CPSC 457 – Fall 2025
Assignment 1 — Parts I & II

Name: Jaiveer Toor
UCID: 30204466
Git Repository URL: https://github.com/jtoor2005/cpsc457-a1-treasure-hunt

Part 1:

For Part I I needed to report the treasure location without shared memory while using only exit and wait. An exit status is only eight bit, so a child cannot return a column index that may be as large as nine hundred ninety nine. Each child scans its assigned row and exits with one for found or zero for not found. The parent stores the process IDs as it forks, waits for all children, identifies the winner PID from exit statuses, maps that PID back to the row number, and then rescans that row in memory to compute the exact column. Every child calls _exit and the parent calls wait for all children, which prevents zombies. I also validate that all input cells are zero or one and warn if there is trailing data. The program prints the required child search lines and the final parent line in the exact format.

Part 2:

For Part II I make sure the program never creates more processes than needed by computing an effective count as the minimum of N and the number of values in the range. I use this effective count for shared memory sizing, forking, waiting, and printing. The inclusive range from lower to upper is split into that many parts using balanced chunking. I compute a base size and give one extra value to the first few children when the division is not exact. This yields even work and no overlapping subranges. The shared memory segment contains an array of counts followed by an array of primes whose capacity equals the effective count times the ceiling of total values divided by the effective count. Child i writes only inside its own block and then stores how many primes it wrote in counts at index i. Because blocks do not overlap there is no need for locking. Each child detaches from shared memory before exiting. After the parent waits for all children it reads the results, detaches, and removes the segment with shmctl using IPC_RMID. Command line inputs are parsed with strtol and validated, including checks that lower is not greater than upper and that N is positive. The work is evenly divided, primality testing runs up to the square root, and there are no unnecessary delays. The number of child processes never exceeds N, and when the range has fewer values than N I use the range length instead.