<u>Part C:</u>

Notice that the flag parcer, PNM reader, and writer are done for us in the starter code (for the flag parser) and pnm.h (for the reader and writer). This means the only component that we have to worry about is the Image mapper and how the operation is passed to the Image Mapper.

Note that the PNM Reader will be in main using pnm.h, which will open the file and close the file before moving on so there are no memory leaks when exiting failure.

Image mapper will take the Pnm_ppm which represents the image to transform, the A2Methods_mapfun, which allows us to use a mapping function, and a void function which performs the transformation. The void function is NOT an A2Methods_applyfun, but is a function that is in the format:

```
void (*transform)(Pnm_ppm image,
                  A2Methods_mapfun mappingFunction);
```

This allows us to hide the apply functions and mapping functions within the transform function, so we can easily change the implementation. Each of the operations that can be performed will be in the format of the transform function, each of which will have a hidden apply function helper to help set the data.

Timing will be handled in the ImageMapper function, (which is the function that runs transform), as it can just put the timer before and stop after the transformation. The function contract for the image mapper is:

```
void imageMapper(Pnm_ppm image,
                 A2Methods_mapfun mappingFunction,
                 void (*transform)(Pnm_ppm image,
                    A2Methods_mapfun mappingFunction),
                 FILE *timerFile);
```

The function will basically just start the timer, run the transform function, then stop the timer and write to the file.

The function Rotate0 will take the Pnm_ppm image, A2Methods_mapfun mappingFunction. However, since there is no transformation, it will just return immediately (voiding the unused parameters).

The function Rotate90 will take in the Pnm_ppm, A2Methods_mapfun mappingFunction. To facilitate this rotation, the Pnm_ppm image A2Methods_T is used to generate an instance of A2_UArray2. The new A2_UArray2 newImage will have the dimensions of the UArray2 that is rotated, so that the rotated UArray2 can be directly put into newImage. These dimensions are to swap width and height. Note that the position in the new array can be calculated to be (x, y) -> (h - y - 1, x), so we know the exact location in the newImage that each pixel object should go to. Our apply function will read each element in the A2_UArray2 in the image, and use the formula to set the value in the newImage of the corresponding location. We will use the A2_get function to place into newImage. Finally, since Pnm_ppm is a struct, we can overwrite what the A2_UArray2 is with our own, and free the old A2_UArray2 up, and fix the other values in the Pnm_ppm to match the newImage.

The function Rotate180 will also take in the Pnm_ppm, A2Methods_mapfun mappingFunction. Similarly to Rotate90, the Pnm_ppm image A2Methods_T will be used to produce an A2_UArray2 instance. With this, we will use an apply function to map each element within the A2_UArray2. The way we would go about rotating the image would be to use the formula (x, y) -> (h - 1 - y, h - x) in order to determine where the new location of the rotated180 will be. After, we will use our A2_get function to place it into a newImage.

Implementation Plan:

1. Open the file, read using pnm.h, and close the file (15 min)
    a. Try reading in from stdin, a regular pnm, an empty file, a file that is not a pnm, and unexpected end in stdin, try different color channels (60 min)
2. Create ImageMapper function that handles timing and calls the transform function (along with opening the timing file if it exists and passing null) (60 minutes)
    a. Create a dummy test transform function that just prints out if everything given is not null, check if the timings can be printed into the file, give it no timing file, and try linking a test a2plain.h which

has weird mapping functions and seeing if it can handle it. (90 min)

3. Create the rotate 0 degrees function that simply keeps the array in the original form that was passed in (15 min)
    a. Create a test which rotates an image 0 degrees and ensure that the image rotation is not altered (15 minutes)
4. Create the rotate 90 degrees function which will create a new A2_Array2 and store the elements into it according to the rotation (120 minutes)
    a. Create tests that will rotate 90 degrees, test on images that have horizontal/vertical bars, a 2D array which just contains all the same numbers (60 min)
5. Create the rotate 180 degrees transformation function, which will create a new A2_UArray2 and copy into it (120 min)
    a. Call the function on test images listed above in the reader section, test on UArrays that have shapes (by overwriting some of the pnm.h images returned) such as a long horizontal and long vertical bar, square images, and images that are rectangular (60 min)
6. Pnm write to stdout using pnm.h (60 min)
    a. Try directly writing an image from pnm reading, writing an empty file, and writing from all of our transformations. (90 min)
7. (Optional) Write rotate 270 degrees, translation, reflection, and transposition (240 min)
    a. Testing image files listed above
8. (Optional) Change 180 degrees to in-place
    a. Same as 180 degree testing (120 min)

Part D:

	Row Major	Column Major	Block Major
90 Degree	4	5	3
180 Degree	1	6	2

Assuming that the UArray2 implementation makes the rows contiguous in the array (aka the rows are all adjacent in a single UArray). We also assume that the blocked version uses a similar UArray2 as above, which holds UArrays (which are each individual block). This means that each block is unboxed in the UArray, and we assume that block major uses row major in the UArray2 that it is in, as that would put the next blocks used adjacent in memory. We also assume that our mapping function has two parts: Take the A2 and map it into a temporary 2D array, then take the temporary 2D array and make a new A2 with the data.

Note that the Column major operation is horrible for cache hits, as in the operation above as it will skip through the rows each time, making it so that each of the requests are not close together in continuous memory. This means that the requests will probably not be in the same block. With width larger than the cache block size, each of the requests will be in its own line. This also means that if height is larger than the cache line count, then the cache will evict the first row that the column major operation would need in the next column, making it the worst case scenario where the requests where almost all of the reads/writes would have to go back to main memory at every step except the last few (which would leave dirty lines in the cache) making it worse than just having to go back to main memory directly. In terms of spatial, column performs really poorly because the data is not contiguous in memory, causing it to be stored into spaces that are not close by. The entire point of the spatial aspect is to be able to go to elements in contiguous memory, however, column does not allow us to do that because we traverse through the elements in a column fashion. Similarly, the column major operation does not perform well. Given that since we traverse in a column major order then we will essentially be getting a ton of misses, and as such it is very costly to keep adding to the cache.

Row major is a much better operation in comparison to the column major because of its ability to store memory contiguously. Each element is essentially subsequent to the previous stored one, giving us the ability to access elements in order of where they are stored in memory. Notice that the line size doesn’t really affect how good the caching is because everything in the line will be read, and since the next block will also contain information that we need, we have done everything with the previous block that we will need, and will not need it in the near future until the next read/write cycle. Then, the next block will be adjacent to the previous block as everything is adjacent in memory, so it will also access everything in that block before moving on to the next without having to go back to the block in the near future. Looking into the spatial aspect of row major, it performs really well because each data is stored contiguously into memory. Furthermore, temporal is also very effective because we will not reach a high level of hits since we know that the data we are accessing are close in memory.

The block major in our opinion would be the 2nd best operation after row major because we are able to maximize space which means we are then able to store as much data into the cache, which translates to much lower hits. This is because in memory, it can be seen as similar to going row major. All elements in the block are next to each other, and the next block will also be adjacent in the memory. This means that the block major will hit as much as possible when doing the mapping function, similar to how row major works. However, it is slightly worse than just doing regular row major, as there is extra padding for the UArray, which makes there be more wasted memory in the lines. However, the block major is close enough to the row major for this to be more on the minor side. Spatial can be regarded as effective for block major because data is stored contiguously. When it comes to temporal, this operation performs as well because memory is near each other, and as such there won't be many misses.

1) 180 degree Row Major
    a) This will perform row major read and row major write when reading from the image, and writing the data to a new image. This is the highest hit rate because it uses the highest hit rate operations both times.
2) 180 degree Block Major
    a) This will perform block major reads and writes, being similar to how the row major does it, although being slightly worse especially because of the padding.
3) 90 degree Block Major
    a) This will perform worse during the reading stage, as when we save the data, it will have to save the blocks in a column fashion (as it reads in a row-wise fashion) before it can write with a fast block-wise fashion. This is because when we read the file, we put the data in a separate 2D array, and writing into our separate 2D array is where the column-wise (aka lots of misses) operations come from. However, this is still faster than a 90 degree row major. This is because only the blocks are read in a column-wise fashion; each block is still contiguous in memory, and can have many cache hits, even if the blocks themselves don't have good cache hitting. Note that the writing from our 2D array back is still fast though.
4) 90 degree Row Major
    a) This will have a higher hit rate reading via row major, but there will be a lower hit rate when writing to the temporary array, as the write locations will probably not be in the same blocks because it is written in a column-wise fashion. Although it will probably be slower than column major (as writes are slower), the row major has a slight advantage in the fact that read comes before write, so it is less likely to evict the read line. However, when copying back into the new A2 array, it will be using row major, which will be fast.
5) 90 degree Column Major
    a) This will have a very little hit rate read via column major, but then there will be a higher hit rate when

writing into the temporary array as the areas will be in rows, which will be adjacent in memory because rows are adjacent to each other in the normal UArray2. However, copying into the new A2 will have many misses, as it will be reading in column major fashion.

6) 180 degree Column Major
    a) Column-wise writing in both reading and writing, so it will pretty much miss everything all of the time (Just the worst).