# Exercise 1

The goal of this exercise is to implement functions used in measuring the distance between two strings, modifying them so common mistakes have less impact, implementing a method of suggesting the correct word, creating a program that takes a file as an input and corrects it.

Functions used in measuring the distance between words are:

**levenshtein\_distance** which calculates distance between two words. It returns a number of deletions, insertions and replacements. It creates a matrix of size *m* times *n* where *m* and *n* are sizes of 2 words, first column and first row are filled with ranges of values starting at 0 and ending at *m* or *n*. Each column values are then filled as in formula:

Text

Description automatically generated

where i and j are row and column and char\_1 and char\_2 are characters at positions i and j in two strings. The final result is in the last row and column.



**hamming\_distance** which calculates replacements in two strings, therefore it just iters over two strings at the same time and counts the characters that do not match. This method requires the two strings to have the same length.

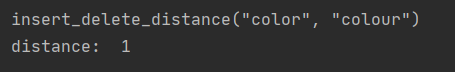


**insert\_delete\_distance** which, similarly as in the levenshtein distance, calculates the distance of two strings as a count of insertions and deletions, but not the replacements. The formula for this method is:

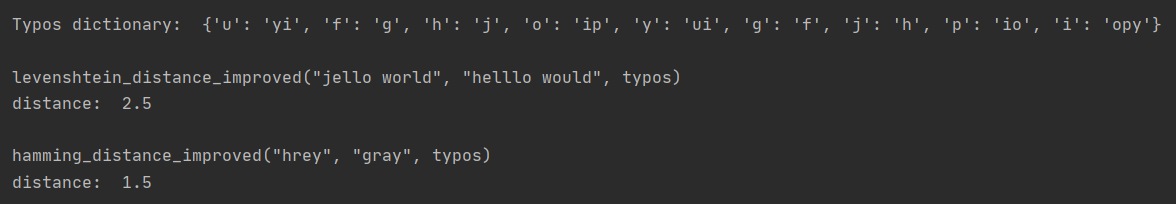
Text

Description automatically generated

And the final result is the absolute value of sum of lengths of two strings minus the two times the value in the last column and row.



Improving the levenshtein and hamming distances is done by checking if replaced character is in a typos dictionary and incrementing by 0.5 instead of 1 as in **levenshtein\_distance\_improved** and **hamming\_distance\_improved**.



Suggesting a correct word is done with a function **find\_similar\_word** which checks if a word is in a list of correct words. If it is not inside, it checks with the **levenshtein\_distance\_improved** every correct word and saves the one with the smallest distance. The list with correct words is read in the next function.

Function **correct\_text** reads from file **text\_raw.txt** and splits the words. Then the **find\_similar\_word** checks every word for correctness. Corrected text is saved in file **words\_corrected.txt**.

**text\_raw.txt**

*somrbody once told me the wotld is gonna rol me i aint the hsarpest toool in the sched she was looking kind of dumb with a finger and a thumb in the hsrape of an l on her foreheadf*

**words\_corrected.txt**

*somebody once told me the woald is gonna bol me i aint the hoariest tool in the ached she was looking kind of dumb with a finger and a thumb in the ascape of an l on her forehead*

# Exercise 2

The goal of this exercise is to implement a priority queue starting with an array and creating Huffman code for a given text.

Priority queue is implemented in a class **PriorityQueue**. Values stored inside the array are in the form of a tuple with character and value. This class has functions:   
**insert** used for inserting a value in the array and correcting the priority of values,   
**heapify** which recursively corrects the priority of values below a certain index,   
**build\_heap** which runs **heapify** for indexes from length/2 - 1 to 0,   
**extract\_min** which pops the first value in the array (the smallest one) and heapifies the array again and also functions **parent, left\_child** and **right\_child** for calculating the indexes.

Huffman code is computed with a class HuffmanCode. It has functions:  
**\_read\_character\_frequency** used for counting the characters and their occurrences in a text, **\_get\_tree** which returns the Huffman tree by taking the two least-valued characters in the heap, joining their values and inserting back to the heap as long as the length of a priority queue is not 1,  
**\_get\_code** which creates codes for characters from the tree and  
**encode\_text** which encodes the text according to the computed codes.

Ratio of compression is calculated as length of a text times 8 bits divided by the length of an encoded text.

Examples:

Text

Description automatically generated

