Project 1 Final Write Up

subella and timkralj

October 2018

1 Final Implementation

Our final version utilizes the triple reversal ($(A^r B^r)^r$) strategy for reversing arrays. This means we are reversing A, then reversing B, and then reversing the entire array again to get the correct solution. To do this, we have two main parts to our code. The first is in *bitarray_rotate* which takes in the original bitarray and splits it into 3 different calls to *bitarray_reverse*. We first find the "pivot" point of the structure which we defined as the location where A and B meet. We find the pivot by subtracting the rotation amount from the end of the bitarray if positive rotation or adding to the beginning if negative rotation. After this we make the 3 calls to *bitarray_reverse* from the beginning of the array to the pivot point ($subarrayA$), then call the same function from pivot point to the end of the bitarray ($subarrayB$), finally we call the function again from the beginning of the bitarray to the end. This returns $(A^r B^r)^r$ which equals the original array rotated by the appropriate amount.

Inside of *bitarray_reverse*, there are different branches depending on the input of the array we are rotating. If the array is small ($\leq$ 128 bits), we just reverse the bits using *bitarray_reverse_bits* function which swaps the first and last bit and moves in until the whole array has been reversed. If the array is large, we instead reverse entire memory addresses at once and then swap. *bitarray_reverse_location* reverses the bits inside the first and last fully filled memory addresses of the array, and then swaps the contents of each location. The process is repeated for the second and second to last addresses and so on. The bits in each address are reversed using an 8-bit look-up table and bit shifts. The look-up table returns the reverse of a byte input. The look-up table greatly speeds up the reversals since there are only 8 calls to the table instead of swapping 64 bits for every location.

bitarray_reverse_location only reverses fully filled address, so "tails" need to be handled separately. "Tails" are the bits at the beginning and end of the array that may only partially fill an address. First, we identify each tail's length. Then we reverse the bits in the tails using our rudimentary *bitarray_reverse_bits* function. After this, 3 cases are handled separately: the tails are the same length, the left tail is longer than right, or the right tail is longer than the left. The

first case is the easiest, all we do is save the left tail in a temp variable, place the right tail in the left location and then put the temp into the right tail spot. Now the array has been fully reversed.

The other cases are more difficult. When swapping the tails, space needs to be made for the larger tail. The entire array is shifted in the appropriate direction by the difference of the tail lengths. An example procedure for the case where the left tail is longer is shown below:

- Save the right tail bits
- Replace right tail with zeros
- OR the left tail bits that fit in the right tail into the right tail
- Save the extra bits of the left tail that didn't fit into the right tail (since it was longer)
- Replace left tail with zeros
- Insert right tail to the leftmost side of the left tail space
- Shift all the bits over all locations left extra bits length, filling in the newly made holes with the bits shifted out from the previous location.
- Append the extra left tail bits to the end of the last full block

After this, we have reversed the bitarray from the start to the end location. Once we have done this for A, B and AB we are done with the rotation of the bitarray.

2 Changes from Beta

All of the changes above were made for our final. We had all these ideas during the beta but were unable to implement them in time for the beta release. As a summary, we added: location swapping, look-up table for 8byte reversals, tail swapping, tail reversal, and equal and uneven length tail swapping.

3 Completeness, Performance, Bugs

Compared to our naive beta implementation, our final implementation has much more uncertainty with regards to correctness. There are now several different cases that are ran depending on the input array's type. For example, separate logic is handled for when the array is less than 128 bits, the array has equal length tails, or the array has uneven length tails. Because of this added complexity, we had to be much more thorough and calculated when constructing

our test suite. Our completed test suite tests every branch possible in our code, and also tests conditions that we think other teams may not be handling well. The final implementation passes our entire test suite, so we have good reason to believe our implementation has high correctness.

Our final implementation reached a high tier 44. While this fell short of the 47 goal to achieve full credit, we felt too much of our design would have to be restructured. We had depended on using a larger look-up table to make reversals faster, but unfortunately found that caching effects prevented this from being a viable strategy.

While still tedious, bug testing became a much more manageable process through the use of GDB. We debugged test by test, until our entire test suite was passing. It should be noted that at one point we had used a combination of both a 10-bit and 8-bit look-up table for faster reversals. This was slightly faster than using just an 8-bit table, but introduced a correctness bug which we were not able to solve in time.

4 MITPOSSE Comments

Meeting with MITPOSSE provided valuable insight on code style choices that we had overlooked. Before our meeting, our code was largely uncommented, which made it difficult for outside viewers to understand. We made sure to include substantial comments directly above important lines explaining their purpose. Additional comments were included above each function declaration explaining the function's purpose.

Initially, our main function was rather large and handled all logic inline. By implementing several helper functions, we were able to reduce clutter in function bodies. Helper functions also helped the debugging process by making it easier to isolate certain processes in our code.

More subtle changes were also brought to our attention in the meeting. Longer lines of code were split into multiple lines and indented to align similar variable types. Lines performing similar operations used the same ordering of variables to provide more clarity.