**Kick Start 2017 - Round D**

# Analysis: Sherlock and Matrix Game

## Sherlock and The Matrix Game : Analysis

### Brute force approaches

In the Small dataset, **N** (which is the size of arrays **A** and **B**) doesn't exceed 200. A naive bruteforce strategy is to actually create the matrix and then iterate over all possible submatrices and calculate the sum for each submatrix. Since the number of submatrices is $O(N^4)$ (observe that for every pair of cells, there exists a unique submatrix), naively iterating over each submatrix in O(*submatrix size*) is bound to run for years(it's a whopping $1.2 * 10^{15}$ operations for 20 test cases in the worst case). However, if we can calculate sum of each submatrix in constant time, the number of operations will reduce to $3 * 10^{10}$, which is feasible for a modern machine in minutes.

#### Inclusion-exclusion to the rescue

Quickly calculating submatrix sums of a matrix M can be done by maintaining another matrix P such that P(i, j) = sum of all cells M(a, b), such that $1 \le a \le i$ and $1 \le b \le j$. Matrix P can be built in O(*size of* M) time by using the recurrence P(i, j) = M(i, j) + P(i-1, j) + P(i, j-1) - P(i-1, j-1). Note that the first three terms on the right hand side count some of the cells twice, which is why we subtract off the fourth term. Further, the sum of a submatrix defined by top-left and bottom-right cells (a, b) and (c, d) can be calculated as P(c, d) - P(a-1, d) - P(c, b-1) + P(a-1, b-1). Again, we add the last term on right hand side to accomodate the double subtraction of some cells.

### A binary search large approach

For the Large dataset, **N** could be up to $10^5$, which implies that creating the matrix in-memory is not possible, let alone iterating over all possible submatrices. We can, however, use the fact that $M(i, j) = \mathbf{A}_i * \mathbf{B}_j$ to our advantage by expressing sum of cells in the submatrix defined by top-left and bottom-right cells (a, b) and (c, d), respectively, as $(\mathbf{A}_a + \mathbf{A}_{a+1} + ... + \mathbf{A}_c) * (\mathbf{B}_b + \mathbf{B}_{b+1} + ... + \mathbf{B}_d)$, i.e., product of sum of contiguous subarrays of **A** and **B**.

If we create two sorted lists U and V consisting of all possible subarray sums of **A** and **B**, respectively, we're trying to find the $\mathbf{K}^{\mathbf{th}}$ largest value among $U_i * V_j$ for all i, j. At this moment, we can exploit the fact that **K** doesn't exceed $10^5$ by observing that we don't need to consider all possible values in U and V. We can work with the **K** largest and smallest values from both arrays(why the smallest? don't forget we also have negative values?) and be guaranteed that answer will be present in product of these values. At this point, we face two subproblems to finally solve this problem.

#### Subproblem 1

Given an array **A** of size **N**, find the **K** largest subarray sums of **A**. Note that the value of **K** is approximately O(**N**). If we can do this, finding the **K** smallest subarray sums is easy by reversing the signs of values in A and running the same algorithm.

#### Solution

We use binary search! The idea is to first find the $\mathbf{K^{th}}$ largest subarray sum by binary searching for it, and then build the actual subarray sums. To apply binary search for the $\mathbf{K^{th}}$ largest subarray sum, we need a function which can quickly count: how many subarray sums are less than X, for a given X?
Let us define $P_i$ as sum of the first i elements of array **A**. Now, a simple idea is to iterate over i and fix the subarray start at position i. Now, we're trying to count possible j(≥ i) such that $P_j \geq P_i + X$. Note that the right size of the inequality is a constant for a fixed i. Things would've been easier if prefix sum array P had been an increasing sequence. However, that is not the case, since there are also negative values in array **A**. Suppose that we iterate over i from **N** to 1. At each step, we'll try to calculate the answer for i(which depends on all values $\mathbf{A}_j$ such that j ≥ i). Imagine we have a data structure DS which supports two operations:

1. Insert y : inserts y in the data structure.
2. Query y : returns the count of values present in data structure that are less than or equal to y.

Using this DS, our job can be done(we can do DS.Insert($\mathbf{A}_i$), add DS.Query($P_i + X$) to our answer, and then decrement i and so on). Such a data structure can be implemented efficiently using any balanced binary search trees such as splay trees. Such trees can handle both operations in O(log(*size of DS*)). If the same data structure can support iterating over all values present in it in increasing order, we can get all subarray sums less than the $\mathbf{K^{th}}$ smallest subarray sum in a similar way. Note that we don't need the "Query" operation for doing this. So, the complexity of solving this subproblem turns out to be O(log(*range of answer*) * **N** * log(**N**) + **K**).

**Subproblem 2**

Given two arrays **A** and **B** of size O(**N**), find the $\mathbf{K^{th}}$ largest among all possible values $\mathbf{A}_i * \mathbf{B}_j$, for all i, j.

**Solution**

Again, we can use binary search on our answer if we, given X, can count how many pairs i, j exist such that $\mathbf{A}_i * \mathbf{B}_j \geq X$.
Just as in to subproblem 1, the idea is to iterate over i, and then count all possible j such that $\mathbf{B}_j \geq X / \mathbf{A}_i$, which is easily doable using binary search if array **B** is kept in a sorted fashion.
However, note that it is a little trickier because of negative values. The complexity of solving this subproblem turns out to be (log(*range of answer*) * **N** * log(**N**)).