INDIRAGANDHINATIONALOPENUNIVERSITY

Regional Centre Delhi-3

Format for Assignment Submission

For Term End Exam June/December- JUNE **(Year)** 2025

(Please read the instructions given below carefully before submitting assignments)

1. **Name of the Student** : Ashu Chaurasiya
2. **Enrollment Number** : 250132 1326
3. **Programme Code** : MCA_NEW
4. **Course Code** (Use this format course-wise separately) : MCS-211
5. **Study Centre Code** : LSC38046
6. **Name of the Study Centre With complete address** : RAJDHANI COLLEGE RAJA GARDEN NEW DELHI
7. **Mobile Number** : 8448713694
8. **E-mail ID** : Intermezzobrilliance@gmail.com
9. **Details if this same assignment has been submitted anywhere else also** : No
10. **Above information is cross checked and it is correct: Yes/No** ✓

Date of Submission: 01-06-25

Ashu

(Signature of the student)

A. Important Instructions:-

1. Please do not send any assignment at any email of the Regional Centre, it will not be considered.
2. Please avoid duplicacy. Do not re-submit the same assignment anywhere else or by any other means.
3. About the mode of submission of assignments, pl wait for instructions from IGNOU Hqtrs. As soon as we shall come to know, we will share it with all.
4. Please do not use plastic covers. Use plain A4 size pages for assignments for uniformity and better management with this cover page format on each assignment.
5. Please write your name and enrollment no. at the bottom of each page of your assignment.
6. Please retain a photocopy set of assignment submitted with you for record(may be asked to submit at later stage) and also keep the assignment submission receipt in safe custody.
7. If assignment awards are not updated in your Grade Card **within next 09 months**, please write to us at rcdelhi3@ignou.ac.in giving your complete details and attaching the proof of assignment submission.
8. Assignment Question Paper can be downloaded from: https://webservices.ignou.ac.in/assignments/

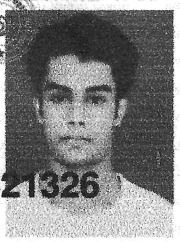
B. Compulsory sequence of the Assignment Set:

1. Duly Filled in Assignment Submission Cover Page (This Format Page).
2. Copy of IGNOU Identity Card.
3. Print out of valid/applicable assignment question paper.
4. Handwritten Assignment, written on both the sides of page (preferably plain A4 size).

ignou
THE PEOPLE'S
UNIVERSITY

इंदिरा गांधी राष्ट्रीय मुक्त विश्वविद्यालय
मैदान गढ़ी, नई दिल्ली - 110068
Indira Gandhi National Open University
Maidan Garhi,New Delhi - 110068

IGNOU - Student Identity Card



2501321326

Enrolment Number :	2501321326
RC Code :	38: DELHI 3 Naraina
Name of the Programme :	MCA_NEW : Master of Computer Applications
Name :	ASHU CHAURASIYA
Father's Name :	VIJAY
Address :	Flat No. 270,G, F Block-14, Pocket 13, Sector-20 RNORTH WEST
Pin Code :	110086

Instructions :
1. This card should be produced on demand at the Study Center, Examination Center or any other Establishment of IGNOU to use its facilities.
2. The facilities would be available only relating to the Programme/course for which the student is registered.
3. This ID Card is generated online. Students are advised to take a color print of this ID Card and get it laminated.
4. The student details can be cross checked with the QR Code at www.ignou.ac.in

Registrar
Student Registration Division

Course Code	:	**MCS-211**
Course Title	:	**Design and Analysis of Algorithms**
Assignment Number	:	**MCA_NEW(I)/211/Assign/2025**
Maximum Marks	:	**100**
Weightage	:	**30%**
Last Dates for Submission	:	**30th April 2025 (for January Session)** **31st October 2025 (for July Session)**

This assignment has four questions (80 Marks). Answer all questions. The remaining 20 marks are for viva voce. You may use illustrations and diagrams to enhance the explanations. Please go through the guidelines regarding assignments given in the Programme guide for the presentation format.

Q1: a) Design and develop an efficient algorithm to find the list of prime numbers in the range 501 to 2000. What is the complexity of this algorithm? **(2 Marks)**

b) Differentiate between Cubic-time and Factorial-time algorithms. Give example of one algorithm each for these two running times. **(2 Marks)**

c) Write an algorithm to multiply two square matrices of order n*n. Also explain the time complexity of this algorithm. **(2 Marks)**

d) What are asymptotic bounds for analysis of efficiency of algorithms? Why are asymptotic bounds used? What are their shortcomings? Explain the Big O and Big Θ notation with the help of a diagram. Find the Big O-notation and Θ-notation for the function: **(4 Marks)**

$$f(n)= 100n^4 +1000n^3 +100000$$

e) Write and explain the Left to Right binary exponentiation algorithm. Demonstrate the use of this algorithm to compute the value of 3^{29} (Show the steps of computation). Explain the worst-case complexity of this algorithm. **(4 Marks)**

f) Write and explain the Bubble sort algorithm. Discuss its best and worst-case time complexity. **(3 Marks)**

g) What are the uses of recurrence relations? Solve the following recurrence relations using the Master's method **(3 Marks)**

a. $T(n) = 4T\left(\frac{n}{4}\right) + n^1$

b. $T(n) = 4T\left(\frac{3n}{4}\right) + n^1$

Q2: a) What is an Optimisation Problem? Explain with the help of an example. When would you use a Greedy Approach to solve optimisation problem? Formulate the Task Scheduling Problem as an optimisation problem and write a greedy algorithm to solve this problem. Also, solve the following fractional Knapsack problem using greedy approach. Show all the steps. **(4 Marks)**

Suppose there is a knapsack of capacity 20 Kg and the following 6 items are to packed in it. The weight and profit of the items are as under:

$(p_1, p_2, \ldots, p_6) = (30,16,18,20,10, 7)$

$(w_1, w_2, \ldots, w_6) = (5, 4, 6, 4, 5, 7)$

Select a subset of the items that maximises the profit while keeping the total weight below or equal to the given capacity.

b) Assuming that data to be transmitted consists of only characters 'a' to 'g', design the Huffman code for the following frequencies of character data. Show all the steps of building a huffman tree. Also, show how a coded sequence using Huffman code can be decoded. **(4 Marks)**

a:5, b:25, c:10, d:15, e:8, f:7, g:30

c) Explain the Merge procedure of the Merge Sort algorithm. Demonstrate the use of recursive Merge sort algorithm for sorting the following data of size 8: [19, 18, 16, 12, 11, 10, 9, 8]. Compute the complexity of Merge Sort algorithm. **(4 Marks)**

d) Explain the divide and conquer approach of multiplying two large integers. Compute the time complexity of this approach. Also, explain the binary search algorithm and find its time complexity. **(4 Marks)**

e) Explain the Topological sorting with the help of an example. Also, explain the algorithm of finding strongly connected components in a directed Graph. **(4 Marks)**

Q3: Consider the following Graph:

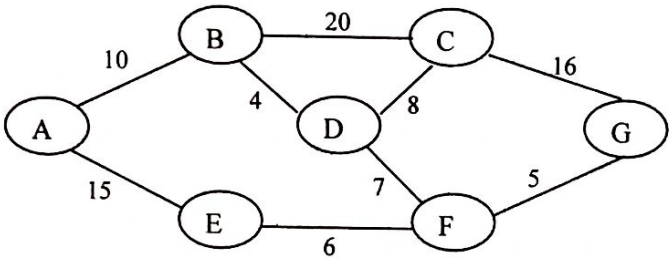


Figure 1: A sample weighted Graph

a) Write the Prim's algorithm to find the minimum cost spanning tree of a graph. Also, find the time complexity of Prim's algorithm. Demonstrate the use of Kruskal's algorithm and Prim's algorithm to find the minimum cost spanning tree for the Graph given in Figure 1. Show all the steps. **(4 Marks)**

b) Write the Dijkstra's shortest path algorithm. Also, find the time complexity of this shortest path algorithm. Find the shortest paths from the vertex 'A' using Dijkstra's shortest path algorithm for the graph given in Figure 1. Show all the steps of computation. **(4 Marks)**

c) Explain the algorithm to find the optimal Binary Search Tree. Demonstrate this algorithm to find the Optimal Binary Search Tree for the following probability data (where p_i represents the probability that the search will be for the key node k_i, whereas q_i represents that the search is for dummy node d_i. Make suitable assumptions, if any) **(6 Marks)**

i	0	1	2	3	4
p_i		0.10	0.15	0.20	0.10
q_i	0.05	0.10	0.10	0.10	0.10

d) Given the following sequence of chain multiplication of the matrices. Find the optimal way of multiplying these matrices: **(2 Marks)**

Matrix	Dimension
A1	10 × 15
A2	15 × 5
A3	5 × 20
A4	20 × 10

e) Explain the Rabin Karp algorithm for string matching with the help of an example. Find the time complexity of this algorithm. **(4 Marks)**

Q4: a) Explain the term Decision problem with the help of an example. Define the following problems and identify if they are decision problem or optimisation problem? Give reasons in support of your answer. **(4 Marks)**

(i) Travelling Salesman Problem
(ii) Graph Colouring Problem
(iii) 0-1 Knapsack Problem

b) What are P and NP class of Problems? Explain each class with the help of at least two examples. **(4 Marks)**

c) Define the NP-Hard and NP-Complete problem. How are they different from each other. Explain the use of polynomial time reduction with the help of an example. **(4 Marks)**

d) Define the following Problems: **(8 Marks)**

(i) SAT Problem
(ii) Clique problem
(iii) Hamiltonian Cycle Problem
(iv) Subset Sum Problem

Enrollment No. 2501321326

Question no.1: a) Design and develop an efficient algorithm to find the list of prime numbers in the range 501 to 2000. What is the complexity of this algorithm?

Answer:

a) Algorithm to Find Prime Numbers (Sieve of Eratosthenes)

Start.

Create a boolean array is_prime of size 2001, initialized to True. (Indices 0 and 1 are not prime, so is_prime[0] = is_prime[1] = False).

Iterate p from 2 up to the square root of 2000 (approx. 44).

If is_prime[p] is True:

For multiple = p* p up to 2000, incrementing by p:

Set is_prime[multiple] = False.

Create an empty list primes_list.

Iterate i from 501 to 2000:

If is_prime[i] is True:

Add i to primes_list.

Print primes_list.

Stop.

Complexity: The time complexity of the Sieve of Eratosthenes algorithm is approximately O(NloglogN), where N is the upper limit of the range (2000 in this case). This is highly efficient for finding primes within a given range.

Question 1 b) Differentiate between Cubic-time and Factorial-time algorithms

. Give example of one algorithm each for these two running times.

Answer:

b) Differentiating Cubic-time and Factorial-time Algorithms

Cubic-time Algorithms (O(N3)): A cubic-time algorithm's execution time grows proportionally to the cube of the input size (N). This means if the input size doubles, the time taken increases by a factor of eight (23). Cubic time is generally considered polynomial time and can be acceptable for moderate input sizes.

Example Algorithm: Standard matrix multiplication of two N×N matrices (e.g., using three nested loops, each running N times) has a time complexity of O(N3).

Factorial-time Algorithms (O(N!)): A factorial-time algorithm's execution time grows proportionally to the factorial of the input size (N!). This is an extremely rapid growth rate. Even for very small values of N (e.g., N=10, 10!=3,628,800), the time required becomes prohibitively large. Factorial time algorithms are impractical for inputs beyond a very small constant.

Example Algorithm: The naive approach to solving the Traveling Salesperson Problem (TSP) by trying every possible permutation of cities to find the shortest route is a factorial-time algorithm. Generating all permutations of N items also typically takes O(N!) time.

Question) c) Write an algorithm to multiply two square matrices of order n*n. Also explain the time complexity of this algorithm.

Answer:

c) Algorithm to Multiply Two Square Matrices (Order $n \times n$)

Start.

Declare three $n \times n$ matrices: MatrixA, MatrixB, and ResultMatrix.

Read the elements of MatrixA (of size $n \times n$).

Read the elements of MatrixB (of size $n \times n$).

Initialize all elements of ResultMatrix to 0.

For i from 0 to n-1 (for rows of ResultMatrix):

For j from 0 to n-1 (for columns of ResultMatrix):

For k from 0 to n-1 (for inner product):

ResultMatrix[i][j] = ResultMatrix[i][j] + (MatrixA[i][k] * MatrixB[k][j]).

Print ResultMatrix.

Stop.

Time Complexity Explanation: This algorithm involves three nested loops.

The outermost loop (for i) runs n times.

The middle loop (for j) runs n times.

The innermost loop (for k) runs n times.

Inside the innermost loop, constant time operations (multiplication, addition, assignment) are performed.

Therefore, the total number of fundamental operations is approximately $n \times n \times n = n3$. The time complexity of this standard matrix multiplication algorithm is $O(n3)$ (Cubic-time).

Question 1 d) What are asymptotic bounds for analysis of efficiency of

algorithms? Why are asymptotic bounds used? What are their shortcomings?

Explainthe Big O and Big notation with the help of a diagram.

Find theBig O-notation and - notation for the function:

$f(n)=100n4+1000n3+100000$

Answer:

d) Asymptotic Bounds for Algorithm Efficiency Analysis

What are Asymptotic Bounds? Asymptotic bounds are mathematical notations used to describe the limiting behavior of an algorithm's running time (or space complexity) as the input size (N) approaches infinity. They provide a high-level classification of an algorithm's efficiency, focusing on its growth rate rather than exact execution times.

Why are Asymptotic Bounds Used? Asymptotic bounds are used for several crucial reasons:

Machine Independence: They abstract away details like processor speed, compiler efficiency, or specific hardware, allowing for a universal comparison of algorithms.

Focus on Growth Rate: They highlight how an algorithm scales with increasing input size, which is critical for large datasets. Constant factors and lower-order terms become less significant for large N.

Predictive Power: They help predict how an algorithm will perform on larger inputs based on its behavior for smaller ones.

Algorithm Comparison: They provide a standardized framework to compare the inherent efficiency of different algorithms solving the same problem.

Shortcomings of Asymptotic Bounds:

Ignores Constant Factors and Lower-Order Terms: For small input sizes, an algorithm with a theoretically higher asymptotic complexity might actually perform faster due to smaller constant factors.

Worst-Case Emphasis: Big O notation typically describes the worst-case scenario, which might not always reflect the average or best-case performance in practice.

Doesn't Reflect Practical Performance: They don't account for specific hardware optimizations, cache performance, or parallel processing capabilities, which can significantly impact real-world execution.

Assumes Infinite Input: The "asymptotic" nature means they are most accurate for extremely large inputs, which might not always be relevant for typical problem sizes.

Big O (Big Oh) Notation (O-notation): Big O notation describes the upper bound of an algorithm's running time. It represents the slowest possible growth rate that the running time will not exceed. If an algorithm is $O(f(N))$, it means that for sufficiently large N, the running time $T(N)$ is at most a constant multiple of $f(N)$. It tells you the "worst-case" or "not worse than" performance.

Big Theta (f xx) Notation (Θ-notation): Big Theta notation describes the tight bound (both upper and lower bound) of an algorithm's running time. It signifies that the running time $T(N)$ grows at the same rate as $f(N)$. If

algorithm is ($f(N)$), it means that for sufficiently large N, the runningtime $T(N)$ is bounded both above and below by constant multiples of $f(N)$.

It describes the "exactly equal to" or "average-case" growth rate.

Finding Big O and Big for $f(n)=100n^4+1000n^3+100000$

To find the Big O and Big notation, we identify the term with the highest growth rate. In the given function, n^4 is the dominant term as n approaches infinity.

Big O-notation: $O(n^4)$

Explanation: For sufficiently large n, $100n^4+1000n^3+100000$ will not grow faster than a constant multiple of n^4. The lower-order terms ($1000n^3$ and 100000) become insignificant as n increases.

Big -notation: (n^4)

Explanation: The function $100n^4+1000n^3+100000$ grows exactly at the rate of n^4. It is both upper-bounded and lower-bounded by constant multiples of n^4 for large n. This signifies that n^4 accurately represents the average-case as well as worst-case growth.

Question 1 e) Write and explain the Left to Right binary exponentiation algorithm. Demonstrate the use of this algorithm to compute the value of 3^{29} (Show the steps of computation). Explain the worst-case complexity of this algorithm.

Answer:

e) Left-to-Right Binary Exponentiation Algorithm

Algorithm Explanation: The Left-to-Right (LTR) binary exponentiation algorithm

(also known as the "square-and-multiply" algorithm) efficiently computesbaseexponent by processing the exponent's binary representation from leftto right (most significant bit to least significant bit). It leveragesthe property that $x^{2k} = (x^k)^2$ and $x^{2k+1} = (x^k)^2 x$.

Start.

Input base and exponent.

Convert the exponent to its binary representation.

Initialize result = 1.

Iterate through the binary digits of the exponent from left to right (from the most significant bit to the least significant bit).

For each bit:

Always Square: result = result * result.

If the current bit is 1: result = result * base.

Return result.

Stop.

Demonstration: Compute 3^{29}

Base: 3

Exponent: 29

Binary representation of 29:

29/2=14 remainder 1

14/2=7 remainder 0

7/2=3 remainder 1

3/2=1 remainder 1

1/2=0 remainder 1 Reading remainders from bottom up: 2910=1110112.

Computation Steps:

(Initial) result = 1 1 Initialize result.

1 (MSB) result = result * result (1 * 1 = 1) 1 Always square.

result = result * base (1 * 3 = 3) 3 Bit is 1, so multiply by base. (Current: 31)

1 result = result * result (3* 3 = 9) 9 Always square. (Current: 32)

result = result * base (9* 3 = 27) 27 Bit is 1, so multiply by base. (Current: 33)

1 result = result * result (27* 27 = 729) 729 Always square. (Current: 36)

result = result * base (729* 3 = 2187) 2187 Bit is 1, so multiply by base. (Current: 37)

0 result = result * result (2187* 2187 = 4782969) 4782969 Always square. (Current: 314)

(No multiply by base) 4782969 Bit is 0.

1 (LSB) result = result * result (4782969* 4782969 = 22876792454961) 22876792454961 Always square. (Current: 328)

result = result * base (22876792454961 * 3 = 68630377364883) 68630377364883 Bit is 1, so multiply by base. (Current: 329)

Therefore, 329=68,630,377,364,883.

Worst-Case Complexity: Let N be the exponent. The number of bits in the

binary representation of N is approximately log2N+1). In the worst case, every bit in the binary representation of the exponent is a 1).

For each bit, we perform:

One squaring operation (result = result * result).

Potentially one multiplication by the base (result = result * base).

So, for an exponent N, there are roughly log2N squaring operations and at most log2N multiplication operations. Thus, the worst-case time complexity of the Left-to-Right Binary Exponentiation algorithm is O(logN) in terms of the number of multiplications.

f) Write and explain the Bubble sort algorithm. Discuss its best and worst-case time complexity. Answer:

f) Bubble Sort Algorithm

Algorithm Explanation: Bubble Sort is a simple comparison-based sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which means the list is sorted. Larger elements "bubble" up to their correct positions at the end of the list with each pass.

Start.

Input a list of N elements, Arr.

For i from 0 to N-2 (This loop controls the number of passes):

Set a flag swapped = False.

For j from 0 to N-2-i (This loop compares adjacent elements in the

unsorted portion):

If Arr[j] is greater than Arr[j+1]:

Swap Arr[j] and Arr[j+1].

Set swapped = True.

If swapped is False after an entire pass (meaning no elements were swapped), the list is sorted, so break from the outer loop.

Print the sorted Arr.

Stop.

Best-Case Time Complexity: The best case for Bubble Sort occurs when the array is already sorted. In this scenario, during the first pass (outer loop i=0), the inner loop (j) will iterate through all elements, but no swaps will occur because all elements are already in their correct order. The swapped flag will remain False. Consequently, the algorithm will detect that no swaps happened in the first pass and will immediately break from the outer loop. The inner loop still performs N-1 comparisons. Therefore, the best-case time complexity is O(N) (linear time), as it only requires one pass to confirm that the list is sorted.

Worst-Case Time Complexity: The worst case for Bubble Sort occurs when the array is sorted in reverse order (e.g., descending order when sorting ascending). In this scenario, every pair of adjacent elements will need to be swapped in almost every pass.

In the first pass, N-1 comparisons and up to N-1 swaps.

In the second pass, N-2 comparisons and up to N-2 swaps.

...and so on, until the last pass which involves 1 comparison. The total number of comparisons (and swaps) will be approximately $(N-1)+(N-2)+\cdots+1 = N(N-1)/2$. This sum is equivalent to $O(N^2)$. Therefore, the worst-case time complexity of Bubble Sort is $O(N^2)$ (quadratic time).

Question 1 part g) What are the uses of recurrence relations? Solve the following recurrence relations using the Master's method

a. $T(n) = 4T(n/4) + n$

b. $T(n) = 4T(3n/4) + n$

Answer:

Recurrence relations are incredibly useful in computer science and mathematics. Here's what they're primarily used for:

Analyzing Algorithm Efficiency: This is a huge one in computer science. Many algorithms, especially those that call themselves (recursive algorithms like Merge Sort or Quick Sort), can be described by a recurrence relation. This relation shows how the running time for a problem of size 'n' depends on the running time of smaller versions of the problem. Solving these relations helps us figure out how fast an algorithm is (for example, its time complexity like $O(n \log n)$ or $O(n^2)$).

Counting Problems: They're great for figuring out the number of ways to do something, where the answer for 'n' items depends on the answer for fewer items. Think about counting different arrangements or combinations.

Modeling Growth and Decay: In various fields, recurrence relations can describe things that change step-by-step over time, like how populations grow,

how much money you earn with compound interest, or how a radioactivesubstance breaks down.

Dynamic Programming: Recurrence relations are the core idea behind dynamic programming. When you solve problems with dynamic programming, you usually define a recurrence that breaks the problem into smaller, overlapping parts. Then, you save the results of those smaller parts to avoid recomputing them

.

Combinatorics: They're heavily used in combinatorics for counting objects with specific properties, such as the number of different binary trees you can make with 'n' nodes.

Discrete Mathematics: Recurrence relations are fundamental in discrete mathematics for studying sequences, series, and various structures.

Now, let's solve the recurrence relations you provided using the Master's Method.

The Master's Method applies to relations that look like this: $T(n) = aT(n/b) + f(n)$

Where 'a' is 1 or greater, 'b' is greater than 1, and 'f(n)' is a function that represents the cost of the work done outside the recursive calls.

We compare 'f(n)' with n to the power of (log base b of a)'.

There are three main scenarios:

Case 1: If 'f(n)' is significantly smaller than n to the power of (log base b of a)', then the solution is dominated by the recursive calls, and

T(n) is approximately n to the power of (log base b of a)'. Case 2: If 'f(n)' is about the same size as n to the power of (log base b of a)', then the solution is n to the power of (log base b of a) multiplied by log n'. Case 3: If 'f(n)' is significantly larger than n to the power of (log base b of a)', and it satisfies a "regularity condition" (meaning the non-recursive part grows fast enough), then the solution is dominated by 'f(n)', and T(n) is approximately 'f(n)'.

a. $T(n) = 4T(n/4) +$ n to the power of 1

Here's what we have:

$a = 4$

$b = 4$

$f(n) =$ n to the power of 1 (which is just n)

First, let's calculate n to the power of (log base b of a): n to the power of (log base 4 of 4) = n to the power of 1.

Now, compare 'f(n)' with n to the power of (log base b of a): 'f(n)' is n to the power of 1' and n to the power of (log base b of a)' is also n to the power of 1'. Since they are of the same approximate size, this falls under Case 2 of the Master's Method.

Therefore, the solution is: T(n) is approximately n to the power of (log base b of a)' multiplied by log n'. T(n) is approximately n to the power of 1' multiplied by 'log n'. T(n) is approximately n log n.

b. $T(n) = 4T(3n/4) +$ n to the power of 1

This one looks a little different because of 3n/4, but we can still use the Master's Method.

Here's what we have:

a = 4

The effective b is 4/3 (because 3n/4 is the same as n divided by 4/3). So, b = 4/3.

f(n) = n to the power of 1 (which is just n)

First, let's calculate n to the power of (log base b of a): n to the power of (log base 4/3 of 4)

Let's estimate the value of log base 4/3 of 4: This is equivalent to (log 4) divided by (log 4/3). Using a calculator, (log 4) is roughly 1.386, and (log 4/3) is roughly 0.288. So, (log base 4/3 of 4) is approximately 1.386 / 0.288, which is about 4.8125.

So, n to the power of (log base b of a) is approximately n to the power of 4.8125.

Now, compare f(n) with n to the power of (log base b of a): f(n) is n to the power of 1, and n to the power of (log base b of a) is approximately n to the power of 4.8125.

Since 1 is much smaller than 4.8125, f(n) is significantly smaller than n to the power of (log base b of a). This falls under Case 1 of the Master's Method.

Therefore, the solution is: T(n) is approximately n to the power of (log base b of a). T(n) is approximately n to the power of (log base 4/3

of 4).

For practical purposes, this is T(n) is approximately n to the power of 4

8) 25.

Question 2. a)

Answer: An Optimization Problem is about finding the best solution from many possible choices. "Best" means either getting the maximum of something good (like profit) or the minimum of something bad (like cost).

Example: Finding the shortest route between two cities on a map is an optimization problem where you want to minimize distance.

When to use a Greedy Approach: Use a Greedy Approach when making the best choice right now at each step leads to the best overall solution.

It works for problems where:

Greedy Choice Property: The best immediate choice helps achieve the overall best solution without needing to backtrack.

Optimal Substructure: The best solution to the whole problem contains best solutions to its smaller parts. If these hold, a greedy algorithm can be fast.

Task Scheduling Problem (Optimization): Goal: Maximize total profit from tasks. Input: Tasks with profit and a deadline. Each task takes one unit of time. Rules: Only one task at a time; profit earned only if completed by deadline. Formulation: Choose a subset of tasks and schedule them to maximize total profit while meeting deadlines.

Greedy Algorithm for Task Scheduling:

Sort Tasks: Arrange tasks by decreasing profit.

Initialize Schedule: Create empty time slots up to the latest deadline.

Schedule Tasks: For each task (from highest profit):

Try to place it in the latest possible empty slot up to its deadline.

If placed, add its profit to the total.

Result: The schedule and total profit.

Example Walkthrough: Tasks: A(10,2), B(15,1), C(5,2), D(8,1). Max deadline = 2.

Sorted: B(15,1), A(10,2), D(8,1), C(5,2)

Schedule: [empty, empty]

Process:

B(15,1): Place in slot 1. Profit = 15. Schedule: [B, empty]

A(10,2): Place in slot 2. Profit = 15+10 = 25. Schedule: [B, A]

D(8,1): Slot 1 taken. Cannot schedule.

C(5,2): Slot 2 taken, Slot 1 taken. Cannot schedule. Final: B in slot 1, A in slot 2. Max Profit = 25.

Fractional Knapsack Problem (Greedy Solution): Knapsack Capacity: 20 Kg Items:

Profits: p1=30, p2=16, p3=18, p4=20, p5=10, p6=7

Weights: w1=5, w2=4, w3=6, w4=4, w5=5, w6=7

Steps:

Calculate Profit-per-Weight (P/W) Ratio:

Item 1: 30/5 = 6.0

Item 2: 16/4 = 4.0

Item 3: 18/6 = 30

Item 4: 20/4 = 50

Item 5: 10/5 = 20

Item 6: 7/7 = 10

Sort Items by P/W (highest to lowest): Item 1 (60), Item 4 (50), Item 2 (40), Item 3 (30), Item 5 (20), Item 6 (10)

Fill Knapsack: (Current Capacity = 20 Kg, Total Profit = 0)

Take Item 1: Weight 5 Kg. Capacity >= 5. Take all.

Remaining Capacity = 20 - 5 = 15 Kg. Total Profit = 30.

Take Item 4: Weight 4 Kg. Capacity >= 4. Take all.

Remaining Capacity = 15 - 4 = 11 Kg. Total Profit = 30 + 20 = 50.

Take Item 2: Weight 4 Kg. Capacity >= 4. Take all.

Remaining Capacity = 11 - 4 = 7 Kg. Total Profit = 50 + 16 = 66.

Take Item 3: Weight 6 Kg. Capacity >= 6. Take all.

Remaining Capacity = 7 - 6 = 1 Kg. Total Profit = 66 + 18 = 84.

Take Item 5: Weight 5 Kg. Capacity = 1 Kg. Can't take all.

Take 1/5 (0.2) of Item 5. Profit from fraction = 0.2 * 10 = 2.

Remaining Capacity = 1 - 1 = 0 Kg. Total Profit = 84 + 2 = 86.

Item 6: No capacity left.

Result: Items 1, 4, 2, 3 (all), and 0.2 of Item 5 are taken. Total Weight = 20 Kg. Maximum Profit = 86.

Question 2 b)

Answer: Huffman Code Design: Huffman coding compresses data by giving

shorter codes to frequently used characters and longer codes to less frequent ones.

Frequencies: a: 5, b: 25, c: 10, d: 15, e: 8, f: 7, g: 30

Steps to Build Huffman Tree:

Start with nodes: (a: 5), (b: 25), (c: 10), (d: 15), (e: 8), (f: 7), (g: 30).

Combine smallest two frequencies repeatedly:

(a: 5), (f: 7) combine to (a+f: 12).

(e: 8), (c: 10) combine to (e+c: 18).

(a+f: 12), (d: 15) combine to (a+f+d: 27).

(e+c: 18), (b: 25) combine to (e+c+b: 43).

(a+f+d: 27), (g: 30) combine to (a+f+d+g: 57).

(e+c+b: 43), (a+f+d+g: 57) combine to (Root: 100). (Imagine this forming a tree where each combination is a new parent node)

Assigning Codes (0 for left branch, 1 for right branch from parent): By tracing from the Root down to each character:

a: 1000

b: 01

c: 001

d: 101

e: 000

f: 1001

g: 11

How to Decode Huffman Code: To decode, you use the Huffman tree. Start at

the trees root. Read the encoded bits one by one:

If you read '0', go to the left child.

If you read '1', go to the right child.

When you reach a character (a "leaf node"), you've decoded one character.

Write it down, then start over from the root with the next bit.

Example Decoding: Coded sequence 1100010011001

11: From root, right (1), then right (1) -> g. (Restart from root)

000: From root, left (0), left (0), left (0) -> e. (Restart from root)

101: From root, right (1), left (0), right (1) -> d. (Restart from root)

1001: From root, right (1), left (0), left (0), right (1) -> f.

(Restart from root) Decoded Sequence: gedf

Question 2 c)

Answer: Merge Procedure of Merge Sort: The Merge procedure is the heart of Merge Sort. It takes two lists that are already sorted and combines them into one single, larger sorted list.

How it works:

Imagine you have two sorted piles of cards.

You look at the top card of each pile.

You pick the smaller card of the two and put it into a new, empty sorted pile.

You repeat this process, always picking the smaller top card, until one of the original piles is empty.

Then, you just put all the remaining cards from the other pile into your new sorted pile (since they're already sorted and bigger than everything you've moved so far). This results in one perfectly sorted pile.

Recursive Merge Sort Example: Sorting: [19, 18, 16, 12, 11, 10, 9, 8] Steps:

Divide: The algorithm repeatedly splits the list in half until each sub-list has only one element (which is by definition sorted).

[19, 18, 16, 12, 11, 10, 9, 8]

[19, 18, 16, 12] | [11, 10, 9, 8]

[19, 18] | [16, 12] | [11, 10] | [9, 8]

[19] [18] | [16] [12] | [11] [10] | [9] [8] (Each is a sorted list of 1 element)

Conquer (Merge): Now, it starts merging these tiny sorted lists back together.

Merge [19] and [18] -> [18, 19]

Merge [16] and [12] -> [12, 16]

Merge [11] and [10] -> [10, 11]

Merge [9] and [8] -> [8, 9]

Merge [18, 19] and [12, 16] -> [12, 16, 18, 19]

Merge [10, 11] and [8, 9] -> [8, 9, 10, 11]

Finally, Merge [12, 16, 18, 19] and [8, 9, 10, 11] -> [8, 9, 10, 11, 12, 16, 18, 19]

Final Sorted List: [8, 9, 10, 11, 12, 16, 18, 19]

Complexity of Merge Sort:

Time Complexity: O(n log n)

Why? Each time you split the list, it's log n' levels of splitting.

At each level, the 'merge' operation (combining all sub-lists) takes about n' steps.

So, it's roughly n' (for merging) multiplied by log n' (for levels).

Merge Sort is consistently O(n log n) in all cases (best, average, worst), which is very efficient for sorting.

Space Complexity: O(n)

Why? The Merge procedure needs extra temporary space to hold the elements while it's combining the two sorted sub-lists. This extra space is roughly proportional to the size of the original list.

Question 2 d):

Answer: Divide and Conquer for Multiplying Two Large Integers: The standard way to multiply two numbers (like you learned in school) takes a long time if the numbers are very large. Divide and Conquer, specifically Karatsuba's algorithm, speeds this up.

Let's say we want to multiply two big numbers, X and Y, each with n' digits. We split X and Y into two halves (roughly n/2 digits each): X = (First half of X) * 10 to the power of n/2) + (Second half of X) Y = (First half of Y) * 10 to the power of n/2) + (Second half of Y)

Instead of doing four separate multiplications of the halves (which would be

slow), Karatsuba's trick shows how to do it with only three multiplications of the halves. Then you combine these three results with some additions and subtractions.

Time Complexity: The time it takes grows roughly by O(n to the power of 1.585). This is faster than the traditional method, which takes O(n squared) time.

Binary Search Algorithm: Binary Search is a very efficient way to find a specific item in a sorted list or array.

How it works:

Start in the Middle: Look at the item in the exact middle of the sorted list.

Compare:

If the middle item is what you're looking for, you found it!

If your target item is smaller than the middle item, you know it must be in the left half of the list (because the list is sorted).

If your target item is larger than the middle item, you know it must be in the right half of the list.

Repeat: You then repeat steps 1 and 2 on the chosen half (discarding the other half). You keep halving the search area until you find the item or the search area becomes empty.

Example: Find 12 in [8, 9, 10, 11, 12, 16, 18, 19]

Middle is 11. Target (12) is > 11. Look in right half: [12, 16, 18, 19]

Middle of [12, 16, 18, 19] is 16. Target (12) is < 16. Look in

left half: [1 2]

Middle of [1 2] is 1 2. Target (1 2) == 1 2. Found it!

Time Complexity: O(log n)

Why? With each step, Binary Search cuts the search space in half. For a list of 'n' items, it takes 'log n' steps to narrow down the search to a single item. This is extremely fast, especially for very large lists.

Question 2 e):

Answer: Topological Sorting: Topological sorting is a way to arrange the nodes (like steps or tasks) of a directed graph (a graph where connections only go one way, like a one-way street) in a linear order. This order ensures that if there's a path from node A to node B, then A always appears before B in the sorted list.

Conditions: It can only be done on Directed Acyclic Graphs (DAGs) - meaning directed graphs that have no cycles (you can't start at a node and follow arrows back to it).

Example: Imagine a list of college courses and their prerequisites:

Course A has no prerequisites.

Course B requires A.

Course C requires A.

Course D requires B and C.

Course E requires D.

A topological sort would give you a valid order to take these courses: One

possible topological sort: A -> B -> C -> D -> E Another valid sort: A -> C -> B -> D -> E (Notice: A always comes before B, C, D, E B and C can be taken after A in any order before D. D comes after B and C. E comes after D)

Algorithm for finding Strongly Connected Components (SCCs) in a Directed Graph: A Strongly Connected Component (SCC) in a directed graph is a set of nodes where every node in the set can be reached from every other node in the same set by following the direction of the arrows. It's like a self-contained "loop" of interconnected nodes.

Algorithm (Kosaraju's Algorithm or Tarjan's Algorithm are common): Ill describe a common approach (similar to Kosaraju's):

First Depth-First Search (DFS) - Find Finishing Times:

Perform a Depth-First Search on the original graph.

During this DFS, keep track of the "finishing time" for each node (the time at which the DFS finishes exploring all paths reachable from that node). Store these nodes in a list, ordered by decreasing finishing times.

Reverse the Graph:

Create a new graph where all the directions of the edges are reversed. If there was an edge from A to B, now there's an edge from B to A.

Second DFS - Find SCCs:

Perform another DFS on the reversed graph.

When you start a new DFS from a node, pick the node that has the

highest finishing time (from your list in step 1) among all unvisited nodes. Every time you start a new DFS from an unvisited node in this step, all the nodes reachable from it in this DFS form one Strongly Connected Component.

This algorithm works because the first DFS helps to "flatten" the graph in a way that reveals the structure of SCCs. The second DFS on the reversed graph then correctly identifies these self-contained components.

Question 3:

a)

Answer:

Prim's Algorithm to find Minimum Cost Spanning Tree (MST):

Prim's algorithm builds an MST by starting from a chosen vertex and iteratively adding the cheapest edge that connects a vertex already in the MST to a vertex outside the MST.

Algorithm Steps:

Initialize:

Start with a single vertex (e.g., A) as the initial MST.

Maintain a set of vertices already in the MST (initially just A).

Maintain a list of edges that connect vertices in the MST to vertices outside the MST.

Iterate: While there are still vertices not in the MST:

Select the edge with the minimum weight from the list that connects a vertex in the MST to a vertex not yet in the MST.

Add this selected edge to the MST.

Add the new vertex (the one that was outside the MST) to the set of vertices already in the MST.

Update the list of potential edges (new edges might now connect to the MST).

Result: The set of selected edges forms the Minimum Spanning Tree.

Time Complexity of Prim's Algorithm:

The time complexity depends on the data structure used to find the minimum-weight edge:

Using a simple array (adjacency matrix): O(V squared) where V is the number of vertices.

Using a min-priority queue (binary heap): O(E log V) or O(E + V log V) where E is the number of edges and V is the number of vertices.

This is more efficient for sparse graphs (fewer edges). For a dense graph (many edges), O(V squared) might be better.

Demonstration for Figure 1:

Graph Edges and Weights:

(A, B): 10

(A, E): 15

(B, C): 20

(B, G): 4

(C, G): 8

(C, ?): 16 (Let's call the unnamed node H, so (C, H): 16)

(E, F): 6

(F, G): 7

(F, H): 5

1. Kruskal's Algorithm to find MST:

Kruskal's algorithm builds the MST by adding edges in increasing order of weight, as long as they don't form a cycle. It uses a Disjoint Set Union (DSU) data structure to efficiently check for cycles.

Steps:

List all edges sorted by weight (ascending):

(B, G): 4

(F, H): 5

(E, F): 6

(F, G): 7

(C, G): 8

(A, B): 10

(A, E): 15

(C, H): 16

(B, C): 20

Initialize sets: Each vertex is in its own set.

Sets: {A}, {B}, {C}, {E}, {F}, {G}, {H}

MST Edges = {}

Total Cost = 0

Process edges:

Edge (B, G): 4

B is in {B}, G is in {G}. They are in different sets.

Add (B, G) to MST. Union {B} and {G}.

Sets: {A}, {C}, {E}, {F}, {H}, {B, G}

MST Edges = {(B, G)}

Total Cost = 4

Edge (F, H): 5

F is in {F}, H is in {H}. Different sets.

Add (F, H) to MST. Union {F} and {H}.

Sets: {A}, {B, G}, {C}, {E}, {F, H}

MST Edges = {(B, G), (F, H)}

Total Cost = 4 + 5 = 9

Edge (E, F): 6

E is in {E}, F is in {F, H}. Different sets.

Add (E, F) to MST. Union {E} and {F, H}.

Sets: {A}, {B, G}, {C}, {E, F, H}

MST Edges = {(B, G), (F, H), (E, F)}

Total Cost = 9 + 6 = 15

Edge (F, G): 7

F is in {E, F, H}, G is in {B, G}. Different sets.

Add (F, G) to MST. Union {E, F, H} and {B, G}.

Sets: {A}, {C}, {B, E, F, G, H}

MST Edges = {(B, G), (F, H), (E, F), (F, G)}

Total Cost = 15 + 7 = 22

Edge (C, G): 8

C is in {C}, G is in {B, E, F, G, H}. Different sets.

Add (C, G) to MST. Union {C} and {B, E, F, G, H}.

Sets: {A}, {B, C, E, F, G, H} (All connected except A)

MST Edges = {(B, G), (F, H), (E, F), (F, G), (C, G)}

Total Cost = 22 + 8 = 30

Edge (A, B): 10

A is in {A}, B is in {B, C, E, F, G, H}. Different sets.

Add (A, B) to MST. Union {A} and {B, C, E, F, G, H}.

Sets: {A, B, C, E, F, G, H} (All vertices now in one set)

MST Edges = {(B, G), (F, H), (E, F), (F, G), (C, G), (A, B)}

Total Cost = 30 + 10 = 40

(At this point, MST has V-1 = 7-1 = 6 edges. All vertices are connected. We can stop)

Remaining edges (A, E): 15, (C, H): 16, (B, C): 20 would form cycles, so they are not added.

Minimum Spanning Tree Edges (Kruskal's): (A, B), (B, G), (C, G), (E, F), (F, G), (F, H)

Total Minimum Cost (Kruskal's): 40

2. Prim's Algorithm to find MST:

Let's start from vertex A.

Steps:

Initialize:

MST Vertices = {A}

MST Edges = {}

Total Cost = 0

Candidate Edges (from A to outside): (A, B) with weight 10, (A, E) with weight 15.

Iteration 1:

Smallest candidate edge: (A, B) with weight 10.

Add (A, B) to MST Edges.

MST Vertices = {A, B}

Total Cost = 10

Update candidate edges:

Edges from {A, B} to outside:

(A, E): 15

(B, G): 4

(B, C): 20

Current candidates: (A, E): 15, (B, G): 4, (B, C): 20

Iteration 2:

Smallest candidate edge: (B, G) with weight 4.

Add (B, G) to MST Edges.

MST Vertices = {A, B, G}

Total Cost = 10 + 4 = 14

Update candidate edges:

Edges from {A, B, G} to outside:

(A, E): 15

(B, C): 20

(C, G): 8

(F, G): 7

Current candidates: (A, E): 15, (B, C): 20, (C, G): 8, (F, G): 7

Iteration 3:

Smallest candidate edge: (F, G) with weight 7.

Add (F, G) to MST Edges.

MST Vertices = {A, B, G, F}

Total Cost = 14 + 7 = 21

Update candidate edges:

Edges from {A, B, G, F} to outside:

(A, E): 15

(B, C): 20

(C, G): 8

(E, F): 6

(F, H): 5

Current candidates: (A, E): 15, (B, C): 20, (C, G): 8, (E, F): 6, (F, H): 5

Iteration 4:

Smallest candidate edge: (F, H) with weight 5.

Add (F, H) to MST Edges.

MST Vertices = {A, B, G, F, H}

Total Cost = 21 + 5 = 26

Update candidate edges:

Edges from {A, B, G, F, H} to outside:

(A, E): 15

(B, C): 20

(C, G): 8

(E, F): 6 (Note: E is outside, F is inside)

(C, H): 16 (Note: C is outside, H is inside)

Current candidates: (A, E): 15, (B, C): 20, (C, G): 8, (E, F): 6, (C, H): 16

Iteration 5:

Smallest candidate edge: (E, F) with weight 6.

Add (E, F) to MST Edges.

MST Vertices = {A, B, G, F, H, E}

Total Cost = 26 + 6 = 32

Update candidate edges:

Edges from {A, B, G, F, H, E} to outside:

(A, E): 15 (A and E are now both inside) - No, E just got added. A is inside, E was outside. Now A is inside, E is inside. This edge is not considered as a candidate to connect outside.

(B, C): 20

(C, G): 8

(C, H): 16

Current candidates: (B, C): 20, (C, G): 8, (C, H): 16

Iteration 6:

Smallest candidate edge: (C, G) with weight 8.

Add (C, G) to MST Edges.

MST Vertices = {A, B, G, F, H, E, C} (All vertices now included)

Total Cost = 32 + 8 = 40

We have included V-1 = 7-1 = 6 edges, and all vertices are part of the MST. We stop.

Minimum Spanning Tree Edges (Prim's): (A, B), (B, G), (F, G), (F, H), (E, F), (C, G)

Total Minimum Cost (Prim's): 40

(Both algorithms yield the same MST and total cost, as expected)

Question 3

b):

Answer:

Dijkstra's Shortest Path Algorithm:

Dijkstra's algorithm finds the shortest paths from a single starting vertex to all other vertices in a graph with non-negative edge weights.

Algorithm Steps:

Initialize:

For each vertex v, set its distance[v] to infinity (a very large number) and its previous[v] to null.

Set distance[start_vertex] to 0.

Create a min-priority queue (or a set of unvisited vertices) and add all vertices to it. The priority is based on distance[v].

Iterate: While the priority queue is not empty:

Extract the vertex u with the smallest distance[u] from the priority queue.

If distance[u] is infinity, then all remaining unvisited vertices are unreachable from the start vertex, so stop.

Relax Edges: For each neighbor v of u:

Calculate the alt_distance = distance[u] + weight(u, v).

If alt_distance < distance[v]:

Update distance[v] = alt_distance.

Set previous[v] = u.

Update v's priority in the priority queue (if using one that supports updates).

Result: distance[v] will hold the shortest distance from the start vertex to v, and previous[v] can be used to reconstruct the path.

Time Complexity of Dijkstra's Algorithm:

Using a simple array (linear scan for min): O(V squared) where V is the number of vertices.

Using a min-priority queue (binary heap): O(E log V) or O(E + V log V) where E is the number of edges and V is the number of vertices. This is more efficient for sparse graphs.

Using a Fibonacci heap (theoretical best): O(E + V log V).

Finding Shortest Paths from Vertex 'A' using Dijkstra's Algorithm for Figure 1:

Vertices: A, B, C, E, F, G, H (unnamed node assumed as H)

Edges and Weights:

(A, B): 10, (A, E): 15

(B, C): 20, (B, G): 4

(C, G): 8, (C, H): 16

(E, F): 6

(F, G): 7, (F, H): 5

Initialization:

dist[A] = 0

dist[B] = infinity

dist[C] = infinity

dist[E] = infinity

dist[F] = infinity

dist[G] = infinity

dist[H] = infinity

prev array: all null

unvisited_vertices = {A, B, C, E, F, G, H} (min-priority queue)

Steps:

Extract A (dist=0):

Remove A from unvisited_vertices.

Neighbors of A: B, E

To B: alt = dist[A] + weight(A, B) = 0 + 10 = 10.

Since 10 < dist[B] (infinity), dist[B] = 10, prev[B] = A.

To E: alt = dist[A] + weight(A, E) = 0 + 15 = 15.

Since 15 < dist[E] (infinity), dist[E] = 15, prev[E] = A.

dist values: {A: 0, B: 10, C: inf, E: 15, F: inf, G: inf, H: inf}

Extract B (dist=10):

Remove B from unvisited_vertices.

Neighbors of B: A, C, G

To A: Already visited. Ignore.

To C: alt = dist[B] + weight(B, C) = 10 + 20 = 30.

Since 30 < dist[C] (infinity), dist[C] = 30, prev[C] = B.

To G: alt = dist[B] + weight(B, G) = 10 + 4 = 14.

Since 14 < dist[G] (infinity), dist[G] = 14, prev[G] = B.

dist values: {A: 0, B: 10, C: 30, E: 15, F: inf, G: 14, H: inf}

Extract G (dist=14):

Remove G from unvisited_vertices.

Neighbors of G: B, C, F

To B: Already visited. Ignore.

To C: alt = dist[G] + weight(G, C) = 14 + 8 = 22.

Since 22 < dist[C] (30), dist[C] = 22, prev[C] = G. (Path A->B->G->C is shorter than A->B->C)

To F: alt = dist[G] + weight(G, F) = 14 + 7 = 21.

Since 21 < dist[F] (infinity), dist[F] = 21, prev[F] = G.

dist values: {A: 0, B: 10, C: 22, E: 15, F: 21, G: 14, H: inf}

Extract E (dist=15):

Remove E from unvisited_vertices.

Neighbors of E: A, F

To A: Already visited. Ignore.

To F: alt = dist[E] + weight(E, F) = 15 + 6 = 21.

Since 21 is not less than dist[F] (which is already 21), no update.

(Path A->G->F is same cost as A->E->F)

dist values: {A: 0, B: 10, C: 22, E: 15, F: 21, G: 14, H: inf}

Extract F (dist=21):

Remove F from unvisited_vertices.

Neighbors of F: E, G, H

To E: Already visited. Ignore.

To G: Already visited. Ignore.

To H: alt = dist[F] + weight(F, H) = 21 + 5 = 26.

Since 26 < dist[H] (infinity), dist[H] = 26, prev[H] = F.

dist values: {A: 0, B: 10, C: 22, E: 15, F: 21, G: 14, H: 26}

Extract C (dist=22):

Remove C from unvisited_vertices.

Neighbors of C: B, G, H

To B: Already visited. Ignore.

To G: Already visited. Ignore.

To H: alt = dist[C] + weight(C, H) = 22 + 16 = 38.

Since 38 is not less than dist[H] (26), no update. (Path A->G->F->H is shorter than A->B->G->C->H)

dist values: {A: 0, B: 10, C: 22, E: 15, F: 21, G: 14, H: 26}

Extract H (dist=26):

Remove H from unvisited_vertices.

No unvisited neighbors to process.

Final Shortest Distances from A:

A to A: 0

A to B: 10 (Path: A -> B)

A to C: 22 (Path: A -> B -> G -> C)

A to E: 15 (Path: A -> E)

A to F: 21 (Path: A -> B -> G -> F or A -> E -> F)

A to G: 14 (Path: A -> B -> G)

A to H: 26 (Path: A -> B -> G -> F -> H or A -> E -> F -> H)

Question 3 c):

Answer:

Algorithm to Find the Optimal Binary Search Tree (OBST):

An Optimal Binary Search Tree (OBST) is a binary search tree (BST) that minimizes the expected search cost (or total cost) for a given set of keys and their search probabilities. It's an "optimization problem" that is solved using Dynamic Programming.

Assumptions:

We have n sorted keys: $k1 < k2 < \ldots < kn$.

pi is the probability that a search will be for key ki.

qi is the probability that a search will be for a value between key ki and k(i+1), represented by dummy nodes (d0, d1, ..., dn). d0 represents values less than k1, d1 values between k1 and k2, etc, dn values greater than kn.

The sum of all probabilities (all p's and all q's) is 1.

Why Dynamic Programming?

A naive approach would be to try every possible BST, which is too many (Catalan numbers). OBST has:

Optimal Substructure: The optimal BST for a set of keys k(i)...k(j) (and their associated dummy nodes) must contain optimal BSTs for its left and right subtrees.

Overlapping Subproblems: The same subproblems (finding OBST for a range of keys) are solved multiple times. Dynamic programming stores these solutions to avoid recomputation.

Algorithm Steps (High-Level):

Define Cost Function:

The cost of a search is the depth of the node (or dummy node) + 1.

The total expected search cost is sum of (probability * cost) for all keys and dummy nodes.

Create Tables: We typically need three tables:

cost[i][j]: Stores the minimum cost for building an OBST from keys ki to kj (and their associated dummy nodes).

root[i][j]: Stores the root of the optimal BST for keys ki to kj.

weight[i][j]: Stores the sum of probabilities for keys ki to kj and dummy nodes di-1 to dj. This is helpful for calculations.

Initialization (Base Cases):

For length = 0 (only dummy nodes, no keys):

cost[i][i] = q[i] (Cost of a single dummy node).

weight[i][i] = q[i].

For length = 1 (a single key with its two dummy nodes):

cost[i][i+1] = p[i+1] + q[i] + q[i+1].

weight[i][i+1] = p[i+1] + q[i] + q[i+1].

Fill Tables Iteratively:

Iterate length from 1 to n (length of key range).

For each i (starting index) from 0 to n - length:

j = i + length (ending index).

Calculate weight[i][j] = weight[i][j-1] + p[j] + q[j].

To find cost[i][j], iterate through all possible roots r from i+1 to j:

current_cost = cost[i][r-1] + cost[r][j] + weight[i][j]

(Here, cost[i][r-1] is cost of left subtree, cost[r][j] is cost of right subtree. weight[i][j] is added because each node in the subtree increases its depth by 1 when it becomes a child of the current root r, so the total search cost increases by the sum of probabilities of all nodes in that subtree)

Choose the r that gives the minimum current_cost.

Set cost[i][j] to this minimum cost and root[i][j] to r.

Result: cost[0][n] will contain the minimum expected search cost for the entire set of keys, and root[0][n] will be the root of the optimal BST.

You can reconstruct the tree using the root table recursively.

Question 4 a) Explain the term Decision problem with the help of an example. Define the following problems and identify if they are decision problem or optimisation problem? Give reasons in support of your answer. (i) Travelling Salesman Problem (ii) Graph Colouring Problem (iii) 0 –) Knapsack Problem

Answer: A decision problem is a problem that can be posed as a yes/no question. Example: Is number x prime?

(i) Travelling Salesman Problem: Definition: Finding the shortest possible route that visits each city exactly once and returns to the origin city. Type: Optimization problem. Reason: It seeks the minimum (optimal) tour length, not a yes/no answer. (The decision version asks if there's a tour with length $\leq k$)

(ii) Graph Colouring Problem: Definition: Assigning colours to vertices of a graph such that no two adjacent vertices share the same colour, using the minimum number of colours. Type: Optimization problem. Reason: It seeks the minimum number of colours (chromatic number). (The decision version asks if the graph can be colored with $\leq k$ colors)

(iii) 0 –) Knapsack Problem: Definition: 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. Type: Optimization problem. Reason: It seeks the maximum total value that can be put into the knapsack. (The decision version asks if a total value of at least V can be achieved)

Question 4 b) What are P and NP class of Problems? Explain each class with the help of at least two examples.

Answer: P class: Problems solvable in polynomial time by a deterministic Turing machine. Examples: 1. Sorting (e.g., Bubble Sort, Merge Sort) 2. Searching in a sorted array (Binary Search) 3. Finding the shortest path in an unweighted graph (Breadth-First Search) NP class: Problems whose solutions can be verified in polynomial time by a deterministic Turing machine, or problems solvable in polynomial time by a non-deterministic Turing machine. Examples: 1. Satisfiability Problem (SAT) 2. Travelling Salesman Problem (Decision version) 3. Subset Sum Problem

Question 4 c) Define the NP-Hard and NP-Complete problem. How are they different from each other. Explain the use of polynomial time reduction with the help of an example.

Answer: NP-Hard problem: A problem H is NP-Hard if every problem in NP can be polynomial-time reduced to H. NP-Complete problem: A problem C is NP-Complete if it is in NP and is also NP-Hard. Difference: All NP-Complete problems are NP-Hard, but not all NP-Hard problems are NP-Complete (NP-Hard problems do not necessarily have to be in NP). For example, the

Halting Problem is NP-Hard but not NP-Complete because it is undecidable(not in NP). Polynomial time reduction example: Proving problem A is NP-Complete by reducing a known NP-Complete problem B to A in polynomial time. Example: To show 3-SAT is NP-Complete, we can reduce SAT to 3-SAT. Anarbitrary SAT formula can be transformed into an equivalent 3-CNF formula inpolynomial time, meaning if we can solve 3-SAT, we can solve SAT. SinceSAT is NP-Complete, and we've reduced it to 3-SAT in polynomial time, and3-SAT is in NP, then 3-SAT is NP-Complete.

Question 4 d) Define the following problems: (i) SAT Problem (ii) Clique problem (iii) Hamiltonian Cycle Problem (iv) Subset Sum Problem

Answer:

(i) SAT Problem: Given a Boolean formula in Conjunctive Normal Form (CNF), determine if there exists an assignment of truth values to its variables that makes the formula true (satisfiable).

(ii) Clique problem: Given a graph $G=(V,E)$ and an integer k, determine if G contains a clique of size at least k (a subset of k vertices where every pair of vertices is connected by an edge).

(iii) Hamiltonian Cycle Problem: Given a graph $G=(V,E)$, determine if there exists a cycle that visits each vertex exactly once and returns to the starting vertex. (iv) Subset Sum Problem: Given a set of integers $S=\{s1, s2, .., sn\}$ and a target sum T, determine if there is a non-empty subset of S whose elements sum up to T.