University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	**Algorithmics II (H)**			
Coursework Number	**1 of 1**			
Deadline	**Time:**	**4.30pm**	**Date:**	**17 November 2017**
% Contribution to final course mark	**20%**		**This should take this many hours:**	**20**
Solo or Group ✓	**Solo**	✓	**Group**	
Submission Instructions	Via Moodle – see Section G			
Marking Criteria	See Section H			
Please Note: This coursework cannot be redone				

Code of Assessment Rules for Coursework Submission

Deadlines for the submission of coursework which is to be formally assessed will be published in course documentation, and work which is submitted later than the deadline will be subject to penalty as set out below. The primary grade and secondary band awarded for coursework which is submitted after the published deadline will be calculated as follows:

(i) in respect of work submitted not more than five working days after the deadline
   a. the work will be assessed in the usual way;
   b. the primary grade and secondary band so determined will then be reduced by two secondary bands for each working day (or part of a working day) the work was submitted late.

(ii) work submitted more than five working days after the deadline will be awarded Grade H.

Penalties for late submission of coursework will not be imposed if good cause is established for the late submission. You should submit documents supporting good cause via MyCampus.

Penalty for non-adherence to Submission Instructions is 2 bands

Algorithmics II (H)

Assessed Exercise, 2017-18

Dynamic Programming with Memoisation

A. General

This is the only assessed practical exercise for Algorithmics II (H), and accounts for 20% of the assessment for this course. As a rough guide, it is intended that an average student should be able to obtain a band in the "B" range by putting in no more than 20 hours of work, and you are advised not to spend significantly more time than this on the exercise.

The exercise is to be done individually. Some discussion of the exercise among members of the class is to be expected, but close collaboration or copying of code, in any form, is strictly forbidden – see the School's plagiarism policy, contained in Appendix A of the Undergraduate Class Guide (available from http://moodle2.gla.ac.uk/mod/resource/view.php?id=749301).

B. Deadline for submission

The hand-out date for the exercise is Friday 27 October 2017, by which time all the relevant material will have been covered in lectures. The deadline for submission is **4.30pm, Friday 17 November 2017**. The course web page on Moodle will be used for this exercise, providing setup files and an electronic submission mechanism. Guidance as to what should be submitted is given in Section G. The aim will be to return marked exercises with individual feedback by Friday 8 December. In addition, general feedback on the exercise will be provided via the course web page on Moodle.

C. Specification

The main purpose of this exercise is to implement iterative and recursive dynamic programming algorithms for various problems involving strings, including the Longest Common Subsequence (LCS) problem, the Edit Distance (ED) problem, and the problem of finding a Highest Scoring Local Similarity (HSLS) using the Smith-Waterman (SW) algorithm. The recursive dynamic programming algorithms are to be implemented firstly without, and then with, memoisation.

The next part of this exercise involves conducting some simple experiments on these implementations using (i) sample input strings that are provided, and (ii) randomly-generated strings. These will aim to compare and contrast the different algorithm implementations with respect to runtime.

To evaluate the true benefit of memoisation, we need to use this technique in combination with "virtual initialisation" (i.e., avoiding explicitly initialising our dynamic programming table). Since Java automatically initialises integer arrays so that all elements are 0 upon instantiation, for a meaningful comparison we need to use a programming language that does not do automatic initialisation. To this end, the language of implementation for this exercise will be C.

D. Getting started

On the course web page (http://moodle2.gla.ac.uk/course/view.php?id=971), click on the link "Setup file" under the heading "Assessed Exercise". You will obtain a zip file containing a C source code file entitled `AssEx.c`, a listing of the given source code and three files containing strings. `AssEx.c` contains some skeleton code, the main parts of which are as follows:

- global variables
- a function to parse command-line arguments (`getArgs`)
- a function to read two strings from a text file (`readStrings`)
- a function to generate two strings randomly (`generateStrings`)
- the main function (`main`)

Take some time to study the source code that has been provided. The code should be compiled using `gcc AssEx.c -o AssEx -lm` and it should be executed using the following command-line arguments: `AssEx -g M N A -f filename.txt -t {LCS|ED|SW} -i -r -m -p` (if you are working on Linux, which is recommended, you may need to prefix "`AssEx`" by "`./`").

A description of these arguments is as follows:
`-g M N A`: generate random strings of lengths `M` and `N` over an alphabet of size `A`;
`-f filename.txt`: read in the two strings from `filename.txt`;
`-t {LCS|ED|SW}`: problem to solve, where
`LCS` = Longest Common Subsequence
`ED` = Edit Distance
`SW` = Highest Scoring Local Similarity
`-i`: carry out iterative dynamic programming;
`-m`: carry out recursive dynamic programming with memoisation;
`-r`: carry out recursive dynamic programming without memoisation;
`-p`: print dynamic programming table and optimal alignment in the case of LCS.

The command line arguments must include:
- exactly one of `-g M N A` and `-f filename.txt`;
- `-t LCS` or `-t ED` or `-t SW`;
- at least one of `-i`, `-m` and `-r`.

The argument `-p` is optional.

As part of this exercise you will be adding code to `AssEx.c` to implement the functionality described by the above command-line arguments. This will involve modifying code in existing functions as well as adding new variables and functions. You should not need to create other modules, but you are not prevented from doing so.

E. Implementation tasks

1. Add code to compute the length of an LCS of the two strings `x` and `y` using iterative dynamic programming, and to measure the time taken. The output should be along the following lines:

```
>AssEx -g 10000 10000 4 -i -t LCS

Longest Common Subsequence

Iterative version
Length of a longest common subsequence is: 6511
Time taken: 1.48 seconds
```

2. Repeat Task 1 to compute the ED between the two strings `x` and `y` and the length of a HSLS between the two strings `x` and `y`, in both cases using iterative dynamic programming. Again the time taken should be measured. The output should be along the following lines:

```
>AssEx -g 10000 10000 4 -i -t ED
Edit Distance
```

```
Iterative version
Edit distance is: 5174
Time taken: 1.67 seconds

>AssEx -g 10000 10000 4 -i -t SW
Smith-Waterman algorithm

Iterative version
Length of a highest scoring local similarity is: 1128
Time taken: 1.75 seconds
```

Try to create suitable helper functions to avoid repetition of code between the LCS, ED and HSLS algorithms.

3. Add code to print the dynamic programming table in the case of LCS, ED or HSLS. The output should be along the following lines:

```
>AssEx -f SWstrings.txt -i -t SW -p
Smith-Waterman algorithm

Iterative version
Length of a highest scoring local similarity is: 4
Dynamic programming table:
        0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
          C G C A T G T G A C A G C T T C A
       ___________________________________
 0     | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 1   G | 0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0
 2   A | 0 0 0 0 1 0 0 0 0 2 1 1 0 0 0 0 0 1
 3   A | 0 0 0 0 1 0 0 0 0 1 1 2 1 0 0 0 0 1
 4   C | 0 1 0 1 0 0 0 0 0 0 2 1 1 2 1 0 1 0
 5   T | 0 0 0 0 0 1 0 1 0 0 1 1 0 1 3 2 1 0
 6   A | 0 0 0 0 1 0 0 0 0 1 0 2 1 0 2 2 1 2
 7   C | 0 1 0 1 0 0 0 0 0 0 2 1 1 2 1 1 3 2
 8   A | 0 0 0 0 2 1 0 0 0 1 1 3 2 1 1 0 2 4
 9   T | 0 0 0 0 1 3 2 1 0 0 0 2 2 1 2 2 1 3
10   C | 0 1 0 1 0 2 2 1 0 0 1 1 1 3 2 1 3 2
11   A | 0 0 0 0 2 1 1 1 0 1 0 2 1 2 2 1 2 4
12   T | 0 0 0 0 1 3 2 2 1 0 0 1 1 1 3 3 2 3
13   C | 0 1 0 1 0 2 2 1 1 0 1 0 0 2 2 2 4 3
14   T | 0 0 0 0 0 1 1 3 2 1 0 0 0 1 3 3 3 3
15   A | 0 0 0 0 1 0 0 2 2 3 2 1 0 0 2 2 2 4
16   C | 0 1 0 1 0 0 0 1 1 2 4 3 2 1 1 1 3 3
17   T | 0 0 0 0 0 1 0 1 0 1 3 3 2 1 2 2 2 2
18   C | 0 1 0 1 0 0 0 0 0 0 2 2 2 3 2 1 3 2

Time taken: 0.00 seconds
```

Ensure that all columns line up regardless of the lengths of the strings or the sizes of the values contained in the table.

4. Add code to implement the recursive dynamic programming algorithms *without* memoisation in the case of LCS and ED (we will not be using recursive dynamic programming with the HSLS problem). Note that in this case, the objective is *not* to compute solutions to the LCS and ED problems, but rather, to use the dynamic programming table entries to keep a count of how many times that particular entry was computed. In addition, the output should include the total number of times the table entries are computed. The output should be along the following lines:

```
>AssEx -g 8 8 4 -r -t ED
Edit Distance
```

```
Recursive version without memoisation
Total number of times a table entry computed: 34096
Time taken: 0.00 seconds

>AssEx -g 8 8 4 -r -t ED -p
Edit Distance

Recursive version without memoisation
Dynamic programming table:
          0    1    2    3    4    5    6    7    8
               B    B    B    B    C    D    D    C
     ____________________________________________________
0    | 1755 2774 1550  765  307   99    8    1    1
1  D | 2192 1755 1019  531  234   73   26    8    1
2  A |  514  437  299  189  108   53   20    6    1
3  A |   97   77   61   49   32   23   10    4    1
4  B |   11   20   16   12   17    9    4    2    1
5  D |   12   11    9    7    5    3    2    2    1
6  D |    1    1    1    1    1    1    2    2    1
7  D |    0    0    0    0    0    0    1    2    1
8  D |    0    0    0    0    0    0    0    1    1

Total number of times a table entry computed: 15241
Time taken: 0.00 seconds
```

For example in the above table, the number of times entry (3,5) was computed during recursive dynamic programming without memoisation was 23.

5. Add code to implement the recursive dynamic programming algorithms *with* memoisation in the case of LCS and ED. Ensure that you use "virtual initialisation" (i.e., the technique described in lectures for avoiding explicit initialisation) of your dynamic programming table. The table should this time be used to compute solutions to the LCS and ED problems. In addition, information is required about the number of table entries computed and the proportion of table entries computed (expressed as a percentage to one decimal place). The output should be along the following lines:

```
>AssEx -g 8 8 3 -m -t LCS
Longest Common Subsequence

Recursive version with memoisation
Length of a longest common subsequence is: 6
Number of table entries computed: 19
Proportion of table computed: 23.5%
Time taken: 0.00 seconds

>AssEx -g 8 8 3 -m -t LCS -p
Longest Common Subsequence

Recursive version with memoisation
Dynamic programming table:
       0 1 2 3 4 5 6 7 8
         C C B B A C B C
     ___________________
0    | 0 0 - 0 0 0 - - -
1  C | - 1 1 1 1 1 - - -
2  B | 0 1 - 2 2 2 - - -
3  C | - 1 2 2 2 2 3 - -
4  B | 0 1 2 3 3 3 3 - -
5  B | - - - 3 4 4 4 - -
6  B | - - - - 4 4 - 5 -
7  C | - - - - - - 5 - 6
8  B | - - - - - - - 6 6
```

```
Length of a longest common subsequence is: 6
Number of table entries computed: 39
Proportion of table computed: 48.1%
Time taken: 0.00 seconds
```

Take special care to update your function to print out the dynamic programming table, bearing in mind that some entries may not have been computed. In the table, '-' should be used to represent a table entry that has not been computed.

6. Add code to print out an *optimal alignment* of the two strings `x` and `y` in the case of LCS. In general, an optimal alignment is a way of lining up the characters of `x` with the characters of `y` to give the minimum number of operations required to transform `x` into `y`, where the allowed operations are (i) delete a character of `x`, (ii) insert a character of `y` or (iii) retain a character that is in the LCS of `x` and `y`. To illustrate this, consider the following example:

```
>AssEx -g 6 6 3 -i -p -t LCS
Longest Common Subsequence

Iterative version
Length of a longest common subsequence is: 4
Dynamic programming table:
       0 1 2 3 4 5 6
         B B C C A B
      ---------------
0     | 0 0 0 0 0 0 0
1   C | 0 0 0 1 1 1 1
2   B | 0 1 1 1 1 1 2
3   B | 0 1 2 2 2 2 2
4   C | 0 1 2 3 3 3 3
5   B | 0 1 2 3 3 3 4
6   C | 0 1 2 3 4 4 4

Optimal alignment:
CBBCBC||
 ||| |
|BBC|CAB
Time taken: 0.00 seconds
```

Here, to transform `x` into `y`, carry out the following operations: delete `C` from `x`, retain `BBC` from LCS of `x` and `y`, delete `B` from `x`, retain `C` from LCS of `x` and `y`, and insert `AB` from `y`.

In general an optimal alignment has three lines: the first line contains characters of `x` that are to be deleted or else belong to the LCS, together with vertical bars corresponding to characters of `y` that are to be inserted; the second line contains vertical bars corresponding to LCS characters; finally the third line contains characters of `y` that are to be inserted or else belong to the LCS, together with vertical bars corresponding to characters of `x` that are to be deleted.

F. Written report

Write a short report (1 page of A4 should suffice) in Word or LaTeX that answers the following questions / addresses the following requirements:

7. Include a status report which, for any non-working implementation, should state clearly what happens on compilation (in the case of compile-time errors) or on execution (in the case of run-time errors). If there are no compile-time or run-time errors, all that is required is a single sentence indicating this.

8. Try running the iterative algorithm and the recursive algorithm with memoisation to compute an LCS for the strings in files `strings10000a.txt` and `strings10000b.txt`. What general behaviour do you notice and why is this the case?

9. Try generating strings of increasing lengths (assume that both strings have the same length) over an alphabet of size 4, and executing all three types of algorithm to find an LCS. What trends do you notice, and why? When answering this question, you should refer to (i) the relative speed of the algorithms and (ii) the largest instances solvable within, say, a minute.

10. Try generating strings of length 5000 (again assume that both strings have the same length) over an alphabet that increases from 2 up to 10, executing the iterative algorithm and the recursive algorithm with memoisation to find an LCS. What do you notice regarding how the time taken in each case varies with the growth in the alphabet size? Why do you think this is?

G. How to submit

Your submission to this exercise should include your source code, code listing file (in pdf form) and written report (again in pdf form). All submissions are to be made electronically and no hard-copy submissions are required. More information about each of these components is given as follows:

- *Source code*: include `AssEx.c` and any other source code files, if you created additional ones. Ensure that you remove any debugging code that may generate large volumes of output on large input files.

- *Code listing file*: include a formatted listing of your source code in pdf form. In order to produce the pdf file for `AssEx.c`, for example, you should use the following Unix commands:

```
a2ps –A fill –Ma4 AssEx.c [more source files] –o codeListing.ps
ps2pdf -sPAPERSIZE=a4 codeListing.ps
```

- *Written report*: this should take the form of a pdf file that addresses Tasks 7-10 in Section F.

In order to make your submission, follow the submission link in the section entitled "Assessed Exercise" within the Moodle page (http://moodle2.gla.ac.uk/course/view.php?id=971) for the course. You will need to submit a zip file or tar.gz file which should be named `AlgII_<family name>_<given name>.zip` or `AlgII_<family name>_<given name>.tar.gz`, e.g., `AlgII_Manlove_David.zip`.

The zip or tar.gz file should contain your source code, code listing file and written report as explained above.

Before you submit, ensure that your zip/tar.gz file contains the version of the files that you wish to have assessed. You can submit as many times as you wish; the last submission made before the exercise deadline will be the one that is used, and your code at the time of that submission should correspond to the code listing pdf files.

You will be required to complete a Declaration of Originality when submitting via Moodle. For the purposes of this exercise, the declarations that you make apply to all parts of your submission. If you have used any external sources, be sure to acknowledge them in your implementation report.

After the deadline, the submissions will be run through the School's in-house plagiarism detection program. As part of the marking, your code for Tasks 1-6 will be run against an acceptance test.

H. Marking scheme

Submissions will be marked according to the following breakdown of numerical marks:

Task	***Description***	***Marks***
1	Iterative LCS	3
2	Iterative ED and HSLS	2
3	Print DP table	3
4	Recursive DP without memoisation	3
5	Recursive DP with memoisation	4
6	Optimal alignment	3
7	Status report	0
8	Experiments on strings10000a.txt and strings10000b.txt	1
9	Experiments on strings with increasing lengths	5
10	Experiments on strings with increasing alphabet sizes	4
	Quality of code and general presentation	2
	Total	**30**

Marks for Tasks 1-6 will be awarded on the basis of correctness, efficiency and demonstration of understanding of the underlying theory of dynamic programming.

Numerical totals will then be converted to bands according to the School's standard percentage-band translation table.