B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2015.

Fourth Semester

Computer Science and Engineering

CS 6402 — DESIGN AND ANALYSIS OF ALGORITHMS

(Common to Information Technology)

(Regulation 2013)

Time : Three hours   Maximum : 100 marks

Answer ALL questions.

PART A — ($10 \times 2 = 20$ marks)

1. Write an algorithm to find the number of binary digits in the binary representation of a positive decimal integer.

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

2. Write down the properties of asymptotic notations.

If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

3. Design a brute-force algorithm for computing the value of a polynomial $p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x \cdots a_0$ at a given point x_0 and determine its worst-case efficiency class.

```
//Input: Array P[0..n] of the coefficients of a polynomial of degree n,
//         stored from the lowest to the highest and a number x
//Output: The value of the polynomial at the point x
p ← 0.0
for i ← n downto 0 do
    power ← 1
    for j ← 1 to i do
        power ← power * x
    p ← p + P[i] * power
return p
```

$$M(n) = \sum_{i=0}^{n} \sum_{j=1}^{i} 1 = \sum_{i=0}^{n} i == \frac{n(n+1)}{2} \in \Theta(n^2).$$

4. Derive the complexity of Binary Search algorithm.
The recurrence equation is:

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 1.$$

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil.$$

5. Write down the optimization technique used for Warshall's algorithm. State the rules and assumptions which are implied behind that.
Dynamic programming. Dynamic programming is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a given problem's solution to solutions of its smaller subproblems. Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained.

6. List out the memory functions used under Dynamic Programming.
Memory functions use under dynamic programming technique is called memoization. The goal is to get a method that solves only subproblems that are necessary and does so only once. It is a top down approach instead for classic dynamic programming, which is bottom up.

7. What do you mean by 'perfect matching' in bipartite graphs?
A perfect matching is a matching in which each node has exactly one edge incident on it.

A Bipartite Graph $G = (V, E)$ is a graph in which the vertex set V can be divided into two disjoint subsets X and Y such that every edge $e \in E$ has one end point in X and the other end point in Y. A matching M is a subset of edges such that each node in V appears in at most one edge in M.

Theorem 5.1.3 *A Bipartite graph G(V,E) has a Perfect Matching iff for every subset* $S \subseteq X or S \subseteq Y$*, the size of the neighbors of S is at least as large as S, i.e* $|\Gamma(S)| \geq |S|$

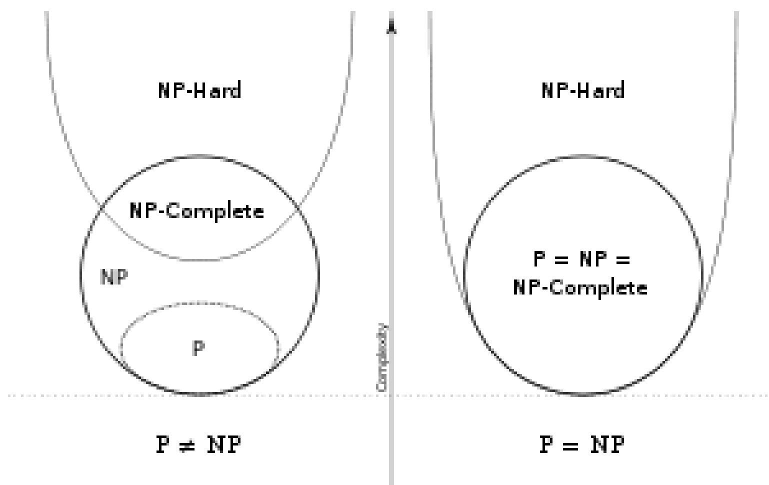
8. Define flow 'cut'.
The "cut" has the following property: if all the edges of a cut were deleted from the network, there would be no directed path from source to sink.

9. How NP-Hard problems are different from NP-Complete?
NP: A decision problem where instances of the problem for which the answer is yes have proofs that can be verified in polynomial time.

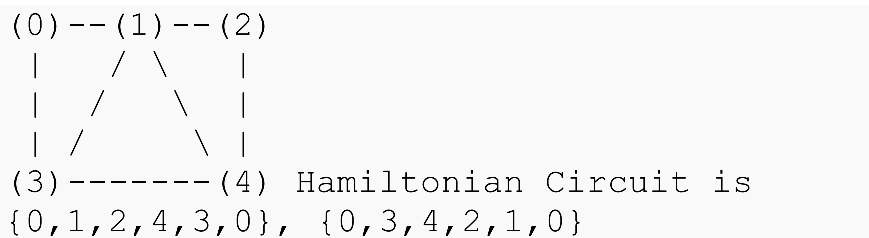
NP-Complete: An NP problem X for which it is possible to reduce any other NP problem Y to X in polynomial time. Intuitively this means that we can solve Y quickly if we know how to solve X quickly.
A decision problem D is said to be NP-complete if:

1. it belongs to class NP
2. every problem in NP is polynomially reducible to D

NP Hard: These are the problems that are even harder than the NP-complete problems.



10. Define Hamiltonian Circuit problem.
Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path.



PART B — (5 × 16 = 80 marks)

11. (a) If you have to solve the searching problem for a list of n numbers, how can you take advantage of the fact that the list is known to be sorted? Give separate answers for

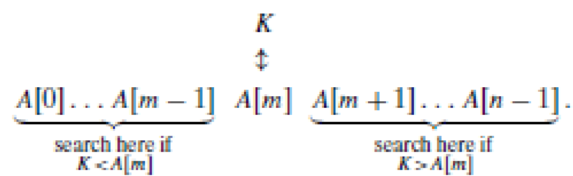
(i) lists represented as arrays.

(ii) lists represented as linked lists.

Compare the time complexities involved in the analysis of both the algorithms. (16)

Answer 11(a) (i) Use Binary Search [Algorithm 4 Marks, Analysis 4 Marks]

```
ALGORITHM  BinarySearch(A[0..n − 1], K)
    //Implements nonrecursive binary search
    //Input: An array A[0..n − 1] sorted in ascending order and
    //        a search key K
    //Output: An index of the array's element that is equal to K
    //          or −1 if there is no such element
    l ← 0; r ← n − 1
    while l ≤ r do
        m ← ⌊(l + r)/2⌋
        if K = A[m] return m
        else if K < A[m] r ← m − 1
        else l ← m + 1
    return −1
```



The recurrence equation is:

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 1.$$

Solving by master's theorem, we get

$$C_{worst}(2^k) = k + 1 = \log_2 n + 1.$$

$$C_{avg}(n) \approx \log_2 n.$$

(ii) When searching in a sorted linked list, stop as soon as an element greater than or equal to the search key is encountered.

[Algorithm 4 Marks, Analysis 4 marks]

ALGORITHM *SequentialSearch2*$(A[0..n], K)$

```
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0..n − 1] whose value is
//            equal to K or −1 if no such element is found
A[n] ← K
i ← 0
while A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Analysis:

Cworst(n) = n +1.
Cavg(n) = (2−p)(n+1) /2

$$\begin{aligned} C_{avg}(n) &= [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \ldots + i \cdot \frac{p}{n} + \ldots + n \cdot \frac{p}{n}] + (n+1) \cdot (1-p) \\ &= \frac{p}{n}[1 + 2 + \ldots + i + \ldots + n] + (n+1)(1-p) \\ &= \frac{p}{n}\frac{n(n+1)}{2} + (n+1)(1-p) = \frac{(2-p)(n+1)}{2}. \end{aligned}$$

Or

(b) (i) Derive the worst case analysis of Merge Sort using suitable illustrations. (8)

(ii) Derive a loose bound on the following equation:

$f(x) = 35x^8 - 22x^7 + 14x^5 - 2x^4 - 4x^2 + x - 15$. (8)

Answer: 11(b)(i) Merge Sort Algoithm 4 Marks, Analysis 4 Marks]

ALGORITHM *Mergesort*($A[0..n-1]$)

```
//Sorts array A[0..n − 1] by recursive mergesort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
if n > 1
    copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
    copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
    Mergesort(B[0..⌊n/2⌋ − 1])
    Mergesort(C[0..⌈n/2⌉ − 1])
    Merge(B, C, A)
```

ALGORITHM *Merge*($B[0..p-1]$, $C[0..q-1]$, $A[0..p+q-1]$)

```
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B and C
i ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i + 1
    else A[k] ← C[j]; j ← j + 1
    k ← k + 1
if i = p
    copy C[j..q − 1] to A[k..p + q − 1]
else copy B[i..p − 1] to A[k..p + q − 1]
```

Setting recurrence equation we get:

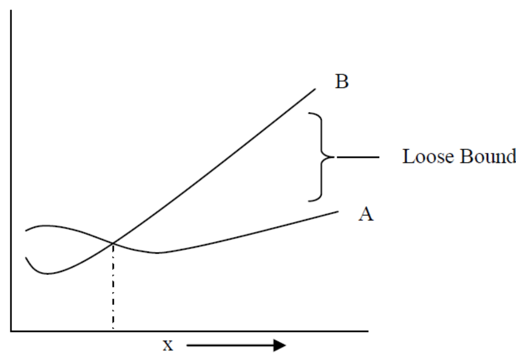
$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad C(1) = 0.$$

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 0.$$

Solving by masters theorem we get

$$C_{worst}(n) \in \Theta(n \log n)$$

11(b)(ii) A bound (lower bound or upper bound) is said to be loose bound if the inequality is strictly less.



Theorem 1.2 If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$.

Proof:

$$\begin{aligned} f(n) &\leq \sum_{i=0}^{m} |a_i| n^i \\ &\leq n^m \sum_{i=0}^{m} |a_i| n^{i-m} \\ &\leq n^m \sum_{i=0}^{m} |a_i| \qquad \text{for } n \geq 1 \end{aligned}$$

So, $f(n) = O(n^m)$ (assuming that m is fixed).

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

$$f(x) = 35x^8 - 22x^7 + 14x^5 - 2x^4 - 4x^2 + x - 15$$

$$|f(x)| = |35x^8 - 22x^7 + 14x^5 - 2x^4 - 4x^2 + x - 15|$$

$$|f(x)| < x^8(35 - 22/x + 14/x^3 - 2/x^4 - 4/f(x^6 + 1/x^7 - 15/x^8)$$

$$|f(x)| < x^8(|35| + |-22| + |14| + |-2| + |-4| + |1| + |-15|)$$

$$f(x) < 93x^8 \in o(x^9)$$

12. (a) (i) Solve the following using Brute-Force algorithm: (10)

Find whether the given string follows the specified pattern and return 0 or 1 accordingly.

Examples:

(1) Pattern: “abba”, input: “redblueredblue’ should return 1

(2) Pattern: “aaaa”, input: “asdasdasdasd’ should return 1

(3) Pattern: “aabb”, input: “xyzabcxzyabc” should return 0

(ii) Explain the convex hull problem and the solution involved behind it. (6)

Answer:

12(a)(i) Brute force algorithm: [analysis 5 marks, algorithm 5 marks]

pattern is abba [2 a's and 2 b's] and string = redbluebluered [14 chars]

Let number of chars in 'a' = x and 'b' = y

$3x + 4y = 14$ find all possibilities of x and y,

here it came : $x = 2$ and $y = 2$

Loop over all possibilities of x and y

Check in one more loop if string is following that pattern or not.

The approach is:

Example: pattern = [a b a b], given string = redblueredblue (14 characters in total)

|a| (length of a) = 1, then that makes 2 characters for as and 12 characters is left for bs, i.e. |b| = 6. Divided string = r edblue r edblue. Whoa, this matches right away!

(just out of curiosity) |a| = 2, |b| = 5 -> divided string = re dblue re dblue -> match

Example 2: pattern = [a b a b], string = redbluebluered (14 characters in total)

|a| = 1, |b| = 6 -> divided string = r edblue b luered -> no match

|a| = 2, |b| = 5 -> divided string = re dblue bl uered -> no match

|a| = 3, |b| = 4 -> divided string = red blue blu ered -> no match

Like this, it should be trial and error for |a|, |b| length.

Algorithm: function brute_force(text[], pattern[])

```
{
  // let n be the size of the text and m the size of the
  // pattern

  for(i = 0; i < n; i++) {
    for(j = 0; j < m && i + j < n; j++)
      if(text[i + j] != pattern[j]) break;
      // mismatch found, break the inner loop
    if(j == m) // match found
  }
}
```

12(a)(ii) Convex Hull problem: [Algorithm 5 Marks Analysis 3 Marks]

DEFINITION A set of points (finite or infinite) in the plane is called convex if for any two points p and q in the set, the entire line segment with the endpoints at p and q belongs to the set.

The convex hull of a set S of points is the smallest convex set containing S. (The “smallest” requirement means that the convex hull of S must be a subset of any convex set containing S.). The convex hull of any set S of $n>2$ points not all on the same line is a convex polygon with the vertices at some of the points of S. (If all the points do lie on the same line, the polygon degenerates to a line segment but still with the endpoints at two points of S.)

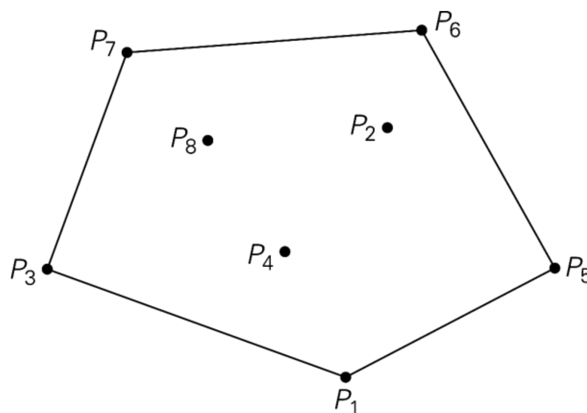


FIGURE 3.6 The convex hull for this set of eight points is the convex polygon with vertices at P_1, P_5, P_6, P_7, and P_3.

The convex-hull problem is the problem of constructing the convex hull for a given set S of n points. To solve it, we need to find the points that will serve as the vertices of the polygon in question. Mathematicians call the vertices of such a polygon "extreme points." By definition, an extreme point of a convex set is a point of this set that is not a middle point of any line segment with endpoints in the set. For example, the extreme points of a triangle are its three vertices, the extreme points of a circle are all the points of its circumference, and the extreme points of the convex hull of the set of eight points.

Analysis:

Finding point farthest away from line P_1P_2 can be done in linear time

Time efficiency:

worst case: $\Theta(n^2)$ (as quicksort)

average case: $\Theta(n)$ (under reasonable assumptions about distribution of points given)

Or

(b) A pair contains two numbers, and its second number is on the right side of the first one in an array. The difference of a pair is the minus result while subtracting the second number from the first one. Implement a function which gets the maximal difference of all pairs in an array (using Divide and Conquer method). (16)

Answer: [Explanation 6 Marks, Implementation 6 Marks, Example and Analysis 4 Marks]

A pair contains two numbers, and its second number is on the right side of the first one in an array. The difference of a pair is the minus result while subtracting the second number from the first one. Please implement a function which gets the maximal difference of all pairs in an array. For example, the maximal difference in the array {2, 4, 1, 16, 7, 5, 11, 9} is 11, which is the minus result of pair (16, 5).

We divide an array into two sub-arrays with same size. The maximal difference of all pairs occurs in one of the three following situations:

(1) two numbers of a pair are both in the first sub-array;

(2) two numbers of a pair are both in the second sub-array;

(3) the minuend is in the greatest number in the first sub-array, and the subtrahend is the least number in the second sub-array.

we get the maximal difference of pairs in the first sub-array (leftDiff), and then get the maximal difference of pairs in the second sub-array (rightDiff). We continue to calculate the difference between the maximum in the first sub-array and the minimal number in the second sub-array (crossDiff). The greatest value of the three differences is the maximal difference of the whole array.

Analysis:

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

solving we get

time complexity is O(n)

13. (a) (i) Given the mobile numeric keypad. You can only press buttons that are up, left, right or down to the first number pressed to obtain the subsequent numbers. You are not allowed to press bottom row corner buttons (i.e. ∗ and #). Given a number N, how many key strokes will be involved to press the given number. What is the length of it? Which dynamic programming technique could be used to find solution for this? Explain each step with the help of a pseudo code and derive its time complexity. (12)

(ii) How do you construct a minimum spanning tree using Kruskal's algorithm? Explain. (4)

Answer: 13(a)(i): [Analysis of the problem+ recurrence equation: 8 Marks, Complexity Analysis: 4 Marks]

For N=1, number of possible numbers would be 10 (0, 1, 2, 3, …., 9)
For N=2, number of possible numbers would be 36
Possible numbers: 00,08 11,12,14 22,21,23,25 and so on.
If we start with 0, valid numbers will be 00, 08 (count: 2)
If we start with 1, valid numbers will be 11, 12, 14 (count: 3)
If we start with 2, valid numbers will be 22, 21, 23,25 (count: 4)
If we start with 3, valid numbers will be 33, 32, 36 (count: 3)
If we start with 4, valid numbers will be 44,41,45,47 (count: 4)

If we start with 5, valid numbers will be 55,54,52,56,58 (count: 5)
If we start with 6, valid numbers will be 66, 63, 65,69 (count: 4)
If we start with 7, valid numbers will be 77,74,78 (count: 3)
If we start with 8, valid numbers will be 88,85,80,87,89 (count: 5)
If we start with 9, valid numbers will be 99,96,98 (count: 3)

Mobile Keypad is a rectangular grid of 4X3 (4 rows and 3 columns)

Lets say Count(i, j, N) represents the count of N length numbers starting from position (i, j)

If N = 1

Count(i, j, N) = 10

Else

Count(i, j, N) = Sum of all Count(r, c, N-1) where (r, c) is new position after valid move of length 1 from current position (i, j).

Travelsal for N=4 from key 8

Few traversals starting for button 8 for N = 4

e.g. 8 -> 7 -> 4 -> 1, 8 -> 9 -> 6 -> 3

8 -> 5 -> 4 -> 1, 8 -> 5 -> 6 -> 3

8 -> 5 -> 2 -> 2, 8 -> 5 -> 2 -> 3

Complexity Analysis: O(n).

13(a)(ii) Kruskal's algorithm [4 Marks]

ALGORITHM *Kruskal(G)*

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \leq \ldots \leq w(e_{i_{|E|}})$
$E_T \leftarrow \emptyset$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
    $k \leftarrow k + 1$
    if $E_T \cup \{e_{i_k}\}$ is acyclic
        $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $ecounter \leftarrow ecounter + 1$
return E_T

Procedure:

1. First we examine the edges of G in order of increasing weight.
2. Then we select an edge (u, v)∈ E of minimum weight and checks whether its end points belongs to same component or different connected components.
3. If u and v belongs to different connected components then we add it to set A,

otherwise it is rejected because it create a cycle.

4. The algorithm stops, when only one connected components remains (i.e. all the vertices of G have been reached).

Or

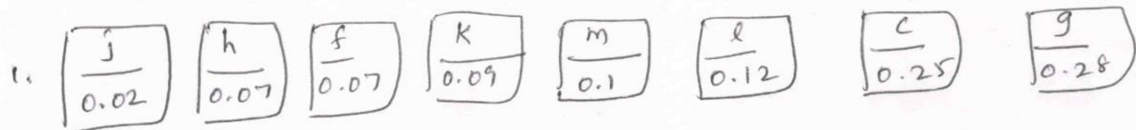
(b) (i) Let $A = \{l/119, m/96, c/247, g/283, h/72, f/77, k/92, j/19\}$ be the letters and its frequency of distribution in a text file. Compute a suitable Huffman coding to compress the data effectively. (8)

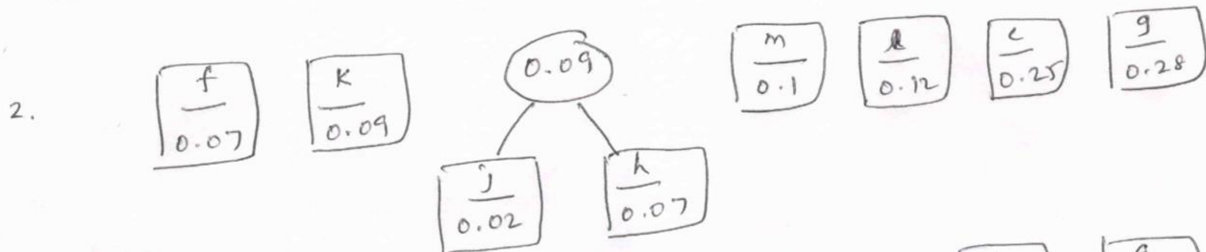
(ii) Write an algorithm to construct the optimal binary search tree given the roots $r(i, j)$, $0 \le i \le j \le n$. Also prove that this could be performed in time $O(n)$. (8)

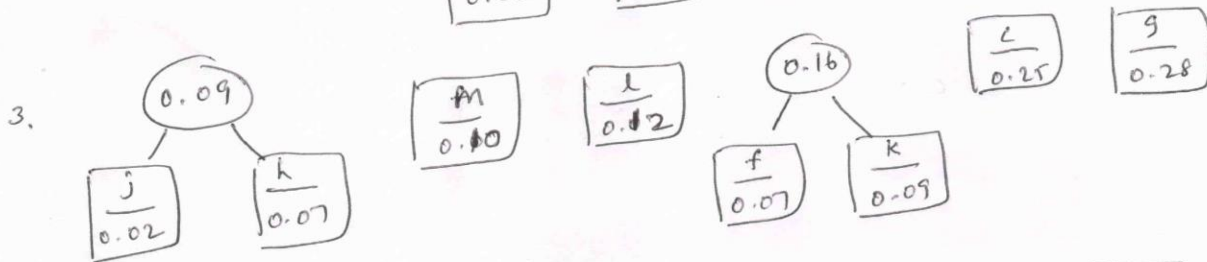
Answer: (b)(i)Total number of letters: 1005

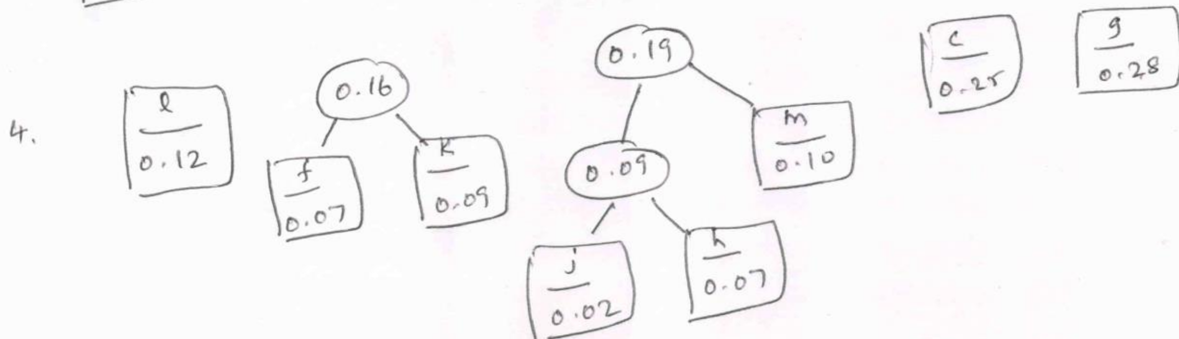
Frequency is:

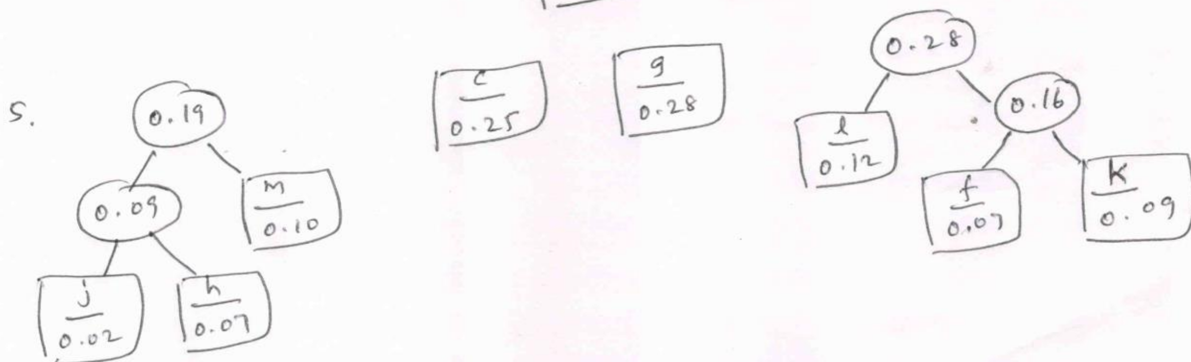
L	m	c	g	h	f	k	j
0.2	0.1	0.25	0.28	0.07	0.07	0.09	0.02
110	001	01	10	0001	1110	1111	0000

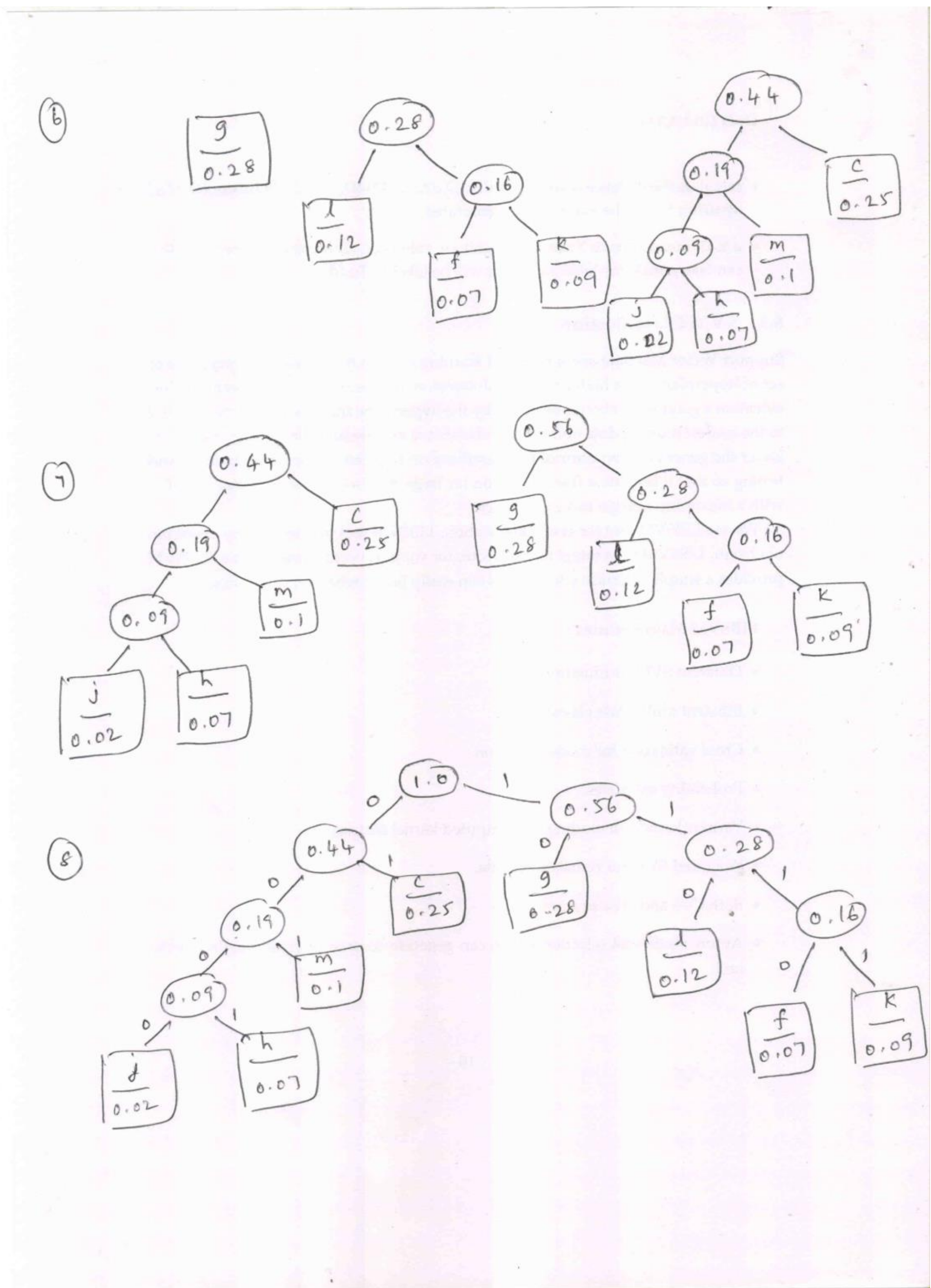

1.
j
0.02
h
0.07
f
0.07
K
0.09
m
0.1
l
0.12
c
0.25
g
0.28


2.
f
0.07
K
0.09
0.09
j
0.02
h
0.07
m
0.1
l
0.12
c
0.25
g
0.28


3.
0.09
j
0.02
h
0.07
m
0.10
l
0.12
0.16
f
0.07
k
0.09
c
0.25
g
0.28


4.
l
0.12
0.16
f
0.07
k
0.09
0.19
0.09
j
0.02
h
0.07
m
0.10
c
0.25
g
0.28


5.
0.19
0.09
m
0.10
j
0.02
h
0.07
c
0.25
g
0.28
0.28
l
0.12
0.16
f
0.07
K
0.09



(b)(ii) root table [4 marks] + [Algorithm 4 Marks]

ALGORITHM *OptimalBST*$(P[1..n])$

```
//Finds an optimal binary search tree by dynamic programming
//Input: An array P[1..n] of search probabilities for a sorted list of n keys
//Output: Average number of comparisons in successful searches in the
//        optimal BST and table R of subtrees' roots in the optimal BST
for i ← 1 to n do
    C[i, i − 1] ← 0
    C[i, i] ← P[i]
    R[i, i] ← i
C[n + 1, n] ← 0
for d ← 1 to n − 1 do //diagonal count
    for i ← 1 to n − d do
        j ← i + d
        minval ← ∞
        for k ← i to j do
            if C[i, k − 1] + C[k + 1, j] < minval
                minval ← C[i, k − 1] + C[k + 1, j]; kmin ← k
        R[i, j] ← kmin
        sum ← P[i]; for s ← i + 1 to j do sum ← sum + P[s]
        C[i, j] ← minval + sum
return C[1, n], R
```

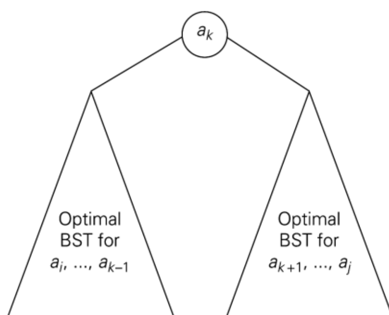


FIGURE 8.9 Binary search tree (BST) with root a_k and two optimal binary search subtrees T_i^{k-1} and T_{k+1}^{j}

root table

	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

14. (a) (i) Maximize $p = 2x + 3y + z$ (8)

subject to $x + y + z \le 40$

$2x + y - z \ge 10$

$-y + z \ge 10$

$x \ge 0,\ y \ge 0,\ z \ge 0.$

(ii) Write down the optimality condition and algorithmic implementation for finding M-augmenting paths in bipartite graphs. (8)

Answer:

14(a)(i) The first constraint is

$x + y + z \le 40.$

To turn it into an equation, we must add a slack variable s to the left-hand side, getting

$x + y + z + s = 40.$

The next constraint is $2x + y - z \ge 10$,

and we must subtract the surplus variable t to the left-hand side, getting

$2x + y - z - t = 10.$

The last constraint is $-y + z \ge 10$,

and we must subtract the surplus variable u to the left-hand side, getting

$-y + z - u = 10.$

Finally, the objective is

$p = 2x + 3y + z.$

We must subtract $2x + 3y + z$ from both sides to get the desired equation:

$-2x - 3y - z + p = 0.$

Tableau #1

x	y	z	s	t	u	p	
1	1	1	1	0	0	0	40
2	1	-1	0	-1	0	0	10
0	-1	1	0	0	-1	0	10
-2	-3	-1	0	0	0	1	0

Tableau #2

x	y	z	s	t	u	p	
0	0.5	1.5	1	0.5	0	0	35
1	0.5	-0.5	0	-0.5	0	0	5
0	-1	1	0	0	-1	0	10
0	-2	-2	0	-1	0	1	10

Tableau #3

x	y	z	s	t	u	p	
0	2	0	1	0.5	1.5	0	20
1	0	0	0	-0.5	-0.5	0	10
0	-1	1	0	0	-1	0	10
0	-4	0	0	-1	-2	1	30

Tableau #4

x	y	z	s	t	u	p	
0	1	0	0.5	0.25	0.75	0	10
1	0	0	0	-0.5	-0.5	0	10
0	0	1	0.5	0.25	-0.25	0	20
0	0	0	2	0	1	1	70

Optimal Solution: p = 70; x = 10, y = 10, z = 20

14(a)(ii) [Condition Explanation 4 Marks and Algorithm 4 Marks]

A maximum matching—more precisely, a maximum cardinality matching—is a matching with the largest number of edges. We limit our discussion in this section to the simpler case of bipartite graphs. In a bipartite graph, all the vertices can be partitioned into two disjoint sets V and U, not necessarily of the same size, so that every edge connects a vertex in one of these sets to a vertex in the other set. In other words, a graph is bipartite if its vertices can be colored in two colors so that every edge has its vertices colored in different colors; such graphs are also said to be 2-colorable. A matching M is a maximum matching if and only if there exists no augmenting path with respect to M

Given a matching M, an M-alternating path is a path that alternates between edges in M and edges not in M. An M-alternating path P that begins and ends at vertices not covered by M is an M-augmenting path;

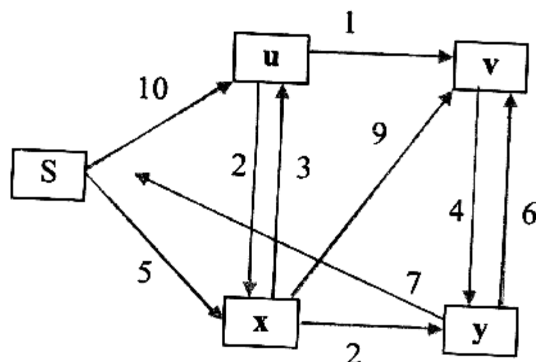
```
ALGORITHM  MaximumBipartiteMatching(G)
    //Finds a maximum matching in a bipartite graph by a BFS-like traversal
    //Input: A bipartite graph G = ⟨V, U, E⟩
    //Output: A maximum-cardinality matching M in the input graph
    initialize set M of edges with some valid matching (e.g., the empty set)
    initialize queue Q with all the free vertices in V (in any order)
    while not Empty(Q) do
        w ← Front(Q); Dequeue(Q)
        if w ∈ V
            for every vertex u adjacent to w do
                if u is free
                    //augment
                    M ← M ∪ (w, u)
                    v ← w

                    while v is labeled do
                        u ← vertex indicated by v's label; M ← M − (v, u)
                        v ← vertex indicated by u's label; M ← M ∪ (v, u)
                    remove all vertex labels
                    reinitialize Q with all free vertices in V
                    break //exit the for loop
                else //u is matched
                    if (w, u) ∉ M and u is unlabeled
                        label u with w
                        Enqueue(Q, u)
        else //w ∈ U (and matched)
            label the mate v of w with "w"
            Enqueue(Q, v)
    return M //current matching is maximum
```

Or

(b) (i) Briefly describe on the Stable marriage problem. (6)

(ii) How do you compute maximum flow for the following graph using Ford-Fulkerson method? (10)



Answer: (b)(i) [Algorithm and discussion 6 Marks]

Stable marriage algorithm

Input: A set of n men and a set of n women along with rankings of the women by each man and rankings of the men by each woman with no ties allowed in the rankings

Output: A stable marriage matching

1. Start with all the men and women being free.
2. While there are free men, arbitrarily select one of them and do the following:

   Proposal The selected free man m proposes to w, the next woman on his preference list (who is the highest-ranked woman who has not rejected him before).

   Response If w is free, she accepts the proposal to be matched with m. If she is not free, she compares m with her current mate. If she prefers m to him, she accepts m's proposal, making her former mate free; otherwise, she simply rejects m's proposal, leaving m free.

3. Return the set of n matched pairs.

14(b)(ii) Ford Fulkerson Method: Algorithm with an example [6 + 4 Marks]

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ G.E
2      (u, v).f = 0
3  while there exists a path p from s to t in the residual network G_f
4      c_f(p) = min {c_f(u, v) : (u, v) is in p}
5      for each edge (u, v) in p
6          if (u, v) ∈ E
7              (u, v).f = (u, v).f + c_f(p)
8          else (v, u).f = (v, u).f − c_f(p)
```

15. (a) (i) Suggest an approximation algorithm for traveling salesperson problem. Assume that the cost function satisfies the triangle inequality. (8)

(ii) Explain how job assignment problem could be solved, given n tasks and n agents where each agent has a cost to complete each task, using Branch and Bound technique. (8)

Answer:

15(a)(i) Greedy Algorithms for the TSP: [Algorithm with example 6 marks, inequality 2 marks]

The simplest approximation algorithms for the traveling salesman problem are based on the greedy technique.

Nearest-neighbor algorithm [always go next to the nearest unvisited city]

Step 1 Choose an arbitrary city as the start.

Step 2 Repeat the following operation until all the cities have been visited:

go to the unvisited city nearest the one visited last (ties can be broken

arbitrarily).

Step 3 Return to the starting city.

- *triangle inequality* $d[i, j] \leq d[i, k] + d[k, j]$ for any triple of cities i, j, and k (the distance between cities i and j cannot exceed the length of a two-leg path from i to some intermediate city k to j)
- *symmetry* $d[i, j] = d[j, i]$ for any pair of cities i and j (the distance from i to j is the same as the distance from j to i)

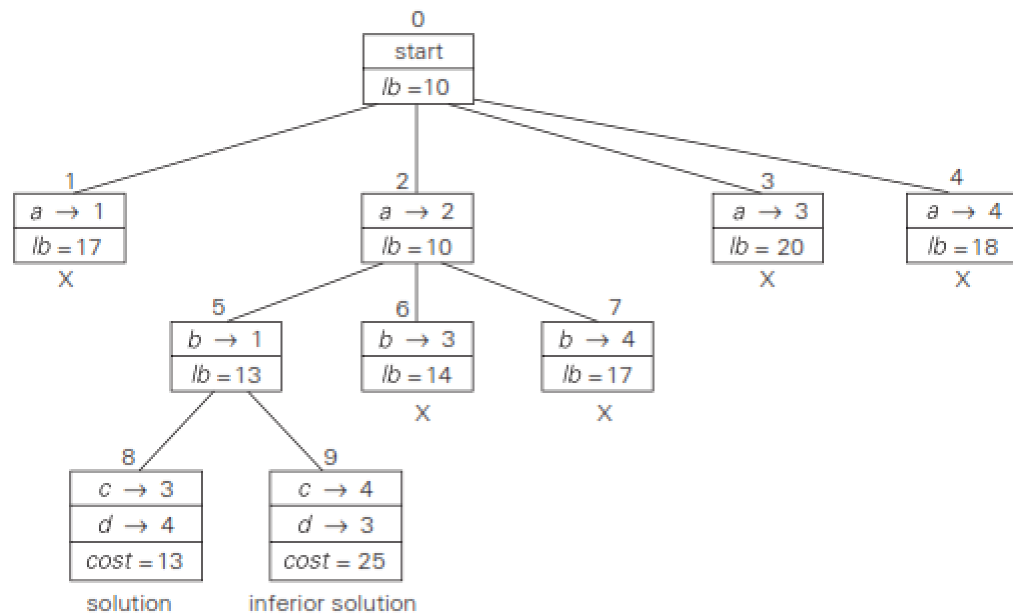
15(a)(ii) [Explanation 4 Marks, Example 4 Marks]

Let us illustrate the branch-and-bound approach by applying it to the problem of assigning n people to n jobs so that the total cost of the assignment is as small as possible. Compare the lower bounds of the live nodes. It is sensible to consider a node with the best bound as most promising. Best first Branch and Bound.

Example

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \begin{matrix} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{matrix}$$

(columns: job 1, job 2, job 3, job 4)



Or

(b) (i) The knight is placed on the first block of an empty board and, moving according to the rules of chess, must visit each square exactly once. Solve the above problem using backtracking procedure. (10)

(ii) Implement an algorithm for Knapsack problem using NP-Hard approach. (6)

Answer:

15(b)(i) Knight's Tour Algorithm: [Explanation 4 Marks, Solution 4 Marks]

we start from an empty solution vector and one by one add items Knight's move. When we add an item, we check if adding the current item violates the problem constraint, if it does then we remove the item and try other alternatives. If none of the alternatives work out then we go to previous stage and remove the item added in the previous stage

Algorithm:

If all squares are visited

print the solution

Else

a) Add one of the next moves to solution vector and recursively

check if this move leads to a solution. (A Knight can make maximum

eight moves. We choose one of the 8 moves in this step).

b) If the move chosen in the above step doesn't lead to a solution

then remove this move from the solution vector and try other

alternative moves.

c) If none of the alternatives work then return false (Returning false will remove the previously added item in recursion and if false is returned by the initial call of recursion then "no solution exists")

Knight's tour in a Chess Board:

0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

(b)(ii) Greedy Algorithms for the Knapsack Problem [Algorithm 5 Marks, Example 3 marks]

One is to select the items in decreasing order of their weights; however, heavier items may not be the most valuable in the set. Alternatively, if we pick up the items in decreasing order of their value, there is no guarantee that the knapsack's capacity will be used efficiently. We can, by computing the value-to-weight ratios v_i/w_i, where ,i= 1, 2,...,n,and selecting the items in decreasing order of these ratios. (

Step 1 Compute the value-to-weight ratios $r_i= v_i/w_i$, where i=1,...,n, for the items given.

Step 2 Sort the items in non increasing order of the ratios computed in Step 1.

Step 3 Repeat the following operation until no item is left in the sorted list:

if the current item on the list fits into the knapsack, place it in the knapsack and proceed to the next item; otherwise, just proceed to the next item.