Owners: lyuan04, cyamam01

Implementation plan (implementation plan for part C starts at step 8 - onwards):

1. Create a uarray2b.c file and create some boilerplate for each function, then try to 'make' it. Figure out the error messages from 'make' and work through them to make sure the files are linked together properly (15 minutes)
2. Implement each function one-by-one in uarray2b.c and see if it compiles (15 minutes)
3. Test each function against our own inserted lines in a2test.c to see if you can actually create, access, and read through uarray2b and ensure that it works as intended (45 minutes)
4. Run a2test.c through valgrind and ensure there are no leaks (and if there are leaks, fix them) (45 minutes)
5. Finish implementing all of the functions in a2plain.c, double checking that the exported functions are matches the ones in a2blocked.c (except for the NULL we export for the blocked functions) (15 mins)
6. Test the functions implemented in a2plain.c (again, by using a2test.c with our own lines of code modifying the A2Methods_T, seeing if all of them work as planned) (30 mins)
7. Ensure that ppmtrans.c can *link* with the above-created files (using the specific linker commands in the spec) (10 mins)
8. Allow ppmtrans to take in a last argument specifying the filename (check if the file can open properly, write to stderr if filename provided, but can't be opened) (10 mins)
9. Check if a -row/-col/-block major tag was provided, then store that method (or the default one) (10 mins)
10. Create rotate_0, rotate_90, rotate_180 functions, fill them with print statements like "you're now in rotate_0 function using row major!" and call them using the terminal to see if we are linking together our main() function and rotate() functions. (10 mins)
11. Create a function readFile(). In it, use the Pnm_ppmread() function (from pnm.h) to read through the open file and store each pixel in a pnm_ppm struct (this will be the main data structure that will hold our pixels, each element is a Pnm_rgb struct). Check if you can pass different methods from A2Methods_T to Pnm_ppmread() correctly.Try printing smaller images to console to see if this works (15 mins).
12. Make a helper function getNewCoordinates() that takes in a pixel's coordinates and specific rotation, then returns the new coordinates it should end up in. (10 mins)
13. Implement the rotate_0() function: takes in a Pnm_ppm struct and prints its 0^{o} rotation (by whatever row/col/block major specified) to standard output (using pnm.h). Use only the A2Methods interface (of mapping by row/column/block major). Test if it works by printing to console and seeing if it looks like the input image (30 minutes)
14. Implement the rotate_90() function: It will take in a Pnm_ppm struct and print its 90 degree rotation. It will read through every pixel in the Pnm_ppm struct, write it to a new temporary data structure (of type A2Methods_UArray2) in a new location (determined by our helper function getNewCoordinates(), and then will use Pnm_ppmwrite() to write the new A2Methods_UArray2 to standard output. Test it on an easily recognizable input image (like a 'blank' image, with just a box of 1s in the corner and you can see if it moved to the right corner) (30 minutes)
15. Implement the rotate_180() function: almost the exact implementation as above, but with a different rotation passed to our helper function getNewCoordinates(). Again, only use A2Methods and test it on easily recognizable input images. (30 minutes)

16. Test above functions on files in /comp/40/bin/images/ and use “display -” to see if we printed them correctly (30 minutes)
17. Implement the *timing* section (start a time at the start of the program, end it at the end of the program, calculate the average for pixel operations). See if our predictions are right. (30 minutes)
18. Go thoroughly through each and every function and *ensure* that *nowhere in this program will a pointer be dereferenced without first checking if it is not null*. Also check that all coordinates and sizes, widths, and heights are valid (>= 1). (10 minutes)
19. Free the pnm and the UArray2b. Ensure program passes valgrind (30 minutes)

Part D predictions

	row major	col major	block major
90 degree	4	4	2
180 degree	1	5	3

I implemented my UArray2 by using a single UArray underneath, where the rows are stored contiguously, back-to-back. Because of this structure, reading through rows will be very efficient, as values in the same row are stored next to each other (capitalizing on spatial locality). However, reading through columns will be much less efficient because the new locations are distant enough in RAM such that they will not be cached together.

Thus, implementing a 180 degree rotation using row major will be very efficient. This is because flipping an image in a 180 degree rotation will require reading through rows in both the reading and writing of the arrays. Because of this structure, addresses of each row will be stored in the cache, thus being much quicker to access. This will be *the most efficient* because 180-degree rotations require the most amount of row-reading.

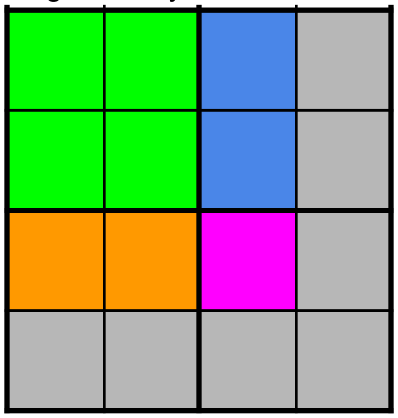
Implementing a 180 degree rotation using column major will be very inefficient. This is because reading through an image by column will *not* capitalize on spatial locality in the way that we implemented it in our UArray2. *Each* access of a new column will be *too far* away to be stored in the cache. This will be the *least efficient* because 180-degree rotations require the least amount of row-reading.

Implementing a 90-degree rotation requires reading and writing from a row to a column or vice versa. Thus, this situation does not benefit one implementation better than the other. In row major, you read by rows (fast) and write to columns (slow). In column major, you read by columns (slow) and write to rows (fast). Thus, they should be roughly equal in performance and hit rate (and in between the performance of 180 degree row major and col major).

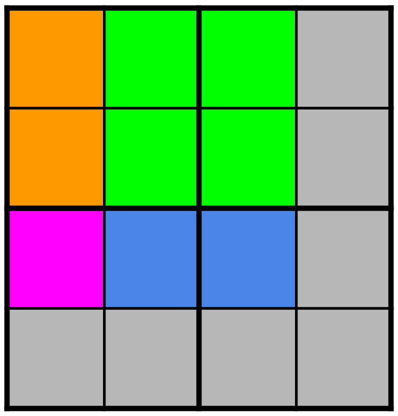
Both block major 90 degree and 180 degree will be faster than the 90-degree row-major and col-major because the *blocking* can *capitalize* on spatial locality. When you are reading (and writing) from any element in a block, the whole block is stored in the cache. Thus, when you need to read from it again *regardless of whether you need the next row or column*, the elements may already be stored in the cache (as the block can store other, nearby elements).

But between block major 90 degree and 180 degree, the block major 90 degree will be *faster* because of how our data structure behaves at the *edges* (the blocks that are not completely filled with elements). When you read from one block that gets written to elements at the edge, there exists a chance where the edge elements must be distributed over different blocks.

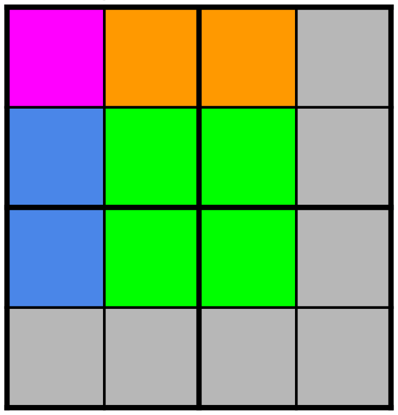
Original array



90 degree rotation

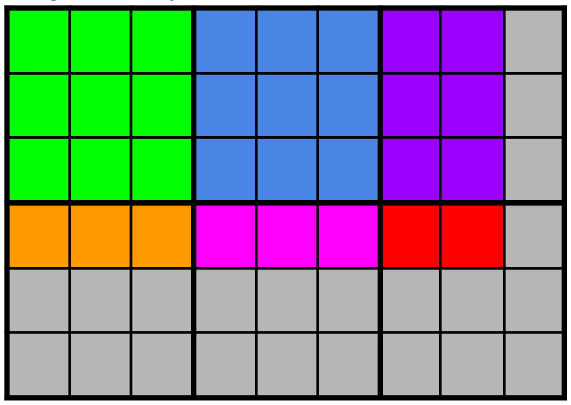


180 degree rotation

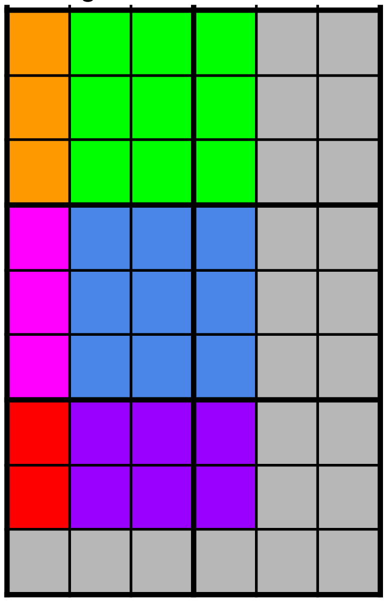


As you can see, when you rotate an image 180 degrees, there exists a chance when an initial block gets read but written to four blocks. This chance does not exist in the 90 degree rotation, as in the 90 degree rotation, each block at most gets 'split' between two blocks when writing.

Original array



90 degree rotation



180 degree rotation