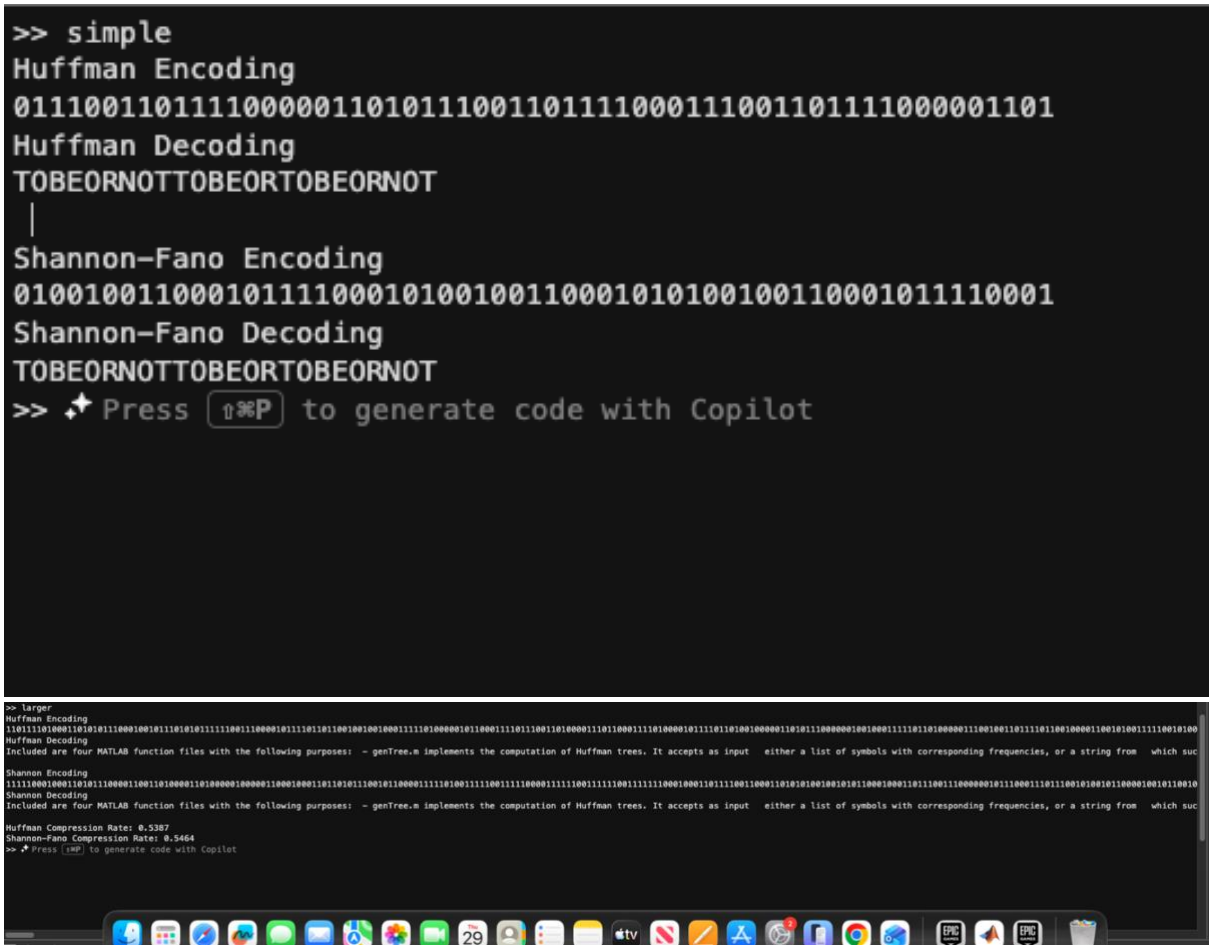

A1 part 1 output.

I implemented the Shannon–Fano lossless compression algorithm. The algorithm first computes the frequency of each symbol and sorts the symbols in descending order of frequency. It then recursively partitions the sorted symbol list into two groups such that the total frequencies of the two groups are as close as possible. At each partition, a binary digit (0 or 1) is assigned to the symbols in each group. This recursive process continues until each symbol is assigned a unique binary code, forming a Shannon–Fano coding tree.

To support both Huffman and Shannon–Fano coding, I modified genTree.m to accept a parameter specifying the tree generation method. When the Shannon–Fano option is selected, the function applies the recursive balanced-partition strategy described above instead of the Huffman merging process. The resulting tree is then used by the existing encoding and decoding functions without further modification.

I tested the implementation on the provided simple example and generated the corresponding Shannon–Fano tree. I also generated both Huffman and Shannon–Fano trees for the larger example and computed their compression rates, assuming a bit-level representation of the encoded symbols as specified in the README. The compression rates produced by the two algorithms were then compared. (See screenshots)