CSE 321 Introduction to Algorithm Design

Tuba TOPRAK- 161044116

Hw5

Q1-)

1-) Divide

Sort the drones by their x-coordinates.
Recursively divide the sorted list into two halves, roughly equal in size.

2- Conquer :

Recursively find the minumum distance in the left and right halves.
Let d-left and d-right be minumum distances found in the respective halves.
Determine the minumum distance d-min among d-left d-right, and the minumum distance between drones in the middle strip.
Return d-min as the minumum distance for the current problem.

3- middle Strip:

Construct a vertical strip with width d-min centered around the middle vertical line that divides the two halves.
Consider only drones within this strip, as any pair of drones with a smaller distance must have one drone from each half.
Sort these drones by their y-coordinates.
Iterate through the sorted drones in the strip, comparing only those within a distance of d-min vertically.
Update d-min if a smaller distance is found.

Time Complexity:

the sorting steps take O(n log n) time.
the recursive call divide the problem into halves, leading to a log n factor.
The middle strip processing takes O(n) time.

So, Algorithm has a time complexity of O(n log n).

2-) → Base case:

If there are very few sensors (3 or fewer), I can directly determine the minumum sensors required using a brute force algorithm.

→ Sorting:

I start by sorting the sensors based on their x-coordinates to divide the region effectively.

→ Divide and conquer:

I recursively divide the sorted sensor list into two halves: left and right
I solve the problem for each half independently, finding the maximum sensors needed for their perimeters.

→ Merging and middle strip:

I carefully merge the results from the halves, prioritizing sensors covering critical exploration areas.
I implement a robust strategy to select sensors in the middle region, ensuring a continous vertical strip of coverage.

→ Coverage and overlap:

I verify that the chosen sensors provide full coverage within the secured perimeter, not just at the boundaries.
I adjust the selection process to ensure sufficient overlap at the perimeter edges.

→ Sensor Ranges and Obstacles:

If sensor detection ranges are avaible, I incorporate them to guarantee adequate coverage.
If there are terrain obstacles, I modify the algorithm to avoid selecting sensors that are blocked or have limited visibility.

Time Complexity: $O(n \log n)$

3 -) Setting up:

I created a 2D table in my mind, like a grid, where rows represent one sequence and columns represent the other.
I filled the first row and column with numbers, indicating the cost of aligning an empty string with either sequnce.

Filling the grid:

I systematically went through each cell in the grid, calculating the minumum cost to align the sequences up to that point.
If the nucleotides in the sequnces matched, great no cost here.
If they didn't match, I consired there options:
Insertion: Add a nucleodide to one sequence, costing 1.
Deleting: Remove a nucleotide from the other sequence, also costing 1.
Substitution: Swap one nucleotide for another, costing 3.

I always chose the option with the lowest cost and recorded it in the cell.

Backtracking the Path:

Once the grid was filled, I started from the bottom-right corner and worked my way back. tracing the path of minumum costs.

At each cell, I choose the operation that led to that minumum cost, revealing the sequence of steps needed for alignment.

Time complexity:

m = len(seq1)
n = len(seq2)

Stetting up takes O(m+n) time.
Filling the grid takes O(mn) time.
Backtracking the Path takes O(m+n) time.

So the algorithm's time complexity O(mn).

4-) Construct a Table: I initialized a table called max_discounts to store the maximum discounts achievable for various subsets of stores. This table plays a crucial role in avoiding redundant calculations, leading to efficiently gains.

→ Iterate through Subsets: I systematicaly iterated through all possible subsets of stores, starting from single-store subsets and gradually expanding the size. This approach ensures that we consider every potential combination of stores.

→ Calculate maximum discount for each subset: For each subset, I calculated the maximum discount achievable by carefully evaluating two distinct cases:

Including the current store: I calculated the discount for the current subset using the calc_discount() function and added it to the maximum discount already calculated for the subset excluding the current store

Excluding the current store: I retrieved the maximum discount achievable from the max_discounts table for the subset that excludes the current store.

→ Choose the optimal discount: I compared the maximum discounts calculated in both cases and selected the larger value, storing it in the max_discounts table for the current subset. This ensures that we always maintain the highest possible discount for each subset.

→ Retrieve the max. discount: The final max. achievable discount for the entire sequence of stores is conveniently stored in the max_discounts table for the full set of stores.

Time Complexity: $O(n^2)$

5-)

→ Sorting Antennas:

I first sorted all the antennas based on their ending points on the street, from left to right. This ensures we consider antennas with shorter coverage ranges first.

→ Activating the First Antenna:

I boldly activated the first antenna in the sorted list, as it has no potential conflicts yet.

→ Iterating and activating:

I then went through the remaining antennas one by one, making crucial decisions:

If an antenna's coverage range didn't overlap with the last activated antenna, I confidently activated it, expanding our coverage.

If it did overlap, I wisely skipped it to avoid interferance.

→ Keeping Track:

I carefully maintained a list of activated antennas to ensure no intersections occurred.

→ Time Complexity:

The sorting step takes O(n log n) time.
The iteration over antennas takes O(n) time.

Algorithm's time complexity O(n log n).