3 Divide and Conquer

3.1 Fixed Point in an array

3.1.1 Algorithm

Algorithm 3 Pseudo code for Fixed Point

Input:
A, An array that has been sorted.
Output:
ans, 1 if array has fixed point, 0 if not.

```
ans ← false
l ← 0, r ← A.size() − 1
while l + 1lr do
    mid ← (l + r)/2
    if A[mid] == mid then
        return 1
    else if A[mid] > mid then
        r ← mid
    else
        l ← mid
    end if
end while
return A[l] == l or A[r] == r
```

3.1.2 Correctness

According to the definition of the values in the array, if $A[i] < i$, then the possible fixed point won't be in the right side of i, since $A[i+k] > A[i] + k > i + k$. Similarly, if $A[i] > i$, then the possible fixed point won't be in the left side of i. For those point that we search between index l and r, we promise that $l < A[l] < A[i] < A[r] < r$. If the interval turn out to not contain fixed point, then we return 0.

3.1.3 Time Complexity

Since we halve the array during each iteration, the time complexity is O($\log n$)

3.2 Organizing a lottery

3.2.1 Algorithm

Algorithm 4 Pseudo code for Lottery

Input:
$points$, A sequence of n points on a line.
$segments$, A sequence of m segments.
Output:
$count$, A vector of number representing number of segments that contains each points.

```
initialize(count, 0)
sort(points)
for segment in segments do
    lval, rval ← boudary(segment)
    lidx ← firstElemBiggerThan(lval)
    ridx ← firstElemSmallerThan(rval)
    for point in [lidx, ridx] do
        count[point] ← count[point] + 1
    end for
end for
return count
```

First we set the count of each point to 0, and sort the points according to their coordinates. Then for each segment, we find the points' coordinates that are closest to the segment's left and right boundaries. Then for each points within the segment, we add the count of this point by 1.

3.2.2 Correctness

The correctness is quite straight forward, each time we are adding the count only within one single segment. After iterates through all the segments, the count are our answer.

3.2.3 Time Complexity

Initialize would take O(n) time.

Sorting the points takes O($n \log n$) time.

Searching the two boundary across a sorted list takes 2O($\log n$), and iterates the all segments takes 2O($m \log n$)

Together, the algorithm would take O(n) + O($n \log n$) + 2O($m \log n$) = O($(m + n) \log n$) time.

3.3 Finding a peak in sublinear time

3.3.1 Algorithm

Algorithm 5 Pseudo code for Finding 2d peak

Input:
$matrix$, a $n \times n$ matrix containing pairwise different integers.
$left$, minimum column for searching.
$right$, maximum column for searching.
Output:
$peak$, an element that is greater than all its neighbors.

```
1: mid ← n/2
2: p, p_idx ← find_column_peak(matrix, mid);
3: if n == 1 then
4:     return p
5: else if matrix[mid][p_idx] < matrix[mid − 1][p_idx] then
6:     p ← find_2d_peak(matrix, 0, mid − 1);
7: else if matrix[mid][p_idx] < matrix[mid + 1][p_idx] then
8:     p ← find_2d_peak(matrix, mid + 1, n);
9: else
10:    return p
11: end if
```

Every time we find the peak in the middle column of the matrix.(the time complexity of this function will be discussed later) Then we compare the neighboring elements in the left and right column. If either one of the two element is bigger than the middle column's peak value, then there must exist a global peak in the corresponding side. Otherwise, the middle column peak itself is the global peak.

3.3.2 Correctness

We will just focus on the left side of the matrix, since the situation in the right will be the same. Let us denote $left = matrix[mid - 1][i]$ such that $peak = matrix[mid][i]$ is the peak of the middle role.(instead of global peak) If $left > peak$, there must exist a global peak in the leaf sub-matrix.

1. Suppose that there doesn't exist a peak in the left sub-matrix. Then $left$ must have a value $left_1$ s.t. $left_1 > left$.
2. But $left_1$ is not a peak too, then there will be a neighboring element $left_2$ s.t. $left_2 > left_1$
3. ...
4. We can't move the neighboring point $left_i$ any point right-ward beyond the middle point. If we did that, $left_i > left_{i-1} > ... > left > peak$. Then $peak$ cannot be the peak element of the middle point, which is contradictory with our condition.
5. Since there will be finitely number of elements in the left half of matrix, we will finally run out of elements to go.
6. Thus, there must exist an element in the left half of the matrix that is the global peak.
7. We can easily prove the correctness on the right half.

3.3.3 Time Complexity

To find the column peak, find_1d_peak can take O(n) time scanning the whole column. Better still, it can take only O($\log n$ time. By compare the neighboring of the middle point and goes to the bigger side. The proof will be ommited since it is the simplified version of the 2d peak finding. So we have

$$T(n, n) = T(n, n/2) + O(\log n) = O((\log n)^2)$$