Design:

- **mymalloc():**
  - With our malloc implementation, we were able to achieve having just **2 bytes of metadata** for our array. What we did was declare a typedef struct called header which stores a single short inside. There are also two helper macros inside the header file, called GETS (obtains the size of the block) and GETA (obtains the allocation bit of the block, which will always be either 0 or 1). With these, we're are able to have just 2 bytes of metadata for each block. Malloc itself works by first checking if the array was ever initialized, and initializes it if it never was. The initialize works by setting the first two bytes to a metadata block, and the last two bytes to a metadata block. Next, we simply check if the size asked for is 0 or less then 0, returning NULL if it was. After that, we head into the main loop, which looks at the first block, obtains the size and allocation bit with GETA and GETS, and determined if it can allocate here or not. If it can, it will turn the bit to 1, and whether or not there is enough room to create a new metadata block afterwards or pad the remaining bytes is also checked. Our malloc also pads the given size so that it is always divisible by 2. This is required in order for GETS and GETA to function, as they use bitwise operations to determine the values. If there was not enough room or the block was already allocated, it will go to the next block and check that. This goes on until it either finds a block to allocate into or reaches the end and cannot allocate, in which case a saturation error is thrown and returned NULL.
- **myfree()**
  - Our free implementations begin with a few checks. These include checking if the given pointer was ever malloc'd and checking if the pointer is even valid. If any of these are true, it will print out the appropriate error and do nothing else within free. If it is a valid pointer, it will set the allocation bit to 0, then send the pointer to coalesce function.

  - **Coalesce:**
    - This function is fairly straight forward. It will check if the block in front is free and the block behind. If either of them is, it will join the block together. If both of them are free, it will join all the blocks together. Sadly, the only way I could get it to check the block before it was to start at the beginning and iterate over the array until the next metadata block is the current pointer. This leaves this step at O(n) time complexity, which could easily be O(1) if I decided to store pointers in the metadata. However, a single pointer takes up 8 bytes, which is 4 times what we have currently. Our goal was space efficiency, not speed, so O(n) is fine in this case.
- **mymalloc.h**
  - This is a simple header file which contains the necessary implementations, following the project guidelines. It has a header guard, the static array, the struct for the metadata, and all the function signatures for mymalloc.c

Workloads:

These are the results for the given workloads:

OUR MALLOC
Test A: 5.390000 Microseconds
Test B: 57.870000 Microseconds
Test C: 8.240000 Microseconds
Test D: 23.680000 Microseconds
Test E: 282.500000 Microseconds
THERE MAY OR MAY NOT BE SATURATION OF DYNAMIC MEMORY ERRORS FROM TEST F. THIS IS EXPECTED, DETAILS IN TESTPLAN.TXT.
Test F: 0.001731 Seconds

There were a few interesting things we found in the workload data. For starters, our implementation is much slower than malloc provided by the system. The systems malloc has the following results:

SYSTEM MALLOC
Test A: 6.310000 Microseconds
Test B: 7.100000 Microseconds
Test C: 5.930000 Microseconds
Test D: 7.440000 Microseconds
Test E: 3.230000 Microseconds
THERE MAY OR MAY NOT BE SATURATION OF DYNAMIC MEMORY ERRORS FROM TEST F. THIS IS EXPECTED, DETAILS IN TESTPLAN.TXT.
Test F: 0.000031 Seconds

Comparing these results, our malloc is more efficient in only Test A, which means we are faster at mallocing 1 byte then immediately freeing it. When it comes to mallocing many bytes and freeing many bytes, the system outdoes us by a lot (in term of microseconds). However, we were expecting these results as our malloc is coded for maximum space efficiency, which means that we suffer time loss in cases where doing things could be O(1) but are O(n) instead. We can see from our data that the malloc is very fast at allocating small sizes, but when it comes to larger sizes is slowed down tremendously. Even so, we are very satisfied with saving the space in the metadata that a hundred or two more microseconds.