We created our mymalloc function by utilizing a char array of size 4096, and using structs of metadata. The first thing the mymalloc function does when it is called is check if the array is empty or not. If it is empty, then we set the struct of metadata into the first spot in the array. If the array is not empty, then we iterate through it until we find the first free location, and place the struct there. If we iterate through the entire array and are unable to find a 'free' spot in the array, then we print an error. We then set the 'size' of this struct equal to the remaining space in the array. Next, we compare this size to the size being requested. If this size is less than the size being requested, then we print an error here as well. If the size is equal to the size being requested, then it is an exact fit and all we need to do is set this struct to 'not free'. If the size of this pointer is more than the size being requested, then we have space remaining and we need to reduce the size of the struct to the size of the request. We do this by creating a new struct. This new struct will have the size of the difference between the remaining space in the array and the size being requested and set this struct to being 'free'. In our struct of metadata, a flag being set to 1 means that it is free and 0 means it is not free. Our original struct will then have a size being set equal to the request and will point to this new one as a 'next'.

Our myfree function works as follows. We first check if the pointer that it accepts as an argument is NULL. If it is, this means that we tried to free something that isn't an actual pointer, and we print an error. For other pointers that simply have not been had its space allocated by malloc, we found multiple instances of where we can identify this. One way, is by checking if the pointer is being stored in memory within the bounds of the array. If not, then we print an error. This does not work when, for example, we malloc a pointer named 'ptr' and try to free ptr + 10. Since ptr was malloced properly, ptr+10 will still be found in the bounds of our char array. This was an interesting case actually, and it why, in our free function, we have two types of meta*, one named 'mp' and another one named 'new'. Initially, I was unsure which one we should go with, so as we were testing, I kept printing out the size and flag of each type to see which one gives the metadata we need. To our surprise, we found that a valid way to handle a case like "ptr+10", is by taking the data of each type. This is represented in our if statement that checks this size of the metadata, and the flags of both struct pointer, 'mp' and 'new'. In the case of p+10, the mp flag was 0, and the new flag was 1. So if this was the case, we knew that it must be an invalid pointer to free, and we were able to properly print out an error (we used this same method to avoid freeing invalid pointers in the workload test cases C and D). Finally, we check for redundant freeing errors by checking if this pointer's flag is set to 'free' or not. If it is, then we print a redundant free error. If none of these 'error conditions' are found, then it must have been a valid input, so we then go into its metadata and set it to being 'free' and then set the size of it to 0.

Each workload is measured in microseconds, seconds are converted to microseconds and then added to microseconds.
Workload A worked perfectly fine with our implementation, throwing no errors. Workload A had the lowest average time as expected. It usually performed under 10 microseconds.
Workload B also worked perfectly fine in our implementation, throwing no errors. Workload B took slightly longer than A but still had a low average time in the 15- 25 microsecond range.

Workload C & D were the most difficult test cases to implement and fine tune. Learning to use the random function was absolutely necessary . The greatest difficulty was getting the balance right, too many free calls were randomly made and there was nothing to free. It was imperative that we guarded against too many malloc calls as well, preventing an overconsumption of data. When attempting to fix either C or D, fixing one would cripple the other somehow when running the entire test workload. Workload C and D required their own respective fixes and guards to prevent unwanted errors and the easily susceptible segmentation fault. In their finely tuned state they both run at an average time of 8 microseconds.

Workload E works as intended. This testcase require just required fine tuning to get just right. Since we are saving a 13 byte string a random number of times in a 2D array it had to be tested numerous times to find the ceiling of what our random number generator should be. Settling on a cap of 100 for amount of times to malloc “french fries”, we got this test case to run smoothly. Since the loop is not a fixed number, its average time will vary by the random number, from low times in the 50 microseconds, up to 200 microseconds.

Workload F works as intended. It was built to detect errors and to further clarify this a message is displayed right before the error signifying which error is to occur. This test helped solidify that our implementation was able to detect important errors and misuse of malloc and free. Workload F had an average time with a large range from 70-250 microseconds.