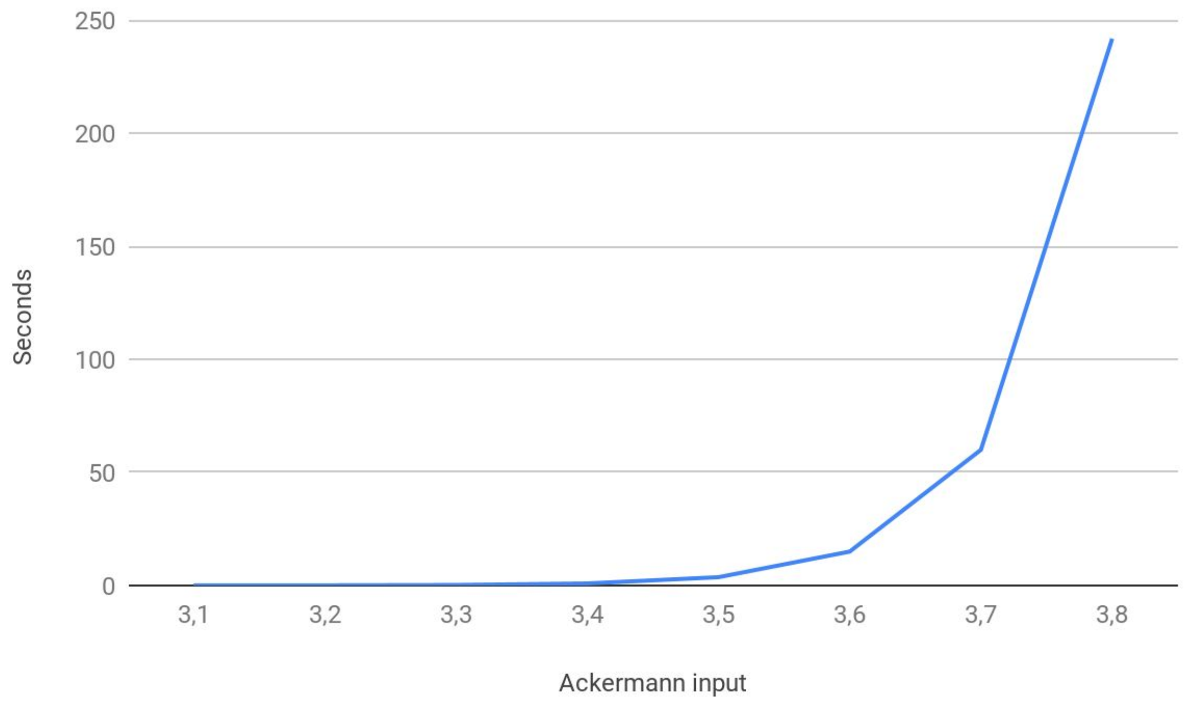
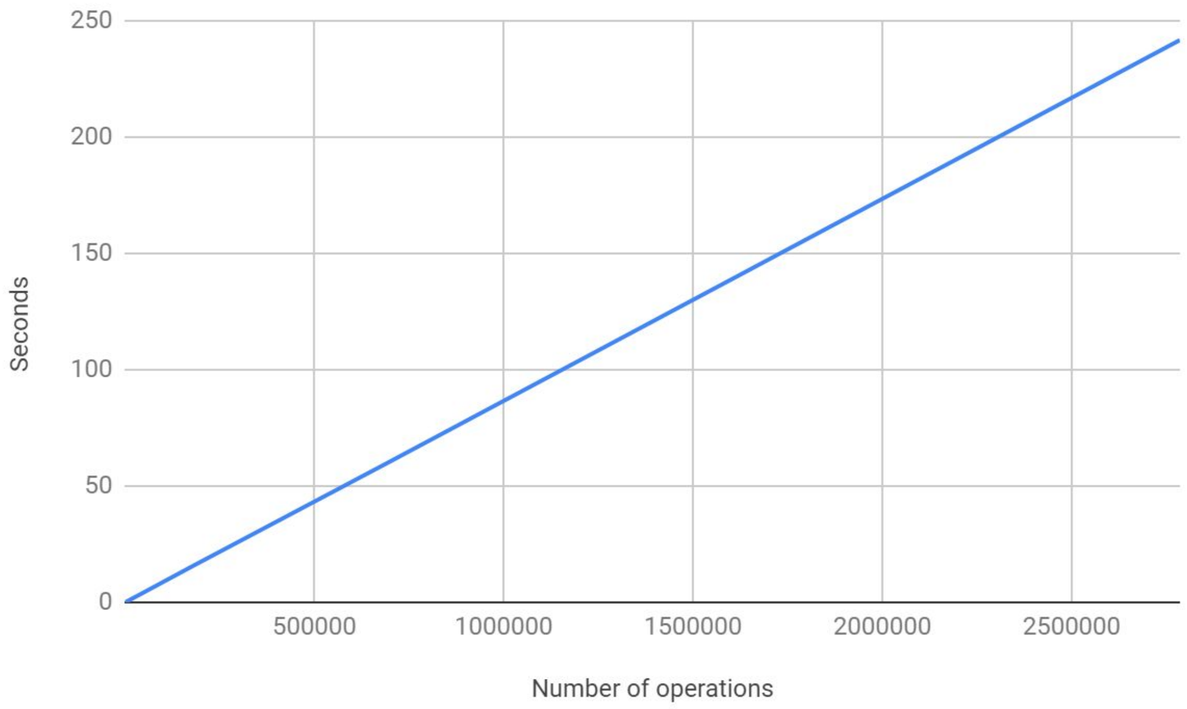
Thomas Hari Budihardjo
UIN: 126009556

Quite unsurprisingly, we find that the more complex the Ackermann input the more time it would take for the program to complete. And since the Ackermann Function is recursive, it should also be unsurprising that the time it takes to complete grows exponentially. I choose to not graph the other values of n (n=1 and n=2) because they all took less than a second to run, and thus would only be represented by a line across the bottom on this same graph.



What is perhaps more interesting is how the time it takes to complete the program is actually linearly related to the number of operations it makes. This graph even goes through the origin without being forced to, which makes sense since if the program takes 0 operations it would complete instantly. But the way that each operation seems to take a set amount of time to complete is quite confounding to me because I thought that different operations would require different amounts of time to complete, based on the state of the free lists. (For example, split would take tons of time to complete for allocating 1 byte of memory when we only have one block in the beginning, but split would not even execute if we try to allocate any valid memory after this because every free list except for the largest would have one block in it.)



In my program, the bottleneck lies in the way the split function is implemented. In my code, whenever we split a block, we would remove that block from the current free list and add the two blocks that are created from the original block into the free list of the smaller size. Instead of having one remove and two insert free list operations for each split, we can have only one remove call at the beginning, removing the block from the initial free list that it exists in, and only one insert call per split block to add the buddy of the block that we keep throughout the split function. The implementation of this would require some type of “first time flag” where it would only remove from the free list if it was the first time that we enter the split while loop, and this change would not increase the overhead cost significantly at all, since we are only adding one variable into the program. This would improve performance because it would require less removing and inserting into the free list, operations that take time and resources. The benefit of this improvement would be felt more when dealing with smaller GBs of memory because they require more splitting.

In my testing, my program works completely as intended. From a program-design perspective, I do not know how much it differs from what we discussed in class because I did not look at the pseudocode provided at all, choosing instead to think about the program wholistically and design the program from the skeleton code provided.