string-algorithms handin 2 - Tandem repeats

Marc Skodborg, 201206073
Stefan Niemann Madsen, 201206043
Simon Fischer, 201206049

May 8, 2016

1 Status

Seems to work, we have tested our implementation on the provided examples and a comparison of outputs showed no discrepancies. Run our code using the following command: `python3 tandem-repeat.py mississippi.txt`

2 Insights

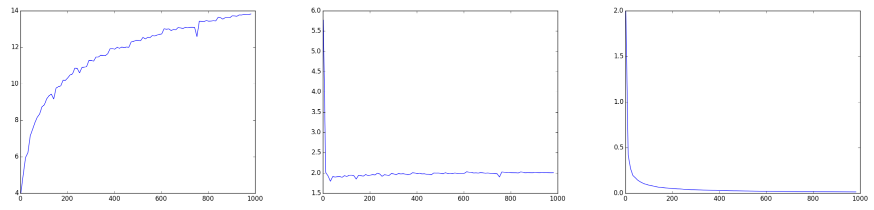
For profiling the code to verify running time, simple seems the way to go. We were determined to use pythons own profiling tools, namely `cProfile`, which ended up being a waste of time as the results we seemed to get was not as expected. We believe the differences in results is primarily due to our unfamiliarity with the tool and misinterpretations of the output, rather than issues with the actual implementation.

3 Problems encountered

See above issues with the profiling

4 Experiment

To verify that our implementation ran in time $O(n \log n)$, we simply added a variable and incremented it once per loop iteration in loops in the algorithm. We did this instead of timing all our code, as the execution times are affected by many uncontrollable, external variables, such as the garbage collector, OS thread scheduling and more. We found counting instructions to be a more reliable way of stating something useful about the time complexity of our implementation. This works, as we generally do not care about constants when discussing time complexity, and most of the code inside the loops is considered evaluated in constant time, i.e. *O(1)*. We did, however, leave some of the loops out of the



(a) $\mathcal{O}(n)$ (b) $\mathcal{O}(n \log n)$ (c) $\mathcal{O}(n^2)$

Figure 1: Plots of number of loops performed in relation to various time complexities as n grows

counting, even though they are never evaluated in constant time. This was due to the fact that they would always iterate through at most the same elements as another possibly bigger loop. As a result, the smaller loop would only influence the final time complexity by a constant factor, which we ignore here.

Once we had modified the implementation for supporting simple profiling, we simply picked one of the supplied input files, in this case the `fib15.txt` file. The file is 988 characters long, so we split it up into 98 cases, each containing 10 characters more from the file than the previous. We then constructed the tree and counted the instructions as we searched the tree for tandem repeats using our implementation. This gave us 98 points of input length and corresponding instructions used to process this input.

Before plotting them, we decided to let us inspire from [1] and plot three separate graphs. These graphs can be seen in Figure 1. What we did was divide the actual number of instructions by the value a function which we guessed would have had given input `n`. As an example, in Figure 1a we see the results of guessing that the performance of our implementation could be described by a linear function of the input size, i.e. *O(n)*. In this graph, the `x`-axis is the values `n`, and the `y`-axis is then `m / n`, where `m` is the actual number of instructions. As the figure obviously shows a growing function, the fraction on the `y`-axis must be growing as `n` becomes larger, i.e. the `m` is growing faster than our guess of `n`, which indicates that our guess was too optimistic. Likewise, the Figure 1c is converging to 0, indicating that the denominator is the dominant part of that fraction, i.e. we were too pessimistic. On Figure 1b, on the other hand, we observe a constant function with a value just around 2. This tells us that the ratio between the numerator and the denominator is 2, i.e. the value of dividing our actual performance measurements with the function $n \log n$ gives us a constant value of 2, regardless of the size of n. This must mean that our performance function is, approximately, equal to $2n \log n \in O(n \log n)$ which verifies our expectations.

5 References

[1] `http://stackoverflow.com/a/3983413`