Undo, Redo, and You!

You've recently been hired by a product company that has its sights on revolutionizing the way the world looks at text editing. While their ideas are solid, their codebase is "alpha" quality, and needs some love before anyone can take it seriously. That's where you come in.

One thing you noticed right away is that whoever wrote the original code must have never tested against even remotely large documents. As the document size increases, the memory footprint starts to skyrocket. A simple benchmarking script makes this obvious:

```
$ ruby memory_test.rb
Starting Memory:
 total           44136K
Adding 100 characters, 1 at a time
Current memory footprint:
 total           44136K
Adding 1000 characters, 1 at a time
Current memory footprint:
 total           44136K
Adding 10000 characters, 1 at a time
Current memory footprint:
 total          103056K
Adding 100000 characters, 1 at a time
Current memory footprint:
 total         6084152K
Took 11.744874s to run
```

Before you lay down more than a couple hundred thousand characters, the app has already consumed 6gb of memory! While you would expect the footprint to increase as you insert more text into the document, there is no good reason why things should explode like this. A quick glance at the source code reveals the true issue. Can you spot it?

Inefficient TextEditor::Document implementation

```
module TextEditor
  class Document
    def initialize
      @contents = ""
      @snapshots = []
      @reverted  = []
    end

    attr_reader :contents

    def add_text(text, position=-1)
      snapshot(true)
      contents.insert(position, text)
    end

    def remove_text(first=0, last=contents.length)
      snapshot(true)
      contents.slice!(first...last)
    end

    def snapshot(tainted=false)
      @reverted = [] if tainted
      @snapshots << @contents.dup
    end

    def undo
      return if @snapshots.empty?

      @reverted << @contents
      @contents = @snapshots.pop
    end

    def redo
      return if @reverted.empty?

      snapshot
      @contents = @reverted.pop
    end

  end
end
```

The Culprit

Buried down in the snapshot definition, we see this suspicious line of code:

```
@snapshots << @contents.dup
```

Although snapshot() seems elegant in its simplicity, it has a critical flaw. Our benchmarks prove that copying the entire contents of the document on every single change is a fairly insane thing to do whenever you aren't dealing with tiny data sets. We will need take a different approach if we want to solve this problem correctly.

Resources

You have the following resources at your disposal:

- A linux based benchmark script for testing memory consumption
- The source for the original Document class
- Tests that cover the basics of how the features should work

While you may need to tweak the benchmark script to run in your environment, or write a new one entirely, it gives you something to start with. The included tests were written by the original developer, and aren't quite comprehensive, but cover the basic functionality your new code will need to support.

Your Task

This exam is meant to serve a broad range of skillsets simultaneously, so you have a choice of how you want to answer this problem. Pick any one of the three challenges listed below, aiming for the one that suits your comfort level the best. Try not to spend more than a couple hours on the problem, it's not meant to be a major exercise, and your response doesn't need to be perfect.

LEVEL 1: Redesign and re-implement the following four features for the `Document` class: `add_text()`, `remove_text()`, `undo()`, and `redo()`. Comment on the reasoning behind your changes, and what you can expect to see in terms of memory and performance gains. Show benchmark outputs to prove your results.

LEVEL 2: Identify an approach to redesigning the `Document` class so that it is more efficient. Describe the changes that would need to be made to the existing code, and what the overall structure would look like. Even if you can't fully implement the idea yourself, do the best you can to get it started, sketching out your ideas in Ruby code and leaving some blanks to be filled in as needed. Submit any open questions you have as comments in your code.

LEVEL 3: Do a bit of searching around the web for discussions about implementing undo/redo functionality. Come up with a good list of questions that you think will help you learn what you need to know to solve this problem. Submit a list of the materials you found, along with your list of questions. If you experiment with writing any code, share that as well.

No matter which level you choose, try your best to get as far into the problem as you can. Clearly documenting the areas where you stumbled a bit will go a long way towards helping me put you in the right session.