A1_Report_122090337

Problem_1

1 Initial Analysis

The core task is finding the steps of **swapping positions** on a given array of integers during the sorting process. So, the task can be divided into two subtasks: **sorting efficiently** and **counting the steps of swapping (disordered pairs)**.

Since at least I need to sort the integers, so I start my thinking by trying different sorting methods.

2 Possible Solutions for the problem

Selection Sort (Brute Force)

This method means iterating through the entire queues and checking each pair of the integers to see if they are in the correct order. By counting the numbers of the disordered pairs, we get the final answers. However, time complexity of $O(N^2)$ means too slow.

Quick Sort

This method chooses a pivot (normally median) from the queue and partition the queue into two sub-arrays recursively: left are lesser integers and right are greater integers. Sorting the sub-arrays while counting the disorder pairs. However, it is not stable, slowest time complexity is $O(N^2)$.

Merge Sort

This method splits the queue of integers into two sub-queues, recursively sorting them, and calculating the number of disorder pairs. This method is fast and stable. I will elaborate on this method's detailed process and advantage in the following parts.

3 How to Solve the Problem

The thinking process is: I already confirm that merge sort is a very good way to solve the problem because of its time complexity and stableness. So the codes need to include:

1. write merge sort functions;
2. add `count` into the merge sort functions;
3. write a main function to input and output.

In this part, I will explain how the codes work function by function. starting from main() function.

```
def main():
    input_num = int(input())
    heights_list = list()
    input_heights = input().split()
    for i in range(input_num):
        heights_list.append(int(input_heights[i]))

    global count
    count = 0
    merge_sort(heights_list)
    print(count)
```

First, get the input divided into integers and add them into the array `heights_list` for sorting. `count` is the variable for recording disorder pairs, which is the result we want. Then we put `heights_list` into the sorting process `merge_sort()`

```
def merge_sort(sort_height):
        ''' dividing process '''
    if len(sort_height) == 1:
        return
    else:
        mid = len(sort_height)//2
        left = sort_height[:mid]
        right = sort_height[mid:]
                '''recurse process'''
        merge_sort(left)
        merge_sort(right)
        sorted_merge(left, right, sort_height)
```

The function divides the array at the middle, put integers on the left to `left` and integers on the right to `right` recursively until there is only one integer left in `left` and `right`. Then, we put `left` and `right` to `sorted_merge()`

```python
def sorted_merge(left, right, sort_arr):
    global count
    left_order = right_order = sort_arr_order = 0

    while left_order < len(left) and right_order < len(right):
        if left[left_order] <= right[right_order]:
            sort_arr[sort_arr_order] = left[left_order]
            left_order += 1
        else:
            count += len(left) - left_order
            sort_arr[sort_arr_order] = right[right_order]
            right_order += 1
        sort_arr_order += 1
```

The function compares integers in `left` and `right` one by one, and replace integers to `sort_arr` in the correct order, from small to big. In the codes,

```python
'''add 'count' into the merge sort functions'''
count += len(left) - left_order
```

is the most important step for counting. It means when integer in `left` is greater than `right` (which is a disordered pair), **the number of disordered pairs added to `count` is the number of the resting integers in `left`**. That's because integers `left` are already sorted in the correct order in the former iterations, so the resting integers in `left` will all be greater than `right`.

```python
    while left_order < len(left):
        sort_arr[sort_arr_order] = left[left_order]
        left_order += 1
        sort_arr_order += 1
    while right_order < len(right):
        sort_arr[sort_arr_order] = right[right_order]
        right_order += 1
        sort_arr_order += 1
```

This is for adding the resting integers to the array after comparison to ensure the completeness of arrays. Finally `print(count)` in `main()` will display the number of disordered pairs, which is what we want.

4 Why is Merge Sort Better

Compared to Selection Sort

Selection sort is inefficient because its time complexity is $O(N^2)$. It is too slow to even pass OJ. While merge sort has time complexity of $O(NlogN)$, which is much faster than selection sort when input is large-scale.

Compared to Quick Sort

The average time complexity of quick sort is $O(NlogN)$, but if there are too many duplicate integers or nearly sorted integers in the input, its time complexity may go up to to $O(N^2)$.

However, merge sort is much stabler. The slowest time complexity of the merge sort method is $O(NlogN)$. So regardless of the input data it guarantees completion within this time complexity. That's why merge sort wins over quick sort.

Problem_2

1 Initial Analysis

This is a calculation problem, but the calculation is very complex. So the core task is to find a way to calculate the modulo efficiently.

First, we need to do some math. The question tells us:

$f_N = af_{N-1} + bf_{N-2}$ therefore,

$$\begin{bmatrix} f_N \\ f_{N-1} \end{bmatrix} = \begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f_{N-1} \\ f_{N-2} \end{bmatrix} = \begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f_{N-2} \\ f_{N-3} \end{bmatrix} = \begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f_{N-3} \\ f_{N-4} \end{bmatrix} \dots$$

$$\dots = \begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}^{N-1} \begin{bmatrix} f_1 \\ f_0 \end{bmatrix}$$

So, we can get f_N by getting $\begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}^{N-1}$

2 Possible Solutions for the problem

Directly Calculating $\begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}^{N-1}$ (Brute Force)

Since we can write the calculation process of matrix multiplication, we can directly use $\begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}^{N-1}$ to multiply itself for N times.

Recursively Calculating $\begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}^{N-1} \mod m$

The goal is ti get $f_N \mod m$ instead of f_N. Since m is consistent, calculating modulo anywhere does not affect the resultSo, it will be much faster to get $f_N \mod m$ by getting $\begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}^{N-1} \mod m$. Besides, using recursive calculation (actually a way of dynamic programming) helps greatly reduces repetitive calculation (elaborate in the following part).

Non-recursively Calculating $\begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}^{N-1} \mod m$ Using DP

Using modulo as well. The difference is, using dynamic programming doesn't necessarily means recursion. We can use binary numbers for non-recursively DP methods (elaborate in the following part).

3 How to Solve the Problem

I will elaborate this part by introducing my thinking process.

First, we need to figure out which solution is the best choice.

Direct calculation will be extremely slow because the result of the multiplication will be extremely big, which greatly slows down the program. In addition, the multiplication of huge matrices will be N times, which is completely impractical. `mod` during multiplication will be better. Let's write the matrix multiplication function.

```python
def matrix_multiply(mx_1, mx_2, mod):
    if len(mx_2) == 4:
        return [(mx_1[0] * mx_2[0] + mx_1[1] * mx_2[2]) % mod,
                (mx_1[0] * mx_2[1] + mx_1[1] * mx_2[3]) % mod,
                (mx_1[2] * mx_2[0] + mx_1[3] * mx_2[2]) % mod,
                (mx_1[2] * mx_2[1] + mx_1[3] * mx_2[3]) % mod]
    else:  # len(mx_2) == 2
        return [mx_1[0] * mx_2[0] + mx_1[1] * mx_2[1],
                mx_1[2] * mx_2[0] + mx_1[3] * mx_2[1]]
```

Now, the goal is: how to calculate $\begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}^{N-1} \mod m$ efficiently.

Recursively calculation sounds really good, and it was my first choice, the codes are like

```python
def matrix_exp(matrix, power, mod):
    if power == 1:
        return matrix
    if power % 2 == 0:
        div_matrix = matrix_exp(matrix, power // 2, mod)
        return matrix_multiply(div_matrix, div_matrix, mod)
    else:
        return matrix_multiply(matrix, matrix_exp(matrix, power - 1, mod), mod)
```

- If the power is 1, the function returns the original matrix.
- If the power is even, it divides the power by 2 and recursively calculates the matrix exponentiation for the reduced power. Then, it squares the result using `matrix_multiply` with modulo `mod`.
- If the power is odd, it recursively reduces the power by 1, calculates the matrix exponentiation for the reduced power, and multiplies it with the original matrix using `matrix_multiply` with modulo `mod`.

However, "any user using Python should know that the upper limit of Python recursion is 1000 times." When power (N) is big enough, the recursion will exceeds 1000, OJ will show "IR". `sys.setrecursionlimit(2000000)` will directly causes "TLE" because recursions take too much time.

Then, non-recursively calculating $\begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}^{N-1} \mod m$ Using DP becomes the final solution.

```
def matrix_exp(matrix, power, mod):
    order = bin(power)
    ans_matrix = [1, 0, 0, 1]
    while order[-1] != 'b':
        if order[-1] == '1':
            ans_matrix = matrix_multiply(ans_matrix, matrix, mod)
        matrix = matrix_multiply(matrix, matrix, mod)
        order = order[:-1]
    return ans_matrix
```

`bin(power)` is the binary representation of `power` and it's a string. e.g. `bin(10) == 0b1010`. It has an important binary meaning, that $a^{10} = a^{2^{4-1}} \times a^{2^{2-1}} = a^8 \times a^2$, it helps to reduce repetitive calculation, because $a^8 = (a^4)^2 = ((a^2)^2)^2$, in which a^2 and a^4 are stored and used again, no need for calculating again (which is a way of DP).

Now, `order` is a string. `ans_matrix` is originally an identity matrix, used to accumulate the result. When `order` doesn't come to its head `0b`, the function will read `order` from the end to head.

`matrix = matrix_multiply(matrix, matrix, mod)` means in each iteration, square the original matrix `matrix` using `matrix_multiply` function, which is equivalent to exponentiating the matrix by 2.

Noted that the $i\,th$ iteration, `matrix` = $\begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}^i$, if binary digit in this iteration is '1', then `ans_matrix` = `ans_matrix` $\times \begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}^i$

`order = order[:-1]` removes the last digit from the binary representation of `power`.

The loop continues until all bits in the binary representation `bin(power)` have been processed, calculating the matrix exponentiation using binary exponentiation. Finally, it returns the resulting `ans_matrix`, which represents the matrix raised to the power `power`.

```
matrix_after_exp = matrix_exp([a, b, 1, 0], n - 1, m)
ans = (matrix_after_exp[0] * f_1 + matrix_after_exp[1] * f_0) % m
print(ans)
```

Finally in main(), we get `ans` $f_N\, mod\, n$ by `matrix_after_exp` $\begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}^{N-1} \mod m$

4 Why is the Method Better

1. Compared to direct calculation, this method is much much faster.
2. Compared to recursion method, recursion method doesn't work when recursion take place for too many times, it's fast but not stable. But the method above is fast and stable. That's why this method of using non-recursive binary representation is a good solution.