Chandler Teigen
CPTS 223
PA2
A.

The purpose of this assignment is to benchmark the Josephus problem with many different input sizes, and using two different data structures from the C++ Standard Template Library, the std::vector and std::list.

B.

For my algorithm design, each of the data structures was initialized by using a for-loop to iterate and using the std::push_back method to add a new Person object to the data structure. The position of each player ranged from 0 to N – 1. In order to eliminate the next player, I used an std::iterator, and found the Person that was M positions ahead of the current holder of the potato. Then I used std::erase to remove that element from the data structure. The potato's position did not need to be updated because the index of the Person before they were erased is the same index as the Person who will next receive the potato. If the next position of the potato holder was beyond the bounds of the array, I used the mod operator to calculate the position that the potato holder would have after the potato had "wrapped around" to the other side of the data structure.

C.

My experiment was performed on the EECS Linux servers, however the code was written and debugged using a Macbook Pro 2015 laptop running MacOS. I ran the code a single time to produce the output files that are present in the project. However, the average elimination times were found by averaging many eliminations during each game. So, the total times were only executed once for each game setup, but the average elimination times are the product of the same function being ran and benchmarked repeatedly.

D.

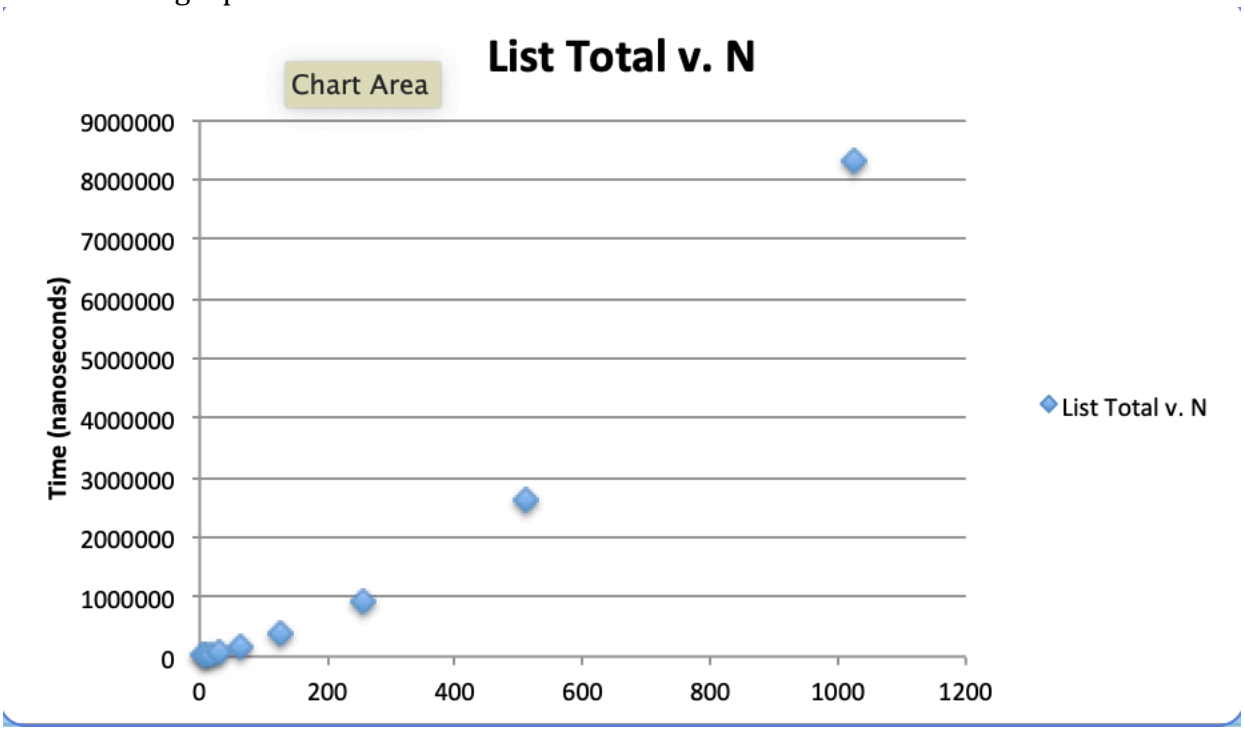
As you can see in the plots below, the total time for the std::list data structure was approximately twice as large as the std::vector implementations total time on average. The main difference between the two data structures is that the list is a linked list, and the vector is essentially a wrapper around an array. This means that the linked list cannot use indexing to get to a random value in the list, but a vector can. The vector has this advantage because when finding the next potato holder, its position in the vector can be randomly accessed using indexes. The List on the other hand needs to follow the links down the list in order to find the next potato holder. The list's advantage comes from when the potato holder is erased from the list. The list can simply erase the Person and rearrange its points, whereas the vector needs to shift the Person objects over to fill the gap in the data structure.

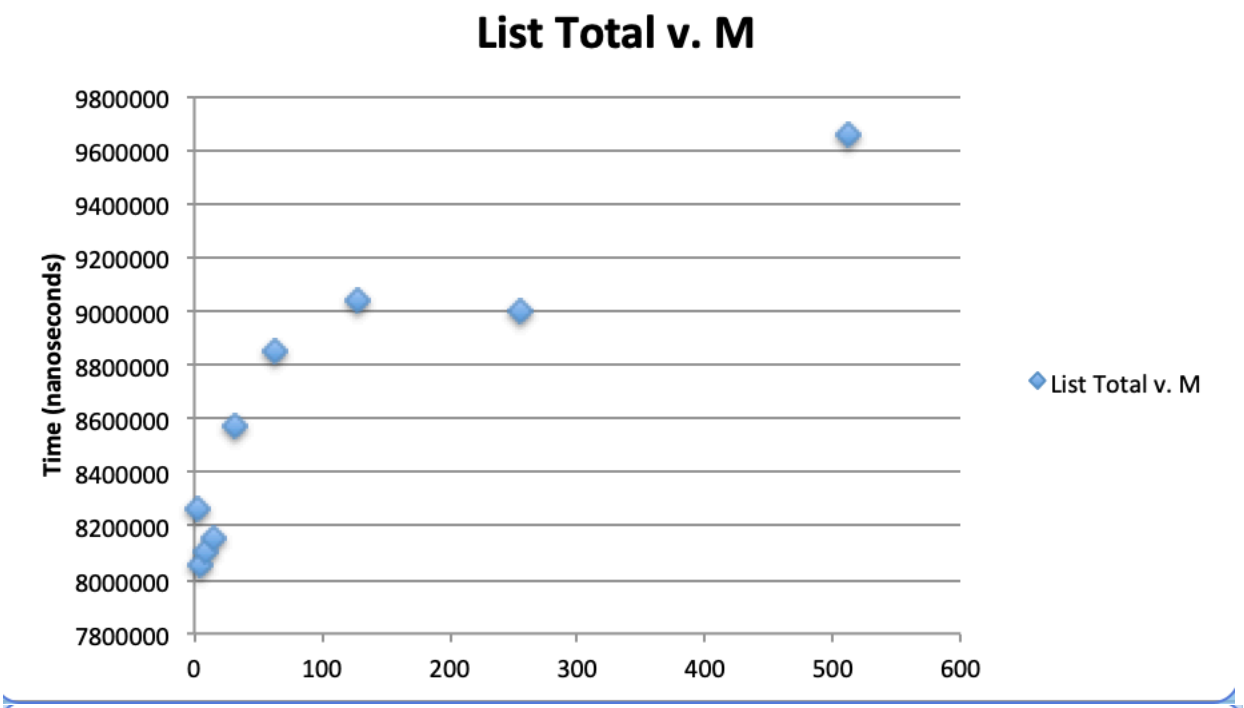
The average elimination time was also in favor of the vector as the list's average time was once again approximately twice as large as the vector's time. This seems backwards in my opinion because I would guess that the shifting of objects in the vector would slow it down for each elimination, but as the data shows, the list must be slower. I would conjecture that the following of the links in the std::list slowed it down more than the shifting of objects in the vector.

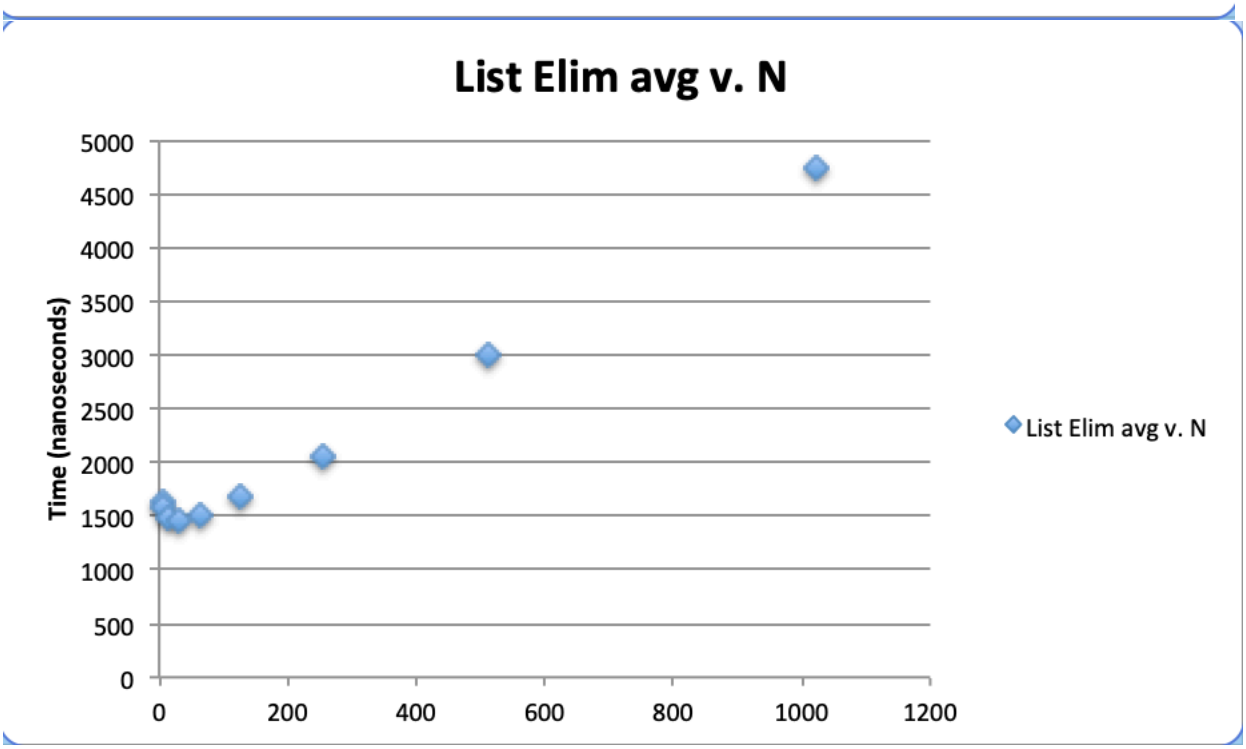
There are some differences between the run-time dependencies of N and M. The total time for both the list and the vector seem to increase linearly as N increases. The List implementation's run-time has a increasing trend as M increases, but it is not as linear as it is for N. The vector implementation on the other hand has a negative correlation with M, but the difference between the run-time values for the vector are not as great as with the list. I believe that M does not affect the vector run-time as much as the list run-time because the vector can use indexing to jump to the next position no matter how high M is, but the list has to use the links to traverse itself, which would take longer as M increases.

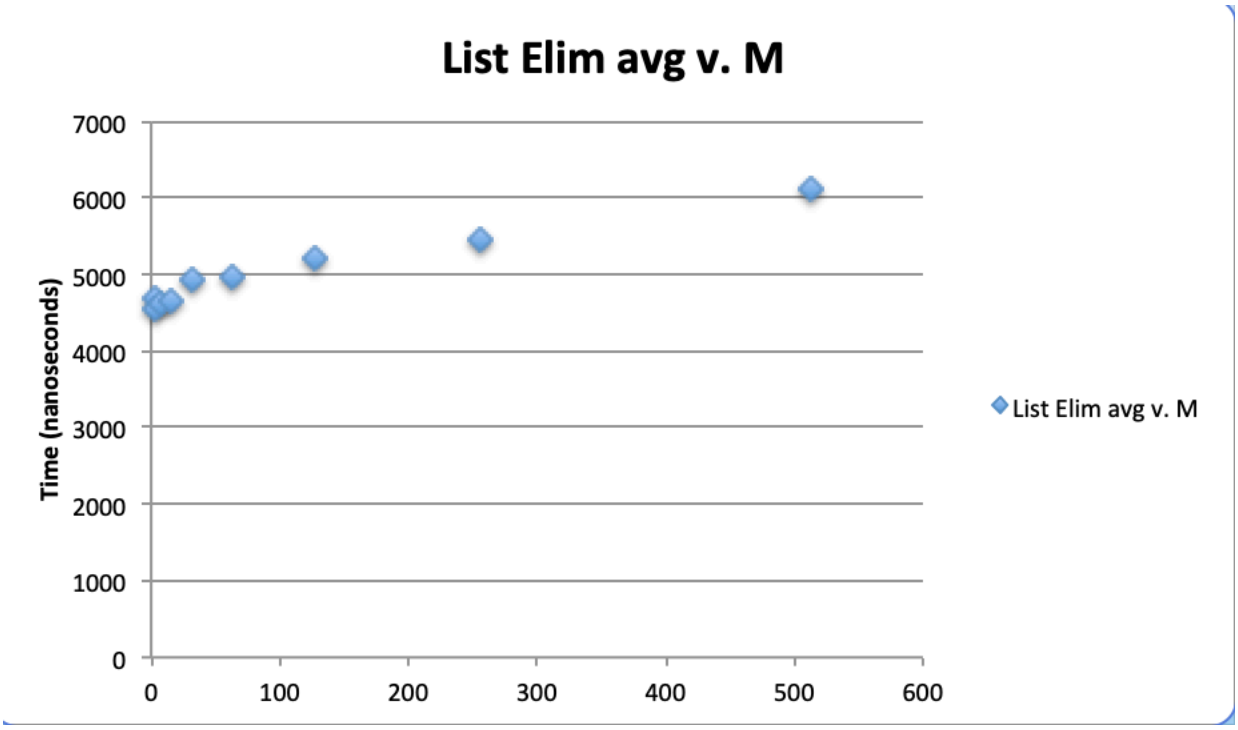
The list and vector also have differences when it comes to the changes in the elimination time with changes in N and M. For the vector, the elimination time increases very drastically when M increases because of the shifting of values that must happen in the vector when a middle object is removed. The changing of M does not change the elimination time as much however. In the list implementation has an increasing elimination time for both N and M, but the increasing N causes the elimination time to increase more. This does not make sense to me like the other results do.

The following 4 plots are for the std::list data structure:

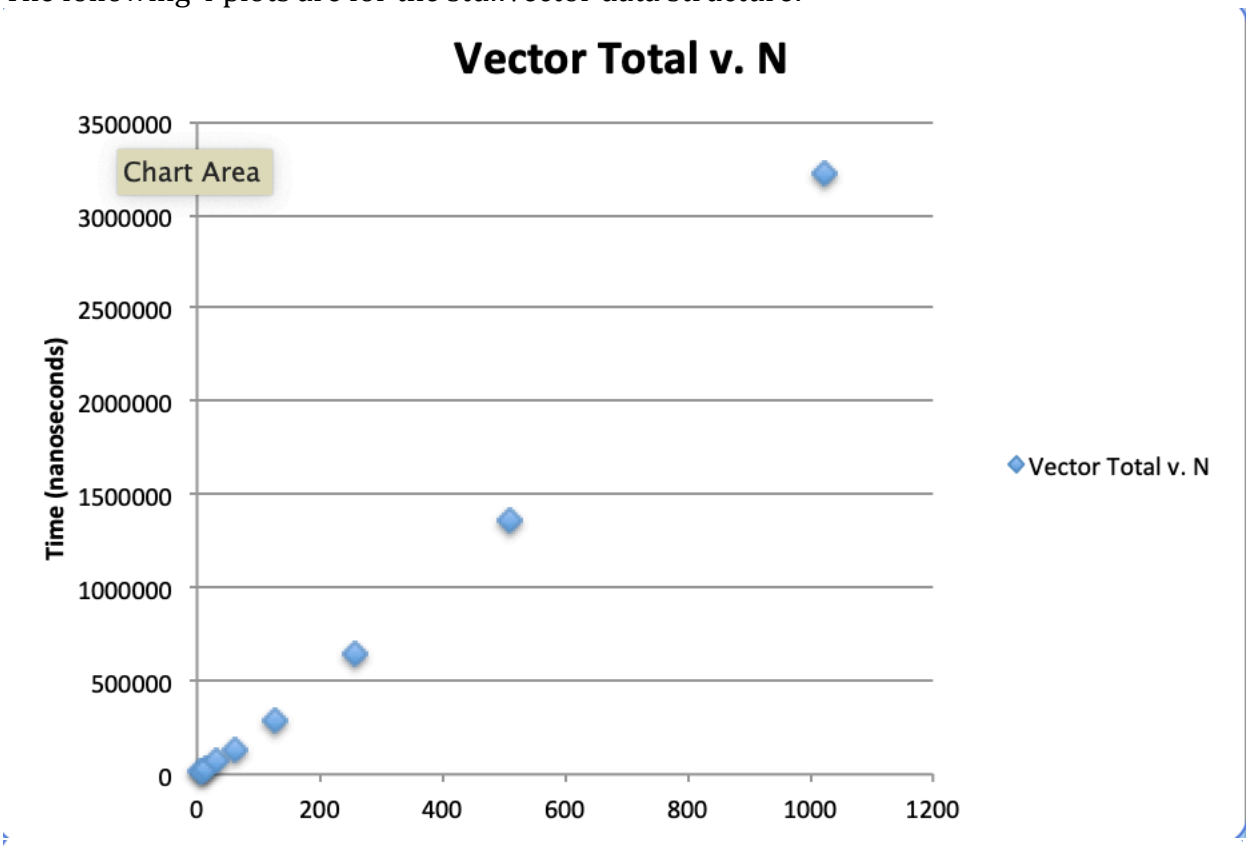


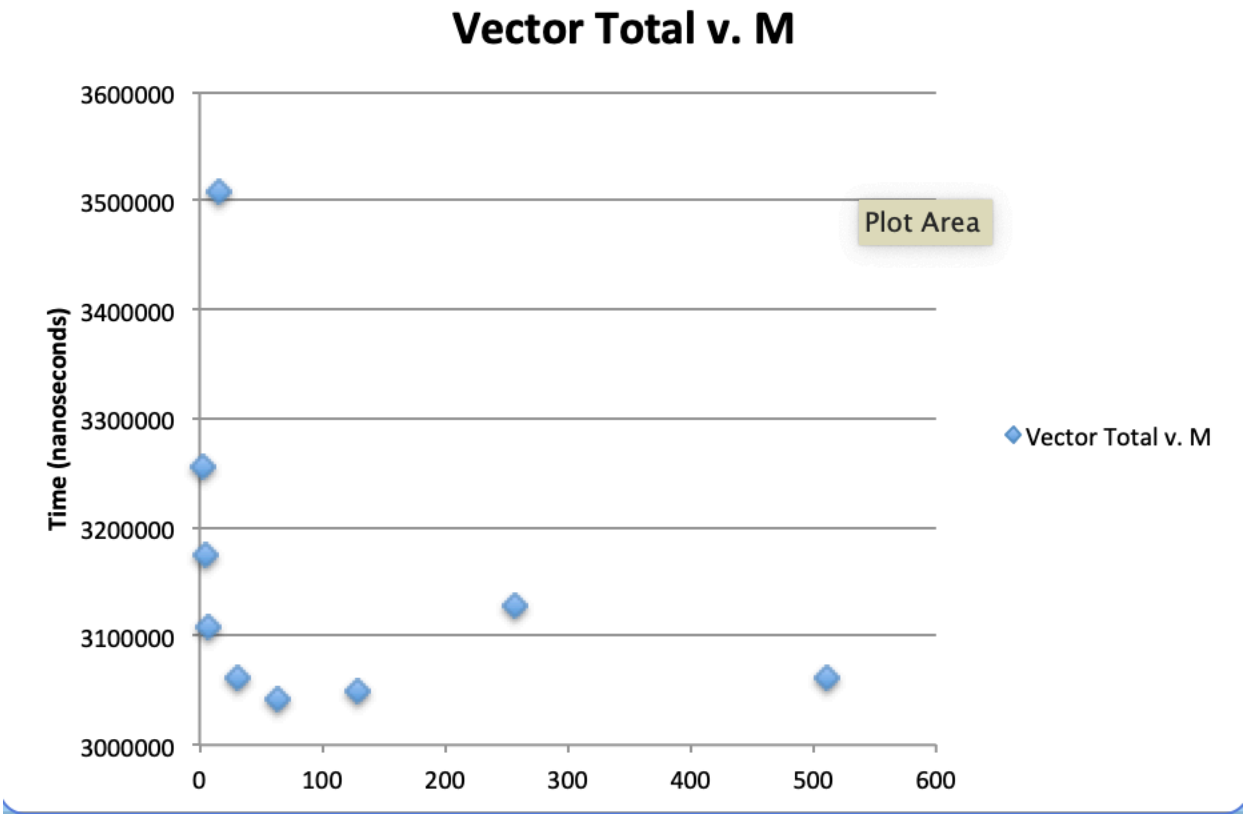

List Total v. M
Time (nanoseconds)
9800000
9600000
9400000
9200000
9000000
8800000
8600000
8400000
8200000
8000000
7800000
0
100
200
300
400
500
600
List Total v. M

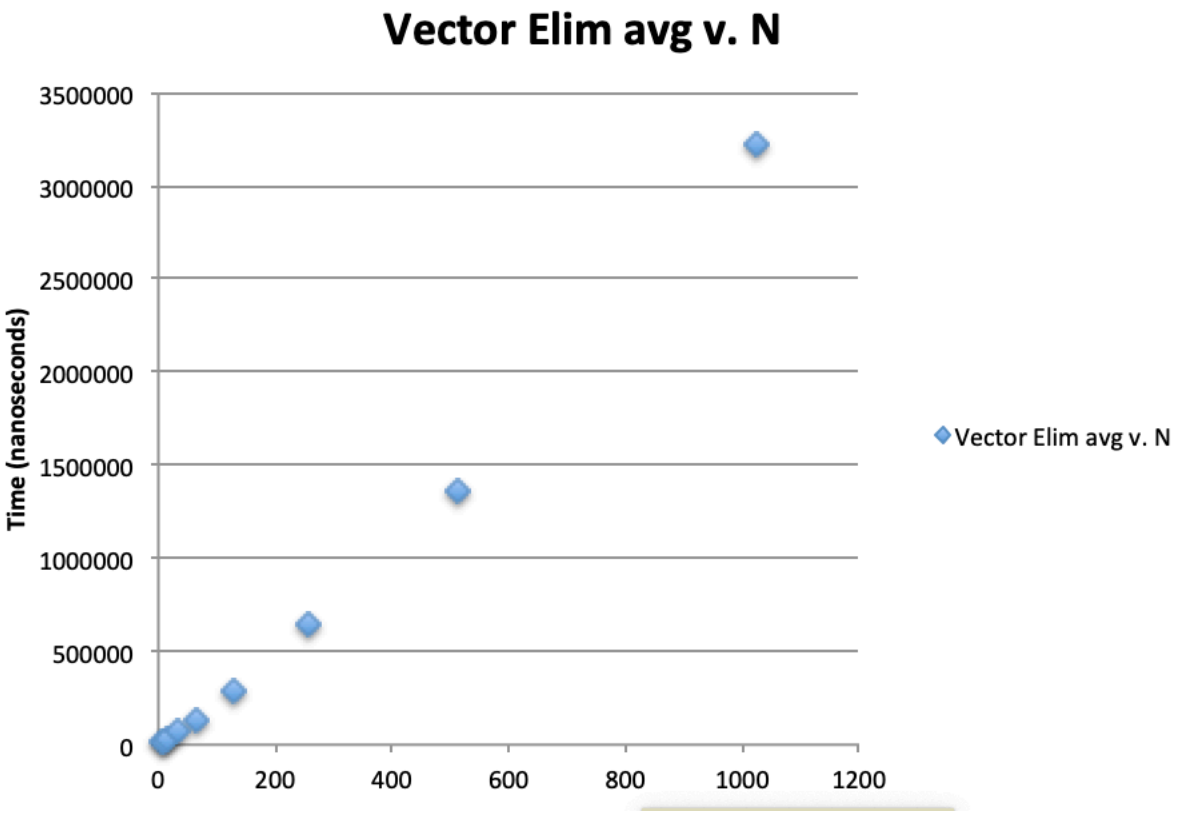

List Elim avg v. N
Time (nanoseconds)
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
0
200
400
600
800
1000
1200
List Elim avg v. N

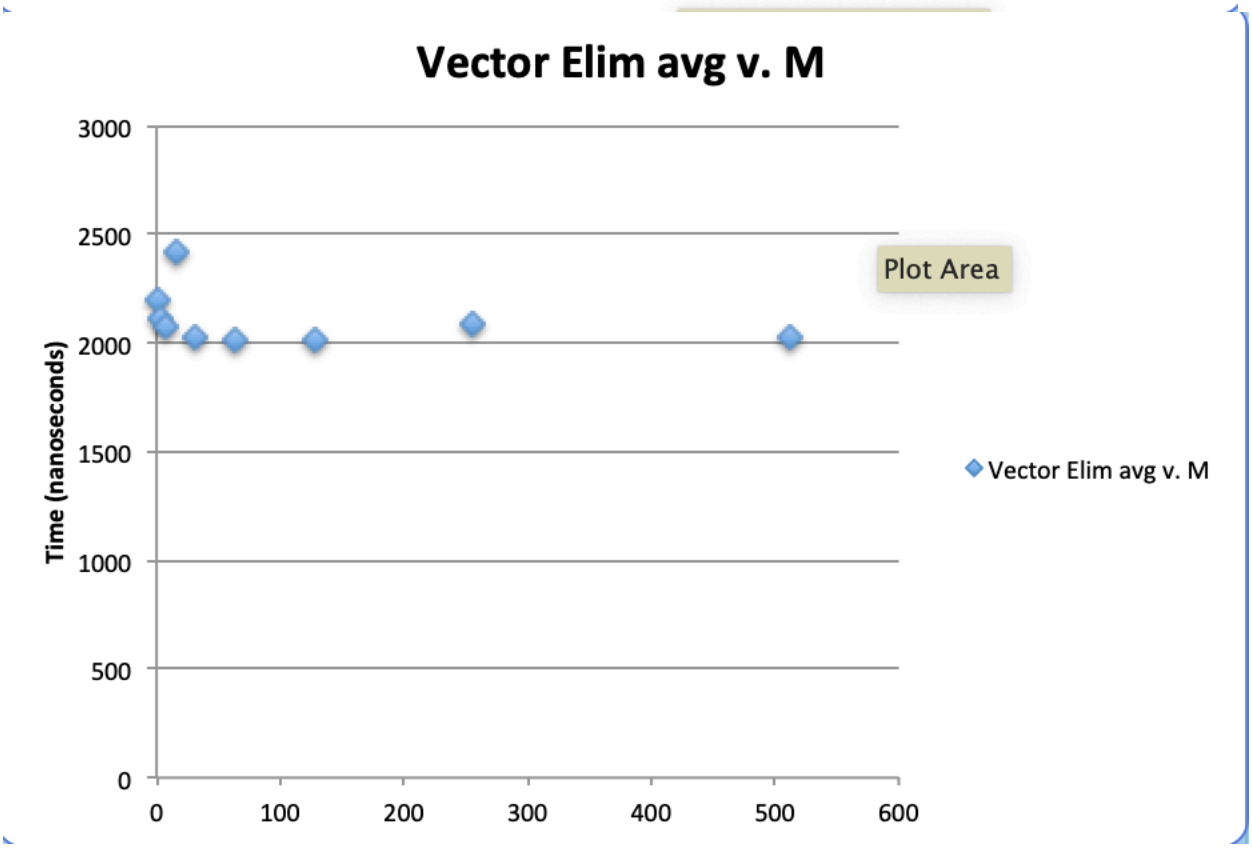

List Elim avg v. M
Time (nanoseconds)
7000
6000
5000
4000
3000
2000
1000
0
0
100
200
300
400
500
600
List Elim avg v. M

The following 4 plots are for the std::vector data structure:






Vector Elim avg v. N
Time (nanoseconds)
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
0
200
400
600
800
1000
1200
Vector Elim avg v. N


Vector Elim avg v. M
Time (nanoseconds)
3000
2500
2000
1500
1000
500
0
0
100
200
300
400
500
600
Plot Area
Vector Elim avg v. M