Problem 1

Description:

1. Main Function: The main function begins by declaring several variables and reading the number of test cases T from the input. It then enters a loop to process each test case.
2. Processing Each Test Case: For each test case, the code reads the number of elements n, the factor k, and the sorting method id. It then reads the elements into a list A. For each element, it calculates the x and y coordinates and stores them in a vector of tuples.
3. Sorting: Depending on the value of id, the code sorts the vector of tuples using one of the four sorting methods required by the question. Here we set up four Comparator Functions: cmp_1, cmp_2, cmp_3, and cmp_4 used for sorting in the four different types of sorting methods. They take two tuples as arguments and return a boolean value indicating whether the first tuple should be placed before the second tuple in a sorted sequence (I used two if sentences to judge whether the case is x_1 == x_2 or not, and then execute the 4 required y sorting). The difference between these functions is the order in which they sort the tuples based on their x and y coordinates, including all four methods. The sorted vector is then added to a vector of all the results.
4. Output: After all test cases have been processed, the code loops through the vector of all the results and prints the x and y coordinates of each tuple. Each test case is separated by an empty line.

Time Complexity:

The time complexity is primarily determined by the sorting operation. The std::sort function in C++ uses a sorting algorithm called Introsort, which is a hybrid sorting algorithm derived from Quicksort and Heap sort. The worst-case time complexity of this algorithm is O(nlog n), where n is the number of elements being sorted. Since the sorting operation is performed for each test case, and each test case involves sorting n elements, the overall time complexity of the program is O(T * nlog n), where T is the number of test cases.

Space Complexity:

The space complexity is determined by the amount of memory used to store the elements and their coordinates. The list A stores n elements, and the vector coordinates stores n tuples, each containing two integers. Therefore, the space complexity is O(n). Note that this is the space complexity for each test case. If you consider all test cases together, the space complexity would be O(T * n), where T is the number of test cases.

Problem 2

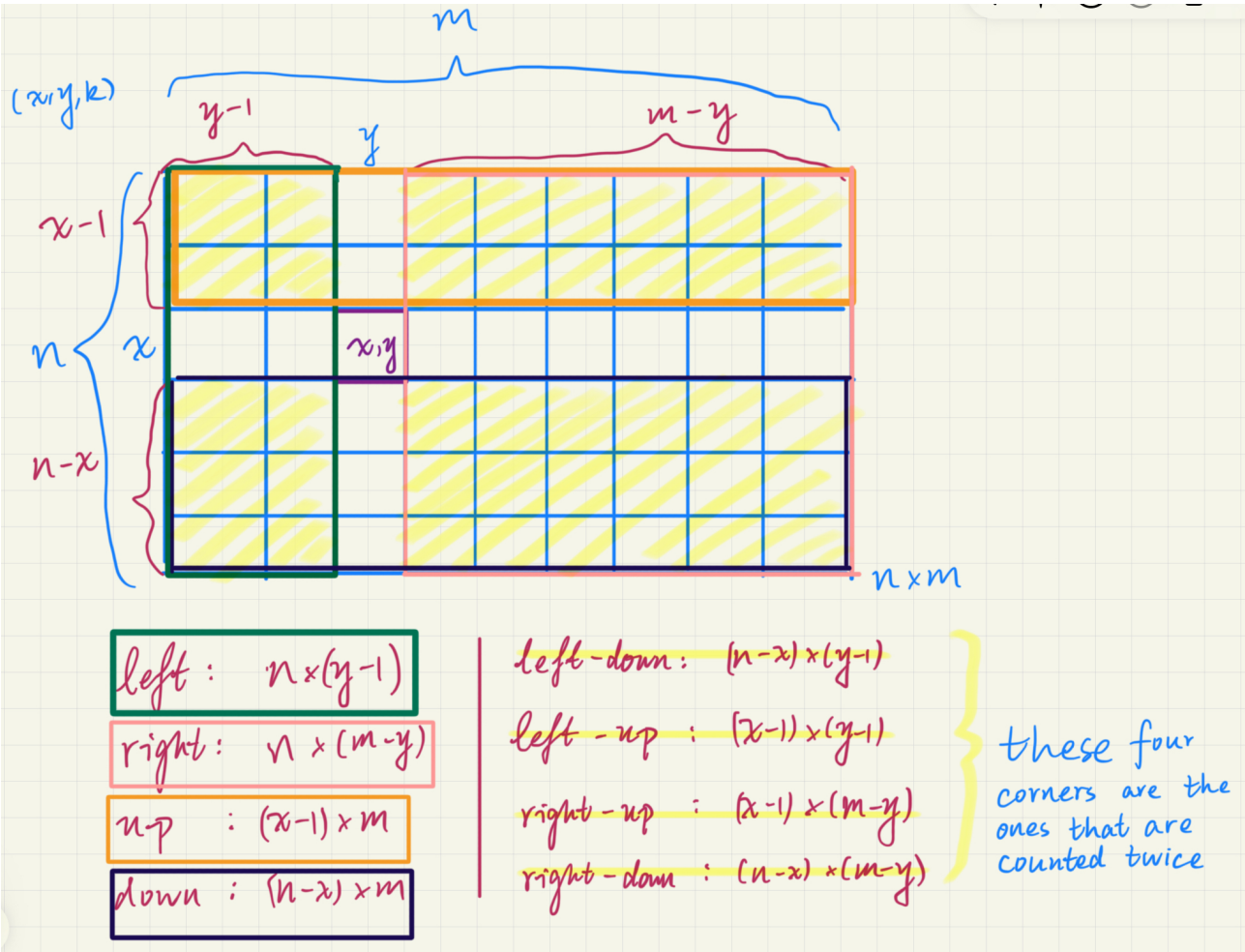
Description:

1. **num_rec Function**: This function calculates the number of any rectangle in a given a*b big rectangle. The calculation is based on the formula: a * b * (a + 1) * (b + 1) // 4. If either a or b is less than or equal to 0, it returns 0.

2. **Main Program**: The program first reads three integers `n`, `m`, and `p` from the input. It then initializes the result to 0.

3. **Loop Over Test Cases**: The program enters a loop that runs `p` times. In each iteration, it reads three integers `x`, `y`, and `k` from the input.

4. **Calculations**: The program calculates several values based on the inputs and the `num_rec` function. Here, the algorithm is designed as follows: in each iteration, we focus on every squad with k_i Tyranids on the n*m map and consider the problem from the 1*1 grid where this squad lies. Let this 1*1 grid be the center, acting as the starting point of the algorithm, we calculate all the other rectangles that do not include this squad, which is done by first calculating the left, right, up and down rectangles, and then calculating the rectangles that are counted twice (leeft_down, left_up, right_up, right_down)(please see the following graph illustrating this algorithm). These values represent different number of any rectangles of 2D grids surrounding the 1*1 (x, y) grid.



After calculating the total, left, right, up, down, left_down, left_up, right_up, and right_down by calling the function num_rec, we can calculate the variable (temp) representing the expected number of Tyranids of this single squad by multiplying the number of all grids containing this squad (x, y, k) with the number of Tyranids in this squad ((total - left - right - up - down + left_down + left_up + right_up + right_down) *k).

5. **Update Result**: After calculating the variable temp, the program then adds `temp` to the `result`, taking the modulus with `MOD` = 998244353 to ensure the result stays within a certain range. As the iteration goes, we calculate all the temps (each temp representing the number of the expected Tyranids in any rectangles) for each squad (in total p times), and finally the result adds up to the expected number of Tyranids of all the squads in any rectangle on the map.

6. **Output**: After all test cases have been processed, the program prints the final result.

Time Complexity:

The time complexity of this program is O(p), where p is the number of test cases. This is because the program performs a constant amount of work for each test case.

Space Complexity:

The space complexity of this program is O(1), which means the amount of memory used does not increase with the size of the input. This is because the program only uses a fixed amount of variables and does not use any data structures that grow with the input size.