ECE220: Computer Systems & Programming **Fall 2023 ZJUI**
Machine Problem 2 **due: Thursday 12 October, 11:59:59 p.m.**

Pretty-Printing Levenshtein Distance

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

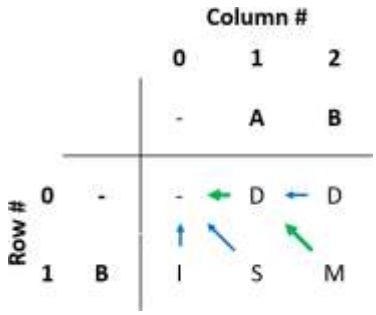
Your task this week is to write a program and two additional subroutines to support measurement and printing of the Levenshtein distance between two strings, as shown in the example to the right. In particular, given two strings, a distance, and a table (described later), you must write a main program to produce the output shown, making use of one subroutine to find the dimensions of the table and a second subroutine that uses the table to produce the output shown in the last two lines of the example. In MP3, you will add subroutines that initialize the table and use dynamic programming to find a Levenshtein distance, then integrate use of those subroutines into your main program.

The objective for this week is to give you some experience with understanding and manipulating arrays of data in memory, some experience using a stack, and a bit more experience with formatting output.

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

Your program is given two NUL-terminated ASCII strings starting at memory addresses x3800 and x3840, the measured Levenshtein distance between the strings (stored in 2s complement at memory address x38C0), and a table in memory starting at address x4000 containing the information necessary to show the operations performed to transform the first string into the second.



An example table for the strings "AB" and "B" is illustrated to the right. The table has N rows and M columns, where M is one more than the length of the first string, and N is one more than the length of the second string. In the example, M is 3, and N is 2. The first subroutine that you must write, FIND_M_N, must use your STRLEN subroutine from MP1 to compute M and N, then store them to memory at addresses x38E0 (for M) and x38E1 (for N). Each entry in the table records both the type of the predecessor entry (Substitution/Match, Deletion, or Insertion) and the offset to the predecessor entry (the arrows). The green arrows show the path from the last (lower right) entry in the table back to the first (upper left). The second subroutine, PRETTY_PRINT, must make use of this path in order to print the two strings with appropriate hyphens added. The hyphens line up the two strings so that insertions, deletions, and substitutions are easy to see.

The next page gives more details as well as a suggested order for implementing the MP.

Step 1: Write the FIND_M_N subroutine.

This subroutine is fairly straightforward, as it must make use of your STRLEN subroutine from MP1. Make a copy of your MP1 code, then add the new subroutine. All registers other than R7 must be callee-saved for the FIND_M_N subroutine (in other words, the registers' values must be preserved by the subroutine). Keep in mind that since FIND_M_N must call STRLEN twice, you must protect the value of R7 (the subroutine return address) by saving it to memory at the beginning of the subroutine and restoring it to R7 before executing RET at the end of the subroutine. The value M is the one more than the length of the first string, which starts at x3800. Store the value M to memory address x38E0. The value N is the one more than the length of the second string, which starts at x3840. Store the value N to memory address x38E1.

Step 2: Write the main program.

Next, write the main program, which prints the first two lines of the output and calls your subroutines. Note that your output must match exactly, so be careful when writing the extra strings that your program uses.

Start your main program by calling your FIND_M_N subroutine.

The text on the first line of output must consist of the first string, a single space, a hyphen, a greater-than sign, another space, the second string, and a line feed (ASCII character x0A). We suggest using PUTS to write strings and OUT to write single characters, but you are welcome to re-implement the traps yourself if you are interested in doing the I/O directly.

The text on the second line of output must be exactly "**`Levenshtein distance =`** " (including all three spaces) followed by the decimal value of the distance given to you in memory address x38C0, and finally a line feed (x0A).

Your main program must make use of PRINT_DECIMAL from MP1 to print the decimal value of the Levenshtein distance, and must make use of the PRETTY_PRINT subroutine after printing the first two lines. Be sure to include a HALT trap at the end of your main program to stop the LC-3.

Step 3: Write the PRETTY_PRINT subroutine.

Finally, write the PRETTY_PRINT subroutine to walk the table and produce the last two lines of output to the display. To do so, you must understand the details of the table structure. The illustration to the right gives the exact contents of memory for the table shown on the previous page, for the strings "AB" and "B". Each table entry actually consists of three consecutive memory locations. The first memory location in each entry should be ignored—you will use it in MP3—these locations are marked as "L.D." in the explanation on the right. The second memory location in each entry records the offset of the predecessor entry. The third and final memory location in each entry records the predecessor type as a small integer: both Substitution and Match are encoded as 2 (you do not need to differentiate them); Deletion is encoded as 1; and Insertion is encoded as 0.

Address	Contents	Meaning
x4000	bits	L.D. (0,0)
x4001	#0	(starting entry)
x4002	#-1	(starting entry)
x4003	bits	L.D. (0,1)
x4004	#-3	offset to (0,0)
x4005	#1	Deletion
x4006	bits	L.D. (0,2)
x4007	#-3	offset to (0,1)
x4008	#1	Deletion
x4009	bits	L.D. (1,0)
x400A	#-9	offset to (0,0)
x400B	#0	Insertion
x400C	bits	L.D. (1,1)
x400D	#-12	offset to (0,0)
x400E	#2	Match/Sub.
x400F	bits	L.D. (1,2)
x4010	#-12	offset to (0,1)
x4011	#2	Match/Sub.

As memory consists of a linear array of addresses, entries in the two-dimensional table must be put into a single linear order before they are placed into memory. To do so, we make use of a row-major order, in which each row forms a contiguous region of memory. The address of the entry (R,C) at Row R and Column C is then given by the expression x4000 + 3(RM+C). For example, entry (1,2) starts at address 0x4000 + 3(1·3+2) = x400F. Looking at entry (1,2) in the illustration on the

previous page, we find the letter M and a green arrow pointing to entry (0,1). Entry (0,1) starts at address x4000 + 3(0·3+1) = x4003, so the offset in entry (1,2) is x4003 – x400F = #-12, as shown at address x4010. And the value 2 at address x4011 records the predecessor type, in this case, M.

To walk the table, your subroutine should use M and N to find the address of the last entry in the table (at row N-1 and column M-1), then use the offset element in each entry to follow the "arrows" back to the first entry. Finding the predecessor entry is easy: add any entry's offset value to its address to obtain the offset of the predecessor entry.

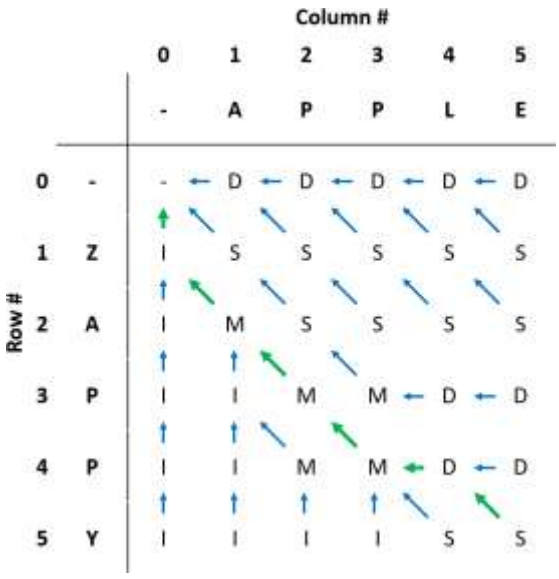
You may assume that all offsets in the table are valid (in other words, adding an entry's offset value to the entry's address produces a valid entry address within the table) and that following the offsets starting from the last entry will always reach the first entry in a reasonable number of steps (the actual bound is M-N-2, but that shouldn't matter to your code).

The first entry (row 0 and column 0, at address x4000) lacks a predecessor entry, so its offset is set to 0 (see memory address x4001 in the example) and its predecessor type is set to #-1 (see memory address x4002). When your code finds a negative predecessor type, it has reached the first entry in the table.

As your subroutine walks the table, it should use a stack to build up a list of operations based on the predecessor types. Use R6 to store the stack pointer, as discussed in class, and use x8000 as the base of your stack (to simplify checking for an empty stack later—this value can be added to itself to produce 0). Starting with the last entry in the table, push the predecessor type for that entry on to the stack before moving to the next entry.

When your subroutine reaches the first entry in the table, the stack should contain the sequence of operations needed to transform the first word into the second. Let's look at a more complex example, as shown in the table to the right. Here, the strings are "APPLE" and "ZAPPY". After your subroutine has walked the green path in the table, starting from entry (5,5) in the lower right, the stack should appear as shown below:

Address	Contents	Meaning	
x7FFA	x0000	Insertion	←top of stack
x7FFB	x0002	Match/Sub.	
x7FFC	x0002	Match/Sub.	
x7FFD	x0002	Match/Sub.	
x7FFE	x0001	Deletion	
x7FFF	x0002	Match/Sub.	
x8000			←base of stack



Reading the example stack from the top, we see the operations necessary to transform "APPLE" into "ZAPPY": first, insert "Z"; next, match three consecutive letters, in this case "APP" in both words; then delete the "L"; finally, substitute "Y" for "E".

```
APPLE -> ZAPPY
Levenshtein distance = 3
-APPLE
ZAPP-Y
```

Using the stack, your subroutine must then produce the last two lines of output shown to the right. To do so, walk down the stack (from top to base) once for each string (be sure to save the top of stack for use with the second string!). For the first string, finding an Insertion predecessor type (the value 0) should print a hyphen (ASCII character x2D), while any other predecessor type should print the next character in the string. After printing the first string, print a line feed (x0A) and move on to the second string. For the second string, finding a Deletion predecessor types (the value 1) should print a hyphen, while any other predecessor type should print the next character in the string. After printing the second string, print another line feed (x0A) to

complete the subroutine. Be sure to terminate on reaching the base of the stack for both strings, not when the end of the string is reached.

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

Specifics

- Your code must be written in LC-3 assembly language and must be contained in a single file called **`mp2.asm`** in the **`mp/mp2`** 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 Levenshtein distance stored as a non-negative 2s complement value at memory address x38C0, and
  - a table starting at address x4000.
- 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).
  - The Levenshtein distance is non-negative.
  - The table size matches the length of the strings (MN entries in total).
  - Starting from the last entry, the table contains a path of offsets back to the first entry.
  - The first entry in the table always has offset 0 and predecessor type -1.
  - All other predecessor types in the table are 0 (Insertion), 1 (Deletion), or 2 (Match/Substitution).
- Your program must start at x3000, must produce the appropriate output exactly, and must call the FIND_M_N, PRINT_DECIMAL, and PRETTY_PRINT subroutines in order to do so. The first line of output must contain the first string, the four-character sequence " **`->`** " (including two spaces), the second string, and a line feed (ASCII character x0A). The second line must start with "**`Levenshtein distance =`** " (including three spaces), then the decimal value of the Levenshtein distance (printed using PRINT_DECIMAL), and finally a line feed. The remaining two lines of output must be printed by PRETTY_PRINT.
- You must write the FIND_M_N subroutine to calculate M, one more than the length of the first string, and store it at memory address x38E0, as well as N, one more than the length of the second string, and store it at memory address x38E1. For both strings, your subroutine must call STRLEN to measure the length of the string. All registers other than R7 must be caller-saved for the FIND_M_N subroutine.
- You must write the PRETTY_PRINT subroutine, for which all registers are caller-saved. Pretty-print must use the values of M and N (stored in memory at address x38E0 and x38E1, respectively) to find the last entry in the table, then walk back along the path of offsets to the first entry, pushing each predecessor type onto the stack with base address x8000. Once the first entry has been reached, the subroutine must use the stack to print the first string, inserting a hyphen (ASCII character x2D) each time an Insertion (value 0) is found on the stack, and printing a character from the string for each other predecessor type. After printing the first string, the subroutine must print a line feed. The subroutine must then use the stack a second time (do not rebuild it—just reuse the data on the stack!) to print the second string, inserting a hyphen (ASCII character x2D) each time an Deletion (value 1) is found on the stack, and printing a character from the string for each other predecessor type. After printing the second string, the subroutine must print a line feed.
- All of your subroutines must be able to execute more than once according to the specifications given, and all of your subroutines must be independent of any other code.

- You may write and use additional subroutines, but your interfaces to FIND_M_N and PRETTY_PRINT must match the specification exactly, and your main program must perform all tasks specified and make use of the subroutines required.
- 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 are used within each part of PRETTY_PRINT: traversing the path and printing each of the two strings. 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. Use the example test files provided to test and debug your code after each step. For example, once you have implemented FIND_M_N, use the example files to check that your subroutine executes correctly. The scripts are designed for use after your program is complete, so you will need to execute them partially or make your own script versions so as to inspect your results.
2. 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, 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 MP2, constructing example input tables on your own is time-consuming, so the tools will be even more helpful in identifying subtle errors in your code. Please note, however, that while the 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 (50%)
- 10% - main program produces correct output
- 10% - FIND_M_N subroutine finds and stores M and N correctly
- 5% - FIND_M_N preserves all registers other than R7
- 15% - PRETTY_PRINT builds stack correctly by traversing path contained in table
- 5% - PRETTY_PRINT prints first string correctly, including terminal line feed
- 5% - PRETTY_PRINT prints second string correctly, including terminal line feed

Style (20%)
- 10% - PRETTY_PRINT builds stack from path through table exactly once
- 5% - main program calls FIND_M_N, PRINT_DECIMAL, and PRETTY_PRINT appropriately
- 5% - FIND_M_N calls the STRLEN subroutine appropriately to find string lengths
- **-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 (30%)
- 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)
- 15% - each of the three parts of PRETTY_PRINT (traversing the path and printing each of the two strings) has a register table (comments) explaining how registers 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. Note also that the remaining LC-3 MP (MP3) will build on these subroutines, so you may have difficulty testing those MPs if your code does not work properly for this MP. You will also be penalized heavily if your code executes data or modifies itself (do not write self-modifying code).

Sharing MP1 Solutions

To help you in testing your code, you may make use of another student's MP1 solution as part of your MP2, provided that you **strictly obey the following**:

- You may not obtain another student's MP1 solution until **after class on Tuesday 28 September**. Violation of this rule is an academic integrity violation, will result in BOTH students receiving 0 for MP1, and may have additional consequences.
- You must clearly mark the other student's code in your own MP2, 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 MP2.**
- As you should know already, you may not share any additional code beyond the solution to MP1.

Please also note that if the other student's code has bugs that lead to your introducing bugs into your MP2 code, you may lose points as a result. We in no way guarantee the accuracy of any student's MP1 solution.