Aditi Singh and Rumeet Goradia (as2811 and rug5)
CS214, Section 4
Asst1

In this program, we were tasked with designing our own versions of malloc() and free() to simulate dynamic allocation in C. Our program uses a header file, mymalloc.h, to replace any calls to malloc() and free() with calls to our versions, which are named mymalloc() and myfree() respectively. The header file also includes the declaration of a static unsigned char array of size 4096 called myblock, which is used to store all allocations and essentially to simulate a heap.

The data in myblock is tracked through various metadata that is stored within the array itself. This metadata is always 3 bytes long; the first byte stores the smaller byte of the inputted size, the second byte stores the larger byte of the size, and the last byte stores a char to specify whether the allocated memory is in use or not. We decided to use individual indices within the array to store the metadata instead of using a struct in an effort to minimize the size of the metadata, as a struct with the same parameters would use up 4 bytes of memory due to padding. We also stored a magic number in the first 2 bytes of the array as a check for whether or not malloc() had been called yet. This magic number was chosen by random prior to coding began, and, as it takes up 2 bytes, has an incredibly small chance of being there prior to the first call to malloc().

mymalloc() and myfree() each operated in a very efficient way, with a total combined time complexity of O(n), where n is the total number of allocated pointers, freed or unfreed. mymalloc() first checks if the magic number is present in myblock, and if it is not, the function returns the pointer returned by init(). If it is present, meaning that malloc() had been called previously, the function traverses through the list of allocated pointers, incrementing each time by the size given in the first two bits of each pointer's metadata and checking to see if the pointer is free and has a big enough size for the current allocation. If it finds that these conditions are met, it returns a pointer returned by create(), and if the function exhausts the list of allocated pointers without meeting these conditions, NULL is returned with an error message stating that the requested size is too big. create() and init() behave somewhat differently since init() deals with the first allocation of a pointer and therefore cannot read any data from the array and must set the magic number in the array; however, both of the functions essentially have the same task, which is setting the current pointer to being not free and adjusting the size of it by changing the values in the aforementioned metadata, and subsequently calling the split() function. The split() function creates a new metadata block after the current pointer, labels it free, and decides the value of this block's size bytes using the difference between the original size of the current pointer and its new size. It is important to note that create() only changes the current pointer's size and calls split() if there is enough space for new metadata so as to be efficient, to stay within the bounds of myblock, and to prevent overriding any metadata located after the current pointer.

myfree() starts with many tests to ensure that the pointer passed to it is valid, meaning it is not NULL, it is a legitimate pointer, and it was allocated using malloc(). The functionality of this function is very simple; it simply changes the value of the in-use byte of the pointer's metadata to false. It also tests whether the following pointer is free, and if it is, these free chunks in myblock are coalesced through coalesce(). An important note here is that in myfree(), only the pointer after the current pointer is tested for being free or not -- the previous pointer is not. Given our implementation of the metadata, there is no simple AND efficient way to find the location of the previous pointer in myfree(). We could have used a for loop to traverse through the list and find any adjacent free pointers to coalesce, but this would not be efficient. Instead, we chose to coalesce the previous pointer with the current pointer on a need basis

rather than automatically doing it. In mymalloc(), we test if the size of all subsequent, adjacent, free pointers added to the size of the current pointer added to the size of the metadata would be enough to contain the size requested by the user. If this condition is satisfied, then coalesce() is called in this location. coalesce() simply changes the current pointer's size to the sum of the size of metadata, the size of the current pointer, and the size of the next pointer(s).

All error messages are descriptive and include a line number and file name, which are automatically passed into mymalloc() and myfree().

After testing the workload of each test case the results we found are listed below. Each test was ran 100 times when the program ran, and we ran the program about 50 times to gather these averages for the runtime of each of the workloads:
- WorkLoadA: 5.1 milliseconds
- WorkLoadB: 38.8 milliseconds
- WorkLoadC: 152.1 milliseconds
- WorkLoadD: 444.2 milliseconds
- WorkLoadE: 43.1 milliseconds
- WorkLoadF: 21.7 milliseconds

Through the testing of the workloads, for WorkLoadC and WorkLoadD we noticed a lot of fluctuations in the runtimes. The range for WorkLoadC was 48 to 369 milliseconds, and the range for WorkLoadD was 170 to 768 milliseconds. This can be attributed to the fact that WorkLoadC and WorkLoadD use random numbers, hence making the timings different depending on what random integer is picked.