Project 2: CS61Classify

Part A Deadline: Monday, October 4, 11:59:59 PM PT

Part B Deadline: Monday, October 11, 11:59:59 PM PT

In this project, you will write RISC-V assembly code to classify handwritten digits with a simple machine learning algorithm.

The goal of this project is to familiarize you with RISC-V, specifically calling convention, calling functions, using the heap, interacting with files, and writing some tests.

Setup: Git

1. Visit Galloc. Log in and start the Project 2 assignment. This will create a GitHub repository for your work.
2. Clone the repository on your workspace. We recommend using your local machine. Windows users should clone outside WSL (Git Bash is recommended).

```
$ git clone https://github.com/61c-student/fa21-proj2-username.git proj2-cs61classify
```

(replace `username` with your GitHub username)

3. Navigate to your repository:

```
$ cd proj2-cs61classify
```

4. Add the starter repository as a remote:

```
$ git remote add starter https://github.com/61c-teach/fa21-proj2-starter.git
```

Setup: Java, Python, and Make

Make sure that you followed the local computer setup directions in Exercise 4 of Lab 0.

Windows users should use WSL for running the Make commands. You can `cd` to the repo you cloned in Windows (e.g. `C:/Users/oski/cs61c/proj2/` in Windows is `/mnt/c/Users/oski/cs61c/proj2/` in WSL), so that you're running Venus and Make in the same repo.

Setup: Venus

We will use the Venus web interface for debugging. The procedure is very similar to the Venus setup in Lab 3, so we highly recommend doing Lab 3 first. As a recap:

1. Navigate to your `proj2-cs61classify` directory and run this command. Windows users should run outside WSL (Git Bash is recommended).

   ```
   $ java -jar tools/venus.jar . -dm
   ```

2. Open https://venus.cs61c.org in your web browser (Chrome or Firefox is recommended). In the Venus web terminal, run `mount local proj2`. In your browser, you may see a prompt saying `Key has been shown in the Venus mount server! Please copy and paste it into here`. You should be able to see a key (string of letters and numbers) in the most recent line of your local terminal output; just copy and paste it into the dialog.
3. Now the project files from your local computer are loaded into the Venus web interface. You can open the Files tab or run `ls proj2` in the Venus terminal to see all your files in your browser.

If you're having trouble with this setup, refer to this video walkthrough.

If you can see your files in Venus, you can skip the rest of this section. If the above steps didn't work, you can follow the guide below to manually upload files.

▼ Manually Uploading Files

1. On your local computer or on the hive, zip your `src`, `inputs`, and `unittests/assembly` directories.
2. Navigate to https://venus.cs61c.org Type `upload` in the Venus terminal.
3. A file upload prompt should appear. Upload the zip file you created in step 1.

Now you can edit and debug your files on Venus.

To download any files you've edited, you can run `zip zipname file1 file2 file3` in the Venus terminal. Replace `zipname` with whatever you want the downloaded zip file to be named. Replace `file1 file2 file3` with a list of files or directories you want to download (there can be more or less than 3 files or directories).

Dark Mode

Part A: Math Functions

In this part, you will implement a few math operations that will be used for classification later.

Task 1: Absolute Value (Walkthrough)

To familiarize you with the workflow of this project, we will walk you through this task. This section is available in both video and written form.

To follow along with the videos, watch the videos in this playlist.

Running Tests

In this project, tests are written in Python and compiled into RISC-V assembly.

To see the Python source for the tests, navigate to the `unittests` directory where the tests are located. Inside the `unittests` directory, `unittests.py` is where we've provided tests for your functions.

Take a look at the contents of `unittests.py`. Although the tests are written for you in Tasks 1-4, it helps to be familiar with the unit testing framework to understand what the tests are doing.

To run the tests, start by running `make help` in the `unittests` directory on your local machine. This gives you an overview of the commands you can run for testing. In particular, `make test-partA` compiles and runs all the tests for Part A. You can also provide the name of a specific function to compile and run all the tests for that particular function. For this task, since we are implementing the `abs` function, you can run `make test-abs` on your local machine.

Since we haven't implemented the `abs` function yet, all of the tests are failing.

Using VDB to debug tests via Venus

First, open up Venus in your web browser and mount your files. (Refer back to the setup section of the spec if you're having trouble.)

You can edit files in a text editor or directly in Venus. To edit files in Venus, switch to the Files tab. Here you can open and edit assembly files. Remember to save your files frequently with `control+S` (Windows) or `command+S` (Mac). Venus does not auto-save as you work.

Open `src/abs.s`. This is the function we need to implement in this task. At the moment, the function just returns 0.

To see why the `abs` function is failing right now, we can set a breakpoint. Type `ebreak` at Line 13. This places a breakpoint just before the `mv a0, zero` instruction.

To start the debugger, go to Venus and navigate to `unittests/assembly/TestAbs_test_one.s`. Note that the `assembly` folder will only be made after you run the `make` command to generate the assembly test files.

Click on VDB to start the debugger. When we click Run, the debugger will pause at the breakpoint we set. While paused, you can inspect the registers and memory. In particular, notice that register `a0` contains the argument `1` here, because this test inputs the argument 1.

You can also step through code line-by-line in the debugger. Click Step to execute the next instruction, `mv a0, zero`. This causes the register `a0` to take the value `0`.

Now we can implement the `abs` function by putting this in `src/abs.s`:

```
abs:
        # Return if non-negative
        bge a0, zero, done

        # Negate a0 if negative
        sub a0, x0, a0

done:
        ret
```

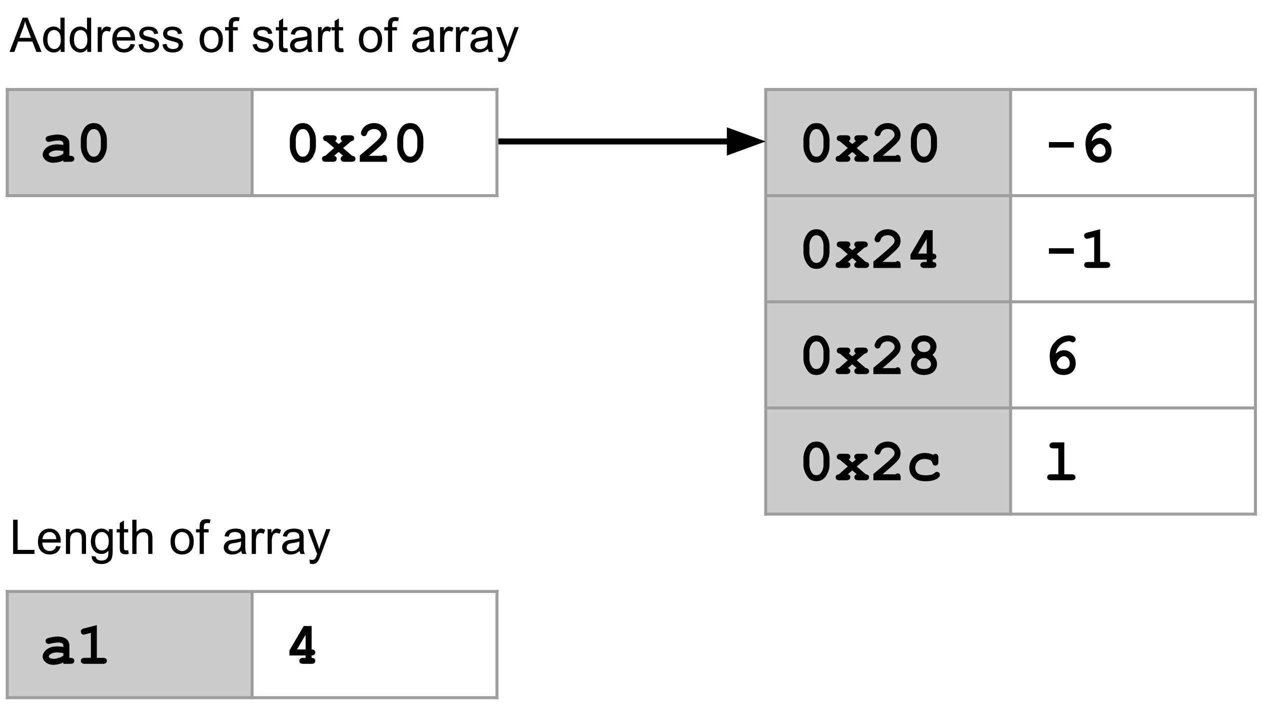
Try running `make test-abs` again, and now you should see that all the tests pass.

Task 2: ReLU

Conceptual Overview: Arrays

In this project, we will be working with integer arrays. Recall that the integers in an integer array are stored in a consecutive block of memory.

To pass an integer array as an argument, we will pass a pointer to the start of the integer array, and the number of elements in the array.

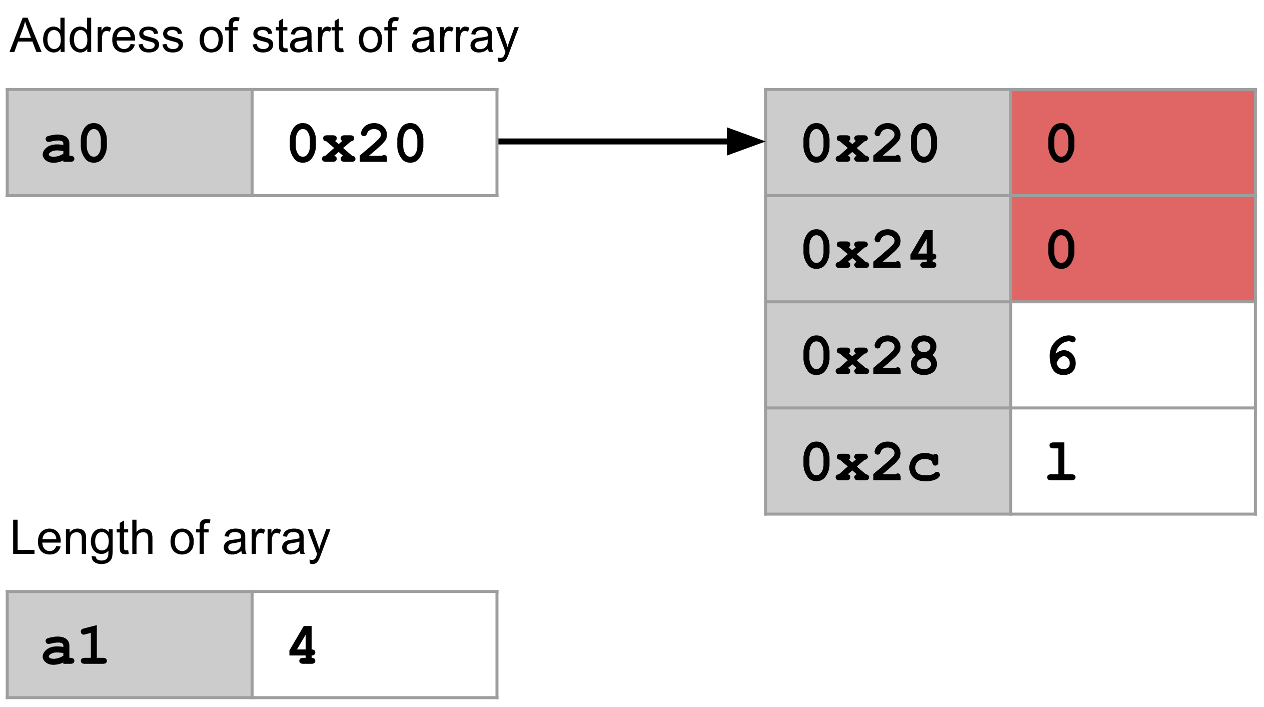


In this diagram, register `a0` stores the first argument (the address of the start of the array). Register `a1` stores the second argument (the number of integers in the array).

Conceptual Overview: ReLU

The ReLU function takes in an integer array and sets every negative value in the array to 0. Positive values in the array are unchanged. In other words, for each element `x` in the array, ReLU computes `max(x, 0)`.

ReLU should modify the array in place. For example, if the above integer array is passed into ReLU, the result would be stored in the same place in memory:



Note that the negative values in the array were set to 0 in memory.

Your Task

Fill in the `relu` function in `src/relu.s`.

`relu`: Task 2.			
Arguments	`a0`	`int *`	A pointer to the start of the integer array.
	`a1`	`int`	The number of integers in the array. You can assume that this argument matches the actual length of the integer array.
Return values	None		

If the input is malformed in the following ways, put the appropriate return code into `a1` and run `call exit2` to quit the program. (For example, if the length of the array is less than 1, run `li a1 57` and `call exit2`.)

Return code	**Exception**

57	The length of the array is less than 1.

To test your function, run `make test-relu`. Refer back to Task 1 for debugging instructions.

Task 3: Argmax

Conceptual Overview: Argmax

The argmax function takes in an integer array and returns the *index* of the largest element in the array. If multiple elements are tied as the largest element, return the smallest index.

For example, if the integer array `[-6, -1, 6, 1]` is passed into the argmax function, the output should be 2, because the largest integer (6) is located at index 2 in the array. If the integer array were instead `[6, 1, 6, 1]`, then the output should be 0, because the largest integer (6) is first found at index 0.

Your Task

Fill in the `argmax` function in `src/argmax.s`.

`argmax`: Task 3.			
Arguments	`a0`	`int *`	A pointer to the start of the integer array.
	`a1`	`int`	The number of integers in the array. You can assume that this argument matches the actual length of the integer array.
Return values	`a0`	`int`	The index of the largest element. If the largest element appears multiple times, return the smallest index.

If the input is malformed in the following ways, put the appropriate return code into `a1` and run `call exit2` to quit the program.

Return code	**Exception**
57	The length of the array is less than 1.

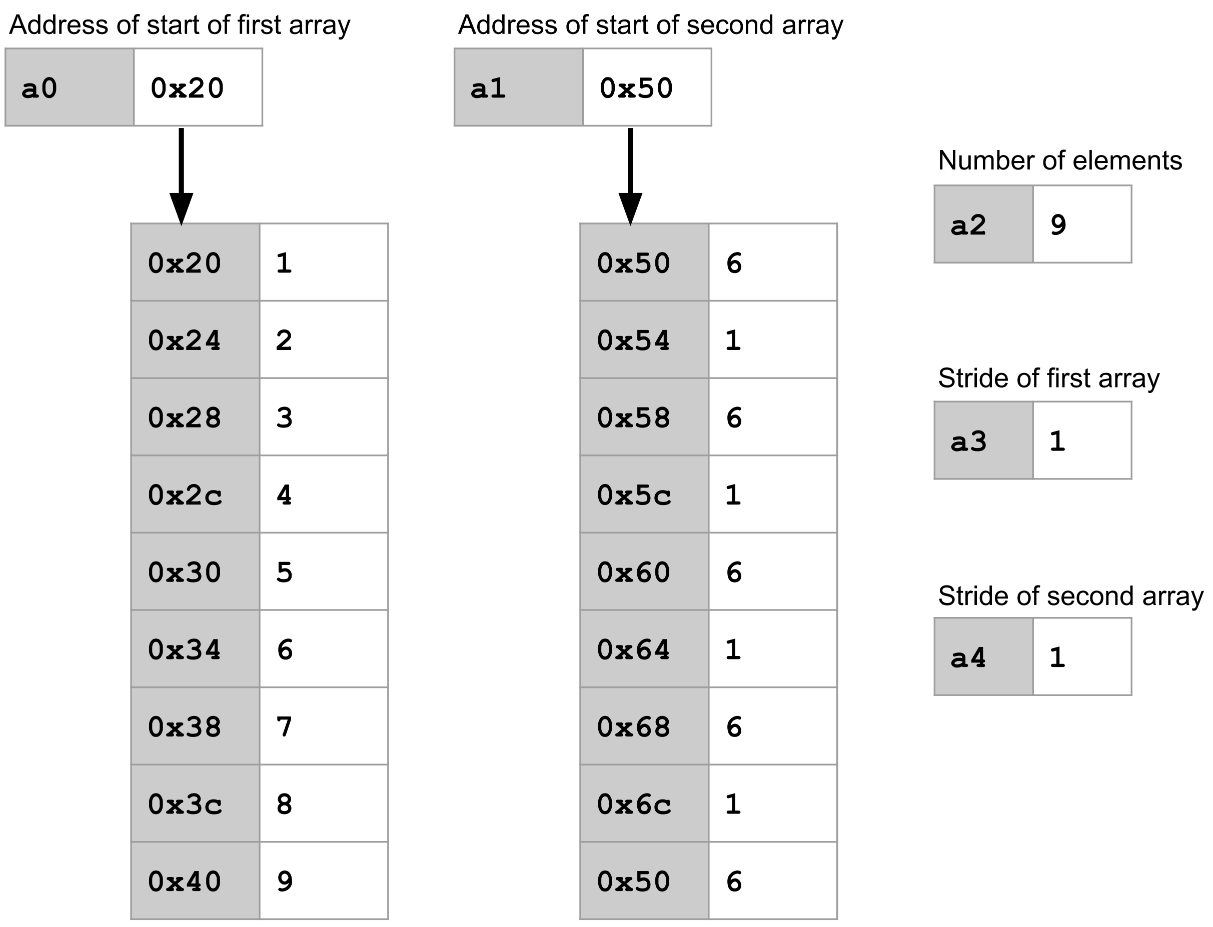
To test your function, run `make test-argmax`. Refer back to Task 1 for debugging instructions.

Task 4: Dot Product

Conceptual Overview: Dot Product

The dot product function takes in two integer arrays, multiplies the corresponding entries of the arrays together, and returns the sum of all the products.

For example, if these two integer arrays were passed into the dot product function, the function would return `(1*6) + (2*1) + (3*6) + (4*1) + (5*6) + (6*1) + (7*6) + (8*1) + (9*6) = 170`.

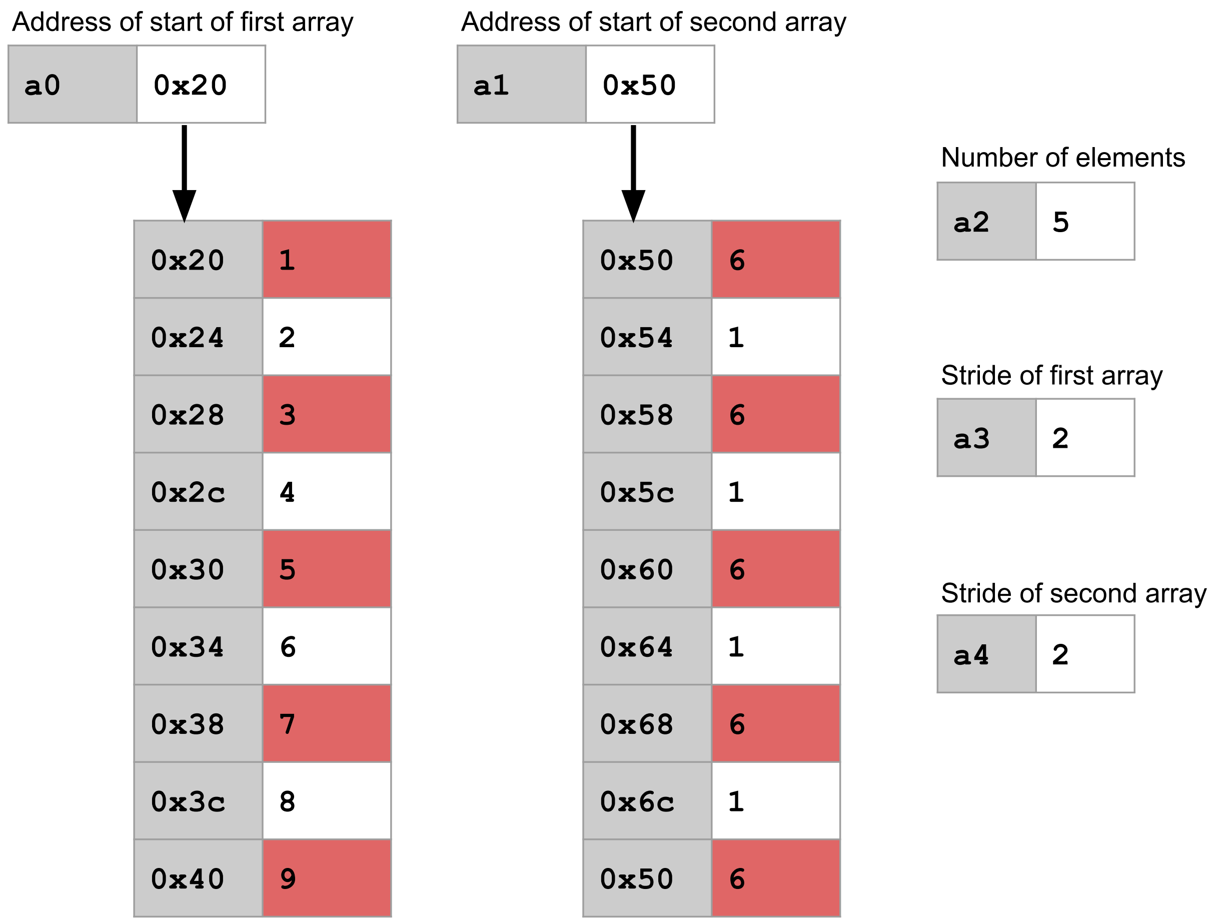


Conceptual Overview: Array Strides

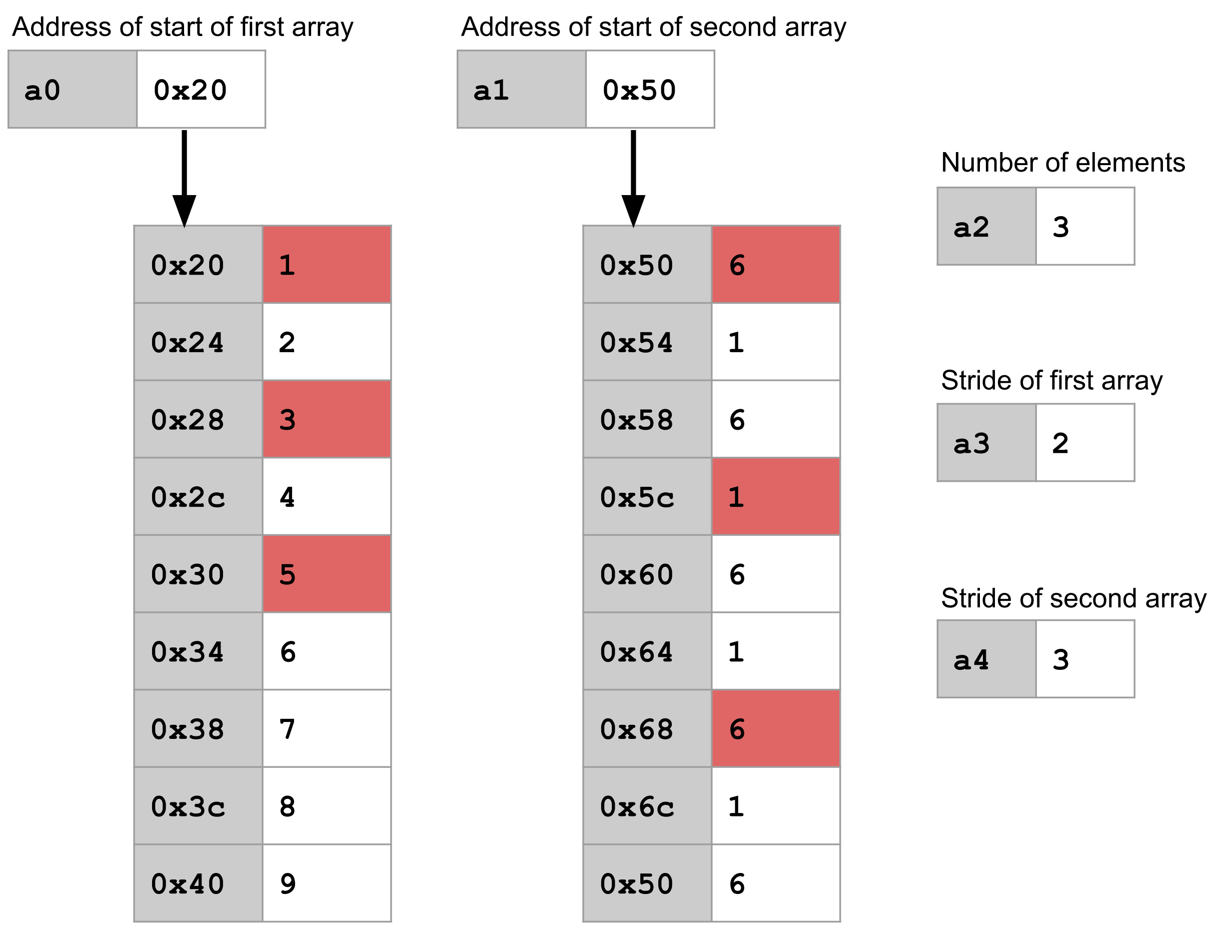
Instead of iterating through every element of the array, what if we want to iterate through every other element, or every third element? To do this, we will define the stride of an array.

To iterate through an array with stride *n*, start at the beginning of the array and only consider every *n*th element, skipping the elements in between.

Note that the stride is given in number of elements, not number of bytes. This means that iterating with stride 1 is equivalent to iterating through every element of the array.



For example, in the above diagram, both arrays are using stride 2, so we skip every other element in the array. 5 elements should be considered, so we stop after multiplying 5 pairs of elements together. The function would return `(1*6) + (3*6) + (5*6) + (7*6) + (9*6) = 150`.



In the above diagram, the first array is using stride 2, so we skip every other element in this array. The second array is using stride 3, so we use every third element in this array. 3 elements should be considered, so we stop after multiplying 3 pairs of elements together. The function would return `(1*6) + (3*1) + (5*6) = 39`.

Your Task

Fill in the `dot` function in `src/dot.s`.

`dot`: Task 4.			
Arguments	`a0`	`int *`	A pointer to the start of the first array.
	`a1`	`int *`	A pointer to the start of the second array.
	`a2`	`int`	The number of elements to use in the calculation.
	`a3`	`int`	The stride of the first array.
	`a4`	`int`	The stride of the second array.

Return values	`a0`	`int`	The dot product of the two arrays, using the given number of elements and the given strides.

If the input is malformed in the following ways, put the appropriate return code into `a1` and run `call exit2` to quit the program.

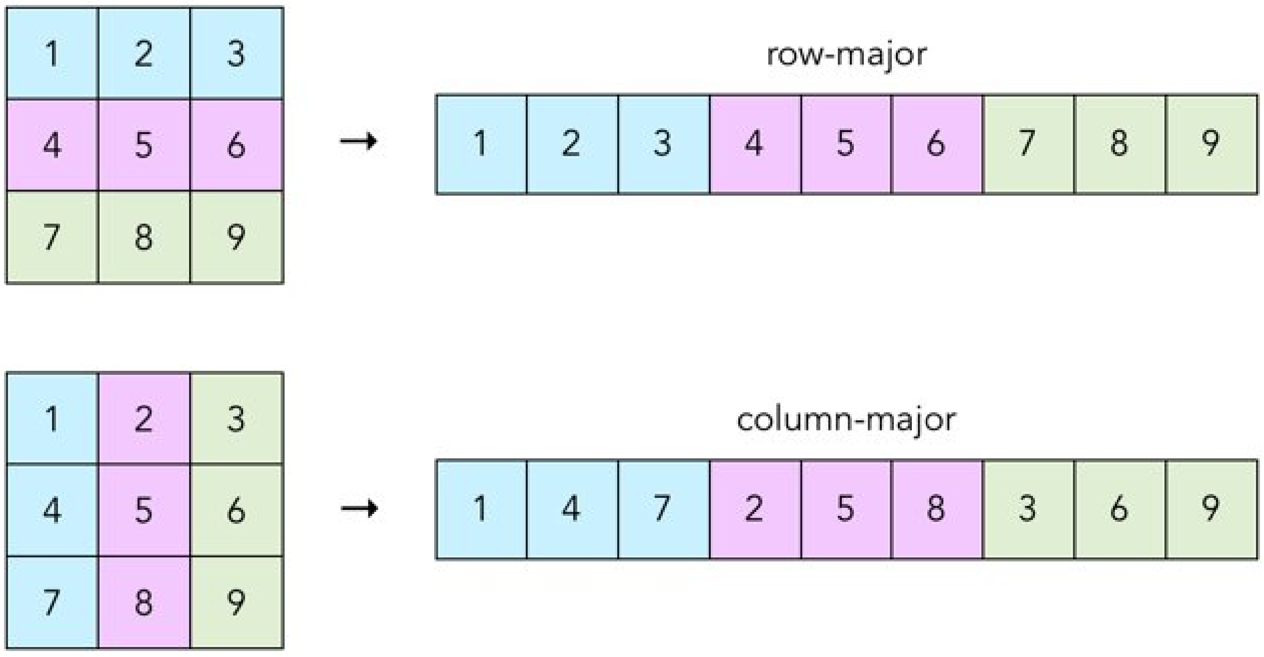
Return code	**Exception**
57	The length of either array is less than 1.
58	The stride of either array is less than 1.

To test your function, run `make test-dot`. Refer back to Task 1 for debugging instructions.

Task 5: Matrix Multiplication

Conceptual Overview: Storing Matrices

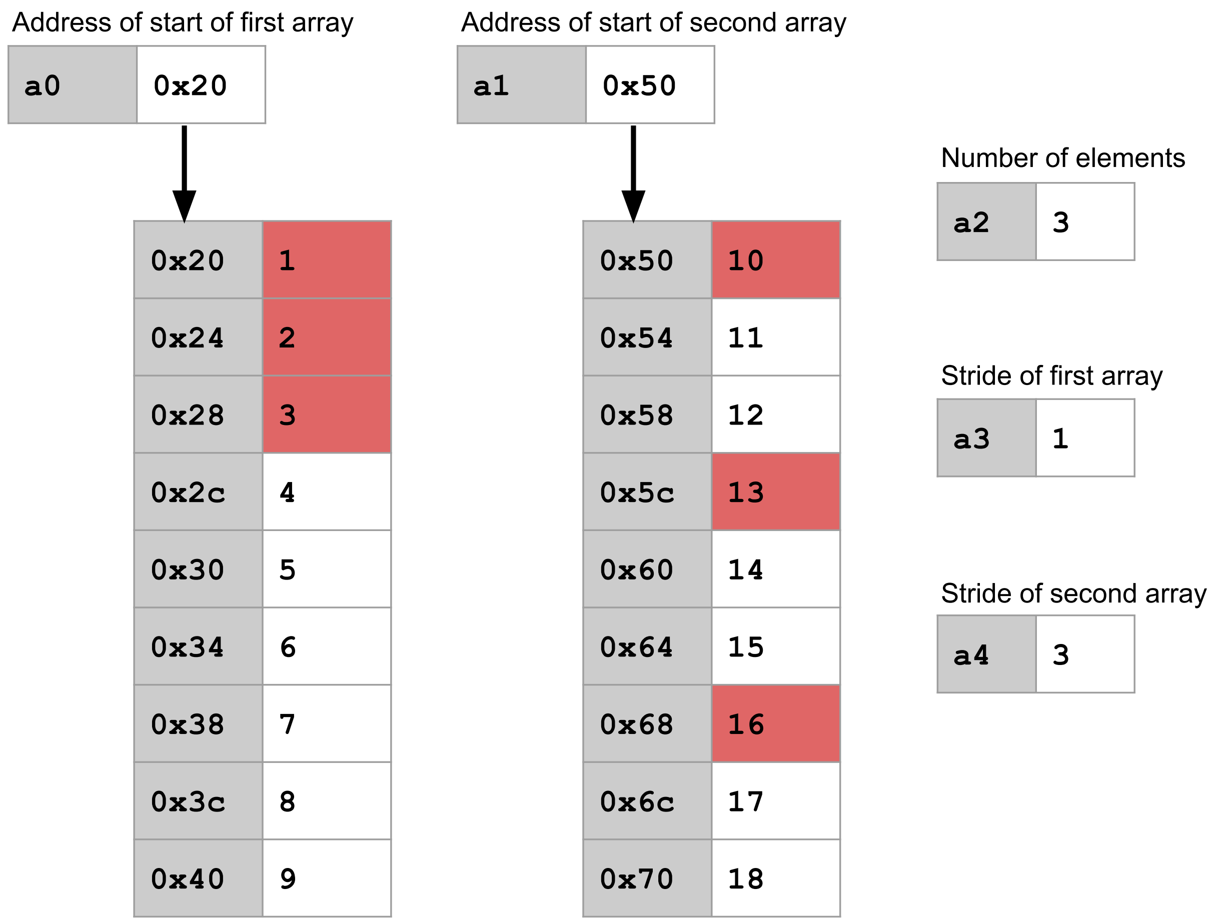
A matrix is a 2-dimensional array of integers. In this project, matrices will be stored as an integer array in row-major order. Row-major order means we each row of the matrix consecutively in memory as a 1-dimensional integer array.



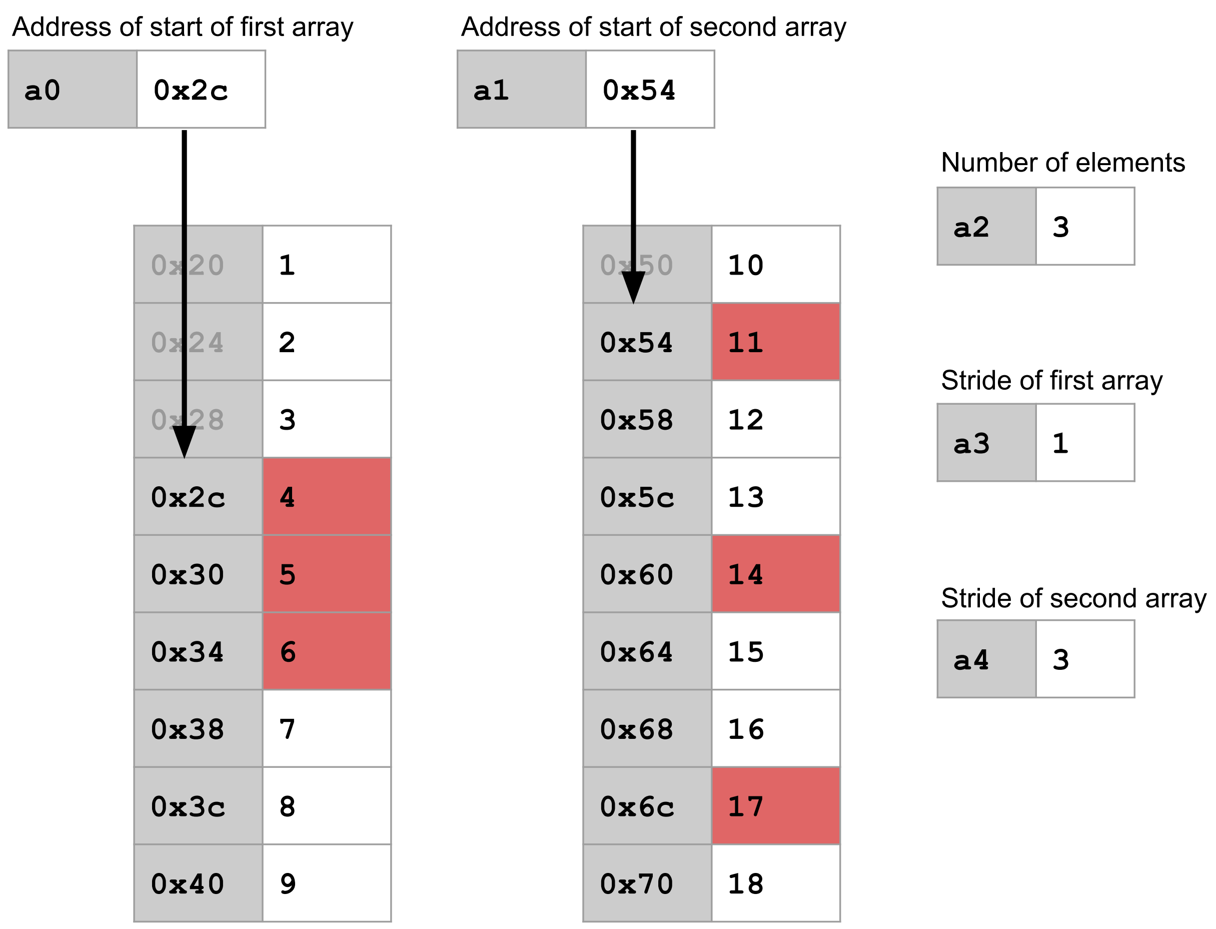
Conceptual Overview: Matrix Multiplication

The matrix multiplication function takes in two integer matrices A (dimension $n \times m$) and B (dimension $m \times k$) and outputs an integer matrix C (dimension $n \times k$).

To calculate the entry at row i, column j of C, take the dot product of the ith row of A and the jth column of B. Note that this can be done by calling the `dot` function with the proper strides.



For example, in the above diagram, we are computing the entry in row 1, column 1 of C by taking the dot product of the 1st row of A and the 1st row of B.



In the above diagram, we are computing the entry in row 2, column 2 of C. Note that we are changing the pointer to the start of the array in order to access later rows and columns.

Your Task

Fill in the `matmul` function in `src/matmul.s`.

`matmul`: Task 5.

Arguments	`a0`	`int *`	A pointer to the start of the first matrix A (stored as an integer array in row-major order).
	`a1`	`int`	The number of rows (height) of the first matrix A.
	`a2`	`int`	The number of columns (width) of the first matrix A.
	`a3`	`int *`	A pointer to the start of the second matrix B (stored as an integer array in row-major order).
	`a4`	`int`	The number of rows (height) of the second matrix B.
	`a5`	`int`	The number of columns (width) of the second matrix B.
	`a6`	`int *`	A pointer to the start of an integer array where the result C should be stored. You can assume this memory has been allocated (but is uninitialized) and has enough space to store C.
Return values	None		

If the input is malformed in the following ways, put the appropriate return code into `a1` and run `call exit2` to quit the program.

Return code	**Exception**
59	The height or width of either matrix is less than 1.
59	The number of columns (width) of the first matrix A is not equal to the number of rows (height) of the second matrix B.

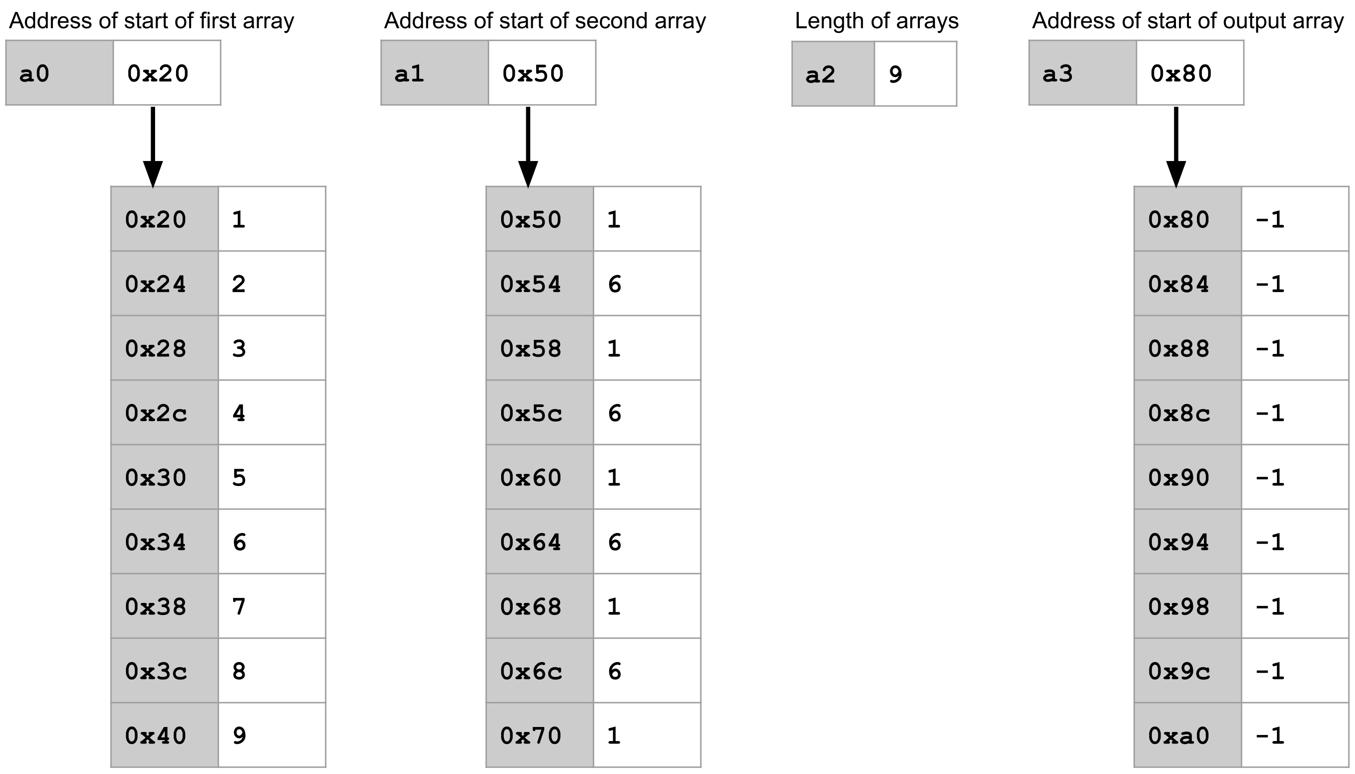
To test your function, run `make test-matmul`. Refer back to Task 1 for debugging instructions.

Task 6: Testing

In this task, you will be writing tests for some mathematical functions that have already been implemented for you.

Conceptual Overview: Loss Functions

A loss function takes in two integer arrays and outputs an integer array containing some measure of how different each pair of corresponding entries are. Some loss functions also output the sum of all the difference measurements. This project uses three different loss functions.



The absolute loss function computes and outputs the absolute difference between each pair of corresponding entries, and then outputs the sum of all the absolute differences.

Sum of elements in the output array

a0	28

Output array

0x80	0
0x84	4
0x88	2
0x8c	2
0x90	4
0x94	0
0x98	6
0x9c	2
0xa0	8

The squared loss function computes and outputs the square of the difference between each pair of corresponding entries, and then outputs the sum of all the squared differences.

Sum of elements in the output array

`a0`	`144`

Output array

`0x80`	`0`
`0x84`	`16`
`0x88`	`4`
`0x8c`	`4`
`0x90`	`16`
`0x94`	`0`
`0x98`	`36`
`0x9c`	`4`
`0xa0`	`64`

The zero-one loss function computes whether each pair of corresponding entries is equal, and does not output any sum.

Output array

0x80	1
0x84	0
0x88	0
0x8c	0
0x90	0
0x94	1
0x98	0
0x9c	0
0xa0	0

These loss functions use a helper function `initialize-zero`. It takes in the length of the array as input and outputs a newly-allocated array of the given length, filled with zeros.

Your Task

Fill in the tests for the three loss functions and the `initialize-zero` helper function in `unittests/studenttests.py`.

We recommend looking through `unittests/unittests.py` to understand how the Python framework for writing tests works.

Arguments	`a0`	`int *`	A pointer to the start of the first input array.
	`a1`	`int *`	A pointer to the start of the second input array.
	`a2`	`int`	The number of integers in the array.
	`a3`	`int *`	A pointer to the start of the output array, where the results will be stored.

Return values	`a0`	`int`	The sum of the elements in the output array.

Submission and Grading

Submit your code to the Project 2A assignment on Gradescope.

To ensure the autograder runs correctly, do not add any `.import` statements to the starter code. Also, make sure there are no `ecall` instructions in your code.

Dark Mode

Part B: File Operations and Classification

In this part, you will implement file operations to read pictures of handwritten digits. Then you will use your math functions from the previous part to determine what digit is in the picture.

If you are curious how the machine learning algorithm works, you can expand the Neural Networks section below. This is optional and not required to finish the project.

▶ Optional: Neural Networks

Task 7: Read Matrix

Conceptual Overview: Matrix Files

Recall from Task 5 that matrices are stored in memory as an integer array in row-major order.

Matrices are stored in files as a consecutive sequence of 4-byte integers. The first and second integers in the file indicate the number of rows and columns in the matrix, respectively. The rest of the integers store the elements in the matrix in row-major order.

To view matrix files, you can run `make read matrix_file.bin`. We have also provided a human-readable version of each of the matrix files at `matrix_file.txt`.

A video walkthrough of how to read and convert matrix files is available at this link.

Your Task

Fill in the `read_matrix` function in `src/read_matrix.s`. This function should do the following:

1. Open the file with read permissions. The filepath is provided as an argument (`a0`).
2. Read the number of rows and columns from the file (recall: these are the first two integers in the file). Store these integers in memory at the provided pointers (`a1` for rows and `a2` for columns).
3. Allocate space on the heap to store the matrix. (Hint: Use the number of rows and columns from the previous step to determine how much space to allocate.)
4. Read the matrix from the file to the memory allocated in the previous step.
5. Close the file.

6. Return a pointer to the matrix in memory.

`read_matrix`: Task 7.			
Arguments	`a0`	`char *`	A pointer to the filename string.
	`a1`	`int *`	A pointer to an integer which will contain the number of rows.
	`a2`	`int *`	A pointer to an integer which will contain the number of columns.
Return values	`a0`	`int *`	A pointer to the matrix in memory.

If the input is malformed in the following ways, put the appropriate return code into `a1` and run `call exit2` to quit the program.

Return code	**Exception**
88	`malloc` returns an error.
89	`fopen` returns an error.
90	`fclose` returns an error.
91	`fread` does not read the correct number of bytes.

To implement this function, you will need to call some utility functions. A complete set of function definitions can be found in the appendix. The relevant function definitions for this task are provided below (expand the section to see them).

▶ Task 7: Relevant Function Definitions

To test your function, run `make test-read_matrix`. Refer back to Task 1 for debugging instructions.

Task 8: Write Matrix

Fill in the `write_matrix` function in `src/write_matrix.s`. This function should do the following:

1. Open the file with write permissions. The filepath is provided as an argument.
2. Write the number of rows and columns to the file. (Hint: The `fwrite` function expects a pointer to data in memory, so you should first store the data to memory, and then pass a pointer to the

data to `fwrite`.)

3. Write the data to the file.
4. Close the file.

`write_matrix`: Task 8.			
Arguments	`a0`	`char *`	A pointer to the filename string.
	`a1`	`int *`	A pointer to the matrix in memory (stored as an integer array).
	`a2`	`int`	The number of rows in the matrix.
	`a3`	`int`	The number of columns in the matrix.
Return values	None		

If the input is malformed in the following ways, put the appropriate return code into `a1` and run `call exit2` to quit the program.

Return code	**Exception**
89	`fopen` returns an error.
92	`fwrite` does not write the correct number of bytes.
90	`fclose` returns an error.

To implement this function, you will need to call some utility functions. A complete set of function definitions can be found in the appendix. The relevant function definitions for this task are provided below (expand the section to see them).

▶ Task 8: Relevant Function Definitions

To test your function, run `make test-write_matrix`. Refer back to Task 1 for debugging instructions.

Task 9: Classify

Fill in the `classify` function in `src/classify.s`. This function should do the following:

1. Read three matrices `m0`, `m1`, and `input` from files. Their filepaths are provided as arguments.

2. Compute `h = matmul(m0, input)`. You will probably need to `malloc` space to fit `h`.
3. Compute `h = relu(h)`. Recall that `relu` is performed in-place.
4. Compute `o = matmul(m1, h)` and write the resulting matrix to the `output` file. The `output` filepath is provided as an argument.
5. Compute and return `argmax(o)`. If the print argument is set to 0, then also print out `argmax(o)` and a newline character.
6. Free any data you allocated with `malloc` in this function.

`classify`: Task 9.			
Arguments	`a0`	`int`	`argc` (the number of arguments provided)
	`a1`	`char **`	`argv`, a pointer to an array of argument strings (`char *`)
	`a1[1] = *(a1 + 4)`	`char *`	A pointer to the filepath string of the first matrix file `m0`.
	`a1[2] = *(a1 + 8)`	`char *`	A pointer to the filepath string of the second matrix file `m1`.
	`a1[3] = *(a1 + 12)`	`char *`	A pointer to the filepath string of the input matrix file `input`.
	`a1[4] = *(a1 + 16)`	`char *`	A pointer to the filepath string of the output file.
	`a2`	`int`	If set to 0, print out the classification. Otherwise, do not print anything.
Return values	`a0`	`int`	The classification (see above).

If the input is malformed in the following ways, put the appropriate return code into `a1` and run `call exit2` to quit the program.

Return code	**Exception**
88	`malloc` returns an error.
72	There are an incorrect number of command line arguments.

To implement this function, you will need to call some utility functions. A complete set of function definitions can be found in the appendix. The relevant function definitions for this task are provided below (expand the section to see them).

▶ Task 9: Relevant Function Definitions

Testing and Debugging

To test your function, run `make test-classify` and `make test-main`.

Debugging instructions are available in both video form and written form (expand the section below).

▶ Text Version

Submission and Grading

Submit your code to the Project 2B assignment on Gradescope.

To ensure the autograder runs correctly, do not add any `.import` statements to the starter code. Also, make sure there are no `ecall` instructions in your code.

Congratulations on finishing Project 2! Just for fun, you can now use your code to classify your own handwritten digits. This is optional and not required to finish the project. If you're interested, expand the section below.

▶ Optional: Classifying Your Own Digits

Dark Mode

Appendix: Function Definitions

This appendix contains the function definitions (arguments, return values, error codes) for all the provided utility functions and all the functions you need to implement.

Utilities: Memory Allocation

`malloc`

`malloc`: Allocates heap memory.			
Arguments	`a0`	`int`	The size of the memory that we want to allocate (in bytes).
Return values	`a0`	`void *`	A pointer to the allocated memory. If the allocation failed, this value is 0.

`free`

`free`: Frees heap memory.			
Arguments	`a0`	`int`	A pointer to the allocated memory to be freed.
Return values	None		

Utilities: File Operations

`fopen`

`fopen`: Open a file for reading or writing.			
Arguments	`a1`	`str *`	A pointer to the filename string.
	`a2`	`int`	Permission bits. 0 for read-only, 1 for write-only.

Return values	`a0`	`int`	A file descriptor. This integer can be used in other file operation functions to refer to the opened file. If opening the file failed, this value is -1.

`fread`

`fread`: Read bytes from a file to a buffer in memory. Subsequent reads will read from later parts of the file.			
Arguments	`a1`	`int`	The file descriptor of the file we want to read from, previously returned by `fopen`.
	`a2`	`int*`	A pointer to the buffer where the read bytes will be stored. The buffer should have been previously allocated with `malloc`.
	`a3`	`int`	The number of bytes to read from the file.
Return values	`a0`	`int`	The number of bytes actually read from the file. If this differs from the argument provided in `a3`, then we either hit the end of the file or there was an error.

`fwrite`

`fwrite`: Write bytes from a buffer in memory to a file. Subsequent writes append to the end of the existing file.			
Arguments	`a1`	`int`	The file descriptor of the file we want to read from, previously returned by `fopen`.
	`a2`	`void *`	A pointer to a buffer containing what we want to write to the file.
	`a3`	`int`	The number of elements to write to the file.
	`a4`	`int`	The size of each element. In total, `a3` × `a4` bytes are written.
Return values	`a0`	`int`	The number of items actually written to the file. If this differs from the number of items specified (`a3`), then we either hit the end of the file or there was an error.

`fclose`

`fclose`: Close a file, saving any writes we have made to the file.			
Arguments	`a1`	`str *`	The file descriptor of the file we want to close, previously returned by `fopen`.
Return values	`a0`	`int`	0 on success, and -1 otherwise.

Utilities: Printing

`print_int`

`print_int`: Prints an integer.			
Arguments	`a1`	`int`	The integer to print.
Return values	None		

`print_char`

`print_char`: Prints a character.			
Arguments	`a1`	`char`	The character to print.
Return values	None		

Part A

`relu`

`relu`: Task 2.			
Arguments	`a0`	`int *`	A pointer to the start of the integer array.
	`a1`	`int`	The number of integers in the array. You can assume that this argument matches the actual length of the integer array.

Return values	None

argmax

`argmax`: Task 3.			
Arguments	`a0`	`int *`	A pointer to the start of the integer array.
	`a1`	`int`	The number of integers in the array. You can assume that this argument matches the actual length of the integer array.
Return values	`a0`	`int`	The index of the largest element. If the largest element appears multiple times, return the smallest index.

dot

`dot`: Task 4.			
Arguments	`a0`	`int *`	A pointer to the start of the first array.
	`a1`	`int *`	A pointer to the start of the second array.
	`a2`	`int`	The number of elements to use in the calculation.
	`a3`	`int`	The stride of the first array.
	`a4`	`int`	The stride of the second array.
Return values	`a0`	`int`	The dot product of the two arrays, using the given number of elements and the given strides.

matmul

`matmul`: Task 5.

Arguments	a0	int *	A pointer to the start of the first matrix A (stored as an integer array in row-major order).
	a1	int	The number of rows (height) of the first matrix A.
	a2	int	The number of columns (width) of the first matrix A.
	a3	int *	A pointer to the start of the second matrix B (stored as an integer array in row-major order).
	a4	int	The number of rows (height) of the second matrix B.
	a5	int	The number of columns (width) of the second matrix B.
	a6	int *	A pointer to the start of an integer array where the result C should be stored. You can assume this memory has been allocated (but is uninitialized) and has enough space to store C.
Return values	None		

Part B

read_matrix

read_matrix: Task 7.			
Arguments	a0	char *	A pointer to the filename string.
	a1	int *	A pointer to an integer which will contain the number of rows.
	a2	int *	A pointer to an integer which will contain the number of columns.
Return values	a0	int *	A pointer to the matrix in memory.

write_matrix

write_matrix: Task 8.

Arguments	`a0`	`char *`	A pointer to the filename string.
	`a1`	`int *`	A pointer to the matrix in memory (stored as an integer array).
	`a2`	`int`	The number of rows in the matrix.
	`a3`	`int`	The number of columns in the matrix.
Return values	None		

`classify`

`classify`: Task 9.			
Arguments	`a0`	`int`	`argc` (the number of arguments provided)
	`a1`	`char **`	`argv`, a pointer to an array of argument strings (`char *`)
	`a1[1] = *(a1 + 4)`	`char *`	A pointer to the filepath string of the first matrix file `m0`.
	`a1[2] = *(a1 + 8)`	`char *`	A pointer to the filepath string of the second matrix file `m1`.
	`a1[3] = *(a1 + 12)`	`char *`	A pointer to the filepath string of the input matrix file `input`.
	`a1[4] = *(a1 + 16)`	`char *`	A pointer to the filepath string of the output file.
	`a2`	`int`	If set to 0, print out the classification. Otherwise, do not print anything.
Return values	`a0`	`int`	The classification (see above).

Dark Mode