**CAT 2**

**Question Bank**

**ADA**

Q1. Analyse the need of Dynamic Programming in multistage graphs with example.

The need for dynamic programming in multistage graphs arises due to two key properties:

Overlapping Subproblems: Multistage graphs often have subproblems that are repeated multiple times during the solution process. These subproblems can be solved independently and their solutions can be reused, which leads to a significant reduction in redundant computations. Dynamic programming takes advantage of this property by storing the solutions to subproblems in a table or memoization array, allowing efficient retrieval and reuse of these solutions when needed.

Optimal Substructure: The optimal solution to a multistage graph can be constructed from the optimal solutions of its subproblems. Dynamic programming utilizes this property by solving the subproblems in a specific order, starting from the final stage and working backward. The solutions to the subproblems are combined in an optimal way to determine the overall optimal solution. By leveraging optimal substructure, dynamic programming guarantees that the final solution is indeed the best possible solution for the given problem.

Let's consider an example to illustrate the need for dynamic programming in multistage graphs. Suppose we have a project with multiple stages, and each stage has a set of tasks that need to be completed. Each task has a certain cost associated with it. The goal is to find the minimum cost path from the starting stage to the final stage, considering that certain tasks can only be performed after completing specific tasks in the previous stages.

To solve this problem using dynamic programming, we can represent the multistage graph as a directed acyclic graph, where each node corresponds to a task, and the edges represent the dependencies between tasks. The cost associated with each task is represented by the weight of the edges.

By applying dynamic programming, we can start from the final stage and compute the minimum cost to reach each task. We store these values in a table or memoization array. Then, we move backward through the stages, considering all possible paths from each task in the current stage to the tasks in the next stage, and update the minimum cost accordingly.

At the end of this process, the minimum cost path from the starting stage to the final stage can be determined by examining the values in the memoization array. The dynamic programming approach avoids redundant computations by reusing the solutions to subproblems, which results in an efficient solution for the multistage graph problem.

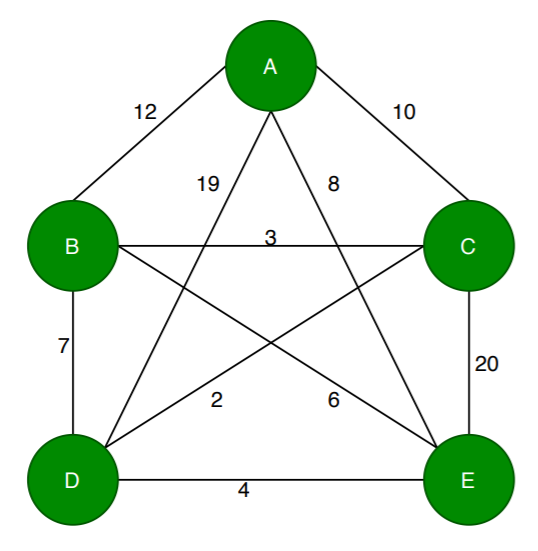
Q2. Examine the travelling salesman problem and discuss how to solve it using dynamic programming?

The [Travelling Salesman Problem (TSP)](https://www.baeldung.com/java-simulated-annealing-for-traveling-salesman) is a very well known problem in theoretical computer science and operations research. The standard version of TSP is a hard problem to solve and belongs to the [NP-Hard class](https://www.baeldung.com/cs/p-np-np-complete-np-hard).

## 2. Introduction to TSP

In the TSP, given a set of cities and the distance between each pair of cities, a salesman needs to choose the shortest path to visit every city exactly once and return to the city from where he started.

Let’s take an example to understand the TSP in more detail:



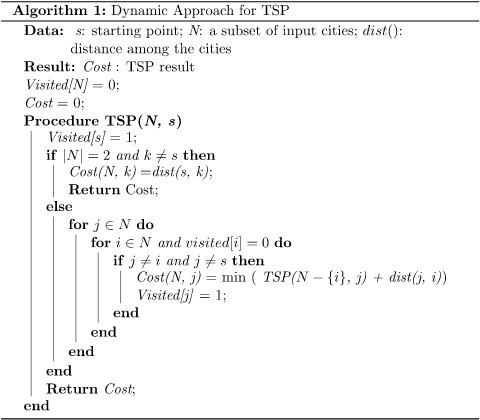
Here, the nodes represent cities, and the values in each edge denote the distance between one city to another. Let’s assume a salesman starts the journey from the city . According to TSP, the salesman needs to travel all the cities exactly once and get back to the city  by choosing the [shortest path](https://www.baeldung.com/cs/floyd-warshall-shortest-path). Here the shortest path means the sum of the distance between each city travelled by the salesman, and it should be less than any other path.

With only  cities, the problem already looks quite complex. As the graph in the example is a [complete graph](https://www.baeldung.com/cs/graph-theory-intro), from each city, the salesman can reach any other cities in the graph. From each city, the salesman needs to choose a route so that he doesn’t have to revisit any city, and the total distance travelled should be minimum.

**How to find out such a path? Let’s discuss an algorithm to solve the TSP problem.**

## 3. Dynamic Programming Approach for Solving TSP

Let’s first see the pseudocode of the dynamic approach of TSP, then we’ll discuss the steps in detail:



In this algorithm, we take a subset  of the required cities needs to be visited, distance among the cities  and starting city  as inputs. Each city is identified by unique city id like .

Initially, all cities are unvisited, and the visit starts from the city . We assume that the initial travelling cost is equal to . Next, the TSP distance value is calculated based on a recursive function. If the number of cities in the subset is two, then the recursive function returns their distance as a base case.

On the other hand, if the number of cities is greater than , then we’ll calculate the distance from the current city to the nearest city, and the minimum distance among the remaining cities is calculated recursively.

Finally, the algorithm returns the minimum distance as a TSP solution.

**Here we use a dynamic approach to calculate the cost function .** Using recursive calls, we calculate the cost function for each subset of the original problem.

## 4. Time Complexity Analysis

In the dynamic algorithm for TSP, the number of possible subsets can be at most . Each subset can be solved in  times. **Therefore, the time complexity of this algorithm would be** .

## 5. Conclusion

TSP is a popular NP-Hard problem, but depending on the size of the input cities, it is possible to find an optimal or a near-optimal solution using various algorithms.

In this tutorial, we’ve discussed a dynamic programming approach for solving TSP. We also presented the time complexity of the given algorithm.

Q3. Inspect and solve for the optimum cost using matrix chain multiplication of A 4x5, B 5x7 , C 7x3 , D 3x2 and E 2x7 matrices.

Q4. Apply dynamic programming to find out the optimal sequence for the matrix chain multiplication of A 4x10, B 10x3 , C 3x12 , D 12x20 and E 20x7 matrices.

Q5. Explain (0/1) knapsack problem and fractional knapsack problem.

| **Sr. No** | **0/1 knapsack problem** | **Fractional knapsack problem** |
| --- | --- | --- |
| **1.** | The 0/1 knapsack problem is solved using dynamic programming approach. | Fractional knapsack problem is solved using a greedy approach. |
| **2.** | The 0/1 knapsack problem has an optimal structure. | The fractional knapsack problem also has an optimal structure. |
| **3.** | In the 0/1 knapsack problem, we are not allowed to break items. | Fractional knapsack problem, we can break items for maximizing the total value of the knapsack. |
| **4.** | 0/1 knapsack problem, finds a most valuable subset item with a total value less than equal to weight. | In the fractional knapsack problem, finds a most valuable subset item with a total value equal to the weight. |
| **5.** | In the 0/1 knapsack problem we can take objects in an integer value. | In the fractional knapsack problem, we can take objects in fractions in floating points. |
| **6.** | The 0/1 knapsack does not have greedy choice property | The fraction knapsack do have greedy choice property |

Q6. Explain the Dynamic Programming (DP) Algorithmic Paradigm? List a few problems which can be solved using the same.

# What is Dynamic Programming? Top-down vs Bottom-up Approach

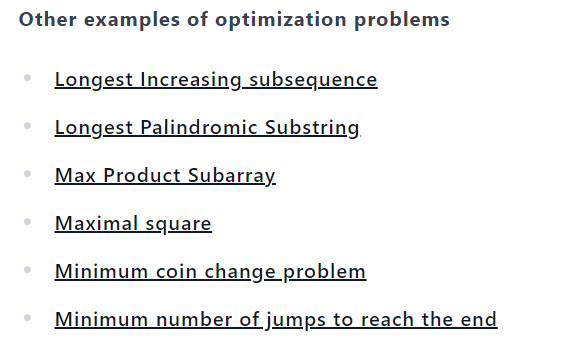
Dynamic programming is an algorithmic paradigm that is majorly used to formulate solutions to these optimization problems. Hence, it is critical to master this problem-solving approach to become a good competitive [programmer](https://www.simplilearn.com/job-roles-for-programmers-article). So, in this article on ‘What is Dynamic Programming’, we will discover the dynamic programming paradigm in detail.

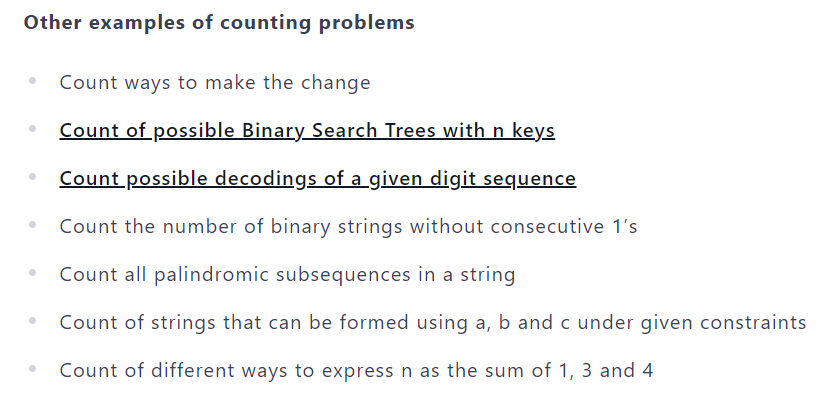
## What Is Dynamic Programming?

Dynamic programming is an [algorithmic](https://www.simplilearn.com/tutorials/data-structure-tutorial/what-is-an-algorithm) paradigm that divides broader problems into smaller subproblems and stores the result for later use, eliminating the need for any re-computation. This problem-solving approach is quite similar to the divide and conquer approach.

We solve problems in both these paradigms by integrating the answers to smaller subproblems. However, unlike divide and conquer, the subproblems in dynamic programming repeat themselves multiple times. This means dynamic programming has different properties than the divide and conquer approach. If the problem abides by properties given below, only then it can be solved using a dynamic programming paradigm:

* Optimal Substructure: A problem is said to have an optimal substructure if we can formulate a recurrence relation.
* Overlapping Subproblem: A problem has an optimal substructure if we can formulate a recurrence relation for it.

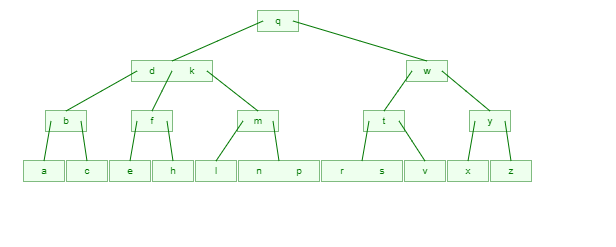




Q7. Examine and solve the following 0/1 Knapsack Problem using Dynamic Programming. There are five items whose weights and values are given in following arrays. Weight w [ ] = {1,2,5,6,7} Value v [ ] = {1, 6, 18, 22, 28}. Show your Equation and find out the optimal knapsack items for weight capacity of 11 units.

Q8. solve for the shortest path using multistage graph method using dynamic programming by taking an example.

Q9. Define a B-Tree. Show the result of inserting the keys F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E in the order to an empty B-Tree of degree 3



Q10. Apply dynamic programming to find out the optimal sequence for the matrix chain multiplication of A 4x10, B 10x3 , C 3x12 , D 12x20 and E 20x7 matrices.

Q11. Find the maximum keys in the case of a B-tree of order 4 and of height 3.

Q12. Solve the Travelling Salesman problem Dynamic programming with example.

Build the maximum profit using dynamic method by considering the Knapsack capacity W=9, w = (3,4,5,7) and v=(12,40,25,42).

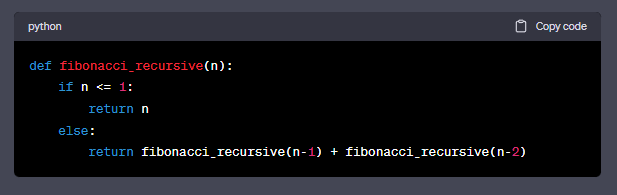
Q13. Interpret the need of Dynamic Programming over Recursion. Take suitable example to illustrate the above need.

Dynamic programming is often preferred over recursion in certain scenarios due to its ability to avoid redundant computations by storing and reusing the solutions to subproblems. This can lead to significant performance improvements and can be especially advantageous when dealing with problems that exhibit overlapping subproblems.

Let's consider an example to illustrate the need for dynamic programming over recursion. Suppose we have a problem of calculating the nth Fibonacci number, where each Fibonacci number is the sum of the two preceding numbers in the sequence: F(n) = F(n-1) + F(n-2). We want to compute this value using both recursion and dynamic programming and compare their efficiency.

Recursive Approach:

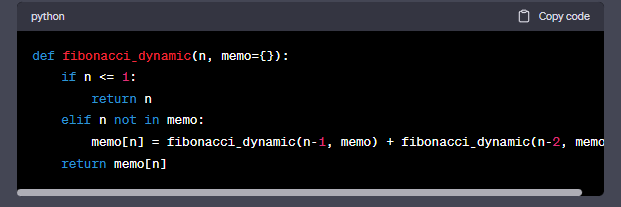
A straightforward way to calculate the nth Fibonacci number is by using recursion. We can define a recursive function that calls itself to calculate the preceding Fibonacci numbers:



While this recursive approach is intuitive, it has a significant drawback: it recalculates the same Fibonacci numbers multiple times. For example, to calculate F(5), the function would calculate F(4) and F(3), and to calculate F(4), it would calculate F(3) and F(2) again. This redundancy increases exponentially as the value of n grows, resulting in a significant number of redundant computations.

Dynamic Programming Approach:

Dynamic programming can efficiently solve this problem by avoiding redundant computations. Instead of recalculating the Fibonacci numbers multiple times, we can store the intermediate results in an array or memoization table for future reuse. Here's an implementation of the dynamic programming approach using memoization:



In this approach, the function fibonacci\_dynamic checks if the Fibonacci number for a particular value of n has already been computed and stored in the memoization table. If so, it retrieves the value from the table; otherwise, it calculates the Fibonacci number using the recursive calls and stores the result in the memoization table for future use.

By storing and reusing the solutions to subproblems, the dynamic programming approach eliminates redundant computations. As a result, it provides a significant performance improvement compared to the recursive approach. The time complexity of the dynamic programming approach is reduced from exponential to linear, making it much more efficient for larger values of n.

In summary, dynamic programming offers a more efficient solution compared to recursion when dealing with problems that have overlapping subproblems. By avoiding redundant computations through memoization, dynamic programming significantly improves the runtime and efficiency of the solution.

Q14. Compare the divide and conquer and dynamic programming problem solving approaches.

The divide and conquer approach and the dynamic programming approach are both problem-solving techniques that aim to break down complex problems into smaller, more manageable subproblems. However, there are fundamental differences between the two approaches. Let's compare them in terms of their key characteristics and use cases:

Subproblem Relationship:

Divide and Conquer: In the divide and conquer approach, the subproblems are independent of each other. The problem is divided into smaller subproblems, each of which is solved independently, and the solutions are combined to obtain the final solution.

Dynamic Programming: In dynamic programming, the subproblems have overlapping structures, meaning that the solutions to subproblems are reused in the computation of larger subproblems or the final solution.

Subproblem Solving:

Divide and Conquer: The divide and conquer approach solves subproblems recursively and independently. Each subproblem is solved in isolation without considering previous solutions.

Dynamic Programming: Dynamic programming solves subproblems iteratively and uses previously computed solutions to optimize the computation of larger subproblems or the final solution.

Solution Combination:

Divide and Conquer: The solutions to subproblems are combined or merged to obtain the solution for the entire problem. This combination step often involves merging or comparing subproblem solutions to derive the final solution.

Dynamic Programming: The solutions to subproblems are stored and reused to optimize the computation of larger subproblems or the final solution. The combination step involves using the stored solutions to build up the final solution iteratively.

Time Complexity:

Divide and Conquer: The time complexity of divide and conquer algorithms can vary depending on the problem. In some cases, the time complexity can be exponential, especially if the subproblems overlap significantly or if there is no mechanism to avoid redundant computations.

Dynamic Programming: Dynamic programming optimizes time complexity by avoiding redundant computations through memoization or tabulation. It reduces the time complexity to polynomial or linear time, depending on the problem structure.

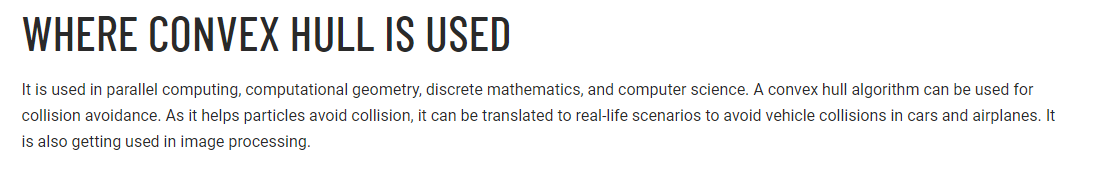
Use Cases:

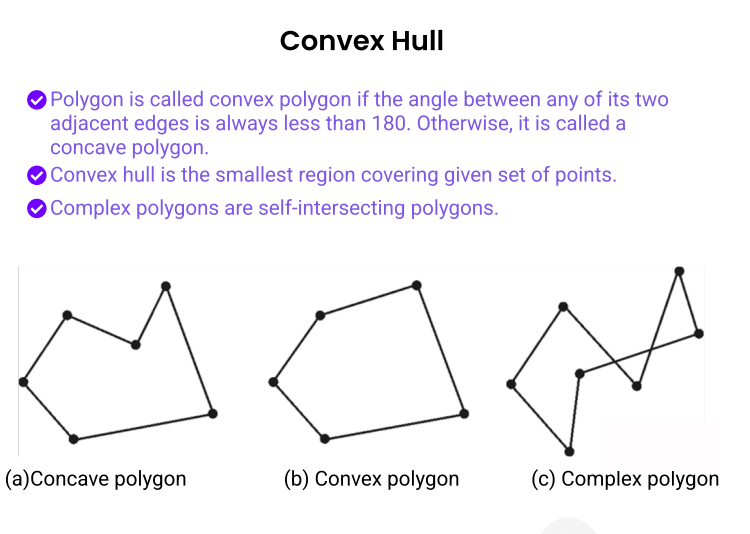
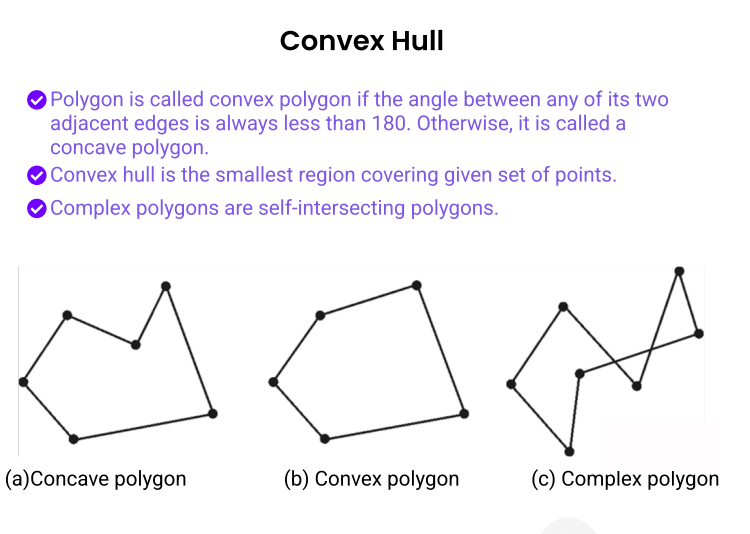
Divide and Conquer: The divide and conquer approach is suitable for problems that can be divided into independent subproblems. Examples include sorting algorithms (e.g., Merge Sort, Quick Sort), finding the maximum subarray, and calculating matrix multiplication.

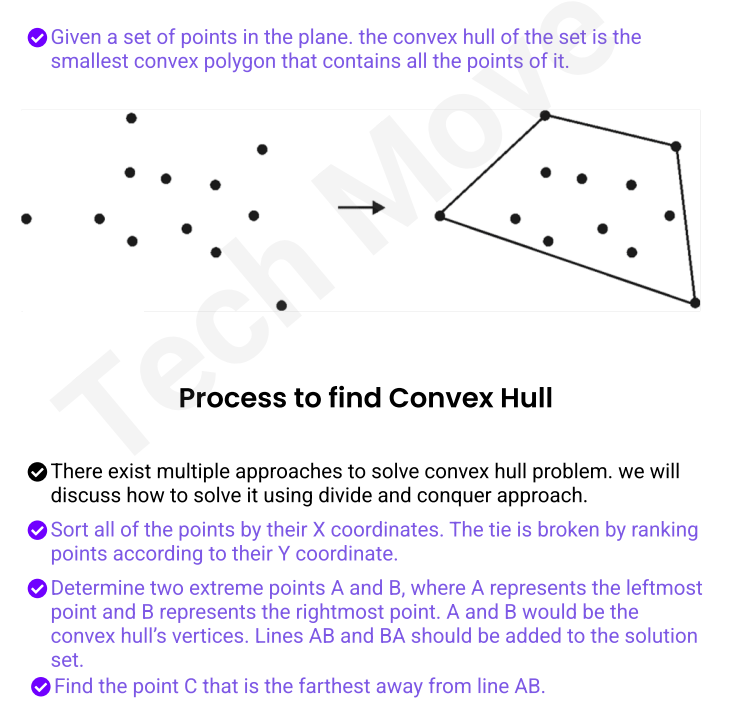
Dynamic Programming: Dynamic programming is suitable for problems with overlapping subproblems, where the optimal solution can be constructed from the solutions to subproblems. Examples include the knapsack problem, finding the shortest path in a graph, and calculating Fibonacci numbers.

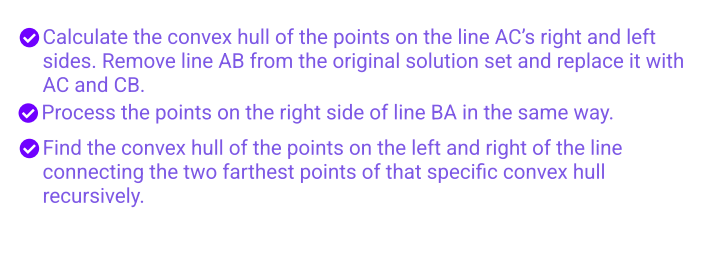
Q15. Create a Binomial Heap using suitable example.

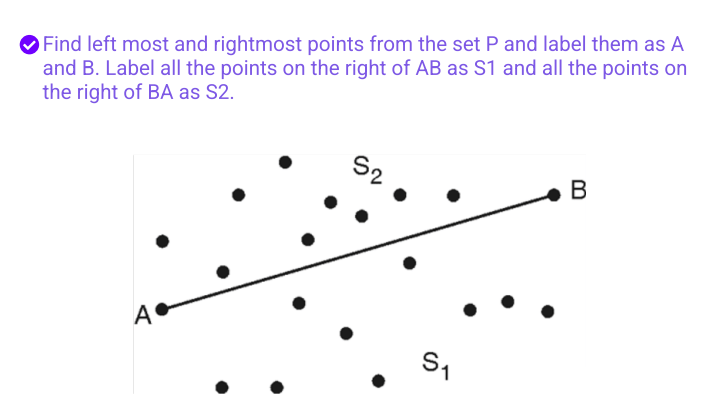
Q16. Identify the use of convex hull problem and explain it with the help of a diagram.

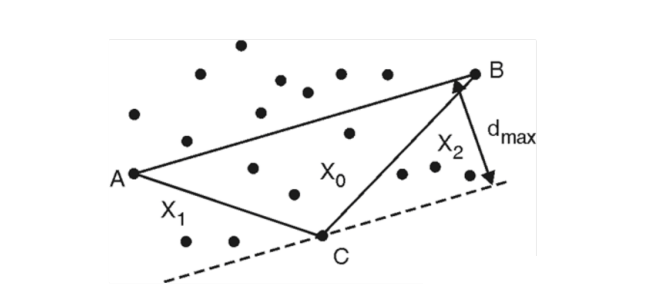


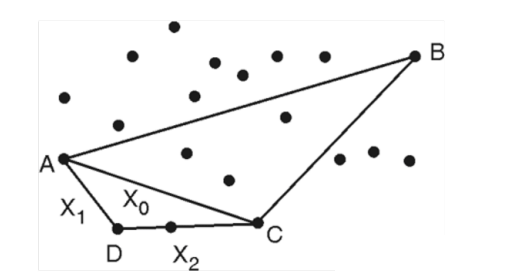


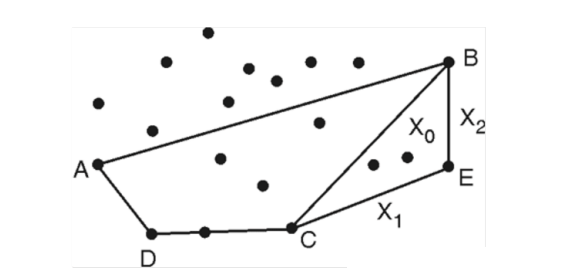


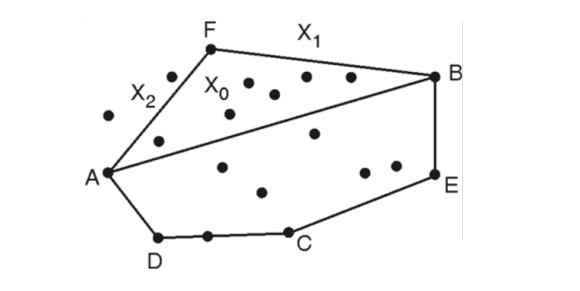


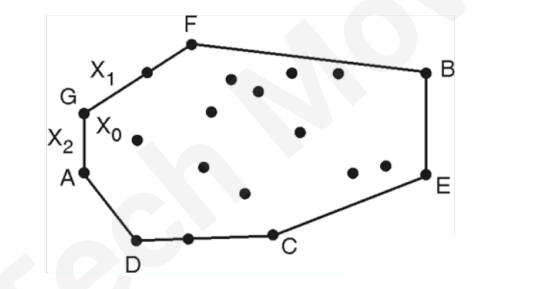












Q17. Explain a Red Black tree.

When it comes to searching and sorting data, one of the most fundamental data structures is the binary search tree. However, the performance of a binary search tree is highly dependent on its shape, and in the worst case, it can degenerate into a linear structure with a time complexity of O(n). This is where Red Black Trees come in, they are a type of balanced binary search tree that use a specific set of rules to ensure that the tree is always balanced. This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.

Red Black Trees are self-balancing, meaning that the tree adjusts itself automatically after each insertion or deletion operation. It uses a simple but powerful mechanism to maintain balance, by coloring each node in the tree either red or black.

Red Black Tree-

Red-Black tree is a binary search tree in which every node is colored with either red or black. It is a type of self balancing binary search tree. It has a good efficient worst case running time complexity.

Properties of Red Black Tree:

The Red-Black tree satisfies all the properties of binary search tree in addition to that it satisfies following additional properties –

1. Root property: The root is black.

2. External property: Every leaf (Leaf is a NULL child of a node) is black in Red-Black tree.

3. Internal property: The children of a red node are black. Hence possible parent of red node is a black node.

4. Depth property: All the leaves have the same black depth.

5. Path property: Every simple path from root to descendant leaf node contains same number of black nodes.

The result of all these above-mentioned properties is that the Red-Black tree is roughly balanced.

Rules That Every Red-Black Tree Follows:

Every node has a color either red or black.

The root of the tree is always black.

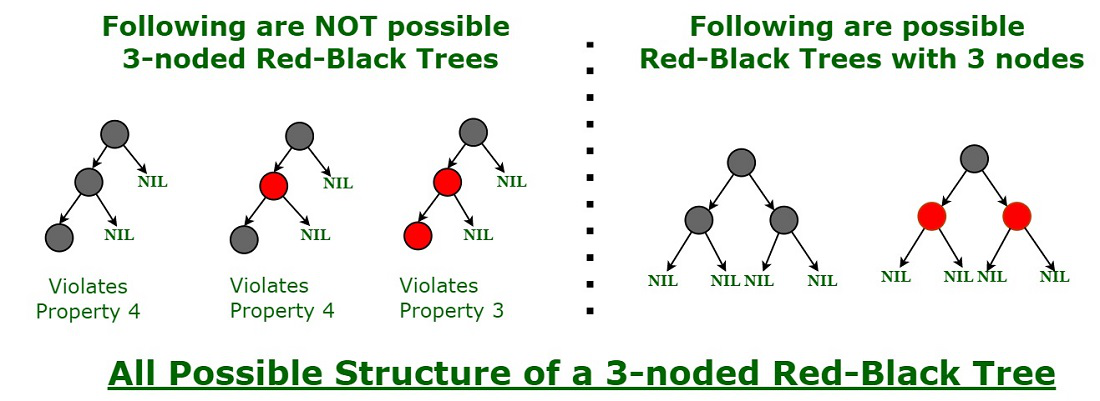
There are no two adjacent red nodes (A red node cannot have a red parent or red child).

Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.

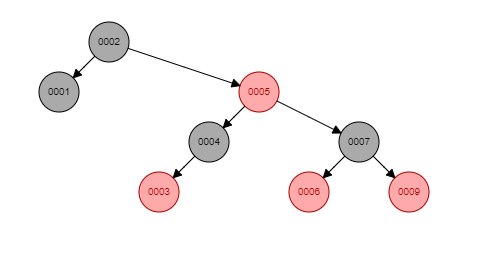
Every leaf (e.i. NULL node) must be colored BLACK.

| **Sr. No.** | **Algorithm** | **Time Complexity** |
| --- | --- | --- |
| 1. | Search | O(log n) |
| 2. | Insert | O(log n) |
| 3. | Delete | O(log n) |

**“n” is the total number of elements in the red-black**



Q 18. Create a Red Black tree given input 2, 1, 4, 5, 9, 3, 6, 7.



Q 19. Explain B tree.

The limitations of traditional binary search trees can be frustrating. Meet the B-Tree, the multi-talented data structure that can handle massive amounts of data with ease. When it comes to storing and searching large amounts of data, traditional binary search trees can become impractical due to their poor performance and high memory usage. B-Trees, also known as B-Tree or Balanced Tree, are a type of self-balancing tree that was specifically designed to overcome these limitations.

Unlike traditional binary search trees, B-Trees are characterized by the large number of keys that they can store in a single node, which is why they are also known as “large key” trees. Each node in a B-Tree can contain multiple keys, which allows the tree to have a larger branching factor and thus a shallower height. This shallow height leads to less disk I/O, which results in faster search and insertion operations. B-Trees are particularly well suited for storage systems that have slow, bulky data access such as hard drives, flash memory, and CD-ROMs.

B-Trees maintain balance by ensuring that each node has a minimum number of keys, so the tree is always balanced. This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.

Time Complexity of B-Tree:

Sr. No. Algorithm Time Complexity

1. Search O(log n)

2. Insert O(log n)

3. Delete O(log n)

Note: “n” is the total number of elements in the B-tree

Properties of B-Tree:

All leaves are at the same level.

B-Tree is defined by the term minimum degree ‘t‘. The value of ‘t‘ depends upon disk block size.

Every node except the root must contain at least t-1 keys. The root may contain a minimum of 1 key.

All nodes (including root) may contain at most (2\*t – 1) keys.

Number of children of a node is equal to the number of keys in it plus 1.

All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.

B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.

Like other balanced Binary Search Trees, the time complexity to search, insert and delete is O(log n).

Insertion of a Node in B-Tree happens only at Leaf Node.

Q 20. Explain binomial heap.

A binomial heap can be defined as the collection of binomial trees that satisfies the heap properties, i.e., min-heap. The min-heap is a heap in which each node has a value lesser than the value of its child nodes. Mainly, Binomial heap is used to implement a priority queue. It is an extension of binary heap that gives faster merge or union operations along with other operations provided by binary heap.

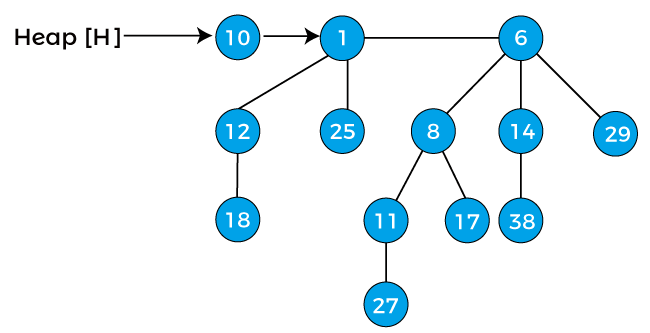
### Properties of Binomial heap

There are following properties for a binomial heap with **n** nodes -

* Every binomial tree in the heap must follow the **min-heap** property, i.e., the key of a node is greater than or equal to the key of its parent.
* For any non-negative integer k, there should be atleast one binomial tree in a heap where root has degree k.

The first property of the heap ensures that the min-heap property is hold throughout the heap. Whereas the second property listed above ensures that a binary tree with **n** nodes should have at most **1 + log2 n** binomial trees, here **log2** is the binary logarithm.

We can understand the properties listed above with the help of an example -



The above figure has three binomial trees, i.e., B0, B2, and B3. The above all three binomial trees satisfy the min heap's property as all the nodes have a smaller value than the child nodes.

**Time Complexity**

|  |  |
| --- | --- |
| **Operations** | **Time complexity** |
| Finding the minimum key | O(log n) |
| Inserting a node | O(log n) |
| Extracting minimum key | O(log n) |
| Decreasing a key | O(log n) |
| Union or merging | O(log n) |
| Deleting a node | O(log n) |

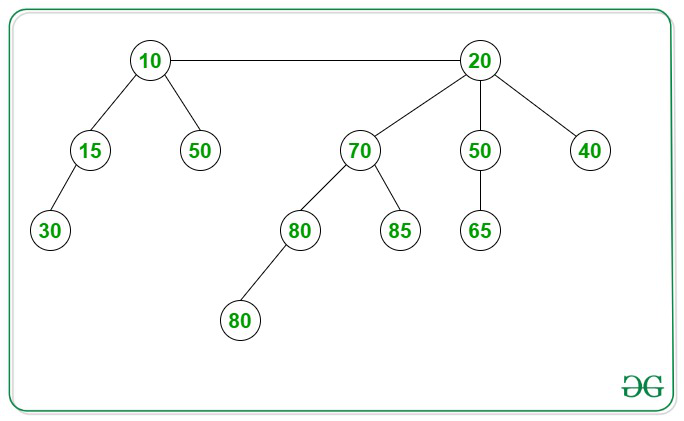
Q 21. Discuss how Fibonacci heap is different from binomial heap?

Binomial Heap:

A Binomial Heap is a collection of Binomial Tree where each Binomial Tree follows the Min-Heap property and there can be at most one Binomial Tree of any degree.

Example of Binomial Heap:

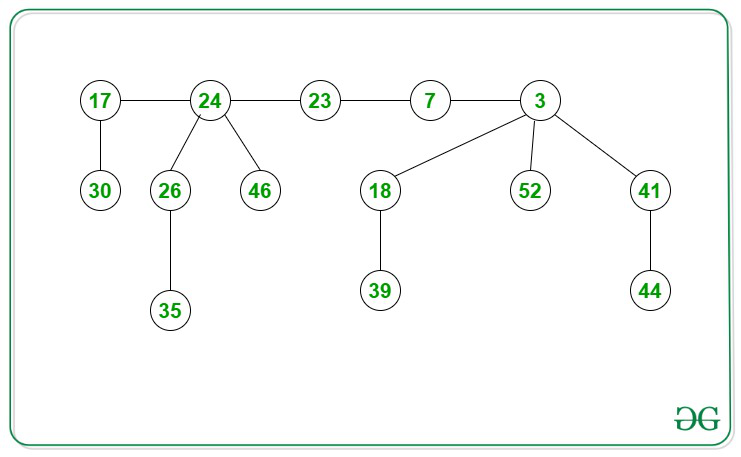
The key difference between a Binary Heap and a Binomial Heap is how the heaps are structured. In a Binary Heap, the heap is a single tree, which is a complete binary tree. In a Binomial Heap, the heap is a collection of smaller trees (that is, a forest of trees), each of which is a binomial tree. A complete binary tree can be built to hold any number of elements, but the number of elements in a binomial tree of some order N is always 2\*N. Consequently, one complete binary tree is needed to back a Binary Heap, but we may need multiple binomial trees to back a Binomial Heap.



Fibonacci Heap:

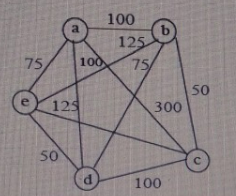
Like Binomial Heap, Fibonacci Heap is a collection of trees with Min-Heap or Max-Heap property. In Fibonacci Heap, trees can have any shape even all trees can be single nodes (This is unlike Binomial Heap where every tree has to be a Binomial Tree). Fibonacci Heap maintains a pointer to a minimum value (which is the root of a tree). All tree roots are connected using a circular doubly linked list, so all of them can be accessed using a single ‘min’ pointer.

Example of Fibonacci Heap:

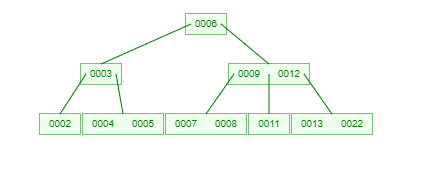


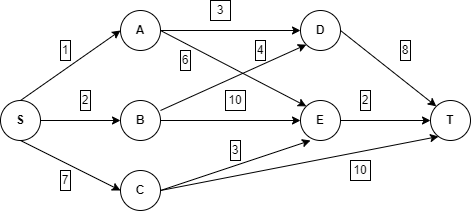
| **Operation** | **Binomial Heap** | **Fibonacci Heap** |
| --- | --- | --- |
| insert | O(log N) | O(1) |
| find-min | O(log N) | O(1) |
| delete | O(log N) | O(log N) |
| decrease-key | O(log N) | O(1) |
| union | O(log N) | O(1) |

The difference in time complexities of various operations associated with Binary heap, Binomial heap, and Fibonacci heaps are mentioned in the following table.

Q 22. Solve the Travelling Salesman problem using Dynamic Programming problem for the following graph and find the shortest path  


Q 23. Examine and create a B-Tree of order 4 by inserting following key values: 6, 5, 22, 9, 2, 13, 3, 7, 11, 12, 4, 8



Q 24. Solve the below graph for the shortest path using multistage graph method with dynamic programming approach.  


Q 25. Solve the following instance of 0/1 Knapsack problem given the knapsack capacity is M=5

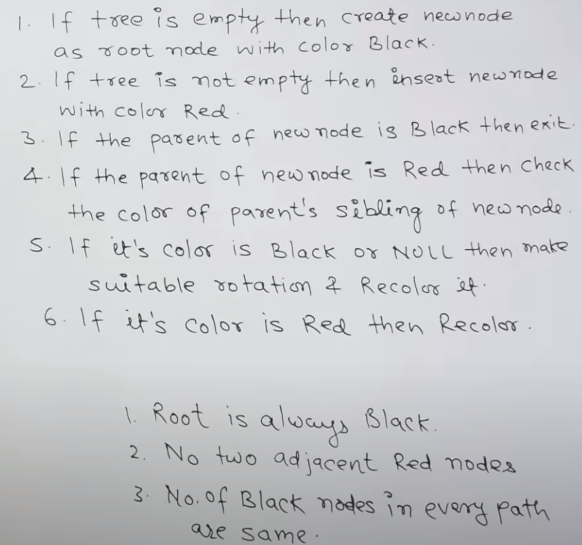
|  |  |  |  |
| --- | --- | --- | --- |
| **Objects** | 1 | 2 | 3 |
| **Profit** | 5 | 3 | 4 |
| **Weight** | 3 | 2 | 1 |

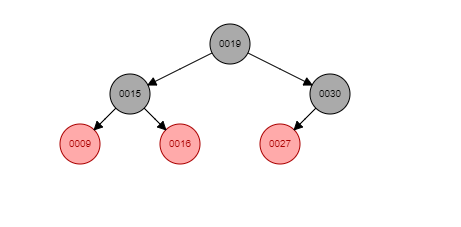
|  |
| --- |
| Q 26. 4 matrices M1, M2, M3, M4 of dimensions’ p *x* q, q *x* r, r *x* s, s *x* t respectively can be multiplied in several ways with different number of multiplication because of associative property. If p=10, q=100, r=20, s=5, t=80. Then what will be the minimum number of multiplications required to multiply M1,M2,M3,M4? |

Q 27. Convex Hull of a set of points, in 2D plane, is a convex polygon with minimum area such that each point lies either on the boundary of polygon or inside it. Now given a set of points the task is to find the convex hull of points.

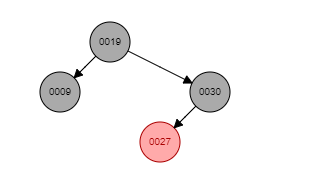
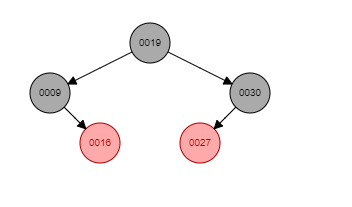
Q 30. Discuss and derive an equation for solving the 0/1 Knapsack problem using dynamic programming method. Design and analyse the algorithm for the same.

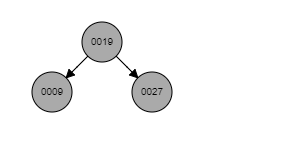
Q31. Construct a Red Black tree by inserting 9,19,30,15,16 and 27 into an initially empty tree and also delete 15,16 and 30 from the tree.





deletion





Q32. Obtain the solution to knapsack problem by Dynamic Programming method n=6, (p1, p2,...p6)=(w1,w2,...w6)=(100,50,20,10,7,3) and m=165.

Q33. Explain the Dynamic Programming (DP) Algorithmic Paradigm? List a few problems which can be solved using the same.

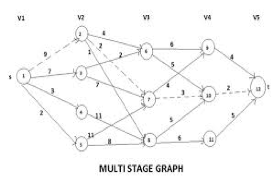
Q34. Differentiate between Greedy approach and dynamic programming problem solving approach.

There are several key differences between the greedy method and dynamic programming, listed below:

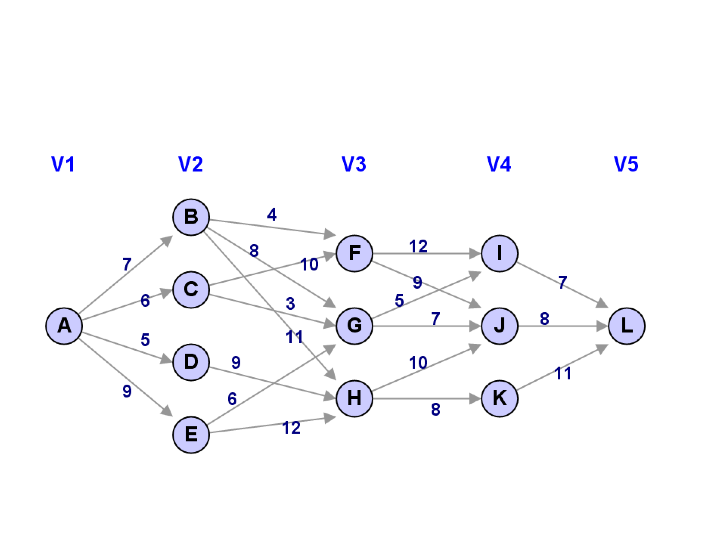
|  |  |
| --- | --- |
| **Greedy Programming** | **Dynamic Programming** |
| * A greedy algorithm chooses the best solution at the moment, in order to ensure a global optimal solution. | * In dynamic programming, we look at the current problem and the current solution to determine whether to make a particular choice or not. We then calculate the optimal choice based on previous problems and solutions. |
| * It is not guaranteed that an optimal solution will be obtained in the greedy method. | * Because of the nature of Dynamic Programming, it is certain that an optimal solution will be generated. |
| * A greedy methodology follows the problem-solving approach of making the locally optimal choice at each step. | * Dynamic programming is an algorithmic approach that uses a recurring formula to calculate new states. |
| * The process of looking back and revisiting previous choices is more costly in terms of memory. | * The memoization solution requires a DP table, which increases memory complexity. |
| * It is easier to engage in greedy procedures than to use Dijkstra’s shortest path algorithm, which takes O(ElogV + VLogV) time. | * Bellman Ford algorithm, which is based on dynamic programming, takes O(V) time. |
| * The greedy approach deterministically obtains its answer by repeatedly selecting a random step in a backward direction and never looking back or changing previous choices. | * Developing a solution top down or bottom up is accomplished by obtaining smaller optimal sub-solutions. |
| * Fractional knapsack is an example of greedy algorithms. | * 0/1 knapsack problem is an example of greedy algorithms. |
| * Every problem can’t be solved by greedy algorithm. | * Every problem can be solved by Dynamic algorithm. |
| * A solution to a specified problem set is contained within the given solution set. | * It is not necessary to insist on a particular set of feasible solutions. |
| * More Efficient because we never look back to other options. | * Less Efficient as compared to a greedy approach becausee it’s required DP table to store the answers of calculated states. |
| * A set of overlapping problems cannot be dealt with. | * A set of overlapping problems can be dealt with. |
| * No memorization is required. | * memorization is required. |
| * A greedy strategy is faster than a dynamic one. | * Compared to greedy programming, it is slower. |
| * Fast results | * Slow results comparatively |
| * Each step is locally optimal. | * Past solutions are used to create new ones. |

Q 35. How Red-Black tree is different from B-Tree. Explain with example.

Q 36. Solve the below graph for the shortest path using multistage graph method with dynamic programming approach.



Q37. Solve the below graph for the shortest path using multistage graph method with dynamic programming approach.



Q38. What is B-Tree? Explain it with its example and rules.

Q39. What is the importance of Convex Hull. Explain with proper example.

Q40. Write down the problem statement of TSP. Explain it with the help of an example.