February 16, 2020

Matan Segal

CS 3353

Project 1 – Memory Management

Introduction:

In this project, we were required to implement a memory management which will the default allocator. In order to so, we will need to make an object which will contain all the chain in memory we will allocate foe the program. This object will be called through a singleton class, make it a unique object. The Allocator class will inherit to three subclasses which will have the kind of method our allocation will manage the memory chain. The first method is the first fit. In this method, the allocator will insert the new object in the first free location in the chain. The second is the best fit, which will search for the smallest size of free memory window which will be able to contain the new object and insert it there. The third method is our choice implementation. For this one, I chose to use a method which will insert the new object where the allocator pointer is pointed to after the last object. Once it will reach to the end of the chain, it will start inserting the object in the first free location, like the first fit.

My assumption is that the method I chose, where the insertion occurs at the last location of the pointer, will be the fastest among the three since it does not need to search for free location each time. However, this will be true until the pointer gets to the end of the chain, and at this moment it will be slower evermore than the best fit. Also, I think that the first fit should be faster than best fir since it does not need to search for best location but just for the first one which the new object can fit in. If we will look on the wastefulness of the methods, the best fit will be the most efficient in this parameter, the first fit will be second and the most waste method is the one I chose.

My implementation:

My Allocator contains a Bookkeeping 2D array, a freeList 2D array, and a freeAddresses map described as follow:

- The bookkeeping store just arrays in order to follow their length. It contains 100,000 elements in the first dimension and 671 in the second. It is ordered by the address. In order to find the index location in the first dimension we need to divide the start address of the node by 2684, where the start address is the difference between the actual address of the node minus the address of the beginning of our memory allocator. It managed like an order map, where the second dimension contains all the nodes between this start address and the next 2684 bytes.

- The freeList 2D array also act like a map where the first dimension is ordered by the size of the node. Once we will free memory, it will add it to the free list. the first dimension divided to groups. each group contains 100 elements (total of 1000 groups). the first group represents size of 0 to 396 bytes, second of 400 to 796 and so on. each first dimension will contain array of nodes with the exact same size. When we will reuse a specific node, It sets the start address to null so it will not use it again.
- The freeAddress is a stl map. the key is the start address, which is the difference between the actual node address and the allocator start address. The element is the size of memory. since it is a map, it will order the start addresses. It serves me when I am looking for resizing back to back nodes.

For all kind of fits, I am first checking that it is in a 4-bytes blocks using the mod operator. Then, I am checking to see if it is an array in order to insert it to the bookkeeping 2D array. In my fit, at this point I am checking that the next pointer, which points to the last location where insertion occurred in the memory chain, plus the size of the new object is less then the length of the chain, and add it at this location. However, in the first fit and the best fit, I am checking for free location in the free list 2D array using its organization by length. The first fit will search for the first location in the first group it fits in, while the best fit will look for the best location in this specific group. If there is no such a location in the group, it will move to the next group. If it does not find after all groups, the allocator will insert the object in the next pointer location. For my fit, if it pass the length of the memory (the “pool” in my program), the allocator will act like first fit and search for the first location in the memory that the new object will fit in.

When I am deleting an object, first the allocator will check if this object is an array by looking in the bookkeeping. Once the allocator has the size and the start address of the object it will check any back to back location using the map. If the delete occurred next to other free location, it will set the previous address value of the existing free location to null, then add the new free node adding the length of those location and choose the start location, which is the lower among the two free locations. Eventually, the allocator will insert the new free location to the free list 2D array.

Stress Tests:

the four test I wrote for my allocator are the following:

- Test 1: in this test I have a loop which each time will allocate a new int array with a random number of items from 1 to 256. Every other time it goes through the loop, it will delete the array it just allocated. The data size for this test is the number of loops the test go through.
- Test 2: in this test I made an object which contains: an int, long, char, string, double, and long long variables. Also here, I have a loop which the data size is the numbers of time it goes through this loop. Each time, it will allocate new array of the object with length of 1 to 10 according to the number of the loop. In addition, it will delete every fifth object.
- Test 3: this test also contains a for loop and the data size is the number of iterations through this loop. Each time it will allocate memory for a long array and a double array. The length of those are different but both depends on the number of the loop. The test also will delete arrays every 5 loops for the long array and every 15 loops for the double array.
- Test 4: this test is basically the combination of test 1 and test 2. It will run test one, then test 2. For the first test it will be similar to test 1, but when it run test 2, the allocator will also have the elements from test 1.

In addition, for those tests, I also add the sentimental analysis project from last semester. I am just parsing the twits and not going trough the testing part of the project. For this test, the data size will represent the number of twits I am paring.

Results of the tests:

When testing the allocator, we focused of two main factors: the running time for the test as well as the wastefulness of the specific kind of fit, where good indication for it is where the next pointer is pointed to after running the test.

Results from testing the running time of each fit kind:

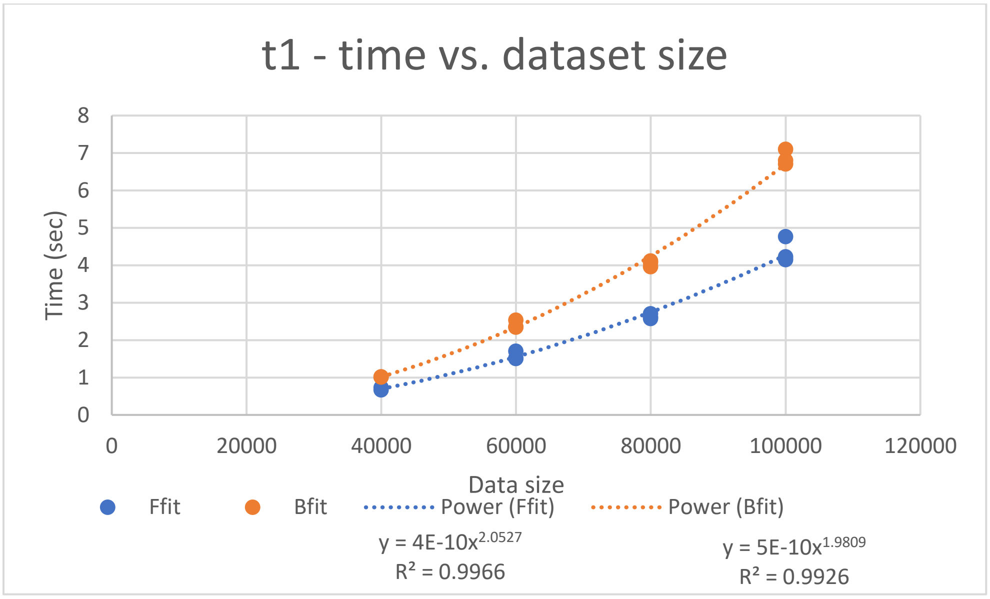

Figure 1 – time vs. data size for first and best fit, for test 1

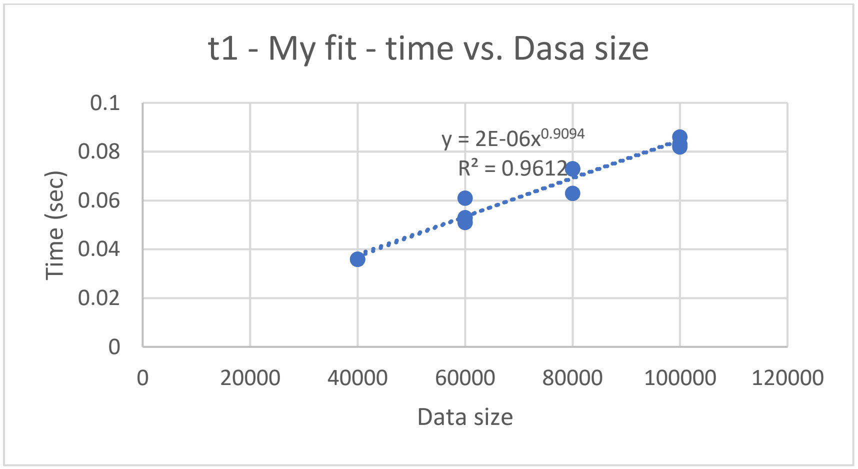

Figure 2 - – time vs. data size for my fit, for test 1

As mention before, the data size is the number of iterations through the for loop in the test. I plotted two graphs because the 'my fit' method was way fester than the other two. As we can see from the figure above, 'my fit' is absolutely the fastest method for the allocator. When using a power fitting for the graph of 'my fit' its power is very close to 1, refers to a linearity and a big O(n). when we look and figure 1, we see that both 'first fit' and 'best fit' are very close to a power 2 and a big O(n^2), however first fit increasing in time less than 'bests fit'.

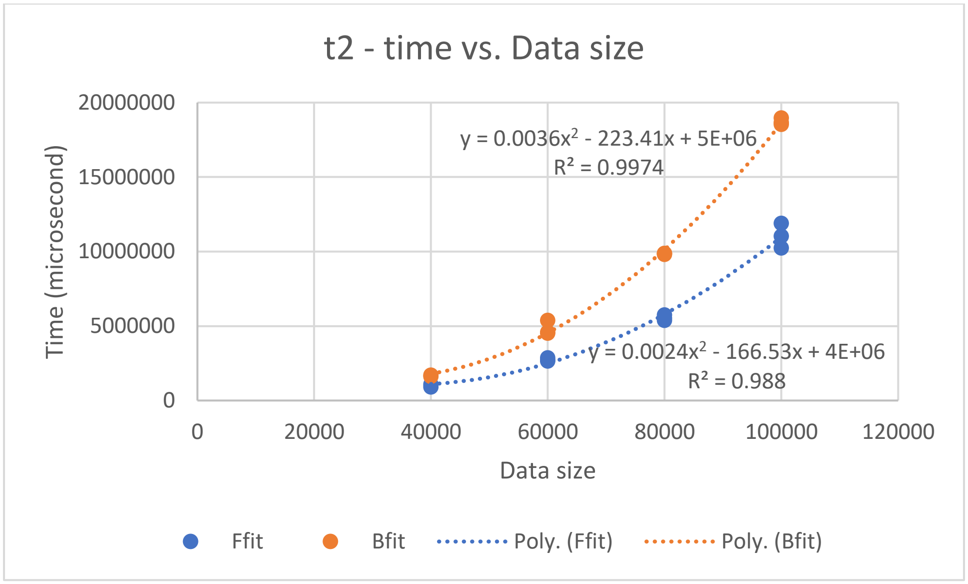

Figure 3 – time vs. data size for first and best fit, for test 2

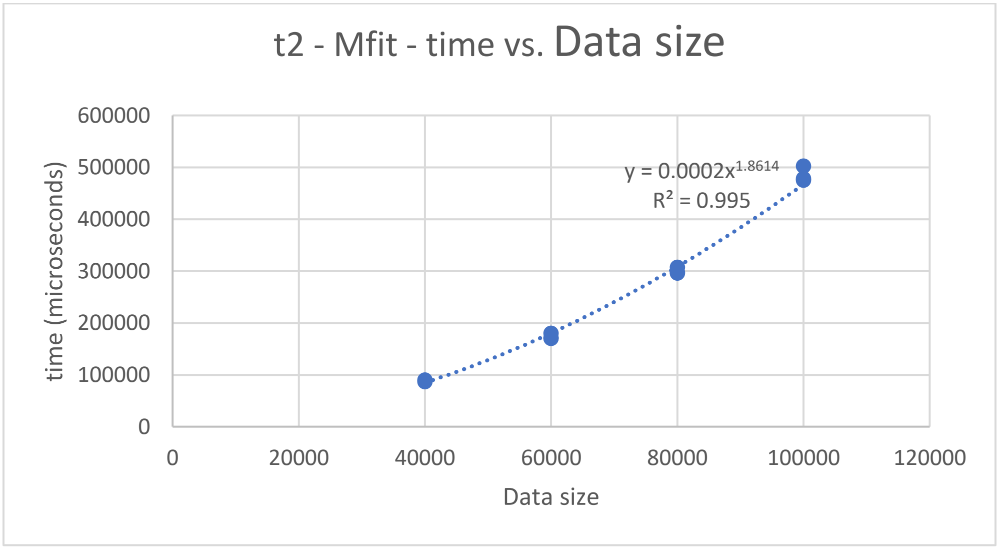

Figure 4 – time vs. data size for my fit, for test 2

From the results of test 2 in figure 3 and 4 we see very similar result like test 1. However, we can see now that ‘my fit’ even tough it still fastest, its power is now closer to 2 more than to 1, which refers to a big O(n^2). The other two were almost completely fitting the second-degree polynomial like before.

In the third test, we also checked about the next pointer:

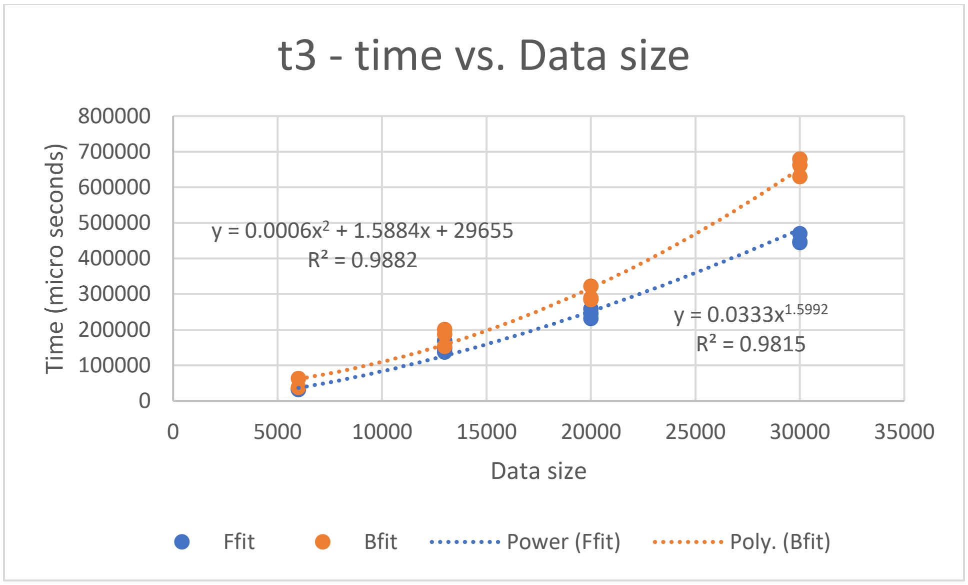

Figure 5 – time vs. data size for first and best fit, for test 3

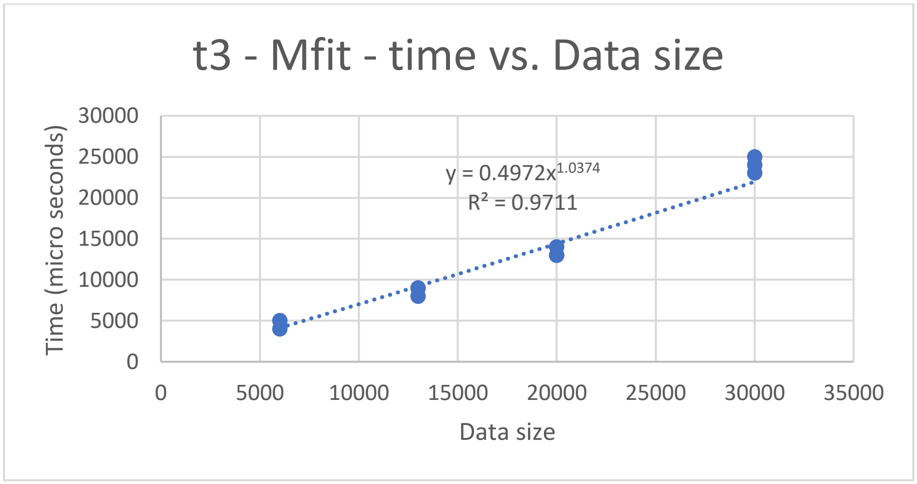

Figure 6 – time vs. data size for my fit, for test 3

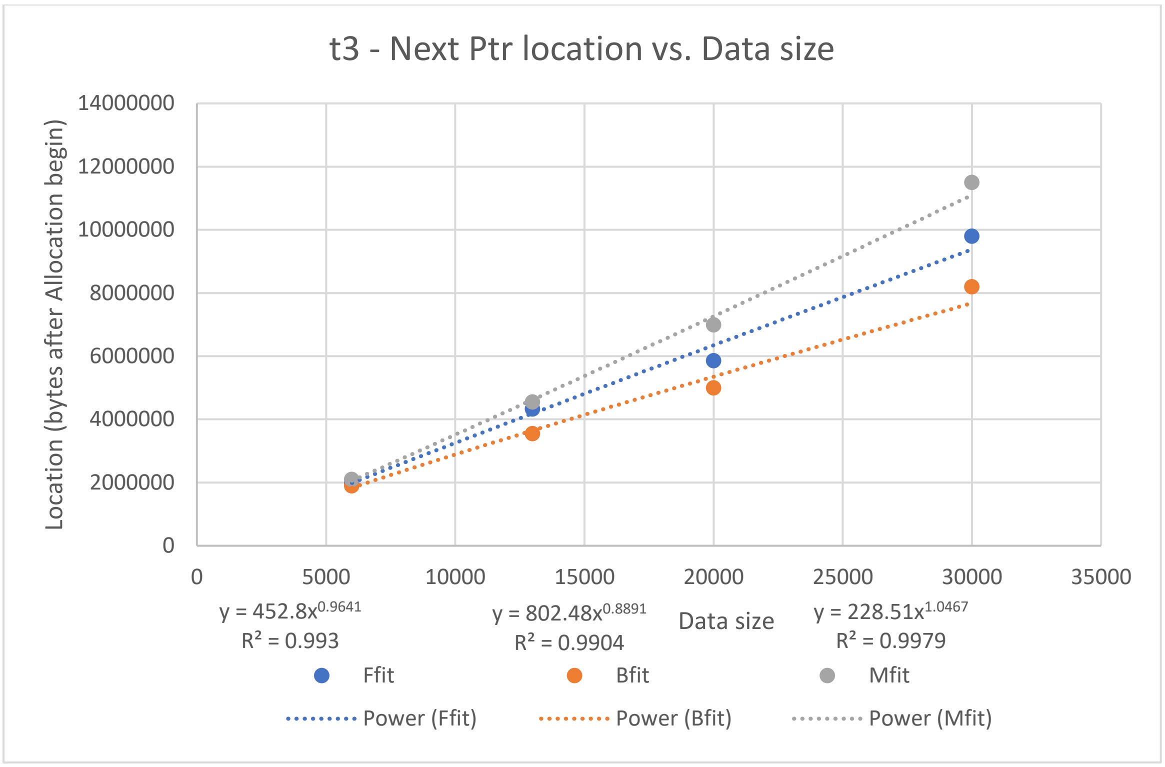

Figure 7 – Next pointer location as function of the data size in test 3

Like in test 1 and 2, we can see from figures 5 and 6 above that the order of the fastest to slowest stays the same. here, once again, the plotting of 'my fit' is more linear like in test 1. When we look at figure 7, we can see the location of the allocator pointer after last insertion to the chain. We can see a clear picture where 'my fit' was the fastest with the cost of being the most wasteful, means it has the most free location holes in the chain. While it is fine for smaller size of data, it will be problematic once the next pointer reaches to the end of the memory chain. We can also see that the 'best fit' is more efficient way in terms of 'holes' in the memory chain, like we expected it to be in the hypothesis in the introduction. While we have clear differences between the kind of fitting and the location of the next operator, all graphs powers are very close to 1, means all can be represented in a linear function and in a big O(n).

For test 4, which is the combination of test 1 and 2 we received very similar result for the time aspect like we had for the last three tests, therefore I will move to the testing of the sentimental analysis program.

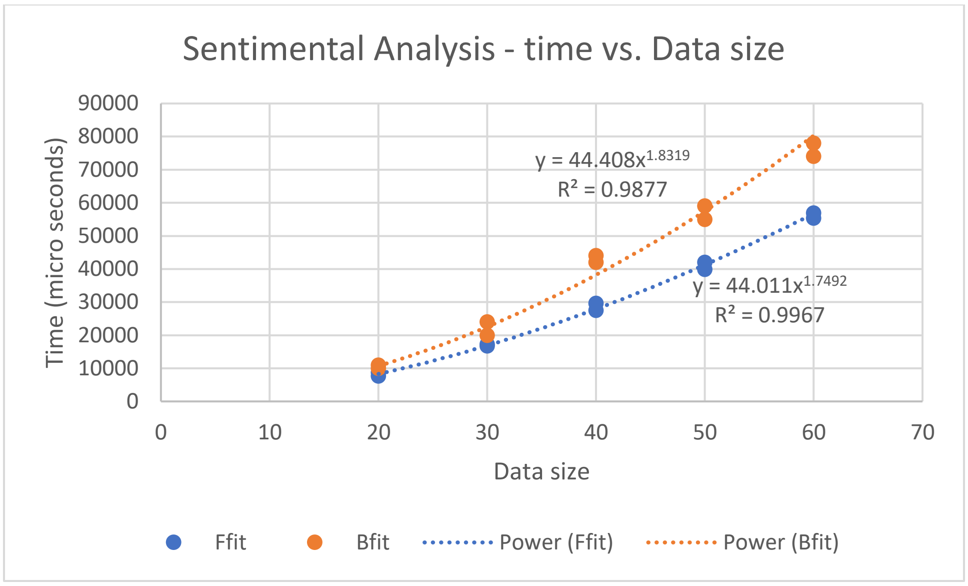

Figure 8 – time vs. data size for first and best fit, for the sentimental analysis program

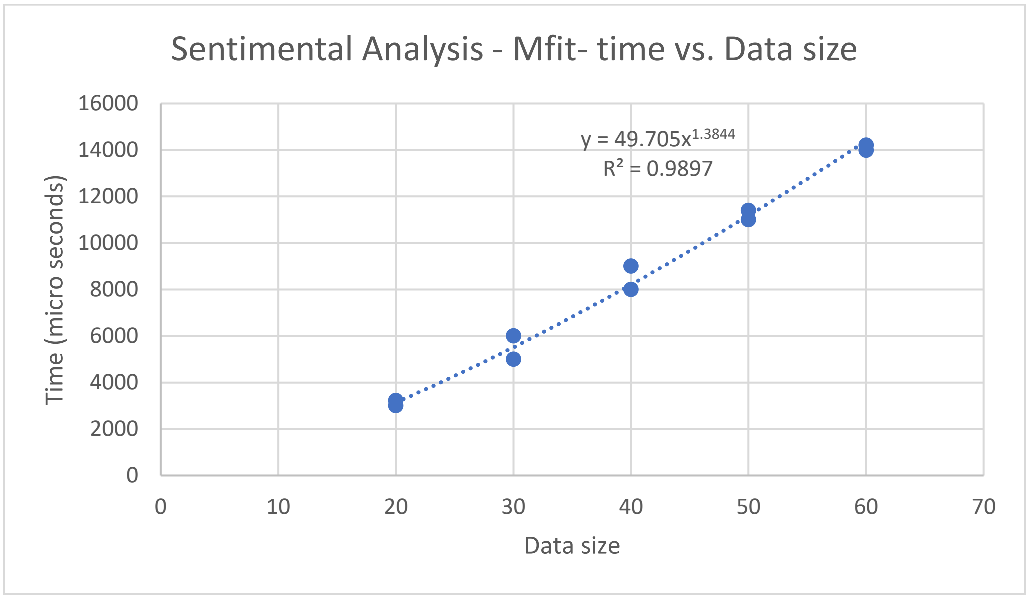

Figure 9 – time vs. data size for my fit, for the sentimental analysis program

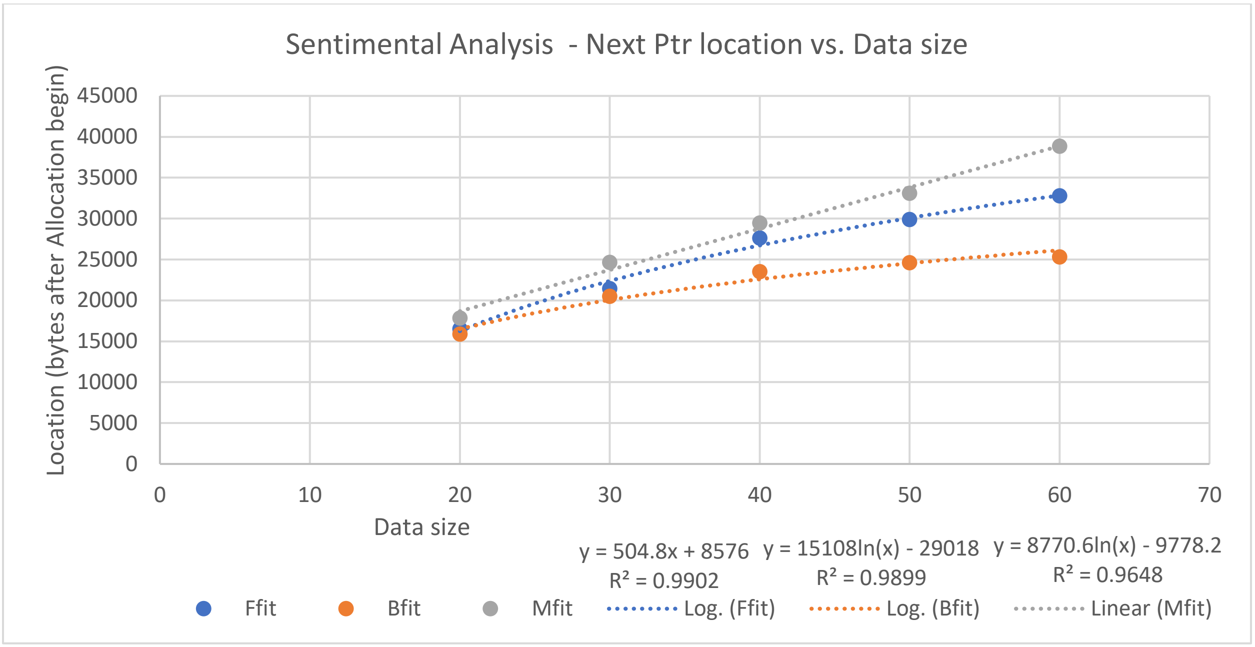

Figure 10 – Next pointer location as function of the data size in sentimental analysis program

In this case, the data size is the number of twits we are parsing. Once again, we see from figures 8 and 9 the same order in the run time aspect and the same order as before in the wastefulness aspect. However, this time we can see that while 'my fit' is linear in the location of the next pointer relate to the data size like before, the other two fit very good the logarithmic function. We saw before that the power of those function where less than 1 but here the both increasing slower. In my opinion it is happening because before the test delete many unrelated size of blocks, in the sentimental analysis most of the words are 4, 8 or 12 bytes blocks (after adjust them to a 4 bytes block manner) which make the insertion fits much better and being more efficient compared to different sizes of blocks like in test 3.

<u>Comparing results with teammates:</u>

In my team, all of us use the same method for 'my fit' which is adding at the next pointer location. However, we implemented it differently. According to one of my teammates, the first fit was faster than 'my fit'. We can see his results from the figures below:

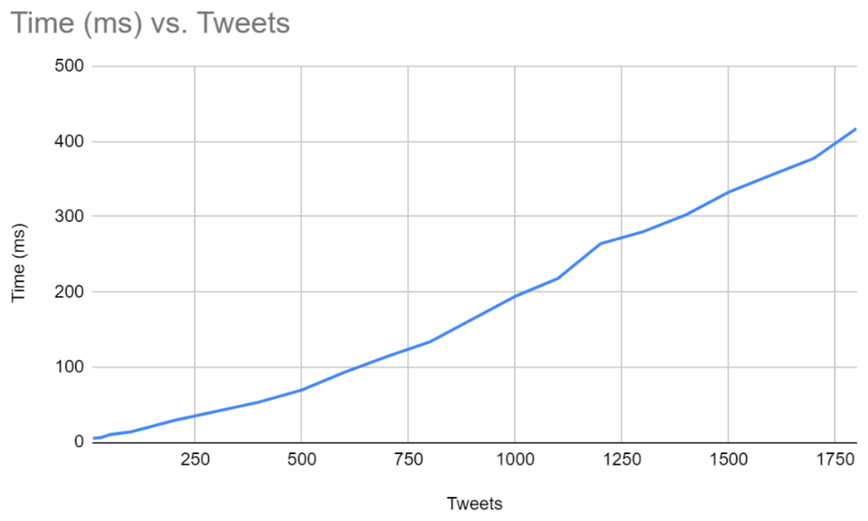

Figure 11 – teammate's 'first fit' time vs. tweets using sentimental analysis test

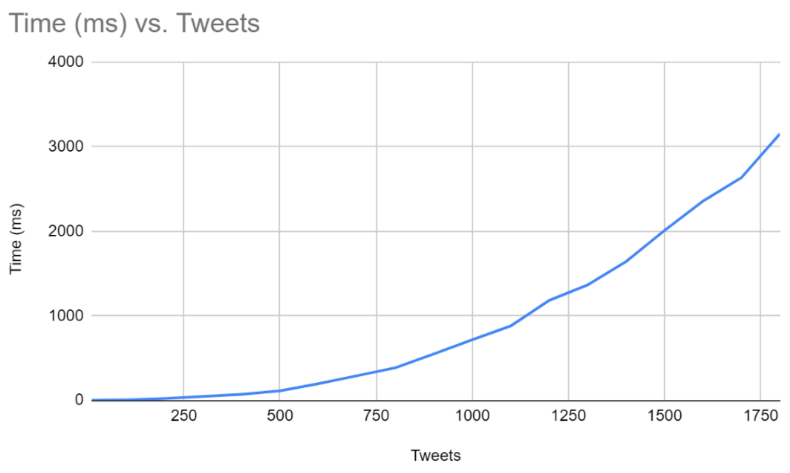

Figure 12 – teammate's 'my fit' time vs. tweets using sentimental analysis test

As we can see in figure 11, 'first fit' is almost linear while 'my fit' is look like a second-degree polynomial. We figured out that in his implementation he did not use a next pointer, but each time went trough the all array in order to get to the end, while I had the pointer to the last location. Therefore, his first fit looks almost linear when the search pointer can find quick free location, and 'my fit' need to go through the whole array which is like a sum of a mathematical series, or a big O(n^2).