The University of Melbourne
School of Computing and Information Systems
COMP10002 Foundations of Algorithms
Semester 1, 2022
Assignment 1
Due: 4pm Friday 29 April 2022

1 Learning Outcomes

In this assignment, you will demonstrate your understanding of arrays, pointers, input processing, and functions. You will also extend your skills in terms of code reading, program design, testing, and debugging.

2 The Story...

Alpha, Beta, Gamma, Delta, and now the highly infectious Omicron variant! The SARS-CoV-2 (a.k.a., COVID-19) virus has kept mutating, creating new variants continuously. But how do scientists track the different variants of the virus? In this assignment, we will take a sneak peek at a key step in the virus variant tracking process – mapping the *genome* of a new variant to that of a *reference genome* (i.e., the genome of a reference virus), which helps identify the genes that have mutated in the variant comparing with the reference virus. You do *not* need to be a biologist to complete this assignment. This assignment specification contains sufficient background knowledge. For further information on virus variant tracking, you may also take a look at a web page on *Genomic Surveillance* (`https://www.cdc.gov/coronavirus/2019-ncov/variants/genomic-surveillance.html`).

To track variants of a virus, scientists obtain "isolates", which are virus isolated from infected patients. DNA or RNA are extracted from the isolates, which need to be translated into a sequence of *bases*. DNA consists of four bases: adenine (**A**), cytosine (**C**), guanine (**G**), and thymine (**T**), while the four bases of RNA are adenine (**A**), cytosine (**C**), guanine (**G**), and uracil (**U**). This translation process is referred to as *sequencing*. After sequencing, the DNA of a virus is represented as a sequence (that is, a string) consisting of four letters 'A', 'C', 'G', and 'T'[1]. Different virus variants can be compared using their DNA strings, and sub-strings of specific patterns (e.g., a gene that makes a variant more infectious) can be identified from the DNA. For example, below are the first 350 bases of the Wuhan-Hu-1 isolate of the SARS-CoV-2 virus:[2]

```
ATTAAAGGTTTATACCTTCCCAGGTAACAAACCAACCAACTTTCGATCTCTTGTAGATCTGTTCTCTAAA
CGAACTTTAAAATCTGTGTGGCTGTCACTCGGCTGCATGCTTAGTGCACTCACGCAGTATAATTAATAAC
TAATTACTGTCGTTGACAGGACACGAGTAACTCGTCTATCTTCTGCAGGCTGCTTACGGTTTCGTCCGTG
TTGCAGCCGATCATCAGCACATCTAGGTTTCGTCCGGGTGTGACCGAAAGGTAAGATGGAGAGCCTTGTC
CCTGGTTTCAACGAGAAAACACACGTCCAACTCAGTTTGCCTGTTTTACAGGTTCGCGACGTGCTCGTAC
```

As a preparation step, the DNA of a virus isolate needs to be broken into short pieces, that is, *DNA fragments*, because it is more difficult to recognise the bases accurately from long DNA fragments. A sequencer machine then recognises the bases from DNA fragments and converts them into short pieces of DNA sequences called *reads*, which are represented by short strings consisting of four letters 'A', 'C', 'G', and 'T'. Below is an example record of a read (using the FASTQ format).

```
@MT734046.1-1990/1
TTTGCGCATCTGTTATGAAATAGTTTTTAACTGTACTATCCATAGGAATAAAATCTTCTA
+
CCCGGGCGGGGGGCCGGGGGGGGGGGGGGCCGG=GGGGGGGGGGGGGGGGGGGGGGGCGG
```

[1] For RNA virus such SARS-CoV-2, the RNA is converted to complementary DNA during a preparation step of sequencing.

[2] Complete genome available at `https://www.ncbi.nlm.nih.gov/nuccore/NC_045512.2?report=fasta`

The record of a read consists of four lines:

- Line 1: An Identifier line starting with '@' (You may assume at least 2 and up to 100 characters in this line).
- Line 2: A DNA sequence consisting of four letters 'A', 'C', 'G', and 'T', where each character represents a DNA base (You may assume at least 2 and up to 100 characters in this line).
- Line 3: A line with a single plus sign '+'.
- Line 4: A line of characters of the same length as Line 2. Each character represents a *quality score* of a base in Line 2, which reflects how confident the sequencer machine is when recognising a base. For example, the first base of the example read, 'T', has a quality score of 'C' (67), while the last base, 'A', has a quality score of 'G' (71). The characters in this line have ASCII values between 33 and 73. We will detail how these characters are converted to sequencer machine error probabilities later.

A challenging problem in DNA sequencing is how to map the reads (that is, short sub-strings) of a new virus variant to a reference DNA sequence, such that the complete DNA sequence of the new virus variant can be reconstructed. This assignment works on a simplified version of the problem.

3 Your Task

You will be given a skeleton code file named `program.c` for this assignment on Canvas. The skeleton code file contains a `main` function that has been completed already. There are a few other functions which are incomplete. You need to add code into them for the following tasks. **Note that you should *not* change the `main` function, but you are free to modify any other parts of the skeleton code (including adding more functions).**

The given input to the program consists of a number of read records (at least one and at most 100 records) followed by a separator line with 5 '#'s and a reference DNA sequence in a single line (at least 2 and at most 1000 'A', 'C', 'G', and 'T' characters). Each read record follows the format as described above. *No input validity checking is needed. Below is a sample input.*

```
@MT734046.1-1990/1
TTTGCGCATCTGTTATGAAATAGTTTTTAACTGTACTATCCATAGGAATAAAATCTTCTA
+
CCCGGGCGGGGGCCGGGGGGGGGGGGGGGCCGG=GGGGGGGGGGGGGGGGGGGGGGGCGG
@MT734046.1-1988/1
AATCAACCACACCCTCTTGTATTTTAATACCC
+
CCCGGGGG#GGGGGG=GGG=GGGGGGG$&GGC
@MT734046.1-1986/1
TGTAGAGAATAAAACATTAAAGTTTGCACAATGCAGAATGCATCTG
+
CCCGGGGGGGGGGGGGGGGG=GGGGG()GCG=GGGGG=GGGGGG8G
@MT734046.1-1984/1
TGTGTGAATTTGGA
+
CCCGGGGGGGGGGC
@MT734046.1-1982/1
TCCTG
+
CCCG3
#####
CAGTATAATTTTTGCGCATCTGTTATGAAATAGTTTTTAACTGTACTATCCATAGGAATAAAATCTTCTA
AATAACTAATTACTGTCGTTGACAGGACACGAGTAACTCGTCTATCTTCTGCAGGCTGCTTACGGTTTCG
TCCGTGTTGCAGCCGATCATCAGCACATCTAGGTTTCGTCCGGGTGTGACCGAAAGGTAAGATGGAGAGC
CTTGTCCCTGGTTTCAACGAGAAAACACACGTCCAACTCAGTTTGCCTGTTTTACAGGTTCGCGACGTGC
TCGTACAATCAACCCCACCCTCTTGTATTTTAACACCCAATCAACCACACCCTCTTGTATTTTAATACCA
```

Here, we have shown the reference DNA sequence in multiple lines for ease of presentation. The sequence in the actual test files will be stored in a single line.

3.1 Stage 1: Process One Read and Print the Base with the Smallest Quality Score (Up to 3 Marks)

Your first task is to understand the skeleton code. Note the use of the types `read_id_t`, `read_t`, `score_t`, and `ref_t` in the skeleton code, which are essentially `char` type arrays. Each `read_id_t` variable stores the Identifier of a read, each `read_t` variable stores the content of a read, each `score_t` variable stores the quality scores of the bases of a read, and each `ref_t` variable stores a reference DNA sequence.

The `stage_one` function calls the `take_one_read` function. You need to add code to the `stage_one` function to call the `take_one_read` function to take the first read from the input data. Here, `take_one_read` is already given to you. The `stage_one` function then calls an `index_of_base_with_smallest_quality_score` function to locate and return the index (that is, the position) of the base that has the smallest quality score in the first read. Here, the quality scores are compared by their ASCII values. If there is a tie, the function should return the smallest index among the tied ones. Given the sample input above, the function should return 33, since the 33-th (index starting from 0) base `'T'` has the smallest quality score `'='` (its ASCII value 61 is smaller than those of the other quality scores `'C'` and `'G'` of the first read).

You need to complete the `index_of_base_with_smallest_quality_score` function as described above.

The output for this stage given the above sample input should be (where "`mac:`" is the command prompt):

```
mac: ./program < test0.txt
Stage 1
==========
Base with the smallest quality score: T
Index: 33
```

As this example illustrates, the best way to get data into your program is to edit it in a text file (with a "`.txt`" extension, any text editor can do this), and then execute your program from the command line, feeding the data in via input redirection (using `<`). In the program, we will still use the standard input functions such as `getchar` to read the data fed in from the text file. Our auto-testing system will feed input data into your submissions in this way as well. You do not need to (and *should not*) use any file operation functions such as `fopen` or `fread`. To simplify the assessment, your program should not print anything except for the data requested to be output (as shown in the output example).

You should plan carefully, rather than just leaping in and starting to edit the skeleton code. Then, before moving through the rest of the stages, you should test your program thoroughly to ensure its correctness.

3.2 Stage 2: Process All Reads and Find the Read with the Smallest Average Quality Score (Up to 7 Marks)

Now add code to the `stage_two` function to loop through all the input read records. The function should print out the total number of reads processed and the read that has the smallest *average quality score*. Here, the average quality score of a read is the average of the ASCII values of the quality scores of all the bases of the read. If there is a tie, print the first read with the smallest average quality score. On the same sample input data, the additional output for this stage should be:

```
Stage 2
==========
Total number of reads: 5
Smallest average quality score: 64.60
Read with the smallest average quality score:
TCCTG
```

Hint: You may call the `take_one_read` function written in Stage 1 to process a read record and write another function to calculate the average quality score. Alternatively, you may write a function to do both in a single pass over the read records. The second approach is faster in practice, but we allow both approaches for the purpose of the assignment. In either approach, you need to check whether input processing has reached the end of the read records. You may modify the `take_one_read` function if needed.

3.3 Stage 3: Revisit the Reads and Convert Any Base with Error Probability Larger than 0.15 to ‘$*$’ (Up to 11 Marks)

Bases with large *error probability* should carry a low importance when mapping a read to a sub-string of the reference DNA sequence.

Add code to the `stage_three` function to prepare the reads for mapping by replacing the bases with an error probability larger than 0.15 by a `'*'` character. Here, the error probability p of a base can be calculated from (the ASCII value of) its quality score Q using the equation below.

$$p = \frac{1}{10^{\frac{Q-33}{10}}} \tag{1}$$

The output of this stage are the reads after the base replacement. For example, given the sample input above, the output of this stage is as follows

```
Stage 3
==========
TTTGCGCATCTGTTATGAAATAGTTTTTAACTGTACTATCCATAGGAATAAAATCTTCTA
AATCAACC*CACCCTCTTGTATTTTAA**CCC
TGTAGAGAATAAAACATTAAAGTTTG**CAATGCAGAATGCATCTG
TGTGTGAATTTGGA
TCCTG
```

Hint: You should `#define` the constants before using them. Note that the constant 33 in Equation 1 is the lower bound of the ASCII value of the quality scores as mentioned earlier.

3.4 Stage 4: Process the Reference Sequence and Print the Numbers of A, C, G, and T Bases (Up to 13 Marks)

Add code to the `stage_four` function to process the reference sequence, that is, the line after the separator line in the input. This stage should output the length (that is, the total number of bases) of the sequence and the numbers of A, C, G, and T bases in the sequence. The output for this stage given the sample input above is as follows.

```
Stage 4
==========
Length of the reference sequence: 350
Number of A bases: 92
Number of C bases: 87
Number of G bases: 64
Number of T bases: 107
```

3.5 Stage 5: Map Reads to the Reference Sequence (Up to 15 Marks)

Add code to the `stage_five` function to further map the reads to sub-strings of the reference sequence. For each read, this stage should print out the sub-string of the reference sequence with the largest *match score*. Here, the match score between a read R and a sub-string S is calculated based on the error probability of the bases that are matched by the sub-string, as defined by the following equation:

$$\text{match\_score}(R, S) = -\sum_{i=0}^{\text{length}(R)-1} \log_2 \Big(\text{match\_score(R[i], S[i])}\Big) \tag{2}$$

$$\text{match\_score}(R[i], S[i]) = \begin{cases} p[i] & \text{if } R[i] = S[i] \\ 0.25 & \text{if } R[i] = \text{‘}*\text{’} \\ 1 & \text{otherwise} \end{cases} \tag{3}$$

Here, R is a full read and S must share the same length with R, length(R) denotes the number of bases in read R, $R[i]$ and $S[i]$ denote the i-th bases in R and S, respectively, and $p[i]$ denotes the error probability of the i-th base in R.

If there is a tie, the matched sub-string that appears earliest in the reference sequence should be printed.

The output for this stage given the above sample input should be (note a final newline '\n' at the end):

```
Stage 5
==========
Read:  TTTGCGCATCTGTTATGAAATAGTTTTTAACTGTACTATCCATAGGAATAAAATCTTCTA
Match: TTTGCGCATCTGTTATGAAATAGTTTTTAACTGTACTATCCATAGGAATAAAATCTTCTA
Read:  AATCAACC*CACCCTCTTGTATTTTAA**CCC
Match: AATCAACCCCACCCTCTTGTATTTTAACACCC
Read:  TGTAGAGAATAAAACATTAAAGTTTG**CAATGCAGAATGCATCTG
Match: TTGACAGGACACGAGTAACTCGTCTATCTTCTGCAGGCTGCTTACG
Read:  TGTGTGAATTTGGA
Match: AGTATAATTTTTGC
Read:  TCCTG
Match: TACTG
```

4 Submission and Assessment

This assignment is worth 15% of the final mark. A detailed marking scheme will be provided on Canvas.

To submit your code, you will need to:

1. Log in to Grok Learning Assignment 1 module via the **"Assignment 1" link in Canvas Assignments** page.
2. Write *all* your code in the `program.c` tab window.
3. Compile your code by clicking on the `Compile` button.
4. Once the compilation is successful, click on the `Mark` button to submit your code. You can submit as many times as you want to. *Only the last submission made before the deadline will be marked.* Submissions made after the deadline will be marked with late penalties as detailed at the end of this document. Do *not* press the `Mark` button after the deadline unless a late submission is intended.
5. Two sample tests will be run automatically after you make a submission. Make sure that your submission passes these sample tests.
6. Two hidden tests will be run for marking purpose. Results of these tests will be released after the marking is done.

You can (and should) submit both **early and often** – to check that your program compiles correctly on our test system, which may have some different characteristics to your own machines.

You will be given a sample test file `test0.txt` and the sample output `test0-output.txt`. You can test your code on your own machine with the following command and compare the output with `test0-output.txt`:

```
mac: ./program < test0.txt    /* Here '<' feeds the data from test0.txt into program */
```

Note that we are using the following command to compile your code on the submission testing system (we name the source code file `program.c`).

```
gcc -Wall -std=c99 -o program program.c -lm
```

The flag "`-std=c99`" enables the compiler to use a modern standard of the `C` language – `C99`. To ensure that your submission works properly on the submission system, you should use this command to compile your code on your local machine as well.

You may discuss your work with others, but what gets typed into your program must be individual work, **not** from anyone else. Do **not** give (hard or soft) copy of your work to anyone else; do **not** "lend" your memory stick to others; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say a very firm "no" when they ask for a copy of, or to see, your program, pointing out that your "no", and their acceptance of that decision, is the only thing that will preserve your friendship. *A sophisticated program that* *undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions in "compare every pair" mode.* See `https://academichonesty.unimelb.edu.au` for more information.

Deadline: Programs not submitted by **4pm Friday 29 April 2022** will lose penalty marks at the rate of 2 marks per day or part day late. Late submissions after 4pm Monday 2 May 2022 will **not** be accepted. Students seeking extensions for medical or other "outside my control" reasons should email the lecturer at jianzhong.qi@unimelb.edu.au. If you attend a GP or other health care professional as a result of illness, be sure to take a Health Professional Report (HRP) form with you (get it from the Special Consideration section of the Student Portal), you will need this form to be filled out if your illness develops into something that later requires a Special Consideration application to be lodged. You should scan the HPR form and send it in connection with any non-Special Consideration assignment extension requests.

Special consideration due to COVID-19: Please refer to "Advice for students affected by COVID-19" here: `https://students.unimelb.edu.au/your-course/manage-your-course/exams-assessments-and-results/special-consideration#advice-for-students-affected-by-covid-19`

And remember, *Algorithms are fun!*

Prepared by Jianzhong Qi