Large Assignment
Word Sequences

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.

27 October

- Example for stage 3 corrected.

Version 1: Released on 22 October 2021

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	2 marks for stage 1 (correctness) 3 marks for stage 2 (correctness) 2 marks for stage 3 (correctness) 3 marks for stage 4 (correctness) 1 mark for complexity analysis 1 mark for style Total: 12 marks
Due	11:00:00am on **Monday** 15 November (week 10)
Late	2 marks (= ⅙ of the maximum mark) off the ceiling per day late (e.g. if you are 25 hours late, your maximum possible mark is 8)

Aim

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

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

in which

- words are ordered alphabetically, and
- two consecutive words ω_i and ω_{i+1} differ by only one letter:
  1. either they are of the same length with one letter different, e.g. `tail` → `tall`
  2. *or* one letter is missing or added, e.g. `grater` → `greater` or `scent` → `sent`.

An example is the following word sequence of length $k = 8$:

`cold` → `cord` → `core` → `corse` → `horse` → `hose` → `host` → `hot`

Your task is to develop a program to compute the *longest* word sequences that can be created given a collection of words.

Input

Your program should start by prompting the user to input a positive number *n*, then prompt the user to input *n* words.

Hint:

You may assume that

- the input is syntactically correct (a number $n \geq 1$ followed by *n* words);
- all words are input in alphabetical order;
- each single word has a maximum of 31 letters;
- only the 26 lowercase letters `a` – `z` will be used;
- the maximum number of words will be 100.

Output

- **Task 1:** For each word ω, compute and output all the words that could be used as the next word after ω in a word sequence.
- **Task 2:** Compute and output
  a. the maximum length of a word sequence that can be built from the words in the input,
  b. all word sequences of maximum length that can be built from the set of words in the input.

Stage 1 (2 marks)

For stage 1, you need to demonstrate that you can build the underlying graph *under the assumption that all words have the same length.*

Any test for this stage will have only words of equal length and such that all the words together form a word sequence. Hence, this stage focuses on Task 1, and for Task 2 you can just output the total number of words (= maximum length of a sequence) and a sequence containing all the words (= the only maximal sequence).

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

```
prompt$ ./words
Enter a number: 6
Enter a word: lad
Enter a word: lag
Enter a word: lap
Enter a word: tap
Enter a word: tip
Enter a word: top

lad: lag lap
lag: lap
lap: tap
tap: tip top
tip: top
top:

Maximum sequence length: 6
Maximal sequence(s):
lad -> lag -> lap -> tap -> tip -> top
```

Stage 2 (3 marks)

For stage 2, you should demonstrate that you can find *one* maximal sequence under the assumption that all words have the same length.

Any test for this stage will be such that all words are of equal length and such that there is only one maximal word sequence.

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

```
prompt$ ./words
Enter a number: 8
Enter a word: bad
Enter a word: ban
Enter a word: dad
Enter a word: dan
Enter a word: lad
Enter a word: lap
Enter a word: mad
Enter a word: tap

bad: ban dad lad mad
ban: dan
dad: dan lad mad
dan:
lad: lap mad
lap: tap
mad:
tap:

Maximum sequence length: 5
Maximal sequence(s):
bad -> dad -> lad -> lap -> tap
```

Stage 3 (2 marks)

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

Tests for this stage are also guaranteed to have just one word sequence of maximum length.

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

```
prompt$ ./words
Enter a number: 6
Enter a word: east
Enter a word: eat
Enter a word: eats
Enter a word: fast
Enter a word: fist
Enter a word: foist

east: eat fast
eat: eats
eats:
fast: fist
fist: foist
foist:

Maximum sequence length: 4
Maximal sequence(s):
east -> fast -> fist -> foist
```

Stage 4 (3 marks)

For stage 4, you should extend your stage 3 program such that it outputs

- all word sequences of maximal length
- in alphabetical order.

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

```
prompt$ ./words
Enter a number: 5
Enter a word: east
Enter a word: eat
Enter a word: eats
Enter a word: fast
Enter a word: fist

east: eat fast
eat: eats
eats:
fast: fist
fist:

Maximum sequence length: 3
Maximal sequence(s):
east -> eat -> eats
east -> fast -> fist
```

Note:

- It is a requirement that the sequences are output in alphabetical order.

Complexity Analysis (1 mark)

Your program should include a time complexity analysis for the worst-case asymptotic running time of your program, in Big-Oh notation, depending on the size of the input:

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.

Your main program file `words.c` should start with a comment: `/* … */` that contains the time complexity of your program in Big-Oh notation, together with a short explanation.

Hints

If you find any of the following ADTs from the course useful, then you can, and indeed are encouraged to, use them with your program:

- linked list ADT : `list.h`, `list.c`
- stack ADT : `stack.h`, `stack.c`
- queue ADT : `queue.h`, `queue.c`
- priority queue ADT : `PQueue.h`, `PQueue.c`
- graph ADT : `Graph.h`, `Graph.c`
- weighted graph ADT : `WGraph.h`, `WGraph.c`

You are free to modify any of the six 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 `words.c` should start with a comment: `/* … */` that contains the time complexity of your program in Big-Oh notation, together with a short explanation.

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. `words`. It expects to find, in the current directory, the program `words.c` and any of the admissible ADTs (`Graph,WGraph,stack,queue,PQueue,list`) that your program is using, even if you use them unchanged. You can use dryrun as follows:

```
prompt$ 9024 dryrun words
```

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 `words.c` and, optionally, any of the ADTs named `Graph,WGraph,stack,queuePQueue,list` 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 `list` ADT, then you should submit:

```
prompt$ give cs9024 assn words.c Graph.h Graph.c list.h list.c
```

Do not forget to add the time complexity to your main source code file `words.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 assn
```

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
- Good commenting

Plagiarism

Group submissions will not be allowed. Your programs must be entirely your own work. Plagiarism detection software will be used to compare all submissions pairwise (including submissions for similar assessments in previous years, if applicable) and serious penalties will be applied, including an entry on UNSW's plagiarism register.

- ***Do not copy ideas or code from others***
- ***Do not use a publicly accessible repository or allow anyone to see your code***

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 Policy Statement
- UNSW Plagiarism Procedure

Help

See FAQ for some additional hints.

Finally …

Have fun! Michael