Fingerprinting for text search

We consider the problem of finding all possible matches of a pattern string P within a text T. This problem has many applications, *e.g.*, search engines, matches of particular DNA sequences in biology, etc. We will consider both the pattern string P and text T to be represented as an array of characters, respectively of size m and n (starting at index 0). We will denote by T_i^m the substring $T[i:i+m]$ (*i.e* the string $T[i]\dots T[i+m-1]$). We will restrict to characters from the (extended) ASCII ones.

Open a new project `CSE202_TD6` in Spyder. Download the file `search.py`, which contains the signatures of the various functions to implement, plus a function `string_compare(P,S)`, which tests whether two strings P and S (of same length) are equal ($O(m)$ maximum number of characters comparisons). We will use this function anytime we need to compare two strings[1]. In addition, download the file `search_tests.py`, which contains several functions for testing your algorithms and analysing their performances (feel free to modify the functions parameters).

1 A naive way for finding string matches within a text

Checking whether the pattern P matches within the text T at position i boils down to checking that $P[j] = T[i+j]$ for all $j = 0, \dots, m-1$. Therefore, a naive way of finding all possible matches of P within T is to iterate over all possible contiguous substrings of T of size m and check whether all the characters equals the one of P (from left to right).

Figure 1 shows an example of the application of this naive string matcher with text $T =$ **abacadabra** and pattern string $P =$ **abra**.

i	0	1	2	3	4	5	6	7	8	9
	a	b	a	c	a	d	a	b	r	a
0	**a**	**b**	r	a						
1		a	b	r	a					
2			**a**	b	r	a				
3				a	b	r	a			
4					**a**	b	r	a		
5						a	b	r	a	
6							**a**	**b**	**r**	**a**

Figure 1: Naive string matching algorithm for the pattern string $P =$ **abra**. The pattern P is compared to every valid starting position within the text T. In green are shown characters of P that match the text and in red the first character that is not matching. Since we compare strings left to right, characters in gray do not need to be checked. In this example, there is only one match of P at position 6 within the text T.

Question 1. Write a function `string_match(T,P)` which returns the list of positions within the text T matching P, following the algorithmic scheme described above. It will serve as a base comparison in the following. You can test your implementation by calling `test1()` from `search_tests.py`.

Remark. The algorithm has an $O((n-m)m)$ worst-case time complexity, considering the time complexity of comparing two strings of length m to be $O(m)$ (*e.g* when two strings match, exactly m characters comparisons). An example with no match where worst-case time is attained is for $P = a^{m-1}b$ ($m-1$ times a followed by b) and $T = a^n$. Then for each test, one has to wait until the last symbol of P to see that it is a mismatch.

[1] We could use the Python built-in equality tests for strings (using the operator "=="). However, Python has an optimised way of managing strings and comparing them. Using this will hide the complexity study of the algorithm in this tutorial. So we rely on `string_compare` to compare strings.

2 Rabin–Karp algorithm: fingerprinting for finding string matches

Rabin and Karp have proposed a string matching algorithm which is not based on comparing the characters of the strings. Instead, the idea of Rabin and Karp for comparing a pattern string P to a substring $T_i^m = T[i : i + m]$ is to compare a fingerprint of P and T_i^m: an integer which is constructed from the strings P and T_i^m.

In order to compute the fingerprints, we consider a function h that takes a string as input and returns a fingerprint. This function acts like a hash function: from the characters of the input string S, it builds an integer that "defines" S, i.e., it should be unlikely that two different strings share the same fingerprint, *but* having two different fingerprints means the two corresponding strings are different.

One difficulty for comparing fingerprints instead of characters is that if, for a string S of size m, the algorithm computing $h(S)$ has an $O(m)$ time complexity (since the fingerprint is computed from all characters of S), then there is no complexity gain (since we will have to compute the fingerprints of all of the T_i^m). However, Rabin and Karp suggested to use a class of fingerprint functions, the class of *rolling hash functions*, which enables to compute $h(T_{i+1}^m)$ from $h(T_i^m)$ in constant time. In particular, it is possible to quickly obtain $h(T_{i+1}^m)$ from $h(T_i^m)$, using the characters $T[i]$ and $T[i + m]$. We will study the Rabin-Karp algorithm through different fingerprint functions.

2.1 A basic fingerprint

First, we consider the following simple fingerprint, given a string S of size m:

$$h(S) := \sum_{i=0}^{m-1} \text{ASCII}(S[i]),$$

where $\text{ASCII}(S[i])$ gives the ASCII code of the character $S[i]$ (in Python, this is the function `ord`). This function is a rolling hash function as:

$$h(T_{i+1}^m) = h(T_i^m) - \text{ASCII}(T[i]) + \text{ASCII}(T[i + m]).$$

Question 2. Write a function `hash_string_sum(S)` which, given a string S, returns the sum of the ASCII code of its characters as described above. Remember that the `ord()` method converts a character to its Unicode code.

Question 3. Write a function `hash_string_sum_update(h, Ti, Tim)` which, given the fingerprint h of T_i^m, $Ti = T[i]$ and $Tim = T[i + m]$, computes the fingerprint of T_{i+1}^m as described above.

Now we can describe the Rabin–Karp algorithm with this fingerprint function for finding substring within T matching P. The algorithm iterates over all possible initial positions i within the text T and compares $h(P)$ with $h(T_i^m)$. The fingerprints $h(P)$ and $h(T_0^m)$ are computed once with `hash_string_sum`, while the fingerprints $h(T_{i+1}^m)$ are computed from $h(T_i^m)$ using `hash_string_sum_update`. During the iterations, if $h(P) = h(T_i^m)$, then we have to check whether we did not fall on a *spurious hit*, that is $P \neq T_i^m$ but their fingerprints are the same. To check this, we simply compare the two strings[2]. Figure 3 shows an example of the application of Rabin–Karp, using `hash_string_sum` as fingerprint function, on the text string **abacaaabra** and pattern string **abra**.

Question 4. Write a function `rabin_karp_sum(T,P)` which implements the Rabin–Karp algorithm described above. The algorithm should return a pair whose first element is the list of positions in T matching P and whose second element is the number of spurious hits encountered. Check that your implementation is correct with the function `test4()` from `search_tests.py`.

Remark. Try the function `test_krsum_worst_case()`, which plots timings of `string_match` and `rabin_karp_sum` for various text sizes and fixed template lengths. The pattern string P has length

[2]We hope that the number of spurious hits is small in order to compete with the naive method.

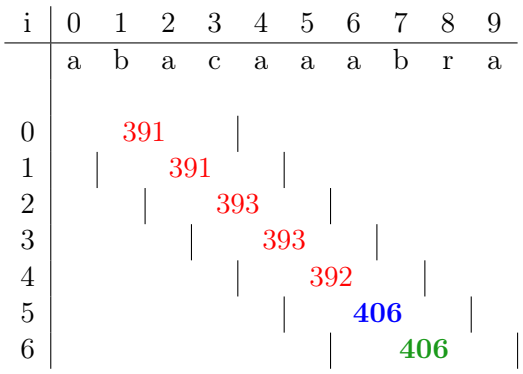

Figure 2: Rabin-Karp using `hash_string_sum` as fingerprint function h with pattern string $P =$ **abra**, for which $h(P) = 406$. The figure show the fingerprint of each T_i^m. At each iteration i, the fingerprint of T_i^m is compared to the fingerprint of P. If they are not equal (shown in red), P and T_i^m do not match. If the fingerpints are equal, we may have fallen for a *spurious hit* (in blue): $h(P) = h(T_i^m)$ but $P \neq T_i^m$. So we need to compare the strings when $h(P) = h(T_i^m)$. In green at position 6 is shown the only match of fingerprints which is not spurious. Note that for computing $h(T_{i+1}^m)$, we consider using the function `hash_string_sum_update`, using the fingerprint of T_i^m.

100, made of 99 'a' followed by a 'b'. The length of the text T varies from 100 to 1000 and is made of concatenations of P. The poor behaviour of `rabin_karp_sum` compared to `string_match` is due to the choice of the fingerprint function. As it is heavily commutative, the test shows a case where fingerprints always match (almost every hit of fingerprints is spurious).

2.2 A better fingerprint function: Rabin polynomials

The previous function for computing fingerprints does not have good properties: it can show many spurious hits, which in turn make the algorithm for matching strings inefficient.

We can think about using a fingerprint function that eliminates all possible spurious hits. This perfect fingerprint can be obtained (in our context) as follows for string S of length m:

$$h(S) = \sum_{i=0}^{m-1} d^{m-1-i}\text{ASCII}(S[i]),$$

where $d = 256$ is the number of ASCII characters. For a given string S of ASCII characters, its fingerprints uniquely defines it: if $h(S) = h(S')$, then $S = S'$. This function is a Rabin polynomial.

This function is also of the class of rolling hashing functions as:

$$h(T_{i+1}^m) = d\left(h(T_i^m) - d^{m-1}\text{ASCII}(T[i])\right) + \text{ASCII}(T[i+m]).$$

Remark. Note that the update formula involves the value of d^{m-1}, which becomes prohibitive when m becomes large (d^m is an integer of bit size $O(m)$). In order to limit this effect and to guarantee a constant time update, we consider the possibility of encountering spurious hits but with the guarantee that every arithmetic operations used for updating fingerprints within the text T operate on fixed size integers.

To do so, we consider evaluating the same Rabin polynomial as before but modulo a (random) prime q, *i.e.*,

$$h_q(S) = \left(\sum_{i=0}^{m-1} d^{m-1-i}\text{ASCII}(S[i])\right) \bmod q = \left(\sum_{i=0}^{m-1} \left((d \bmod q)^{m-i-1}(\text{ASCII}(S[i]) \bmod q)\right)\right) \bmod q.$$

Analogously, this fingerprint modulo q function is also a rolling hash function as:

$$h(T_{i+1}^m) = \left(d\left(h(T_i^m) - (d^{m-1} \bmod q)\text{ASCII}(T[i])\right) + \text{ASCII}(T[i+m])\right) \bmod q.$$

We will consider evaluating $d^{m-1} \bmod q$ once, and reuse the value as needed.

Question 5. Following the previous questions, we will write the Rabin–Karp algorithm with respect to this fingerprint function.

1. Write a function `hash_string_mod(S, q)` which, given a string S of size m and a prime q, evaluates the Rabin'polynomial modulo q as described above and return its valuation. Note: use the Horner rule for evaluating the polynomial (see Horner's method).

2. Write a function `hash_string_mod_update(h, q, Ti, Tim,dm)` which, given the fingerprint h of T_i^m, a prime q, $Ti = T[i]$, $Tim = T[i+m]$, and $dm = d^{m-1} \bmod q$, computes the fingerprint of T_{i+1}^m as described above.

3. Write a function `karp_rabin_mod(T,P,q)` which implements the Rabin–Karp algorithm for a text T and pattern P and prime q, using the two above fingerprint functions `hash_string_mod` and `hash_string_mod_update`. The algorithm should return a pair whose first element is the list of positions in T matching P and whose second element is the number of spurious hits encountered. Check that your implementation is correct with the function `test5()` from `search_tests.py`.

Question 6. Call the function `test_krmod_primes()`, which calls `karp_rabin_mod` on random pattern strings (of lowercase alphabetical characters) of varying lengths m, a random text T (of lowercase alphabetical characters) of size $100m$, and four random primes of respectively 8, 16, 32, and 64 bits[3]. For each prime number, `test_krmod_primes()` plots the computation time of `karp_rabin_mod` for various value of m and the number of spurious hits encountered. Observe the behavior of the algorithm.

Remark. The Rabin–Karp algorithm has a good expected complexity but is not competitive compared to other algorithms for string matching (that we will see later in the course). However, Rabin–Karp can be easily and efficiently extended for finding the matches of multiple patterns at the same time. You can try to adapt the Rabin–Karp algorithm for a set of pattern strings (of same length first, then of different lengths).

[3]We do not consider the problem of generating random primes. You can go to the following website to generate new prime numbers `https://asecuritysite.com/encryption/random3?val=8`.