Assignment 2
Word Chains

Change Log

We may make minor changes to the spec to address/clarify some outstanding issues. These may require minimal changes in your design/code, if at all. Students are strongly encouraged to check the change log regularly.

25 September

- Example for stage 3 corrected

Version 1: Released on 22 September 2017

Objectives

The assignment aims to give you more independent, self-directed practice with

- advanced data structures, especially graphs
- graph algorithms
- asymptotic runtime analysis

Admin

Marks	3 marks for stage 1 (correctness) 3 marks for stage 2 (correctness) 3 marks for stage 3 (correctness) 3 marks for stage 4 (correctness) 2 marks for complexity analysis 1 mark for style ——————— Total: 15 marks
Due	23:59 on **Wednesday** 18 October (week 12)
Late	2.25 marks (15%) off the ceiling per day late (e.g. if you are 25 hours late, your maximum possible mark is 10.5)

Aim

A **word chain** is a sequence of $k \geq 1$ words:

$$\omega_1 \rightarrow \omega_2 \rightarrow \omega_3 \rightarrow \ldots \rightarrow \omega_{k-1} \rightarrow \omega_k$$

in which

- words appear in alphabetical order, and
- each word ω_{i+1} is related to its predecessor ω_i by
    1. changing one letter, e.g. `bear` → `beer`
    2. *or* adding or removing one letter, e.g. `compete` → `complete` or `bear` → `ear`.

An example is the following word chain of length $k = 6$:

`bear` → `dear` → `ear` → `ears` → `hears` → `hearts`

Your task is to write a program for computing the *longest* word chains that can be built from a given collection of words.

Your program should:

- prompt the user to input
  - a positive number *n*
  - *n* words
- **Task 1:** compute and output, for each word ω, all words that could immediately follow ω in a word chain,
- **Task 2:** compute and output
  a. the maximum length of a word chain that can be built from the given words,
  b. all word chains of maximum length that can be built from the given words.

Your program should include a time complexity analysis, in Big-Oh notation, for

1. your implementation for task 1, depending on the number *n* of words and the maximum length *m* of a word;
2. your implementation for task 2, depending on the number *n* of words.

Note:

- You may assume that
  - the user input is correct (a number $n \geq 1$ followed by *n* words);
  - the user inputs the words in alphabetical order;
  - no word has more than 19 characters;
  - there will be no more than 1000 words.

- It may not be immediately obvious, but this problem is best understood and solved as a graph problem. Hence, if you find any of the following ADTs from the lectures useful, then you can, and indeed are encouraged to, use them with your program:

  - stack ADT : `stack.h`, `stack.c`
  - queue ADT : `queue.h`, `queue.c`
  - graph ADT : `Graph.h`, `Graph.c`
  - weighted graph ADT : `WGraph.h`, `WGraph.c`

  You are free to modify any of the four ADTs for the purpose of the assignment (but without changing the file names). If your program is using one or more of these ADTs, you should submit both the header and implementation file, even if you have not changed them.

- Your main program file should start with a comment: `/* … */` that contains the time complexity of your solutions for task 1 and task 2, together with an explanation.

Stage 1 (3 marks)

For stage 1, you should demonstrate that you can build the underlying graph correctly *under the assumption that all words are of equal length.*

All tests for this stage will be such that all words have the same length and all given words together form a word chain. Hence, all you need to do for task 2 at this stage is to output the total number of words (= maximum length of a chain) and all words (= the only maximal chain).

Here is an example to show the desired behaviour of your program for a stage 1 test:

```
prompt$ ./wordchains
Enter a number: 6
Enter word: bear
Enter word: dear
Enter word: hear
Enter word: heat
Enter word: neat
Enter word: seat

bear: dear hear
dear: hear
hear: heat
heat: neat seat
neat: seat
seat:

Maximum chain length: 6
Maximal chains:
bear -> dear -> hear -> heat -> neat -> seat
```

Stage 2 (3 marks)

For stage 2, you should demonstrate that you can find a *single* maximal chain under the assumption that all words are of equal length.

All tests for this stage will be such that all words have the same length and there is only one maximal word chain.

Here is an example to show the desired behaviour of your program for a stage 2 test:

```
prompt$ ./wordchains
Enter a number: 8
Enter word: bear
Enter word: beer
Enter word: dear
Enter word: deer
Enter word: hear
Enter word: heat
Enter word: rear
Enter word: seat

bear: beer dear hear rear
beer: deer
dear: deer hear rear
deer:
hear: heat rear
heat: seat
rear:
seat:

Maximum chain length: 5
Maximal chains:
bear -> dear -> hear -> heat -> seat
```

Stage 3 (3 marks)

For stage 3, you should extend your program for stage 2 such that words can be of different length.

All tests for this stage will be such that there is only one maximal word chain.

Here is an example to show the desired behaviour of your program for a stage 3 test:

```
prompt$ ./wordchains
Enter a number: 6
Enter word: cast
Enter word: cat
Enter word: cats
Enter word: cost
Enter word: most
Enter word: moyst

cast: cat cost
cat: cats
cats:
cost: most
most: moyst
moyst:

Maximum chain length: 4
Maximal chains:
cast -> cost -> most -> moyst
```

Stage 4 (3 marks)

For stage 4, you should extend your program for stage 3 such that it outputs, in alphabetical order, all word chains of maximal length.

Here is an example to show the desired behaviour of your program for a stage 4 test:

```
prompt$ ./wordchains
Enter a number: 5
Enter word: cast
Enter word: cat
Enter word: cats
Enter word: cost
Enter word: most

cast: cat cost
cat: cats
cats:
cost: most
most:

Maximum chain length: 3
Maximal chains:
```

```
cast -> cat -> cats
cast -> cost -> most
```

Note:

- It is required that the maximal chains be printed in alphabetical order.

Testing

We have created a script that can automatically test your program. To run this test you can execute the `dryrun` *program for the corresponding assignment, i.e.* `assn2`*. It expects to find, in the current directory, the program* `wordchains.c` *and any of the admissible ADTs (*`Graph,WGraph,stack,queue`*) that your program is using, even if you use them unchanged. You can use dryrun as follows:*

```
prompt$ ~cs9024/bin/dryrun assn2
```

Please note: Passing the dryrun tests does not guarantee that your program is correct. You should thoroughly test your program with your own test cases.

Submit

For this project you will need to submit a file named `wordchains.c` and, optionally, any of the ADTs named `Graph,WGraph,stack,queue` that your program is using, even if you have not changed them. You can either submit through WebCMS3 or use a command line. For example, if your program uses the `Graph` ADT and the `queue` ADT, then you should submit:

```
prompt$ give cs9024 assn2 wordchains.c Graph.h Graph.c queue.h queue.c
```

Do not forget to add the time complexity to your main source code file `wordchains.c`.

You can submit as many times as you like — later submissions will overwrite earlier ones. You can check that your submission has been received on WebCMS3 or by using the following command:

```
prompt$ 9024 classrun -check assn2
```

Marking

This project will be marked on functionality in the first instance, so it is very important that the output of your program be ***exactly*** correct as shown in the examples above. Submissions which score very low on the automarking will be looked at by a human and may receive a few marks, provided the code is well-structured and commented.

Programs that generate compilation errors will receive a very low mark, no matter what other virtues they may have. In general, a program that attempts a substantial part of the job and does that part correctly will receive more marks than one attempting to do the entire job but with many errors.

Style considerations include: readability, structured programming and good commenting.

Plagiarism

Group submissions will not be allowed. Your program must be entirely your own work. Plagiarism detection software will be used to compare all submissions pairwise (including submissions for similar projects in previous years, if applicable) and serious penalties will be applied, particularly in the case of repeat offences.

Do not copy from others; do not allow anyone to see your code, not even after the deadline

Please refer to the on-line sources to help you understand what plagiarism is and how it is dealt with at UNSW:

- Plagiarism and Academic Integrity
- UNSW Plagiarism Procedure

Help

See FAQ for some hints on how to get started.

Finally …

Best of luck and have fun! Michael