CISC/CMPE-365
Fall 2019

Week 10 AND Week 11 Lab Problem: Settling Our Differences

This is Assignment 4

Due Date: November 23, 11:59 PM

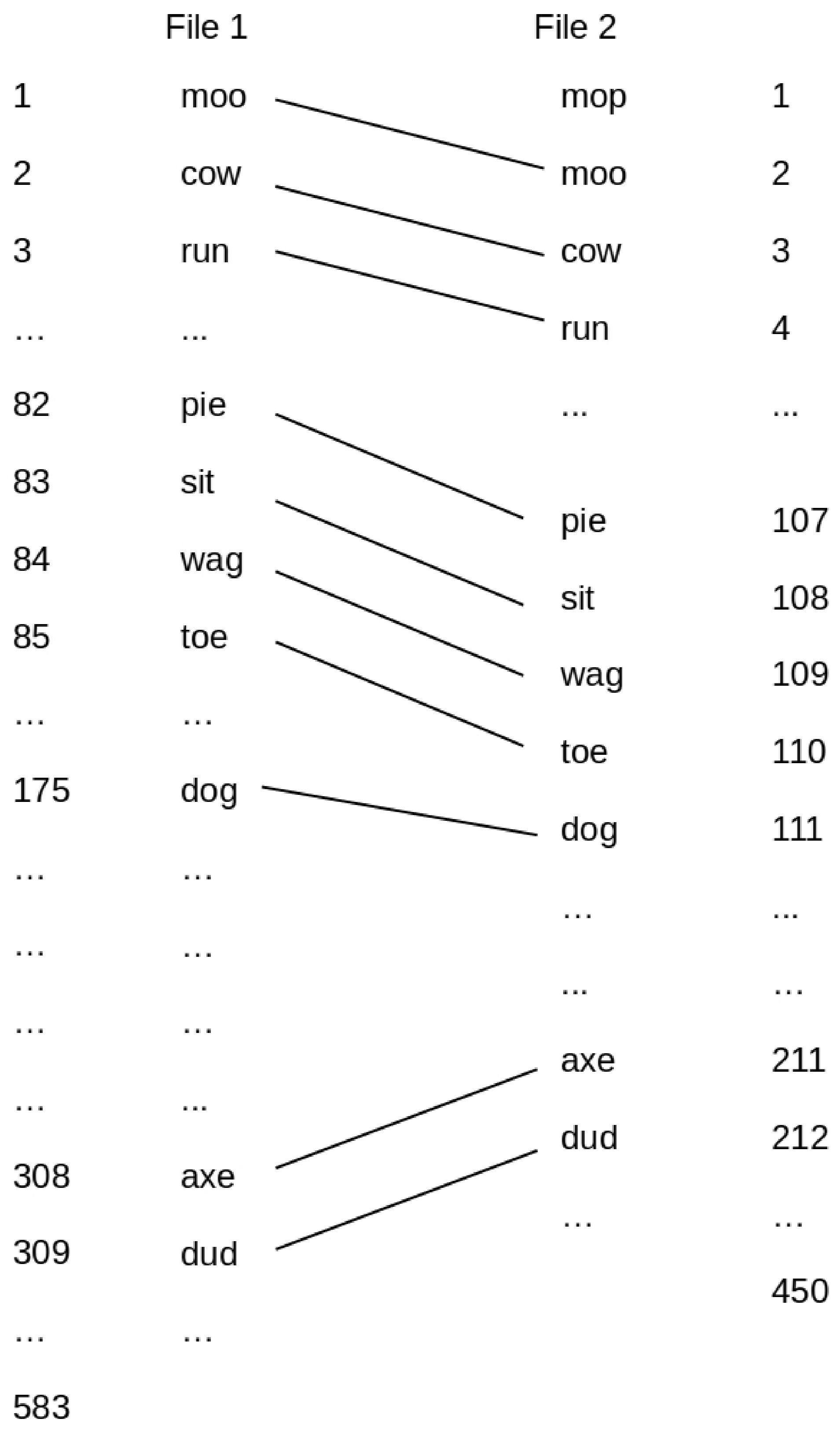
This 2-week lab deals with the problem of finding the differences between two files. UNIX and Linux systems include an utility called "diff" which does this. In this assignment you will create your own implementation using the Longest Common Subsequence algorithm that we discussed in class.

Here are the specifications:

Your program should take two text files as input. The files can be hard-coded into your program, but it would be better to have the file names as command line parameters or prompt the user to enter the file names during execution.

We will refer to the first file as "File 1" and the second file as "File 2", but your program should use the actual file names.

Treating the lines of the files as tokens, use the Dynamic Programming LCS algorithm to find the longest common subsequence of lines for File 1 and File 2. This will result in a matching up of lines that we can visualize like the figure on the next page.. Note that I have added line numbers starting at 1 for both files.


File 1
File 2
1 moo
2 cow
3 run
…
82 pie
83 sit
84 wag
85 toe
…
175 dog
…
…
…
…
308 axe
309 dud
…
583
mop 1
moo 2
cow 3
run 4
...
pie 107
sit 108
wag 109
toe 110
dog 111
… ...
... …
axe 211
dud 212
… …
450

Your program must report the differences between the two files in the following format. Note that this is different from the output of diff, which is designed to form the input to another UNIX/Linux utility called "patch".

Blocks of lines (from one line up to many) that form part of the LCS must be reported as

```
Match:      File 1 : <a .. b>                File 2 : <c .. d>
```

Blocks of lines (from one line up to many) that come between parts of the LCS must be reported as

```
Mismatch:   File 1 : <a .. b>           File 2 : <c .. d>
```

Note that in the case of a mismatch, either <a .. b> or <c .. d> could be "None"

For the two files shown above, the output would be

```
Mismatch:   File 1 :  None                  File 2 : <1 .. 1>

Match:      File 1 : <1 .. 3>               File 2 : < 2 .. 4>

Mismatch:   File 1 : <4 .. 81>              File 2 : <5 .. 106>

Match:      File 1 : <82 .. 85>             File 2 : <107 .. 110>

Mismatch:   File 1 : <86 .. 174>            File 2 : None

Match:      File 1 : <175 .. 175>           File 2 : <111 .. 111>

Mismatch:   File 1 : <176 .. 307>           File 2 : <112 .. 210>

Match:      File 1 : <308 .. 309>           File 2 : <211 .. 212>

Mismatch:   File 1 : <310 .. 583>           File 2 : <213 .. 450>
```

Note that your output should give the actual file names, not "File 1" and "File 2"

If you approach this by actually comparing the lines of the two files your program will be much slower than it needs to be … but fortunately in Lab 9 you experimented with the method of comparing strings quickly by representing each string by an integer. For Assignment 4, you must incorporate this method.

Remember that unless you have found a perfect method for representing strings by integers there is always a possibility that two different lines will be represented by the same integer (or set of integers, if you have adopted one of the methods described in Lab 9). For this reason, if two lines from the files are being compared and their integer representatives are equal, the lines then need to be compared in the traditional way ("`line1 == line2`" or equivalent) to see if they really do match. See Lab 9 for more details.

Deliverables:

- All of your source code, appropriately documented.
- An explanation and justification of the method you use to represent strings by integers.
- The output from applying your program to these pairs of files:
    - Dijkstra.py and Dijkstra.py3.py
    - Three_Bears.v1.txt and Three_Bears.v2.txt

Optional Problem to Consider:

This is NOT part of the requirements of this assignment, but if you have some spare time, it's interesting.

If the files being compared are large, filling in the Dynamic Programming table may take more time than we are willing to wait. We can accelerate the solution by deleting all lines that are in one file but not the other (they cannot be in the LCS). This has two things working against it:

- we need to somehow keep track of the original line numbers so that our output is correct
- if the two files are closely related (eg. different versions of a program) then the probability is high that most of the lines will be in both files.

An alternative is to work top-down instead of bottom-up. Using the notation that LCSL(i,j) represents the length of the longest common subsequence of File_1[1.. i] and File_2[1..j], we know the problem we need to solve is LCSL(n,m) where n and m are the line counts of the two files.

If File_1[n] and File_2[m] match then LCSL(n,m) = 1 + LCSL(n-1,m-1) … we don't need to work out any other values in the last row or column of the table. And if the two files are closely related, the probability is high that their last lines ***will*** match. In fact, the probability that each subproblem LCSL(i,j) we encounter satisfies LCSL(i,j) = 1 + LCSL(i-1,j-1) is high.

So if we work from the top down and only solve the subproblems that we need to, we may be able to leave most of the table empty. It may even be better to abandon the table completely and store the subproblem results in a hash-table as we go. We can write this version completely recursively but if n and m are large we may exceed the maximum permitted recursion depth. It would be better to use a stack to simulate recursion.