Mitchell Gouzenko

Library Design C++

Homework 2

**Introduction**

The purpose of this homework was to test the speed of insertion and deletion in vectors and linked lists in the experiment specified by professor Stroustrup. Here, I will highlight my findings, surprises, and subsequent experiments that I performed to justify the results.

**Approach**

As specified by Professor Stroustrup, I timed the insertion of random elements into the list, as well as the deletion of elements at random indices.

To do so, I wrote a function called **generate\_fillers** that creates a vector of random integers from a uniform distribution. These fillers are subsequently fed into vectors and linked lists, and inserted in the correct positions. The function **generate\_removal\_indices** creates a vector of random positions. The positions are used to remove elements from the vectors and linked lists.

I timed the insertion and deletion various amounts of elements. I started from 0 elements and worked my way up to 20,000, taking steps of 1,000.

**Inserting elements**

Since the assignment specifications called for the same algorithm for both linked lists and vectors, I decided to use an iterative method for insertion.

**Hypothesis**

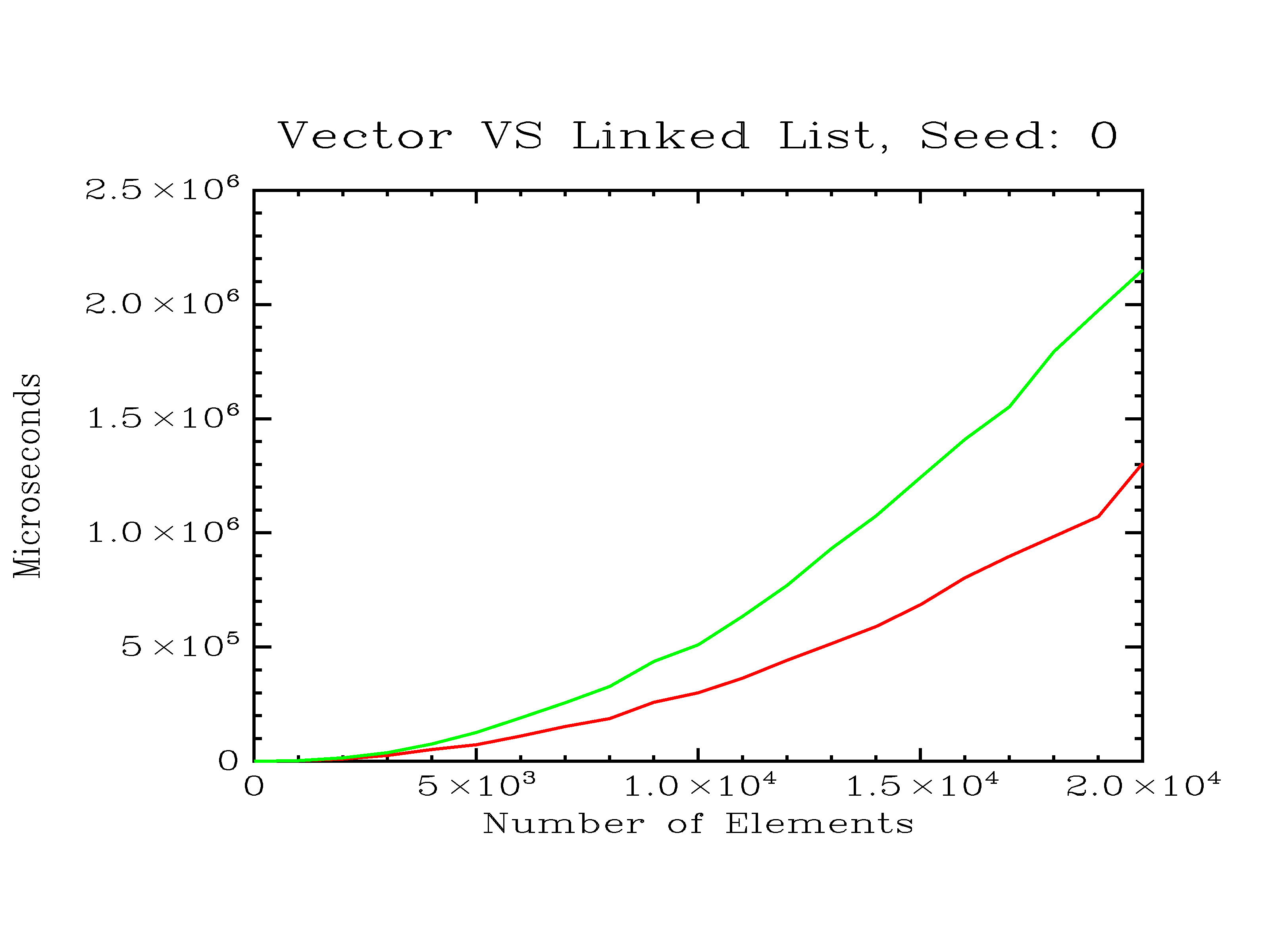
When I started this assignment, I simply assumed that the vectors would outperform the linked lists for a low number of elements. Then, at some point, the linked lists would start outperforming the vector. My logic was as follows: vectors incur an O(n) cost for insertions, since they store underlying elements in arrays. When an element is inserted into its appropriate position, all other elements need to be shifted over by one spot to make space for the new element. For low n, there is not much overhead to do this. But, for high n, many elements need to be moved, so insertion should be slow.

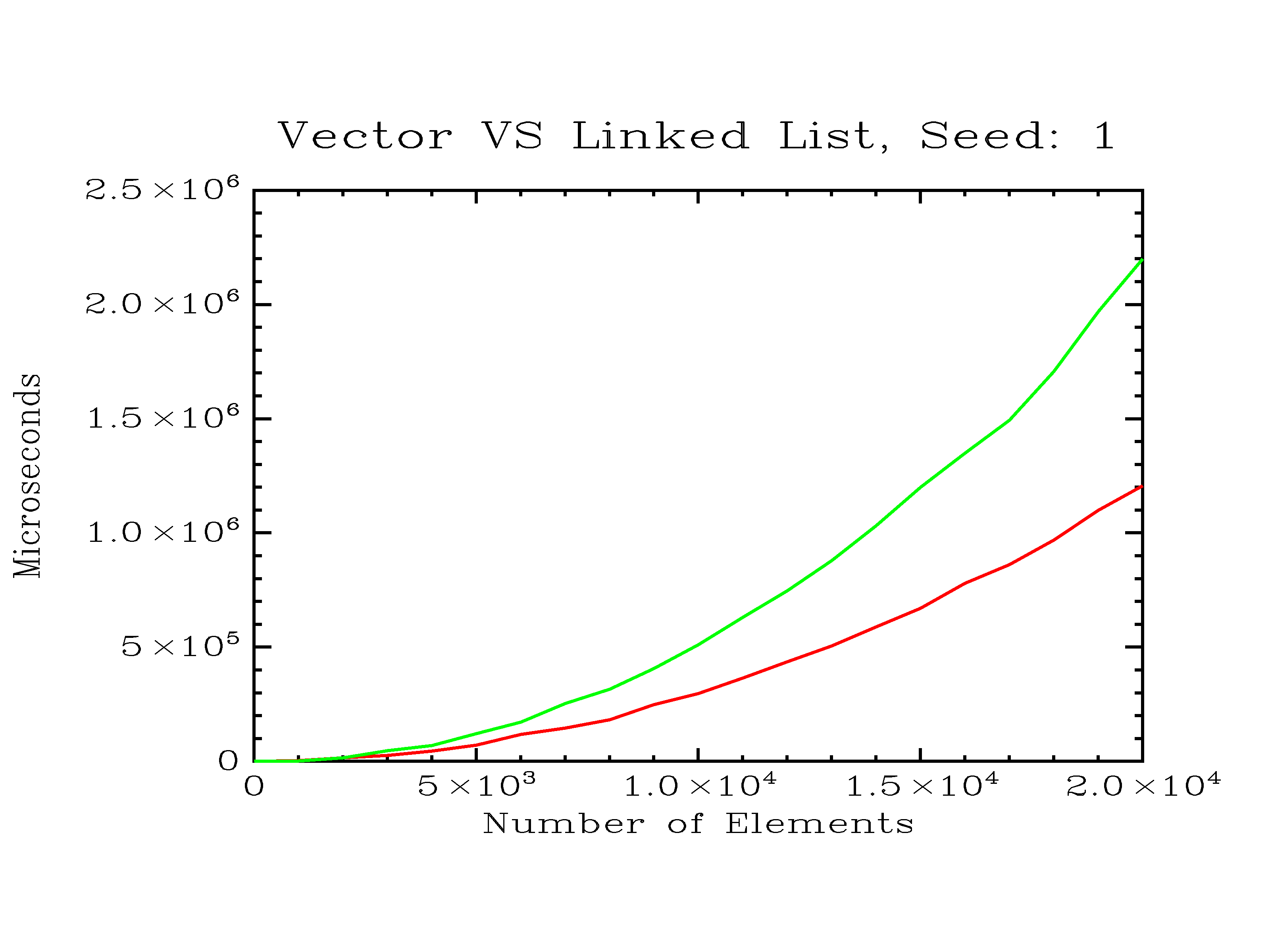
Linked lists, on the other hand, perform insertions in constant time – assuming that an iterator that points to the desired insertion location is readily available. Thus, for large n, the vector’s insertion time should grow linearly, while the linked list’s remains the same.

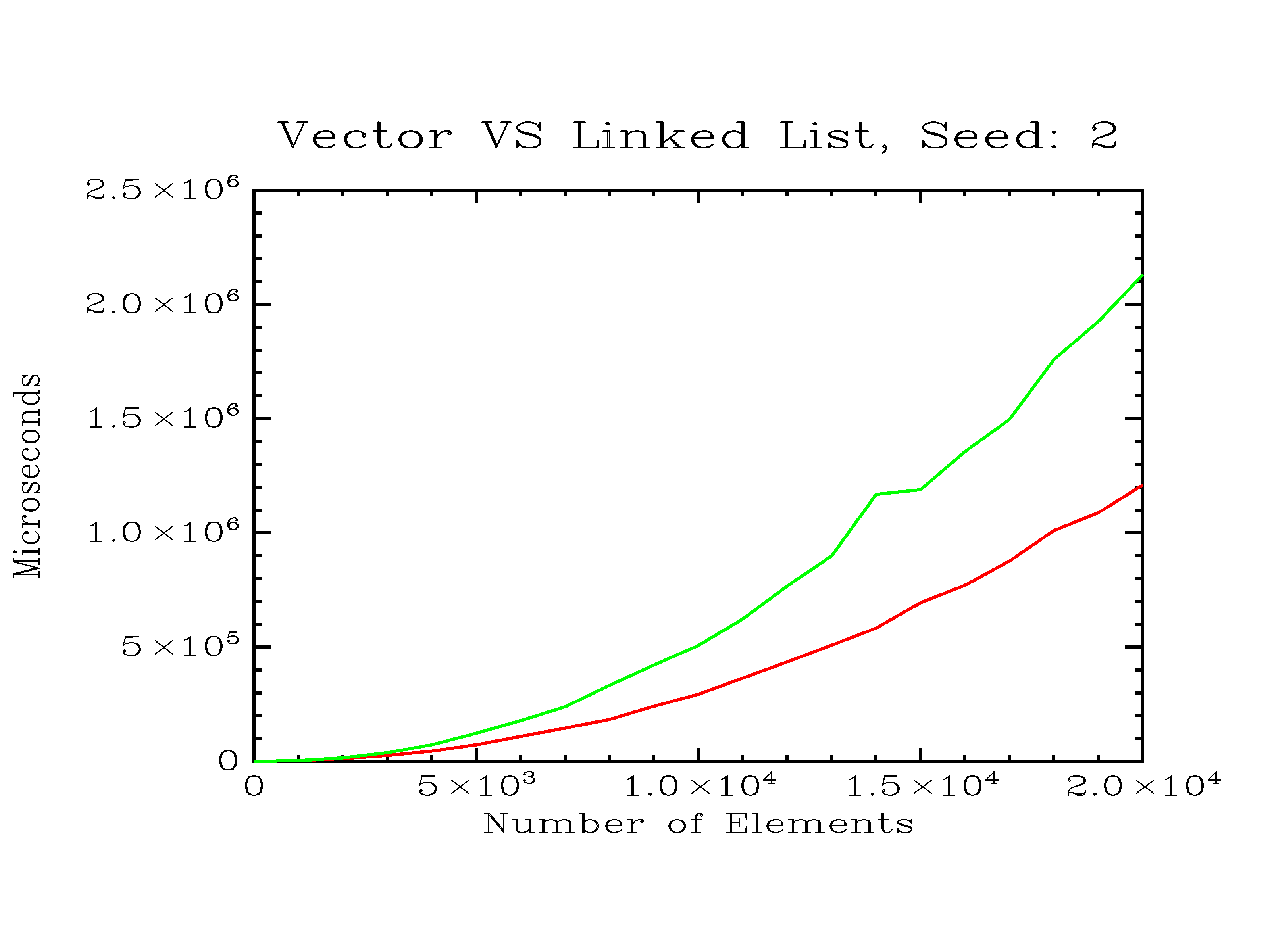
In total, the time complexity for vectors should be O(n3), dominated by element insertion. This is because for each element, an O(n) traversal is performed, followed by an O(n) insertion. For linked lists, the time complexity should be O(n2). For each element, and O(n) traversal is performed, followed by an O(1) insertion. Because of this difference in complexities, the linked list should eventually outperform the vector.

**Results**

The results of the experiment are shown below for various seeds. The green line represents the linked list’s performance, and the red line represents the vector’s performance. The vector **consistently outperformed the linked list**. This is contrary to my original hypothesis.







**Analysis**

The vector’s superiority over the linked list must be somehow accounted for. After discussing my results with several classmates, I formed the hypothesis that vector operations were being optimized through usage of the cache. That is, when a portion of the vector is accessed in memory, the memory locations surrounding it are pulled into cache. Then, since we’re accessing data sequentially, all of the shifting that occurs during insertion happens within the cache. For linked lists, on the other hand, all of the data is scattered across memory instead of getting stored contiguously. Thus, the cache cannot help the list.

For this hypothesis to be true, we should be able to bring down the speed of the vector by storing large pieces of data – so large that a relatively small amount of them will exhaust the size of the cache.