Harry Langford
hjel2@cam.ac.uk

1 Genome Assembly

1. In the *genome assembly* problem, we augment the previously covered sequencing framework with an additional *reference genome*. In what way does this aid sequencing?

   Most genomes are overwhelmingly similar. This means that when sequencing a new individual, a lot of the work has already been done. Typically, animals of the same species share $\sim 99.9\%$ of their DNA with each other. So we can get a reference genome of a "typical specimen", find where the differences are and patch over them to create a genome for the new individual much easier than constructing one from scratch.

2. Explain the basic operations supported by the trie data structure and their complexities. Highlight how it can be deployed into the problem of genome assembly in different ways. Your answer should provide a discussion of *tries*, *suffix tries* and *suffix arrays*.

   Trie's are a type of lookup data structure which is optimised for the case where keys are prefixes of each other. They support lookups, insertion and deletion. Lookups, insertions and deletions all have time complexities $\in \mathcal{O}(\ell)$ where the length of the key is ℓ.

   In a trie, you start at the root, move to the node corresponding to the next symbol in the key and repeat until there are no more symbols. If you are at a node, then you are done and the value of this node is the value of the lookup.

   A Suffix Trie is a trie which contains all suffixes of a particular piece of text.

   A Suffix Array is an array where the ith element holds the index of first element of the suffix which comes ith when ordered lexicographically. This can be used to do longest prefix match by binary searching the array for matches using $sequence[i:]$ as the key. This corresponds to the indices of the symbols in the first row in the intermediate BWT rotation matrix.

3. Explain how the methods of the Burrows-Wheeler transform (BWT) and run-length encoding (RLE) can be applied to reducing the storage requirements of large genomes. Prove that your scheme is efficiently invertible.

   We start with a genome which isn't particularly amenable to RLE (and path compression in Suffix Tries). The BWT takes this structure and creates a new representation which has a high probability of being amenable to RLE.

   The BWT is a form of string compression which works particularly well with genome data. Firstly, consider all possible rotations of the string. Then sort the permutations lexicographically. The last column of the rotation matrix is the result of the BWT. This representation is more likely to have sequences of characters than the original string if the original string has any commonly repeated sequences and so can be compressed better using RLE.

   BWT has the property that the predecessor to the ith occurrence of a symbol s in the last column is at the index corresponding to the ith occurrence of that symbol in the first column. Since we know the index of the last symbol (EOF), we can inductively use this to find the preceding symbol. This *can* be done in linear time (although my implementation below is quadratic for implementation simplicity).

4. How can one efficiently pattern-match to the string obtained by the above? Your answer should contain a discussion of the properties and construction time complexities of *suffix arrays*.

   - Suffix Trie

     We can pattern match in a Suffix Trie by simply looking up the string in the Trie. If we use the BWT'd matrix and path compression then we are likely to have a lower memory complexity.

- Suffix Array

  We can pattern match in a Suffix Array by binary searching the array for the prefix desired. This will return a range; where all elements in the range have the same prefix and this is the prefix we search for.

5. Demonstrate all of the techniques outlined in questions 2–4 on the sequence `CATATATAG$`.

   I start by demonstrating the BWT on the sequence `CATATATAG$`. I then invert it.

C	A	T	A	T	A	T	A	G	$
A	T	A	T	A	T	A	G	$	C
T	A	T	A	T	A	G	$	C	A
A	T	A	T	A	G	$	C	A	T
T	A	T	A	G	$	C	A	T	A
A	T	A	G	$	C	A	T	A	T
T	A	G	$	C	A	T	A	T	A
A	G	$	C	A	T	A	T	A	T
G	$	C	A	T	A	T	A	T	A
$	C	A	T	A	T	A	T	A	G

Table 1: BWT starts by creating a matrix of rotations

$	C	A	T	A	T	A	T	A	G
A	G	$	C	A	T	A	T	A	T
A	T	A	G	$	C	A	T	A	T
A	T	A	T	A	G	$	C	A	T
A	T	A	T	A	T	A	G	$	C
C	A	T	A	T	A	T	A	G	$
G	$	C	A	T	A	T	A	T	A
T	A	G	$	C	A	T	A	T	A
T	A	T	A	G	$	C	A	T	A
T	A	T	A	T	A	G	$	C	A

Table 2: Sort lexicographically

The result of BWT is the last column. In this case, `GCTTT$AAAA`.

We can invert the BWT by the following:

- **Initialision:**

  Reconstruct the first column by sorting the last column.

  Start with $ as the last symbol and set the index to the index of this in the last column

- **Iteration:**

  If the current index corresponds to the ith occurrence of that character in the BWT representation, add that character to the string; and set the current index to the index corresponding to the ith occurrence of that character in the first column.

- **Termination:**

  Terminate when the index corresponds to $ again.

We have now parsed every symbol; and can reconstruct the original string as `CATATATAG$`, which is correct.

6. Implement either a suffix trie, a suffix array or the BWT in a language of your choice, and use it to verify the outcome of applying / constructing it on the same sequence as above.

```python
class Node:

    def __init__(self, value, subtrie):
        super().__init__()
        self.value = value
        self.subtrie = subtrie

    def __getitem__(self, key):
        if key == '':
            return self.value
        for seq, rem in map(lambda i: (key[:i], key[i:]), range(len(key) + 1)):
            if seq in self.subtrie:
                return self.subtrie[seq][rem]

    def __setitem__(self, key, value):
        if key == '':
            self.value = value
        for seq, rem in map(lambda i: (key[:i], key[i:]), range(len(key))):
            if seq in self.subtrie:
                self.subtrie[seq][rem] = value

        newtrie = Node(
            value, {
                suffixkey[len(key):]: self.subtrie.pop(suffixkey)
                for suffixkey in list(
                    filter(
                        lambda x: len(x) != len(key) and x[:len(key)] == key,
                        self.subtrie))
            })

        self.subtrie[key] = newtrie

    def __repr__(self):
        return str(dict(self.subtrie))


class SuffixTrie(Node):

    def __init__(self):
        super().__init__(None, {})
        self.value_set = False

    def __getitem__(self, key):
        if key == '' and self.value_set is False:
            raise KeyError
        else:
            return super().__getitem__(key)

    def __setitem__(self, key, value):
        if key == '' and self.value_set is False:
            self.value_set = True
        else:
            super().__setitem__(key, value)
```

```python
def bwt(sequence):
    """
    :param sequence: the sequence to transform; terminated by '#'
    :type sequence: str
    :return: burrels wheeler transformation of the input
    :rtype: str
    """
    # sorted rotation matrix
    matrix = sorted(
        (sequence[i:] + sequence[:i] for i in range(len(sequence))),
        key=lambda x: "".join(x))
    # n.b. sorted is stable

    # last column
    return ''.join((s[-1] for s in matrix))


def invert(sequence):
    """
    :param sequence: burrels wheeler transformed sequence
    :type sequence: str
    :return: original sequence
    :rtype: str
    """
    # indices of first occurrence for each character in the last column
    last_indices = {base: sequence.index(base) for base in set(sequence)}

    # first column
    firstcol = ''.join(sorted(sequence))

    # indices of first occurrence for each character in the first column
    first_indices = {base: firstcol.index(base) for base in set(sequence)}

    # reversed accumulator of genomes
    genome = ['$']
    index = 0
    for _ in range(len(sequence) - 1):
        genome.append(sequence[index])
        count = sequence[:index].count(sequence[index])
        index = first_indices[sequence[index]] + count

    return ''.join(reversed(genome))


if __name__ == '__main__':
    # CATTATATAG$
    seq = 'CATTATATAG$'

    # GCTTT$AATAA
    seq = bwt(seq)

    # CATTATATAG$
    invert(seq)

from itertools import chain
```

```python
def rle(sequence):
    """
    :param sequence: the string to encode
    :type sequence: str
    :return: a run length encoding of the sequence
    :rtype: str
    """

    rle_str = ''

    count = 1
    current = sequence[0]

    # "#" is a character not occurring in sequence
    for c in chain(sequence[1:], '#'):
        if c == current:
            count += 1
        else:
            rle_str += f'{count}{current}'
            count = 1
            current = c

    return rle_str


if __name__ == '__main__':
    # 1C1A2T1A1T1A1T1A1G1$
    rle('CATTATATAG$')

    # 1G1C3T1$2A1T2A
    rle(bwt('CATTATATAG$'))
```

7. Carefully explain why we might be, in fact, generally more interested in *inexact matchings* of reads to the reference. Outline the *seeding* and BWT approaches to this problem, stating their complexities.

   Reconstructions aren't perfect. Genomes in all cells aren't identical. There are slight differences between species. We therefore want to allow slight mismatches and so want to be able to know when we have a near-match.

   The intuition behind "seeding" is that "if we want a string which has at most d mismatches, then we can split it up into $d+1$ chunks and at least one of them will match perfectly". So we can search for each of these "seeds" (substrings) and if any match then test the whole string at that point.

   We can use the BWT to search for inexact matchings using the following: start with the BWT string. Then apply the reconstruction algorithm to every index until there is more than a certain threshold of mismatches. We essentially reconstruct all subsequences of increasing length until they are no longer good enough.

2 Hidden Markov Models

1. Describe the key stateful components of a Hidden Markov Model (HMM), outlining the differences between it and a Markov Chain.

   A HMM has a Markov Transition Matrix; a hidden state and an emitted state. The difference between this and a Markov Chain; is that the Markov Chain state is not hidden and there is no emission.

2. Outline the inputs, outputs and time complexities of the following HMM algorithms:

   - Viterbi

     Inputs: A Markov Transition Matrix; a Matrix of Emission probabilities; a Vector of initial probabilities; and a sequence of observations.

     Outputs: A sequence of states; where the ith element corresponds to the most likely state at the ith timestep.

     Time Complexity: $\mathcal{O}(n \cdot |\Sigma|^2)$ where n is the number of timesteps and Σ is the number of states.

   - Forward

     Inputs: A Markov Transition Matrix; a Matrix of Emission probabilities; a Vector of initial probabilities; and a sequence of observations.

     Outputs: The most likely sequence of states to have generated the outputs.

     Time Complexity: $\mathcal{O}(n \cdot |\Sigma|^2)$ where n is the number of timesteps and Σ is the number of states.

   - Viterbi training

     Inputs: A set of sequences of hidden states and emitted states.

     Outputs: A Markov Transition Matrix; Emission Matrix; and Initial Probability Vector.

     Time Complexity: $\mathcal{O}(n \cdot \Sigma^2)$ where n is the number of symbols across all input sequences.

   - Baum-Welch

     Inputs: A set of sequences of emitted states.

     Outputs: A Markov Transition Matrix and Emission Probability Matrix likely to have generated the emitted states.

     Time Complexity: $\mathcal{O}(n \cdot \Sigma^2)$ where n is the number of symbols across all input sequences per timestep.

3. Consider the following three-state HMM, modelling an underlying process on DNA sequences (dotted lines represent the probabilities of starting in each of the three states):

$$\begin{tikzcd}
 & {} \arrow[d, dashed, "0.3"] & \\
x \arrow[r, bend left, "0.4"] \arrow[rr, bend left=60, "0.4"] \arrow[loop left, "0.2"] & y \arrow[l, bend left, "0.1"] \arrow[r, bend left, "0.3"] \arrow[loop below, "0.6"] & z \arrow[l, bend left, "0.1"] \arrow[ll, bend left=60, "0.8"] \arrow[loop right, "0.1"] \\
{} \arrow[u, dashed, "0.3"'] & & {} \arrow[u, dashed, "0.4"]
\end{tikzcd}$$

   Furthermore, you know the likelihoods of producing each of the nucleotides from each of the states:

	A	T	C	G
x	0.7	0.1	0.1	0.1
y	0.7	0.1	0.1	0.1
z	0.7	0.1	0.1	0.1

You observed the DNA sequence `CCGAAGTG`.

(a) What is the most-likely sequence of states that produced it?

20202020

(b) What is the probability that it was produced by this HMM?

20000000

(c) What is the probability that the HMM was in state x when producing the first `G`?

0.426

You may wish to consider implementing some of the required subroutines, rather than computing values by hand

```python
import numpy as np
from itertools import accumulate

# For simplicity, I ignore the vanishing probabilities problem


def forward(transition_matrix, input_probs, emission_probs, sequence):
    sequence_probs = [input_probs * emission_probs[:, sequence[0]]]
    for i in range(1, len(sequence)):
        probs = sequence_probs[-1]
        state_probs = probs @ transition_matrix
        sequence_probs.append(state_probs * emission_probs[:, sequence[i]])
    return [np.argmax(p) for p in sequence_probs]


def viterbi(transition_matrix, input_probs, emission_probs, sequence):
    sequence_probs = [input_probs + emission_probs[:, sequence[0]]]
    previous_most_likely = []

    for i in range(1, len(sequence)):
        probs = sequence_probs[-1]
        previous = np.argmax(probs * transition_matrix, axis=0)
        previous_most_likely.append(previous)
        state_probs = np.max(probs * transition_matrix, axis=0)
        sequence_probs.append(state_probs + emission_probs[:, sequence[i]])

    index = np.argmax(sequence_probs[-1])

    return list(
        reversed(
            list(
                accumulate(reversed(previous_most_likely),
                           lambda x, prev: prev[x],
                           initial=index))))
```

```
if __name__ == '__main__':
    transition_matrix = np.array([
        [0.2, 0.4, 0.4],
        [0.6, 0.1, 0.3],
        [0.8, 0.1, 0.1],
    ])

    initial_vector = np.array([0.3, 0.3, 0.4])

    emission_probs = np.array([
        [0.7, 0.1, 0.1, 0.1],
        [0.7, 0.1, 0.1, 0.1],
        [0.7, 0.1, 0.1, 0.1],
    ])

    forward(transition_matrix, initial_vector, emission_probs,
            [1, 1, 2, 0, 0, 2, 3, 2])
```

4. For the following two scenarios, explain (on an abstract level) how you would model the problem using HMMs, and which algorithms would be useful (and in what way):

   (a) Analysis of transmembrane (located around the cellular membrane) protein secondary structure – namely, for each amino acid of the protein, determining whether it's located *inside the cell*, *inside the membrane*, or *outside the cell.* You are provided with a training set containing transmembrane protein sequences, along with a labelled sequence of the same length, determining the location of each amino acid. You are also aware that any region of a protein within the membrane will consist of at least 5 and at most 25 amino acids.

   Hidden States: whether the amino acid is inside the cell, inside the membrane or outside the cell

   Observed States: amino acid

   Transition Probabilities: matrix showing the probability of amino acid x following amino acid y

   Initial Probabilities: probability of the amino acid being the first in the protein

   Emission Probabilities: probability of an amino acid being inside the cell, inside the membrane or outside the cell

   We can model the constraint by considering states as pairs of $(position, count)$ where count is the number of preceding amino acids which are in the membrane. This means that there will be 27 states; however each individual state only has 3 possible states it can transition to; meaning that the noise in the markov transition matrix is low. We then force any number of transitions in the transition matrix to 0 probability to model the constraint. How best to implement this is domain-specific knowledge and there are tradeoffs involved which depend on the size of the dataset and biological properties.

   The simplest solution would be to train the transition and emission matrices with all 27 states. However, this may lead to noise especially if long sequences 25 inside the membrane are very unlikely. However, this does allow us to model information about the distribution of lengths of sequences which are inside the membrane.

   A potentially better method use basic HMM training on the training set to learn the probability matrices (and vectors) using only 3 states; then duplicate the probabilities for "inside the membrane" into all the inside the membrane states; zeroing the relevant ones. This means there is no noise; but also means we cannot exploit any other information about the distribution of lengths that proteins are in the membrane for.

Note that since proteins have no "start" or "end", we could train the HMM using both directions of the data.

We can then use the viterbi algorithm to compute the most likely sequence of states.

(b) Classifying patients for presence or absense of a genetic disease, based on their DNA sequences. You are provided with a training set containing DNA sequences labelled as either "patient" or "normal".

We encode Bayes rule into a HMM and then use the result of the forward algorithm to infer whether the patient is more likely to be sick or healthy.

The initial probability vector corresponds to $P(health)$. The markov transition matrix is a disjoint graph. One set of nodes corresponds to nodes for a healthy individual and the other corresponds to nodes for a diseased individual. The transition probabilities between any nodes in different sets is zero. The combination of the transition matrix and emission matrix allows computation of $P(data \mid health)$. Forward multiplies these probabilities together and thus works out $\kappa \cdot P(health \mid data)$ for each health.

We train the model on the DNA sequences normally. We could use bigram or trigram models to learn marginally more complicated relationships between the DNA sequences.

To perform inference on this model, we run forward algorithm and compute the relative probabilities of the last state being in either set. The probability of the last state being in the healthy cluster is the probability that the person is healthy. In the simplest case, we could have only one node in each cluster (but this is a very limited model...) and then the result of classification is the last node that forward predicts.