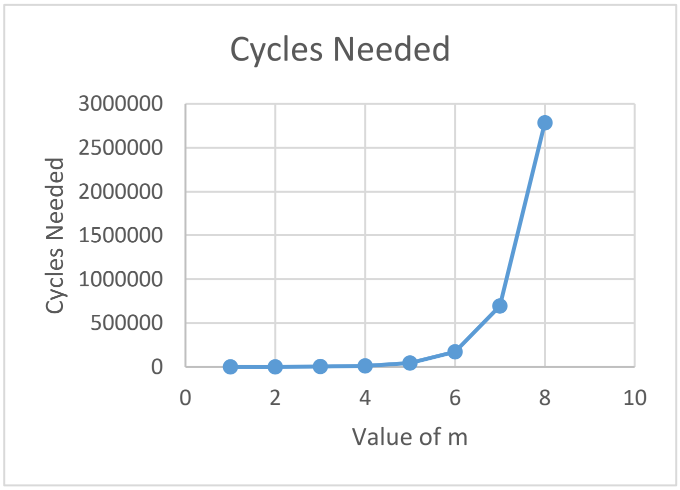
David Qin
PA1: Buddy Allocator
Dr. Ahmed
9/14/18

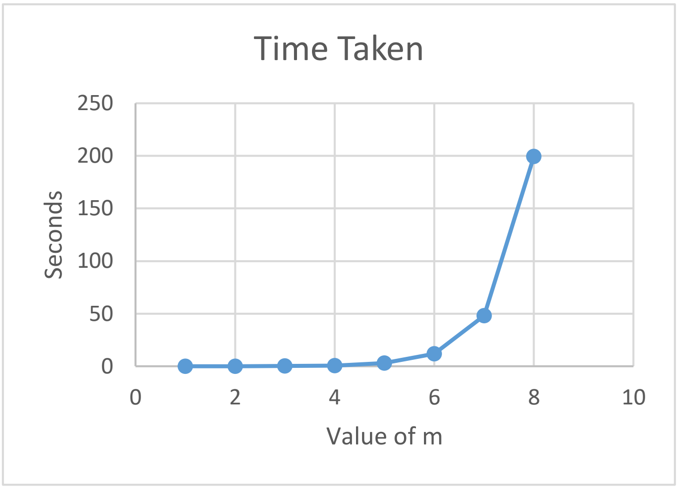
My Buddy-System allocator will take two command line arguments, basic block size and maximum available memory size. If user don't input, the default value will be 128 and 512K. From here, the program will do memory management. The program will create a vector of block headers. The vectors will be separated based on the size of the memory block, and the linked list headers will have new block of equal size that will be used by the caller. The program will try to split block if the caller's request is small or merge blocks together if not in use. This allows the program to act faster since the memory blocks aren't returned to system directly. Nonetheless, the program was able to run Ackermann's function successfully with 128 MB of memory at A(3,8) for the maximum input.

Below are the results of running the program using n = 3 and m = {1,2…8} as user inputs as well as 128MB of memory.

Table 1: Results of test using n = 3

m Input	Time Taken(s)	Cycles Needed
1	0.008669	106
2	0.041118	541
3	0.163145	2432
4	0.721883	10307
5	2.902932	42438
6	11.928764	172233
7	48.043031	693964
8	199.361953	2785999





The largest bottlenecks can be observed when large amounts of memory are initialized but only very small block sizes are needed. Maybe the caller only needs 10 of the 128 bytes from the smallest memory block size, causing the program to continually break and merge, taking up time, and only using a small amount of the memory. There will be increased fragmentation as a result. This is the worst case scenario, but if you knew that the caller would only use the allocator for small amounts, you could alter the program to only break blocks, no merging, in order to save processing time.

The slowest part of my implementation would probably be the alloc function. Here, the algorithm takes O(n^2) time in order to finish. When the caller asks for a certain amount of memory, the program must first find the adjusted size to the nearest power of 2, which in my case uses a while loop. This could've been altered to take less time without iteration. Meanwhile, searching for an available block requires traversing through the memoryheader vector at a speed of O(n) and traversing the list at O(n) again. One way to improve the speed, would be to change how the linked list is structured. Give it a tail, and when a block is needed, use one from the head and move it to the tail, this allows the list to operate in O(1) time instead, and can be faster in large-scale allocations.