The University of Melbourne
School of Computing and Information Systems
COMP10002 Foundations of Algorithms
Semester 1, 2020
Assignment 1
Due: 4pm Tuesday 12 May 2020

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...

An emoticon, short for "emotion icon", is a pictorial representation of a facial expression using characters (punctuation marks only, for this assignment) to express a person's feelings or mood. As social networks such as Facebook and Twitter and messaging applications such as Snapchat have become part of our daily lives, a large number of emoticons are being used as part of messages and posts. Various individuals and organisations are harvesting and analysing these posts and using emoticons to understand the emotion of the users for business, research, and political decision support.

While emoticons contain rich information, they cannot be handled directly by traditional text analytical tools that were developed based on well written documents such as Wall Street Journal articles. The first step towards utilising emoticon contents is to clean up data and obtain only the valid emoticons. In this assignment, you will learn basic principles in data cleansing and write a program that will cleanse a list of emoticons and find the longest emotion in a post.

3 Your Task

You will be given an input that consists of multiple lines. Each line contains a message from a messaging service with **at least 1 and at most 280 characters**. There are **at least 1 and at most 100** lines. A sample input with five messages is shown below.

```
That,was,really,funny,=)),:))
I,have,no,clue,:/,:?,on,this,matter
^-^,You,are,the,best,*-*,:-*
See,you,|->,bye
Thank,you,:)
```

Note that we have replaced all whitespaces (' ') with commas (',') to simplify the debugging and testing process. There will *not* be any whitespaces in the input data. A comma will not be a part of any emoticon. An emoticon consists of only punctuation marks (no English letters or digits).

At the end of the messages, there is one special line, consists of three hash symbols ("`###`"). This special line separates the messages from the rest of the inputs. After this special line, there are inputs of multiple lines. These lines are a "dictionary" of emoticons, where each line contains a single emoticon followed by the emotion associated with that emoticon. Each such line has **at least 1 and at most 50 characters**. There are **at least 1 and at most 50** lines. A full sample input including five messages, the separating line, and dictionary of ten emoticons is shown below.

```
That,was,really,funny,=)),:))
I,have,no,clue,:/,:?,on,this,matter
^-^,You,are,the,best,*-*,:-*
See,you,|->,bye
Thank,you,:)
###
:),happy
:)),happy
^-^,happy
:?,confused
:/,confused
=)),fun
:(,sad
:((,sad
**_**,love
:-*,love
```

Note that we have replaced all whitespaces (' ') with commas (',') in the dictionary as well. There will *not* be any whitespaces in the input data.

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).**

3.1 Stage 1: Reading the First Message (Up to 3 Marks)

Your first task is to understand the skeleton code. Note the use of the types `msg_t` and `emtcn_t` in the skeleton code, which are essentially `char` type arrays. Each `msg_t` variable stores the content of a message, and each `emtcn_t` variable stores the content of an emoticon.

The `stage_one` function calls the `read_one_msg` function. You need to add code to the `stage_one` function to call the `read_one_msg` function to read the first message. Here, the `read_one_msg` function is already given to you. The `stage_one` function then calls the `count_tokens` function to count and return the number of *tokens* in the first message. A token is a basic grammatical unit in a sentence. Words and emoticons are both tokens. For simplicity, function `count_tokens` counts the number of tokens by counting the number of commas, and it should return the number of commas plus 1. You need to add code to complete the function as described above.

The output for this stage given the above sample input should be (where "`mac:`" is the command prompt; there are 5 commas in the first message, so `count_tokens` returns $5 + 1 = 6$):

```
mac: ./program < test0.txt
Stage 1
==========
Number of tokens: 6
```

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, `jEdit` 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. 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: Removing Alphanumeric Characters (Up to 7 Marks)

Now add code to the `stage_two` function to loop through the input messages until reaching the separating line. For each message, your program should cleanse the message by removing all English letters and digits, while keeping all the other characters including commas. On the same sample input data, the additional output for this stage should be:

```
Stage 2
==========
,,,,=)),:))
,,,,:/,:?,,,
^-^,,,,,*-*,:-*
,,|->,
,,:)
```

Hint: You may call the `read_one_msg` function written in Stage 1 to read a message and write another function to cleanse the message. Alternatively, you may write a function to do both in a single pass over the message. 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 the line that has been read is the separating line ("`###`") or not.

You may assume that no lines will become empty after this stage.

3.3 Stage 3: Removing Extra Commas (Up to 11 Marks)

Add code to the `stage_three` function to further cleanse the messages by removing any leading, trailing, and consecutive commas. After this cleanse, for every message, there should *not* be commas at the start or the end. Also, there should be just one comma between any two adjacent tokens remaining in a message. For example, after this stage, a message "`,,,,:/,:?,,,`" should become "`:/,:?`".

The output for this stage given the sample input above is as follows.

```
Stage 3
==========
=)),:))
:/,:?
^-^,*-*,:-*
|->
:)
```

Hint: Study the `getword` function (Figure 7.13 in the textbook, link to source code available in lecture slides `lec06.pdf`) to learn how to skip leading and trailing commas.

You may assume that no lines will become empty after this stage.

3.4 Stage 4: Reading the Dictionary and Finding the Longest Emoticon (Up to 13 Marks)

Add code to the `stage_four` function to read the dictionary, that is, the lines after the separating line ("`###`") in the input. Recall that, each line in the dictionary contains a single emoticon followed by a text that represents the corresponding emotion. This stage should output the total number of emoticons in the dictionary, the longest emoticon in the dictionary (that is, the emoticon with the maximum length), and its length. Here, the length refers to the number of characters in an emoticon. In the case of ties, the emoticon that appears the earliest in the dictionary should be the output. The output for this stage given the sample input above is as follows.

```
Stage 4
==========
Emoticon total: 10
Longest: **_**
Length: 5
```

3.5 Stage 5: Removing Invalid Emoticons with the Help of the Dictionary (Up to 15 Marks)

Note that you are NOT allowed to used any function from the `string.h` *library for Stage 5.* You are free to use any library functions in Stages 1 to 4.

Add code to the `stage_five` function to further cleanse the messages by removing the emoticons that are invalid. For this stage, you need to use the input dictionary of emoticons. That is, if there is no entry for an emoticon in the dictionary, that emoticon needs to be removed from the message. The output for this stage given the above sample dictionary input should be (note a final newline character '`\n`' at the end):

```
Stage 5
==========
=)),:))
:/,:?
^-^,,:-*
:)
```

Here, the emoticon "`*-*`" in the third message and the emoticon "`|->`" in the fourth message have been removed as they are not in the dictionary. If a message becomes an empty string after this stage (for example, the fourth message in the sample data), you should *not* print it. Note that, this stage may again leave extra commas. You do *not* need to worry about them for the assignment purpose. We design the assignment stages in this order such that you do not need to consider the dictionary in the first three stages.

4 Submission and Assessment

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

You need to submit your code in Grok Learning (`https://groklearning.com`**) for assessment**. Submission will NOT be done via the Canvas. Instead, you will need to:

1. Log in to Grok Learning using your student login details.
2. Navigate to the Assignment 1 module of our subject COMP10002 2020 S1: Foundations of Algorithms.
3. Write your code in the `program.c` tab window.
4. Compile your code by clicking on the `Compile` button.
5. 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.*)
6. Two sample tests will be run automatically after you make a submission. Make sure that your submission passes these sample tests.
7. Two hidden tests will be run for marking purpose. Results of these tests will be released after our 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 the lab machines and 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 server.

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

The flag "`-std=c99`" enables the compiler to use a more modern standard of the C language – `C99`. To ensure that your submission works properly on the submission server, 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** copied from anyone else. Do **not** give hard copy 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 Tuesday 12 May 2020** will lose penalty marks at the rate of two marks per day or part day late. Late submissions after 4pm Friday 15 May 2020 will **not** be accepted. Students seeking extensions for medical or other "outside my control" reasons should **email the subject coordinator 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 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 the "Special Consideration" section in this page: `https://students.unimelb.edu.au/student-support/coronavirus/information-for-all-students/`

And remember, *Algorithms are fun!*