Aashray Shrestha

Detailed Report
CS 3310 : Data and File Structures
Assignment 2
Instructor : Dr.Ajay Gupta
TA : Ms.Rajani Pingili

Problem Specification :

The assignment will check to see if a given DNA (or RNA) double-stranded is palindromic and search for palindromic strands in a genomic sequence. A palindrome in language is a word or a string that can be read the same way backwards and forwards. The palindromes in a genetic sequence are referred to as genetic palindromes in this project.

PHASE 1 : SPECIFICATION

i. Store the genomic sequence in a linked list to use pointers and nodes
ii. Linked List Node data portion has at least one attribute, called **nucleotide** of type char
iii. Read the DNA/RNA sequences from a file and store one strand per line
iv. Identify the genetic palindromes in the sequence
v. Find similar genetic palindromes for the ones that are found
vi. Test and measure time complexities of isGeneticPalindrome() , findGeneticPalindrome(), and countSimilarGeneticPlaindromes()
vii. Input validate on the given to check the DNA/RNA strands
viii. Sort the list and print it out

PHASE 2 : DESIGN

2.1 Modules and Basic Structure

4 classes
- Data
- Main
- PerfectList
- SortList

Interface
- MyList

2.2 Pseudocode for Each Class and Interface

I. Data Class

The data class was designed to store and return the data from the input file.

i. **public static final** List<String> readStringsFromFile(String filename) **throws** IOException
to read the line from the file as a String

ii. **public static final** List<PerfectList> getListsOfSequences(String filename) **throws** IOException
to get the list of Sequences from the file
- the method loops through each line and stores it in the the "PerfectList"
- the storing is done using the .add method as described

II. MyList

MyList is an interface with the following methods:

```
public interface MyList {
    void add(Character character);
    void append(MyList myList);
    PerfectList.Node getFirst();
    PerfectList.Node getLast();
    int count(char c);
    void insert(int position, char c);
    void sortedInsert(char c);
    void clear();
    boolean remove(Character character);
    Object[] toArray();
    int size();
    boolean contains(Character character);
    Object get(int index); //additional
}
```

III. Perfect List

public class PerfectList implements MyList, Iterable

Fields :

```
Node<Character> first;
Node<Character> last;
private int size =0;
```

Constructor

```
public PerfectList() {
    last = new Node<Character>(null, null, first);
    first = new Node<Character>(null, last, null);
}
```

Methods

getFirst ()

to get the first character of the list

getLast()

to get the last character of the list

int count

to get the count

void insert (int position, char c**)**
to insert the given character at the given position

sortedInsert(char c)
to insert into the given list

void add(Character character)
add the given character to the end of the list

void append(MyList myList)
append to the List

void clear()
clear out the list

public boolean remove(Character character)
to remove a character from the list

public boolean contains(Character character)
to see if the list contains the character

public Object get(int index)
get the Object at that index

public Iterator iterator()

Node class within the PerfectList

```
public class Node<Character> {
    Character character;
    Node<Character> next;
    Node<Character> prev;
    public Node(Character character, Node<Character> next, Node<Character> prev) {
        this.character = character;
        this.next = next;
        this.prev = prev;
    }

    public Character getCharacter() {
        return character;
    }
    public void setCharacter(Character character) {
        this.character = character;
    }
    public Node<Character> getNext() {
        return next;
    }
    public void setNext(Node<Character> next) {
        this.next = next;
    }
    public Node<Character> getPrev() {
        return prev;
    }
    public void setPrev(Node<Character> prev) {
        this.prev = prev;
    }

}
```

III. SortList

The class is to create a sorted list from the given

Method : **listSort() to sort the list**
display() to display the results

IV. Main Class

The main class has the following methods which is the driver of the code

1. **Main method :** get the input from the user and drive the program
2. **countSimilarGeneticPalindrome :** to count similar geneticPalindromes
3. **isSimilar :** to see if there are similar palindromes in the list
4. **testCase :** testCase to check whether the given is a DNA or RNA then if it is a genetic palindrome and go on from there
5. **findGeneticPalindromes :** find the genetic palindromes inside the sequence
6. **isDNAorRNA :** to check if the given input is valid to check if the input is T, A, G, C, or U

PHASE 3 : RISK ANALYSIS

No risk

PHASE 4 : VERIFICATION

The program will run for all .txt files which have some lines in it. The program does however, have a few errors here and there that could have been fixed.

PHASE 5 : Coding

The code is attached with this file and has comments for better understanding.

Aashray Shrestha

PHASE 6 : TESTING

Study 1 : Time to Create Linked List from Sequence

- Check the Genetic Palindromes
- Similar Sequences

Test 1:

```
Time taken to create linked lists from sequences: 9.0 milliseconds

DNA sequence 1 : TATA

TATA is a genetic palindrome.
However, it has the following genetic palindromes TATA-
Time to test whether a genomic subsequence in Sequence 0 is palindromic:0.0
TATA has 42 similar sequences in the file, and it took 12.0  milliseconds to determine that.

DNA sequence 2 : UAGCTA

UAGCTA is a genetic palindrome.
However, it has the following genetic palindromes UAGCTA-AGCT-
Time to test whether a genomic subsequence in Sequence 1 is palindromic:1.0
UAGCTA has 41 similar sequences in the file, and it took 3.0  milliseconds to determine that.

DNA sequence 3 : ACCTAGGT

ACCTAGGT is a genetic palindrome.
However, it has the following genetic palindromes ACCTAGGT-CCTAGG-CTAG-
Time to test whether a genomic subsequence in Sequence 2 is palindromic:1.0
ACCTAGGT has 0 similar sequences in the file, and it took 3.0  milliseconds to determine that.
```

Test 2:

```
Time taken to create linked lists from sequences: 9.0 milliseconds

DNA sequence 1 : TATA

TATA is a genetic palindrome.
However, it has the following genetic palindromes TATA-
Time to test whether a genomic subsequence in Sequence 0 is palindromic:0.0
TATA has 42 similar sequences in the file, and it took 12.0  milliseconds to determine that.

DNA sequence 2 : UAGCTA

UAGCTA is a genetic palindrome.
However, it has the following genetic palindromes UAGCTA-AGCT-
Time to test whether a genomic subsequence in Sequence 1 is palindromic:1.0
UAGCTA has 41 similar sequences in the file, and it took 3.0  milliseconds to determine that.

DNA sequence 3 : ACCTAGGT

ACCTAGGT is a genetic palindrome.
However, it has the following genetic palindromes ACCTAGGT-CCTAGG-CTAG-
Time to test whether a genomic subsequence in Sequence 2 is palindromic:1.0
ACCTAGGT has 0 similar sequences in the file, and it took 3.0  milliseconds to determine that.
```

Test 3 :

```
Time taken to create linked lists from sequences: 9.0 milliseconds

DNA sequence 1 : TATA

TATA is a genetic palindrome.
However, it has the following genetic palindromes TATA-
Time to test whether a genomic subsequence in Sequence 0 is palindromic:0.0
TATA has 42 similar sequences in the file, and it took 11.0  milliseconds to determine that.

DNA sequence 2 : UAGCTA

UAGCTA is a genetic palindrome.
However, it has the following genetic palindromes UAGCTA-AGCT-
Time to test whether a genomic subsequence in Sequence 1 is palindromic:0.0
UAGCTA has 41 similar sequences in the file, and it took 6.0  milliseconds to determine that.

DNA sequence 3 : ACCTAGGT

ACCTAGGT is a genetic palindrome.
However, it has the following genetic palindromes ACCTAGGT-CCTAGG-CTAG-
Time to test whether a genomic subsequence in Sequence 2 is palindromic:1.0
ACCTAGGT has 0 similar sequences in the file, and it took 4.0  milliseconds to determine that.
```

Cases	Time Millissecond
Case 1	9
Case 2	9
Case 3	9

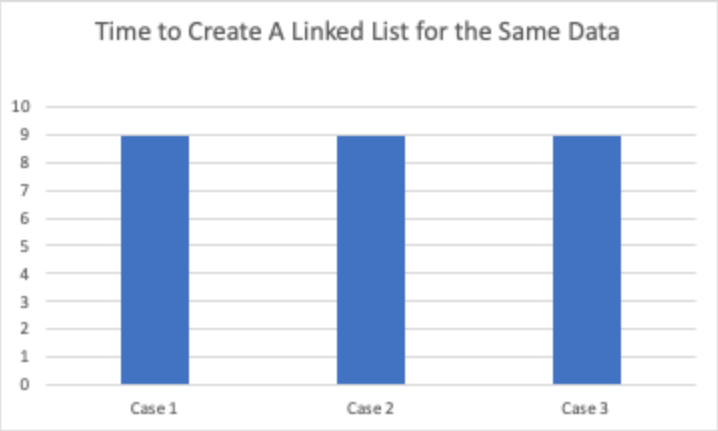


Fig : Time taken to Create A Linked List
Conclusion : Linear time when taken in Milliseconds

Cases	Time to find Similar Sequences #1	Time to find Similar Sequences #2	Time to find Similar Sequences #3
Case 1	12	3	9
Case 2	12	3	3
Case 3	11	6	4

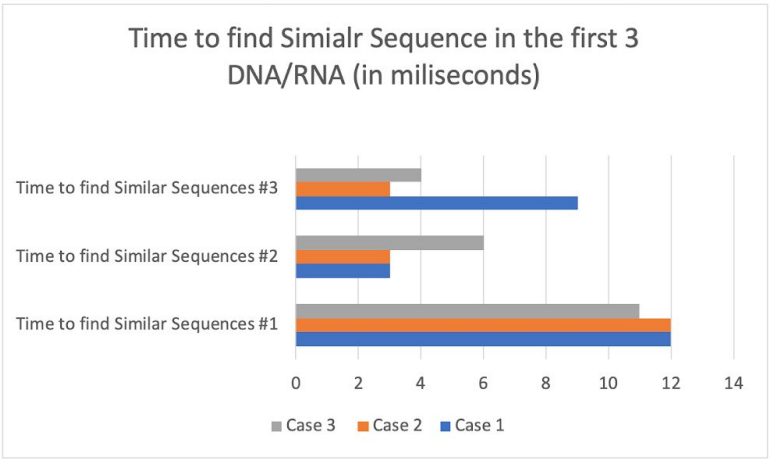


Fig : Time taken to find Similar Sequences in the File in Milliseconds

STUDY 2 :

Case 1:

```
The first sorted line is :AAAAAAAAAACCCCCCCGGGGGTTTTTTUU

The last sorted line is :AAAAAAAAAACCCCCCCGGGGGTTTTTTUU

Time taken to sort 5 milliseconds
```

Case 2:

```
The first sorted line is :AAAAAAAAAACCCCCCCGGGGGTTTTTTUU

The last sorted line is :AAAAAAAAAACCCCCCCGGGGGTTTTTTUU

Time taken to sort 10 milliseconds
```

Case 3 :

```
The first sorted line is :AAAAAAAAAACCCCCCCGGGGGTTTTTTUU

The last sorted line is :AAAAAAAAAACCCCCCCGGGGGTTTTTTUU

Time taken to sort 7 milliseconds
```

Case	Time Taken to Sort in Milliseconds
1	5
2	10
3	7

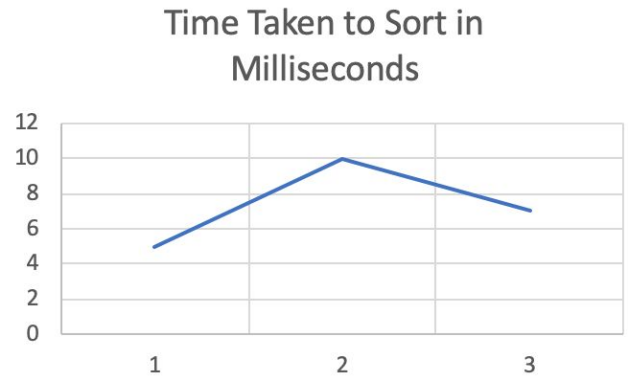


Fig : Time taken to sort the list

PHASE 7: REFINING THE PROGRAM

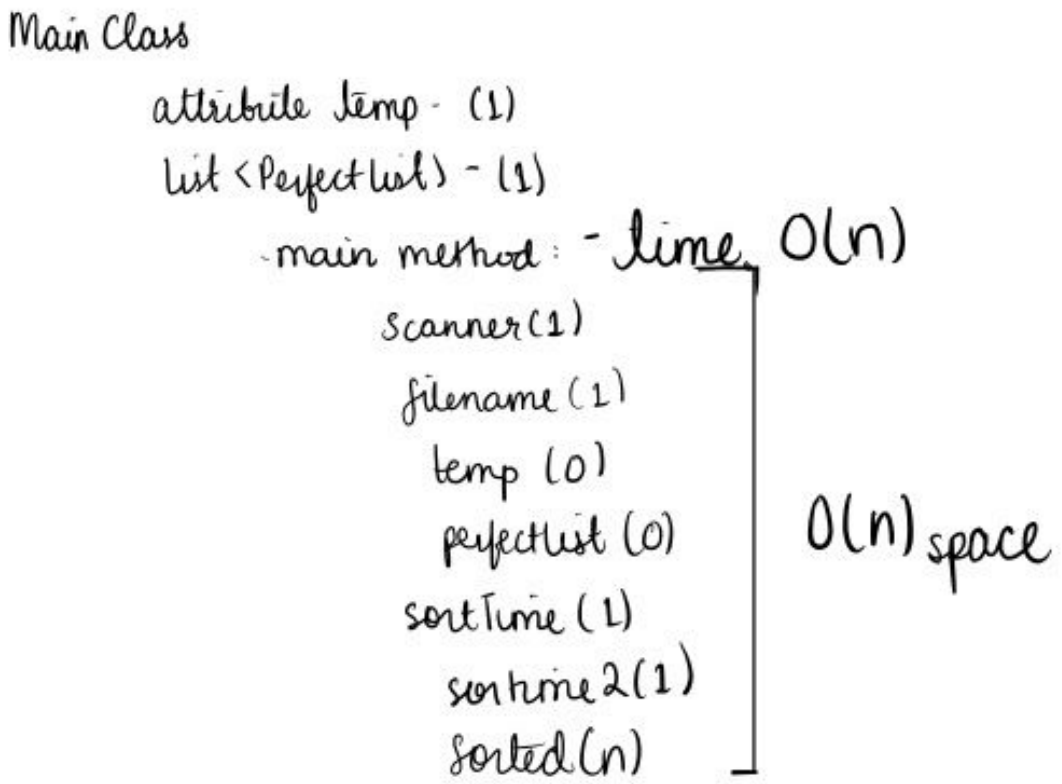
The program was refined along the way with making several changes in each step.

PHASE 8 : PRODUCTION

Includes the Source Code and documentation

PHASE 9: MAINTENANCE

The program can be maintained and improved on to have better time complexity and space complexity. However, since the code does not follow the exact instructions mentioned in the assignment, I have decided to submit the code.



countSimilarPalindrome: $O(n^3)$ time — $O(n)$ space
isSimilar: $O(1)$ time
testcase: $O(n)$ time — $O(n)$ space
findGeneticPalindrome: $O(n^3)$ time — $O(n)$ space
subList: $O(n^2)$ time — $O(n)$ space
isDNARNA: $O(n)$ time — $O(2)$ space

isGeneticPalindrome :	O(n) time	O(3) space
isPair :	O(n) time	O(2) space

Perfect List:

attributes :		O(2) space
perfectList() :	O(2) time	O(2) space
getFirst () :	O(1) time	
getLast () :	O(1) time	
count () :	O(n) time	O(3) space
insertNth() :	O(n) time	O(3) space
add () :	O(3) time	O(4) space
append() :	O(2) time	O(n) space
clear () :	O(2) time	O(0) space
remove () :	O(2) time	O space

Node Class:

4 attributes : O(4) space

Analysis of the algorithm space and time complexity

- The reuse of Node variable and temp variable could reduce the space complexity if defined at the beginning. However, to clear it, it would require (O(1)) time.

References :

StackOverFlow
GeeksforGeeks