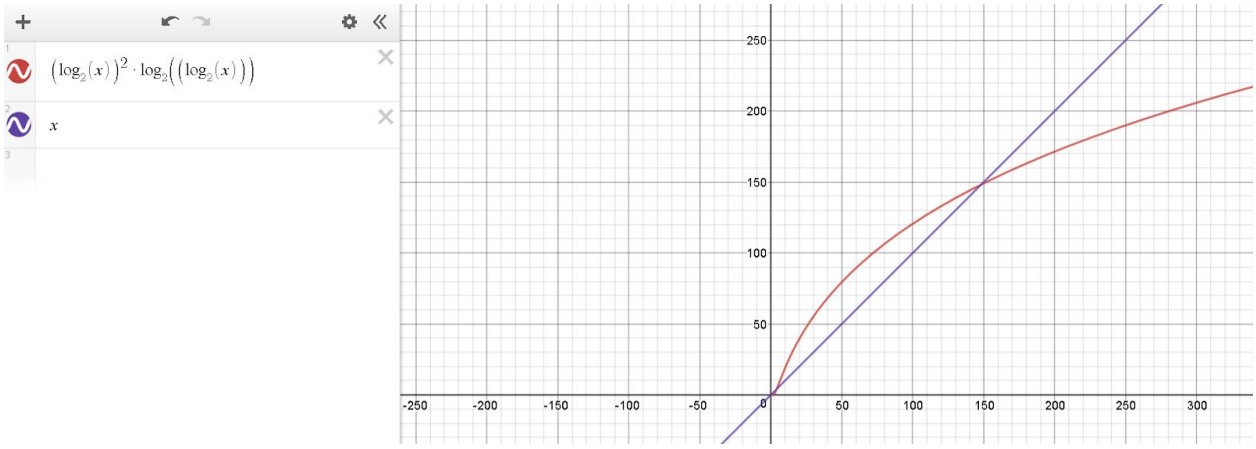
Zack Campbell: zcc254
Paul Heath: pah2276

Homework 3

1.
    a. The first history is sequentially consistent and linearizable. A history exists such that real-time order is preserved and all operations are legal:
    write(x, 1) -> read(x, 1) -> write(x, 2) -> read(x, 2)
    b. The second history is sequentially consistent and linearizable. A history exists such that real-time order is preserved and all operations are legal:
    write(x, 2) -> write(x, 1) -> read(x, 1) -> read(x, 1)
    c. The third history is sequentially consistent but not linearizable. A sequential history exists such that there is process order and legality:
    write(x, 2) -> write(x, 1) -> read(x, 1) -> read(x, 1)
    But by the ordering, you must write(x, 2) after write(x, 1) but before you read(x,1) so there is no way you can read(x,1) without breaking real-time order
2.
    a. This output is sequentially consistent because P2 can set “b” to 1 before P1 sets “a” to 1 and after P3 sets “c” to 1. Then P1 can set “a” to 1 before P2 and P3 finish printing and the history will preserve process order and legality
    b. This output is not sequentially consistent because in order for P1 to output “00” the full process of P1 must complete before the other processes start. Suppose this is the case. Then “a” is set to 1 and “b” and “c” are set to 0. Then P3 completes it’s first operation, setting “c” to 1. Then P2 can complete legally because both “a” and “c” are set to 1, outputting “11”. Then P3 attempts to complete it’s operations (printing “a” and “b”) but “a” is set to 1 which violates the assumption that the output is “01” for P3. Therefore, the outputs are not sequentially consistent.
3. You can use any algorithm that computes the max with distinct elements on an array without distinct elements by first condensing the array. By reducing the array by combining like elements, you will create an array with only distinct elements on which you can use any algorithm.
4. Assume the algorithm is run in place. Split the input array into log(n) pieces. Run the non-optimal parallel prefix sum algorithm on each of these pieces. These pieces can be processed in parallel, so the time complexity for processing these pieces is O(log(log(n))). Each of these pieces requires O(log(n) * log(log(n))) work. Thus, the total work for processing all of the pieces in parallel requires O(log(n) * log(n) * log(log(n))). After processing all of the pieces with the non-optimal parallel prefix sum algorithm, add the right most element in each piece to all of the elements in the adjacent piece to the right. Do this this process in order starting with the left-most piece. This second process is similar to the sequential algorithm for prefix sum. Each piece can be added to another in O(1) time and there are O(log(n)) pieces so the total time complexity for the sequential

process is O(log(n)). The sequential process requires one add to each element in the array not including the first log(n) elements so the work complexity for the second part is O(n - log(n)) which reduces to O(n). The work complexity of the entire algorithm is O(n + log(n) * log(n) * log(log(n))) which reduces to O(n). The time complexity of the entire algorithm is O(log(n) + log(log(n))) which reduces to O(log(n))

Work Reduction:



Example:

```
01  02  03 | 04  05  06 | 07  08        // split into log(n) pieces
01  03  06 | 04  09  15 | 07  15        // run non-optimal parallel prefix on pieces
             +6  +6  +6
01  03  06 | 10  15  21 | 07  15        // first iteration of sequential sum algorithm
                         +21 +21
01  03  06 | 10  15  21 | 28  36        // second iteration of sequential sum algo
                                       // complete
```