Homework 4

Points: 100 points + 15 bonus (a full score is 100). The extra bonus points will be added to the total homework points when computing the average. The bonus points do not come from specific components. Anything above 100 points is bonus.
Topics: Dynamic programming: Edit Distance (compute the distance and print the table to verify correctness); given a word find the most similar words in the dictionary; fix words.

Program (115 points)

Watch the Canvas video that presents this homework (under the Canvas entry for this assignment).

You will write all your code in the file `spell.c` (provided). A client file, `spell_checker.c`, is provided. It implements the high level behavior of the program and calls the specific functions that you must implement in spell.c. The program has 2 main parts:

- Part 1: the user repeatedly enters pairs of words to compute the edit distance for. The table and final answer will be printed for each pair. It will stop with -1 -1.
- Part 2: implements part of a spell checker functionality. It will take a dictionary file name and a test filename and for each word in the test file it will offer a set of suggestions from the dictionary file and let the user decide what to do.

You will implement:

- `int edit_distance(char * first_str, char * second_str, int print_on)`
- `void spell_check(char * dictname, char * testname)`
- any helper functions you need so that the above functions are not too big. E.g. you should write a separate function for printing the table and call it from the `edit_distance` function.
- Calculate the **worst case** time complexity for computing the edit distance from T test words to D dictionary words where all words have length MAX_LEN.
    - Write this time complexity at the top of your file as a comment.
    - In your answer, use variables: T, D, MAX_LEN where:
    - T = number of words in the test file,
    - D = number of words in the dictionary file
    - MAX_LEN = maximum length of a word (from the test file or dictionary file).
    - Since the word length can vary, you should assume the worst case, that is, assume that every test word and every dictionary word is size MAX_LEN. Give the Θ for this worst case scenario.

Details:

1. `int edit_distance(char * first_str, char * second_str, int print_on)`
    - Is **case sensitive**. E.g.`edit_distance("dog","DoG")` is 2, not 0.
    - Returns the edit distance between `first_str` and `second_str`.

- If `print_on` is 1, it will print the edit distance table (including the corresponding letters) and also the distance. If `print_on` is anything other than 1, this function will not print anything. (I recommend writing a separate function, `print_table`, that takes the strings and the table and prints whatever needs to be printed, and calling it from the `edit_distance` function.)
- Implements the BOTTOM-UP method (i.e. no recursion) for the Edit Distance.
- You can use either the function we used in the lecture, or the one below. They will both give the same answer (same numbers in the table).

```
Dist(0,0) = 0
Dist(0,j) = j
Dist(i,0) = i
Dist(i,j) = Dist(i-1,j-1) if x_{i-1} == y_{j-1}
            1 + min { Dist(i-1,j), Dist(i,j-1), Dist(i-1,j-1) }  if x_{i-1}≠y_{j-1}
```

2. `void spell_check(char * dictname, char * testname)`
   0. Open the dictionary file and store the words in a table (an array of strings). Make sure you allocate space correctly for the strings. Otherwise it can be a source of bugs, Valgrind errors, crashes.
   1. For each word in the test file:

print it to the screen. Put two vertical bars around it to be able to tell if you read any extra space or not with the word. E.g. print |*can*| , not just *can*.
Print these options to the user as to what correction to be used for this word:

```
-1  - user will type the correct spelling
 0  - leave the word as is (do not apply any correction)
      Minimum distance: d
      Words that give minimum distance:
 1 word1
 2 word2
 ...
 x wordx
Enter your choice (from the above options):
```

(x will be different based on how many words are at that minimum edit distance from the test word).

The user will enter their choice. If the choice is (-1) the program will also allow the user to type in a word. See sample runs.
Print the corrected word.

   2. To find the most similar words (1 to x) in the dictionary file, I recommend the steps below, but you can follow another method, if you prefer.

      - compute, and store in an array the edit distances between the test word and all the dictionary words.
      - find the smallest distance in that array
      - print that distance
      - print all the dictionary words that have that edit distance: find all the distances that are equal to the smallest distance and print the corresponding dictionary words.
      - If the user chooses one of these words as the correction, you must be able to print it again. To do that, you can store the words, or you can store their index (from

the dictionary array) and use that to print the word. (E.g. if the first similar word, word1, is "your" and it is at index 5 the dictionary array, you could store "your", or you could store 5).

Files:

1. **spell_checker.c** - do not modify
2. **spell.h** - do not modify
3. **spell.c** - write your code and time complexity here.
4. **dsmall.txt , dmed.txt , dbig.txt** - dictionary files of 3 sizes (small, medium, big): dmed.txt is a slight modification of the 1-1000.txt file from https://gist.github.com/deekayen/4148741. dbig.txt is taken from https://github.com/first20hours/google-10000-english/blob/master/google-10000-english.txt
5. **list1.txt** - test file with misspelled words.
6. **run_dmed_list1.txt** - Sample run with user input for part 2 (skips part 1 by giving -1 -1 right away) uses the medium size dictionary, **dmed.txt**, and the **list1.txt** test file .
7. For testing part 1, instead of typing the words every time, you can use input redirection: whatever you would type during program execution you put in a file and redirect input from that file. The main goal is to test part 1, so the redirection file will have pairs of words for testing part 1 and bad names for dictionary and test files so that part 2 will not run. Note that you do NOT have to write any C code to read from this file. The Unix system will do that for you. Input redirection will be used in grading part 1. Example 1:
   - download or create **redir1.txt** , file to be used for input redirection.
   - execute: `valgrind --leak-check=full ./a.out < redir1.txt`
   - produces output shown in **run_redir1.txt** .

   Example 2:

   - download or create **redir_100_4.txt** . It contains words of max length (100) for part 1.
   - execute: `valgrind --leak-check=full ./a.out < redir_100_4.txt`
   - produces output shown in **run_redir_100_4.txt**

Input constraints:

1. Any word has at most 100 characters. That includes: the file names, dictionary words, test words, words given by the user for part 1 or part 2 (including the words from input redirection files used in testing part 1).
2. All the words in the files are lower case.
3. Every file has on the first line the number of words it contains:

```
N
word1
word2
...
wordN
```

where $1 \le N \le 20000$ and `N` is valid integer.

Requirements:

1. Standard: no compilation errors, no compilation warnings, global variables not allowed, proper and consistent indentation (only tabs or only spaces used), signature of the functions in the .h file should not be modified.
2. Unless otherwise specified, **every function must work for all sizes and variations of the data**. E.g. do not hardcode file names or the number of words/lines in files.
3. The provided files are given with the Unix EOL (end-of-line) Since your program will run on a Unix system (omega/VM/Ubuntu) it will have to run with this type of files. Since you are working with strings, this may be more important now than in past assignments.
4. The table showing the edit distance calculation must match the one in the sample run:
    1. prints corresponding letters
    2. 3 spaces reserved for number display in each cell (cells in each column must be aligned),
    3. horizontal bars (|) between cells and
    4. horizontal lines (of dashes) between rows have same length as the data.

    ("Why do we have to print the table and the letters?" - It is important to be able to print the data from your program in a formatted, easily readable way that allows you to easily check and verify that the program does what you want it to do. Printing the table for the edit distance along with the indices and corresponding letters from the strings allows you to check that the program generates the same table as we did in class or for any other test case you develop on your own on paper.)

What might be challenging or a source of bugs here?

1. Working with strings: storing them, not having \n at the end as it will affect the length of the words and thus the table and possibly the edit distance
2. Printing the corresponding letters in the table. Possibly also aligning the cells. Hint here: for a row in the table, most of the data might be printed by printing in a loop, but other parts of that row (at beginning and/or end) can be printed outside of the loop.
3. Finding the words at the smallest edit distance. This is not hard, but it would be if you try to do it all in a single pass instead of the 2-pass method recommended here.

Suggestions:

1. DO NOT leave it for the last few days. Start early! There are small, and independent, or almost independent, components that you can work on:
    - Read and save in an array the dictionary words.
    - Read and print each test word.
    - Print the table in the required format (with letters).
    - Implement the `edit_distance` function
2. Check that your program works with user input. This may be used when grading.
3. After that check that it works with input redirection. This will be the main method used in grading as it can be automated.
4. Write helper functions and call them as needed. (Do not put all the code in just 2 functions.)

Extra resources for programming components needed for this assignment:

1. For strings, remember to allocate at least one extra char in addition to the max length to hold the '\0'.
2. If you want to store the dictionary as an array of pointers and see how to pass that as an argument to a function, review the dynamic allocation for 2D arrays. In both cases, a row is a pointer to a 1D malloc-ed/calloc-ed array.
3. Passing 2D array as an argument in C depends on how the array is created. Review our past assignments (for dynamically allocated 2D arrays) or check this page: https://www.geeksforgeeks.org/pass-2d-array-parameter-c/ .
4. To print an integer on 5 reserved spaces you should use printf and specify the minimum width for printing. E.g. see how the numbers printed by the following printf statements print the numbers aligned because 5 spaces are reserved (and so even the 'shorter' numbers use 5 spaces and they align well with the 'longer' numbers. For your homework solution you need to reserve a different number of spaces (not 5 as here).

   printf("-%5d-",16); printf("-%5d-",3976); printf("-%5d-",2); printf("\n"); printf("-%5d-",8257); printf("-%5d-",8); printf("-%5d-",52);

5. In order to print the horizontal line of dashes, you should count how many cells are in a row and what the width of one cell is. Remember to count 1 for the "wall" (i.e. "|") of the cell. Should you count one wall or two walls for each cell? Test you calculations for a small table like the one for "cs" and "cat" in the sample output.
6. If you are still struggling with printing a formatted table, you can work on the following helper problem (with a classmate or on your own). Working on this will NOT be considered collusion. Problem: read a word from the user and print it repeatedly in a table of 3 rows and 4. E.g.
7.
8. ```
   Enter a word: cat
   +  cat+  cat+  cat+  cat+
   ~~~~~~~~~~~~~~~~~~~~~~~~~
   +  cat+  cat+  cat+  cat+
   ~~~~~~~~~~~~~~~~~~~~~~~~~
   +  cat+  cat+  cat+  cat+
   ~~~~~~~~~~~~~~~~~~~~~~~~~
   ```

   Another sample run:

   ```
   Enter a word: cs
   +   cs+   cs+   cs+   cs+
   ~~~~~~~~~~~~~~~~~~~~~~~~~
   +   cs+   cs+   cs+   cs+
   ~~~~~~~~~~~~~~~~~~~~~~~~~
   +   cs+   cs+   cs+   cs+
   ~~~~~~~~~~~~~~~~~~~~~~~~~
   ```

How to submit

Submit only the spell.c file. You can submit other file(s) with your own test cases if you develop any, but your code will be tested with the posted data files and other test files we prepare. We will only use your spell.c file.

Include the compilation instructions (especially if they differ from the one shown here). The assignment should be submitted via Canvas. As stated on the course syllabus, programs must be in C, and must run on omega.uta.edu/VM/Ubuntu.
   ```

IMPORTANT: Pay close attention to all specifications on this page, including file names and submission format. Even in cases where your answers are correct, points will be taken off liberally for non-compliance with the instructions given on this page (such as wrong file names). The reason is that non-compliance with the instructions makes the grading process significantly (and unnecessarily) more time consuming. Contact the instructor or TA if you have any questions.

Grading criteria

Penalties:

Compilation ERROR(S) : 0 credit (final grade 0)

If only 1 or 2 minor syntax errors such as missing ; or misspelled variable name: 10 points penalty per error.

If major errors, or 3 or more minor errors: no credit.

Compilation WARNING(S): 10%-50% penalty

Code crashes : 30%-100% penalty

Global variables : 50% penalty if global variables are used in any place.

(Macros used to store limits such as max word size are not global variables, and are allowed).

Indentation incorrect or inconsistent (mixed spaces and tabs) : 5 points penalty

Submitted file is named something other than spell.c : 5 points penalty

Point distribution:

115 points total

10 pts - Valgrind

20 pts - correct edit distance computation (part 1 of program)

17 - correct calculation of all edit distance (for all cases tested on)

If the table cannot be printed, we cannot verify that it is correct: 15 points penalty

If recursive, instead of iterative (with loop) method is used: 15 points penalty.

3 - correct behavior for print/no print argument: prints for value 1, does not print for any other value

25 pts printing the table - partial credit:

5 - correct top row, i.e. column labels (letter)

5 - correct leftmost column, i.e. row labels (letter)

8 - inside cells (all cells that contain a number) are aligned.

5 - cells are aligned.

3 - cells have exactly 3 spaces (including the number in the table) (between | |)

5 - horizontal lines of dashes are printed and match the length of the row

2 - The final edit distance is printed on a separate line after the table.

5 pts - Give the worst case time complexity to compute the edit distance from T test words to D dictionary words where all words (from dictionary and misspelled) have length MAX_LEN.

5 pts - if any of the files fails to open, program will not crash and will not generate memory errors.

10 pts - correct display for each processed word: the word is printed with | around it (see |tis| in example below), the options menu for corrections is printed, takes the user's answer(s). Here a 2pt penalty is applied for each missed component.

40 pts - Correct spell check part (part 2 of program: read in files and do spell check)

15 - minimum distance is correct (10pts) and printed before the words are printed (5pts).

15 - all words at minimum distance are found and printed (10pts), and their corresponding option numbers start from 1 (5 pts)

10 - User selection works. If the user selects option 1 or higher, the corresponding word is printed. If an invalid option is given, a message is displayed and the original word is used (see demo in video).

E.g. :

```
---> |tis|
-1 - type correction
 0 - leave word as is (do not fix spelling)
     Minimum distance: 1
     Words that give minimum distance:
 1 - his
 2 - is
 3 - this
```

```
 4 - tie
Enter your choice (from the above options): 3
The corrected word is: this
```