Fast String Matching

Modern day computers have a word-size of 64-bit or 32-bit (not so much prevalent now). That means the CPU can process an assembly instruction on data of size 64-bit. Every character in C is represented using 1 byte (8 bits). A lot of string matching algorithms compare character by character. This means that at CPU level the 24 higher order bits have been padded with 0s and the next 8 bits represent the character. This wastes CPU power.

So, instead of comparing 8 bits/1byte/1character at a time, we can compare 64bits/8bytes/8characters in a single shot on a 64 bit machine or 4 characters on a 32-bit machine.

Lets say the word size is 32-bits and the text string is “ABCD” and the pattern string is “DEFG”. We can interpret the text and the pattern as an unsigned int32 and compare them directly using ==. However, since we also need some information of how many characters matched to figure out the shift amount, we can do two things: subtract the two integers or take an xor. Subtraction involves the usage of an adder at machine level, which involves the carrying-over of bits if the need arises. This is not as efficient as an XOR where a comparison can be done directly and doesn't require and carry bits to be forwarded to higher order bits. So, doing an XOR will give us sufficient information of how many characters matched, and then we can directly use the mismatched character to figure out the shift required using the shift tables used in Horspool's and Boyer Moore's algorithms.

When the algorithm has to be implemented for practical uses, we’ll have to take care to figure out the endian-ness of the machine (for interpreting the int32 form of the strings), and take care of the cases when the number of characters is not a multiple of 4 (in case of a 32bit machine).