Assignment 1

Instructions (READ CAREFULLY!)

1. Follow the directory format provided. Assignments that violate the directory format will be considered as not having been submitted. The directory can be downloaded here. **Remember to change the name of root folder to your student ID and to re-zip into a zip folder named your student ID!**

2. Complete this assignment in one of either Python 3.7, C, C++, or Java. If using Java, do not use packages as they would violate the directory format and complicate command-line compilation. Also, if using Java, ensure that your file paths for input and output files are not specific for your machine; submissions with this problem will be awarded 0.

3. Ensure that you edit the JSON file in the root of the directory with your name, Student ID, programming language used, and Email address.

4. Sign and attach either a soft copy or clear image of the plagiarism declaration with your submissions. There is a folder in the directory for you to store this declaration. You are allowed to discuss the assignment with your classmates; however, your code and write-up MUST be your own. You are free to use the sample code provided on the course Github repo.

5. After unzipping the directory structure, edit with the confines of that directory structure and then places it back into a .zip file with your Student ID. Other compression formats such as .rar or .7zip, or archives not named your Student ID will be rejected.

6. After completion of your assignment, fill out this form and email a copy of the assignemnt to irahamancourses@gmail.com. Your email should have the subject COMP2611 A1 1234 where 1234 is your student ID number.

7. For this assignment, you are free to use the Hashtable code provided on the Repo; however, you are not allowed to use the in-built dictionary or set types in Python.

Part A - DNA Alignment

One of the earliest problems in Bioinformatics is the read alignment problem. In this problem you are given several DNA sequences. One of these sequences is called the text, denoted as T, and the rest of the sequences is called the set of patterns P. Each DNA sequence comprises characters drawn from the set $\{A, C, G, T\}$. The text contains n characters, and each pattern in P contains m characters where $n < m$.

Given a set of patterns, P, The goal of the (exact) read alignment problem is to find all occurrences of each P in T.

Consider $T = ACACGAAC$, and let $P = \{AC, GA\}$. The pattern AC occurs at positions 2 and 6 in T, and the pattern GA occurs at position 4 in T.

Naively, we can simply iterative over the strings to compute the positions as in the following psuedocode.

```
ALIGN(T, P)
1  n = length(T)
2  m = length(P)
3  arr = []
4  for i = 0 to n − m
5        if T[i : i + m] == P
6              append(arr, i)
7  return arr
```

However, this is very inefficient for large DNA sequences - and DNA sequences can get quite large indeed! Instead, we would use a hashtable to store every sub-string of length m of T and then simply look into the hashtable to get all occurrences of a particular pattern.

Using a hashtable, write a programme in Python, C, C++, or Java that solves this read alignment problem. Your input comprises two files: dna.txt and patterns.txt. The first file, dna.txt contains the DNA sequence for referencing T on a single line. The second file contains several DNA sequences, each on a separate line. Your output should be a file named solutions.txt. Each line in solutions.txt contains the alignment positions of the corresponding line from patterns.txt. If a pattern is not found in T you should use the keyword *NONE* in place of the alignment positions. Consult the sample input files and output file for further clarification.

NB: You may assume that T is no longer than 500 characters, each P is no longer than 10 characters, and that for a single run, you will have no more than 10 patterns in patterns.txt

Part B - Blocking Websites

Recall that every machine on the Internet has an IP address. In order to retrieve webpages, we need to map their URL to an IP address. This process is called DNS resolution. We can use a type that implements the Dictionary ADT discussed in class to implement such a mapping.

Sometimes when running a proxy server, we want to block out entire machines (i.e. IP addresses) instead of individual domains. We can assemble an efficient blacklist by using a type that implements the Set ADT discussed in class.

You are given three input files:

- dns.txt - stores URL-to-IP address mappings. Each line on this file stores a single mapping, and the URL and IP address are separated by a single space
- blacklist.txt - each line contains an IP address to block. This file can contain IP addresses not present in dns.txt
- queries.txt - each line contains a url. This file can contain urls not present in dns.txt

Write a programme that reads these three files and determines if we should block access to a URL. For each line in query, you should have a corresponding line in a file solutions.txt with either Y, indicating that we block a site, or N, indicating that we didn't block a site. Consult the sample input and output files to get further clarification.

NB: You may assume that each IP address is an IPv4 address containing no more than 15 characters in lenght. You may assume that each URL is no more than 100 characters in length. You may assume that you will process no more than 10 IP addresses in blacklist.txt, no more than 25 pairs in dns.txt, and no more than 10 queries in queries.txt.