

Ethan Yu
COMP IV: Project Portfolio
Spring 2021

Contents:

1.	PS0	Hello World with SFML	·	·	·	·	·	·	4
1.1.		Overview	·	·	·	·	·	·	4
1.2.		Key Concepts and Lessons Learned	·	·	·	·	·	·	4
1.3.		main.cpp	·	·	·	·	·	·	4-5
1.4.		Output	·	·	·	·	·	·	6
2.	PS1a	Linear Feedback Shift Register (LFSR) and Image Encoding (Part A)							7
2.1.		Overview	·	·	·	·	·	·	7
2.2.		Key Concepts and Lessons Learned	·	·	·	·	·	·	7
2.3.		Makefile	·	·	·	·	·	·	8
2.4.		test.cpp	·	·	·	·	·	·	8-9
2.5.		FibLFSR.h	·	·	·	·	·	·	9-10
2.6.		FibLFSR.cpp	·	·	·	·	·	·	10-12
2.7.		Output	·	·	·	·	·	·	12
3.	PS1b	Linear Feedback Shift Register (LFSR) and Image Encoding (Part B)							13
3.1.		Overview	·	·	·	·	·	·	13
3.2.		Key Concepts and Lessons Learned	·	·	·	·	·	·	13
3.3.		Makefile	·	·	·	·	·	·	13-14
3.4.		PhotoMagic.cpp	·	·	·	·	·	·	14-16
3.5.		FibLFSR.h	·	·	·	·	·	·	16-17
3.6.		FibLFSR.cpp	·	·	·	·	·	·	17-19

	3.7.	Output	·	·	·	·	·	·	·	19
4.	PS2a	N-Body Simulation (Part A)	·	·	·	·	·	·	·	20
	4.1.	Overview	·	·	·	·	·	·	·	20
	4.2.	Key Concepts and Lessons Learned	·	·	·	·	·	·	·	20
	4.3.	Makefile	·	·	·	·	·	·	·	20
	4.4.	main.cpp	·	·	·	·	·	·	·	21-22
	4.5.	NBody.h	·	·	·	·	·	·	·	22-24
	4.6.	NBody.cpp	·	·	·	·	·	·	·	24-25
	4.7.	planets.txt	·	·	·	·	·	·	·	25-26
	4.8.	Output	·	·	·	·	·	·	·	26
5.	PS2b	N-Body Simulation (Part B)	·	·	·	·	·	·	·	27
	5.1.	Overview	·	·	·	·	·	·	·	27
	5.2.	Key Concepts and Lessons Learned	·	·	·	·	·	·	·	27
	5.3.	Makefile	·	·	·	·	·	·	·	27-28
	5.4.	main.cpp	·	·	·	·	·	·	·	28-31
	5.5.	NBody.h	·	·	·	·	·	·	·	31-34
	5.6.	NBody.cpp	·	·	·	·	·	·	·	34-37
	5.7.	planets.txt	·	·	·	·	·	·	·	37
	5.8.	Output	·	·	·	·	·	·	·	37
6.	PS3	DNA Sequence Alignment	·	·	·	·	·	·	·	38
	6.1.	Overview	·	·	·	·	·	·	·	38
	6.2.	Key Concepts and Lessons Learned	·	·	·	·	·	·	·	38
	6.3.	Makefile	·	·	·	·	·	·	·	38-39
	6.4.	main.cpp	·	·	·	·	·	·	·	39-40
	6.5.	ps3.h	·	·	·	·	·	·	·	40
	6.6.	ps3.cpp	·	·	·	·	·	·	·	40-43
	6.7.	Output	·	·	·	·	·	·	·	44
7.	PS4a	Synthesizing a Plucked String Sound (Part A)	·	·	·	·	·	·	·	45
	7.1.	Overview	·	·	·	·	·	·	·	45
	7.2.	Key Concepts and Lessons Learned	·	·	·	·	·	·	·	45
	7.3.	Makefile	·	·	·	·	·	·	·	45

	7.4.	test.cpp	·	·	·	·	·	·	·	45-46
	7.5.	CircularBuffer.h	·	·	·	·	·	·	·	46-47
	7.6.	CircularBuffer.cpp	·	·	·	·	·	·	·	47-48
	7.7.	Output	·	·	·	·	·	·	·	48
8.	PS4b	Synthesizing a Plucked String Sound (Part B)	·	·	·	·	·	·	·	49
	8.1.	Overview	·	·	·	·	·	·	·	49
	8.2.	Key Concepts and Lessons Learned	·	·	·	·	·	·	·	49
	8.3.	Makefile	·	·	·	·	·	·	·	49
	8.4.	KSGuitarSim.cpp	·	·	·	·	·	·	·	50-52
	8.5.	CircularBuffer.h	·	·	·	·	·	·	·	52
	8.6.	StringSound.h	·	·	·	·	·	·	·	53
	8.7.	CircularBuffer.cpp	·	·	·	·	·	·	·	53-54
	8.8.	StringSound.cpp	·	·	·	·	·	·	·	55-56
	8.9.	Output	·	·	·	·	·	·	·	56
9.	PS5	Recursive Graphics	·	·	·	·	·	·	·	57
	9.1.	Overview	·	·	·	·	·	·	·	57
	9.2.	Key Concepts and Lessons Learned	·	·	·	·	·	·	·	57
	9.3.	Makefile	·	·	·	·	·	·	·	57
	9.4.	TFractal.cpp	·	·	·	·	·	·	·	58-59
	9.5.	Triangle.h	·	·	·	·	·	·	·	59-60
	9.6.	Triangle.cpp	·	·	·	·	·	·	·	60-61
	9.7.	Output	·	·	·	·	·	·	·	61
10.	PS6	Kronos Time Clock	·	·	·	·	·	·	·	62
	10.1.	Overview	·	·	·	·	·	·	·	62
	10.2.	Key Concepts and Lessons Learned	·	·	·	·	·	·	·	62
	10.3.	Makefile	·	·	·	·	·	·	·	62
	10.4.	main.cpp	·	·	·	·	·	·	·	63-65
	10.5.	Output	·	·	·	·	·	·	·	65

Time to complete: 12 hours

PS0 Hello World with SFML

Overview -

In this assignment, we learned the basics of the Simple and Fast Multimedia Library (SFML) and got familiar with how to load, display, and modify sprites in a window.

The assignment was to get your virtual machine set up and make a SFML program that would draw a sprite, move it with user inputs, and do some other feature. I made my program so it would move the sprite using the arrow keys and expand or shrink the sprite if the Z or X keys were pressed respectively.

Key Concepts and Lessons Learned -

Important aspects from this assignment included how to work with the `RenderWindow`, `Texture`, `Sprite`, `Event`, `Keyboard`, and `Vector2f` SFML classes. Using these classes, we could create a window to draw sprites in, listen for and take in user events, and store coordinates.

I had not worked with the SFML library before this so this was all new to me.

Code files in this assignment: **main.cpp**

main.cpp

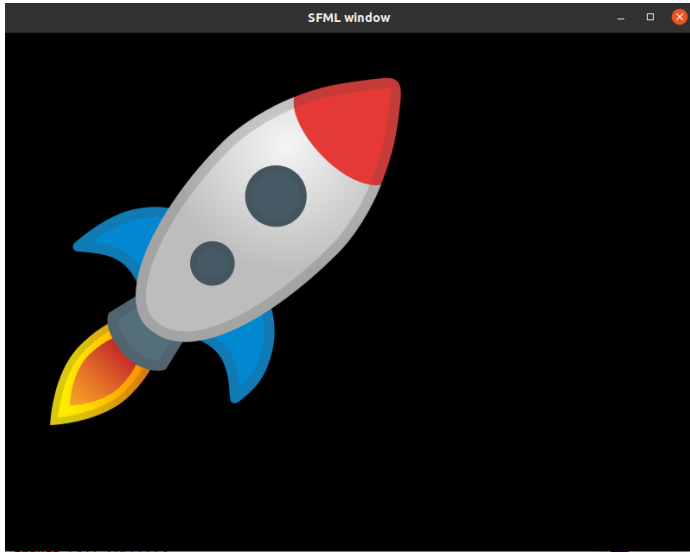
```
1. //Author: Ethan Yu
2. #include <SFML/Graphics.hpp>
3. int main()
4. {
5.     // Create the main window
6.     sf::RenderWindow window(sf::VideoMode(800, 600), "SFML window");
7.     // Load a sprite to display
8.     sf::Texture texture;
9.     if (!texture.loadFromFile("./sprite.png"))
10.         return EXIT_FAILURE;
11.     sf::Sprite sprite(texture);
12.
13.     // Start the game loop
14.     while (window.isOpen())
15.     {
16.         // Process events
17.         sf::Event event;
18.         while (window.pollEvent(event))
19.         {
20.             // Close window: exit
21.             if (event.type == sf::Event::Closed)
22.                 window.close();
23.         }
24.     }
```

```

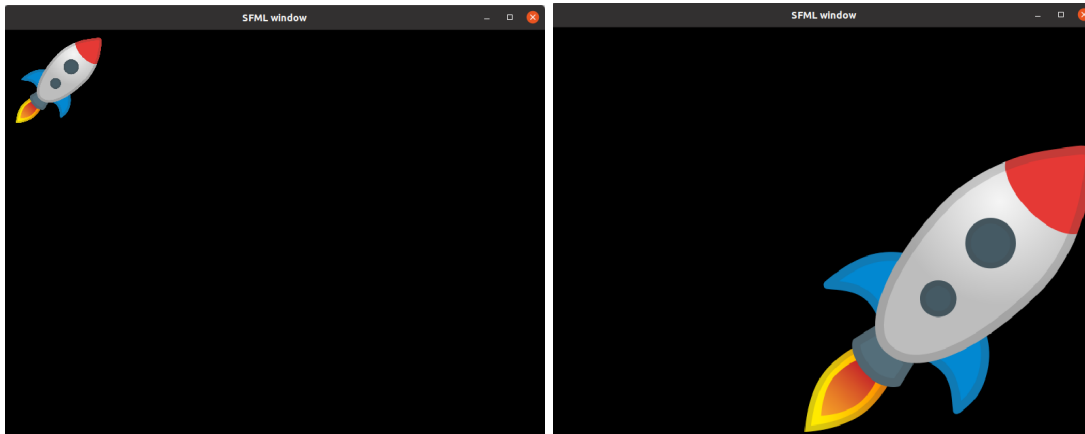
25.         //Vector2 objects used to adjust sprite properties
26.         sf::Vector2f newPosition = sprite.getPosition();;
27.         sf::Vector2f newScale = sprite.getScale();
28.
29.         // Use the arrow keys to move the sprite
30.         if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up))
31.         {
32.             newPosition.y = newPosition.y - 5;
33.         }
34.         else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down))
35.         {
36.             newPosition.y = newPosition.y + 5;
37.         }
38.         else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left))
39.         {
40.             newPosition.x = newPosition.x - 5;
41.         }
42.         else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right))
43.         {
44.             newPosition.x = newPosition.x + 5;
45.         }
46.
47.         // Use the Z key to expand the sprite, X key to contract it
48.         if (sf::Keyboard::isKeyPressed(sf::Keyboard::Z))
49.         {
50.             newScale.x = newScale.x * 1.01f;
51.             newScale.y = newScale.y * 1.01f;
52.         }
53.         else if (sf::Keyboard::isKeyPressed(sf::Keyboard::X))
54.         {
55.             newScale.x = newScale.x * .99f;
56.             newScale.y = newScale.y * .99f;
57.         }
58.
59.         sprite.setPosition(newPosition.x, newPosition.y);
60.         sprite.setScale(newScale);
61.
62.         // Clear screen
63.         window.clear();
64.         // Draw the sprite
65.         window.draw(sprite);
66.         // Update the window
67.         window.display();
68.     }
69.     return EXIT_SUCCESS;
70. }

```

Output -



Above: The original starting size and position of the sprite when drawn in the window.



Left: The sprite is moved to the top left corner using arrow keys and shrunk with X.

Right: The sprite is moved to the bottom right corner using arrow keys and expanded with Z.

PS1a Linear Feedback Shift Register (LFSR) and Image Encoding (Part A)

Overview -

In this assignment, the task was to simulate a linear feedback shift register (LFSR) that would generate a new bit by XOR'ing the leftmost bit and the tap bits and append that to the end of the register after it is left-shifted. The shifting and appending happened in the `step()` function and the `generate(n)` function just calls the `step()` function `n` times.

I also made an error exception class called `FibLFSRException` for `FibLFSR.cpp` to throw if something happened when making a `FibLFSR` object (its main purpose is to check if the seed supplied is valid).

The output operator `<<` is also overloaded for `FibLFSR` objects so it will display the contents of the register and the most recent bit added.

Key Concepts and Lessons Learned -

One key concept and something I learned was how to use the Boost Library for unit testing. Using the Boost Library for unit testing allows you to test parts of your program and set parameters as to what you expect the program to do and if it doesn't do that, then you know where it went wrong. It allows for arguably easier debugging because you don't need to set breakpoints in multiple places or put print statements in places to track progress.

Another key concept was how to make Makefiles. These allow you to compile and create an executable from files using a single command. This eliminates the need to type compilation or deletion commands individually for each file so now you can just use the "make" or "make clean" commands to do each of those tasks respectively. Before this assignment I did not know how to make Makefiles but after this I was able to learn how to make and use them.

Code files in this assignment: **Makefile**, **test.cpp**, **FibLFSR.h**, **FibLFSR.cpp**

Makefile:

To make the `ps1a` executable, you need the `FibLFSR.o` and `test.o` object files. One thing to note is to make the `test.o` file, the `-lboost_unit_test_framework` (which I used the `BOOST_ARG` label for) is needed for it to compile because we use Boost library functions in `test.cpp`

```
1. CC = g++
2. CFLAGS = -Wall -Werror -std=c++11 -pedantic
3. BOOST_ARG = -lboost_unit_test_framework
4. OBJ = FibLFSR.o test.o
5. EXE = ps1a
6.
7. all: ps1a
8.
9. FibLFSR.o: FibLFSR.cpp FibLFSR.h
10. $(CC) $(CFLAGS) -c FibLFSR.cpp -o FibLFSR.o
11.
12. test.o: test.cpp
13. $(CC) $(CFLAGS) -c test.cpp -o test.o $(BOOST_ARG)
14.
15. ps1a: $(OBJ)
16. $(CC) $(CFLAGS) -o $(EXE) $(OBJ)
17.
18. clean:
19. rm -f $(EXE) $(OBJ) *.o
20.
```

test.cpp:

The first `BOOST_AUTO_TEST_CASE` was provided which checks if the program will check to see if the bits were pushed and calculated correctly and resulted in the right values. The second test tests if the seed given is 16 bits long. If not, then an exception is thrown and a `FibLFSRException` object will be made with a string member saying "Incorrect number of bits". The third test tests if the seed given contains any non-binary digits (any number besides 0 and 1). If so, then an exception is thrown and a `FibLFSRException` object will be made with a string member saying "Non-binary digit detected".

Since I used the `BOOST_CHECK_THROW()` function to check for these possibilities and I passed in bad seeds for the tests, then it will only pass these tests if the exceptions are thrown.

```
1. #include <iostream>
2. #include <string>
3.
```



```

4.  #include "FibLFSR.h"
5.
6.  #define BOOST_TEST_DYN_LINK
7.  #define BOOST_TEST_MODULE Main
8.  #include <boost/test/unit_test.hpp>
9.
10. BOOST_AUTO_TEST_CASE(sixteenBitsThreeTaps)
11. {
12.     FibLFSR l("1011011000110110");
13.     BOOST_REQUIRE(l.step() == 0);
14.     BOOST_REQUIRE(l.step() == 0);
15.     BOOST_REQUIRE(l.step() == 0);
16.     BOOST_REQUIRE(l.step() == 1);
17.     BOOST_REQUIRE(l.step() == 1);
18.     BOOST_REQUIRE(l.step() == 0);
19.     BOOST_REQUIRE(l.step() == 0);
20.     BOOST_REQUIRE(l.step() == 1);
21.
22.     FibLFSR l2("1011011000110110");
23.     BOOST_REQUIRE(l2.generate(9) == 51);
24. }
25.
26. //Tests to see if the program will throw an exception when the seed given isn't 16 bits long
27. //If FibLFSR.cpp works correctly, then an exception will be thrown
28. BOOST_AUTO_TEST_CASE(fifteenBits)
29. {
30.     BOOST_CHECK_THROW(FibLFSR incorrectNumOfBits("1010101010"), FibLFSRException);
31. }
32.
33. //Tests to see if the program will throw an exception if the seed contains bits
34. //that are not 0 or 1
35. //If FibLFSR.cpp works correctly, then an exception will be thrown
36. BOOST_AUTO_TEST_CASE(sixteenBitsWithNonbinaryDigits)
37. {
38.     BOOST_CHECK_THROW(FibLFSR invalidBits("1234567890123456"), FibLFSRException);
39.
40. }
41.

```

FibLFSR.h:

I used a vector to represent the register bits because you can easily add to the back of it using the `.push_back()` function and you can erase the first bit using the `.erase()` function. You can also easily get its size using the `.size()` or `.length()` functions. I used this over a dynamically allocated array because a vector "shifts" automatically while an array would need to have its contents reassigned after every shift to match its new place in the register.

```

1.  #ifndef FIBLFSR_H
2.  #define FIBLFSR_H
3.
4.  #include <iostream>
5.  #include <string>

```

```

6.  #include <vector>
7.
8.  class FibLFSR {
9.  public:
10.     // constructor to create LFSR with the given initial seed
11.     FibLFSR(std::string seed);
12.
13.     // simulate one step and return the new bit as 0 or 1
14.     int step(void);
15.
16.     // simulate k steps and return k-bit integer
17.     int generate(int k);
18.
19.     //overloading the output operator
20.     friend std::ostream& operator<< (std::ostream& outputStream, const FibLFSR& fibLFSR);
21.
22. private:
23.     std::vector<int> bitVector; // what we put the seed into and use for step()
24.     int vectorLength;          // size of the vector
25.     int tapBits[4];             // these are the bits you XOR together
26.     int resultValue;           // the value returned after step() or generate()
27. };
28.
29. std::ostream& operator<< (std::ostream& outputStream, const FibLFSR& fibLFSR);
30.
31. class FibLFSRException
32. {
33. public:
34.     FibLFSRException(std::string message);
35.
36. private:
37.     std::string errorMessage;
38. };
39.
40. #endif
41.

```

FibLFSR.cpp

For the tap bits used for the XOR function, since I pushed in the values into the vector so the first bit in was actually the most significant bit and so on, the indexes needed to be flipped to access the correct value (so for example, the leftmost most significant bit would be at index [0] while the rightmost least significant bit would be at index [15] contrary to how we would count it in binary where it would be reversed).

As mentioned before, using a vector allowed me to use vector functions which made adding and removing elements easy as seen in the constructor and step() functions where push_back() and erase() functions came in handy.

```

1.  #include "FibLFSR.h"
2.

```

```

3.  const int ASCII_0 = 48;
4.  const int ASCII_1 = 49;
5.  const int NUM_BITS_WE_WANT = 16;
6.  const int LEFTMOST_BIT_POSITION = 0; //15
7.  const int TAP_BIT_POSITION_1 = 2; //13
8.  const int TAP_BIT_POSITION_2 = 3; //12
9.  const int TAP_BIT_POSITION_3 = 5; //10
10.
11. std::ostream& operator<< (std::ostream& outputStream, const FibLFSR& fibLFSR)
12. {
13.     for (int i = 0; i < fibLFSR.vectorLength; i++)
14.     {
15.         outputStream << fibLFSR.bitVector[i];
16.     }
17.
18.     outputStream << "\t" << fibLFSR.resultValue;
19.     return outputStream;
20. }
21.
22. FibLFSRException::FibLFSRException(std::string message)
23. {
24.     errorMessage = message;
25. }
26.
27. FibLFSR::FibLFSR(std::string seed)
28. {
29.     tapBits[0] = TAP_BIT_POSITION_1;
30.     tapBits[1] = TAP_BIT_POSITION_2;
31.     tapBits[2] = TAP_BIT_POSITION_3;
32.     tapBits[3] = LEFTMOST_BIT_POSITION;
33.
34.     resultValue = 0;
35.     vectorLength = seed.length();
36.
37.     if (vectorLength != NUM_BITS_WE_WANT)
38.     {
39.         throw FibLFSRException("Incorrect number of bits");
40.     }
41.
42.     for (int i = 0; i < vectorLength; i++)
43.     {
44.         if (seed[i] == ASCII_0 || seed[i] == ASCII_1)
45.         {
46.             //since seed[i] is a char, we need to adjust it to its int value
47.             bitVector.push_back(seed[i] - ASCII_0);
48.         }
49.         else
50.         {
51.             throw FibLFSRException("Non-binary digit detected");
52.         }
53.     }
54. }
55.
56. int FibLFSR::step(void)
57. {

```

```

58.         int xorResult = (bitVector[TAP_BIT_POSITION_1] ^ bitVector[TAP_BIT_POSITION_2]) ^
(bitVector[TAP_BIT_POSITION_3] ^ bitVector[LEFTMOST_BIT_POSITION]);
59.         bitVector.erase(bitVector.begin()); //Erase the first element of the bitVector
60.         bitVector.push_back(xorResult);      //Add the xor result to the back of the bitVector
61.         resultValue = xorResult;
62.
63.         return xorResult;
64.     }
65.
66. int FibLFSR::generate(int k)
67. {
68.     int resultInteger = 0;
69.     for (int i = 0; i < k; i++)
70.     {
71.         //left shift the bitVector then add the XOR bit to the end
72.         resultInteger = (resultInteger << 1) + this->step();
73.     }
74.     resultValue = resultInteger;
75.     return resultInteger;
76. }

```

Output -

```

ethan@ethan-VirtualBox:~/ps1a$ make
g++ -Wall -Werror -std=c++17 -pedantic -c FibLFSR.cpp -o FibLFSR.o
g++ -Wall -Werror -std=c++17 -pedantic -c test.cpp -o test.o -lboost_unit_test_framework
g++ -Wall -Werror -std=c++17 -pedantic -o ps1a FibLFSR.o test.o -lboost_unit_test_framework
ethan@ethan-VirtualBox:~/ps1a$ ls
FibLFSR.cpp  FibLFSR.o  ps1a          test.cpp
FibLFSR.h   Makefile  ps1a-readme.txt  test.o
ethan@ethan-VirtualBox:~/ps1a$ ./ps1a
Running 3 test cases...

*** No errors detected
ethan@ethan-VirtualBox:~/ps1a$

```

Above: Compiling the code and creating the executable using the Makefile and running the executable, resulting in no errors detected after the tests (all tests passed)

PS1b Linear Feedback Shift Register (LFSR) and Image Encoding (Part B)

Overview -

This assignment takes a source image and encrypts it using the given seed. This is done by taking 3 command-line arguments: the source image name, the output image name, and the seed to run the program.

The way the program works is the source image is taken, the colors are inverted, then the seed is used along with the remaining color data to be XOR'ed and generate a new color. Do this for all the pixels in the image and what comes out should be an image that looks like TV static. After that, you can decrypt it by putting the output image as the new source image and running the program with the same seed and the resulting image will be the original source image you gave.

Key Concepts and Lessons Learned -

While PS0 taught us how to draw sprites and manipulate its position and size, PS1b taught us how to manipulate the sprite itself by modifying the individual pixels in the sprite's image.

Code files in this assignment: **Makefile**, **PhotoMagic.cpp**, **FibLFSR.h**, **FibLFSR.cpp**

Makefile:

Since we have to work with a SFML window and display things in it, we need to include -lsfml-graphics -lsfml-window -lsfml-system (which I put into the SFML label) when compiling the PhotoMagic.o so it can use the SFML functions.

1. CC = g++
2. CFLAGS = -Wall -Werror -std=c++11 -pedantic
3. SFML = -lsfml-graphics -lsfml-window -lsfml-system
4. OBJ = FibLFSR.o PhotoMagic.o
- 5.
6. all: PhotoMagic
- 7.
8. PhotoMagic: PhotoMagic.o FibLFSR.o
9. \$(CC) \$(CFLAGS) -o PhotoMagic PhotoMagic.o FibLFSR.o \$(SFML)
- 10.
11. PhotoMagic.o: PhotoMagic.cpp FibLFSR.h
12. \$(CC) \$(CFLAGS) -c PhotoMagic.cpp -o PhotoMagic.o
- 13.
14. FibLFSR.o: FibLFSR.cpp FibLFSR.h

```

15. $(CC) $(CFLAGS) -c FibLFSR.cpp -o FibLFSR.o
16.
17. clean:
18. rm PhotoMagic *.o
19.

```

PhotoMagic.cpp:

When it comes to drawing the picture, we invert the colors of the starter picture to create its photonegative then pass it to the transform function. How transform() works is a seed is given by the user which is used to create “random” numbers by running it through generate(8). For each pixel, the red, green, and blue color components are taken and XOR’d with the random number and then set to the resulting new color. Once this is done for every pixel, the new image is displayed on screen and saved as a .png file with the name the user used as the third argument in the command line.

```

1. #include <SFML/System.hpp>
2. #include <SFML/Window.hpp>
3. #include <SFML/Graphics.hpp>
4. #include "FibLFSR.h"
5.
6. void transform(sf::Image& sourceImage, FibLFSR* seed);
7.
8. void transform(sf::Image& sourceImage, FibLFSR* seed)
9. {
10.     //get the size of the image
11.     sf::Vector2u size = sourceImage.getSize();
12.     unsigned int imageWidth = size.x;
13.     unsigned int imageHeight = size.y;
14.     unsigned int randomNum, redComponent, greenComponent, blueComponent, redXORResult,
15.     greenXORResult, blueXORResult;
16.     sf::Color color;
17.
18.     for (unsigned int i = 0; i < imageWidth; i++)
19.     {
20.         for (unsigned int j = 0; j < imageHeight; j++)
21.         {
22.             //extract red and XOR with random number
23.             color = sourceImage.getPixel(i, j);
24.             randomNum = seed->generate(8);
25.             redComponent = color.r;
26.             redXORResult = redComponent ^ randomNum;
27.
28.             //extract green and XOR with random number
29.             randomNum = seed->generate(8);
30.             greenComponent = color.g;
31.             greenXORResult = greenComponent ^ randomNum;
32.
33.             //extract blue and XOR with random number
34.             randomNum = seed->generate(8);

```

```

34.             blueComponent = color.b;
35.             blueXORResult = blueComponent ^ randomNum;
36.
37.             //set the pixel to the new color
38.             sourceImage.setPixel(i, j, sf::Color(redXORResult, greenXORResult, blueXORResult));
39.         }
40.     }
41. }
42.
43. int main(int argc, char* argv[])
44. {
45.     //arg[0] = name of executable, arg[1] = inputfile, arg[2] = outputFile, arg[3] = seed
46.     const std::string inputFile = argv[1];
47.     const std::string outputFile = argv[2];
48.
49.     sf::Image image;
50.     if (!image.loadFromFile(inputFile))
51.     {
52.         return -1;
53.     }
54.
55.     // p is for pixelimage.getPixel(x, y);
56.     sf::Color p;
57.
58.     // get the size (width, height) of the image
59.     sf::Vector2u size = image.getSize();
60.
61.     // make a FibLFSR object to pass into transform()
62.     FibLFSR seed(argv[3]);
63.
64.     // create photographic negative image
65.     for (unsigned int x = 0; x < size.x; x++)
66.     {
67.         for (unsigned int y = 0; y < size.y; y++)
68.         {
69.             p = image.getPixel(x, y);
70.             p.r = 255 - p.r;
71.             p.g = 255 - p.g;
72.             p.b = 255 - p.b;
73.             image.setPixel(x, y, p);
74.         }
75.     }
76.
77.     //encrypt the file using the seed
78.     transform(image, &seed);
79.
80.     //set up and display the encrypted image
81.     sf::RenderWindow window(sf::VideoMode(size.x, size.y), outputFile);
82.
83.     sf::Texture texture;
84.     texture.loadFromImage(image);
85.
86.     sf::Sprite sprite;
87.     sprite.setTexture(texture);
88.
89.     while (window.isOpen())

```

```

90.  {
91.      sf::Event event;
92.      while (window.pollEvent(event))
93.      {
94.          if (event.type == sf::Event::Closed)
95.              window.close();
96.      }
97.
98.      window.clear(sf::Color::White);
99.      window.draw(sprite);
100.     window.display();
101. }
102.
103. // write the file
104. if (!image.saveToFile(outputFile))
105. {
106.     return -1;
107. }
108.
109. return 0;
110. }
111.

```

FibLFSR.h:

This file should be the same as the one from PS1a because no new functions were added or functionality changed in terms of function parameters or classes.

```

1.  #ifndef FIBLFSR_H
2.  #define FIBLFSR_H
3.
4.  #include <iostream>
5.  #include <string>
6.  #include <vector>
7.
8.  class FibLFSR {
9.  public:
10.     // constructor to create LFSR with the given initial seed
11.     FibLFSR(std::string seed);
12.
13.     // simulate one step and return the new bit as 0 or 1
14.     int step(void);
15.
16.     // simulate k steps and return k-bit integer
17.     int generate(int k);
18.
19.     //overloading the output operator
20.     friend std::ostream& operator<< (std::ostream& outputStream, const FibLFSR& fibLFSR);
21.
22. private:
23.     std::vector<int> bitVector; // what we put the seed into and use for step()
24.     int vectorLength;           // size of the vector
25.     int tapBits[4];             // these are the bits you XOR together
26.     int resultValue;           // the value returned after step() or generate()

```



```

27. };
28.
29. std::ostream& operator<< (std::ostream& outputStream, const FibLFSR& fibLFSR);
30.
31. class FibLFSRException
32. {
33. public:
34.     FibLFSRException(std::string message);
35.
36. private:
37.     std::string errorMessage;
38. };
39.
40. #endif
41.

```

FibLFSR.cpp:

This is largely the same as the one in PS1a but now the constructor takes initial seed values that contain values from 0-9, A-Z, a-z. The way I implemented it was if any of those valid characters are present, then a 1 would be added to the seed in its place. (For example: 10A0Z0a0z09050f0 would become 1010101010101010). The seed must still be 16 characters long for it to work.

```

1. #include "FibLFSR.h"
2.
3. const int ASCII_0 = 48;
4. //const int ASCII_1 = 49;
5. const int ASCII_9 = 57;
6. const int ASCII_A = 65;
7. const int ASCII_Z = 90;
8. const int ASCII_a = 97;
9. const int ASCII_z = 122;
10. const int NUM_BITS_WE_WANT = 16;
11. const int LEFTMOST_BIT_POSITION = 0; //15
12. const int TAP_BIT_POSITION_1 = 2; //13
13. const int TAP_BIT_POSITION_2 = 3; //12
14. const int TAP_BIT_POSITION_3 = 5; //10
15.
16. std::ostream& operator<< (std::ostream& outputStream, const FibLFSR& fibLFSR)
17. {
18.     for (int i = 0; i < fibLFSR.vectorLength; i++)
19.     {
20.         outputStream << fibLFSR.bitVector[i];
21.     }
22.
23.     outputStream << "\t" << fibLFSR.resultValue;
24.     return outputStream;
25. }
26.
27. FibLFSRException::FibLFSRException(std::string message)

```

```

28. {
29.     errorMessage = message;
30. }
31.
32. FibLFSR::FibLFSR(std::string seed)
33. {
34.     tapBits[0] = TAP_BIT_POSITION_1;
35.     tapBits[1] = TAP_BIT_POSITION_2;
36.     tapBits[2] = TAP_BIT_POSITION_3;
37.     tapBits[3] = LEFTMOST_BIT_POSITION;
38.
39.     resultValue = 0;
40.     vectorLength = seed.length();
41.
42.     if (vectorLength != NUM_BITS_WE_WANT)
43.     {
44.         throw FibLFSRException("Incorrect number of bits");
45.     }
46.
47.     for (int i = 0; i < vectorLength; i++)
48.     {
49.         //since seed[i] is a char, we need to compare it with its int equivalent
50.         if (((seed[i] >= ASCII_0) && (seed[i] <= ASCII_9)) || ((seed[i] >= ASCII_A) &&
(seed[i] <= ASCII_Z)) || ((seed[i] >= ASCII_a) && (seed[i] <= ASCII_z)))
51.         {
52.             //if seed[i] is '0', then add a 0 to the vector
53.             if (seed[i] == ASCII_0)
54.             {
55.                 bitVector.push_back(0);
56.             }
57.             //if seed[i] is any other valid character, then add a 1 to the vector
58.             else
59.             {
60.                 bitVector.push_back(1);
61.             }
62.         }
63.         else
64.         {
65.             throw FibLFSRException("Non-symbol or alphanumeric character detected");
66.         }
67.     }
68. }
69.
70. int FibLFSR::step(void)
71. {
72.     int xorResult = (bitVector[TAP_BIT_POSITION_1] ^ bitVector[TAP_BIT_POSITION_2]) ^
(bitVector[TAP_BIT_POSITION_3] ^ bitVector[LEFTMOST_BIT_POSITION]);
73.     //Erase the first element of the bitVector
74.     bitVector.erase(bitVector.begin());
75.     //Add the xor result to the back of the bitVector
76.     bitVector.push_back(xorResult);
77.
78.     resultValue = xorResult;
79.     return xorResult;
80. }
81.

```

```

82. int FibLFSR::generate(int k)
83. {
84.     int resultInteger = 0;
85.     for (int i = 0; i < k; i++)
86.     {
87.         //left shift the bitVector then add the XOR bit to the end
88.         resultInteger = (resultInteger << 1) + this->step();
89.     }
90.
91.     resultValue = resultInteger;
92.
93.     return resultInteger;
94. }

```

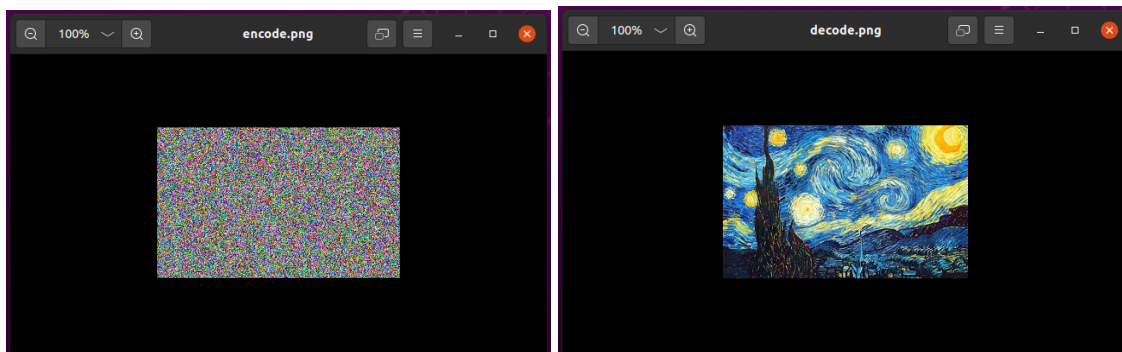
Output -

```
% PhotoMagic input-file.png output-file.png 1011011000110110
```

Above: Example command line input from the user. Would use input-file.png as the starting image and output the final image as output-file.png. The initial seed would be 1011011000110110



Left: An example starter image



Left: The resulting image after running the starter image through the program

Right: The resulting image after running the encoded image through the program. It is the same as the original starter image given.

PS2a N-Body Simulation (Part A)

Overview -

The assignment was to create a universe simulation that would read and make objects from a file and show celestial bodies after scaling their distances to fit the screen. The celestial bodies are just sprites and do not move for now. For ease of use, you can feed in a text file using input redirect in the command line (so do `./Nody < planets.txt` to let the program read from the `planets.txt` file) instead of feeding in inputs manually.

Key Concepts and Lessons Learned -

One major concept from this assignment was working with smart pointers. Before this, I was only using raw pointers but with smart pointers, I would not need to worry about deallocating memory and it would be safer handling pointers in general.

Code files in this assignment: **Makefile**, **main.cpp**, **NBody.h**, **NBody.cpp**, **planets.txt**

Makefile

For some reason in this assignment it would not work unless I added the `-ansi` and `-std=c++17` flags when compiling. Also for testing purposes, I made it so you needed the necessary planet image files in order to compile.

1. `CC = g++`
2. `CFLAGS = -Wall -Werror -ansi -std=c++17 -pedantic`
3. `SFML = -lsfml-graphics -lsfml-window -lsfml-system`
4. `OBJ = NBody.o main.o`
5. `ASSETS = earth.gif mars.gif mercury.gif sun.gif venus.gif starfield.jpg`
6. `EXE = NBody`
- 7.
8. `all: $(EXE)`
- 9.
10. `$(EXE): $(OBJ) $(ASSETS)`
11. `$(CC) $(CFLAGS) -o $(EXE) $(OBJ) $(SFML)`
- 12.
13. `NBody.o: NBody.cpp NBody.h`
14. `$(CC) $(CFLAGS) -c NBody.cpp -o NBody.o`
- 15.
16. `main.o: main.cpp`
17. `$(CC) $(CFLAGS) -c main.cpp -o main.o`
- 18.
19. `clean:`
20. `rm $(EXE) *.o`
- 21.

main.cpp

For this file, the tricky part was working with a vector of pointers because I would need to remember to dereference them to use their functions. How this works is the universe default constructor is called which reads from the input file and makes CelestialBody objects to fill its vector container. This vector is then used to access the positions of where the CelestialBody objects should be drawn and draws them there.

```
1. #include "NBody.h"
2.
3. int main(int argc, char* argv[])
4. {
5.     //Check to see if we have the background file
6.     sf::Image background;
7.     if (!background.loadFromFile("starfield.jpg"))
8.     {
9.         return -1;
10.    }
11.
12.    //Set up the window and its dimensions
13.    sf::Vector2u size = background.getSize();
14.    sf::RenderWindow window(sf::VideoMode(size.x, size.y), "Solar System");
15.
16.    //Make a Universe object which will read from file upon creation
17.    Universe universe;
18.
19.    //Set up variables for when we draw the CelestialBodies
20.    sf::Sprite tempPlanetSprite;
21.    float newXPosition;
22.    float newYPosition;
23.
24.    while (window.isOpen())
25.    {
26.        sf::Event event;
27.        while (window.pollEvent(event))
28.        {
29.            if (event.type == sf::Event::Closed)
30.                window.close();
31.        }
32.
33.        window.clear(sf::Color::Black);
34.
35.        //Set up the Background
36.        sf::Texture backgroundTexture;
37.        backgroundTexture.loadFromImage(background);
38.
39.        //Draw the background
40.        sf::Sprite backgroundSprite;
41.        backgroundSprite.setTexture(backgroundTexture);
42.        window.draw(backgroundSprite);
43.
```

```

44.         //Draw the CelestialBodies
45.         std::vector<std::shared_ptr<CelestialBody>> temp = universe.getVector();
46.         for (auto bodyPtr : temp)
47.         {
48.             //Create a temp CelestialBody sprite, modify it, then set it back
49.             tempPlanetSprite = (*bodyPtr).getBodySprite();
50.             newXPosition = calculatePosition(universe.getUniverseRadius(),
(*bodyPtr).getXPosition(), size.x);
51.             newYPosition = calculatePosition(universe.getUniverseRadius(),
(*bodyPtr).getYPosition(), size.y);
52.             tempPlanetSprite.setPosition(newXPosition, newYPosition);
53.             (*bodyPtr).setBodySprite(tempPlanetSprite);
54.
55.             //Actually draw the CelestialBody
56.             window.draw(*bodyPtr);
57.         }
58.
59.         //Display the screen
60.         window.display();
61.     }
62.
63.     return 0;
64. }
65.

```

NBody.h

The Universe and CelestialBody classes are declared here along with the standalone calculatePosition(). The CelestialBody class needs to inherit from the sf::Drawable in order to make it drawable in the SFML window.

```

1.  #ifndef NBODY_H
2.  #define NBODY_H
3.
4.  #include <SFML/System.hpp>
5.  #include <SFML/Window.hpp>
6.  #include <SFML/Graphics.hpp>
7.  #include <iostream>
8.  #include <string>
9.  #include <vector>
10. #include <memory>
11.
12. class CelestialBody : public sf::Drawable
13. {
14. public:
15.     CelestialBody();
16.
17.     CelestialBody(double xPos, double yPos,
18. double xVel, double yVel, double mass,
19. std::string name);
20.
21.     double getXPosition(void)
22.     {

```

```

23.         return xPosition;
24.     }
25.
26.     double getYPosition(void)
27.     {
28.         return yPosition;
29.     }
30.
31.     sf::Sprite getBodySprite(void)
32.     {
33.         return bodySprite;
34.     }
35.
36.     void setBodySprite(sf::Sprite newSprite)
37.     {
38.         bodySprite = newSprite;
39.     }
40.
41.     friend std::istream &operator>> (std::istream &in, CelestialBody &body);
42.
43. private:
44.     double xPosition;
45.     double yPosition;
46.     double xVelocity;
47.     double yVelocity;
48.     double bodyMass;
49.     std::string fileName;
50.     sf::Image bodyImage;
51.     sf::Texture bodyTexture;
52.     sf::Sprite bodySprite;
53.
54.     virtual void draw(sf::RenderTarget &target, sf::RenderStates states) const;
55. };
56.
57.
58. class Universe
59. {
60. public:
61.     Universe();
62.
63.     double getUniverseRadius(void)
64.     {
65.         return universeRadius;
66.     }
67.
68.     void setUniverseRadius(double newRadius)
69.     {
70.         universeRadius = newRadius;
71.     }
72.
73.     std::vector<std::shared_ptr<CelestialBody>> getVector(void)
74.     {
75.         return bodyVector;
76.     }
77.
78. private:

```

```

79.     std::vector<std::shared_ptr<CelestialBody>> bodyVector;
80.
81.     int numPlanets;
82.     double universeRadius;
83.
84. };
85.
86. //Stand-alone function for calculating place on the screen
87. float calculatePosition(double universeRadius, double position, double screenDimension);
88.
89. #endif

```

NBody.cpp

The function doing the most work here is the Universe default constructor. It reads from the input file and figures out how many planets are in the universe and how big the universe is. Then it starts filling its vector of CelestialBody smart pointers. For every CelestialBody in Universe, a smart pointer is created for it, pushed to the back of the vector, then input is read to create the CelestialBody for the pointer just added using an overloaded insertion operator.

```

1.  #include "NBody.h"
2.
3.  //CelestialBody value constructor (not used)
4.  CelestialBody::CelestialBody(double xPos, double yPos, double xVel, double yVel, double mass, std::string
    name)
5.  {
6.      xPosition = xPos;
7.      yPosition = yPos;
8.      xVelocity = xVel;
9.      yVelocity = yVel;
10.     bodyMass = mass;
11.     fileName = name;
12.
13.     if (!bodyImage.loadFromFile(fileName))
14.     {
15.         exit(1);
16.     }
17.
18.     bodyTexture.loadFromImage(bodyImage);
19.     bodySprite.setTexture(bodyTexture);
20. }
21.
22. //CelestialBody Default Constructor
23. CelestialBody::CelestialBody()
24. {
25.     xPosition = 0;
26.     yPosition = 0;
27.     xVelocity = 0;
28.     yVelocity = 0;
29.     bodyMass = 0;
30.     fileName = "";
31. }

```



```

32.
33. //CelestialBody's draw function
34. void CelestialBody::draw(sf::RenderTarget &target, sf::RenderStates states) const
35. {
36.     target.draw(bodySprite, states);
37. }
38.
39. //The insertion operator will automatically set all the CelestialBody's values
40. //And create the image, texture, and sprite
41. std::istream &operator>> (std::istream &in, CelestialBody &body)
42. {
43.     in >> body.xPosition >> body.yPosition >> body.xVelocity >> body.yVelocity >> body.bodyMass >>
        body.fileName;
44.
45.     body.bodyImage.loadFromFile(body.fileName);
46.     body.bodyTexture.loadFromImage(body.bodyImage);
47.     body.bodySprite.setTexture(body.bodyTexture);
48.
49.     return in;
50. }
51.
52. Universe::Universe()
53. {
54.     std::cin >> numPlanets >> universeRadius;
55.     for (int i = 0; i < numPlanets; i++)
56.     {
57.         //Make a new shared_ptr and add it to the back of the vector
58.         std::shared_ptr<CelestialBody> celestialBodyPtr(new CelestialBody);
59.         bodyVector.push_back(celestialBodyPtr);
60.         //Use the overloaded insertion (>>) operator to create the CelestialBody
61.         std::cin >> *bodyVector.back();
62.     }
63. }
64.
65. //Since we want (0,0) to be in the middle, we do + (screenDimension / 2)
66. //We do (position / universeRadius) * (screenDimension / 2) to get the
67. //offset of the planets from the middle
68. float calculatePosition(double universeRadius, double position, double screenDimension)
69. {
70.     float newPosition = (position / universeRadius) * (screenDimension / 2) + (screenDimension / 2);
71.     return newPosition;
72. }
73.

```

planets.txt

This is an example input file that can be fed into the program from the command line using input redirect. The first line tells how many planets are in the Universe. Second line tells the universe's radius. After that, all other lines specify the attributes of the planets starting with the x-position, then y-position, x-velocity, y-velocity, mass, and the planet's image file name.

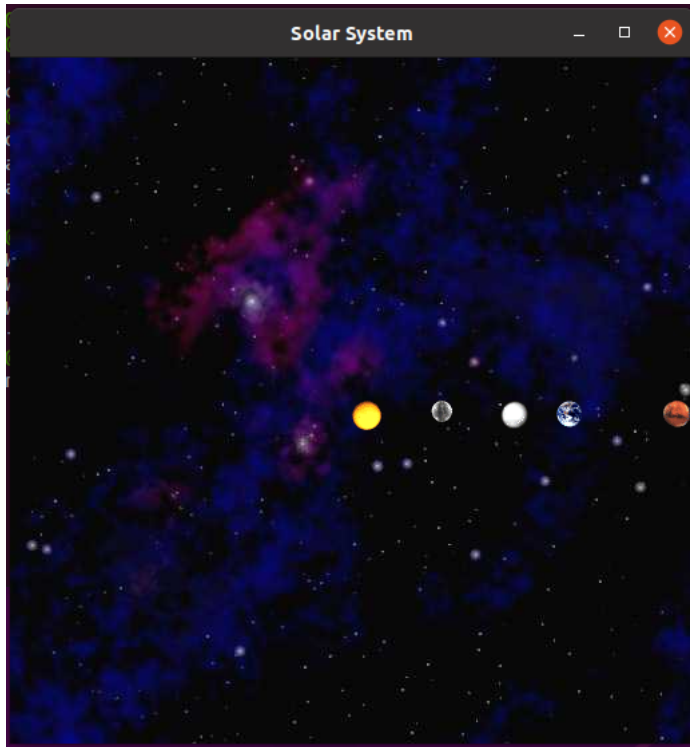
```

1. 5
2. 2.50e+11
3. 1.4960e+11 0.0000e+00 0.0000e+00 2.9800e+04 5.9740e+24 earth.gif
4. 2.2790e+11 0.0000e+00 0.0000e+00 2.4100e+04 6.4190e+23 mars.gif

```

5. 5.7900e+10 0.0000e+00 0.0000e+00 4.7900e+04 3.3020e+23 mercury.gif
6. 0.0000e+00 0.0000e+00 0.0000e+00 0.0000e+00 1.9890e+30 sun.gif
7. 1.0820e+11 0.0000e+00 0.0000e+00 3.5000e+04 4.8690e+24 venus.gif
- 8.
9. This file contains the sun and the inner 4 planets of our Solar System.

Output -



Above: The planets.txt file mentioned before, when run through the program, would generate this as output to the SFML window.

PS2b N-Body Simulation (Part B)

Overview -

For this assignment, we take the stationary program from PS2a and turn it into a moving simulation. It takes in data from planets.txt and the planetary data is used to calculate the force on each planet which is used to calculate the acceleration which itself is used to calculate the new velocity and new position of each planet. Also for the command line inputs, now the second parameter must be the total time (in seconds) that you want the program to simulate and the amount of simulated seconds each real second is. This will be discussed in the output section.

Key Concepts and Lessons Learned -

Here, the challenge was calculating the position and speeds for moving objects whose positions and speeds rely on those of another object. Before, we just had to worry about the objects by themselves and they didn't interact with the others. We also now have objects moving at different speeds because of their position so we need to adjust their speeds as they're moving. This was hard to do and although I got the planets to move, I was only able to get them to move clockwise instead of counterclockwise like the assignment wanted us to do.

Code files in this assignment: **Makefile**, **main.cpp**, **NBody.h**, **NBody.cpp**, **planets.txt**

Makefile:

This is similar to the one from PS2a, but some extra features I added were a time counter and music so to use those, I needed to add some things to ASSETS and to make the music work, I had to add -lsfml-audio to the SFML label.

1. CC = g++
2. CFLAGS = -Wall -Werror -ansi -std=c++17 -pedantic -O1
3. SFML = -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio
4. OBJ = NBody.o main.o
5. ASSETS = earth.gif mars.gif mercury.gif sun.gif venus.gif starfield.jpg Worship_the_Endless.ogg
OpenSans-Regular.ttf
6. EXE = NBody
- 7.
8. all: \$(EXE)
- 9.
10. \$(EXE): \$(OBJ) \$(ASSETS)
11. \$(CC) \$(CFLAGS) -o \$(EXE) \$(OBJ) \$(SFML)

```

12.
13. NBody.o: NBody.cpp NBody.h
14. $(CC) $(CFLAGS) -c NBody.cpp -o NBody.o
15.
16. main.o: main.cpp
17. $(CC) $(CFLAGS) -c main.cpp -o main.o
18.
19. clean:
20. rm $(EXE) *.o
21.

```

main.cpp:

Lots of new additions from PS2a. The first is a timer in the upper left corner of a window when the program runs which displays the number of real world seconds passed in yellow text. The second is music will play when the program runs. The song I chose is called “Worship the Endless” from the game Endless Space 2.

Now onto how the planets are set up and moved. The Universe default constructor still makes a vector filled with all the CelestialBodies and we go through it once to find where the Sun CelestialBody is because we base our calculations for the other planets using it. Inside the window display loop, we do the calculations and draw the planets and update their member values to their new speeds and positions.

At the end of the program, the contents of the input file will be printed out into the terminal except the x- and y-positions and x- and y-velocities will be updated to the final values calculated.

```

1. #include "NBody.h"
2.
3. int main(int argc, char* argv[])
4. {
5.     //Set up the font for displaying the time
6.     sf::Font font;
7.     if (!font.loadFromFile("OpenSans-Regular.ttf"))
8.     {
9.         return -1;
10.    }
11.
12.    //Set up the text used to display the time
13.    sf::Text elapsedTimeText;
14.    elapsedTimeText.setFont(font);
15.    elapsedTimeText.setCharacterSize(16);
16.    elapsedTimeText.setFillColor(sf::Color::Yellow);
17.
18.    //Check to see if we have the background file
19.    sf::Image background;
20.    if (!background.loadFromFile("starfield.jpg"))

```

```

21.  {
22.      return -1;
23.  }
24.
25.  //Set up and play the music
26.  sf::Music music;
27.  if (!(music.openFromFile("Worship_the_Endless.ogg")))
28.  {
29.      return -1;
30.  }
31.  music.setVolume(100);
32.  music.play();
33.
34.  //Set up the window and its dimensions
35.  sf::Vector2u size = background.getSize();
36.  sf::RenderWindow window(sf::VideoMode(size.x, size.y), "Solar System");
37.
38.  //Set the time variables by converting the command line arguments into doubles
39.  double maxTime = std::stod(argv[1]);
40.  double deltaTime = std::stod(argv[2]);
41.  double timePassed = 0; //Elapsed simulated time of the program
42.
43.  //Make a Universe object which will read from file upon creation
44.  Universe universe;
45.
46.  //Set up variables for when we draw the CelestialBodies
47.  sf::Sprite tempPlanetSprite;
48.  std::vector<std::shared_ptr<CelestialBody>> temp = universe.getVector();
49.
50.  //Set up the time and clock
51.  sf::Clock clock;
52.  sf::Time elapsedTimeIRL = clock.restart();
53.
54.  //Get the sun's mass
55.  std::shared_ptr<CelestialBody> sunPtr(nullptr);
56.  for (auto sunFinderPtr : temp)
57.  {
58.      if ((*sunFinderPtr).getFileName() == "sun.gif")
59.      {
60.          sunPtr = sunFinderPtr;
61.          break;
62.      }
63.  }
64.
65.  while (window.isOpen() && (timePassed < maxTime))
66.  {
67.      sf::Event event;
68.      while (window.pollEvent(event))
69.      {
70.          if (event.type == sf::Event::Closed)
71.              window.close();
72.      }
73.
74.      window.clear(sf::Color::Black);
75.
76.      //Set up the Background

```

```

77.     sf::Texture backgroundTexture;
78.     backgroundTexture.loadFromImage(background);
79.
80.     //Draw the background
81.     sf::Sprite backgroundSprite;
82.     backgroundSprite.setTexture(backgroundTexture);
83.     window.draw(backgroundSprite);
84.
85.     //Draw CelestialBodies
86.     for (auto bodyPtr : temp)
87.     {
88.         if (!((*bodyPtr).getFileName() == "sun.gif"))
89.         {
90.
91.             double deltaX = calculateDeltaX(bodyPtr, sunPtr);
92.             double deltaY = calculateDeltaY(bodyPtr, sunPtr);
93.
94.             double hypotenuseR = calculateHypotenuseR(deltaX, deltaY);
95.             double totalForce = calculateForce(bodyPtr, sunPtr, hypotenuseR);
96.             double newForceX = calculateForceX(totalForce, deltaX, hypotenuseR);
97.             double newForceY = calculateForceY(totalForce, deltaY, hypotenuseR);
98.
99.             (*bodyPtr).setForceX(newForceX);
100.            (*bodyPtr).setForceY(newForceY);
101.
102.            universe.step(deltaTime, bodyPtr);
103.
104.        }
105.
106.        //Create a temp CelestialBody sprite, modify it, then set it back
107.        tempPlanetSprite = (*bodyPtr).getBodySprite();
108.        float newXPosition = calculatePosition(universe.getUniverseRadius(),
109.        (*bodyPtr).getXPosition(), size.x);
110.        float newYPosition = calculatePosition(universe.getUniverseRadius(),
111.        (*bodyPtr).getYPosition(), size.y);
112.        tempPlanetSprite.setPosition(newXPosition, newYPosition);
113.        (*bodyPtr).setBodySprite(tempPlanetSprite);
114.
115.        //Actually draw the CelestialBody
116.        window.draw(*bodyPtr);
117.
118.        //Update the time label
119.        elapsedTimeIRL = clock.getElapsedTime();
120.        std::string timeText = std::to_string(elapsedTimeIRL.asSeconds());
121.        elapsedTimeText.setString(timeText);
122.        window.draw(elapsedTimeText);
123.    }
124.
125.    //Display the screen
126.    window.display();
127.    timePassed = timePassed + deltaTime;
128. }
129.
130. //Ending print statements for the state of universe
131. std::cout << universe.getNumPlanets() << std::endl <<
132.     universe.getUniverseRadius() << std::endl;

```

```

131.
132.     std::cout.precision(4);
133.     //std::vector<std::shared_ptr<CelestialBody>> temp = universe.getVector();
134.     for (auto bodyPtr : temp)
135.     {
136.         std::cout << std::scientific << (*bodyPtr).getXPosition() << "\t" <<
137.         (*bodyPtr).getYPosition() << "\t" << (*bodyPtr).getXVelocity() <<
138.         "\t" << (*bodyPtr).getYVelocity() << "\t" << (*bodyPtr).getMass() <<
139.         "\t" << (*bodyPtr).getFileName() << std::endl;
140.     }
141.
142.     return 0;
143. }
144.

```

NBody.h:

This is many things from PS2a, but now there are a lot more accessor functions and standalone math functions to help with the calculations for updating the member values for the CelestialBody objects.

```

1.  #ifndef NBODY_H
2.  #define NBODY_H
3.
4.  #include <SFML/System.hpp>
5.  #include <SFML/Window.hpp>
6.  #include <SFML/Graphics.hpp>
7.  #include <SFML/Audio.hpp>
8.  #include <iostream>
9.  // #include <string> //already included in the other header files
10. // #include <vector>
11. #include <memory>
12. #include <cmath>
13.
14. //gravitational constant G
15. const double G = 6.67e-11;
16.
17. class CelestialBody : public sf::Drawable
18. {
19. public:
20.     CelestialBody();
21.
22.     CelestialBody(double xPos, double yPos,
23.     double xVel, double yVel, double mass,
24.     std::string name);
25.
26.     double getXPosition(void) {
27.         return xPos;
28.     }
29.
30.     void setXPosition(double newXPos) {
31.         xPos = newXPos;
32.     }

```

```

33.
34. double getYPosition(void) {
35.     return yPosition;
36. }
37.
38. void setYPosition(double newYPos) {
39.     yPosition = newYPos;
40. }
41.
42. double getXVelocity(void) {
43.     return xVelocity;
44. }
45.
46. void setXVelocity(double newXVel) {
47.     xVelocity = newXVel;
48. }
49.
50. double getYVelocity(void) {
51.     return yVelocity;
52. }
53.
54. void setYVelocity(double newYVel) {
55.     yVelocity = newYVel;
56. }
57.
58. double getMass(void) {
59.     return bodyMass;
60. }
61.
62. std::string getFileName(void) {
63.     return fileName;
64. }
65.
66. sf::Sprite getBodySprite(void) {
67.     return bodySprite;
68. }
69.
70. void setBodySprite(sf::Sprite newSprite)
71. {
72.     bodySprite = newSprite;
73. }
74.
75. double getForceX(void) {
76.     return forceX;
77. }
78.
79. void setForceX(double newForceX) {
80.     forceX = newForceX;
81. }
82.
83. double getForceY(void) {
84.     return forceY;
85. }
86.
87. void setForceY(double newForceY) {
88.     forceY = newForceY;

```



```

89.     }
90.
91.     double getAccelerationX(void) {
92.         return accelerationX;
93.     }
94.
95.     void setAccelerationX(double newAccelerationX) {
96.         accelerationX = newAccelerationX;
97.     }
98.
99.     double getAccelerationY(void) {
100.        return accelerationY;
101.    }
102.
103.    void setAccelerationY(double newAccelerationY) {
104.        accelerationY = newAccelerationY;
105.    }
106.
107.    friend std::istream &operator>> (std::istream &in, CelestialBody &body);
108.
109. private:
110.     double xPosition;
111.     double yPosition;
112.     double xVelocity;
113.     double yVelocity;
114.     double bodyMass;
115.     std::string fileName;
116.     sf::Image bodyImage;
117.     sf::Texture bodyTexture;
118.     sf::Sprite bodySprite;
119.
120.     double forceX;
121.     double forceY;
122.     double accelerationX;
123.     double accelerationY;
124.
125.     virtual void draw(sf::RenderTarget &target, sf::RenderStates states) const;
126. };
127.
128.
129. class Universe
130. {
131. public:
132.     Universe();
133.
134.     void step(double deltaTime, std::shared_ptr<CelestialBody> bodyPtr);
135.
136.     double getNumPlanets(void) {
137.         return numPlanets;
138.     }
139.
140.     double getUniverseRadius(void) {
141.         return universeRadius;
142.     }
143.
144.     void setUniverseRadius(double newRadius) {

```

```

145.         universeRadius = newRadius;
146.     }
147.
148.     std::vector<std::shared_ptr<CelestialBody>> getVector(void)
149.     {
150.         return bodyVector;
151.     }
152.
153. private:
154.     std::vector<std::shared_ptr<CelestialBody>> bodyVector;
155.
156.     int numPlanets;
157.     double universeRadius;
158.
159. };
160.
161. //Standalone functions
162.
163. //Function for calculating place on the screen
164. float calculatePosition(double universeRadius, double position, double screenDimension);
165.
166. //Functions for calculating components for the physics calculations
167. double calculateDeltaX(std::shared_ptr<CelestialBody> bodyPtr,
168.     std::shared_ptr<CelestialBody> sunPtr);
169. double calculateDeltaY(std::shared_ptr<CelestialBody> bodyPtr,
170.     std::shared_ptr<CelestialBody> sunPtr);
171.
172. double calculateHypotenuseR(double deltaX, double deltaY);
173.
174. double calculateForce(std::shared_ptr<CelestialBody> bodyPtr, std::shared_ptr<CelestialBody> sunPtr,
175.     double hypotenuseR);
176. double calculateForceX(double totalForce, double deltaX, double hypotenuseR);
177. double calculateForceY(double totalForce, double deltaY, double hypotenuseR);
178. #endif
179.

```

NBody.cpp:

The important new additions are the math functions but there are too many to go over here. Some deal with forces, acceleration, hypotenuse of the distance between planet and Sun, and position to name a few. One thing to note is when calculating deltaX and deltaY, the Sun is always the second object so if you're doing $x_2 - x_1$, you do Sun.x - planet.x .

```

1.  #include "NBody.h"
2.
3.  //CelestialBody value constructor (not used)
4.  CelestialBody::CelestialBody(double xPos, double yPos,
5.      double xVel, double yVel, double mass,
6.      std::string name)
7.  {
8.      xPosition = xPos;

```

```

9.     yPos = yPos;
10.    xVelocity = xVel;
11.    yVelocity = yVel;
12.    bodyMass = mass;
13.    fileName = name;
14.
15.    if (!bodyImage.loadFromFile(fileName))
16.    {
17.        exit(1);
18.    }
19.
20.    bodyTexture.loadFromImage(bodyImage);
21.    bodySprite.setTexture(bodyTexture);
22.
23. }
24.
25. //CelestialBody Default Constructor
26. CelestialBody::CelestialBody()
27. {
28.     xPos = 0;
29.     yPos = 0;
30.     xVelocity = 0;
31.     yVelocity = 0;
32.     bodyMass = 0;
33.     fileName = "";
34. }
35.
36. //CelestialBody's draw function
37. void CelestialBody::draw(sf::RenderTarget &target, sf::RenderStates states) const
38. {
39.     target.draw(bodySprite, states);
40. }
41.
42. //The insertion operator will automatically set all the CelestialBody's values and create the image, texture,
    and sprite
43. std::istream &operator>> (std::istream &in, CelestialBody &body)
44. {
45.     in >> body.xPos >> body.yPos >> body.xVelocity >> body.yVelocity >> body.bodyMass >>
        body.fileName;
46.
47.     body.bodyImage.loadFromFile(body.fileName);
48.     body.bodyTexture.loadFromImage(body.bodyImage);
49.     body.bodySprite.setTexture(body.bodyTexture);
50.
51.     return in;
52. }
53.
54. Universe::Universe()
55. {
56.     std::cin >> numPlanets >> universeRadius;
57.     for (int i = 0; i < numPlanets; i++)
58.     {
59.         //Make a new shared_ptr and add it to the back of the vector
60.         std::shared_ptr<CelestialBody> celestialBodyPtr(new CelestialBody);
61.         bodyVector.push_back(celestialBodyPtr);
62.

```

```

63.         //Use the overloaded insertion (>>) operator to create the CelestialBody
64.         std::cin >> *bodyVector.back();
65.     }
66. }
67.
68. //Since we want (0,0) to be in the middle, we do + (screenDimension / 2)
69. //We do (position / universeRadius) * (screenDimension / 2) to get the
70. //offset of the planets from the middle
71. float calculatePosition(double universeRadius, double position, double screenDimension)
72. {
73.     float newPosition = (position / universeRadius) * (screenDimension / 2) + (screenDimension / 2);
74.     return newPosition;
75. }
76.
77. void Universe::step(double deltaTime, std::shared_ptr<CelestialBody> bodyPtr)
78. {
79.     double newAccelerationX = ((*bodyPtr).getForceX() / (*bodyPtr).getMass());
80.     (*bodyPtr).setAccelerationX(newAccelerationX);
81.     double newAccelerationY = ((*bodyPtr).getForceY() / (*bodyPtr).getMass());
82.     (*bodyPtr).setAccelerationY(newAccelerationY);
83.
84.     double newVelocityX = ((*bodyPtr).getXVelocity() + (deltaTime *
85.         (*bodyPtr).getAccelerationX()));
86.     (*bodyPtr).setXVelocity(newVelocityX);
87.     double newVelocityY = ((*bodyPtr).getYVelocity() + (deltaTime *
88.         (*bodyPtr).getAccelerationY()));
89.     (*bodyPtr).setYVelocity(newVelocityY);
90.
91.     double newPositionX = ((*bodyPtr).getXPosition() + (deltaTime *
92.         (*bodyPtr).getXVelocity()));
93.     (*bodyPtr).setXPosition(newPositionX);
94.     double newPositionY = ((*bodyPtr).getYPosition() + (deltaTime *
95.         (*bodyPtr).getYVelocity()));
96.     (*bodyPtr).setYPosition(newPositionY);
97. }
98.
99. double calculateDeltaX(std::shared_ptr<CelestialBody> bodyPtr, std::shared_ptr<CelestialBody> sunPtr) {
100.     return ((*sunPtr).getXPosition() - (*bodyPtr).getXPosition());
101. }
102.
103. double calculateDeltaY(std::shared_ptr<CelestialBody> bodyPtr, std::shared_ptr<CelestialBody> sunPtr) {
104.     return ((*sunPtr).getYPosition() - (*bodyPtr).getYPosition());
105. }
106.
107. double calculateHypotenuseR(double deltaX, double deltaY) {
108.     return (sqrt(pow(deltaX, 2) + pow(deltaY, 2)));
109. }
110.
111. double calculateForce(std::shared_ptr<CelestialBody> bodyPtr, std::shared_ptr<CelestialBody> sunPtr,
112.     double hypotenuseR) {
113.     return ((G * (*bodyPtr).getMass() * (*sunPtr).getMass()) / (pow(hypotenuseR, 2)));
114. }
115. double calculateForceX(double totalForce, double deltaX, double hypotenuseR) {
116.     return (totalForce * (deltaX / hypotenuseR));
117. }

```

```

118.
119. double calculateForceY(double totalForce, double deltaY, double hypotenuseR) {
120.     return (totalForce * (deltaY / hypotenuseR));
121. }
122.

```

planets.txt:

The contents of this should be the same as from PS2a. The only changes to input isn't in the input file that you pass to the program, it's the command line arguments you use when executing the program as will be discussed in the next section.

```

1. 5
2. 2.50e+11
3. 1.4960e+11 0.0000e+00 0.0000e+00 2.9800e+04 5.9740e+24 earth.gif
4. 2.2790e+11 0.0000e+00 0.0000e+00 2.4100e+04 6.4190e+23 mars.gif
5. 5.7900e+10 0.0000e+00 0.0000e+00 4.7900e+04 3.3020e+23 mercury.gif
6. 0.0000e+00 0.0000e+00 0.0000e+00 0.0000e+00 1.9890e+30 sun.gif
7. 1.0820e+11 0.0000e+00 0.0000e+00 3.5000e+04 4.8690e+24 venus.gif
8.
9. This file contains the sun and the inner 4 planets of our Solar System.

```

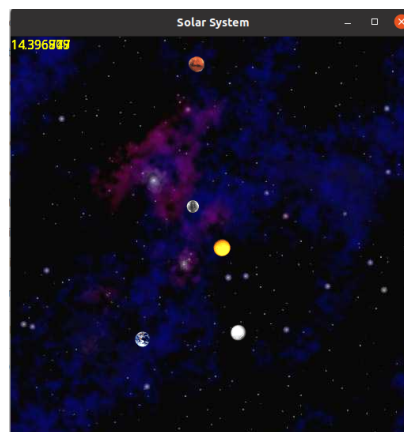
Output -

```

ethan@ethan-VirtualBox:~/ps2b$ ./NBody 157788000.0 25000.0 < planets.txt
Setting vertical sync not supported
5
2.5e+11
-1.3309e+10 -1.4961e+11 2.9561e+04 -2.6684e+03 5.9740e+24 earth.gif
-2.2088e+11 5.1569e+10 -5.5550e+03 -2.3569e+04 6.4190e+23 mars.gif
-3.7806e+10 4.3449e+10 -3.6905e+04 -3.0946e+04 3.3020e+23 mercury.gif
0.0000e+00 0.0000e+00 0.0000e+00 0.0000e+00 1.9890e+30 sun.gif
-1.0531e+11 2.3566e+10 -7.9309e+03 -3.4186e+04 4.8690e+24 venus.gif

```

Above: Example of command line arguments and final output by the program



Above: Program during runtime as planets are moving and the timer is running in the top left

PS3 DNA Sequence Alignment

Overview -

The purpose of this assignment is to take 2 pieces of DNA and perform global sequence alignment to find the shortest edit distance for them. To do this, you take 2 proteins and you compare them and assign penalties to the edit distance: if the proteins are the same, then 0 penalty, if there is a protein mismatch, then 1 penalty, if a gap needs to be added, then 2 penalty. There are different ways to approach this problem like with recursion, Needleman-Wunsch algorithm, Hirschberg's algorithm, etc.

Key Concepts and Lessons Learned -

The way I approached this is to use the Needleman-Wunsch algorithm for my program. The way I implemented it was so it would create a (m x n) matrix (m and n being the lengths of the pieces of DNA inputted) and to fill in the edges before moving from the bottom right corner to the top left corner. This is not the most efficient algorithm to use for this situation, but it was the one I was most familiar with because we spent the most time in class discussing it.

Another thing introduced in this assignment is lambda functions. They are like functions that are able to capture outside variables for use in them (alongside passed in arguments). I had heard of them before but this was my first time implementing one.

A tool that we had to use for this assignment was Valgrind which let us see how much memory and time our program used. I had used it in labs for previous computing classes but did not really dive into it. We used it in this assignment in order to see the memory usage of our program.

Something I was supposed to learn for the post-assignment analysis was the doubling method and how I applied it to my program. I did not understand what they were referring to when they said “doubling method” so I did not answer that.

Code files in this assignment: **Makefile, main.cpp, ps3.h, ps3.cpp**

Makefile:

This Makefile is fairly standard and the only optimization I did was add the -O2 optimization flag in order for it to run faster. The only reason we need -lsfml-system is to use SFML's time class in the main.cpp file.

1. CC = g++
2. CFLAGS = -Wall -Werror -std=c++11 -pedantic -O2 -g
3. SFML = -lsfml-system
4. OBJ = ps3.o main.o
5. EXE = ED
- 6.
7. all: \$(EXE)

```

8.
9. $(EXE): $(OBJ)
10. $(CC) $(CFLAGS) -o $(EXE) $(OBJ) $(SFML)
11.
12. ps3.o: ps3.cpp ps3.h
13. $(CC) $(CFLAGS) -c ps3.cpp -o ps3.o
14.
15. main.o: main.cpp
16. $(CC) $(CFLAGS) -c main.cpp -o main.o
17.
18. clean:
19. rm $(EXE) *.o
20.

```

main.cpp:

A lambda function does most of the work in this file. First the 2 DNA strings are taken from the command line/input file. Then the lambda function creates an ED object out of the DNA strings and runs them through the ED class' OptDistance() function to find the optimal edit path for the 2 DNA strings. The optimal path is calculated and returned along with the program run time.

```

1. #include "ps3.h"
2. #include <SFML/System.hpp>
3.
4. int main(int argc, char* argv[])
5. {
6.     sf::Clock clock;
7.     sf::Time t;
8.
9.     std::string rowString;    //string for the y-axis of matrix
10.    std::string columnString;  //string for the x-axis of matrix
11.    std::cin >> rowString >> columnString;
12.
13.    //Lambda function
14.    auto runProgram = [rowString, columnString]()
15.    {
16.        ED test(rowString, columnString);
17.        int optimalDistance = test.OptDistance();
18.        std::cout << "Edit Distance = " << optimalDistance << std::endl;
19.        std::string pathMessage = test.Alignment();
20.        std::cout << pathMessage << std::endl;
21.    };
22.
23.    runProgram();
24.
25.    //Return program run time
26.    t = clock.getElapsedTime();
27.    std::cout << "Execution time is " << t.asSeconds() << " seconds \n";
28.

```

```

29.     return 0;
30. }
31.

```

ps3.h:

The class declaration of ED (edit distance) is here. A vector of vector<int> is used to represent the matrix because it felt easier to manage than a dynamically allocated 2D array.

```

1.  #ifndef PS3_H
2.  #define PS3_H
3.
4.  #include <iostream>
5.  #include <vector>
6.  #include <string>
7.
8.  class ED {
9.  public:
10.     ED(std::string initRowString, std::string initColumnString);
11.     int OptDistance(void);
12.     std::string Alignment(void);
13.
14. private:
15.     std::string rowString;
16.     std::string columnString;
17.     std::vector<std::vector<int>>> matrix;
18. };
19.
20. #endif
21.

```

ps3.cpp:

The important functions here are the OptDistance() and Alignment() functions.

The OptDistance() function fills in an ED object's vector matrix starting with the bottom row and right column and then filling it row by row starting from the bottom and working upwards. The values to put in each cell for non-bottom row and non-right column cell is to look at the right cell next to it, the bottom-right diagonal cell to it, and the cell on the bottom of it and then finding the smallest number after taking into account the penalty between that cell and the current cell and choosing the one with the lowest value to put in the current cell.

The Alignment() function finds the path in the matrix with the lowest numbers and follows that from the top leftmost cell to the bottom rightmost cell and prints its path along the way.

```

1.  #include "ps3.h"
2.  static int penalty(char a, char b);
3.  static int min(int a, int b, int c);
4.  const int EDGE_VAL = 99999;    //arbitrarily large number used to denote the edge of the matrix
5.
6.  ED::ED(std::string initRowString, std::string initColumnString)
7.  {

```



```

8.   rowString = initRowString;           //string for the y-axis of matrix
9.   columnString = initColumnString; //string for the x-axis of matrix
10.
11.  //The <= is to get an extra row for the - row at the end
12.  for (unsigned int i = 0; i <= rowString.length(); ++i)
13.  {
14.      //Set up the y-axis
15.      std::vector<int> row;
16.      matrix.push_back(row);
17.
18.      //The <= is to get an extra column for the - column at the end
19.      for (unsigned int j = 0; j <= columnString.length(); ++j)
20.      {
21.          //Set up the x-axis, fill each entry with 0
22.          matrix[i].push_back(0);
23.      }
24.  }
25. }
26.
27. int ED::OptDistance(void)
28. {
29.     int bottomRow = matrix.size() - 1;
30.     int rightmostColumn = matrix[bottomRow].size() - 1;
31.
32.     //Start from the bottom row, then move up
33.     //Fills the right edge of matrix
34.     for (int i = bottomRow; i >= 0; --i)
35.     {
36.         matrix[i][rightmostColumn] = ((bottomRow - i) * 2);
37.     }
38.
39.     //Start from the rightmost column, then move to the left
40.     //Fills in the bottom edge
41.     for (int j = rightmostColumn; j >= 0; --j)
42.     {
43.         matrix[bottomRow][j] = ((rightmostColumn - j) * 2);
44.     }
45.
46.     //Fill in the rest of the entries
47.     for (int r = bottomRow - 1; r >= 0; --r)
48.     {
49.         for (int c = rightmostColumn - 1; c >= 0; --c)
50.         {
51.             int rightVal = matrix[r][c + 1];
52.             int diagonalVal = matrix[r + 1][c + 1];
53.             int bottomVal = matrix[r + 1][c];
54.             int diagonalPenalty = penalty(rowString[r], columnString[c]);
55.
56.             matrix[r][c] = min(rightVal + 2, diagonalVal + diagonalPenalty, bottomVal + 2);
57.         }
58.     }
59.
60.     /*
61.     //Test to see if matrix is set up right
62.     for (unsigned int i = 0; i < matrix.size(); ++i)
63.     {

```

```

64.         for (unsigned int j = 0; j < matrix[i].size(); ++j)
65.         {
66.             std::cout << matrix[i][j] << "\t";
67.         }
68.         std::cout << std::endl;
69.     }
70.     */
71.
72.     return matrix[0][0];
73. }
74.
75. std::string ED::Alignment(void)
76. {
77.     int bottomRow = matrix.size() - 1;
78.     int rightmostColumn = matrix[bottomRow].size() - 1;
79.     std::string pathMessage = "";
80.     int r = 0;
81.     int c = 0;
82.
83.     while ((r != bottomRow) || (c != rightmostColumn))
84.     {
85.         int rightVal, diagonalVal, bottomVal;
86.
87.         if (c != rightmostColumn)
88.         {
89.             rightVal = matrix[r][c + 1];
90.         }
91.         else
92.         {
93.             rightVal = EDGE_VAL;
94.         }
95.
96.         if (r != bottomRow)
97.         {
98.             bottomVal = matrix[r + 1][c];
99.         }
100.        else
101.        {
102.            bottomVal = EDGE_VAL;
103.        }
104.
105.        if (!((rightVal == EDGE_VAL) || (bottomVal == EDGE_VAL)))
106.        {
107.            diagonalVal = matrix[r + 1][c + 1];
108.        }
109.        else
110.        {
111.            diagonalVal = EDGE_VAL;
112.        }
113.
114.        int penaltyVal = penalty(rowString[r], columnString[c]);
115.
116.        //If the next step is to the diagonal entry
117.        if (matrix[r][c] == (diagonalVal + penaltyVal))
118.        {
119.            pathMessage = pathMessage + rowString[r] + " "

```

```

120.             + columnString[c] + " " + std::to_string(penaltyVal) + "\n";
121.             r = r + 1;
122.             c = c + 1;
123.         }
124.
125.         //If the next step is to the bottom entry
126.         else if (matrix[r][c] == (bottomVal + 2))
127.         {
128.             pathMessage = pathMessage + rowString[r]
129.                 + " - " + std::to_string(2) + "\n";
130.             r = r + 1;
131.         }
132.
133.         //If the next step is to the right entry
134.         else if (matrix[r][c] == (rightVal + 2))
135.         {
136.             pathMessage = pathMessage + "- " + columnString[c] +
137.                 " " + std::to_string(2) + "\n";
138.             c = c + 1;
139.         }
140.         else
141.         {
142.             std::cout << "Idk what happened \n";
143.             pathMessage = pathMessage;
144.         }
145.     }
146.
147.     return pathMessage;
148. }
149.
150. //Aligning 2 characters that match = 0, mismatch = 1
151. static int penalty(char a, char b)
152. {
153.     return !(a == b);
154. }
155.
156. //Finds the minimum of 3 values
157. static int min(int a, int b, int c)
158. {
159.     int minimum = a;
160.     if (b < minimum)
161.     {
162.         minimum = b;
163.     }
164.     if (c < minimum)
165.     {
166.         minimum = c;
167.     }
168.
169.     return minimum;
170. }

```

Output -

Let's say for example you pass in a file called endgaps7.txt that contains the strings:

atattat

tattata

Once you run it through the program, it generates this:

```
ethan@ethan-VirtualBox:~/ps3$ ./ED < endgaps7.txt
Edit Distance = 4
a - 2
t t 0
a a 0
t t 0
t t 0
a a 0
t t 0
- a 2
Execution time is 0.000558 seconds
```

The minimal path and program run time are printed to the terminal.

PS4a Synthesizing a Plucked String Sound (Part A)

Overview -

In this program, we lay the groundwork for simulating the plucking of a guitar string by first implementing a circular buffer. The circular buffer is a first-in-first-out (FIFO) container so if it is full, then the first index is deleted and everything is moved up and something is added to the end. The functions we need for this part of the assignment only deal with the basic functions of the circular buffer and making sure they behave properly.

Key Concepts and Lessons Learned -

One concept introduced with this assignment is making our code pass cpplint. The files included in this section and future sections should be able to pass cpplint standards originally but when pasting to this document the formatting may be off so they may need to be readjusted to fit cpplint standards.

Code files in this assignment: **Makefile**, **test.cpp**, **CircularBuffer.h**, **CircularBuffer.cpp**

Makefile:

Nothing special about this file besides the -g flag is added in order for Valgrind to analyze the memory usage of the files.

1. CC = g++
2. CFLAGS = -Wall -Werror -std=c++14 -pedantic -O1 -g
3. BOOST_ARG = -lboost_unit_test_framework
4. OBJ = CircularBuffer.o test.o
5. EXE = ps4a
- 6.
7. all: \$(EXE)
- 8.
9. \$(EXE): \$(OBJ)
10. \$(CC) \$(CFLAGS) -o \$(EXE) \$(OBJ) \$(BOOST_ARG)
- 11.
12. CircularBuffer.o: CircularBuffer.cpp CircularBuffer.h
13. \$(CC) \$(CFLAGS) -c CircularBuffer.cpp -o CircularBuffer.o
- 14.
15. test.o: test.cpp
16. \$(CC) \$(CFLAGS) -c test.cpp -o test.o \$(BOOST_ARG)
- 17.
18. clean:
19. rm \$(EXE) *.o

test.cpp

Since there was no real output to check to see if the program was working correctly or not, we used Boost unit tests to check for expected behavior. The first test cases check to see if the functions work as expected. The second test cases check to see if exceptions are thrown. The third test cases check to see if exceptions are not thrown.

```

1. // Copyright 2021 Ethan Yu
2. #define BOOST_TEST_DYN_LINK
3. #define BOOST_TEST_MODULE Main
4. #include <boost/test/unit_test.hpp>
5. #include "CircularBuffer.h"
6.
7. BOOST_AUTO_TEST_CASE(testFunctionsWork) {
8.     // std::cout << "Testing good function output\n";
9.     CircularBuffer testObj(3);
10.    // std::cout << "Size of deque is " << testObj.size() << std::endl;
11.    testObj.enqueue(1);
12.    // std::cout << "Size of deque is " << testObj.size() << std::endl;
13.    testObj.enqueue(2);
14.    testObj.enqueue(3);
15.
16.    BOOST_REQUIRE(testObj.size() == 3);
17.    BOOST_REQUIRE(testObj.isEmpty() == false);
18.    BOOST_REQUIRE(testObj.isFull() == true);
19.    BOOST_REQUIRE(testObj.dequeue() == 1);
20.    BOOST_REQUIRE(testObj.peek() == 2);
21. }
22.
23. BOOST_AUTO_TEST_CASE(testExceptionsThrow) {
24.    // std::cout << "Testing exception throws\n";
25.    BOOST_CHECK_THROW(CircularBuffer testObj(0), std::invalid_argument);
26.    BOOST_CHECK_THROW(CircularBuffer testObj(-1), std::invalid_argument);
27.
28.    CircularBuffer testObj(1);
29.    BOOST_CHECK_THROW(testObj.dequeue(), std::runtime_error);
30.    BOOST_CHECK_THROW(testObj.peek(), std::runtime_error);
31.
32.    testObj.enqueue(1);
33.    BOOST_CHECK_THROW(testObj.enqueue(2), std::runtime_error);
34. }
35.
36. BOOST_AUTO_TEST_CASE(testExceptionsDontThrow) {
37.    // std::cout << "Testing exceptions aren't thrown\n";
38.    BOOST_CHECK_NO_THROW(CircularBuffer testObj(1));
39.
40.    CircularBuffer testObj(1);
41.    BOOST_CHECK_NO_THROW(testObj.enqueue(1));
42.    BOOST_CHECK_NO_THROW(testObj.peek());
43.    BOOST_CHECK_NO_THROW(testObj.dequeue());
44. }
45.

```

CircularBuffer.h

Here is the class declaration for CircularBuffer. I used a deque to represent the FIFO container because you can easily remove from the front and add to the back with it.

```

1. // Copyright 2021 Ethan Yu
2. #ifndef _HOME_ETHAN_PS4A_CIRCULARBUFFER_H_
3. #define _HOME_ETHAN_PS4A_CIRCULARBUFFER_H_

```

```

4.
5. #include <stdint.h>
6. #include <iostream>
7. #include <deque>
8.
9.
10. class CircularBuffer {
11. public:
12.     explicit CircularBuffer(int capacity);
13.     int size(void);
14.     bool isEmpty(void);
15.     bool isFull(void);
16.     void enqueue(int16_t x);
17.     int16_t dequeue(void);
18.     int16_t peek(void);
19.
20. private:
21.     std::deque<int16_t> container;
22.     int max_container_size;
23. };
24.
25. #endif // _HOME_ETHAN_PS4A_CIRCULARBUFFER_H_

```

CircularBuffer.cpp

Most of the functions here use the built-in functions from deque to perform the actions. The most of the additions I did were to throw exceptions when conditions were not met. I also used a lambda in the value constructor to check for bad input.

```

1. // Copyright 2021 Ethan Yu
2. #include "CircularBuffer.h"
3. #include <stdexcept>
4.
5. const int ERROR_VAL = 1337;
6.
7. // Value constructor for CircularBuffer
8. CircularBuffer::CircularBuffer(int capacity) {
9.     auto checkBadInput = [=]() {
10.         if (capacity > 0) {
11.             max_container_size = capacity;
12.         } else {
13.             throw std::invalid_argument
14.                 ("CircularBuffer constructor: capacity must be greater than zero.");
15.         }
16.     };
17.
18.     checkBadInput();
19. }
20.
21. // Returns the current size of the container
22. int CircularBuffer::size(void) {
23.     return container.size();
24. }
25.
26. // Returns status reflecting if container is empty (true) or not (false)

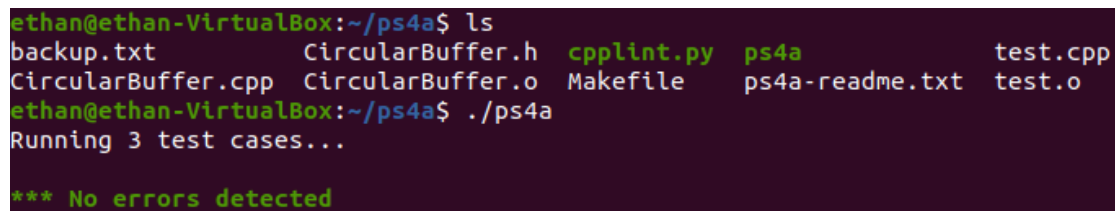
```

```

27. bool CircularBuffer::isEmpty(void) {
28.     return container.empty();
29. }
30.
31. // Returns status reflecting if container is full (true) or not (false)
32. bool CircularBuffer::isFull(void) {
33.     return (container.size() == static_cast<unsigned int>(max_container_size));
34. }
35.
36. // Adds value to back of container (unless it's full, then it throws an error)
37. void CircularBuffer::enqueue(int16_t x) {
38.     if (!isFull()) {
39.         container.push_back(x);
40.     } else {
41.         throw std::runtime_error("Enqueue: can't enqueue to a full ring.");
42.     }
43. }
44.
45. // Returns and removes the first element of container
46. // (or if empty, throws an error)
47. int16_t CircularBuffer::dequeue(void) {
48.     int16_t frontVal = 0;
49.     if (!isEmpty()) {
50.         frontVal = container.front();
51.         container.pop_front();
52.     } else {
53.         throw std::runtime_error("Dequeue: can't dequeue from an empty ring.");
54.         frontVal = ERROR_VAL;
55.     }
56.     return frontVal;
57. }
58.
59. // Returns the first element of the container
60. // (or if container is empty, throws an error)
61. int16_t CircularBuffer::peek(void) {
62.     if (!isEmpty()) {
63.         return container.front();
64.     } else {
65.         throw std::runtime_error("Peek: can't peek at an empty ring.");
66.         return ERROR_VAL;
67.     }
68. }

```

Output -



```

ethan@ethan-VirtualBox:~/ps4a$ ls
backup.txt      CircularBuffer.h  cpplint.py  ps4a          test.cpp
CircularBuffer.cpp  CircularBuffer.o  Makefile    ps4a-readme.txt  test.o
ethan@ethan-VirtualBox:~/ps4a$ ./ps4a
Running 3 test cases...

*** No errors detected

```

Above: Terminal output from my program showing that no errors occurred so it passed all the test cases.

PS4b Synthesizing a Plucked String Sound (Part B)

Overview -

With this assignment we were supposed to build off of the previous one and actually make a usable piano keyboard. Using the CircularBuffer from the previous part, we were supposed to implement the Karplus-Strong algorithm in order to simulate a key's sound dying out over time.

Key Concepts and Lessons Learned -

A key concept was how to manage a large range of inputs because we needed to find a way to manage 37 possible inputs in an efficient way. I unfortunately was unable to find a way to do this as will be explained later.

Another concept introduced was using `std::random_device`, `std::mt19937`, and `std::uniform_int_distribution` to generate truly random numbers because I learned that `rand()` does not generate truly random numbers because the probability of getting a number at the end of your range is slightly lower than the rest of your range so there is not a truly even random probability of getting all numbers in your range.

Code files in this assignment: **Makefile**, **KSGuitarSim.cpp**, **CircularBuffer.h**, **StringSound.h**, **CircularBuffer.cpp**, **StringSound.cpp**

Makefile:

This is similar to the other Makefiles except to get sound, `-lsfml-audio` needs to be added to `CFLAGS` in order to use the sound libraries in SFML.

1. `CC = g++`
2. `CFLAGS = -Wall -Werror -std=c++11 -pedantic -O2 -g`
3. `SFML = -lsfml-system -lsfml-audio -lsfml-graphics -lsfml-window`
4. `OBJ = CircularBuffer.o StringSound.o KSGuitarSim.o`
5. `EXE = KSGuitarSim`
- 6.
7. `all: $(EXE)`
- 8.
9. `$(EXE): $(OBJ)`
10. `$(CC) $(CFLAGS) -o $(EXE) $(OBJ) $(SFML)`
- 11.
12. `CircularBuffer.o: CircularBuffer.cpp CircularBuffer.h`
13. `$(CC) $(CFLAGS) -c CircularBuffer.cpp -o CircularBuffer.o`
- 14.

```

15. StringSound.o: StringSound.cpp StringSound.h
16. $(CC) $(CFLAGS) -c StringSound.cpp -o StringSound.o
17.
18. KSGuitarSim.o: KSGuitarSim.cpp
19. $(CC) $(CFLAGS) -c KSGuitarSim.cpp -o KSGuitarSim.o $(SFML)
20.
21. clean:
22. rm $(EXE) *.o
23.

```

KSGuitarSim.cpp:

How this works is a sound object is set up and then based on the key pressed in the window event loop, a different frequency will play. The problem is whenever I tried to implement catching user keyboard input in the event loop, it would somehow ignore my default statement and tell me to make a case for every single input type (too many to mention here) and even after doing that, it would still yell at me saying not every possibility was covered. That is why I have a dreaded 37 case switch statement that plays the same frequency. Nothing else I tried would work and this was the best I could do.

```

1.  /* Copyright 2015 Fred Martin, Y.
2.    Rykalova, 2020
3.    Ethan Yu, 2021
4.  */
5.  #include "CircularBuffer.h"
6.  #include "StringSound.h"
7.  #include <limits.h>
8.  #include <string>
9.  #include <exception>
10. #include <SFML/Graphics.hpp>
11. #include <SFML/System.hpp>
12. #include <SFML/Window.hpp>
13.
14. #define CONCERT_A 220.0
15. #define SAMPLES_PER_SEC 44100
16.
17. std::vector<sf::Int16> makeSamples(StringSound& gs) {
18.     std::vector<sf::Int16> samples;
19.
20.     gs.pluck();
21.     int duration = 8; // seconds
22.     int i;
23.     for (i = 0; i < SAMPLES_PER_SEC * duration; i++) {
24.         gs.tic();
25.         samples.push_back(gs.sample());
26.     }
27.     return samples;
28. }
29.
30. int main() {

```

```

31. sf::RenderWindow window(sf::VideoMode(300, 200),
32. "SFML Plucked String Sound Lite");
33. sf::Event event;
34. std::vector<sf::Int16> samples;
35. double freq = CONCERT_A;
36. StringSound gs1(freq);
37. sf::Sound sound1;
38. sf::SoundBuffer buf1;
39. samples = makeSamples(gs1);
40. if (!buf1.loadFromSamples(&samples[0], samples.size(), 2, SAMPLES_PER_SEC))
41.     throw std::runtime_error("sf::SoundBuffer: failed to load from samples.");
42. sound1.setBuffer(buf1);
43.
44. while (window.isOpen()) {
45.     while (window.pollEvent(event)) {
46.         switch (event.type) {
47.             case sf::Event::Closed:
48.                 window.close();
49.                 break;
50.
51.             case sf::Event::KeyPressed:
52.                 switch (event.key.code) {
53.                     case sf::Keyboard::Q:
54.                     case sf::Keyboard::Num2:
55.                     case sf::Keyboard::W:
56.                     case sf::Keyboard::E:
57.                     case sf::Keyboard::Num4:
58.                     case sf::Keyboard::R:
59.                     case sf::Keyboard::Num5:
60.                     case sf::Keyboard::T:
61.                     case sf::Keyboard::Y:
62.                     case sf::Keyboard::Num7:
63.                     case sf::Keyboard::U:
64.                     case sf::Keyboard::Num8:
65.                     case sf::Keyboard::I:
66.                     case sf::Keyboard::Num9:
67.                     case sf::Keyboard::O:
68.                     case sf::Keyboard::P:
69.                     case sf::Keyboard::Hyphen:
70.                     case sf::Keyboard::LBracket:
71.                     case sf::Keyboard::Equal:
72.                     case sf::Keyboard::Z:
73.                     case sf::Keyboard::X:
74.                     case sf::Keyboard::D:
75.                     case sf::Keyboard::C:
76.                     case sf::Keyboard::F:
77.                     case sf::Keyboard::V:
78.                     case sf::Keyboard::G:
79.                     case sf::Keyboard::B:
80.                     case sf::Keyboard::N:
81.                     case sf::Keyboard::J:
82.                     case sf::Keyboard::M:
83.                     case sf::Keyboard::K:
84.                     case sf::Keyboard::Comma:
85.                     case sf::Keyboard::Period:
86.                     case sf::Keyboard::Semicolon:

```

```

87.         case sf::Keyboard::Slash:
88.         case sf::Keyboard::Quote:
89.         case sf::Keyboard::Space:
90.             sound1.play();
91.             break;
92.         default:
93.             break;
94.     }
95.     default:
96.         break;
97.     }
98.     window.clear();
99.     window.display();
100.    }
101. }
102. return 0;
103.}

```

CircularBuffer.h:

This is largely similar to the one in PS4a except accessor functions were added to return the container and its max size.

```

1.  // Copyright 2021 Ethan Yu
2.  #ifndef _HOME_ETHAN_PS4B_CIRCULARBUFFER_H_
3.  #define _HOME_ETHAN_PS4B_CIRCULARBUFFER_H_
4.
5.  #include <stdint.h>
6.  #include <iostream>
7.  #include <deque>
8.
9.  class CircularBuffer {
10. public:
11.     explicit CircularBuffer(int capacity);
12.     int size(void);
13.     bool isEmpty(void);
14.     bool isFull(void);
15.     void enqueue(int16_t x);
16.     int16_t dequeue(void);
17.     int16_t peek(void);
18.     inline std::deque<int16_t> getContainer(void) const {
19.         return container;
20.     }
21.     inline unsigned int getMaxContainerSize(void) const {
22.         return max_container_size;
23.     }
24.
25. private:
26.     std::deque<int16_t> container;
27.     unsigned int max_container_size;
28. };
29.
30. #endif // _HOME_ETHAN_PS4B_CIRCULARBUFFER_H_

```

StringSound.h:

The StringSound class declaration is here. It includes functions to simulate a guitar string pluck and time passing and holds a pointer to a CircularBuffer object to add and remove frequency values from.

```
1. // Copyright 2021 Ethan Yu
2. #ifndef _HOME_ETHAN_PS4B_STRINGSOUND_H_
3. #define _HOME_ETHAN_PS4B_STRINGSOUND_H_
4.
5. #include "CircularBuffer.h"
6. #include <math.h>
7. #include <vector>
8. #include <SFML/Audio.hpp>
9.
10. class StringSound {
11. public:
12.     explicit StringSound(double frequency);
13.     explicit StringSound(std::vector<sf::Int16> init);
14.     // StringSound (const StringSound &obj) {}; // no copy const
15.     ~StringSound(void);
16.     void pluck(void);
17.     void tic(void);
18.     sf::Int16 sample(void);
19.     int time(void);
20. private:
21.     CircularBuffer* _cb;
22.     int _time;
23. };
24.
25. #endif // _HOME_ETHAN_PS4B_STRINGSOUND_H_
26.
```

CircularBuffer.cpp:

This file should be exactly the same as the one in PS4a.

```
1. // Copyright 2021 Ethan Yu
2. #include "CircularBuffer.h"
3. #include <stdexcept>
4.
5. const int ERROR_VAL = 1337;
6.
7. // Value constructor for CircularBuffer
8. CircularBuffer::CircularBuffer(int capacity) {
9.     auto checkBadInput = [=]() {
10.         if (capacity > 0) {
11.             max_container_size = capacity;
12.         } else {
13.             throw std::invalid_argument("CircularBuffer constructor: capacity must be greater than
zero.");

```

```

14.         }
15. };
16.
17.         checkBadInput();
18. }
19.
20. // Returns the current size of the container
21. int CircularBuffer::size(void) {
22.     return container.size();
23. }
24.
25. // Returns status reflecting if container is empty (true) or not (false)
26. bool CircularBuffer::isEmpty(void) {
27.     return container.empty();
28. }
29.
30. // Returns status reflecting if container is full (true) or not (false)
31. bool CircularBuffer::isFull(void) {
32.     return (container.size() == max_container_size);
33. }
34.
35. // Adds value to back of container (unless it's full, then it throws an error)
36. void CircularBuffer::enqueue(int16_t x) {
37.     if (!isFull()) {
38.         container.push_back(x);
39.     } else {
40.         throw std::runtime_error("Enqueue: can't enqueue to a full ring.");
41.     }
42. }
43.
44. // Returns and removes the first element of container
45. // (or if empty, throws an error)
46. int16_t CircularBuffer::dequeue(void) {
47.     int16_t frontVal = 0;
48.     if (!isEmpty()) {
49.         frontVal = container.front();
50.         container.pop_front();
51.     } else {
52.         throw std::runtime_error("Dequeue: can't dequeue from an empty ring.");
53.         frontVal = ERROR_VAL;
54.     }
55.
56.     return frontVal;
57. }
58.
59. // Returns the first element of the container
60. // (or if container is empty, throws an error)
61. int16_t CircularBuffer::peek(void) {
62.     if (!isEmpty()) {
63.         return container.front();
64.     } else {
65.         throw std::runtime_error("Peek: can't peek at an empty ring.");
66.         return ERROR_VAL;
67.     }
68. }

```

StringSound.cpp:

Some of the important functions include the pluck() and tic() functions. The pluck() function fills the CircularBuffer object with truly random numbers using a lambda function. The tic() function implements the Karplus-Strong algorithm and increments the time.

```
1. // Copyright 2021 Ethan Yu
2. #include "CircularBuffer.h"
3. #include "StringSound.h"
4. #include <random>
5.
6. #define SAMPLING_RATE 44100
7. #define ENERGY_DECAY_FACTOR 0.996
8. #define HALF 0.5
9.
10. StringSound::StringSound(double frequency) {
11.     if (frequency <= 0) {
12.         throw std::runtime_error("StringSound constructor: Can't have 0 or negative frequency");
13.     } else {
14.         _cb = new CircularBuffer(std::ceil(SAMPLING_RATE / frequency));
15.         _time = 0;
16.     }
17. }
18.
19. StringSound::StringSound(std::vector<sf::Int16> init) {
20.     _cb = new CircularBuffer(init.size());
21.     unsigned int i = 0;
22.     while (i < init.size()) {
23.         (*_cb).enqueue(init[i]);
24.         ++i;
25.     }
26.     _time = 0;
27. }
28.
29. // StringSound::StringSound(const StringSound &obj) {} // no copy const
30.
31. StringSound::~StringSound(void) {
32.     delete _cb;
33.     _time = 0;
34. }
35.
36. void StringSound::pluck(void) {
37.     if ((*_cb).isFull()) {
38.         (*_cb).getContainer().clear();
39.     }
40.     auto generateRandomNums = [=] () {
41.         std::random_device rd;
42.         std::mt19937 gen(rd());
43.         std::uniform_int_distribution<int16_t> dist(-32768, 32767);
44.         while (!(*_cb).isFull()) {
45.             (*_cb).enqueue(dist(gen));
46.         }
47.     };
```

```

48.         generateRandomNums();
49.     }
50.
51. void StringSound::tic(void) {
52.     pluck();
53.     sf::Int16 oldFrontVal = (*_cb).dequeue();
54.     sf::Int16 newVal = ENERGY_DECAY_FACTOR * (HALF * (oldFrontVal + (*_cb).peek()));
55.     (*_cb).enqueue(newVal);
56.     _time = _time + 1;
57. }
58.
59. sf::Int16 StringSound::sample(void) {
60.     if ((*_cb).isEmpty()) {
61.         throw std::runtime_error
62.             ("StringSound sample: Can't peek at empty container");
63.         return 0;
64.     }
65.     return (*_cb).peek();
66. }
67.
68. int StringSound::time(void) {
69.     return _time;
70. }

```

Output -

There is no output to show visually because all the output is audio.

PS5 Recursive Graphics

Overview -

The purpose of this assignment was to make a Sierpinski Triangle using recursion. 2 arguments should be read from the command line (first being the side length of the base triangle, second being the recursion depth) and each iteration will add 3^n triangles to the base triangle, n being the current recursion depth.

Key Concepts and Lessons -

The main focus of this assignment was the power of recursion. With each iteration, I was able to call the function 3 more times (1 for each child triangle) and keep this going until the desired recursion depth was reached. One drawback to recursion is it may take up a lot of space on the stack and cause a stack overflow, so we had to practice tail-end recursion to avoid that happening. I had not used tail-end recursion before but it seems very efficient and useful for recursive problems.

Code files in this assignment: **Makefile**, **TFractal.cpp**, **Triangle.h**, **Triangle.cpp**

Makefile:

This is your average Makefile; nothing special about it.

```
1. CC = g++
2. CFLAGS = -Wall -Werror -std=c++11 -pedantic
3. SFML = -lsfml-system -lsfml-graphics -lsfml-window
4. OBJ = TFractal.o Triangle.o
5. EXE = ps5
6.
7. all: $(EXE)
8.
9. $(EXE): $(OBJ)
10.    $(CC) $(CFLAGS) -o $(EXE) $(OBJ) $(SFML)
11.
12. TFractal.o: TFractal.cpp
13.    $(CC) $(CFLAGS) -c TFractal.cpp -o TFractal.o
14.
15. Triangle.o: Triangle.cpp Triangle.h
16.    $(CC) $(CFLAGS) -c Triangle.cpp -o Triangle.o
17.
18. clean:
19.    rm $(EXE) *.o
```

TFractal.cpp:

Here, the `fTree()` function handles all the recursion. You load it with a `vector<Triangle>` and the starting values for the triangle size and desired recursion depth and it starts creating triangles until it reaches the desired recursion depth. The SFML window event loop then iterates through the `Vector<Triangle>` to draw the triangles. The initial triangle will always be at the center of the window.

```
1. // Copyright 2021 Ethan Yu
2. #include "Triangle.h"
3.
4. void fTree(std::vector<Triangle>& triangle_vector, // NOLINT
5.           sf::Vector2f center_coordinates, double length,
6.           int target_depth, int current_depth);
7.
8. int main(int argc, char* argv[]) {
9.     double base_size = atoi(argv[1]);
10.    int target_depth = atoi(argv[2]);
11.
12.    // Set up the window and get starting variables
13.    sf::RenderWindow window(sf::VideoMode(800, 600), "Sierpinski Triangle");
14.    int screen_x = window.getSize().x;
15.    int screen_y = window.getSize().y;
16.    sf::Vector2f starting_center((screen_x / 2), (screen_y / 2));
17.    std::vector<Triangle> triangle_vector;
18.
19.    while (window.isOpen()) {
20.        // Process events
21.        sf::Event event;
22.        while (window.pollEvent(event)) {
23.            // Close window: exit
24.            if (event.type == sf::Event::Closed) {
25.                window.close();
26.            }
27.
28.            // Call recursive function with starting arguments
29.            fTree(triangle_vector, starting_center, base_size, target_depth, 0);
30.
31.            window.clear(); // Clear screen
32.
33.            // Go through the vector and draw the triangles
34.            std::vector<Triangle>::iterator iter;
35.            for (iter = triangle_vector.begin(); iter != triangle_vector.end(); ++iter) {
36.                window.draw((*iter).get_vertex_array()); // Draw shape
37.            }
38.            window.display(); // Update window
39.        }
40.        return 0;
41.    }
42.
43. void fTree(std::vector<Triangle>& triangle_vector, // NOLINT
```

```

44. sf::Vector2f center, double length,
45. int target_depth, int current_depth) {
46.     // Calculate the triangle's height
47.     double height = (sqrt(3) * length)/2;
48.
49.     // Create and add the current triangle to the vector
50.     Triangle newTriangle(length, height, center);
51.     triangle_vector.push_back(newTriangle);
52.
53.     // Stop iterating
54.     if (current_depth + 1 > target_depth) {
55.         return;
56.     }
57.
58.     // Calculate the centers for the child triangles
59.     sf::Vector2f new_center_1(center.x - (length / 2),
60. center.y - (2 * height/3));
61.     sf::Vector2f new_center_2(center.x + (3 * length/4),
62. center.y - (height/6));
63.     sf::Vector2f new_center_3(center.x - (length/4),
64. center.y + (5 * height/6));
65.
66.     // Top triangle
67.     fTree(triangle_vector, new_center_1,
68. length/2, target_depth, current_depth + 1);
69.
70.     // Right triangle
71.     fTree(triangle_vector, new_center_2,
72. length/2, target_depth, current_depth + 1);
73.
74.     // Bottom Left triangle
75.     fTree(triangle_vector, new_center_3,
76. length/2, target_depth, current_depth + 1);
77. }
78.

```

Triangle.h:

The class declaration for Triangle. It needs to inherit from sf::Drawable in order to be drawn in the SFML window. I had to use Vector2f and VertexArray for creating and storing points for the triangles because it was the easiest way to put 2 values together.

```

1. // Copyright 2021 Ethan Yu
2. #ifndef _HOME_ETHAN_PS5_TRIANGLE_H_
3. #define _HOME_ETHAN_PS5_TRIANGLE_H_
4.
5. #include <iostream>
6. #include <cmath>
7. #include <vector>
8. #include <SFML/Graphics.hpp>
9. #include <SFML/System.hpp>
10. #include <SFML/Window.hpp>
11.

```

```

12. class Triangle : public sf::Drawable {
13. public:
14.     Triangle(double init_length, int init_height, sf::Vector2f center);
15.     inline sf::VertexArray get_vertex_array(void) const {
16.         return vertex_array;
17.     }
18.     virtual void draw(sf::RenderTarget &target, // NOLINT
19.         sf::RenderStates states) const;
20.
21. private:
22.     double length;
23.     double height;
24.     sf::Vector2f center;
25.     sf::VertexArray vertex_array;
26. };
27.
28. #endif // _HOME_ETHAN_PS5_TRIANGLE_H_

```

Triangle.cpp:

Surprisingly, triangles need 4 sides (or in this case, linestrips) to be made. The linestrip makes it so a line will be drawn from one point to another. All points are calculated from the center point of the triangle. Each point also is designated a color and when a line is drawn between points, it creates a color gradient on the line.

```

1. // Copyright 2021 Ethan Yu
2. #include "Triangle.h"
3.
4. // Triangle Value Constructor
5. Triangle::Triangle(double init_length, int init_height,
6.     sf::Vector2f init_center) {
7.     length = init_length;
8.     height = init_height;
9.     sf::Vector2f center(init_center);
10.
11.     // You need 4 points for the triangle's lines
12.     // 3 for the triangle's vertices, 1 wrap around back to the initial vertice
13.     // Line sequence: 0 -> 1, 1 -> 2, 2 -> 3 (3 = 0)
14.     vertex_array = sf::VertexArray(sf::LineStrip, 4);
15.
16.     // Top Left Vertice
17.     vertex_array[0].position = sf::Vector2f(center.x - (length/2),
18.         center.y - (height/3));
19.
20.     // Top Right Vertice
21.     vertex_array[1].position = sf::Vector2f(center.x + (length/2),
22.         center.y - (height/3));
23.
24.     // Bottom Vertice
25.     vertex_array[2].position = sf::Vector2f(center.x, center.y +
26.         (2 * height/3));
27.

```

```

28.         // Back to Top Left Vertice
29.         vertex_array[3].position = sf::Vector2f(center.x - (length / 2),
30.         center.y - (height/3));
31.
32.         // Set the colors for each vertice
33.         // Will create a gradient in the line
34.         vertex_array[0].color = sf::Color::Red;
35.         vertex_array[1].color = sf::Color::Blue;
36.         vertex_array[2].color = sf::Color::Green;
37.         vertex_array[3].color = sf::Color::Red;
38.     }
39.
40. // Triangle Draw Function
41. void Triangle::draw(sf::RenderTarget &target, sf::RenderStates states) const {
42.     target.draw(vertex_array, states);
43. }
44.

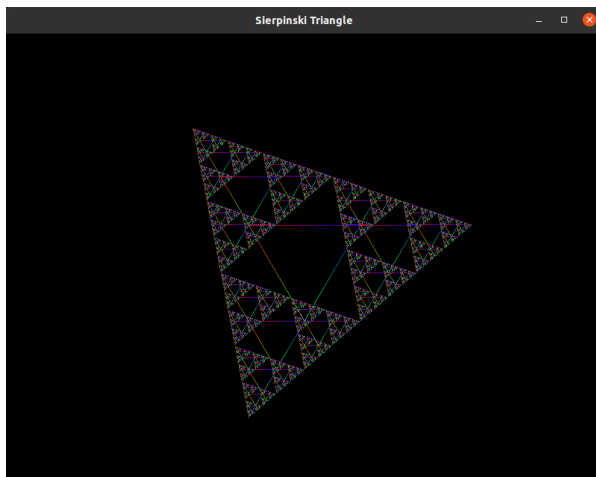
```

Output -

```

ethan@ethan-VirtualBox:~/ps5$ ls
Ethan_Yu_Sierpinski_Triangle_Screenshot.png  ps5-readme.txt  Triangle.cpp
Makefile                                     TFractal.cpp    Triangle.h
ps5                                           TFractal.o      Triangle.o
ethan@ethan-VirtualBox:~/ps5$ ./ps5 150 7
Setting vertical sync not supported

```



Above: The command line argument generates the Sierpinski Triangle seen in the window.

PS6 Kronos Time Clock

Overview -

With this assignment, we find the time it takes for a Kronos inTouch system to boot up. To do this, we read line by line for special startup beginning and ending lines and then we calculate the time taken between these lines. What we also do is copy the input file and output it as a different file type by appending .rpt to the end of the original file name.

Key Concepts and Lessons Learned -

One big tool used in this assignment was regular expressions. They allow us to search for matches by using search parameters and specific criteria based on the content of the line. I had never used regular expressions before this but they were very useful although they can be hard to set up and hard to decipher if you do not know what you're looking for.

Code files in this assignment: **Makefile**, **main.cpp**

Makefile:

A basic Makefile except we need to add `-lboost_regex` in order to use regular expressions and `-lboost_date_time` in order to use posix and gregorian class functions.

```
1. CC = g++
2. CFLAGS = -Wall -Werror -std=c++11 -pedantic
3. OTHER_FLAGS = -lboost_regex -lboost_date_time
4. OBJ = main.o
5. EXE = ps6
6.
7. all: $(EXE)
8.
9. $(EXE): $(OBJ)
10.    $(CC) $(CFLAGS) -o $(EXE) $(OBJ) $(OTHER_FLAGS)
11.
12. main.o: main.cpp
13.    $(CC) $(CFLAGS) -c main.cpp -o main.o $(OTHER_FLAGS)
14.
15. clean:
16.    rm $(EXE) *.o
17.
```

main.cpp:

First, the input file is taken and the output file is created and every line read from the input file is written into the output file. While reading from the input file, if a line matches the regex for beginning startup, then we take note of the timestamp of the line and look for a line matching the regex for the ending of startup. If we find a pair of starting and ending lines, then we return the success and how long it took for it to startup. If we encounter another starting line, then we say the first one failed and start a new start one. If we reach the end of file before finishing a startup, then we say that one failed too.

```
1. // Copyright 2021 Ethan Yu
2. #include <iostream>
3. #include <fstream>
4. #include <string>
5. #include "boost/date_time/gregorian/gregorian.hpp"
6. #include "boost/date_time/posix_time/posix_time.hpp"
7. #include <boost/regex.hpp>
8.
9. using boost::gregorian::date;
10. using boost::gregorian::from_simple_string;
11. using boost::gregorian::date_period;
12. using boost::gregorian::date_duration;
13. using boost::posix_time::ptime;
14. using boost::posix_time::time_duration;
15.
16. int main(int argc, char* argv[]) {
17.     std::string fileName = argv[1];
18.     const std::string inputFileName(fileName);
19.     std::ifstream inputFile{ fileName };
20.     fileName = fileName + ".rpt";
21.     const std::string outputFileName(fileName);
22.     std::ofstream outputFile{ fileName };
23.
24.     boost::regex
25.     startExpr{"^([\\d]{4})\\-([\\d]{2})\\-([\\d]{2})\\s([\\d]{2}):([\\d]{2}):([\\d]{2})\\.\\s((log\\.c\\.166\\) server
26.     started)\\s$"}; // NOLINT
27.
28.     boost::regex
29.     endExpr{"^([\\d]{4})\\-([\\d]{2})\\-([\\d]{2})\\s([\\d]{2}):([\\d]{2}):([\\d]{2})\\.([\\d]{3}):INFO:oejs.Abstra
30.     ctConnector:Started SelectChannelConnector@0\\.0\\.0\\.0:9080$"}; // NOLINT
31.
32.     boost::smatch matches;
33.
34.     int startUpInProgress = 0;
35.     std::string line;
36.     int lineNum = 0;
37.     int startYear, startMonth, startDay, startHour, startMinute, startSecond;
38.     int endYear, endMonth, endDay, endHour, endMinute, endSecond;
39.     ptime startTime, endTime;
40.
41.     auto findElapsedTime = [](ptime startTime, ptime endTime) {
42.         time_duration elapsedTime = endTime - startTime;
```

```

37.         std::cout << "Elapsed Time: " << elapsedTime.hours() <<
38.         ":" << elapsedTime.minutes() << ":" <<
39.         elapsedTime.seconds() << "\n\n";
40.     };
41.
42.     if (inputFile && outputFile) {
43.         while (getline(inputFile, line)) {
44.             lineNum = lineNum + 1;
45.             outputFile << line << std::endl;
46.
47.             if (boost::regex_match(line, matches, startExpr)) {
48.                 if (startUpInProgress) {
49.                     std::cout << "Status: Failure \n\n";
50.                 }
51.
52.                 startUpInProgress = 1;
53.                 startYear = std::stoi(matches.str(1));
54.                 startMonth = std::stoi(matches.str(2));
55.                 startDay = std::stoi(matches.str(3));
56.                 startHour = std::stoi(matches.str(4));
57.                 startMinute = std::stoi(matches.str(5));
58.                 startSecond = std::stoi(matches.str(6));
59.                 date startDate(startYear, startMonth, startDay);
60.                 ptime start(startDate, time_duration(startHour, startMinute, startSecond, 0));
61.                 startTime = start;
62.
63.                 std::cout << "Starting Up on Line " << lineNum << std::endl;
64.                 std::cout << "Timestamp: " << startYear << "-" << startMonth
65.                 << "-" << startDay << " " << startHour << ":" <<
66.                 startMinute << ":" << startSecond << std::endl;
67.             } else if (boost::regex_match(line, matches, endExpr)) {
68.                 if (startUpInProgress) {
69.                     std::cout << "Status: Success \n";
70.
71.                     endYear = std::stoi(matches.str(1));
72.                     endMonth = std::stoi(matches.str(2));
73.                     endDay = std::stoi(matches.str(3));
74.                     endHour = std::stoi(matches.str(4));
75.                     endMinute = std::stoi(matches.str(5));
76.                     endSecond = std::stoi(matches.str(6));
77.                     date endDate(endYear, endMonth, endDay);
78.                     ptime end(endDate,
79.                             time_duration(endHour, endMinute, endSecond, 0));
80.                     endTime = end;
81.
82.                     findElapsedTime(startTime, endTime);
83.                 }
84.                 startUpInProgress = 0;
85.             }
86.         }
87.         if (startUpInProgress) {
88.             std::cout << "Status: Failure \n\n";
89.         }
90.     } else {
91.         std::cout << "There was a problem with the files \n";
92.     }

```

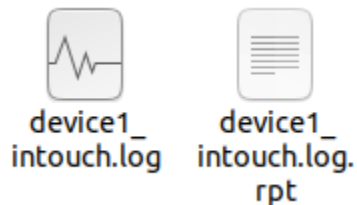


```
93.  
94.     inputFile.close();  
95.     outputFile.close();  
96.  
97.     return 0;  
98. }
```

Output -

```
ethan@ethan-VirtualBox:~/ps6$ ./ps6 device1_intouch.log  
Starting Up on Line 435369  
Timestamp: 2014-3-25 19:11:59  
Status: Success  
Elapsed Time: 0:3:3  
  
Starting Up on Line 436500  
Timestamp: 2014-3-25 19:29:59  
Status: Success  
Elapsed Time: 0:2:45  
  
Starting Up on Line 440719  
Timestamp: 2014-3-25 22:1:46  
Status: Success  
Elapsed Time: 0:2:41  
  
Starting Up on Line 440866  
Timestamp: 2014-3-26 12:47:42  
Status: Success  
Elapsed Time: 0:2:47
```

Above: Partial terminal output when running the supplied device1_intouch.log file through the program (full output was too long to capture in photo)



Left: The input file is copied over to a new file but with .rpt at the end