"Beauty is more important in computing than anywhere else in technology because software is so complicated. Beauty is the ultimate defense against complexity. ... The geniuses of the computer field, on the the other hand, are the people with the keenest aesthetic senses, the ones who are capable of creating beauty. Beauty is decisive at every level: the most important interfaces, the most important programming languages, the winning algorithms are the beautiful ones." – D. Gelernter ('Machine Beauty', Basic Books, 1998)

"The road to wisdom? Well, it's plain and simple to express: Err and err and err again, but less and less and less." – Piet Hein

"The key to understanding recursion is to begin by understanding recursion. The rest is easy." – Koenig/Moo, Accelerated C++

Learning Objectives

1. To manage pointers and data structures in MIPS assembly
2. To use function calls and recursion in MIPS assembly

Work that needs to be handed in (via SVN)

First deadline

1. `draw_line.s`: implement the `draw_line` function in MIPS. Run with:
   `QtSpim -file common.s draw_line_main.s draw_line.s lab8_given.s`

Second deadline

1. `flood_fill.s`: implement the `flood_fill` function in MIPS. Run with:
   `QtSpim -file common.s flood_fill_main.s flood_fill.s lab8_given.s`
2. `count_disjoint_regions_step.s`: implement the `count_disjoint_regions_step` function in MIPS. Run with:
   `QtSpim -file common.s count_disjoint_regions_step_main.s count_disjoint_regions_step.s\`
   `flood_fill.s lab8_given.s`
3. `count_disjoint_regions.s`: implement the `count_disjoint_regions` function in MIPS. Run with:
   `QtSpim -file common.s count_disjoint_regions_main.s count_disjoint_regions.s\`
   `draw_line.s count_disjoint_regions_step.s flood_fill.s lab8_given.s`

Important!

Unlike previous labs, this is a solo lab and you may not work with anyone else on this lab.

We highly encourage you to start working on Part 2 during the lab section.

We also cannot stress enough the importance of reading the *entire* lab handout.

Rules

- Rules are the same as Lab 7.
- You'll find the assignment *so much easier* if you try to understand the C++ code you're translating before starting.
- Just as in Lab 7, follow all calling and register-saving conventions you've learned. It's even more important in this lab.
- Don't try to change the algorithms at this point; just write the code in MIPS as closely to the provided C++ code as possible.

We're helpful!

We've provided the C++ code we want you to rewrite. It's all in `lab8.cpp` in your SVN for your reference.

We've provided some print functions, in `common.s` and `lab8_given.s`. Use them to help you debug!

Structs

What is a Struct?

In this lab, you will utilize structs to represent canvas and lines. A `struct` is a high-level language tool, a way to reference an aggregation of data with just a single variable or pointer. Physically, a struct is a region of memory big enough to hold all the members inside it, like with an array, except that the elements can be all different sizes and types.

One useful note about `structs` in C++: how do you access their members? Say we have the following struct and variable declaration:

```
struct Node {
    int node_id;
    struct Node **children;
};
Node mynode;
```

If we want the `node_id` element of `mynode`, we reference `mynode.node_id`. Same thing if we wanted the assignments. But what if all we had was a *pointer* to `mynode`?

```
Node *myptr = &mynode;
```

We could always just dereference the pointer before accessing a member: `(*myptr).node_id`. But C++ provides a shorthand for this: `myptr->node_id`. You'll see a lot of this arrow operator in the C++ code for this lab.

Node Struct

We will use the Node struct to represent the graph. These struct is declared as follows:

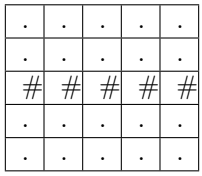
```
struct Node {
    int node_id;
    struct Node **children;
};
```

The Node struct contains two member variables: an integer `node_id` and a pointer to a null terminated array of node pointers `children`. That is, the second element in the node struct is word that represents the address of another place in memory, where an array of node pointers is stored. To know when you have reached the end of the array, the last element of the array will be a NULL pointer. Each element of the array is a Node pointer which points to a place in memory where the struct is allocated. If you're confused about this, treat the second element of the Node struct as an array of integers. Those integers just happen to be addresses that point to nodes.

Part 1 [20 points]

After learning more advanced painting skills, you need a better canvas to make the masterpiece. You choose the Canvas struct, which has four member variables: two integers for the height and width of the canvas, a char for the pattern to draw, and an array of char pointers where each char pointer represents a row and each char represents a single pixel on the canvas.

Below is an example of a 5×5 canvas, with a single line using the pattern #. We use . to represent empty pixel.



Draw Lines On Canvas [20 points]

To get familar with this new canvas, you will practice the basic skill: darwing a line on the canvas. Given the start position and the end position of a horizontal or vertial line, you need to set each pixel along the line to the pattern by storing the pattern value into the char pointer array. As in Lab 7, the pixels are labeled in row-major order from 0 to `height` × `width` - 1.

```
struct Canvas {
    // Height and width of the canvas.
    unsigned int height;
    unsigned int width;
    // The pattern to draw on the canvas.
    unsigned char pattern;
    // Each char* is null-terminated and has same length.
    char** canvas;
};

// Draw a line on canvas from start_pos to end_pos.
// start_pos will always be smaller than end_pos.
void draw_line(unsigned int start_pos, unsigned int end_pos, Canvas* canvas) {
    unsigned int width = canvas->width;
    unsigned int step_size = 1;
    // Check if the line is vertical.
    if (end_pos - start_pos >= width) {
        step_size = width;
    }
    // Update the canvas with the new line.
    for (int pos = start_pos; pos != end_pos + step_size; pos += step_size) {
        canvas->canvas[pos / width][pos % width] = canvas->pattern;
    }
}
```

The C++ code and a test case for `draw_line` function are in the file named `Lab8.cpp`. You have to write a MIPS translation of this function in `draw_line.s`. There are some test cases in the assembly file named `draw_line_main.s`, and you can compile and run the C++ code and compare its output to your MIPS code to verify its correctness. Feel free to add more test cases to both the C++ and `draw_line_main.s`.

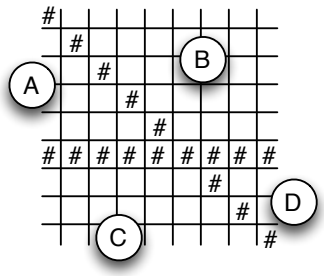
It may seem tricky that this function calls many other functions. But as long as you set up each function call correctly, you shouldn't run into any major problems. Just make sure to follow the calling conventions just like with any other function call, otherwise strange things will happen. We recommend learning how to use 'callee saved registers' (the `$s` registers), as you will likely find that they simplify the code. We've provided a number of video examples on using `$s` registers!

Learn and use S registers! They will prevent many years from being shaved off your life expectancy due to stress.

This function has arrays allocated on the stack! This may seem scary to deal with, but it's not too different from accessing arrays in the data segment.[1] You simply need to subtract enough stack space to fit the arrays, and then index the stack accordingly. For example, let's say I were writing a function where I needed to allocate array `int A[4]` on the stack, and wanted to set each number to 0. Then I would write:

```
sub         $sp, $sp, 16     # 16 bytes for four words!
move        $t0, $sp         # $t0 = &A[0]

# "normal" stack stuff
sub         $sp, $sp, _
sw          $ra, 0($sp)
# ...

sw          $0, 0($t0)       # A[0] = 0
sw          $0, 4($t0)       # A[1] = 0
sw          $0, 8($t0)       # A[2] = 0
sw          $0, 12($t0)      # A[3] = 0

# ... (normal function stuff)

# unroll the stack
lw          $ra, 0($sp)
# ...
add         $sp, $sp, _
add         $sp, $sp, 16     # Don't forget to add those A[4] bytes back!
jr          $ra
```

[1] Don't use `la` for stack arrays, though. If you're using it, you're doing something wrong.



Part 2 [80 points]

As the complexity of your paintings increases, you are looking for a way to keep track of the empty regions on your canvas. An empty region is a set of connected pixels that are not yet set to the pattern. Two pixels are connected if they share an edge. Two empty regions are disjoint if none of the pixels from one region is connected with any pixel from the other region. That is, two empty regions are disjoint if there is no path through connected edges from one region to the other that doesn't pass through a pixel marked with the pattern. In the example image above, there are four disjoint regions if we consider the '#' character as the pattern. Your want to measure the number of disjoint empty regions on your canvas.

Flood Fill Algorithm [30 points]

To make sure each disjoint empty region is only counted once, you mark each empty region using the **flood fill algorithm**. The algorithm starts with marking an empty pixel and recursively marking all the connected pixels.

```
// Mark an empty region as visited on the canvas using flood fill algorithm.
void flood_fill(int row, int col, unsigned char marker, Canvas* canvas) {
    // Check the current position is valid.
    if (row < 0 || col < 0 || row >= canvas->height || col >= canvas->width) {
        return;
    }
    unsigned char curr = canvas->canvas[row][col];
    if (curr != canvas->pattern && curr != marker) {
        // Mark the current pos as visited.
        canvas->canvas[row][col] = marker;
        // Flood fill four neighbors.
        flood_fill(row - 1, col, marker, canvas);
        flood_fill(row, col + 1, marker, canvas);
        flood_fill(row + 1, col, marker, canvas);
        flood_fill(row, col - 1, marker, canvas);
    }
}
```

Note that this function is recursive. This may seem like a tricky thing to deal with, but in actuality, a recursive function call isn't really any different than a normal function call. Just make sure to follow the calling conventions just like with any other function call, otherwise strange things will happen. Also, we recommend learning how to use 'callee saved registers' (the `$s` registers), as you will likely find that they simplify the code. We've provided a number of video examples on using `$s` registers!

We've provided some test cases in `flood_fill_main.s` and their C++ versions in `Lab8.cpp`. You should read the given test cases, make sure you understand them, and then run your code against them to check its correctness. You're also heavily encouraged to add some more test cases, because we will when autograding. You can do so by adding them to `Lab8.cpp` to see what the correct output should be. Then you can add them to `flood_fill_main.s` to see if your MIPS does the same.

Count Disjoint Regions on the Canvas at a Given Step [30 points]

This step function iterates through each pixel on the canvas in row-major order and checks if a pixel is empty. If the pixel is empty, you find another empty region. You will increment the region counter and call the flood fill algorithm to mark the entire empty region as visited.

```
// Count the number of disjoint empty area in a given canvas.
unsigned int count_disjoint_regions_step(unsigned char marker,
                                         Canvas* canvas) {
    unsigned int region_count = 0;
    for (unsigned int row = 0; row < canvas->height; row++) {
        for (unsigned int col = 0; col < canvas->width; col++) {
            unsigned char curr_char = canvas->canvas[row][col];
            if (curr_char != canvas->pattern && curr_char != marker) {
                region_count ++;
                flood_fill(row, col, marker, canvas);
            }
        }
    }
    return region_count;
}
```

We've provided some test cases in `count_disjoint_regions_step_main.s` and their C++ versions in `Lab8.cpp`. You should read the given test cases, make sure you understand them, and then run your code against them to check its correctness. You're also heavily encouraged to add some more test cases, because we will when autograding. You can do so by adding them to `Lab8.cpp` to see what the correct output should be. Then you can add them to `count_disjoint_regions_step_main.s` to see if your MIPS does the same.

Count Disjoint Regions on the Canvas [20 points]

Now you will use functions implemented above to track the number of disjoint empty regions after drawing each line on the canvas.

The Lines struct has two merber variables: an integer indicating the number of lines and an integer pointer array of size 2 storing a sequence of start positions and end positions. You will draw each line in the Lines struct in the given order using the draw_line function implemented in Part 1. After drawing each line, you will call the count_disjoint_region_step function to count the number of disjoint empty regions. The region count will be stored in counts field of the Solution struct.

```
struct Lines {
    unsigned int num_lines;
    // An int* array of size 2, where first element is an array of start pos
    // and second element is an array of end pos for each line.
    unsigned int* coords[2];
};

struct Solution {
    unsigned int length;
    // Number of disjoint regions after drawing each line.
    int* counts;
};

void count_disjoint_regions(const Lines* lines, Canvas* canvas,
                            Solution* solution) {
    // Iterate through each step.
    for (unsigned int i = 0; i < lines->num_lines; i++) {
        unsigned int start_pos = lines->coords[0][i];
        unsigned int end_pos = lines->coords[1][i];
        // Draw line on canvas.
        draw_line(start_pos, end_pos, canvas);
        // Run flood fill algorithm on the updated canvas.
        // In each even iteration, fill with marker 'A', otherwise use 'B'.
        unsigned int count = count_disjoint_regions_step('A' + (i % 2), canvas);
        // Update the solution struct. Memory for counts is preallocated.
        solution->counts[i] = count;
    }
}
```

The C++ code and a test case for `count_disjoint_regions` function are in the file named `Lab8.cpp`. You have to write a MIPS translation of this function in `count_disjoint_regions.s`. There are some test cases in the assembly file named `count_disjoint_regions_main.s`, and you can compile and run the C++ code and compare its output to your MIPS code to verify its correctness. Feel free to add more test cases to both the C++ and `count_disjoint_regions_main.s`.