TSP -

```java
// branch and bound

import java.util.*;

public class TSP {
    static int n = 4;
    static int dist[][] = {
        {0, 20, 42, 25},
        {20, 0, 30, 34},
        {42, 30, 0, 10},
        {25, 34, 10, 0}
    };

    // The memoization table
    static int dp[][] = new int[(1 << n)][n];

    // Initialize dp array with -1 (representing unvisited states)
    static {
        for (int i = 0; i < (1 << n); i++) {
            Arrays.fill(dp[i], -1);
        }
    }

    // Recursive function to solve TSP with bitmasking
    static int tsp(int mask, int pos) {
        // If all cities have been visited, return the cost to return to the start city
        if (mask == (1 << n) - 1) {
            return dist[pos][0];
```

```
        }

        // If this state has already been computed, return the result
        if (dp[mask][pos] != -1) {
            return dp[mask][pos];
        }

        int ans = Integer.MAX_VALUE;
        // Try to visit all cities
        for (int city = 0; city < n; city++) {
            // If the city hasn't been visited yet
            if ((mask & (1 << city)) == 0) {
                int newAns = dist[pos][city] + tsp(mask | (1 << city), city);
                ans = Math.min(ans, newAns);
            }
        }

        // Store the result in dp table and return it
        return dp[mask][pos] = ans;
    }

    public static void main(String[] args) {
        int mask = 1;  // Starting at the first city (0th city)
        int pos = 0;   // Initial position is 0th city

        System.out.println("Minimum cost of traveling salesman tour: " + tsp(mask, pos));
    }
}
```

```
// Represent the cities and distances in a matrix dist[][].
// Use a bitmask mask to track visited cities and a memoization table dp to store intermediate results.
// Start from the first city with only it visited (mask = 1, pos = 0).
// Use recursion to explore all unvisited cities, updating the bitmask and calculating the cost.
// Return to the starting city when all cities are visited (mask == (1 << n) - 1).
// Memoize results to avoid redundant calculations.
// Output the minimum cost from the starting point.
// Time - o(n * 2^n) as there are 2^n subsets (masks) and n cities to explore for each state.
// Best - still o(n * 2^n), but memoization can reduce actual computations.
// Space - dp table - o(n * 2^n) to store intermediate results.
// Recursion stack - o(n) for the maximum depth of the recursion.
```

QUICK SORT -

```
public class QuickSort {

  public static void quickSort(int[] arr, int low, int high) {
    if (low < high) {
      int pivot = partition(arr, low, high); // Partition the array
      quickSort(arr, low, pivot - 1);      // Recursively sort left part
      quickSort(arr, pivot + 1, high);     // Recursively sort right part
    }
  }

  public static int partition(int[] arr, int low, int high) {
    int pivot = arr[low]; // Choose pivot as the first element
    int i = low;
    int j = high;
    while (i < j) {
      while (arr[i] <= pivot && i < high) i++;  // Find element > pivot
```

```
        while (arr[j] > pivot && j > low) j--;    // Find element <= pivot
        if (i < j) {
            swap(arr, i, j); // Swap out-of-place elements
        }
    }
    swap(arr, j, low); // Place pivot in its correct position
    return j;         // Return pivot index
}

public static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

public static void main(String[] args) {
    int[] arr = {4, 8, 6, 2, 5, 7, 9, 1, 3};
    int n = arr.length;

    System.out.println("Original Array");
    printArray(arr);

    quickSort(arr, 0, n - 1);
    System.out.println("Sorted Array");
    printArray(arr);
}

public static void printArray(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
```

```
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }
}
```

/*

Algorithm:

1. Choose a pivot (here, the first element of the current range).

2. Partition the array into two parts: elements ≤ pivot and elements > pivot.

3. Recursively apply the same process to the left and right parts of the pivot.

4. Base case: Stop recursion when the range has one or zero elements.

5. Output the sorted array after all recursive calls complete.

Time Complexity:

- **Worst Case:** O(n^2), when the pivot divides the array into extremely unbalanced parts (e.g., already sorted array).

- **Best Case:** O(n log n), when the pivot divides the array into two balanced halves repeatedly.

Space Complexity:

- **Space for Recursion Stack:** O(log n) (average) for balanced partitioning, O(n) (worst case) for unbalanced partitioning.

- **No additional space used for sorting as it's in-place: O(1).**

*/

QUEENS –

```
public public class NQueens {

    // Function to print the solution board
    public static void printSolution(int[][] board) {
```

```
    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board.length; j++) {
            System.out.print(board[i][j] + " ");
        }
        System.out.println();
    }
}

// Function to check if a queen can be placed on board[row][col]
public static boolean isSafe(int[][] board, int row, int col) {
    // Check the column
    for (int i = 0; i < row; i++) {
        if (board[i][col] == 1) {
            return false;
        }
    }

    // Check the left diagonal
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1) {
            return false;
        }
    }

    // Check the right diagonal
    for (int i = row, j = col; i >= 0 && j < board.length; i--, j++) {
        if (board[i][j] == 1) {
            return false;
        }
```

```java
    }

    return true;
}

// Function to solve the N Queens problem using Backtracking
public static boolean solveNQueens(int[][] board, int row) {
    // If all queens are placed
    if (row >= board.length) {
        return true;
    }

    // Try placing the queen in all columns one by one
    for (int col = 0; col < board.length; col++) {
        if (isSafe(board, row, col)) {
            board[row][col] = 1;  // Place the queen

            // Recur to place the queen in the next row
            if (solveNQueens(board, row + 1)) {
                return true;
            }

            // If placing queen in the current column doesn't lead to a solution
            board[row][col] = 0;  // Backtrack
        }
    }

    return false; // If the queen cannot be placed in any column in this row
}
```

```
    public static void main(String[] args) {
        int n = 8;  // Change the value of n for different sizes of the board
        int[][] board = new int[n][n];

        // Solve the N Queens problem
        if (solveNQueens(board, 0)) {
            printSolution(board);
        } else {
            System.out.println("Solution does not exist.");
        }
    }
}
```

/*

Algorithm:

1. Start by placing a queen in the first row, trying each column.

2. For each queen, check if it is safe to place in that column (check column and diagonals).

3. Recursively place queens in subsequent rows.

4. If placing the queen leads to a solution, continue. If not, backtrack (remove the queen and try the next column).

5. Print the solution board when all queens are placed successfully.

Time Complexity:

- Worst Case: O(N!) because there are N possible positions for each queen and we try placing queens in each column recursively.

- Best Case: O(N^2) if the solution is found early (for example, if the board is small or the backtracking prunes the search quickly).

Space Complexity:

- O(N^2) for the board (an N x N matrix to store the placement of queens).

- O(N) for the recursion stack due to the recursive function `solveNQueens`.

- Overall Space Complexity: O(N^2) due to the board storage.

For comparison of time:

- You can compare the time taken by changing the value of `n` (4, 5, 6, 7, 8) and measuring how long it takes for the program to find the solution.

For example, to measure time:

```
   long startTime = System.nanoTime();

   solveNQueens(board, 0);

   long endTime = System.nanoTime();

   System.out.println("Time taken: " + (endTime - startTime) + " nanoseconds.");

*/

 {


}
```

MERGE SORT –

```
public class MergeSort {

   // Recursive function to sort an array using merge sort

   public static void mergeSort(int[] arr, int left, int right) {

      if (left < right) {

         int mid = left + (right - left) / 2; // Find the middle point

         // Recursively sort both halves

         mergeSort(arr, left, mid);

         mergeSort(arr, mid + 1, right);
```

```
        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

// Function to merge two sorted subarrays
public static void merge(int[] arr, int left, int mid, int right) {
    int n1 = mid - left + 1; // Size of left subarray
    int n2 = right - mid;    // Size of right subarray

    // Create temporary arrays
    int[] leftArr = new int[n1];
    int[] rightArr = new int[n2];

    // Copy data to temp arrays
    for (int i = 0; i < n1; i++) leftArr[i] = arr[left + i];
    for (int j = 0; j < n2; j++) rightArr[j] = arr[mid + 1 + j];

    // Merge the temp arrays back into the original array
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k++] = leftArr[i++];
        } else {
            arr[k++] = rightArr[j++];
        }
    }

    // Copy remaining elements of leftArr and rightArr if any
```

```
        while (i < n1) arr[k++] = leftArr[i++];
        while (j < n2) arr[k++] = rightArr[j++];
    }

    // Utility function to print an array
    public static void printArray(int[] arr) {
        for (int num : arr) System.out.print(num + " ");
        System.out.println();
    }

    public static void main(String[] args) {
        int[] arr = {38, 27, 43, 3, 9, 82, 10};
        System.out.println("Original Array:");
        printArray(arr);

        mergeSort(arr, 0, arr.length - 1); // Call mergeSort on the array

        System.out.println("Sorted Array:");
        printArray(arr);
    }
}

/*
Algorithm:
1. Divide the array into two halves until each subarray has one element.
2. Merge adjacent subarrays by sorting them into one sorted array.
3. Repeat this process until the whole array is merged and sorted.

Time Complexity:
```

- **Best, Worst, and Average Case:** O(n log n), as the array is always divided into two halves and merging takes linear time.

Space Complexity:

- **Auxiliary Space:** O(n), as temporary arrays are created during the merge step.

- **Recursion Stack Space:** O(log n), for the recursive calls.

- **Overall Space Complexity:** O(n).

*/

MATRIX –

```
import java.util.Random;

public class MatrixMultiplication {

    // Sequential Matrix Multiplication
    public static void sequentialMatrixMultiplication(int[][] A, int[][] B, int[][] C, int N) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                for (int k = 0; k < N; k++) {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
    }

    // Multithreaded Matrix Multiplication
    public static class MatrixMultiplier extends Thread {
        private int[][] A, B, C;
        private int N, row;
```

```java
    public MatrixMultiplier(int[][] A, int[][] B, int[][] C, int N, int row) {
        this.A = A;
        this.B = B;
        this.C = C;
        this.N = N;
        this.row = row;
    }

    @Override
    public void run() {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
                C[row][j] += A[row][k] * B[k][j];
            }
        }
    }
}

// Function to perform multithreaded matrix multiplication
public static void multithreadedMatrixMultiplication(int[][] A, int[][] B, int[][] C, int N) {
    Thread[] threads = new Thread[N];

    // Create a thread for each row of the result matrix
    for (int i = 0; i < N; i++) {
        threads[i] = new MatrixMultiplier(A, B, C, N, i);
        threads[i].start();
    }

    // Wait for all threads to finish
```

```
    try {
        for (int i = 0; i < N; i++) {
            threads[i].join();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

// Function to initialize a random matrix
public static void initializeMatrix(int[][] matrix, int N) {
    Random rand = new Random();
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            matrix[i][j] = rand.nextInt(10); // Random values between 0 and 9
        }
    }
}

// Function to print the matrix (optional)
public static void printMatrix(int[][] matrix, int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            System.out.print(matrix[i][j] + " ");
        }
        System.out.println();
    }
}
```

```
public static void main(String[] args) {
    int N = 4; // Size of the matrix (N x N)

    int[][] A = new int[N][N];
    int[][] B = new int[N][N];
    int[][] CSequential = new int[N][N];
    int[][] CMultithreaded = new int[N][N];

    // Initialize matrices A and B with random values
    initializeMatrix(A, N);
    initializeMatrix(B, N);

    // Sequential matrix multiplication
    long startTime = System.nanoTime();
    sequentialMatrixMultiplication(A, B, CSequential, N);
    long endTime = System.nanoTime();
    long sequentialTime = endTime - startTime;

    // Multithreaded matrix multiplication
    startTime = System.nanoTime();
    multithreadedMatrixMultiplication(A, B, CMultithreaded, N);
    endTime = System.nanoTime();
    long multithreadedTime = endTime - startTime;

    // Print results (optional)
    // printMatrix(CSequential, N);
    // printMatrix(CMultithreaded, N);

    // Compare the time taken by both methods
```

```
        System.out.println("Time taken for sequential multiplication: " + sequentialTime + " nanoseconds");
        System.out.println("Time taken for multithreaded multiplication: " + multithreadedTime + "
nanoseconds");
    }
}
```

/*

Algorithm:

1. Initialize two matrices A and B with random values.

2. Sequential Matrix Multiplication: Multiply the matrices row by row and column by column.

3. Multithreaded Matrix Multiplication: For each row, create a separate thread that computes the corresponding row in the result matrix.

4. Use join() to ensure the main thread waits for all the threads to finish their computation.

5. Compare the time taken by sequential and multithreaded multiplication.

Time Complexity:

- Sequential Matrix Multiplication: O(N^3) where N is the number of rows/columns in the matrix. Three nested loops iterate over the rows, columns, and elements of the matrix.

- Multithreaded Matrix Multiplication: Time complexity remains O(N^3), but the computation is split across multiple threads, allowing parallel execution. The actual speedup depends on the number of threads and system hardware.

Space Complexity:

- Both methods have space complexity of O(N^2) as we are storing the matrices (A, B, and C) of size N x N.

- Additional space for threads: O(N) for storing the thread references.

Comparison:

- For small values of N (e.g., 4), the difference in time may not be significant. However, as N increases, multithreading can offer better performance by utilizing multiple CPU cores.

- For larger matrices, the multithreaded approach will reduce the overall computation time, but the performance gain depends on the system's available cores and thread management overhead.

*/

KNAPSACK –

```
import java.util.Arrays;

public class Knapsack01 {

    // Recursive helper function with memoization
    public static int helper(int ind, int capacity, int[] wt, int[] val, int[][] dp) {

        // Base case: if at the first item, check if it can fit in the knapsack
        if (ind == 0) {
            if (wt[0] <= capacity) return val[0];
            return 0;
        }

        // If already computed, return the value from dp table
        if (dp[ind][capacity] != -1) return dp[ind][capacity];

        // Do not include the current item
        int nonTake = 0 + helper(ind - 1, capacity, wt, val, dp);

        // Include the current item if it fits
        int take = Integer.MIN_VALUE;
        if (wt[ind] <= capacity) {
            take = val[ind] + helper(ind - 1, capacity - wt[ind], wt, val, dp);
        }

        // Store the result and return the maximum of both choices
        return dp[ind][capacity] = Math.max(take, nonTake);
```

```
    }

    // Main function to solve the 0/1 Knapsack problem
    public static int knapsack(int[] weight, int[] value, int n, int capacity) {
        int[][] dp = new int[n][capacity + 1];
        for (int[] row : dp) Arrays.fill(row, -1); // Initialize dp table with -1

        return helper(n - 1, capacity, weight, value, dp);
    }

    public static void main(String[] args) {
        int[] weight = {1, 3, 4, 5};   // Weights of items
        int[] value = {1, 4, 5, 7};    // Values of items
        int maxWeight = 7;             // Maximum capacity of the knapsack
        int n = weight.length;         // Number of items

        // Output the maximum value that can be achieved
        System.out.println("Maximum value in Knapsack: " + knapsack(weight, value, n, maxWeight));
    }
}

/*
Algorithm:
1. Use recursion with memoization to explore all subsets of items.
2. Base case: If only one item is considered, include it if it fits within the capacity.
3. For each item, calculate two possibilities:
   a. Exclude the item and move to the next.
   b. Include the item if it fits and adjust the remaining capacity.
4. Store intermediate results in a dp table to avoid recomputation.
```

5. Return the maximum value obtained.

Time Complexity:

- **Worst Case:** O(n * capacity), as there are n items and capacity values to evaluate.

- **Best Case:** O(n * capacity), as memoization ensures each state is computed only once.

Space Complexity:

- **DP Table:** O(n * capacity), for storing intermediate results.

- **Recursion Stack:** O(n), as the depth of recursion is equal to the number of items.

- **Overall Space Complexity:** O(n * capacity).

*/

GENETIC –

```
import java.util.Arrays;

import java.util.HashMap;


public class Genetic {


   // Function to perform Partially Mapped Crossover (PMX)

   public static int[] partiallyMappedCrossover(int[] parent1, int[] parent2, int crossoverPoint1, int
crossoverPoint2) {

       int n = parent1.length;  // Length of parent chromosomes

       int[] child = new int[n];  // Initialize child chromosome

       Arrays.fill(child, -1);  // Fill with -1 to indicate unassigned positions


       // Step 1: Copy crossover segment from Parent 1 to child

       for (int i = crossoverPoint1; i <= crossoverPoint2; i++) {

          child[i] = parent1[i];

       }
```

```java
    // Step 2: Create a mapping between Parent 1 and Parent 2 within crossover range
    HashMap<Integer, Integer> mapping = new HashMap<>();
    for (int i = crossoverPoint1; i <= crossoverPoint2; i++) {
        mapping.put(parent2[i], parent1[i]);
    }

    // Step 3: Fill remaining positions in child from Parent 2, resolving duplicates
    for (int i = 0; i < n; i++) {
        if (child[i] == -1) {  // If position is unassigned
            int city = parent2[i];
            while (mapping.containsKey(city)) {  // Resolve duplicates using mapping
                city = mapping.get(city);
            }
            child[i] = city;  // Assign resolved city to child
        }
    }

    return child;  // Return the child chromosome
}

public static void main(String[] args) {
    // Example parent chromosomes (routes for TSP)
    int[] parent1 = {1, 2, 3, 4, 5, 6, 7};  // Parent 1 route
    int[] parent2 = {4, 1, 2, 7, 6, 5, 3};  // Parent 2 route

    // Define crossover points
    int crossoverPoint1 = 2;
    int crossoverPoint2 = 4;
```

```
        // Perform PMX crossover
        int[] child = partiallyMappedCrossover(parent1, parent2, crossoverPoint1, crossoverPoint2);

        // Print results
        System.out.println("Parent 1: " + Arrays.toString(parent1));
        System.out.println("Parent 2: " + Arrays.toString(parent2));
        System.out.println("Child:    " + Arrays.toString(child));
    }
}

/*
Algorithm:
1. Copy the segment of the chromosome between the crossover points from Parent 1 to the child.
2. Create a mapping of values between the segment of Parent 2 and Parent 1.
3. For remaining positions, fill the child with values from Parent 2, resolving duplicates using the mapping.
4. Return the resulting child chromosome.

Time Complexity:
- **Best and Worst Case:** O(n), as each element of the chromosome is processed once.

Space Complexity:
- **Auxiliary Space:** O(n), for the mapping and child chromosome arrays.
- **Overall Space Complexity:** O(n), as the algorithm works in linear space.
*/
```

DINING –

```
import java.util.concurrent.Semaphore;
```

```
public class DiningPhilosophers {

   // Number of philosophers
   static final int numPhilosophers = 5;

   // Create a semaphore for each fork
   static Semaphore[] forks = new Semaphore[numPhilosophers];

   static {
      // Initialize each fork with a semaphore
      for (int i = 0; i < numPhilosophers; i++) {
         forks[i] = new Semaphore(1);
      }
   }

   // Philosopher function
   static class Philosopher implements Runnable {
      private final int id;

      public Philosopher(int id) {
         this.id = id;
      }

      @Override
      public void run() {
         int leftFork = id; // Fork on the left
         int rightFork = (id + 1) % numPhilosophers; // Fork on the right
```

```
while (true) {
   // Thinking
   System.out.println("Philosopher " + id + " is thinking.");
   try {
      Thread.sleep(1000);  // Simulate thinking
   } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
   }

   System.out.println("Philosopher " + id + " is hungry.");

   // Pick up forks in a defined order to avoid deadlock
   if (id % 2 == 0) {
      try {
         forks[leftFork].acquire();
         System.out.println("Philosopher " + id + " picked up left fork.");
         forks[rightFork].acquire();
         System.out.println("Philosopher " + id + " picked up right fork.");
      } catch (InterruptedException e) {
         Thread.currentThread().interrupt();
      }
   } else {
      try {
         forks[rightFork].acquire();
         System.out.println("Philosopher " + id + " picked up right fork.");
         forks[leftFork].acquire();
         System.out.println("Philosopher " + id + " picked up left fork.");
      } catch (InterruptedException e) {
         Thread.currentThread().interrupt();
```

```
            }
        }

        // Eating
        System.out.println("Philosopher " + id + " is eating.");
        try {
            Thread.sleep(1000);  // Simulate eating
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        // Put down forks
        forks[leftFork].release();
        forks[rightFork].release();
        System.out.println("Philosopher " + id + " put down both forks.");
      }
    }
  }

public static void main(String[] args) {
    // Create and start threads for each philosopher
    Thread[] philosopherThreads = new Thread[numPhilosophers];
    for (int i = 0; i < numPhilosophers; i++) {
        philosopherThreads[i] = new Thread(new Philosopher(i));
        philosopherThreads[i].start();
    }

    // Run the simulation for a limited time
    try {
```

```
            Thread.sleep(10000);  // Simulate for 10 seconds
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        // Optionally, interrupt all philosophers (not really necessary for the purpose of this example)
        for (Thread t : philosopherThreads) {
            t.interrupt();
        }
    }
}
```

/*

Algorithm:

1. Initialize a semaphore for each fork to ensure mutual exclusion.

2. Each philosopher thinks, gets hungry, picks up forks, eats, and puts down forks.

3. Philosophers pick forks in a defined order to prevent deadlock:

   - Even-index philosophers pick left fork first, then right.

   - Odd-index philosophers pick right fork first, then left.

4. Simulate thinking and eating with sleep calls.

5. Use semaphores to acquire and release forks to avoid race conditions.

Time Complexity:

- Worst Case: O(1) per philosopher for each fork acquisition/release, as semaphore operations are constant-time.

- Best Case: O(1) as the operations are simple semaphore locks and releases.

Space Complexity:

- Forks Array: O(n) where n is the number of philosophers (5).

- Threads: O(n) as we create one thread per philosopher.

- Overall Space Complexity: O(n), considering the space used by the semaphore array and threads.

*/