Search for a substring pattern P in a text T

Naive method

- compare character-by-character, shifting P by 1 in T every time there is a mismatch. For P of length M and T of length N, this approach runs in time that is in O(N*M)

- also called a brute-force search

Knuth-Morris-Pratt

- compute a skip table (or skip array) that indicates how far P can be shifted in T on a mismatch when comparing character-by-character from the front of P

e.g. P = "XYXYZ"

skip table: row indexed by symbol found in T, column indexed by position in P

	X	Y	X	Y	Z
X	0	1	0	3	2
Y	1	0	3	0	5
Z	1	2	3	4	0
*	1	2	3	4	5

* denotes the set of all symbols not found in P

Boyer-Moore

-compute a skip array (also called skip table) that indicates how far P can be shifted in T on a mismatch when comparing character-by-character from the end of P

	X	Y	X	Y	Z
X	0	5	0	5	2
Y	5	0	5	0	1
Z	5	5	5	5	0
*	5	5	5	5	5

- both KMP and BM run in O(N+M), where N is the length of T and M is the length of P; however, BM tends to run in time proportionate to N/M because the maximum skip tends to occur a lot.

We note that the skip table (array) represents a FSM indicating what to do in every possible state. We further note that no pair of characters between P and T are considered more than once in KMP or BM because we either already know a match will

occur if our skip maintains some overlap, so we don't need to reconfirm that match, or we skip past characters that would produce a mismatch, never looking at them again. This explains why both are in O(N+M).

In many discussions about KMP and BM, the skip is computed by a function ... called the good_prefix function for KMP, and the good_suffix function for BM, where the function computes (in the case of KMP) the longest prefix of the matched pattern thus far that matches up to and including the mismatched character (i.e. matching the next longest prefix of the pattern), or (in the case of BM) the longest prefix of the matched portion thus far (i.e. a suffix) that is also a good suffix of the pattern. Just think about it for a while, and it'll make sense to you.

Rabin-Karp

  -test for a match by effectively hashing every subsequence of T with the same length as P; then confirm a hash-match with a string comparison. Unlike standard hashing, where a random uniform distibution over the address space is the principal goal (i.e. fast lookup) ..... for string searching, the goal is to be able to compute the hash function quickly for every position in T, with as few spurious hits as possible. Rabin-Karp is useful for snooping serial communication lines (i.e. bit sniffing).

Suffix trie

  -build a [compact] suffix trie for T, then a search for P goes directly to every point where P occurs within T in time proportionate to the length of P (plus the number of times P occurs if all instances are required).