

CSCI-1200 Data Structures — Fall 2022

Homework 4 — Visual Difference Lists

In this assignment we will compute and visualize the differences between two text files. We will use this assignment as a chance to review and practice using the STL `list` container class, as well as Big 'O' Notation and Recursion. *Please carefully read the entire assignment before beginning your implementation.*

To get started, let's study the sequence of words in these two input files:

input_original.txt

```
the quick brown fox jumped over the lazy dogs
```

input_revised.txt

```
the quick fox jumps over the big lazy dogs
```

We can see 3 differences between the files: the word “brown” has been erased early in the sentence, the word “jumped” has been replaced by the word “jumps”, and finally the word “big” has been inserted near the end of the sentence. We will represent these differences in the following file format:

output_original_revised.diff

```
2 ERASE
4 REPLACE "jumps"
7 INSERT "big"
```

The operations above are necessary and sufficient to transform the *original* text into the *revised* text. Each operation has an integer which indicates the position in the *original* file where the operation is performed. Given the original file and the operations to transform original to revised, we can also prepare the inverse list of operations necessary to go the opposite way (transform the *revised* text to the *original* text):

output_revised_original.diff

```
2 INSERT "brown"
3 REPLACE "jumped"
6 ERASE
```

We can visualize the edits discussed above using HTML background highlighting and any modern web browser (Chrome, Firefox, Safari, Edge, etc.). The pink areas have been erased or replaced, and the green areas have been replaced or inserted. The grey regions add space to visually align the portions of the files that match.

```
the quick brown fox jumped over the lazy dogs
the quick fox jumps over the big lazy dogs
```

In the example above, the unit for comparison was *words* separated by whitespace. But we can also do the comparison *per character* as illustrated in the example below, using the first paragraph of the poem *Still I Rise* by Maya Angelou. These are the operations necessary to remove all of the punctuation and replace the uppercase letters with the lowercase versions:

```
You may write me down in history
With your bitter, twisted lies,
You may trod me in the very dirt
But still, like dust, I'll rise.

you may write me down in history
with your bitter, twisted lies
you may trod me in the very dirt
but still, like dust, i'll rise
```

output_still_i_rise_char.diff

```
0 REPLACE "y"
33 REPLACE "w"
49 ERASE
63 ERASE
65 REPLACE "y"
98 REPLACE "b"
107 ERASE
118 ERASE
120 REPLACE "i"
121 ERASE
129 ERASE
```

This visualization technique is very helpful in collaborative software development projects. When one member of the team proposes changes to the code base, their requested changes can be visualized for review and approval by the other team members using a visual difference. Here is a small example where we consider the input as *whole lines* instead of words or characters as in the above examples.

<pre>#include <iostream> int main() { std::cout << "Hello World!" << std::endl; return 0; }</pre>	<pre>#include <string> #include <iostream> int main(int argc, char* argv[]) { std::cout << "Hello World!" << std::endl; }</pre>
--	--

You can see more complex code difference visualizations on GitHub, <https://github.com/>, which hosts many open-source software development projects, including *Submittity*. Here's a example *pull request* from one of the Submittity developers which changes multiple files in the code base:

<https://github.com/Submittity/Submittity/pull/7128/files>

Interactive and Incremental Commands for Testing and Debugging

We provide a framework of code to read and write most of the different input, output, and visualization files for this homework, but you must write several key functions to complete the program. You may not modify any of the provided code. All of your work will be in two new files, `solution.h` and `solution.cpp`. Please study all of the provided code and the example input and output files posted on the course web site.

The program accepts commands interactively from `std::cin` (the keyboard), but typically we will run the program by *redirecting* a text file of commands on the command line. See also “Redirecting Input & Output” from http://www.cs.rpi.edu/academics/courses/fall22/csci1200/programming_information.php. Here is a example file of commands to visualize the first example in this handout:

requests.txt

```
compute_diff WORD input_original.txt input_revised.txt output_original_revised.diff
render_diff WORD input_original.txt output_original_revised.diff output_original_revised.html
```

And here is the same example, saving more of the intermediate steps for inspection and debugging:

requests_debugging.txt

```
compute_diff WORD input_original.txt input_revised.txt output_original_revised.diff
apply_diff WORD input_original.txt output_original_revised.diff output_applied.txt
assert_same WORD input_revised.txt output_applied.txt
invert_diff WORD input_original.txt output_original_revised.diff output_inverted.diff
apply_diff WORD input_revised.txt output_inverted.diff output_reapplied.txt
assert_same WORD input_original.txt output_reapplied.txt
compute_diff WORD input_revised.txt input_original.txt output_revised_original.diff
assert_same_diff WORD output_inverted.diff output_revised_original.diff
render_diff WORD input_original.txt output_original_revised.diff output_original_revised.html
```

The `main` function in `main.cpp` will parse these commands, read the necessary input files, call several functions you will implement, and write intermediate or final results to files (functions in `input_output.cpp` and `render.cpp`). You should make your own test cases for this assignment, and incrementally test each operation and inspect every output file. You must implement the missing functions, writing the function prototypes in a file named `solution.h` and implementing the body of the functions in `solution.cpp`.

This is how you will compile and run your program:

```
g++ main.cpp input_output.cpp render.cpp solution.cpp -Wall -Wextra -std=c++11 -o run.out
./run.out < requests.txt
./run.out < requests_debugging.txt
```

Study the expected output files for this first example posted on the course website.

Multiple Solutions: Using Recursion to Find the Minimum Edit Distance

The most complex operation is the `compute_diff` function. We recommend saving the implementation of this function for last. Why is this a complicated problem? There are *multiple valid solutions* – multiple different combinations of insert, erase, and replace operations that when applied to the first file result in the second file. In particular, the problem is non-trivial if two or more adjacent/neighbors chars/words/lines must be erased, inserted, or replaced – as shown in the example below. Here are the visualizations of five different, *valid* ways to edit the original file – all resulting in the same revised2 file:

input_original.txt

```
the quick brown fox jumped over the lazy dogs
```

input_revised2.txt

```
yesterday the foolish and debatably quick brown fox jumped over dogs
```

output_prioritize_erase.diff

```
0 ERASE
1 ERASE
2 ERASE
3 ERASE
4 ERASE
5 ERASE
6 ERASE
7 ERASE
8 ERASE
9 INSERT "yesterday"
9 INSERT "the"
9 INSERT "foolish"
9 INSERT "and"
9 INSERT "debatably"
9 INSERT "quick"
9 INSERT "brown"
9 INSERT "fox"
9 INSERT "jumped"
9 INSERT "over"
9 INSERT "dogs"
```

output_prioritize_insert.diff

```
0 INSERT "yesterday"
1 INSERT "foolish"
1 INSERT "and"
1 INSERT "debatably"
6 INSERT "dogs"
6 ERASE
7 ERASE
8 ERASE
```

output_prioritize_replace.diff

```
0 REPLACE "yesterday"
1 REPLACE "the"
2 REPLACE "foolish"
3 REPLACE "and"
4 REPLACE "debatably"
5 REPLACE "quick"
6 REPLACE "brown"
7 REPLACE "fox"
8 REPLACE "jumped"
9 INSERT "over"
9 INSERT "dogs"
```

output_default.diff

```
0 INSERT "yesterday"
1 REPLACE "foolish"
2 REPLACE "and"
3 REPLACE "debatably"
4 REPLACE "quick"
5 REPLACE "brown"
6 REPLACE "fox"
7 REPLACE "jumped"
8 REPLACE "jumped"
8 INSERT "over"
```

output_recursive.diff

```
0 INSERT "yesterday"
1 INSERT "foolish"
1 INSERT "and"
1 INSERT "debatably"
6 ERASE
7 ERASE
```

output_prioritize_erase.diff

output_prioritize_insert.diff

output_prioritize_replace.diff

output_default.diff

output_recursive.diff

When we have two or more adjacent elements (words, characters, or lines) that are different in our two files, we cannot easily determine which operation (erase, replace, or insert) to perform. We hope that the original and revised input files will line up and match again in the near future, but we can't be sure how far its necessary to "look ahead" in each file to find that alignment.

The sample diff files above were created with five different algorithms that (as their names suggest) prioritize the different operations. These are only a few of the many possible valid answers for this small problem. How do we decide which one is best? How do we find the best answer? Usually we decide that the *best or optimal solution* has the fewest total insert, erase, and replace operations (e.g., shortest length/number of lines in the `.diff` file). This is the *Minimum Edit Distance* problem – a dynamic programming problem that is often covered in Intro to Algorithm courses. **IMPORTANT NOTE: For this homework, we are not asking you for an efficient, optimal solution to every input, which is beyond the scope of this course!** You should not search for, read in detail, or use any code available online to solve this problem. Once the homework deadline has passed, if you are curious, you are welcome to do your own research and learn more about this topic and this famous problem.

For *EXTRA CREDIT* you can compute the optimal solution for small problems by writing an exhaustive recursive search. Caution: This search is very expensive! As you test your recursive function you will appreciate how quickly this search becomes impractical for larger examples. Summarize your testing and your Big O Notation analysis in your `README.txt`.

Since finding the optimal solution is rather expensive, your *DEFAULT* (non-extra-credit) algorithm for `compute_diff` should efficiently find a good, (near) minimum edit distance solution for inputs that *do not require editing two or more adjacent/neighborng chars/words/lines*. The rest of the code should correctly work with and visualize `.diff` files produced from any `compute_diff` algorithm, as shown above and available on the course website. You do *not* need to implement multiple versions/algorithms for the `compute_diff` operation, but you are welcome to experiment with different algorithms if you decide to tackle the extra credit.

Additional Assignment Requirements and Suggestions

- **You may not use vectors or arrays or subscript [] for this assignment.** Use STL lists instead. You may not use maps, or sets, or things we haven't seen in class yet. Be sure to use `const` and `pass/return by reference` where appropriate (refer to the diagram in Lecture 4).
- You should practice using a traditional debugger, e.g., `gdb/lldb` with this assignment. Set breakpoints, walk line-by-line through loops or function calls, print variables, examine stack frames, etc. Even though you will not be directly calling `new` or `delete` yourself, you may find a memory debugger, e.g., `drmemory` or `valgrind` helpful if you get stuck with a segmentation fault or other memory error.
- In your `README.txt` file, provide a Big O Notation complexity analysis of each operation/function in the program assuming w words in the first input file, and requiring i inserts, e erases, and r replace operations. Write a few sentences justifying each of your answers.
- Be sure to make up new test cases to fully test your program. Use the template `README.txt` to list your collaborators, your time spent on the assignment, and any notes you want the grader to read.