Finding a needle in a haystack

Contact: Maks Ovsjanikov
Difficulty: medium

1 Introduction

In this project your goal will be to experiment with several algorithms for performing a basic operation – finding a substring in a long piece of text. As you will see, there is no ideal solution to this problem and choosing the right algorithm highly depends on the properties of the query, of the text (and in particular the size of the alphabet) as well as the number of queries that you are expecting to perform on a given text. You will also see that the most naive algorithm is almost never the best choice, and with a small number modifications, one can obtain significant improvements in performance. As a way of motivation, note that beyond the obvious everyday application of looking for a word or a phrase in a large document, substring searching is very important in bio-informatics, where, for example, the human genome can be represented as a string with roughly 3 billion letters, and a substring can correspond to a particular gene, whose length is approximately 15 thousand letters on average (with a huge variability in size, however).

2 Problem Definition

Throughout the exercise, we will assume that we are given a string S of length n and a query string q of length m, where $m \leq n$ and both S and q consist of letters that belong to a particular alphabet Σ. In other words, $S[i], q[j] \in \Sigma \; \forall i \in [0..n-1], j \in [0..m-1]$. Our goal then is to find all occurrences of q in S, which are integers k, such that $q[i] = S[k+i] \; \forall i \in [0..m-1]$. The decision version of this problem is to state whether q occurs in S or not, which for our purposes will be equivalent to returning the first instance of q in S.

3 Data

In this project, you will use several datasets of increasing complexity to test the various algorithms:

1. Random strings of ASCII characters for increasing n and m.
2. Mysterious Island by Jules Verne (approximately 1.1M characters).
3. War and Peace by Leo Tolstoy (approximately 3.2M characters).
4. Fibrobacter succinogenes, chromosome (approximately 3.8M characters).
5. Aspergillus fumigatus, chromosome (approximately 5M characters).

You can find an archive conaining all of these texts at the following address: http://goo.gl/aU1AdU

Note that we only provide you with the target text, not the queries. It will be your job to create your own query strings of increasing lengths to test the performance of your implementations.

4 Naive Algorithm

The simplest "naive" algorithm for the substring search problem is the method that tries all possible offset positions k between 0 and $n-m$ and verifies whether $q[i] = S[k+i]$ for all i between 0 and $m-1$. Of course, as soon as $q[i] \neq S[k+i]$ for some i, one can safely increment k and try the next position.

Problem 1. Implement the naive algorithm and test it on the data mentioned above, by considering query strings of increasing length.

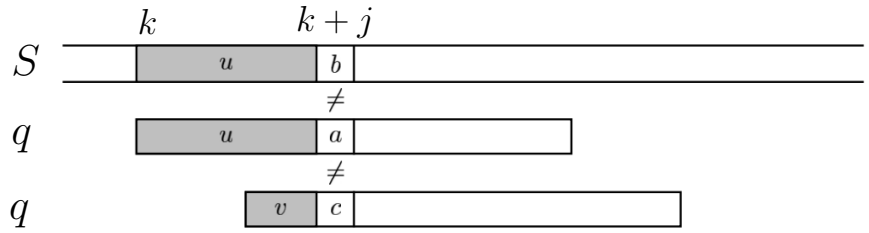

Figure 1: Illustration of the Knuth-Morris-Pratt algorithm

5 Karp-Rabin Algorithm

The main problem with the naive algorithm is that it ignores the work done at position $i-1$, when testing position i. Ideally, we would like to re-use some of the information that we already have so that in most cases testing position i would be a constant-time operation, independent of the length of the query string q. One of the simplest variants of this idea is the Karp-Rabin algorithm [KR87], which is based on hashing.

Consider a query string q of length m, and suppose we have a hash function `hash` that assigns an integer value to every string of length m, so that if $\texttt{hash}(s1) \neq \texttt{hash}(s2)$ then $s1$ definitely differs from $s2$. If the hash values match, the strings might still be different, and we can test this by using the naive algorithm.

Let the hash function be:

$$\texttt{hash}(s) = (s[0] * 2^{m-1} + s[1] * 2^{m-2} + ... + s[m-1]) \bmod w,$$

where w is some large number. Note that if we have a string $s2$ such that $s2[i] = s1[i+1], i \in [0..m-1]$ then $\texttt{hash}(s2) = ((\texttt{hash}(s1) - s1[0] * 2^{m-1}) * 2 + s2[m-1]) \mod w$. In other words, when searching through S, we can compute the hash function of the string at string at position $k+1$ if we know that of the string at position k in constant time. The Karp-Rabin algorithm then computes the hash value of the query string q and compares it to the hash values of the substrings of the same size in text S which are built incrementally.

Problem 2. Implement the Karp-Rabin algorithm as described above and test it on the given data by considering queries of increasing length. Does your code work for query strings of length bigger than 32?

6 Knuth-Morris-Pratt algorithm

Note that the worst-time complexity of the Karp-Rabin algorithm is the same as that of the naive algorithm, although in practice it works significantly better. The Knuth-Morris-Pratt (KMP) algorithm [KMP77], however, is linear even in the worst case.

To understand the algorithm, consider Figure 1. When we're checking the presence of q in S starting at position k, suppose the first mis-match happens at position j. I.e., $q[i] = S[k+i] \ \forall \ i \in [0..j-1]$, and $q[j] \neq S[k+j]$. Let u be the string formed by the first $j-1$ letters of q. Now when shifting the index from k forward, we should expect that some prefix of q (i.e., a substring starting at the beginning of q), match some suffix of u. In Figure 1 this matching substring is denoted by v. Moreover, since we know that j^{th} character doesn't match, we would like the character following v to be different from the one following u in q, since otherwise we would immediately get a mismatch.

Now let us create an auxiliary variable `next` of size m, such that $\texttt{next}(i)$ is the length of the longest prefix of $q[0..i]$ satisfying the above requirements. Then, as soon as we find a mismatch, we can shift by $j - \texttt{next}(j)$ and start the comparison between characters $S[k+j]$ and $q[j - \texttt{next}(j)]$ without risking missing any occurence of q in S, and avoiding any backtracking.

The KMP algorithm starts by creating the auxiliary variable `next` (which can easily be done in time $O(m)$) and then using it during the search for q in S. The worst-time complexity of the algorithm is $O(n+m)$ independent of the size of the alphabet.

Problem 3. Implement the KMP algorithm as described above and test it on the given data. Can you demonstrate the worst-case linear time behavior? Are there cases where it works consistently better than the other two methods?

7 Boyer-Moore algorithm

Although worst-case linear is of course theoretically optimal without placing assumptions on the strings S and q, in practice we can sometimes do even better provided the strings are not too regular.

One of the main insights of the Boyer-Moore algorithm is that if we check for the presence of the substring starting *from the right* rather than from the left, then we might be able to skip some parts of S without ever looking at them. To illustrate this suppose we are considering offset k and see a character at position $k+i$ of S that does not occur in q. Then we can safely skip to a new offset at $k+i$ without looking at the characters between k and $k+i$.

The Boyer-Moore algorithm [BM77] is considered one of the most efficient substring search algorithms in typical scenarios and its variants are often implemented in text editor applications. The algorithm is based on two rules: the *bad-character shift* and the *good suffix shift*, and, as mentioned above, on testing the given substring from right to left, while advancing the starting index k from left to right. For both of these rules we consider what happens when we find a character $S[k+i]$ that mismatches the character $q[i]$ during our search. The bad character rule consists in shifting the starting index k so that the character $S[k+i]$ aligns with its rightmost occurence in $q[0..i]$. For this, we need to store, for every character in the alphabet its occurrences in the query string, which can be found by creating an auxiliary variable with the size of the alphabet.

The good suffix shift consists in aligning the segment $S[k+i+1..m-1] = q[i+1..m-1]$ with its rightmost occurrence in q that is preceded by a character different from $q[i]$. If there exists no such segment, the shift consists in aligning the longest suffix v of $S[k+i+1..k+m-1]$ with a matching prefix of q. Again, this can be found by creating an extra variable that stores the necessary references.

Problem 4. Implement the Boyer-Moore algorithm as described above and test it on the given data. What do you observe when the size of the alphabet grows? In what cases does this algorithm outperform the others?

8 Remarks

Note that all the algorithms described above are based on searching through the long string S of size n and in the worst case still have complexity $O(n)$. Now if S is fixed in advance and we would like to support many search queries against it, then it might make sense to spend extra time preprocessing S so that during the search, the complexity depends primarily on the length of the query string q. This functionality is supported by a data-structure called Suffix Trees, introduced by Weiner [Wei73] that allow linear time search for a substring of length m independently of the length of the text S, at the cost of some preprocessing and extra storage. This is especially useful when the text S is very long and fixed and we are expecting to perform many search queries against it, as is often the case in bioinformatics. If you are feeling ambitious, as a bonus question, implement and test a basic suffix tree based on a description for example found in this document http://goo.gl/VM6PYL. An excellent book that describes all of the algorithms mentioned above, as well as suffix trees and many other data structures is [Gus97]

References

[BM77] Robert S Boyer and J Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[Gus97] Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge university press, 1997.

[KMP77] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[KR87] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.