Robert DeLaus
Yianni Giannaris
October 2, 2018
6.172 Project 1
Final Writeup

In our beta implementation, the algorithm we used was the reversal method described in the project handout where the rotation of the string *ab* can be performed using the identity $ba=(a^R b^R)^R$. This algorithm seemed like a good choice to us as it did not require too many operations and used constant storage. The bulk of the work for this algorithm is found in the reversals, so we set out to optimize a reversal function. Our first implementation of the reverse function was fairly simple, swapping bits one at a time. After spending a great deal of time trying to improve this method by swapping bytes or larger chunks at a time, we were unable to produce a bug-free implementation and had to revert back to our simple approach for the beta submission.

After taking a few days to talk with other classmates, review other beta submissions and think about the problem, we decided to use an entirely different algorithm for our final implementation. While we believe it would be possible to optimize the reversal method to make it very high performing, we decided to use a simpler algorithm that we had naively discounted because we thought would be too slow. The algorithm we used to rotate string *ab* at the highest level simply copies *a* into a temporary memory, moves *b* to *a'*s original position, then moves *a* from the temporary string into its final position. To make the implementation of this algorithm efficient, we would only move substrings in 64-bit aligned chunks. Because *a* and *b* are not necessarily 64-bit aligned, we are not always able to move all of *a* and all of *b*. To handle this, we first copy the 128 bits at the start and 128 bits at the end of both *a* and *b* and set them aside so that they can be put back into place at the end. This works because neither *a* or *b* will be shifted more then 63 bits, and neither *a* or *b* will be more then 63 bits away from the ends of the substring we are rotating.

We wrote a python script to generate thousands of test rotations, so with our final implementation passing all of those tests, we are fairly confident in the correctness of our code. By the time we passed all of our test cases and saw that we were completing tier 47 we did not have too much more time to work on our code before the deadline. So even though we recognize a few more simple optimizations that could be made, we did not take the time to implement them. We also realize that we left unused commented out sections of code in our final submission and are missing some documentation.

We learned several valuable pointers from out MITPOSSE meetings that will be useful in future 6.172 projects. One cool idea we learned was to add #ifdef DEBUG #enddef blocks into helper methods that compare the output of the helper method to the output of an implementation that we know is correct (for example testing a bit-hacky modulo function with the built in % modulo operator). If the output is not correct, we can assert False or print a statement. This would allow us to use our end to end tests to easily find bugs in helper methods. This is much easier then writing separate tests for helper methods or tracing a failed test case to the helper method. The beauty about this method is that if we remove the #define DEBUG line, then none of the helper method tests get compiled.