Report for Assignment 1

Note:

In Possible Solution sections, I also include information in my solution. Plz be well-checked.

Problem 1. Queue Disorder

Given an array A with length n, we want to find the number of pairs $(i, j), i < j$, where we have $A[i] > A[j]$.

1.1 Possible Solutions

Clearly, it is trivial to have the brute force solution. That is, we let $i = 0, \cdots, A.\text{length}$ and compare every $A[j], j \geq i$ with $A[i]$. The complexity is $O(n^2)$, which is impossible to handle large arrays e.g. $n = 10^6$.

To improve the algorithm, it is naturally to think about the algorithm with $O(n \log n)$. To achieve this, we may think about ***divide and conquer algorithm***. More specifically, each time we divide the algorithm into two parts and count the number of disordered pairs separately and add the number of disordered pairs when combining the two parts. It is natural to think about the merge sort, which is in the similar pattern.

Now, we want to calculate what is the number of disordered pairs when merging two arrays. Since it may be analogous to the merge sort, we step into this issue further. Assume we have two sorted arrays:

$$A = [a_1, a_2, \cdots], \qquad B = [b_1, b_2, b_3, \cdots]$$

Now, to combine them, ideally, $\forall a \in A$, we have $a < b, \forall b \in B$. Assume that, for $b \in B$ s.t. $b < a_i$ for some i. Then, it is clear that all $a_k : k > i$ will be disordered w.r.t. b. Therefore, through merging, we can keep track of all the disorders occurred during merging. Since counting is $O(1)$, the algorithm should have complexity $O(n \log n)$ as merge sort.

1.2 My Solution

Effectively, my algorithm works recursively as:

1. Divide the array into two parts
2. Calculate the disorders specific to each part, and do sorting simultaneously to get two ordered arrays
3. Calculate the disorders owing to combining to sorted arrays

```
def mergeSort(array, left, right):
    count = 0
    if left >= right-1:
        return 0

    mid_point = (left+right)//2
    count_left = mergeSort(array,left,mid_point)
    count_right = mergeSort(array, mid_point, right)
    count_merge = merge(array,left,right)
    count = count_merge + count_left + count_right
    return count
```

This part is similar to merge sort, except that we are sorting the array while keeping track of the disorders. For each sub-array, we count the disorders recursively, and to avoid counting-twice, we sorted them after counting.

As for the merging part, it is also similar to merge sort. We use two indices i, j to denote the position of elements in the two sorted parts (denoted as A, B).

1. If $A[i] \leq B[j]$, there is no disorder
2. If $A[i] > B[j]$, then for any elements l in A later than $A[i]$, we must have $A[l] > B[j]$, which is also disorder.

Therefore, to place $B[j]$ to the right position, it occurs $A.\text{length} - i$ disorders. We sum all the number of disorders up to get the disorders occurred in merging.

1.3 Advantages

1. Compared with brute force, its time complexity is way lower ($O(n \log n)$). Therefore, it can handle array with large size efficiently.
2. Except for counting disordered pairs, it can be used to automatically sort the disordered array.

Problem 2. Star Sequence

Given number of iterations n, parameters a, b, initial value f_0, f_1 and modulo m, we want to

2.1 Possible Solutions

It is trivial to think about the naive approach, say recursively calculate f_n directly. However, when n is sufficiently large, f_n is unnecessarily large, which is huge burden for time and space (it may exceeds the space allocated for long) .

Besides, using matrix multiplication directly on f_n is not ideal, say

$$\begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} f_1 \\ f_0 \end{bmatrix} = \begin{bmatrix} f_n \\ f_{n-1} \end{bmatrix}$$

By exploration, taking $a = 1, b = 1, n = 300000$, $\begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}^n$ is unreasonably large in scale.

Therefore, we ***<u>abandon</u> brute force approach***.

Now, it is useful to consider mod. We have two properties:

$$\begin{aligned} (a \mod m) + (b \mod m) \mod m &= (a + b \mod m) \\ (a \mod m)(b \mod m) \mod m &= (ab \mod m) \end{aligned}$$

Therefore, we have

$$f_n \mod m = [(a \mod m)(f_{n-1} \mod m) + (b \mod m)(f_{n-2} \mod m)] \mod r$$

Therefore, we have

$$\begin{bmatrix} (a \mod m) & (b \mod m) \\ (1 \mod m) & (0 \mod m) \end{bmatrix}^n \begin{bmatrix} f_1 \mod m \\ f_0 \mod m \end{bmatrix} \mod m = \begin{bmatrix} f_n \mod m \\ f_{n-1} \mod m \end{bmatrix} \mod m$$

This could enormously reduce the scale of the calculation.

Now, the problem is remained to calculate

$$\begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} = \begin{bmatrix} (a \mod m) & (b \mod m) \\ (1 \mod m) & (0 \mod m) \end{bmatrix}^n$$

Direct calculation would lead to $O(n)$ complexity. When n is large enough, say $n = 10^{18}$, it is quasi-impossible to calculate in short time. Therefore, we consider recursively calculate it using algorithm:

```
Matrix_power(n):
    Matrix_power(n//2) * Matrix_power(n-n//2)
```

with base case return the matrix itself. The time complexity is $O(\log n)$.

After computing the matrix power, we could compute

$$f_n \mod m = a_1 f_{n-1} + a_2 f_{n-2} \mod m$$

2.2 My Solution

As in section 2.1:

Recursively calculate the power

```
def matrix_power(a,b,n):
    if n in memory_dict.keys():
        return memory_dict[n]
    if n == 1:
        return (a%m,b%m,1,0)
    n1 = n//2
    n2 = n - n1
    result =
matrix_multiplication(matrix_power(a,b,n1),matrix_power(a,b,n2))
    memory_dict[n]= result
    return result
```

On top of what I have mentioned in section2.1, I also speed the algorithm using memory disk as the code displays

Using memory disk, This algorithm will store the calculated matrix power for use. Therefore, when encounter the same power, it can direct get the matrix from dictionary using key.

2.3 Advantage:

1. Using modulo instead of the absolute value in the calculating process could reduce the scale and improve the space and speed efficiency
2. The algorithm to calculate the power of 2D matrix to bisect the power is $O(\log n)$ complexity, which is useful especially when n is sufficiently large
3. Using memory dictionary to record the power value could reduce at least half of the time, say assume $n = 100000$, to calculate the power, we will divide it into (power(50000)*power(50000)). With memory dictionary, we no longer need to calculate the second half, since we can directly get such value from the dictionary.