SID: 510 444 907 14/03/2025

COMP3027 – Assignment 1

Problem 1 – Star Wars

Part A – Algorithm

- Sort clone data by ascending perishing time.
- Initialize an empty reference clone.
- Iterate through the sorted clone data and compare the reference clone with each clone. (Runs in $O(n)$ time)
  In each iteration:
    - If the reference clone is empty, fill the reference clone with the current clone. (Runs in $O(1)$ time)
    - If the compared clone's emergence time is less than or equal to the reference clone's perishing time, we add them to the bribe list. (Runs in $O(1)$ time)
    - If the compared clone's emergence time is greater than the reference clone's perishing time, we replace the reference clone with the currently compared clone. (Runs in $O(1)$ time)

Part B – Correctness

Proof by Exchange Argument

Let X be the bribe list generated by our greedy algorithm. Let X_{opt} be an optimal bribe list with the maximum number of bribed clones.

If $X = X_{opt}$, then we are done. Otherwise, suppose $X \neq X_{opt}$. Then, there exists a pair of clones i and j such that:

- In X_{opt}, clone j is selected before clone i,
- But in X, clone i appears before clone j, and
- The perishing time $p_i > p_j$.

Pair (i, j) are an inversion, because in the optimal list, a clone with a later perishing time appears before one with an earlier perishing time.

Consider the effect of swapping the order of i and j in the optimal solution so that clone j comes before clone i.

In this swapped schedule:

- Clone j is selected earlier, which is beneficial since its perishing time is earlier.
- Clone i can still be considered afterward, and since $emergence_i > emergence_j$, the reference window for clone j still overlaps with i if it did previously.

Because $p_j < p_i$, clone j's tighter expiration window does not interfere with any clones that could overlap with i, and this swap does not reduce the number of valid bribes.

Thus, this exchange preserves feasibility. By repeatedly eliminating inversions (applying these swaps), we can eventually transform X_{opt} into the schedule produced by our greedy algorithm without decreasing the number of bribes.

Hence, the greedy solution X is also optimal.

Part C – Time Complexity

Using a merge sort to sort the clone data by ascending perishing times takes $O(n \log n)$ time.

Initializing an empty reference clone takes $O(1)$ time.

Iterating through the sorted clone data and compare the reference clone with each clone runs in $O(n)$ time.

In each iteration:

- Filling the reference clone with the current clone runs in $O(1)$ time.
- Adding a clone to the bribe list runs in $O(1)$ time.
- Replacing the reference clone with the currently compared clone runs in $O(1)$ time.

Summing these running times up, we get:

$$O(n \log n) + O(1) + O(n) + O(n) + O(1) + O(1) + O(1) = O(n \log n)$$

Hence, $O(n \log n)$ running time.

SID: 510 444 907 14/03/2025

Problem 2 – Age of Empires

Part A – Algorithm

- All checks within Manhattan distance r from any cell runs in $O(4r^2)$ time

Iterate through the middle column and middle row and check each cell.

- Within each check, record the highest recorded value
- Within each check, record the position of the highest recorded value

Now that we know which quadrant the highest position is (breaking ties by random selection), we move to that quadrant and repeat the above process within that quadrant until a suitable cell is found.

Part B – Correctness

Base Case: The base case is $n = 1$, where the algorithm trivially returns the only cell, which is clearly a peak.

Inductive Hypothesis: For the induction hypothesis, assume that the algorithm correctly finds a peak for all grids of size $k \times k$, where $1 \leq k < n$.

Consider a grid of size$n \times n$. The algorithm proceeds as follows:

1. It examines the middle row and the middle column.
2. For each cell in the middle row and column, it performs a check within Manhattan distance r.
3. It records the maximum value and its position among all checked cells.
4. Based on the location of the highest value, the algorithm selects the corresponding quadrant and recursively applies the same process.

Let (i, j) be the selected maximum-value cell in the current step. By construction, (i, j) has the highest value in all scanned cells in the middle row and column. While this does not guarantee global optimality, the recursive step of moving into the quadrant containing (i, j) ensures that the search continues in a region that includes at least one candidate for a valid peak. Because the quadrant size is strictly smaller (at most $\frac{n}{2} \times \frac{n}{2}$), the induction hypothesis guarantees that the recursive call returns a potential returns a candidate that satisfies the peak condition within that quadrant.

Eventually, this process reaches a grid of size 1 × 1 (the base case), ensuring termination. The returned cell is thus guaranteed to be a peak.

Therefore, by mathematical induction, the algorithm always returns a cell that satisfies the peak condition for any $n \times n$ grid.

Part C – Time Complexity

Iterating through the middle column and middle row runs in $O(2n)$. Adding a check to each cell causes the time complexity to run in $O(2n \times 4r^2)$.

The next iteration in one of the quadrants runs in $O(n)$. Adding a check to each cell causes the time complexity to run in $O(n \times 4r^2)$.

Similarly, the next iteration in one of the smaller quadrants runs in $O\left(\frac{n}{2}\right)$. Adding a check to each cell causes the time complexity to run in $O\left(\frac{n}{2} \times 4r^2\right)$.

And so on. To calculate all the tiles checked, we do a sum of cells checked per iteration, which leads to a geometric sum:

$$n + \left(\frac{n}{2} \times 4r^2\right) + \left(\frac{n}{4} \times 4r^2\right) + \left(\frac{n}{8} \times 4r^2\right) + \cdots = n \times 4r^2 \times \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right)$$

As such, while treating $4r^2$ as a constant, we can get the following time complexity:

$$O\left(n \cancel{\times 4r^2 \times \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right)}\right) = O(n)$$

Hence, $O(n)$ running time.