Group: Logan Yuan (lyuan04), Dylan Perkins (dperki01)

1. How will data in the program be represented and interconnected?
2. How will the program logic be organized, and how will the computation be done?

Restoration Architecture

- Each line is read and parsed into their character infusion and integer arrays. For example, a12b13c13 is split into "abc" and [12,13,13]. This will be placed into a struct, and a Hanson Sequence will be used to store all of the structs (representing each line's char infusion sequence and integer array).
- We will then find the *repeat* char infusion sequence. This will represent any *valid* row that we want to keep.
    - We create a Hanson Set containing char*
    - We will iterate through every element in the original list
    - We will check if struct->charInfusion is in the set. If it's not, add the string. If it is, keep track of the charInfusion code.
- Read through the original list and compare each line->charInfusion code to *our* charInfusion code. If they align, add it to our new Hanson Sequence (containing just the arr[] codes for the line).
- Print the pixels to standard output, along with their proper heading.
- Free all memory allocated

High level Hanson data structures:

- Hanson Sequence: Containing structs representing each line's characters and integers (void * will point to instances* of our struct). A second sequence will also be used to keep track of the final pixel grid. The list will be of *int[]. Our commands will be Seq_new, Seq_addhi, and Seq_get. We will also free using Seq_free.
- Hanson Set: This will contain char* and will be used to check for repeat sequences. We will use Set_new, Set_put, and Set_member. We will free using Set_free.

Implementation plan

- [The first bullet points given to us in the spec]
- Create readaline() function, making sure to adhere to all style guidelines, and have it output an int to terminal to test that the function works - 5 minutes

- Add the ability for readaline() to accept file input, and test that it does read the opened file (print file contents to terminal) - 5 minutes
- Have readaline() stop at '\n' or EOF (test by printing a single line to console) - 5 minutes
- Count the number of bits in the inputted line. It is important to remember that the inputted files will be plain pgms or P2s. In this form, each char takes up a byte's worth of space, so all the ASCII numerals are added to the sum as the total number of digits in the ASCII numeral. In addition, there will be ' \n' which take up a specific amount of space. Also, C 'strings' have an '\0' at the end of them, and we will have to account for that as well. (count manually the number of chars in a file and print the function results, see if they match up) - 20 minutes
- Make sure that readaline throws a CRE when the spec specifies that it should. Read the exceptions doc again, and raise all exceptions required in the spec - 15 minutes
- Create a new file to hold the code surrounding the large Hanson list that will be used to hold all the lines in the pgm file - 5 minutes
- Add to the file all necessary libraries, the readaline file, and a struct containing two members: a pointer to the char * array containing the given line, and a void * pointer that will be set to each line's character infusion - 10 minutes
- Create a function that will take in a file name, and output a pointer to the first member of a list of all the lines in the file. But for now have it take standard input, and use readaline to read and return the inputted string. This tests that readaline was added correctly and sets up the main function for this file. - 15 minutes
- Have the function take in a file when called, and make sure the file opens correctly - 10 minutes
- Add Hansons list to the file that contains the struct created earlier - 5 minutes
- Create a while loop, and each time it is called a new member of the list is added, the address of it's first member is set to the pointer to the newest line inputted to readaline() - 10 minutes
- While testing this section make sure to free all the memory you use - 15 minutes
- Test first with many small self created files that this function works, by comparing the file to the output from the function - 15 minutes
- Now make the function add any char that is not representing a ASCII character ( so all letters and random symbols ) to a char * array, and set the the address of the second member of the struct to the first element of that char * array - 30 minutes
- Test the above code heavily, now adding in more inputted test files, this time containing char infusions - 20 minutes
- create a new file that will contain the set that will be used to check for the right infusion -5 min
- now create the set, and create a function that will add the infusions from each struct in the list to this set - 5 min

- For loop through all lines, and add scramble. Before each addition, check if that scramble already exists in the list. If it's already contained, return the char* infusion sequence. -10min
- Free set - 5min
- Create another Hanson list, where each index is the row number for a correct, original row of pixels. -2min
- Loop through the original Hanson list of Lines, find char* infusions that match our char* code. Add int[] to new List Pixels. -10min
- test that on a number of inputs including lines with different scrambles of infused characters, split between numbers in different ways, with different numbers.
- Output to standard output by printing the header, then iterating through the new Sequence and printing its pixels
- free all memory