SPRING END SEMESTER EXAMINATION-2019

4th Semester B.Tech & B.Tech Dual Degree

Design & Analysis of Algorithms
CS-2008

[For 2018 (LE), 2017 & Previous Admitted Batches]

Time: 3 Hours **Full Marks: 50**

Answer any SIX questions.
Question paper consists of four sections-A, B, C, D.
Section A is compulsory.
Attempt minimum one question each from Sections B, C, D.
The figures in the margin indicate full marks.
Candidates are required to give their answers in their own words as far as practicable and all parts of a question should be answered at one place only.

DAA SAMPLE ANSWER(S) & EVALUATION SCHEME

Q1 Answer the following questions: [1 x 10]

(a) Rank the following functions by order of their growth in increasing sequence? log n, log √n, √n, n log √n, n!, 2n

Evaluation Scheme:

- Correct Answer : 1 Mark
- Wrong Answer : Zero

Answer:

log √n, log n, √n, 2n, n log √n, n!

(b) Find out the complexity of the given function $\sum_{k=1}^{n} \log k$

Evaluation Scheme:

- Correct Answer : 1 Mark
- Wrong Answer : Zero

Answer:

O(log n!) or O(n log n)

(c) How many elements will not participate at 5th level of partitioning of an n-length array in Quick-Sort?

A. n/4 B. $2^4 – 1$ C. 2^5 -1 D. $2^4 + 1$

Evaluation Scheme:

- Correct Answer : 1 Mark
- Wrong Answer : Zero

Answer:

B or C

(d) What is the worst case time complexity of Insertion-Sort where position of the data to be inserted into the sorted array is calculated using Linear-Search?

(A) n **(B) nlogn** **(C) n^2** **(D) n^2logn**

Evaluation Scheme:

- Correct Answer : 1 Mark
- Wrong Answer : Zero

Answer:

C

(e) **Let s be a sorted array of n integers. Let t(n) denote the time taken for the most efficient algorithm to determine if there are two elements with sum less than 1000 in s. which of the following statements is true?**

a) t (n) is O(1) **b) t (n) is O(nlogn)** **c) t (n) is O(n)** **d) t (n) is O(n^2)**

Evaluation Scheme:

- Correct Answer : 1 Mark
- Wrong Answer : Zero

Answer:

a

(f) **Let $Z = \langle z_1 . . . z_k \rangle$ be any Longest-Common-Subsequence of the sequences X and Y. Which of the following is (are) true?**

A. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z is an LCS of X_m and Y_n.

B. If $x_m \neq y_n$, then $z_k \neq x_m$ and Z is an LCS of $X_{m\text{-}1}$ and Y.

C. If $x_m \neq y_n$, then $z_k \neq y_n$ and Z is an LCS of $X_{m\text{-}1}$ and Y.

D. If $x_m \neq y_n$, then $z_k \neq x_m$ and Z is an LCS of X and $Y_{n\text{-}1}$.

Evaluation Scheme:

- Inadequate information
- Grace Mark: 1 Mark

Answer:

NA

(g) **Differentiate between Divide-and-Conquer and Dynamic-programming approach.**

Evaluation Scheme:

- Correct Answer : 1 Mark
- Wrong Answer : Zero

Answer:

Divide-And-Conquer (D-n-C)	**Dynamic Programming (DP)**
1. break problems into simpler & **independent sub problems.**	1. break problems into simpler & **dependent sub-problems** that is sub-problems share sub sub-problems.
2. **splits** its input at prespecified deterministic points (e.g., **always in the middle**)	2. **splits** its input at **every possible split points** rather than at a pre-specified points. After trying all split points, it determines which split point is optimal.
3. solves an sub-problem independently, in turn create a	3. solves every sub sub-problem just once and then **saves its**

large number of identical sub-problems during any given computation because of duplicating certain pieces of computation / sub-problem resulting in an inefficient algorithm. This property called **over lapping sub-problems**. Due to over lapping sub-problem, it does more work than necessary, repeatedly solving the common sub sub-problems. 4. It is a **top-down technique/method** which logically progresses from the initial instance down to the smallest sub-instances via intermediate sub-instances.	**answer in a table**, thereby avoiding the work of re computing the answer every time the sub sub-problems is encountered. 4. It is a **bottom up technique** in which the smallest sub instances are explicitly solved first and the results of these used to construct solution to progressively larger sub-instances.
Example Compute n th Fibonacci number F(n) using the recurrence relation. F(n)=F (n-1)+ F (n-2) with F(0)=0, F(1)=1 **Solution** **D-n-C Method of Solving** F (n) =F(n-1)+F(n-2) Calculate F(2)=F(1)+F(0)=0+1=1 F(3)=F(2)+F(1) =F(1)+F(0)+F(1) To compute F(3),F(1)is overlapped. Calculate F(4)=F(3)+F(2) =F(2)+F(1)+F(1)+F(0) =F(1)+F(0)+F(1)+F(1)+F(0) Mark in the process of computing F(4)F(1)is duplicated thrice F(0)is twice	**DP Method of Solving** F(2)=F(1)+F(0)=1+0=1 F(3)=F(2)+F(1)=1+1=2 In calculating F(3), F(2) is not called recursively because the preventing stored answer of F(2) we can use directly.

(h) Match the following Algorithms and its Complexities.

(A) O (logn) **(a) Finding max-min in an array**

(B) O (n) **(b) Heap-sort**

(C) O (nlogn) **(c) Binary search**

(D) O (n^2) **(d) Insertion sort**

Evaluation Scheme: -

- Correct Answer : 1 Mark
- Wrong Answer : Zero

Answer:

(A) - (c)

(B) - (a)

(C) - (b)
(D) - (d)

(i) Which of the following statement is true about adjacency-list representation?

i. Space complexity for both directed and undirected graphs is $O(V^2)$.

ii. Space complexity for directed graph is O(V) and Space complexity for undirected graphs O(E)

iii. Space complexity for directed graph is $O(V^2)$ and Space complexity for undirected graphs O(V+E)

iv. Space complexity for directed graph is O(V+E) and Space complexity for undirected graphs $O(V^2)$

v. Space complexity for both directed graph and undirected graphs is O(V+E)

<u>Evaluation Scheme:</u>

- Correct Answer : 1 Mark
- Wrong Answer : Zero

<u>Answer:</u>

v

(j) Which of the following is an NP-Complete Problem?

A. 2CNF B. Euler tour C. Hamiltonian Cycle D. Shortest Path

<u>Evaluation Scheme:</u>

- Correct Answer : 1 Mark
- Wrong Answer : Zero

<u>Answer:</u>

C

Q2 (a) What is the significance of asymptotic notations? Define different asymptotic notations used in algorithm analysis. [4]

<u>Evaluation Scheme:</u>

- Significance of asymptotic notations: 1.5 Marks
- Definition of asymptotic notations: 2.5 Marks

<u>Answer:</u>

- To analyze the efficiency of an algorithm, it is not necessary to conduct a detailed analysis of the running time. The asymptotic analysis is the method that efficiently describes the efficiency of an algorithm.
- The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared.
- Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.
    - The asymptotic run time of an algorithm gives a simple and machine independent, characterization of its complexity.

- The notations works well to compare algorithm efficiencies because we want to say that the growth of effort of a given algorithm approximates the shape of a standard function.

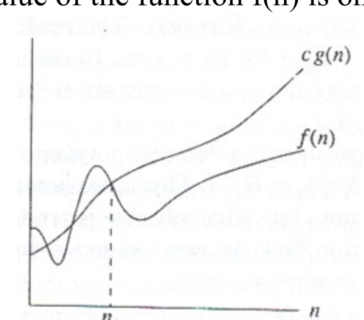
- The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.
  i) O - Notation (Big-Oh Notation) : Asymptotic upper bound
  ii) Ω - Notation (Big-Omega Notation) : Asymptotic lower bound
  iii) Θ - Notation (Theta Notation) : Asymptotic tight bound

i) O - Notation (Big-Oh Notation)

- It represents the upper bound of the resources required to solve a problem.(worst case running time)
- **Definition:** Formally it is defined as
  For any two functions f(n) and g(n), which are non-negative for all $n \geq 0$, f(n) is said to be g(n), $f(n) = O(g(n))$, if there exists two positive constants c and n_0 such that

$$0 \leq f(n) \leq c\, g(n) \text{ for all } n \geq n_0$$

- Less formally, this means that for all sufficiently big n, the running time of the algorithm is less than g(n) multiplied by some constant. For all values n to the right of n_0, the value of the function f(n) is on or below g(n).

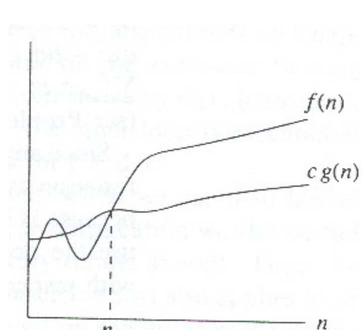


$f(n) = O(g(n))$

ii) Ω - Notation (Big-Omega Notation)

- **Definition:** For any two functions f(n) and g(n), which are non-negative for all $n \geq 0$, f(n) is said to be g(n), $f(n) = \Omega(g(n))$, if there exists two positive constants c and n0 such that

$$0 \leq cg(n) \leq f(n) \text{ for all } n \geq n0$$

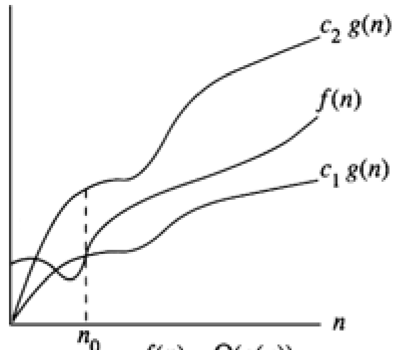


$f(n) = \Omega(g(n))$

iii) Θ - Notation (Theta Notation)

- **Definition:** For any two functions f(n) and g(n), which are non-negative for all $n \geq 0$, f(n) is said to be g(n), $f(n) = \Theta(g(n))$, if there exists positive constants c1, c2 and n0 such that

$$0 \leq c1g(n) \leq f(n) \leq c2g(n) \text{ for all } n \geq n0$$



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

(b) State and explain master's method, and use the method to give tight asymptotic bounds for the recurrence [4]

$T(n) = 4T(n/2) + n^3$.

Evaluation Scheme:

- State & Explain Master Theorem: 2 Marks
- Solving the recurrence: 2 Marks

Answer:

$T(n) = \Theta(n^3)$

Explanation

Type:-1 (Master Theorem as per CLRS)	
Master Theorem	**Solution of recurrence** $T(n) = 4T(n/2) + n^3$
The Master Theorem applies to recurrences of the following form: $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. $T(n)$ is defined on the non-negative integers by the recurrence. T(n) can be bounded asymptotically as follows: There are 3 cases: **a) Case-1:** If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $\mathbf{T(n) = \Theta(n^{\log_b a})}$ **b) Case- 2:** If $f(n) = \Theta(n^{\log_b a})$, then $\mathbf{T(n) = \Theta(n^{\log_b a} \log n)}$ **c) Case-3:** If $f(n) = \Omega(n^{\log_b a+\epsilon})$ with $\epsilon > 0$, and : $af(n/b) \leq cf(n)$, then $T(n) = \Theta(f(n))$, for some constant $c < 1$ and all sufficiently large n, then $\mathbf{T(n) = \Theta(f(n))}$	Given, a=4, b=2, $f(n)=n^3$ **Step-1 (Guess)** $n^{\log_b a}=n^{\log_2 4}=n^2$, Comparing $n^{\log_b a}$ with f(n), f(n) is found asymptotically larger than $n^{\log_b a}$. So case-3 of master theorem is guessed. **Step-2 (Verify)** As per case-3 of master theorem, Let $f(n)=\Omega(n^{\log_b a+\epsilon})$ is true $\Rightarrow f(n) \geq c.n^{\log_b a+\epsilon}$ $\Rightarrow n^3 \geq c.n^{2+\epsilon}$ $\Rightarrow n \geq c.n^{\epsilon}$, This inequality is valid for c=1 and $0<\epsilon \leq 1$. Now $af(n/b) \leq cf(n)$ must valid $\Rightarrow 4(n/2)^3 \leq c.n^3$ $\Rightarrow c \geq 0.5$ (True as c is a valid constant < 1) So the solution is

	$T(n) = \Theta(f(n) = \Theta(n^3)$
Type:-2 (Master Theorem)	
Master Theorem	**Solution of recurrence** $T(n) = 4T(n/2) + n^3$
If the recurrence is of the form $T(n) = aT(n/b) + n^k \log^p n$, where $a \geq 1$, $b > 1$, $k \geq 0$ and p is a real number, then comapre a with b^k and conclude the solution as per the following cases. **Case-1:** If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$ **Case-2:** If $a = b^k$, then a) If $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$ b) If $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$ c) If $p < -1$, then $T(n) = \Theta(n^{\log_b a})$ **Case-3:** If $a < b^k$, then a) If $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$ b) If $p < 0$, then $T(n) = \Theta(n^k)$	Here, a=4, b=2, k=3, p=0 $b^k = 2^3 = 8$ Comparing a with b^k, we found a is less than b^k., so this will fit to case-3. Now p=0, so case-3.a solution is the recurrence solution. **$T(n)= \Theta(n^3 \log^0 n)=\Theta(n^3)$**

Q3 (a) **Solve the recurrence $T(n) = \sum_{1=1}^{n} T(i) + 1$, for $n \geq 2$.** **[4]**

Evaluation Scheme:

- Inadequate information
- Grace Mark: 4 Marks

Answer:

NA

(b) **Define and differentiate between P, NP and NP-complete problems with examples.** **[4]**

Evaluation Scheme:

- Insertion Algorithm: 1.5 Marks
- Explanation through step count method : 2.5 Marks

Answer:

CLASSIFICATION OF PROBLEMS

The subject of computational complexity theory is dedicated to classifying problems by how hard they are. There are many classifications, some of the most common and useful are the following:

1. **The Class of P problems**
   - P stands for Polynomial.
   - The problems that are **solvable** in polynomial time (that is the time $O(n^k)$, for some constant k, where n is the size of the input to the problem), are called

class of **P problems**.

2. **The Class of NP problems**
    - **NP** stands for **Non-deterministic Polynomial.**
    - The problems that are **verifiable** in polynomial time, are called class of N**P problems**.
    - Here the term verifiable mean is that if we were somehow given a certificate of a solution, then we would verify the certificate is correct in polynomial time.
    - In other words, a problem is in NP if we can quickly (in polynomial time) test whether a solution is correct without worrying about how hard it might be to find the solution.
    - Problems in NP are still relatively easy, if only we could guess the right solution, we could then quickly test it.
    - Any problem in P is also in NP, since if a problem is in P then we can solve it in polynomial time without even being given a certificate. We believe P is a subset of NP.
    - Example: Determining whether a directed graph has a Hamiltonian cycle is NP-Complete, Traveling salesperson ($O(n^2 2^n)$), knapsack ($O(2^{n/2})$)

3. **The Class of NP-Complete & NP-Hard Problems**
    - A problem A can be **reduced** to another problem B if any instance of A can be rephrased as an instance of B, the solution to the instance of B provides a solution to the instance of A.
    - A decision problem C is NP-complete if:
        a) C is in NP
        b) 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.
    - Note that **a problem satisfying condition b)** is said to be **NP-hard**, whether or not it satisfies condition a)
    - A problem is said to be NP-hard if an algorithm for solving it can be translated into one for solving any other NP-problem. It is much easier to show that a problem is NP than to show that it is NP-hard. **A problem which is both NP and NP-hard is called an NP-complete problem.**
    - For example, the problem of solving linear equations in an indeterminate x reduces to the problem of solving quadratic equations. Given an instance $ax + b = 0$, we transform it to $0x^2 + ax + b = 0$, whose solution provides a solution to $ax + b = 0$.
    - NP-complete problems are the hardest problems in NP. Formally, problem P is NP-complete if it is in NP and every problem in NP reduces to P.
    - If an NP-complete problem were solvable in polynomial time, then every NP problem would be solvable in polynomial time.
    - $P \neq NP$ widely believed (but still unproven).
    - Example: Satisfiability, Travelling Salesman Problem, and Graph 3-Coloring. Finding the longest path between two vertices is N P-complete, even if the weight of each edge is 1

Q4 (a) Given 10 different jobs along with their start time (s_i) and finish time (f_i) as S_i = < 1, 2, 3, 4, 7, 8, 9, 9, 11, 12 > and f_i = < 3, 5, 4, 7, 10, 9, 11, 13, 12, 14 >. These jobs **[4]**

are to be scheduled on a single processor machine and all these jobs are associated with equal profit values. Write an efficient procedure to generate a schedule of these jobs on the machine to obtain maximum profit.

Evaluation Scheme:

- Greedy Activity Selector Algorithm: 4 Marks
- Solution to the problem with proper explanation: 0-2 marks (if algorithm is not written properly and score is<4)

Answer:

Arranging the activities in increasing order with their finishing time.

a_i	s_i	f_i	Selection (Yes/No)
a1	1	3	√
a3	3	4	√
a2	2	5	x
a4	4	7	√
a6	8	9	√
a5	7	10	x
a7	9	11	√
a9	11	12	√
a8	9	13	x
a10	12	14	√

Optimal Schedule: <a1, a3, a4, a6, a7, a9, a10>

(b) Write the algorithm for MAX-HEAPIFY(A, i). Consider the array A = {27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0} to be used for constructing a Heap and answer the following questions. **[4]**

i. Whether the Heap satisfies the Max-Heap property at each internal node.

ii. If it is not then fix the Max-Heap property at those positions where it is not satisfying Max-Heap property.

Evaluation Scheme:

- MAX-HEAPIFY(A, i) Algorithm: 2 Marks
- Correct answer whether the Heap satisfies Max-Heap properties: 0.5 Mark
- Application of MAX-HEAPIFY() procedure to the correct the internal node: 1.5 Marks

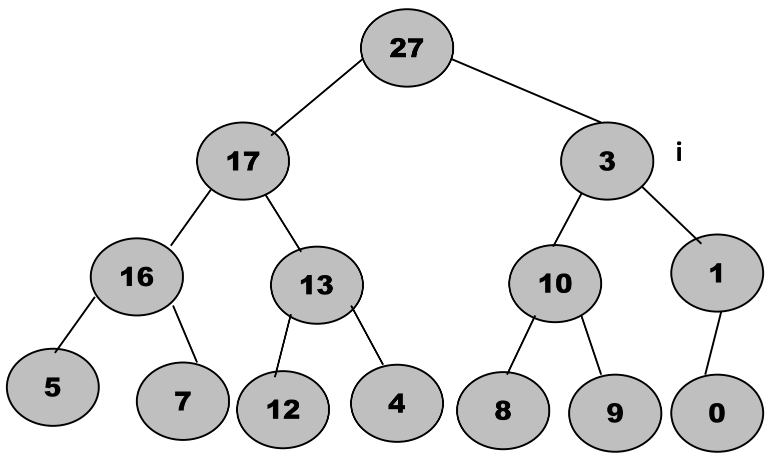
Answer:

/***Max-Heapify :** Given a tree that is a heap except for node i, Max-Heapify function arranges node i and it's subtrees to satisfy the heap property.*/

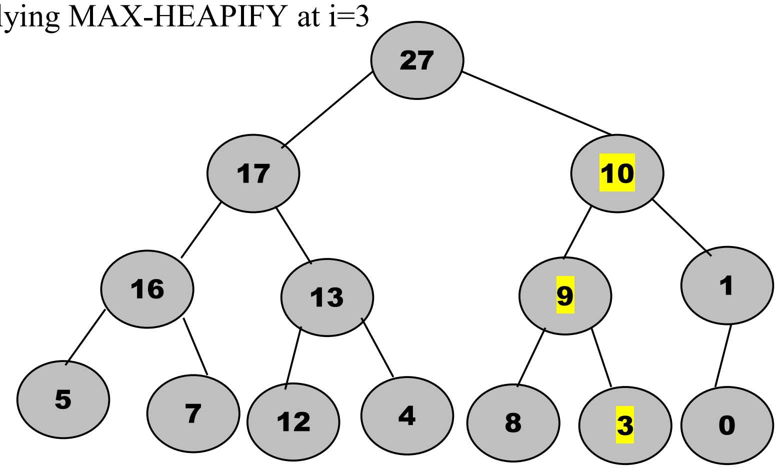
```
MAX-HEAPIFY(A, i)
{
  l ← LEFT(i);
  r ← RIGHT(i);
  if  (l ≤ n and A[l] > A[i])
      largest = l;
  else
```

```
        largest = i;
    if  (r ≤ n and A[r] > A[largest])
        largest = r;
    if (largest != i)
    {
       A[i] ↔ [largest]; // swaping
       MAX-HEAPIFY(A, largest);
    }
}
```

i. No, Heap does not satisfies the Max-heap property at each internal nodes.
ii. At i=3, Heap does not satisfies the Max-heap property. So MAX-HEAPIFY(A, 3) is applied.



Applying MAX-HEAPIFY at i=3



Q5 (a) Write the algorithms to find maximum and minimum of an array using (i) Straightforward approach and (ii) Divide-And-Conquer approach. Find out the number of basic operations performed by these approaches and make a comparison analysis. **[4]**

<u>Evaluation Scheme:</u>

- Correct Algorithm (s) : each 1.5 Marks
- Comparison: 1 Mark
- Some valid steps in algorithms/explanation: 1-3 Marks

Answer:

/* A[1..n] is a array of n elements. Parameters p and r are integers, represents lower bound and upper bound of the array/subarray $1 \le p \le r \le n$.

```
STRAIGHT-MAX-MIN(A, p, r, max, min)
{
    max := min := a[p];
    for i := p+1 to r
    {
        if(a[i] > max) then max := a[i];
        if(a[i] < min) then min := a[i];
    }
}
```

/* A[1..n] is a array of n elements. Parameters p and r are integers, represents lower bound and upper bound of the array/subarray $1 \le p \le r \le n$.

```
DIVIDE-MAX-MIN(A, p, r, max, min)
{
    if (p==r) then max := min := a[p]; // if array contains one element
    else if (p==r-1) then // if array contains two elements
    {
        if (a[p] < a[r]) then max := a[r]; min := a[p];
       else max := a[p]; min := a[r];
    }
    else // if array contains more than two elements
    {
        //Divide into two subproblems
         q ← (p + r )/2;
        // Solve the sub-problems.
        MAX-MIN( A, p, q, max, min );
        MAX-MIN( q+1,  r, max1, min1 );
        // Combine the solutions.
        if (max1 > max) then max := max1;
        if (min1 < min) then min := min1;
    }
}
```

Time Complexity of Divide-Conquer Max-Min

If T(n) represents the time complexity, then the resulting recurrence relation is

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

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 2^2T(n/2^2) + 2^2 + 2 \\ &= 2^2\{2T(n/2^3) + 2\} + 2^2 + 2 \\ &= 2^3T(n/2^3) + 2^3 + 2^2 + 2 \\ &\ldots\ldots\ldots\ldots \\ &\ldots\ldots\ldots\ldots \\ &= 2^{k-1}T(n/2^{k-1}) + (2^{k-1} + 2^{k-2} + 2^{k-3} + \ldots + 2^3 + 2^2 + 2^1) \end{aligned}$$

If $n=2^k$, then

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

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

$= 2^{k-1} + 2\sum_{i=0}^{k-2} 2^i$

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

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

$= 2^k/2 + 2^k - 2 = n/2 + n - 2 = 3n/2 - 2 = O(n)$

$3n/2 - 2$ is the best, average, worst case number of comparison when n is a power of two

Comparisons with Straight Forward Method

. The straight max-min algorithm requires 2n-2 element comparisons in the best, average, and worst cases. Compared with the 2n – 2 comparisons for the Straight Forward method, Divide & Conquer max-min algorithm is a saving of 25% in comparisons.

(b) State and explain the Longest Common Subsequence problem. Determine an LCS of the given two sequences < a, b, b, a, b, a, b, a > and <b, a, b, a, a, b, a, a, b>. **[4]**

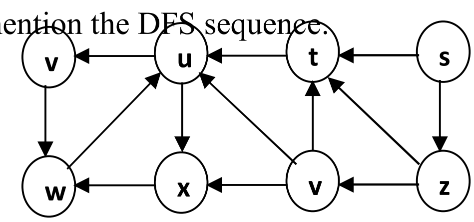
Answer:

Longest Common Subsequencce Problem

- Given two sequences $X=x_1x_2x_3\ldots x_m$ and $Y=y_1y_2y_3\ldots y_n$ and find a longest subsequence $Z=z_1z_2z_3\ldots z_k$ that is common to both subsequence X and Y.
- The LCS of <a, b, b, a, b, a, b, a> and <b, a, b, a, a, b, a, a, b> is **abbaab.**

	y_j	b	a	b	a	a	b	a	a	b
x_i	0	0	0	0	0	0	0	0	0	0
a	0	0 ↑	↖ 1	← 1	↖ 1	↖ 1	← 1	↖ 1	↖ 1	← 1
b	0	↖ 1	1 ↑	↖ 2	← 2	← 2	↖ 2	← 2	← 2	↖ 2
b	0	↖ 1	1 ↑	↖ 2	2 ↑	2 ↑	↖ 3	← 3	← 3	↖ 3
a	0	1 ↑	↖ 2	2 ↑	↖ 3	↖ 3	← 3	↖ 4	↖ 4	← 4
b	0	↖ 1	2 ↑	↖ 3	← 3	← 3	↖ 4	4 ↑	4 ↑	↖ 5
a	0	1 ↑	↖ 2	3 ↑	4	↖ 4	4 ↑	↖ 5	↖ 5	5 ↑
b	0	↖ 1	2 ↑	↖ 3	4 ↑	4 ↑	↖ 5	5 ↑	5 ↑	↖ 6
a	0	1 ↑	↖ 2	3 ↑	↖ 4	↖ 5	5 ↑	↖ 6	↖ 6	6 ↑

Q6 a) Traverse the following graph by DFS technique with 's' as source vertex. Draw the DFS tree/forest and mention the DFS sequence. **[4]**

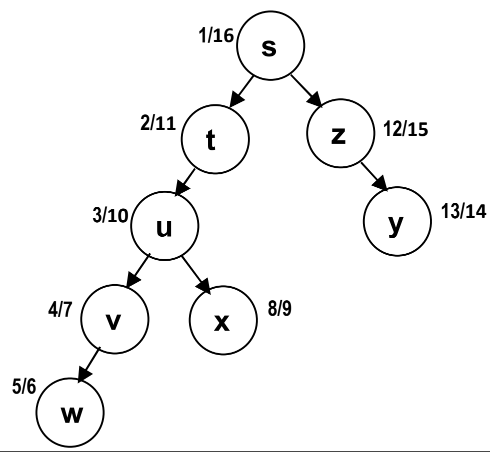
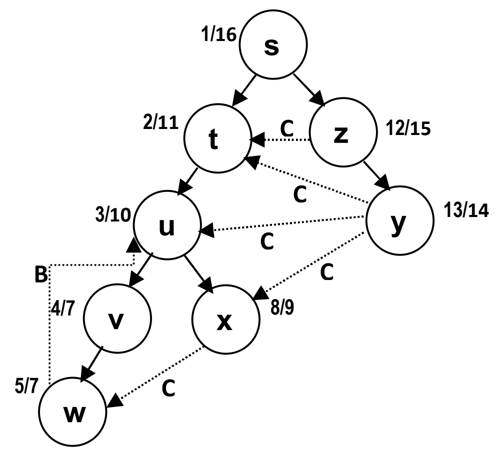


Evaluation Scheme:

- Construction of DFS tree/forest: 3 Marks
- Correct DFS sequence: 1 Mark

Answer:

- In this case the DFS tree/forest is not unique. The sample answer is given as follows:

<table>
<tr><th>DFS tree/forest without forward edge, back age, cross edge (only with tree edges)</th><th>DFS tree/forest with tree edge (solid line forward edge (labeled as F), back age (labeled as B), cross edge (labeled as C)</th></tr>
<tr><td>
</td><td>
</td></tr>
<tr><td>DFS Sequence for this DFS tree/forest is
s, t, u, v, w, x, z, y</td><td>DFS Sequence for this DFS tree/forest is
s, t, u, v, w, x, z, y</td></tr>
</table>

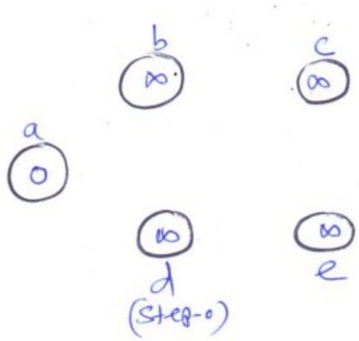
b) **Use suitable shortest path algorithm to find out shortest path between a to c and a to e.** [4]

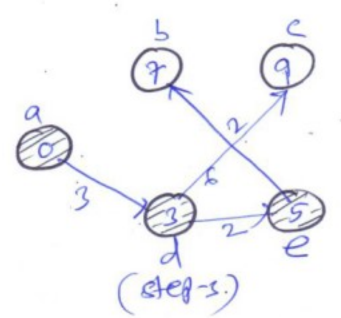
Evaluation Scheme:

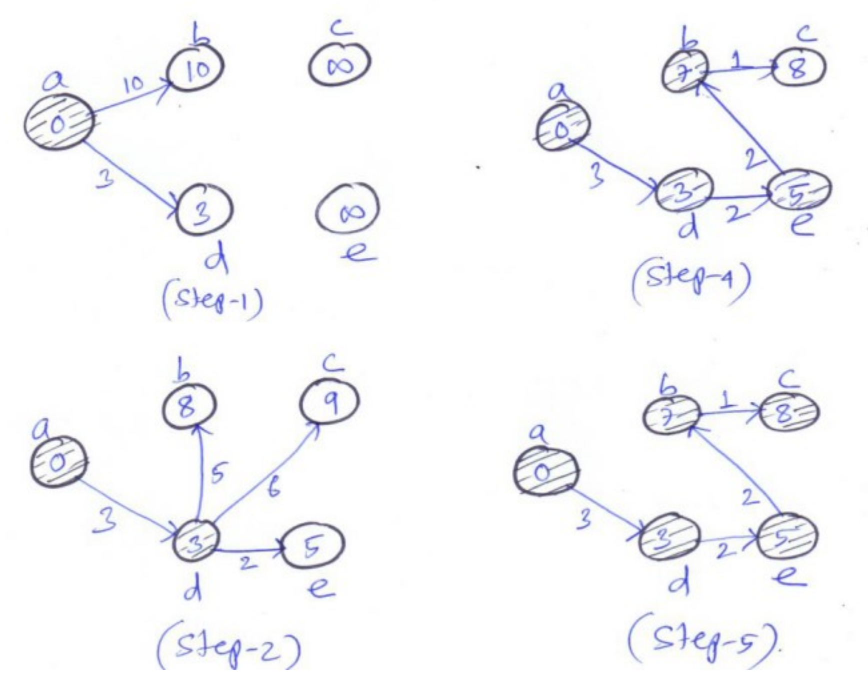
- Mentioning the name of shortest path algorithm: 1 Marks
- Explanation of algorithms through the given example: 3 Marks

Answer:

The name of shortest path algorithm: Dijkstra's algorithm.
The shortest path from a to c is 8 and a to e is 5.







Q7 (a) Write a Quick-Sort algorithm that randomly selects an element as pivot element and derive the average case time complexity of this algorithm. [4]

<u>Evaluation Scheme:</u>

- Randomized Quick-Sort Algorithm: 2 Marks
- Average case Time Complexity: 2 Marks

<u>Answer:</u>

<u>A randomized version of quicksort</u>

- The quicksort algorithm can be randomized explicitly permuting the input. But a different randomization technique, called random sampling, yields a simpler analysis.
- Instead of always using A[r] as the pivot, we will select a randomly chosen element from the subarray A[p..r]. We do so by first exchanging element A[r] with an element chosen at random from A[p..r].
- The following procedure implements quicksort that chooses pivot as a random element.

Randomized Quick Sort Algorithm

```
RANDOMIZED-QUICKSORT (A, p, r)
{
    if (p<r)
    {
        q←RANDOMIZED-PARTITION (A, p, r);
        RANDOMIZED-QUICKSORT (A, p, q-1);
        RANDOMIZED-QUICKSORT (A, q+1, r);
    }
}
```

Randomized Partition Algorithm

RANDOMIZED-PARTITION (A, p, r)
{
    i⟵RANDOM(p, r);
    A[r]↔A[i]; First swap random element with last element.
    return PARTITION(A, p, r);;
}

- The following procedure implements quicksort that uses the PARTITION procedure, which rearranges the subarray A[p..r] in place.

Quick Sort Algorithm	Partition Algorithm

QUICKSORT (A, p, r)
{
    if (p<r)
    {
        q⟵PARTITION (A, p, r);
        QUICKSORT (A, p, q-1);
        QUICKSORT (A, q+1, r);
    }
}
	PARTITION (A, p, r)
	{
	    x⟵A[r] //Taking last element as pivot
	    i ← p-1;
	    for (j⟵p to r-1)
	    {
	        if A[j] ≤ x)
	        {
	            i ← i+1;
	            A[i] ↔ A[j]; //Internal Swap
	        }
	    }
	    i ← i+1;
	    A[i] ↔ A[r]; //Final Swap
	    return i;
	}

Analysis of Quick Sort Algorithm

- To sort an entire array A, the initial call is QUICKSORT(A, 1, n), where n is the number of elements stored in the array. (p=1, r=n)
  QUICKSORT (A, p, r) => QUICKSORT (A, 1, n) => n-1+1=n elements=>T(n)
  QUICKSORT (A, p, q-1) => QUICKSORT (A, 1, q-1) => q-1-1+1=q-1 elements=>T(q-1)
  QUICKSORT (A, q+1, r) => QUICKSORT (A, q+1, n) => n-(q+1)-1=n-q elements=>T(n-q)
  The running time of PARTITION on the subarray A[p..r] is $\Theta(n)$.
- So the recurrence for Quick Sort algorithm is as follows:

$$\mathbf{T(n) = T(q-1) + T(n-q) + n \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots(1)}$$

Average Case Analysis

- The average-case running time of quicksort is much closer to the best case than to the worst case. Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split, which at first blush seems quite unbalanced (putting suitable value of q). We then obtain the recurrence.

$$\mathbf{T(n) = T(9n/10) + T(n/10) + cn \ldots\ldots\ldots\ldots\ldots\ldots\ldots(2)}$$

- Solving the recurrence equation (1), $T(n) = O(\log_2 n)$

(b) **Devise a "Binary Search" algorithm that splits the set into two sets, one of which is twice the size of the other. How does this algorithm compare with standard binary search?** [4]

Evaluation Scheme:

- Modified Binary Search: 3 Mark
- Comparison of Modified Binary Search with Standard Binary Search: 1 Mark

Answer:

Binary Search Algorithm

<table>
<tr><td colspan="2">/* A[1..n] is a array of n elements sorted already in ascending order. Parameters p and r are integers, represents lower bound and upper bound of the array/subarray 1≤p≤r≤n. key is the element to be searched through the array A.*/</td></tr>
<tr><td>Recursive Binary Search Algorithm (Standard: splits the set into two sets of equal size)</td><td>Recursive Binary Search Algorithm (Splits the set into two sets, one of which is twice of the size of other)</td></tr>
<tr><td><pre>BINARY-SEARCH-REC (A, p, r, key)
{
  if (p≤r)
  {
     q ← (p+r)/2;
      if (key == A[q])
          return q; //Successful Search
      else if (key < A[q])
          return (A, p, q-1, key);
      else
         return (A, p+1, r, key);
   }
   return -1; //Unsuccessful Search

  }
}</pre></td><td><pre>BINARY-SEARCH-REC (A, p, r, key)
{
  if (p≤r)
  {
     q ← (p+2r+1)/3;
      if (key == A[q])
          return q; //Successful Search
      else if (key < A[q])
          return (A, p, q-1, key);
      else
         return (A, p+1, r, key);
   }
   return -1; //Unsuccessful Search

  }
}</pre></td></tr>
<tr><td><u>Recurrence Equation</u>
T(n) = T(n/2) + 1, T(1)=1</td><td><u>Recurrence Equation</u>
T(n) = T(2n/3) + 1, T(1)=1</td></tr>
</table>

- In worst case, the above modified binary search will take more time (number of comparison) than standard binary search algorithm.

Q8 a) **Given a sorted array with n distinct elements, we are required to find 3 such distinct elements whose sum is equal to 0. Design an efficient algorithm that implements the above mentioned requirements and analyze its time complexity.** [4]

Evaluation Scheme:

- Efficient Algorithm with $O(n^2)$ Time Complexity: 4 Marks
- Other Correct Algorithm more than $O(n^2)$: 3 Marks
- Incorrect algorithm, but some valid steps: step marks (0.5-2.5 Marks)

Answer:

```
void findTripletsSumZero(int a[], int n)
{
        int i, j, k;
        for(i = 0; i < n-2; i++)
        {
              // index of the first element in the remaining elements.
              j = i + 1;
             // index of the last element.
             k = n - 1;
            while(j < k)
            {
                if ((a[i] + a[j]) == -a[i])
                {
                      printf("\n %d + %d +  %d = 0", a[i], a[j], a[k]);
                      return;
                }
                else if ((a[j] + a[k]) > -a[i])
                              k--;
                else
                              j++;
            }
        }
        printf("\nNo three elements are found, whose sum is equal to zero.").
 }
```

Time Complexity = $O(n^2)$

b) **Suppose a file to be transferred through the network contains the following characters with their number of occurrences as < a: 15, b: 25, c: 5, d: 35, e: 20 >. Determine an efficient strategy that can minimize the total cost of transferring that file of 1000 characters. Find out the total cost of transfer if transferring cost for 1-bit of data is 4 units.** **[4]**

Evaluation Scheme:

- Naming the correct efficient strategy: 0.5 Mark
- Construction of Huffman Tree:1.5 Marks
- Finding optimal Huffman code: 1 Mark
- Finding out total cost of transferring the file of 1000 characters. : 1 Mark.

Answer:

- The efficient strategy is Huffman Algorithm.

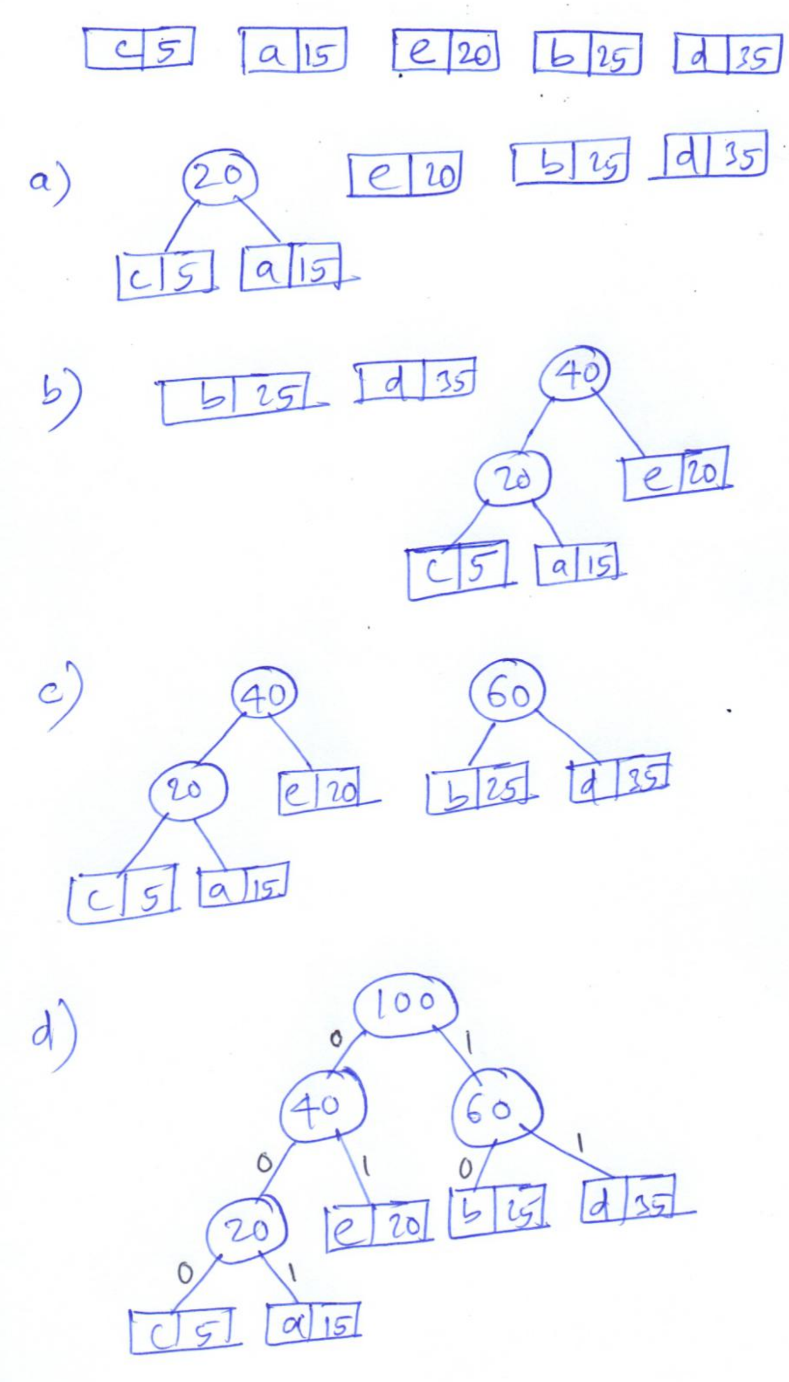
Item	Code	or
a	001	110
b	10	01
c	000	111
d	11	00
e	01	10

Comparison

Huffman Code (Variable Code)	Fixed Code
Total bits required (100 characters) = nob(a)xfeq(a) + nob(b)xfeq(b) + nob(c)xfeq(c) + nob(d)xfeq(d) + nob(d)xfeq(d) + nob(e)xfeq (e) = 3x15 + 2x25 + 3x5 + 2x35 + 2x20 =45 + 50 + 15 + 70 + 40 = 220 Cost of transfer = 220 x 4 = 880	Total bits required (100 characters) = 100x3 = 300 (as 5 character, so 3 bits) Cost of transfer = 300 x 3 = 900 **(THIS PART IS OPTIONAL**)

N. B: nob-Number of Bits, feq-Frequency

<u>Construction of Huffman Tree</u>



*** E N D ***