- Q1

The most immediate response to the first problem is the brute force solution: using a nested For-loop, any two heights are compared, from which a matching disorder pair is selected and counted. However, since the input n is directly related to the number of times the loop is executed, its time complexity is too large: O(n^2), so the simple idea is not an efficient solution. And then I thought, can we omit some of these comparisons? Then came the idea of sorting and moreover the idea of divide and conquer: merge. In this program, I first split the whole sequence into two sub-sequences until each sub-sequence has only one element left. Then, as I learned in class, through a loop, the two sub-sequences are combined into one large sequence in ascending order. In this process, when we put the elements in the right sub-sequence into the large sequence, it indicates that this element is smaller than all the numbers in the current left sub-sequence that are not in the large sequence, and each of these numbers can form a disorder pair that meets the requirements of the problem, so the number of all the elements in the left sub-sequence that are not in the large sequence should be counted. At the same time, it should be noted that when the corresponding elements of the left and right sub-sequences are equal, we preferentially put the elements of the left sub-sequence into the larger sequence, because the problem requires pair equality is not disorder pair, and we want to avoid redundant counts. Then write two parallel loops to ensure that all the elements are in the large sequence. In the process of merging and counting, finally we get the answer. The time complexity of this solution is much less than (O (n^2)), and it is more efficient.

- Q2

For the second problem, the most immediate response is to first write a recursion to compute f_n, and then compute the value of f_n modulo m. However, the time complexity of the process of recursion f_n is too high (O (n^2)), and f_n as the final result, modulo m will be more complicated. Then, combined with the hint given by the problem, it comes to the idea that the value of f_n modulo m can be taken partly in the process of calculation. At the same time, the matrix in the hint also reminds me that the coefficient can be calculated first and then multiplied with f_0 and f_1. Therefore, I wrote two functions, one is "two_matrix_mult", responsible for the multiplication of two matrices, and then directly take the value of the result modulo m; The other is "mult_matrix_mult", which is responsible for calculating the final result of the coefficient, because here the coefficient is a fixed matrix to the n power, so we use the idea of divide-and-conquer recursion, and keep dichotomizing until it can be calculated directly using the known matrix (Note the odd-even discussion here). Then through the obtained results, the results of the upper level are constantly calculated, and finally, the total coefficient is obtained. Of course, in the process of solving, be careful to discuss the n = 1 case separately. The final result is a further modulo of the multiplication of coefficients with f0, f1. For the whole process, the time complexity is much lower than (O (n^2)) and the efficiency is much higher.