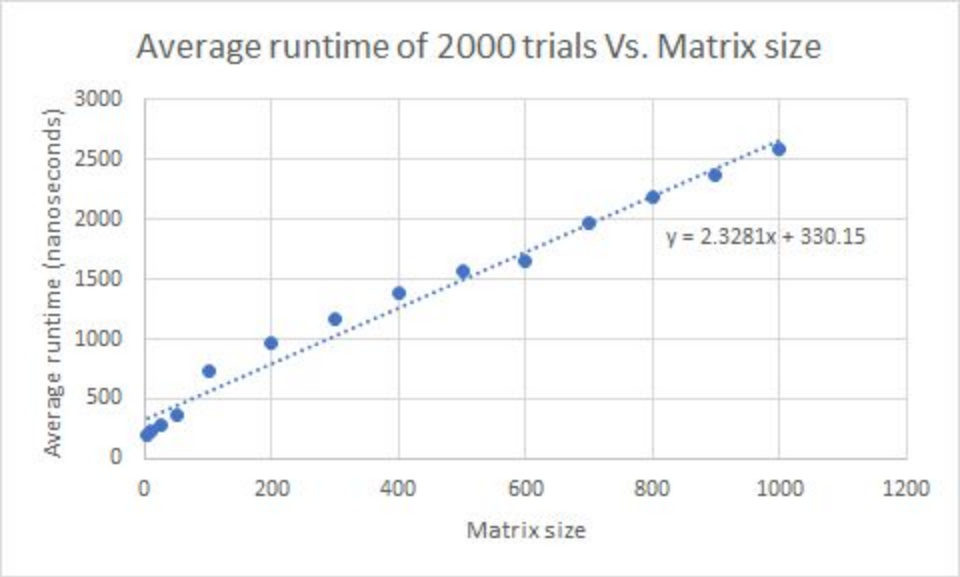
Devin Lin
Dxl702

Data Structure and Algorithms Programming Report

I ran a test for the runtime inMatrix consisting of 2000 trials of size 5, 10, 25, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000 matrices. I summed the runtime of the trials and took the average. The results of the test can be seen in the figure below.



From an analytical standpoint, I know that the worst case runtime of my code is O(2n) because I am never looking at irrelevant portions of the matrix. My algorithm to find the element in question 'x' is to start at the rightmost column of the matrix and traverse downward until we hit an element that is greater than or equal to the element in question. If it is equal to the element, the function instantly returns true and we are done. Otherwise, we perform a search in a zigzag pattern in the southwest direction. For example, the algorithm will start looking for 'x' in the row (horizontal direction) that we found in the previous clause. Once we hit 'x' or a value less than 'x', we increment the row and start looking for 'x' in the vertical direction downward. We repeat these two steps until we either find 'x' or we pass the bounds of the matrix in which case the function returns false. The worst case runtime of this function is O(2n) because at worst, we are looking for the element in the bottom left corner position and a total of 2n elements will have to be traversed to reach that position. In all other cases, the runtime will be slightly faster than O(2n). Regardless, from an analytical standpoint, this implementation of inMatrix has a linear runtime.

To test my theory, I wrote a function to run inMatrix 2000 times and had it return the average runtime of the trials. From the graph above we can see that in fact my function's runtime grows linearly with respect to the size of the matrix as the linear line-of-best-fit barely deviates from the data points.

I would say that this program is very efficient when it comes to a search algorithm for a 2-D array. Normally an easier to write but much slower $O(n^2)$ implementation is used to search up elements in a matrix. However the drawback of this is that when the size of the matrix reaches extreme numbers or even approaches infinity, the O(n) function that I implemented would eclipse the $O(n^2)$ algorithm in performance. However, it is noteworthy to mention that the constraints on the 2-D array that are necessary to make this algorithm work are definitely a drawback as there is a lack of ease of use for this search function.