Baseline Benchmark

```
100 cut paste operations took 6.489900000000715e-05 s
100 copy paste operations took 5.094199999998266e-05 s
100 text retrieval operations took 1.7796999999930563e-05 s
100 mispelling operations took 5.8294999999986e-05 s
```

Optimized Benchmark

```
Evaluating case: hello friends
100 cut paste operations took 0.00012612000000000734 s
100 copy paste operations took 8.20890000000074e-05 s
100 text retrieval operations took 3.683899999995521e-05 s
100 mispelling operations took 0.0003892949999999562 s
```

High Level Overview

My high level approach to this problem was to figure out which methods took the longest and search the code for any inefficiencies. The slowest methods were cutting and pasting. However, after looking through the code, I realized that python slicing is highly optimized, so there were no better in-place methods. I then realized that Python has dynamic resizing strategies: as long as the number of characters you are deleting / adding is less than `size(document) // 2`, then removing / adding a character in-place is constant time. Otherwise, the underlying code will copy over the elements into a larger array. Thus, I decided to keep the document as an array of characters rather than a string (as strings are immutable). For cutting, I would go through and manually delete the indexes of characters needed to be cut and for pasting I would insert characters into the array.

How to Run

To open an interactive python session, run `python -i editory.py`.

```
editor = SimpleEditor("hello friendsmy")
s.copy(13, 15)
s.paste(6)
s.undo()
s.redo()
```

To simply run the tests and benchmark, run `python editory.py`.

New Features

I then moved on to finding new features I could implement. Since QwickType developers spend so much time copying and pasting code, they must make mistakes sometimes. Thus, I decided to implement redo / undo functionality. I used a stack because when we want to undo, we want to undo the most recent change. Thus a LIFO data structure would be necessary. Specifically, I used Python’s deque class which has constant pop and append time. I append to the stack whenever text is pasted or cut, storing the type of operation as well as any arguments necessary to undo it. Whenever the undo function is called, I simply pop off from the stack and perform the necessary changes to the document. I repeated a similar procedure for redo.

Runtime Changes

By choosing to edit the document in place rather than use slicing, I save on time for the cut/paste operations. Normally, what you cut / paste is much smaller than the entire document size. Thus, it is a waste to copy over the entire document every time you make a change. In-place cutting / pasting will run in time proportional to amount cut / pasted rather than the entire document. Although the new benchmark shows slower times for both of these methods, I have shown that their theoretical runtime is faster after my optimization. For larger and larger documents, this will be more apparent. However, this optimization slows down the text retrieval method as I must join the array of characters into a string, which takes linear time. Furthermore, the runtime of the redo / undo operations are the same as the runtimes of copy / cut because redoing or undoing reduces to copying or cutting plus constant time stack operations. Although I added two stacks, they never grow past a constant size. Thus, the space complexity of the original text-editor is maintained.

If I Had More Time

I would improve the misspellings method. Each time you call it, you are searching the entire document over and over again. However, this is very inefficient if you are constantly cutting / pasting. A better way to do this would be to keep a variable for the number of misspellings within the document. Every time you update the document, update the misspellings variable. You won’t need to search the entire document because each cut / paste would only possibly change words directly adjacent to the cut / pasted text.