Carol Iglesias
FINAL PROJECT

(a) Data

I downloaded the genome from *Candidatus Carsonella Ruddii* from the NCBI database, formatted as a Fasta file. I decided to use this genome as my running example upon recommendation from our textbook. In chapter Four of the textbook, focused on genome assembly, the authors recommend attempting to solve the problems only with simulated data or with a small genome, to avoid problems related to repeats and subsequent bubbles in the de Bruijn or Eulerian graphs.

I did not manually modify the Fasta file.

Since the information that I needed to use was not an actual genome but rather a sequence of k-mers or of reads that would amount back to a genetic sequence, I had to first create a program that would (a) break the sequence into its k-mer composition; (b) break the sequence into randomly segmented reads. I organized this information as two dictionaries. The first section corresponded to Problem BA3A in Rosalind, from where I downloaded sample data sets to try out my program.

(b) Steps of my programs

"Problem 3E"

I created two programs in order to solve this problem: readFile and adjList. (When applied to the *Candidatus Carsonella Ruddii* sequence it is only necessary to use the second program, as the information has already been processed from the Fasta file and organized into a list).

readFile parsed through a txt (i.e., a Rosalind data sample) and converted a list of k-mers separated into lines into a list *Pattern*, in which the elements are of type string.

adjList
input: list of k-mers Pattern
output: a dictionary (adjacency list) in which each prefix or term (a k-mer up to its prior to last character) is paired with its corresponding suffix (i.e. a k-mer with which it shares all characters but the first one, and which thus corresponds to the following k-mer in the sequence if the k-mers had been produced as the different moments of a sliding window moving one character at a time).

It creates a list of prefixes, that is, a list of the k-mers minus their last term.
For each element of this list [each prefix], it looks for all those k-mers in *Pattern* that coincide in everything but their last term. It creates a list for each prefix, including these k-mers from their second character to the last. In a dictionary, we make each key a prefix, and its values the list of suffixes differing only in their last character.

Evaluation: I was able to solve this problem and results were checked in Rosalind.

"Problem 3F"

I created one program to solve this problem: eulerianCycle, which both read a txt file and produced a list of edges, or trajectory for the graph.

eulerianCycle

input: txt file in which an adjacency list is given as paired characters (type: string), separated by the character '->' and commas when there are multiple options/suffixes.
output: eulerian cycle in the form of a trajectory, list of edges.

PartI: create a dictionary out of the txt adjacency list. I had to eliminate all characters that weren't letters or integers (for the Rosalind examples).
PartII: eulerian cycle out of the dictionary/adjacency list.

I created a list that included all edges, i.e., all the possible points in which the trajectory could stop, including those elements that had more than one possible exits as many times as they could be traversed. [unusedEdges]

In a FOR loop, the program would start at each of the possible unusedEdges and generate a trajectory list [EdgeList], taking random decisions (through random.choice) at each point where more than one possible path was possible. Every time the program run through the FOR loop, unusedEdges was 'renewed' so as to be able to produce a new trajectory and re-use all points.

Inside the FOR loop, I created a WHILE loop that split into two possibilities. The first possibility took place if the point taken in the graph had more than one exit available. [This could be verified if the length of the dictionary value was longer than 1 character] In this case, the program chose randomly between one of the two options, if the randomly chosen option had already been taken, the program deleted it from the list of possibilities and took the other. It appended this new point (chosen path) to the EdgeList and it became the new 'starting point' from which the WHILE loop departed. In the second case, if the point gave way only to another point (one possibility), this became the new starting point and it was appended to the EdgeList and removed from unusedEdges. The WHILE loop run until there were no more unusedEdges.

Each time the program run through the FOR loop, I had the EdgeList printed. That way one could see all possible Eulerian Cycles for a given adjacency list.

Evaluation: I was unable to deal with 'wrong paths'. Every time the program randomly produced a shorter route that left unusedEdges, it gave me an error as it wanted to keep running through the WHILE loop but had gotten to a point with no 'possible' exits, as they had all been used.
If I manually input the right first choice, and for a short adjacency list, it can give me the right Eulerian Cycle. Since I was unable to solve this, I couldn't move on to the following problem, the Eulerian Path. This ended up meaning I couldn't run my program on the reads from *Candidatus Carsonella Ruddii* but only in the k-mer list, which gave only one possible path.

"Problem 3H"

I reused readFile and adjList and created a program stringR in order to solve this problem.

input: list of k-mers (Pattern)
output: reconstructed sequence (string)

stringR
input: adjacency list in the form of a dictionary (prefix/suffix)
output: reconstructed sequence

The program consisted of two FOR loops. The first FOR loop went through the list of suffixes and made a list in which every element in the Dictionary values was of type string. The second FOR loop checked every prefix against the suffix list. The prefix that did not appear in suffix list had to be the first one of the sequence. Once the program had determined the starting point for the sequence, it just followed the ordered assigned by the adjacency list. An embedded FOR loop, running for n-1 times the number of k-mers or elements in the adjacency list, appended to a list *Seq* each of the prefixes.

Once *Seq* had been produced, it was made into a string by adding the last character of each of the elements in *Seq* to the starting prefix. (That is to say, each of the prefixes merely gives us the information of the character that follows, for all the rest of characters overlap with the prior term).

Evaluation: this program worked and I verified it through Rosalind. The only problem is that the k-mers list did not have multiple options, I was not really making use of the Eulerian or de Bruijn graphs and thus, this could not be applied to the list of reads but only to the k-mer composition of *Candidatus Carsonella ruddii* genome.

"Final"

The python file *Final* combined many of the above described programs and applied them to the genome of *Candidatus Carsonella ruddii*, which I downloaded as a Fasta file.

readFasta
It eliminated the first line of the txt file, which is introduced by the character '>'
Then it returned a string (dnaSeq) out of the annexation of all the lines. I called this string Carsonella for this program.

kmerComp
input: string (text), and an integer k (length of k-mers into which I'm trying to break the string).
output: dictionary, the values of which are the k-mer composition list (as opposed to a list itself, the dictionary is unordered)
I just had to pass a sliding window through the sequence, which was spliced at each character and appended to the dictionary.

readsDict
I incorporated kmerComp to this program.
input: string (text), integer k, integer c (number of copies I want to make of the k-mer composition list)
output: dictionary of reads, in which every read is of length k, some may be repeated, and some may be missing.
The program runs through a FOR loop 'c' number of times. [This replicates the amount of copies one would have of a given DNA sequence for the reads to be produced]. A second FOR loop is embedded, of the average length of the sequence divided between the length of k-mers (we want to replicate what would happen if for each of the copies 'c', the sequence was fragmented at random points – i.e., producing not consecutive k-mers with overlap – produced by a sliding window – but instead fragments or reads with the possibility of missing areas or repeats. This average number (no) is the amount of k-mers that are selected and appended to the dictionary every time(c times) that the program runs through the dictionary of k-mers.

The reads dictionary was longer than the k-mer dictionary.

Example from k-mer dictionary:
```
0: 'ATGAATAATATTTTTGCAAAAATAA',
1:  'TGAATAATATTTTTGCAAAAATAAC',
2:   'GAATAATATTTTTGCAAAAATAACT',
3:    'AATAATATTTTTGCAAAAATAACTG',
4:     'ATAATATTTTTGCAAAAATAACTGC',
```

Example from reads dictionary:
```
0: 'TTTTTTTTTTTAAAAAAAAAAATAT',
1: 'CTAATAGAAAAATAATTTTTTATTT',
2: 'GAACAAAATGATATAAAAAAAATTA',
3: 'TATGTGCTGGGACTTTTATTAATTC',
4: 'TTTAATTTAACAATGGAAAAACAAA',
```

adjList, stringR

Once the two dictionaries are produced, I pass them through the programs adjList and stringR, already explained. AdjList worked for both k-mer and reads dictionary, but stringR could only be applied to the k-mer composition. Here is a fragmentary example:

```
ADJACENCY LIST FROM K-MERS:

'TTTTGTGTTGGAAAATAATGATTT':
 'TTTGTGTTGGAAAATAATGATTTA,

'TTGCAGGAATAAATGCAGCTAGAA':
 'TGCAGGAATAAATGCAGCTAGAAA
```

```
ADJACENCY LIST FROM READS: works! (but has repeats!)

TTGCAGGAATAAATGCAGCTAGAA':
'TGCAGGAATAAATGCAGCTAGAAA'
'TGCAGGAATAAATGCAGCTAGAAA'

'GCTAAAAATATAATTTTATGTGCT':
 'CTAAAAATATAATTTTATGTGCTG'
 'CTAAAAATATAATTTTATGTGCTG'
```

SeqMaking

This program made a string(sequence) from the list of prefixes produced by StringR. (The same as at the end of Problem3H file) At the end of this program, I added a check/comprobation. It prints 'success!' when the sequence produced is the same as the sequence produced with readFasta and called 'Carsonella'.

Evaluation: as explained before, this worked perfectly for the k-mer list, but I couldn't solve the problems necessary in order to deal with repeated reads, or, what comes down to the same, having more than one possible option to follow at a particular node.

(c) Discussion/Conclusion

I had no idea how difficult dealing with random starts and randomly produced reads (which would amount to 'senseless' repetitions and missing parts) would be. It was interesting to produce my own information because already in that section I had to give up the 'realistic' multiplication of the sequence and instead use a FOR loop to run through the already produced list of k-mers. Otherwise the length of reads could not have been regular.

When dealing with Problem 3E I noticed that I wasn't sure how applicable the Eulerian problems described in the text book would be to my list of reads, given that the 'repeats' at least were different to each other. I wasn't sure what to glue nodes as it is described in the de Bruijn graphs.

I think that my project shows that having incomplete or excessive information are equally misleading and that the difficulty of genome reassembly seems to derive mostly from the fact that we cannot isolate and conserve a single DNA string. My simulated data did not even start to account for the difficulty that dealing with the two strands of DNA would cause (inability to know whether a read is 'following' or the 'reverse').