**Algorithm**

For the two outer processes (P1 and P3), check the two adjacent pairs they each control, making sure to use semaphore wait/signal when operating on the shared character index. If the two characters are out of order, swap them. If the process goes 2 cycles without making any swaps, decrement the process associated semaphore to zero. If a swap was done in the cycle, increment the semaphore. If the END semaphore is raised while the process has the process associated semaphore decremented, stop the process.

For the centre process (P2), check the two adjacent pairs it controls, making sure to use semaphore wait/signal when operating on the shared character indices of each pair. If the two characters are out of order, swap them. If the process goes a cycle without making any swaps, check to see if the two semaphores associated with P1 and P3 have been reduced to 0. If this is true, and the next cycle also has no swaps, then the END semaphore will be incremented. The process will then end.

This algorithm probably has a race condition with the ending condition (between the last swap from P2 and the two cycles before raising the END flag), but I rushed through this due to other classwork and I don’t want to risk breaking my program now.

**Goals**

The program uses a parent process and 3 children processes. Those children run the algorithm above, while the parent process handles user input and output, sends the starting semaphore signals, sets up and removes shared memory/semaphores, and waits for the children to be finished. The processes only use semaphores for the critical sections, and there two separate semaphores for the two different shared characters. This allows the program to process two different operations at the same time which achieves the concurrency goal. The tracking of state in the processes should ensure that the elements are in order.

**Results**

While the program works, the debug output shows that there are many times that the program cannot continue due to one of the processes not yet doing a swap. The program appears to perform a stable sort (Bubblesort). For the two mandatory test sets the results are: “AMWZ596” and “CDEF123”.