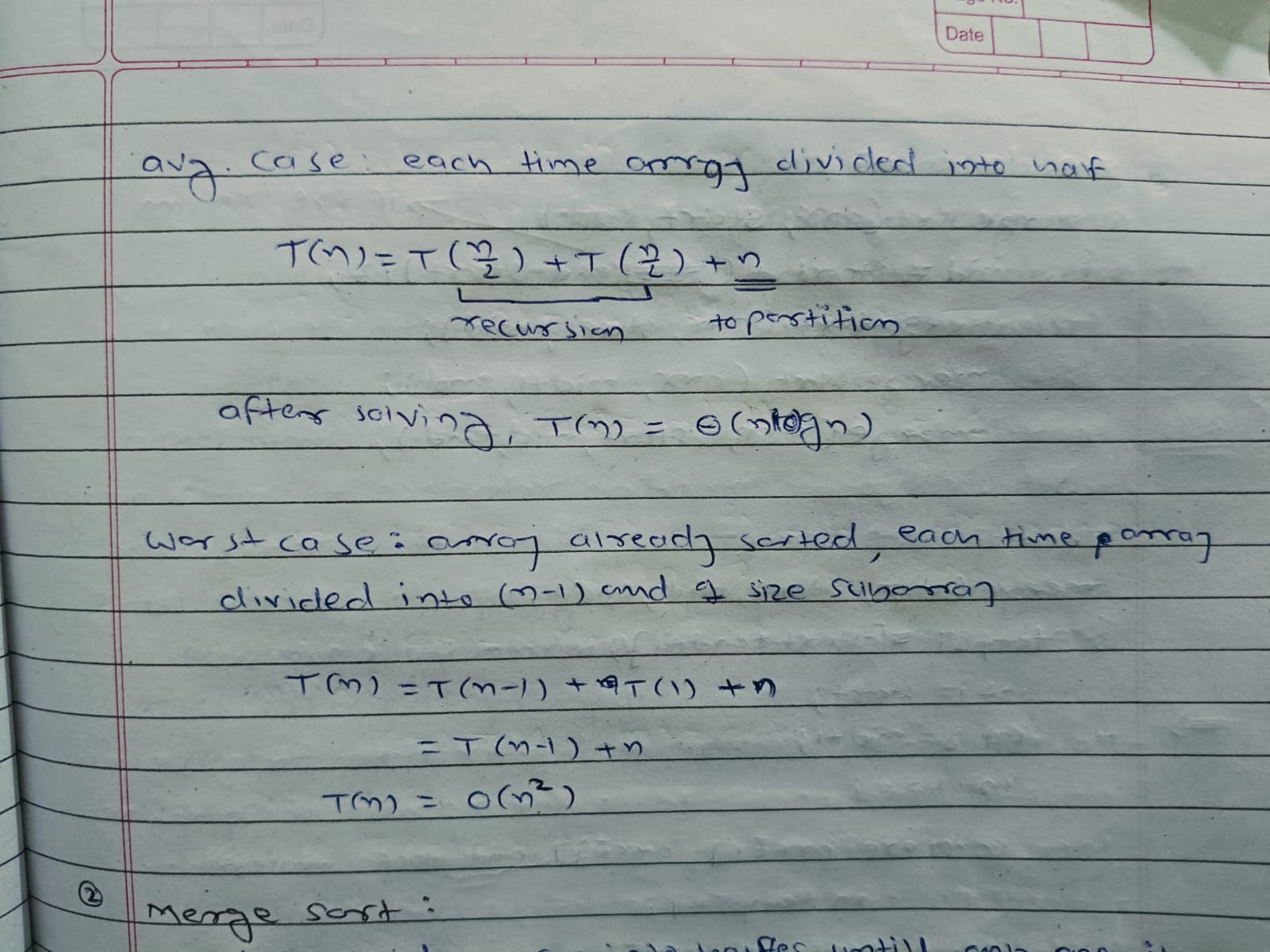
# Assignment 1 - Quick Sort and Merge Sort

**1. Quicksort Algorithm**

1. Choose a pivot element (typically the last element of the array).
2. Partition the array:
   * Elements smaller than the pivot go to the left.
   * Elements larger than the pivot go to the right.
3. Recursively apply the above steps to the subarrays on the left and right of the pivot.
4. Stop when the subarray has one or no elements.



**Time Complexity**:

* Best Case: O(N\*logN) (when the pivot divides the array evenly).
* Average Case: O(N\*logN) (expected for random input).
* Worst Case: O(N^2) (when the pivot is always the smallest or largest element, e.g., sorted array).

**Space Complexity**:

* In-place sorting: O(logN) (due to recursive call stack in balanced partitions).
* Worst case: O(N) (when recursion depth equals array size).

--------------------------------------------------------------------------------------------------------------------------------

**2. Mergesort Algorithm**

1. Divide the array into two halves recursively until each subarray has one element.
2. Merge the two sorted halves back together.
3. Continue merging until the whole array is sorted.

**Time Complexity**:

* Best Case: O(n log n) (always divides the array evenly).
* Average Case: O(n log n)
* Worst Case: O(n log n)

**Space Complexity**:

* Requires O(n) auxiliary space to store temporary subarrays.

# Assignment 2- Quick Sort Vs Randomized Quick Sort

**Randomized Quicksort Algorithm**

1. Randomly select a pivot index and swap it with the last element.
2. Partition the array based on the pivot:
   * Elements smaller than or equal to the pivot go to the left.
   * Elements larger than the pivot go to the right.
3. Recursively apply the above steps to the left and right subarrays.
4. Stop when subarrays have one or zero elements.

**Comparative Analysis**

| **Aspect** | **Standard Quicksort** | **Randomized Quicksort** |
| --- | --- | --- |
| **Pivot Selection** | Deterministic (last element) | Randomized |
| **Best Time** | O(nlogn) | O(nlogn) |
| **Average Time** | O(nlogn) | O(nlogn) |
| **Worst Time** | O(n^2) | (n^2)O (but less likely) |
| **Space Complexity** | O(logn), O(n) | O(logn), O(n) |
| **Use Case** | Simple arrays, performance may degrade for sorted or nearly sorted inputs | Larger datasets requiring randomized performance optimization |

# Assignment 3- Mutation Of Chromosome and TSP

**Write a function to demonstrate Mutation of a chromosome representing solution of Traveling Salesperson Problem (TSP)**



**Explanation**

1. **Chromosome Representation**:
   * The chromosome is represented by an array of integers. Each integer in the array represents a city. For example, [0, 1, 2, 3] represents the solution visiting cities in the order 0→1→2→3.
2. **Mutation Mechanism**:
   * We perform **swap mutation** by selecting two random indices from the chromosome array and swapping their values. This alters the order of cities in the solution.
3. **Random Index Selection**:
   * The Random object is used to select two random indices for mutation. A while loop ensures that the two indices are different.
4. **Mutation Output**:
   * After the mutation, we print the original and mutated chromosome arrays to show the change in the tour.

**Time Complexity**

* **Mutation Operation**: O(1) — The swap mutation is a constant-time operation.

**Space Complexity**

* **Space for Chromosome Copy**: O(n)

# Assignment 4- Crossover of Chromosome and TSP

**Crossover Function**

**Step 1**: A random crossover point (crossoverPoint) is selected.

**Step 2**: The first part of the child chromosome (up to the crossover point) is copied from parent1.

**Step 3**: The second part of the child chromosome (from the crossover point onwards) is copied from parent2, ensuring that no duplicates are introduced. If a city from parent2 is already present in the child, it is skipped and replaced by the next available city from parent2.

**Time Complexiy= O(n)**

# Actual Solution of TSP Using Genetic Algorithm

**Explanation:**

1. **Initial Population**:
   * generateRandomChromosome() generates a random solution by shuffling the cities.
2. **Fitness Function**:
   * calculateTourLength() calculates the total distance of a tour (path).
3. **Crossover**:
   * crossover() applies **single-point crossover**, combining two parent chromosomes into one child while ensuring no duplicate cities in the child.
4. **Mutation**:
   * mutate() randomly swaps two cities in the tour, introducing genetic diversity.
5. **Selection**:
   * We select the two best parents using a **simple selection** method by finding the parents with the shortest paths.
6. **Evolution**:
   * In runTSPGA(), we evolve the population through multiple generations by creating new offspring using crossover and mutation, then replacing the old population with the new one.
7. **Output**:
   * After running the genetic algorithm, the program outputs the best tour and its total distance.

**Time Complexity:**

* **Crossover and Mutation**: Each crossover and mutation operation takes O(n), where n is the number of cities.
* **Fitness Evaluation**: Each fitness evaluation takes O(n).
* **Population Evolution**: For each generation, we perform crossover and mutation for each individual in the population, so the time complexity per generation is O(n\*populationSize)
* **Overall Complexity**: The overall time complexity is O(generations\*populationSize\*n).

**Space Complexity:**

* **Chromosomes**: We need space to store the population, which is O(populationSize\*n).
* **Total Space Complexity**: O(populationSize\*n)

# Assignment 5- Knap Sack Using DP

**Steps of the Algorithm:**

1. **Input**:
   * objects[][]: A 2D array where each row represents an object with its profit and weight. objects[i][0] is the profit and objects[i][1] is the weight of the i-th object.
   * W: The maximum capacity of the knapsack.
2. **Initialization**:
   * Create a DP table KS[][] of size (N+1) x (W+1) where N is the number of objects. This table will store the maximum profit that can be obtained with a given number of objects and weight capacity.
3. **Filling the DP table**:
   * For each object n and weight w:
     + If n == 0 or w == 0, then KS[n][w] = 0 (no objects or no capacity means no profit).
     + Otherwise, for each object, if its weight is less than or equal to the current capacity w, calculate the maximum of:
       - Not including the object: KS[n-1][w]
       - Including the object: KS[n-1][w-weight] + profit
4. **Final Maximum Profit**:
   * The maximum profit for a given capacity W and all objects will be in KS[N][W].
5. **Backtracking to find the selected objects**:
   * Starting from KS[N][W], track which objects are included in the optimal solution by comparing the current value KS[i][j] with KS[i-1][j]. If they are different, the object i is included in the solution.
6. **Output**:
   * Print the **DP Table**.
   * Print the **Maximum Profit**.
   * Print the **Selected Objects**.

**Time Complexity Analysis:**

* **Filling the DP Table**:
  + filling the entire DP table takes O(N \* W) time.
* **Backtracking**:
  + Backtracking to find the selected objects involves going from KS[N][W] to KS[0][0], and in the worst case, this takes O(N) time because in each iteration we either move up one row or backtrack the column.

Therefore, the total **time complexity** of the algorithm is: O(N×W+O(N)

Thus, the overall time complexity is **O(N \* W)**.

**Space Complexity: O(N×W)** *(DP table)+***O(N)** *(selected objects array)*

# Assignment 6- N Queen Using Backtracking

**Algorithm for N-Queens Problem using Backtracking**

1. **Function solveNQueen(n, solution, row)**:
   * If row == n, print the current solution as a valid arrangement of queens and return.
   * For each column i in the current row:
     + If placing a queen at solution[row] = i is safe (checked by isSafe function):
       - Assign solution[row] = i and recursively call solveNQueen(n, solution, row + 1).
       - After the recursive call, backtrack by resetting solution[row].
2. **Function isSafe(solution, row, col)**:
   * Check if placing a queen at position (row, col) is safe by verifying:
     + No other queen is placed in the same column (solution[i] == col).
     + No queens are placed in the diagonals (Math.abs(i - row) == Math.abs(solution[i] - col)).
3. **Function printSolution(solution)**:
   * Print the current solution array, where each element represents the column index of the queen in each row.

**Time Complexity**

* The time complexity of the N-Queens backtracking algorithm is **O(N!)** in the worst case because, in the worst case, we try placing a queen in every row and check all possible columns (which grows factorially as n increases).
* For each row, we check n columns, and there are n rows to consider, leading to an **O(N!)** complexity.

**Space Complexity**

* The space complexity is **O(N)** due to the recursion stack and the space required for storing the solution (the array solution of size n).
* Each recursive call adds a new entry to the recursion stack, which can go as deep as n, hence **O(N)** space complexity.

**Summary of Complexity**

* **Time Complexity**: O(N!)
* **Space Complexity**: O(N)

# Assignment 11- dining philosopher