Project 6: Hashing: Karp-Rabin Algorithm

Problem Description Karp-Rabin hashing[1] is an algorithm for computing the hash codes of successive k-element substrings. For example, if we have the string ABCDEFG, we can quickly compute the hashes of ABCD, of BCDE, CDEF, and DEFG respectively. This kind of hash is sometimes called a rolling hash.

In this project, you need to write a program to search a database of DNA sequences, represented as strings of characters, to find matches of other DNA subsequences. Both the database and the query strings are made up of only four characters: A, C, G and T. Your program should report the location of an exact match within any given input DNA sequence for each input search query string. If the query string matches within multiple sequences within the database, each result must be reported; and if the query string matches multiple locations within the same database sequence, the earliest position that matches exactly must be reported. The file names for this problem (database file, queries file, output results) will be given on the command line.

The input database and query files will have the same format. Each sequence will be prefixed by a line starting with a greater than character (“>”) followed by a description of the origin of the sequence of no more than 131 characters. The sequence will then begin on the next line for some number of lines. Each line will contain exactly 80 characters from the set A, C, G and T, except the last line, which may hold fewer than 80 characters. Following this last line will be the descriptor line from the next sequence until the end of the file has been reached, which will be signified by the descriptor (“>EOF”). For each query string contained within the second input file, the output file should print the descriptor of the query sequence and the descriptors of any database sequences that contain a match as well as the position within the database sequence of that exact match. If the query sequence string is not found within any database sequences, a message to that effect should be printed after the query descriptor. Thus the program takes as input two sets of strings: the Database strings and the Query strings. We will refer to these two sets of strings as the database and the querybase respectively. We make some assumptions about the input:

- Each string is less than 1,000,000 bytes.
- The entire database fits in main memory, but the entire querybase may be too big to fit in main memory.
- There are less than 232 strings in a database or a querybase.

[1] https://en.wikipedia.org/wiki/Rabin-Karp_algorithm

The output format is described here[2]. In that example, the database is:

>DB description string 1

AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTC
TGATAGCAGCTTCTGAACTGGTTACCTGCCGTGAGTAAATTAAAATTTTATTGACTTAGGT
CACTAAATACTTTAACCAATATAGGCATAGCGCACAGACAGATAAAAATTACA

>DB description string 2

AACGGTGCGGGCTGACGCGTACAGGAAACACAGAAAAAAGCCCGCACCTGACAGTGCGG
GCTTTTTTTTTTCGACCAAAGGTAACGAGGTAACAACCATGCGAGTGTTGAAGTTCGGCGG
TACATCAGTGGCAAATGCAGAACGTTTTCTGCGTGTTGCCGATATTCTGGAAAGCAATGC

>EOF

and an example querybase is:

>Query description string 1

CATTCTGACTGCAA

>Query description string 2

AAAAAAG

>Query description string 3

GTAA

>Query description string 4

AGAGAGAGAGAGAGAGAGAGAGAGAGAGAG

>EOF

The output would then be:

Query description string 1

[DB description string 1] at offset 7

Query description string 2

[DB description string 1] at offset 47

[DB description string 2] at offset 33

Query description string 3

[DB description string 1] at offset 94

[DB description string 2] at offset 79

Query description string 4

NOT FOUND

[2]http://seu.wangmengsd.com/ds/data.zip

Karp-Rabin hashing works as follows. First we chose a prime number P. The hash of a string q is computed as:

$$\sum_{0 \le i \le |q|} h(q_i) P^{|q|-i-1}, \tag{1}$$

where $h(q_i)$ is a non-negative integer less than P. In this case, since our alphabet consists of the four letters A, C, T, G, we can choose $P = 5$, and $h(A) = 0, h(C) = 1, h(T) = 2, h(G) = 3$. Thus, if we have the string TATGTGAGAAGA, we can encode it as 202323030030.

To compute the hash of the first four characters we have

$$\begin{aligned} h(2023) &= 2 \cdot 5^3 + 0 \cdot 5^2 + 2 \cdot 5^1 + 3 \cdot 5^0 \\ &= 2 \cdot 125 + 0 \cdot 25 + 2 \cdot 5 + 3 \cdot 1 \\ &= 261. \end{aligned}$$

To compute the hash of the next 4-character subsequence we have

$$\begin{aligned} h(0232) &= 0 \cdot 5^3 + 2 \cdot 5^2 + 3 \cdot 5^1 + 2 \cdot 5^0 \\ &= 0 \cdot 125 + 2 \cdot 25 + 3 \cdot 5 + 2 \cdot 1 \\ &= 57. \end{aligned}$$

But that can be computed incrementally by multiplying the previous hash by 5, adding in the newly added character and subtracting the newly removed character.

$$\begin{aligned} h(0232) &= h(2023) \cdot 5 - 2 \cdot 5^4 + 2 \cdot 5^0 \\ &= 261 \cdot 5 - 2 \cdot 625 + 2 \\ &= 57 \end{aligned}$$

Thus in general we have

$$h(s_{<i+1,j+1>}) = (h(s_{<i,j>})) \cdot P - h(s_i) \cdot P^{|j-i|} + h(s_{j+1}). \tag{2}$$

Since we know the length of the s in advance (4 in this example), we can compute $P^{|s|}$ in advance (in this example $54 = 625$). Furthermore, we can do this arithmetic in a fixed-size word (e.g., in a 32-bit int in C++), and it will work out.