CS61C Spring 2014 Project 1: C/MIPS

TA: Jeffrey Dong / William Ku
Part 1: Due 02/16 @ 23:59:59
Part 2: Due 02/23 @ 23:59:59

Updates

- 02/19 : Some important annoucements/reminders updated in Part 2 Task B spec.
- 02/16 : Part 2 released.
- 02/13 : Added a missing free() statement to `sparsify.c`. Using the older version won't affect your submission's correctness, but you should re-copy it.
- 02/11 : Submission available.
- 02/10 : In `calc_dist.c`, "assets/" renamed to "templates/". If you copied the file before then, please re-copy it.
- 02/09 : Project 1 released.

Clarifications/Reminders

- Start Early!
- This project should be done on either the **hive machines** or the **the computers in 271,273,275, or 277 Soda**. Parts of your program may not work on other computers.
- It is suggested that you work with **one** partner for this project. You may share code with your partner, but ONLY your partner. The submit command will ask you for your partner's name, so only one partner needs to submit. (It would be nice if only one partner submitted).
- **Make sure you read through the project specs before starting.**
- It is a known issue that some .GIFs are loaded improperly. For example, if you test with `translated_5.bmp`, your program should identify it as a 5, but will assign it a small, but non-zero, distance. This will not affect your functions during grading. Unfortunately, since we will focus on answering questions/developing proj1-2, this will probably remain as-is.
- For part 2 task B, make sure to run `s2d_main.s` as that is the main entrance point for the program.
- For part 2 task B, **do not** assume the matrix is 10x10. The spec was written using 10x10 matrix to give you a concrete example. Your functions **should** be able to handle all sizes of matrices, since the necessary size information (e.g. width, height) are given in the function arguments.

Goals

This project will expose you to C and MIPS in greater depth. The first part gives you the opportunity to sharpen your C programming skills that you have learned in the past few weeks, while the second part dives deeper into coding MIPS. Part of this project will aslo serve as the basis for the upcoming project 3. We hope you will have fun!

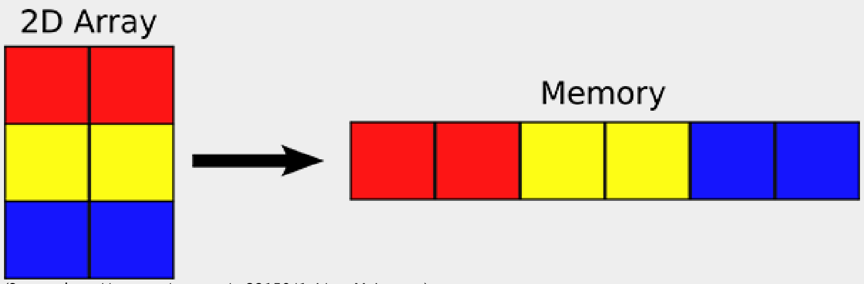
Part 1 (Due 2/16 @ 23:59:59)

Background

Bitmap Images

We will be working with 8-bit grayscale bitmap images in the first part of our project. In this file format, each pixel takes on a value between 0 and 255, with 0 corresponding to black, 255 to white, and values in between to various shades of gray. Together, the pixels form a 2D matrix with *image_height* rows and *image_width* columns.

Since each pixel may be one of 256 values, we can represent an image in memory using an array of unsigned chars of size *image_width* * *image_height*. We map the 2D matrix into the 1D array such that each row will occupy a contiguous part of the array, and rows at the bottom are put in the latter part of the array (see illustration below):



(Source: http://cnx.org/content/m32159/1.4/rowMajor.png)

We can refer to individual pixels of the array by specifying the row and column it's located in. Recall that in a matrix, rows and columns are numbered from the top left. We will follow this numbering scheme as well, so the leftmost red square is at row 0, column 0, and the rightmost blue square is at row 2, column 1. In this project, we will also refer to an element's column # as its *x position*, and it's row # as its *y position*. We can also call the # of columns of the image as its *width*, and the # of rows of the image as its *height*. Thus, the image above has a width of 2, height of 3, and the element at x=1 and y=2 is the rightmost blue square.

Digit Recognition

We are trying to implement a simple version of digit recognition - without any complicated machine learning - through comparing the "digit distances" between a set of ten template images (of digit 0 to 9) and a test image. The digit of the template image that yields the minimum "digit distance" (think of "distance" as resemblence) will be the classification result for the test image.

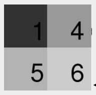
<--- Template Images

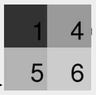
For any two images, the way we are going to compute their distance is using the squared Euclidean distance, computed by summing up the squared differences bewteen each pixel in the two images. Formally, if *x* and *y* are two M by N images represented in a 1-D array (i.e. *x* = (*x_1*, *x_2*, ... *x_MN*), *y* = (*y_1*, *y_2*, ... *y_MN*)), the squared Euclidean distance is given by:

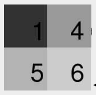
$$d_E^2(x,y)=\sum_{k=1}^{MN}\left(x^k-y^k\right)^2$$

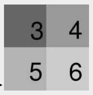
(Source: http://www.cis.pku.edu.cn/faculty/vision/wangliwei/pdf/IMED.pdf)

For example, given two sets of two 2x2 images below:



<--- Euclidean distance is 0+0+0+0 = 0 --->




<--- Euclidean distance is 4+0+0+0 = 4 --->


(Source: http://cybertron.cg.tu-berlin.de/pdci08/imageflight/descriptors.html)

To maximize the use of our digit recognition program, we are also supporting rotated (90, 180, 270 degrees only), flipped (horizontal and vertical), and translated images (but NOT scaling). Therefore, our "digit distance" for between each of the digit template and the test image will be the **minimum** squared Euclidean distance obtained through rotating, flipping, and translating the template image. For example, if I obtain 4 (your implementation will have more) squared Euclidean distances, say (50, 65, 90, 10), from comparing the test image to all different transformations (i.e. rotations, flip, translations) of my digit 0 template, then the "digit distance" for digit 0 will be min(50, 65, 90, 10) = 10.

Sparse Representations

Notice that most of the pixels in our images are white. While using an array of unsigned chars to represent an image certainly works, for large images we would be using a lot more memory than we need as there will be a lot of repeated values. Instead, we can store just the locations and values of the non-white pixels and stipulate that any pixel which is not stored is white-colored. While we do need to specify more data per pixel (location in addition to value), as long as the ratio of white to non-white pixels is high enough we will be saving space.

Formally, matrices with many elements of the same value are known as sparse matrices. In contrast with dense matrices, where the value of every element needs to be stored, sparse matrices can be efficiently stored with specialized representations. Different representations have different advantages, and we will be using the list-of-lists representation for its relative simplicity (the exact specifications of this format will be described later). Do note that typical sparse matrices have a large amount of 0s, while our images will have a large amount of 255s (the value of white pixels). *Normally, sparse matrices omit 0s, but for this project, we will be omitting the value 255.* Most literature regarding sparse matrices assume that 0s are omitted, so don't get confused!

Getting started

Copy the files in the directory `~cs61c/proj/01` to your `proj1` directory, by entering the following command:

```
$ mkdir ~/proj1
$ cp -r ~cs61c/proj/01/* ~/proj1
```

The files you will need to modify and submit are:

- `calc_dist.c`: You will be implementing the `calc_min_dist()` function. The `swap()`, `flip_horizontal()`, `transpose()`, `rotate_ccw_90()` are optional (but may be helpful).
- `make_sparse.c`: You will be implementing the `dense_to_sparse()` and `free_sparse` functions.
- `proj1_1A.txt`: You will report your results and analysis for task A in this file.

You are free to define and implement additional helper functions, but if you want them to be run when we grade your project, you must include them in `calc_dist.c` or `make_sparse.c`. **Changes you make to any other files will be overwritten when we grade your project.**

The rest of the files are part of the framework. It may be helpful to look at all the other files.

- `Makefile`: Defines all the compilation commands.
- `digit_rec.h`: Defines template width and height and function signatures related to digit recognition.
- `digit_rec.c`: Loads bitmap images and calls `calc_min_dist()` to calculate distances.
- `test_digitrec.c`: Examines the calculated distances and prints out the digit recognition result, which is the digit with shortest distance to the test image.
- `sparsify.h`: Defines the `Row` and `Elem` structs, as well as sparsify function headers.
- `sparsify.c`: Calls `dense_to_sparse()` and `free_sparse()`.
- `test_sparsify.c`: Contains functions to test your sparse representation.
- `utils.h`: Defines `Image` struct and utility function signatures.
- `utils.c`: Defines bitmap loading and printing functions.
- `templates/`: Contains the template images.
- `basic_tests/`: Contains test images for basic test cases (eg. only rotated, etc).
- `advanced_tests/`: Contains test images for more advanced test cases.

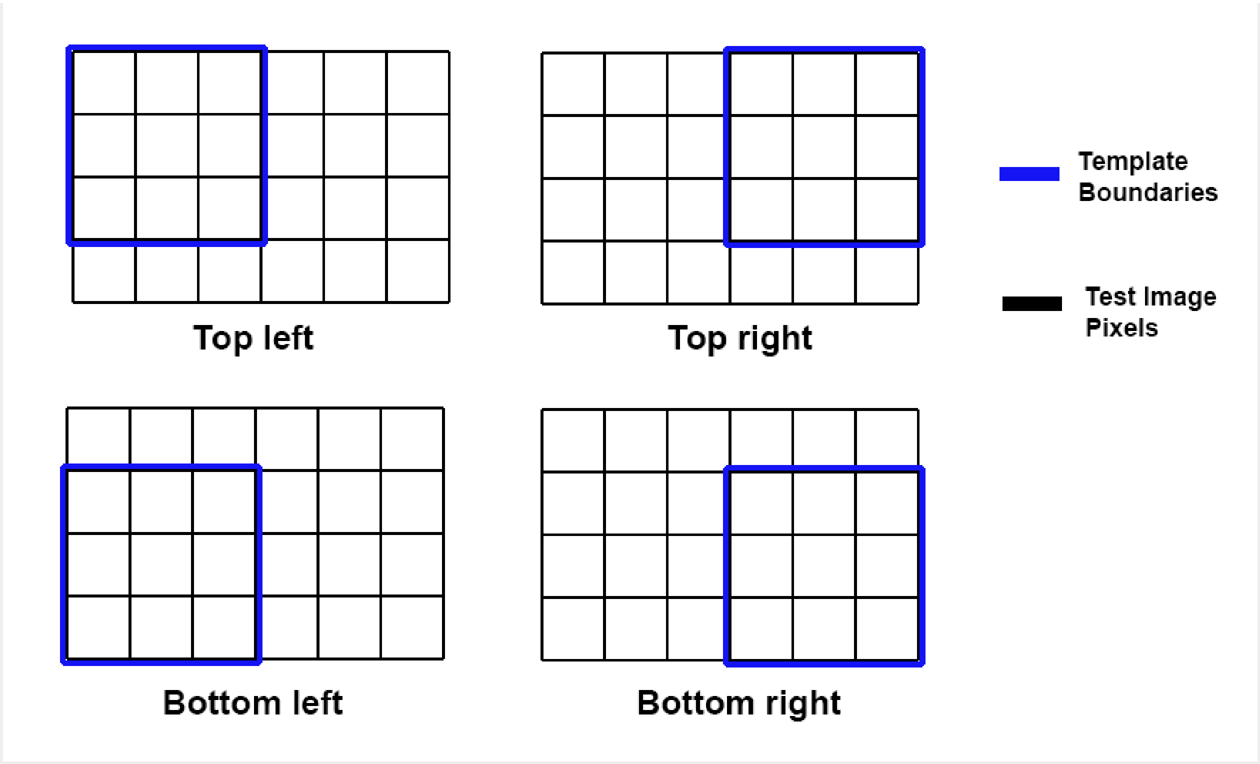
Your Job

Task A (Due 2/16): Your first task will be to implement the digit recognition.

Here are some tips and reminders:

- When supplied with a test image, the program `digit-rec` will call `calc_min_dist()`, which will take each digit template and calculate the squared Euclidean distance between the test and template images.
- In `calc_min_dist()`, you should be returning the smallest distance out of all the possible translations/rotations/flips.
- In case you need access to template width and height, they are defined in `digit_rec.h`
- The program will examine the distances and declare the digit with the minimum distance as the classification result.
- Make sure to take into account the cases when a test image is rotated (90, 180, 270 degrees only), flipped, or translated (**Hint 1**: You only need to implement either horizontal or vertical flip, but not both, why? **Hint 2**: For translation, start by aligning the corners of the images, and then move the smaller image across the larger image).
- All test images will have a white margin of at least the same width as the template images (i.e. translation image will not move pass the margins).
- The test image is **always** larger or equal to the size of the template images.
- Template images in this project are square images (i.e. width = height).
- Test images **may or may not** be square images.
- You may assume the fonts in all of the images are approximately the same size (i.e. no need to detect scaling).

Below is an illustration of by how much you should translate if you're testing a 6x4 image with a 3x3 template. Notice that the template never "leaves" the test image.



A sample output (and a sanity check) for your program is as follows:

```
$ make digit-rec
gcc -g -std=c99 -o digit-rec digit_rec.c calc_dist.c test_digitrec.c utils.c
$ ./digit-rec templates/3_64.bmp
0: <some number>
1: <some number>
2: <some number>
3: 0
4: <some number>
5: <some number>
6: <some number>
7: <some number>
8: <some number>
9: <some number>
I think your image is a 3
```

After implementing your digit recognition algorithm, please answer the following questions in proj1_1A.txt:

1. For each of the test image provided in `advanced_tests/`, which images are not recognized correctly?
2. For each test image listed in the previous question, What is the difference in "digit distances" between the recognized digit and the digit it is supposed to be (e.g. if a digit 3 is recognized as 8, and 3 has a "digit distance" of 10 while 8 has a "digit distance" of 20, then difference in distances is 20 - 10 = 10).
3. Give a one to two sentence explanation on why you think the images mentioned above are recognized incorrectly.

Task B (Due 2/16): Your second task will be to implement the matrix sparsify functionality.

As mentioned earlier, we will use the list-of-list format, which consists of a NULL-terminated linked list with each node containing a NULL-terminated linked list. The nodes of the outer linked list will be of type `Row` and each represent a row of the matrix. The nodes of the inner linked list will be of type `Elem` and each represent an element in the row. The declarations of the structs `Row` and `Elem` are in `sparsify.h`.

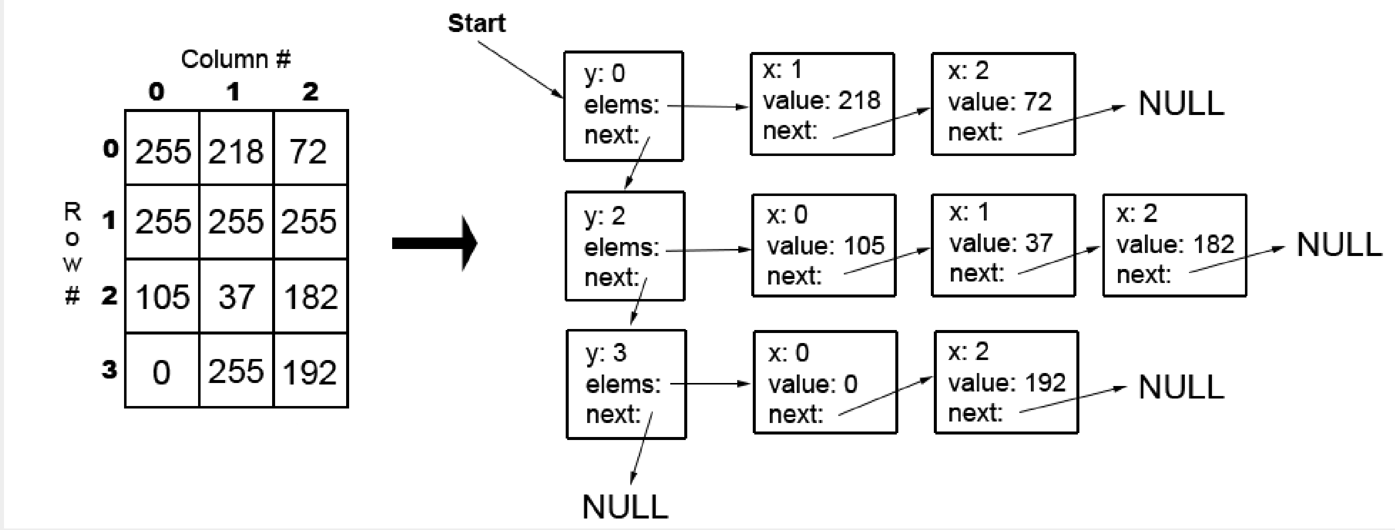
Your task is to write the `dense_to_sparse()` and `free_sparse()` functions located in `make_sparse.c`. `dense_to_sparse` takes an array of unsigned chars, converts it into a linked list of linked lists, and returns a pointer to the first `Row` in the list. Keep in mind that local variables are no longer in scope, so **you must use dynamic memory allocation** in the function. Since memory allocation could fail, you need to check whether the pointer returned is valid or not. If it is NULL, you should call `allocation_failed` (defined in `utils.h`).

The function `free_sparse()` frees the memory associated with sparse matrix. You must make sure there are no memory leaks, and you should test with Valgrind (see Lab 3 for a refresher) to make sure.

Here are a few key points regarding your sparse representation:

1. White pixels (value = 255) are not stored, whereas elements with values 0-254 are. If an element in the matrix is not present in the sparse representation, we know that it has the value of 255.
2. The list of Row structs should be sorted by y coordinate from smallest to largest.
3. Each list of Elem structs should be sorted by x coordinate from smallest to largest.
4. If an entire row is white, no Row node should be created (since it will only contain an empty list, we might as well save some space). This means it is possible for the `dense_to_sparse()` function to return NULL.

The following example illustrates these points:



Turning a dense matrix into a sparse matrix.

You can compile your code for task B with the following command:

```
$ make sparsify
```

Running the program with no arguments will test your code with a few basic tests. You can also pass in the name of a bitmap image, and the code will sparsify the image (although we have not included any tests).

Valgrind can be run on the sparsify program via the following command:

```
$ make sparsify-memcheck
```

There should be 0 errors for your code (you can ignore any suppressed errors).

Debugging and Testing

Your code is compiled with the -g flag, so you can use GDB to help debug your program. While adding in print statements can be helpful, GDB can make debugging a lot quicker, especially for memory access-related issues. While you are working on the project, we encourage you to keep your code under version control. **However, if you are using a hosted repository, please make sure that it is private, or else it could be viewed as a violation of academic integrity.**

In addition, we have included a few functions to help make development and debugging easier:

- **`print_bmp(const unsigned char *data, int w, int h)`**: This function takes in an array of pixels and prints their values in hex. w refers to the width of the image, while h refers to the height. This function is declared in `utils.h`, which is included in both `calc_dist.c` and `make_sparse.c`.
- **`print_sparse(Row *sparse_matrix)`**: This function takes in a sparse matrix and prints the x coordinate, y coordinate, and value of each element in the sparse matrix. Note: this function will only work properly if your linked lists are correct. This function is declared in `sparsify.h`, so it is already included in `make_sparse.c`. For each row, it will print the row #, followed by each element in the format: `[( <x> , <y> ): <value> ]`. For the sparse matrix in the image above, you should expect to see:

```
Printing sparse matrix:
Row 0: [(1,0):218] [(2,0):72]
Row 2: [(0,2):105] [(1,2):37] [(2,2):182]
Row 3: [(0,3):0] [(2,3):192]
end of matrix
```

The test cases we provide you are not all the test cases we will test your code with. You are highly encouraged to write your own tests before you submit. Feel free to add additional tests into the skeleton code, **but do not make any modifications to function signatures or struct declarations.** This can lead to your code failing to compile during grading.

If you mistakenly break parts of the skeleton, you can always grab a new copy of the file by running the following command, but BE VERY CAREFUL not to overwrite all your work.

```
$ cp ~cs61c/proj/01/<NAME OF FILE> ~/proj1
```

Before you submit, **make sure you test your code on the Hive machines.** We will be grading your code there. IF YOUR PROGRAM FAILS TO COMPILE, YOU WILL AUTOMATICALLY GET A ZERO FOR THAT PORTION (ie. If `digit-rec` works but `sparsify` doesn't compile, you will receive points for `digit-rec` but none for `sparsify`). There is no excuse not to test your code.

Submission/Early Submission Incentive

We *highly* encourage everyone to get started on this project early, especially since debugging could take a lot longer than you plan for. To encourage everyone to get started early, we will be offering one bonus point for anyone that submits a working `calc_min_dist()` that can handle **non-translated, non-flipped, and non-rotated images by Thursday night (11:59 PM).** This means if we give you any of the template images, your function should return a distance of 0 (of course we will be using other 64x64 images to test your code). As this is for bonus points, slip days will not apply.

To submit for the early bonus, enter in the following command. You will only need to turn in `calc_dist.c`.

```
$ cd ~/proj1
$ submit proj1-1-early
```

The full proj1-1 is due Sunday. To submit the full proj1-1, enter in the following. You should be turning in `calc_dist.c`, `make_sparse.c`, and `proj1_1A.txt`.

```
$ cd ~/proj1
$ submit proj1-1
```

Part 2 (Due 2/23 @ 23:59:59)

Background

The Collatz Conjecture

The Collatz conjecture was first proposed by Lothar Collatz in 1937. It states that repeatedly applying the following operation to any positive integer will eventually yield 1.

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod{2} \\ 3n+1 & \text{if } n \equiv 1 \pmod{2}. \end{cases}$$

(Source: Wikipedia)

In other words, if a positive integer *n* is even, divide it by 2, and if *n* is odd, multiply it by 3 and add 1. Repeat until *n* equals 1. The sequence of numbers generated by applying this operation is known as "hailstone sequences," and the smallest number of steps taken for *n* to reach 1 is known as its **stopping time**. In this project, we are interested in computing the stopping time of a number.

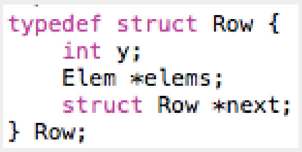
Sparse to Dense Matrix

In the first part of the project, we looked at how to turn a dense 2D matrix into sparse representation. One may ask, what about the other way around? This is the motivation behind why we are going to implement the sparse to dense matrix functionality – in MIPS!

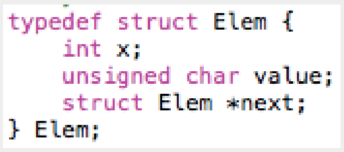
Since our implementation will be in MARS – the MIPS simulator – we are going to **assume each element in the matrix as one word/integer** to simplify things a little bit. Recall that each word/integer is 32 bits (or 4 bytes). You will be given a file, `s2d_sparse_matrix_data.s`, which loads a sparse matrix into the data segment of the simulator. We want to be able to turn that sparse matrix into its dense matrix representation, as well as to print it to the I/O of the simulator.

Additionally, we are going to make the **default value for sparse matrix as 0**, instead of 255 (as in part 1). In another word, the sparse matrix representation only stores values that are non-zero into the linked-list.

Recall our definitions of `Row` and `Elem` from part 1, which are:

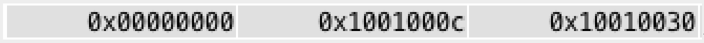


```
typedef struct Row {
    int y;
    Elem *elems;
    struct Row *next;
} Row;
```



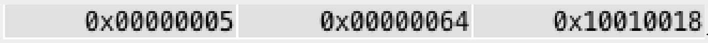
```
typedef struct Elem {
    int x;
    unsigned char value;
    struct Elem *next;
} Elem;
```

Since we have decided that our matrix values are integers, each field in `Row` and `Elem` occupies **one** word space (32bits; 4 bytes), as shown below:



0x00000000	0x1001000c	0x10010030

<--- Row 0 represented in memory (first integer: row number; second integer: Elem pointer, third integer: next Row pointer).



0x00000005	0x00000064	0x10010018

<--- Elem 5 represented in memory (first integer: column number; second integer: value, third integer: next elem pointer).

`mult` in MIPS

If you used multiplication somewhere in part 1, then you might wonder how to use multiplication in MIPS. Multiplication in MIPS is handled by the instruction `mult`, which takes as arguments two registers that hold the values to be multiplied (e.g. **`mult $s0, $s1`**). The result of the multiplication is stored inside the `hi` and `lo` special registers. These two register values can be obtained via the `mfhi` (which stands for "move from hi") and `mflo` (which stands for "move from lo") instructions. These two instrucitons each takes one argument, which is the register that you would like the hi or lo value to move to (e.g. **`mflo $t0`**). The reason behind having to use two registers (32 bit each) to hold our miltiplication result is that, when we multiple two 32-bit integers, the resulting value can be up to 64 bits. However, many times we are simply multiplying small values (say, 10*10). Therefore, we can only worry about the `lo` register.

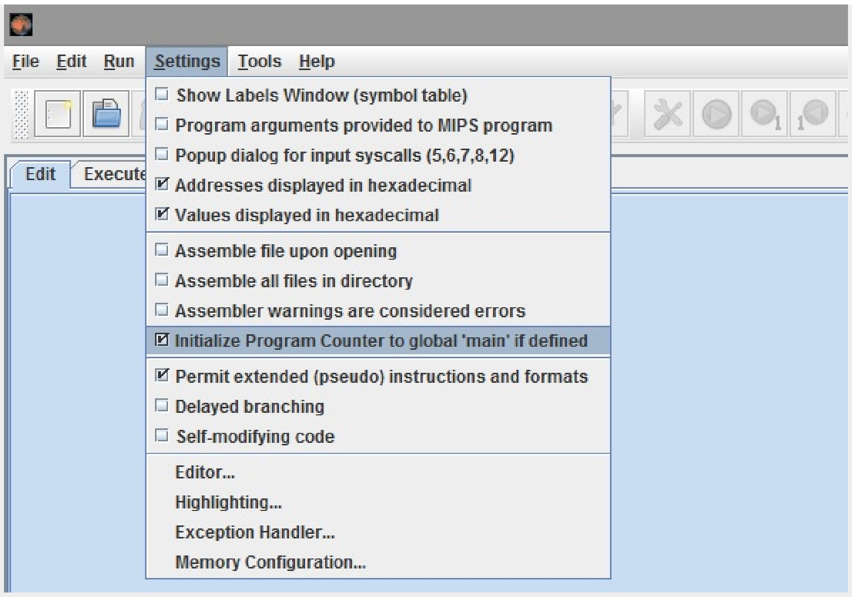
Getting started

Copy the files in the directory `~cs61c/proj/01_pt2` to your `proj1` directory, by entering the following command:

```
$ mkdir ~/proj1-2
$ cp -r ~cs61c/proj/01_pt2/* ~/proj1-2
```

It is recommended that you download MARS and run it from your own computer. Using MARS by SSH-ing from an instructional machine is *very slow.* (Since MARS is a Java program, you do not have to worry about compatibility between operating systems). The latest version of MARS is v4.4, and it can be found here: http://courses.missouristate.edu/KenVollmar/MARS/index.htm

Before you start, make sure the option **Initialize Program Counter to global 'main' is defined** (located in the Settings menu) is checked.



- [ ] Show Labels Window (symbol table)
- [ ] Program arguments provided to MIPS program
- [ ] Popup dialog for input syscalls (5,6,7,8,12)
- [✓] Addresses displayed in hexadecimal
- [✓] Values displayed in hexadecimal
- [ ] Assemble file upon opening
- [ ] Assemble all files in directory
- [ ] Assembler warnings are considered errors
- [✓] Initialize Program Counter to global 'main' if defined
- [✓] Permit extended (pseudo) instructions and formats
- [ ] Delayed branching
- [ ] Self-modifying code

Editor...
Highlighting...
Exception Handler...
Memory Configuration...

The files you will need to modify and submit are:

- `collatz_iterative.s`: Implement the `collatz_iterative` function.
- `collatz_recursive.s`: Implement the `collatz_recursive` function.
- `proj1_2A.txt`: Put your answer to the question in proj1-2 task A here.
- `s2d.s`: You will inteprett the sparse matrix representation and fill in the provided dense matrix with the correct values.
- `s2d_print_dense.s`: You will print out the values in the dense matrix in the specified format described in the next section.

The rest of the files are part of the framework. Remember, **changes you make to any other files will be overwritten when we grade your project.**

- `collatz_common.s`: Contains functions used by both `collatz_iterative.s` and `collatz_recursive.s`.
- `s2d_main.s`: This file loads the sparse matrix and calls `sparse2dense` and `print_dense`.
- `s2d_sparse_matrix_data.s`: This file stores a sparse matrix into the data segment of the simulator.

Your Job

Task A (Due 2/23): Your second task will be to implement the Collatz functions.

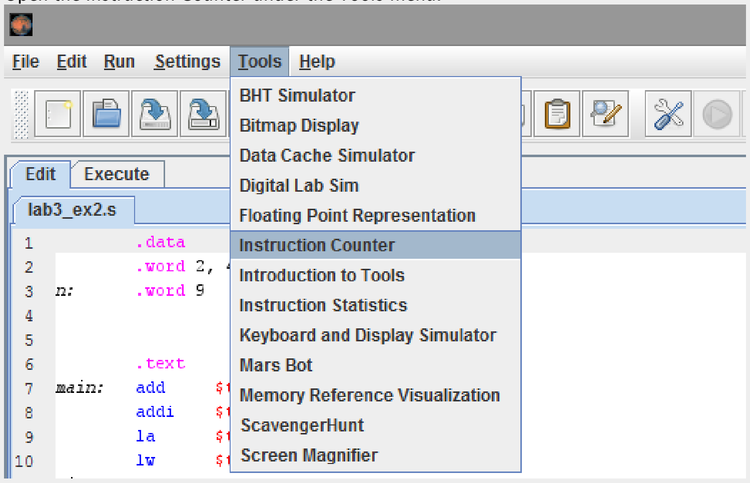
You task is to implement the collatz_iterative and collatz_recursive functions, which will return the stopping time of a given number. For each function, the number will be passed in via the `$a0` register, and you must return the correct result in `$v0`. You may assume that the input is guaranteed to be a positive integer. **Make sure to follow the appropriate function call conventions when implementing your solution. the `collatz_recursive` function MUST call `collatz_recursive` or points will be deducted.**

Note that the stopping time is defined as the number of iterations *until* the number reaches one. Hence, if the number passed in is 1, your function should return 0. Here are a few samples to verify your code:

```
If n = 1, stopping time = 0.
If n = 4, stopping time = 2.
If n = 29, stopping time = 18.
If n = 171, stopping time = 124.
```

In addition, we would like to compare the performance of the iterative and recursive versions of our functions by keeping track of the number of machine instructions executed by each. For this section, you will need a pencil and a (large) sheet of paper... well that would suck. Fortunately, MARS can keep track of the number of instructions executed via the Instruction Counter tool. To use the Instruction Counter:

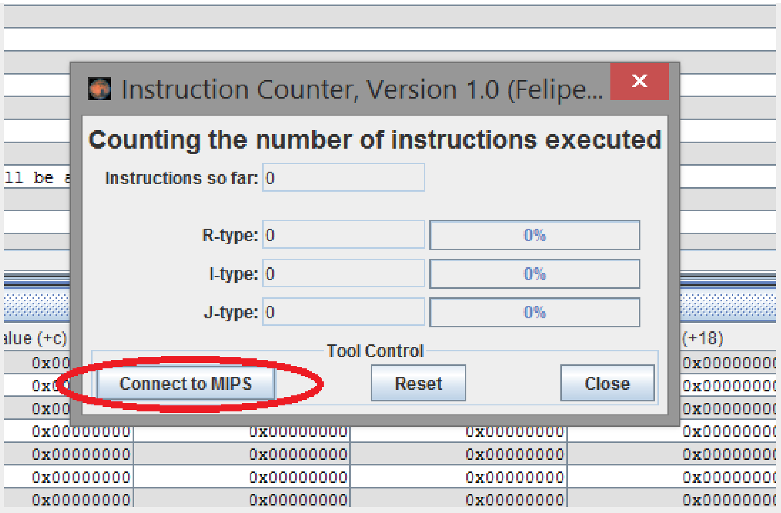
1. Open the Instruction Counter under the Tools menu.



```
        .data
        .word 2,
n:      .word 9


        .text
main:   add     $
        addi    $
        la      $
        lw      $
```

2. Assemble your program, then press the "Connect to MIPS" button.


Instruction Counter, Version 1.0 (Felipe...
Counting the number of instructions executed
Tool Control
Connect to MIPS
Reset
Close

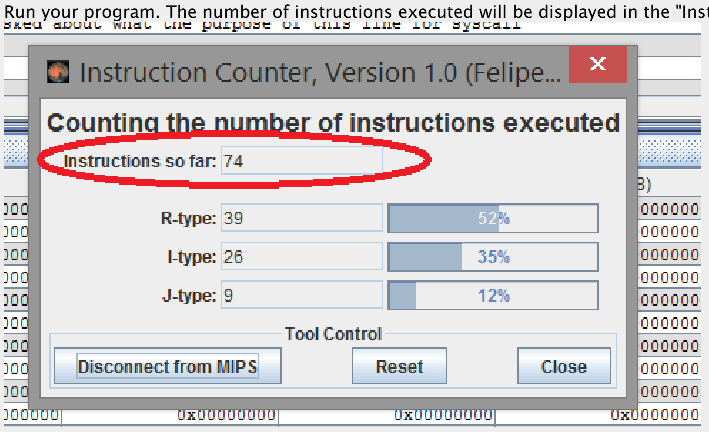
Instructions so far: 0

R-type: 0 0%

I-type: 0 0%

J-type: 0 0%

3. Run your program. The number of instructions executed will be displayed in the "Instructions so far" box.


Instruction Counter, Version 1.0 (Felipe...
Counting the number of instructions executed
Tool Control
Disconnect from MIPS
Reset
Close

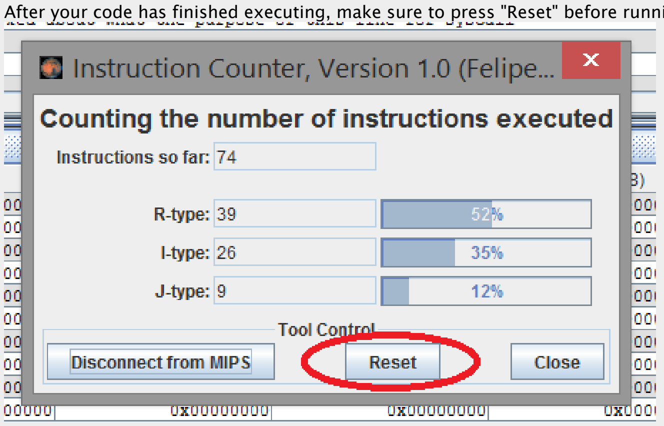
Instructions so far: 74

R-type: 39 52%

I-type: 26 35%

J-type: 9 12%

4. After your code has finished executing, make sure to press "Reset" before running again.


Instruction Counter, Version 1.0 (Felipe...
Counting the number of instructions executed
Tool Control
Disconnect from MIPS
Reset
Close

Instructions so far: 74

R-type: 39 52%

I-type: 26 35%

J-type: 9 12%

Run your code with a few inputs (preferably including something with stopping time > 10). Using the instruction counter, answer the following question in `proj1_2A.txt` using 1-2 sentences:

1. Which version (iterative or recursive) executed more instructions, and why? (Hint: What do you have to do in the recursive version that you don't need to in the iterative version?)

Task B (Due 2/23): Your second task will be to implement the sparse to dense matrix and print dense functionality.

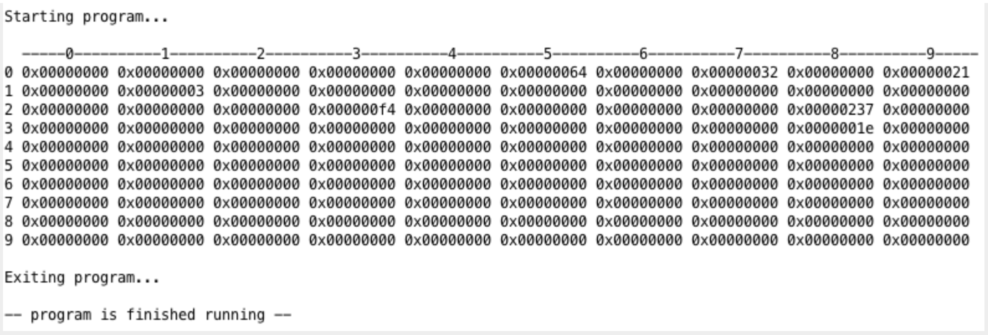
There are two subtasks in Task B: the `sparse2dense` function (inside `s2d.s`) and the `print_dense` function (inside `s2d_print_dense.s`).

For `sparse2dense`, you are provided with the pointer to the first `Row` of the sparse matrix linked list, the pointer to the dense matrix, and the width of the dense matrix. Your job is to iterate through each `Elem` of the sparse matrix and store its value in the appropriate position inside the dense matrix, in **row major**.

For `print_dense`, you are given the pointer to the dense matrix, the matrix width, and the matrix height. You will iterate through each element inside the dense matrix and print out its value, in a specific format.
Here is the required format for printing the dense matrix:

1. The first line of the output from `print_dense` should be the string `head`, provided to you on top of `s2d_print_dense.s` (due to some synchronization issue, we will accept either 4 or 5 dashes at the end of the first line).
2. The row number should be printed before each row, followed by a space character.
3. Each element in the dense matrix should be printed in hexidecimal.
4. There should be **one** space character between each element in the matrix.
5. There can be zero or one space character at the end of each line.

A sample output of a 10x10 dense matrix generated from the sparse matrix provide is as follow:



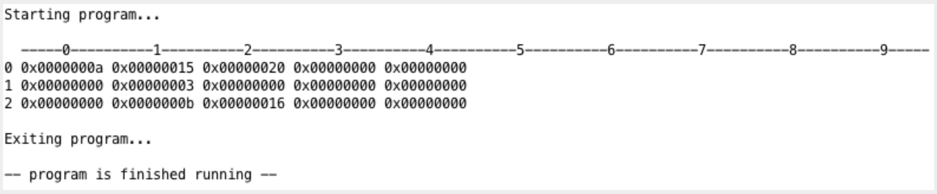
```
Starting program...

  -----0----------1----------2----------3----------4----------5----------6----------7----------8----------9-----
0 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000064 0x00000000 0x00000032 0x00000000 0x00000021
1 0x00000000 0x00000003 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
2 0x00000000 0x00000000 0x00000000 0x000000f4 0x00000000 0x00000000 0x00000000 0x00000000 0x00000237 0x00000000
3 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x0000001e 0x00000000
4 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
5 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
6 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
7 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
8 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
9 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000

Exiting program...

-- program is finished running --
```

A sample output of a 5x3 dense matrix generated from the sparse matrix provide is as follow:



```
Starting program...

  -----0----------1----------2----------3----------4----------5----------6----------7----------8----------9-----
0 0x0000000a 0x00000015 0x00000020 0x00000000 0x00000000
1 0x00000000 0x00000003 0x00000000 0x00000000 0x00000000
2 0x00000000 0x0000000b 0x00000016 0x00000000 0x00000000

Exiting program...

-- program is finished running --
```

Here are some tips and reminders:

- `s2d_main.s` is the only file you need to run to start the program, as it is the entrance point.
- You are provided with the dense matrix (in the default file: 10 x 10; note this size may change so do not hard-code the value in your code) already, with all 0 values.
- In `s2d_main.s`, several print functions are already defined for you. You may use those to your will.
- Each element in the matrix is a word/integer (32 bits; 4 bytes).
- The sparse matrix only stores elements with non-zero values.
- Make sure to handle you prologues and epilogues (i.e. stack memory management), as many bugs can stem from the fact that `$ra` was not stored properly and got overwritten.
- You may want to start by implementing the `print_dense` function first, as it will allow you to visualize your dense matrix.
- Make sure the output of your `print_dense` matches the suggested output above, or the autograder may mark you wrong.
- Make sure to test your code against other sparse matrix. You can generate your own sparse matrix by following the format in `s2d_sparse_matrix_data.s`.
- NOTE on 02/19: **DO NOT** assume the matrix is 10x10. The spec was written using 10x10 matrix to give you a concrete example. Your functions **should** be able to handle all sizes of matrices, since the necessary size information (e.g. width, height) are given in the function arguments.
- NOTE on 02/19: For `print_dense`, you may simply print out `head` as the first line, regardless of the width of the matrix.

Debugging and Testing

For task A, we have included the function `print_value` (located in `collatz_common.s`). It will print the value in `$a0`, followed by a space. The registers `$a0` and `$v0` will be clobbered by the function, so if it contains anything important, make sure to save it first!

For task B, it may be helpful to first be able to print out the dense matrix. Then, you may want to try accessing each element in the sparse matrix. Lastly, you can take each element and store them in the corresponding position in the dense matrix.

Setting breakpoints is also a good way to see the details of each register. You may find your bugs that way.

Again, code that does not assemble will AUTOMATICALLY RECEIVE A ZERO FOR THAT SECTION. For the `.s` files, you need not test on the hive machines, but you definitely should make sure that MARS will run your file with no errors.

Submission/Early Submission Incentive

We are offering an early submission incentive again. We will give one bonus point for anyone that submits a working `collatz_iterative.s` by **Thursday night (2/20/14 11:59 PM).** As this is for bonus points, slip days will not apply.

To submit for the early bonus, enter in the following command. You will only need to turn in `collatz_iterative.s`.

```
$ cd ~/proj1-2
$ submit proj1-2-early
```

The full proj1-2 is due Sunday. To submit the full proj1-2, enter in the following. You should be turning in `collatz_iterative.s`, `collatz_recursive.s`, `proj1_2A.txt`, `s2d.s`, and `s2d_print_dense.s`.

```
$ cd ~/proj1-2
$ submit proj1-2
```