Lab 1 – Steganography

Algorithm:

* **Prime Iterator**
  + The PrimeIterator follows the Sieve of Eratosthenes method of finding prime numbers. The algorithm begins during the construction of the PrimeIterator object, which takes in a positive integer that represents the maximum number the program will go reach to find the prime numbers. Next, a Boolean array of the max number size is created, with each index representing a number in the array of numbers from 0 to the max number. Then, for each element in the Boolean array starting from index 2, if the Boolean at the specified index is set to false, then each element whose index is a multiple of the current number is set to true (In this case, true means that the number the element represents is divisible by other numbers than itself and 1). This continues until the square root of the max number is reached, indicating that every prime number has been determined.
  + Once each prime number is determined, the program goes through the Boolean array and counts the number of times “false” appears (In this case, false represents numbers not divisible by number other than itself and 1). This count is used to instantiate an integer array that contains the prime numbers. This array is filled by looping through the Boolean array and finding all of the “false” elements, using the index of the Boolean array plus 1 to fill in each index of the integer array of prime numbers.
* **Steganog**
  + **Embedding:**

When embedding into an image, the method first takes in two inputs, the Picture object containing the image and a string representation of the message we want to embed. First, the algorithm determines the number of prime-numbered pixels based on the max number of pixels the image has. Then, while the character index for the message is less than the length of the message AND there is still an available prime number, we encode the message into the image, character by character. This is done by getting a character from the message at the specified index. This character is then converted into a 6-bit binary value whose decimal value is 32 less than the character’s actual ASCII value. Next, the pixel affected is determined by modding the current prime number by the image’s width and dividing the current prime number by the image’s height to get the location of the pixel in the image. The RGB values of that pixel are then converted into 8-bit binary strings. Each value’s last two bits are then replaced by two bits of the character’s binary (Red gets the first two bits, green gets the middle two bits, and then blue gets the last two bits). The new RGB values are then put together to make a new color, which is then set as the current pixel’s new color. The message-embedded image is then returned by the algorithm.

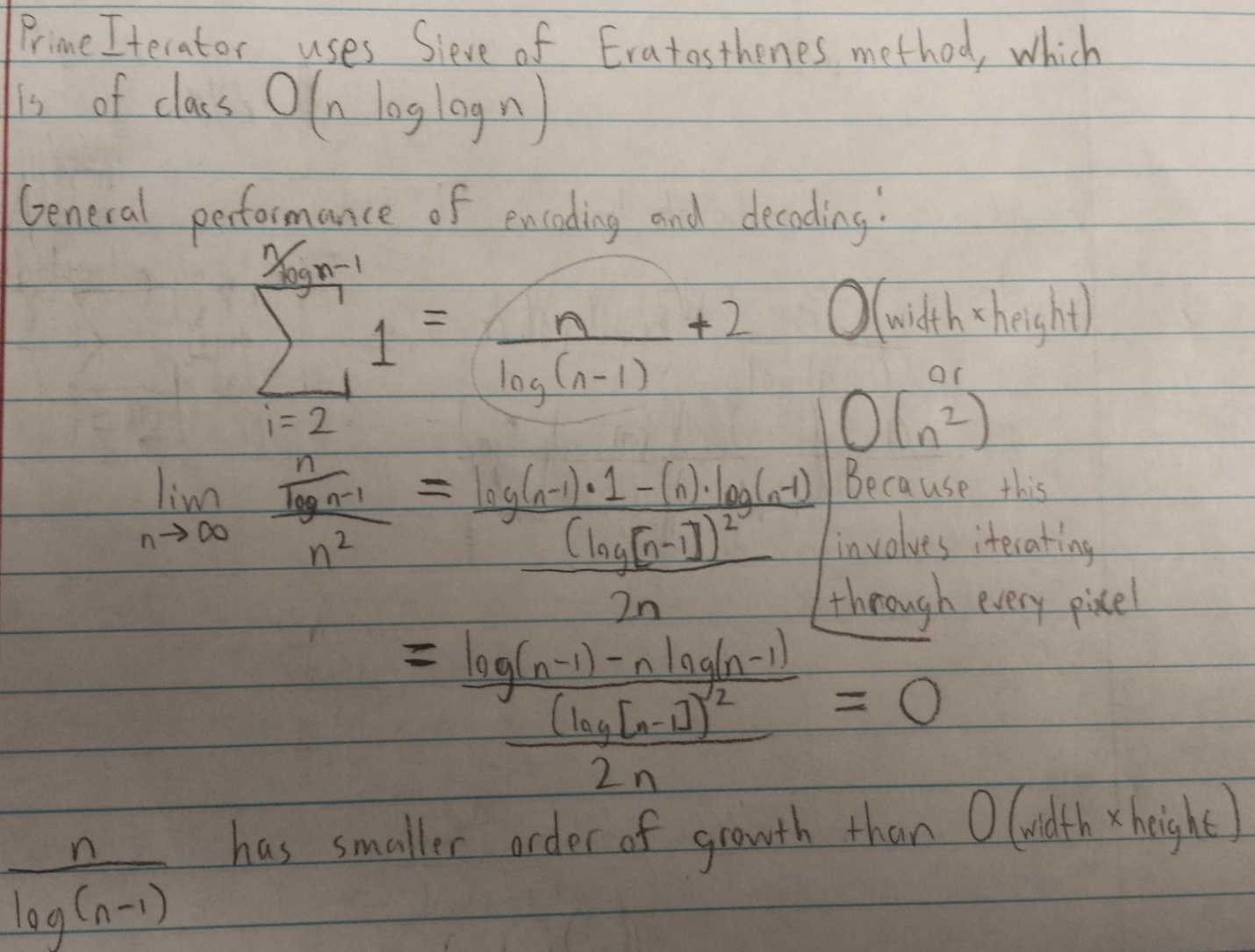
* + **Decoding:**

When retrieving a message from an image, the method takes in only the Picture with the image we need to decode. First, the algorithm determines the number of prime numbers based on the size of the image. Then, while there is still an available prime number, the algorithm will decode the message, character by character. In this loop, the algorithm first gets the color of the current prime-numbered pixel based on its x and y position in the image by modding by the width and dividing by the height, respectively. Then, each RGB value of that pixel is converted into an 8-bit binary number. The binary number of the hidden character is then retrieved by hijacking the last two bits of each RGB value and combining them into one binary string. This is converted into an int, add by 32, then converted into its corresponding character by using the integer as an ASCII value to find the character. This character is then concatenated into a String object to build up the message. The completed message is then returned.

Help from other students and classmates

* I asked some students if they could help me understand how the Sieve of Eratosthenes worked in general. I figured out how to implement the sieve in code, but I needed help understanding how the process worked in general before I could actually code the algorithm.
* I had one student help me understand how the Picture class is arranged in order to get the pixels we need. I was having trouble understanding how to get each pixel when the Picture’s get() method required two ints, one for *X* and one for *Y*. He helped me understand how the X represents a “column” and the Y represents each “row” in the Picture’s array of pixels. This helped me understand how to directly retrieve the pixels I wanted without having to loop through each pixel until I got a prime-numbered pixel.

Mathematical Analysis:



Empirical Analysis:

I chose these experiments because these variables are what affect the overall speed of the algorithm, which is displayed in milliseconds at the far right of the table. By consistently increasing the overall size of the image, I can determine how the algorithm handles larger and larger images and number of primes. I also learn how the algorithm handles message length and if the length has any real impact on the time needed to complete the algorithm.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Width | Height | Max # of Pixels | Square Root of Max | # of Primes | Message Length | Completion  Time (Encoding and decoding in milliseconds) |
| 100 | 100 | 10000 | 100 | 1229 | 67 | 28 ms |
| 200 | 200 | 40000 | 200 | 4203 | 67 | 55 ms |
| 300 | 300 | 90000 | 300 | 8713 | 67 | 90 ms |
| 400 | 400 | 160000 | 400 | 14683 | 67 | 163 ms |
| 500 | 500 | 250000 | 500 | 22044 | 67 | 272 ms |
| 100 | 100 | 10000 | 100 | 1229 | 1000 | 36 ms |
| 200 | 200 | 40000 | 200 | 4203 | 1000 | 60 ms |
| 300 | 300 | 90000 | 300 | 8713 | 1000 | 96 ms |
| 400 | 400 | 160000 | 400 | 14683 | 1000 | 173 ms |
| 500 | 500 | 250000 | 500 | 22044 | 1000 | 276 ms |