be stated as follows:
Consider a 8 × 8 chessboard on which we have to place 8-queens so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

Coding**:**

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

```
int z, a[ ], n, i, j;
void nqueen(int z, int n);
void main( )
{
   Printf("\n program for nqueen problem");
}
void nqueen (int z, int n)
{
   for (i = 1; i <= n; i++)
   {
   if (placing(z, i))
   {
         a[z]=i;
         if(z = = n)
         {
            for (i = 0; i <= n; i++)
            {
               Printf(x[i]);
            }
         }
         else
         {
            nqueen (z + 1, n));
         }
   }
}
Boolean placing (int z, int i)
{
   for (int j = 1; j <= k–1; j++)
}
if ((a[j] = = i) || (Abs(a[j]–i) = Abs (j – z)))
   return false;
else
   return true;
}
}
```

Example:

Solve 8-queen problem for a feasible sequence 7, 5, 3, 1.

	1	2	3	4	5	6	7	8
1							Q_1	
2					Q_2			
3			Q_3					
4	Q_4							
5						Q_5		
6								Q_6
7		Q_7						
8				Q_8				

Condition:

If the positions are $p(i, j)$ and $p(k, l)$, then
$i - j = k - l$ (or)
$i + j = k + 1$.

Positions								**Conditions**
1	2	3	4	5	6	7	8	$i - j = k - l$ (or) $i + j = k + l$
7	5	3	1	2				$3 = 3$
7	5	3	1	6	6			$3 \neq 1$ $5 \neq 11$
7	5	3	1	6	8			$1 \neq 2$ $11 \neq 14$
7	5	3	1	6	8	2		$5 \neq 1$ $9 \neq 14$
7	5	3	1	6	8	2	4	$5 \neq 4$ $9 \neq 12$

Feasible solution = {7, 5, 3, 1, 6, 8, 2, 4}

15) (a) **Biconnected Components:**

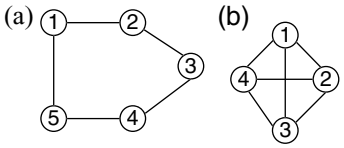
A graph G is biconnected if and only if it contains no articulation points.

Articulation Point (Or) Cut Vertex:

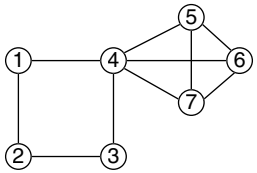
Let $G = (V, E)$ be connected Graph then the Articulation point is a vertex whose removal disconnects the graph into two or more non empty components.

Example:

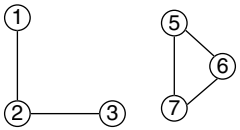
(i) **Biconnected Components:**



(ii) **Articulation Point:**



Here 4 is an Articulation point, whose removal changes the Graph into:



Identification of Articulation point:

Depth first search (DFS) provides a linear time algorithm to find all articulation points in a connected graph.

For every vertex V in the depth first spanning tree we compute $L(u)$.

Where,

$L(u)$-Lowest depth number.

Computation of $L[u]$:

$L(u) = \min\{\text{dfn}[u], \min\{L[w]\}/\text{being child of } u\}, \min\{\text{dfn } [w]/(u, w) \text{ being a back edge}\}\}$ vertex u will be an articulation point if;

1. u is not a root and
2. $L[w] \geq \text{dfn}[u]$

Algorithm to compute dfn and L(u):

Void Artpoint (u, v)

// G is a connected directed Graph with n nodes.

//This algorithm performs depth first search beginning from a

```
//vertex u, where u is a start vertex of graph G.
//variable index is initialized to zero.
dfn [u] = index;
L[u] = index;
index = index + 1;
for each vertex w adjacent from u do
{
   if (dfn [W] = = 0) then
   {
      Artpoint (w, u);
      L[u] = min (L[u], L[w]);
   }
   else if (w! = u) then
      L[u] = min (L[u], dfn [w]);
}
}
```

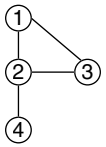
Algorithm to determine biconnected components:

```
Void Bicomp (u, v)
// G is a connected and directed graph with n nodes.
/*This algorithm computes biconnected components of a Graph
G using depth first search, beginning from a vertex u */
//variable "index" is initialized to one
{
   dfn [u] = index;
   L[u] = index;
   index = index + 1;
   For each vertex w =
   {
      if ((v! = w) and (dfn[w] < dfn [u])) then push (v, w) to the
      stack s;
      if (dfn [w] = = 0) then
      {
         if (L[w] ≥ dfn [u]) then
         {
            write ("New bicomponent");
            repeat
            {
            pop the top edge (X, Y) from the stack s;
            write (X, Y);
   }
until(((x, y) = (u, w)) or ((x, y) = (w, u)));
```

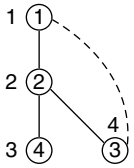
}
Bicomp (*w*, *u*);
L[*u*] = min (*L*[*u*], *L* [*w*]);
}
else if (*w*! = *v*) then
L [*u*] = min (*L*[*u*], dfn [*w*]);
}
}

Example:

- Graph:



- Spanning Tree:



- Computation of dfn values:
  dfn [1] = 1
  dfn [2] = 2
  dfn [3] = 4
  dfn [4] = 3
- Computation of *L*[*u*] values.
  L[*u*] = min{dfn[*u*], min{*L*[*w*]/being child of *u*},
  min {dfn[*w*]/ (*u*, *w*)being a back edge})
  L[1] = min {dfn (1), min (*L*[2]), min {dfn (3)}}
  = min{1, *L*[2], 4}
  = 1
  L[2] = min {dfn (2), min (*L*[4], *L*[3]), dfn (0)}
  = min {2, min {*L*[4], *L*[3]}}
  = min {2, 1} = 1
  L[3] = min {dfn(3), –, min{dfn(1)}
  = min {4, –, 1}
  =1
  L[4] = min {dfn (4), –, – }
  = 3

Identification of Articulation point:
$L[W] \geq$ dfn (u)
$u = 2, w = 3, 4$
$L[3] = 1$ dfn $(2)=1$.
$L[4] = 3$
$L[3] \geq$ dfn (1)
$L[4] \geq$ dfn (1)
$\therefore 2$ is the articulation point.
Complexity for finding articulation point.
The articulation point can be determined in $(n + e)$ time.
where,
n – no. of nodes in Graph G.
e – no. of edges in Graph G.

15. (b) Draw the portion of a state space tree related by FIFO, LCBB for job sequencing with deadlines instance $n = 5$. $P_i = (6, 3, 4, 8, 5)$, $T_i = (2, 1, 2, 1, 1)$, $D_i = (3, 1, 4, 2, 4)$. What is the penality corresponding to an optimal solution.

$$\hat{C}(x) = \sum_{i=m} P_i \qquad U(x) = \sum_{i=1}^{n} P_i$$

Node 1:

$$\hat{C}(x) = 0$$

$$U(x) = \sum_{i=1}^{n} P_i$$

$$= P_1 + P_2 + P_3 + P_4 + P_5$$

$$= 6 + 3 + 4 + 8 + 5$$

$$= 26$$

Node 2:

$$\hat{C}(x) = 0$$

$$U(x) = \sum_{i=1}^{n} P_i$$

$$= P_2 + P_3 + P_4 + P_5$$

$$= 3 + 4 + 8 + 5$$

$$= 20$$

Node 3:

$$\hat{C}(x) = P_1 = 6$$
$$U(x) = P_1 + P_3 + P_4 + P_5$$
$$= 6 + 4 + 8 + 5$$
$$= 23.$$

Node 4:

$$\hat{C}(x) = P_1 + P_2 = 9$$
$$U(x) = P_1 + P_2 + P_4 + P_5$$
$$= 6 + 3 + 8 + 5$$
$$= 22$$

Node 5:

$$\hat{C}(x) = P_1 + P_2 + P_3 = 6 + 3 + 4$$
$$= 13$$
$$U(x) = P_1 + P_2 + P_3 + P_5$$
$$= 6 + 3 + 4 + 5$$
$$= 18.$$

Node 6:

$$\hat{C}(x) = P_1 + P_2 + P_3 + P_4 = 6 + 3 + 4 + 8$$
$$= 21$$
$$U(x) = P_1 + P_2 + P_3 + P_4 = 21.$$

Node 7:

$$\hat{C}(x) = 0$$
$$U(x) = P_3 + P_4 + P_5 = 4 + 8 + 5 = 17.$$

Node 8:

$$\hat{C}(x) = P_2 = 3$$
$$U(x) = P_2 + P_4 + P_5 = 3 + 8 + 5 = 16.$$

Node 9:

$$\hat{C}(x) = P_2 + P_3 = 3 + 4 = 7$$
$$U(x) = P_2 + P_3 + P_5 = 3 + 4 + 5 = 12.$$

Node 10:

$$\hat{C}(x) = P_2 + P_3 + P_4 = 3 + 4 + 5 = 12$$
$$U(x) = P_2 + P_3 + P_4 = 12$$

Node 11:

$$\hat{C}(x) = P_1 = 6$$
$$U(x) = P_1 + P_4 + P_5 = 6 + 8 + 15 = 19$$

Node 12:

$$\hat{C}(x) = P_1 + P_3 = 6 + 4 = 10$$
$$U(x) = P_1 + P_3 + P_5 = 6 + 4 + 5 = 15$$

Node 13:

$$\hat{C}(x) = P_1 + P_3 + P_4 = 6 + 4 + 8 = 18$$
$$U(x) = P_1 + P_3 + P_4 = 18$$

Node 14:

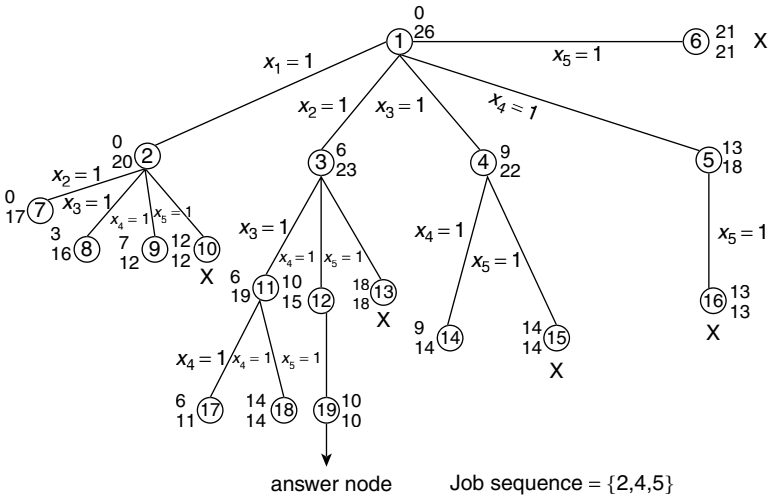
$$\hat{C}(x) = P_1 + P_2 = 6 + 3 = 9$$
$$U(x) = P_1 + P_2 + P_5 = 6 + 3 + 5 = 14$$

Node 15:

$$\hat{C}(x) = P_1 + P_2 + P_4 = 6 + 3 + 5 = 14$$
$$U(x) = P_1 + P_2 + P_4 = 14$$

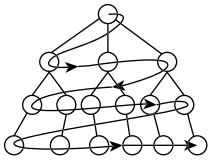
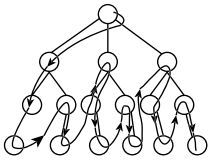
Node 16:

$$\hat{C}(x) = P_1 + P_2 + P_3 = 6 + 3 + 4 = 13$$
$$U(x) = 13$$



Branch and Bound:

The term branch and bound refers to all state – space search method, in which all children of the ε -node are generated before any other live node become the ε-node.

S.No	FIFO	L.C
•	BFS like state space search in which the exploration of a new node cannot begin until the node currently being explored is fully explored, will be called as FIFO[First-In-First-Out] search as the list of live nodes in a first-in-first-out manner (or stack)	A search strategy that uses a cost function $c(x) = f(h(x)) + g(x)$ to select the next ε-node would always choose for its next ε-node a line node with least $\hat{C}(\cdot)$. Hence a search strategy is called an LC search (Least-Cost Search)
•	Node Traversal 	Node Traversal 
•	In the above job sequencing problem, FIFO branch-and-bound algorithm begin with upper bound on the cost of a minimum cost answer node.	In the above job sequencing problem, LC branch and bound search of a true will begin with upper bound
•	Starting with node 1 as the ε-node	starting with node 1 as the ε-node
•	Using variable tuple size formulation, nodes 2, 3, 4, 5, 6 are generated.	When node 1 is expanded, nodes 2, 3, 4, 5, 6 are generated.
•	Node 2 corresponds to the inclusion of job 1, is generated	Here also node 2 corresponds to the inclusion of job 1, is generated
•	Next node 3 is generated and the variable upper is updated.	Next node 3 is generated and the variable upper is updated.
•	Here $\hat{C} = U(6) = 21$ $\therefore$ node 6 is killed (or bounded).	Here also node 6 is killed because of $\hat{C} = U(6) = 21$.
•	Now 2, 3, 4, 5 are live node	2, 3, 4, 5 remains as live nodes
•	Next node 2 become ε – node and 7, 8, 9, 10 are generated	Next node 2 become ε – node and 7, 8, 9, 10 are generated

S.No	FIFO	L.C
•	Node 10 will be bounded because $\hat{C}(10) = U(10) = 12$ and node 8, 9, 7 are live nodes. i.e., node 10 is infeasible.	Node 12 will be bounded same as in FIFO and live nodes are 8, 9, 7
•	Next ε-node is 3 and 11, 12, 13, are generated	But here node 7 becomes next ε-node and its children are infeasible.
•	Here 13 is infeasible, next ε-node is 4 and goes on	Here, next ε-node is 8 and goes on
•	The node 19 representing minimum cost answer node because least upper value is 10.	The node 19 representing minimum cost answer node.

B.E./B.Tech. DEGREE EXAMINATION, NOV/DEC 2011

Fourth Semester

Computer Science and Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

(Common to Information Technology)

(Regulation 2008)

Time: Three hours **Maximum: 100 marks**

Answer All Questions

PART A – (10 × 2 = 20 marks)

1. What do you mean by linear search?
2. What is the properties of big-Oh notation.
3. What are greedy algorithms?
4. What is a Knapsack problem?
5. What is a traveling Salesperson problem?
6. What do you mean by multistage graphs?
7. State the general backtracking method?
8. What is a graph cloning?
9. What is a spanning tree? Give an example.
10. What is an NP Completeness?

PART B – (5 × 16 = 80 marks)

11. (a) (i) Define Asymptotic notations. Distinguish between Asymptotic notation and conditional asymptotic notation. (10)

    (ii) Explain how the removing condition is done from the conditional asymptotic notation with an example. (6)

or

(b) (i) Explain how analysis of linear search is done with a suitable illustration. (10)

(ii) Define recurrence equation and explain how solving recurrence equations are done. (6)

12. (a) What is divide and conquer strategy and explain the binary search with suitable example problem. (16)

or

(b) Distinguish between quick sort and merge sort, and arrange the following numbers in increasing order using merge sort (18, 29, 68, 32, 43, 37, 87, 24, 47, 50) (16)

13. (a) (i) Explain the multistage graph problem with an example. (8)

(ii) Find an optimal solution to the knapsack instance $n = 7$, $m = 15$ $(p1, p2, p3, p7) = (10, 5, 15, 7, 6, 18, 3)$ and $(w1, w2, w3, \ldots, w7)$ (2, 3, 5, 7, 1, 4, 1) (8)

or

(b) Describe binary search tree with three traversal patterns? Give suitable example with neat diagram for all three traversal of binary search tree. (16)

14. (a) (i) How does backtracking work on the 8 queens problem with suitable example? (8)

(ii) Explain elaborately recursive backtracking algorithm? (8)

or

(b) What is Hamiltonian problem? Explain with an example using backtracking? (16)

15. (a) Write a complete LC branch and bound algorithm for the job sequencing with deadlines problem. Use the fixed tuple size formulation. (16)

or

(b) Write a non deterministic algorithm to find whether a given graph contains a Hamiltonian cycle. (16)Solutions

Part A

1. What is meant by linear search?
   Linear search is a method for finding a particular value in a list, that contains of checking everyone of its elements, one at a time and in sequence, until the desired one is found.

   Linear search is the simplest search algorithm which is a special case of brute-force search.

   For a list with n items, the best case is when the value is equal to the first element of the list, in which are only one comparison is needed. The worst case is when the value is not in the list in which n comparisons are needed.

   If the value being sought occurs K times, and all orderings of the list are likely to be expected as numbers of comparisons,

$$\begin{cases} n & \text{if } K = 0 \\ n+1, & \text{if } 1 \le K \le n \end{cases}$$

2. What is the properties of big-oh notations.
   If a function $f(n)$ can be written as a finite sum of other function, then the fastest growing one determines the order of $f(n)$.
   Product

$$f_1 \in O(g_1) \text{ and } f_2 \in O(g_2) \Rightarrow f_1 f_2 \in O(g_1, g_2)$$
$$f.0(9) \subset 0(f9)$$

   Sum

$$f_1 \in O(g_1) \text{ and } f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(|g_1| + |g_2|)$$

   This implies $f_1 \in O(g)$ and $f_2 \in O(g) \Rightarrow f_1 + f_2 \in O(g)$, which means that $O(g)$ is a convex cone.
   If f and g are positive function $f + O(g) \in O(f + g)$
   Multiplication by a constant
   Let k be a constant then:

$$O(kg) = O(g) \text{ if } k \text{ is non zero}$$
$$F \in O(g) \Rightarrow kf \in O(g).$$

3. What is meant by greedy algorithm
   A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum; In many problems a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time.

E.g.: Greedy strategy for the travelling sales man problem "At each stage visit an unvisited city nearest to the current city" . This A heuristic need not find a best solution but terminates in a reasonable number of steps; finding a optimal solution typically requires unreasonably many steps. In Mathematical optimization greedy algorithms solve combinational problems having the properties of matroids.

4. What is knapsack problem?
'Fractional knapsack problem: The setup is same but the thief can take carry a maximal fractions of items, meaning that the items can be broken into smaller pieces so that thief may decide to carry only a fraction of x_i of items i where $0 \leq x_i \leq 1$.

Exhibit greedy choice property and therefore greedy algorithm exists

– Given a set of items each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives a problem faced by someone who is constrained by the fixed size knapsack and must fill it with most valuable items.

5. What is travelling salesperson problem.
The travelling sales person problem is an NP-hard problem in combinatorial optimization studied in operational search.

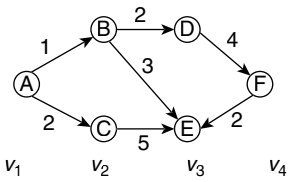
The problem TSP has several application even in its purest formulation such as planning and logistics, even though the problem is computationally difficult a large number of heuristics and exact methods are known.

Graph problem:

TSP can be modelled as an undirected weighted graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's length. It is a minimization problem starting and finished at a specified vertex after having visited each and other vertex exactly once. If no path exist between two cities, adding an arbitarily long edge will complete the graph without affecting optimal tour.

6. Define multistage graph.
A multistage graph $G = \{V, E\}$ is a directed graph in which Vertices are partioned into $k >= 2$ disjoint sets $V_i, 1 \leq i \leq k$ where v = set of vertices and E = set of edges.

- Also for each edge $<U, V>$ is E, $u \in v_i$ and $v \in v_i + 1$ for $1 \leq i \leq k$
- The sets v_1 and v_k have cardinality one.
- Let s and t be two vertices, such that s belongs to V_1, and t belongs to V_K
- S is the source vertex and t be the sink vertex.

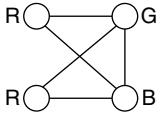


7. General backtracking method:
   Backtracking is a general method for finding all solutions to some computational problems. E.g., for backtracking is 8 queen problem, chess queens on the standard chessboard so that no queen attacks any other. In common backtracking approach, the partial candidates are arrangements of k queens in the first K rows of the board, all in different rows and columns. Any partial solutions that contains two mutually attacking queens can be abandoned, since it cannot possibly be completed to a valid solution.

8. What is graph coloring?
   Using Backtracking graph coloring is done. Back tracking is a search and optimization procedure when at each step we consider a number of possible choices, and for each one recursively solve a smaller version of the original. If the subprograms are not similar to the original, we may still be able to restate the problem to give it this similarity, usually by generalizing out original problem.

   Graph coloring is the way where the vertices of graph are coloured in such a way that no two adjacent vertices has same color.

   Consider the directed map $G = (V, E)$ and an integer K. Assign one of k colours to each node, such that no nodes get same colour.



9. What is spanning tree?
   A spanning tree T of a connected, undirected graph G is a true composed of all vertices and some of the edges of G. Informally, a spanning tree of G is a selection of edges of G that form a spanning tree for every vertex. That is, every vertex lies in the tree, but no cycles or loops are formed.

   A spanning tree of a connected graph G can also be defined as a maximal set of edges of G that contains no cycle, or as a minimal set of edges that connect all vertices.

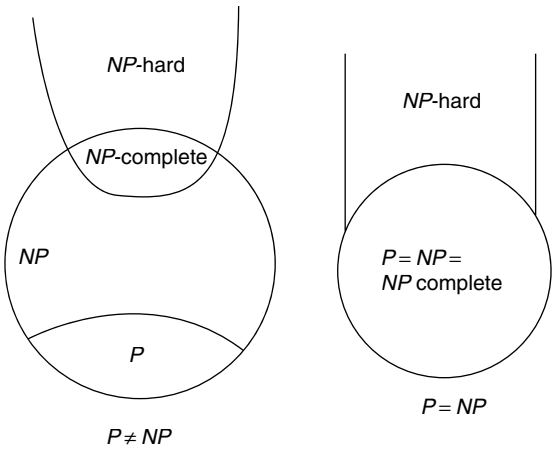
10. Define *NP* completeness.

The *NP* complete is a class of decision problems. A problem *L* is a *NP* - complete if it is in the set of *NP* problems and also in the set of *NP* - hard problems.

A decision problem *C* is *NP*-complete if

1. *C* is in *NP*, and
2. Every problem in *NP* is reducible to *C* in Polynomial Time.

C can be shown to be in *NP* by demonstrating that a candidate solution to *C* can be verified in polynomial time.



11. (a) (i) **Asymptotic Notation:**

Asymptotic algorithm is used to find the best algorithm for particular task among the other algorithms.

Asymptotic Notation:

A problem may have numerous algorithm solutions. In order to choose the best algorithm for a particular task you need to able to judge how long a particular solution will take to run or more accurately you need to able to judge how long two solutions will take to run, and choose the better of the two. You don't need to know how many minutes & seconds they will take but you do need some way to compare algorithm against one another.

Asymptotic complexity is a way of expressing the main component of the cost of an algorithm, using idealized units of computational work.

Consider, how we would go about comparing the complexity of two algorithms. Let $f(n)$ be the cost in the worst case, of one algorithm expressed as a function of the input size n and $g(n)$ be the cost functions for the other algorithm.

Conditional asymptotic Notation:

When we start the analysis of algorithm n as the size is satisfied in certain conditions. Consider n as the size of integer to be multiplied. The algorithm moves forward if $n = 1$, that needs microseconds for a suitable constant a if $n > 1$ the algorithm proceeds by accomplish addition tasks it takes some linear amount of time.

This algorithm takes a worst case time which is given by functions.

T: $N \to R^{\geq 0}$ recursively defined where R is a set of non negative integer.

$T(1) = a$

$T(n) = \Delta T([n/2]) + b_n$ for $n > 1$

Conditional asymptotic notation is a simple notation convenience. Its main interest is that we can eliminate it after using if for analysing the algorithm. A function F: $N\ R^{\geq 0}$ is non decreasing function if there is an integer threshold. No such that $F(n) \leq F(n+1)$ for all $n \geq n_0$. This implies that mathematical induction is $F(n) \leq F(m)$ whenever $m \geq n \geq n_o$.

Consider $b \geq 2$ as an integer function F is b smooth if it is non decreasing & satisfies the condition $F(b_n) < 0((F(n))$. Otherwise there should be constant C. Such that $F(b_n) < C(F(n))$ for all $n \geq n_o$.

(ii) Removing condition from the conditional asymptotic notation.

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

Consider c and no as Constant such that $F(b_n) \leq c.\ F(n)$ & $F(n) \leq F(n+1)$ for all $n = n_0$. Let $i = \left\lfloor \log_b{}^a \right\rfloor$ by definition of the algorithm $a = b$

Consider $n \geq n_0$ if is obvious to show by mathematical induction from b smoothrule of F that $F(b'_n) \leq c'\ (f(n))$. But $F(a_n) = F(b'_n)$ because F is eventually non-decreasing and approximate to $b^1{}_n \geq a_n \geq n_0$ If implies that $F(a_n) \in \hat{c}\ F(n)$for $\hat{c} = c^1$ and therefore F is a smooth

Smoothness rule:

Smooth functions seem to be interesting because of the smoothness rule. Consider $F : N \to R^{\geq 0}$ (where $R^{\geq 0}$ is non Negative integer) as a smooth function and T: $N \to R^{\geq 0}$ as an eventually non decreasing function. Consider an integer where $b \geq 2$. The

smoothness rule states that $T(n) < \theta(F(n))$ whenever $T(n) < \theta$ and $(F(n))/n$ is a power of b) we apply this rule equally to θ and Ω notation. The smoothness rules assumes directly that $T(n) < \theta(n^2)$ if n^2 is a smooth function and $T(n)$ is eventually non decreasing function. The first condition is immediate since the function is approximate $(2n)^2 = 4n^2$. Therefore the conditional asymptotic notation is a stepping stone that generates the final result unconditionally i.e., $T(n) = \theta(n^2)$.

11. (b) (i) Analysis if linear search:

Algorithms are usually analyzed to get best – case worst-case and average case of asymptotic values. Every problem is defined on a certain domain.

Let $J \in A_n$ be an instance of the problem, taken from the domain A_n Also, taken $s(j)$ as the computation time of the algorithm for the instance $J \in A_n$. It consists of three cases to analyse

Best – Case analysis:

In this case, it gives the minimum computed time of the algorithm with respect to all instances from the respective domain. It can be stated as.

$$C(n) = \min\{s(J) / J \in A_n\}$$

Worst – case analysis:

In this case, it gives the maximum computation time of algorithm with respect to all instance from the respective domain. It can be stated as

$$D(n) = \text{Max}\{s(J)/J \in A\}$$

Average - case analysis:

The average case is stated as.

$$E(n) = \sum_{J \in A_n} N(J)S(J)$$

Where $N(J)$ is the average probability with respect to the instance I.

E.g.,:

Consider an array, the elements may be in any, location in the array. To find the location of the particular element we use the

linear search algorithm.

a	b				y			k
0	1		...		i		...	n-1

Elements in the array

```
int linearsearch (const char A[], const unsigned int size, char ch)
{
for (int i = 0; i < size; i++)
{
if (A[i] = = ch)
return (i);
}
return (−1);
}
```

The searching is done in sequential as one location after the other, the searching is begin from the first location to the end the array. The program gets terminate when it finds the element or fails to find the element in the array after searching the whole array. If it cannot find the value it return it as-1.

From the information the comparison is made for three case.

For Best – case analysis $c(n) = \min \{1, 2, \ldots n\}$

$= 1$

$= 0\ (1)$

The comparisons for finding the elements at various location is

Location of the Element	**Number of Comparisons Required**
0	1
1	2
2	3
$n - 1$	n
not in the array	n

For the worst cases the searching is done by

$$D(n) = \max \{1, 2, 3 \ldots .n\}$$
$$= n$$
$$= 0(n)$$

Here the element could either be in the last location or could not be in the array

For the average case analysis,

Let k be the probability of y being in the array. the average probability is,

$$N(J_j) = k/n \text{ for } 0 \le j \le n-1$$
$$N(J_n) = 1-k$$

The probability for *y* not being in the array. Now,

$$E(n) = \sum_{j=0}^{n} N(J_j)s(J_j)$$
$$= \left(\frac{k}{n}\right)\sum_{j=0}^{n}(j+1) + (1-k)n$$
$$= \frac{k}{n}\frac{n(n+1)}{2} + (1-k)n$$
$$= \frac{k(n+1)}{2} + (1-k)n.$$

Suppose *y* is in the array, then $k = 1$.

$$\therefore E(n) = \frac{n+1}{2} = 0(n)$$

In case of *y* being in the array or not, $k = 1/2$.

$$\Rightarrow E(n) = \frac{n+1}{4} + \frac{n}{2}$$
$$= \left(\frac{3}{4}\right)n + 1$$
$$= 0(n)$$

$\therefore$ The computation time for linear searching is,

Algorithm	Best – case	Worst – case	Average – case
Linear search	0(1)	0(*n*)	0(*n*)

(ii) Recurrance equation:

Recurrance equation can be classified into homogeneous and inhomogeneous.

Suppose $X(n)$ is the time complexity of our algorithm for the size of the input *n*. Assume that $X(n)$is recursively defined as,

$$X(n) = b_1X(n-1) + b_2X(n-2) + \ldots + b_kX(n-k)$$
$$\Rightarrow a_0X(n) + a_1X(n-1) + \ldots + a_k \times (n-k) = 0 \quad (1)$$

The constants, which are b_i's are now converted to a_i's for sim-

plicity. Let us denote $x(i)$ as t^1, 1 becomes.

$$a_o t^n + a_1 t^{n-1} + \ldots + a_k t^{n-k} = 0 \qquad (2)$$

which is a homogeneous recurrence equation.

$$a_o t^k + a_1 t^{k-1} + \ldots + a_t = 0$$

which is the characteristics equation have k roots. Let the Roots $r_1, r_2, \ldots r_k$, the roots are may or may not be same.

Case (i)
when the roots are distinct.The general solution is,

$$X(n) = \sum_{i=L}^{K} a_i r_i^n$$

Where a_i is the constants.
Example, the characteristics equation is,

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

$\therefore$ General solutions is, $x(n) = C_1(1)^n + C_2(2)^n$
The general form of inhomogeneous recurrance equation is,

$$x_0 t_n + x_1 t_{n-1} + \ldots + x_k t_{n-k} = y_1^n B_1(n) + y_2^n B_2(n) + \cdots$$

The characteristics equation tends to be

$$(x_o t^k + x, t^{k-1} + \ldots x_k)(a-b_1)^{d_1+1}(a-b_2)^{d_2+1} = 0$$
$$\Rightarrow x_0 t^k + x_1 t^{k-1} + \ldots + x_k = 0$$
$$(a-b_1)^{d_1+1} = 0$$
$$(a-b_2)^{d_2+1} = 0$$
$$\ldots$$

e.g.,:

$$g(n) = 3g\,(n/3) + n$$
$$g(1) = 1$$

Which is the recurrence equation of the complexity of merge – sort algorithm.

Let $n = 2^k$ for simplicity, we say $g(2^k) = s_k$

$$\Rightarrow s_k - 2s_{k-1} = 2^k$$

e.g.: the characteristics equation of the recurrence equation

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

$\therefore$ The roots are 3, 3

$$\therefore s_k = C_1 3^k + C_2 k 3^k \qquad \{\therefore g(n) = C_1 n + C_2 n \log n\} \qquad (3)$$

Given: $g(1) = 1 \Rightarrow C_1 = 1$

$$g(2) = 3\, g(1) + 3 = 3 + 3 = 6$$

from equation (2)

$$6 = C_1 3 + C_2 3$$
$$6 = 2 + C_2 2$$
$$C_2 = 1$$

$\therefore$ The equation (2) becomes.

$$g(n) = n + n \log n$$
$$g(n) = 0(n \log n)$$

since $n \log n$ is 2–smooth and (n) is eventually non-decreasing.

12. (a) What is divide and conquer strategy and explain the binary search with suitable example problem?

Divide and conquer is an important algorithm design paradigm based on multi branched recursion. A divide and conquer strategy works by recursively breaking down a problem into two or more sub-problems of the same type, until these become simple to be solved directly. The solution to the sub problems are then combined to give a solution to the original problem.

This technique is the basis of efficient algorithm for binary search, merge sort, quick sort etc.

In binary search the divide and conquer is applied to reduce the problem to only one subproblem for finding the record in the sorted list.

Algorithm: Recursive binary search

```
void Bsearch(int a, int b, int c, int d)
{
int x[20];
if (c = = b)
{
if (d = = x[b])
return b;
else
return 0;
}
else
```

```
{
int mid = [(b +1)/2];
if (d = x[mid]
return mid;
else if (d < x[mid])
return Bsearch (a, b, mid −1, d);
else
return Bsearch (a, mid +1, c, d);
}
}
```

Iterative binary search

```
void Bisearch (int x, int y, int z);
{
int c[20];
int low, high, mid;
low = 1;
high = y;
while (low ≤ high)
{
mid = [(low + high)/2];
if (z < c[mid])
{
high = mid −1;
}
else if (z > c[mid])
{
low = mid +1;
}
else
return mid;
}
return 0;
}
```

Example:

a	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
Elements	−15	−16	0	7	9	23	54	82	101	112	125	131	142	151
Comparison	3	4	2	4	3	4	1	4	3	4	2	4	3	4

Space complexity: $n + 4$

$$\text{Midelement} = \frac{n}{2} = \frac{14}{2} = 7$$

Worst case = 4.

Best case = 1

Average case

(i) Average case if it is a successful search

$$= \frac{\text{Number of comparison}}{\text{total number of elements}}$$

The best case is 0(1).

(ii) Average case if it is a unsuccessful search.

$$= \frac{\text{Number of search} + (\text{total} \times \text{maximum})}{\text{Number of possiblities}}$$

$$= \frac{3 + (14 \times 4)}{15}$$

$$= 3.43$$

The time complexity for unsuccessful search is $O(\log_n)$

(b) Distinguish between quicksort and mergesort and arrange the following numbers in increasing order using mergesort {18, 29, 68, 32, 43, 37, 87, 24, 47, 50}

Mergesort is a natural approach. It is a nautral because simply divides the list into two equal sublists and get sorted these two partitions applying the same rule and merged later. The difficult part of merge sort is merging the sorted sub –lists. By performance measurement merge sort is slower than quicksort Mergesort algorithms are fast and stable and easy to implement Mergesort requires additional space, which could affect the performance.	In quicksort we have to choose an element from the list then we must put all the elements with value less than the elements on the left side and all the items with value greater than the element on the right side. There is no need of merging. The difficult part of quicksort is choosing an element. If we choose the greatest value in the list it will be the worst scenario. Quicksort is faster than mergesort though the average complexity O($n$log$n$) same as mergesort, it usually performes well in practice. Recursive implementation is easy and results elegant solution with no tricky merging as mergesort. Quicksort saving the performance and memory by not creating the extra space.

{18, 29, 68, 32, 43, 37, 87, 24, 47, 50}

Step 1:

Divide the array into two subarrays of equal size.

18, 29, 68, 32, 43| 37, 87, 24, 47, 50

Step 2:
Divide the first subarrays into size of two and three.
18, 29, 68| 32, 43| 37, 87, 24, 47, 50
Step 3:
Divide the first subarray further and merge it.
18, 29| 68| 32, 43| 37, 87, 24, 47, 50
18|29| 68| 32, 43| 37, 87, 24, 47, 50
18, 29| 68| 32, 43| 37, 87, 24, 47, 50
18, 29, 68| 32, 43| 37, 87, 24, 47, 50
18, 29, 32, 43, 68| 37, 87, 24, 47, 50
Step 4:
Divide the next subarray using same rules
18, 29, 32, 43, 68 | 37, 87, 24 | 47, 50
18, 29, 32, 43, 68 | 37, 87 | 24 | 47, 50
18, 29, 32, 43, 68 | 37 | 87 | 24 | 47, 50
18, 29, 32, 43, 68 | 37, 87 | 24| 47, 50
18, 29, 32, 43, 68 | 24, 37, 87|47, 50
18, 29, 32, 43, 68 | 24, 37, 47, 50, 87
Step 5:
Divided and sorted subarrays are merged.
18, 24, 29, 32, 37, 43, 47, 50, 68, 87

13. (a) (i) Explain the multistage graph with an example.

A multistage graph $G = (V, E)$ is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets, $V_i, 1 \leq i \leq k$. In addition, if (U, V) is an edge in E, then $u \in v_i$ and $u \in v_{i+1}$ for some i, $1 \leq i \leq k$. The sets V_i and V_k are such that $|V_i| = |V_k| = 1$

Let s and t, respectively, be the vertices in V_i and V_k. The vertex s is the source, and t the sink. Let $C(i, j)$ be the cost of edge(i, j).The cost of a path from s to t is the sum of the costs of the edges on the path.

The multistage graph problem is to find a minimum-cost path from s to t. Each set V_i defines a stage in the graph. Because of the constraints on E, every path from s to t starts in stage 1, goes to stage 2 than to stage 3, than 4 and so on and eventually terminates in stage K. A minimum-cost s to t path is indicated by the broken edges.

A dynamic programming formulation for a k-stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of k–2 decisions. It is easy to see that principles of optimality holds. Let $P(i, j)$ be a minimum-cost path from vertex j in V_i to Vertex t.

Let cost (i, j) be the cost of this path. Then using the forward

approach, thus obtain

$$\text{Cost}(i, j) = \min\{c(j, l) + \text{cost}(i+1, l)\}$$
$$l \in v_i + 1$$
$$(j, l) \in E$$

Algorithm: FGRAPH (G, K, N, P)
// The input is a *K*-stage graph $G = (V, E)$ with *n* vertices.
// indexed in order of stages. *E* is a set of edges and $c[i, j]$
// is the cost of (i, j). $P[1: k]$ is a minimum-cost path.
{
Let bcost (i, j) be the cost of bP(i, j).
From the backward approach we obtain

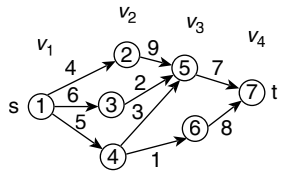
$$\text{bcost}(i, j) = \min\{\text{bcost}(i-1, l) + c(l, j)\}$$
$$l \in V_i - 1$$
$$(l, j) \in E.$$

Algorithm BGraph (G, K, N, P)
// Same function as FGraph
{
bcost [1]: = 00;
for j: = 2 to *n* do
{// compute bcost [j]
Let r be such that <r, j> is an edge of G and 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]];
}
{
cost[n]:= 0.0;
for j: = n − 1 to 1 step-1do
{// compute cost[j]
Let r be a vertex such that <j, r> is an edge of G and c[i, j] + cos[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[i]:=d[P[j – 1]];
}
The multistage graph problem can also be solved using the backward approach.

Let bp(i, j) be a minimum-cost path from vertex *s* to a vertex j in v_i

Backward Approach

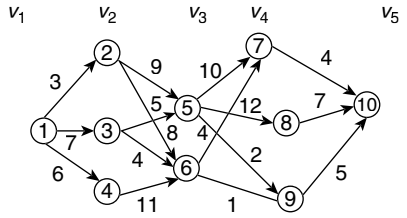


bcost (i, j) =min {bcost((i–1), l) + c(l, j)}
$j \in v_{i-1}$
$(l, j) \in E$
Step 1:
i = 2, j = 2, 3, 4
bcost (2, 2) = min {bcost (1, 1) + ((1, 2)}
bcost (1,1) = min { bcost (0)} =0
bcost (2,2) = min { 0 + 4} = 4 $l \in$ 1
bcost (2,3) = min {bcost (1,1) + *c*(1,3)}
= min {0 + 6} = 6 l$\in$ 1.
bcost (2,4) = min {bcost (4,1) + c(1,4)}
= min {0 + 5} = 5 l$\in$ 1.
Step 2:
i = 3, j = 5, 6
bcost (3,5) = min {bcost (2,2) + *c*(2,5), bcost (2,3) + c(3,5), bcost (2,4) + c(4,5)}
= min {(4 + 9), (6 + 2). (5 + 3)}
= min {13,8,8} (8 l $\in$ 3,4)
bcost (3,6) = min {bcost (2,4) + c(4,6)}
= min {5+1} = 6 *l* $\in$ 4
Step 3:
i = 4, j = 7
bcost (4, 7) = min { bcost (3, 5) + (5,7), bcost (3, 6) + *c* (6, 7)}
= min {(2 + 7), (0 + 8)}
= min {9, 8} = 8 l $\in$ 6.
The shortest path

$1 \rightarrow 4 \rightarrow 6 \rightarrow 7$

Forward Approach

$$Cost(i, j) = \min\{C(j, l) + \cos t(i+1, l)\}.$$



Step 1:

Initially

i = k –2

= 5 –2 = 3 ∴ $i = 3, j = 5, 6$

Cost (3, 5) = min {C(5, 7) + cost (4, 7), C (5, 8) + cost (4, 8), C(5, 9) + cost (4, 9)}

= min {10 + cost(4, 7),12 + cost(4, 8),4 + cost(4, 9)}

Cost (4, 7) = min {C(7, 10) + cost (5, 10)}

= min {4 + 0} = 4

Cost (4, 8) = min {C(8, 10) + cost (5, 10)}

= min {7 + 0} = 7

Cost (4, 9) = min {C(9, 10) + cost (5, 10)}

= min {5 + 0} = 5

∴ Cost (3, 5) = min {C(10 + 4),(12 + 7),(2 + 5)}

= min {14, 19, 7} (7 l ∈ 9)

Cost(3, 6) = min{C(6,7) + cost(4,7), C(6,9) + cost(4,9)}

= min{(4 + 4), (1 + 5)}

= min{8, 6} = 6l∈ 9

Step 2:

i = k – 3 = 5 – 3 = 2

∴ i = 2, j = 2, 3, 4

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

= min {(9 + 5), (8 + 4)}

= min {14, 12} = 12 l ∈ 6.

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

= min {(5 + 5),(4 + 4)}

= min {10, 8} = 8 l ∈ 6.

Cost (2, 4) = min {C(4, 6) + cost (3, 6)}
= min {11 + 4}
= 15 l ∈ 6.

Step 3:

j = k − 4 (i.e.,) 5 − 4 = 1
i = 1, j = 1
Cost (1, 1) = min {C(1, 2) + cost (2, 2), C(1, 3) + cost (2, 3), C(1, 4) + cost (2, 4)}
= min {(3 +12), (7 + 8), (6 +15)}
= min {15, 15, 21} ∴ (15 l ∈ 2, 3).

The shortest path which is obtained for
The figure is
(i) ①→②→⑥→⑨→⑩
(ii) ①→③→⑥→⑨→⑩

13. (a) (ii) Find an optimal solution to the Knapsack instance when $n = 7$, $m = 15$, $(P_1, P_2, P_3, P_4, P_5, P_6, P_7) = (10, 5, 15, 7, 6, 18, 3)$ and $(W_1, W_2, W_3, W_4, W_5, W_6, W_7) = (2, 3, 5, 7, 1, 4, 1)$

Solution:

Arrange $\frac{P_i}{W_i}$ in descending order such that

$$\frac{T_i}{W_i} \geq T(i+1)/W(i+1)$$

Therefore, $\frac{P_i}{W_i} = \frac{10}{2} = 5$

$$\frac{P_2}{W_2} = \frac{5}{3} = 1-6, \frac{P_3}{W_3} = \frac{15}{5} = 3, \frac{P_4}{W_4} = 7 = 1, \frac{P_5}{W_5} = \frac{6}{1} = 6,$$

$$\frac{P_6}{W_6} = \frac{18}{4} = \frac{9}{2} = 4.5, \frac{P_7}{w_7} = \frac{3}{1} = 3$$

$$\therefore \frac{P_i}{W_i} = \{5, 1.66, 3, 1, 6, 4.5, 3\}$$

$$\frac{P_5}{W_5} > \frac{P_1}{W_1} > \frac{P_6}{W_6} > \frac{P_7}{W_7} > \frac{P_3}{W_3} > \frac{P_2}{W_2} > \frac{P_4}{W_4}$$

Action	Remaining Weights In Knapsack	

$i = 5$	$15 - 1 = 14$	$0 + 6 = 6$
$i = 1$	$14 - 2 = 12$	$6 + 10 = 16$
$i = 6$	$12 - 4 = 8$	$16 + 18 = 34$
$i = 7$	$8 - 1 = 7$	$34 + 3 = 37$
$i = 3$	$7 - 5 = 2$	$37 + 15 = 52$

Select fraction parts of

$$\text{Object} = \frac{\text{remaining space in knapsack}}{\text{actual weight of selected object}} \times \text{profit of the selected object}$$

$$\Rightarrow \frac{2}{3} \times 15 = \frac{10}{13}$$

$$\Rightarrow 52 + \frac{10}{3} = 55.3$$

$$\therefore \text{The optimal solution is } \{1, \frac{2}{3}, 1, 0, 1, 1, 1, \}$$

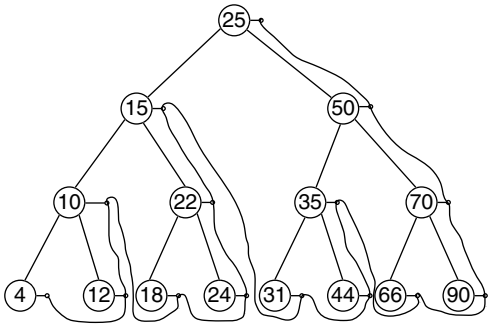
13. (b) **Binary Search Tree**

- BINARY SEARCH TREE (BST) is a special kind of binary tree which is *n* also called as ordered or sorted binary tree.
- It is a node based binary tree data structure.
- It may be empty and if it is not empty then it satisfies the following properties:
  1. The left subtree of a node contains only nodes with key value less than the node's key value.
  2. The right subtree of a node contains only nodes with keys greater than or equal to the node's key.
  3. Both the left and right subtrees must also be binary search trees.
- There are three kinds of operation involved with BST: search, insert and delete.
- TRAVERSAL: **Tree Traversal** refers to the process of visiting each node in a tree data structure, exactly once, in a systematic way. Such traversals are classified by the order in which the nodes are visited.
- There are three steps to traversal:
  1. Visit the current node
  2. Traverse its left subtree
  3. Traverse its right subtree
- The order in which you perform these three steps results in three different traversal orders:
  1. Pre-order traversal: (1) (2) (3)
  2. In-order traversal: (2) (1) (3)

3. Post-order traversal: (2) (3) (1)

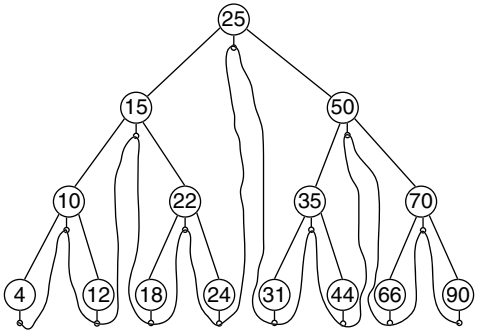
- i.e., In preorder traversal we visit the current node first then traverse left subtree and then the right subtree.
- In post order traversal we visit the left subtree first, then the right subtree and then the current node.
- In Inorder traversal we first traverse the left subtree then we visit the current node and then we traverse the right subtree.

Post order:



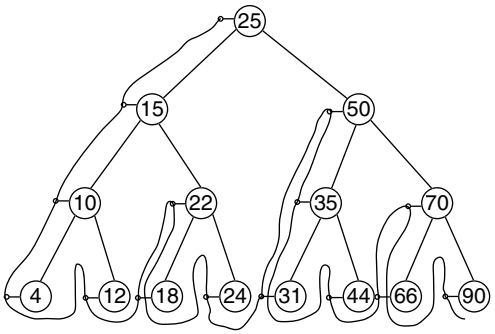
In Post order traversal, we visit the left subtree first, then the right subtree and then the current node, we can simplify the traversal process as shown above in the diagram. We can draw a hook towards the right for each node & start putting the thread from the left most leaf node.

In order:



In order traversal, we visit left subtree first, then the (Current node/ root node) & then the right subtree. We can simplify the traversal process as shown above, we can draw a hook below the node for each node and start putting the thread from leftmost leaf node. In this case node 4.

Preorder:



In preorder traversal, we visit the current node i.e., root node 25 & then the left subtree and then the right subtree of the root.

We can simplify the traversal process as shown in the above diagram. We draw a hook towards the left for each node & then start putting the thread from the top i.e., the root node.

Inorder (root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90
A Pre- order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90
A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

14. (a) (i) How does backtracking work on the 8-queens problem with suitable example

Backtracking:

Backtracking is one of the most general technique. In this method,

- the desired solution is expressed as an n tuple $(x_1, x_2, \ldots x_n)$ where x_i is choosen from some finite set s_i.
- The solution maximizes or minimizes or satisfies a criterion function $C(x_1, x_2, \ldots x_n)$

8- queens problem.

The 8-queens problem can be stated as follows.

On a 8×8 chessboard we have to place 8-queens such that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

Diagonal conflicts:

Let $k_1 = (i, j)$ and $k_2 = (k, l)$ are two positions. Then k_1 and k_2 are the positions that are on the Same diagonal if

$$i + j = k + l \quad \text{(or)}$$
$$i - j = k - l$$

Coding**:**

```
void n-queens (int z, int n)
{
for (int i = 1; i < = n; i++)
{
if (placing(z, i))
{
a[z] = i;
if (k = = n )
{
for (int i = 0;i < = n; i++)
{
print f(x[i]);
}
}
else
n - queens (z + 1,n);
}
}
}
Boolean placing (int z, int i)
{
for (int j = 0;j < k – 1; j ++)
{
if ((a[j] = = i) || (Abs (a[j] –i) = Abs (j – z)))
{
return false;
}
return true;
}
}
```

Example

Solve the 8-queens problem for a feasible sequence (6, 4, 7, 1).

1	2	3	4	5	6	7	8

1						Q_1		
2				Q_2				
3							Q_3	
4	Q_4							
5			Q_5					
6								
7		Q_7						
8								Q_8

Solution:

Positions								Diagonal Conflict
1	2	3	4	5	6	7	8	$i - j = k - 1$ (or) $i + j = k + 1$
6	4	7	1	2				3 = 3 (conflict)
6	4	7	1	3				$4-1 \neq 5-3$ $4+1 \neq 5+3$
6	4	7	1	3	2			$6-2 \neq 5-3$(not Feasible) $6+2 \neq 5+3$
6	4	7	1	3	5			$6+5 \neq 5-3$ $6+5 \neq 5+3$
6	4	7	1	3	5	2		$7+2 \neq 6+5$ $7-2 \neq 6-5$
6	4	7	1	3	5	2	8	$8-8 \neq 7-2$ $8+8 \neq 7+2$

The feasible solution is {6, 4, 7, 1, 3, 5, 2, 8}

14. (a) (ii) Explain elaborately recursive backtracking algorithm Backtracking:

Backtracking is one of the most general technique. In this method,

- The desired solution is expressed as an n - tuple $(x_1, x_2, \ldots, x_n)$ where x_i is choosen from some finite set S_i.
- The solution maximizes or minimizes or satisfies a criterion function $C(x_1, x_2, \ldots x_n)$

Algorithm:

```
void RBacktrack (int z)
{
for (z = 1; a[z] ∈ T (a[1], … , a(z −1); z + +)
{
if (B_z (a[1], a[2],. … , a[z]) ≠ 0)
{
for (i = 1; i < = z; i + +)
{
Printf (x[i]);
}
}
if (z < n)
{
RBacktrack (z + 1);
}
}
}
```

- A recursive backtracking technique is used because of the postorder traversal of a tree initially it is invoked by RBacktrack(1);
- The solution vector (a_1, ... a_n) is treated as a global array a[1:N].
- All the possible elements for the z^{th} position of the tuple that satisfy B_z are generated one by one, and adjoined to the current vector (a_1,a_2, … a_{z-1}).
- Each time a_z is attached, a check is made to determine whether a solution has been found.
- Thus the algorithm is recursively invoked.
- When the for loop is exited, there are no values for a_z and the current copy of RBacktrack ends.
- The algorithms prints all solutions and assumes the variable sized tuples forms a solution. If only a angle solution is desired, then a flag can be added as a parameter to indicate the first occurrence.

(b) What is Hamiltonian problem? Explain with an example using backtracking.

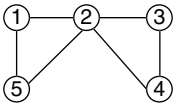
- Given an undirected connected graph and two nodes a and b then find a path from a to b visiting each node exactly once.
- To solve the hamiltonian problem, backtrack approach is used.
- The state space tree is generated to find all possible hamiltonian cycles in the graph.
- Only distinct cycles are the output of this algorithm.

Algorithm:

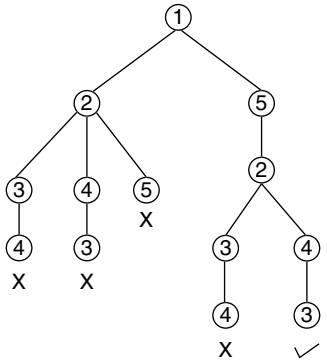
```
void nxtver (int z)
{
while (!False)
{
a[z] = (a[z] + 1]) % (n + 1); //Next vertex
if (a[z] = = 0)
{
break;
}
if (G[a[z − 1], a[z]] ! = 0)
{
for (int j = 1; j < = z −1; j++)
{
if (a[j] = = a [z])
break;
if (j = = z)
{
 if ((z < n) || ((z = = n) && G [a[n], a[i]] ! = 0))
return;
}
}
}
}
}
void Hcycle (int z)
{
while (! False)
{
nxtver (z); //Generates value for a[z].
if (a[z] = = 0)
break;
if (z = = n)
{
for (int i = 1; i < = n; i + +)
{
Printf (x[i]);
}}
else
Hcycle (z + 1);
}
}
```

Example:

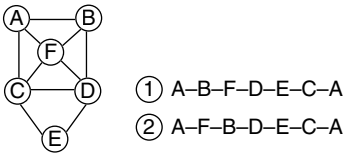
Determine the Hamiltonian cycles in the following graph

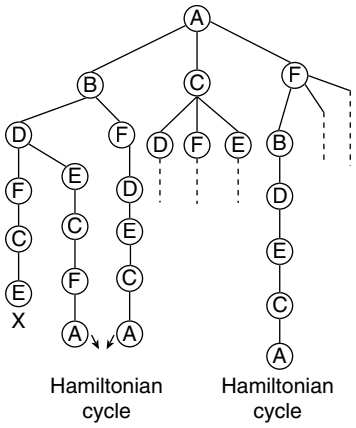


Solution:



This graph does not contain any hamiltonian cycle.





15. (a) (i) **Algorithm of LC Search**

```
list node = record {
listnode * next, *parent; float cost;
}
1 Algorithm LC Search(t)
2 // Search t for an answer node
3 {
4 if *t is an answer node then output *t and return;
5 ε: = t; // ε-node
6 Initialize the list of live nodes to be empty;
7 Repeat
8 {
9 for each child x of E do
10 {
11 if x is answer node then output the path
12 from x to t and return;
13 Add (x); // x is a new live node
14 (x → parent): = E; //Pointer for path to root.
15 }
16 if there are no more live node then
17 {
18 write ("No answer node"); return;
19 }
20 ε: = Least ( );
21 }
22 Until (false);
23 }
```

Let t be a state space tree and C ( ) a 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 subtree with root x.

If x is either an answer node or a leaf node. The algorithm uses $\hat{c}$ to find an answer node.

The algorithm uses two functions Least ( ) and Add (x) to delete and add a live node from or to the list of live nodes, respectively.

Least ( ) finds a live node with Least $\hat{c}$ ( ). This node is deleted from the list of live nodes and returned.

Add (x) adds the new live node x to the list of live nodes. The list of live nodes will usually be implemented as a min-heap.

Algorithm LC Search outputs the path from the answer node it finds to the root node t.

This is easy to do if with each node x that becomes live, we associate a field parent which gives the parent of node x.

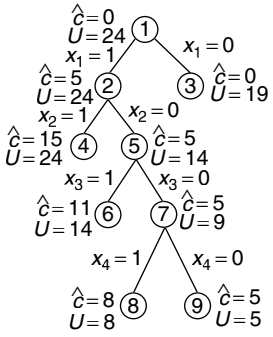
When an answer node *g* is found, the path from *g* to *t* can be determined by following a sequence of parent values starting from the current ε-node (which is the parent of *g*) and ending at node *t*.

By the definition of LC-search, the root node is the first ε node. Initially this list should be empty. The for loop examines all the children of the ε-node. If one of the children is an answer node, then the algorithm outputs the path from *x* to *t* and terminates.

If a child of ε is not an answer node, then it becomes a live node. It is added to the list of live nodes and its parent field set to *E*. When all the children of *E* have been generated, *E* becomes a dead node and this happens only if none of *E*'S children is an answer node. So, the search must continue further. If there are no live nodes left, then the entire state space tree has been searched and no answer nodes found. The algorithm terminates. Otherwise, Least ( ), by definition, correctly chooses the next ε -node and the search continues from here.

(ii) Write a complete Least Cost Branch and Bound algorithm for job sequencing with deadline problem. Use fixed tuple size formulation.

E.g.,: Consider the following instance $n = 4$, $(p_1, d_1, t_1) = (5, 1, 1)$. $(p_2, d_2, t_2) = (10, 3, 2)$, $(p_3, d_3, t_3) = (6, 2, 1)$ and $(p_4, d_4, t_4) = (3, 1, 1)$.



Calculation for

node 1: $\hat{C}(1) = 0$

$$U(1) = \sum_{i=1}^{n} P_i = P_1 + P_2 + P_3 + P_4 = 5 + 10 + 6 + 3 = 24$$

node 2: $\hat{C}(2) = 5$

$$U(2) = \sum_{i=1}^{n} P_i = P_1 + P_2 + P_3 + P_4 = 5 + 10 + 6 + 3 = 24$$

node 3: $\hat{C}(3) = 0$

$$U(3) = \sum_{i=2}^{n} P_i \quad = P_1 + P_2 + P_3 + P_4 = 10 + 6 + 3 = 19$$

node 4: $\hat{C}(4) = 15$

$$U(4) = P_1 + P_2 + P_3 + P_4 = 5 + 10 + 6 + 3 = 24$$

node 5: $\hat{C}(5) = 5$

$$U(5) = 5 + 6 + 3 = 14$$

node 6: $\hat{C}(6) = 11$

$$U(6) = P_1 + P_3 + P_4 = 5 + 6 + 3 = 14$$

node 7: $\hat{C}(7) = 5$

$$U(7) = P_3 + P_4 = 6 + 3 = 9$$

node 8: $\hat{C}(8) = 8$

$$U(8) = P_1 + P_4 = 5 + 3 = 8$$

node 9: $\hat{C}(9) = 5$

$$U(9) = P_1 = 5$$

As the node 1 is explored with rank = 0 and cost = 24 and 2, 3 nodes are generated.

Node 2 has the upper value as 24 and $\hat{c}(2) = 5$ where as the node 3 has the rank $\hat{c}(3) = 0$ and the $U(3) = 19$. Here the rank says 0 so the node 3 gets bounded.

So node 2 is expanded and the nodes 4 and 5 are generated. Likewise node 4 is bounded and node 5 is explored.
So nodes 6 and 7 are generated with inclusion and exclusion of job 3.

Node 6 gets bounded Since $\hat{c}(6) = 11 > U(7)$

So Node 7 is explored and node 9, 8 are generated.

Node 8 is a inclusion of job 4 and node 9 is exclusion of job 4.

The answer node is 8.

$U(8) = 8$

Job = {1, 4}

15. (b) Write a non-deterministic algorithm to find whether a given graph contains hamiltonian cycle.

Hamiltonian cycle:

- A Hamiltonian cycle of a graph is a cycle that contains every vertex of the graph.
- A Graph can be Hamiltonian if it follows the conditions given below:
  a) Ore's theorem – A simple graph with n vertices ($n > 2$) is Hamiltonian if sum of degrees of every pair of non adjacent vertices is atleast n.
  b) Dirac's theorem – A simple graph with n vertices ($n > 2$) is Hamiltonian if degree of every vertex is at least ($n/2$).

Problem: To write a non-deterministic algorithm to find whether a given graph contains Hamiltonian cycle.

Preprocessing condition:

- No nodes should have degree 1.
- No node should have more than two adjacent nodes having degree 2.

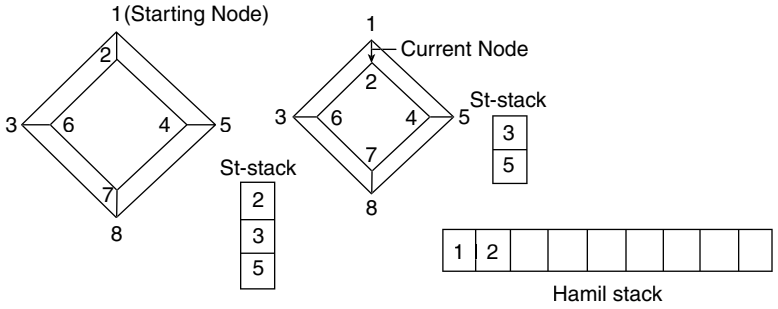
Algorithm:

Step 1: Select a node of highest degree from graph, to start travelling .we call this node the starting node. Store the adjacent nodes of the starting node in a stack called St-stack.

Step 2: Select one of the St-stack nodes and name it as next node according to following priorities.

- The St-stack having least degree.
- If nodes having least degree are more than one, select any one of them as next node. And add current node in the stack junction.

Step 3: Go to next node and delete it from the St-stack Mark the connecting edge between starting node and next node. Now next node becomes current node.



Step 4: As we proceed from one node to next, add all visited node is a stack called Hamil. Up to now, two nodes have been visited. Hamil stores nodes of complete Hc.

Step 5: While (Next Node ≠ starting Node)

if (we at a St-stack node and some other nodes are left which have not been processed yet) then

{

Mark the connecting edge between St-stack node and the starting node (Do not select starting node as next node).

}

Note: Marking an edge, similar to deleting it but it can be restored while backtracking, "Adjacent" means adjacent to current node.

/* In selection conditions below, we select next node from the adjacent nodes of current node, in order of the following conditions. After a node has been selected in one of the following steps, remaining steps will not work.*/

Selecting conditions:

If (we are standing at St-stack node while there is no-node (including St-stack nodes) left which has not been processed) then

{

Select starting node as next node.

}

If (There is an adjacent node of degree 2, to the current node) then

{

If (There are more than one adjacent node having degree 2) then

Call Backtrack( )

Else if (There is only one adjacent node of degree 2)

then

{

Select that node as next node

If (this selected node is St-stack node) then

{

If (one St-stack node left while all other nodes of graph have been processed) then

The selected node is next node.

If (one St-stack node left while some other node of graph exists that have not been processed) then

call Backtrack( )

}

}
}
If (There is one St-stack node adjacent to current node) then
{
If (The stack (St-stack) have more than one nodes) then
Pick this node as next node and delete it from St-stack.
Else if (The stack (St-stack) have onlyone node left) then
{
// there are two possibilities
If (If all nodes of graph processed) then
choose this node as next node and delete it from St-stack.
If (Some nodes of graph are left, to be processed) then mark this connecting edge (b/w current node and St-stack node (not taking it as current node)
}
}
else if (there are more than one St-stack nodes adjacent to current node)
Select the St-stack node with least degree.
//if there are more than one St-stack nodes of least degree.
//then do not add this in stack "junction".
//fourth condition selects a node with least degree
If (next node has not been selected yet in previous steps) then
{
select a node with least degree.
If (there are more than one nodes having least degree) then
Place this current node in a stack called "junction"
}
/* A next node has been selected because there is no appropriate adjacent node) then
Call Backtrack ( ).
(/* A next node has been selected in previous step next node will become current node, after following changes*/

- All the edges from current node (on which we are standing) to all its adjacent nodes are marked, this path in future will not be taken. It means there is no path from any vertex to come back at this node.
- Decrement the degree of all adjacent nodes by one (except the previous node form which we have shifted at this node).
- Add current node to stack hamil
- Now rename next node as current node.

End while.

Procedure Backtrack ( )
{
If (the stack' junction' is empty) then stop processing and prompt that it is not Hamiltonian
circuit.
//Otherwise all the following reverse processes are carried out:

- Take the last node from stack junction as current node.
- Select this current node from stack hamil.

/* From this node till the last stored node in stack hamil, the reverse changes occur, these changes are described below. If were the current node is starting node then neither increment degree of St – stack nodes work demark the edges. We increment degree of only those nodes which we have decremented in previous processing and demark only those edges which have been marked*/

- In graph start from current node and start demarking all previously marked edges. to the last node of hamil.
- Increment degrees of adjacent nodes by one.
- After all the reverse changes, the next alternative node adjacent to current node is chosen as next node.
- If there is no more alternate path after selecting next node, this node is deleted from stack junction.
- If current node is starting node then after selecting next node, delete current node from the stack junction (As we are checking only two paths form start, if blocking condition occurs on both paths HC will not exist).
- Then new next selected is sent back for further processing

}
End of algorithm.

B.E./B.Tech. DEGREE EXAMINATION
APRIL/MAY 2011

Fourth Semester

(Regulation 2008)

Computer Science and Engineering

2251 – DESIGN AND ANALYSIS OF ALGORITHM

(Common to Information Technology)

Time : Three hours **Maximum : 100 marks**

Answer All Questions

PART A – (10 × 2 = 20 marks)

1. Using the step count method analyze the time complexity when 2 m × n matrices are added.

2. An array has exactly n nodes. They are filled from the set {0, 1, 2,...,n-1, n}. There are no duplicates in the list. Design an O(n) worst case time algorithm to find which one of the elements from the above set is missing in the array.

3. Show the intermediate steps when the numbers 123, 23, 1, 43, 54, 36, 75, 34 are sorted using merge sort.

4. Devise an algorithm to make a change for 1655 using the Greedy strategy. The coins available are {1000, 500, 100, 50, 20, 10, 5}.

5. Write the difference between Greedy Method and Dynamic Programming.

6. Write an algorithm to find the shortest path between all pairs of nodes.

7. Define the chromatic number of a graph.

8. Draw a graph with a cycle but with no Hamiltonian cycle.

9. Define a strongly connected digraph and give the minimum in degree of all the nodes in the graph.

10. Perform depth first and breadth first search on the following graph and find all the nodes reachable from ‘a’.

PART B — (5 × 16 = 80 marks)

11. (a) Write the recursive and non-recursive versions of the factorial function. Examine how much time each function requires as 'n' becomes large.

Or

(b) (i) Explain asymptotic notations in detail. (8)

(ii) Write an algorithm for linear search and analyze the algorithm for its time complexity. (8)

12. (a) Write an algorithm to perform binary search on a sorted list of elements. Analyze the algorithm for the best case, average case and worst case.

Or

(b) Using the divide and conquer approach to find the maximum and minimum in a set of 'n' elements. Also find the recurrence relation for the number of elements compared and solve the same.

13. (a) Use function OBST to compute w(i, j), r(i, j), and c(i, j), 0<=i<j<=4, for the identifier set ( ) 4 3 2 1 , , , a a a a = (cout, float, if, while) with p(1) = 1/20, p(2) = 1/5, p(3) = 1/10, p(4) = 1/20, q(0) = 1/5, q(1) = 1/10, q(2) = 1/5, q(3) = 1/20, and q(4) 1/20. Using the r(i, j)'s, construct the optimal binary search tree.

Or

(b) Using dynamic approach programming, solve the following graph using the backward approach.

14. (a) Using backtracking, find the optimal solution to a knapsack problem for the knapsack instance n = 8, m = 110, (p1, p2. ... p7) = (11, 21, 31, 33, 43, 53, 55, 65) and (w1, w2,...,w7) = (1, 11, 21, 33, 43, 53, 55, 65).

Or

(b) Write an algorithm for N QUEENS Problem and Trace it for n=6.

15. (a) Write the Kruskal's algorithm apply it to find a minimum spanning tree for the following graph :

Or

(b) Write short notes on NP-hard and NP-completeness.

Solutions

PART A

1. For computing the step count for the matrix addition, consider the following code:-

```
for(i = 0; i < m; i++ )
{
 for(j = 0; j< n;j++ )
  {
   c[i][j] = A[i][j] + B[i][j]
  }
}
```

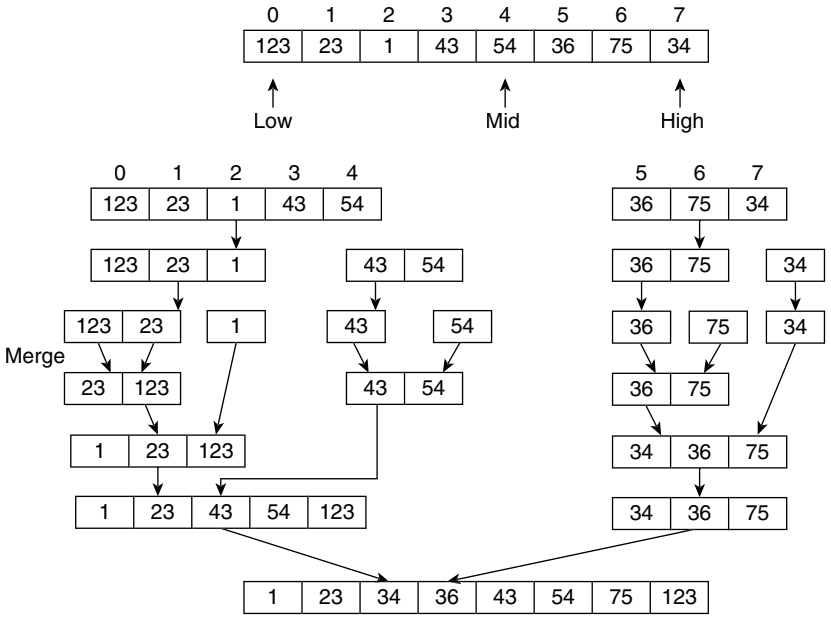
Statement	Step Count
i = 0	1
i < m	m + 1
i++	m + 1
j = 0	m
j = n	(n + 1)m
j++	(n + 1)m
c[i][j] = A[i][j] + B[i][j]	n(m)
Total	3mn + 5m + 3.

The time Complexity = 0(MN)

2. Missed Element (Array[n])

```
{
 Total = [n + 1] *(n + 2)/2.
 for (i = 0; i < n; i++ )
 total = a[i];
 return Total
}
```

3.



4. For the change of 1655 the coins that can be used are

1000 = 1 coin
500 = 1 coin
100 = 1 coin
50 = 1 coin
5 = 1 coin
1655 = 5 coins

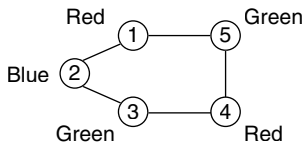
5.

Greedy Method	Dynamic Programming
1. Greedy method is used for obtaining optimum solution.	Dynamic programming is also for obtaining optimum solution.
2. In greedy method a set of feasible solutions and the picks up the optimum solution	There is no special set of feasible solutions in this method.
3. In greedy method the optimum selection is without revising previously generated solutions.	Dynamic programming consider all possible sequence in order to obtain the optimum solution.
4. In greedy method there is no as such guarantee of getting optimum solution.	It is guaranteed that the dynamic programming will generate optimal solution using principal of optimality.

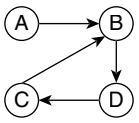
6. All pair(int w, int a)

```
{
   for (i = 1; i < n; i + + )
{
   for (j  = 1; j < n; j + + )
{
   a[i][j]  = w[i][j]
   for (k  = 1; k < n;k + + )
{
   for (i  = 1; i < n;i + + )
{
   for (j  = 1; j < n;j + + )
{
   A[i][j]  = min(A[i][j],A[i][k] + A[k][j]);
}
}
}
}
```

7. Let $G = (V,E)$ is a graph. Let $V \rightarrow \{1, 2, \ldots m\}$. defined for all the vertices of the graph. This is a function used for coloring the graph. The coloring should be such that no two adjacent vertices should have the same color. For ex. the function f returns m = 3 for coloring this graph. The number 3 is called the chromatic number.

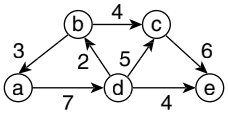


8. The Graph *G* contains cycle but no Hamiltonian cycle



9. In undirected graphs, two vertices are connected if they have a path connecting them. But in the case of directed graphs vertex *a* is strongly connected to *b* if there exists two paths one from *a* to *b* and another from *b* to *a*.

10.



Depth first search -ad_{bce}
Breadth first search-ad_{bec}.

PART B

11. (a) Recursive

```
# include<stdio.h>
#include<conio.h>
void main()
{
int n;
printf("\ n Enter the value of n");
scanf("%d", &n);
printf ("\n The factorial is  = %d", fact(n));
}
int fact(int n)
{
int a,b;
if(n  =  = 0)
return 1;
a  = n – 1,
b  = fact(a);
return(n*b);
}
```

Non- recursive

```
# include<stdio.h>
#include<conio.h>
void main()
{
int n;
printf("\n Enter the value of n");
scanf("%d", &n);
printf("\n The factorial is = %d", fact(n));
}
int fact(int n)
        {
```

```
int prod = 1;
int x;
x = n;
while(x > 0)
{
 prod = prod *x;
 x – – ;
 }
 return prod;
 }
```

In the recursive routine when each call is made, it gets stored in the stack the larger value of n may cause the stack overflow error.

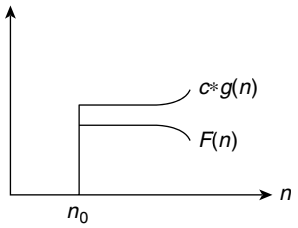
11. (b) (i) Asymptotic Notations

Using asymptotic notations we can give time complexity as "fastest possible", "slowest possible" or " average time".

Various notations such as Ω, θ and 0 used are called asymptotic notations.

Big Oh Notation

Let $F(n)$ and $g(n)$ be two non-negative functions. Let no and constant c are two integers such that no denotes some value of inputs and $n > n_0$. Similarly c is some constant such that $c > 0$.
$F(n) \leq c*g(n)$. then *f(n)* is big oh of *g(n)*.

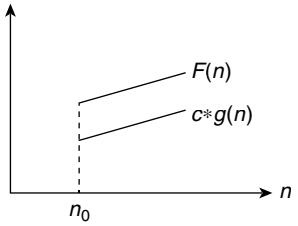


Ex:
$F(n) = 2n + 2$
$g(n) = n^2$
$n = 1$ then
$F(n) = 2(1) + 2 = 4.$
$g(n) = n^2 = 1^2$
$g(n) = 1.$
$F(n) > g(n).$

Omega Notation

A function $F(n)$ is said to be in $\Omega\, g(n)$ if f(n) is bounded below by some positive constant multiple of $g(n)$ such that

$$F(n) \geq c * g(n) \text{ For all } n \geq n_0.$$



Ex: $F(n) = 2n^2 + 5$ $g(n) = 7n$

$n = 0,$

$F(n) = 2(0)^2 + 5$

$= 5$

$g(n) = 7(0)$

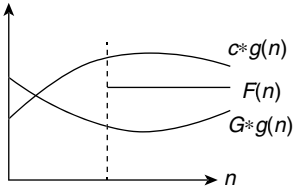
$= 0$ (ie) $F(n) > g(n)$.

ϕ Notation:

Let $F(n)$ and $g(n)$ be two non-negative functions there are two positive constants namely

c_1 and c_2 such that

$c_1\, g(n) \leq (n) \leq c_2\, g(n)$.



Ex: $F(n) = 2n + 8$, $g(n) = 7n$, $n \geq 2$

$5n \leq 2n + 8 \leq 7n$

Here $c_1 = 5$ & $c_2 = 7$ with $n_0 = 2$

The theta notation is more precise with big oh & omega.

11. (b) (ii) **Linear Search**

```
Linear Search (x[0] to x[n − 1], key)
{
for (i = 0; i < n − 1; i + + )
{
```

```
if (x[i] = = Key)
return i;
}}
```

Time Complexity:

Best Case: $C_{best} = 1$

Worst *C*ase

$C_{worst} = n$

Average Case

$C_{avg}(n)$ Probability of Successful Search + Probability of Unsuccessful Search

$$\text{ie } C_{avg}(n) = [1\ \frac{p}{n} + 2\ \frac{p}{n} + ...i\frac{p}{n}] + n(1-p)$$

$$= \frac{p}{n}[1+2+...+i...n] + n(1-p)$$

$$= \frac{p}{n}\ \frac{(n)(n+)}{2} + n(1-p)$$

$$= \frac{p(n+1)}{2} + n(1-p)$$

If $p = 0$

$C_{avg}(n) = 0(n + 1)/2 + n(1 - 0)$

$= n.$

If $p = 1$

$C_{avg}(n) = 1(n + 1)/2 + n(1 - 1)$

$= (n + 1)/2.$

Best Case	Worst Case	Average Case
0(1)	0(n)	0(n)

12. (a)

```
BinSearch (A[0,…n − 1],key)
{
 low = 0
 high = n − 1
 while(low < high)
 {
  m = (low & high) /2.
  if(key = A[m])
  return m
  else if (key < A[m])
```

```
  high = m − 1
else
low = m + 1
}
return − 1
}
```

Worst Case:

$C_{worst}(n) = C_{worst}(n/2) + 1$ for h > 1

Time required to Compare left Sublist or right Sublist. — One comparison made with Middle element.

$C_{worst}(1) = 1$

$C_{worst}(n) = C_{worst}(n/2) + 1$ (1)

$C_{worst}(1) = 1$ (2)

Assume $n = 2^k$.

$C_{worst}(2^k) = C_{worst}(2^{k/2}) + 1$

$C_{worst}(2^k) = C_{worst}(2^{k-1}) + 1$ (3)

Using backward substitution method.

$C_{worst}(2^{k-1}) = C_{worst}(2^{k-2}) + 1$

Eqn (3) becomes

$C_{worst}(2^k) = [C_{worst}(2^{k-2}) + 1] + 1$

$= C_{worst}(2^{k-2}) + 2$

$C_{worst}(2^k) = [C_{worst}(2^{k-3}) + 1] + 2$

$C_{worst}(2^k) = C_{worst}(2^{k-3}) + 3$

⋮

$C_{worst}(2^k) = C_{worst}(1) + k$ (4)

$C_{worst}(1) = 1$

$C_{worst}(2^k) = 1 + k.$

$C_{worst}(n) = 1 + \log_2 n$

$C_{worst}(n) = \log_2 n$ for $n > 1$

Worst Case of binary search is $\theta(\log_2 n)$

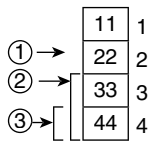
Average Case:

To obtain average case we will consider some sample of input n.

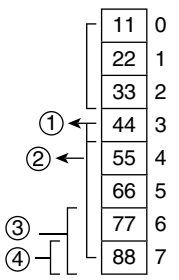
If $n = 1$ only element 11 is there.

1→ [11]

If $n = 4$ and search key = 44



M = (0 + 3)/2 = 1
A[1] = 22 and 22 < 44
Search right Sublist, again
M = (2 + 3)/2 = 2
Again A[2] = 33 and 33 < 44 we divide list.
In right sublist A[4] = 44 and key is 44
Thus total 3 comparisons are made to search 44.
If n = 8 & search key = 88.



m = (0 + 7)/2 = 3
As A[3] = 44 & 44 < 88 search right sublist, again.
m = (4 + 7)/2 = 5.
Again A[5] = 66 and 66 < 88 search right sublist.
again.
m = (6 + 7) /2 = 6.
But A[6] = 77 and 77 < 88
Then search right sublist.
m = (7 + 7) /2 = 7.
A[m] = A[7] = 88. Is A[7] = 88.
Yes, the desired element is present.
Thus total 4 comparison are made to search 88

n	**Total Comparison**
1	1
2	2
4	3
8	4
10	5

$$\log_2 n + 1 = C.$$

For instance if n = 2.

$$\log_2 2 = 1.$$
$$C = 1 + 1$$
$$C = 2.$$

Similarly if n = 8, then

$$C = \log_2 n + 1$$
$$= \log_2 8 + 1$$
$$= 3+1$$
$$C = 4.$$

Average case time Complexity of binary Search is $\theta(\log n)$.

Best Case	Average Case	Worst Case
$\theta(1)$	$\theta(\log_2 n)$	$\theta(\log_2 n)$

12. (b)

Algo:

```
Max_val(A[1…n])
{
 max = A[1]
 for (i = 2; i < n; i + + )
{
 If(A[i] > max)
 max = A[i]
 }
 return max
 }
```

Algo:

```
Min_val(A[1…n])
{
 min = A[1]
 for (i = 2; i < n; i + + )
{
 If(min > A[i])
 min = A[i]
}
 return min;
 }
```

Algo:

```
Max_ Min (A[1…n], Max, Min)
{
 Max = Min = A[1]
 For (i = 2; i < n; i + + )
{
 if (A[i] > Max)
 Max = A[i];
 if (A[i] < Min)
 Min = A[i];
 }
 }
```

13. (a) We will multiply values of p's and Q' s by 20 for convenience. Hence $p(1) = 1$, $p(2) = 4$, $p(3) = 2$, $p(4) = 1$, $Q(0) = 4$, $Q(1) = 2$, $Q(2) = 4$, Q(3) = 1, Q(4) = 1

Now we will compute,

$i = q$; $R_{ii} = 0$, $c_{ii} = 0$

$W_i, i + 1 = q_i + q(i + 1) + p(i + 1)$

$R_i, i + 1 = i + 1$

$C_i, i + 1 = q_i + q(i + 1) + p(i + 1)$

Let $i = 0$

$W_{00} = q_0 = 4$

$W_{11} = q_1 = 2$

$W_{22} = q_2 = 4$

$W_{33} = q_3 = 1$

$W_{44} = q_4 = 1$

$W_{01} = q_0 + q_1 + p_1$

$W_{01} = 4 + 2 + 1 = 7$

$W_{12} = q_1 + q_2 + p_2$

$= 2 + 4 + 4$

$W_{12} = 10$

$W_{23} = q_2 + q_3 + p_3$

$= 4 + 1 + 2$

$W_{23} = 7$

$W_{34} = q_3 + q_4 + p_4$

$= 1 + 1 + 1$

$W_{34} = 3$

We will use – Formula

$W_{ij} = W_{ij} - 1 + p_j + q_j$

let $i = 0, j = 2$

$W_{02} = W_{01} + p_2 + q_2$

$= 7 + 4 + 4$

$W_{02} = 15$

$i = 1\ j = 3$

$W_{13} = W_{12} + p_3 + q_3$

$= 10 + 2 + 1$

$W_{13} = 13$

$i = 2 \quad j = 4$

$W_{24} = W_{23} + p_4 + q_4$

$= 7 + 1 + 1$

$W_{24} = 9$

Let $i = 2\ j = 4$

$W_{24} = W_{23} + p_4 + q_4$

$= 7 + 1 + 1$

Let $i = 0\ j = 3$

$W_{03} = W_{02} + p_3 + q_3$

$= 15 + 2 + 1$

$W_{03} = 18$

Let $i = 1\ j = 4$

$W_{14} = W_{13} + p_4 + q_4$

$= 13 + 1 + 1$

$W_{14} = 15$

Let $i = 0 \quad j = 4$

$W_{04} = W_{03} + p_4 + q_4$

$= 18 + 1 + 1$

$W_{04} = 20$

The tabular expressions for values of W are as shown below.

$\rightarrow i$

$j\text{-}i \downarrow$	0	1	2	3	4
0	$W_{00} = 4$	$W_{11} = 2$	$W_{22} = 4$	$W_{33} = 1$	$W_{44} = 1$
1	$W_{01} = 7$	$W_{12} = 10$	$W_{23} = 7$	$W_{34} = 3$	
2	$W_{02} = 15$	$W_{13} = 13$	$W_{24} = 9$		
3	$W_{03} = 18$	$W_{14} = 15$			
4	$W_{04} = 20$				

The computation of C and R values is done

$C_{00} = 0$ $\quad C_{33} = 0$

$R_{00} = 0$ $\quad R_{33} = 0$

$C_{11} = 0$ $\quad C_{44} = 0$

$R_{11} = 0$ $\quad R_{44} = 0$

$C_{22} = 0$

$R_{22} = 0$

For computation of $Ci, i + 1$ and R_i, i + 1

we will use formula

$Ci, i + 1 = q_i + q(i + 1) + P(i + 1)$

$R_i, i + 1 = i + 1$

$i = 0$ then

$C_{01} = 7$

$C_{12} = 10$

$C_{23} = 7$

$C_{34} = 3$

To compute Cij and Rij we will use formula

$C_{ij} = i < k^{Min} \leq j\{C^{(i,k-1)+0kj}\} + w_{ij}$

$R_{ij} = k$

values of k lies between $R_i, j - 1$, and $R\ (i + 1), j$

Let us compute C_{02}

Hence $i = 0\ j = 2$ the values of k lies between $R_{01} = 1$ to $R_{12} = 2$

$C_{02} = 10$

$C_{02} = 7 \rightarrow$ Minimum value of C

$R_{02} = 2$ then $C_{02} = W_{02} + C_{02} = 15 + 7 = 22$

$R_{12} = 2\ R_{23} = 3$

$C_{13} = 7 \rightarrow$ Minimum value.

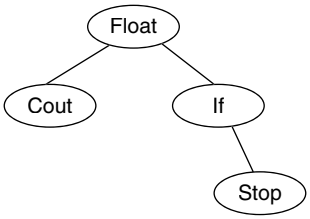
$C_{24} = 3 \rightarrow$ Minimum value.

$C_{03} = 32$

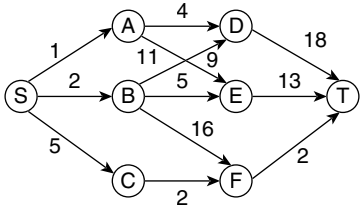
$C_{14} = 27 \rightarrow$ Minimum value

	0	1	2	3	4
0	$C_{00} = 0$ $R_{00} = 0$	$C_{11} = 0$ $R_{11} = 0$	$C_{22} = 0$ $R_{22} = 0$	$C_{33} = 0$ $R_{33} = 0$	$C_{44} = 0$ $R_{44} = 0$
1	$C_{01} = 7$ $R_{01} = 1$	$C_{12} = 10$ $R_{12} = 2$	$C_{23} = 7$ $R_{23} = 3$	$C_{34} = 3$ $R_{34} = 4$	
2	$C_{02} = 22$ $R_{02} = 2$	$C_{13} = 20$ $R_{13} = 2$	$C_{24} = 12$ $R_{24} = 3$		
3	$C_{03} = 32$ $R_{03} = 2$	$C_{14} = 27$ $R_{14} = 2$			
4	$C_{04} = 39$ $R_{04} = 2$				

The final tree is



13. (b)



Backward approach

$b\ cost\ (i,j) = \min\ \{b\ cost\ (i-1, l) + C(l,j)\}$
$l \in V_i - 1$
$< l,j > \in t$

$i = 2,$

$b\ cost\ (2,A) = \min\ \{b\ cost\ (1,S) + C(S,A)\}$
$= \min\ \{0 + 1\}$
$= 1.$

$b\ cost\ (2,B) = \min\ \{b\ cost\ (1,S) + C(S,B)\}$
$= \min\ \{0 + 2\}$
$= 2.$

$b\ cost\ (2,C) = \min\ \{b\ cost\ (1,S) + C(S,C)\}$
$= \min\ \{0 + 5\}$
$= 5$

$i = 3,$

$b\ cost\ (3,D) = \min\ \{b cost\ (2,A) + C(A,D), b\ cost\ (2,B) + C(B,D)\}$
$= \min\ \{1 + 4, 2 + 9)$
$= 5.$

$b\ cost\ (3,E) = \min\ \{b cost\ (2,A) + C(A,E), b\ cost\ (2,B) + C(B,E)\}$
$= \min\ \{1 + 11, 2 + 5)$
$= 7.$

$b\ cost\ (3,F) = \min\ \{b cost\ (2,B) + C(B,F), b\ cost\ (2,C) + C(C,F)\}$
$= \min\ \{2 + 16, 5 + 2)$
$= 7.$

$i = 4,$

$b\ cost\ (4,T) = \min\ \{b\ cost\ (3,D) + C(D,T), b\ cost\ (3,E) + C(Z,T), b\ cost\ (3,F) + C(F,T)\}$
$= \min\ \{5 + 18, 7 + 13, 7 + 2)$
$= 9$

Path:

S – C – F – T

Minimum cost: 5 + 2 + 2 = 9.

14. (a)

Item	P_i	W_i	P_i/w
1	11	1	11
2	21	11	1.90
3	31	21	1.47
4	33	33	1

Continued

(*Continued*)

4	33	33	1
5	43	43	1
6	53	53	1
7	55	55	1
8	65	65	1

M = 110
Initially profit = –1
$p = 0$, $CW = 0$, Let $k = 0$,
$Cp = 11$, $CW = 1 < M$
$K = 1$, $P[i] = 21$, $w[i] = 1$,
$Cp = 32$, $CW = 12 < M$
$K = 2$, P[3] = 31, $w[3] = 21$
$Cp = 63$, $CW = 33 < M$
$K = 3$, $P[4] = 33$, $w[4] = 33$
$Cp = 96$, $CW = 66 < M$
$K = 4$, $P[5] = 43$, $w[5] = 43$
$Cp = 139$, $CW = 109 < M$
$K = 5$, $P[6] = 53$, $w[6] = 53$
$Cp = 192$, $CW = 162 > M$
Calculate upper bound value.
$U_b = Cp = 139$
$C = CW = 109$
$Ub = Ub + (1 - (c - m)/w[i])*P[i]$
$= 139 + (1 - (109 - 110)/w[6])*P[6]$
$= 141$
if we try for 7 & 8, we backtrack to $Cp = 139$, $Cw = 109$
Hence temp[t] = temp[7] = temp[8] = 0.
∴ Item 1,2,3,4,5 is selected.
Profit = 139

14. (b) Algorithm for NQueens and Trace if for $n = 6$.

Problem Statement:

The *n*-Queens problem is a generalization of the 8-queens (or) 4-queens problems. Now n queens are to be placed on an $n \times n$ chess-board so that no two queens attack each other; that is no two queens are on the same row, column or diagonal.

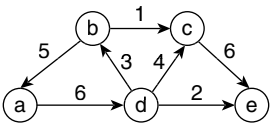
```
{
for i = 1to n do
{
If place (k,i) then
```

```
{
n[k] = i;
if (k = n) then write(x{1 :n]);
else N queens (k + 1,n);
 }
 }
 N queens (k – 1, n); i = x[k – 1] + 1;
 }
 Algorithm place(k,i.)
 {
  for j = 1to k – 1do
  if((x[j] = i) // Two queens in the same column.
  or(Abs(x[j] – i) = Abs(j – k))) // Two queens in the same diagonal.
  then return false;
  return true;
  }
```

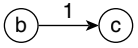
	1	2	3	4	5	6
1		Q				
2				Q		
3						Q
4	Q					
5			Q			
6					Q	

15. (a) Kruskal's algorithm:

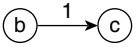
```
Kruskal (G)
{
    ecounter = 0;
    k = 0;
    While(ecounter < |v| − 1)
    {
        k = k + 1
        if E_TU {e_ik} is a cyclic
        E_T = E_TU {e_ik}; ecounter = ecounter + 1;
        return E_T;
    }
}
```



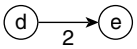
Ans: step 1:



Total cost = 1

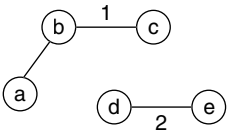


step 2:



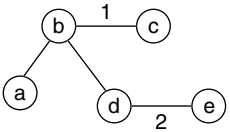
Total cost = 3

step 3:



Total cost = 8

step 4:



Total cost = 11

15. (b) NP Hard and NP Completeness

There are 2 groups in which a problem can be classified. First group consists of the problems that can be solved in polynomial time.

Ex – Sorting of elements & O (log n).

Second group consists of problems that can be solved in non-deterministic Polynomial time.

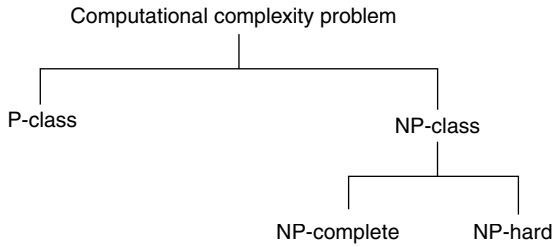
Ex – KnapSack problem O ($2^{n/2}$).

- Any problem for which answer is either yes or no is called decision problem.
- Any problem that involves the identification of optimal cost is called optimization problem.
- Definition of P- Problem that can be solved in polynomial time
  Ex – Sorting of elements.

- Definition of NP- It stands for non-deterministic polynomial time".

  Ex – Graph Colouring problem.

- The NP class problems can be further categorized into NP- complete & NP hard.



A problem D is called NP complete if

(i) It belongs to class NP

(ii) Every problem in NP can also be solved in polynomial time.

- If an NP- hard problem can be solved in polynomial time both them all NP – complete problem can also be solved in polynomial time.
- All NP –complete problem are NP-hard but all NP-hard problem cannot be NP- complete.
- The NP class problem are the decision problem that can be solved by non-deterministic polynomial Algorithm.
- Example of p Class problem.
  – Kruskal's algorithm.
- Example of NP Class problem.
  – Travelling salesman's problem.

Properties of NP –complete and NP-hard problems:

- As we know, p denotes the class of all deterministic polynomial language problem and NP denotes the class of all non-deterministic polynomial language problem.

$$P \leq NP.$$

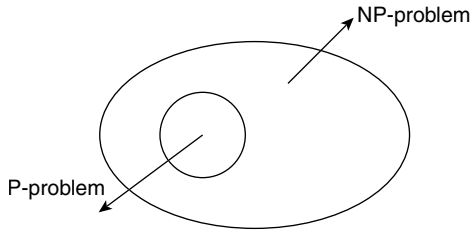
- The question of whether or not,

$$P = NP.$$

- Problem which are known to lie in P are often called as trace table. Problems which lie Outside of P are often termed as intractable.

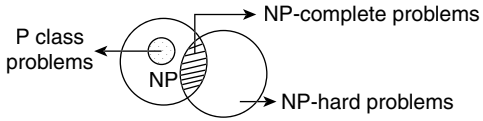
$$P = NP \text{ or } P \neq NP.$$

The relationship between P and NP is depicted by



- If P = NP. In 1971. S.A. COOK proved that a particular NP problem known as SAT has the property, that if it is solvable in polynomial time so are all NP problems. This is called NP –Complex Problem.
- Let *A* & *B* are two problems then problem *A* reduce to *B* if and only if there is a way to solve *A* be deterministic polynomial time algorithm using a deterministic algorithm that solves *B* in polynomial time.

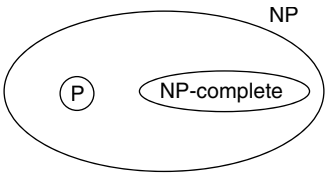
  A reduces to *B* can be denoted as $A \propto B$. In other words we can say that if there exists any polynomial time algorithm which solves *B* then we can solve A in Polynomial time. We can also share that if $A \propto B$ and $B \propto C$ then $A \propto C$



  Normally the decision problem are NP-complete but optimization problems are NP-hard. However if problem *A* is a decision problem and *B* is optimization problem then it is possible that $A \propto B$. For instance the knapsack decision problem then it is possible that $A \propto B$. For instance the knapsack decision problem can be Knapsack optimization problem

  There are some NP-hard problem that are not NP – complete. For eg: hating problem. The halting problem states that: "It is possible to determine whether an algorithm will ever halt or enter in a loop on certain input.

  Two problems *P* & *Q* are said to be polynomially equivalent if and only if $P \propto Q$ and $Q \propto P$.



B.E./B.Tech. DEGREE EXAMINATION, NOV/DEC 2010

Fourth Semester

Computer Science and Engineering

CS2251 − DESIGN AND ANALYSIS OF ALGORITHMS

(Common to Information Technology)

(Regulation 2008)

Time: Three hours **Maximum: 100 marks**

Answer All Questions

PART A − (10 × 2 = 20 Marks)

1. What are the metrics that are used to measure the complexity of an algorithm?
2. What is conditional asymptotic notation?
3. What kind of problems can be best solved using divide and conquer method?
4. What are the demerits of greedy algorithm?
5. State the principle behind dynamic programming.
6. What is a multi-stage graph?
7. What is graph coloring problem?
8. What is Hamiltonian path?
9. What are biconnected components?
10. When is a problem said to be NP-Hard?

PART B (5 × 16 = 80 marks)

11. (a) (i) What are asymptotic notations? Discuss briefly about various asymptotic notations? (8)

(ii) List the properties of Big-oh notation. (8)

Or

(b) What are recurrence equations? Explain how recurrence relations are solved? (16)

12. (a) Explain how merge-sort is carried out using divide and conquer method with an example. (16)

Or

(b) State what is container loading problem? Explain how greedy method is used to solve this problem. (16)

13. (a) With an example, show how Dynamic programming is used to find all-pairs shortest path is graph. (16)

Or

(b) Explain Travelling Salesman problem and show how dynamic programming is used to solve the problem. (16)

14. (a) What is backtracking? Explain how Backtracking technique is applied to solve the 8-queens problem. (16)

Or

(b) Explain how sum of subsets is computed using Backtracking technique with an example. 16)

15. (a) (i) Explain graph traversal algorithms with an example. (8)

(ii) Explain the technique used to find minimal spanning tree with an example. (8)

Or

(b) What is the principle behind Branch and bound technique? Solve 0/1 knapsack problem using branch and bound technique. (16)

Solutions

PART A

1. The two important categories of metrics that are used to measure the complexity of an algorithm are
   1. Space Complexity
   2. Time Complexity

2. Many algorithms are easier to analyze if initially we restrict our attention to instances whose size satisfies a certain condition, such as being apower of 2. Consider, for example, the divide and conquer algorithm for multiplying large integers.
   Let n be the size of the integers to be multiplied. The algorithm proceeds directly if $n = 1$, which requires a microseconds for an appropriate constant a. If $n>1$, the algorithm proceeds by multiplying four pairs of integers of size $n/2$ (or three if we use the better algorithm). Moreover, it takes a linear amount of time to carry out additional tasks. For simplicity, let us say that the additional work takes at most bn microseconds for an appropriate constant b.

3. Divide and conquer is applied to those algorithms that reduce each problem to only one subproblem, such as the binary search algorithm for finding a record in a sorted list.

4. Greedy algorithms don't work for all problems.
   Even though these algorithms are known for their simplicity, correct greedy algorithms can be intangible.
   It's easy to fool yourself into believing an incorrect greedy algorithm is correct.
   Using this algorithm is not an automatic process.

5. Dynamic programming is a technique for solving problems with overlapping sub problems. Typically, these sub problems arise from a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the same type.

   Rather than solving overlapping sub problems again and again, dynamic programming suggests solving each of the smaller sub problems only once and recording the results in a table from which we can then obtain a solution to the original problem.

6. A multistage graph is a graph G, denoted by $G = (V, E)$ with V partitioned into $K >= 2$ disjoint subsets such that if (a, b) is in E, then a is in Vi, and b is in $Vi+1$ for some subsets in the partition; and $| V1 | = | VK | = 1$.

The vertex *s* in *V*1 is called the source; the vertex t in *VK* is called the sink.G is usually assumed to be a weighted graph.The cost of a path from node vto node w is sum of the costs of edges in the path.
The "multistage graph problem" is to find the minimum cost path from *s* to *t*.

7. A coloring of a graph is an assignment of a color to each vertex of the graph so that no two vertices connected by an edge have the same color. It is not hard to see that our problem is one of coloring the graph of incompatible turns using as few colors as possible.

   The problem of coloring graphs has been studied for many decades, and the theory of algorithms tells us a lot about this problem. Unfortunately, coloring an arbitrary graph with as few colors as possible is one of a large class of problems called "NP- complete problems," for which all known solutions are essentially of the type "try all possibilities." The graph-coloring problem is to determine the minimum number of colors needed to color a given graph.

8. A *Hamiltonian path* or *traceable path* is a path that is visited each vertex exactly once.

9. A biconnected component (or 2-connected component) is a maximal biconnected subgraph. Any connected graph which decomposes into a tree of biconnected components is called the block tree of the graph.

10. NP stands for Non-deterministic Polynomial time.
    This means that the problem can be solved in Polynomial time using a Non-deterministic Turing machine (like a regular Turing machine but also including a non-deterministic "choice" function).

    Basically, a solution has to be testable in poly time. If that's the case, and a known NP problem can be solved using the given problem with modified input (an NP problem can be reduced to the given problem) then the problem is NP complete.
    A problem *H* is NP-hard if and only if there is an NP-complete problem L that is polynomial time reducible to *H* (i.e., $L \leq_p H$)

PART B

11. (a) (i) **Asymptotic Notations**

    Step count is to compare time complexity of two programs that compute same function and also to predict the growth in run time as instance characteristics changes. Determining exact step count is difficult and not necessary also. Because the values are not exact quantities. We need only comparative statements like $c1n2 \leq tp(n) \leq c2n2$.

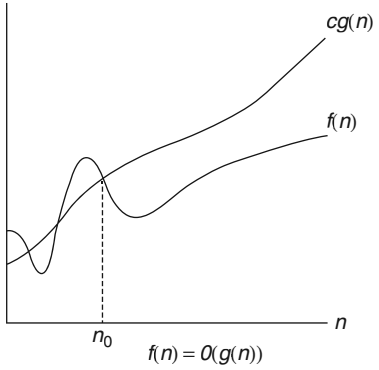
For example, consider two programs with complexities $c1n2 + c2n$ and $c3n$ respectively. For small values of n, complexity depend upon values of $c1$, $c2$ and $c3$. But there will also be an n beyond which complexity of $c3n$ is better than that of $c1n2 + c2n$.This value of n is called break-even point. If this point is zero, c3n is always faster (or at least as fast). Common asymptotic functions are given below.

Function	**Name**
1	Constant
log *n*	Logarithmic
n	Linear
n log n	*n* log *n*
*n*2	quadratic
*n*3	Cubic
2*n*	Exponential
N!	factorial

Big 'Oh' Notation (O)

$O(g(n)) = \{f(n)$: there exist positive constants c and $n0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n0\}$

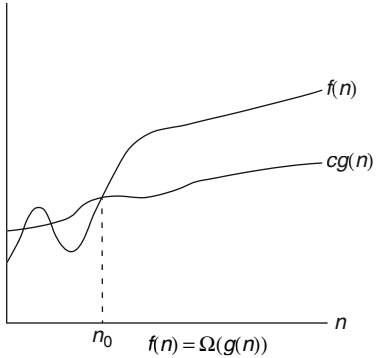
It is the upper bound of any function. Hence it denotes the worst casecomplexity of any algorithm. We can represent it graphically as



Omega Notation (Ω)

Ω ($g(n)$) = { $f(n)$: there exist positive constants c and $n0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n0$ }

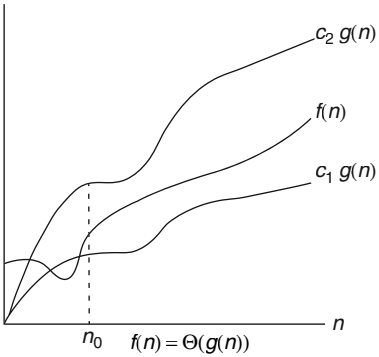
It is the lower bound of any function. Hence it denotes the best case complexity of any algorithm. We can represent it graphically as



Theta Notation (Θ)

Θ($g(n)$) = {$f(n)$: there exist positive constants $c1$, $c2$ and $n0$ such that $c1g(n) \leq f(n) \leq c2g(n)$ for all $n \geq n0$ }

If $f(n)$ = Θ($g(n)$), all values of n right to $n0$ $f(n)$ lies on or above $c1g(n)$ and on or below $c2g(n)$. Hence it is asymptotic tight bound for $f(n)$.



Little-O Notation

For non-negative functions, $f(n)$ and g(n), f(n) is little o of $g(n)$ if and only

if $f(n) = O(g(n))$, but $f(n) \neq \Theta(g(n))$. This is denoted as "$f(n) = o(g(n))$".

This represents a loose bounding version of Big $O.g(n)$ bounds from the top, but it does not bound the bottom.

Little Omega Notation

For non-negative functions, $f(n)$ and $g(n)$, $f(n)$ is little omega of $g(n)$ if and only if $f(n) = \Omega(g(n))$, but $f(n) \neq \Theta(g(n))$. This is denoted as "$f(n) = \Omega(g(n))$".

Much like Little Oh, this is the equivalent for Big Omega. g(n) is a loose lower boundary of the function f(n); it bounds from the bottom, but not from the top.

(a) (ii) **Properties of Big-Oh Notation**

Generally, we use asymptotic notation as a convenient way to examine what can happen in a function in the worst case or in the best case. For example, if you want to write a function that searches through an array of numbers and returns the smallest one:

```
function find-min(array a[1..n])
let j :=
for i := 1 to n:
j := min(j, a[i])
repeat
return j
end
```

Regardless of how big or small the array is, every time we run find-min, we have to initialize the i and j integer variables and return j at the end. Therefore, we can just think of those parts of the function as constant and ignore them.

So, how can we use asymptotic notation to discuss the find-min function? If we search through an array with 87 elements, then for loop iterates 87 times, even if the very first element we hit turns out to be the minimum. Likewise, for n elements, for loop iterates n times. Therefore we say the function runs in time $O(n)$.

```
functionfind-min-plus-max(array a[1..n]
// First, find the smallest element in the array
let j := ;
for i := 1 to n:
j := min(j, a[i])
repeat
let minim := j
// Now, find the biggest element, add it to the smallest and
j := ;
for i := 1 to n:
j := max(j, a[i])
repeat
let maxim := j
```

```
// return the sum of the two
return minim + maxim;
end
```

What's the running time for find-min-plus-max? There are two for loops, that each iterate n times, so the running time is clearly $O(2n)$. Because 2 is a constant, we throw it away and write the running time as $O(n)$. Why can you do this? If you recall the definition of Big-O notation, the function whose bound you're testing can be multiplied by some constant. If $f(x) = 2x$, we can see that if $g(x) = x$, then the Big-O condition holds. Thus $O(2n) = O(n)$. This rule is general for the various asymptotic notations.

(b) When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence equation

$X_{n+1} = rx_n (1 - x_n)$ (1)

Solving Recurrence Equation

The substitution method for solving recurrences entails two steps:

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

The name comes from the substitution of the guessed answer for the function when the inductive hypothesis is applied to smaller values. This method is powerful, but it obviouslycan be applied only in cases when it is easy to guess the form of the answer. The substitution method can be used to establish either upper or lower bounds on a recurrence. As an example, let us determine an upper bound on the recurrence

$T(n) = 2T(\lfloor n/2 \rfloor) + n$, (2)

Which is similar to recurrences (1) and (2), We guess that the solution is $T(n) = O(n \lg n)$.Our method is to prove that $T(n) \leq$ cnlg n for an appropriate choice of the constant $c > 0$. We start by assuming that this bound holds for $\lfloor n/2 \rfloor$, that is, that $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$.

Substituting into the recurrence yields

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn\lg(n/2) + n \\ &= cn\lg n - cn\lg 2 + n \\ &= cn\lg n - cn + n \\ &\leq cn\lg n, \end{aligned}$$

Where the last step holds as long as $c \geq 1$

Mathematical induction now requires us to show that our solution holds for the boundary conditions. Typically, we do so by showing that the boundary conditions are suitable as base cases for the inductive proof. We must show that we can choose the constant c large enough so that the bound $T(n) = \text{cnlg n}$ works for the boundary conditions as well. This requirement can sometimes lead to problems. Let us assume, for the sake of argument, that $T(1) = 1$ is the sole boundary condition of the recurrence. Then for $n = 1$, the bound $T(n) = cnlg\ n$ yields $T(1) = c1 \lg 1 = 0$, which is at odds with $T(1) = 1$. Consequently, the base case of our inductive proof fails to hold.

This difficulty in proving an inductive hypothesis for a specific boundary condition can be easily overcome. For example, in the recurrence (4.4), we take advantage of asymptotic notation only requiring us to prove $T(n) = cnlg\ n$ for $n \geq n0$, where n0 is a constant of our choosing. The idea is to remove the difficult boundary condition $T(1) = 1$ from consideration

A recurrence relation is an equation that recursively defines a sequence. Each term of the sequence is defined as a function of the preceding terms. A difference equation is a specific type of recurrence relation.

12. (a) Merge sort.
    1. As an example, let us consider the problem of computing the sum of n numbers a_0, $an-1$.
    2. If $n > 1$, we can divide the problem into two instances of the same problem.
       They are To compute the sum of the first $\lfloor n/2 \rfloor$numbers
    3. To compute the sum of the remaining $[n/2]$numbers. (Of course, if $n = 1$, we simply return a0 as the answer.)
    4. Once each of these two sums is computed (by applying the same method, i.e., recursively),we can add their values to get the sum in question. i.e., $a0 + \ldots + an-1 = (a0 + \ldots + a\lfloor n/2 \rfloor) + (a\lfloor n/2 \rfloor + \ldots + an-1)$
    5. More generally, an instance of size n can be divided into several instances of size nib, with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$).
    6. Assuming that size n is a power of b, to simplify our analysis, we get the following recurrence for the running time $T(n).T(n) = aT(n/b) + f(n)$
    7. This is called the general divide and-conquer recurrence. Where $f(n)$ is a function that accounts for the time spent on dividing

the problem into smaller ones and on combining their solutions. (For the summation example, $a = b = 2$ and $f(n) = 1$.

8. Obviously, the order of growth of its solution $T(n)$ depends on the values of the constants a and b and the order of growth of the function $f(n)$. The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem.

(b) The greedy algorithm constructs the loading plan of a single container layer by layer from the bottom up. At the initial stage, the list of available surfaces contains only the initial surface of size L × W with its initial position at height 0. At each step, the algorithm picks the lowest usable surface and then determines the box type to be packed onto the surface, the number of the boxes and the rectangle area the boxes to be packed onto, by the procedure select layer.

The procedure select layer calculates a layer of boxes of the same type with the highest evaluation value. The procedure select layer uses breadth-limited tree search heuristic to determine the most promising layer, where the breadth is different depending on the different depth level in the tree search. The advantage is that the number of nodes expanded is polynomial to the maximal depth of the problem, instead of exponentially growing with regard to the problem size. After packing the specified number of boxes onto the surface according to the layer arrangement, the surface is divided into up to three sub-surfaces by the procedure divide surfaces.

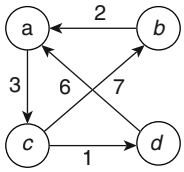
Then, the original surface is deleted from the list of available surfaces and the newly generated sub-surfaces are inserted into the list. Then, the algorithm selects the new lowest usable surface and repeats the above procedures until no surface is available or all the boxes have been packed into the container. The algorithm follows a similar basic framework.The pseudo-code of the greedy algorithm is given by the greedy heuristic procedure.

```
procedure greedy heuristic()
list of surface := initial surface of L × W at height 0
list of box type := all box types
while (there exist usable surfaces) and (not all boxes are
packed) do
select the lowest usable surface as current surface
set depth := 0
set best layer := select layer(list of surface, list of box
type, depth)
pack best layer on current surface
reduce the number of the packed box type by the packed
amount
```

set a list of new surfaces := divide surfaces(current surface,
best layer, list of box type)
delete current surface from the list of surfaces
insert each surface in list of new surfaces into list of
surfaces
end while

13. (a) ALL PAIR SHORTESET PATH

Given a weighted connected graph (undirected or directed), the all-pair shortest paths problem asks to find the distances (the lengths of the shortest paths) from each vertex to all other vertices. It is convenient to record the lengths of shortest paths in an n-by-n matrix D called the distance matrix: the element dij in the ith row and the jth column of this matrix indicates the length of the shortest path from the ith vertex to the jth vertex ($1 \le i,j \le n$). We can generate the distance matrix with an algorithm called Floyd's algorithm. It is applicable to both undirected and directed weighted graphs provided that they do not contain a cycle of a negative length.



(a)

$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array} \qquad D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

(b) (c)

Fig: (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

Floyd's algorithm computes the distance matrix of a weighted graph with vertices through a series of n-by-n matrices:

$D^{(0)},\ldots, D^{(k-1)}, D^{(k)},\ldots, D^{(n)}$

Each of these matrices contains the lengths of shortest paths with certain constraints on the paths considered for the matrix. Specifically, the element in the ith row and the jth column of matrix D ($k = 0,1,\ldots,n$) is equal to the length of the shortest path among all paths from the ith vertex to the jth vertex with each intermediate vertex, if any, numbered not higher than k. In particular, the series starts with $D(0)$, which does not allow any intermediate vertices in its paths; hence, $D(0)$ is nothing but the weight matrix of the graph. The last matrix in the series, $D(n)$, contains the lengths of the shortest paths among all pathsthat can use all n vertices as intermediate and hence is nothing but the distance matrix being sought.

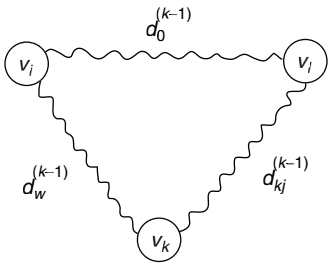
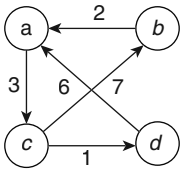


Fig: Underlying idea of Floyd's algorithm

$$d_{ij}^{k} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)}, d_{kj}^{(k-1)}\} \text{ for } k \geq 1, d_{ij}^{(0)} = w_{ij}$$

ALGORITHM *Floyd* (W [1..n, 1..n])
// Implements Floyd's algorithm for the all-pairs shortest-paths problem
// Input: The weight matrix W of a graph
// Output: The distance matrix of the shortest paths' lengths
$D \leftarrow W$ // is not necessary if W can be overwritten
for $k \leftarrow 1$ to n do
for $i \leftarrow 1$ to n do
for j $\leftarrow 1$ to n do
$D[i, j] \leftarrow \min \{D[i, j], D[i, k] + D[k, j]\}$
Return D



$$D^{(0)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$$D^{(1)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 1. i.e., just a (note two new shortest paths from b to c and from d to c).

$D^{(2)}=$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	9	7	0	1
d	6	∞	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 2. i.e., *a* and *b* (note a new shortest paths from *c* to *a*).

$D^{(3)}=$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	**16**	9	0

Lengths of the shortest paths with no intermediate vertices numbered not higher than 3. i.e., *a*, *b*, and *c* (note four new shortest paths from *a* to *b* from *a* to *d*, from *b* to *d*, and from *d* to *b*).

$D^{(4)}=$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	7	7	0	1
d	6	16	9	0

Lengths of the shortest paths with no intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note a new shortest paths from *c* to *a*).

Fig: Application of Floyd's algorithm to the graph shown. Updated elements are shown in bold.

(b) We will be able to apply the dynamic programming technique to instances of the traveling salesman problem, if we come up with a reasonable lower bound on tour lengths.

One very simple lower bound can be obtained by finding the smallest element in the intercity distance matrix *D* and multiplying it by the number of cities it. But there is a less obvious and more informative lower bound, which does not require a lot of work to compute.

It is not difficult to show that we can compute a lower bound on the length I of any tour as follows.

For each city i, $1 \le i \le n$, find the sum si of the distances from city i to the two nearest cities; compute the sums of these *n* numbers; divide the result by 2, and, it all the distances are integers, round up the result to the nearest integer:

l*b*=[*S*/2]

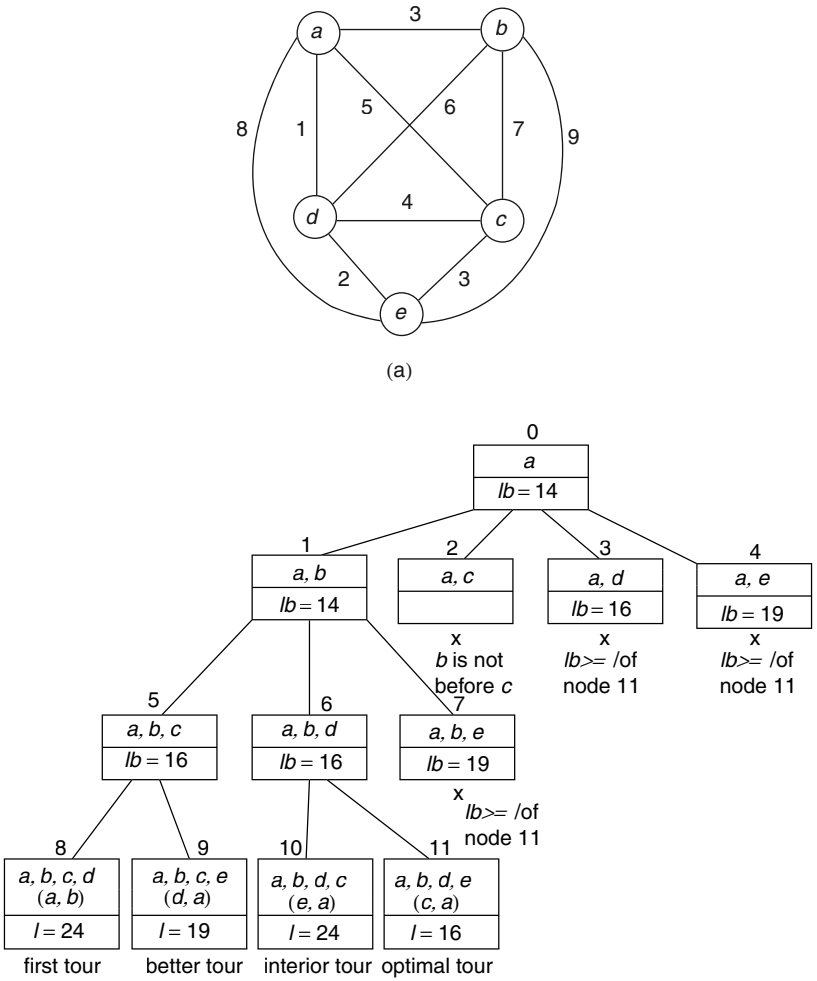


Fig: (a) Weighted graph.
(b) State-space tree of the branch-and- bound algorithm applied to this graph.

14. (a) Many problems are difficult to solve algorithmically. Backtracking makes it possible to solve at least some large instances of difficult combinatorial problems.

GENERAL METHOD

1. The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows.

2. If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component.
3. If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered. In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.

The problem is to place it queens on an n-by-n chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal. For n = 1, the problem has a trivial solution, and it is easy to see that there is no solution for n = 2and n =3. So let us consider the four-queen problem and solve it by the backtracking technique.

This can also be solved for eight-queen.

Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board presented in the following figure.

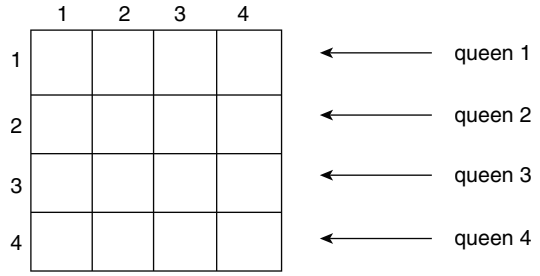


Fig: Board for the Four-queen problem

Steps to be followed

1. We start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1.
2. Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2,3), the square in row 2 and column 3.
3. This proves to be a dead end because there i no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2,4).
4. Then queen 3 is placed at (3,2), which proves to be another dead end.
5. The algorithm then backtracks all the way to queen 1 and moves it to (1,2). Queen 2 then goes to (2, 4), queen 3 to (3,1), and queen 4 to (4,3), which is a solution to the problem.

The state-space tree of this search is given in the following figure.

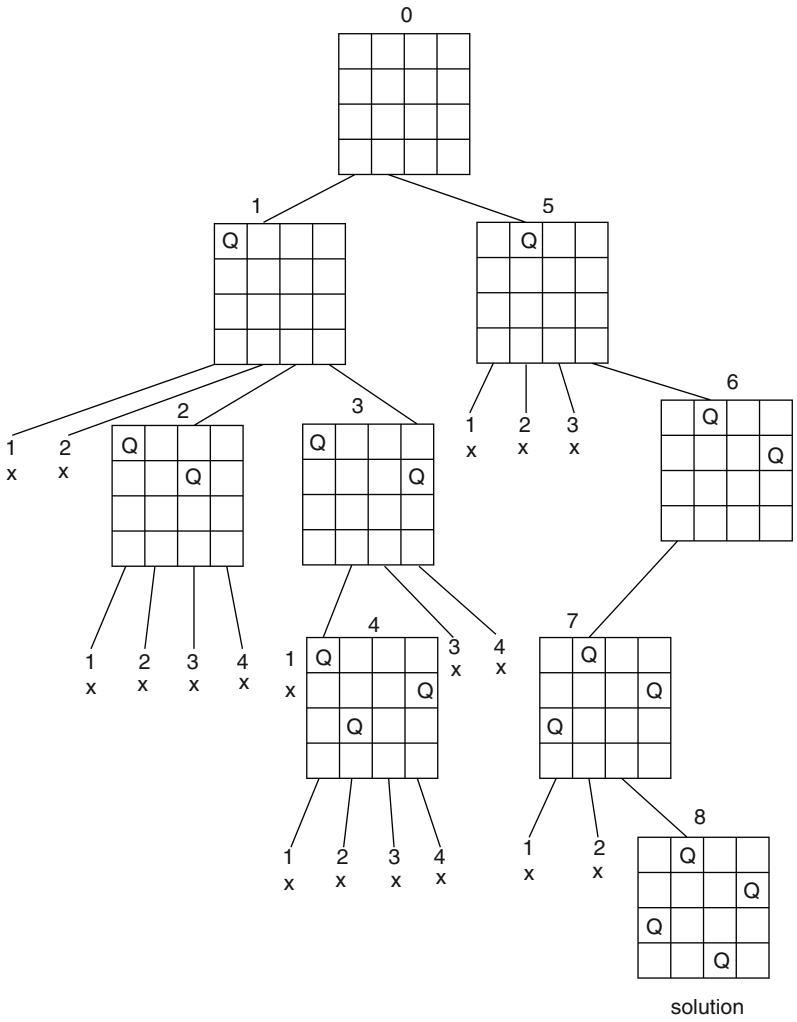


Fig: State-space tree of solving the four-queen problem by back tracking.

(b) SUBSET-SUM PROBLEM

Subset-Sum Problem is finding a subset of a given set $S = \{s1, s2\ldots. s_n\}$ of n positive integers whose sum is equal to a given positive integer d.

For example, for $S = \{1, 2, 5, 6, 8\}$ and $d = 9$, there are two solutions: $\{1, 2, 6\}$ and $\{1, 8\}$. Of course, some instances of this problem may have no solutions.

It is convenient to sort the set's elements in increasing order. So we will assume that

$s1 \leq s2 \leq \ldots\ldots \leq s_n$

The state-space tree can be constructed as a binary tree as that in the following figure for the instances S = (3, 5, 6, 7) and d = 15. The root of the tree represents the starting point, with no decisions about the given elements made as yet. Its left and right children represent, respectively, inclusion and exclusion ofs1 in a set being sought.

Similarly, going to the left from a node of the first level corresponds to inclusion of s2, while going to the right corresponds to its exclusion, and soon.Thus, a path from the root to a node on the ith level of the tree indicates which of the first i numbers have been included in the subsets represented by that node. Thus, a path from the root to a node on the ith level of the tree indicates which of the first i numbers have been included in the subsets represented by that node. We record the value of s the sum of these numbers, in the node, Ifs is equal to d. we have a solution to the problem.

We can either, report this result and stop or, if all the solutions need to he found, continue by backtracking to the node's parent.
If s is not equal to d, we can terminate the node as non-promising if either of the two inequalities holds:

$$s' + s_{i+1} > d \text{(the sum } s' \text{ is too large)}$$

$$s' + \sum_{j=i+1}^{n} s_j < d \text{(the sum } s' \text{ is too small).}$$

Pseudocode For Backtrack Algorithms
ALGORITHM *Backtrack* (X [1..i])
// Gives a template of a generic backtracking algorithm
// Input: X [1..i] specifies first i promising components of a solution
// Output: All the tuples representing the problem's solution
if X [1.. i] is a solution write X [1.. i]
else // see Problem 8
for each element $x \in S_{i+1}$ consistent with X [1.. i] and the constraints do
X [i + 1] ← x
Backtrack (x [1..i + 1])
(Applied to the instance S = (3, 5, 6, 7) and d = 15 of the subset-sum problem. The number inside a node is the sum of the elements already included in subsets represented by the node. The inequality below a leaf indicates the reason for its termination)

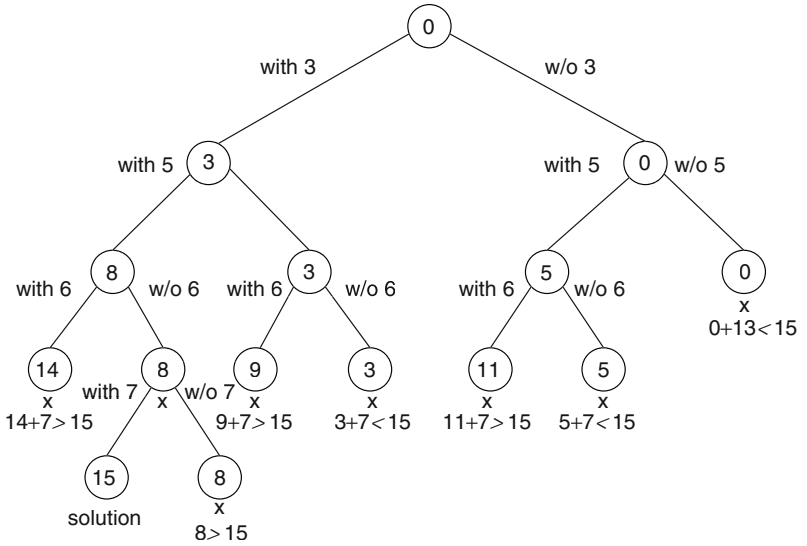


Fig: Complete state-space tree of the backtracking algorithm

15 (a) (i) WORKING PRINCIPLE:

1. It starts from the arbitrary vertex
2. It visits all vertices adjacent to starting vertex.
3. Then all unvisited vertices between two edges apart from it.
4. If there still remain unvisited vertices, the algorithm has to be restarted at an arbitrary vertex of another connected component of the graph.
    1. Queue is used to trace BFS.
    2. The queue is initialized with the traversal's starting vertex, which is marked as visited.
    3. On each iteration, the algorithm identifies all unvisited vertices that are adjacent to the front vertex, marks them as visited, and adds them to the queue; after that, the front vertex is removed from the queue.

ALGORITHM BFS(G)
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = (V, E)
//Output: Graph G with its vertices marked with consecutive integers in the order they have been visited by the BFS traversal mark each vertex in V with 0 as a mark of being "unvisited"
count 0
for each vertex v in V do

if v is marked with 0
bfs(v)
bfs(v)
//visits all the unvisited vertices connected to vertex v and assigns them the numbers in the order they are visited via global variable count
count count + 1; mark *v* with count and initialize a queue with v
while the queue is not empty do
for each vertex w in V adjacent to the front's vertex v do
if w is marked with 0
count count + 1; mark w with count add w to the queue
remove vertex v from the front of the queue

(ii) A spanning tree of a connected graph is its connected acyclic sub graph (i.e., a tree) that contains all the vertices of the graph. A minimum spanning tree of a weighted connected graph is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges. The minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.

Two serious difficulties to construct Minimum Spanning Tree

1. The number of spanning trees grows exponentially with the graph size (at least for dense graphs).
2. Generating all spanning trees for a given graph is not easy.

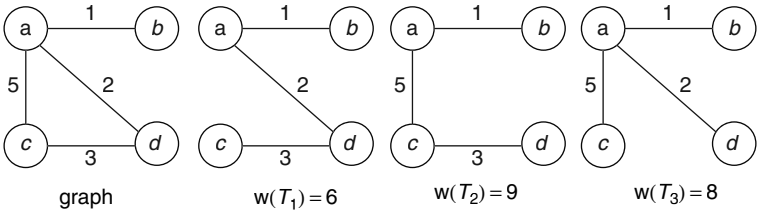


Fig.: Graph and its spanning trees; T1 is the Minimum Spanning Tree

Prim's algorithm constructs a minimum spanning tree through a sequence of expanding sub trees. The initial sub tree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices.

On each iterations, we expand the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. The algorithm stops after all the graph's vertices have been included in the tree being constructed.

Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is n-1,

where n is the number of vertices in the graph. The tree generated by the algorithm is obtained as the set of edges used for the tree expansions.

The nature of Prim's algorithm makes it necessary to provide each vertex not in the current tree with the information about the shortest edge connecting the vertex to a tree vertex.

We can provide such information by attaching two labels to a vertex: the name of the nearest tree vertex and the length (the weight) of the corresponding edge. Vertices that are not adjacent to any of the tree vertices can be given the label indicating their —infinite| distance to the tree vertices a null label for the name of the nearest tree vertex.

With such labels, finding the next vertex to be added to the current tree T = (VT, ET) become simple task of finding a vertex with the smallest distance label in the set V–VT. Ties can be broken arbitrarily. After we have identified a vertex u* to be added to the tree, we need to perform two operations:

- Move u* from the set V–VT to the set of tree vertices VT.
- For each remaining vertex U in V–VT - that is connected to u* by a shorter edge than the u's current distance label, update its labels by u* and the weight of the edge between u* and u, respectively.

(b) Solve 0/1 knapsack problem using branch and bound technique.

Branch and Bound

For example, for the instance of the above figure, formula yields lb = [(1 + 3) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 3)] /2= 14.

We now apply the branch and bound algorithm, with the bounding function given by formula, to find the shortest Hamiltonian circuit for the graph of the above figure (a). To reduce the amount of potential work, we take advantage of two observations.

First, without loss of generality, we can consider only tours that start at a.

Second, because our graph is undirected, we can generate only tours in which b is visited before c. In addition, after visiting $n-1$ = 4 cities, a tour has no choice but to visit the remaining unvisited city and return to the starting one.

KNAPSACK PROBLEM

Given n items of known weights w_i and values v_i, i = 1,2,..., n, and a knapsack of capacity W, find the most valuable subset of the items that fit in 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, with ties resolved arbitrarily:

$$v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_n/w_n.$$

It is natural to structure the state-space tree for this problem as a binary tree constructed as follows (following figure).

Each node on the ith level of this tree, $0 \leq i \leq n$, represents all the subsets of n items that include a particular selection made from the first i ordered items. This particular selection is uniquely determined by a path from the root to the node: a branch going to the left indicates the inclusion of the next item while the branch going to the right indicates its exclusion.

We record the total weight w and the total value v of this selection in the node, along with some upper bound ub on the value of any subset that can be obtained by adding zero or more items to this selection.

A simple way to compute the upper bound ub is to add to 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:

$$ub = v + (W - w)\ (v_{i+1}/w_{i+1})$$

As a specific example, let us apply the branch-and-bound algorithm to die same instance of the knapsack problem

item	***weight***	***value***	$\frac{\textbf{\textit{value}}}{\textbf{\textit{weight}}}$
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

Fig.: Knapsack Problem (The knapsacks capacity W is 10.)

At the root of the state-space tree (in the following figure), no items have been selected as yet. Hence, both the total weight of the items already selected w and their total value v are equal to 0.

The value of the upper bound computed by formula (ub=v+(W-w) (vi+1/wi+1) is $100. Node 1, the left child of the root, represents the subsets that include item, 1.

The total weight and value of the items already included are 4 and $40, respectively; the value of the upper bound is 40 + (10 – 4)*6 = $76.

Node 2 represents the subsets that do not include item 1. Accordingly, $w = 0$, $v = \$0$, and $ub = 0 + (10 - 0)*6 = \$60$.

Since node 1 has a larger upper bound than the upper bound of node 2, it is more promising for this maximization problem, and we branch from node 1 first. Its children— nodes 3 and 4—represent subsets with item 1 and with and without item 2, respectively.

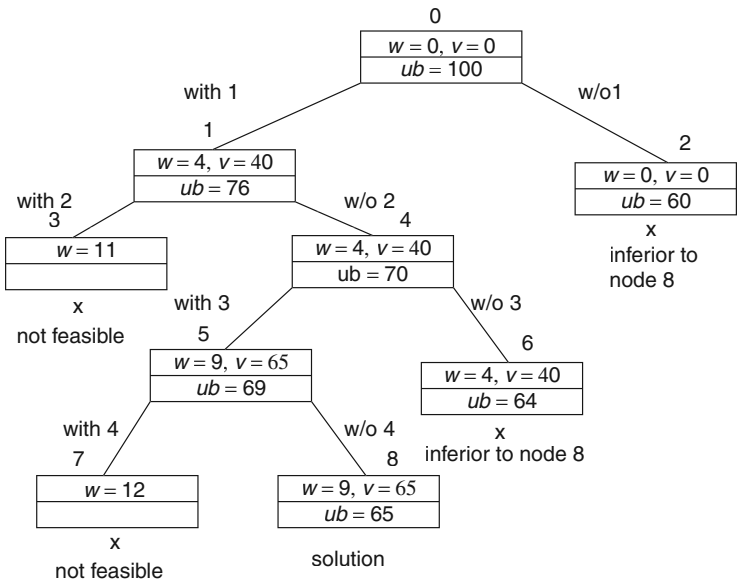


Fig.: State-space tree of the branch-and-bound algorithm for the instance of the knapsack problem

Since the total weight w of every subset represented by node 3 exceeds the knapsack's capacity, node 3 can be terminated immediately. Node 4 has the same values of w and u as its parent; the upper bound ub is equal to $40 + (10 - 4)*5 = \$70$.

Selecting node 4 over node 2 for the next branching, we get nodes 5 and 6 by respectively including and excluding item 3. The total weights and values as well as the upper bounds for these nodes are computed in the same way as for the preceding nodes.

Branching from node 5 yields node 7, represents no feasible solutions and node 8 that represents just a single subset {1, 3}.

The remaining live nodes 2 and 6 have smaller upper-bound values than the value of the solution represented by node 8. Hence, both can be terminated making the subset {1, 3} of node 8 the optimal solution to the problem.

Solving the knapsack problem by a branch-and-bound algorithm has a rather unusual characteristic.

Typically, internal nodes of a state-space tree do not define a point of the problem‘s search space, because some of the solution‘s components remain undefined. For the knapsack problem, however, every node of the tree represents a subset of the items given.

We can use this fact to update the information about the best subset seen so far after generating each new node in the tree. If we did this for the instance investigated above, we could have terminated nodes 2 & 6 before node 8 was generated because they both are I inferior to the subset of value $65 of node 5.

B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2010

Fourth Semester

Computer Science and Engineering

CS2251 − DESIGN AND ANALYSIS OF ALGORITHMS

(Common to Information Technology)

(Regulation 2008)

Time: Three hours **Maximum: 100 marks**

Answer All Questions

PART A − (10 × 2 = 20 Marks)

1. Differentiate Time Complexity from Space complexity.
2. What is a Recurrence Equation?
3. What is called Substitution Method?
4. What is an optimal solution?
5. Define Multistage Graphs.
6. Define Optimal Binary Search Tree.
7. Differentiate Explicit and Implicit Constraints.
8. What is the difference between a Live Node and a Dead Node?
9. What is a Biconnected Graph?
10. What is a FIFO branch-and-bound algorithm?

PART B − (5 × 16 = 80 Marks)

11. (a) Explain how Time Complexity is calculated. Give an example.

Or

(b) Elaborate on Asymptotic Notations with examples.

12. (a) With a suitable algorithm, explain the problem of finding the maximum and minimum items in a set of n elements.

Or

(b) Explain Merge Sort Problem using divide and conquer technique. Give an example.

13. (a) Write down and explain the algorithm to solve all pairs shortest paths problem.

Or

(b) Explain how dynamic programming is applied to solve travelling salesperson problem.

14. (a) Describe the backtracking solution to solve 8-Queens problem.

Or

(b) With an example, explain Graph Coloring Algorithm problem statement.

15. (a) Explain in detail the graph traversals.

Or

(b) With an example, explain how the branch-and-bound technique is used to solve 0/1 knapsack problem.Solutions

Solutions

PART A

1. • The total number of steps involved in a solution to solve a problem is the function of the size of the problem, which is the measure of that problem's time complexity
   • Space complexity is measured by using polynomial amounts of memory, with an infinite amount of time.
     ▪ The difference between space complexity and time complexity is that space can be reused. Space complexity is not affected by determinism or non-determinism.

2. • Recurrence equation defines each element of a sequence in terms of one or more earlier elements.
   • E.g. The Fibonacci sequence,

$$X[1] = 1\ X[2] = 1$$

$$X[n] = X[n-1] + X[n-2]$$

   • Some recurrence relations can be converted to "closed form" where $X[n]$ is defined purely in terms of n, without reference to earlier elements.

3. • Substitution Method works by solving one of the chosen equations for one of the variables and then plugging this back into the other equation, "substituting" for the chosen variable and solving for the other. Then solve for the first variable

4. • Optimal solution is a feasible solution with the best value of the objective function
   • Eg: The shortest Hamiltonian Circuit

5. • A multistage graph is a graph
     ▪ $G=(V,E)$ with V partitioned into $K >= 2$ disjoint subsets such that if (a,b) is in E, then a is in V_i , and b is in V_{i+1} for some subsets in the partition
     ▪ And $| V_1 | = | V_K | = 1$.
   • The vertex s in V_1 is called the source; the vertex t in V_K is called the sink.
   • G is usually assumed to be a weighted graph.
   • The cost of a path from node v to node w is sum of the costs of edges in the path.
   • The "multistage graph problem" is to find the minimum cost path from s to t.

6. • One of the principal applications of Binary Search Tree is to implement the operation of searching.
   • If probabilities of searching for elements of a set are known, it is natural to pose a question about an optimal binary search tree for which the average number of comparisons in a search is the smallest possible.

7. • Explicit constraints are rules that restrict each x_i to take on values only from a given set.
   • Explicit constraints depend on the particular instance I of problem being solved
   • Examples of explicit constraints
     ▪ $x_i = 0$, or all nonnegative real numbers
     ▪ $x_i = \{0, 1\}$
   • Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function.
   • Implicit constraints describe the way in which the $x_i s$ must relate to each other.

8. • Live node is a node that has been generated but whose children have not yet been generated.
   • Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.
   • 'Live' nodes are able to acquire new links whereas 'dead' nodes are static.

9. **Biconnected graph** is a connected and "non-separable" graph, meaning that if any vertex were to be removed, the graph will remain connected. Therefore a Biconnected graph has no articulation vertices.

10. • Branch-and-bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.
    • BFS is an FIFO search in terms of live nodes.
      ▪ List of live nodes is a queue
    • FIFO branch-and-bound algorithm
      Initially, there is only one live node; no queen has been placed on the chessboard
      ▪ The only live node becomes E-node
      ▪ Expand and generate all its children
      ▪ Next E-node is the node with queen in row 1 and column 1
      ▪ Expand this node, and add the possible nodes to the queue of live nodes
      ▪ Bound the nodes that become dead nodes

PART B

11. (a) **Time complexity** of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the string representing the input. The time complexity of an algorithm is commonly expressed using big O notation, which suppresses multiplicative constants and lower order terms. When expressed this way, the time complexity is said to be described *asymptotically*, i.e., as the input size goes to infinity.

Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform. Thus the amount of time taken and the number of elementary operations performed by the algorithm differ by at most a constant factor.

Since an algorithm's performance time may vary with different inputs of the same size, one commonly uses the worst-case time complexity of an algorithm, denoted as $T(n)$, which is defined as the maximum amount of time taken on any input of size n. Time complexities are classified by the nature of the function $T(n)$. For instance, an algorithm with $T(n) = O(n)$ is called a linear time algorithm, and an algorithm with $T(n) = O(2^n)$ is said to be an exponential time algorithm.

For example, if the time required by an algorithm on all inputs of size n is at most $5n^3 + 3n$, the asymptotic time complexity is O (n^3).

(b) Asymptotic notations are methods to estimate and represent efficiency of an algorithm

- Big-oh notation (O)
- Big-omega notation (Ω)
- Big-Theta notation (Θ)
- Little-oh notation (o)

Big-oh notation is used to define the worst-case running time of an algorithm and concerned with large values of n.

$O(g(n))$ is the set of all functions with a smaller or same order of growth as $g(n)$.

Example

$n \in O(n2)$

$100n + 5 \in O(n2)$

$n(n-1) / 2 \in O(n2)$

Big-Omega Notation

Definition: A function $t(n)$ is said to be in $\Omega(g(n))$, denoted as $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n.

$$t(n) \leq cg(n) \text{ for all } n \geq n0$$

Big-theta notation

A function $t(n)$ is said to be in $\Theta(g(n))$, $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n

$$C2g(n) \leq t(n) \leq c1g(n) \text{ for all } n \geq n0$$
$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

Little-oh notation

Definition: a function $t(n)$ is said to be in $o(g(n))$, $t(n) \in o(g(n))$, if there exists a constant c and some nonnegative integer such that

$$t(n) \leq cg(n)$$
$$\lim_{n \to \infty} t(n) / g(n) = 0$$

12. (a) **Selection by sorting**

Selection can be reduced to sorting by sorting the list and then extracting the desired element. This method is efficient when many selections need to be made from a list, in which case only one initial, expensive sort is needed, followed by many cheap extraction operations. In general, this method requires $O(n \log n)$ time, where n is the length of the list.

Linear minimum/maximum algorithms

Linear time algorithms to find minima or maxima work by iterating over the list and keeping track of the minimum or maximum element so far.

Nonlinear general selection algorithm

Using the same ideas used in minimum/maximum algorithms, we can construct a simple, but inefficient general algorithm for finding the kth smallest or kth largest item in a list, requiring $O(kn)$ time, which is effective when k is small. To accomplish this, we simply find the most extreme value and move it to the beginning until we reach our desired index. This can be seen as an incomplete selection sort.

Minimum-based algorithm:

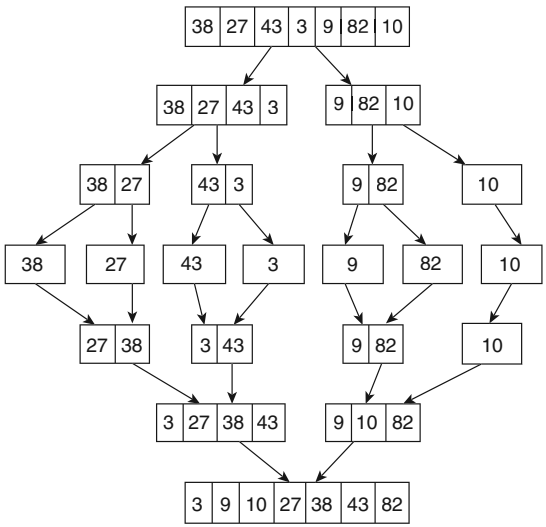
```
function select(list[1..n], k)
   for i from 1 to k
      minIndex = i
      minValue = list[i]
      for j from i+1 to n
         if list[j] < minValue
            minIndex = j
            minValue = list[j]
   swap list[i] and list[minIndex]
return list[k]
```

(b) **Divide and conquer** is probably the best known general algorithm design technique. It work according to the following general plan

- A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
- The smaller instances are solved
- If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original problem

Merge sort is a perfect example of a successful application of the divide and conquer technique. It sorts a given array $A[0...n-1]$ by dividing it into two halves $A[0...[n/2]-1]$ and $A[[n/2]....n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

Example:



13. (a) **Floyd–Warshall** is a graph analysis algorithm for finding shortest paths in a weighted graph (with positive or negative edge weights) and also for finding transitive closure of a relation R. A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between *all* pairs of vertices, though it does not return details of the paths themselves.

```
procedure FloydWarshallWithPathReconstruction ()
   for k := 1 to n
      for i := 1 to n
         for j := 1 to n
            if path[i][k] + path[k][j] < path[i][j] then {
               path[i][j] := path[i][k]+path[k][j];
               next[i][j] := k; }

function GetPath (i, j)
   if path[i][j] equals infinity then
   return "no path";
int intermediate := next[i][j];
if intermediate equals 'null' then
   return " "; /* there is an edge from i to j, with no vertices
   between */
else
   return GetPath (i, intermediate) + intermediate + GetPath (inter-
   mediate, j);
```

(b) The **travelling salesman problem (TSP)** is an NP-hard problem in combinatorial optimization studied in operations research and theoretical computer science. Given a list of cities and their pair wise distances, the task is to find the shortest possible route that visits each city exactly once and returns to the origin city.

Let,

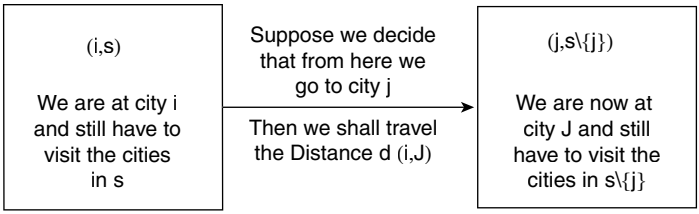
$f(i, s)$:= shortest sub-tour given that we are at city i and still have to visit the cities in s (and return to home city)

Then clearly,

$$f(i,\phi) = d(i,0), \quad \phi = \text{empty set}$$

$$f(i,s) = \min_{j \in S} \{d(i,j) + f(j, s \setminus \{j\})\}, \; s \neq \phi$$

$$s \setminus A := \{k \in s, k \notin A\}.$$



$$f(i,\phi) = d(i,0), \quad \phi = \text{empty set}$$

$$f(i,s) = \min_{j \in S} \{d(i,j) + f(j, s \setminus \{j\})\}, \; s \neq \phi$$

$$s \setminus A := \{k \in s, k \notin A\}.$$

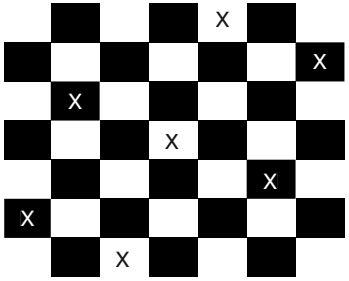
14. (a) Backtracking is kind of solving a problem by trial and error. However, it is a well organized trial and error. We make sure that we never try the same thing twice. We also make sure that if the problem is finite we will eventually try all possibilities (assuming there is enough computing power to try all possibilities).

The **eight queens' puzzle** is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens attack each other. Thus, a solution requires that no two queens share the same row, column, or diagonal. The eight queens' puzzle is an example of the more general **n-queens problem** of placing n queens on an $n \times n$ chessboard, where solutions exist for all natural numbers n with the exception of 2 and 3.

The eight queens' puzzle has 92 **distinct** solutions. If solutions that differ only by symmetry operations (rotations and reflections) of the board are counted as one, the puzzle has 12 unique (or fundamental) solutions.

A fundamental solution usually has eight variants (including its original form) obtained by rotating 90, 180, or 270 degrees and then reflecting each of the four rotational variants in a mirror in a fixed position. However, should a solution be equivalent to its own 90 degree rotation (as happens to one solution with five queens on a 5×5 board) that fundamental solution will have only two variants (itself and its reflection). Should a solution be equivalent to its own 180 degree rotation (but not to its 90 degree rotation) it will have four variants (itself, its reflection, its 180 degree rotation and the reflection of that). It is not possible for a solution to be equivalent to its own reflection (except at n=1) because that would require two queens to be facing each other. Of the 12 fundamental solutions to the problem with eight queens on an 8×8 board, exactly one is equal

to its own 180 degree rotation, and none are equal to their 90 degree rotation, thus the number of distinct solutions is 11*8 + 1*4 = 92 (where the 8 is derived from four 90-degree rotational positions and their reflections, and the 4 is derived from two 180-degree rotational positions and their reflections).



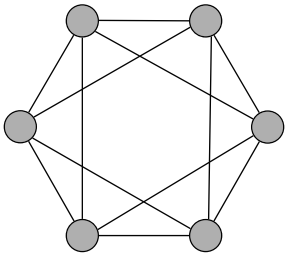
(b) In graph theory, graph coloring is a special case of graph labeling; it is an assignment of labels traditionally called "colors" to elements of a graph subject to certain constraints. In its simplest form, it is a way of coloring the vertices of a graph such that no two adjacent vertices share the same color; this is called a **vertex coloring**. Similarly, an **edge coloring** assigns a color to each edge so that no two adjacent edges share the same color, and a **face coloring** of a planar graph assigns a color to each face or region so that no two faces that share a boundary have the same color.

Vertex coloring is the starting point of the subject, and other coloring problems can be transformed into a vertex version. For example, an edge coloring of a graph is just a vertex coloring of its line graph, and a face coloring of a planar graph is just a vertex coloring of its planar dual. However, non-vertex coloring problems are often stated and studied as is. That is partly for perspective, and partly because some problems are best studied in non-vertex form, as for instance is edge coloring.

Example:

The convention of using colors originates from coloring the countries of a map, where each face is literally colored. This was generalized to coloring the faces of a graph embedded in the plane. By planar duality it became coloring the vertices, and in this form it generalizes to all graphs. In mathematical and computer representations it is typical to use the first few positive or nonnegative integers as the "colors". In general one can use any finite set as the "color set". The nature of the coloring problem depends on the number of colors but not on what they are.

3-color example



15. (a) **Graph Traversal**

To traverse a graph is to process every node in the graph exactly once. Because there are many paths leading from one node to another, the hardest part about traversing a graph is making sure that you do not process some node twice.

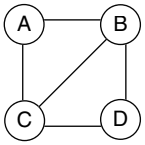
There are two general solutions to this difficulty:

- When you first encounter a node, mark it as REACHED. When you visit a node, check if it's marked REACHED; if it is, just ignore it. This is the method our algorithms will use.
- When you process a node, delete it from the graph. Deleting the node causes the deletion of all the arcs that lead to the node, so it will be impossible to reach it more than once.

General Traversal Strategy

- Mark all nodes in the graph as NOT REACHED.
- Pick a starting node. Mark it as REACHED and place it on the READY list.
- Pick a node on the READY list. Process it. Remove it from READY. Find all its neighbours: those that are NOT REACHED should marked as REACHED and added to READY.
- Repeat 3 until READY is empty.

Example:

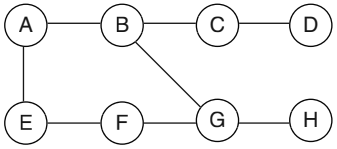


- Step 1: A = B = C = D = NOT REACHED.
- Step 2: READY = {A}. A = REACHED.
- Step 3: Process A. READY = {B, C}. B = C = REACHED.
- Step 3: Process C. READY = {B, D}. D = REACHED.

- Step 3: Process B. READY = {D}.
- Step 3: Process D. READY = {}.

The two most common traversal patterns are *breadth-first* **traversal and** *depth-first* **traversal.**

- In breadth-first traversal, READY is a QUEUE, not an arbitrary list. Nodes are processed in the order they are reached (FIFO). This has the effect of processing nodes according to their distance from the initial node. First, the initial node is processed. Then all its neighbours are processed. Then all of the neighbors' neighbours etc.
- In depth-first traversal, READY is a STACK; the most recently reached nodes are processed before earlier nodes.



Initial Steps: READY = [A]. Process A. READY = [B,E]. Process B. It is at this point that two traversal strategies differ. Breadth-first adds B's neighbours to the *back* of READY; depth-first adds them to the *front*:

Breadth First:

- READY = [E,C,G].
- process E. READY = [C,G,F].
- process C. READY = [G,F,D].
- process G. READY = [F,D,H].
- process F. READY = [D,H].
- process D. READY = [H].
- process H. READY = [].

Depth First:

- READY = [C,G,E].
- process C. READY = [D,G,E].
- process D. READY = [G,E].
- process G. READY = [H,F,E].
- process H. READY = [F,E].
- process F. READY = [E].
- process E. READY = [].

(b) Mathematically the 0-1-knapsack problem can be formulated as:
Let there be n items, x_1 to x_n where x_i has a value u_i and weight w_i. The maximum weight that we can carry in the bag is W. It is common to assume that all values and weights are nonnegative. To simplify the representation, we also assume that the items are listed in increasing order of weight.

- Maximize $\sum_{i=1}^{n} u_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \le W, \qquad x_i \in \{0,1\}$

Maximize the sum of the values of the items in the knapsack so that the sum of the weights must be less than the knapsack's capacity.

The **bounded knapsack problem** removes the restriction that there is only one of each item, but restricts the number x_i of copies of each kind of item to an integer value c_i.
Mathematically the bounded knapsack problem can be formulated as:

- maximize $\sum_{i=1}^{n} u_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \le W, \qquad x_i \in \{0,1,\ldots,c_i\}$

The **unbounded knapsack problem** (**UKP**) places no upper bound on the number of copies of each kind of item and can be formulated as above except for that the only restriction on x_i is that it is a non-negative integer. If the example with the colored bricks above is viewed as an unbounded knapsack problem, then the solution is to take three yellow boxes and three grey boxes.

Example

- Capacity W is 10
- Upper bound is \$100 (use fractional value)

Item	Weight	Value	Value / weight
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

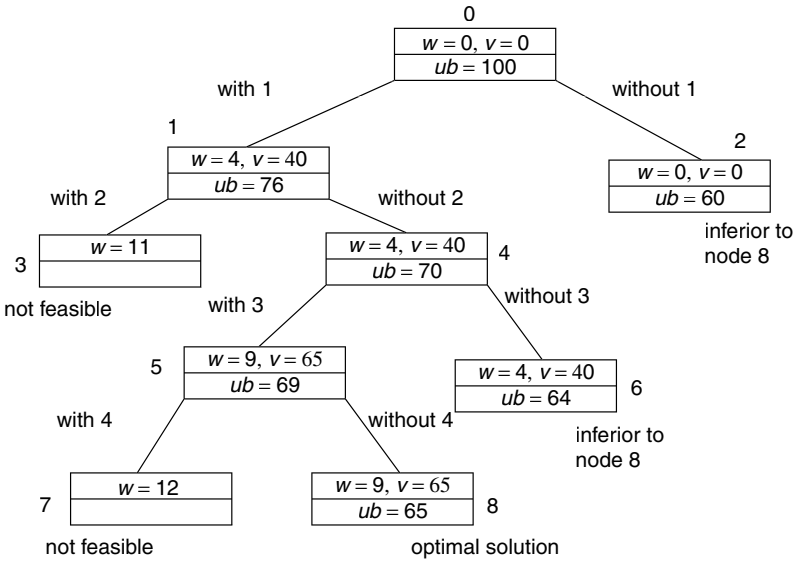
- To compute the upper bound, use
  - $ub = v + (W - w)(v_{i+1}/w_{i+1})$
- So the maximum upper bound is
  - pick no items, take maximum profit item
  - ub = (10 – 0)*(\$10) = \$100

After we pick item 1, we calculate the upper bound as

- all of item 1 (4, $40) + partial of item 2 (7, $42)
- $40 + (10-4)*6 = $76

If we don't pick item 1:

- ub = (10 – 0) * ($6) = $60



B.E./B.Tech. DEGREE EXAMINATION
NOV/DEC 2009

Fourth Semester

Computer Science and Engineering

CS2251 − DESIGN AND ANALYSIS OF ALGORITHMS

(Common to Information Technology)

(Regulation 2008)

Time: Three hours **Maximum: 100 marks**

Answer All Questions

PART A − (10 × 2 = 20 Marks)

1. If $f(n) = a_m n^m + \ldots\ldots a_1 n + a_0$, then prove that $f(n) = O(n^m)$.
2. Establish the relationship between O and Ω.
3. Trace the operation of the binary search algorithm for the input −15, −6, 0, 7, 9, 23, 54, 82, 101, 112, 131, 142, 151, if you are searching for the element 9.
4. State the general of principle of greedy algorithm.
5. State the principle of optimality.
6. Compare divide and conquer with dynamic programming and dynamic programming with greedy algorithm.
7. Explain the idea behind backtracking.
8. Define and solve the graph coloring problem.
9. Define NP Hard and NP Completeness.
10. What is a Minimum spanning tree?

PART B (5 × 16 = 80 MARKS)

11 (a) (i) Explain briefly the Time Complexity estimation, space complexity estimation and the trade off between Time and Space complexity. (6)

(ii) Solve the following recurrence equations completely.

(1) $T(n) = \sum_{i=1}^{n-1} T(i) + 1$, if $n \geq 2$ (4)

$T(n) = 1$; if $n = 1$

(2) $T(n) = 5T(n-1) - 6T(n-2)$ (3)

(3) $T(n) = 2T\left(\frac{n}{2}\right) + nl\,g\,n$ (3)

Or

(b) (i) Write the linear search algorithm and analyse for its best, worst and average case time complexity. (10)

(ii) Prove that for any two functions $f(n)$ and $g(n)$, we have $f(n) = \theta(g(n))$ if and only if $f(n) = O(g(n) = \Omega(g(n))$. (6)

12. (a) (i) Write a recursive algorithm to determine the max and min from a given element and explain. Derive the time complexity of this algorithm and compare it with a simple brute force algorithm for finding max and min. (10)

(ii) For the following list of element trace the recursive algorithm for finding max and min and determine how many comparisons have made. 22, 13, –5, –8, 15, 60, 17, 31, 47 (6)

Or

(b) (i) Write the container loading greedy algorithm and explain. Prove that this algorithm is optimal. (8)

(ii) Suppose you have 6 containers whose weights are 50, 10, 30, 20, 60, 5 and a ship whose capacity is 100. Use the above algorithm to find an optimal solution to this instance of the container loading problem. (8)

13. (a) (i) Write and explain the algorithm to computer the all pairs source shortest path using dynamic programming and prove that it is optimal. (8)

(ii) For the following graph having four nodes represented by the matrix given below determine the all pairs source shortest path (8)

$$\begin{matrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{matrix}$$

Or

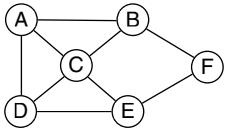
(b) (i) Write the algorithm to compute the 0/1 knapsack problem using dynamic programming and explain it (8)

(ii) Solve the following instance of the 0/1, knapsack problem given the knapsack capacity is W = 5 (8)

Items	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

14. (a) Write an algorithm to determine the Sum of Subsets for a given Sum and a Set of numbers. Draw the tree representation to solve the subset sum problem given the numbers set as {3, 5, 6, 7, 2} with the Sum = 15. Derive all the subsets. (8 + 8)

(b) Write an algorithm to determine Hamiltonian cycle in a given graph using back tracking. For the following graph determine the Hamiltonian cycle (8 + 8)



15. (a) Explain with an algorithm as to how 0/1 knapsack problem is solved using branch and bound technique. Apply branch and bound technique to solve the following 0/1 knapsack instance if W = 10. (8 + 8)

Items	Weight	Value
1	4	40
2	7	42
3	5	25
4	3	12

Or

(b) Write an algorithm and explain to determine Biconnected Components. Prove the theorem that two biconnected components can have at most one vertex as common and this vertex is an articulation point. (10 + 6)

Solutions

PART A

1. Consider $f(n)=a_m n^m+\ldots a_1 n+a_0$
   $a_m n^m+\ldots a_1 n+a_0 \leq a_m n^m+\ldots a_1 n+n \leq a_m n^m+\ldots (a_1+1)n$
   $f(n) = O(n^m)$

2. Relationship between O and Ω
   - A function $t(n)$ is said to be in $O(g(n))$ denoted $t(n)$ ε O $(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some non negative integer $n0$ such that
     - $T(n) < c\ g$ (n) for $n > n0$
   - A function $t(n)$ is said to be in Ω $(g(n))$, denoted $t(n)$ $\hat{I}$ ε $(g(n))$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some non negative integer $n0$ such that
     - $T(n) < c\ g\ (n)$ for $n > n0$

3. –15,–6,0,7,9,23,54,82,101,112,131,142,151. Search for 9 using binary search.
   - Pos=(1+14)/2=8 Value=82
   - Pos=(1+8)/2=5 Value =9

4. A **greedy algorithm** is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. In many problems, a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time.

5. **Principle of optimality.**
   - Optimal solution to any instance of an optimization problem is composed of optimal solutions to its sub-instances.

6. • Divide-&-conquer works best when all sub problems are independent. So, pick partition that makes algorithm most efficient & simply com-

bine solutions to solve entire problem.

- Dynamic programming is needed when sub problems are dependent; we don't know where to partition the problem
- Divide-&-conquer is best suited for the case when no "overlapping sub problems" are encountered.
- In dynamic programming algorithms, we typically solve each sub problem only once and store their solutions. But this is at the cost of space

Dynamic Programming solves the sub-problems bottom up. The problem can't be solved until we find all solutions of sub-problems. The solution comes up when the whole problem appears.

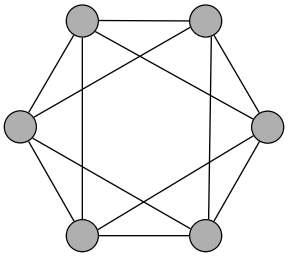
- Greedy solves the sub-problems from top down. We first need to find the greedy choice for a problem, then reduce the problem to a smaller one. The solution is obtained when the whole problem disappears.
- Dynamic Programming has to try every possibility before solving the problem. It is much more expensive than greedy. However, there are some problems that greedy cannot solve while dynamic programming can. Therefore, we first try greedy algorithm. If it fails then try dynamic programming.

7. **Idea behind backtracking.**

- The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows.
- If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component.
- If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered.
- In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option

8. **Graph coloring problem.**

- The graph coloring problem asks us to assign the smallest number of colors to vertices of a graph so that no two adjacent vertices are the same color.



9. NP Hard and NP completeness.

NP-Hard

A problem *H* is NP-hard if and only if there is an NP-complete problem L that is polynomial time Turning-reducible to H

NP-complete

A decision problem *C* is NP-complete if:

1. *C* is in NP, and
2. Every problem in NP is reducible to *C* in polynomial time.
   C can be shown to be in NP by demonstrating that a candidate solution to C can be verified in polynomial time.

10. A **minimum spanning tree** (**MST**) or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree.

Part B

11. (a) (i) The total number of steps involved in a solution to solve a problem is the function of the size of the problem, which is the measure of that problem's time complexity

Space complexity is measured by using polynomial amounts of memory, with an infinite amount of time.

The difference between space complexity and time complexity is that space can be reused. Space complexity is not affected by determinism or non-determinism.

(ii) Recurrence relation solution

1. $T(1) = 1$, $T(2) = 2$, $T(3) = 4$, $T(4) = 8$, $T(5) = 16$……
   Powers of 2 ranging from 0 and hence it is
   2^{n-1}
2. $T_n = 5T_{n-1} - 6T_{n-2}$
   $T_n - 5T_{n-1} + 6T_{n-2} = 0$
   Solving we get 2 and 3
   Applying in equations $F(n) = c_1(-2^n) + c_2(-3^n)$
   Solving $T_0 = c_1 + c_2 = 0$
   $T_1 = -2c_1 - 3c_2 = 1$
   $\Rightarrow$ We get
   $\Rightarrow -2^n + 3^n$

3. Applying this case

$$\text{If } f(n) = \Theta(n^{\log_b a} \log^k n) \text{ with } k \geq 0,$$
$$\text{then } T(n) = \Theta(n^{\log_b a} \log^{k+1} n).$$

$\Rightarrow$ We get ,
$\Rightarrow n\log^2 n$

Or

b. (i) **Linear Search**:

The problem: Search an array of size *n* to determine whether the array contains the value key; return index if found, –1 if not found

```
Set k to 0
While (k< n) and (a[k]is not key)
Add 1 to k.
If k == n
        Return –1.
Return k.
```

Analysis

In worst case, loop will be executed n times, so amount of work done is proportional to *n*, and algorithm is $O(n)$

***Average* case for a *successful* search**

- Probability of key being found at index *k* is 1 in n for each value of *k*
- Add up the amount of work done in each case, and divide by total number of cases:
- $((a*1 + d) + (a*2 + d) + (a*3 + d) + \ldots + (a*n + d))/n = (n*d + a*(1 + 2 + 3 + \ldots + n))/n = n*d/n + a*(n*(n + 1)/2)/n = d + a*n/2 + a/2 = (a/2)*n + e$,
- Where constant $e = d + a/2$, so expected case is also $O(n)$

Simpler approach to expected case:

- Add up the number of times the loop is executed in each of the n cases, and divide by the number of cases n
- $(1 + 2 + 3 + \ldots + (n-1) + n)/n = (n*(n + 1)/2)/n = n/2 + 1/2$; algorithm is therefore $O(n)$

(ii) Theorem: If $f(n) = O(g(n))$ then

$$f(n) - \Omega(g(n)). \text{ and } f(n) = O(g(n))$$

Conversely, if $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$

then $f(n) = \theta(g(n))$

The above relations can be established by using basic definitions
Example (1): Since, $n\ (n-1)/2 = \theta\ (n^2)$, therefore it follows that

$$n(n-1)/2 = \Omega\ (n^2)$$
$$n(n-1)/2 = O\ (n^2)$$

Example (2): It can be shown that

$$5n^2 + 1 = \Omega\ (n^2)$$
$$\text{and } 5n^2 + 1 = O\ (n^2)$$
$$\text{therefore, } 5n^2 + 1 = \theta\ (n^2)$$

Informally, $f(n) = *(g(n)$ means that $f(n)$ is asymptotically equal to $g(n)$.

12. (a) (i) **Finding maximum and minimum**

```
procedure MINMAX (A, L, U, mx, mn);
(* A is an array ranging between L and U; mx and mn will have
the maximum and minimum value respectively when the proce-
dure terminates *)
var l1 , l2 , U1,U2: integer ; (* Local Variables *)
mx1, mx2, mn1, mn2: integer ; (* Local Variables *)
if (U– L+1) = 2 then
1. if A[L] > A[U] then
     mx:= A[L]; (* maximum *)
     mn:= A[U]; (* minimum *)
else
     mx:= A[U]; (* maximum *)
     mn:= A[L]; (* minimum *)
 endif
else
     Split A into two halves A1 and A2 with lower and upper
     indices to be l1 , U1 and l2 , U2 respectively;
2. call MINMAX (A, l1 , U1 , mx1 , mn1 );
3. call MINMAX (A, l2, U2, mx2, mn2);
4. mx : = maximum (mx1, mx2 ) ; (* maximum returns the maxi-
     mum of the two values *)
5. mn : = minimum (mn1, mn2 ); (* minimum returns the mini-
     mum of the two values *)
endprocedure{MINMAX}
```

In brute force:

To find the maximum and minimum of n numbers, $(n-1) + (n-2) = 2n - 3$ comparisons.

In recursive:

Let us analyze the number of comparisons made by the above algorithm. Explicit comparisons are made in line (1) where maximum and minimum is computed between two array elements and in line (5) where maximum and minimum of two elements are computed. The recursive algorithms can be effectively analyzed through recurrence equations

Let $T(n)$ be the total number of comparisons made by the procedure MINMAX when $U - L + 1 = n$. Obviously, $T(2) = 1$. One comparison each is needed for the operations of line 4 and line 5. If $n > 2$, the size of the arrays in lines 2 and 3 will be $n/2$ each. Thus,

$T(n) = 1$ if $n = 2$

$2*T(n)+2$ if $n > 2$

It can be shown that the function $T(n) = (3/2)n - 2$ is the solution to the above recurrence equation. Further, we can show that $(3/2)n - 2$ comparisons are necessary for finding the maximum and minimum from n elements. In a sense, the above algorithm is optimal when n is a power of 2.

(ii) Solution to the problem

- Apply recursion and find the min and max.
- Min= −8
- Max=60
- Number of Comparisons for Min and Max

  $T(n)= (3/2)n-2$

  $T(9)=(3/2)*9-2$ = approx 12 comparisons for recursive

  Brute force =8+7=15 comparisons

Or

(b). (i) Loading problem

- Ship has capacity c.
- m containers are available for loading.
- Weight of container i is w_i.
- Each weight is a positive number.
- Sum of container weights > c.
- Load as many containers as is possible without sinking the ship.
- Load containers in increasing order of weight until we get to a container that doesn't fit.

(ii) Solution to the problem

Given: 50, 10, 30, 20, 60, 5

Capacity = 100

- Sort then in order
  5, 10, 20, 30, 50, 60
- Loading order 10, 30,60 ⇒ Capacity = 100

13. (a) (i) Dynamic-programming algorithm for all pairs shortest path
- Input: A weighted graph, represented by its weight matrix W.
- Problem: find the distance between every pair of nodes.
- Dynamic programming Design:
- Notation: $A^{(k)}(i,j)$ = length of the shortest path from node i to node j where the label of every intermediary node is ⇐ k.
- $A^{(0)}(i, j) = W[i,j]$.
- Principle of Optimality: We already saw that any sub-path of a shortest path is a shortest path between its end nodes.
- recurrence relation:
- Divide the paths from i to j where every intermediary node is of label <=k into two groups:
- 1. Those paths that do go through node k
- 2. Those paths that do not go through node k.
- the shortest path in the first group is the shortest path from i to j where the label of every intermediary node is ⇐ k–1.
- therefore, the length of the shortest path of group 1 is $A^{(k-1)}(i,j)$
- Each path in group two consists of two portions: The first is from node i to node k, and the second is from node k to node j.
- the shortest path in group 2 does not goe through K more than once, for otherwise, the cycle around K can be eliminated, leading t o a shorter path in group 2.
- Therefore, the two portions of the shortest path in group 2 have their intermediary labels ⇐ k–1.
- Each portion must be the shortest of its kind. That is, the portion from i to k where intermediary node is ⇐ k–*1* must be the shortest such a path from i to k. If not, we would get a shorter path in group 2. Same thing with the second portion (from j to k).
- Therefore, the length of the first portion of the shortest path in group 2 is $A^{(k-1)}(i,k)$
- Therefore, the length of the 2nd portion of the shortest path in group 2 is $A^{(k-1)}(k,j)$
- Hence, the length of the shortest path in group 2 is $A^{(k-1)}(i,k) + A^{(k-1)}(k,j)$
- Since the shortest path in the two groups is the shorter of the shortest paths of the two groups, we get
- $A^{(k)}(i,j)$=min($A^{(k-1)}(i,j)$, $A^{(k-1)}(i,k) + A^{(k-1)}(k,j)$).

Algorithm

```
Procedure APSP(input: W[1:n,1:n];A[1:n,1:n])
begin
   for i=1 to n do
      for j=1 to n do
         A(0)(i, j) := W[i,j];
   endfor
   endfor
   for k=1 to n do
      for i=1 to n do
         for j=1 to n do
            A(k)(i, j)=min(A(k–1)(i, j), A(k–1)(i, k) + A(k–1)(k, j))
         endfor
      endfor
   endfor
end
```

- Note that once $A^{(k)}$ has been computed, there is no need for $A^{(k-1)}$
- Therefore, we don't need to keep the superscript
- By dropping it, the algorithm remains correct, and we save on space and hence the solution is optimal

```
Procedure APSP(input: W[1:n,1:n];A[1:n,1:n])
begin
   for i=1 to n do
      for j=1 to n do
         A(i, j) := W[i,j];
      endfor
   endfor
   for k = 1 to n do
      for i = 1 to n do
         for j = 1 to n do
            A(i, j) = min(A(i, j),A(i, k) + A(k,j));
         endfor
      endfor
   endfor
end
```

- **Time Complexity Analysis:**
  The first double for-loop takes $O(n^2)$ time.
  The tripl-for-loop that follows has a constant-time body, and thus takes $O(n^3)$ time.

Thus, the whole algorithm takes $O(n^3)$ time.

(ii) Solution to the problem

	A	B	C	D
A	0	∞	3	∞
B	2	0	∞	∞
C	∞	7	0	1
D	6	∞	∞	0

	A	B	C	D
A	0	10	3	4
B	2	0	5	6
C	9	7	0	1
D	6	16	9	0

Final solution

	A	B	C	D
A	0	10	3	4
B	2	0	5	6
C	7	7	0	1
D	6	16	9	0

Or

(b). (i) **Dynamic programming**

If all weights (w_1, …, w_n, w) are nonnegative integers, the knapsack problem can be solved in pseudo-polynomial time using dynamic programming. The following describes a dynamic programming solution for the *unbounded* knapsack problem.

To simplify things, assume all weights are strictly positive ($w_i > 0$). We wish to maximize total value subject to the constraint that total weight is less than or equal to W. Then for each $w \le W$, define $m[w]$ to be the maximum value that can be attained with total weight less than or equal to w. $m[W]$ then is the solution to the problem.

$m\ [0] = 0$ (the sum of zero items, i.e., the summation of the empty set)

- $m[0] = 0$ (the sum of zero items, i.e., the summation of the empty set)
- $m[w] = \max_{w_i \le w} \left(u_i + m[w - w_i] \right)$

where u_i is the value of the i-th kind of item.

The following is pseudo code for the dynamic program:

Input:

```
Values (stored in array v)
Weights (stored in array w)
Number of distinct items (n)
Knapsack capacity (W)
for w from 0 to W do
   T[0, w] := 0
end for
for i from 1 to n do
   for j from 0 to W do
    if j >= w[i] then
      T[i, j] := max(T[i–1, j], T[i–1, j–w[i]] + v[i])
else
      T[i, j] := T[i–1, j]
      end if
   end for
end for
```

0-1 knapsack problem

A similar dynamic programming solution for the 0–1 *knapsack problem* also runs in pseudo-polynomial time. As above, assume (w_1, w_2, …, $w_{n,}$ w) are strictly positive integers. Define $m[i, w]$ to be the maximum value that can be attained with weight less than or equal to w using items up to i.

We can define $m[i, w]$ recursively as follows:

- $m[i,w] = m[i–1, w]$ if $w_i > w$ (the new item is more than the current weight limit)
- $m[i, w] = \max (m[i–1, w], m[i–1, w–w_i] + u_i)$ if $w_i \leq w$.

The solution can then be found by calculating $m[n, W]$. To do this efficiently we can use a table to store previous computations. This solution will therefore run in $O(nW)$ time and $O(nW)$ space. Additionally, if we use only a 1-dimensional array $m[W]$ to store the current optimal values and pass over this array $i + 1$ times, rewriting from $m[W]$ to $m[1]$ every time, we get the same result for only $O(W)$ space.

(ii)

Item	Weight	Value	Value / weight
1	2	12	6
2	1	10	10
3	3	20	6.33
4	2	15	7.5

- $w = 5$
- First choose 2 – Next 1- next 3
- This exceeds $w = 5$
- So choose the next weight 2
- Optimal solution s = {2,1,2}
- $V = \{12,10,15\} = 37$

14. (a) **Sum of Subsets**

Given n distinct positive numbers usually called as weights, the problem calls for finding all the combinations of these numbers whose sums are m.

initialize a list S to contain one element 0.
for each i from 1 to N do
let T be a list consisting of $x_i + y$, for all y in S
let U be the union of T and S
sort U
make S empty
let y be the smallest element of U
add y to S
for each element z of U in increasing order do
//trim the list by eliminating numbers close to one another
//and throw out elements greater than s
if $y + cs/N < z \leq s$, set $y = z$ and add z to S
if S contains a number between $(1 - c)s$ and s, output *yes*, otherwise *no*

Solution to the problem

- Given set {3, 5, 6, 7, 2} Sum=15
- Construct the tree such that sum=15
- First Start with 3 and then with other numbers such that we obtain the subset whose summation is 15

Possible solutions are:

- [3,5,7]
- [6,7,2]

(b) A *Hamiltonian path* or *traceable path* is a path that visits each vertex exactly once. A graph that contains a Hamiltonian path is called a **traceable graph**. A graph is **Hamilton-connected** if for every pair of vertices there is a Hamiltonian path between the two vertices. A *Hamiltonian cycle*, *Hamiltonian circuit*, *vertex tour* or *graph cycle* is a cycle that visits each vertex exactly once (except the vertex which is both the start and end, and so is visited twice). A graph that con-

tains a Hamiltonian cycle is called a **Hamiltonian graph**. Similar notions may be defined for *directed graphs*, where each edge (arc) of a path or cycle can only be traced in a single direction (i.e., the vertices are connected with arrows and the edges traced "tail-to-head").A **Hamiltonian decomposition** is an edge decomposition of a graph into Hamiltonian circuits.

Any Hamiltonian cycle can be converted to a Hamiltonian path by removing one of its edges, but a Hamiltonian path can be extended to Hamiltonian cycle only if its endpoints are adjacent. The line graph of a Hamiltonian graph is Eulerian. The line graph of an Eulerian graph is Hamiltonian. A tournament (with more than 2 vertices) is Hamiltonian if and only if it is connected. A Hamiltonian cycle may be used as the basis of a zero-knowledge proof.

Number of different Hamiltonian cycles for a complete graph = $(n-1)! / 2$.

Number of different Hamiltonian cycles for a complete Digraph = $(n-1)!$

Algorithm

Recurse(Path p, endpoint e)
While (e has unvisited neighbors)
{ GetNewNode x; (add x node to P)
PruneGraph. (Prune graph. If result graph does not permit a HC forming, remove x from P and continue)
FormCycle (If P includes all nodes then try to form cycle. Fail, remove x and continue; Succ, return success)
BackTrack: Recurse(P,x)
}
Return fail.

Solution to the problem

A-B-C-D Not possible
A-C-E-F Not possible

Similarly try all possibilities

If a particular path fails then back track and find the solution

Final solution:

A-B-F-E-C-D-A

15. (a) 0-1 Knapsack using the branch and bound
Mathematically the 0-1-knapsack problem can be formulated as:

Let there be n items, x_i to x_n where x_i has a value u_i and weight w_i. The maximum weight that we can carry in the bag is W. It is common to assume that all values and weights are nonnegative. To simplify the representation, we also assume that the items are listed in increasing order of weight.

- Maximize $\sum_{i=1}^{n} u_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \le W, \qquad x_i \in \{0,1\}$

Maximize the sum of the values of the items in the knapsack so that the sum of the weights must be less than the knapsack's capacity.

The **bounded knapsack problem** removes the restriction that there is only one of each item, but restricts the number x_i of copies of each kind of item to an integer value c_i.

Mathematically the bounded knapsack problem can be formulated as:

$$\text{maximize} \sum_{i=1}^{n} u_i x_i \quad \text{subject to} \sum_{i=1}^{n} w_i x_i \le W, \qquad x_i \in \{0,1,\ldots,c_i\}$$

The **unbounded knapsack problem** (**UKP**) places no upper bound on the number of copies of each kind of item and can be formulated as above except for that the only restriction on x_i is that it is a non-negative integer. If the example with the colored bricks above is viewed as an unbounded knapsack problem, then the solution is to take three yellow boxes and three grey boxes.

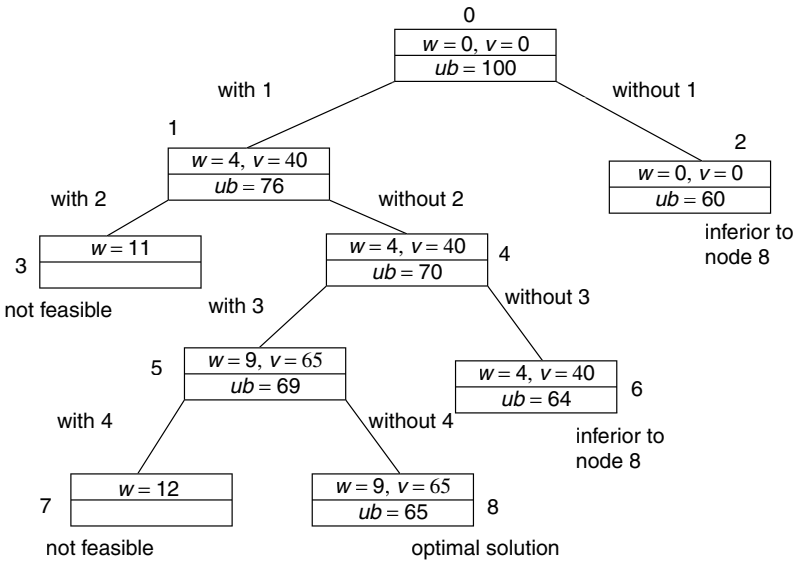
Example

- Capacity W is 10
- Upper bound is \$100 (use fractional value)

Item	Weight	Value	Value / weight
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

- To compute the upper bound, use
  - $ub = v + (W - w)(v_{i+1}/w_{i+1})$
- So the maximum upper bound is
  - pick no items, take maximum profit item
  - $ub = (10 - 0)*(\$10) = \$100$

- After we pick item 1, we calculate the upper bound as
  - all of item 1 (4, $40) + partial of item 2 (7, $42)
  - $40 + (10 – 4)*6 = $76
- If we don't pick item 1:
  - ub = (10 – 0) * ($6) = $60



Or

(b) Biconnected graph

Definition A graph G is biconnected if and only if it contains no articulation points.

Algorithm to determine if a connected graph is biconnected

- Identify all the articulation points in a connected graph
- If graph is not biconnected, determine a set of edges whose inclusion makes the graph biconnected
  - Find the maximal subgraphs of *G* that are biconnected
  - Biconnected component

A Graph *G* is Biconnected if and only if it contains no articulation point. A vertex *V* in a graph is said to be an articulation point if and only if the deletion of vertex *v* together with all its edges incident to v disconnects the graph into to two or more non-empty components. A maximal Biconnected component is Biconnected graph. The graph *G* can be transformed into a Biconnected graph by using the edge addition scheme.

For each articulation point a do

{ let B1, B2,.. BK be the Biconnected components containing vertex a.
le vi, vi != a be the vertex in I
Add to G the edges (*vi*, vi+1)
}
Doing DFS gives DFN numbers to the nodes. From it the L value is found. Calculations and methods are devised to find the articulation point and then the edge connection is made to make it Biconnected. The time taken for TVS is $O(n)$time.

Proof

Lemma 1 Two biconnected components can have at most one vertex in common and this vertex is an articulation point

- No edge can be in two different biconnected components; this will require two common vertices
- Graph G can be transformed into a biconnected graph by using the edge addition scheme.
- If G has p articulation points and b biconnected components, the above algorithm introduces exactly $b - p$ new edges into G

B.E./B.Tech. DEGREE EXAMINATION, MAY/JUNE 2009

Fourth Semester

(Regulation 2004)

Computer Science and Engineering

CS 1201-DESIGN AND ANALYSIS OF ALGORITHMS

(Common to Information Technology)

Time: Three hours **Maximum: 100 marks**

Answer ALL questions

PART A–($10 \times 2 = 20$ marks)

1. Design an algorithm for computing $\sqrt{n}$.
2. Prove or disprove
   If $t(n) \in O(g(n))$ then $g(n) \in \Omega\, t(n)$
3. What is empirical analysis?
4. Justify the need for algorithm visualization.
5. Write a recursive algorithm to perform binary search.
6. State the basic principle of Decrease and Conquer strategy.
7. Differentiate Dynamic Programming and Divide and Conquer.
8. Give two examples of real time problems that could be solved using Greedy algorithm.
9. State the Hamitonian circuit problem.
10. What is the real time application of the assignment problem?

PART B–(5 × 16 = 80 marks)

11. (a) (i) Prove that for every polynomial

$p(n) = a_k n^k + a_{k-1} n^{k-1} + a_{k-2} n^{k-2} + \ldots + a_0$ with $a_k > 0$ belongs to O (n^k) (8)

(ii) Write the sequential search algorithm and analyse for its best, worst and average case behaviour and express this analysis using asymptotic notations. (8)

Or

(b) (i) Explain in detail the algorithm design and analysis process and explain the need for analysis the algorithm for time and space. (10)

(ii) Prove that exponential functions a^k have different orders of growth for different values of base $a > 0$. (6)

12. (a) (i) Consider the following algorithm: (12)

```
Mystery (n)
//Input: A non-negative integer n
S ← 0
For i ← 1 to n do
S ← S + i ∗ i
returns S
```

1. What does this algorithm compute?
2. What is its basic operation?
3. How many times is the basic operation executed?
4. What is the efficiency class of this algorithm?
5. Suggest improvements for this algorithm and justify your improvement by analyzing its efficiency.

(ii) Write the factorial algorithm using recursion and analyse for its time and space complexity. (4)

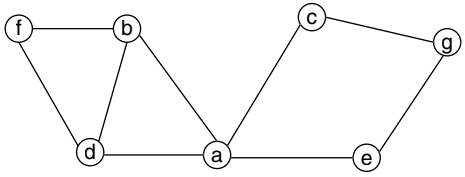
Or

(b) Design a recursive algorithm for computing 2^n for any non negative integer n which is based on the formula $2^n = 2^{n-1} + 2^{n-1}$. Also set up a recurrence relation for the number of additions made by the algorithm and solve it. (16)

13. (a) Write the Quick sort algorithm in detail and analyse for its time and space complexity. What will be the efficiency of quick sort algorithm if the elements are already in ascending order? Using Quicksort algorithm, sort the numbers 21, 34, 2, 65, 36, 98, 20, 11, 32. (16)

Or

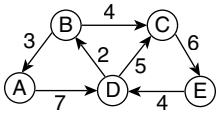
(b) Write the DFS algorithm and analyse for its time and space complexity. What is the data structure used to implement this algorithm? For the following graph perform the DFS and list the nodes. (16)



14. (a) Write the heapsort algorithm and analyse for its time and space complexity. Construct a heap for the following numbers: 23, 54, 3, 27, 98, 57, 65, 35, 90, 21 and use the above algorithm to sort these numbers. What is the limitation of heapsort? (16)

Or

(b) Write Dijkstra's shortest path algorithm and analyse for its time and space complexity. For the following graph, determine the shortest path from vertex A to all other vertices. (16)



15. (a) What is branch and bound technique? Explain how knapsack problem could be solved using branch and bound technique. Solve the

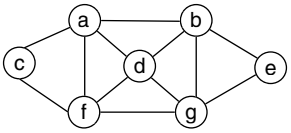
following instance of the knapsack problem by branch and bound algorithm for $W = 16$. (16)

Item	Weight	Value in Rs.
1	10	100
2	7	63
3	8	56
4	4	12

Or

(b) (i) Explain the Subset-sum problem in detail by justifying it using backtracking algorithm. (8)

(ii) Apply backtracking to the problem of finding a Hamiltonian circuit for the following graph: (8)



Solution

PART A

1.

//Problem Description: Computing $\sqrt{n}$
//Input: non-negative value-n
//Output: Square root of n:
If n = 0 return 0
Else return Sqrt (n, 2)

2.

Consider:
$t(n) \in O(g(n))$ is true
using Symmetry and Transpose Symmetry property of asymptotic notation
$f(n) \in O(g(n))$ if and only if
$g(n) \in \Omega(t(n))$
Hence proved.

3.

The principal alternative to the mathematical analysis of an algorithm's efficiency is its empirical analysis.
General plan for empirical analysis of algorithm time efficiency:

1. Understand the experiment's purpose.
2. Decide on the efficiency metric M to be measured and the measurement unit (an operation's count Vs. a time unit)
3. Decide on characteristics of the input sample (its range, size, and so on)
4. Prepare a program implementing the algorithm (or algorithms) for the experimentation.
5. Generate a sample of inputs.
6. Run the algorithm (or algorithms) on the sample's inputs and record the data observed.
7. Analyze the data obtained.

4. Algorithm visualization can be defined as the use of images to convey some useful information about algorithms. That information can be a visual illustration of an algorithm's operation, of its performance on different kinds of inputs, or of its execution speed versus that of other algorithms for the same problem. To accomplish this goal, an algorithm visualization uses graphic elements (points, line segments, two-or-three-dimensional bars and so on) to represent some interesting events in the algorithm's operation.

5.

```
int BinSrch (Type a[], int i, int 1, Type X)
//Given an array a [i : 1] of elements in nondecreasing order, K = i < = 1,
determine whether X is present, and if so, return j such that X = = a[j];
else return 0
  {
     if (1 = i) { //If Small (P)
        if (X = = a[i]) return i;
        else return 0;
     }
     else {//Reduce P into a smaller sub problem.
        int mid = (i + 1)/2;
        if (X = = a[mid]) return mid;
        else if (X < a[Mid]) return BinSrch (a, i, mid-1, X);
        else return BinSrch (a, mid + 1, 1, X);
     }
  }
```

6. In decrease and conquer method, a given problem is,
   1. Divided in to smaller sub problems.
   2. These sub problems are solved independently
   3. Combining all the solutions of sub problems into a solution of the whole.

   → If the sub problems are large enough them decrease and conquer is reapplied.

   → The generated sub problems are usually of same type as the original problem

   Hence recursive algorithms are used in decrease and conquer strategy.

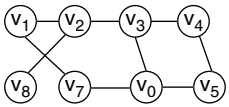
7.

Dynamic programmes	**Divide and Conquer**
It is an algorithm design method that can be used when the solution to a problem can be viewed as a result of sequence of decision	It is easier to solve several small instances of a problem than one large one. They divide the problem into smaller instances of the same problem
It is a general algorithm decision technique for solving problems defined by recurrences with overlapping sub problems	It is the smaller instances recuresvely and finally combine the solution to obtain the solution for the original input.

8.

1. Machine scheduling
2. Container loading

9. Hamilton circuit:



This contains Hamilton an circuits
$v_1, v_2, v_8, v_7, v_6, v_5, v_3, v_1$

10.

- Successive shortest path algorithm
- Job sequencing scheduling

PART B

11. (a) (i) Let,

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + a_{k-2} n^{k-2} + \ldots + a_0 \quad (1)$$

If we treate a_k^i as a constant then the eqn (1) becomes

$$p(n) = A \sum_{i=0}^{k} a^k \quad (2)$$

Where A is $a_0^0 + a_1^1 + a_2^2 + \ldots$
If eqn. (2) is

$$p(n) = A\left[\sum_{i=0}^{n} 1\right] = A[1+1+1\ldots+1] = A(n^1)$$
$$= A \cdot \mathrm{O}(n^1)$$

If eqn. (2) is

$$p(n) = A \sum_{i=0}^{n} n = A[1+2+3\ldots n] = A(n^2)$$
$$= A \cdot \mathrm{O}(n^2)$$

If eqn. (2) is

$$p(n) = A \sum_{i=0}^{n} n^2 = A \cdot \mathrm{O}(n^3)$$

Thus

$$p(n) = A\sum_{i=0}^{k} n = A \cdot O(n^k)$$

Thus neglecting the constant team we will have

$$P(n) = O(n^K)$$

11. (a) (ii) **Analysis of linear search:**

Algorithms are usually analysed to get best case, worst case and average case asymptonic values

Let D_n be the domain of a problem. Where n be the size of the input. Let $I \in D_n$ be an instance of the problem, taken from the domain D_n. Also, take $T(1)$ as the computation time of the algorithm for the instance $I \in D_n$

Best case analysis:

$Bn = \min \{T(1)/I \in D_n\}$

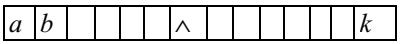
Worst case analysis:

$W(n) = \max \{T(1)/I \in D_n\}$

Average case analysis:

$$A(n) = \sum_{I \in D_n} P(1)T(1)$$

Eg: In an array of some elements, the array is to be searched for the existence of an element x. If the element x is in the array, the corresponding location of the array should be returned as an output. Notice that the element is in the array, if may be in any location that is in the first location, second location, anywhere in the middle or at the end of the array.



01 … I … $n-1$

```
int linear search const char A[], const unseqned int she
{ for (int c = 0; I < size; if t)
   { if (A[i] = = ch)
      return (i);
   } return (−1)
}.
```

B(n) = min {1, 2,… n}
= 1
= O(1)

Required numbers of comparisons for finding the elements at various location

Location of the elements	Number of comparisons required
0	1
1	2
2	3
.	.
.	.
.	.
.	.
$n-1$	n
not in the array	n

Worst case

$W(n) = \max \{1, 2, \ldots, n\}$
$= 2$
$= O(n)$

Average case

$p(I_i) = k/n$ for $0 \le i \le n-1$
$p(I_i) = 1 - k$

$$A(n) = \sum_{i=0}^{n} P(I_i)T(I_i)$$

$$= (k/n)\sum_{n=0}^{n-1}(i+1) + (1-k)n$$

$$= \frac{k}{n} \quad \frac{n(n+1)}{2} + (1-k)n$$

$$= \frac{k(n+1)}{2} + (1-k)n$$

Suppose x is in the array, then $k = 1$
Therefore
$A(n) = (n + 1)/2 = O(n)$
$k = ½$
$\Rightarrow A(n) = (n + 1)/4 + n/2 = (3/4)\ n + 1 = O(n)$

Algorithm	Best case	Worst case	Average case
Linear search	O(1)	O(n)	O(n)

Time complexity

For instances take T(n) = T(n − 1) + 2. The value 2 accounts for the testing conditions and the computations at the second return statement. Now we say bacically that $T(n)$ = O(n)

11 (b) (i) Study of algorithm:

The study of algorithm involves 3 major parts.

1. Designing the algorithm
2. Proving the correctness of the algorithm
3. Analyzing the algorithm

Designing the algorithm

(i) It is a method for solving a problem.
(ii) Each step of an algorithm must be precisely defined and no vague statement should be used.
(iii) Pseudo code is used to describe the algorithm.

Proving the correctness of the algorithm

(i) We have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.
(ii) A Human must be able to perform each step using pencil and paper by giving the required input, use the algorithm and should get the required output in a finite amount of time

Analyzing the algorithm

(i) If deals with the amount of time and space consumed it
(ii) Efficient algorithm can be computed with minimum requirement of time and space.
(iii) Time and space complexity can be reduced only to certain levels.

The need for algorithm analysis:

Criteria:

(i) Time efficiency
– Time required to run an algorithm.
(ii) space efficiency
–space need for an algorithm.
(iii) Measuring an Input's size
–The algorithm efficiency depends on i/p size.
(iv) units for measuring running time
–some unit of time such as sec, millisecond and so on

(v) order of growth
(vi) worst case, best case and average case

11 (b) (ii) $\Rightarrow$ $f(n) = a^k$

When $a = 1$
$f(n) = 1^k$
$f(n) = 1$
$\therefore f(n) = O(1)$

When $a = 2$
$f(n) = 2^k$
$\therefore f(n) = O(2^k)$

When $a = 3$
$f(n) = 3^k$
$\therefore f(n) = O(3^k)$
$\therefore$ If is proved a^k has different order of growth her different values of base $a > 0$.

12 (a) (i) Consider the following algorithm

```
S = 0
for i = 1 to n do
   S = S + i * i
return S
```

Solution:

(a) What does the algorithm compute?
Algorithm computes the sum of quadric series ($1^2 + 2^2 + \ldots + n^2$)

(b) What is its basic operation?
Basic operation is multiplication

(c) How many times is the basic operation executed?
if input series is n
Let $t(n)$ denote the number of times the basic operation is executed
$\therefore t(n) = n$

(d) What is the efficiency of the algorithm?
The efficiency class is linear n
$t(n)$ is in $O(n)$

(e) Suggest improvement for this algorithm and justify your improvement by analyzing its efficiency.

```
S ← 0
i ← 1
```

```
while (i ≤ n)
{
 i ++
 S ← S + i * i
}
return S
```

As the number of execution time for the given algorithm is n, it fails at $n + 1$ condition. In this algorithm, it executes upto n, it does not goes beyond n.
Hence the efficiency is improved in this algorithm compared to previous algorithm.

12. (a) (ii)

```
# include <stdio.h>
# include <conio.h>
void main ()
{
    int n;
    printf ("\n Enter the value of n");
    scanf ("%d", &n);
    printf ("/n The factorial is = %d", fact(n))
    scanf("/n The factorial is = %d", func(n));
}
    int fact (int n)
   {int a, b;
    if (n = = 0)
    return 1;
    a = n – 1;
    b = fact(a);
    return (n * b);
}
```

Non-recurrsive

```
# include <stdio.h>
# include <conio.h>
void main ()
{
    int n;
    printf ("/n Enter the value of n");
```

Factorial using recursion

```
def fact(n):
```

if n < –1:
result – 1
else:
result = n ∗ fact (n-1)
return result
Let $T(n)$ = amount of time to compute fact(n)
Don't need exact solution for $T(n)$, just big O

Best case: n ≤ 1
Takes small constant amount of time to compute.
Like that constant an arbitrary name:
$T(n) = a, n \leq 1$

12. (b) //Algorithm f(2^n)
// Computing 2^n
// Input non-negative integer
// Output: The value of 2^n
if $n = 0$ return 1
else return $F(2^{n-1}) * 2$

$F(2^n) = 2 \cdot F(2^{n-1}) + 1$ for $n > 0$
$F(2^0) = 0$
$F(2^n) = 2 \cdot F(2^{n-1}) + 1$

∴ Substitute

$F(2^{n-1}) = 2 \cdot F(2^{n-2}) + 1$
$F(2^n) = 2[2 \cdot F(2^{n-2}) + 1] = 4 \cdot F(2^{n-2}) + 2$

∴ Substitute

$F(2^{n-2}) = 2 \cdot F(2^{n-3}) + 1$
$F(2^n) = 4[2 \cdot F(2^{n-3}) + 1] + 2$
$= 8 \cdot F(2^{n-3}) + 3$

⋮

$= 2^i \cdot F(2^{n-i}) + i$
$= 2^n \cdot F(2^{n-n}) + n$

∴ $F(2^n) = 2^n F(2^0) + n = n$
∴ The time complexity should be n.

13. (a) Quick-sort:
ALGORITHM: Quicksort(A[l…r])
// sort a subarray by quicksort
// Input: A subarray A[l…r] of [0…n – 1], defined by its
// left and right indices l and r.
// Output: subarray A[l…r] sorted in nondecreasing order
if l < r

S ← partition (A[l…r]) // S is a split position
Quickshort (A[l…s − 1])
Quickshort (A[s + 1…r])

If the elements are already in ascending order there will be a Best case scenario of quicksort algorithm

$C_{best}(1) = 0$ [No. of key comparisons]

Elements: 21, 34, 2, 65, 36, 98, 20, 11, 32.
Step 1: (21), 34, 2, 65, 36, 98, 20, 11, 32
Step 2: (2), 34, (21), 65, 36, 98, 20, 11, 32
Step 3: 2, (11), 21, 65, 36, 98, 20, (34), 32.
Step 4: 2, 11, (20), 65, 36, 98, (21), 34, 32
Step 5: 2, 11, 20, 21, (32), 98, 65, 34, (36)
Step 6: 2, 11, 20, 21, 32, (34), 65, (98), 36
Step 7: 2, 11, 20, 21, 32, 34, (36), 98, (65)
Step 8: 2, 11, 20, 21, 32, 34, 36, (65), (98)

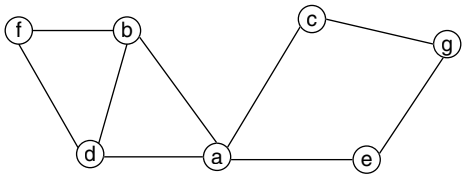
∴ Quicksort: {2, 11, 20, 21, 32, 34, 36, 65, 98

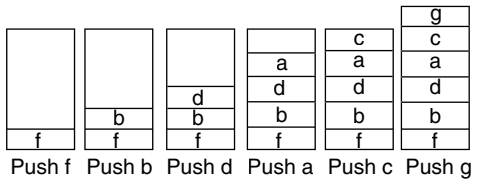
Space complexity of Quicksort:
O(n) auxillary (naive)
O(log n) auxillary
Time complexity: O(n^2)

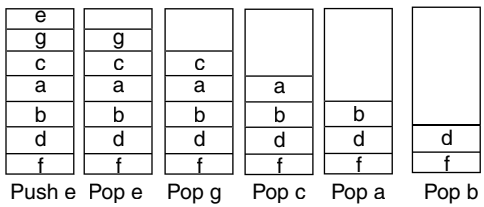
13. (b) **Depth first search**

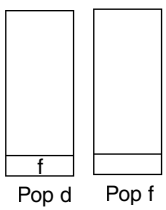
ALGORITHM:

```
// Loop, Initialisation
for u ← v1, v2, … vn // for each u in v
{
   set color [u] ← white
   set pre [u] ← NULL
}
set time ← 0
for u ← v1, v2, …vn
{
   if (color = white)
   case to DFS visit (u)
}
return
```

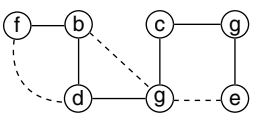






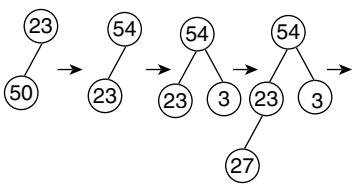


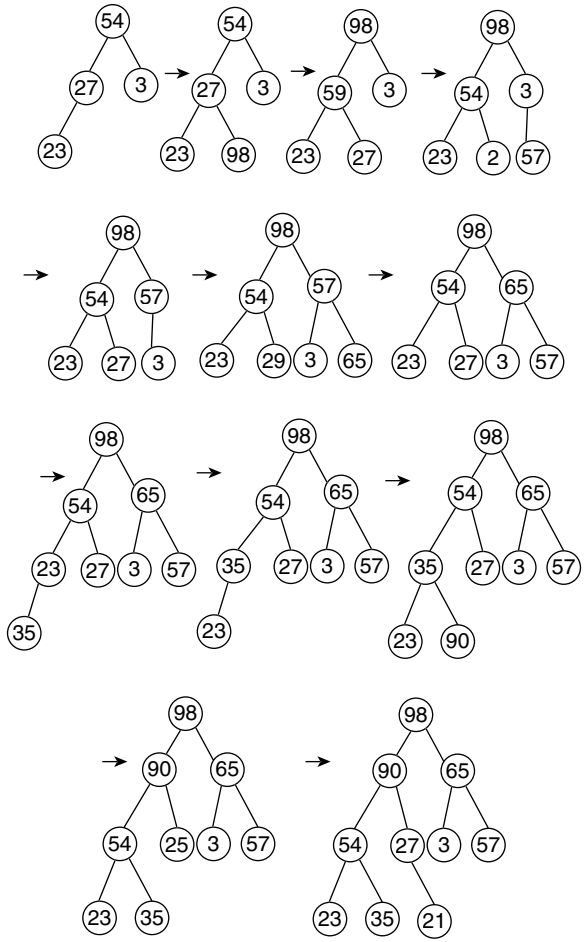
DFS



Time complexity O(|V| + |E|)
Space complexity O(|V|)

14 (a) Heapsort:





Time complexity of heap sort is O(n)
Min heap O(log n) WC
Max heap O(log n) WC
Space complexity of heap sort O(n)total, O(1) auxicllary.

ALGORITHM:

call to Build-Heap (H)
for i ← length [H] down to 2.
set H[1] ↔ H[i]
set heap-size [H] ← heapsize [H] − 1
call to heapify (H, 1)
return

Limitations:

- heap sort is not a stable sort
- heap sort is not used in external sorting. If faces the issue of locality of reference.

14. (b) The Dijkstra procedure finds the shortest path from the given graph. It uses the concept of relaxing all edges of graph repeatedly once.

→ Initialization loop

for $V_i \leftarrow V_1, V_2, \ldots V_n \rightarrow$ for each V_i in V

{

set dist $[V_i] \leftarrow +\infty$

set color $[V_i] \leftarrow$ White

set pre $[V_i] \leftarrow$ NULL

}

set dist $[V_s] \leftarrow 0$

→Putting all vertices in Q

Set Q ← Call to prio Qnene(V) →Put vertices in Q

→ Loop 1, Processing all vertices

While (non-empty (Q)]

→ Start Loop 2

for $V_j \leftarrow V_1, V_2, \ldots$ Adj$(V_i) \rightarrow$ for each V_j in Adj (V_i)

{

if (dist $[V_i]$ + W(V_i, V_j) < dist $\{V_j\}$ then

{

set (dist $[V_i] \leftarrow$ dist $[V_i]$ + (V_i, V_j)

call to Decrease -key (Q, V_i, dist $\{V_j\}$)

} set pre $[V_j] \leftarrow V_i$

→ End of loop 2

→ Coloring

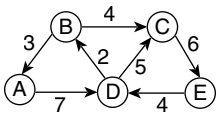
set color $[V_j] \leftarrow$ Block

→ end of loop 1

→ Return at the point of call return.

This operation like Insert (), Extract_minimum (), and Decrease-key () can be performed on priority quene, each in O(log n) time.

→ Space complexity O (|V| |E|)



Dijkstra's shortest path:
5 nodes, 7 edges in the graph. There are 5 nodes in this graph path from which? BA distances are:

A·····[0]···→A
A·····[9]..···→B
A·····[12]·····→C
A·····[7]..···→D
A·····[22]·····→E

15. (a) knapsack problem by branch and bound.

Item	Weight	Values
1	10	$ 100
2	7	$ 63
3	8	$ 56
4	4	$ 12

$$u_b = v + (W - w) * v_{i+1} / W_{i+1}$$

First arrange the items $v_1/W_1 > v_2/W_2 \ldots v_i / W_i$

$v_i W_i$
10
9
7
3

Computation at node I
No item is selected initially.
Hence $w = 0$, $v = 0$, $W = 16$, $v_{i+1}/W_{i+1} = v_1/W_1 = 10$

$$u_b = v + (W - w) * v_{i+1}/W_{i+1}$$
$$u_b = 0 + (16 - 0) * 10$$
$$u_b = 160.$$

Computation at Node II
Select item 1
Hence $w = 10$, $v = \$100$, $v_{i+1}/W_{i+1} = v_2/W_2 = 9$

$$u_b = v + (W - w) * v_{i+1}/W_{i+1}$$
$$= 100 + (16 - 10) * 9.$$
$$= 100 + 54$$
$$u_b = 154$$

Computation at Node III
Select without item 1
Hence $w = 0$, $v = 0$, $v_{i+1}/W_{i+1} = v_2/W_2 = 9$

$$u_b = v + (W - w) * v_{i+1}/W_{i+1}$$
$$= 0 + (16 - 0) * 9$$
$$u_b = 144$$

Computation at Node IV

Select item 1 and item 2

$\therefore w = 10 + 7 = 17$, $v = 100 + 63 = \$163$.

But this exceeds the capacity W.

Computation at Node V

Select item 1, without item 2

$$\therefore w = 10,\ v = 100\ v_{i+1}/W_{i+1} = v_3/W_3 = 7$$
$$u_b = v + (W - w) * v_{i+1}/W_{i+1}$$
$$= 100 + (16 - 10) * 7$$

Computation at Node IX

Select item 1 and item 4

$\therefore w = 10 + 4$, $v = 100 + 12 = 112\ v_{i+1}/W_{i+1} = 6$

as there is no next item for selection

$$u_b = v + (W - w) * v_{i+1}/W_{i+1}$$
$$= 112 + (16 - 14) * 0$$
$$= 112$$

Computation at Node X

Select item 2 and item 3

$\therefore w = 7 + 8 = 15$, $v = 63 + 56 = 119\ v_{i+1}/W_{i+1} = v_4/W_4 = 3$

$$u_b = v + (W - w) * v_{i+1}/W_{i+1}$$
$$= 119 + (16 - 15) * 3$$
$$= 119 + 3$$
$$= 122$$

As we get profit with selection of item 2 and 3. Also u_b is also greates. Hence the solution to this knapsack problemes (item 2, item 3)

$$= 100 + 42$$
$$u_b = 142$$

Computation at Node VI

Select without item, but select item 2,

$w = 7$, $v = 63\ v_{i+1}/W_{i+1} = v_3/W_3 = 7$

$$u_b = v + (W - w) * v_{i+1}/W_{i+1}$$
$$= 63 + (16 - 7) * 7$$
$$= 63 + 63$$
$$= 126$$

Computation at Node VII

Select item 1, item 3,

$\therefore w = 10 + 8 = 18$, $v = 100 + 56 = \$156$,

As this exceeds the knapsack's capacity. So, It is not feasible solution.

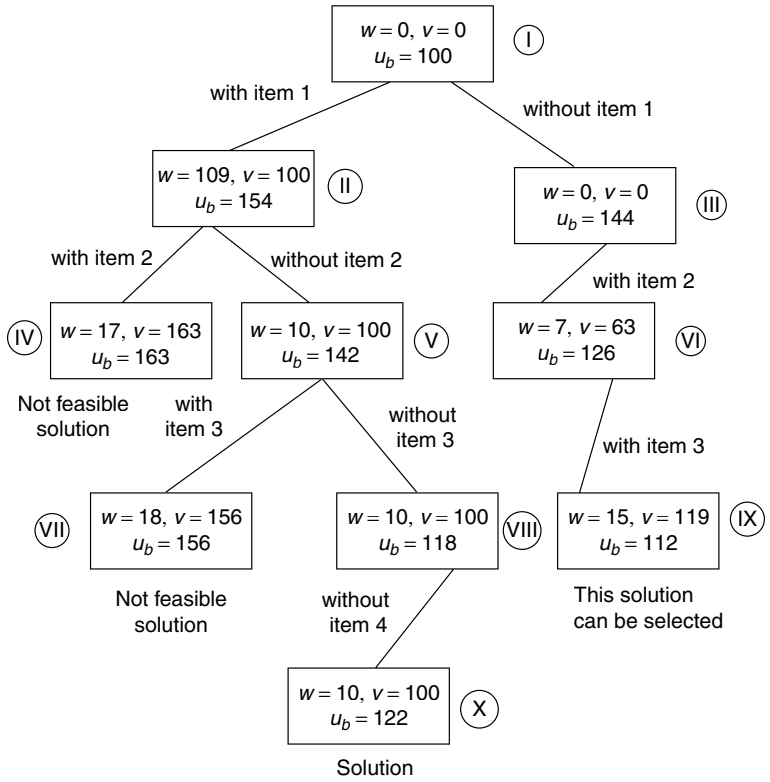
Computation at Node VIII

Select item 1,without item 2, and without item 3,

$w = 10$, $v = 100$, $v_{i+1}/W_{i+1} = 3$

$$u_b = v + (W - w) * v_{i+1}/W_{i+1}$$
$$= 100 + (16 - 10) * 3$$
$$= 100 + 18$$
$$= 118$$

State space tree:



15. (b) (i) **Sum of subsets:**

We are given n district positive numbers and we desire to find all combination of these numbers whose sum are 'm' this is called the sum of subset problem.

Solution to sum of subsets problem:

- sort the weights in non decreasing order.

- The root of the space free represent the starting point, with no decision about the given elements.
- Its left and right child represent inclusion and exclusion of X_i in a set.
- The node to be expanded, check it with the following condition

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

The handed node can be identified with the following condition.

$$B_k (x_1, x_2 \ldots., x_k) = \text{tree}$$

iff $$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

- Backtrack the bounded node and find alternative solution
- Thus a path from root to a node on the i^{th} level of the tree indicates which of the first i numbers have been included in the subsets represented by that node.
- We can terminate the node as non promising if either of the 2 ingratiation holds

$S' + W_{i+1} > m$ (Where S' in too large)
$S' + W_{i+1} < m$ (Where S' in too small)

$$S^1 = \sum_{i=1}^{k} W_1 x_i$$

Example for sum of subset problem

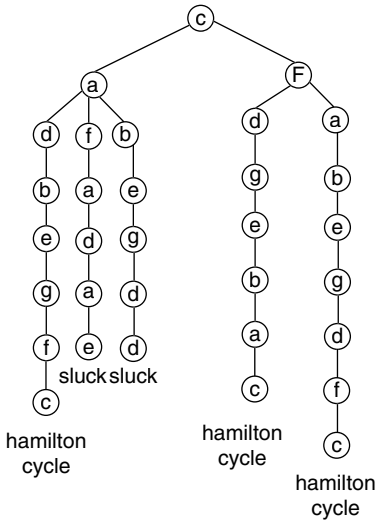
Let $W = \{5, 10, 12, 13, 15, 18\}$ and $m = 30$. Find all possible subsets of 'W' that sum 'm' and draw state space tree.
Solution set = {(5, 10, 5), (5, 12, 13), (12, 18)} solution set can also represented in another way.
Solution set = {(1, 1, 0, 0, 1, 0), (1, 0, 1, 1, 0, 0), (0, 0, 1, 0, 0, 1)}

Complexity of sum of subsets problem

- The algorithm splits arbitrarily N elements into 2 sets of $N/2$ each
- Sort the arrays, it require $O(2^{N/2}N)$

15. (b) (ii) Then the Hamilton cycle is C-a-b-e-g-d-f-c. This can be solved by backtracking approach.

Clearly the backtrack approach is adopted. For instance c-a-f-d-b-g-e for returning to A we have to reverse at least one vertex. Hence we backtracked and from D node another path is chosen. C-a-b-e-g-d-f-c Which is Hamilton cycle.




```
Algorithm
H_cycle (k)
{
 repeat
{
Next_vertex (k);
If (x[k] = 0) then
return;
if (k = n) then
Write (x[1:n])
 else
   H_cycle (k + 1)
} until (false);
}
Generating Hamilton cycle
{
 while (1)
{
 x[k] = = x[k + 1] mod (n + 1);
```

```
 if (x[k] = 0) then
 return;
if (G[x[k − 1], x[k]] = 1) then
   {
for (j = 1 to k − 1) do
if (x[j] = x[k]) then
   break;
if (j = k) then
{
if (j = k) then
{ if (k < n) OR ((k = n) AND G(x[n], x[1]] = 1)) then
return:
          }
            }
              }
                }
```

B.E./B.Tech. DEGREE EXAMINATION, NOV/DEC 2008

Fourth Semester

(Regulation 2004)

Computer Science and Engineering

CS 1201 – DESIGN AND ANALYSIS OF ALGORITHMS

(Common to Information Technology)

Time: Three hours **Maximum: 100 marks**

Answer All Questions

PART A – ($10 \times 2 = 20$ marks)

1. Define an algorithm.
2. Design an algorithm for checking whether the given word is a palindrome or not i.e., whether the word is the same even when reversed. E.g., MADAM is a palindrome.
3. List out the steps for empirical analysis of algorithm efficiency
4. What is the significance of Fibonacci number of sequence?
5. Give an example problem that cannot be solved by a Brute–Force attack.
6. Write a pseudo code for a divide and conquer algorithm for finding the position of the largest element in an array of n numbers.
7. Define a heap.
8. Give the pseudo code of the Warshall's algorithm.
9. When do you terminate the search path in a state-space tree of a branch and bound algorithm?
10. Define a Knapsack problem.

PART B – (5 × 16 = 80 marks)

11. (a) (i) Elaborate the various asymptotic metrics used to evaluate the efficiency of the algorithm. (10)

(ii) Use the most appropriate notation to indicate the time efficiency class of a sequential search. (6)

(1) in the worst case
(2) in the best case
(3) in the average case

Or

(b) Specify the Euclid's algorithm, the consecutive integer checking algorithm and the middle-school algorithm for computing the greatest common divisor of two integers. Which of them is simpler? Which is more efficient? Why? (16)

12. (a) Consider the following algorithm:

```
ALGORITHM Secret (A [0,…,n-1])
    //Input: An array A[0,…,n-1] of n real numbers
    minval ← A [0]; maxval ← A [0];
    for i ← 1 to n-1 do
        if A [i] < minval
                minval ← A [i];
        if A [i] > maxval
                maxval ← A [i];
        return maxval-minval;
```

(i) What does this algorithm compute? (2)
(ii) What is the basic operation? (2)
(iii) How many times is the basic operation computed? (4)
(iv) What is the efficiency class of this algorithm? (3)
(v) Suggest an improvement or a better algorithm altogether and indicate the efficiency class. If you can't do it, prove that in fact it can't be done. (5)

Or

(b) Consider the following recursive algorithm for computing the sum of the first n cubes: $S(n) = 1^3 + 2^3 + \ldots + n^3$;

ALGORITHM S (n)

// Input: A +ve integer n;

// Output: The sum of the first n cubes.

if n = 1 return 1

else return S(n – 1) + n* n* n

(i) Set up and solve a recurrence relation for the number of the algorithm basic operation is executed. (8)

(ii) How does this algorithm compare with the straight forward non-recursive algorithm for computing this function? (8)

13. (a) (i) Describe the Quick sort algorithm. (10)

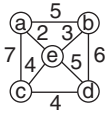
(ii) Apply Quick sort to sort the list E, X, A, M, P, L, E in alphabetical order. Draw the tree of the recursive calls made. (6)

Or

(b) Compare and contrast the Depth First Search and Breadth First Search algorithms. How do they fit into the decrease and conquer strategy? (16)

14. (a) (i) Describe the Prim's algorithm for finding the minimum cost spanning tree. (10)

(ii) Apply the Prim's algorithm to the following graph: (6)



Or

(b) For each of the following lists, construct an AVL tree by inserting their elements successively, starting with an empty tree.

(i) 1, 2, 3, 4, 5, 6 (5)

(ii) 6, 5, 4, 3, 2, 1 (5)

(iii) 3, 6, 5, 1, 2, 4 (6)

15. (a) Using backtracking enumerate how can you solve the following problems:

(i) 8-Queens problem. (8)

(ii) Hamiltonian circuit problem. (8)

Or

(b) (i) Solve the following instance of the Knapsack problem by branch and bound algorithm. (8)

Item	Weight	Values
1	10	$100
2	7	$63
3	8	$56
4	4	$12
$W = 16$		

(ii) Give an example for the best case input for the branch and bound algorithm for the assignment problem. (4)

(iii) In the best case, how many nodes will be in the state space tree of the branch and bound algorithm for the assignment problem. (4)

B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2008

Fourth Semester

(Regulation 2004)

Computer Science and Engineering

CS 1201 – DESIGN AND ANALYSIS OF ALGORITHMS

(Common to Information Technology)

Time: Three hours **Maximum: 100 marks**

Answer All Questions

PART A – ($10 \times 2 = 20$ marks)

1. Define Big oh notation.
2. Enumerate some important types of problems.
3. How will you measure Input size of algorithms?
4. What is the average case complexity of linear search algorithm?
5. Write the procedure for selection sort.
6. Define depth first searching technique.
7. What is the fundamental philosophy of transform and conquer method?
8. What is AVL tree?
9. State if backtracking always produces optimal solution.
10. Explain briefly branch and bound technique for solving problems.

PART B – ($5 \times 16 = 80$ marks)

11. (a) (i) Define the asymptotic notations used for best case average case and worst case analysis of algorithms.

(ii) Write an algorithm for finding maximum element of an array, perform best, worst and average case complexity with appropriate order notations.

Or

(b) Write an algorithm to find mean and variance of an array perform best, worst and average case complexity, defining the notations used for each type of analysis.

12. (a) Derive the recurrence equation for Fibonnocci series. Perform complexity analysis for the same.

Or

(b) Explain in detail, the techniques for algorithm visualization.

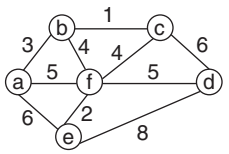
13. (a) Explain in detail quick sorting method. Provide a complete analysis of quick sort.

Or

(b) Explain in detail merge sort. Illustrate the algorithm with a numeric example. Provide complete analysis of the same.

Or

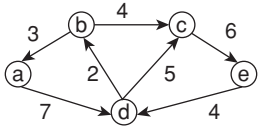
14. (a)



Apply Prim's algorithm and Kruskal algorithm to the graph to obtain minimum spanning tree. Do these algorithms always generate same output – Justify.

Or

(b) Solve the following instance of the single-source shortest-paths problem with vertex *a* as the source:



Write the algorithm, for the above problem.

15. (a) Explain *N*-queens problem with an algorithm. Explain why back tracking is defined as a default procedure of last resort for solving problems. (10 + 6)

Or

(b)

	Job1	Job2	Job3	Job4
Person *a*	9	2	7	8
b	6	4	3	7
c	5	8	1	8
d	7	6	9	4

Consider the above matrix for assignment problem involving persons and jobs. Explain in detail how branch and bound technique is useful is solving assignment problems.

B.E./B.Tech. DEGREE EXAMINATION, NOV/DEC 2007

Fourth Semester

(Regulation 2004)

Computer Science and Engineering

CS 1201 – DESIGN AND ANALYSIS OF ALGORITHMS

(Common to Information Technology)

Time: Three hours **Maximum: 100 marks**

Answer All Questions

PART A – ($10 \times 2 = 20$ marks)

1. What is meant by stepwise refinement?
2. How is the efficiency of an algorithm defined?
3. Give the smoothness rule applied for recurrence relation.
4. What is algorithm visualization?
5. State the time complexity of bubble sort algorithm.
6. List out any two drawbacks of binary search algorithm.
7. What is mathematical modeling?
8. What is pre-structuring? Give examples.
9. What is heuristics?
10. What is the metric used to measure the accuracy of approximation of algorithms?

PART B – ($5 \times 16 = 80$ marks)

11. (a) (i) What are the important problem types focused by the researchers? Explain any two with example. (10)

(ii) What is empirical analysis of an algorithm? Discuss its strength and weakness. (6)

Or

(b) (i) Discuss the fundamentals of analysis framework. (10)

(ii) Explain the various asymptotic notations used in algorithm design. (6)

12. (a) (i) Discuss the general plan for analyzing the efficiency of Non-recursive algorithms. (6)

(ii) Write an algorithm to find the number of binary digits in the binary representation of a positive decimal integer and analyse its efficiency. (10)

Or

(b) What is the principal alternative to mathematical analysis of an algorithm? Explain the steps in analyzing the efficiency of this analysis with example. (16)

13. (a) Find the number of comparisons made by the sentinel version of sequential search algorithm for (i) in worst case and (ii) in average case. (16)

Or

(b) Give a suitable example and explain the Depth-First search algorithm. (16)

14. (a) (i) Define Heap. Explain the properties of heap. (6)

(ii) With a simple example explain heap sort algorithm. (10)

Or

(b) Define spanning tree. Discuss the design steps in Kruskal's algorithm to construct minimum spanning tree with an example. (16)

15. (a) Explain subset – sum problem and discuss the possible solution strategies using backtracking. (16)

Or

(b) Discuss the solution for Knapsack problem using branch and bound technique. (16)

B.E./B.Tech. DEGREE EXAMINATION, MAY/JUNE 2007

Fourth Semester

(Regulation 2004)

Computer Science and Engineering

CS 1201 – DESIGN AND ANALYSIS OF ALGORITHMS

(Common to Information Technology)

Time: Three hours **Maximum: 100 marks**

Answer All Questions

PART A – (10 × 2 = 20 marks)

1. Define an algorithm.
2. Define order of an algorithm.
3. Define loop.
4. What is recursive call?
5. What are the objectives of sorting algorithms?
6. Why is bubble sort called by that name?
7. Define AVL tree.
8. What is a minimum cost spanning tree?
9. What is meant by optimal solution?
10. What do you mean by row major and column major?

PART B – (5 × 16 = 80 marks)

11. (a) Describe briefly the notions of complexity of an algorithm. (16)

Or

(b) (i) What is pseudo-code? Explain with an example. (8)

(ii) Find the complexity ($C(n)$) of the algorithm for the worst case, best case and average case. (evaluate average case complexity for $n = 3$, where n is number of inputs) (8)

12. (a) Write an algorithm for a given numbers n to generate the n^{th} number of the Fibonacci sequence. (16)

Or

(b) Explain the pros and cons of the empirical analysis of algorithm. (16)

13. (a) (i) Write an algorithm to sort a set of 'N' numbers using insertion sort. (8)

(ii) Trace the algorithm for the following set of numbers: 20, 35, 18, 8, 14, 41, 3, 39. (8)

Or

(b) (i) Explain the difference between depth-first and breadth-first searches. (8)

(ii) Mention any three search algorithms which is preferred in general? Why? (8)

14. (a) Write the Dijikstra's algorithm. (16)

Or

(b) Explain KRUSKAL'S algorithm. (16)

15. (a) What is back tracking? Explain in detail. (16)

Or

(b) What is branch and bound? Explain in detail. (16)

B.E./B.Tech. DEGREE EXAMINATION, NOV/DEC 2006

Fourth Semester

Computer Science and Engineering

CS 1201 – DESIGN AND ANALYSIS OF ALGORITHMS

(Common to Information Technology)

Time: Three hours **Maximum: 100 marks**

Answer All Questions

PART A – ($10 \times 2 = 20$ marks)

1. What is an algorithm design technique?
2. Compare the order of growth $n!$ and 2^n.
3. What is the tool for analyzing the time efficiency of a non recursive algorithm?
4. What is algorithm animation?
5. Compare DFS and BFS.
6. Find the number of comparisons made by the sequential search in the worst and best case.
7. How efficient is Prim's algorithm?
8. What do you mean by Huffman Code?
9. What is state space tree?
10. What are the additional items required for branch and bound compare backtracking technique?

PART B – (5 × 16 = 80 marks)

11. (a) (i) Discuss briefly the sequence of steps in designing and analyzing an algorithm. (10)

(ii) Explain some of the problem types used in the design of algorithm. (6)

Or

(b) (i) Explain the general frame work for analyzing the efficiency of algorithms. (8)

(ii) Explain the various asymptotic efficiencies of an algorithm. (8)

12. (a) (i) Design a recursive algorithm to compute the factorial function $F(n) = n!$ for an arbitrary non negative integer n and also derive the recurrence relation. (10)

(ii) Discuss the features of animation of an algorithm. (6)

Or

(b) (i) Design a non recursive algorithm for computing the product of two $n \times n$ matrices and also find the time efficiency of the algorithm. (10)

(ii) Write short notes on algorithm visualization and its applications. (6)

13. (a) (i) Write a pseudo code for divide and conquer algorithm for merging two sorted arrays into a single sorted one. Explain with an example. (12)

(ii) Set up and solve a recurrence relation for the number of key comparisons made by the above pseudo code. (4)

Or

(b) Design a recursive decrease-by-one algorithm for sorting the n real numbers in an array with an example and also determine the number of key comparisons and time efficiency of the algorithm. (16)

14. (a) (i) Construct a minimum spanning tree using Kruskal's algorithm with your own example. (10)

(ii) Explain single L-rotation and of the double RL-rotation with general form. (6)

Or

(b) Solve the all-pairs shortest-path problem for the diagraph with the weight matrix given below: (16)

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

15. (a) Apply backtracking technique to solve the following instance of the subset sum problem. $S = \{1, 3, 4, 5\}$ and $d = 11$. (16)

Or

(b) Solve the following instance of the Knapsack problem by the branch-and-bound algorithm. (16)

Item	Weight	Value
1	4	\$40
2	7	\$42
3	5	\$25
4	3	\$12

The Knapsack's capacity $W = 10$.

B.E./B. Tech. DEGREE EXAMINATION, MAY/JUNE 2006

Fourth Semester

(Regulation 2004)

Computer Science and Engineering

CS 1201 – DESIGN AND ANALYSIS OF ALGORITHMS

(Common to Information Technology)

Time: Three hours **Maximum: 100 marks**

Answer All questions

PART A – (10 × 2 = 20 marks)

1. Define algorithm.
2. What is Big 'Oh' notation?
3. What is an activation frame?
4. Define external path length.
5. Give the recurrence equation for the worst case behavior of merge sort.
6. Name any two methods for pattern matching.
7. When do you say a tree as minimum spanning tree?
8. How will you construct an optimal binary search tree?
9. Define backtracking.
10. What is Hamiltonian cycle in an undirected graph?

PART B – (5 × 16 = 80 marks)

11. (i) Explain the various criteria used for analyzing algorithms. (10)
    (ii) List the properties of various asymptotic notations. (6)

12. (a) (i) Explain the necessary steps for analyzing the efficiency of recursive algorithms. (10)

(ii) Write short notes on algorithm visualization. (6)

Or

(b) (i) Explain the activation frame and activation tree for finding the Fibonacci series. (10)

(ii) What are the pros and cons of the empirical analysis of algorithm? (6)

13. (a) (i) Sort the following set of elements using Quick Sort. (10)
12, 24, 8, 71, 4, 23, 6

(ii) Give a detailed note on divide and conquer techniques. (6)

Or

(b) (i) Write an algorithm for searching an element using binary search method. Give an example. (12)

(ii) Compare and contrast BFS and DFS. (4)

14. (a) Explain the method of finding the minimum spanning tree for a connected graph using Prim's algorithm. (16)

Or

(b) How will you find the shortest path between two give vertices using Dijkstra's algorithm? Explain. (16)

15. (a) (i) Describe the travelling salesman problem and discuss how to solve it using dynamic programming. (10)

(ii) Write short notes on n – Queen's problem. (6)

Or

(b) Discuss the use of Greedy method in solving Knapsack problem and subset-sum problem. (16)