Project 1 Report

Paul Chon

September 2020

Note:

I mainly used Intellij to code, but I used Eclipse for the JUnit testing and exporting the project. Also, the instructions recommended to use System.currentTimeMillis(), but for Part B it kept on returning 0 ms, so, instead, I used System.nanoTime(), which records in nano seconds or ns.

Part A: Data Shuffling

OBJECTIVE:

The objective of Part A is to shuffle a dataset of Erdos Numbers using the Fisher-Yates algorithm, which is shown in Figure 1 and has a running time of O(n) and a space complexity of O(1). Also, pseudo-random numbers have to be created in O(1) running time when you generate random numbers for the algorithm.

```
-- To shuffle an array a of n elements (indexes 0..n-1)
for i=n-1 downto 1 //index starts from 0
      j = random integer within 1 <= j <= i
      exchange a[j] and a[i]
```

Figure 1: Fisher-Yates Algorithm Pseudocode

IMPLEMENTATION:

The Big-O Notation of this program is probably O(n) because I use for loops, but there are no nested for loops. The actual dataset is in a txt file called ErdosCA.txt and it has two columns (the left column is like an ID number for each individual person and the right column is their Erdos Number) and 7,516 rows, but the first line is just metadata, so just 7,515 rows of data. Looking at the Fisher-Yates Algorithm from before, n would equal the numbers of rows, 7,515, but we also need to generate random numbers, so there is a random

number generator using a seed of 20 and I used nextInt. Now the algorithm can be implemented, which is seen in Figure 2.

```
Random r = new Random( seed: 20);
time = System.nanoTime();

for (int i = data.length - 1; i > 0; i--) {
    int j = r.nextInt(i);
    String temp = data[j];
    data[j] = data[i];
    data[i] = temp;
}
```

Figure 2: Fisher-Yates Algorithm Implemented

To be able to use the algorithm at all, I created a constructor, as seen in Figure 3, that gets input from a file, which would be ErdosCA.txt, using BufferedReader and outputs it into a String array without the first line (I did this by doing a readLine() without saving it to any variables), since the first line is just metadata. Also, since the constructor is using BufferedReader, I made it "throw IOException" to handle that exception.

```
public DataShuffling(File file) throws IOException {
    time = System.nanoTime();
    data = new String[7515];
    BufferedReader br = new BufferedReader(new FileReader( fileName: "ErdosCA.txt"));
    br.readLine();
    for (int i = 0; i < data.length; i++) {
        data[i] = br.readLine();
    }
    br.close();
    time = System.nanoTime() - time;
    System.out.println("Time to read from file (in ns): " + time);
}
```

Figure 3: Input of ErdosCA.txt implemented in constructor

To output the shuffled dataset, I used PrintWriter, instead of just BufferedReader because with PrintWriter you can use the print() and println() functions, which is much more convenient than readLine() from BufferedReader. Also, when creating the FileWriter, I set the append to true, but because of that I created a PrintWriter, as seen in Figure 4, which clears the file or else the output keeps on getting added each time it's run. Figure 4 shows the implementation.

```
PrintWriter clear = new PrintWriter( fileName: "ChonPaulShuffled.txt");
PrintWriter pw = new PrintWriter(new BufferedWriter(
        new FileWriter( fileName: "ChonPaulShuffled.txt",  append: true)));

for (String s : data) {
    pw.println(s);
}

time = System.nanoTime() - time;
System.out.println("Time to write to file (in ns): " + time);
pw.close();
```

Figure 4: Implementing Output

TESTING:

To test the program, I first used @BeforeEach to create a new DataShuffling class that inputs the ErdosCA.txt file before each run of the test, which can be seen in Figure 5. Then, in the actual @Test, I call the program, shuffle(), and I compared the output of the program to the expected output using BufferedReader, which is Target2.txt, since I used nextInt() to generate random numbers.

```
@BeforeEach
public void setUp() throws IOException {
    ds = new DataShuffling(new File( pathname: "ErdosCA.txt"));
}
```

Figure 5: Before each test, create a new DataShuffling class that inputs the ErdosCA.txt file

```
@Test
public void testShuffle() throws IOException {
    ds.shuffle();
    String expectedLine;
    BufferedReader In = new BufferedReader(new FileReader( fileName: "Target2.txt"));
    BufferedReader Out = new BufferedReader(
            new FileReader( fileName: "ChonPaulShuffled.txt"));

    while ((expectedLine = In.readLine()) != null) {
        String actualLine = Out.readLine();
        assertEquals(expectedLine, actualLine);
    }

    Out.close();
    In.close();
}
```

Figure 6: Comparing the output of the program to the expected output

TIMING:

As I said previously, I used nanoTime() rather than currentTimeMillis(), so I got much larger numbers rather than just using currentTimeMillis(). The times of 10 different runs, and the averages can be seen in Figure 7.

Time to read from file (in ns)
5012000
7200100
5285400
1.21299E7
7412000
5169900
9309500
6052400
8802400
5158000

Time to shuffle (in ns)
1745500
1147900
1518900
2376200
1658500
2606800
1874600
1042300
1526800
1724100

Time to write to file (in ns)
1.02484E7
1.08817E7
1.03447E7
1.32449E7
1.1166E7
1.06927E7
1.00733E7
8415600
1.12886E7
1.21951E7

	Average (in ns)
Time to read from file (in ns)	7153160
Time to shuffle (in ns)	1722160
Time to write to file (in ns)	10855100

Figure 7: Time to read from file, shuffle, and write to file

Part B: Prisoner Game

OBJECTIVE:

The objective of Part B is to use a circular singly-linked list to recreate a prisoner game where prisoners, every certain number of steps are eliminated until one prisoner is remaining, who is given freedom, and an example of this can be seen in Figure 8.

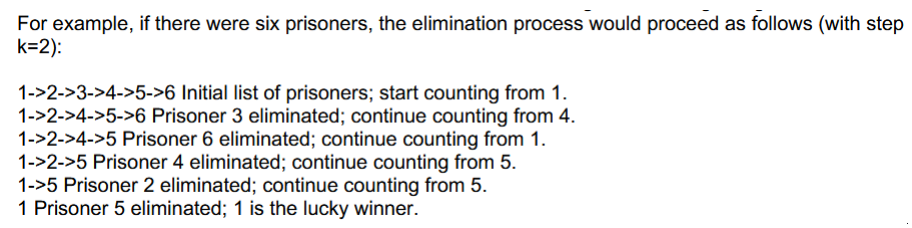

Figure 8: Example game with 6 prisoners and eliminating every other two prisoners

IMPLEMENTATION:

The Big-O Notation of this program is probably O(n^2) because I use the freedom() method in Figure 11 which has a for loop inside of a while loop. First, I created a Node class, a CircularLinkedList class, and a CircularLinkedListGame class to create the linked list class and run the game. As shown in Figure 9, the constructor in CircularLinkedListGame initializes private instance variables and also creates a new and empty linked list.

```
public CircularLinkedListGame(int prisoners, int step) {
    this.prisoners = prisoners;
    this.step = step;
    this.prisonerGame = new CircularLinkedList();
}
```

Figure 9: Constructor that creates a new and empty linked list

In addition, in the CircularLinkedListGame class, I created a method, insertPrisoners(), as seen in Figure 10, that adds number of prisoners specified by the user into the linked list. To find the winner, I created a method, freedom(), as seen in Figure 11, that will find the winner by removing prisoners every certain number of steps until the head of the linked list equals the tail of the linked list, which means the size of the linked list is 1, and the winner is found. Also, the remove method in the CircularLinkedList class that was used to remove prisoners can be seen in Figure 12.

```java
public void insertPrisoners() {
    time = System.nanoTime();
    //
    for (int i = 0; i < prisoners; i++) {
        prisonerGame.add(new Node( data: i + 1));
    }
    time = System.nanoTime() - time;
    System.out.println("Time to create list (in ns): " + time);
}
```

Figure 10: Adds nodes into the linked list matching the number of prisoners

```java
public int freedom() {
    Node eliminatedPrisoner = prisonerGame.getHead();
    int removeCounter = 0;
    time = System.nanoTime();
    while (prisonerGame.getHead() != prisonerGame.getTail()) {
        for (int i = 1; i <= step; i++) {
            eliminatedPrisoner = eliminatedPrisoner.getNext();
        }

        time = System.nanoTime();
        prisonerGame.remove(eliminatedPrisoner);
        if (removeCounter == 0) {
            time = System.nanoTime() - time;
            System.out.println("Time to delete one node (in ns): " + time);
        }
        removeCounter++;
        eliminatedPrisoner = eliminatedPrisoner.getNext();
    }
    time = System.nanoTime() - time;
    System.out.println("Time to find winner (in ns): " + time);

    return prisonerGame.getHead().getData();
}
```

Figure 11: Finds the winner of the prisoner game

```java
public void remove(Node removeMe) {
    Node prev = tail;

    while (prev.getNext().getData() != removeMe.getData()) {
        prev = prev.getNext();
    }

    if (prev.getNext() == tail) {
        prev.setNext(head);
        tail = prev;
    } else if (prev.getNext() == head) {
        head = head.getNext();
        tail.setNext(head);
    } else {
        prev.setNext(prev.getNext().getNext());
    }
}
```

Figure 12: Removes a specific node from the linked list

TESTING:

To test the program, five test cases were given: n = 6 and k = 2, n = 1 and k = 9, n = 7 and k = 7, n = 12 and k = 8, and n = 5 and k = 1. I put the n's and k's in separate integer arrays and used a for loop to go through each n and k pair, which is one test case. As Figure 13 shows, in the for loop, I tested that after creating the CircularLinkedListGame class, that the CircularLinkedList was empty, I tested that after adding the prisoners that the CircularLinkedList was not empty, and that after finding the winner with the freedom() method that the size of the CircularLinkedList was 1 and that the winner was correct.

```java
@Test
public void playGame() {
    int[] n = new int[]{6, 1, 7, 12, 5};
    int[] k = new int[]{2, 9, 7, 8, 1};
    int[] output = new int[]{1, 1, 4, 2, 3};

    for (int i = 0; i < n.length; i++) {
        game = new CircularLinkedListGame(n[i], k[i]);
        assertTrue(game.isEmpty());
        assertEquals( expected: 0, game.getCircularLinkedList().getSize());

        game.insertPrisoners();
        assertFalse(game.isEmpty());
        assertEquals(n[i], game.getCircularLinkedList().getSize());

        game.freedom();
        assertEquals( expected: 1, game.getCircularLinkedList().getSize());
        assertEquals(output[i], game.getCircularLinkedList().getHead().getData());
    }
}
```

Figure 13: Tests the program with five test cases

As a sidenote, I didn't call prisonerGame, which was the CircularLinkedList, for some of the assertions because the isEmpty() (shown in Figure 14), which is in the CircularLinkedListGame class, calls the getSize() method of the CircularLinkedList class (shown in Figure 15) and checks whether it is equal to 0. Now, if I wanted to directly call the getSize() method I implemented a method, getCircularLinkedList(), that returns the linked list, so I could call the getSize() method

```
public boolean isEmpty() {
    return prisonerGame.getSize() == 0;
}
```

Figure 14: Checks that the linked list has a size of 0

```
public int getSize() {
    size = 0;

    if (head == null || tail == null) {
        return 0;
    } else if (head == tail) {
        return 1;
    } else {
        Node temp = head;
        //TODO
        do{
            temp = temp.getNext();
            size++;
        }
        while (temp != head);
        return size;
    }
}
```

Figure 15: Function in CircularLinkedList that gets the size of the linked list

TIMING:

As I said previously, I used nanoTime() rather than currentTimeMillis(), so I got much larger numbers rather than just using currentTimeMillis(). Unlike in Part A, for Part B, I only ran the test twice because the test runs the program five times, so running the test twice would get 10 numbers, except for the time to delete one node because one of the test cases only has one node, so that one only has 8 numbers. The times of 2 different runs, and the averages can be seen in Figure 16.

Time to create list (in ns)
524800
1500
8000
4500
4300
531800
1100
2000
3700
2400

Time to delete one node (in ns)
3400
1200
2200
1100
2600
900
2400
1100

Time to find a winner (in ns)
800
500
2900
1300
700
600
300
2400
800
800

	Average (in ns)
Time to create list (in ns)	108410
Time to delete one node (in ns)	1862.5
Time to find winner (in ns)	1110

Figure 16: Time to create list, delete one node, and find a winner