|  |
| --- |
| 1. Define an algorithm.   An algorithm is a step-by-step set of well-defined instructions or rules designed to solve a specific problem or accomplish a particular task. Algorithms provide a systematic way to perform computations, process data, make decisions, or perform any other operation in a clear and repeatable manner. They are fundamental to computer science and play a crucial role in various fields, including mathematics, computer programming, data analysis, and problem-solving.  What is Algorithm  **lgorithm: Calculate Sum of Two Numbers**  **Input**: Two numbers, **a** and **b**.  **Output**: The sum of **a** and **b**.  **Steps**:   1. Add **a** and **b** together. 2. The result of the addition is the sum of **a** and **b**. 3. Return the result as the output. |
| 1. List three characteristics of a good algorithm.   **What are the Characteristics of an Algorithm?**  Characteristics of an Algorithm   * **Clear and Unambiguous**: The algorithm should be unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning. * **Well-Defined Inputs**: If an algorithm says to take inputs, it should be well-defined inputs. It may or may not take input. * **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well. It should produce at least 1 output. * **Finite-ness:** The algorithm must be finite, i.e. it should terminate after a finite time. * **Feasible:** The algorithm must be simple, generic, and practical, such that it can be executed with the available resources. It must not contain some future technology or anything. * **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected. * **Input**: An algorithm has zero or more inputs. Each that contains a fundamental operator must accept zero or more inputs. * **Output**: An algorithm produces at least one output. Every instruction that contains a fundamental operator must accept zero or more inputs. * **Definiteness:** All instructions in an algorithm must be unambiguous, precise, and easy to interpret. By referring to any of the instructions in an algorithm one can clearly understand what is to be done. Every fundamental operator in instruction must be defined without any ambiguity. * **Finiteness:** An algorithm must terminate after a finite number of steps in all test cases. Every instruction which contains a fundamental operator must be terminated within a finite amount of time. Infinite loops or recursive functions without base conditions do not possess finiteness. * **Effectiveness:**An algorithm must be developed by using very basic, simple, and feasible operations so that one can trace it out by using just paper and pencil. |
| 1. Calculate the factorial of 5.   Flowchart for factorial of a number - YouTube   1. **class** FactorialExample{ 2. **public** **static** **void** main(String args[]){ 3. **int** i,fact=1; 4. **int** number=5;//It is the number to calculate factorial 5. **for**(i=1;i<=number;i++){ 6. fact=fact\*i; 7. } 8. System.out.println("Factorial of "+number+" is: "+fact); 9. } 10. }   Output:  Factorial of 5 is: 120 |
| 1. Recall the main goal of algorithm analysis.   The main goal of algorithm analysis is to evaluate and understand the performance characteristics of algorithms. It involves studying how algorithms consume computational resources, such as time and memory, as a function of the size of the input data or problem instances. The primary objectives of algorithm analysis are as follows:   1. **Efficiency Comparison**: To compare and contrast different algorithms for solving the same problem and determine which one is more efficient in terms of resource usage. This helps in selecting the best algorithm for a particular application. 2. **Predict Performance**: To predict and quantify how an algorithm will perform on various inputs or problem sizes without actually running the algorithm on those inputs. This predictive capability is essential for making informed decisions in algorithm selection. 3. **Optimization**: To identify opportunities for optimizing algorithms, making them faster or more memory-efficient. Algorithm analysis can reveal bottlenecks and areas where improvements can be made. 4. **Scalability Assessment**: To understand how an algorithm's performance scales as the size of the input data or problem instances grows. This is crucial for designing algorithms that can handle larger datasets or more complex problems. 5. **Resource Utilization**: To analyze and manage the utilization of computational resources like CPU time, memory, and storage. This is important for efficient resource allocation in computing systems. 6. **Trade-offs**: To help in making trade-offs between different aspects of algorithm performance. For example, an algorithm may be faster but require more memory, and algorithm analysis helps in making informed decisions based on these trade-offs. 7. **Real-world Application**: To apply algorithm analysis to real-world problems and systems, ensuring that algorithms meet the performance requirements of practical applications. |
| 1. Determine the time complexity of a linear search algorithm.  **Algorithm**  1. Linear\_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search 2. Step 1: set pos = -1 3. Step 2: set i = 1 4. Step 3: repeat step 4 while i **<**= n 5. Step 4: if a[i] == val 6. set pos = i 7. print pos 8. go to step 6 9. [end of if] 10. set ii = i + 1 11. [end of loop] 12. Step 5: if pos = -1 13. print "value is not present in the array " 14. [end of if] 15. Step 6: exit   The time complexity of a linear search algorithm is O(n), where "n" represents the number of elements in the input list or array that is being searched.  In a linear search, also known as a sequential search, the algorithm systematically examines each element in the list one by one until it finds the target element (if it exists) or reaches the end of the list. This means that in the worst-case scenario, where the target element is the last element in the list or is not present at all, the algorithm will have to perform n comparisons, where "n" is the number of elements in the list.  In big O notation, we express this time complexity as O(n), indicating that the time taken by the linear search algorithm grows linearly with the size of the input list. As the number of elements increases, the number of comparisons and the time taken will also increase linearly.  public class LinearSearch {  public static void main(String[] args) {  int[] arr = {4, 2, 9, 7, 1, 6};  int target = 7;  int index = linearSearch(arr, target);  if (index != -1) {  System.out.println("Element " + target + " found at index " + index);  } else {  System.out.println("Element " + target + " not found in the array.");  }  }  public static int linearSearch(int[] arr, int target) {  for (int i = 0; i < arr.length; i++) {  if (arr[i] == target) {  return i; // Found the target element, return its index  }  }  return -1; // Target element not found in the array  }  } |
| 1. Identify the key characteristics of an algorithm. |
| 1. State a red-black tree and its key properties.   A red-black tree is a type of self-balancing binary search tree data structure. It is designed to ensure that the tree remains approximately balanced during insertions and deletions, which helps maintain efficient search and retrieval operations. Red-black trees have several key properties that distinguish them:   1. **Binary Search Tree Property**: Like any binary search tree, a red-black tree satisfies the binary search tree property. This property ensures that for each node:    * All nodes in its left subtree have values less than or equal to the node's value.    * All nodes in its right subtree have values greater than the node's value. 2. **Color Property**: Each node in a red-black tree is assigned one of two colors: red or black. This color assignment follows certain rules, which are outlined in the next properties. 3. **Root Property**: The root node of the tree is always black. 4. **Red Property**: Red nodes cannot have red children. In other words, no two red nodes can be adjacent (i.e., they cannot be parent and child nodes of each other). Red nodes can have black children. 5. **Black Height Property**: The black height of a node is the number of black nodes on the path from that node to any of its descendant leaves. All paths from the root to the leaves must have the same black height. This property ensures that the tree remains balanced, preventing it from becoming highly skewed. |
| 1. Describe the concept of memoization in dynamic programming.   Memoization is a technique used in dynamic programming to optimize the performance of algorithms, especially in situations where the same subproblems are repeatedly encountered during the computation of a larger problem. It involves storing the results of expensive function calls and returning the cached result when the same inputs occur again, rather than re-computing the result. This can greatly reduce redundant calculations and improve the efficiency of algorithms.  Here are the key concepts and steps involved in memoization in dynamic programming:   1. **Recursive Structure**: Many dynamic programming problems can be solved using a recursive approach, breaking down a larger problem into smaller subproblems. However, this recursive approach often leads to redundant calculations. 2. **Caching Results**: With memoization, when a subproblem is solved, the result is stored (or "cached") in a data structure like an array, dictionary, or memoization table. The key to accessing this stored result is typically based on the parameters or inputs of the subproblem. 3. **Check Before Compute**: Before solving a subproblem, the algorithm checks whether the result is already available in the cache. If it is, the algorithm simply returns the cached result instead of recomputing it. This eliminates unnecessary calculations. 4. **Recursive Calls with Memoization**: Recursive calls in dynamic programming problems are usually modified to check the cache before making further recursive calls. If the result for a subproblem is in the cache, it's returned immediately, saving time and computational resources. 5. **Initialization**: The memoization data structure is typically initialized with initial values or placeholders to indicate that results for certain subproblems haven't been computed yet. |
| 1. Describe the concept of "asymptotic analysis" and its role in analyzing algorithm efficiency.   Asymptotic analysis is a fundamental concept in computer science and mathematics used to analyze the efficiency or performance of algorithms. It provides a way to describe the growth rate of an algorithm's resource usage (typically time and space) as the input size becomes very large. The primary role of asymptotic analysis in algorithm analysis is to characterize how an algorithm's efficiency scales with input size, helping us make informed decisions about algorithm selection and optimization.  Key components of asymptotic analysis include:   1. **Big O Notation (O)**: Big O notation is a mathematical notation used to describe the upper bound or worst-case scenario of an algorithm's time complexity or space complexity. It provides an upper limit on the resource usage as the input size approaches infinity. For example, if an algorithm has a time complexity of O(n), it means that its runtime grows linearly with the size of the input. 2. **Theta Notation (Θ)**: Theta notation defines a tight bound on an algorithm's complexity, both upper and lower bounds. It describes the average-case behavior of an algorithm. For example, if an algorithm has a time complexity of Θ(n), it means that the runtime grows linearly, and it is an accurate description of the algorithm's performance. 3. **Omega Notation (Ω)**: Omega notation defines the lower bound on an algorithm's complexity. It describes the best-case behavior of an algorithm. For example, if an algorithm has a time complexity of Ω(n), it means that the algorithm cannot perform better than linear time.   The primary role of asymptotic analysis in analyzing algorithm efficiency is as follows:   1. **Comparing Algorithms**: Asymptotic analysis allows us to compare and contrast different algorithms for solving the same problem. By examining their time or space complexities using Big O notation, we can identify which algorithm is likely to be more efficient for large inputs. For example, an algorithm with O(n) complexity may be more efficient than one with O(n^2) complexity for sufficiently large input sizes. 2. **Predicting Algorithm Behavior**: It helps us predict how an algorithm will perform as the input size grows. This predictive capability is crucial for choosing the most appropriate algorithm for a specific problem or dataset. 3. **Identifying Bottlenecks**: Asymptotic analysis can identify performance bottlenecks in algorithms, which can guide optimization efforts. For example, if an algorithm has a quadratic time complexity (O(n^2)) and is the main bottleneck in a system, optimizing it can lead to significant performance improvements. 4. **Designing Efficient Algorithms**: It assists in the design of efficient algorithms by providing insights into how to minimize resource consumption and improve overall performance. |
| 1. Distinguish between worst-case and average-case time complexity.   Worst-case time complexity and average-case time complexity are two different ways to analyze the time efficiency of an algorithm in different scenarios. They represent different aspects of an algorithm's performance:  **1. Worst-Case Time Complexity:**   * **Definition:** The worst-case time complexity of an algorithm represents the maximum amount of time an algorithm will take to complete for any input of a given size. It considers the input that leads to the most inefficient execution of the algorithm. * **Focus:** It focuses on the "worst-case scenario," which means that it provides an upper bound on the algorithm's runtime. * **Use:** Worst-case time complexity is useful for guaranteeing that an algorithm will not take more time than a certain limit, ensuring that the algorithm performs well under all circumstances. * **Notation:** It is typically denoted using Big O notation (O), which describes the upper bound of an algorithm's time complexity.   **2. Average-Case Time Complexity:**   * **Definition:** The average-case time complexity of an algorithm represents the expected or average amount of time an algorithm will take to complete for all possible inputs of a given size, taking into account the probabilities of different inputs. * **Focus:** It focuses on the "average" or "expected" performance of the algorithm when considering various inputs and their likelihood of occurring. * **Use:** Average-case time complexity provides a more realistic view of an algorithm's performance under real-world conditions. It helps assess how well an algorithm is expected to perform in practice. * **Notation:** It is often notated using Theta notation (Θ) or Omega notation (Ω), which describe tight bounds or lower and upper bounds, respectively, for the average performance. |
| 1. Explain the types of Asymptotic notations?   Asymptotic notation is used to describe the growth rate of an algorithm's resource usage (typically time or space) as the input size becomes very large. There are several types of asymptotic notations commonly used in algorithm analysis to characterize different aspects of an algorithm's performance. The three main types of asymptotic notations are:   1. **Big O Notation (O-notation):**    * **Description:** Big O notation provides an upper bound on the worst-case time or space complexity of an algorithm. It represents an upper limit on how the resource usage of the algorithm scales with input size.    * **Usage:** It is most commonly used to describe the upper bound or worst-case scenario of an algorithm's complexity. It describes the maximum amount of resources an algorithm will use.    * **Notation:** O(f(n)), where "f(n)" represents a mathematical function that bounds the algorithm's complexity. 2. **Theta Notation (Θ-notation):**    * **Description:** Theta notation provides both upper and lower bounds on the time or space complexity of an algorithm, providing a tight bound on its performance. It describes the average-case behavior.    * **Usage:** It is used to describe the average-case complexity of an algorithm when upper and lower bounds match.    * **Notation:** Θ(f(n)), where "f(n)" represents a mathematical function that tightly bounds the algorithm's complexity. 3. **Omega Notation (Ω-notation):**    * **Description:** Omega notation provides a lower bound on the best-case time or space complexity of an algorithm. It describes the lower limit of the resource usage.    * **Usage:** It is used to describe the best-case scenario or lower bound of an algorithm's complexity. It specifies the minimum amount of resources an algorithm will use.    * **Notation:** Ω(f(n)), where "f(n)" represents a mathematical function that provides a lower bound on the algorithm's complexity.   These three types of asymptotic notations are complementary and are used in different contexts to provide a comprehensive understanding of an algorithm's performance:   * **Big O notation (O)** helps us focus on the upper bound, which is essential for ensuring that an algorithm performs efficiently even in worst-case scenarios. * **Theta notation (Θ)** provides a tight, average-case bound when the upper and lower bounds match, giving us insights into the expected behavior of an algorithm. * **Omega notation (Ω)** focuses on the lower bound, highlighting the best-case scenario and helping us understand the lower limit of an algorithm's efficiency. |
| 1. Explain the concept of stability in sorting algorithms.   In the context of sorting algorithms, "stability" refers to a property where equal elements retain their relative order in the sorted output as they were in the original input. In simpler terms, when you have multiple items with the same value, a stable sorting algorithm ensures that the order of these items remains the same in the sorted result as it was in the initial unsorted list.  Here's a more detailed explanation of the concept of stability in sorting algorithms:  Imagine you have a list of objects with two properties: a numerical score and a name. You want to sort this list by the numerical score in ascending order. However, if two objects have the same score, you also want to maintain their original order based on the names. A stable sorting algorithm will do just that.  For example, consider the following list:   1. Score: 90, Name: Amit 2. Score: 85, Name: Ankit 3. Score: 90, Name: Ankur 4. Score: 80, Name: Aman   If you use a stable sorting algorithm to sort this list by score in ascending order, you would expect the result to be:   1. Score: 80, Name: Aman 2. Score: 85, Name: Ankit 3. Score: 90, Name: Amit 4. Score: 90, Name: Ankur   Notice that Amit and Ankur, who both had a score of 90, are still in the same order as they were in the original list. This is what stability in sorting means – it preserves the relative order of equal elements.  Stability can be important in various applications. For instance, in a database, if you sort records first by one field and then by another (e.g., first by age and then by name), a stable sorting algorithm ensures that the original order within each age group is preserved.  Not all sorting algorithms are stable, but some, like merge sort and insertion sort, are inherently stable, while others can be modified to be stable. Stability is a valuable characteristic when maintaining the order of equivalent elements is essential for the correctness or meaningfulness of the sorting operation. |
| 1. Describe the process of "greedy" algorithm design.   A "greedy" algorithm design is a problem-solving approach that involves making a series of locally optimal choices at each step with the hope that these choices will lead to a globally optimal solution. In other words, at each decision point, a greedy algorithm selects the best available option based on a specific criterion without considering the potential consequences of that choice on future steps. Greedy algorithms are simple and often easy to implement, but they may not always guarantee an optimal solution.  Here is a general process for designing and implementing a greedy algorithm:   1. **Problem Definition:** Clearly define the problem you want to solve using the greedy approach. Identify the input, constraints, and the objective you are trying to optimize. 2. **Select a Criterion:** Determine the criterion or rule that will guide the algorithm's decision-making at each step. This criterion should be based on the problem's characteristics and objectives. It often involves selecting the option that provides the maximum immediate benefit or minimizes a specific cost. 3. **Initialization:** Initialize any necessary variables or data structures. Set the initial state of the algorithm, including the starting point or configuration. 4. **Iterative Process:** Implement a loop or iterative process to make a sequence of decisions. At each iteration, apply the chosen criterion to select the best available option. Update the algorithm's state and any relevant variables. 5. **Termination Condition:** Define a termination condition that determines when the algorithm should stop. This condition could be reaching a specific goal, exhausting the available choices, or satisfying certain constraints. 6. **Solution Construction:** If applicable, construct the final solution based on the decisions made during the algorithm's execution. This may involve combining the selected choices to form the overall solution. 7. **Analysis:** Analyze the resulting solution to determine whether it meets the desired objectives or constraints. Evaluate the quality of the solution and assess its optimality. 8. **Optimality Considerations:** Greedy algorithms do not guarantee the globally optimal solution in all cases. To assess the algorithm's correctness and optimality, you may need to prove that the locally optimal choices lead to a globally optimal solution or that the algorithm produces an acceptable solution based on specific criteria. 9. **Iterate and Refine:** If necessary, iterate and refine the algorithm or adjust the chosen criterion to improve its performance or achieve a better solution. |
| 1. Explain how dynamic programming differs from divide and conquer.   Dynamic programming and divide and conquer are two common techniques used in algorithm design and problem-solving. While they share some similarities, they differ in their fundamental approaches and when they are most applicable.   | **Aspect** | **Dynamic Programming** | **Divide and Conquer** | | --- | --- | --- | | **Table Usage** | Utilizes a table (usually a 2D array) to store and reuse solutions to subproblems. | Does not inherently involve table usage. Subproblems are solved independently. | | **Rows and Columns** | Rows and columns in the table represent different parameters or states of the problem. | May conceptually organize subproblems in rows and columns (e.g., dividing a 2D array), but no table is maintained for storing solutions. | | **Value at Each Cell** | The value at each cell in the table represents the solution or a value related to the subproblem. | No concept of cell values as solutions; subproblems are solved independently. | | **Algorithm Flow** | Fills in the table iteratively, often row by row or column by column. Each cell's value depends on previously computed cells. | Divides the problem into smaller subproblems, often recursively. Subproblems are solved independently, and their solutions may later be combined. | | **Optimization Goal** | Focuses on optimizing time complexity by avoiding redundant calculations through solution reuse. | Typically focuses on efficient problem decomposition and independent subproblem solutions. | | **Examples** | Examples include the Fibonacci sequence, the Knapsack problem, and shortest path algorithms like Dijkstra's algorithm. | Examples include merge sort, quicksort, and problems solved using recursive binary search. | |
| 1. Describe the steps of the binary search algorithm and find the index of 19 in [1, 5, 12, 19, 27, 35, 42]. |
| 1. Explain how merge sort works with example .   **Merge Sort in Simple Terms:**  Imagine you have a deck of unsorted playing cards, and you want to arrange them in order from the smallest to the largest. Here's how merge sort works step by step:   1. **Divide:**    * You start by dividing the deck into two roughly equal halves.    * You do the same for each half, dividing them into smaller groups, until each group has only one card. These groups are considered sorted because there's only one card in each. 2. **Conquer (Merge):**    * Now, you start merging the small groups back together in pairs, comparing the cards in each pair and placing them in the correct order.    * You continue this process, merging pairs of groups into larger sorted groups. 3. **Repeat Merging:**    * You keep merging until you have one big, fully sorted group, which is your sorted deck of cards.   **Example:**  Let's say you have a deck of cards with these numbers: **[38, 27, 43, 3, 9, 82, 10]**. Using merge sort:   1. Divide:    * Split the deck into two halves: **[38, 27, 43]** and **[3, 9, 82, 10]**.    * Continue dividing until you have single cards: **[38] [27] [43] [3] [9] [82] [10]**. 2. Conquer (Merge):    * Start merging the single-card groups: Compare **38** and **27**, place them in order: **[27, 38]**. Do the same for **[3]** and **[43]** and **[10]** and **[82]**.    * Now, you have groups of two: Merge **[27, 38]** and **[3, 43]** to get **[3, 27, 38, 43]**. Merge **[9]** and **[10, 82]** to get **[9, 10, 82]**. 3. Repeat Merging:    * Finally, merge **[3, 27, 38, 43]** and **[9, 10, 82]** to get the fully sorted deck: **[3, 9, 10, 27, 38, 43, 82]**. |
| 1. Explain how a greedy algorithm works with an example of the coin change problem.   A greedy algorithm works by making the best choice at each step of a problem with the hope of finding a globally optimal solution. It follows a straightforward strategy: select the locally optimal option without considering the consequences of that choice on future steps. The key characteristics of a greedy algorithm are:   1. **Initialization:** Start with an empty solution or some initial state. 2. **Greedy Choice:** At each step, choose the best option available based on a defined criterion or rule. This choice is made without considering how it might affect future steps. 3. **Update Solution:** Update the current solution with the chosen option. 4. **Repeat:** Continue making greedy choices and updating the solution until a termination condition is met or a satisfactory solution is achieved.   **Coin Change Problem (Example of a Greedy Algorithm):**  Imagine you need to give someone 63 cents in change, and you have coins in denominations of 25 cents, 10 cents, 5 cents, and 1 cent. Here's how a greedy algorithm would work:   1. Start with 0 cents given as change and 63 cents to go. 2. Choose the largest coin denomination (25 cents) that can be given without exceeding the remaining change. Give one 25-cent coin as change. 3. Update: 25 cents given and 38 cents remaining. 4. Repeat: Choose another 25-cent coin. Now, 50 cents given and 13 cents remaining. 5. Continue: Choose a 10-cent coin. Now, 60 cents given and 3 cents remaining. 6. Keep going with 1-cent coins until the remaining change is zero.   In the end, you've given change using six coins: two 25-cent coins, one 10-cent coin, and three 1-cent coins. |
| 1. Explain how the "Master Theorem" can be used to analyze the time complexity of divide and conquer algorithms. |
| 1. Solve a basic "knapsack problem" with given weights and values. |
| 1. Show a simple "knapsack problem" with weights [2, 3, 4, 5] and values [3, 4, 5, 6], with a weight capacity of 5. |
|  |
| 1. Solve a simple insertion sort problem with input [4, 2, 2, 8, 3, 3, 1].   Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time. It works by iteratively taking each element from the input list and inserting it into its correct position in the sorted part of the array. Here's how to solve a simple insertion sort problem with the input **[4, 2, 2, 8, 3, 3, 1]**:  **Step-by-Step Insertion Sort:**   1. Start with the first element (index 0) since it's the first element of the sorted part of the array. The unsorted part contains the remaining elements.   **[4, 2, 2, 8, 3, 3, 1]**   1. Compare the second element (index 1), which is **2**, with the first element (**4**) in the sorted part.    * Since **2** is smaller than **4**, swap them:   **[2, 4, 2, 8, 3, 3, 1]**   1. Move to the next unsorted element (index 2), which is **2**, and compare it with the elements in the sorted part.    * **2** is smaller than **4**, so we swap them:   **[2, 2, 4, 8, 3, 3, 1]**   * + Now, we compare **2** with the previous element **2**, and they are the same, so no further swaps are needed.  1. Continue this process for the remaining unsorted elements:    * **8** stays in its position, as it's greater than **4**.   **[2, 2, 4, 8, 3, 3, 1]**   * + **3** is smaller than **8**, so we swap them:   **[2, 2, 4, 3, 8, 3, 1]**   * + **3** is smaller than **4**, so we swap them:   **[2, 2, 3, 4, 8, 3, 1]**   * + **3** is smaller than **2**, so we swap them:   **[2, 2, 3, 3, 4, 8, 1]**   * + Finally, **1** is smaller than **8**, so we swap them:   **[1, 2, 2, 3, 3, 4, 8]**   1. The sorting process is complete. The final sorted array is **[1, 2, 2, 3, 3, 4, 8]**. |
| 1. Implement a linear search algorithm.  **Algorithm**  1. Linear\_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search 2. Step 1: set pos = -1 3. Step 2: set i = 1 4. Step 3: repeat step 4 while i **<**= n 5. Step 4: if a[i] == val 6. set pos = i 7. print pos 8. go to step 6 9. [end of if] 10. set ii = i + 1 11. [end of loop] 12. Step 5: if pos = -1 13. print "value is not present in the array " 14. [end of if] 15. Step 6: exit   Working of Linear search  Now, let's see the working of the linear search Algorithm.  To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.  Let the elements of array are -  Linear Search Algorithm  Let the element to be searched is **K = 41**  Now, start from the first element and compare **K** with each element of the array.  Linear Search Algorithm  The value of **K,** i.e., **41,** is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.  Linear Search Algorithm |
| 1. Find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach. Consider-  n = 4  w = 5 kg  (w1, w2, w3, w4) = (2, 3, 4, 5)  (b1, b2, b3, b4) = (3, 4, 5, 6) |
| 1. Distinguish between 0-1 knapsack algorithm and fractional knapsack algorithm. |
| 1. Solve a problem using the bubble sort algorithm with values [0.,5, 1, 6, 15, 17,20].   Bubble sort is a simple 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, indicating that the list is sorted. Here's how to use bubble sort to sort the values **[0, 5, 1, 6, 15, 17, 20]** in ascending order:  **Bubble Sort Algorithm:**   1. Start by comparing the first two elements (0 and 5). 2. If the first element is greater than the second element, swap them. In this case, no swap is needed because 0 is smaller than 5. 3. Move to the next pair of elements (5 and 1). Since 5 is greater than 1, swap them.   **[0, 1, 5, 6, 15, 17, 20]**   1. Continue this process for all adjacent pairs in the list, performing swaps as necessary.    * Next, swap 5 and 6.   **[0, 1, 5, 6, 15, 17, 20]**   * + Then, swap 6 and 15.   **[0, 1, 5, 6, 15, 17, 20]**   * + Finally, swap 15 and 17.   **[0, 1, 5, 6, 15, 17, 20]**   1. After one pass through the list, the largest element (20) has "bubbled up" to the end of the list. 2. Repeat the entire process, excluding the last element (20) since it's already in its correct position. 3. Continue these passes until no swaps are needed in a full pass through the list.   **Sorted Result:**  After multiple passes, you'll find that no more swaps are needed, indicating that the list is sorted. Here's the sorted list in ascending order:  **[0, 1, 5, 6, 15, 17, 20]**  The bubble sort algorithm has sorted the values **[0, 5, 1, 6, 15, 17, 20]** in ascending order. |
|  |
| 1. The A file contains the following characters with the frequencies as shown. If Huffman Coding is used for data compression, determine-  1.Huffman Code for each character 2.Average code length 3.Length of Huffman encoded message (in bits)Characters Frequencies a 10 e 15 i 12 o 3 u 4 s 13 t 1   Huffman code for each character and calculate the average code length and length of the Huffman encoded message, we'll follow these steps:  Step 1: Build a Huffman Tree  Step 2: Assign Codes to Characters  Step 3: Calculate Average Code Length  Step 4: Calculate the Length of the Huffman Encoded Message  Let's go through each step:  Step 1: Build a Huffman Tree  To build a Huffman Tree, we start with a priority queue (min-heap) of nodes, where each node represents a character along with its frequency. We repeatedly combine the two nodes with the lowest frequencies into a new node until there is only one node left, which becomes the root of the Huffman Tree.  Here's the initial priority queue based on the provided frequencies:  Copy code  Characters Frequencies  t 1  o 3  u 4  a 10  i 12  s 13  e 15  After building the Huffman Tree, we get the following structure:  Copy code  [70]  / \  [33] [37]  / \ / \  a s [15] [22]  / \ / \  i e [10] [12]  / \  t o  Step 2: Assign Codes to Characters  We can assign binary codes to characters by traversing the Huffman Tree. Moving to the left child corresponds to adding a '0' to the code, and moving to the right child corresponds to adding a '1' to the code. The codes for the characters are as follows:  Copy code  a: 00  s: 01  i: 100  e: 101  t: 1100  o: 1101  u: 111  Step 3: Calculate Average Code Length  To calculate the average code length, we need to multiply the length of each character's code by its frequency, sum these products, and then divide by the total number of characters.  Average Code Length = (10 \* 2 + 13 \* 2 + 12 \* 3 + 15 \* 3 + 1 \* 4 + 3 \* 4 + 4 \* 3) / (10 + 13 + 12 + 15 + 1 + 3 + 4) = 407 / 58 ≈ 7 bits per character  Step 4: Calculate the Length of the Huffman Encoded Message  To calculate the length of the Huffman encoded message, we need to encode each character using its Huffman code and sum up the lengths.  Original message: "a e i i i i i i i i o o o u u u u s s s s s s s s s t"  Encoded message:  "a" is encoded as "00" (2 bits)  "e" is encoded as "101" (3 bits)  "i" is encoded as "100" (3 bits)  "o" is encoded as "1101" (4 bits)  "u" is encoded as "111" (3 bits)  "s" is encoded as "01" (2 bits)  "t" is encoded as "1100" (4 bits)  Now, sum up the lengths of these encoded characters:  2 + 3 + 3 + 4 + 3 + 2 + 4 = 21 bits  So, the length of the Huffman encoded message is 21 bits. |
| 1. Demonstrate a greedy algorithm to solve the interval scheduling problem.   The interval scheduling problem is a classic optimization problem that can be solved using a greedy algorithm. In this problem, you are given a set of tasks, each with a start time and an end time. The goal is to find the maximum number of non-overlapping tasks that can be scheduled. Here's how you can solve this problem using a greedy algorithm:  **Greedy Interval Scheduling Algorithm:**   1. Sort the tasks based on their end times in ascending order. 2. Initialize an empty list to store the selected tasks. 3. Iterate through the sorted tasks:    * If the list of selected tasks is empty, add the current task to the list.    * If the end time of the current task is compatible with the last selected task (i.e., it doesn't overlap), add the current task to the list. 4. The list of selected tasks now contains the maximum non-overlapping tasks.   Let's demonstrate this algorithm with an example:  **Example:**  Suppose you have the following tasks with their start and end times:   | **Task** | **Start Time** | **End Time** | | --- | --- | --- | | A | 1 | 3 | | B | 2 | 5 | | C | 4 | 7 | | D | 6 | 9 | | E | 8 | 10 |   **Step 1:** Sort the tasks by their end times in ascending order:   | **Task** | **Start Time** | **End Time** | | --- | --- | --- | | A | 1 | 3 | | B | 2 | 5 | | C | 4 | 7 | | D | 6 | 9 | | E | 8 | 10 |   **Step 2:** Initialize an empty list to store selected tasks.  Selected Tasks: []  **Step 3:** Iterate through the sorted tasks:   * Start with Task A (1, 3):   + Since the list is empty, add Task A to the selected tasks.   + Selected Tasks: [A] * Move to Task B (2, 5):   + Task B ends at 5, which is compatible with Task A (ends at 3), so add Task B to the selected tasks.   + Selected Tasks: [A, B] * Move to Task C (4, 7):   + Task C starts at 4 and ends at 7. It overlaps with Task B (ends at 5), so skip it. * Move to Task D (6, 9):   + Task D starts at 6 and ends at 9. It overlaps with Task B (ends at 5), so skip it. * Move to Task E (8, 10):   + Task E starts at 8 and ends at 10. It overlaps with Task B (ends at 5), so skip it.   **Step 4:** The list of selected tasks now contains the maximum non-overlapping tasks: [A, B].  So, the greedy algorithm selects Tasks A and B, which is the optimal solution for this interval scheduling problem. |
| 1. Analyze the time complexity of a dynamic programming algorithm with overlapping subproblems. End |
| 1. Analyze the time complexity of a greedy algorithm for the Huffman coding problem. |
| 1. Analyze the time complexity of bubble sort.   public class bubbleSort {  public static void bubblesort(int arr[],int n) {  for(int i=0;i<n-1;i++){  for(int j=0;j<n-1-i;j++){  if(arr[j]>arr[j+1]){  int temp=arr[j];  arr[j]=arr[j+1];  arr[j+1]=temp;  }  }  }  }  public static void main(String[] args) {  int arr[]={34,5,45,30,89,66};  int n=arr.length;  bubblesort(arr,n);  for(int a:arr){  System.out.print(a+" ");  }  }  }  Time complexity of bubble sort is =n^2; |
| 1. Analyze the time complexity of a basic recursive factorial algorithm.\   The time-complexity of recursive factorial would be:  factorial (n) {  if (n = 0)  return 1  else  return n \* factorial(n-1)  }  So,  The time complexity for one recursive call would be:  T(n) = T(n-1) + 3 (3 is for As we have to do three constant operations like  multiplication,subtraction and checking the value of n in each recursive  call)  = T(n-2) + 6 (Second recursive call)  = T(n-3) + 9 (Third recursive call)  .  .  .  .  = T(n-k) + 3k  till, k = n  Then,  = T(n-n) + 3n  = T(0) + 3n  = 1 + 3n  To represent in Big-Oh notation,  T(N) is directly proportional to n,  Therefore, The time complexity of recursive factorial is O(n). |
| 1. Analyze the time complexity of a recursive Fibonacci algorithm.   public class Fibonacci {  public static int fibonacci(int n) {  if (n <= 0) {  return 0;  } else if (n == 1) {  return 1;  } else {  return fibonacci(n - 1) + fibonacci(n - 2);  }  }  public static void main(String[] args) {  int n = 10; // You can change n to any non-negative integer.  int result = fibonacci(n);  System.out.println("Fibonacci(" + n + ") = " + result);  }  }  The time complexity for one recursive call would be:   * T(n) = T(n-1) + T(n-2) + O(1) * T(n-1) = T(n-2) + T(n-3) + O(1) * T(n-2) = T(n-3) + T(n-4) + O(1) * ... * T(2) = T(1) + T(0) + O(1) * T(1) = O(1) * T(0) = O(1)   Adding up all these equations:  T(n) = O(1) + O(1) + O(1) + ... + O(1)  The number of O(1) terms added up equals approximately the number of Fibonacci numbers between 0 and **n**, which is roughly 2^n. Therefore:  T(n) ≈ O(2^n) |
| 1. Examine the time complexity of quick sort for sorting n numbers with d digits. 2. Ans- **Recursive Steps:**    * To analyze the time complexity as we move up the recursion, let's consider the recurrence relation:      + T(n) = T(n-1) + T(n-2) + O(1)      + T(n-1) = T(n-2) + T(n-3) + O(1)      + T(n-2) = T(n-3) + T(n-4) + O(1)      + ...    * At each recursive step, we divide the array into two subarrays, and the partitioning process takes O(n) time. 3. **Summing Up:**    * Now, if we sum up all these equations, we notice that the O(1) work is consistently present in each step:      + O(1) + O(1) + O(1) + ... + O(1)    * The number of O(1) terms being added is roughly the number of recursive levels in the quicksort, which can vary depending on the choice of pivot and the input data.   So, when we add all these constant O(1) terms together, the total time complexity for quicksort can be expressed as a sum of these terms. In the average case, quicksort efficiently sorts **n** numbers with **d** digits in O(n log n) time. However, in the worst case, when the pivot selection results in highly imbalanced partitions, the time complexity can degrade to O(n^2). |
| 1. Prove that any comparison-based sorting algorithm has a best-case time complexity of O(n log n).   Ans.Think of sorting like a card game where we compare cards to put them in order. Here's why:  0.There are many ways to arrange cards (like shuffling), say a billion ways.  1.Sorting algorithms compare cards, asking questions like "Is this card bigger or smaller?"  2.We want to know the best possible number of questions (comparisons) we need.  3.Imagine playing a smart game where each question cuts possibilities in half.  4.In this smart game, you'll ask questions like dividing a deck in two equal parts.  5.If you keep dividing until you can't anymore (individual cards), you'll take about log₂(n) steps, where 'n' is the number of cards.  6."log₂(n)" means it's "logarithmically proportional to the number of cards."  7.So, the best you can do with smart comparisons is O(n log n) steps. |
| 1. Prove the correctness of quicksort algorithm's partitioning step.   Let's say we have an array [5, 2, 9, 1, 5, 6], and we choose 5 as the pivot.  1.We start with two pointers, i at the beginning and j at the end of the array:  [5, 2, 9, 1, 5, 6]  ^ ^  i j  2.We increment i until we find an element greater than the pivot:  [5, 2, 9, 1, 5, 6]  ^ ^  i j  3.We decrement j until we find an element less than or equal to the pivot:  [5, 2, 9, 1, 5, 6]  ^ ^  i j  4.We swap the elements at i and j:  [5, 2, 1, 9, 5, 6]  ^ ^  i j  5.We repeat steps 2-4 until i is greater than or equal to j.  After this process, the pivot 5 is in its correct position, and elements to the left are smaller or equal, while elements to the right are larger. This demonstrates how the partitioning step rearranges the elements correctly in a simple textual format. |
| 1. Using substitution method ,solve the recurrence relation T (n) = 2T(n/2) + n, n>1   Ans: Certainly, here's a shorter textual version of the proof using the substitution method:  \*\*Step 1: Guess the Solution\*\*  We assume that T(n) has the form T(n) = O(n log n).  \*\*Step 2: Base Case\*\*  For the base case (n = 2), we have:  T(2) = 2T(1) + 2 (where T(1) is represented as a constant C)  \*\*Step 3: Inductive Hypothesis\*\*  We assume that for all k such that 2 <= k <= n, it holds that T(k) ≤ C\*k\*log₂(k).  \*\*Step 4: Inductive Step\*\*  We want to prove that T(n+1) ≤ C\*(n+1)\*log₂(n+1).  Starting with the recurrence relation:  T(n+1) = 2T((n+1)/2) + (n+1)  Substituting our assumption:  T(n+1) ≤ 2[C\*((n+1)/2)\*log₂((n+1)/2)] + (n+1)  Simplifying the log term:  log₂((n+1)/2) = log₂(n+1) - 1  Substituting back:  T(n+1) ≤ 2C\*((n+1)/2)\*(log₂(n+1) - 1) + (n+1)  T(n+1) ≤ C\*(n+1)\*log₂(n+1) - C\*(n+1) + (n+1)  T(n+1) ≤ C\*(n+1)\*log₂(n+1) (as the last two terms are dominated by the first term)  By mathematical induction, T(n) = O(n log n) is a valid solution for T(n) = 2T(n/2) + n. |
| 1. Solve the following recurrence relation using Master’s theorem-  T(n) = 3T(n/2) + n2   Ans: To solve the recurrence relation T(n) = 3T(n/2) + n^2 using the Master Theorem, we need to compare the recurrence with the standard form of the Master Theorem:  T(n) = aT(n/b) + f(n)  In this case:  - a = 3  - b = 2  - f(n) = n^2  Now, we need to compute log\_b(a), which is log\_2(3). The Master Theorem has three cases:  \*\*Case 1:\*\* If f(n) = O(n^c) for some c < log\_b(a), then T(n) = Θ(n^(log\_b(a))).  \*\*Case 2:\*\* If f(n) = Θ(n^c) for some c = log\_b(a), then T(n) = Θ(n^c \* log n).  \*\*Case 3:\*\* If f(n) = Ω(n^c) for some c > log\_b(a), and if af(n/b) <= kf(n) for some k < 1 and sufficiently large n, then T(n) = Θ(f(n)).  In our case, log\_2(3) is between 1 and 2, so we're in \*\*Case 2\*\* since c = log\_2(3).  Now, let's compute T(n):  T(n) = Θ(n^c \* log n)  T(n) = Θ(n^(log\_2(3)) \* log n)  So, the solution to the recurrence relation T(n) = 3T(n/2) + n^2 is:  T(n) = Θ(n^(log\_2(3)) \* log n) |
| 1. Evaluate the trade-offs between different sorting algorithms based on their time and space complexity. |
| | **Aspect** | **Dynamic Programming** | **Greedy Algorithms** | | --- | --- | --- | | Approach | Careful and systematic, solving problems by breaking them into smaller parts | Simple and immediate decisions at each step | | Optimality Guarantee | Guarantees finding the best overall solution | May not guarantee the best overall solution | | Problem Types | Versatile, works for various problem types | Best suited for specific problem types | | Time Complexity | Can be slower due to thorough examination of all possibilities | Usually faster as it makes quick decisions | | Space Complexity | Can be higher due to storing intermediate results | Typically lower as it doesn't store many results | | Complexity of Implementation | Can be more complex to implement | Simpler and more straightforward | | Example Analogy | Solving a jigsaw puzzle, piece by piece | Choosing the most appealing option from a buffet | | Trade-off | Precision and optimality | Efficiency and simplicity | |  |  |  | |
| 1. Evaluate the advantages and disadvantages of using an array vs. a linked list.   Evaluating the advantages and disadvantages of using an array vs. a linked list involves considering the characteristics and trade-offs of each data structure. Here's a comparison of the two:  **Advantages of Using an Array:**   1. **Constant-Time Access:** Arrays offer constant-time access to elements by index. You can directly access any element using its index, which makes them efficient for random access. 2. **Sequential Access:** Iterating through an array sequentially is fast and cache-friendly, which can lead to better performance for certain operations like linear search. 3. **Contiguous Memory:** Array elements are stored in contiguous memory locations, which reduces memory overhead and is efficient for memory management. 4. **Predictable Memory Usage:** Arrays have a fixed size, making it easy to predict and manage memory usage.   **Disadvantages of Using an Array:**   1. **Fixed Size:** Arrays have a fixed size, which can be limiting when you need to dynamically resize or shrink the data structure. 2. **Memory Allocation:** Allocating a large array in advance may lead to wasted memory if not fully used or the need for reallocation if exceeded. 3. **Insertions and Deletions:** Insertions and deletions at arbitrary positions in an array are relatively slow because shifting elements is required.   **Advantages of Using a Linked List:**   1. **Dynamic Size:** Linked lists can dynamically resize and adapt to the amount of data, making them more flexible. 2. **Efficient Insertions/Deletions:** Insertions and deletions in a linked list are efficient because you can add or remove nodes without shifting other elements. 3. **Memory Efficiency:** Linked lists use memory more efficiently for dynamic data structures because memory is allocated as needed. 4. **No Fixed Size:** Linked lists don't have a fixed size, so you can add elements without worrying about running out of space.   **Disadvantages of Using a Linked List:**   1. **Slower Access:** Accessing elements in a linked list requires traversing from the head (or tail), which can result in slower access times compared to arrays for random access. 2. **Sequential Access:** While iterating through a linked list is efficient for forward traversal, it can be less efficient for backward traversal due to the lack of direct access to the previous element. 3. **Overhead:** Linked lists have extra memory overhead due to the need to store references (pointers) to the next element. 4. **Complexity:** Implementing and managing linked lists can be more complex due to the need to handle pointers and dynamic memory allocation. |
|  |
| 1. Evaluate the space complexity of an iterative algorithm for finding factorials.   The space complexity of an iterative algorithm for finding factorials is generally O(1), which means it uses a constant amount of extra memory that does not depend on the input size.  In an iterative factorial algorithm, you typically use a single variable (e.g., an integer) to keep track of the result and iterate from 1 to n, updating the result in each iteration. This means that regardless of the value of n, you are only using a fixed number of variables, which does not grow with the input size.  public class FactorialCalculator {  public static long iterativeFactorial(int n) {  if (n < 0) {  throw new IllegalArgumentException("Factorial is not defined for negative numbers.");  }  long result = 1; // A single variable to store the result  for (int i = 1; i <= n; i++) {  result \*= i;  }  return result;  }  public static void main(String[] args) {  int n = 5; // Example input  long factorial = iterativeFactorial(n);  System.out.println("Factorial of " + n + " is " + factorial);  }  }  When you calculate the factorial of a number iteratively, you typically use a single variable to store the result and loop through the numbers from 1 to n. The memory used for this single variable doesn't depend on the value of n, so it's considered constant memory usage, which is O(1). |
| 1. Evaluate the trade-offs between quicksort and mergesort.  * **Quicksort:** It's like sorting a deck of cards by repeatedly picking a card as a "pivot," then dividing the cards into two piles: one with cards smaller than the pivot and one with cards larger. You then sort each pile separately. It's fast but might occasionally take longer (like when cards are almost sorted). * **Mergesort:** It's like sorting a deck by dividing it into halves, sorting each half separately, and then merging them back together. It's reliable and consistent but uses more space.   **Quicksort:**   1. **Time Complexity:**    * Average-Case Time Complexity: O(n log n)    * Worst-Case Time Complexity: O(n^2), but with good pivot selection strategies, it can be mitigated. 2. **Space Complexity:**    * In-Place Sorting: Yes, Quicksort can be implemented in-place, which means it doesn't require additional memory proportional to the input size. 3. **Stability:**    * Quicksort is not stable, meaning it can change the relative order of equal elements. 4. **Adaptivity:**    * Quicksort is not adaptive, as its performance doesn't depend on the initial order of elements. 5. **Best Use Case:**    * Quicksort is efficient for large datasets and is often the preferred choice for general-purpose sorting when space is a concern. It performs well in practice and is commonly used in programming libraries and applications.   **Mergesort:**   1. **Time Complexity:**    * Average-Case Time Complexity: O(n log n)    * Worst-Case Time Complexity: O(n log n) 2. **Space Complexity:**    * Mergesort typically requires additional memory proportional to the input size, which can make it less memory-efficient for large datasets. 3. **Stability:**    * Mergesort is stable, meaning it preserves the relative order of equal elements. 4. **Adaptivity:**    * Mergesort is adaptive and performs well on partially sorted data, as it has a consistent time complexity. 5. **Best Use Case:**    * Mergesort is suitable for scenarios where stability is essential, such as sorting objects with multiple key attributes. It is also a good choice when memory usage is not a critical concern or when working with linked lists.   **Trade-offs:**   1. **Time Complexity:** Both algorithms have similar average-case time complexity, but in the worst-case scenario, Mergesort has a more predictable and consistent performance. 2. **Space Complexity:** Quicksort is generally more memory-efficient because it can be implemented in-place, while Mergesort requires additional memory for merging. 3. **Stability:** If preserving the relative order of equal elements is crucial, Mergesort is preferred because it is a stable sorting algorithm. 4. **Adaptivity:** Mergesort is adaptive and consistently performs well, regardless of the initial order of elements. Quicksort's performance can vary depending on the choice of pivot and the input data. |
| 1. Critique the limitations of the Big-O notation in analyzing algorithm efficiency.   The Big-O notation is a valuable tool for analyzing algorithm efficiency, but it has certain limitations and drawbacks. Let's critique some of the limitations of the Big-O notation:   1. **Lack of Precision:** Big-O notation provides an upper bound on an algorithm's growth rate, which means it focuses on the worst-case scenario. However, it doesn't provide precise information about an algorithm's actual performance in all cases. It can lead to overly pessimistic assessments of some algorithms. 2. **Ignores Constant Factors:** Big-O notation disregards constant factors and lower-order terms. This means two algorithms with the same Big-O complexity may have significantly different actual runtimes. For practical comparisons, constant factors can be crucial. 3. **Doesn't Consider Data Distribution:** Big-O notation doesn't take into account the distribution of data. An algorithm may perform differently on different types of input, and Big-O notation doesn't capture this variability. 4. **Ignores Hardware and Implementation Details:** The actual performance of an algorithm can vary based on hardware, compiler optimizations, and implementation details. Big-O notation abstracts away these factors, making it less accurate for predicting real-world performance. 5. **Limited to Asymptotic Analysis:** Big-O notation focuses on the behavior of algorithms as the input size approaches infinity. It's less useful for small input sizes, which are common in practice. Algorithms with better Big-O complexities may still be slower for small inputs due to their constant factors. 6. **Inaccurate for Non-Comparison-Based Algorithms:** Big-O notation is primarily designed for comparison-based algorithms. It may not accurately represent the efficiency of algorithms that don't rely on comparisons, such as hash table-based algorithms. 7. **Ignores Memory and Space Usage:** Big-O notation mainly addresses time complexity. It doesn't provide information about an algorithm's memory usage or space complexity, which can be crucial in memory-constrained environments. 8. **Overemphasizes Worst-Case Analysis:** While worst-case analysis is essential, it can lead to overly conservative assessments of an algorithm's performance. Algorithms that perform exceptionally well on average cases but have a poor worst-case scenario may be underestimated. 9. **Ignores Constants in Practical Performance:** In practice, small differences in algorithmic complexity might not be significant due to variations in input data, hardware, and real-world scenarios. Yet, Big-O notation treats these differences as significant. 10. **Not Useful for Fine-Grained Analysis:** Big-O notation is not well-suited for fine-grained performance analysis or micro-optimizations within algorithms. It doesn't provide insights into the detailed behavior of algorithms at lower levels. |
| 1. Generate the time complexity of a brute-force solution to the traveling salesman problem for a large number of cities. |
| 1. Apply master theorem to solve T (n) = 2T (n/2) + n.   Ans- To apply the Master Theorem to solve the recurrence relation T(n) = 2T(n/2) + n, we need to identify the parameters a, b, and f(n) in the standard recurrence relation form:  T(n) = aT(n/b) + f(n)  In this case:  - a = 2  - b = 2  - f(n) = n  Now, we'll compare log\_b(a) with the exponent in f(n). In our case:  - log\_2(2) = 1  Since log\_b(a) is equal to the exponent in f(n), we are in Case 2 of the Master Theorem.  Case 2 states that if log\_b(a) equals the exponent in f(n), the time complexity is:  T(n) = Θ(n^c \* log n)  In our case, c is 1 (the exponent in f(n)), so the time complexity of T(n) is:  T(n) = Θ(n \* log n)  Therefore, the solution to the recurrence relation T(n) = 2T(n/2) + n is T(n) = Θ(n \* log n). |
| 1. Design a dynamic programming algorithm to find the length of the longest common subsequence of two strings.   Ans-Algorithm to Find Length of Longest Common Subsequence (LCS):  Given two strings, str1 of length m and str2 of length n.  Create a 2D array dp of size (m+1) x (n+1).  Initialize all elements of dp to 0.  Iterate through the characters of both strings:  For each character str1[i] in str1 and each character str2[j] in str2:  If str1[i] is equal to str2[j], increment dp[i+1][j+1] by 1.  Otherwise, set dp[i+1][j+1] to the maximum of dp[i][j+1] and dp[i+1][j].  After filling the dp array, dp[m][n] will contain the length of the LCS of str1 and str2.  Return dp[m][n] as the length of the LCS.  This algorithm efficiently finds the length of the Longest Common Subsequence of two strings and can be implemented in various programming languages.t  Ans:- public class LCStringTopDown {  public static void main(String[] args) {  String x="abdes";  String y="awdes";  int ans= LCString(x,y,x.length(),y.length());  System.out.println(ans);  }  private static int LCString(String x, String y, int m, int n ) {  int t[][]=new int[m+1][n+1];  int MaxLength=0;  for(int i=0;i<m+1;i++){  for(int j=0;j<n+1;j++){  if(i==0 || j==0){  t[i][j]=0;  }  }  }  for(int i=1;i<m+1;i++){  for(int j=1;j<n+1;j++){  if(x.charAt(i-1)==y.charAt(j-1)){  t[i][j]=1+t[i-1][j-1];  }else{  t[i][j]=0;  }  MaxLength=Math.max(MaxLength, t[i][j]);  }  }  return MaxLength;  }  } |
| 1. Describe the application of a greedy algorithm to the traveling salesman problem.   \*\*Application of Greedy Algorithm to TSP:\*\*  Imagine a traveling salesperson who needs to visit multiple cities in the most efficient way possible, returning to the starting city. A greedy algorithm helps them make quick decisions:  1. Starting Point: Begin at any city (the starting point).  2. \*\*Next Stop:\*\* Choose the nearest unvisited city as the next stop.  3. \*\*Repeat:\*\* Keep selecting the closest unvisited city and move there until all cities are visited.  4. \*\*Return Home:\*\* Finally, return to the starting city, completing the tour.  \*\*Example:\*\* If you have cities A, B, C, and D, and you start at A, the greedy algorithm might lead you from A to B, then C, then D, and back to A, aiming for efficiency at each step.  \*\*Real-World Application:\*\* This approach is used in real life for tasks like optimizing delivery routes, where a driver wants to minimize travel time and distance by choosing the nearest stops one by one.  \*\*Advantages:\*\*  - Simplicity and speed make it practical for smaller-scale problems.  - Provides a reasonably good solution quickly.  \*\*Limitations:\*\*  - Doesn't guarantee the absolute best route.  - Outcome may vary depending on the starting city.  - Less effective for very complex or large-scale problems. |
| 1. Develop a novel algorithmic approach to solve a well-known problem and provide a theoretical analysis of its efficiency. |
| 1. Construct a Red Black tree by inserting 9,19,30,15,16 and 27 into an initially empty tree and also delete 15,16 and 30 from the tree.   Initial Tree (Empty)  Empty Tree  Step 2: Insert 9  9 (Black)  Step 3: Insert 19  9 (Black)  \  19 (Red)  Step 4: Insert 30  19 (Black)  / \  9 (Red) 30 (Red)  Step 5: Insert 15  19 (Black)  / \  9 (Red) 30 (Black)  /  15 (Red)  Step 6: Insert 16  19 (Black)  / \  9 (Red) 30 (Black)  / \  15 (Black)  \  16 (Red)  Step 7: Insert 27  19 (Black)  / \  9 (Red) 27 (Black)  / \  15 (Black)  \  16 (Red)  \  30 (Red)  Step 8: Delete 15  19 (Black)  / \  9 (Red) 27 (Black)  / \  16 (Black)  \  30 (Red)  Step 9: Delete 16  19 (Black)  / \  9 (Red) 27 (Black)  /  30 (Black)  Step 10: Delete 30  19 (Black)  / \  9 (Red) 27 (Red) |
| 1. Construct the binary search tree by inserting 11,6,8,19,4,10,5,7,43,49,31 and also delete the nodes 5,4,49   Step 1: Initial Tree (Empty)  Empty Tree  Step 2: Insert 11  11  Step 3: Insert 6  11  /  6  Step 4: Insert 8  11  /  6  \  8  Step 5: Insert 19  11  / \  6 19  \  8  Step 6: Insert 4  11  / \  6 19  / \  4 8  Step 7: Insert 10  11  / \  6 19  / \  4 8  \  10  Step 8: Insert 5  11  / \  6 19  / \  4 8  \  10  \  5  Step 9: Insert 7  11  / \  6 19  / \  4 8  \  10  / \  5 7  Step 10: Insert 43  11  / \  6 19  / \ \  4 8 43  \  10  / \  5 7  Step 11: Insert 49  11  / \  6 19  / \ \  4 8 43  \ \  10 49  / \  5 7  Step 12: Insert 31  11  / \  6 19  / \ \  4 8 43  \ \  10 49  / \ /  5 7 31  Step 13: Delete 5 (Leaf Node)  11  / \  6 19  / \ \  4 8 43  \ \  10 49  \ /  7 31  Step 14: Delete 4 (Leaf Node)  11  / \  6 19  \ \  8 43  \ \  10 49  \ /  7 31  Step 15: Delete 49 (Node with One Child)  11  / \  6 19  \ \  8 43  \  10  \  7  |  31 |
| 1. Construct a Red Black tree by inserting 9,19,30,15,16 and 27 into an initially empty tree and also delete 15,16 and 30 from the tree.   : Initial Tree (Empty)  Empty Tree  Step 2: Insert 9  19(B)  / \  9(R)  Step 3: Insert 19  19(B)  / \  9(R) 30(R)  Step 4: Insert 30  19(B)  / \  9(R) 30(R)  \  27(R)  Step 5: Insert 15  19(B)  / \  9(R) 30(B)  \  27(R)  /  15(R)  Step 6: Insert 16  19(B)  / \  9(R) 27(B)  \  30(R)  /  15(R)  \  16(R)  Step 7: Delete 15  19(B)  / \  9(R) 27(B)  \  30(R)  /  16(R)  Step 8: Delete 16  19(B)  / \  9(R) 27(B)  \  30(R)  Step 9: Delete 30  19(B)  / \  9(R) 27(B) |
| 1. **Develop a new hashing technique and explain its advantages over standard hashing.** |
| 1. **Develop a new sorting algorithm and analyze its time complexity.** |
| 1. Write an algorithm implement Dijkstra’s Algorithm and also analyze its complexity.   Dijkstra's Algorithm:  Dijkstra's Algorithm is used to find the shortest path from a starting vertex to all other vertices in a weighted graph.  Algorithm Steps:  Create a set to keep track of visited vertices and initialize it as empty.  Create a dictionary to store the shortest distances from the starting vertex to all other vertices. Initialize the distances with infinity except for the starting vertex, which is set to 0.  Create a priority queue (or min-heap) to store vertices ordered by their tentative distances. Initialize it with the starting vertex and its distance.  While the priority queue is not empty:  a. Extract the vertex with the smallest tentative distance from the priority queue.  b. If this vertex has already been visited, skip it.  c. Mark the vertex as visited.  d. For each neighboring vertex that hasn't been visited:  i. Calculate the tentative distance from the starting vertex to this neighbor through the extracted vertex.  ii. If the tentative distance is smaller than the current recorded distance for the neighbor, update the distance.  iii. Add the neighbor to the priority queue with its updated distance.  Continue this process until all vertices have been visited or until the destination vertex is reached.  The shortest distances from the starting vertex to all other vertices are now calculated.  Complexity Analysis:  Time Complexity: In the worst case, where all vertices and edges are considered, Dijkstra's algorithm has a time complexity of O(E + V log V), where E is the number of edges and V is the number of vertices. The use of a priority queue or min-heap for efficient extraction of minimum distances contributes to this time complexity.  Space Complexity: The space complexity is O(V) for storing the distances and O(V log V) for the priority queue. |
| 1. Define a graph and its basic components. |

A **graph** is a fundamental data structure used in computer science and mathematics to represent a collection of objects and the relationships between them. Graphs are versatile and find applications in various fields, including computer networking, soctyial networks, transportation systems, and more.

* **Vertices (Nodes):** Think of these as points or dots. Each dot represents something, like a city on a map or a person in a social network. Each dot can have a name or label.
* **Edges:** These are like the lines connecting the dots. They show how things are related. For example, if the dots represent cities, the lines can show the roads connecting them.
* **Directed vs. Undirected:** Sometimes the lines have arrows to show direction, like a one-way street. This is called a "directed" graph. Other times, there are no arrows, and the relationship is two-way, like a regular street. This is an "undirected" graph.
* **Weighted Graph:** Imagine each line has a number on it, like the distance between two cities or the cost of a flight. This is a "weighted" graph, where the numbers represent values associated with the relationships.
* **Paths:** You can trace a path from one dot to another along the lines. For example, you can find the shortest path from one city to another on a map.
* **Degrees:** Each dot can have a number associated with it, telling you how many lines connect to it. This number is called the "degree" of the dot.

|  |
| --- |
|  |
| **List three common types of graphs.** **Types of Graphs:** **1. Null Graph:** A null graph is defined as a graph which consists only the isolated vertices.  **Example:** The graph shown in fig is a null graph, and the vertices are isolated vertices.  Types of Graphs  **2. Undirected Graphs:** An Undirected graph G consists of a set of vertices, V and a set of edge E. The edge set contains the unordered pair of vertices. If (u, v)∈E then we say u and v are connected by an edge where u and v are vertices in the set V.  **Example:** Let V = {1, 2, 3, 4} and E = {(1, 2), (1, 4), (3, 4), (2, 3)}.Draw the graph.  **Solution:** The graph can be drawn in several ways.  Two of which are as follows:  Types of Graphs  **3. Multigraph:** If in a graph multiple edges between the same set of vertices are allowed, it is known as Multigraph. In other words, it is a graph having at least one loop or multiple edges.  Types of Graphs  **4. Directed Graphs:** A directed graph or digraph G is defined as an unordered pair (V, E), where V is the set of points called vertices and E is the set of edges. Each edge in the graph G is assigned a direction and is identified with an ordered pair (u, v), where u is the initial vertex, and v is the end vertex.  **Example:** Consider the graph G = (V, E) as shown in fig. Determine the vertex set and edge set of graph G.  Types of Graphs  **Solution:** The vertex and edge set of graph G =(V, E) is as follow  G={{1,2,3},{(1,2),(2,1),(2,2),(2,3),(1,3)}}.  **5. Undirected Complete Graph:** An undirected complete graph G=(V,E) of n vertices is a graph in which each vertex is connected to every other vertex i.e., and edge exist between every pair of distinct vertices. It is denoted by Kn.A complete graph with n vertices will have Types of Graphs edges.  **Example:** Draw Undirected Complete Graphs k4and k6.  **Solution:** The undirected complete graph of k4 is shown in fig1 and that of k6is shown in fig2.  Types of Graphs  .  **8. Directed Complete Graph:** A directed complete graph G = (V, E) on n vertices is a graph in which each vertex is connected to every other vertex by an arrow. It is denoted by Kn.  **Example:** Draw directed complete graphs K3 and K5.  **Solution:** Place the number of vertices at the appropriate place and then draw an arrow from each vertex to every other vertex as shown in fig:  Types of Graphs |
| **Define "dynamic programming" and its main principle.**  Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.  **Main Principle:**  The main principle of dynamic programming is often summarized as "optimal substructure" and "overlapping subproblems."   1. **Optimal Substructure:**    * This means that an optimal solution to the problem can be constructed from optimal solutions of its subproblems. In other words, if you have an optimal solution to a smaller instance of the problem, you can use it to construct an optimal solution to the larger problem. 2. **Overlapping Subproblems:**    * This implies that the problem can be broken down into subproblems which are reused several times. Rather than solving a subproblem multiple times, dynamic programming suggests solving it only once and storing the solution, so that when the same subproblem is encountered again, you can simply look up the previously computed solution. |
| **Enumerate the kind of problems can be solved by Divide and Conquer technique**  The Divide and Conquer technique is a problem-solving strategy that involves breaking a problem into smaller subproblems, solving the subproblems independently, and then combining their solutions to solve the original problem. This approach is particularly effective for problems that exhibit certain characteristics. Here are some types of problems that can be effectively solved using the Divide and Conquer technique:   1. **Sorting:**    * **Example:** Merge Sort and Quick Sort are classic sorting algorithms that use the Divide and Conquer approach. They divide the array into smaller subarrays, sort each subarray, and then merge or combine them to obtain the final sorted array. 2. **Searching:**    * **Example:** Binary Search is a classic example of a search algorithm that uses Divide and Conquer. It repeatedly divides a sorted array in half and narrows down the search space until the desired element is found or determined to be absent. 3. **Matrix Multiplication:**    * **Example:** Strassen's algorithm is a Divide and Conquer algorithm for matrix multiplication. It divides each matrix into smaller submatrices, performs recursive multiplications, and combines the results to obtain the final matrix product. 4. **Closest Pair of Points:**    * **Example:** Given a set of points in the plane, the Closest Pair of Points problem involves finding the two points that are closest to each other. The problem can be solved efficiently using a Divide and Conquer approach known as the "closest pair of points algorithm." 5. **Maximum Subarray Sum:**    * **Example:** The Maximum Subarray Sum problem involves finding the contiguous subarray with the largest sum in an array of numbers. The Divide and Conquer approach is used in algorithms like Kadane's algorithm and others to efficiently solve this problem. 6. **Convex Hull:**    * **Example:** The Convex Hull is the smallest convex polygon that encloses a set of points. Algorithms like the Graham scan and the QuickHull algorithm use Divide and Conquer to compute the Convex Hull of a set of points. 7. **Integer Multiplication:**    * **Example:** Karatsuba algorithm is an efficient algorithm for multiplying two numbers using the Divide and Conquer technique. It breaks down the multiplication into smaller multiplications and combines the results. 8. **Tree Traversal:**    * **Example:** In tree data structures, various traversal algorithms like in-order, pre-order, and post-order traversals use the Divide and Conquer strategy to process the left and right subtrees of each node separately. |
| 1. **Discuss any 2 applications of finding Minimum Spanning Tree (MST)**   Finding the Minimum Spanning Tree (MST) in a graph is a fundamental problem in computer science and network design. MSTs have numerous applications in various domains. Here are two applications:   1. **Network Design:**    * **Scenario:** Imagine you are tasked with designing a network of cities and want to connect them using a telecommunication network. The goal is to establish communication links between the cities while minimizing the cost of laying down cables or establishing connections.    * **Application of MST:** By finding the Minimum Spanning Tree of the graph representing these cities (with edges weighted by the cost of connecting each pair), you can determine the most cost-effective way to establish the network. The edges of the MST represent the optimal connections that minimize the total cost while ensuring all cities are connected. 2. **Circuit Board Wiring:**    * **Scenario:** In electronic circuit design, engineers often need to connect various components on a circuit board. The objective is to create connections between components in the most efficient and cost-effective way possible.    * **Application of MST:** The components can be represented as nodes in a graph, and the connections between them can be represented as edges with weights corresponding to the cost or distance. Finding the Minimum Spanning Tree of this graph ensures that you establish the necessary connections with the minimum total cost. This is critical in scenarios where minimizing the length of wires or the cost of manufacturing is important. |
| 1. **Explain the Divide and Conquer Algorithmic Paradigm. Also list a few algorithms which use this paradigm.**   The Divide and Conquer algorithmic paradigm is a problem-solving strategy that involves breaking down a problem into smaller subproblems of the same type, solving each subproblem independently, and then combining their solutions to obtain the solution to the original problem. The three main steps of the Divide and Conquer approach are:   1. **Divide:** Break the problem into smaller subproblems. 2. **Conquer:** Solve each subproblem independently, usually recursively. 3. **Combine:** Combine the solutions of the subproblems to obtain the solution to the original problem.   This paradigm is often applied to problems that can be naturally divided into smaller, similar subproblems and where combining the solutions of the subproblems efficiently leads to a solution for the overall problem.  **Examples of Algorithms using Divide and Conquer:**   1. **Merge Sort:**    * **Description:** Merge Sort is a sorting algorithm that uses the Divide and Conquer strategy. It divides the unsorted list into �*n* sublists, each containing one element, and then repeatedly merges sublists to produce new sorted sublists until there is only one sublist remaining—the sorted list.    * **Steps:**      + Divide: Split the unsorted list into �*n* sublists.      + Conquer: Recursively sort each sublist.      + Combine: Merge the sorted sublists to produce a single sorted list. 2. **Quick Sort:**    * **Description:** Quick Sort is another sorting algorithm that uses the Divide and Conquer paradigm. It selects a pivot element from the array and partitions the other elements into two subarrays according to whether they are less than or greater than the pivot. The subarrays are then sorted recursively.    * **Steps:**      + Divide: Choose a pivot element and partition the array.      + Conquer: Recursively sort the subarrays.      + Combine: No explicit combining step; the array is already sorted in place. 3. **Binary Search:**    * **Description:** Binary Search is a searching algorithm that efficiently finds the position of a target value within a sorted array. It repeatedly divides the search interval in half.    * **Steps:**      + Divide: Compare the target value with the middle element of the array.      + Conquer: If the target value is smaller, repeat the search in the lower half; if larger, repeat in the upper half.      + Combine: No explicit combining step; the search interval is narrowed down until the target is found. 4. **Strassen's Matrix Multiplication:**    * **Description:** Strassen's algorithm is an efficient method for matrix multiplication that uses the Divide and Conquer approach. It divides each matrix into smaller submatrices, performs recursive multiplications, and combines the results.    * **Steps:**      + Divide: Split each matrix into four submatrices.      + Conquer: Recursively compute seven products.      + Combine: Combine the products to obtain the final matrix product. |
| 1. **Examine the worst case complexity of Naïve string matching algorithm**   The Naïve String Matching algorithm is a simple and straightforward method for finding occurrences of a pattern within a text. The algorithm works by checking all possible alignments of the pattern with the text. Let's examine the worst-case complexity of the Naïve String Matching algorithm.  **Naïve String Matching Algorithm:**   * **Input:**   + *n*: Length of the text   + *m*: Length of the pattern   + *T*[1...*n*]: Text   + *P*[1...*m*]: Pattern * **Output:**   + List of indices where occurrences of the pattern start in the text.   **Algorithm Steps:**   1. Slide the pattern over the text one position at a time. 2. Compare each character of the pattern with the corresponding character of the text. 3. If a mismatch is found, slide the pattern one position to the right and continue the comparison.   **Worst-Case Complexity:** In the worst case, the Naïve String Matching algorithm may have to check all possible alignments of the pattern with the text. Suppose there is a match at the last position of the text and the pattern has to be shifted by one position each time to reach that position.   * In the first comparison, *m* character comparisons are needed. * In the second comparison, *m*−1 comparisons are needed. * In the *i*-th comparison, *m*−*i*+1 comparisons are needed.   The worst-case number of comparisons is obtained when the pattern has to be shifted*n*−*m*+1 times. Therefore, the worst-case time complexity *T*(*n*,*m*) of the Naïve String Matching algorithm can be expressed as:  *T*(*n*,*m*)=*m*+(*m*−1)+(*m*−2)+…+1+(*n*−*m*+1)  This is an arithmetic series, and the sum of the first *k* natural numbers is given by the formula *k*⋅(*k*+1)/2​. Therefore, the worst-case time complexity can be simplified to:  *T*(*n*,*m*)=*m*⋅(*m*+1)/2​+(*n*−*m*+1) |
| 1. **Analyse 0/1 and fractional Knap sack problems. Find an optimal solution to the following Knapsack problem with 7 items.   The Capacity of the Knapsack is 45. The weight and profit vectors are as follows: Weight: <12, 13, 15, 17, 11, 14, 11>  Profit: <6,26,15,51,44, 21, 22>** |
| 1. **Apply Prim’s and Kruskal’s algorithm on** |
| 1. **Explain the concept of a minimum spanning tree in a graph.**   A Minimum Spanning Tree (MST) in a graph is a tree that spans all the vertices of the graph, connecting them with the minimum possible total edge weight. In other words, an MST is a subset of the edges of the graph that forms a tree and includes every vertex, minimizing the sum of edge weights.  Key properties of a Minimum Spanning Tree:   1. **Spanning Tree:** An MST must be a spanning tree, meaning it includes all the vertices of the original graph without forming any cycles. Thus, it connects all vertices, and there are �−1*V*−1 edges in the tree, where �*V* is the number of vertices. 2. **Acyclic:** An MST is acyclic, ensuring that no cycles are formed within the tree. This is a consequence of being a spanning tree. 3. **Minimum Weight:** The sum of edge weights in an MST is minimized. Among all possible spanning trees for a graph, an MST has the smallest total edge weight.   There are multiple algorithms for finding the Minimum Spanning Tree of a graph, with Prim's algorithm and Kruskal's algorithm being two common approaches.   * **Prim's Algorithm:**   + Starts with an arbitrary vertex and grows the MST one vertex at a time by adding the minimum-weight edge that connects a vertex in the MST to a vertex outside the MST.   + Continues this process until all vertices are included in the MST. * **Kruskal's Algorithm:**   + Sorts all the edges in non-decreasing order of their weights.   + Selects the edges one by one in ascending order of weights and includes them in the MST, ensuring that the MST remains acyclic. |
| 1. **Explain the key difference between dynamic programming and greedy algorithms.**   Dynamic programming and greedy algorithms are both strategies used for solving optimization problems, but they differ in their approach to making decisions and solving subproblems. The key difference between dynamic programming and greedy algorithms   |  |  | | --- | --- | | **Dynamic Programming** | **Greedy Method** | | 1. Dynamic Programming is used to obtain the optimal solution. | 1. Greedy Method is also used to get the optimal solution. | | 2. In Dynamic Programming, we choose at each step, but the choice may depend on the solution to sub-problems. | 2. In a greedy Algorithm, we make whatever choice seems best at the moment and then solve the sub-problems arising after the choice is made. | | 3. Less efficient as compared to a greedy approach | 3. More efficient as compared to a greedy approach | | 4. Example: 0/1 Knapsack | 4. Example: Fractional Knapsack | | 5. It is guaranteed that Dynamic Programming will generate an optimal solution using Principle of Optimality. | 5. In Greedy Method, there is no such guarantee of getting Optimal Solution. | |
| 1. **Define minimum cost spanning tree. Generate minimum cost spanning tree for the following graph using Kruskal’s algorithm.**   A Minimum Cost Spanning Tree (MCST) is a tree that spans all the vertices of a graph with the minimum possible total edge weight. In other words, it is a subset of the edges of the graph that connects all vertices while minimizing the sum of edge weights.  **Kruskal's Algorithm:** Kruskal's algorithm is a greedy algorithm for finding the Minimum Cost Spanning Tree of a connected, undirected graph. The steps of Kruskal's algorithm are as follows:   1. Sort all the edges in non-decreasing order of their weights. 2. Initialize an empty graph as the result. 3. Iterate through the sorted edges. For each edge, add it to the result if adding the edge does not create a cycle in the result graph. 4. Repeat step 3 until the result graph has �−1*V*−1 edges, where �*V* is the number of vertices.   **Example:** Consider the following graph:  4  A---------B  | \ / |  | \ / |  | C |  | / \ |  | / \ |  D---------E  3  Edges and their weights: AB (4), AC (2), BC (1), BD (3), BE (5), CE (4).  **Kruskal's Algorithm Execution:**   1. Sort edges in non-decreasing order of weights: AC (2), BC (1), BD (3), AB (4), CE (4), BE (5). 2. Start with an empty graph. 3. Add edges to the graph if they do not form a cycle: AC, BC, BD, CE. 4. Stop when the graph has �−1*V*−1 edges: AC, BC, BD.   The Minimum Cost Spanning Tree is formed by the edges AC, BC, and BD.  **Resulting Minimum Cost Spanning Tree:**  4  A---------B  |  |  |  |  |  D  3  The total weight of this Minimum Cost Spanning Tree is 2+1+3=62+1+3=6. This tree connects all vertices with the minimum possible total edge weight. |
| 1. **Describe Kruskal's algorithm for finding a minimum spanning tree.**   Kruskal's algorithm is a greedy algorithm for finding the Minimum Spanning Tree (MST) of a connected, undirected graph with weighted edges. The goal of Kruskal's algorithm is to find a subset of edges that connects all vertices of the graph with the minimum possible total edge weight. The algorithm works by adding edges to the MST in ascending order of their weights, avoiding the creation of cycles.  **Steps of Kruskal's Algorithm:**   1. **Sort Edges:**    * Sort all the edges of the graph in non-decreasing order of their weights. 2. **Initialize Result:**    * Initialize an empty graph (the result) to represent the Minimum Spanning Tree. 3. **Iterate Through Edges:**    * Iterate through the sorted edges. 4. **Add Edges:**    * For each edge, check if adding it to the result would create a cycle. If adding the edge does not create a cycle, include it in the result graph. 5. **Repeat Until MST is Formed:**    * Repeat step 4 until the result graph has �−1*V*−1 edges, where �*V* is the number of vertices in the graph.   **Pseudocode:**  Kruskal(Graph G):  1. Sort edges of G in non-decreasing order of weights.  2. Initialize an empty graph (result) as the Minimum Spanning Tree.  3. For each edge (u, v) in the sorted order:  If adding (u, v) to the result does not create a cycle:  Add (u, v) to the result.   1. Return the result graph.   **Example:**  Consider the following graph:  4  A---------B  | \ / |  | \ / |  | C |  | / \ |  | / \ |  D---------E  3  Edges and their weights: AB (4), AC (2), BC (1), BD (3), BE (5), CE (4).  **Execution of Kruskal's Algorithm:**   1. Sort edges in non-decreasing order of weights: AC (2), BC (1), BD (3), AB (4), CE (4), BE (5). 2. Start with an empty graph. 3. Add edges to the graph if they do not form a cycle: AC, BC, BD. 4. Stop when the graph has �−1*V*−1 edges: AC, BC, BD.   The Minimum Spanning Tree is formed by the edges AC, BC, and BD.  Top of Form |
| 1. **Explain Dijkstra's algorithm for finding the shortest path in a graph.**   Dijkstra's algorithm is a greedy algorithm used for finding the shortest path between a source vertex and all other vertices in a weighted, directed or undirected graph with non-negative edge weights. The algorithm maintains a set of vertices whose shortest distance from the source is known and continuously updates the distances as it explores the graph.  **Steps of Dijkstra's Algorithm:**   1. **Initialization:**    * Assign a tentative distance value to every vertex. Set the distance of the source vertex to 0 and all other vertices to infinity. Maintain a priority queue (or a min-heap) to keep track of the vertices and their tentative distances. 2. **Process Vertices:**    * While there are vertices in the priority queue:      + Extract the vertex �*u* with the minimum tentative distance from the priority queue.      + For each neighbor �*v* of �*u*:        - Calculate the total distance from the source to �*v* through �*u*.        - If the calculated distance is less than the current tentative distance of �*v*, update the distance of �*v*. 3. **Continue Until All Vertices are Processed:**    * Continue this process until all vertices have been processed or the destination vertex is reached. 4. **Result:**    * The final distances stored in the vertices represent the shortest paths from the source vertex to all other vertices.   4  A------------B  |\ |  | \ |  | \ |  | \ |  | \ |  | \ |  | \ |  | \ |  | \ |  D------------C  3  Edges and their weights: AB (4), AC (3), AD (1), BC (7), BD (2), CD (5).  **Execution of Dijkstra's Algorithm:**   1. Initialization:    * distance[A] = 0, distance[B] = ∞, distance[C] = ∞, distance[D] = ∞. 2. Process Vertices:    * Start with the source vertex A.    * At each step, update the tentative distances to neighbors: AB (4), AC (3), AD (1).    * Choose vertex D (with distance 1) as the next vertex to process.    * Update the tentative distances: BD (2), CD (5).    * Choose vertex B (with distance 2) as the next vertex to process.    * Update the tentative distance: BC (7).    * Choose vertex C (with distance 5) as the next vertex to process.    * No further updates are possible. 3. Result:    * Final distances: distance[A] = 0, distance[B] = 2, distance[C] = 5, distance[D] = 1.   The shortest paths from vertex A to all other vertices are obtained based on the calculated distances. |
| 1. **Describe the "optimal substructure" property in dynamic programming.**   The "optimal substructure" property is a key concept in dynamic programming. A problem exhibits optimal substructure if an optimal solution to the overall problem can be constructed from optimal solutions to its subproblems. In other words, the problem can be broken down into smaller, overlapping subproblems, and solving these subproblems independently contributes to finding the optimal solution for the entire problem.  **Key Characteristics of Optimal Substructure:**   1. **Recursive Structure:**    * The optimal solution to the larger problem can be constructed by combining optimal solutions to smaller instances (subproblems) of the same problem. 2. **Overlapping Subproblems:**    * The problem can be broken down into subproblems, and the same subproblems are solved multiple times in the course of solving the overall problem. 3. **Memoization or Tabulation:**    * Optimal substructure allows for the use of memoization (caching previously computed solutions) or tabulation (building a table of solutions) to avoid redundant computations and speed up the algorithm.   **Example:**  Consider the Fibonacci sequence as an example to illustrate the optimal substructure property:  The Fibonacci sequence is defined as follows: �(�)=�(�−1)+�(�−2)*F*(*n*)=*F*(*n*−1)+*F*(*n*−2)  The optimal substructure property is evident here:   * The optimal solution to �(�)*F*(*n*) can be constructed from optimal solutions to �(�−1)*F*(*n*−1) and �(�−2)*F*(*n*−2).   Dynamic programming can be applied to efficiently compute Fibonacci numbers using either memoization or tabulation to avoid redundant calculations.  **Pseudocode with Memoization:**  **memo = {}**  **def fibonacci(n):**  **if n in memo:**  **return memo[n]**  **if n <= 1:**  **result = n**  **else:**  **result = fibonacci(n-1) + fibonacci(n-2)**  **memo[n] = result**  **return result**  In this example, the recursive function **fibonacci** uses memoization to store and reuse the results of subproblems, leveraging the optimal substructure property. |
| 1. **Discuss the different ways of representing a Graph in the memory of a computer? Represent the following graph using those methods.**   Graphs can be represented in the memory of a computer using various data structures. The two most common representations are the **Adjacency Matrix** and the **Adjacency List**. Here's a brief overview of each: Representations of Graph Here are the two most common ways to represent a graph :   1. Adjacency Matrix 2. Adjacency List  [Adjacency Matrix](https://www.geeksforgeeks.org/adjacency-matrix-meaning-and-definition-in-dsa/) An adjacency matrix is a way of representing a graph as a matrix of boolean (0’s and 1’s).  Let’s assume there are **n** vertices in the graph So, create a 2D matrix **adjMat[n][n]** having dimension n x n.   * *If there is an edge from vertex****i****to****j****, mark****adjMat[i][j]****as****1****.* * *If there is no edge from vertex****i****to****j****, mark****adjMat[i][j]****as****0****.*  Representation of Undirected Graph to Adjacency Matrix: The below figure shows an undirected graph. Initially, the entire Matrix is ​​initialized to **0**. If there is an edge from source to destination, we insert **1** to both cases (**adjMat[destination]**and**adjMat**[**destination])** because we can go either way.  *Undirected Graph to Adjacency Matrix* Representation of Directed Graph to Adjacency Matrix: The below figure shows a directed graph. Initially, the entire Matrix is ​​initialized to **0**. If there is an edge from source to destination, we insert **1**for that particular **adjMat[destination]**.  Directed_to_Adjacency_matrix  *Directed Graph to Adjacency Matrix* [Adjacency List](https://www.geeksforgeeks.org/adjacency-list-meaning-definition-in-dsa/) An array of Lists is used to store edges between two vertices. The size of array is equal to the number of **vertices (i.e, n)**. Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex **i**.  Let’s assume there are **n** vertices in the graph So, create an **array of list** of size **n** as **adjList[n].**   * *adjList[0] will have all the nodes which are connected (neighbour) to vertex****0****.* * *adjList[1] will have all the nodes which are connected (neighbour) to vertex****1****and so on.*  Representation of Undirected Graph to Adjacency list: The below undirected graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has two neighbours (i.e, 1 and 2). So, insert vertex 1 and 2 at indices 0 of array. Similarly, For vertex 1, it has two neighbour (i.e, 2 and 1) So, insert vertices 2 and 1 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.  Graph-Representation-of-Undirected-graph-to-Adjacency-List  *Undirected Graph to Adjacency list* Representation of Directed Graph to Adjacency list: The below directed graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has no neighbours. For vertex 1, it has two neighbour (i.e, 0 and 2) So, insert vertices 0 and 2 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.  Graph-Representation-of-Directed-graph-to-Adjacency-List  *Directed Graph to Adjacency list* |
| 1. **Solve the following Activity-Selection problem using Greedy Technique. (The starting and finishing times of 11 activities are given as follows: (12,23), (18,22), (22,24), (13,15), (10,16), (11,14), (16,20), (15,17), (13,18), (15,19), (18,21)** |
| 1. **Define P, NP, NP-Hard and NP-Complete Problem.**   **P (Polynomial Time):** P is the class of decision problems (problems with a yes/no answer) for which an algorithm exists that can determine the answer in polynomial time. Polynomial time means that the running time of the algorithm is a polynomial function of the input size. In simpler terms, if a problem is in P, it is considered efficiently solvable.  **NP (Non-deterministic Polynomial Time):** NP is the class of decision problems for which a given solution can be verified quickly (in polynomial time), but finding a solution is not necessarily quick. If a solution is proposed, it can be checked in polynomial time. The term "non-deterministic" refers to the fact that, in a theoretical non-deterministic computer, a solution could be guessed and checked in polynomial time.  **NP-Hard (Non-deterministic Polynomial Time Hard):** NP-Hard is a class of problems that are at least as hard as the hardest problems in NP. In other words, an NP-Hard problem is a problem for which a polynomial-time algorithm to solve it would also solve any problem in NP in polynomial time. However, NP-Hard problems themselves may not be in NP; they are just as hard as the hardest problems in NP.  **NP-Complete (Non-deterministic Polynomial Time Complete):** NP-Complete is a class of problems that are both in NP and NP-Hard. These problems are considered to be the most difficult problems in NP in the sense that if you could solve one of them efficiently (in polynomial time), you could solve all problems in NP efficiently. Finding a polynomial-time algorithm for any NP-Complete problem would imply a polynomial-time algorithm for all problems in NP. |
| 1. **Create a Red Black tree given input 2, 1, 4, 5, 9, 3, 6, 7.**  Insert 2, 1, 4, 5, 9, 3, 6, 7 Below is a trace of inserting the following values into an initially empty red/black tree... |
| 1. **Implement and demonstrate the runtime behavior of an algorithm with different input sizes to observe its growth rate.** |
| 1. **Solve a basic problem of finding the shortest path between two nodes using Dijkstra's algorithm.**   Dijkstra's algorithm is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge weights. The algorithm works by iteratively selecting the vertex with the smallest tentative distance from the source and updating the distances of its neighbors.  Here is a mathematical representation of Dijkstra's algorithm:  Given a graph �*G* with vertices �*V* and edges �*E*, and a source vertex �*s*, the algorithm maintains a set of vertices �*S* whose final shortest distance from the source is known. Initially, �*S* is empty.   1. **Initialization:**    * Assign to every vertex a tentative distance value. Set the distance to 0 for the source vertex and to ∞∞ for all other vertices.    * Set the initial node as the current node. 2. **Iterate:**    * For the current node, consider all of its neighbors and calculate their tentative distances through the current node. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.    * Mark the current node as "visited" (add it to �*S*).    * Select the unvisited node with the smallest tentative distance as the next "current node" and go back to step 2. 3. **Termination:**    * The algorithm terminates when all nodes have been visited (added to �*S*) or when the smallest tentative distance among the unvisited nodes is ∞∞.   Now, let's illustrate this with a simplified example:  Consider the graph:  cssCopy code  A --3-- B | /| | / | 2 1 4 | / | | / | C --5-- D  Assume we start from node A and want to find the shortest paths to all other nodes.   1. Initialization:    * �(�)=0*d*(*A*)=0, �(�)=∞*d*(*B*)=∞, �(�)=∞*d*(*C*)=∞, �(�)=∞*d*(*D*)=∞.    * Current node: A. 2. Iteration:    * Consider neighbors of A: B and C.      + �(�)=min⁡(�(�),�(�)+�(�,�))=min⁡(∞,0+3)=3*d*(*B*)=min(*d*(*B*),*d*(*A*)+*w*(*A*,*B*))=min(∞,0+3)=3      + �(�)=min⁡(�(�),�(�)+�(�,�))=min⁡(∞,0+2)=2*d*(*C*)=min(*d*(*C*),*d*(*A*)+*w*(*A*,*C*))=min(∞,0+2)=2    * Mark A as visited.    * Select the unvisited node with the smallest tentative distance (C) as the current node. 3. Iteration:    * Consider neighbors of C: A, B, D.      + �(�)=min⁡(�(�),�(�)+�(�,�))=min⁡(0,2+2)=0*d*(*A*)=min(*d*(*A*),*d*(*C*)+*w*(*C*,*A*))=min(0,2+2)=0      + �(�)=min⁡(�(�),�(�)+�(�,�))=min⁡(3,2+5)=3*d*(*B*)=min(*d*(*B*),*d*(*C*)+*w*(*C*,*B*))=min(3,2+5)=3      + �(�)=min⁡(�(�),�(�)+�(�,�))=min⁡(∞,2+5)=7*d*(*D*)=min(*d*(*D*),*d*(*C*)+*w*(*C*,*D*))=min(∞,2+5)=7    * Mark C as visited.    * Select the unvisited node with the smallest tentative distance (B) as the current node. 4. Iteration:    * Consider neighbors of B: A, C, D.      + �(�)=min⁡(�(�),�(�)+�(�,�))=min⁡(0,3+3)=0*d*(*A*)=min(*d*(*A*),*d*(*B*)+*w*(*B*,*A*))=min(0,3+3)=0      + �(�)=min⁡(�(�),�(�)+�(�,�))=min⁡(2,3+1)=2*d*(*C*)=min(*d*(*C*),*d*(*B*)+*w*(*B*,*C*))=min(2,3+1)=2      + �(�)=min⁡(�(�),�(�)+�(�,�))=min⁡(7,3+4)=7*d*(*D*)=min(*d*(*D*),*d*(*B*)+*w*(*B*,*D*))=min(7,3+4)=7    * Mark B as visited.    * Select the unvisited node with the smallest tentative distance (C) as the current node. 5. Iteration:    * Consider neighbors of C: A, D.      + �(�)=0*d*(*A*)=0 (unchanged)      + �(�)=7*d*(*D*)=7 (unchanged)    * Mark C as visited.    * Select the unvisited node with the smallest tentative distance (D) as the current node. 6. Iteration:    * Consider neighbors of D: B, C.      + �(�)=3*d*(*B*)=3 (unchanged)      + �(�)=2*d*(*C*)=2 (unchanged)    * Mark D as visited.    * All nodes have been visited, terminate.   The final shortest path distances are �(�)=0*d*(*A*)=0, �(�)=3*d*(*B*)=3, �(�)=2*d*(*C*)=2, �(�)=7*d*(*D*)=7. The shortest path from A to D is A -> C -> B -> D with a total distance of 7. |
| 1. **Implement the Prim's algorithm for finding a minimum spanning tree.**  **Prim's Algorithm** In this article, we will discuss the prim's algorithm. Along with the algorithm, we will also see the complexity, working, example, and implementation of prim's algorithm.  Before starting the main topic, we should discuss the basic and important terms such as spanning tree and minimum spanning tree.  **Spanning tree -** A spanning tree is the subgraph of an undirected connected graph.  **Minimum Spanning tree -** Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.  Backward Skip 10sPlay VideoForward Skip 10s  Now, let's start the main topic.  **Prim's Algorithm** is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.  Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected. How does the prim's algorithm work? Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -   * First, we have to initialize an MST with the randomly chosen vertex. * Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree. * Repeat step 2 until the minimum spanning tree is formed.   The applications of prim's algorithm are -   * Prim's algorithm can be used in network designing. * It can be used to make network cycles. * It can also be used to lay down electrical wiring cables.  Example of prim's algorithm Now, let's see the working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.  Suppose, a weighted graph is -  Prim's Algorithm  **Step 1 -** First, we have to choose a vertex from the above graph. Let's choose B.  Prim's Algorithm  **Step 2 -** Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.  Prim's Algorithm  **Step 3 -** Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.  Prim's Algorithm  **Step 4 -** Now, select the edge CD, and add it to the MST.  Prim's Algorithm  **Step 5 -** Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.  Prim's Algorithm  So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -  Cost of MST = 4 + 2 + 1 + 3 = 10 units. Algorithm  1. Step 1: Select a starting vertex 2. Step 2: Repeat Steps 3 and 4 until there are fringe vertices 3. Step 3: Select an edge 'e' connecting the tree vertex and fringe vertex that has minimum weight 4. Step 4: Add the selected edge and the vertex to the minimum spanning tree T 5. [END OF LOOP] 6. Step 5: EXIT  Complexity of Prim's algorithm Now, let's see the time complexity of Prim's algorithm. The running time of the prim's algorithm depends upon using the data structure for the graph and the ordering of edges. Below table shows some choices -   * **Time Complexity**  |  |  | | --- | --- | | **Data structure used for the minimum edge weight** | **Time Complexity** | | Adjacency matrix, linear searching | O(|V|2) | | Adjacency list and binary heap | O(|E| log |V|) | | Adjacency list and Fibonacci heap | O(|E|+ |V| log |V|) | |
| 1. **Solve a problem of finding the shortest path between two nodes using the Bellman-Ford algorithm.**   Consider the following directed graph:  javascript  Copy code  A --2--> B --4--> C  ^ / | /  | / | /  -3 1 2 -1  | / | /  D --5---> E  In this graph:  �  ,  �  ,  �  ,  �  ,  �  A,B,C,D,E are vertices.  (  �  ,  �  ,  2  )  ,  (  �  ,  �  ,  −  3  )  ,  (  �  ,  �  ,  4  )  ,  (  �  ,  �  ,  2  )  ,  (  �  ,  �  ,  5  )  ,  (  �  ,  �  ,  −  1  )  ,  (  �  ,  �  ,  −  1  )  (A,B,2),(A,D,−3),(B,C,4),(B,E,2),(D,B,5),(D,E,−1),(E,C,−1) are directed edges with weights.  Let's apply the Bellman-Ford algorithm to find the shortest paths from vertex A.  Initialization:  Set  �  [  �  ]  =  0  d[A]=0 and  �  [  �  ]  =  ∞  d[v]=∞ for all other vertices.  �  [  �  ]  =  0  �  [  �  ]  =  ∞  �  [  �  ]  =  ∞  �  [  �  ]  =  ∞  �  [  �  ]  =  ∞  ​  d[A]=0  d[B]=∞  d[C]=∞  d[D]=∞  d[E]=∞  ​  Relaxation:  Iterate through all edges  ∣  �  ∣  −  1  ∣V∣−1 times.  Iteration 1:  Relax  (  �  ,  �  ,  2  )  (A,B,2):  �  [  �  ]  =  min  ⁡  (  ∞  ,  0  +  2  )  =  2  d[B]=min(∞,0+2)=2  Relax  (  �  ,  �  ,  −  3  )  (A,D,−3):  �  [  �  ]  =  min  ⁡  (  ∞  ,  0  −  3  )  =  −  3  d[D]=min(∞,0−3)=−3  �  [  �  ]  =  0  �  [  �  ]  =  2  �  [  �  ]  =  ∞  �  [  �  ]  =  −  3  �  [  �  ]  =  ∞  ​  d[A]=0  d[B]=2  d[C]=∞  d[D]=−3  d[E]=∞  ​  Iteration 2:  Relax  (  �  ,  �  ,  2  )  (A,B,2): No change (  �  [  �  ]  =  2  d[B]=2)  Relax  (  �  ,  �  ,  −  3  )  (A,D,−3): No change (  �  [  �  ]  =  −  3  d[D]=−3)  �  [  �  ]  =  0  �  [  �  ]  =  2  �  [  �  ]  =  ∞  �  [  �  ]  =  −  3  �  [  �  ]  =  ∞  ​  d[A]=0  d[B]=2  d[C]=∞  d[D]=−3  d[E]=∞  ​  Iteration 3:  Relax  (  �  ,  �  ,  2  )  (A,B,2): No change (  �  [  �  ]  =  2  d[B]=2)  Relax  (  �  ,  �  ,  −  3  )  (A,D,−3): No change (  �  [  �  ]  =  −  3  d[D]=−3)  �  [  �  ]  =  0  �  [  �  ]  =  2  �  [  �  ]  =  ∞  �  [  �  ]  =  −  3  �  [  �  ]  =  ∞  ​  d[A]=0  d[B]=2  d[C]=∞  d[D]=−3  d[E]=∞  ​  Final Distances:  After 3 iterations, no further changes occur.  The final shortest distances from A are:  �  [  �  ]  =  0  �  [  �  ]  =  2  �  [  �  ]  =  ∞  �  [  �  ]  =  −  3  �  [  �  ]  =  ∞  ​  d[A]=0  d[B]=2  d[C]=∞  d[D]=−3  d[E]=∞  ​  The shortest paths from A are:  �  →  �  A→B with weight 2.  �  →  �  A→D with weight -3.  It's important to note that if there are negative cycles in the graph, the Bellman-Ford algorithm will detect them. In this example, there is no negative cycle reachable from A. |
| 1. **Implement a dynamic programming algorithm to find the longest common subsequence of two strings.**  **Longest Common Subsequence** Here longest means that the subsequence should be the biggest one. The common means that some of the characters are common between the two strings. The subsequence means that some of the characters are taken from the string that is written in increasing order to form a subsequence.  **Let's understand the subsequence through an example.**  Suppose we have a string 'w'.  **W1 = abcd**  **The following are the subsequences that can be created from the above string:**   * ab * bd * ac * ad * acd * bcd   The above are the subsequences as all the characters in a sub-string are written in increasing order with respect to their position. If we write ca or da then it would be a wrong subsequence as characters are not appearing in the increasing order. The total number of subsequences that would be possible is 2n, where n is the number of characters in a string. In the above string, the value of 'n' is 4 so the total number of subsequences would be 16.  **W2= bcd**  By simply looking at both the strings w1 and w2, we can say that bcd is the longest common subsequence. If the strings are long, then it won't be possible to find the subsequence of both the string and compare them to find the longest common subsequence.  **Finding LCS using dynamic programming with the help of a table.**  **Consider two strings:**  X= a b a a b a  Y= b a b b a b  **(a, b)**  **For index i=1, j=1**  Longest Common Subsequence  Since both the characters are different so we consider the maximum value. Both contain the same value, i.e., 0 so put 0 in (a,b). Suppose we are taking the 0 value from 'X' string, so we put arrow towards 'a' as shown in the above table.  **(a, a)**  **For index i=1, j=2**  Longest Common Subsequence  Both the characters are the same, so the value would be calculated by adding 1 and upper diagonal value. Here, upper diagonal value is 0, so the value of this entry would be (1+0) equal to 1. Here, we are considering the upper diagonal value, so the arrow will point diagonally.  **(a, b)**  **For index i=1, j=3**  Longest Common Subsequence  Since both the characters are different so we consider the maximum value. The character 'a' has the maximum value, i.e., 1. The new entry, i.e., (a, b) will contain the value 1 pointing to the 1 value.  **(a, b)**  **For index i=1, j=4**  Longest Common Subsequence  Since both the characters are different so we consider the maximum value. The character 'a' has the maximum value, i.e., 1. The new entry, i.e., (a, b) will contain the value 1 pointing to the 1 value.  **(a, a)**  **For index i=1, j=5**  Longest Common Subsequence  Both the characters are same so the value would be calculated by adding 1 and upper diagonal value. Here, upper diagonal value is 0 so the value of this entry would be (1+0) equal to 1. Here, we are considering the upper diagonal value so arrow will point diagonally.  **(a, b)**  **For index i=1, j=6**  Longest Common Subsequence  Since both the characters are different so we consider the maximum value. The character 'a' has the maximum value, i.e., 1. The new entry, i.e., (a, b) will contain the value 1 pointing to the 1 value.  **(b, b)**  **For index i=2, j=1**  Longest Common Subsequence  Both the characters are same so the value would be calculated by adding 1 and upper diagonal value. Here, upper diagonal value is 0 so the value of this entry would be (1+0) equal to 1. Here, we are considering the upper diagonal value so arrow will point diagonally.  **(b, a)**  **For index i=2, j=2**  Longest Common Subsequence  Since both the characters are different so we consider the maximum value. The character 'a' has the maximum value, i.e., 1. The new entry, i.e., (a, b) will contain the value 1 pointing to the 1 value.  In this way, we will find the complete table. The final table would be:  Longest Common Subsequence  In the above table, we can observe that all the entries are filled. Now we are at the last cell having 4 value. This cell moves at the left which contains 4 value.; therefore, the first character of the LCS is 'a'.  The left cell moves upwards diagonally whose value is 3; therefore, the next character is 'b' and it becomes 'ba'. Now the cell has 2 value that moves on the left. The next cell also has 2 value which is moving upwards; therefore, the next character is 'a' and it becomes 'aba'.  The next cell is having a value 1 that moves upwards. Now we reach the cell (b, b) having value which is moving diagonally upwards; therefore, the next character is 'b'. The final string of longest common subsequence is 'baba'.  **Why a dynamic programming approach in solving a LCS problem is more efficient than the recursive algorithm?**  If we use the dynamic programming approach, then the number of function calls are reduced. The dynamic programming approach stores the result of each function call so that the result of function calls can be used in the future function calls without the need of calling the functions again.  In the above dynamic algorithm, the results obtained from the comparison between the elements of x and the elements of y are stored in the table so that the results can be stored for the future computations.  The time taken by the dynamic programming approach to complete a table is O(mn) and the time taken by the recursive algorithm is 2max(m, n). |
| 1. **Create a Red Black tree given the following inputs a) 2, 1, 4, 5, 9, 3, 6, 7and b) 8,18,5,15,17,25,40,80.** |
| 1. **Compare the various programming paradigms such as divide-and-conquer, dynamic programming and greedy approach.**   Programming paradigms, including divide-and-conquer, dynamic programming, and greedy approach, are problem-solving strategies that guide the structure and organization of algorithms. Let's compare these paradigms based on their key characteristics: 1. **Divide-and-Conquer:**  * **Basic Idea:** Break down a problem into smaller, manageable subproblems, solve them independently, and then combine their solutions to solve the original problem. * **Characteristics:**   + **Divide:** Break the problem into subproblems.   + **Conquer:** Solve subproblems recursively.   + **Combine:** Merge solutions of subproblems to obtain the solution for the original problem. * **Examples:**   + Merge Sort   + Quick Sort   + Binary Search * **Pros:**   + Well-suited for problems that can be naturally divided into subproblems.   + Parallelization is often possible. * **Cons:**   + Overhead from recursive calls and combining solutions.  2. **Dynamic Programming:**  * **Basic Idea:** Solve a problem by breaking it down into overlapping subproblems and solving each subproblem only once, storing the solutions in a table to avoid redundant computations. * **Characteristics:**   + **Overlapping Subproblems:** The same subproblem is solved multiple times.   + **Optimal Substructure:** An optimal solution to the problem can be constructed from optimal solutions of its subproblems. * **Examples:**   + Longest Common Subsequence   + Knapsack Problem   + Fibonacci (with memoization) * **Pros:**   + Avoids redundant computations by storing intermediate results.   + Efficiently solves problems with optimal substructure and overlapping subproblems. * **Cons:**   + Requires additional space for memoization or tabulation.  3. **Greedy Approach:**  * **Basic Idea:** Make locally optimal choices at each stage with the hope that these choices will lead to a globally optimal solution. * **Characteristics:**   + **Greedy Choice Property:** A global optimum can be arrived at by selecting a local optimum.   + **Optimal Substructure:** An optimal solution to the problem contains optimal solutions to its subproblems. * **Examples:**   + Dijkstra's Algorithm   + Kruskal's Algorithm   + Huffman Coding * **Pros:**   + Simplicity and efficiency for certain types of problems.   + Often easier to implement. * **Cons:**   + Greedy algorithms may not always yield globally optimal solutions.   + Careful consideration of correctness is necessary.  **Comparison:**  * **Optimality:**   + **Divide-and-Conquer:** May or may not lead to optimal solutions, depending on the problem.   + **Dynamic Programming:** Generally leads to optimal solutions if the problem exhibits optimal substructure.   + **Greedy Approach:** May or may not lead to optimal solutions, and correctness needs to be carefully established. * **Space Complexity:**   + **Divide-and-Conquer:** Often requires less space compared to dynamic programming.   + **Dynamic Programming:** May require additional space for memoization or tabulation.   + **Greedy Approach:** Typically requires minimal space. * **Applicability:**   + **Divide-and-Conquer:** Suitable for problems that can be divided into independent subproblems.   + **Dynamic Programming:** Suitable for problems with optimal substructure and overlapping subproblems.   + **Greedy Approach:** Suitable for problems where a locally optimal choice leads to a globally optimal solution. |
| 1. **Determine the LCS of < H,B,C,F,G,M,N,A,P,Q> and < C,B,H,G,R,S,F,N,M,Q>** |
| 1. **Analyze the time complexity of the dynamic programming solution to the longest common subsequence problem.**   The time complexity of the dynamic programming solution to the Longest Common Subsequence (LCS) problem can be analyzed by considering the number of subproblems and the time required to solve each subproblem.  Let's denote the lengths of the input sequences as �*m* and �*n*, where �*m* is the length of the first sequence and �*n* is the length of the second sequence. Time Complexity Analysis:  1. **Filling the Table:**    * The dynamic programming table �*c* is of size (�+1)×(�+1)(*m*+1)×(*n*+1).    * Each entry in the table requires constant time to compute based on the recurrence relation.    * Filling the entire table involves computing (�+1)×(�+1)(*m*+1)×(*n*+1) entries. 2. **Overall Time Complexity:**    * The time complexity of filling the dynamic programming table is �(�⋅�)*O*(*m*⋅*n*).  Explanation: The dynamic programming solution uses a bottom-up approach to fill the table by iteratively solving subproblems. The table is of size (�+1)×(�+1)(*m*+1)×(*n*+1) because it includes entries for the empty sequences (when �=0*i*=0 or �=0*j*=0).  For each entry �[�,�]*c*[*i*,*j*], the algorithm takes constant time to compute based on the recurrence relation. The overall time complexity is dominated by the number of entries in the table, which is �(�⋅�)*O*(*m*⋅*n*). |
| 1. **Shortest path does not exist for a graph with negative weight edge cycle. Illustrate with an example.**   In a graph with negative weight edge cycles, the concept of a "shortest path" becomes ambiguous. This is because you can keep traveling around the negative weight cycle indefinitely, continuously reducing the cost of the path. As a result, there is no well-defined shortest path in such a graph.  Let's illustrate this with a simple example:  Consider the following directed graph with a negative weight cycle:  A --2--> B  ^ / |  | / |  -3 1 -2  | / |  D --5---> C  In this graph:   * Vertices: A, B, C, D * Directed edges with weights: (A, B, 2), (A, D, -3), (B, C, 1), (B, D, 5), (C, D, -2), (D, A, -2)   Now, let's try to find the shortest path from A to C using Dijkstra's algorithm:   1. Start with A, initialize distances: �:0,�:∞,�:∞,�:∞*A*:0,*B*:∞,*C*:∞,*D*:∞. 2. Relax edges: �→�*A*→*B* (distance to B becomes 2), �→�*A*→*D* (distance to D becomes -3). 3. Relax edges: �→�*D*→*A* (distance to A becomes -5), �→�*B*→*C* (distance to C becomes -3), �→�*B*→*D* (distance to D becomes -1). 4. Relax edges: �→�*D*→*A* (distance to A becomes -3), �→�*D*→*C* (distance to C becomes -5).   Now, if you follow the path A -> D -> C -> D -> A -> ..., you can keep going around the negative weight cycle, reducing the cost indefinitely. There is no well-defined shortest path from A to C because you can always find a shorter path by going around the cycle one more time.  In such cases, algorithms like Dijkstra's algorithm, which assume that the shortest path is well-defined, may not terminate or provide meaningful results. If a graph contains a negative weight cycle, it's common to use algorithms like the Bellman-Ford algorithm, which can detect the presence of negative cycles and handle them appropriately |
| 1. **Analyze the time complexity of BFS in an unweighted graph.**   The time complexity of Breadth-First Search (BFS) in an unweighted graph is typically analyzed in terms of the number of vertices (�*V*) and edges (�*E*). Time Complexity Analysis:  1. **Initialization:**    * Initializing data structures (e.g., queues, visited array): �(�)*O*(*V*) 2. **Exploration of Vertices:**    * Each vertex is visited at most once.    * Visiting a vertex involves enqueuing it and marking it as visited.    * In an unweighted graph, each vertex has the same number of neighbors (constant).    * Overall, visiting all vertices takes �(�)*O*(*V*) time. 3. **Exploration of Edges:**    * In an unweighted graph, the number of edges explored from each vertex is proportional to the degree of the vertex (number of neighbors).    * Exploring all edges takes �(�)*O*(*E*) time.  Overall Time Complexity: The overall time complexity of BFS in an unweighted graph is �(�+�)*O*(*V*+*E*). Explanation: In an unweighted graph, each edge has the same weight, and BFS explores the graph level by level. The queue used in BFS ensures that vertices are visited in the order of their distance from the source vertex.  The time complexity is dominated by the exploration of vertices and edges. Visiting each vertex takes constant time, and exploring all edges takes �(�)*O*(*E*) time. Therefore, the overall time complexity is �(�+�)*O*(*V*+*E*).  It's important to note that if the graph is represented using an adjacency matrix, the time complexity for exploring edges can be �(�2)*O*(*V*2) in the worst case. However, if the graph is represented using an adjacency list, the time complexity for exploring edges is �(�)*O*(*E*), making the algorithm more efficient. In practice, BFS is often very efficient for exploring unweighted graphs. |
| 1. **Analyze the time complexity of Kruskal's algorithm in terms of sorting the edges.**   Kruskal's algorithm is a greedy algorithm used to find the minimum spanning tree of a connected, undirected graph. The time complexity of Kruskal's algorithm is dominated by the sorting of the edges based on their weights. Time Complexity Analysis:  1. **Sorting Edges:**    * The main time-consuming operation in Kruskal's algorithm is sorting the edges based on their weights.    * If the graph has �*V* vertices and �*E* edges, the time complexity of sorting �*E* edges is typically �(�log⁡�)*O*(*E*log*E*) using efficient sorting algorithms like quicksort or mergesort.    * In the worst case, sorting dominates the time complexity. 2. **Union-Find Operations:**    * After sorting, Kruskal's algorithm performs union-find operations to determine whether adding an edge creates a cycle in the current spanning forest.    * In the worst case, there can be �(�)*O*(*V*) union-find operations (each vertex starts in its own disjoint set, and at most �−1*V*−1 edges can be added to create the minimum spanning tree).    * Union-find operations typically take nearly constant time per operation with path compression and union by rank.  Overall Time Complexity: The overall time complexity of Kruskal's algorithm, dominated by the sorting step, is �(�log⁡�)*O*(*E*log*E*). Explanation:  * The sorting step is the most time-consuming operation in Kruskal's algorithm. * Sorting �*E* edges takes �(�log⁡�)*O*(*E*log*E*) time using efficient sorting algorithms. * The union-find operations after sorting contribute �(�)*O*(*V*) time in the worst case.   The efficiency of Kruskal's algorithm depends on the efficiency of the sorting algorithm used. The �(�log⁡�)*O*(*E*log*E*) time complexity makes Kruskal's algorithm particularly suitable for sparse graphs, where �*E* is close to �*V* or smaller. If the graph is dense, and �*E* is close to �2*V*2, then other algorithms like Prim's algorithm (which has a �(�2)*O*(*V*2) time complexity using efficient data structures) might be more efficient. |
| 1. **Analyze the time complexity of Dijkstra's algorithm using a priority queue.**   Dijkstra's algorithm is a greedy algorithm used for finding the shortest paths from a single source vertex to all other vertices in a weighted graph. When implemented using a priority queue to efficiently extract the minimum distance vertex, the time complexity is influenced by the data structures and operations involved. Time Complexity Analysis:  1. **Initialization:**    * Initializing distances and priority queue: �(�)*O*(*V*) time, where �*V* is the number of vertices. 2. **Priority Queue Operations:**    * Dijkstra's algorithm repeatedly extracts the vertex with the minimum distance from the priority queue.    * If a binary heap is used as the priority queue, the time complexity for extracting the minimum element is �(log⁡�)*O*(log*V*).    * If a Fibonacci heap is used (which allows decrease-key operations in constant time), the time complexity can be reduced to �(1)*O*(1) for decrease-key and �(log⁡�)*O*(log*V*) for extracting the minimum.    * The total time spent on priority queue operations is �((�+�)log⁡�)*O*((*E*+*V*)log*V*) in the worst case, where �*E* is the number of edges. 3. **Relaxation:**    * Each edge is relaxed at most once.    * If adjacency lists are used, the total time for relaxation is �(�)*O*(*E*).  Overall Time Complexity: The overall time complexity of Dijkstra's algorithm with a priority queue is �((�+�)log⁡�)*O*((*E*+*V*)log*V*), where �*E* is the number of edges and �*V* is the number of vertices. Explanation:  * The dominant factor in the time complexity is usually the priority queue operations. * Using a binary heap, each extraction of the minimum element takes �(log⁡�)*O*(log*V*) time, and there are at most �*V* vertices, so �(�log⁡�)*O*(*V*log*V*). * Relaxing all edges contributes �(�)*O*(*E*) time. * The overall time complexity is �((�+�)log⁡�)*O*((*E*+*V*)log*V*).   Note that if a Fibonacci heap is used, the decrease-key operation has a constant time complexity, and the overall time complexity can be further improved. However, Fibonacci heaps have a more complex implementation compared to binary heaps. The choice of priority queue depends on the specific requirements and characteristics of the graph being processed.  Top of Form |
| 1. **Analyze the time complexity of the dynamic programming solution to the knapsack problem.**   The time complexity of the dynamic programming solution to the 0/1 Knapsack Problem is typically analyzed in terms of the number of items (�*n*) and the capacity of the knapsack (�*W*). The dynamic programming approach uses a table to store intermediate results for subproblems. Time Complexity Analysis:  1. **Initialization:**    * Initializing a table of size (�+1)×(�+1)(*n*+1)×(*W*+1): �(��)*O*(*nW*) 2. **Filling the Table:**    * Filling in each entry of the table involves constant time operations.    * Each entry in the table corresponds to a subproblem representing the maximum value achievable with a subset of items and a specific knapsack capacity.    * The table is filled in a bottom-up manner, considering each item and each possible knapsack capacity.    * The time complexity for filling the table is �(��)*O*(*nW*).  Overall Time Complexity: The overall time complexity of the dynamic programming solution to the 0/1 Knapsack Problem is �(��)*O*(*nW*), where �*n* is the number of items and �*W* is the capacity of the knapsack. Explanation:  * The initialization step involves creating a table of size (�+1)×(�+1)(*n*+1)×(*W*+1), where each entry represents a subproblem. * Filling in the table involves iterating through each item and each possible knapsack capacity, calculating the maximum value achievable. This process takes �(��)*O*(*nW*) time. * The dominant factor in the time complexity is the size of the table, which is directly proportional to the product of the number of items and the knapsack capacity.   It's important to note that the dynamic programming solution to the knapsack problem is efficient when the knapsack capacity and the number of items are not too large. The time complexity makes it suitable for moderate-sized problem instances, but for very large instances, more efficient algorithms or heuristics might be considered. |
| 1. **Write down Floyd Warshall’s algorithm to find solution to the all-pairs shortest path algorithm. Run your algorithm on the following graph.**   The Floyd-Warshall algorithm is used to find the shortest paths between all pairs of vertices in a weighted directed graph, including negative weight edges. It works for both dense and sparse graphs and is based on dynamic programming. Floyd-Warshall Algorithm:  1. **Initialization:**    * Let �*n* be the number of vertices in the graph.    * Create a 2D array ����*dist* of size �×�*n*×*n* to represent the shortest distances between vertices.    * Initialize ����[�][�]*dist*[*i*][*j*] to the weight of the edge from �*i* to �*j* if there is an edge; otherwise, set it to ∞∞.    * Initialize ����[�][�]*dist*[*i*][*i*] to 0 for all �*i* (the distance from a vertex to itself is 0). 2. **Dynamic Programming Iteration:**    * For each intermediate vertex �*k*, iterate through all pairs of vertices �*i* and �*j*.    * Update ����[�][�]*dist*[*i*][*j*] to be the minimum of ����[�][�]*dist*[*i*][*j*] and the sum of ����[�][�]*dist*[*i*][*k*] and ����[�][�]*dist*[*k*][*j*]. 3. **Final Result:**    * After the dynamic programming iterations, the matrix ����*dist* contains the shortest distances between all pairs of vertices.  Example Graph: Consider the following graph:  3  (A)---->(C)  ^ \ / |  | \ / |  | \/ |  | /\ |  | / \ |  | / \v  (B)---->(D)  The adjacency matrix representing the graph is:  A B C D  A [ 0, 3, ∞, ∞]  B [ ∞, 0, 1, ∞]  C [ 3, ∞, 0, ∞]  D [ ∞, ∞, ∞, 0] |
| 1. **Write down Prim’s / Kruskal’s algorithm that determine the Minimum Spanning Tree of a graph. Run both the algorithms on the following graph. What is the difference .** |
| 1. **Write down Bubble Sort algorithm. What is the best and worst case complexities of Bubble Sort algorithm?Solve the bubble sort 8,22,7,9,31,19,5 and 13.find the number of swaps?**   Bubble Sort is a simple 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 the list is sorted.  Here is the Bubble Sort algorithm:   1. Start from the first element (index 0) and traverse to the end of the list. 2. Compare each pair of adjacent elements. 3. If the elements are in the wrong order (i.e., the element on the left is greater than the element on the right), swap them. 4. Continue this process for each pair of adjacent elements until the end of the list is reached. 5. After the first pass, the largest element will be at the end of the list. Repeat steps 1-4 for the remaining elements, excluding the already sorted ones. 6. Continue these passes until the entire list is sorted.   The best-case complexity of Bubble Sort is O(n) when the list is already sorted. The worst-case and average-case complexities are O(n^2), where n is the number of elements in the list.  Now, let's solve the Bubble Sort for the given list [8, 22, 7, 9, 31, 19, 5, 13]:  Pass 1:   * Compare and swap 8 and 22 (no swap) * Compare and swap 22 and 7 (swap) * Compare and swap 22 and 9 (swap) * Compare and swap 22 and 31 (no swap) * Compare and swap 31 and 19 (swap) * Compare and swap 31 and 5 (swap) * Compare and swap 31 and 13 (swap)   Pass 2:   * Compare and swap 8 and 7 (swap) * Compare and swap 22 and 9 (swap) * Compare and swap 22 and 19 (swap) * Compare and swap 31 and 5 (swap) * Compare and swap 31 and 13 (swap)   Pass 3:   * Compare and swap 8 and 7 (swap) * Compare and swap 22 and 9 (swap) * Compare and swap 22 and 19 (swap) * Compare and swap 31 and 13 (swap)   Pass 4:   * Compare and swap 8 and 7 (swap) * Compare and swap 22 and 9 (swap) * Compare and swap 22 and 19 (swap)   Pass 5:   * Compare and swap 8 and 7 (swap) * Compare and swap 22 and 9 (swap)   Pass 6:   * Compare and swap 8 and 7 (swap)   The sorted list is [7, 8, 9, 13, 19, 22, 31]. The total number of swaps is 16. |
| 1. **Determine the LCS of < P,Q,R,S,T,P,Q,R,S > and < P,R,A,T,P,B,R,Q,R,P,S,>** |
| 1. **Summarise P.NP and NP-complete decision problems.** |
| 1. **Determine the LCS of < A,B,C,D,B,A,C,D,F > and < C,B,A,>** |
| 1. **Sort the following set of keys using Merge sort 12, 3, 34, 45, 11, 7, 26, 98, 5, 43, 9, 15, 8**   Merge Sort is a divide-and-conquer algorithm that recursively divides the input array into smaller halves, sorts them, and then merges them back together. Here's how you can use Merge Sort to sort the given set of keys:  plaintext  Copy code  Original array: 12, 3, 34, 45, 11, 7, 26, 98, 5, 43, 9, 15, 8  Step 1: Split the array into halves  [12, 3, 34, 45, 11, 7, 26, 98, 5, 43, 9, 15, 8]  [12, 3, 34, 45] | [11, 7, 26, 98, 5, 43, 9, 15, 8]  Step 2: Recursively sort each half  [12, 3, 34, 45] => [3, 12, 34, 45]  [11, 7, 26, 98, 5, 43, 9, 15, 8] => [5, 7, 8, 9, 11, 15, 26, 43, 98]  Step 3: Merge the sorted halves  [3, 12, 34, 45] | [5, 7, 8, 9, 11, 15, 26, 43, 98]  Merge: [3, 5, 7, 8, 9, 11, 12, 15, 26, 34, 43, 45, 98]  The sorted array: 3, 5, 7, 8, 9, 11, 12, 15, 26, 34, 43, 45, 98  So, the sorted set of keys using Merge Sort is [3, 5, 7, 8, 9, 11, 12, 15, 26, 34, 43, 45, 98]. |
| 1. **Evaluate the trade-offs between dynamic programming and greedy algorithms.** |
| 1. **Evaluate the trade-offs between Dijkstra's algorithm and the Bellman-Ford algorithm for finding shortest paths.** |
| 1. **Generate the applicability of Kruskal's algorithm to graphs with negative edge weights.** |
| 1. **Rewrite the applicability of dynamic programming to problems with overlapping subproblems and optimal substructure.** |
| 1. **Prove that the height of RB tree with n internal nodes is at most 2lg(n+1).** |
| 1. **Produce two graphs with negative weight edges on which (a) Dijkstra's algorithm works (b) Dijkstra's algorithm fails** |
| 1. **Find the shortest path from source node 5 to all other destinations using Bellman-Ford Algorithm for the following graph** |
| 1. **Deduce the complexity of Quick Sort algorithm in its best, worst and average cases.**   Quick Sort is a divide-and-conquer sorting algorithm that works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.  Here are the complexities for Quick Sort in its best, worst, and average cases:   1. **Best Case Complexity (Ω(n log n)):**    * The best-case scenario occurs when the pivot chosen at each step divides the array into two equal halves.    * In this case, each element is compared only once with the pivot, and the partitioning is perfectly balanced.    * The best-case time complexity is Ω(n log n), where 'n' is the number of elements in the array. 2. **Worst Case Complexity (O(n^2)):**    * The worst-case scenario occurs when the pivot chosen at each step is the smallest or largest element in the array.    * This leads to an unbalanced partition, and the algorithm degrades into a quadratic time complexity.    * The worst-case time complexity is O(n^2), where 'n' is the number of elements in the array.    * However, with proper pivot selection strategies (such as the median of three), the likelihood of encountering the worst case is minimized. 3. **Average Case Complexity (O(n log n)):**    * On average, Quick Sort performs well and has an average time complexity of O(n log n).    * This assumes a random distribution of pivot elements and effective partitioning of the array.    * The average-case analysis is commonly used because, in practice, Quick Sort tends to perform efficiently and is widely used for its average-case performance. |
| 1. **Design an algorithm to solve a specific problem by combining elements of dynamic programming and greedy strategies.** |
| 1. **Design a dynamic programming algorithm to find the shortest path between all pairs of nodes using Floyd-Warshall algorithm.** |
| 1. **Design an algorithm to find the shortest path between all pairs of nodes using Floyd-Warshall algorithm.** |
| 1. **Develop an algorithm to solve the traveling salesman problem using dynamic programming.** |
| 1. **Compute the optimal solution for knapsack problem using greedy method N=3, M= 20, (p1,p2,p3)= (25,24,15), (w1,w2,w3) =(18,15,10)** |
| **Problem: Consider the following instances of the fractional knapsack problem: n = 3, M = 20, V = (24, 25, 15) and W = (18, 15, 20) find the feasible solutions.**  **Solution:**  Let us arrange items by decreasing order of profit density. Assume that items are labeled as X = (I1, I2, I3), have profit V = {24, 25, 15} and weight W = {18, 15, 20}.   | **Item (xi)** | **Value (vi)** | **Weight (wi)** | ***pi = vi / wi*** | | --- | --- | --- | --- | | I2 | 25 | 15 | 1.67 | | I1 | 24 | 18 | 1.33 | | I3 | 15 | 20 | 0.75 |   We shall select one by one item from Table. If the inclusion of an item does not cross the knapsack capacity, then add it. Otherwise, break the current item and select only the portion of item equivalent to remaining knapsack capacity. Select the profit accordingly. We should stop when knapsack is full or all items are scanned.  Initialize, Weight of selected items, SW = 0,  Profit of selected items, SP = 0,  Set of selected items, S = { },  Here, Knapsack capacity M = 20.  **Iteration 1 :** SW= (SW + w2) = 0 + 15 = 15  SW ≤ M, so select I2  S = { I2 }, SW = 15, SP = 0 + 25 = 25  **Iteration 2 :** SW + w1> M, so break down item I1.  The remaining capacity of the knapsack is 5 unit, so select only 5 units of item I1.  *frac* = (M – SW) / W[i] = (20 – 15) / 18 = 5 / 18  S = { I2,I1\* 5/18 }  SP = SP + v1 \* *frac* = 25 + (24 \* (5/18)) = 25 + 6.67 = 31.67  SW = SW + w1 \* *frac* = 15 + (18 \* (5/18)) = 15 + 5 = 20  The knapsack is full. Fractional [Greedy algorithm](https://codecrucks.com/greedy-algorithm/) selects items { I2,I1\* 5/18 }, and it gives a profit of 31.67 units.  **Problem: Find the optimal solution for knapsack problem (fraction) where knapsack capacity = 28, P = {9, 5, 2, 7, 6, 16, 3} and w = {2, 5, 6, 11, 1, 9, 1}.**  **Solution:**  Arrange items in decreasing order of profit to weight ratio   | **Item** | **Profit pi** | **Weight wi** | **Ratio vi/wi** | | --- | --- | --- | --- | | I5 | 6 | 1 | 6.00 | | I1 | 9 | 2 | 4.50 | | I7 | 3 | 1 | 3.00 | | I6 | 16 | 9 | 1.78 | | I2 | 5 | 5 | 1.00 | | I4 | 7 | 11 | 0.64 | | I3 | 2 | 6 | 0.33 |   Initialize, Weight = 0, P = 0, M = 28, S = { }  Where S is the solution set, P and W is profit and weight of included items, respectively. M is the capacity of the knapsack.  **Iteration 1**  (Weight + w5) ≤ M, so select I5  So, S = { I5 }, Weight = 0 + 1 = 1, P = 0 + 6= 6  **Iteration 2**  (Weight + w1) ≤ M, so select I1  So, S = {I5 ,I1 }, Weight = 1 + 2 = 3, P = 6 + 9= 15  **Iteration 3**  (Weight + w7) ≤ M, so select I7  So, S = {I5, I1, I7 }, Weight = 3 + 1 = 4, P = 15 + 3= 18  **Iteration 4**  (Weight + w6) ≤ M, so select I6  So, S = {I5, I1, I7, I6 }, Weight = 4 + 9 = 13, P = 18 + 16= 34  **Iteration 5**  (Weight + w2) ≤ M, so select I2  So, S = {I5, I1, I7, I6, I2 }, Weight = 13 + 5 = 18, P = 34 + 5= 39  **Iteration 6**  (Weight + w4) > M, So I4 must be broken down into two parts x and y such that x = capacity left in knapsack and y = I4 – x.  Available knapsack capacity is 10 units. So we can select only (28 – 18) / 11 = 0.91 unit of I4.  So S = {I5, I1, I7, I6, I2, 0.91 \* I4 }, Weight = 18 + 0.91\*11 = 28, P = 39 + 0.91 \* 7= 45.37   1. **Write down Dijkstra’s algorithm for solving the single source shortest path on a weighted, directed graph. Execute your algorithm on the following graph with vertex A as the source vertex.**   **http://d2vlcm61l7u1fs.cloudfront.net/media/7dc/7dcc48aa-8b7f-4492-bb95-df7c141f26b8/phpaNFvua.png** |