Students:
Section 14.5 is a part of 1 assignment:
**Programming Project 05 - Link Lint List - Find Shortest Word Ladder**

This assignment's due date has passed. Activity will still be recorded, but will not count towards this assignment (unless the due date is changed). See this article for more info.

Includes: 🟩 zyLab

Due: 11/07/2024, 11:59 PM CST

Instructor created

# 14.5 P05 - Link Lint List - Find Shortest Word Ladder

## Word Ladder Solver

### Introduction to Word Ladders (reproduced from Project03)

A **word ladder** is a bridge between one word and another, formed by changing one letter at a time, with the constraint that at each step the sequence of letters still forms a valid word. For example, here is a word ladder starting from the word **"data"** and *climbing up* the ladder to the word **"code"**. Each changed letter is bolded as an illustration:

```
|   code   |
|     ^    |
|   cove   |
|    ^     |
|   cave   |
|   ^      |
|   gave   |
|     ^    |
|   gate   |
|   ^      |
|  date  |
|     ^  |
|  data  |
```

This word ladder has height 7. There are many other word ladders that connect these two words. For example, here is another ladder of height 5 that uses some more obscure words:

```
|  code  |
|   ^    |
|  cade  |
|    ^   |
|  cate  |
|  ^     |
|  date  |
|     ^  |
|  data  |
```

In fact, this ladder has the shortest height that connects **data** to **code**. Note that there might be other ladders of the same height, but none with fewer rungs than this one. Notice that the shortest a word ladder can be is 2, e.g. **cove** -> **code**. There are also pairs of words for which a word ladder cannot be formed, e.g. there is no word ladder to connect **stack** -> **queue**.

In this project, you will write functions to support a program that prompts the user for two words and finds a minimum-height ladder linking the words, which uses linked lists of words to store ladders, linked lists of partially completed ladders to organize all possible solutions, and a prescribed algorithm to find the shortest word ladder.

## Dictionary File, Word Array, Word Struct, and a Linked List for the Word Ladder

The provided file *dictionary.txt* contains the full contents of the "Official Scrabble Player's Dictionary, Second Edition." This word list has over 120,000 words, which should be more than enough for the purposes of making word ladders for small to moderate sized words. Smaller dictionaries are also provided for testing purposes: *simple3.txt* contains a limited number of 3-letter words, *simple4.txt* contains a limited number of 4-letter words, and *simple5.txt* contains a limited number of 5-letter words.

The complete primary application is provided in **main()** of the starter code; however, it involves many calls to functions that still need to be written. Here are the main steps of the program:

- The user interactively sets the length of the starting and final words for the word ladder.

This, in turn, also sets the **wordSize** for the full set of words to be read in from the dictionary file.

- The user also interactively inputs the dictionary file name to be used for reading words into the full array of possible words that could later make up the ladder.
- The dictionary file is opened for the first time and is scanned to count the number of words it contains that has the desired **wordSize**; this count is stored as **numWords**.
- We now know the user-specified word length (**wordSize**) AND the number of words in the dictionary that have the correct length (**numWords**). Thus, we can now allocate array space for all of the words, open the file again, and read the words in from the file to fill the newly-allocated array. The full set of words is stored using a heap-allocated array of pointers to C-strings. That is, **words** is an array of pointers with size **numWords**, where each pointer points to a heap-allocated C-string of size **wordSize+1** (to allow space for a word and the null character). See the figure below for a diagram of the **words** array.
- The user interactively inputs both the starting word (**startWord**) and the final word (**finalWord**). If either entered word has an incorrect size (i.e. not equal to **wordSize**) or the word is not found in the **words** array, then the user is requested to enter another word.
- With the **words** array filled from the dictionary and the two ends of the word ladder set, an *algorithm* is run to produce the minimum-height word ladder. This entails building a linked list of **WordNode** structs, which contain pointers to C-strings in the **words** array and a pointer to the **next** element in the linked list. Full details on the minimum-height word ladder *algorithm* are in the next section.
- If a word ladder connecting the two words is possible, the minimum-height ladder is displayed (with the starting word on the bottom and the final word at the top of the ladder), along with the ladder height.

**Example:**

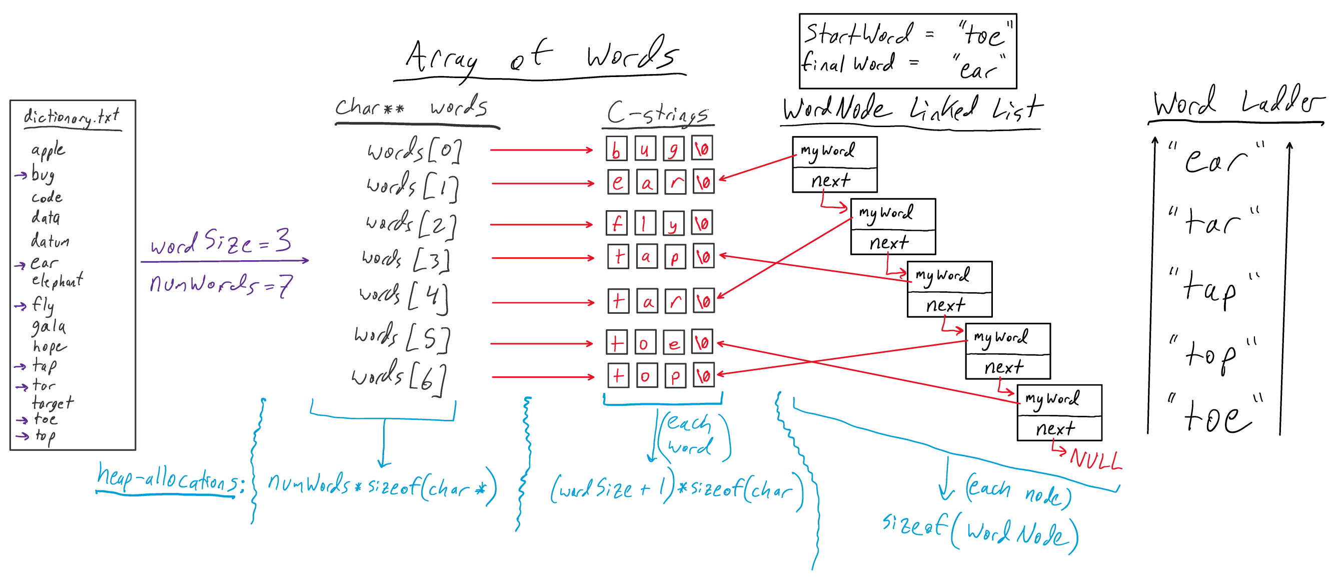
The diagram below uses a sample dictionary of only 15 words, where the word length varies from 3 to 8. If **wordSize** is selected to be 3, then **numWords** would be set to 7, since there are seven 3-letter words in the dictionary file: **bug, ear, fly, tap, tar, toe, top**.  Note that the seven words are heap-allocated C-strings with an extra character for **'\0'**. Then, **startWord** is chosen to be **"toe"** and **final**Word is chosen to be "**ear**". The *algorithm* produces a linked list where the head node's **myWord** pointer points to the **finalWord** C-string in the **words** array, i.e. **"ear"**. It is important to notice that the linked list node does NOT store characters for the word, nor is an additional copy of the word made; instead the linked list node simply points to an element of the **words** array. So...

- the **head** node points to **"ear"** in the **words** array, then following to **next**,
- the **next** node points to **"tar"** in the **words** array, then following to **next**,
- the **next** node points to **"tap"** in the **words** array, then following to **next**,

- the **next** node points to **"top"** in the **words** array, then following to **next**,
- the **next** node points to **"toe"** in the **words** array, then following to **next**,
- the **next** node points to **NULL**

Thus, the linked list is the word ladder, with the **finalWord** pointed to by the head node and the **startWord** pointed to by the last node.

Figure 14.5.1: Diagram of a sample word ladder - words array & ladder linked list



## The Minimum-Height Word Ladder Finding *Algorithm*

Finding a word ladder is a specific instance of what is known as a shortest-path problem: a problem in which we try to find the shortest possible route from a given start to a given end point. Shortest-path problems come up in routing Internet packets, comparing gene mutations, Google Maps, and many other domains. The strategy we will use for finding the shortest path between our start and end words is called breadth-first search ("BFS"). This is a search process that gradually expands the set of paths among which we search *outwards*: BFS first considers all possible paths that are one step away from the starting point, then all possible paths two steps away, and so on, until a path is found connecting the start and end point. A step can be understood as one unit of measurement; depending on the problem, this could be a millisecond, a minute, a mile, a subway stop, and so on. By exploring all possible paths of a given length before incrementing to the next length, BFS guarantees that the first solution you find will be as short as possible.

For word ladders, start by examining ladders that contain only words that are one "step" away from the original word; i.e., words in which only one letter has been changed. If you find your target word among these one-step-away ladders, congratulations, you're done! If not, look for your target word in all ladders containing words that are two steps away; i.e., ladders in which two letters have been changed. Then check three letters, four, etc., until your target word is located.

We implement the breadth-first algorithm using a linked list of partially complete ladders, each of which represents a possibility to explore (i.e., each item in the list of partial ladders is examined in turn, checking to see if it contains a path to our target word). Each partial ladder is represented as a linked list of words, which means that your overall collection will be a linked list of partially complete ladders, which are themselves linked lists of words.

A word ladder is a linked list of **WordNode** structs, which is defined as follows:

```
typedef struct WordNode_struct {
    char* myWord;
    struct WordNode_struct* next;
} WordNode;
```

Here **myWord** is a pointer to a C-string element of the **words** array and **next** is a pointer to the next element in the list of words.

The algorithm centers on storing partially completed word ladders in a linked list of **LadderNode** structs, which is defined as follows:

```
typedef struct LadderNode_struct {
    WordNode* topWord;
    struct LadderNode_struct* next;
} LadderNode;
```

Here **topWord** is a pointer to the head node of a partially completed word ladder and **next** is a pointer to the next element in the list of ladders.

Below is a partial pseudo-code description of the *algorithm* to solve the word-ladder problem *(Note: some students may complain at this point that we are giving too much information and that they want to figure the problem out on their own. In that case, great! Don't look at the pseudocode below if you want to try it on your own!)*

```
To find the shortest word ladder between words w1 and w2:
    Create myList, an empty list of LadderNode structs
    Create myLadder, an empty list of WordNode structs
    Prepend w1 to the front of myLadder
    Append myLadder to the back of myList
```

```
   While myList is not empty:
      Pop the head LadderNode off the front of myList, call it myLadder
      For each word in the words array that is a neighbor of the head myWord of myLadder:
         If the neighbor word has not already been used in a ladder to this point:
            If the neighbor word is w2:
               Prepend w2 to the front of myLadder
               Hooray! We found the shortest word ladder, so return myLadder
            Otherwise:
               Copy myLadder to anotherLadder
               Prepend neighbor word to the front of anotherLadder
               Append anotherLadder to the back of myList
   If no ladder was returned, then no ladder is possible
```

Some of the pseudo-code corresponds almost one-to-one with actual C code. Other parts are more abstract, such as the instruction to "pop" a **LadderNode** from the front of **MyList.** Popping a node from the front of a list means that the the head node is removed from the list, such that the node that was one away from the head node is the new head node. In this case, we are interested in keeping the popped ladder for further analysis, but it is no longer connected to the linked list.

Another instruction that needs more explanation is to examine each "neighbor" of a given word. A neighbor of a given word **w** is a word of the same length as **w** that differs by exactly 1 letter from **w**. For example, **“date”** and **“data”** are neighbors; **“dog”** and **“bog”** are neighbors; **“debug”** and **“shrug”** are NOT neighbors.

Your solution is not allowed to look for neighbors by looping over the full dictionary, nor looping over the entire **words** array, every time; this is much too inefficient. To find all neighbors of a given word, use two nested loops: one that goes through each character index in the word, and one that loops through the letters of the alphabet from a-z, replacing the character in that index position with each of the 26 letters in turn. For example, when examining neighbors of “date”, you'd try:

- **aate, bate, cate, ..., zate** ← *all possible neighbors where only the 1st letter is changed.*
- **date, dbte, dcte, ..., dzte** ← *all possible neighbors where only the 2nd letter is changed.*
- **...**
- **data, datb, datc, ..., datz** ← *all possible neighbors where only the 4th letter is changed.*

Note that many of the possible letter combinations along the way (**aate, dbte, datz**, etc.) are not valid English words. Your algorithm has access to the **words** array, which is built from an

English dictionary, and each time you generate a word using this looping process, you should look it up in the **words** array to make sure that it is actually a legal English word. Only valid English words should be included in your group of neighbors. Since we will be searching the **words** array many times, we will take advantage of the dictionary file being in alphabetical order, which produces a **words** array that is also in alphabetical order, which allows an efficient process for finding words using a *binary search*.

Another way of visualizing the search for neighboring words is to think of each letter index in the word as being a "spinner" that you can spin up and down to try all values A-Z for that letter. The diagram below tries to depict this:

Figure 14.5.2: Efficient search for neighbor words

```
index      0   1   2   3
          ... ... ... ...
           a   m   b   c
           b   n   c   d
         +---+---+---+---+
         | c | o | d | e |
         +---+---+---+---+
           d   p   e   f
           e   q   f   g
          ... ... ... ...
```

Another subtle issue is that you should not reuse words that have been included in a previous, shorter ladder. For example, suppose that you have the partial ladder **cat** → **cot** → **cog** in your list. Later on, if your code is processing the ladder **cat** → **cot** → **con**, one neighbor of **con** is **cog**, so you might want to examine **cat** → **cot** → **con** → **cog**. But doing so is unnecessary. If there is a word ladder that begins with these four words, then there must be a shorter one that, in effect, cuts out the middleman by eliminating the unnecessary word **con**. As soon as you've entered a ladder in your list ending with a specific word, you've found a minimum-length path from the starting word to that end word in that ladder, so you never have to enter that end word again in any later ladder.

To implement this strategy, we keep track of the set of words that have already been used in any ladder, and ignore those words if they come up again. Keeping track of which words you've used also eliminates the possibility of getting trapped in an infinite loop by building a circular ladder, such as **cat → cot → cog → bog → bag → bat → cat.**

The set of previously-used words is recorded in the Boolean array **usedWord**, which has size **numWords** to align with the **words** array by index. The elements of **usedWord** are all initialized to **false**, representing no words have been added to any ladders yet. Thus, if the algorithm seeks to add **words[i]** to a ladder, first check **usedWord[i]**; if it is **false**, then proceed to add **words[i]** to the ladder and set **usedWord[i]** to **true**; otherwise, ignore **words[i]** and move on. This should feel familiar, as it is a form of memoization.

Lastly, whereas the pseudo-code does a nice job laying out the important structural elements of the algorithm and the underlying linked list structures, it makes no effort at complete memory management. The algorithm requires a great deal of heap-memory allocations, i.e. for each **WordNode** and **LadderNode**. The vast majority of these allocations (but not all) go out of scope once the algorithm has finished. Thus, whenever you are done investigating an incomplete partial ladder, make sure to free up all heap-memory that was allocated for it before moving on with the algorithm. Furthermore, when you have found a complete ladder, make sure to free up all remaining heap-memory that was allocated for the algorithm before returning the ladder (do NOT free the memory for the complete ladder as you need to return the ladder to be displayed to screen, etc.). Lastly, in the case where no ladder is possible, the list of LadderNode structs should be empty and all heap-memory should be free'd naturally within the algorithm.

A final tip on the algorithm: it may be helpful to test your program on smaller dictionary files first to find bugs or issues related to your dictionary or to word searching. Thus, the *simple#.txt* files fit this bill nicely.

---

## Programming Tasks

First, read through the code in **main()** to get an understanding for the primary components of the program. Keep in mind that it includes MANY calls to functions that you need to write. As you peruse **main()** and run across function calls, you may want to find each of the function headers above to get acquainted with them.

Then, find the following twelve function headers:

- `Functions related to the words array:`
    - `countWordsOfLength()`
    - `buildWordArray()`

    - `findWord()`
    - `freeWords()`
- `Functions related to linked lists of WordNode structs, i.e. a word ladder:`
    - `insertWordAtFront()`
    - `getLadderHeight()`
    - `copyLadder()`
    - `freeLadder()`
- `Functions related to linked lists of LadderNode structs, i.e. a list of ladders:`
    - `insertLadderAtBack()`
    - `popLadderFromFront()`
    - `freeLadderList()`
- `A function to perform the minimum-height word ladder finding algorithm:`
    - `findShortestWordLadder()`

The comments in each function body of the starter code provide an explanation of what each function should accomplish, implicitly defining each parameter and returned quantity. Your task in this project is to complete the program by writing the twelve functions listed above, without modifying the struct definitions or the code in main().

Whereas, there is no structured student testing requirement for this project, you are expected to test the functions on your own before relying on the autograder. This should be done by writing your own test case functions and/or using the main() application outputs to compare expected vs. actual results. The executable **demo.exe** is a fully functioning program that is provided with the starter code to help with student testing (prior to submission to autograder). As always, you need to change the permissions to allow execution as follows:

```
>> chmod a+x demo.exe
>> ./demo.exe
```

There are many of good options for developing and using your own test cases; here are two:

- Write individual test cases in **main.c**, right below each function that you want to test. For example, right underneath **countWordsOfLength()**, write a Boolean test case function called **test_countWordsOfLength()**, to fully test the functionality of **countWordsOfLength()**. Then, write one additional master testing function that calls all of the **test_*()** functions and has nicely formatted print statements to communicate if test cases are passed or not. Finally, call this master testing function as the first line of **main()**, but ONLY if **testing mode** is ON, where **testing mode** can be turned ON by a command-line argument (defaulted to OFF). Note that the autograder does not use command-line arguments, so it will not run your test cases if you use this option.
- Copy/paste your **main.c** functions (in full) that you want to test into a test case file

inside a subdirectory where all testing is compiled and run; that is, the **main.c** functions are copied into **tests/test.c**, which also have a test case function (e.g. **test_countWordsOfLength()**) for each function (e.g. **countWordsOfLength()**) copied in, and also has its own **main()** that calls all of the **test_*()** functions. This subdirectory (**tests/**) also may have its own **makefile** and is where the **test.c** is compiled and the testing suite is executed. This option is nice since it keeps the primary main.c application clean, putting all testing in a separate location. The major downside to this option is having multiple copies of the same code, which needs to be managed carefully.

You will submit an explanation of how you tested your code. Include samples of running the program and/or test case code that you developed for testing purposes. In order to earn all of the points for testing, you do need to write and run test cases of your own. Script your explanation of testing in a word-processor and **save it as a .pdf**. Submit this **testing.pdf** with your Gradescope submission.

Lastly, you should develop a *makefile* with many useful targets. Here are some example targets:

- a target to compile the code
- a target to run the program interactively
- a target to run the program interactively under valgrind
- a target to run the program using the redirection operator to supply input values non-interactively (you will need to make a .txt file that contains preset user-inputs)
- additional targets you find useful for developing, testing, and running your program

The *makefile* will not be tested as part of the autograded test cases. Instead, you will submit the makefile with your code submission to Gradescope.

## Run the Application

With all functions written, tested, and passing the autograded test cases, run the program with various inputs using the full *dictionary.txt* to investigate the following questions:

1. Finding very short word ladders is easy, e.g. connecting **debug** and **debut** is trivial. However, your program is so good at finding short word ladders that it takes some effort to find word pairs where the shortest word ladder is actually long. What is the longest optimal word ladder you can find between 3-letter words? 4-letter words? 5-letter words? After some systematic attempts, your instructors were able to find (can you find word-pairs where the shortest word ladder is longer than these???)...
    - a 3-letter-word-pair with shortest word ladder of height 7,

    - a 4-letter-word-pair with shortest word ladder of height 9, and
    - a 5-letter-word-pair with shortest word ladder of height 14.
2. Describe your method for finding word pairs where the shortest word ladder is relatively long. Note: in the **setWords()** function of the provided starter code, after 5 invalid words are inputted, the program chooses a word at random so it can proceed. Can you use this feature to your advantage?
3. Based on your experience developing and running the program, briefly explain why word ladders tend to be shorter between smaller words (e.g. 3-letter word pairs) than longer words (e.g. 5-letter word pairs).
4. How does the word length affect the ease of being able to connect words with a word ladder? You may find that almost any 3-letter-word-pairs can be connected. Conversely, you may find it difficult to connect any 9-letter-word-pairs. At what word size do you begin to find it difficult to connect randomly chosen words (you may be surprised at how small this number is)? Explain this phenomenon based on your experience running the program.

*Note: even if you are not able to get a fully-functioning program, you can use **demo.exe** to answer these reflection prompts.*

Script your reflection in a word-processor and **save it as a .pdf**. Submit this **reflection.pdf** with your Gradescope submission.

## Optional Extension (Extra Credit)

After you have completed all required tasks and have built a fully functioning program, consider a modification to the problem: can you find the *longest* word ladder than connects the two input words?

Modify the algorithm to find the *longest* word ladder that connects the two words, while still avoiding duplicate words. Here, the avoidance of duplicate words prevents infinite loops only. In order to find the longest word ladder, it no longer works to prevent words from being entered into multiple ladders. Develop your modified algorithm in a new function called **findLongestWordLadder().** There are no autograded test cases for this function. Instead, display the long word ladder at the end of main, similarly to how the shortest word ladder is displayed.

This component is **optional** and only for **extra credit**. To receive the extra credit you MUST include a file titled **extension.pdf** with your submission, which fully explains how you modified the algorithm to find the *longest* word ladder and with instructions on how to run your extended program. The file must also include sample program outputs displaying some examples for longest possible word ladders.

## Requirements

- Use the starter code as provided, adding code to complete functions, without making any structural changes. Violations of this requirement will receive manually graded deduction(s).
    - Do NOT modify the **WordNode** and **LadderNode** struct definitions (no name change, do not add subitems, do not remove subitems, do not modify subitem definitions, etc.); one exception is that minor changes to the struct definitions are allowed if you prefer to alias the node pointer, e.g. **typedef struct struct_WordNode* WordNodePtr**.
    - Do NOT change the function headers (no name changes, do not add parameters, do not remove parameters, do not modify parameter types, etc.).
    - Whereas minor changes to **main()** may be helpful when developing, testing, and debugging, the primary application in **main()** should not be modified in your final submission, except for student-written testing (such as command-line argument for *TESTING MODE* to call a master test case function, or similar) and additional code related to the optional extra credit extensions.
    - Additional helper functions are allowed/encouraged when appropriate for proper functional decomposition.
- Solve each task and the program at large as intended. Violations of this requirement will receive manually graded deduction(s).
    - For example, when creating a word ladder linked list, each **WordNode** should simply point to a C-string element of the **words** array that is already allocated on the heap; i.e. do not allocate additional space for a copy of the word to point to.
    - Also, the search for **neighbor** words must NOT involve looping through the entire words array; instead, use a nested loop over the character index and the 26 letters of the alphabet and check if each possible neighbor word is a valid entry in the dictionary using a binary search.
- Student testing of the required functions and the program at large is required; however, there is no structural requirements for how the testing is done and it is not checked by the autograder. Instead, you will submit an explanation on how you continually tested your code as you developed functions. Include a separate test case file if you developed one for testing your program.
- The development of a useful makefile with meaningful targets is required; however, it is not checked by the autograder. Use past programming assignments that involved a makefile as a template. Your makefile should AT LEAST include targets for the following:

    - a target to compile the code
    - a target to run the program interactively
    - a target to run the program interactively under valgrind
    - a target to run the program using the redirection operator to supply input values non-interactively (you will need to make a .txt file that contains preset user-inputs)
    - additional targets you find useful for developing, testing, and running your program
- All dynamic heap-allocated memory must be freed to prevent possible memory leaks. This issue is checked by the autograder but may also receive a manually graded deduction.
- Coding style issues are manually graded using deductions, worth up to 25% of the total project score. Style points are graded for following the course standards and best practices laid out in the syllabus, including a header comments, meaningful identifier names, effective comments, code layout, functional decomposition, and appropriate data and control structures.
- Programming projects must be completed and submitted individually. Sharing of code between students in any fashion is not allowed. Use of any support outside of course-approved resources is not allowed, and is considered academic misconduct. Examples of what is allowed: referencing the zyBook, getting support from a TA, general discussions about concepts on piazza, asking questions during lecture, etc. Examples of what is NOT allowed: asking a friend or family member for help, using a "tutor" or posting/checking "tutoring" websites (e.g. Chegg), copy/pasting portions of the project description to an AI chatbot, etc. Check the syllabus for Academic Integrity policies. Violations of this requirement will receive a manually graded deduction, and may be reported to the Dean of Students office.

---

## Tips for Success on this Programming Project

- **Trickiest task/function is implementing a prescribed algorithm -** the project focuses on implementing a prescribed algorithm to extend an earlier project (that let the user build word ladders) to find the **shortest possible word ladder** using **linked lists** in a **C program.** The resulting program centers on an algorithm that finds the shortest word ladder that connects two English words. The algorithm involves organizing a linked list of partially complete word ladders, which are themselves linked lists of words. A full pseudo-code for the algorithm is provided in the description above. The trickiest part of this project is converting the pseudo-code to actual C-code, within the context of the primary application, **main()**, as written. Thus, it is important that you understand all of the steps of the primary application, the struct definitions and how they are used to construct various linked lists, AND all steps of the algorithm's pseudo-code prior to actually beginning to code.

- **NOT all tasks/functions are created equal -** the *Programming Tasks* for this project are primarily to develop many functions to support the primary application, most of which is provided for you in the starter code. The level of scaffolding is designed to lead you through the program development that leaves room for creatively applying course concepts and tools. A natural consequence is that..
    - some Tasks are straightforward and some Tasks are challenging;
    - some Tasks ask you to do one thing specifically and some Tasks are purposefully open-end with multiple components;
    - some Tasks test your ability to follow instructions verbatim and some Tasks require you to practice problem-solving and critical-thinking skills;
    - some Tasks may only take you a few minutes and some Tasks may require an initial attempt followed by a break to do something else only to come back multiple times to tackle the challenge;
- **Continual Testing -** you should continue building the best practice habit of continual testing functionality as you develop code. There is no formal required structure to the testing you do, but you must explain and demonstrate your testing process when you submit your project. Continual testing in small chunks of code is essential for efficient code development.
- **No print statements -** the functions you write should have no print statements. All of the console output is handled in the primary application and/or provided helper functions. Of course, as you develop, test, and debug your code, you may introduce some print statements to achieve those purposes. These print statements must be removed and/or commented out before you submit to the autograder.

---

## Submission & Grading

Develop your code in the IDE below. The provided starter code is substantial. Thus, make sure to read all instructions carefully and only make changes to the specific files referenced in the project description. Use the terminal to test your code interactively as you develop your program. Part of your tasks is to write a helpful and meaningful makefile for compiling and running your program. Once you have fully tested your code for a given task, use the "Submit for grading" button to test your code against the suite of autograded test cases.

When you are ready to formally submit, download the following files from the zyLab IDE file trees, and be prepared to upload the files to the **Project 05 Gradescope submission form**; look for *Project 05*:

- **main.c**
- **makefile** (not included in the starter code, but you are required to create one).
- Additionally, be prepared to describe your method of continually testing functionality as you developed your code, with supporting documentation where appropriate, and make sure your **testing.pdf** is ready to upload with the other files.
- Also, prepare your reflection responses to the four prompts listed in the *Run the Application* section above, and make sure your **reflection.pdf** is ready to upload with the other files.
- Lastly, if you extended the program to find the longest possible word ladders for extra credit, then you will also upload **extension.pdf**, which should thoroughly describe your extension.

Formally, you must submit your final program files to Gradescope, where your code will be manually inspected for style (details in the syllabus) and project requirements (details listed above). The official code submission is the version you submit to Gradescope (look for *Project05*). If you fail to submit to Gradescope, you have not formally submitted anything, and the resulting score is a zero for the project.

The standard deadline for the project is **Thursday, November, 7th**. As detailed in the syllabus, early submissions receive +1 extra credit point/day early, up to a maximum of +3 points. Also, students are allocated a certain number of late days throughout the term, where no late penalty is applied. Students can use their late days whenever they choose, simply by submitting their work to Gradescope after the deadline. However, no more than two late days are allowed for each project.

The grading breakdown is as follows:

- **90 points for functionality** - autograder points with manual deductions for any style issues (check syllabus for how style is graded) and program requirements not adhered to. Style is checked for the functions you write in main.c.
  - **6 points** - countWordsOfLength()
  - **6 points** - buildWordArray()
  - **9 points** - findWord()
  - **9 points** - insertWordAtFront()
  - **3 points** - getLadderHeight()

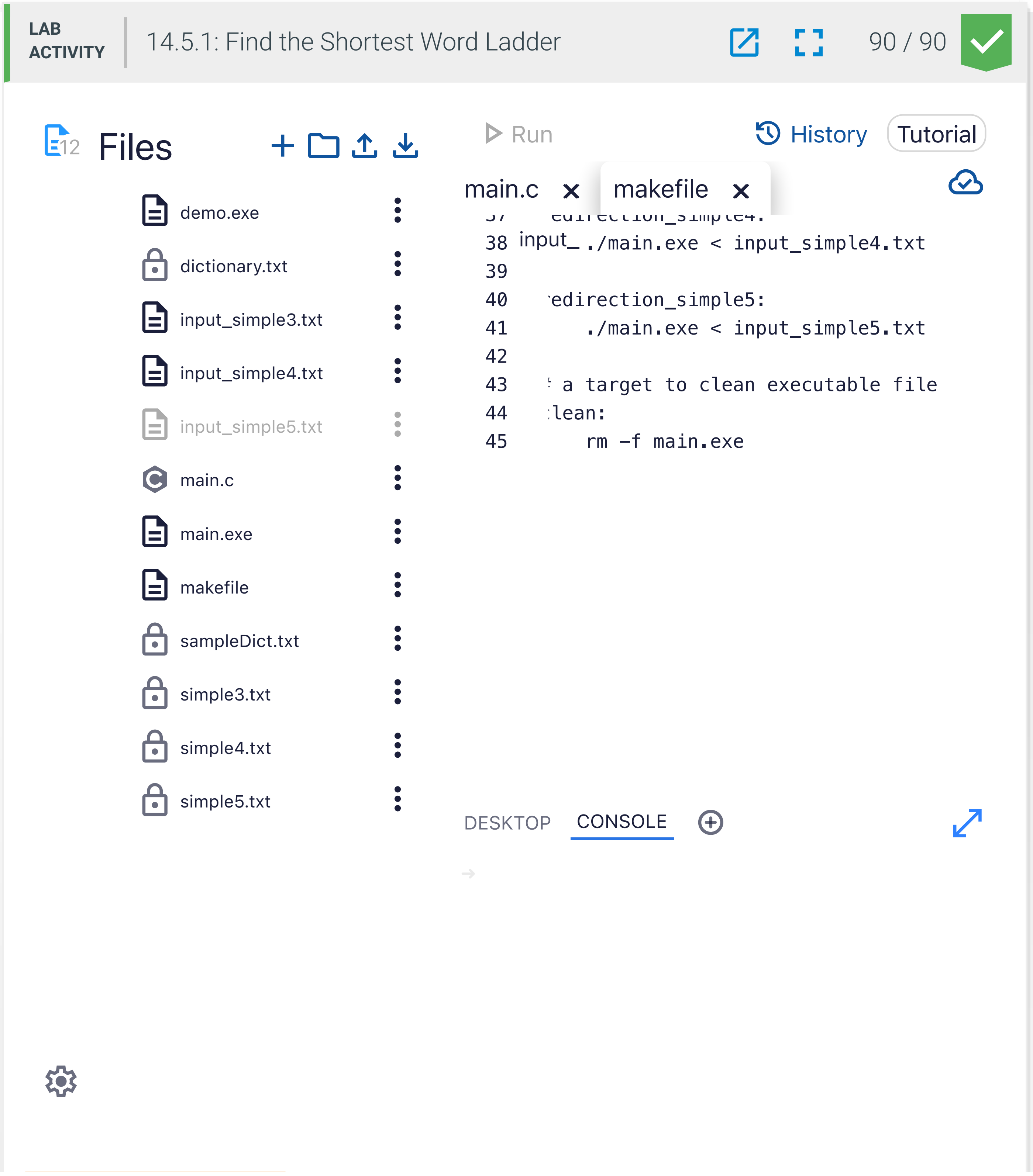
  - **9 points** - copyLadder()
  - **9 points** - insertLadderAtBack()
  - **9 points** - popLadderFromFront()
  - **9 points** - findShortestWordLadder()
  - **13 points** - program output checks on the shortest word ladder heigh for various word lengths and start/final word pairs
  - **8 points** - valgrind memory leak and error checks, which implicitly checks freeWords(), freeLadder(), and freeLadderList()
- **3 points for makefile** - a meaningful makefile with useful targets
- **3 points for student testing** - a full explanation/demonstration of how you tested functionality as you developed code, submitted as a pdf (see above for details)
- **4 points for free-response prompts** - from the *Run the Application* section, submitted as a pdf (see above for details)
- **extension extra credit** - up to +3 extra credit points for an innovative idea to find the longest word ladder to connect the word pairs. To receive any extra credit, up to a maximum of +3 extra credit points, the extension must be fully implemented and working correctly. Additionally, you MUST include a file titled **extension.pdf** with your submission, which presents your program extension to the grader with a full explanation of how the program works, how to run your extension, AND sample program outputs displaying some examples for longest possible word ladders.
- **early submission extra credit** - +1 point/day early with a maximum of +3 extra credit points.

---

## Citation/Inspiration

The idea for this programming project is greatly inspired by Chris Gregg at Stanford University, with much of the introduction and algorithm description directly borrowed.

## Copyright Statement



```
     edirection_simple4:
input_ ./main.exe < input_simple4.txt

    edirection_simple5:
        ./main.exe < input_simple5.txt

    a target to clean executable file
    lean:
        rm -f main.exe
```

Submit for grading

Coding trail of your work   What is this?

```
10/25 F0,21,26,35,35,35,35,44,44,53,62,62,62,62,62,53
 M0,0,62 T60,60,60,60,0,62,60,60,60,60,60,0,60,60,60
,60,60 F60,60,60,60,0,60,60,60,60,60,60,0,60,60,42,0
,0,0,60,60,42,51,60,51,51,60,60,60,51,70,62,70,62,62
,62,53,53,51,62,62,51,0,60,60,70,70,70,62,0,0,0,0,62
,51,62,60,66,66 S62,62,62,66,66,70,70,70,70,70,70 U70
,51,70,70,70,90,66,0,66,90,90 M90,86,90,90 min:450
```

Latest submission - 8:59 PM CST on 11/04/24

Submission passed all tests ✓

Total score: 90 / 90

✓ Only show failing tests *(40 tests hidden)*

Open submission's code

5 previous submissions

| | | |
|---|---|---|
| 8:57 PM on 11/4/24 | 90 / 90 | View |
| 8:56 PM on 11/4/24 | 86 / 90 | View |
| 1:42 PM on 11/4/24 | 90 / 90 | View |
| 1:28 PM on 11/3/24 | 90 / 90 | View |
| 1:26 PM on 11/3/24 | 90 / 90 | View |

Trouble with lab?

Feedback?

Activity summary for assignment: Programming Project 05 - Link Lint List - Find Shortest Word Ladder

90 / 90 points

Due: 11/07/2024, 11:59 PM CST

This assignment's due date has passed. Activity will still be recorded, but will not count towards this assignment (unless the due date is changed). See this article for more info.

**Completion details**