Monash University
Faculty of Information Technology
2nd Semester 2022

FIT2014
Assignment 1
Linux tools, regular expressions, induction
DUE: 11:55pm, Friday 19 August 2022 (Week 4)

How to manage this assignment

- Start work on this assignment early, and do as much as possible of it *before* your week 4 prac class. There will not be time during the class itself to do the assignment from scratch; there will only be time to get some help and clarification.

Instructions

Please read these instructions carefully **before** you attempt the assessment:

- To begin working on the assignment, download the workbench `asgn1.zip` from Moodle. Create a new Ed Workspace and upload this file, letting Ed automatically extract it. Edit the `student-id` file to contain your name and student ID. Refer to Lab 0 for a reminder on how to do these tasks.
- The workbench provides locations and names for all solution files. These will be empty, needing replacement. Do **not** add or remove files from the workbench.
- Solutions to written questions must be submitted as PDF documents. You can create a PDF file by scanning your **legible** (use a pen, write carefully, etc.) hand-written solutions, or by directly typing up your solutions on a computer. If you type your solutions, be sure to create a PDF file. There will be a penalty if you submit any other file format (such as a Word document). Refer to Lab 0 for a reminder how to upload your PDF to the Ed workspace and replace the placeholder that was supplied with the workbench.
- Before you attempt any problem—or seek help on how to do it—be sure to read and understand the question, as well as any accompanying code.
- When you have finished your work, download the Ed workspace as a zip file by clicking on "Download All" in the file manager panel. **You must submit this zip file to Moodle by the deadline given above.** To aid the marking process, you must adhere to **all** naming conventions that appear in the assignment materials, including files, directories, code, and mathematics. Not doing so will cause your submission to incur a one-day late-penalty (in addition to any other late-penalties you might have). Be sure to check your work carefully.

Your submission must include:

- a `sed` script, `decomposeSyllablesIntoParts`, for Problem 1(a);
- a one-line text file, `Korean-NameStructure2`, for Problem 1(a);
- a `sed` script, `decomposePartsIntoLetters`, for Problem 1(b);
- a one-line text file, `Korean-NameStructure3`, for Problem 1(b);
- an `awk` script, `matchKorean`, for Problem 1(c);
- the output file `outputFile` you produced by running your `awk` script on the provided input file, `inputFileOfNames`, in Problem 1(c);
- a file `prob2.pdf` with your solution to Problem 2.

Introduction to the Assignment

In Lab 0, you met the stream editor `sed`, which detects and replaces certain types of *patterns* in text, processing one line at a time. These patterns are actually specified by *regular expressions*. You will use `sed` again in Problem 1 of this Assignment, to help construct regular expressions.

You will also learn about `awk`, which is a simple programming language that is widely used in Unix/Linux systems and which also uses regular expressions. In Problem 1, you will construct an `awk` program to identify a class of Korean names.

Finally, Problem 2 is about applying induction to a problem of counting on graphs.

Introduction to `awk`

In an `awk` program, each line has the form

/*pattern*/ { *action* }

where the *pattern* is a regular expression (or certain other special patterns) and the *action* is an instruction that specifies what to do with any line that contains a match for the *pattern*. The *action* (and the {...} around it) can be omitted, in which case any line that matches the *pattern* is printed.

Once you have written your program, it does not need to be compiled. It can be executed directly, by using the `awk` command in Linux:

```
$ awk -f programName   inputFileName
```

Your program is then executed on an input file in the following way.

```
//   Initially, we're at the start of the input file, and haven't read any of it yet.
If the program has a line with the special pattern BEGIN, then
    do the action specified for this pattern.
Main loop, going through the input file:
{
    inputLine := next line of input file
    Go to the start of the program.
    Inner loop, going through the program:
    {
        programLine := next line of program (but ignore any BEGIN and END lines)
        if inputLine contains a string that matches the pattern in programLine, then
            if there is an action specified in the programLine, then
            {
                do this action
            }
            else
                just print inputLine        //   it goes to standard output
    }
}
If the program has a line with the special pattern END, then
    do the action specified for this pattern.
```

Any output is sent to standard output.

You should read about the basics of `awk`, including the way it represents regular expressions and the main instruction types used in its actions. Any of the following sources should be a reasonable place to start:

- A. V. Aho, B. W. Kernighan and P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, New York, 1988.
  (The first few sections of Chapter 1 should have most of what you need, but be aware also of the regular expression specification on p28.)

- `https://www.grymoire.com/Unix/Awk.html`
- the Wikipedia article looks ok
- the `awk` manpage
- the GNU Awk User's Guide.

Introduction to Problem 1

> The Master said, 'What is necessary is to rectify names .... If names are not rectified, then words are not appropriate. If words are not appropriate, then deeds are not accomplished.'
>
> – Confucius (孔夫子), *The Analects* (transl. R. Dawson), Oxford University Press, 1993, §13.3.

Most organisations today deal with people from many different cultures and language groups, and they must often record and process people's names in systems that work mainly with English language text. In such contexts, it is helpful to be able to recognise names from different language groups. Example applications include: determining how to pronounce students' names when reading them out from a list at graduation ceremonies; determining how to greet a person with whom you have an appointment; determining how to enter the various parts of a person's name into a database; determining how automatically-generated emails, sent to many different people listed in some file, should address each recipient; determining the most likely native language of a person in situations where their name is known but they cannot be spoken to directly at the time (e.g., in some emergency situations). Recognising the language group that a name belongs to is an important first step in all these situations.

In this problem you will write some code in `sed` and `awk` to try to recognise one type of names in a long file of Asian names. More specifically, suppose you are given an input file in which each line starts with a person's name in some language, with each name transcribed somehow into English text. Your task is to detect which of these names come from *Korean*. [1]

In the input file, all text from the start of each line until the first colon (`:`) on the line (but not including the colon itself) is taken to be a person's name. In most cases, each line ends with a string of non-blank letters specifying which language the name is believed to come from. An example input file is provided, as *inputFileOfNames*. If you browse through the file, you will notice that it contains names from a variety of Asian languages: Mandarin, Cantonese, Hokkien, Teochew, Hakka, Korean, Japanese, Thai, Vietnamese, Malay and Indonesian. They have been represented in English text using a variety of transcription schemes, and with all extra marks on letters (accents, tone marks, other diacritical marks, etc.) removed. [2] In many cases, the line about a person also contains some other information about them, but our name recognition task will ignore that information.[3]

Further information about working with names from different cultures can be found in:

- Fiona Swee-Lin Price, *Success with Asian Names*, Allen & Unwin, Crows Nest, NSW, 2007.
- J. Greenberg Motamedi, Z. Jaffery, A. Hagen, and S. Y. Yoon, *Getting it right: Reference guides for registering students with non-English names, 2nd edition. (REL 2016-158 v2)*, U.S. Department of Education, Institute of Education Sciences, National Center for Education Evaluation and Regional Assistance, Regional Educational Laboratory Northwest, Washington, DC, 2017. `https://ies.ed.gov/ncee/edlabs/regions/northwest/pdf/REL_2016158.pdf`
- SBS, *The Cultural Atlas*, `https://culturalatlas.sbs.com.au/`. You can look up a country or culture (e.g., by clicking on a map) and then click on a link to "Naming" for that culture.

[1] There are a few romanisation systems for transcribing Korean into English.

[2] These marks carry important information about meaning and pronunciation. But they are often removed when names are represented using other alphabets.

[3] The file was compiled from a number of sources, mainly Wikipedia lists of names of type `https://en.wikipedia.org/wiki/List_of_`*CultureName*`_people`, where *CultureName* is one of the cultures listed above; also `https://www.goratings.org/en/`. The lists obtained from Wikipedia are rather imperfect, with people's names often not written in a form that clearly shows the claimed cultural background.

Names in Korean

Korean names consist of a family name followed by a given name (the reverse of the usual order in English). The family name usually consists of just a single syllable, and the given name usually consists of two syllables. We restrict ourselves to names with these numbers of syllables from now on.[4]

We treat a Korean name as consisting of a syllable, followed by a space, followed by two consecutive syllables which may have a hyphen between them.

Each syllable consists of an optional *prefix*, a compulsory *middle* and an optional *suffix*.[5]

- The possibilities for the *prefix* are:

  `g`, `kk`, `n`, `d`, `tt`, `r`, `m`, `b`, `pp`, `s`, `ss`, `j`, `jj`, `ch`, `tch`, `k`, `t`, `p`, `h`

  These are also listed, one per line, in the file `prefix-file`.

- The possibilities for the *middle* are:

  `a`, `ae`, `ya`, `yae`, `eo`, `e`, `yeo`, `ye`, `o`, `wa`, `wae`, `oe`, `yo`, `u`, `wo`, `we`, `wi`, `yu`, `eu`, `ui`, `i`, `oo`, `ah`

  These are also listed, one per line, in the file `middle-file`.

- The possibilities for the *suffix* are:

  `k`, `n`, `t`, `l`, `m`, `p`, `ng`

  These are also listed, one per line, in the file `suffix-file`.

For example, the syllable `kyung` is formed from the prefix `k` followed by the middle `yu` and then the suffix `ng`.

Constructing a regular expression for Korean names

We will use this information, together with `sed`, to generate a regular expression to try to match Korean names. (The match will be quite imperfect, and later you will do some simple computations to get some idea of how well, or poorly, it does.[6]) Throughout, we will only do case-insensitive matching, so there is no need to capitalise initial letters of syllables in order to match names.

We describe the steps in detail now, and then list the assignment submission requirements for Problem 1.

We start with the file `Korean-NameStructure0`, which just contains one line consisting of the text `<NAME>`. Think of this as a special symbol, eventually to be transformed to a regular expression for Korean names.

We first transform `<NAME>` to a regular expression representing the *syllable structure* of Korean names. From our description of the syllable structure above, we can represent this by the regular expression `<SYLLABLE> <SYLLABLE>-?<SYLLABLE>` where `<SYLLABLE>` is a special symbol to represent any Korean syllable and putting "`?`" after a symbol means it can occur just once or not at all at that position.[7] Each occurrence of `<SYLLABLE>` will eventually be transformed to a regular expression to represent such syllables. The first step of this transformation can be done using the

[4]There are exceptions, though they are uncommon: some family names have two syllables, and some given names have one syllable or more than two syllables. But in this assignment we consider only one-syllable family names and two-syllable given names.

[5]This terminology is different to the more usual linguistic terminology of *initials* and *finals*. This is deliberate, since they represent slightly different sets of strings here to the standard initials and finals. This change was just a small simplification for the purposes of the assignment.

[6]Some reasons for the imperfect matching include: not all the combinations of prefixes and suffixes listed above give valid syllables in Korean; some people write Korean names without any hyphen or with a space between the two syllables of the given name; Korean names are sometimes represented using other schemes; and some family or given names may have different numbers of syllables than those we have allowed for.

[7]The operator "`?`" is a standard feature of regular expression syntax in `sed` and `awk`. Question to consider (but not part of the assignment): does using it enlarge the class of languages that can be represented by regular expressions?

following `sed` command:

```
$  sed "s/<NAME>/<SYLLABLE> <SYLLABLE>-?<SYLLABLE>/" Korean-NameStructure0 > Korean-NameStructure1
```

Alternatively, you can execute the `sed` instruction "`s/<NAME>/<SYLLABLE> <SYLLABLE>-?<SYLLABLE>/`" from the the file `decomposeNameIntoSyllables` (provided with this assignment) by doing the following:

```
$  sed -f decomposeNameIntoSyllables Korean-NameStructure0
> Korean-NameStructure1
```

Try these, using the given files, and check that you get the required result in the file `Korean-NameStructure1`. This use of `sed` is a template for what you will do next.

Now you need to replace each `<SYLLABLE>` by a regular expression representing possible syllable structures, using `<PREFIX>`, `<MIDDLE>` and `<SUFFIX>`.

Write the `sed` instruction to do this, in the file `decomposeSyllablesIntoParts`, and do the transformation from your Linux command line using

```
$  sed -f decomposeSyllablesIntoParts Korean-NameStructure1
> Korean-NameStructure2
```

Now you need to replace each occurrence of each part, `<PREFIX>`, `<MIDDLE>` and `<SUFFIX>`, by regular expressions representing possible letter strings, according to the sets of strings given on the previous page.

Write the `sed` instructions to do this, in the file `decomposePartsIntoLetters`. You will need three lines: one line giving the `sed` instruction for each part type, `<PREFIX>`, `<MIDDLE>` and `<SUFFIX>`. Then do the transformation from your Linux command line using

```
$  sed -f decomposePartsIntoLetters Korean-NameStructure2
> Korean-NameStructure3
```

This should put, into the file `Korean-NameStructure3`, a regular expression that is intended to match Korean names.

Matching names in the input file

We will now use this regular expression to construct an `awk` program to apply it to names in the input file, and to do some simple computations to determine how well the regular expression matches Korean names.

For each name in the input file, checking it against your regular expression gives one of four possible outcomes:

- **Correct Match:** a name that belongs to Korean (according to the last word on its line in *inputFileOfNames*) is also matched by your regular expression.
- **False Positive:** a name that does *not* belong to Korean (according to the last word on its line in *inputFileOfNames*), but is matched by your regular expression.
- **False Negative:** a name that belongs to Korean (according to the last word on its line in *inputFileOfNames*), but is *not* matched by your regular expression.
- **Correct Non-match:** a name that does *not* belong to Korean (according to the last word on its line in *inputFileOfNames*) and is *not* matched by your regular expression.

Your awk program should compute the number of names, in the input file, of each of these four types, and report those numbers at the end.

You can write the `awk` program manually, cutting-and-pasting the regular expression as needed (and making any necessary modifications to it) from `Korean-NameStructure3`.

The heart of your `awk` program will be a statement that does the following.

- The *pattern* matches a Korean name whenever it occurs in the specified position at the start of a line.
  - Recall the general structure of an `awk` statement. By default, matches in `awk` are case-sensitive. To do a case-*in*sensitive match, you need to match the pattern against a version of the current line in which all alphabetic characters have been converted to lower case. This can be done using a statement of this form:

    `tolower($0) ~ /`*pattern*`/` { *action* }

    Here, `$0` is a built-in `awk` variable that always contains the current line of the input file, and `tolower()` is a function that converts all alphabetic characters to lower case. You can read " ~ " as "matches".
- The *action* increments appropriate variables in order to update the number of correct matches, false positives, false negatives, or correct non-matches, as appropriate.

You may need another line or few too, including an END line, at the end, in order to print the total numbers of matches of each type.

The files and commands required to complete this problem are shown in Figure 1.

Problem 1. [12 marks]

First, run `sed`, as described, using the one-line `sed` script `decomposeNameIntoSyllables` to produce `Korean-NameStructure1`.

(a) Write the one-line `sed` script, `decomposeSyllablesIntoParts`, and run `sed`, as described, to produce `Korean-NameStructure2`.

(b) Write the three-line `sed` script, `decomposePartsIntoLetters`, and run `sed`, as described, to produce `Korean-NameStructure3`.

(c) Write the `awk` script, `matchKorean`, and run it, as described, on input file `inputFileOfNames`, to produce `outputFile`.

For **bonus marks**:

NOTE: This is a more advanced and challenging exercise, requiring exploration, experimentation, and research. To attempt it, you must have mastered the rest of this question first. Bonus marks are only available when the total mark for the rest of Problem 1 is at least 10 out of 12.

Can you write a *much better* regular expression for Korean names? If you attempt this, then, in addition to the files required above, you should also provide:

- your new regular expression, in a one-line file called `Korean-NameStructure4`;
- your new `awk` program, in a file called `matchKoreanBonus`;
- the output from running your `awk` program on the input file, as `outputFileBonus`.

To qualify for bonus marks, your approach must use a *significantly different* approach to the one used above, *and* offer a significant advantage. If your approach is more complex, then it should give better results (i.e., lower error rate) in general. If your approach is significantly *simpler* but does almost as well, then that also may qualify for bonus marks.

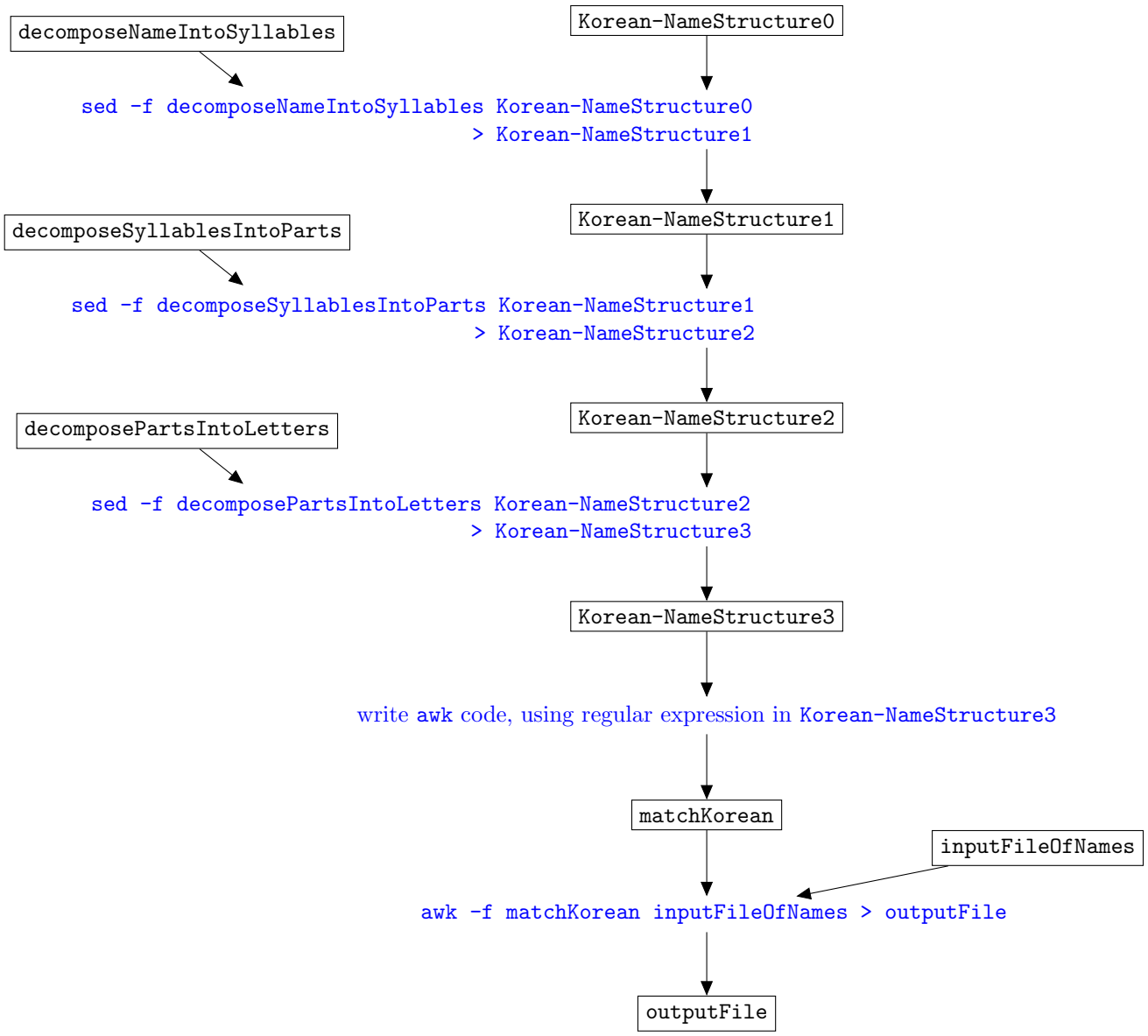


Figure 1: The plan.

Your solution will not just be evaluated on how well it does on the provided input file. It will be run on other files of names too. Also, do not base your regular expression solely on the specific details of the Korean names provided in the input file. They may not be totally representative of the full range of names from that culture.

As usual, be sure to cite any sources used.

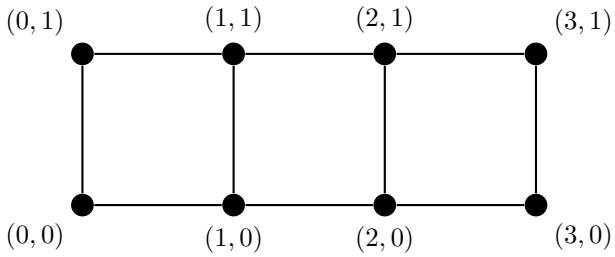
Problem 2. [8 marks]

An *independent set* in a graph is a set of mutually non-adjacent vertices in the graph. So, no edge can have both its endpoints in an independent set.

In this problem, we will count independent sets in ladder graphs. A *ladder graph* L_n is a graph obtained from the points and lines formed by a row of n squares. More formally:

- its vertices correspond to ordered pairs $(0,0), (0,1), (1,0), (1,1), \ldots, (n,0), (n,1)$;
- two vertices are adjacent if they represent points at distance exactly 1 from each other.

The number n here is called the *order* of the ladder. Here is L_3, a ladder of order 3:



Examples of independent sets in L_3 include: $\emptyset$, $\{(2,1)\}$, $\{(0,0),(2,0),(3,1)\}$, $\{(0,1),(1,0),(2,1),(3,0)\}$.

The simplest ladder is L_0, consisting of just the vertices $(0,0)$ and $(0,1)$ with a single edge between them.

For all n, define

$$\begin{aligned} a_n &= \text{number of independent sets in } L_n \text{ which include neither } (n,0) \text{ nor } (n,1); \\ b_n &= \text{number of independent sets in } L_n \text{ which include } (n,0) \text{ but not } (n,1); \\ c_n &= \text{total number of independent sets in } L_n. \end{aligned}$$

(a) Compute a_0, b_0, a_1, b_1.

(b) Give, with justification, recurrence relations that express a_{n+1} and b_{n+1} in terms of a_n and b_n. (Here, *each* of a_{n+1} and b_{n+1} is expressed using *both* a_n and b_n.)

(c) Prove, by induction on n, that for all $n \geq 0$,

$$\begin{aligned} a_n &\leq \sqrt{2}(\sqrt{2}+1)^n \quad \underline{\text{and}} \\ b_n &\leq (\sqrt{2}+1)^n. \end{aligned}$$

(d) Hence give a good closed-form upper bound for c_n (i.e., an upper-bound expression in terms of n, *not* a recurrence relation). Your bound should be significantly better than 3^n (which is the upper bound you get by ignoring horizontal edges and only taking vertical edges into account).

For **bonus marks**:

NOTE: This is a more advanced and challenging exercise. To attempt it, you must have mastered the rest of this question first. Bonus marks are only available when the total mark for the rest of Problem 2 is at least 7 out of 8.

Find, and prove using similar techniques including induction, good *lower* bounds for a_n, b_n and c_n.