Travelling Salesman Problem (TSP)

Task 1:Identification of the Problem Statement

The **Traveling Salesman Problem (TSP)** is a well-known puzzle in math and computer science. It asks a simple question: If a salesman has to visit a list of cities and knows the distance between each pair of cities, what is the shortest route that visits all the cities exactly once and returns to the starting point?

At first, this might seem easy, but it becomes much harder as the number of cities increases. For a small number of cities, you could try all possible routes to find the shortest one. But as the number of cities grows, the number of possible routes grows rapidly. For example, if you have just 10 cities, there are already nearly 400,000 possible routes! For 20 cities, this number jumps to about 60 trillion. This huge number of possible routes makes it very hard to find the best solution quickly.

Despite the challenge, the TSP is important because it has many real-world uses. Here are a few examples:

- Delivery and logistics: Companies need to plan the shortest delivery routes to save time and fuel.
- Circuit design: In electronics, TSP helps in designing efficient layouts for circuits.
- DNA sequencing: In biology, TSP is used in organizing genetic data to map genomes more efficiently.
- Travel planning: When planning trips, finding the best route to visit multiple places can be solved using ideas from TSP.

Because the problem is so difficult for larger numbers of cities, exact methods that find the best possible solution are often only useful for small cases. These methods can guarantee the best solution but take too long to solve for many cities.

For larger problems, we often use heuristic methods that don’t always find the perfect solution but can give a good solution quickly. Some examples include:

- Nearest Neighbour Method: This simple method always goes to the closest city next. It’s fast but doesn’t always give the best solution.
- Genetic Algorithms: These use ideas from biology, like evolution, to improve solutions over time.
- Simulated Annealing: A technique that tries to avoid getting stuck in bad solutions by occasionally accepting worse routes in hopes of finding a better one later.

Overall, the TSP is both a challenging and important problem with practical applications in many areas. Even though it’s hard to solve perfectly for large numbers of cities, we can still find good solutions using various methods.

Task 2:Explanation of the problem and Requirement Analysis.

Explanation:

Problem:

The Travelling Salesman Problem (TSP) is a classic puzzle in the world of mathematics and computer science. It's about a salesman who needs to visit a list of cities, and the challenge is to find the shortest possible route that allows the salesman to visit each city exactly once and return to the starting point. This problem might seem simple at first, but as the number of cities increases, finding the best route becomes extremely complicated.The problem can be applied to many real-world scenarios, such as logistics, route planning, manufacturing, and circuit design.

Given:

- A set of n cities.
- A distance matrix D[i][j], where D[i][j] represents the distance between city i and city j.

Find:

- The shortest path (tour) that visits each city exactly once and returns to the starting point.

Example:

Consider 5 cities (A, B, C, D, E) with distances between each pair of cities. The task is to find the shortest possible path that visits all 5 cities once and returns to the starting point.

Challenge:

The TSP is known to be NP-hard, meaning that the problem is computationally difficult to solve as the number of cities increases. The number of possible routes grows factorially with the number of cities, making it impractical to evaluate every possible route for large instances.

In essence, the TSP is about finding the most efficient way to visit a series of locations, which has numerous practical applications in logistics, manufacturing, and network design.

Requirement Analysis for the Traveling Salesman Problem (TSP)

1. Input Requirements

- Number of Cities (n): The total number of cities to be visited.
- Distance Matrix: A 2D matrix D[i][j] representing the distance between city i and city j. The matrix can be symmetric or asymmetric.

2. Output Requirements

- Optimal Tour (Path): The sequence of cities that the salesman should visit, starting and ending at the same city.
- Total Distance (Cost): The total distance of the optimal tour.

3. Constraints

- Visit Each City Once: Every city must be visited exactly one time.
- Return to Starting City: The tour must return to the starting city after visiting all other cities.
- Non-Negative Distances: All distances between cities must be non-negative.

4. Optimization Goals

- Minimize Total Distance: The objective is to minimize the total distance of the tour while visiting all cities and returning to the starting point.

5. Performance Considerations

- Scalability: For small numbers of cities, exact methods (e.g., dynamic programming) can be used, but for larger instances, approximation algorithms or heuristics (e.g., genetic algorithms, ant colony optimization) may be required due to the NP-hard nature of the problem.

6. Domain-Specific Requirements (Optional)

- Real-World Factors: Depending on the application, additional constraints like city accessibility, dynamic conditions (traffic, roadblocks), or time windows may need to be considered.

Task 3:Explanation of algorithmic steps/method/paradigm

1. Greedy Strategy

Overview: The Greedy Strategy aims to build a solution piece by piece, always choosing the next piece that offers the most immediate benefit. For TSP, this typically means visiting the nearest unvisited city.

Steps:

1. Select a Starting City:
    - Choose any city to start the tour.
2. Create a List of Unvisited Cities:
    - Maintain a list of cities that have not yet been visited.
3. Visit the Nearest Unvisited City:
    - From the current city, calculate the distances to all unvisited cities.
    - Choose the nearest unvisited city and move to that city.
4. Mark the City as Visited:
    - Add the newly visited city to the list of visited cities and remove it from the unvisited list.
5. Repeat Until All Cities are Visited:
    - Continue the process of visiting the nearest unvisited city until all cities are visited.
6. Return to the Starting City:
    - After visiting all cities, return to the starting city to complete the tour.
7. Output the Tour and Total Distance:
    - The result will be the order of cities visited and the total distance of the tour.

Example:

- Cities: A, B, C, D
- Distances:
    - A-B: 10, A-C: 15, A-D: 20
    - B-C: 35, B-D: 25
    - C-D: 30

Starting from A, the nearest city might be B (10), then D (25), then C (35), and finally return to A.

2. Dynamic Programming

Steps Explained Simply:

1. Define the Problem:
    - You have a list of cities and the distances between them. The goal is to find the shortest path that visits each city once and goes back to the starting point.
2. Create a Table:
    - Set up a table dp where dp[i][j] represents the shortest distance to visit all cities in the first i cities, ending at city j.
3. Initialization:
    - Start with the distance from the starting city to itself, which is zero: dp[1][0]=0dp[1][0] = 0dp[1][0]=0
    - This means if we are at the starting city (let's say city A), the cost is zero because we haven't moved.
4. Fill the Table:
    - For each possible number of cities visited:

        - For each city that could be the last visited city:
            - For each other city that could have been visited before this last city:
                - Update the distance: $dp[i][j]=\min(dp[i-1][k]+distance(k,j))$
                - Here, k is any other city that was visited before j. This means we are checking all possible paths that lead to city j and keeping track of the shortest one.

5. Calculate the Final Result:
    - After filling in the table, find the shortest route that returns to the starting city:
        - Check the distances for all cities being the last visited city and add the distance back to the starting city.
        - This gives you the minimum distance for the complete tour.
6. Return the Result:
    - The value you calculate in this step is the shortest route for the TSP.

Example:

Imagine you have three cities: A, B, and C with the following distances:

- A to B: 10
- A to C: 15
- B to C: 20

1. Start with the distance from A to A, which is zero.
2. Fill the table by considering routes that go from A to B, A to C, and then back to A through different combinations.
3. Finally, calculate the distance for routes that cover all cities and return to A.

In addition to using a **Greedy strategy** and **Dynamic programming** to solve the

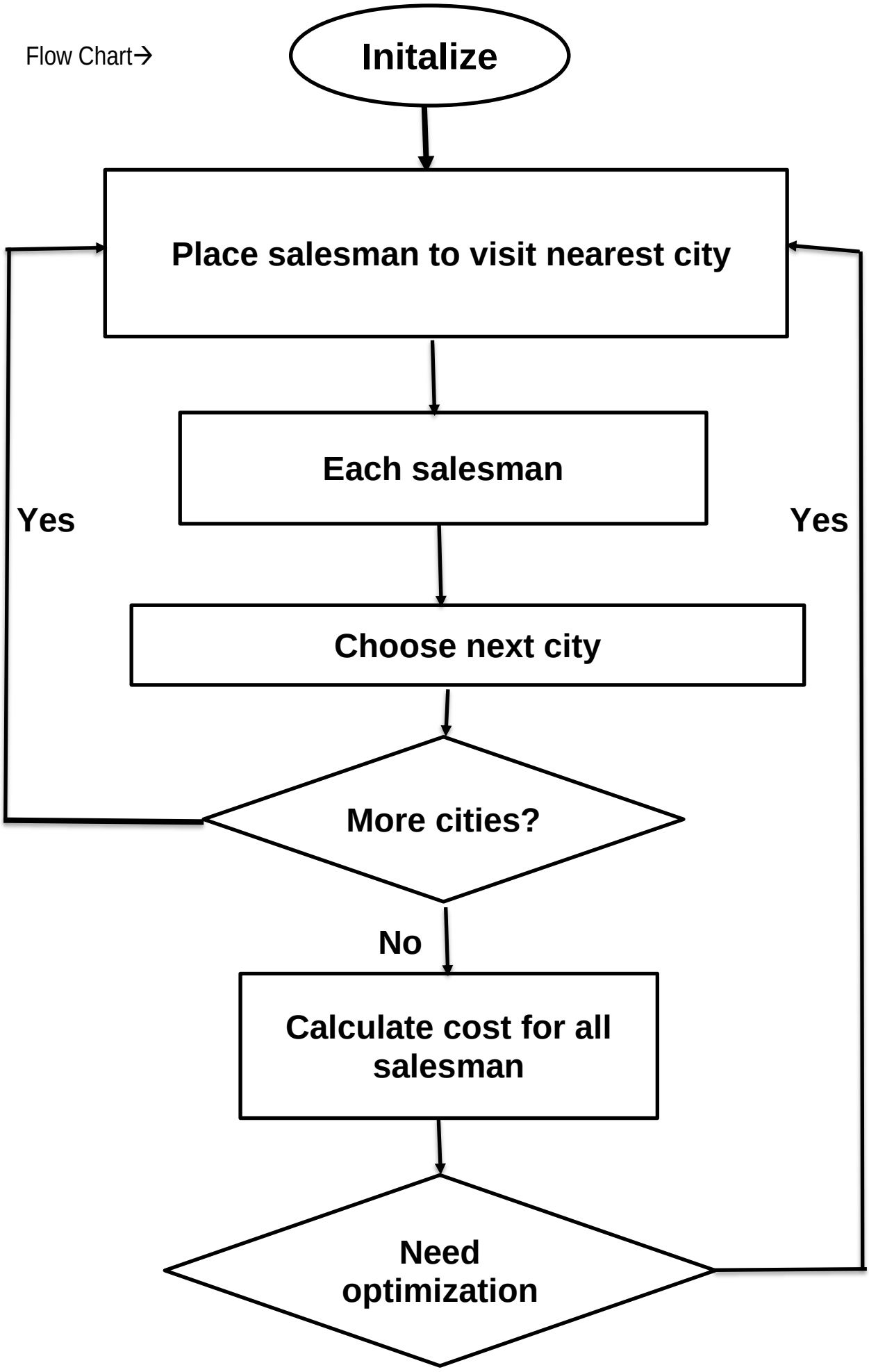
Traveling Salesman Problem, there are also other algorithms that can find approximations to the shortest route.

These algorithms are popular because they are much more effective than to actually check all possible solutions, but as with the greedy algorithm above, they do not find the overall shortest route.

Algorithms used to find a near-optimal solution to the Traveling Salesman Problem include:

- **2-opt Heuristic**: An algorithm that improves the solution step-by-step, in each step removing two edges and reconnecting the two paths in a different way to reduce the total path length.
- **Genetic Algorithm**: This is a type of algorithm inspired by the process of natural selection and use techniques such as selection, mutation, and crossover to evolve solutions to problems, including the TSP.
- **Simulated Annealing**: This method is inspired by the process of annealing in metallurgy. It involves heating and then slowly cooling a material to decrease defects. In the context of TSP, it's used to find a near-optimal solution by exploring the solution space in a way that allows for occasional moves to worse solutions, which helps to avoid getting stuck in local minima.
- **Ant Colony Optimization**: This algorithm is inspired by the behaviour of ants in finding paths from the colony to food sources. It's a more complex probabilistic technique for solving computational problems which can be mapped to finding good paths through graphs.

Flow Chart→


Initalize
Place salesman to visit nearest city
Each salesman
Yes
Yes
Choose next city
More cities?
No
Calculate cost for all salesman
Need optimization

Task 4: Explanation Of Example With Time & Space Complexity

The Traveling Salesman Problem (TSP) is a well-known optimization problem in which a salesman must visit a set of cities, exactly once, and return to the starting city, while minimizing the total travel distance. It is an NP-hard problem, meaning no known efficient algorithm solves it for all instances.

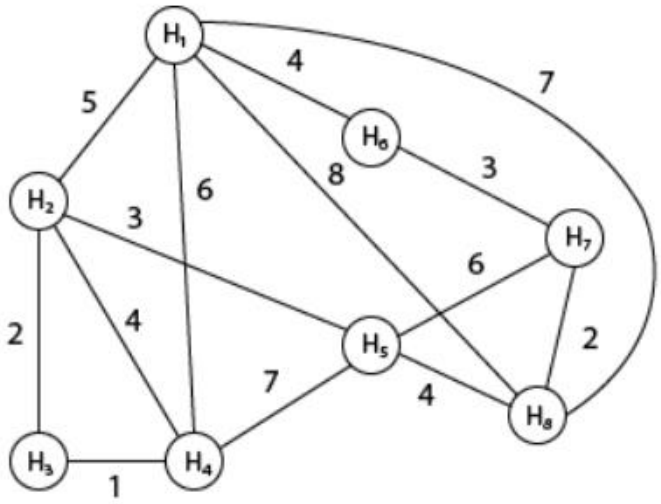
The traveling salesman problems abide by a salesman and a set of cities. The salesman has to visit every one of the cities starting from a certain one (e.g., the hometown) and to return to the same city. The challenge of the problem is that the traveling salesman needs to minimize the total length of the trip.

Suppose the cities are x1 x2..... xn where cost cij denotes the cost of travelling from city xi to xj. The travelling salesperson problem is to find a route starting and ending at x1 that will take in all cities with the minimum cost.

Example of TSP:

A newspaper agent daily drops the newspaper to the area assigned in such a manner that he has to cover all the houses in the respective area with minimum travel cost. Compute the minimum travel cost.

The area assigned to the agent where he has to drop the newspaper is shown in fig:



Solution: The cost- adjacency matrix of graph G is as follows:

costij =

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

The tour starts from area H1 and then select the minimum cost area reachable from H1.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H_1)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H6 because it is the minimum cost area reachable from H1 and then select minimum cost area reachable from H6.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
(H6)	4	0	0	0	0	0	(3)	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H7 because it is the minimum cost area reachable from H6 and then select minimum cost area reachable from H7.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
(H6)	4	0	0	0	0	0	(3)	0
(H7)	0	0	0	0	6	3	0	(2)
H_8	7	0	0	0	4	0	2	0

Mark area H8 because it is the minimum cost area reachable from H8.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
(H6)	4	0	0	0	0	0	(3)	0
(H7)	0	0	0	0	6	3	0	(2)
(H8)	7	0	0	0	(4)	0	2	0

Mark area H5 because it is the minimum cost area reachable from H5.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
(H5)	0	(3)	0	7	0	0	6	4
(H6)	4	0	0	0	0	0	(3)	0
(H7)	0	0	0	0	6	3	0	(2)
(H8)	7	0	0	0	(4)	0	2	0

Mark area H2 because it is the minimum cost area reachable from H2.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
(H2)	5	0	(2)	4	3	0	0	0
H3	0	2	0	1	0	0	0	0
H4	6	4	1	0	7	0	0	0
(H5)	0	(3)	0	7	0	0	6	4
(H6)	4	0	0	0	0	0	(3)	0
(H7)	0	0	0	0	6	3	0	(2)
(H8)	7	0	0	0	(4)	0	2	0

Mark area H3 because it is the minimum cost area reachable from H3.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
(H2)	5	0	(2)	4	3	0	0	0
(H3)	0	2	0	(1)	0	0	0	0
H4	6	4	1	0	7	0	0	0
(H5)	0	(3)	0	7	0	0	6	4
(H6)	4	0	0	0	0	0	(3)	0
(H7)	0	0	0	0	6	3	0	(2)
(H8)	7	0	0	0	(4)	0	2	0

Mark area H4 and then select the minimum cost area reachable from H4 it is H1.So, using the greedy strategy, we get the following.

4 3 2 4 3 2 1 6

H1 → H6 → H7 → H8 → H5 → H2 → H3 → H4 → H1.

Thus the minimum travel cost = 4 + 3 + 2 + 4 + 3 + 2 + 1 + 6 = 25

Time & Space Complexity:

1. Brute Force (Naive Approach):

- How it works: Try all possible routes and choose the one with the least travel cost.
- Time Complexity: O(n!) (too slow for large inputs)
- Space Complexity: $O(n^2)$ (for storing distances between houses)

2. Dynamic Programming (Efficient Approach):

- How it works:
    - We break down the problem into smaller subproblems (subsets of houses) and solve them step by step, storing the results so that we don't have to calculate them again.
    - The dynamic programming approach to TSP is called the Held-Karp algorithm.
    - For example, it calculates the shortest path for visiting groups of houses first, and then uses that information to find the shortest path for the full set of houses.
    - It uses a bit masking technique to represent which houses have been visited.
- Time Complexity: $O(n^2 * 2^n)$, which is much better than brute force but still gets slow as the number of houses increases.
- Space Complexity: $O(n * 2^n)$ for storing the results of subproblems.

3. Greedy Method (Fast but not Always Optimal):

- How it works:
    - The greedy method makes decisions step by step, picking the shortest immediate travel option (the house that is closest to the current location).
    - The idea is to always go to the nearest unvisited house.

    - However, this doesn’t always guarantee the best overall solution—it’s like taking the locally best option without thinking of the whole picture.

- Time Complexity: $O(n^2)$, much faster because it just looks at the next closest house.
- Space Complexity: $O(n^2)$ for storing the distances between houses.

Comparison of Approaches:

Method	How it works	Time Complexity	Space Complexity
Brute Force	Try every possible route	O(n!)	$O(n^2)$
Dynamic Programming	Break the problem into smaller parts and use past results (Held-Karp algorithm)	$O(n^2 * 2^n)$	O(n * 2^n)
Greedy Method	Go to the nearest house at each step	$O(n^2)$	$O(n^2)$

Which One to Use?

- Brute Force: Not practical for more than a few houses, as it takes too long.
- Dynamic Programming (DP): Gives the optimal solution but is still a bit slow for large inputs. Works well for up to 20-30 houses.
- Greedy Method: Very fast, but may not give the best solution. Works well if you need a quick solution for many houses, but you should be okay with some inaccuracy.