University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	**Algorithmics II (H)**			
Coursework Number	**1 (of 1)**			
Deadline	**Time:**	**4.30pm**	**Date:**	**7 November 2025**
% Contribution to final course mark	**20**		**This should take at most this many hours:**	**20**
Solo or Group ✓	**Solo**	✓	**Group**	
Submission Instructions	**Via Moodle – see Page 8**			
Who Will Mark This? ✓	**Lecturer** ✓	**Tutor**	**Other**	
Feedback Type? ✓	**Written** ✓	**Oral**	**Both**	
Individual or Generic? ✓	**Generic**	**Individual**	**Both** ✓	
Other Feedback Notes				
Please Note: This Coursework cannot be Re-Done				

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

Marking Criteria
See Page 8

Algorithmics II (H)

Assessed Exercise 2025-26

Applications of Suffix Trees

1. General

This is the only practical exercise for Algorithmics II (H). It accounts for 20% of your final mark for this course. A numerical mark out of 30 will be computed (see Section 12 for the marking scheme), and the corresponding band on the University's 23-point scale (A1-H) will be returned to you.

As a rough guide, it is intended that you should be able to complete this exercise by putting in no more than 20 hours of work on average, and you are advised not to spend significantly more time than this on the exercise. The language of implementation is Java.

The exercise is to be done individually. Close collaboration or copying of code, in any form, is strictly forbidden – see the School's plagiarism policy, which is contained in Appendix A of the Undergraduate Class Guide, available from:
https://moodle.gla.ac.uk/course/view.php?id=21505#section-3.

2. Deadline for submission

The hand-out date for the exercise is Thursday 16 October 2025, by which time all the relevant material will have been covered in lectures. The deadline for submission is **4.30pm, Friday 7 November 2025**. The course web page on Moodle will be used for this exercise, providing setup files and an on-line submission mechanism. Guidance as to exactly what should be submitted is given in Section 10. The intention is that marked exercises, together with individual feedback, will be returned by Friday 28 November 2025. In addition, general feedback regarding this exercise will be provided via the course web page.

3. Use of AI tools

It is recognised that there are a wide variety of AI tools that are readily available, that some students might decide to use to assist with the completion of this exercise. You must declare the use of these tools, and any other external sources, when submitting your exercise (see Section 10 for details about how to do this). Furthermore, **there will be a question relating to the assessed exercise in the final exam**, and therefore it is strongly in your interests to attempt the exercise yourself, to ensure that you are able to understand the structure of the suffix tree and its applications.

4. Specification

This assessed exercise is concerned with a number of text and string problems, each of which can be solved by an efficient algorithm using an application of the suffix tree data structure. The aim of the exercise is to illustrate the wide range of practical applications of this versatile data structure, and to provide experience of manipulating the suffix tree – this should aid understanding of the structure itself.

The text and string problems that we will be concerned with are as follows:

Task 1: searching for a given string in a piece of text;
Task 2: determining all occurrences of a given string in a piece of text;
Task 3: finding a longest repeated substring in a piece of text;
Task 4: finding a longest common substring of two pieces of text.

You are given Java code for reading a specified text file into a string `s` and building the suffix tree for `s`. (The suffix tree construction algorithm works on the principle of repeated insertion of suffixes together with edge-splitting; for the purposes of this assessed exercise, it may be helpful, but it is not essential, to understand how the construction code works.) Your task is to extend the given code in order to implement solutions to each of Tasks 1-4 as indicated above, displaying the results for the user in an appropriate format (as suggested below).

5. Setup

Download the zip file containing the setup code from the Assessment section of the course web page on Moodle (https://moodle.gla.ac.uk/course/view.php?id=49793). This contains the Java classes `Main.java` and `FileInput.java`, two large sample text files `text1.txt` and `text2.txt`, and the subdirectory `SuffixTreePackage`. This subdirectory contains the Java classes `SuffixTree.java`, `SuffixTreeNode.java`, `SuffixTreeAppl.java`, `Task1Info.java`, `Task2Info.java`, `Task3Info.java` and `Task4Info.java`.

To compile the program, issue the command `javac Main.java` from the top-level directory. To run the program, issue commands according to the following required syntax from the same directory.

`java Main SearchOne <filename> <query string>` for Task 1
`java Main SearchAll <filename> <query string>` for Task 2
`java Main LRS <filename>` for Task 3
`java Main LCS <filename1> <filename2>` for Task 4

6. The given code

`Main.java` is the main class, and the entry point for the program. It is intended that this class should implement a command-line application for accessing the suffix tree algorithms that have been implemented as part of Tasks 1-4 listed in Section 4. The `main` method in this class should use `FileInput.java` to handle input from text files.

The package `SuffixTreePackage` contains seven classes, as follows. The class `SuffixTreeNode` is used to represent a node of a suffix tree; it includes a method to add a child to a given node and simple accessor and mutator methods. The class `SuffixTree` is used to represent a suffix tree; it includes methods for building a suffix tree and also various accessor and mutator methods. The class `SuffixTreeAppl` contains methods for accessing the functions relating to Tasks 1-4. The given method corresponding to Task N returns an object of class `TaskNInfo` (a class of the package `SuffixTreePackage`), containing the information to be displayed to the user.

7. Incomplete parts of the code

You will find that one constructor and a number of methods in `Main.java`, `SuffixTree.java` and `SuffixTreeAppl.java` are incomplete. Firstly, in `Main.java`, the `main` method is incomplete, as follows:

```
public class Main {

  /**
```

```
   * The main method.
   * @param args the arguments
   */
  public static void main(String args[]) {
    String errorMessage = "Required syntax:\n";
    errorMessage += "  java Main SearchOne <filename> <query string> for
                     Task 1\n";
    errorMessage += "  java Main SearchAll <filename> <query string> for
                     Task 2\n";
    errorMessage += "  java Main LRS <filename> for Task 3\n" ;
    errorMessage += "  java Main LCS <filename1> <filename2> for Task 4";

    if (args.length < 2)
      System.out.println(errorMessage);
    else {
      // get the command from the first argument
      String command = args[0];

      switch (command) {
      case "SearchOne":
      case "SearchAll": {
        if (args.length < 3) {
          System.out.println(errorMessage);
          break;
        }
        System.out.println("Search results should be displayed here");
        break;
      }
      case "LRS": {
        System.out.println("LRS results should be displayed here");
        break;
      }
      case "LCS": {
        if (args.length < 3) {
          System.out.println(errorMessage);
          break;
        }
        System.out.println("LCS results should be displayed here");
        break;
      }
      default: System.out.println(errorMessage);
      }
    }
  }
}
```

Secondly, in `SuffixTree.java`, the second constructor is incomplete, as follows:

```
/**
 * Builds a generalised suffix tree for two given strings.
 *
 * @param sInput1 the first string
 * @param sInput2 the second string
 * - assumes that '$' and '#' do not occur as a character
 *   anywhere in sInput1 or sInput2
 * - assumes that characters of sInput1 and sInput2 occupy
 *   positions 0 onwards
 */
public SuffixTree (byte[] sInput1, byte[] sInput2) {
  // to be completed!
}
```

Thirdly, in `SuffixTreeAppl.java`, the following four methods are incomplete:

```
/**
 * Search the suffix tree t representing string s for a target x.
 * Stores -1 in Task1Info.pos if x is not a substring of s,
 * otherwise stores p in Task1Info.pos such that x occurs in s
 * starting at s[p] (p counts from 0)
 * - assumes that characters of s and x occupy positions 0 onwards
 *
 * @param x the target string to search for
 *
 * @return a Task1Info object
 */
public Task1Info searchSuffixTree(byte[] x) {

  return null; // replace with your code!
}

/**
 * Search suffix tree t representing string s for all occurrences of target
 * x. Stores in Task2Info.positions a linked list of all such occurrences.
 * Each occurrence is specified by a starting position index in s
 * (as in searchSuffixTree above).  The linked list is empty if there
 * are no occurrences of x in s.
 * - assumes that characters of s and x occupy positions 0 onwards
 *
 * @param x the target string to search for
 *
 * @return a Task2Info object
 */
public Task2Info allOccurrences(byte[] x) {
  return null; // replace with your code!
}

/**
 * Traverse suffix tree t representing string s and stores ln, p1 and
 * p2 in Task3Info.len, Task3Info.pos1 and Task3Info.pos2 respectively,
 * so that s[p1..p1+ln-1] = s[p2..p2+ln-1], with ln maximal;
 * i.e., finds two embeddings of a longest repeated substring of s
 * - assumes that characters of s occupy positions 0 onwards
 * so that p1 and p2 count from 0
 *
 * @return a Task3Info object
 */
public Task3Info traverseForLrs () {
  return null; // replace with your code!
}

/**
 * Traverse generalised suffix tree t representing strings s1 (of length
 * s1Length), and s2, and store ln, p1 and p2 in Task4Info.len,
 * Task4Info.pos1 and Task4Info.pos2 respectively, so that
 * s1[p1..p1+ln-1] = s2[p2..p2+ln-1], with len maximal;
 * i.e., finds embeddings in s1 and s2 of a longest common substring
 * of s1 and s2
 * - assumes that characters of s1 and s2 occupy positions 0 onwards
 * so that p1 and p2 count from 1
 *
 * @param s1Length the length of s1
 *
 * @return a Task4Info object
 */
```

```
public Task4Info traverseForLcs (int s1Length) {
  return null; // replace with your code!
}
```

8. The required output

Your task is to complete the bodies of the above constructor and the above methods so that, when the user executes the application, providing command-line arguments (i.e., the name(s) of text file(s) and a search string if appropriate), the output is as follows, for each task:

1. `java Main SearchOne <filename f> <query string x>` should return a starting position of `x` in the string `s` read from text file `f` if `x` occurs in `s`, or else a message stating that `x` does not occur in `s` (if `x` has multiple occurrences as a substring of `s`, then the starting position of just one of these occurrences, and not necessarily the first, should be returned);
2. `java Main SearchAll <filename f> <query string x>` should return all occurrences (given by starting positions; not necessarily in sorted order) of `x` in the string `s` read from text file `f`, together with the total number of such occurrences;
3. `java Main LRS <filename f>` should return a longest repeated substring `x` of the string `s` read from text file `f`, the length of `x` and two distinct starting positions of `x` in `s` (note: your code should handle the possibility that `s` has no repeated substring);
4. `java Main LCS <filename1 f1> <filename2 f2>` should return a longest common substring `x` of strings `s1` and `s2` read from text files `f1` and `f2`, the length of `x`, a starting position of `x` in `s1` and a starting position of `x` in `s2` (note: your code should handle the possibility that `s1` and `s2` have no common substring).

Note that you should complete this exercise without altering any of the instance variables, methods or constructors in the supplied code apart from those explicitly shown in Section 7 above (with the exception that you may find it helpful to add instance variables and methods to `SuffixTreeAppl.java`, though it should not be necessary to add inner classes here). A small penalty may be applied if you do alter other parts of the supplied code.

9. Example output

Your output should *exactly* match the following example sequence when executed on the same commands (except that any starting positions output by Task 2 need not be in the same order as in the example below):

```
> java Main SearchOne text1.txt "there"
Search string "there" occurs at position 1132 of text1.txt

> java Main SearchOne text1.txt "clutch"
Search string "clutch" not found in text1.txt

> java Main SearchAll text2.txt "clutch"
The string "clutch" occurs in text2.txt at positions:
178007
77871
479220
The total number of occurrences is 3

> java Main SearchAll text1.txt "clutch"
The string "clutch" does not occur in text1.txt

> java Main LRS text1.txt
An LRS in text1.txt is "
```

```
ECONOMIC EVOLUTIONS.--FIRST PERIOD.--THE DIVISION OF LABOR"
Its length is 59
Starting position of one occurrence is 538
Starting position of another occurrence is 235301

> java Main LRS text2.txt
An LRS in text2.txt is ".
  "Mamma! What sweets are we going to have?" "
Its length is 47
Starting position of one occurrence is 155839
Starting position of another occurrence is 156193

> java Main LCS text1.txt text2.txt
An LCS of text1.txt and text2.txt is " it is absolutely necessary t"
Its length is 29
Starting position in text1.txt is 212557
Starting position in text2.txt is 120985
```

You may find it useful to test your program using the two sample text files `text1.txt` and `text2.txt` that were provided in the zip file. Additionally, after submission, your program will be run against unseen acceptance tests. **It is very important to ensure that your output is exactly consistent with the above example**, as automated testing will be used with the acceptance tests.

10. What to submit

Your submission to this exercise should take the form of a zip/tar.gz file containing the following files:

1. A document (written using e.g., Word or LaTeX) containing:
   - 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), and how these might be fixed if you had more time. If there are no compile-time or run-time errors, all that is required is one line stating this! You should also list here any external sources, including AI tools, that you have used in order to complete the exercise, and what parts of the exercise the use of these sources or tools relates to.
   - A detailed implementation report justifying the operation of each of your methods corresponding to Tasks 1-4 above, making sure to explain how you manipulate the suffix tree data structure in each case. At most half a page of text should be sufficient for each task.

2. A pdf file containing a formatted listing of `Main.java`, `SuffixTree.java` and `SuffixTreeAppl.java`. Ensure that all lines are readable. These are the only classes that should be modified. However if you did find it necessary to modify other classes, you should ensure that you include a listing of these classes, and highlight exactly what additional modifications have been made in your implementation report. In order to produce this pdf file, the preferred option is to use the following Unix commands (e.g., on the server `stlinux01`), but this is *not* obligatory:

   ```
   a2ps -A fill -Ma4 --columns=1 --font-size=9 filename1.java
   filename2.java (etc) -o code-listing.ps
   ps2pdf -sPAPERSIZE=a4 code-listing.ps
   ```

3. Your program code. Be sure to submit *all* of your classes, and not just the ones that you have altered. Also, ensure that you remove any debugging code that may generate large volumes of output on large input files.

The submission mechanism for this exercise is entirely electronic and no hard-copy submission is required.

11. How to submit

In order to make your submission, click on the submission link under the Assessment section of the course web page on Moodle (https://moodle.gla.ac.uk/course/view.php?id=49793). You will need to submit a zip file or tar.gz file containing your files listed in Section 10, which should be named `AlgII_<family name>_<given name>.zip` or `AlgII_<family name>_<given name>.tar.gz`, e.g., `AlgII_Manlove_David.zip`.

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 file.

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 status report.

12. Marking scheme

The assessed exercise will be marked according to the following breakdown of numerical marks (with the numerical mark then being converted to a band):

Implementation of Tasks 1-4 listed in Section 4 above: 3, 2, 3 and 4 marks respectively (awarded on the basis of correctness and efficiency):	(12)
Implementation of the method for building a generalised suffix tree:	(1)
Completion of `main` method:	(3)
Implementation report: 3 marks corresponding to each of Tasks 1-4: 3, 2, 3 and 4 marks respectively:	(12)
Quality of code (i.e. layout, comments etc.) and general presentation:	(2)
Total:	(30)

13. Miscellaneous final remarks

- Various methods in the supplied code assume that the given text file(s) and the given search string (if appropriate) are represented as an array of type `byte` – this is to save space, since a variable of type `char` uses two bytes in Java. You may find the method `getBytes()` in the `String` class helpful for converting a string into an array of type `byte`.

- It is vital to bear in mind that all of the methods which take as input an array of type `byte` assume that the relevant information is stored from **position 0** onwards, as is conventional in Java. All pointers into text files and search strings should therefore count from 0.

- Beware of creating a text file in a Windows or Mac environment and then accessing the file in Unix. In a Windows or Mac text file, a carriage return may appear as two characters (`\r`, `\n`; ASCII values 13 and 10 respectively), whilst in a Unix text file a carriage return appears as one

character (`\n`). The existence of`\r` characters is highlighted when using the command `od -c filename` in Unix. You may find it useful to use the command `dos2unix filename` in Unix to remove any `\r` characters if necessary.

- It is not necessary to add error checking routines into the `main` method to check for incorrect syntax when calling the application from the command line (e.g., forgetting to include the correct number of arguments etc). Some basic error checking has been provided in the setup code, but this need not be extended.

- To try to discourage people from spending too long on this exercise, it's worth noting that each of the methods corresponding to Tasks 1-3 can be completed using at most 60 lines of code (including comments), with 80 lines in the case of Task 4. This is, of course, just a guideline, but a small penalty may be applied to *excessively* complicated solutions to any of the tasks.