(To be sorted, formalized, and put into paragraph form)

1. **Cache-aligned lookup tables**: Aligning decoding table(s) with the cache to reduce cache misses, and speed-up symbol lookup. Since the lookup table has to be accessed every time a symbol binary code is detected, if the table is broken up across several memory locations, or not cache-aligned, cache misses will occur frequently, causing great time inefficiencies. To implement:
    a. use memory allocators or attributes to align lookup table to cache (64 bits in our case)
    b. Keep the total size of the lookup table to lower than the total size of the L1 cache (I think that's 512 in our case, so lookup table shouldn't exceed 256)
2. **Fixed-size bit buffer management**: Maintaining of a 64-bit input buffer from which bits are consumed during decoding. The buffer is only "refilled" from the byte stream when it's been emptied. This reduces the number of times memory has to be accessed, as well as reduces the overhead, since dynamic memory is not necessary because the buffer is fixed-size, making it predictable and fast. To implement:
    a. In the loop that reads from the text string, use a **preallocated** uint64_t
    b. Keep a running count of how many bits from the uint64_t have been consumed (so you know when memory has to be accessed again - ie the next iteration of the loop needs to be started)
3. **Register-resident local variables**: Keeping frequently accessed states in the CPU registers, as opposed to in memory (it's way more expensive to access external memory). This drastically reduces the number of times the program has to traverse between memories. To implement:
    a. Declare those frequently used values as "register," or provide further hints for the compiler like certain flags (apparently -03?)
4. **Parallel multi-stream decoding**: The usage of SIMD vector instructions to decode multiple streams at once. Ie instead of process each symbol individually, we can process several symbols at the same time (parallelism, pipelining, whatever you want to call it). To implement:
    a. Use SIMD instructions (allegedly vld1q_u8, vshlq_n_u64, etc) for the buffers **(combine with optimization #2)**
    b. Split text string into independent blocks
    c. Use in-class tricks to force decoding to happen in separate SIMD lanes
5. **Branch-free decode loops**: Avoiding if/else statements inside loops; use table-driven FSMs and conditional select instructions (csinc, csel, etc) instead. *Branches are bad and should be avoided at all costs.* To implement:
    a. Use select or ternary operations instead of if (eg. result = condition ? a : b;)
    b. Certain compiler flags can assists with this as well (-03 again for example)