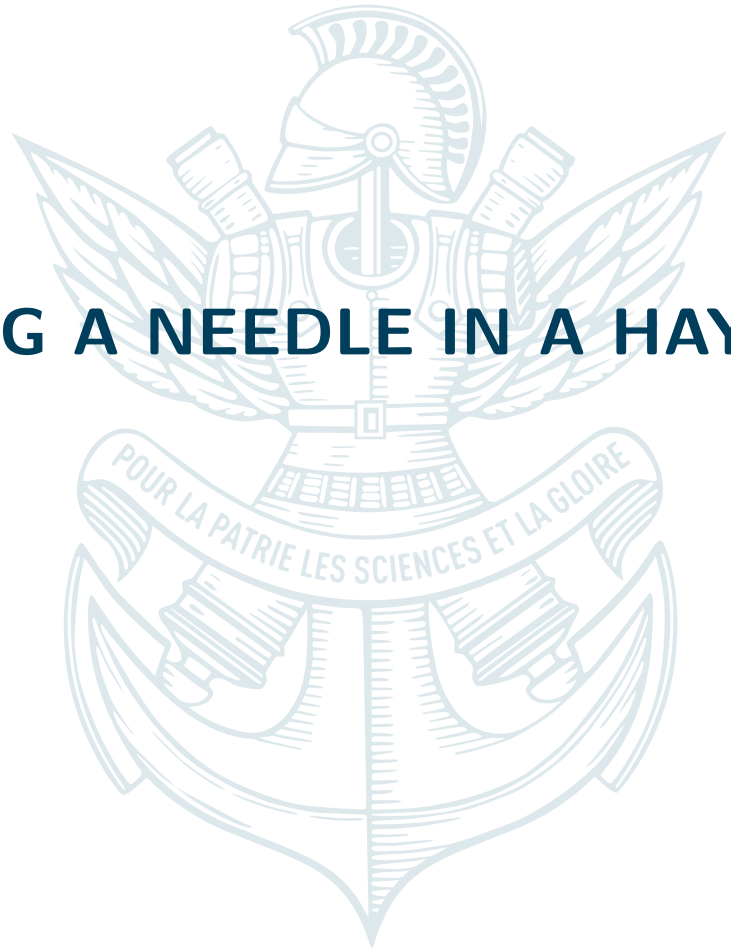

FINDING A NEEDLE IN A HAYSTACK

12 janvier 2016

Chloé Papin et Victor Quach

1 IMPLEMENTATION OF THE FOUR ALGORITHMS

We use char arrays to store our strings.

1.1 Naive algorithm

See file `Naive.java`
This simply implements the algorithm described in the subject. See file `Naive.java`

1.2 Karp-Rabin algorithm

See file `KarpRabin.java`.
We chose $w = 3000000$.

`hash` calculates the hash of the string defined by the char array s. It uses Ruffini-Horner's algorithm to do so (hash seen as a polynomial evaluated for the value 2).

Similarly, `hash_index` computes the hash of the substring of query, starting from given index.

All calculations are done mod w so that queries of length greater than 32 can be found.

Otherwise, we would be limited by `MAX_INT` (and queries would fail after 32 characters). At first, the function `twotothe` was implemented using exponentiation by squaring, then with a recursive implementation but it caused a stack overflow, so we ended up using a library function.

1.3 Knuth-Morris-Pratt

See file `Knuth-morris-Pratt.java`.
The function `setNext` fills the array `next` as described by the subject.
`next[0]` is not defined (nor used) : a mismatch in a first character gives no information on the haystack.

To set the `next` array, we use an auxiliary array in which we store, for each index `i`, the length of the maximal string which is both a prefix of `needle` and a suffix of `needle[0..i]`, whether or not these are followed by unmatching characters. Then we can keep only the information concerning the case where the last character differs.

1.4 Boyer-Moore

See file `Boyer-Moore.java`.
To store the occurences of the characters, we decided to use a hashmap. It is an efficient way to store these, whatever the size of the alphabet.

The implementation of the array in which we stored the indices for the good-suffix rule was not optimal. The first version we used consisted in using two different arrays, the first one in case the

suffix did exist somewhere else in the string, and the second one in case we had to find the best prefix matchnig the suffix. However, this did not pass all the tests, so we kept a naive version of this search.

2 COMPLEXITY

2.1 Naive algorithm

This algorithm has a $O(mn)$ complexity.

2.2 Karp-Rabin algorithm

If there are no hash collisions, this algorithm runs in $O(n)$. However, in worst-time, this is still $O(mn)$.

2.3 Knuth-Morris-Pratt algorithm

Let's show that the complexity of this algorithm is linear in the worst-case. Each character of the haystack might be read more than once : exactly each time there is a mismatch and following a strike of successes. But this happens only at most the number of times there were successes, that is a runtime of at most $2n$.

2.4 Boyer-Moore algorithm

The preprocessing for the Good shift rule that me implemented runs in $O(m^3)$. We have not implemented the $O(m)$ algorithm.

3 PERFORMANCE

In most of our tests, the naive algorithm was the fastest of all 4.
Tests used :

— Easy tests to check that the algorithm worked
— Random needles of increasing length against all data files given by the subject
— Handcrafted tests with repeated patterns to try to make Boyer-Moore outperform the naive algorithm.