Olympiads in Informatics, 2013, Vol. 7, 90–100

Where to Use and How not to Use Polynomial String Hashing

Jakub PACHOCKI, Jakub RADOSZEWSKI
Faculty of Mathematics, Informatics and Mechanics, University of Warsaw
Banacha 2, 02-097 Warsaw, Poland
e-mail: {pachocki,jrad}@mimuw.edu.pl

Abstract. We discuss the usefulness of polynomial string hashing in programming competition tasks. We show why several common choices of parameters of a hash function can easily lead to a large number of collisions. We particularly concentrate on the case of hashing modulo the size of the integer type used for computation of fingerprints, that is, modulo a power of two. We also give examples of tasks in which string hashing yields a solution much simpler than the solutions obtained using other known algorithms and data structures for string processing.

Key words: programming contests, hashing on strings, task evaluation.

1. Introduction

Hash functions are used to map large data sets of elements of an arbitrary length (the *keys*) to smaller data sets of elements of a fixed length (the *fingerprints*). The basic application of hashing is efficient testing of equality of keys by comparing their fingerprints. A *collision* happens when two different keys have the same fingerprint. The way in which collisions are handled is crucial in most applications of hashing. Hashing is particularly useful in construction of efficient practical algorithms.

Here we focus on the case of the keys being strings over an integer alphabet $\Sigma = \{0, 1, \ldots, A-1\}$. The elements of Σ are called *symbols*.

An ideal hash function for strings should obviously depend both on the multiset of the symbols present in the key and on the order of the symbols. The most common family of such hash functions treats the symbols of a string as coefficients of a polynomial with an integer variable p and computes its value modulo an integer constant M:

$$H(s_1 s_2 s_3 \ldots s_n) = (s_1 + s_2 p + s_3 p^2 + \cdots + s_n p^{n-1}) \bmod M.$$

A careful choice of the parameters M, p is important to obtain “good” properties of the hash function, i.e., low collision rate.

Fingerprints of strings can be computed in $O(n)$ time using the well-known Horner’s method:

$$h_{n+1} = 0, \quad h_i = (s_i + p h_{i+1}) \bmod M \quad \text{for } i = n, n-1, \ldots, 1. \tag{1}$$

Polynomial hashing has a *rolling* property: the fingerprints can be updated efficiently when symbols are added or removed at the ends of the string (provided that an array of powers of p modulo M of sufficient length is stored). The popular Rabin–Karp pattern matching algorithm is based on this property (Karp and Rabin, 1987). Moreover, if the fingerprints of all the suffixes of a string are stored as in (1), one can efficiently find the fingerprint of any *substring* of the string:

$$H(s_i \dots s_j) = (h_i - p^{j-i+1} h_{j+1}) \bmod M. \tag{2}$$

This enables efficient comparison of any pair of substrings of a given string.

When the fingerprints of two strings are equal, we basically have the following options: either we consider the strings equal henceforth, and this way possibly sacrifice the correctness in the case a collision occurs, or simply check symbol-by-symbol if the strings are indeed equal, possibly sacrificing the efficiency. The decision should be made depending on a particular application.

In this article we discuss the usefulness of polynomial string hashing in solutions of programming competition tasks. We first show why several common ways of choosing the constants M and p in the hash function can easily lead to a large number of collisions with a detailed discussion of the case in which $M = 2^k$. Then we consider examples of tasks in which string hashing applies especially well.

2. How to Choose Parameters of Hash Function?

2.1. *Basic Constraints*

A good requirement for a hash function on strings is that it should be difficult to find a pair of different strings, preferably of the same length n, that have equal fingerprints. This excludes the choice of $M < n$. Indeed, in this case at some point the powers of p corresponding to respective symbols of the string start to repeat. Assume that $p^i \equiv p^j \bmod M$ for $i < j < n$. Then the following two strings of length n have the same fingerprint:

$$\underbrace{0 \dots 0}_{i} a_1 a_2 \dots a_{n-j} \underbrace{0 \dots 0}_{j-i} \quad \text{and} \quad \underbrace{0 \dots 0}_{j} a_1 a_2 \dots a_{n-j}.$$

Similarly, if $\gcd(M, p) > 1$ then powers of p modulo M may repeat for exponents smaller than n. The safest choice is to set p as one of the generators of the group $U(\mathbb{Z}_M)$ – the group of all integers relatively prime to M under multiplication modulo M. Such a generator exists if M equals 2, 4, q^a or $2q^a$ where q is an odd prime and $a \geqslant 1$ is integer (Weisstein, on-line). A generator of $U(\mathbb{Z}_M)$ can be found by testing a number of random candidates. We will not get into further details here; it is simply most important not to choose M and p for which $M \mid p^i$ for any integer i.

A slightly less obvious fact is that it is bad to choose p that is too small. If $p < A$ (the size of the alphabet) then it is very easy to show two strings of length 2 that cause a collision:

$$H(01) = H(p0).$$

2.2. *Upper Bound on M*

We also need to consider the magnitude of the parameter M. Let us recall that most programming languages, and especially the languages C, C$++$, Pascal that are used for IOI-style competitions, use built-in integer types for integer manipulation. The most popular such types operate on 32-bit or 64-bit numbers which corresponds to computations modulo 2^{32} and 2^{64} respectively. Thus, to be able to use Horner's method for fingerprint computation (1), the value $(M-1) \cdot p + (M-1) = (M-1) \cdot (p+1)$ must fit within the selected integer type. However, if we wish to compute fingerprints for substrings of a string using the (2), we need $(M-1)^2 + (M-1)$ to fit within the integer type, which bounds the range of possible values of M significantly. Alternatively, one could choose a greater constant M and use a fancier integer multiplication algorithm (which is far less convenient).

2.3. *Lower Bound on M*

On the other side M is bounded due to the well-known birthday paradox: if we consider a collection of m keys with $m \geqslant 1.2\sqrt{M}$ then the chance of a collision to occur within this collection is at least 50% (assuming that the distribution of fingerprints is close to uniform on the set of all strings). Thus if the birthday paradox applies then one needs to choose $M = \omega(m^2)$ to have a fair chance to avoid a collision. However, one should note that not always the birthday paradox applies. As a benchmark consider the following two problems.

Problem 1: ***Longest Repeating Substring.*** Given a string s, compute the longest string that occurs at least twice as a substring of s.

Problem 2: ***Lexicographical Comparison.*** Given a string s and a number of queries (a, b, c, d) specifying pairs of substrings of s: $s_a s_{a+1} \dots s_b$ and $s_c s_{c+1} \dots s_d$, check, for each query, which of the two substrings is lexicographically smaller.

A solution to Problem 1 uses the fact that if s has a repeating substring of length k then it has a repeating substring of any length smaller than k. Therefore we can apply binary search to find the maximum length k of a repeating substring. For a candidate value of k we need to find out if there is any pair of substrings of s of length k with equal fingerprints. In this situation the birthday paradox applies. Here we assume that the distribution of fingerprints is close to uniform, we also ignore the fact that fingerprints of consecutive substrings heavily depend on each other – both of these simplifying assumptions turn out not to influence the chance of a collision significantly.

The situation in Problem 2 is different. For a given pair of substrings, we apply binary search to find the length of their longest common prefix and afterwards we compare the symbols that immediately follow the common prefix, provided that they exist. Here we have a completely different setting, since we only check if specific pairs of substrings are equal and we do not care about collisions *across* the pairs. In a uniform model, the chance of a collision within a single comparison is $\frac{1}{M}$, and the chance of a collision occurring within m substring comparisons does not exceed $\frac{m}{M}$. The birthday paradox does not apply here.

2.4. *What if $M = 2^k$?*

A very tempting idea is not to select any value of M at all: simply perform all the computations modulo the size of the integer type, that is, modulo 2^k for some positive integer k. Apart from simplicity we also gain efficiency since the modulo operation is relatively slow, especially for larger integer types. That is why many contestants often choose this method when implementing their solutions. However, this might not be the safest choice. Below we show a known family of strings which causes many collisions for such a hash function.

This family is the ubiquitous Thue–Morse sequence (Allouche and Shallit, 1999). It is defined recursively as follows:

$$\tau_0 = 0;\ \tau_i = \tau_{i-1}\bar{\tau}_{i-1} \quad \text{for } i > 0$$

where $\bar{x}$ is the sequence resulting by negating the bits of x. We have:

$$\tau_0 = 0,\ \tau_1 = 01,\ \tau_2 = 0110,\ \tau_3 = 01101001,\ \tau_4 = 0110100110010110, \ldots$$

and clearly the length of τ_i is 2^i.

Let $W(s)$ be the fingerprint of a string s without any modulus:

$$W(s_1 s_2 s_3 \ldots s_n) = s_1 + s_2 p + s_3 p^2 + \cdots + s_n p^{n-1}.$$

We will show the following property of the Thue–Morse sequence that uncovers the reason for its aforementioned especially bad behavior when hashing modulo $M = 2^k$.

Theorem 1. *For any $n \geqslant 0$ and $2 \nmid p$, $2^{n(n+1)/2} \mid W(\bar{\tau}_n) - W(\tau_n)$.*

Proof. By the recursive definition – $\tau_n = \tau_{n-1}\bar{\tau}_{n-1}$ and, similarly, $\bar{\tau}_n = \bar{\tau}_{n-1}\tau_{n-1}$ – we have:

$$\begin{aligned} W(\bar{\tau}_n) - W(\tau_n) &= W(\bar{\tau}_{n-1}) + p^{2^{n-1}} W(\tau_{n-1}) - W(\tau_{n-1}) - p^{2^{n-1}} W(\bar{\tau}_{n-1}) \\ &= W(\bar{\tau}_{n-1})(1 - p^{2^{n-1}}) - W(\tau_{n-1})(1 - p^{2^{n-1}}) \\ &= (1 - p^{2^{n-1}})(W(\bar{\tau}_{n-1}) - W(\tau_{n-1})). \end{aligned}$$

Now it is easy to show by induction that:

$$W(\bar{\tau}_n) - W(\tau_n) = (1 - p^{2^{n-1}})(1 - p^{2^{n-2}}) \ldots (1 - p).$$

To conclude the proof, it suffices to argue that

$$2^i \mid 1 - p^{2^{i-1}} \quad \text{for any } i \geqslant 1. \tag{3}$$

This fact can also be proved by induction. For $i = 1$ this is a consequence of the fact that p is odd. For the inductive step ($i > 1$) we use the following equality:

$$1 - p^{2^{i-1}} = \left(1 - p^{2^{i-2}}\right)\left(1 + p^{2^{i-2}}\right).$$

The second factor is even again because p is odd. Due to the inductive hypothesis, the first factor is divisible by 2^{i-1}. This concludes the inductive proof of (3) and, consequently, the proof of the whole theorem. □

By Theorem 1, the strings τ_n and $\bar{\tau}_n$ certainly yield a collision if $n(n+1)/2$ exceeds the number of bits in the integer type. For instance, for 64-bit integers it suffices to take τ_{11} and $\bar{\tau}_{11}$ which are both of length 2048. Finally, let us note that our example is really vicious – it yields a collision regardless of the parameter p, provided that it is odd (and choosing an even p is surely bad if M is a power of two). This example can also be extended by sparsifying the strings (i.e., inserting segments consisting of a large number of zeros) to eliminate with high probability a heuristic algorithm that additionally checks equality of a few random symbols at corresponding positions of the strings if their fingerprints are equal.

The same example was described in a recent blog post related to programming competitions (Akhmedov, 2012).

3. Usefulness of String Hashing

In the previous section we have described several conditions that limit the choice of constants for string hashing. In some applications it is difficult to satisfy all these conditions at the same time. In contrast, in this section we show examples of problems that are more difficult to solve without using string hashing. Two particularly interesting problems we describe in detail, and at the end we list several other examples of tasks from Polish programming competitions. In the algorithms we do not describe how to handle collisions, that is, we make an optimistic assumption that no collisions occur and thus agree to obtain a heuristic solution.

We have already mentioned that string hashing can be used to check equality of substrings of a given string. For the same purpose one could use the Dictionary of Basic Factors ($O(n \log n)$ time and space preprocessing for a string of length n) or the suffix tree/suffix array ($O(n)$ time and space preprocessing for a constant-sized alphabet in the basic variant, both data structures are relatively complex). More on these data structures can be found, for instance, in the books Crochemore *et al.* (2007) and Crochemore and Rytter (2003). However, in the following problem it is much more difficult to apply any of these data structures instead of string hashing. This is a task from the final round of a Polish programming competition Algorithmic Engagements 2011 (see `http://main.edu.pl/en/archive/pa/2011/bio`).

Problem 3: ***Computational Biology.*** We are given a string $s = s_1 \dots s_n$ over a constant-sized alphabet. A *cyclic substring* of s is a string t such that all cyclic rotations of t are substrings of s [1]. For a cyclic substring t of s, we define *the number of cyclic occurrences* of t in s as the total number of occurrences of distinct cyclic rotations of t within s. We would like to find a cyclic substring of s of a given length m that has the largest number of cyclic occurrences. We are to output this number of occurrences.

For example, consider the string $s =$ `BABABBAAB` and $m = 3$. The string `AAB` is its cyclic substring with 3 cyclic occurrences: one as `AAB`, one as `ABA` and one as `BAA`. The string `ABB` is also a cyclic substring and it has 4 cyclic occurrences: one as `ABB`, two as `BAB` and one as `BBA`. Thus the result is 4.

On the other hand, consider the string $s =$ `ABAABAABAABAAAAA` and $m = 5$. Here the result is just 1: a single cyclic occurrence of a cyclic substring `AAAAA`. Note that none of the strings `ABAAB` and `AABAA` is a cyclic substring of the string and therefore they are not included in the result.

Using string hashing, we solve this problem as follows. We start by computing fingerprints of all m-symbol substrings of s: $s_1 \dots s_m$, $s_2 \dots s_{m+1}$,. . . – this can be done in $O(n)$ time using the rolling property of the hash function. Using a map-like data structure storing fingerprints, for each of these substrings we count the number of times it occurs in s.

Now we treat these substrings as vertices of a directed graph. The vertices are identified by the fingerprints. Each vertex is assigned its multiplicity: the number of occurrences of the corresponding substring in s. The edges of the graph represent cyclic rotations by a single symbol: there is an edge from $u_1u_2 \dots u_m$ to $u_2 \dots u_m u_1$ provided that the latter string occurs as a substring of s (see Fig. 1). Note that the fingerprint of the endpoint of any edge can be computed in $O(1)$ time, again due to the rolling property of the hash function.

Thus our problem reduces to finding the heaviest cycle in a graph in which each vertex has in-degree and out-degree at most 1. This can be done in linear time, since such a graph is a collection of cycles and paths. The whole solution has $O(n \log n)$ time complexity due to the application of a map-like data structure.

The part of the above solution that causes difficulties in using the Dictionary of Basic Factors or the suffix tree/suffix array data structures is the fact that if $u_1u_2 \dots u_m$ is a substring of s then $u_2 \dots u_m u_1$ is (almost never) a substring of s that could be easily identified by its position in s.

We proceed to the second task example. Recall that in the pattern matching problem we are given two strings, a pattern and a text, and we are to find all the occurrences of the

[1] *A cyclic rotation* of a string is constructed by moving its first letter to its end, possibly multiple times. For example, there are three different cyclic rotations of `ABAABA`, namely `BAABAA`, `AABAAB` and `ABAABA`.

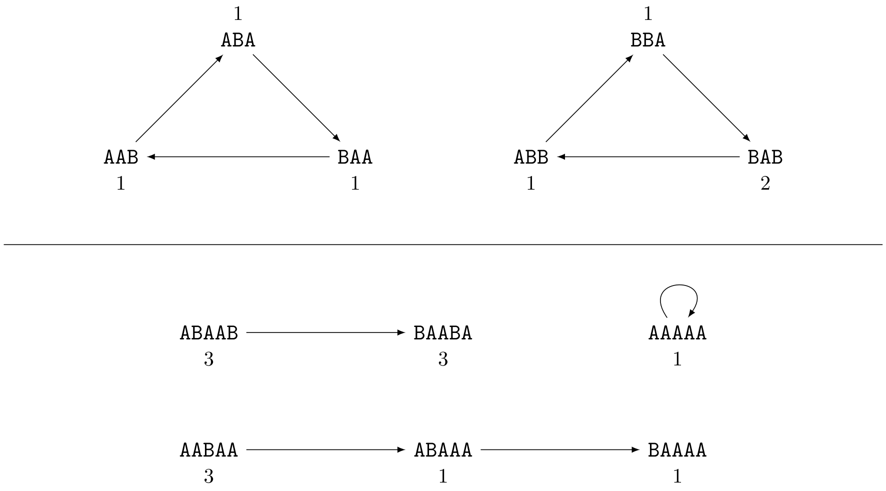


Fig. 1. The graphs of substrings for: $s =$ BABABBAAB and $m = 3$ (above) and $s =$ ABAABAABAABAAAAA and $m = 5$ (below)

pattern in the text. This problem has a number of efficient, linear time solutions, including the Morris–Pratt algorithm, Boyer–Moore algorithm, and a family of algorithms working in constant space. Also suffix trees/suffix arrays can be used for this problem. It is, however, rather difficult to extend any of these algorithms to work for the 2-dimensional variant of the problem:

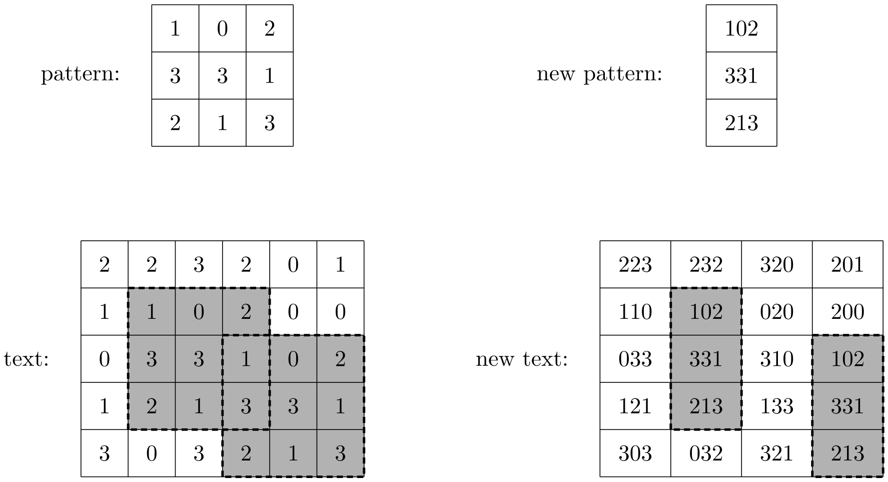
Problem 4: ***2-Dimensional Pattern Matching.*** Let the pattern be an array composed of $m \times m'$ symbols and the text be an array composed of $n \times n'$ symbols. Find all occurrences of the pattern as a subarray of the text.

The Rabin–Karp pattern matching algorithm that is based on string hashing can easily be extended to two dimensions. In each row of the text we compute the fingerprint of each substring of m symbols, this is done in $O(nn')$ time using the rolling property of the hash. We create a new array of size $n \times (n' - m' + 1)$ that contains these fingerprints. Thus the problem reduces to 1-dimensional pattern matching in the columns of the new array with the pattern composed of the fingerprints of the rows of the original pattern (see Fig. 2). This problem can be solved in linear time using the standard Rabin–Karp algorithm.

We obtain a linear time algorithm that appears simpler than, e.g., the known Bird/Baker 2-dimensional pattern matching algorithm which generalizes the Morris–Pratt algorithm to the case of multiple patterns, see Crochemore and Rytter (2003).

Below we list four other examples of tasks in which string hashing can be applied to obtain efficient solutions. The examples are ordered subjectively from the easiest to the hardest task. For each task a short comment on its solution is provided.

Problem 5: ***Quasi-Cyclic-Rotations.*** We are given two strings s and t of the same length n over English alphabet. We are to check if we can change exactly one letter in s so that



pattern:

1	0	2
3	3	1
2	1	3

new pattern:

102
331
213

text:

2	2	3	2	0	1
1	1	0	2	0	0
0	3	3	1	0	2
1	2	1	3	3	1
3	0	3	2	1	3

new text:

223	232	320	201
110	102	020	200
033	331	310	102
121	213	133	331
303	032	321	213

Fig. 2. Reduction of 2-dimensional pattern matching to 1-dimensional pattern matching in columns via string hashing (in this example $p = 10$, $M > 1000$)

it becomes a cyclic rotation of t. This is a task from Algorithmic Engagements 2007: `http://main.edu.pl/en/archive/pa/2007/pra`.

To check if s is an exact cyclic rotation of t, one could check if s occurs as a substring of the string tt – this is a pattern matching problem. For quasi-cyclic-rotations we can modify this approach: for each position $i = 1, 2, \ldots, n$ in tt we need to check if the substring u of length n starting at this position differs from s exactly at one position. For this it suffices to find the longest common prefix and the longest common suffix of u and s. The model solution computes these values in linear total time for all u's using the *PREF* table, also called the table of prefixes (see Section 3.2 of Crochemore and Rytter (2003)). An alternative solution applies binary search for the longest commom prefix/suffix and checks a candidate prefix/suffix using string hashing. This solution requires storing of fingerprints for s and tt and works in $O(n \log n)$ time.

Problem 6: ***Antisymmetry.*** We are given a string $t = t_1 \ldots t_n$ over $\{0, 1\}$ alphabet. For a substring u of t, by u^R we denote the reversed string u and by $\bar{u}$ we denote the negation of u obtained by changing all the zeroes to ones and ones to zeroes. A substring u of t is called *antisymmetric* if $u = \bar{u}^R$. We are to count the number of substrings of t that are antisymmetric (if the same antisymmetric substring occurs multiple times, we count each of its occurrences). This is a task from 17th Polish Olympiad in Informatics: `http://main.edu.pl/en/archive/oi/17/ant`.

Antisymmetric strings resemble palindromic strings. Recall Manacher's algorithm that finds in linear time, for each position i, the radius $R[i]$ of the longest even-length palindromic substring centered at this position (see Section 8.1 of Crochemore and Rytter (2003)). The model solution is based on a modification of Manacher's algorithm that finds, for each position i, the radius $R'[i]$ of the longest antisymmetric substring centered at this position. However, $R'[i]$ could alternatively be computed by applying binary search and checking if a candidate radius is valid via string hashing. Here string fingerprints for both t and $\bar{t}^R$ need to be stored. The solution based on string hashing has $O(n \log n)$ time complexity.

Problem 7: *Prefixuffix.* We are given a string $t = t_1 \dots t_n$ over English alphabet. A *prefixuffix* of t is a pair (p, s) of a prefix and a suffix of t, each of length at most $n/2$, such that s is a cyclic rotation of p. The goal of the task is to find the longest prefixuffix of t. This is a task from 19th Polish Olympiad in Informatics: `http://main.edu.pl/en/archive/oi/19/pre`.

The notion of a prefixuffix generalizes the notion of a *border*, which is a pair formed by an equal prefix and suffix of t. The model solution for this task works in linear time and is really tricky. Assume $2 \mid n$ and let $t = xy$, where x and y have the same length. Consider the word $Q(x, y)$ (a "crossover" of x and y) defined as $x_1 y_{n/2} x_2 y_{n/2-1} \dots x_{n/2} y_1$. Then the result is $l/2$, where l is the length of the longest prefix of $Q(x, y)$ that is a concatenation of two even-length palindromes. The value of l can be found in linear time using the output of Manacher's algorithm (already mentioned in Problem 6).

This solution deserves a short explanation. Note that (p, s) is a prefixuffix if $p = uv$ and $s = vu$ for some words u and v. Then $t = uvwzvu$ for some words w and z of equal length. Thus $x = uvw$, $y = zvu$ and $Q(x, y) = Q(u, u)Q(v, v)Q(w, z)$. Now it suffices to note that $Q(u, u)$ and $Q(v, v)$ are palindromes. E.g., if $t = \texttt{ababbabbabbaab}$ then $x = \texttt{ababbab}$, $y = \texttt{babbaab}$ and

$$Q(x, y) = \texttt{abbaaabbbbaabb} = \texttt{abba} \cdot \texttt{aabbbbaa} \cdot \texttt{bb}.$$

Here $l = 12$ which yields a prefixuffix of t of length 6: $(\texttt{ababba}, \texttt{abbaab})$.

An alternative solution using string hashing was much simpler to come up with. Recall that we need to find a representation $t = uvwzvu$ with uv as long as possible. To find u, we consider all the borders of t, and to find v, we need to know the longest border of each substring of t obtained by removing the same number of letters from the front and from the back of t, that is, $a[i] = \mathit{border}(t_i t_{i+1} \dots t_{n-i} t_{n-i+1})$. All the requested borders can be found using string hashing in linear time. In particular, the latter ones, $a[i]$, can be computed for $i = 1, 2, \dots, n/2$ by observing that $a[i] \geqslant a[i-1] - 2$.

Problem 8: *String Similarity.* We consider a family of strings, $S_1, \dots, S_n$, each of length l. We perform m operations, each of which consists in swapping a pair of letters across some pair of the strings. The goal is to compute, for each $i = 1, \dots, n$, how many strings among $\{S_j\}$ were equal to S_i at some moment in time, at maximum. This is a task from 15th Polish Olympiad in Informatics: `http://main.edu.pl/en/archive/oi/15/poc`.

Here the first idea that comes to mind is to use string hashing to identify which pairs of strings are equal at respective moments in time. Note that fingerprints of the strings can be updated easily when the letters are swapped. However, in the case of this task string hashing is only the beginning of the solution. An efficient, $O((nl + m) \log (nl + m))$

time solution requires keeping track of groups of equal strings. Roughly speaking, for each valid fingerprint we compute the number of strings with this fingerprint at each moment in time and then for each single string we find the maximum size of a group of equal strings it belongs to at any moment in time.

4. Final Remarks

In general, polynomial string hashing is a useful technique in construction of efficient string algorithms. One simply needs to remember to carefully select the modulus M and the variable of the polynomial p depending on the application. A good rule of thumb is to pick both values as prime numbers with M as large as possible so that no integer overflow occurs and p being at least the size of the alphabet.

There is a number of string processing problems in which hashing enables to present solutions that are competitive with the ones obtained using non-randomized algorithms. This includes pattern matching in one and multiple dimensions and searching for specific patterns, which includes palindromic substrings, cyclic rotations and common substrings. The major virtue of hashing is its $O(n)$ time and space consumption. However, one should always keep in mind that hashing is a heuristic algorithm.

References

Akhmedov, M. (2012). Anti-hash test. *Codeforces Blog*. `http://codeforces.com/blog/entry/4898`.

Allouche, J.-P., Shallit, J. (1999). The ubiquitous Prouhet-Thue-Morse sequence. In: Ding, C., Helleseth, T., Niederreiter, H. (Eds.), *Sequences and Their Applications, Proceedings SETA'98*, New York, Springer-Verlag, 1–16.

Crochemore, M., Hancart, C., Lecroq, T. (2007). *Algorithms on Strings*, Cambridge University Press.

Crochemore, M., Rytter, W. (2003). *Jewels of Stringology*, World Scientific.

Karp, R.M., Rabin, M.O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development – Mathematics and Computing*, 31(2), 249–260.

Weisstein, E.W. Modulo multiplication group. *MathWorld – a Wolfram Web Resource*. `http://mathworld.wolfram.com/ModuloMultiplicationGroup.html`.

J. Pachocki (1991), computer science student at Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Poland. Silver medalist of IOI'2009 in Plovdiv and winner of BOI'2009 in Stockholm, member of the runner-up team at the ACM ICPC World Finals 2012 in Warsaw. Problem setter for Polish Olympiad in Informatics and CEOI'2011 in Gdynia, Poland.

J. Radoszewski (1984), teaching assistant at Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Poland. Chair of the jury of Polish Olympiad in Informatics, Polish team leader at IOI 2008–2010 and 2012, former chair of the jury of CEOI'2011 in Gdynia and member of the HSC of IOI'2005 in Nowy Sącz, CEOI'2004 in Rzeszów, and BOI'2008 in Gdynia, Poland. His research interests focus on text algorithms and combinatorics.