Spring
2016

Assignment 7

109561770
MOHAMAD MAHDI ZIAEE

Contents

Description

This assignment contains two major tasks as follow:

1. Exon region mask complement
2. read mapping and read count

Each of the tasks above have their own dedicated application and they must be executed in the same order. All the application files must be placed under `Assignment7/io` folder. The application names are:

1. `/Assignment7/src/com/bio/main/ExonRegionMaskComplementApp.java`
2. `/Assignment7/src/com/bio/main/MappabilityApp.java`

The output of the first application execution must be used as an input to BowTie application with parameters v=3 and m=1. Then, the output of BowTie must be placed under the same `io` folder and together with the rest of files needed, the second application must be executed.

Application specifications

Programming language:	JAVA
Other required installations:	JRE 1.8 to run the application, JDK 1.8 to compile the application.
Version Control:	Git (GitHub) - https://github.com/momazia/Bioinformatic/tree/master/Assignment7
IDE:	Eclipse (Mars 4.5.1)
Build automation tool:	Maven (Refer to `/Assignment7/pom.xml` for the list of the libraries used)

Source code

com.bio.main.ExonRegionMaskComplementApp

```
package com.bio.main;

import java.io.IOException;

import com.bio.util.ExonMaskUtils;
import com.bio.util.FileUtils;

/**
 * The main application to be executed for the task 1 of the assignment in which it will replace N for all the non Exon parts by reading the Exon
 * annotation file, chromosome 1 file. The result of the process will be saved in io folder, namely MASKED_CHR1 and the list of collapsed Exons will
 * be saved in another file name under the same folder, called value of COLLAPSED_EXON.
 *
 * @author Mohamad Mahdi Ziaee
 *
 */
public class ExonRegionMaskComplementApp {

	/**
	 * The main method to be executed in the application.
	 *
	 * @param args
	 */
	public static void main(String[] args) {
		try {
			ExonMaskUtils.getInstance().run(FileUtils.HG19_REFSEQ_EXON_ANNOT, FileUtils.CHR1, FileUtils.MASKED_CHR1, FileUtils.COLLAPSED_EXON);
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

com.bio.main.MappabilityApp

```
package com.bio.main;

import java.io.IOException;

import com.bio.util.FileUtils;
import com.bio.util.MappabilityUtils;

/**
 * The main application to be executed in order to completed part 2 of the assignment. All the input and output files must be placed under io folder.
 * The application reads the Exon annotations file and the bow tie output and finally, it counts the number of hits at gene level and saves it into an
 * output file.
 *
 * @author Mohamad Mahdi Ziaee
```

```
 *
 */
public class MappabilityApp {

	/**
	 * The main method to be executed.
	 *
	 * @param args
	 */
	public static void main(String[] args) {
		try {
			MappabilityUtils.getInstance().run(FileUtils.HG19_REFSEQ_EXON_ANNOT, FileUtils.BOW_TIE_OUTPUT, FileUtils.GENE_EXPRESSION_COUNT);
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

com.bio.pojo.RefSeq

```
package com.bio.pojo;

/**
 * A POJO to hold the data related to RefSeq.
 *
 * @author Mohamad Mahdi Ziaee
 *
 */
public class RefSeq {

	private int startIndex;
	private int endIndex;
	private String id;

	/**
	 * Default constructor setting all the attributes of the class.
	 *
	 * @param startIndex
	 * @param endIndex
	 * @param id
	 */
	public RefSeq(int startIndex, int endIndex, String id) {
		this.startIndex = startIndex;
		this.endIndex = endIndex;
		this.id = id;
	}

	public int getStartIndex() {
		return startIndex;
```

```
	}

	public void setStartIndex(int startIndex) {
		this.startIndex = startIndex;
	}

	public int getEndIndex() {
		return endIndex;
	}

	public void setEndIndex(int endIndex) {
		this.endIndex = endIndex;
	}

	public String getId() {
		return id;
	}

	public void setId(String id) {
		this.id = id;
	}
}
```

com.bio.util.ExonMaskUtils

```
package com.bio.util;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

import com.bio.pojo.RefSeq;

/**
 * The main utility class for Exon Mask.
 *
 * @author Mohamad Mahdi Ziaee
 *
 */
public class ExonMaskUtils {
	private static final char CHAR_N = 'N';
	private static ExonMaskUtils instance = null;

	/**
	 * Private constructor for Singleton design pattern purpose. Declared private so it is not accessible from outside.
	 */
```

```
	private ExonMaskUtils() {
	}

	/**
	 * Gets instance of this class. It will instantiate it if it is not done yet, once.
	 *
	 * @return
	 */
	public static ExonMaskUtils getInstance() {
		if (instance == null) {
			instance = new ExonMaskUtils();
		}
		return instance;
	}

	/**
	 * The method reads the Exon annotation file, collapses the overlapping Exon regions into one and save it into collapsedExonFileName file. Later
	 * it will use the collapsed Exons to replace non Exon regions with N.
	 *
	 * @param exonAnnotationFileName
	 * @param chr1FileName
	 * @param maskedChr1FileName
	 * @param collapsedExonFileName
	 * @throws IOException
	 */
	public void run(String exonAnnotationFileName, String chr1FileName, String maskedChr1FileName, String collapsedExonFileName) throws IOException {
		// Reading the Exon annotation file
		List<RefSeq> refSeqs = readRefSeqs(exonAnnotationFileName);
		// Collapsing the Exons
		System.out.println("Collapsing Exons...");
		List<RefSeq> collapsedExons = collapseExons(refSeqs);
		// Save collapsed Exons into a file
		System.out.println("Saving collapsed Exons into output file...");
		saveCollapsedExons(collapsedExonFileName, collapsedExons);
		// Creating masked Chr1 file
		System.out.println("Masking non-Exon regions and saving it into output file...");
		maskNonExons(collapsedExons, chr1FileName, maskedChr1FileName);
		System.out.println("Done!");
	}

	/**
	 * Reads the Exon annotation file name given and convert it into RefSeq.
	 *
	 * @param exonAnnotationFileName
	 * @return
	 * @throws IOException
	 */
	public List<RefSeq> readRefSeqs(String exonAnnotationFileName) throws IOException {
		System.out.println("Reading the Exon annotation file...");
		List<String> exonAnnotLines = FileUtils.getInstance().readFile(exonAnnotationFileName);
```

```
                // Converting the file lines into RefSeq
                System.out.println("Converting Exon annotations...");
                return toRefSeq(exonAnnotLines);
        }

        /**
         * Saves the list of given RefSeqs into a file using collapsedExonFileName file name.
         *
         * @param collapsedExonFileName
         * @param collapsedExons
         * @throws IOException
         */
        private void saveCollapsedExons(String collapsedExonFileName, List<RefSeq> collapsedExons) throws IOException {
                // Deleting the output file if it already exists
                FileUtils.getInstance().deleteIfExists(collapsedExonFileName);
                PrintWriter out = FileUtils.getInstance().getPrinterWriter(collapsedExonFileName);
                for (RefSeq refSeq : collapsedExons) {
                        out.println(format(refSeq));
                }
                out.close();
        }

        /**
         * Formats the RefSeq given by harcoding chr1 as prefix, start index, end index and the refseq ID. It will also add 0 and + at the end of the
         * string. Each of the mentioned columns will be separated with a tab.
         *
         * @param refSeq
         * @return
         */
        private String format(RefSeq refSeq) {
                return String.join(FileUtils.TAB, "chr1", Integer.toString(refSeq.getStartIndex()), Integer.toString(refSeq.getEndIndex()), refSeq.getId(), "0",
"+");
        }

        /**
         * The method uses the collapsedExons regions and those which are not within the region in chr1FileName will be replaced with N. The result will
         * be saved in maskedChr1FileName method.
         *
         * @param collapsedExons
         * @param chr1FileName
         * @param maskedChr1FileName
         * @throws IOException
         */
        public void maskNonExons(List<RefSeq> collapsedExons, String chr1FileName, String maskedChr1FileName) throws IOException {
                // Deleting the output file if it already exists
                FileUtils.getInstance().deleteIfExists(maskedChr1FileName);
                int chr1Index = -1; // Chr1 file index.
                List<String> chr1Lines = FileUtils.getInstance().readFile(chr1FileName);
                PrintWriter out = FileUtils.getInstance().getPrinterWriter(maskedChr1FileName);
                out.println(chr1Lines.get(0));
```

```
		StringBuffer resultLine = null;
		int collapsedExonsIndex = 0;
		int startIndex = collapsedExons.get(collapsedExonsIndex).getStartIndex();
		int endIndex = collapsedExons.get(collapsedExonsIndex).getEndIndex();
		for (int i = 1; i < chr1Lines.size(); i++) {
			resultLine = new StringBuffer();
			char[] chr1LineChars = chr1Lines.get(i).toCharArray();
			for (char ch : chr1LineChars) {
				chr1Index++;
				// If we are within the Exon start and end indexes, copy the same character, otherwise print N.
				if (startIndex <= chr1Index && chr1Index < endIndex) {
					resultLine.append(ch);
				} else {
					resultLine.append(CHAR_N);
				}
				// If we have reached the current collapsedExons end Index, then we get the next one in the list.
				if (collapsedExonsIndex < collapsedExons.size() - 1 && chr1Index >= endIndex) {
					collapsedExonsIndex++;
					startIndex = collapsedExons.get(collapsedExonsIndex).getStartIndex();
					endIndex = collapsedExons.get(collapsedExonsIndex).getEndIndex();
				}
			}
			out.println(resultLine.toString());
		}
		out.close();
	}

	/**
	 * Goes through the list of RefSeqs and combines those which are overlapping.
	 *
	 * @param refSeqs
	 * @return
	 */
	public List<RefSeq> collapseExons(List<RefSeq> refSeqs) {
		// Sorting the refSeqs based on their startIndex
		sortRefSeqs(refSeqs);
		List<RefSeq> result = new ArrayList<>();
		// Setting the initial end and start indexes.
		int endIndex = refSeqs.get(0).getEndIndex();
		int startIndex = refSeqs.get(0).getStartIndex();
		for (int i = 1; i < refSeqs.size(); i++) {
			// Going through all the ones which are overlapping and finding the maximum end index.
			while (i < refSeqs.size() && endIndex >= refSeqs.get(i).getStartIndex()) {
				if (endIndex < refSeqs.get(i).getEndIndex()) {
					endIndex = refSeqs.get(i).getEndIndex();
				}
				i++;
			}
			result.add(new RefSeq(startIndex, endIndex, refSeqs.get(i - 1).getId()));
			// When we are on the last index, we don't want to set the end and start indexes because we are done!
```

```
                        if (i < refSeqs.size()) {
                                startIndex = refSeqs.get(i).getStartIndex();
                                endIndex = refSeqs.get(i).getEndIndex();
                        }
                }
                return result;
        }

        /**
         * Sorts the given RefSeqs based on their start indexes.
         *
         * @param refSeqs
         */
        private void sortRefSeqs(List<RefSeq> refSeqs) {
                Collections.sort(refSeqs, new Comparator<RefSeq>() {
                        @Override
                        public int compare(RefSeq arg0, RefSeq arg1) {
                                // Sorting two RefSeqs based on their start indexes.
                                return Integer.compare(arg0.getStartIndex(), arg1.getStartIndex());
                        }
                });
        }

        /**
         * Converts the lines read from a file into Refseq, given each of the elements are separated by \t.
         *
         * @param lines
         * @return
         */
        public List<RefSeq> toRefSeq(List<String> lines) {
                List<RefSeq> result = new ArrayList<>();
                for (String line : lines) {
                        String[] columns = line.split(FileUtils.TAB);
                        result.add(new RefSeq(Integer.valueOf(columns[1]), Integer.valueOf(columns[2]), columns[3]));
                }
                return result;
        }
}
```

com.bio.util.FileUtils

```
package com.bio.util;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.nio.file.Files;
import java.nio.file.Paths;
```

```
import java.util.List;

/**
 * The utility class for all the file related operations.
 *
 * @author Mohamad Mahdi Ziaee
 *
 */
public class FileUtils {

	private static FileUtils instance = null;
	public static final String IO_PATH = "../Assignment7/io/";
	public static final String TAB = "\t";
	public static final String UNDER_SCORE = "_";
	public static final String HG19_REFSEQ_EXON_ANNOT = "HG19-refseq-exon-annot-chr1-2016";
	public static final String CHR1 = "chr1.fa";
	public static final String MASKED_CHR1 = "chr1_masked.fa";
	public static final String COLLAPSED_EXON = "collapsed_exon.txt";
	public static final String BOW_TIE_OUTPUT = "BTout-v3-m1-final";
	public static final String GENE_EXPRESSION_COUNT = "gene_level_expression.txt";

	/**
	 * Private constructor for Singleton design pattern purpose. Declared private so it is not accessible from outside.
	 */
	private FileUtils() {
	}

	/**
	 * Gets instance of this class. It will instantiate it if it is not done yet, once.
	 *
	 * @return
	 */
	public static FileUtils getInstance() {
		if (instance == null) {
			instance = new FileUtils();
		}
		return instance;
	}

	/**
	 * Reads the file name given in io folder and returns a list of strings representing each line in the file.
	 *
	 * @param fileName
	 * @return
	 * @throws IOException
	 */
	public List<String> readFile(String fileName) throws IOException {
		return Files.readAllLines(Paths.get(FileUtils.IO_PATH + fileName));
	}
```

```
	/**
	 * Deletes the file placed in io folder if it already exists.
	 *
	 * @param fileName
	 * @throws IOException
	 */
	public void deleteIfExists(String fileName) throws IOException {
		Files.deleteIfExists(Paths.get(FileUtils.IO_PATH + fileName));
	}

	/**
	 * Creates a Printer Writer for the given file name by setting the append = true.
	 *
	 * @param fileName
	 * @return
	 * @throws IOException
	 */
	public PrintWriter getPrinterWriter(String fileName) throws IOException {
		return new PrintWriter(new BufferedWriter(new FileWriter(FileUtils.IO_PATH + fileName, true)));
	}
}
```

com.bio.util.MappabilityUtils

```
package com.bio.util;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import com.bio.pojo.RefSeq;

/**
 * The main utility class related to mappability.
 *
 * @author Mohamad Mahdi Ziaee
 *
 */
public class MappabilityUtils {

	private static MappabilityUtils instance = null;

	/**
	 * Private constructor for Singleton design pattern purpose. Declared private so it is not accessible from outside.
	 */
	private MappabilityUtils() {
```

```
        super();
}

/**
 * Gets instance of this class. It will instantiate it if it is not done yet, once.
 *
 * @return
 */
public static MappabilityUtils getInstance() {
        if (instance == null) {
                instance = new MappabilityUtils();
        }
        return instance;
}

/**
 * The method counts the number of reads on Exons and then convert the count to gene level. Later it will save the result into outputFileName
 * file.
 *
 * @param exonAnnotationFileName
 * @param bowTieOutputFileName
 * @param outputFileName
 * @throws IOException
 */
public void run(String exonAnnotationFileName, String bowTieOutputFileName, String outputFileName) throws IOException {
        // Counting mapped reads on each Exon.
        Map<String, Integer> countMappedReadsOnExons = countMappedReadsOnExons(exonAnnotationFileName, bowTieOutputFileName);
        // Converting the count to gene level
        Map<String, Integer> countMappedReadsOnGenes = convertCounterToGeneLevel(countMappedReadsOnExons);
        // Writing the end result to output file.
        saveGeneExpressionResult(outputFileName, countMappedReadsOnGenes);
        System.out.println("Done!");
}

/**
 * Saves the gene expression result into outputFileName for all the given counters.
 *
 * @param outputFileName
 * @param countMappedReadsOnGenes
 * @throws IOException
 */
private void saveGeneExpressionResult(String outputFileName, Map<String, Integer> countMappedReadsOnGenes) throws IOException {
        System.out.println("Saving the final result...");
        PrintWriter out = FileUtils.getInstance().getPrinterWriter(outputFileName);
        for (String geneId : countMappedReadsOnGenes.keySet()) {
                out.println(String.format("%s\t%s", geneId, countMappedReadsOnGenes.get(geneId)));
        }
        out.close();
}
```

```
	/**
	 * Convers the Exon level counter into gene level counter by checking the prefix of the Exons.
	 *
	 * @param countMappedReadsOnExons
	 * @return
	 */
	public Map<String, Integer> convertCounterToGeneLevel(Map<String, Integer> countMappedReadsOnExons) {
		System.out.println("Converting the counter to Gene level...");
		Map<String, Integer> result = new HashMap<>();
		// Initializing the gene counter
		for (String exonId : countMappedReadsOnExons.keySet()) {
			String[] columns = exonId.split(FileUtils.UNDER_SCORE);
			String geneId = columns[0] + FileUtils.UNDER_SCORE + columns[1];
			result.put(geneId, Integer.valueOf(0));
		}
		// Counting by looking at the prefix of the exonId to see if it is the same geneId.
		for (String geneId : result.keySet()) {
			for (String exonId : countMappedReadsOnExons.keySet()) {
				if (exonId.startsWith(geneId)) {
					result.put(geneId, result.get(geneId) + countMappedReadsOnExons.get(exonId));
				}
			}
		}
		return result;
	}

	/**
	 * Counts the number of reads mapped on Exons using the bow tie output and the Exon annotation file.
	 *
	 * @param exonAnnotationFileName
	 * @param bowTieOutputFileName
	 * @return
	 * @throws IOException
	 */
	public Map<String, Integer> countMappedReadsOnExons(String exonAnnotationFileName, String bowTieOutputFileName) throws IOException {
		List<RefSeq> refSeqs = ExonMaskUtils.getInstance().readRefSeqs(exonAnnotationFileName);
		List<RefSeq> bowTieRefSeqs = readBowTieOutput(bowTieOutputFileName);
		Map<String, Integer> countMap = new HashMap<>();
		// Initializing the counter map
		System.out.println("Initializing the counter map...");
		for (RefSeq refSeq : refSeqs) {
			countMap.put(refSeq.getId(), Integer.valueOf(0));
		}
		// Counting ...
		System.out.println("Counting the mapped reads on each exons...");
		for (RefSeq btSeq : bowTieRefSeqs) {
			for (RefSeq refSeq : refSeqs) {
				if (refSeq.getStartIndex() <= btSeq.getStartIndex() && btSeq.getEndIndex() <= refSeq.getEndIndex()) {
					String id = refSeq.getId();
					countMap.put(id, countMap.get(id) + 1);
```

```
                                }
                        }
                }
                return countMap;
        }

        /**
         * Reads the bow tie output file and converts the result into a list of RefSeqs.
         *
         * @param bowTieOutputFileName
         * @return
         * @throws IOException
         */
        private List<RefSeq> readBowTieOutput(String bowTieOutputFileName) throws IOException {
                List<String> btOutputLines = FileUtils.getInstance().readFile(bowTieOutputFileName);
                List<RefSeq> result = new ArrayList<>();
                for (String line : btOutputLines) {
                        String[] columns = line.split(FileUtils.TAB);
                        result.add(new RefSeq(Integer.valueOf(columns[1]), Integer.valueOf(columns[2]), columns[3]));
                }
                return result;
        }
}
```