School of Computing and Information Systems

COMP10002 Foundations of Algorithms
Semester 2, 2017
Assignment 1

Learning Outcomes

In this project you will demonstrate your understanding of arrays, strings, and functions. You may also use `typedefs` and `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. Nor do you need to make any use of `malloc()` in this project.

Text Search

The Unix command-line tool `grep` provides the ability to identify the lines in a file that exactly match a pattern supplied as a command-line argument. There are also useful options in `grep` for the search to be case-insensitive ("`-i`") and to match on whole words only ("`-w`").

Prefix matching searching

However there are also times when we want to perform a less precisely defined search, looking for partial matches to (possibly multiple) strings, rather than exact matches relative to one string. For example, we might be unsure of how to spell "`latitude`" and "`longitude`", and want to be able to use "`lat long`" as a query to identify – in some kind of decreasing-score order – the lines in an input document that contain one or more words that start with those two strings. Similarly, we might be unsure of how to spell someone's name, and not be bothered if a search for "`alis mof`" finds all of "`Alistair Moffat`", "`Alison Mofet`", "`Alistair, plus Bob Moffat`", and so on.

In this project you will write a program that reads text from `stdin`, generates a "match score" for each line relative to a query supplied on the command-line, and then prints out the lines that have the highest scores – a bit like documents are scored and ranked in web search engines. But, unlike Google and Bing, you will not use an index, and instead you are encouraged (in this project) to make use of obvious approaches. Over a query of up to a five or eight words, and an input text of up to a few megabytes (which is actually quite big), your program should (and had better!) still operate in a second or so. That is, you do *not* need to implement the pattern search algorithms that are being discussed in class; and may use straightforward matching techniques, including any suitable functions in `string.h`.

Input Data

Input to your program will come in two parts: a *query*, specified on the command-line (see Section 7.11 of the textbook) as a sequence of lowercase alphanumeric strings; and a stream of text, to be (always) read from `stdin`. *If you vary away from these interface requirements the automated testing process will fail your program!*

A range of text input data will be used during the post-submission testing. As you develop your program according to the stages listed below, the output will evolve. Output examples for both the `alice-eg.txt` and full *Alice's Adventures in Wonderland* `pg11.txt` file are linked from the FAQ page. You should also check your program against other queries and inputs, of course. Testing and debugging is *your* responsibility.

Stage 1 – Checking the Command-Line (5/15 marks)

In this stage you are to demonstrate that you can access the first of the two required inputs, the query from the command-line. The query itself will be provided to you via `argc` and `argv`. If `argc` is zero

when your program is called you should print the required error message and exit; and if any character in any of the strings making up the query is not a lowercase alphabetic or numeric character, you should print that particular string and the required error message. For example:

```
mac: ./ass1 < alice-eg.txt
S1: No query specified, must provide at least one word
mac: ./ass1 lat 66 loNg 32 words < alice-eg.txt
S1: query = lat 66 loNg 32 words
S1: loNg: invalid character(s) in query
mac:
```

Note how each output line is prefixed by the stage number that generated it.

Stage 2 – Reading the Input (8/15 marks)

In this next stage, you are to demonstrate that you can correctly access the text from `stdin`, by printing out each input line, its length in characters, and the number of words it contains. For the example query and text, the first few lines of your output should be (exactly):

```
mac: ./ass1 lat long < alice-eg.txt
S1: query = lat long
---
Down, down, down. Would the fall NEVER come to an end! 'I wonder how
S2: line = 1, bytes = 68, words = 14
---
```

and so on, see the FAQ page for the full required output. A word is defined to be a maximal length sequence of alphanumeric characters.

You may assume that no input line contains more than 1,000 characters. *Note the item in the FAQ page about newline differences between PC and Unix systems. You should copy the* `mygetchar()` *function into your program and use it (and only it) when you are reading input lines.*

Stage 3 – Scoring Lines (12/15 marks)

Each input line next needs to be given a *score* relative to the query. If there are q query words specified on the command-line (that is when `argc` $= q + 1$), if w_i is the i th of the query words (that is, when `argv`$[i] = w_i$), and if f_i is the number of times that word w_i is a case-insensitive prefix match against a word that appears in that input line, then the score S of that line is given by

$$S = \frac{\sum_{i=1}^{q} \log_2(1.0 + f_i)}{\log_2(8.5 + \ell)},$$

where ℓ is the number of words in the line. A prefix match occurs if every character of the query term matches at the beginning of a words in the input line. For example, "`ali`" is a prefix match of all of "`Ali`", "`Alistair`", "`Alison`", and "`alimentary`"; and is *not* a prefix match of any of "`alloy`", "`al`", "`ai`", or "`malice`". Lines that have no matches against any query terms will automatically get a score of zero according to this formula. Scores are to be calculated and represented as `doubles`; be aware that rounding in `double` arithmetic might lead to your program giving slightly different values to mine in some cases. When printed to three decimal places, the values are probably going to agree, but small implementation-dependent (based primarily on the exact order the operations are carried out by the compiled program) differences in computed values are always possible.

The required output for this stage is a score per line, interleaved with the previous Stage 2 output. The FAQ shows example executions, so that you can confirm that you understand what it is that you are to compute and how it is to be output – look for the lines that commence with "`S3:`". Note that $\log_2$ `x` can be computed via `log(x)/log(2.0)`, with `log` (natural logs) available in `math.h`.

Stage 4 – Ranked Summary Output (15/15 marks)

Once you have the Stage 3 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) five highest-scoring lines, and their scores. Then, once all of the input lines have been read, print those five lines (or up to five lines, if there are not five lines with non-zero scores) and their line numbers and scores, in decreasing score order. If the scores are tied (when doing simple comparisons on `double` values using == and <=, don't try and be clever), then lines should be presented in line number order. For the test file `alice-eg.txt` and the three-term query "`ali lat long`", the required output from this stage is:

```
-----------------------------------------------
S4: line 9, score = 0.668
or Longitude I've got to?' (Alice had no idea what Latitude was, or
---
S4: line 10, score = 0.233
Longitude either, but thought they were nice grand words to say.)
---
S4: line 4, score = 0.229
thousand miles down, I think--' (for, you see, Alice had learnt several
---
S4: line 8, score = 0.226
'--yes, that's about the right distance--but then I wonder what Latitude
---
```

The FAQ page shows some other example interactions with the desired program.

Note that 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 five (as a `#defined` value, of course) lines and their scores at any given time, plus the current line that is being processed; plus access the supplied query via `argv`.

General tips..

You will probably find it helpful to include a `DEBUG` mode in your program that prints out intermediate data and variable values. Use `#if DEBUG` and `#endif` around such blocks of code, and then `#define DEBUG 1` or `#define DEBUG 0` at the top. Disable the debug mode when making your final submission, but leave the debug code in place. The FAQ page has more information about this.

Finally, note that the sequence of stages described in this handout is deliberate – it represents a sensible path though to the required program. You can, of course, ignore the advice and try and write final program in a single effort, without developing it incrementally and testing it in phases. You might even get away with it, this time and at this somewhat limited scale, and develop a program that works. But in general, one of the key things that makes some people better at programming than others is the ability to see a design path through simple programs, to more comprehensive programs, to final programs, that keeps the complexity under control at all times.

The boring stuff...

This project is worth 15% of your final mark. A rubric explaining the marking expectations will be provided on the FAQ page.

You need to submit your program for assessment; detailed instructions on how to do that will be posted on the FAQ page 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. *Failure to follow this simple advice is highly likely to result in tears.* 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 else; 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 for possible disciplinary action without further warning. This message **is** the warning.* See `https://academicintegrity.unimelb.edu.au` for more information. Note also that solicitation of solutions via posts to online forums, whether or not there is payment involved, is also taken very seriously. In the past students have had their enrolment terminated for such behavior.

Deadline: Programs not submitted by **10:00am on Monday 18 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 `ammoffat@unimelb.edu.au` as soon as possible after those circumstances arise. 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 in to 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.

Marks and a sample solution will be available on the LMS before Tuesday 3 October.

And remember, algorithms are fun!