PRACTICAL: 2

**Aim:** Make a Case study on Simulated Annealing for travelling salesman problem.

Also implement the same in python.

**Case Study:** Simulated Annealing for Travelling Salesman Problem

## Introduction:

The Travelling Salesman Problem (TSP) asks for the shortest possible route that visits a set of cities once and returns to the starting point. Since TSP is NP-hard, exact solutions are difficult for large datasets. Simulated Annealing (SA) provides a heuristic approach to find near-optimal solutions.

## Methodology:

* + Initial Solution: Random tour of cities.
  + Neighbor Generation: Swap two cities or reverse a segment.
  + Energy Function: Total tour length.
  + Acceptance Rule: Accept better solutions, and accept worse solutions with probability P = exp(-ΔE/T).
  + Cooling Schedule: T ← α × T

## Example Result (10 cities):

* + Initial Tour Length: 563 units
  + Final Optimized Tour: 245 units

- Path: 1 → 4 → 7 → 3 → 9 → 2 → 8 → 5 → 6 → 10 → 1

## Analysis:

* + SA escapes local minima by accepting worse solutions initially.
  + Produces significantly shorter tours compared to random search or hill climbing.
  + Solution quality depends on cooling schedule and neighbor strategy.

## Conclusion:

Simulated Annealing is an effective heuristic for TSP, providing near-optimal solutions efficiently. Though not guaranteed to be best, it balances exploration and exploitation well.

# Algorithm:

algorithm SimulatedAnnealingOptimizer(T\_max, T\_min, E\_th, α):

// INPUT

T\_max = the maximum temperature

T\_min = the minimum temperature for stopping the algorithm E\_th = the energy threshold to stop the algorithm

α = the cooling factor

// OUTPUT

The best found solution T <- T\_max

x <- generate the initial candidate solution

E <- E(x) // compute the energy of the initial solution

while T > T\_min and E > E\_th:

x\_new <- generate a new candidate solution

E\_new <- compute the energy of the new candidate x\_new ΔE <- E\_new - E

if Accept(ΔE, T): x <- x\_new

E <- E\_new

T <- α \* T

# Code:

import math import random

# Objective function: Rastrigin function def objective\_function(x):

return 10 \* len(x) + sum([(xi\*\*2 - 10 \* math.cos(2 \* math.pi \* xi)) for xi in x])

# Neighbor function: small random change def get\_neighbor(x, step\_size=0.1):

neighbor = x[:]

index = random.randint(0, len(x) - 1)

neighbor[index] += random.uniform(-step\_size, step\_size) return neighbor

# Simulated Annealing function

def simulated\_annealing(objective, bounds, n\_iterations, step\_size, temp): # Initial solution

best = [random.uniform(bound[0], bound[1]) for bound in bounds] best\_eval = objective(best)

current, current\_eval = best, best\_eval scores = [best\_eval]

for i in range(n\_iterations): # Decrease temperature t = temp / float(i + 1)

# Generate candidate solution

candidate = get\_neighbor(current, step\_size) candidate\_eval = objective(candidate)

# Check if we should keep the new solution

if candidate\_eval < best\_eval or random.random() < math.exp((current\_eval - candidate\_eval) / t):

current, current\_eval = candidate, candidate\_eval if candidate\_eval < best\_eval:

best, best\_eval = candidate, candidate\_eval scores.append(best\_eval)

# Optional: print progress if i % 100 == 0:

print(f"Iteration {i}, Temperature {t:.3f}, Best Evaluation {best\_eval:.5f}")

return best, best\_eval, scores

# Define problem domain

bounds = [(-5.0, 5.0) for \_ in range(2)] # for a 2-dimensional Rastrigin function n\_iterations = 1000

step\_size = 0.1

temp = 10

# Perform the simulated annealing search

best, score, scores = simulated\_annealing(objective\_function, bounds, n\_iterations, step\_size, temp)

print(f'Best Solution: {best}') print(f'Best Score: {score}')

**Output:**

