Problem C. *(Adopted from CMU 21-387)*
Euclidean traveling salesman problem (TSP): Given a set of points $\vec{x}_1, \ldots, \vec{x}_n \in \mathbb{R}^2$ representing the positions of cities on a map, we wish to visit each city exactly once while minimizing the total distance traveled. While Euclidean TSP is NP-hard, simulated annealing method provides a practical way to approximate its solution.

Solution
Let us define the permutation $\pi : I_n \to I_n$ where $I_n = \{1, 2, \ldots, n\}$. A desired solution is a tour consisting of given $\vec{x_i} \in \mathbb{R}^2$, thus the feasible set of our problem should be in S_n which is the set of all permutations on I_n. This implies:

$$\pi \in S_n$$

For the weight, let us define $f : S_n \to \mathbb{R}$ that calculates the Euclidean distance for a given tour π along with the distance between the last point back to the initial point in the tour.

$$f(\pi) = \|x_{\pi_n} - x_{\pi_1}\|_2 + \sum_{i=1}^{n-1} \|x_{\pi_{i+1}} - x_{\pi_i}\|_2$$

Writing in optimization term, our objective now is:

$$\pi^* \equiv \underset{\pi \in S_n}{\arg\min} \|x_{\pi_n} - x_{\pi_1}\|_2 + \sum_{i=1}^{n-1} \|x_{\pi_{i+1}} - x_{\pi_i}\|_2$$

A simple shuffle function to generate a random tour that reach each city exactly once.

Algorithm 1 Generate a random tour

```
function RandomTour(n):
    tour ← [ ]
    for i ← 1 to n do
        tour[i] ← i
    for i ← n downto 1 do
        j ← randInt(1, i)
        swap(tour[i], tour[j])
    return tour
```

Next up is a simple function to evaluate the total distance of a given tour.

Algorithm 2 Calculate total distance of tour

```
function TourDistance(tour, cityCoordinates):
    totalDistance ← 0
    n ← length(tour)
    for i ← 0 to n − 2 do
        p1 ← cityCoordinates[tour[i]]
        p2 ← cityCoordinates[tour[i + 1]]
        totalDistance ← totalDistance + distance(p1, p2)
    p1 ← cityCoordinates[tour[n − 1]]
    p2 ← cityCoordinates[tour[0]]
    totalDistance ← totalDistance + distance(p1, p2)
    return totalDistance
```

In each step, we will generate a neighbor tour by performing a slight modification to the current tour, simply swapping two random cities from the current tour will be sufficient to implement the GenerateNeighbor function.

Algorithm 3 Generate neighbor tour

```
function GenerateNeighborTour(tour):
    newTour ← tour
    n ← length(newTour)
    i ← randInt(0, n − 1)
    repeat
        j ← randInt(0, n − 1)
    until i ≠ j
    swap(newTour[i], newTour[j])
    return newTour
```

Last but not least, we apply the simulated annealing method to approximate π^*.

Algorithm 4 TSP Simulated Annealing

```
function Simulated_Annealing(cities, T_0, α):
    T ← T_0
    n ← length(cities)
    currentTour ← RandomTour(n)
    bestTour ← currentTour

    for i ← 1, 2, 3, . . . do
        neighborTour ← GenerateNeighbor(currentTour)
        currentEnergy ← TourDistance(currentTour, cities)
        neighborEnergy ← TourDistance(neighborTour, cities)

        Δf ← neighborEnergy − currentEnergy
        if Δf < 0 then                                  ▷ Objective improved at neighborTour
            currentTour ← neighborTour
        else
            if random_uniform(0, 1) < e^(−Δf/T) then
                currentTour ← neighborTour              ▷ True with probability e^(−Δf/T)

        d_1 ← TourDistance(currentTour, cities)
        d_2 ← TourDistance(bestTour, cities)
        if d_1 < d_2 then
            bestTour ← currentTour
        T ← T × α                                       ▷ Cool the temperature

    return bestTour
```

■