Shawn Tan
CSCE 313

Ackerman Analysis

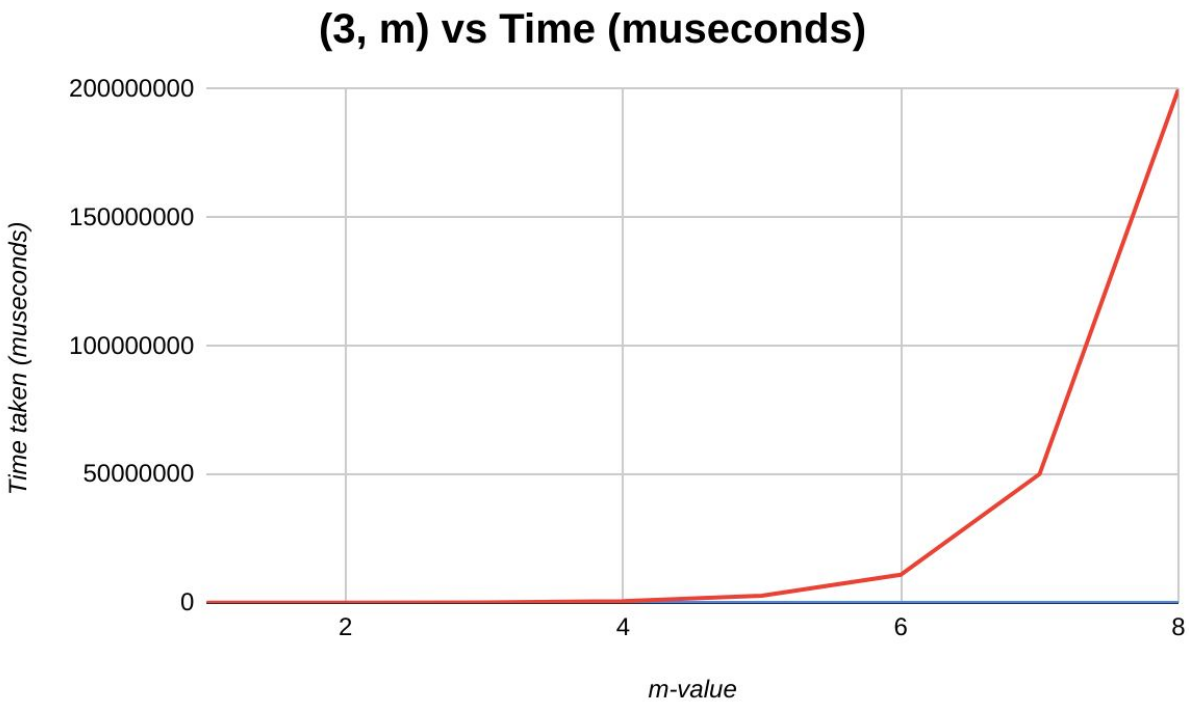
Though the majority of the code was written in an attempt to pass the "easy" tests provided by the programming assignment, the meat of the debugging came from trying to successfully run a series of Ackerman function calls. The results of said tests are listed below (each graph represents tests with a fixed n-value and varying m-values):

(n,m) = (1,x)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Ackerman value	3	4	5	6	7	8	9	10
Time Taken (musec)	212	2845	831	8567	9450	5933	5128	3803
Number of Cycles	4	6	8	10	12	14	16	18

(n,m) = (2,x)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Ackerman value	5	7	9	11	13	15	17	19
Time Taken (musec)	8034	7410	7394	11841	36275	21945	42564	35396
Number of Cycles	14	27	44	65	90	119	152	189

(n,m) = (3,x)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Ackerman value	13	29	61	125	253	509	1021	2045
Time Taken (musec)	20693	79055	200293	709381	2774172	10931727	50128368	199865851
Number of Cycles	106	541	2432	10307	42438	172233	693964	2785999

However, the table-representations of these runtimes don't paint a very good picture of the trend that is occuring with each of these tables. A better representation of this data can be seen graphically, as you will see below:



As you can see, the trend for (3,m) seems to be growing at an exponential rate as one increases the m-value. However, if you look at the table data for (2,m) and (1,m), you will see that the trend is somewhat erratic. This is because the difference in time of completion between tests is so small that it essentially makes it negligible. However, (2,m) does show a bit of an upward increase in runtime as you make your way to the right of the table.

There were different ways in which I could have increased the efficiency of the program. During the initial process of writing the program, I initially looped through the freelist in order to find the index I needed to insert/remove from. However, this was incredibly inefficient and (unexpectedly) led to a large number of segmentation faults. Thus, I derived a formula in order to compute the index values based on the basic block size and the block size of the blockheader I were trying to insert. Not only did this resolve multiple segmentation faults in my program, but the formula also helped me directly jump to the index in which I wanted to insert into. This helped to decrease the runtime of my program by removing the need for a search on my freelist. However, one problem that I didn't manage to fix was the random pointer declarations that pointed to the same address as a formally delcared pointer. This increased the overhead and runtime significantly as there my many needless variables being added to memory that could've slowed down the process of running various functions in my code.