SCHOOL OF COMPUTING AND INFORMATION SYSTEMS

COMP20007 Design of Algorithms, Semester 1, 2018

Assignment 2: Spelling correction

Due: 12 noon Monday 21 May.

Overview

In this assignment you will work towards implementing a spelling corrector: a program that takes a document and attempts to correct misspelled words within that document.

The assignment consists of four coding tasks and one short written task. Each task is broadly related to the theme of word spelling differences, the final tasks dealing with implementing a full spelling corrector. Each task is described in detail in the sections below.

Provided files

Before you attempt this assignment you will need to download the following code and data files from the LMS:

`main.c`	Entry point to program. **Do not change**.
`list.h, list.c`	Singly-linked list module. **Do not change**.
`spell.h`	Prototypes for the 4 functions you must implement. **Do not change**.
`spell.c`	**Define your functions here**, according to specification below.
`Makefile`	To assist with compilation. Edit if you add any extra files.
`data/`	Directory containing some lists of words considered 'correctly spelled'.

At this point, you should be able to compile the supplied code (above) by running `make`. Once your assignment is completed, you will be able to execute it by running a command of a form depending on the coding task to execute (see the 'Running your program' section).

Data structures

Throughout the assignment, 'string' refers to a null-terminated array of `char` suitable for use with the functions declared in `string.h` (standard library string manipulation functions). Furthermore, for this assignment, every such string given to your program will only consist of lower-case alphabetic characters (and the null byte at the end).

Your solution will need to work with the data structures defined in `list.h`. Here's a brief overview (consult the header files for the finer details and a full list of available functions): A `List` is a linked list of `Node`s, with each `Node` containing a pointer to a single piece of data and to the next `Node` in the linked list. These lists can easily be traversed using an appropriate `while` or `for` loop.

In this assignment, the data with each list node is usually a string. That is, the node's `data` field (of type `void *`) can be cast to type `char *`, and can be passed to the functions from `string.h`.

Additionally, you may require further modules to support your implementation of each coding task. It is up to you exactly how to design and implement these modules. Provided code and sample solutions from previous labs may be used in your assignment (with proper attribution), or you may create your own. If you add additional modules, it is your responsibility to correctly extend your makefile—you must ensure that your solution can be compiled after submission simply by running `make`.

Coding tasks

The four coding tasks of the assignment require you to implement functions inside `spell.c`.

Task 1: Computing edit distance (3 marks)

Implement the function `print_edit_distance()` defined in `spell.c`. This function has two input parameters: `word1` and `word2` (strings).

The function should compute the Levenshtein edit distance between the strings `word1` and `word2`. That is, your program should compute the minimum number of 'edits' required to transform `word1` into `word2`, where edits are defined as letter substitutions, deletions or insertions.

The function should print the result (a single integer) on a single line to the standard output. It should not print anything else to the standard output. The following format string will be useful: `"%d\n"`.

Note: You are advised to use the dynamic programming method discussed in lectures and tutorials (with detailed pseudo-code given in the week 7 tutorial solutions) for finding this edit distance. Other methods are allowed but may not be efficient enough to calculate edit distances for large words, which may also make it difficult to complete the remainder of the assignment tasks.

Task 2: Enumerating all possible edits (3 marks)

Implement the function `print_all_edits()` defined in `spell.c`. This function has one input parameter: `word` (a string).

The function should generate and print all lower case alphabetic strings within a Levenshtein edit distance of 1 from `word`. That is, all of the strings made of alphabetic characters ('abcdefghijklmnopqrstuvwxyz') that can be made from substituting a single letter in `word`, inserting a single letter into `word`, or deleting a single letter from `word`.

The function should print the results (a large number of strings) to the standard output, one string per line. It should not print anything else to the standard output. It's okay for the function to print some words multiple times and it may also print the string `word` itself.

Task 3: Spell checking (3 marks)

Implement the function `print_checked()` defined in `spell.c`. This function has two input parameters: `dictionary` (a `List` pointer), a list of strings representing correctly-spelled words in approximate order of decreasing probability of occurrence; and `document` (another `List` pointer), a list of strings to check for spelling mistakes.

For each string (word) in `document`, the function should check if the string occurs in `dictionary` (that is, if it's a correctly-spelled word).

If the string does occur in `dictionary`, the function should print the string directly to the standard output on its own line. If the string does *not* occur in `dictionary`, then the function should print the string to the standard output followed by a question mark, on its own line. The following format string will be useful: `"%s?\n"`. Strings should be printed in their order of occurrence in `document`.

Note: One method for solving this task simply iterates through the linked list `dictionary` to look for each word in `document`. This method will work, but will not be efficient enough to check large documents. You are advised to come up with an alternative approach.

Task 4: Spelling correction (6 marks)

Implement the function `print_corrected()` defined in `spell.c`. This function has two input parameters: `dictionary` (a `List` pointer), a list of strings representing correctly-spelled words in approximate order of decreasing probability of occurrence; and `document` (another `List` pointer), a list of strings to check for spelling mistakes (and attempt to correct).

For each string (word) in `document`, in order, the function should do the following:

1. If the string occurs in `dictionary` (that is, if it's a correctly-spelled word) the function should print this word directly to the standard output on its own line.
2. If the string does not occur in `dictionary` your function should try to find a 'corrected' version of the string with a Levenshtein edit distance of 1 in `dictionary`.
   If such a word exists, the function should print this word to the standard output on its own line. If multiple such words exist, the function should print only the word that occurs first among them in `dictionary`.
3. If the string does not occur in `dictionary` and no words in `dictionary` have a Levenshtein edit distance of 1 with the string, then your program should try to find a 'corrected' version of the string with a Levenshtein edit distance of 2 in `dictionary`.
   If such a word exists, the function should print this word to the standard output on its own line. If multiple such words exist, the function should print only the word that occurs first among them in `dictionary`.
4. If none of the first three conditions are met, then the function should attempt to find a 'corrected' version of the string with a Levenshtein edit distance of 3 in `dictionary`.
   If such a word exists, the function should print this word to the standard output on its own line. If multiple such words exist, the function should print only the word that occurs first among them in `dictionary`.
5. If none of the previous conditions are met, then the function should print the original string to the standard output followed by a question mark, on its own line. The following format string will be useful: `"%s?\n"`.

Running your program

Once your assignment is completed, you will be able to execute it by running a command of a form depending on the coding task to execute, as follows:

Task 1: To execute your task 1 function, run a command of the following form:

```
./a2 dist word1 word2
```

where `word1` and `word2` are the two alphabetic strings to be used as the two arguments to your function (`word1` and `word2`, respectively). For example, the command `./a2 dist sweet sleep` will run your task 1 function with the strings `"sweet"` as `word1` and `"sleep"` as `word2`.

Task 2: To execute your task 2 function, run a command of the following form:

```
./a2 edits word
```

where `word` is an alphabetic string to be passed as the argument to your function. For example, the command `./a2 edits soup` will run your task 2 function with the string `"soup"` as the argument `word`.

Task 3: To execute your task 3 function, run a command of the following form:

```
./a2 check dictionary_filename < document_filename
```

where `dictionary_filename` is the path of a text file containing 'correctly-spelled' lower-case alphabetic words, one per line; and `document_filename` is the path of a text file containing words which may or may not be 'correctly-spelled', one per line. The strings in these files will be read into the linked lists arguments passed into your function (`dictionary` and `document`, respectively).

For example, the command `./a2 check data/words.txt < document.txt` will run your task 3 function with the list of 'correctly-spelled' words coming from the file `words.txt` inside the local directory `data`, and input words coming from the local file `document.txt`.

Alternatively, run a command of the following form:

```
./a2 check dictionary_filename
```

and type a sequence of lower-case alphabetic words (one per line) directly into your program's standard input. Once you have finished entering words to check, enter a blank line or an end of file marker (either input will cause the program to stop reading from the standard input and start running your function).

Task 4: To execute your task 4 function, run a command of either of the following forms:

```
./a2 spell dictionary_filename < document_filename
```

```
./a2 spell dictionary_filename
```

with similar meanings as the commands for running task 3.

Example output

To help you validate the output format of your functions, we provide some samples of correct output for four basic inputs (one for each function). Download the sample files from the LMS. The files contain one example of correct output from a selection of commands, described in the following table.

These examples are intended to help you confirm that your output follows the formatting instructions. Note that these samples represent only a small subset of the inputs your solution will be tested against after submission: matching output for these inputs doesn't guarantee a correct solution. You are expected to test your functions comprehensively to ensure that they behave correctly for all inputs.

Filename	Command
`jabberwocky.txt`	(this is an example document, place it in a local folder `docs/`)
`dist-helloworld.txt`	`./a2 dist hello world`
`edits-cats.txt`	`./a2 edits cats`
`check-jabberwocky.txt`	`./a2 check data/words-100K.txt < docs/jabberwocky.txt`
`spell-jabberwocky.txt`	`./a2 spell data/words-100K.txt < docs/jabberwocky.txt`

Note that for `edits-cats.txt` there may be **more than one correct output** due to different possible orderings of strings, inclusion or exclusion of repeated strings, and inclusion or exclusion of the word 'cats' itself.

Warning: These files contain Unix-style newlines. **They will not display properly in some text editors, including the default Windows text editor Notepad.**

Written task

The final task of the assignment requires you to write a short report addressing the following topics.

Task 5: Design of algorithms (2 marks)

First, discuss your approach to **task 3 and task 4** in terms of the data structures and algorithms you used and the resulting asymptotic time complexity of these functions. Second, comment on any alternative approaches you considered, and why you ended up choosing your approach for each task.

Your report must be no more than two pages in length (but does not need to be a full two pages long). You may use any document editor to create your report, but **you must export the document as a PDF for submission**. You should name the file `report.pdf`.

Submission

Via the LMS, submit a single archive file (e.g. `.zip` or `.tar.gz`) containing **all files required to compile your solution (including `Makefile`) plus your report (as a PDF).** When extracted from this archive on the School of Engineering student machines (a.k.a. `dimefox`), your submission should compile without any errors simply by running the `make` command.

Please note that when compiling your program we will use the original versions of the files marked '**Do not change**' in the 'Provided files' list. Any changes you have made to these files will be lost. This may lead to compile errors, which will result in a mark penalty. **Do not modify these files.**

Furthermore, **do not include the `data` directory** in your submission. This will make your submission very large which will may cause large delays uploading to the LMS. We will supply our own data folder during testing.

Submissions will close automatically at the deadline. As per the Subject Guide, the late penalty is 20% of the available marks for this assignment for each day (or part thereof) overdue. Note that network and machine problems right before the deadline are not sufficient excuses for a late or missing submission. Please see the Subject Guide for more information on late submissions and applying for an extension.

Academic honesty

All work is to be done on an individual basis. Any code sourced from third parties must be attributed. All submissions will be subject to automated similarity detection. Where academic misconduct is detected, all parties involved will be referred to the School of Engineering for handling under the University Discipline procedures. Please see the Subject Guide and the 'Academic Integrity' section of the LMS for more information.

Marking

Each of the four coding tasks will be marked as follows:

- Half of the available marks will be for the suitability of your approach to solving the problem. Note that this mark is available even for a program that does not always produce the correct output.
- Half of the available marks will be for the correctness of your program's output on a suite of test inputs. You will lose partial marks if there are minor discrepancies in output formatting, if there are mistakes in the output, if your program crashes on certain inputs, or if your program takes too long to produce results.
  Note that a small portion of our tests will involve a very large number of words. For full marks, your program is expected to be able to handle these large input sizes in a reasonable amount of time.
  A reminder that we will compile and test your program on the School of Engineering student machines (a.k.a. `dimefox`).

The written report will be marked as follows:

- 1 of the available marks will be for the clarity and accuracy of your discussion of the asymptotic time complexity of your approach(es).

- 1 of the available marks will be for the clarity and accuracy of the discussion of your choice of algorithms and data structures and any alternatives you considered.
- Additionally, 1 mark will be deducted if your report is too long (past the two-page limit) or is not a PDF document.

So far, the available marks add up to only 17. However, there are a total of 20 marks available for this assignment. The final 3 marks are allocated to the following:

- 1 mark will be for the quality of your code. You will lose part or all of these marks if your program is poorly designed (e.g. with lots of repetition, poor functional decomposition, or a lack of modularity), or if your program is difficult to follow (e.g. due to missing or unhelpful comments, or unclear variable names).
- 1 mark will be for the memory safety of your program. You will lose part or all of this mark if your program has memory leaks.
- 1 mark will be for your code's compilation and your Makefile. You will lose part or all of this mark if your program does not compile on `dimefox` when we run `make` inside the directory containing your submission.