Department of Computing and Information Systems

COMP10002 Foundations of Algorithms
Semester 2, 2015
Assignment 1

Learning Outcomes

In this project you will demonstrate your understanding of arrays, strings, and functions. You may also use `structs` if you wish – and will find the program easier to assemble if you do – but you are not required to use them in order to obtain full marks.

Text Search

The Unix tool `grep` provides the ability to identify the lines in a file that exactly match a pattern supplied as a command-line argument. For example, if the file `alice-eg.txt` contains

```
Down, down, down. Would the fall NEVER come to an end! 'I wonder how
many miles I've fallen by this time?' she said aloud. 'I must be getting
somewhere near the centre of the earth. Let me see: that would be four
thousand miles down, I think--' (for, you see, Alice had learnt several
things of this sort in her lessons in the schoolroom, and though this
was not a VERY good opportunity for showing off her knowledge, as there
was no one to listen to her, still it was good practice to say it over)
'--yes, that's about the right distance--but then I wonder what Latitude
or Longitude I've got to?' (Alice had no idea what Latitude was, or
Longitude either, but thought they were nice grand words to say.)
```

(taken from the Project Gutenberg version of Lewis Carroll's *Alice in Wonderland*, available for download from `http://www.gutenberg.org/cache/epub/11/pg11.txt` in plain text format), then the interaction

```
mac: grep "down" < alice-eg.txt
Down, down, down. Would the fall NEVER come to an end! 'I wonder how
thousand miles down, I think--' (for, you see, Alice had learnt several
mac:
```

identifies and prints the lines that contain (anywhere) the string "`down`". To help search English text, there are options in `grep` for the search to be case-insensitive ("`-i`") and to match on whole words only ("`-w`"). It is also possible to include "wildcard" characters. But it is always the case that a line either matches or does not, and that the lines that match are presented in the order they appear in the file.

Sometimes we want to perform a less precisely defined search, looking for approximate or best matches to the string, rather than exact Boolean matches. For example, we might want to search for "`fall down`", knowing that Alice does indeed do something like that, but be unsure of the exact phrasing used in the book; and would like "`fall down`" to provide a strong match to the first line of the example text, and weaker matches to the second and fourth lines.

In this project you will write a simple sequential ranked search process that processes text files, treating each input line as a possible match string, and generating a score for each line relative to a supplied query. Once all of the scores have been generated, the best-matching lines are output, in decreasing score order – a bit like whole documents are scored and ranked in web search engines such as Google and Bing. Unlike Google and Bing, you will not construct an index, and instead you are encouraged to make use of

obvious approaches, rather than seek out efficient but complex-to-implement techniques. In particular, if the query is m characters long and an input line is n characters long, your scoring function might take $O(m^2n)$ time. Over a query of up to a few dozen bytes, and an input text of up to a few megabytes (which is actually quite big), your program will (and had better!) still operate in a second or so. Note that you do *not* need to implement or use the pattern matching algorithms that are being discussed in class; and may use straightforward matching techniques.

Input Data

Input to your program will come in two parts: a *query*, specified on the command-line used to execute the program (see Section 7.11 of the textbook) as a single string (it will need to be quoted if it contains blanks); and a stream of text, to be (always) read from `stdin`. *You may not vary from these expectations!* A range of text input data will be used during the testing that will be carried out after submissions have closed, and your program must be able to execute over the whole `pg11.txt` file (linked from the LMS page) too. As you develop your program according to the stages listed below, the output will evolve. Full output examples for both the `alice-eg.txt` and full `pg11.txt` file are linked from the LMS. You should also check your program against other queries and inputs.

Stage 0 and Stage 1 – Reading Text (5/15 marks)

In this first stage, you are to demonstrate that you can access the two required inputs, the query from the command-line and the text from `stdin`. For Stage 0, you should print out the query string; for Stage 1, you should print out each input line, and its length in characters. For the example query and text, the first few lines of your output should be (exactly):

```
mac: ./ass1 "fall down" < alice-eg.txt
S0: query = fall down
---
Down, down, down. Would the fall NEVER come to an end! 'I wonder how
S1: line     1, bytes = 68
---
many miles I've fallen by this time?' she said aloud. 'I must be getting
S1: line     2, bytes = 72
```

You may assume that no input line contains more than $1{,}000$ characters, and that no query is longer than 100 characters.

Stage 2 – Simple Scoring (8/15 marks)

Now add a simple scoring regime that computes the longest sequence of consecutive characters from (anywhere) in the query string that matches (anywhere) in each of the input lines, including blanks and other punctuation characters, and without case folding on the alphabetic characters. For the same example execution, the first few lines of output (interleaved with the previous Stage 1 output which is not shown again, see the LMS for full output details) should be:

```
S2: line     1, score =  5.000
S2: line     2, score =  4.000
S2: line     3, score =  1.000
S2: line     4, score =  5.000
S2: line     5, score =  1.000
S2: line     6, score =  2.000
S2: line     7, score =  3.000
S2: line     8, score =  2.000
```

```
S2: line     9, score =  1.000
S2: line    10, score =  1.000
```

For example, the score for line 1 is 5 because "`#down`" and "`fall#`" (with `#` representing a blank) are the longest strings that match between the query and the line. Note that scores are calculated and represented as `doubles`, you'll see why in Stage 4.

Stage 3 – Ranked Summary Output (12/15 marks)

Once you have the Stage 2 scoring regime working correctly, it is time to move on to the main goal – presenting lines in decreasing score order. Add data structures to your program that retain the (up to) 10 highest-scoring lines, and their scores. Then, once all of the input lines have been read, print those 10 lines (or up to 10 lines, if there are not 10 lines with non-zero scores) and their line numbers and scores, in decreasing score order. If the scores are tied, then lines should be presented in line number order. For the example inputs, the start of the required output is:

```
-----------------------------------------------
S3: line     1, score =  5.000
Down, down, down. Would the fall NEVER come to an end! 'I wonder how
---
S3: line     4, score =  5.000
thousand miles down, I think--' (for, you see, Alice had learnt several
---
S3: line     2, score =  4.000
many miles I've fallen by this time?' she said aloud. 'I must be getting
---
S3: line     7, score =  3.000
was no one to listen to her, still it was good practice to say it over)
---
```

Note, you may *not* retain every input line in an array of strings, and your program may *not* assume some maximum number of lines in the input. You can only retain 10 (as a `#defined` value, of course) lines and their scores at any given time, plus the current line that is being processed.

Stage 4 – Better Scoring (15/15 marks)

Now implement a more sophisticated scoring computation. For $0 \leq i \leq j < m$, where m is the length of the query q, define $f_{i,j}$ to be the number of occurrences of $q[i...j]$ in the current line. Then compute

$$\text{score} = \left(\sum_{i=0}^{m-3} \sum_{j=i+2}^{m-1} (j-i-1)^2 \log_2(2+f_{i,j}) \right) / \log_2(30+n) \tag{1}$$

where n is the length of the line, and subject to the following additional restrictions:

- the pattern is *reduced*, and all non-alphanumeric characters are removed from it (use `isalnum()`);
- the character comparisons performed are case-insensitive, so that `'T'` matches `'t'`, etc; and
- strings for which $f_{i,j}$ are zero are not included in the summation.

For example, for the query "`fall down`" and the line

```
Down, down, down.  Would the fall NEVER come to an end!  'I wonder how
```

the values of $f_{i,j}$ (with case folding and query reduction applied) are:

$$f_{i,j} = \begin{array}{c|cccccccc} & \texttt{f} & \texttt{a} & \texttt{l} & \texttt{l} & \texttt{d} & \texttt{o} & \texttt{w} & \texttt{n} \\ \hline i=0 & 1 & 1 & \boxed{1} & \boxed{1} & 0 & 0 & 0 & 0 \\ i=1 & & 2 & 1 & \boxed{1} & 0 & 0 & 0 & 0 \\ i=2 & & & 3 & 1 & 0 & 0 & 0 & 0 \\ i=3 & & & & 3 & 0 & 0 & 0 & 0 \\ i=4 & & & & & 6 & 3 & \boxed{3} & \boxed{3} \\ i=5 & & & & & & 8 & 4 & \boxed{3} \\ i=6 & & & & & & & 6 & 3 \\ i=7 & & & & & & & & 7 \end{array}$$

The six numbers in boxes are then used in Equation 1, so that all matching character tri-grams (and longer) contribute to the final score. Your output from this stage will be interleaved with your Stage 1 and Stage 2 output, and will mean that your Stage 3 output also changes. See the LMS for a full example of the required output; be sure to study it carefully so that know what your program is expected to generate. You will probably find it helpful to include a `DEBUG` mode in your program that prints out intermediate data, including the frequency matrix for each input line, and the LMS has an example. Disable the debug mode when making your final submission, but leave the debug code in place. (Use `#if DEBUG` and `#endif` around such blocks of code, and then `#define DEBUG 1` or `#define DEBUG 0` at the top.)

The boring stuff...

This project is worth 15% of your final mark. A rubric explaining the expectations against which you will be marked will be provided via the LMS.

You need to submit your program for assessment; detailed instructions on how to do that will be posted on the LMS once submissions are opened. Submission will *not* be done via the LMS; instead you will need to log in to a Unix server and submit your files to a software system known as `submit`. You can (and should) use `submit` **both early and often** – to get used to the way it works, and also to check that your program compiles correctly on our test system, which has some different characteristics to the lab machines. Only the last submission that you make before the deadline will be marked.

You may discuss your work during your workshop, and with others in the class, but what gets typed into your program must be individual work, not copied from anyone else. So, do **not** give hard copy or soft copy of your work to anyone; do **not** "lend" your "Uni backup" memory stick to others for any reason at all; 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. Students whose programs are so identified will be referred to the Student Center.* See `https://academichonesty.unimelb.edu.au` for more information.

Deadline: Programs not submitted by **10:00am on Monday 21 September** will lose penalty marks at the rate of two marks per day or part day late. Students seeking extensions for medical or other "outside my control" reasons should email Alistair, `ammoffat@unimelb.edu.au` as soon as possible after those circumstances arise. Marks and a sample solution will be available on the LMS by Monday 5 October.

And remember, algorithms are fun!