Project 2 – Fuzzy Search

Overview: In this project, you will be creating code to search a document for a word/phrase that may contain a typographical error.

Goal: To gain experience with string matching algorithms.

Requirements: Your program should prompt the user for the name of a file to search and a phrase to search for. Your program should search for the original phrase in the contents of the file. If it finds the phrase, then it should display the first index at which it found the phrase as well as that it found an exact match.

If the original phrase is not found, try searching for all one-letter deviations from the original phrase and display **all** of them that are found – both their index and the phrase in context. One letter deviations mean that one letter (and only one letter) of the original phrase was omitted, one letter was replaced with any other possible character, one letter swapped with an adjacent letter, or one letter was added at any position in the original phrase. Since you don't have a fixed alphabet for your searches, you may want to consider how to represent the concept of a wildcard character in your code. Your code should display the index as well as the approximate match for all of the approximate matches that are found. After the list of approximate matches, your code should display how many approximate matches there were.

Algorithm: In addition to your code, you need to turn in a well-formatted document containing the pseudo-code for your core algorithm (written in the same format as the textbook uses for pseudo-code) along with any proofs you decide to include about your algorithm – see grading section below.

Grading: Your algorithm will be graded according to the rubric below.

	1	2	3	4	5
Matching	Finds exact match if exists	Finds some approximate matches – most likely deletions and swaps	Finds all approximate matches – including wildcard replacements and additions		
Time	Code always terminates in finite time.	Runtime of code no worse than brute force.	Runtime at least as good as benchmark code – sample runtimes to be provided.	Runtime substantially better than benchmark code.	Best time performance of all student submissions.
Efficiency Analysis	Submits accurate	Submits accurate	Has pseudo-code and		

	pseudo-code of algorithm or implements code to count basic operations and show runtime.	pseudo-code of algorithm and implements code to count basic operations and show runtime.	timing and counting code. Defines input size and basic operation for domain. Formally proves that algorithm belongs to Θ(g(n)).		

Your project will receive a final score based on the number of points from the above rubric that it accrues. The table below summarizes the minimum number of points from the rubric that would be necessary to earn various grades. Extra credit is possible.

Grade:	50	47.5	45	43.5	42	40	37.5	35
Points on Rubric:	10	9	8	7	6	5	4	3

Runtime Guides: The following benchmarks can be used in determining whether your code is matching the performance of brute force and the benchmark algorithm. The benchmark times were based on running the code on one of the lab computers in CTC 114.

File	Search Phrase	Exact Match or # of Approx Matches	Brute Force runtime (in msecs)	Benchmark runtime (in msecs)
small.txt	arra	Exact	0.0156	0.0000
small.txt	ware	10	0.1563	0.1563
small.txt	xe	44	0.1094	0.1093
dictionary.txt	zenith	Exact	49.112	25.898
dictionary.txt	etnse	85	1118.45	554.067
shakespeare.txt	O Romeo, Romeo! wherefore art thou Romeo?	Exact	762.498	95.966
shakespeare.txt	to be or not	1	44710	12738
code.cpp	Board::	Exact	1.1873	0.4531
code.cpp	Board:isFull()	1	73.728	17.172
webpage.html	<a href="/wiki/File:Euclid%27s_algorithm_structured_blocks_1.png"	Exact	38.992	3.1094

Approximate Matches: Depending on your search approach, you may find that you're getting different numbers of matches. My implementation leaves in newline characters – if you strip them out, you may find additional matches. My implementation also removes duplicate matches – for example, if searching for "hello" and my code found an approximate match for "helo" by deleting the third letter, it wouldn't also flag the same "helo" as a match for the search term with the fourth letter deleted.

Sample Approximate Match Outputs:

Searching small.txt for "ware":

```
4 deletions: [5-7: are], [19-21: are], [35-37: are], [43-45: are]
0 transpositions
6 substitutions: [4-7: hare], [18-21: rare], [34-37: dare], [42-45: eare],
[8-11: were], [12-15: wore]
0 insertions
```

Searching small.txt for "xe":

```
22 deletions: [1: e], [3: e], [7: e], [9: e], [11: e], [15: e], [17: e], [21:
e], [23: e], [29: e], [31: e], [37: e], [38: e], [42: e], [45: e], [48: e],
[51: e], [54: e], [56: e], [58: e], [59: e], [62: e]
0 transpositions
22 substitutions: [0-1: he], [2-3: re], [6-7: re], [8-9: we], [10-11: re],
[14-15: re], [16-17: he], [20-21: re], [22-23: te], [28-29: re], [30-31: de],
[36-37: re], [37-38: ee], [41-42: re], [44-45: re], [47-48: re], [50-51: he],
[53-54: se], [55-56: re], [57-58: se], [58-59: ee], [61-62: se]
0 insertions
```

Submission Instructions: Please submit a .zip or .7z file containing the code for your project, the files you tested on as well as your document containing the algorithm pseudo-code and any proofs.