ECE220: Computer Systems & Programming **Fall 2021 ZJUI**
Machine Problem 3 **due: Saturday 16 October, 11:59:59 p.m.**

Using Dynamic Programming to Compute Levenshtein Distance

```
possibility -> capitalization
Levenshtein distance = 27
--possibili--t--y
cap---italization
```

Your task this week is to write subroutines that use dynamic programming to compute the Levenshtein distance between two strings, then integrate these subroutines into your MP2 code to complete the application. Given two strings and the costs for Insertion, Deletion, and Substitution, your first subroutine must initialize a table, and the second subroutine must use dynamic programming to compute the correct substring Levenshtein distances, predecessor offsets, and predecessor types for each entry in the table. Essentially, your subroutine builds the input table to MP2.

The objective for this week is to give you additional experience with understanding and manipulating arrays of data in memory.

As explained in MP2, the Levenshtein distance represents the minimum number of insertions, deletions, and substitutions necessary to transform one string into another, and is widely used to measure the difference between two strings, with applications ranging from correction of typos to genetic sequence alignment. We have generalized the idea slightly to assign costs for each operation: insertion, deletion, and substitution. In the example shown, the word "possibility" is transformed into "capitalization" by inserting the first two letters ("ca"), deleting the "oss," substituting "ta" in place of "bi," inserting "za" and "io," and finally substituting "n" for "y." The cost of each operation in the example was 2 for insertions and deletions and 3 for substitutions. In total, the change required six insertions, three deletions, and three substitutions, for a total Levenshtein distance of 27.

The Task

Computing the Levenshtein distance is fundamentally a recursive operation: any distance can be computed based on the distances for shorter (prefix) strings. Given the costs $\mathbf{C_I}$ for insertion, $\mathbf{C_D}$ for deletion, and $\mathbf{C_S}$ for substitution, let's say that we want to know the Levenshtein distance between "PARTY" and "TALL", which we denote as **`[PARTY,TALL]`**. All three of the following inequalities must hold:

$$\texttt{[PARTY,TALL]} \leq \texttt{[PARTY,TAL]} + C_I \qquad (1)$$

$$\texttt{[PARTY,TALL]} \leq \texttt{[PART,TALL]} + C_D \qquad (2)$$

$$\texttt{[PARTY,TALL]} \leq \texttt{[PART,TAL]} + C_S \qquad (3)$$

For **(1)**, we transform "PARTY" to "TAL" then insert "L" to obtain "TALL".

For **(2)**, we transform "PART" to "TALL" then delete "Y" (from "PARTY").

For **(3)**, we transform "PART" to "TAL" then substitute "L" for "Y".

Since the final characters of "PARTY" and "TALL" do no match, the last operation in transforming "PARTY" to "TALL" must be one of the three shown, and thus

$$\texttt{[PARTY,TALL]} = \texttt{MIN}\ (\texttt{[PARTY,TAL]} + C_I\ ,\ \texttt{[PART,TALL]} + C_D\ ,\ \texttt{[PART,TAL]} + C_S)$$

Rather than computing Levenshtein distances repeatedly for substrings, however, we use a technique called dynamic programming. Dynamic programming uses additional` memory to reduce computation. In this case, we use a two-dimensional table to record the Levenshtein distances for all substrings and eventually build up the desired answer. Dynamic programming is an important algorithmic technique and is used frequently in both software and hardware.

In MP3, your program is again given two NUL-terminated ASCII strings starting at memory addresses x3800 and x3840, but now you must use your new subroutines to fill in the table at x4000 and to compute the Levenshtein distance between the strings. The insertion cost $\mathbf{C_I}$ is given to your program at memory address x3880; the deletion cost $\mathbf{C_D}$ is at memory address x3881; and the substitution cost $\mathbf{C_S}$ is at memory address x3882. Note that your program is no longer given the final Levenshtein distance, as it was in MP2.

Your first subroutine, INIT_WORK, must fill in column 0 and row 0 of the table. Recall that the table has M columns and N rows, where M is the value stored at memory address x38E0, and N is the value stored at memory address x38E1, both written by your FIND_M_N subroutine. Recall also that the table uses row-major order, so address of entry (R,C) at Row R and Column C is then given by the expression x4000 + 3(RM+C).

To understand how INIT_WORK should fill in the necessary entries, let's start by thinking about the meaning of the entry (0,0). This entry corresponds to the base case, the Levenshtein distance **`[,]`** (between two empty strings). Since the two strings are the same, the Levenshtein distance is 0, and there is no preceding operation. As a result, entry (0,0) is always the same: 0 for the first memory address (x4000), 0 for the offset in the second memory address (x4001), and #-1 for the predecessor type in the third memory address (x4002). Be sure that INIT_WORK sets these three values—in particular, while the simulator may initially contain the value 0 at addresses x4000 and x4001, failing to store 0s into both addresses in your subroutine is considered a bug and will result in lost points.

Column 0—entries (R,0) for R > 0—corresponds to the Levenshtein distance between an empty string and all prefixes of the second string. Since the first string is empty, the only possible preceding operation is Insertion, and thus we can fill in each of the entries with R times the insertion cost, as shown to the right for the example second string "ZAPPY". In this case, entry (1,0) corresponds to **`[,Z]`**, entry (2,0) corresponds to **`[,ZA]`**, entry (3,0) corresponds to **`[,ZAP]`**, and so forth. For each entry (R,0)—again, for R > 0—the predecessor offset should point to entry (R - 1,0), which means a value of -3M, and the predecessor type should be 0 for insertion. Again, your subroutine must fill these entries explicitly—relying on memory already having a 0 in it will result in lost points.

	-
-	0/-
	↑
Z	$1C_I$/I
	↑
A	$2C_I$/I
	↑
P	$3C_I$/I
	↑
P	$4C_I$/I
	↑
Y	$5C_I$/I

Row 0—entries (0,C) for C > 0—corresponds to the Levenshtein distance between all prefixes of the first string and an empty string. Since the second string is empty, the only possible preceding operation is Deletion, and thus we can fill in each of the entries with C times the deletion cost, as shown below for the example first string "APPLE". In this case, entry (0,1) corresponds to **`[A,]`**, entry (0,2) corresponds to **`[AP,]`**, entry (3,0) corresponds to **`[APP,]`**, and so forth. For each entry (0,C)—again, for C > 0—the predecessor offset should point to entry (0,C - 1), which means a value of -3, and the predecessor type should be 1 for deletion.

	-	A	P	P	L	E
-	0/-	← $1C_D$/D	← $2C_D$/D	← $3C_D$/D	← $4C_D$/D	← $5C_D$/D

For simplicity, **all registers are caller-saved for the INIT_WORK subroutine**.

The second subroutine, CALC_DISTANCE, must fill in the rest of the table. You may assume that your INIT_WORK subroutine has been called before CALC_DISTANCE is called.

As you saw earlier, any given entry in the table depends only on the entries above it (insertion), to the left of it (deletion), and diagonally up and to the left (substitution/match) of it. CALC_DISTANCE can thus proceed to fill in the table row by row, starting with row 1, and to fill each row from left to right, starting with column 1. Your subroutine should use a doubly-nested loop for this purpose.

This part of the code will need more values than the LC-3 has registers, so think carefully about where you plan to put all the values you need as your write your code, and fill in comments (a register table, as well as notes about values that don't fit into registers) as you go.

For each entry, start by comparing the appropriate characters from the strings to see if they match. If they do, the cost for a match is exactly the same as the cost with one less character in each string (the entry at offset -3(M+1) from the entry being computed)—in other words, drop the $\mathbf{C_S}$ term from the right side of inequality **(3)**. If the characters do not match, the substitution cost $\mathbf{C_S}$ must be added to the potential predecessor entry's distance (the first memory location in that entry).

We suggest that you mark the entry with the substitution/match predecessor offset (as -3(M+1)) and value (as 2) regardless, allowing your later code to simply overwrite these values if a shorter distance is found, or leave them intact if no shorter distance is found.

Next, compare the distance just calculated (for substitution/match) with the distance possible using deletion. Read the Levenshtein distance from the entry at offset -3 from the entry being computed and add the deletion cost $\mathbf{C_D}$ to find the distance with deletion as a predecessor. If it is better (smaller) than that possible with substitution/match, adjust the entry appropriately (offset -3 and predecessor type 1).

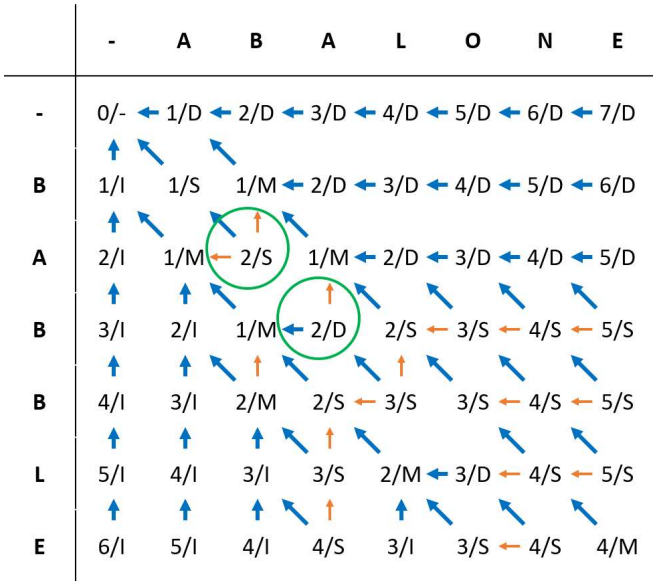
Finally, compare the best distance found so far with the distance possible using insertion. Read the Levenshtein distance from the entry at offset -3M from the entry being computed and add the insertion cost $\mathbf{C_I}$ to find the distance with insertion as a predecessor. If it is better (smaller) than that found previously, adjust the entry appropriately (offset -3M and predecessor type 1).

Be sure that you also record the best distance found in the entry before moving on to fill in the next entry.

Once you have filled in the entire table, return the Levenshtein distance of the final entry (in the lower right corner of the table) in R1.

Note that ties are possible between the three possible predecessor operations, and must be broken as implied by the preceding description. Specifically, **substitution/match must be preferred over deletion and insertion, and deletion must be preferred over insertion**. You are welcome to write your code in any way that works (do pay attention to the style guidelines, of course), but you must match the tie-breaking rules as well as all other parts of the specification.

An example in which ties occur (here $\mathbf{C_S} = \mathbf{C_D} = \mathbf{C_I} = 1$) appears to the right, with all variants of tie-breaking illustrated by the two green circles. The orange arrows represent other ties. The blue arrows represent the correct solution.



For simplicity, **all registers are caller-saved for the CALC_DISTANCE subroutine**.

Note that you will need to make minor adjustments to your main program to make use of your new subroutines. We suggest that you make these adjustments first so that you can test your subroutines more easily as you develop them. In your main program from MP2, you used PRINT_DECIMAL to print the Levenshtein distance. To do so, you loaded the distance from memory address x38C0 into R1. Replace that part of the code with two subroutine calls, first to INIT_WORK, then to CALC_DISTANCE.

You may, of course, want to make other adjustments as you develop the subroutines in order to focus more directly on whether they are working, but the changes mentioned above are needed for the final program to work completely.

Specifics

- Your code must be written in LC-3 assembly language and must be contained in a single file called `mp3.asm` in the `mp/mp3` subdirectory of your repository. We **will not grade** any other files.
- You are given the following:
  - a NUL-terminated ASCII string starting at memory address x3800,
  - a second NUL-terminated ASCII string starting at memory address x3840,
  - a positive insertion cost $\mathbf{C_I}$ at memory address x3880,
  - a positive deletion cost $\mathbf{C_D}$ at memory address x3881, and
  - a positive substitution cost $\mathbf{C_S}$ at memory address x3882.
- You may make the following assumptions about the values provided to you (and no other assumptions):
  - Both strings are valid, but either or both could be empty (0-length).
  - All three costs are positive.
- Your program must start at x3000, and must produce the appropriate output exactly. Our testing and the rubric will focus on your new subroutines, but you will lose points if you do not integrate them properly into the main program.
- You must write the INIT_WORK subroutine to use the stored values of M and N and the costs given (see above) to fill in row 0 and column 0 of the table starting at address x4000. The table has N rows and M columns. Table entries consist of three memory locations each and are stored consecutively in row-major order. Each entry consists of a computed Levenshtein distance, a predecessor offset, and a predecessor type. Entry (0,0) must have distance 0, offset 0, and type -1. Entries (R,0) for R > 0 must have distance R· $\mathbf{C_I}$, offset -3M, and type 0 (insertion). Entries (0,C) for C > 0 must have distance C· $\mathbf{C_D}$, offset -3, and type 1 (deletion). All registers are caller-saved.
- You must write the CALC_DISTANCE subroutine to complete the rest of the table based on the stored values of M and N, the strings at x3800 and x3840, the costs given in memory (see above) and the initialized table at address x4000. All remaining entries must be filled in to indicate the minimum predecessor operation between substitution/match, deletion, and insertion. If ties occur, substitution/match must be preferred over deletion and insertion, and deletion must be preferred over insertion. All registers are caller-saved.
- Both of your subroutines must be able to execute more than once according to the specifications given, and both of your subroutines must be independent of any other code EXCEPT that you may assume that we call INIT_WORK before we call CALC_DISTANCE. Note, for example, that **you may NOT assume that we call FIND_M_N** before calling either—we may instead fill M and N in memory directly, and your code must work in this case.
- You may write and use additional subroutines, but your interfaces to INIT_WORK and CALC_DISTANCE must match the specification exactly, and your main program must call both subroutines and make use of the results appropriately.
- Your code must not access the contents of memory locations other than those required for this MP or declared within your program (using .FILL or .BLKW).
- Your code must be well-commented and must include a table describing how registers and additional memory locations are used within each subroutine. Follow the style of examples provided to you in class and in the textbook.
- Do not leave any additional code in your program when you submit it for grading.

Testing

Remember that **testing is your responsibility.** We have provided several test files and scripts to help you debug your code, and suggest that you adopt the following strategy when developing your program:

1. Start by adjusting the main program, which should be easy, and adding placeholders for your new subroutines. Note that PRETTY_PRINT is likely to loop infinitely until you finish CALC_DISTANCE, so you may want to turn it off (comment out the call) at first. Don't forget to turn it back on!
2. Develop INIT_WORK, then use the example test files provided to test and debug this subroutine. Inspect the table by hand to make sure that entry (0,0) as well as the first row and column are filled in appropriately for different values of M, N, $\mathbf{C_I}$, and $\mathbf{C_D}$.
3. Next, develop CALC_DISTANCE, and test with small examples first. The scripts that we have provided are designed for use after your program is complete, so be sure to put your main program in order before using them directly.
4. Once your code seems to be working with the given examples, leverage the tools made available to you (courtesy of the ZJUI alumni) to help you to improve your coding style and programming ability. As in MP1 and MP2, while you are not required to use the VSCode extension, you will lose points if the extension reports any warnings or errors in your program. Please try out the Webtool interface when debugging your code, too. When you submit a copy of your code to Github, we will automatically test it against our solution and give you feedback about errors in your code as well as differences between your code and our solution. For MP3, you need only make up new strings and costs to create new examples, but verifying the results by hand may be slow, so think carefully. Please note, as always, that while our tools are fairly thorough, they do not guarantee that your code is free of bugs, nor that you will earn full points.

Grading Rubric

Functionality (70%)

- 5% - main program produces correct output
- 5% - INIT_WORK fills in entry (0,0) appropriately
- 10% - INIT_WORK fills in column 0 appropriately
- 10% - INIT_WORK fills in row 0 appropriately
- 10% - CALC_DISTANCE finds the correct Levenshtein distance
- 20% - CALC_DISTANCE fills in the table correctly
- 10% - CALC_DISTANCE breaks ties correctly

Style (5%)

- 5% - entry pointer is never calculated based on M nor N in CALC_DISTANCE (note: offsets MUST use M, but think about how your subroutine fills in entries in the table)
- **-10% PENALTY** - VSCode extension reports any errors or warnings; note that **even a single warning** incurs the full 10% penalty

Comments, Clarity, and Write-up (25%)

- 5% - a paragraph appears at the top of the program explaining what it does (this is given to you; you just need to document your work)
- 10% - each subroutine has a register/memory table (comments) explaining how registers/memory are used in that part of the code
- 10% - code is clear and well-commented

Note that some categories in the rubric may depend on other categories and/or criteria. For example, if your code does not assemble, you will receive no functionality points. You will also be penalized heavily if your code executes data or modifies itself (do not write self-modifying code).

Sharing MP2 Solutions

To help you in testing your code, you may make use of another student's MP2 solution as part of your MP3, provided that you **strictly obey the following**:

- You may not obtain another student's MP2 solution until **after class on Tuesday 12 October**. Violation of this rule is an academic integrity violation, will result in BOTH students receiving 0 for MP2, and may have additional consequences.
- You must clearly mark the other student's code in your own MP3, and must include the student's name in comments indicating that you are using their code. **Failure to mark their code appropriately will result in your receiving a 0 for MP3.**
- As you should know already, you may not share any additional code beyond the solution to MP2.

Please also note that if the other student's code has bugs that lead to your introducing bugs into your MP3 code, you may lose points as a result. We in no way guarantee the accuracy of any student's MP2 solution.