Mandatory Assignment 1, IN3130, 2022

Deadline: Tuesday October 4 @ 23:59

Comments concerning this mandatory assignment:

- This assignment consists of three exercises, and each student must turn in his/her own, independent solution to all exercises.
- Exercises 1 and 2 require code. Your programs must be written in Java. Variable and function names should be self-explanatory; comment as you see fit. Please make sure your code compiles at the department computers. Some datasets for testing your programs are on the course site.
- Exercise 3 requires a textual answer, including some discussion. It is equally important to give good answers to this exercise as it is to program well!
- In the event of minor errors or ambiguities in the assignment text, you are generally expected to state your reasonable assumptions in the PDF or commented in the source code as to the intended meaning in your answer.
- We might give further comments or corrections to this assignment. If so, they will appear as messages on the course page. Check regularly.
- For 3-day extension, submit your request in the nettskjema at the course page.

Exercise 1 (Tries)

Write a program that builds an uncompressed trie (prefix-tree) from a collection of strings, and then searches this trie for a number of given strings. For simplicity we assume the strings consist only of lower-case letters. Repeated insertions of already existing strings are accepted. You are free to choose how to implement access from a node to the relevant subnode, and how new nodes are inserted when necessary.

Input

The program is to read its input from a designated file. The filename is given to the program, together with the name of an output file, as command line arguments. The input file contains an integer *N* and a set of strings (words) separated by newlines. The first *N* strings are to be inserted into the trie, the remaining strings are to be looked up in the finished data structure.

Output

Output is written to the file with name given as the second argument to the program. For each of the first *N* strings: The string itself followed by a colon, and the data structure as it looks after insertion of the string is to be written. Output the data structure should be given in prefix format with parentheses around each subtree, but not around single letters in long paths. List the subtrees in alphabetical order.

If the first three strings we insert into our tree are “internet”, “interview”, and “inter”, the output should look like this:

```
internet: (internet)
interview: (inter(net)(view))
inter: (inter(net)(view))
```

Likewise, if the first three strings we insert are “inter” “inter” “internet”, the output should look like this:

```
inter: (inter)
inter: (inter)
internet: (inter(net))
```

A prefix representation of the trie with the following words “internet”, “interview”, “internally”, “algorithm”, “all”, “web” and “world” looks like this:

```
(al(gorithm)(l))(inter(n(ally)(et))(view))(w(eb)(orld))
```

For the remaining strings (after the first *N* ones), the string itself, and the result of the lookup in the data structure built with the first *N* strings is to be written, like this:

```
internet YES
interney NO
```

Note that a string that *only* exits as a prefix, not as separately inserted word, like “al” in the example trie above should give a negative answer.

Implementation tips

When reading from and writing to files, the classes `java.util.Scanner` and `java.io.PrintWriter` can be used. In the following example an interger is read from an input file, and the string `<input-number> * 2 = <number times two>` is written to an output file:

```
public static void main(String[] args) throws FileNotFoundException
{
   Scanner in = new Scanner(new File(args[0]));
   int input;
   try {
      input = in.nextInt();
      // see documentation for java.util.Scanner for more examples
   } finally {
      in.close();
   }

   int output = input * 2;


PrintWriter out = new PrintWriter(new File(args[1]));
   try {
      out.printf("%d * 2 = %d\n", input, output);
      // or out.println
   } finally {
      out.close();
   }
}
```

It will be helpful to add an attribute in nodes with information that a word ends. Another possibility can be to add a child node with some character with signals end of a word.

Test data is easy to produce, but there will also be a link on the course web page to a file containing a test set.

Exercise 2 (Dynamic Programming)

The problem "Sum of Selections" (SOS) is defined as follows: Given a sequence of n positive integers $t_1, t_2, \ldots, t_n$ and a positive integer K, we are asked to make a selection of the t_i's with sum exactly K (or determine that no such selection exists).

Example: For $n = 5$, let the integers be 2, 5, 6, 3, 10 and $K = 8$. For this instance of the problem we can select 2 and 6, which sums to 8. If we only want a YES/NO answer, the answer is YES for this instance. In several of the questions below we are, however, interested in the selection itself, not merely a YES or NO.

The exercise is divided into multiple parts to illustrate dynamic programming, with and without memoization, to show the differences between the two methods, and how they are both based on the same recursive formula.

a) During execution the sum of different selections can assume values in the interval [0, S] where S is the sum of all the t_i's**.**To find out, using dynamic programming, whether a certain instance is solvable, you can use a boolean array with dimension $[n] \times [S]$. Explain what you want the values of this table to mean, give the recursive formula you base your algorithm on, and explain briefly why it is correct. Submit these three as comments in the source code.

b) Implement a standard bottom-up dynamic programming algorithm that solves SOS. Your program should answer YES or NO depending on whether a selection with the correct sum is possible or not, and for YES-instances also output the selection itself (see output example below).

c) Implement a recursive, memoized algorithm (top-down) using the formula from a) above to solve SOS. Use an array with dimensions as described above in a). Output should be as described for b).

Input

The program is to read its input from a designated file. The filename is given to the program, together with the name of an output file, as command line arguments. The input file contains a series of instances, one on each line; each instance consists of the number n, then the number K, a and finally the t_i's (all integers, separated by blanks).

An input file with two instances can look like this:

```
5 14 1 5 6 3 10
5 4 2 5 6 3 10
```

Output

For each instance in the input file the output should contain:

- A line starting with: “INSTANCE”, and then the numbers: n, K, t_1, …, t_n, divided by single spaces, but with a colon (and a blank) after K.
- A new line with either “YES” or “NO” depending on whether a selection of sum *K* was possible to make or not
- If the answer is “YES”, then yet another line indicating the selected number in the order of growing index i, separated by single blanks. Each selected number should be given as: t_i[i] (see example below).
- followed by the selection as a sequence of (<t_i>,<0 or 1, depending on whether t_i is in the selection or not>).

All of this is to be written to file given as the second argument to the program. Only one solution is to be written for each instance. If more than one solution is possible, choose one.

Output for the instances in the input example above should be:

```
INSTANCE 5 14: 1 5 6 3 10
YES
SELECTION 1[1] 3[4] 10[5]

INSTANCE 5 4: 2 5 6 3 10
NO
```

Implementation tips

See Exercise 1 above for I/O-handling in Java.

To avoid passing all data needed by the recursive method as parameters to every call, you should store much of it as attributes in a suitable object, and make the method local to the class of that object. The array containing the values of solutions to sub-problems should be one such attribute. This object is easily set up by a main method that also starts the recursion.

Test data should be easy to write for this exercise too, but there is a link to a file containing a simple set on the course web page.

Exercise 3 (Undecidability vs decidability)

Decide whether the following problems (formal languages) are decidable or not. If a language is decidable, describe an algorithm for deciding it. If a language is undecidable, prove this by showing a reduction from the halting problem.

1. $L_1 = \{w \in \{0 \mid 1\}^* \mid w$ is a sequence of one or more 1's $\}$
   (corresponding to the problem of checking if the input is a sequence of one or more 1's)
   If L_1 is decidable, give the full Turing machine definition including transition table.
2. $L_2 = \{<M> \mid$ Turing machine M decides $L_1\}$
3. $L_3 = \{<M> \mid$ Turing machine M decides $L_2\}$

Handing it all in

Hand in source code plus *one* PDF-document.

- Put the source code and test files with Exercise 1 and 2 in separate folders and a document called oblig1-<username>.pdf in a folder called: `oblig1_<your username>.`

- Compress the folder into a zip file with the same name.

- Hand in the compressed file at: `https://devilry.ifi.uio.no`

[end]