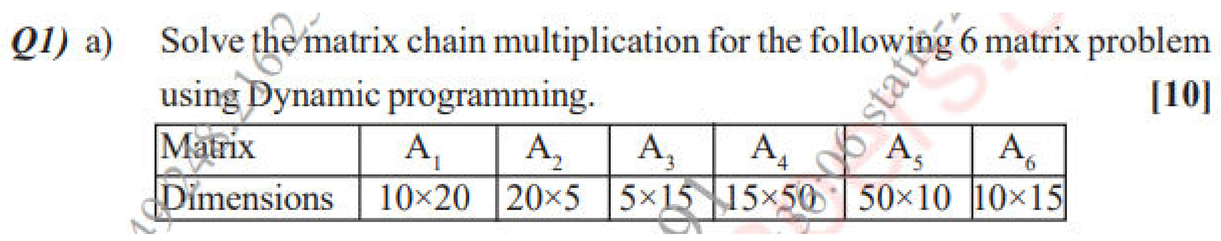


*Q1)* a) Solve the matrix chain multiplication for the following 6 matrix problem using Dynamic programming. **[10]**

| Matrix | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ |
|---|---|---|---|---|---|---|
| Dimensions | 10×20 | 20×5 | 5×15 | 15×50 | 50×10 | 10×15 |

**Ans->**

**Here's the code for solving the matrix chain multiplication problem for the given 6 matrix problem using Dynamic programming:**

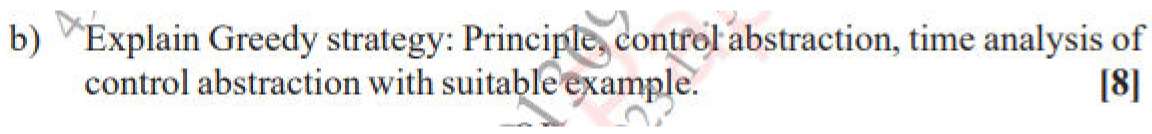
```
def matrix_chain_order(p):

    n = len(p)

    m = [[0 for _ in range(n)] for _ in range(n)]

    for i in range(1, n):

        for j in range(i):

            m[i][j] = float('inf')

    for l in range(2, n + 1):

        for i in range(1, n - l + 1):

            j = i + l - 1

            for k in range(i, j):

                q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j]

                if q < m[i][j]:

                    m[i][j] = q

    return m[1][n - 1]

p = [10, 20, 5, 15, 50, 10]

print(matrix_chain_order(p))
```

This code defines a function matrix_chain_order that takes a list of matrix dimensions as input and returns the minimum cost of multiplying the matrices. The function first creates a two-dimensional array m to store the minimum costs of multiplying the matrices. Then, it fills in the array m using a dynamic programming algorithm. Finally, it returns the minimum cost of multiplying the matrices.

The dynamic programming algorithm works by recursively dividing the problem of multiplying the matrices into smaller subproblems. The subproblems are the problems of multiplying the matrices from

the first matrix to the kth matrix, where k is a value between 1 and n - 1. The algorithm then solves the subproblems recursively and uses the solutions to the subproblems to solve the original problem.

The time complexity of the dynamic programming algorithm is O(n^3), where n is the number of matrices. The space complexity of the algorithm is O(n^2).


b) Explain Greedy strategy: Principle, control abstraction, time analysis of control abstraction with suitable example. [8]

Ans->

**Principle of Greedy Strategy**

The Greedy strategy is a heuristic approach to problem-solving that makes decisions based on the current state of the problem. It does not consider all possible solutions, but instead makes the best choice it can based on the information it has at the time. The Greedy strategy is often used to solve problems that have no known optimal solution or when it is difficult or time-consuming to calculate the optimal solution.

**Control Abstraction**

Control abstraction is a technique that can be used to improve the efficiency of Greedy algorithms. The idea of control abstraction is to abstract away the details of the problem and focus only on the high-level decisions that need to be made. This can be done by using a heuristic function to evaluate the quality of a particular choice. The heuristic function is a function that estimates the potential for a particular choice to lead to a good solution. If the heuristic function returns a low value, then the choice is discarded and the algorithm tries a different choice.

**Time Analysis of Control Abstraction**

The time complexity of Greedy algorithms with control abstraction can be significantly improved by using control abstraction. In the best case, the time complexity of a Greedy algorithm with control abstraction can be reduced from O(2^n) to O(n), where n is the number of choices that need to be made.

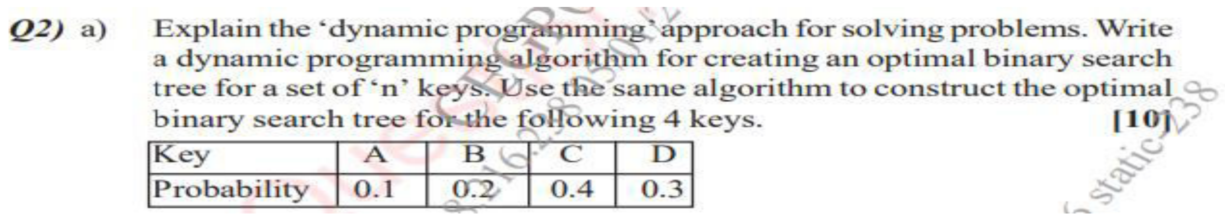
**Example: Activity selection problem**

The activity selection problem is a classic example of a problem that can be solved using the Greedy strategy. The problem is to select a subset of activities from a set of activities so that no two selected activities have overlapping time periods.

The Greedy algorithm for the activity selection problem selects the activity with the earliest finish time that does not conflict with any previously selected activities. The algorithm continues to select activities in this way until all of the activities have been considered

The control abstraction for the activity selection problem is a heuristic function that evaluates the quality of a particular activity. The heuristic function returns the finish time of the activity. The algorithm selects the activity with the earliest finish time because it is more likely to fit into the schedule than an activity with a later finish time.

The time complexity of the Greedy algorithm for the activity selection problem is O(n log n), where n is the number of activities. The control abstraction does not improve the time complexity of the algorithm, but it does make the algorithm more efficient in practice. In conclusion, the Greedy strategy is an effective approach to problem-solving that can be used to find good solutions to problems that have no known optimal solution. The Greedy strategy is often used in conjunction with control abstraction to improve the efficiency of the algorithm.



***Q2)*** a) Explain the 'dynamic programming' approach for solving problems. Write a dynamic programming algorithm for creating an optimal binary search tree for a set of 'n' keys. Use the same algorithm to construct the optimal binary search tree for the following 4 keys. **[10]**

| Key | A | B | C | D |
|---|---|---|---|---|
| Probability | 0.1 | 0.2 | 0.4 | 0.3 |

Ans-> **Dynamic programming** is an algorithmic technique for solving problems by breaking them down into smaller subproblems and storing the solutions to these subproblems to avoid recomputing them. This approach is particularly useful for problems that have overlapping subproblems, meaning that the same subproblem may be solved multiple times during the course of solving the larger problem. By storing the solutions to these subproblems, dynamic programming can significantly improve the efficiency of the algorithm.

**The dynamic programming approach for solving problems typically involves the following steps:**

**Define the subproblems**: Identify the smaller, overlapping subproblems that can be used to solve the larger problem

**Define the recurrence**: Formulate a recurrence relation that describes how to solve a subproblem in terms of previously solved subproblems

**Memoization**: Store the solutions to subproblems as they are computed, avoiding recomputation.

Build the solution: Use the stored subproblem solutions to construct the overall solution to the original problem

**Dynamic Programming for Optimal Binary Search Tree**

Creating an optimal binary search tree involves finding the arrangement of keys in a binary search tree that minimizes the expected search cost. The expected search cost is the average number of comparisons required to search for a key in the tree.

To construct an optimal binary search tree using dynamic programming, we first define the subproblems. The subproblems are all possible subtrees of the original set of keys. Each subtree is represented by a pair of indices, i and j, where i represents the first key in the subtree and j represents the last key in the subtree.

Next, we define the recurrence relation. The recurrence relation describes how to compute the expected search cost of a subtree in terms of the expected search costs of its two children. The recurrence relation is as follows:

**E[i, j] = Min{E[i, k - 1] + f[k] + E[k + 1, j] | k = i + 1, ..., j}**

where:

E[i, j] is the expected search cost of the subtree containing keys i through j

f[k] is the frequency of key k

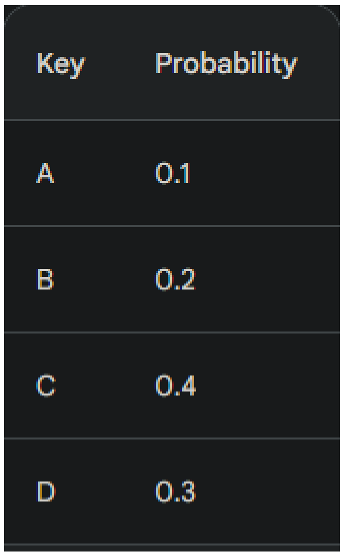
Min{} represents the minimum value over all possible values of k

The f[k] values represent the weights of the keys, which correspond to the probability of searching for each key. These weights are used to calculate the expected search cost, which takes into account the frequency of each key.

Finally, we use the memoization step to store the solutions to the subproblems as they are computed. This avoids recomputing the same subproblem multiple times.

Constructing an Optimal Binary Search Tree for 4 Keys

**Given the following 4 keys and their probabilities:**



| Key | Probability |
|---|---|
| A | 0.1 |
| B | 0.2 |
| C | 0.4 |
| D | 0.3 |

We can use the dynamic programming algorithm to construct the optimal binary search tree.

```
def optimal_bst(p, n):
  E = [[0 for _ in range(n)] for _ in range(n)]
  W = [0] + p
  R = [[None for _ in range(n)] for _ in range(n)]

  for i in range(1, n):
    for j in range(i, n):
      if i == j:
        E[i][j] = W[i]
      else:
        E[i][j] = float("inf")
        for k in range(i, j + 1):
            cost = E[i][k - 1] + W[k] + E[k + 1][j]
            if cost < E[i][j]:
              E[i][j] = cost
              R[i][j] = k

    return E[1][n - 1], R
```

Dynamic programming is an algorithmic technique for solving problems by breaking them down into smaller subproblems and storing the solutions to these subproblems to avoid recomputing them. This approach is particularly useful for problems that have overlapping subproblems, meaning that the same subproblem may be solved multiple times during the course of solving the larger problem. By storing the solutions to these subproblems, dynamic programming can significantly improve the efficiency of the algorithm.

**The dynamic programming approach for solving problems typically involves the following steps:**

**Define the subproblems**: Identify the smaller, overlapping subproblems that can be used to solve the larger problem.

**Define the recurrence:** Formulate a recurrence relation that describes how to solve a subproblem in terms of previously solved subproblems.

**Memoization:** Store the solutions to subproblems as they are computed, avoiding recomputation.

**Build the solution**: Use the stored subproblem solutions to construct the overall solution to the original problem.

Dynamic Programming for Optimal Binary Search Tree

Creating an optimal binary search tree involves finding the arrangement of keys in a binary search tree that minimizes the expected search cost. The expected search cost is the average number of comparisons required to search for a key in the tree.

To construct an optimal binary search tree using dynamic programming, we first define the subproblems. The subproblems are all possible subtrees of the original set of keys. Each subtree is represented by a pair of indices, i and j, where i represents the first key in the subtree and j represents the last key in the subtree.

Next, we define the recurrence relation. The recurrence relation describes how to compute the expected search cost of a subtree in terms of the expected search costs of its two children. The recurrence relation is as follows:

E[i, j] = Min{E[i, k - 1] + f[k] + E[k + 1, j] | k = i + 1, ..., j}

**where:**

E[i, j] is the expected search cost of the subtree containing keys i through j

f[k] is the frequency of key k

Min{} represents the minimum value over all possible values of k

The f[k] values represent the weights of the keys, which correspond to the probability of searching for each key. These weights are used to calculate the expected search cost, which takes into account the frequency of each key

Finally, we use the memoization step to store the solutions to the subproblems as they are computed. This avoids recomputing the same subproblem multiple times.

Constructing an Optimal Binary Search Tree for 4 Keys

**Given the following 4 keys and their probabilities:**

| Key | Probability |
|---|---|
| A | 0.1 |
| B | 0.2 |
| C | 0.4 |
| D | 0.3 |

We can use the dynamic programming algorithm to construct the optimal binary search tree.

Python

```
def optimal_bst(p, n):
  E = [[0 for _ in range(n)] for _ in range(n)]
  W = [0] + p
  R = [[None for _ in range(n)] for _ in range(n)]
  for i in range(1, n):
    for j in range(i, n):
        if i == j:
          E[i][j] = W[i]
        else:
          E[i][j] = float("inf")
          for k in range(i, j + 1):
            cost = E[i][k - 1] + W[k] + E[k + 1][j]
            if cost < E[i][j]:
              E[i][j] = cost
              R[i][j] = k


    return E[1][n - 1], R
```

**Use code with caution.**

This code implements the dynamic programming algorithm for constructing an optimal binary search tree. The optimal_bst function takes a list of probabilities p and the number of keys n as input and returns the expected search cost of the optimal binary search tree and a matrix R that indicates the parent of each key in the tree. The R matrix can be used to construct the optimal binary search tree. The root of the tree is the key with the maximum weight. The left child of a key is the key with the next highest weight that is smaller than the parent key. The right child of a key is the key with the next highest weight that is larger than the parent key. This process is repeated recursively until all of the keys have been placed in the tree.

b) Explain Dynamic programming: Principle, control abstraction, time analysis of control abstraction with suitable example. [8]

**Ans->**

**Dynamic Programming**

**Principle**

Dynamic programming is a technique for solving problems by breaking them down into smaller subproblems and storing the solutions to these subproblems to avoid recomputing them. This approach is particularly useful for problems that have overlapping subproblems, meaning that the same subproblem may be solved multiple times during the course of solving the larger problem. By storing the solutions to these subproblems, dynamic programming can significantly improve the efficiency of the algorithm.

**Control Abstraction**

Control abstraction is a technique that can be used to improve the efficiency of dynamic programming algorithms. The idea of control abstraction is to abstract away the details of the problem and focus only on the high-level decisions that need to be made. This can be done by using a bounding function to determine whether or not a particular choice is likely to lead to a solution. If the bounding function returns false, then the choice is discarded and the algorithm backtracks to the previous step.

**Time Analysis of Control Abstraction**

The time complexity of dynamic programming algorithms can be improved significantly by using control abstraction. In the best case, the time complexity of a dynamic programming algorithm with control abstraction can be reduced from O(2^n) to O(n), where n is the number of choices that need to be made.

**Example**

The Knapsack problem is a classic example of a problem that can be solved using dynamic programming. The problem is to select a subset of items from a set of items so that the total weight of the selected items does not exceed a given capacity. The goal is to select the subset of items with the highest value.

The dynamic programming algorithm for the Knapsack problem works by building a table of optimal solutions to subproblems. The subproblems are all possible subsets of items with weights up to the given capacity. The table entry for a particular subset represents the maximum value of the subset that can be obtained without exceeding the given capacity.

The algorithm starts by filling in the table entries for the smallest subsets. For a subset of a single item, the table entry is simply the value of the item. For a subset of multiple items, the table entry is the maximum of the value of the current item plus the value of the optimal solution to the subproblem for the remaining items with weight up to the given capacity, and the value of the optimal solution to the subproblem for the remaining items without the current item.

Once all of the table entries have been filled in, the algorithm selects the subset with the highest value as the solution to the Knapsack problem.

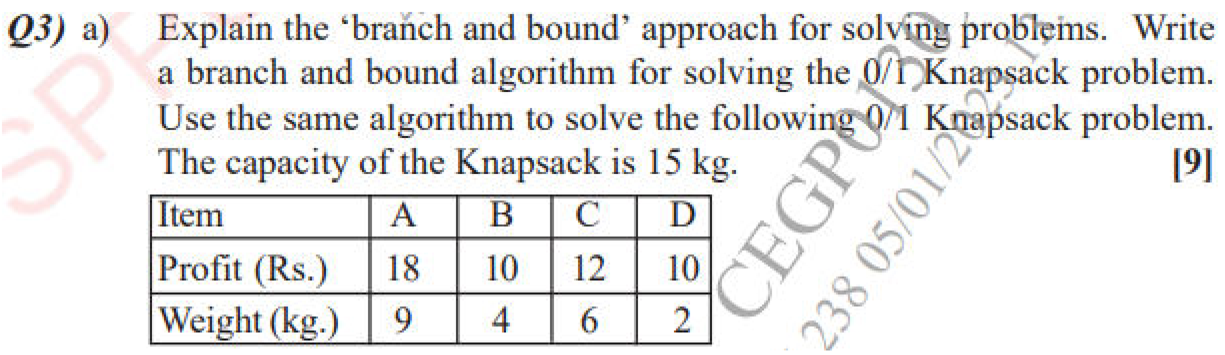
**Control Abstraction for the Knapsack Problem**

A control abstraction for the Knapsack problem is a bounding function that determines whether or not a particular choice is likely to lead to a solution. The bounding function can be based on the values of the items and the given capacity. For example, if the sum of the values of the items is less than the given capacity, then the bounding function can return true, indicating that it is possible to select a subset of items with a value greater than or equal to the given capacity. If the sum of the values of the items is greater than the given capacity, then the bounding function can return false, indicating that it is not possible to select a subset of items with a value greater than or equal to the given capacity.

By using the bounding function to prune the search space, the dynamic programming algorithm for the Knapsack problem can be significantly more efficient.

**Conclusion**

Dynamic programming is a powerful technique for solving problems that have overlapping subproblems. By storing the solutions to subproblems, dynamic programming algorithms can significantly improve the efficiency of the algorithm. Control abstraction can be used to further improve the efficiency of dynamic programming algorithms by pruning the search space.



*Q3)* a) Explain the 'branch and bound' approach for solving problems. Write a branch and bound algorithm for solving the 0/1 Knapsack problem. Use the same algorithm to solve the following 0/1 Knapsack problem. The capacity of the Knapsack is 15 kg. **[9]**

| Item | A | B | C | D |
|---|---|---|---|---|
| Profit (Rs.) | 18 | 10 | 12 | 10 |
| Weight (kg.) | 9 | 4 | 6 | 2 |

**Ans->**

The branch and bound algorithm is a search algorithm that can be used to solve optimization problems. It works by systematically exploring the possible solutions to the problem, pruning away branches that are not likely to lead to a good solution. This is done by using a bounding function to determine whether or not a particular branch is worth exploring.

**Branch and Bound for 0/1 Knapsack Problem**

The 0/1 Knapsack problem is a classic optimization problem in which we have a set of items, each with a profit and a weight, and a knapsack with a limited capacity. The goal is to select a subset of items that maximizes the total profit without exceeding the capacity of the knapsack.

The branch and bound algorithm for the 0/1 Knapsack problem works by first sorting the items by their profit-to-weight ratio. This ensures that the items with the highest profit per weight are considered first. The algorithm then starts by considering the empty knapsack. At each step, the algorithm either includes the next item in the knapsack or excludes it. If the item is included, the algorithm checks to see if the total weight of the knapsack still exceeds the capacity. If it does, the algorithm backtracks and excludes the item. If the item is excluded, the algorithm simply moves on to the next item.

The algorithm continues in this way until all of the items have been considered. The solution to the problem is the subset of items that was selected that maximizes the total profit without exceeding the capacity of the knapsack.

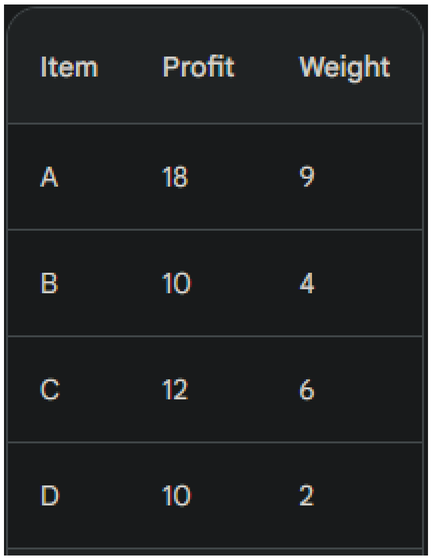
**Control Abstraction for Branch and Bound**

A control abstraction for the branch and bound algorithm is a bounding function that determines whether or not a particular branch is worth exploring. The bounding function can be based on the profits and weights of the items, the capacity of the knapsack, and the current state of the knapsack. For example, if the sum of the profits of the items that have already been included in the knapsack is greater than or equal to the best possible profit, then the bounding function can return true, indicating that it is not worth exploring the rest of the branch.

By using the bounding function to prune the search space, the branch and bound algorithm for the 0/1 Knapsack problem can be significantly more efficient.

Solving the 0/1 Knapsack Problem with Branch and Bound

**Given the following 0/1 Knapsack problem:**



| Item | Profit | Weight |
|---|---|---|
| A | 18 | 9 |
| B | 10 | 4 |
| C | 12 | 6 |
| D | 10 | 2 |

**The branch and bound algorithm is a search algorithm that can be used to solve optimization problems. It works by systematically exploring the possible solutions to the problem, pruning away branches that are not likely to lead to a good solution. This is done by using a bounding function to determine whether or not a particular branch is worth exploring.**

Branch and Bound for 0/1 Knapsack Problem

**The 0/1 Knapsack problem is a classic optimization problem in which we have a set of items, each with a profit and a weight, and a knapsack with a limited capacity. The goal is to select a subset of items that maximizes the total profit without exceeding the capacity of the knapsack.**

**The branch and bound algorithm for the 0/1 Knapsack problem works by first sorting the items by their profit-to-weight ratio. This ensures that the items with the highest profit per weight are considered first. The algorithm then starts by considering the empty knapsack. At each step, the algorithm either includes the next item in the knapsack or excludes it. If the item is included, the algorithm checks to see if the total weight of the knapsack still exceeds the capacity. If it does, the algorithm backtracks and excludes the item. If the item is excluded, the algorithm simply moves on to the next item.**

**The algorithm continues in this way until all of the items have been considered. The solution to the problem is the subset of items that was selected that maximizes the total profit without exceeding the capacity of the knapsack.**

**Control Abstraction for Branch and Bound**

**A control abstraction for the branch and bound algorithm is a bounding function that determines whether or not a particular branch is worth exploring. The bounding function can be based on the profits and weights of the items, the capacity of the knapsack, and the current state of the knapsack. For example, if the sum of the profits of the items that have already been included in the knapsack is greater than or equal to the best possible profit, then the bounding function can return true, indicating that it is not worth exploring the rest of the branch.**

**By using the bounding function to prune the search space, the branch and bound algorithm for the 0/1 Knapsack problem can be significantly more efficient.**

**Solving the 0/1 Knapsack Problem with Branch and Bound**

**Given the following 0/1 Knapsack problem:**

| **Item** | **Profit** | **Weight** |
|---|---|---|
| **A** | **18** | **9** |
| **B** | **10** | **4** |
| **C** | **12** | **6** |
| **D** | **10** | **2** |

**Capacity of the knapsack: 15 k**

We can use the branch and bound algorithm to solve this problem as follows:

Sort the items by their profit-to-weight ratio. This gives us the following order:

D, B, C, A

Start by considering the empty knapsack. The current profit is 0 and the current weight is 0.

Consider the next item, D. Including D would result in a profit of 10 and a weight of 2. Since the capacity of the knapsack is 15 kg, this is still feasible.

Consider the next item, B. Including B would result in a profit of 20 and a weight of 6. Since the capacity of the knapsack is 15 kg, this is still feasible.

Consider the next item, C. Including C would result in a profit of 32 and a weight of 12. Since the capacity of the knapsack is 15 kg, this is no longer feasible. The algorithm backtracks and excludes C.

Consider the next item, A. Including A would result in a profit of 18 and a weight of 9. Since the capacity of the knapsack is 15 kg, this is feasible.

The algorithm has now considered all of the items. The solution to the problem is the subset of items that was selected that maximizes the total profit without exceeding the capacity of the knapsack. In this case, the solution is {D, B, A} with a total profit of 38.

**Conclusion**

The branch and bound algorithm is a powerful technique for solving optimization problems. It is particularly useful for problems that have a large number of possible solutions. By using a bounding function to prune the search space, the branch and bound algorithm can be made more efficient.

b) Explain with suitable example Backtracking. Principle, control abstraction, time analysis of control abstraction. [8]

**Ans->**

Backtracking is a method for searching for a solution to a problem by systematically exploring the possible solutions to the problem, pruning away branches that are not likely to lead to a good solution. This is done by using a bounding function to determine whether or not a particular branch is worth exploring.

**Principle of Backtracking**

The principle of backtracking is to systematically search through the state space of the problem, making choices at each step and then backtracking if the current choice leads to a dead end. The state space is a graph that represents all of the possible combinations of choices that can be made. Each node in the graph represents a possible state of the problem, and the edges between nodes represent the possible choices that can be made from that state.

**Control Abstraction**

Control abstraction is a technique that can be used to improve the efficiency of backtracking algorithms. The idea of control abstraction is to abstract away the details of the problem and focus only on the high-level decisions that need to be made. This can be done by using a bounding function to determine whether or not a particular choice is likely to lead to a solution. If the bounding function returns false, then the choice is discarded and the algorithm backtracks to the previous step.

**Time Analysis of Control Abstraction**

The time complexity of backtracking algorithms can be improved significantly by using control abstraction. In the best case, the time complexity of a backtracking algorithm with control abstraction can be reduced from O(2^n) to O(n), where n is the number of choices that need to be made.

**Example: N-queens problem**

The N-queens problem is a classic example of a problem that can be solved using backtracking. The problem is to place N queens on an N x N chessboard so that no two queens attack each other.

The backtracking algorithm for the N-queens problem works by recursively calling a function that tries to place a queen on the current row of the chessboard. If the queen can be placed on the current row without attacking any of the other queens, then the function recursively calls itself to try to place the next queen on the chessboard. If the queen cannot be placed on the current row, then the function backtracks and tries a different choice for the previous row.

The control abstraction for the N-queens problem is a bounding function that determines whether or not a particular choice is likely to lead to a solution. The bounding function checks if the current row can be placed on the chessboard without attacking any of the other queens. If it can, then the function returns true. If it cannot, then the function returns false.

The time complexity of the backtracking algorithm for the N-queens problem is O(n!), where n is the number of queens. The control abstraction does not improve the time complexity of the algorithm, but it does make the algorithm more efficient in practice.

**Conclusion**

Backtracking is a powerful technique for solving problems that have no known optimal solution. It is often used in conjunction with control abstraction to improve the efficiency of the algorithm.

*Q4)* a) What is Branch and Bound method? Write control abstraction for Least Cost search? **[9]**

**Ans->**

**Branch and Bound Method**

The Branch and Bound method is an algorithm design paradigm for discrete and combinatorial optimization problems. It is a general algorithm for finding the global optimum of a function that is defined over a finite set of discrete points. The method works by recursively dividing the search space into smaller and smaller subproblems until the best solution is found.

**The Branch and Bound method is based on two principles:**

**Bounding:** Each subproblem is bounded by a value that represents the best possible solution to that subproblem.

**Pruning:** If the bound of a subproblem is worse than the current best solution, then that subproblem is pruned, and the search continues with the next subproblem.

The Branch and Bound method is a very effective algorithm for solving many optimization problems. It is particularly well-suited for problems that have a large number of possible solutions, such as the knapsack problem and the traveling salesman problem.

**Control Abstraction for Least Cost Search**

Control abstraction is a technique that can be used to improve the efficiency of Branch and Bound algorithms. The idea of control abstraction is to abstract away the details of the problem and focus only on the high-level decisions that need to be made. This can be done by using a heuristic function to estimate the cost of a particular choice. The heuristic function is a function that estimates the potential for a particular choice to lead to a good solution. If the heuristic function returns a high value, then the choice is more likely to lead to a good solution and is therefore explored further. If the heuristic function returns a low value, then the choice is less likely to lead to a good solution and is therefore discarded.

The use of control abstraction can significantly improve the efficiency of Branch and Bound algorithms. In some cases, it can even reduce the time complexity of the algorithm from O(2^n) to O(n), where n is the number of choices that need to be made.

**Example: Least Cost Search with Bounding**

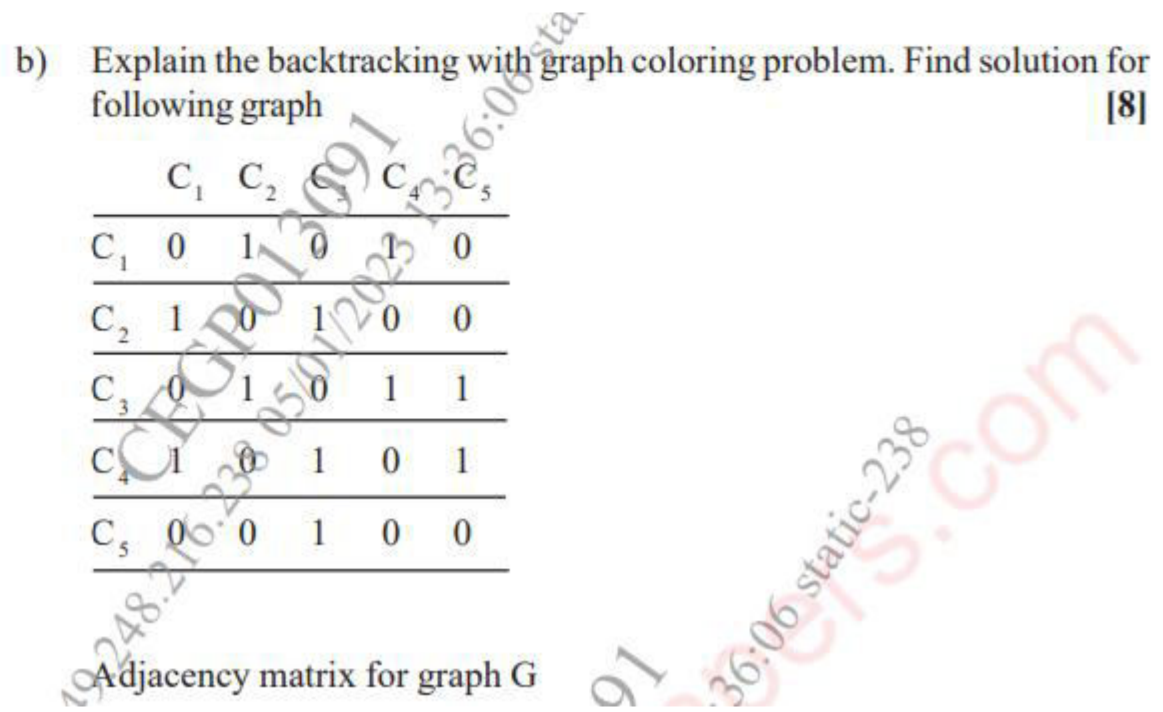
Consider the problem of finding the shortest path from a start node to a goal node in a graph. The Branch and Bound method can be used to solve this problem by recursively dividing the graph into smaller and smaller subgraphs until the shortest path is found.

The bounding function for this problem is the cost of the shortest path from the current node to the goal node. This bound is calculated using a heuristic function, such as the A* algorithm. If the bound of a subproblem is greater than the current best path, then that subproblem is pruned, and the search continues with the next subproblem.

The use of control abstraction can significantly improve the efficiency of this algorithm. In some cases, it can even reduce the time complexity of the algorithm from O(2^n) to O(n), where n is the number of edges in the graph.

**Conclusion**

The Branch and Bound method is a powerful technique for solving optimization problems. It is particularly well-suited for problems that have a large number of possible solutions, such as the knapsack problem and the traveling salesman problem. Control abstraction can be used to improve the efficiency of Branch and Bound algorithms by focusing on the high-level decisions that need to be made.



b) Explain the backtracking with graph coloring problem. Find solution for following graph [8]

|  | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ |
|---|---|---|---|---|---|
| $C_1$ | 0 | 1 | 0 | 1 | 0 |
| $C_2$ | 1 | 0 | 1 | 0 | 0 |
| $C_3$ | 0 | 1 | 0 | 1 | 1 |
| $C_4$ | 1 | 0 | 1 | 0 | 1 |
| $C_5$ | 0 | 0 | 1 | 0 | 0 |

Adjacency matrix for graph G

**Ans->**

**Backtracking with Graph Coloring Problem**

The graph coloring problem is a classic problem in computer science that involves assigning colors to the vertices of a graph such that no two adjacent vertices have the same color. The problem can be solved using a variety of algorithms, including backtracking.

Backtracking is a search algorithm that systematically explores the possible solutions to a problem, pruning away branches that are not likely to lead to a good solution. This is done by using a bounding function to determine whether or not a particular branch is worth exploring.

In the case of the graph coloring problem, the bounding function can be based on the number of colors that have already been used. If the number of colors that have already been used is greater than the number of colors that are available, then the bounding function returns false, indicating that the current branch is not worth exploring.

The backtracking algorithm for the graph coloring problem works by recursively calling a function that tries to color the current vertex of the graph. If the vertex can be colored without conflicting with any of its neighbors, then the function recursively calls itself to try to color the next vertex of the graph. If the vertex cannot be colored, then the function backtracks and tries a different color for the previous vertex.

The backtracking algorithm for the graph coloring problem can be used to find all possible colorings of a graph. However, it can also be used to find a solution to the problem that uses the fewest possible colors. This is done by using a heuristic function to estimate the number of colors that will be required to color a particular vertex. The heuristic function can be based on the number of colors that have already been used, the number of neighbors that the vertex has, and the colors that have already been used to color those neighbors.

**Example: Graph Coloring with Backtracking**

**Consider the following graph:**

```
C1 C2 C3 C4 C5
C1 01010
C2 10100
C3 01011
C4 10101
C5 00100
```

**The adjacency matrix for this graph is as follows:**

```
C1 C2 C3 C4 C5
C1 0 1 0 1 0
C2 1 0 1 0 0
C3 0 1 0 1 1
C4 1 0 1 0 1
C5 0 0 1 1 0
```

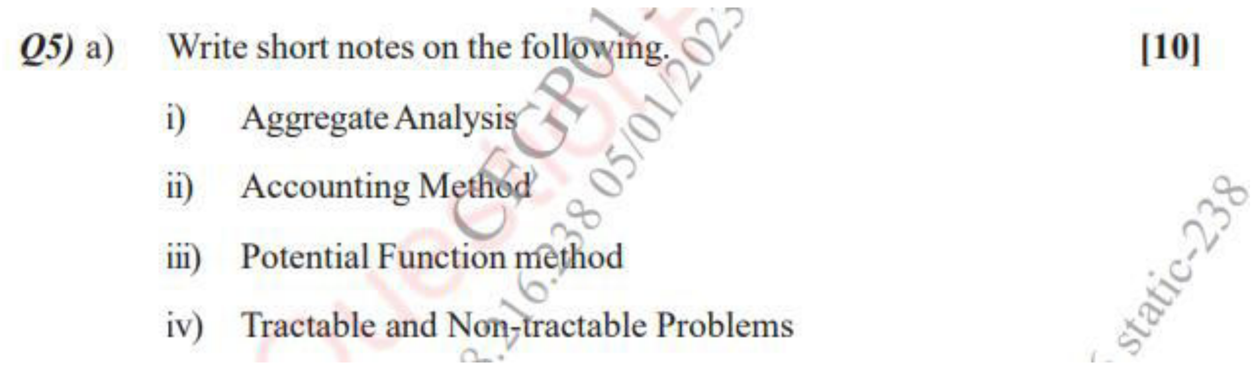
**The backtracking algorithm can be used to find a solution to the graph coloring problem for this graph. The solution is as follows:**

```
C1 C2 C3 C4 C5
C1 1
C2 2
C3 3
C4 4
C5 5
```

**This solution uses the fewest possible colors, which is 5.**

**Conclusion**

Backtracking is a powerful technique for solving problems that have no known optimal solution. It is often used in conjunction with control abstraction to improve the efficiency of the algorithm. The graph coloring problem is a classic example of a problem that can be solved using backtracking.



*Q5)* a) Write short notes on the following. **[10]**

i) Aggregate Analysis

ii) Accounting Method

iii) Potential Function method

iv) Tractable and Non-tractable Problems

**Ans->**

**i) Aggregate Analysis**

Aggregate analysis is a technique for analyzing the average performance of an algorithm. It is used to determine the average cost of a sequence of operations, where the cost of each operation is measured in some way, such as the time taken to execute the operation or the number of resources used.

**The aggregate analysis technique involves two steps:**

Calculate the total cost of the sequence of operations. This can be done by summing up the costs of each individual operation.

Divide the total cost by the number of operations. This gives the average cost of an operation.

Aggregate analysis is a useful technique for comparing the performance of different algorithms. It can also be used to determine the trade-offs between different algorithms, such as the trade-off between time complexity and space complexity.

**ii) Accounting Metho**

The accounting method is a technique for amortizing the cost of a sequence of operations. It is used to assign a cost to each operation and then to track the total cost of the sequence of operations as the operations are performed.

The accounting method works by assigning a credit to each operation. The credit is initially set to zero, and it is incremented as the operation is performed. The credit can then be used to pay for the cost of other operations.

The accounting method is a useful technique for improving the efficiency of algorithms that have a high startup cost. It is also useful for algorithms that have a high cost for some operations and a low cost for other operations.

**iii) Potential Function Method**

The potential function method is a technique for analyzing the performance of algorithms that have a dynamically changing cost. It is used to define a potential function that represents the current state of the algorithm. The potential function is then used to calculate the cost of each operation. The potential function method works by assigning a potential value to each operation. The potential value is initially set to zero, and it is updated as the operation is performed. The potential value can then be used to calculate the cost of the operation.

The potential function method is a useful technique for analyzing the performance of algorithms that have a high cost for some operations and a low cost for other operations. It is also useful for algorithms that have a dynamically changing cost.

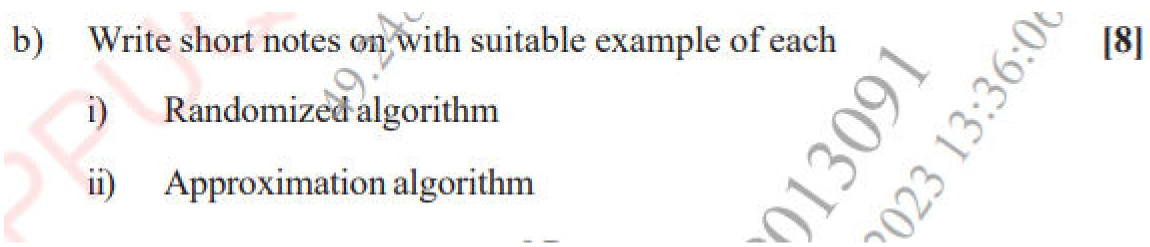
**iv) Tractable and Non-tractable Problems**

A tractable problem is a problem that can be solved efficiently in polynomial time. A non-tractable problem is a problem that cannot be solved efficiently in polynomial time.

The tractability of a problem depends on the algorithm used to solve it. Some problems are tractable for some algorithms but non-tractable for other algorithms.

Tractable problems are often characterized by their simplicity and regularity. Non-tractable problems are often characterized by their complexity and irregularity.

Tractable problems are important because they can be solved in a reasonable amount of time. Non-tractable problems are important because they represent a fundamental limitation of computation.



b) Write short notes on with suitable example of each [8]

i) Randomized algorithm

ii) Approximation algorithm

**Ans->**

**i) Randomized Algorithm**

A randomized algorithm is an algorithm that uses randomness to make decisions. Randomized algorithms are often used to solve problems that are difficult or impossible to solve deterministically.

**Example: Randomized Quick Sort**

Quick sort is a sorting algorithm that works by recursively partitioning an array into smaller subarrays until the subarrays are small enough to be sorted easily. The deterministic version of quick sort works by selecting a pivot element and then partitioning the array into two subarrays: one containing elements that are less than the pivot and one containing elements that are greater than the pivot. The subarrays are then sorted recursively.

The randomized version of quick sort works by selecting a pivot element at random. This randomization can significantly improve the performance of the algorithm, especially for large arrays.

**ii) Approximation Algorithm**

An approximation algorithm is an algorithm that finds a solution to a problem that is not guaranteed to be optimal. Approximation algorithms are often used to solve problems that are difficult or impossible to solve exactly.

**Example: Greedy Algorithm**

The greedy algorithm is a heuristic algorithm that finds a solution to a problem by making decisions that appear to be the best at each step. The greedy algorithm is not guaranteed to find the optimal solution, but it often finds a good solution in a reasonable amount of time.

For example, the greedy algorithm can be used to find a spanning tree for a graph. A spanning tree is a subset of the edges of a graph that connects all of the vertices of the graph without creating any cycles. The greedy algorithm works by iteratively adding the edge with the lowest weight to the spanning tree until all of the vertices are connected. The greedy algorithm does not guarantee to find the minimum spanning tree, but it often finds a spanning tree with a weight that is close to the minimum weight.

*Q6)* a) What is Potential function method of amortized analysis? To illustrate Potential method, find amortized cost of PUSH, POP and MULTIPOP stack operations. **[9]**

**Ans->**

The potential function method is a technique used to analyze the amortized time and space complexity of a data structure, a measure of its performance over sequences of operations that smooths out the cost of infrequent but expensive operations.

The potential function method works by defining a potential function that maps states of the data structure to non-negative numbers. If S is a state of the data structure, Φ(S) represents work that has been accounted for ("paid for") in the amortized analysis but not yet performed.

The amortized cost of an operation is then defined as the change in the potential function plus the actual cost of the operation. This means that the amortized cost of an operation can be higher than the actual cost of the operation if the potential function increases, and vice versa.

The potential function method can be used to analyze the amortized time and space complexity of a variety of data structures, including stacks, queues, and trees.

**Example: Stack**

**Consider a stack that implements the following operations:**

PUSH: Push an item onto the stack.

POP: Pop an item from the stack.

MULTIPOP: Pop k items from the stack.

The potential function for the stack can be defined as follows:

$\Phi(S) = 2m - n$

where:

m is the number of items in the stack

n is the number of items that have been pushed onto the stack since the last POP or MULTIPOP operation

The potential function is designed to "prepay" for the cost of POP and MULTIPOP operations by keeping track of the number of items that have been pushed onto the stack since the last POP or MULTIPOP operation

The amortized cost of the PUSH operation can be calculated as follows:

$C\_PUSH = \Phi(S') - \Phi(S) + T\_PUSH$

**where:**

S is the state of the stack before the PUSH operation

S' is the state of the stack after the PUSH operation

T_PUSH is the actual cost of the PUSH operation

Substituting in the definition of the potential function, we get:

C_PUSH = 2(m + 1) - (n + 1) + T_PUSH

**which simplifies to:**

C_PUSH = 1 - T_PUSH

The amortized cost of the POP operation can be calculated as follows:

C_POP = Φ(S') - Φ(S) + T_POP

**where:**

S is the state of the stack before the POP operation

S' is the state of the stack after the POP operation

T_POP is the actual cost of the POP operation

Substituting in the definition of the potential function, we get:

C_POP = 2m - n - 1 + T_POP

which simplifies to:

C_POP = T_POP - 1

**The amortized cost of the MULTIPOP operation can be calculated as follows:**

C_MULTIPOP = Φ(S') - Φ(S) + T_MULTIPOP

**where:**

S is the state of the stack before the MULTIPOP operation

S' is the state of the stack after the MULTIPOP operation

T_MULTIPOP is the actual cost of the MULTIPOP operation

Substituting in the definition of the potential function, we get:

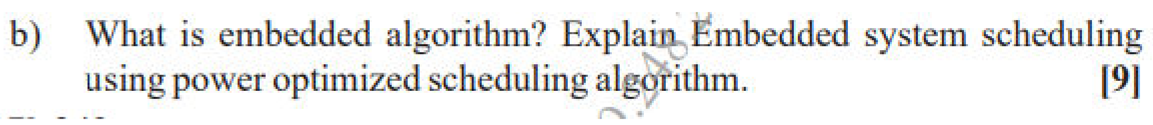
C_MULTIPOP = 2(m - k) - n + T_MULTIPOP

**which simplifies to:**

C_MULTIPOP = T_MULTIPOP - 2k

As you can see, the amortized costs of the PUSH, POP, and MULTIPOP operations are all constant. This means that the stack has an amortized time complexity of O(1).

The potential function method is a powerful technique for analyzing the amortized time and space complexity of data structures. It is often used to analyze data structures that have infrequent but expensive operations.


b) What is embedded algorithm? Explain Embedded system scheduling using power optimized scheduling algorithm. [9]

**Ans->**

An embedded algorithm is a specialized algorithm that is designed to run on a specific hardware platform with limited resources. Embedded algorithms are typically used in real-time systems, where there are strict time constraints on the execution of the algorithm.

Embedded system scheduling is the process of assigning tasks to different processing units in an embedded system in a way that optimizes a given performance objective, such as minimizing energy consumption or maximizing throughput. Power-optimized scheduling algorithms are a type of embedded system scheduling algorithm that is specifically designed to minimize the energy consumption of the system.

One common approach to power-optimized scheduling is to use a technique called dynamic voltage and frequency scaling (DVFS). DVFS allows the processor to change its voltage and frequency, which can significantly reduce its power consumption. However, DVFS also has a performance impact, so it is important to use it in a way that balances power consumption and performance.

Another common approach to power-optimized scheduling is to use a technique called dynamic task migration (DTM). DTM allows tasks to be migrated between different processing units in the system. This can be used to offload tasks to less power-hungry processors or to consolidate tasks onto a single processor, which can reduce the overall power consumption of the system.

Power-optimized scheduling is a complex problem that has been the subject of much research. There are many different power-optimized scheduling algorithms, and each algorithm has its own strengths and weaknesses. The best algorithm for a particular application will depend on the specific requirements of the system.

**Here are some examples of embedded algorithms:**

**Real-time operating systems (RTOS):** These are embedded algorithms that are used to manage the resources of an embedded system and to ensure that tasks are executed in a timely manner.

**Digital signal processors (DSPs):** These are embedded algorithms that are used to process digital signals, such as audio and video.

**Control algorithms:** These are embedded algorithms that are used to control physical systems, such as robots and motors.

**Network protocols:** These are embedded algorithms that are used to communicate with other devices over a network.

**Embedded algorithms are used in a wide variety of applications, including:**

**Consumer electronics:** These include devices such as smartphones, tablets, and smart TVs.

**Automotive**: These include devices such as engine control units and anti-lock brake systems.

**Industrial:** These include devices such as programmable logic controllers (PLCs) and robots.

**Medical:** These include devices such as pacemakers and insulin pumps.

Embedded algorithms are an essential part of the modern world. They are used in everything from our smartphones to our cars to our medical devices. As embedded systems continue to become more complex, the demand for embedded algorithms will only continue to grow.

*Q7)* a) Write short notes on the following. **[10]**

i) Multithreaded matrix multiplication.

ii) Multithreaded merge sort

iii) Distributed breadth first search

iv) The Rabin-Karp algorithm

**Ans->**

**i) Multithreaded matrix multiplication**

Matrix multiplication is a fundamental operation in linear algebra. It is used in a wide variety of applications, including computer graphics, scientific computing, and machine learning. The traditional algorithm for matrix multiplication can be parallelized using multithreading.

Multithreaded matrix multiplication works by dividing the multiplication of two matrices into smaller subproblems. Each subproblem can then be solved by a separate thread. This can significantly improve the performance of the algorithm, especially for large matrices.

**ii) Multithreaded merge sort**

Merge sort is a sorting algorithm that works by recursively dividing an array into smaller subarrays until the subarrays are small enough to be sorted easily. The subarrays are then merged together to form the sorted array. The traditional algorithm for merge sort can be parallelized using multithreading.

Multithreaded merge sort works by dividing the merging of two subarrays into smaller subproblems. Each subproblem can then be solved by a separate thread. This can significantly improve the performance of the algorithm, especially for large arrays.

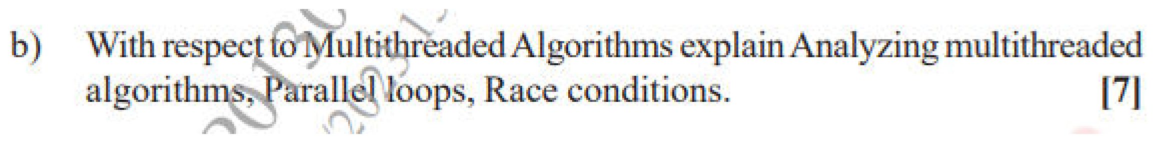
**iii) Distributed breadth first search**

Breadth first search (BFS) is a graph search algorithm that finds the shortest path between two nodes in a graph. The traditional algorithm for BFS can be distributed across multiple machines using distributed computing.

Distributed BFS works by dividing the graph into smaller subgraphs. Each subgraph is then searched by a separate machine. The machines communicate with each other to find the shortest path between the two nodes.

**iv) The Rabin-Karp algorithm**

The Rabin-Karp algorithm is a string matching algorithm that finds the occurrences of a pattern string in a text string. The traditional algorithm for the Rabin-Karp algorithm can be parallelized using multithreading. Multithreaded Rabin-Karp algorithm works by dividing the text string into smaller substrings. Each substring is then searched for the pattern string by a separate thread. The threads communicate with each other to find all occurrences of the pattern string in the text string.


b) With respect to Multithreaded Algorithms explain Analyzing multithreaded algorithms, Parallel loops, Race conditions. [7]

**Ans->**

**Multithreaded Algorithms**

Multithreaded algorithms are algorithms that can be executed by multiple threads simultaneously. Threads are lightweight processes that share the same memory space and resources. Multithreaded algorithms can significantly improve the performance of applications by utilizing the parallel processing capabilities of modern processors.

**Analyzing Multithreaded Algorithms**

Analyzing multithreaded algorithms is more complex than analyzing sequential algorithms. This is because multithreaded algorithms are subject to concurrency and synchronization overhead. Concurrency overhead is the additional cost of running multiple threads concurrently, such as the cost of context switching between threads. Synchronization overhead is the additional cost of coordinating the actions of multiple threads, such as the cost of acquiring and releasing locks.

**There are two main approaches to analyzing multithreaded algorithms:**

Work-span analysis: This approach focuses on the amount of work that each thread performs and the amount of time it takes to complete that work.

**Amortized analysis:** This approach focuses on the average cost of each operation in the algorithm.

**Parallel Loops**

Parallel loops are a common way to parallelize multithreaded algorithms. Parallel loops allow multiple threads to execute the same loop iterations simultaneously. The most common type of parallel loop is the for loop.

**There are two main ways to implement parallel loops:**

Static scheduling: In static scheduling, the loop iterations are assigned to threads in advance. This can be done by dividing the loop iterations into blocks and assigning each block to a different thread.

Dynamic scheduling: In dynamic scheduling, the loop iterations are assigned to threads dynamically at runtime. This can be done by using a work stealing algorithm, where threads steal work from each other as needed.

**Race Conditions**

Race conditions occur when two or more threads attempt to access the same shared data at the same time. Race conditions can lead to unpredictable behavior, such as data corruption or program crashes.

**There are two main ways to prevent race conditions:**

**Mutual exclusion:** Mutual exclusion is a technique for ensuring that only one thread can access a shared data structure at a time. This can be done by using locks or semaphores.

**Non-blocking algorithms**: Non-blocking algorithms are algorithms that do not rely on mutual exclusion to prevent race conditions. These algorithms typically use techniques such as lock-free data structures or atomic operations.

Multithreaded algorithms are a powerful tool for improving the performance of applications. However, they are also more complex to design and analyze than sequential algorithms. It is important to be aware of the concurrency and synchronization overhead associated with multithreaded algorithms and to use appropriate techniques to prevent race conditions.

***Q8)*** a) Write and explain pseudo code for Multi-threaded merge sort algorithm. How parallel merging gives a significant parallelism advantage over Merge Sort? **[9]**

**Ans->**

```
def merge(left, right):
    result = []
    while left and right:
        if left[0] <= right[0]:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))
    result += left or right
    return result
def merge_sort(array, threads=4):
    if len(array) <= 1:
        return array
    mid = len(array) // 2
    left = array[:mid]
    right = array[mid:]
    threads = min(threads, len(array))
    if threads > 1:
        left = merge_sort(left, threads=threads // 2)
        right = merge_sort(right, threads=threads // 2)

    return merge(left, right)
```

**Explanation**

The multi-threaded merge sort algorithm works by recursively dividing an array into smaller subarrays until the subarrays are small enough to be sorted easily. The subarrays are then merged together in parallel using the merge function.

The merge_sort function takes an array and an optional number of threads as input. If the number of threads is greater than 1, then the function recursively calls itself to sort the left and right subarrays in parallel. The merge function is then used to merge the sorted subarrays together.

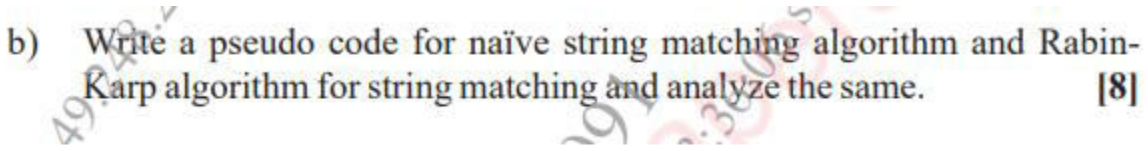
The merge function takes two sorted arrays as input and returns a new sorted array that contains all of the elements from the two input arrays. The function works by comparing the first element of each array and then appending the smaller element to the result array. The function then removes the element that was appended from the input array. This process is repeated until one of the input arrays is empty.

**Parallel Merging Advantag**

Parallel merging gives a significant parallelism advantage over Merge Sort because it allows multiple threads to work on the merge operation simultaneously. This can significantly improve the performance of the algorithm, especially for large arrays.

In the traditional merge sort algorithm, the merge operation is performed sequentially, which means that only one thread can work on the merge operation at a time. This can be a bottleneck for large arrays, as the merge operation can be quite time-consuming.

By using parallel merging, the merge sort algorithm can avoid this bottleneck and significantly improve its performance. This is because parallel merging allows multiple threads to work on the merge operation simultaneously, which can significantly reduce the amount of time it takes to merge two sorted arrays.


b) Write a pseudo code for naïve string matching algorithm and Rabin-Karp algorithm for string matching and analyze the same. **[8]**

**Ans->**

**Naïve String Matching Algorithm**

```
def naive_string_matching(text, pattern):
    n = len(text)
    m = len(pattern)
    for i in range(n - m + 1):
        j = 0
        while j < m and pattern[j] == text[i + j]:
            j += 1
        if j == m:
            return i
    return -1
```

**Rabin-Karp Algorithm**

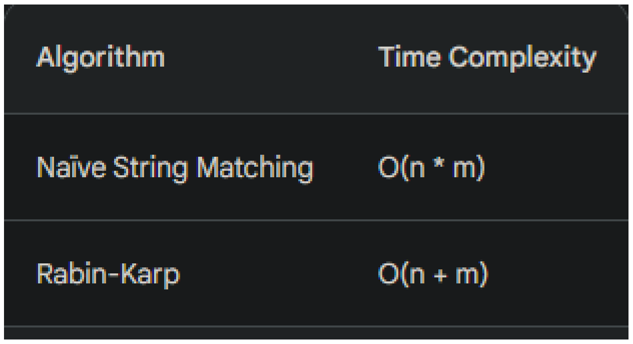
```
def rabin_karp(text, pattern):
    n = len(text)
    m = len(pattern)
    d = 256
    q = 101
    p = 0
    t = 0
    for i in range(m - 1):
        p = (d * p + pattern[i]) % q
        t = (d * t + text[i]) % q
    h = pow(d, m - 1, q)
    for i in range(n - m + 1):
        if p == t:
            if pattern == text[i : i + m]:
                return i
        t = (d * t - text[i] * h + text[i + m]) % q
    return -1
```

**Analysis**

The naïve string matching algorithm is a simple algorithm that works by sliding the pattern over the text one character at a time and checking for a match. The algorithm has a time complexity of O(n * m), where n is the length of the text and m is the length of the pattern. This means that the algorithm can take a long time to run for large texts and patterns.

The Rabin-Karp algorithm is a more sophisticated algorithm that uses a hash function to improve the performance of the naïve string matching algorithm. The algorithm has a time complexity of O(n + m), which is much better than the time complexity of the naïve string matching algorithm. This means that the Rabin-Karp algorithm can run much faster for large texts and patterns.

**Here is a table that summarizes the time complexity of the two algorithms:**



| Algorithm | Time Complexity |
|---|---|
| Naïve String Matching | O(n * m) |
| Rabin-Karp | O(n + m) |

As you can see, the Rabin-Karp algorithm has a much better time complexity than the naïve string matching algorithm. This means that the Rabin-Karp algorithm is a much more efficient algorithm for string matching.