**PA1 Report**

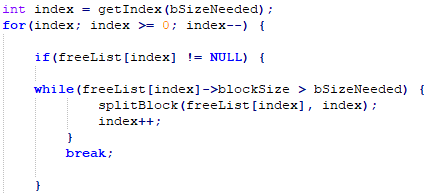
I managed to get my allocator working perfectly in time after many hours of struggle. I like the struggle though, that’s how I learn the most and I definitely learned a lot in the process of completing this assignment.

The performance of this program ran in linear time after testing it against different inputs with the Ackerman function.

This means that the program ran in O(n) linear time where n is the number of times the my\_malloc and my\_Free function were called. This is a pretty good performance because the only way to improve it is by either improving the constant factor of the linear time, or somehow making the program run in logarithmic or constant time.

Init runs in O(1) time, addBlock runs in O(1) time since a new block is always added to the beginning of the freeList, deleteBlock runs in O(n) time because in the worst case, the block that needs to get deleted is at the very end of the linked list and the while loop has to iterate all the way to the end. This however doesn’t happen too often because the block to be deleted won’t be the first block unless it is the address of the buddy. SplitBlock runs in O(1) time because the block to be deleted when it is split will always be the very first block of the linked list so delete happens in O(1) time in this case. And add is O(1) so split will be O(1). MyMalloc is O(n) time because in the worst case the for loop will have to iterate from the largest index to the smallest index if the bbs is the block size needed and the only block available is the size of the memory. MergeBlocks is O(n) time because in the worst case a buddy that is going to be deleted is all the way at the end of the linked list. Therefore, my\_free is also O(n) time.

When the user requests memory, I calculate the block size needed to be allocated by adding the size of the header to the requested size from the user and then finding the next power of two. I then calculate the index I need to access the freeList of that block size. If the freelist is null at that location, meaning that there are no free blocks for that size, then the index would be decremented to search if a block of a bigger size exists.



This becomes inefficient when the block size needed is the size of the basic block size and the only block available is the size of the memory because then the for loop would have to iterate all the way through to the very first index. If you knew what blocks were available, you wouldn’t need to check through all the empty blocks to see if they can be split. To do this I could implement an ordered set and insert the indices of the available blocks into the set. So now, instead of starting at the basic block size and iterating all the way to the block that is the size of the memory, you could simply call set.upper\_bound(index of block size needed, which is bbs in this example). The lower\_bound function would return an iterator to the location of the first occurrence of an item not less than the index (you would need to subtract 1 from the iterator to get desired value). So for if bbs was at index 20 and the only available block was the size of the memory at index 20 then you would set.insert(0) and then set.lower\_bound(0)-1 would return an iterator to index 20. Both insert and lower\_bound functions run in O(logn) time which is much more efficient than linear time. Set is a c++ data structure implementation.

gcc -std=c99 -lm -g -o memtest \*.c && ./memtest -b 128 -s 134217728 can be used on putty to run the function

some function prototypes were added in the .h file so that they could be called by other functions. Header was also included in the .h