Drew Hurdle
July 6, 2014
Algorithms and Data Structures I
Professor Cummings

Lab 1
Steganog

Section 1: How it works

The algorithm is broken into two parts: Prime Generation and Steganography.

Prime Generation

The generation of prime numbers is handled by an algorithm based on the Sieve of Eratosthenes.

n is equal to width * height of the image.
When the PrimeIterator is instantiated, it creates the following fields:

- primeBools[]
  - A boolean array of n values. Java initializes all these values to false by default, so I use true as being “marked out.” Because we know we won’t need 0 or 1, the bool at index i represents the integer of i + 2.
- nextPrime
  - An int which stores the next prime to be given via the next() method. This starts at 2.
- primeWalkerIndex
  - An int used to keep track of where in the primeBools array the program is when it calculated the next prime. This starts at 0.

Getting the next prime

When an instance of PrimeIterator is asked for the next prime, it first marks out all the multiples of the old nextPrime. (See Marking out multiples below)
It then finds the prime after the one it is about to return.
This is done by walking through the boolean array and finding the index of the next boolean not yet marked out.
The old value stored as nextPrime is returned and is replaced by the new value.
If no values are found, the old value stored as nextPrime is still returned, but nextPrime is set to -1 to indicate that there are no more primes under the maximum value passed in at construction.

Checking if there is a next prime

The method hasNext() returns whether or not another prime exists below the maximum value passed in at construction. This works by simply testing if the nextPrime value is equal to -1. nextPrime is only set as -1 if there are no more primes under the max value. Otherwise, it is set as the next prime to be returned.

Marking out multiples

Marking out multiples is done efficiently. First, an int representing the current index is created and set to prime^2 - 2.

The prime is squared because all the multiples of any prime number p less than p^2 have already been marked out. This is because all numbers are products of primes (Fundamental theorem of arithmetic), and all the primes lower than p have already been found, and their multiples have thus already been marked out. Since any number lower than p^2 must have at least one integer factor less than p, we can be certain it will have been marked out.

After setting the starting value of the current index, we perform the following until the current index is greater than the length of the array of booleans:

Mark out the boolean at the current index.

Add the prime to the current index.

Steganography

The Steganography half of the program involves writing and reading to and from specific bits stored in the color of pixels of an image file.

Writing

Writing to an image works in the following manner.

•A Picture to write to and a String to be written are passed in.
•All the characters in the String are set to their upper-case equivalents.
•The end string (a string to indicate to the reader that the end of the message has been found) is appended to the end of the string.
•The meta-data string is calculated…
- A String is created with the meta data header (a string to indicate that the file has meta data).
- The number of total characters of which the message is comprised (including the length of the meta-data and the end string) is calculated.
- This number is masked and converted to five characters. This will allow 30 bits for the value.
- These characters are appended to the end of the meta-data string.

•The meta-data string is appended to the beginning of the string to be written.
•The number to pass as the max to the Prime Iterator is approximated with n (log n + log log n) where n is the number of characters in the entire string. (This returns a value higher than the actual number of primes needed for every integer over 6. Since my metadata string is longer than 6 characters by itself, we need not worry about cases where n is under 6).
•An instance of PrimeIterator is created with a max set to the calculated value.
•For each character in the string to be written…
- The character is cast to an integer.
- 32 is subtracted from the character integer.
- The next prime is retrieved from the PrimeIterator.
- This value is converted into x and y coordinates.

- The current color of the pixel is retrieved from the image as an int containing Alpha, Red, Green, and Blue.
- This color is masked with & 0xFFFCFCFC to remove the last two bits from the red, green, and blue values.
- The integer representing the character is masked…
  - By & 0x30 and shifted << 12 for Red
  - By & 0xC and shifted << 6 for Green
  - By & 0x3 and not shifted for Blue.
- The integer to be combined with the current color integer is created by performing an | bitwise operation on all three of the resulting values from the character masking and shifting.
- The new color to be placed at the x and y values is created by performing an | bitwise operation on the masked original color.
- This new color is set as the color for the pixel at position x, y.

Reading

Reading from the image works in the following manner.

- A Picture to be read is sent in.
- An integer called primeMax is set to the image's width * the image's height.
- An integer called stringBuilderInitialCapacity is set to 16.
  - NOTE: This will only be used if no meta-data is found.
- The meta-data is read and returns the number of characters in the message if meta-data is found and 0 otherwise.
  - The meta-data is read by walking along the first metaDataString.length() primes and checking if the characters in each position are the same as the meta-data string.
    - If they are not the same, 0 is returned.
    - Otherwise…
      - And integer representing the value stored in the next five pixels is created and set to 0.
      - For the next five primes…
        - The Red, Green, and Blue values at the position are retrieved from the image.
        - The values are masked with & 0x3.
        - The iterationShift value is calculated by 6 * (4 - i).
        - The red, green, and blue values are shifted by iterationShift + 4, 2, and 0 respectively.
        - The integer representing the number stored is updated by performing an | operation with each of the shifted and masked integers from the color bits.
- If the meta-data was found, the primeMax integer is set n (log n + log log n) where n is the value it returned. stringBuilderInitialCapacity is also set to this value.
- A StringBuilder with an initial size of stringBuilderInitialCapacity is created.
- A PrimeIterator with a max of primeMax is created.
- If the meta-data was found, the Prime Iterator's next function is called (metaDataString.length + 5) times to get past the meta-data.
- An integer called currentEscapeStringIndex is created and set to 0.
- While the PrimeIterator has a next value and the escape string has not been found…
  - The next prime is retrieved from the prime iterator.
  - The x and y coordinates of the prime are calculated.

    - The red, green, and blue values at the x, y coordinates are retrieved.
    - These are masked with & 0x3 to reduce them to the last two bits.
    - The red is shifted << 4 and the green is shifted << 2.
    - A bitwise I operation is performed on the three values.
    - 32 is added to the result.
    - The value is cast to a char.
    - The char is added to the StringBuilder.
    - If the char is the same as the char at the (currentEscapeStringIndex) position of the escape string, the value of currentEscapeStringIndex is incremented by 1.
    - Otherwise the value of currentEscapeStringIndex is set to 0.
    - If the value of currentEscapeStringIndex is equal to the length of escapeString…
        - The escape string has been found and breaks the loop.
- The string-builder's string is converted to a String.
- If the escape string was found, the converted string's last (escapeString.length) characters are removed.

Section 2 : Help Received

I talk a lot about assignments with friends, and this lab was no different. I discussed it most with Jon Pearl. Before I had started coding, I asked him what his thoughts on performing the conversion of integers to colors were. I suggested some method of subtracting values. He told me he had done it with bitmasking. As I hate to be out-done by Jon (as frequently as it happens) I decided to brush up on bitwise operations too.

Jon and I also argued about the best way to perform the Sieve code, and he gave me the idea of approximating the value of the nth prime to limit the array size of the PrimeIterator for write. I expanded this idea with meta-data and applied it to reading too.

Past that, I received no other significant help.

Section 3 : Analysis

Without the generation of primes, my algorithm is linear. O(n).

The generation of primes is contained within the rest of the algorithm, as the primes are not generated until next() is called.

The Sieve of Eratosthenes performs in (n log log n) time where n is the maximum value (the size of the array). This is because the Prime Harmonic Series approaches log log n and the function is called n times.

However, since our function is only called the message.length number of times, and n is j (log j + log log j) (where j is the message length), the complexity of the algorithm is…

O(n log log (n (log n + log log n)))

*where n is the message length

Which is equivalent to…

O(n log(log(n log(n log(n))))

*where n is positive (which it always is).

As for proving that this is better than O (width * height), the message length can never be greater than the number of primes under (width * height). The number of primes under (width * height) can be approximated as (width * height) / log (width * height).

Thus, n <= (width * height) / log (width * height).

Width * height can be thought of as the number of pixels, hereafter referred to as p.

This means that, if the message is as long as it can be while still fitting in the image, the performance is…

(p / log p) log(log((p / log p) log((p / log p) log((p / log p)))).

The limit of

$$\frac{(p / \log p) \log(\log((p / \log p) \log((p / \log p) \log((p / \log p))))}{p}$$

as p approaches infinity is 0.

Section 4 - Empirical Testing

Testing was performed on a 2013 Macbook Air i7 with 8 GB of RAM running OS X. I forced the JIT to compile to machine code before testing. Testing was performed by comparing System.currentTimeMillis() before and after running the code 4,000 times. Recorded times include only the algorithm runtime and not loading nor saving the images from the hard drive.

I decided on these techniques for testing for the folioing reasons:

- Averaging over many runs gives more accurate results
- Without forcing JIT to compile, averaged results are inaccurate
- System.currentTimeMillis was used because it's very cheap and reliable
- Loading and saving images to the hard drive would vary depending on the hard drive speed and current activity and wasn't a part of my algorithm

Averaged Algorithm Runtime

Test number	Image height	Image width	Character count	Write time (ms)	Read Time (ms)
1	128	128	10	0.0065	0.00625
2	128	128	350	0.088	0.04275
3	600	600	10	0.00825	0.00725
4	600	600	350	0.06475	0.0375
5	4000	6016	10	0.027	0.0065
6	4000	6016	350	0.091	0.04575
7	4000	6016	43687	8.9535	5.23575
8	4000	6016	175,071	39.04875	23.41475