Part C

Problem Statement:
We want to find the best implementation for transforming images. We want to use the cache to its full effect.

Use Cases:
- Rotating an image by 0, 90, 180, or (if implemented) 270 degrees
- (if implemented) Mirror an image vertically or horizontally
- (if implemented) Transpose an image across the upper left axis to the lower right axis
- Create timing data for the image transformation process and output it to a file
- For each of the transformation options:
    - The image can be rotated by mapping through the image in a row-major fashion, column-major fashion, or block major fashion

Assumptions and Constraints:
- We will receive a valid ppm image as input either via standard input or through a file using pnm reader.
- Will output a transformed/rotated/mirrored/transposed copy of the original image in a ppm format.
- We can't give the image to chatGPT to rotate it. - Rigo 2k25
- We don't need to do any explicit error handling for incorrect file inputs.

Implementation Plan:
- Part C will begin implementation once we have our blocked array and a2plain subclass.
- **Handle processing the imputed ppm image and storing into uarray/array2b the A2methods interface and the pnm.h file.**
    - Use asserts to test and see if there were any errors in inserting the data for the image into the 2d array.
- **We will make a print function that prints the output of our function to standard output (or a given filestream) using the Pnm_ppmwrite function.**
    - We will test this out first with a small ppm image
    - This function will be used to test our other functions.
- Create the rotate 0 function and have it print "Hello World" to make sure that the query in main calls the right function.
- **Replace the contents of rotate0 with a function that just outputs the original image to std cout.**
    - Test with the display command and images of various sizes.
    - Valgrind test for memory leaks
- **Implement the rotate 90 function**
    - Test with images of various sizes and use the display command
    - Valgrind test for memory leaks

- **Implement the rotate 180 function**
    - Test with images of various sizes and use the display command
    - Valgrind test for memory leaks
- **If there is time:**
    - Implement the rotate 270 function
        - Test with images of various sizes and use the display command
        - Valgrind test for memory leaks
    - Implement the mirror horizontal function
        - Test with images of various sizes and use the display command
        - Valgrind test for memory leaks
    - Implement the mirror vertical function
        - Test with images of various sizes and use the display command
        - Valgrind test for memory leaks
    - Implement the transpose function
        - Test with images of various sizes and use the display command
        - Valgrind test for memory leaks

Architecture:

- **We will have either 2 uarray2s or 2 uarray2bs**
    - One will be the 2d array that holds the original image and the other will be a 2nd 2d array that holds the image after transformation
    - if the command line arguments say the image will be transposed using block mapping, then the image contents will be held in a array2b
- Once the command line input is processed using SET_METHODS (or the default options are used if a command input isn't specified) in the main function,
- **Methods will be accessed through A2Methods_T interface class**
    - To handle mapping with row or column major use functions originating in a2plain
        - Note: The default mapping for a2plain will be row major
    - To handle mapping with block major use functions originating in a2blocked
    - Methods from A2Methods_T:
        - Functions needed to create the destination 2d array
            - new, width, height, size
            - new_with_blocksize, blocksize will be needed if the original 2d array was a uarray2b
            - None of these have void pointers
        - free will be used to free the uarray2 or uarray2b
            - Does not have void pointer
        - Mapping functions:
            - map_row_major, map_col_major, small_map_row_major, and small_map_col_major will be used with uarray2
            - map_block_major and small_map_block_major will be used with uarray2b

        - For all the mapping functions, the void *cl will be a pointer to the uarray2 or uarray2b holding the contents of the destination image
        - Note: due to the A2Methods_T interface class, the apply function in the original mapping functions that would otherwise be void pointers are now of type A2Methods_applyfun or A2Methods_smallapplyfun
- **Methods from pnm.h:**
    - Pnm_ppmread, Pnm_ppmwrite, and Pnm_ppmfree
        - None of these have void pointers
- **Invariants**
    - Invariants for image rotation (taken from spec):
        - 90 degree rotation: If you have an original image of size w × h, then when the image is rotated 90 degrees, pixel (i, j) in the original becomes pixel (h − j − 1, i) in the rotated image.
        - 180 degree rotation: When the image is rotated 180 degrees, pixel (i, j) becomes pixel (w − i − 1, h − j − 1)
    - All invariants for 2d arrays (uarray2 and uarray2b) are handled in their respective data structure implementations.

Part D

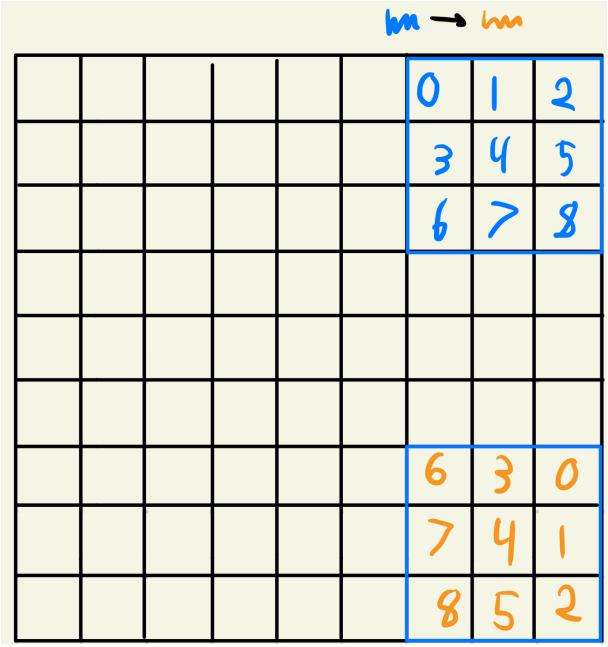
Our Uarray2 implementation has entries in a row next to each other in memory and entries in a column are a width of the 2d array apart in memory.

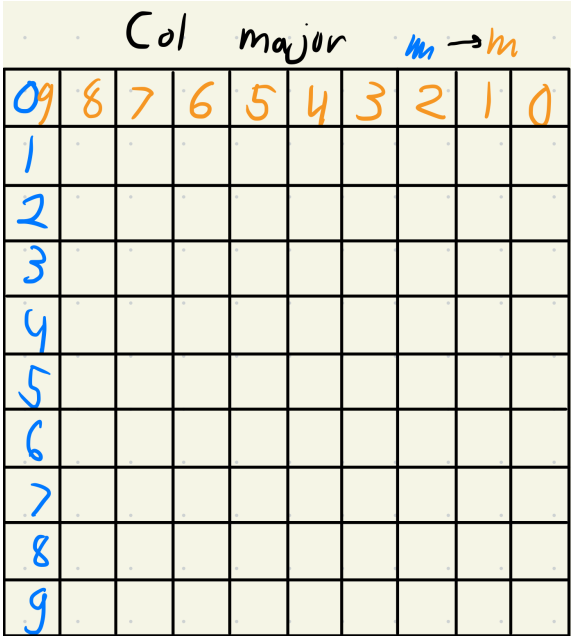
1. **Row 180**
    a. As long as you have a at minimum 2 line cache, you can store both the row from the array that you are reading to and the row in the array you are writing to. This is because the entries in a row are adjacent to each other in memory. (If you can't store the whole line, you will still have the parts that of the row that you are writing to).
2. **Block 90/180**
    a. Blocked 90/180 degree rotation is the second best because you get a decent amount of cache hits. Rotations should not be written into more than 4 blocks at a time, because it's impossible for data stored in one block to partially occupy more than 4 blocks of equal size when stored in the same shape. When the 2d array inside the blocks doesn't perfectly fit, since the blocks are always squares, if the 2d array is offset by only the width or the height, you need to use the 2 adjacent blocks to handle putting data into the new 2d array when translating.

When the 2d array is offset by both width and height, to rotate you need to use 4 adjacent blocks to handle putting data into the new 2d array.

3. **Row/Col 90**
    a. Row major/Column major 90-degree rotation is the third best because our implementation of Uarray2 has elements in the same row closer to each other, whereas elements in the same column are further from each other (elements in a column are the width of the 2d array apart). The 90-degree row-major rotation reads from a row in the original image and writes to a column in the translated image, giving good locality when reading and bad locality when writing. The 90-degree column-major rotation reads from a column in the original image and writes to a row in the transformed image, giving bad locality when reading and good locality when writing. Since both are reading/writing from a row, and writing/reading to a column, the cache hits are the same.
4. **Col 180**
    a. Column major 180-degree rotation is the worst because elements in a single column are further apart. Each element in a column is a distance of the width of the 2d array apart. Since a 180-degree rotation repeatedly accesses elements by column, the cache isn't being used to our advantage due to poor spatial locality.

90 degree rotation





180 degree rotation

Blocked

0	1	2	3						
4	5	6	7						
8	9	10	11						
12	13	14	15						
						15	14	13	12
						11	10	9	8
						7	6	5	4
						3	2	1	0

Row

0	1	2	3	4	5	6	7	8	9
9	8	7	6	5	4	3	2	1	0

Col

0									9
1									8
2									7
3									6
4									5
5									4
6									3
7									2
8									1
9									0