CS208 · Algorithm Design and Analysis

Assignment 5

张文灏 / 11812103

April 29, 2020

Exercise 5.1, 5.2

1 Exercise 5.1

We can give an algorithm based on binary search to find the median value, with $O(\log n)$ times to query.

Assume that no two values are the same. Suppose $a_1, a_2, \cdots, a_n$ and $b_1, b_2, \cdots, b_n$ are the values in the first and second database respectively, satisfing that $a_1 < a_2 < \cdots < a_n$ and $b_1 < b_2 < \cdots < b_n$.

Consider the situation that the median is in a_i, let it be a_k. Easily we can find that $b_{n-k} < a_k < b_{n-k+1}$ (or only $k = n, a_n < b_1$, which can be checked separated). Then using binary search to find k. Initially the values between 1 and n can possibly be the value of k. Then we check x, when $b_{n-x} < a_x < b_{n-x+1}$, then $k = x$ and the answer is a_x. When $a_x < b_{n-x}$, then x is relative small and the possible range for k should be $[x + 1, r]$. When $b_{n-x+1} < a_x$, then x is relative large and the possible range for k should be $[l, x - 1]$.

After several iterations, either we can find the k or the range will be empty. When the range is empty, we can say that the median is in b_i. Using the similar way to find the value of k such that $a_{n-k} < b_k < a_{n-k+1}$ (or $k = n, b_n < a_1$), then we can find the answer.

It is easy to find that we use at most twice of binary search, with the initial range $O(n)$. Each time we only query $O(1)$ times. Then the total times we need to query is $O(\log n)$.

2 Exercise 5.2

Recall the regular way to count the number of inversions by merge sorting. We divide the array into 2 parts, solving each part recursively, then merge two parts and conquer the answer.

Here to count **significant** inversions, we only need to modify some details of the algorithm to count regular inversions by merge sorting. For the regular inversions, when we merge two parts, we use two pointers to indicate the smallest value not merged in each part, and add the number of elements that have not been merged in the left part into the total answer, when each time an element in the right part is merged.

Then for significant inversions, we add a new pointer to indicate the smallest value in the left part, which is larger than **twice** of the value indicated by the right pointer. Then when each time an element a_j in the right part is merged, we add the number of the element in the right of the new pointer (including itself) to the answer. Easy to find all and only these elements in the left part can make significant inversions with a_j.

Since both the left and right parts have been sorted, we only need to move the new pointer linearly, and move other pointers in the same way as we have described in merge-sort algorithm. The total time complexity is still $O(n \log n)$.