



dmkozyrev's blog

[Tutorial] Rolling hash and 8 interesting problems [Editorial]

By [dmkozyrev](#), [history](#), 13 months ago, translation, , 

UPD: while I was translating this post from Russian to English, [dacin21](#) wrote his post, more advanced, [link](#). I hope that my post will help beginners, but in my post more rough estimates.

And in Russia we call **rolling hashes** as a **polynomial hashes**.

Hello, codeforces! This blogpost is written for all those who want to understand and use polynomial hashes and learn how to apply them in solving various problems. I will briefly write the theoretical material, consider the features of the implementation and consider some problems, among them:

1. Searching all occurrences of one string of length n in another string length m in $O(n + m)$ time
2. Searching for the largest common substring of two strings of lengths n and m ($n \geq m$) in $O((n + m \cdot \log(n)) \cdot \log(m))$ and $O(n \cdot \log(m))$ time
3. Finding the lexicographically minimal cyclic shift of a string of length n in $O(n \cdot \log(n))$ time
4. Sorting of all cyclic shifts of a string of length n in lexicographic order in $O(n \cdot \log(n)^2)$ time
5. Finding the number of sub-palindromes of a string of length n in $O(n \cdot \log(n))$ time
6. The number of substrings of string of length n that are cyclic shifts of the another string length m in $O((n + m) \cdot \log(n))$ time
7. The number of suffixes of a string of length n , the infinite extension of which coincides with the infinite extension of the given string for $O(n \cdot \log(n))$ (extension is a duplicate string an infinite number of times).
8. Largest common prefix of two strings length n with swapping two chars in one of them $O(n \cdot \log(n))$

Note 1. It is possible that some problems can be solved more quickly by other methods, for example, sorting the cyclic shifts — this is exactly what happens when constructing a suffix array, to search for all occurrences of one string in another will allow the Knut-Morris-Pratt algorithm, the Manaker algorithm works well with the sub-palindromes, and for own suffixes there is a prefix function.

Note 2. In the problems above, an estimate is made when a hash search is performed by sorting and binary searching. If you have your own hash table with open mixing or overflow chains, then you — lucky, boldly replace the hash search for a search in your hash table, but do not try to use `std::unordered_set`, as in practice the search in `std::unordered_set` loses sorting and binary search in connection with the fact that this piece obeys the C++ standard and has to guarantee a lot to the user, which is useful for industrial coding and, often, is useless in the competitive programming, so sorting and binary search for simple structures gain absolute primacy in C++ in speed of work, if not used additional own structures.

Note 3. In cases where comparison of elements is slow (for example, comparison by hash in $O(\log(n))$ time), in the worst case `std::random_shuffle` + `std::sort` always loses `std::stable_sort`, because `std::stable_sort` guarantees the minimum number of comparisons among all sorts (based on comparisons) for the worst case.

→ Pay attention

Before contest
[Codeforces Round #577 \(Div. 2\)](#)
 04:23:23
[Register now »](#)
 *has extra registration?

Like

151 people like this. [Sign Up](#) to see what your friends like.

→ Top rated

#	User	Rating
1	tourist	3645
2	Radewoosh	3403
3	LHiC	3336
4	wxhtxdy	3329
5	Benq	3320
6	Um_nik	3301
7	V--o--V	3275
8	mnvmar	3193
9	yutaka1999	3190
10	ainta	3180

[Countries](#) | [Cities](#) | [Organizations](#) [View all →](#)

→ Top contributors

#	User	Contrib.
1	Errichto	193
2	Radewoosh	184
3	rng_58	164
4	PikMike	163
5	Vovuh	160
6	majk	158
7	300iq	153
8	Um_nik	150
9	Petr	147
10	kostka	144

[View all →](#)

→ Find user

Handle:

Find

→ Recent actions

[Um_nik](#) → [Round #576 Editorial](#) 
[maths.noob](#) → [Determinant of Sparse Matrix??](#) 
[Errichto](#) → [TCO 2019 Round 4](#) 

The solution of the listed tasks will be given below, the source codes also.

As **advantages of polynomial hashing** I can notice that you often do not need to think, you can immediately take and write a naive algorithm to solve the problem and speed it up with polynomial hashing. Personally, firstly, I think about solution with a polynomial hash, perhaps that's why I'm blue.

Among the **disadvantages of polynomial hashing**: a) Too many operations getting remainder from the integer division, sometimes on the border with TLE for large problems, and b) on the codeforces in C++ programs are often small guarantees against hacking due to MinGW: `std::random_device` generates the same number every time, `std::chrono::high_resolution_clock` ticks in microseconds instead of nanoseconds. (The Cygwin compiler for windows wins against MinGW).

``UPD``: Solved a)

``UPD``: Solved b)

What is polynomial hashing?

Hash-function must assign to the object a **certain value** (hash) and possess the following properties:

1. If two objects are equal, then their hashes are equal.
2. If two hashes are equal, then the objects are equal with a high probability.

A **collision** is the very unpleasant situation of equality of two hashes for not equal objects. Ideally, when you choose a hash function, you need to ensure that **probability of collision lowest of possibles**. In practice — just a probability to successfully pass a set of tests to the task.

There are **two approaches** in choosing a function of a polynomial hash that depend on **directions**: from left to right and from right to left. To begin with, consider the option from left to right, and below, after describing the problems that arise in connection with the choice of the first option, consider the second.

Consider the sequence $\{a_0, a_1, \dots, a_{n-1}\}$. Under the polynomial hash from left to right for this sequence, we have in mind the result of calculating the following expression:

$$\text{hash}(a, p, m) = (a_0 + a_1 \cdot p + a_2 \cdot p^2 + \dots + a_{n-1} \cdot p^{n-1}) \bmod m$$

Here p and m — point (or base) and a hash module, respectively.

The conditions that we will impose: $\max(a_i) < p < m, \gcd(p, m) = 1$.

Note. If you think about interpreting the expression, then we match the sequences $\{a_0, a_1, \dots, a_{n-1}\}$ number of length n in the number in system with base p and take the remainder from its division by the number m , or the value of the polynomial $(n - 1)$ -th power with coefficients a_i at the point p modulo m . We'll talk about the choice of p and m later.

Note. If the value of $\text{hash}(a, p, m)$ (not by modulo), is placed in an integer data type (for example, a 64-bit type), then each sequence can be associated with this number. Then the comparison by greater / less / equal can be performed in $O(1)$ time.

Comparison by equal in $O(1)$ time

Now let's answer the question, how to compare arbitrary segments of sequence for $O(1)$? We show that to compare the segments of given sequence $\{a_0, a_1, \dots, a_{n-1}\}$, it is sufficient to compute the polynomial hash on **each prefix** of the original sequence.

Define a polynomial hash on the prefix as:

$$\text{pref}(k, a, p, m) = (a_0 + a_1 \cdot p + a_2 \cdot p^2 + \dots + a_{k-1} \cdot p^{k-1}) \bmod m$$

Briefly denote $\text{pref}(k, a, p, m)$ as $\text{pref}(k)$ and keep in mind that the final value is taken modulo m . Then:

[kasper08](#) → [Memory limit exceeded](#) 🔒

[JanchoMath](#) → [IOI mirror? live scoreboard?](#) 🔒

[bhasky_06](#) → [Inverse counting using segment tree](#) 🔒

[darkshadows](#) → [DP on Trees Tutorial](#) 🔒

[dalex](#) → [Gym — XII Samara Regional Intercollegiate Programming Contest](#) 🔒

[SPatrik](#) → [Codeforces Round #577 \(Div 2\)](#) 🔒

[we1erstrass](#) → [Polynomial calculation of N! modulo M](#) 🔒

[siddharth2000](#) → [is there a more efficient way of writing this code\(problem 762A\)](#) 🔒

[Malmo](#) → [Hackerrank frozen scoreboard issue](#) 🔒

[shpvb](#) → [Doing modular division when denominator and modulus not coprime](#) 🔒

[jmrh_1](#) → [Data sets of a problem](#) 🔒

[mohammad.h915](#) → [Space for member comments for each question](#) 🔒

[pilk](#) → [Guide to Competitive Programming and CSES Problem Set](#) 🔒

[aka.Sohieb](#) → [\[GYM\] 2019 ICPC Malaysian National Contest](#) 🔒

[MrDecomposition](#) → [Another copy of Codeforces \(Codeforc.es #2\)](#) 🔒

[chokudai](#) → [AtCoder Beginner Contest 135 Announcement](#) 🔒

[JetBrains](#) → [Kotlin Heroes Announcement](#) 🔒

[algo3rhythm](#) → [How to solve IOI 2019 Data Transfer Practice task](#) 🔒

[Musin](#) → [Checking lower constraints in networks with cycles](#) 🔒

[algo3rhythm](#) → [IOI 2019 Practice Task CYCLE](#) 🔒

[PikMike](#) → [Educational Codeforces Round 68 Editorial](#) 🔒

[TozanSoutherpacks](#) → [YouTube channel: Any requests for algorithm \(contest technique\) lectures?](#) 🔒

[Detailed →](#)

$$\text{pref}(0) = 0, \text{pref}(1) = a_0, \text{pref}(2) = a_0 + a_1 \cdot p, \text{pref}(3) = a_0 + a_1 \cdot p + a_2 \cdot p^2$$

General form:

$$\text{pref}(n) = a_0 + a_1 \cdot p + a_2 \cdot p^2 + \dots + a_{n-1} \cdot p^{n-1}$$

The polynomial hash on each prefix can be calculated in $O(n)$ time, using recurrence relations:

$$p^k = p^{k-1} \cdot p, \text{pref}(k+1) = \text{pref}(k) + a_k \cdot p^k$$

Let's say we need to compare two substrings that begin with i and j and have the length len , for equality:

$$a_i, a_{i+1}, \dots, a_{i+len-1} =? a_j, a_{j+1}, \dots, a_{j+len-1}$$

Consider the differences $\text{pref}(i+len) - \text{pref}(i)$ and $\text{pref}(j+len) - \text{pref}(j)$. It's not difficult to see that:

$$\begin{aligned} \text{pref}(i+len) - \text{pref}(i) &= a_i \cdot p^i + a_{i+1} \cdot p^{i+1} + \dots + a_{i+len-1} \cdot p^{i+len-1} \\ \text{pref}(j+len) - \text{pref}(j) &= a_j \cdot p^j + a_{j+1} \cdot p^{j+1} + \dots + a_{j+len-1} \cdot p^{j+len-1} \end{aligned}$$

We multiply the first equation by p^j , and the second by p^i . We get:

$$\begin{aligned} p^j \cdot (\text{pref}(i+len) - \text{pref}(i)) &= p^{i+j} \cdot (a_i + a_{i+1} \cdot p + \dots + a_{i+len-1} \cdot p^{len-1}) \\ p^i \cdot (\text{pref}(j+len) - \text{pref}(j)) &= p^{i+j} \cdot (a_j + a_{j+1} \cdot p + \dots + a_{j+len-1} \cdot p^{len-1}) \end{aligned}$$

We see that on the right-hand side of the expressions in brackets polynomial hashes were obtained from the segments of sequence:

$$a_i, a_{i+1}, \dots, a_{i+len-1} \text{ and } a_j, a_{j+1}, \dots, a_{j+len-1}$$

Thus, in order to determine whether the required segments of sequence have coincided, we need to check the following equality:

$$p^j \cdot (\text{pref}(i+len) - \text{pref}(i)) = p^i \cdot (\text{pref}(j+len) - \text{pref}(j))$$

One such comparison can be performed in $O(1)$ time, assuming the degree of p modulo precalculated. With the module m , we have:

$$p^j \cdot (\text{pref}(i+len) - \text{pref}(i)) \bmod m = p^i \cdot (\text{pref}(j+len) - \text{pref}(j)) \bmod m$$

Problem: Comparing one segment of sequence depends on the parameters of the other segment of sequence (from j).

The first solution of this problem (given by [veschii_nevstrui](#)) is based on multiplying the first equation by p^{-i} , and the second by p^{-j} . Then we get:

$$\begin{aligned} p^{-i} \cdot (\text{pref}(i+len) - \text{pref}(i)) &= a_i + a_{i+1} \cdot p + \dots + a_{i+len-1} \cdot p^{len-1} \\ p^{-j} \cdot (\text{pref}(j+len) - \text{pref}(j)) &= a_j + a_{j+1} \cdot p + \dots + a_{j+len-1} \cdot p^{len-1} \end{aligned}$$

We can see that in the right-hand parts of equations we get a polynomial hash from the needed segments of sequence. Then, the equality is checked as:

$$p^{-i} \cdot (\text{pref}(i+len) - \text{pref}(i)) = p^{-j} \cdot (\text{pref}(j+len) - \text{pref}(j))$$

To implement this, we need to find the inverse element for p modulo m . From the condition $\gcd(p, m) = 1$, the inverse element always exists. To do this, we need calculate or just know the value of the Euler function for the selected module $\varphi(m)$ and get power $\varphi(m) - 1$ for p . If we precalculate the powers of the inverse element for the selected module, then the comparison can be performed in $O(1)$ time.

The second solution we can use if we know the maximum lengths of compared segments of sequences. Let's denote the maximum length of compared lines as $mxPow$. We multiply 1-th equation by power $mxPow - i - len + 1$ of p , and 2-nd equation by $mxPow - j - len + 1$ power of p . We get:

$$p^{mxPow-i-len+1} \cdot (\text{pref}(i+len) - \text{pref}(i)) = p^{mxPow-len+1} \cdot (a_i + a_{i+1} \cdot p + \dots + a_{i+len-1} \cdot p^{len-1})$$

$$p^{mxPow-j-len+1} \cdot (\text{pref}(j+len) - \text{pref}(j)) = p^{mxPow-len+1} \cdot (a_j + a_{j+1} \cdot p + \dots + a_{j+len-1} \cdot p^{len-1})$$

We can note that on the right-hand sides of equals a polynomial hash of segments of sequence. Then, the equality is checked as follows:

$$p^{mxPow-i-len+1} \cdot (\text{pref}(i+len) - \text{pref}(i)) \bmod m = p^{mxPow-j-len+1} \cdot (\text{pref}(j+len) - \text{pref}(j)) \bmod m$$

This approach allows you to compare **one substring** of length len with **all substrings** of length len by **equality**, including **substrings of another string**, since the expression $p^{mxPow-i-len+1} \cdot (\text{pref}(i+len) - \text{pref}(i)) \bmod m$ for the substring of the length len starting at the position i , depends only on **the parameters of the current substring** i , len and **constant** $mxPow$, and not from the parameters of another substring.

Now consider **another approach** for choosing polynomial hash function. Define a polynomial hash on the prefix as:

$$\text{pref}(k, a, p, m) = (a_0 \cdot p^{k-1} + a_1 \cdot p^{k-2} + \dots + a_{k-2} \cdot p + a_{k-1}) \bmod m$$

Briefly denote $\text{pref}(k, a, p, m)$ as $\text{pref}(k)$ and keep in mind that the final value is taken modulo m . Then:

$$\text{pref}(0) = 0, \text{pref}(1) = a_0, \text{pref}(2) = a_0 \cdot p + a_1, \text{pref}(3) = a_0 \cdot p^2 + a_1 \cdot p + a_2$$

The polynomial hash on each prefix can be calculated in $O(n)$ time, using recurrence relations:

$$p^k = p^{k-1} \cdot p, \text{pref}(k+1) = \text{pref}(k) \cdot p + a_k$$

Let's say we need to compare two substrings that begin with i and j and have the length len , for equality:

$$a_i, a_{i+1}, \dots, a_{i+len-1} \stackrel{?}{=} a_j, a_{j+1}, \dots, a_{j+len-1}$$

Consider the differences $\text{pref}(i+len) - \text{pref}(i) \cdot p^{len}$ and $\text{pref}(j+len) - \text{pref}(j) \cdot p^{len}$. It's not difficult to see that:

$$\begin{aligned} \text{pref}(i+len) - \text{pref}(i) \cdot p^{len} &= a_i \cdot p^{len-1} + \dots + a_{i+len-2} \cdot p + a_{i+len-1} \\ \text{pref}(j+len) - \text{pref}(j) \cdot p^{len} &= a_j \cdot p^{len-1} + \dots + a_{j+len-2} \cdot p + a_{j+len-1} \end{aligned}$$

We see that on the right-hand side of the expressions in brackets polynomial hashes were obtained from the segments of sequence:

$$a_i, a_{i+1}, \dots, a_{i+len-1} \neq a_j, a_{j+1}, \dots, a_{j+len-1}$$

Thus, in order to determine whether the required segments of sequence have coincided, we need to check the following equality:

$$(\text{pref}(i+len) - \text{pref}(i) \cdot p^{\text{len}}) \bmod m = (\text{pref}(j+len) - \text{pref}(j) \cdot p^{\text{len}}) \bmod m$$

One such comparison can be performed in $O(1)$ time, assuming the degree of p modulo m precalculated.

Comparison by greater / less in $O(\log(n))$ time

Consider two substrings of (possibly) different strings of lengths len1 and len2 , ($\text{len1} \leq \text{len2}$), starting in the positions i and j respectively. Note that the ratio greater / less is determined **by the first non-equal symbol** in these substrings, and before this position strings are equal. Thus, we need to find the **position of the first non-equal symbol** by the **binary search method**, and then compare the found symbols. By comparing substrings to equality in $O(1)$ time, we can solve the problem of comparing substrings by greater / less in $O(\log(\text{len1}))$ time:

Pseudocode

Minimizing the probability of collision

Using [approximations in birthday problem](#), we get (perhaps a rough) estimate of the probability of collision. Suppose we compute a polynomial hash modulo m and, during the program, we need to compare n strings. Then the probability that the collision will occur:

$$p \approx 1 - \exp\left(-\frac{n^2}{2m}\right)$$

Hence it is obvious that m needs to be taken much more than n^2 . Then, approximating the exponential as Taylor series, we get the probability of collision on one test:

$$p \approx 1 - \exp\left(-\frac{n^2}{2m}\right) \approx \frac{n^2}{2m}$$

If we look at the problem of searching of occurrences of all cyclic shifts of one row in another string of lengths to 10^5 , then we can get 10^{15} comparisons of strings.

If we take a prime modulo of the order 10^9 , then we will not go through any of the maximum tests.

If we take a module of the order 10^{18} , then the probability of collision on one test is ≈ 0.001 . If the maximum tests are 100, the probability of collision in one of the tests is ≈ 0.1 , that is 10%.

If we take the module of the order 10^{27} , then on the 100 maximum tests the probability of collision is $\approx 10^{-10}$.

Conclusion: the higher the value of module, the more likely that we must pass the test. This probability not includes estimate probability of hack your solution.

Double polynomial hash

Of course, in real programs we can not take modules of the order 10^{27} . How to be? To the aid comes the Chinese theorem on the remainders. If we take two mutually simple modules m_1 and m_2 , then the ring of residues modulo $m = m_1 \cdot m_2$ is equivalent to the product of rings modulo m_1 and m_2 , i.e. there is bijection between them, based on the idempotents of the residue ring modulo m . In other words, if you calculate **hash₁** modulo m_1 and **hash₂** modulo m_2 , and then compare two segments of sequence with **hash₁** and **hash₂** simultaneously, then this is equivalent to comparing a polynomial hash modulo m . Similarly, we can take three mutually prime modules m_1, m_2, m_3 .

Features of the implementation

So, we came to the implementation of the above. Goal — **the minimum of the number of the remainder calculation from the integer division**, i.e. get two multiplications in a 64-bit type and one take the remainder from division in 64-bit type for one calculation of a double polynomial hash, **get a hash modulo about 10^{27}** and **protect the code from hacking** on codeforces

Selection of modules. It is advantageous to use a double polynomial hash on the modules `m1 = 1000000123` and `m2 = 2^64`. If you do not like this choice of `m1`, you can select `1000000321` or `2^31-1`, the main thing is to choose such a prime number so that the difference of the two residues lies within the signed 32-bit type (int). A prime number is more convenient, since the conditions `gcd(m1, m2) = 1` and `gcd(m1, p) = 1` are automatically provided. The choice of `m2 = 2^64` is not accidental. The C++ standard ensures that all calculations in the `unsigned long long` are executed modulo `2^64` automatically. Separately, the module `2^64` can not be taken, because there is an [anti-hash test](#), which does not depend on the choice of the hash point `p`. The module `m1` should be specified as **constant** to speed up the taking the remainder (the compiler (not MinGW) optimizes, replacing by multiplication and bitwise shifting).

Sequence encoding. If given a sequence of characters, consisting, for example, of small Latin letters, then you not need to encode anything, because each character already corresponds to its code. If a sequence of integers is given that is reasonable for a representation in memory of length, then it is possible to collect all the occurring numbers into one array, sort, delete the repeats and assign to each number in the sequence its index in sorted set. **Code zero is forbidden:** all sequences of the form `0,0,0, ..., 0` of different length will have the same polynomial hash.

Choosing of the base. As the base `p` it suffices to take any odd number satisfying the condition `max(a[i]) < p < m1` (odd, because then `gcd(p, 2 ^ 64) = 1`). If you **can be hacked**, then it is necessary to generate `p` randomly with every new program start, and generation with `std::srand(std::time(0))` and `std::rand()` is a bad idea, because `std::time(0)` ticks very slowly, and `std::rand()` does not provide enough uniformity. If the compiler **is not MINGW** (unfortunately, MinGW is installed on the codeforces), then you can use `std::random_device`, `std::mt19937`, `std::uniform_int_distribution<int>` (in cygwin on windows and gnu gcc on linux this set provides almost absolute randomness). If you were unlucky and you use only MinGW, then there is nothing left to do but replace `std::random_device` with `std::chrono::high_resolution_clock` and hope for the best (or is there a way to get some counter from the processor?). On MinGW, this timer ticks in microseconds, on cygwin and gnu gcc in nanoseconds.

Warranties against hacking. Odd numbers up to a module of the order of 10^9 are also of the order of 10^9 . The cracker will need to generate an anti-hash test for each odd number so that there is a collision in the space to 10^{27} , compile all the tests into one big test and hack you! This is if you do not use MinGW on Windows. On MinGW, the timer is ticking, as already mentioned, in microseconds. Knowing **when solution was started** on the server with an accuracy of seconds, it is possible for each of the 10^6 microseconds to calculate what random `p` was generated, and then the number of variants in the 1000 times less. If 10^9 is some cosmic number, then 10^6 already seems not so safe. If you use `std::time(0)` only 10^3 number of variants (milliseconds) — can be hacked! In the comments I saw that grandmasters know how to break a polynomial hash modulo of order of 10^{36} .

Ease of use. It is convenient to write a **universal object** for a polynomial hash and **copy** it to the task where it might be needed. It is better to write independently for your needs and goals in the style in which you write to understand the source code if necessary. All problems in this post are solved by copying the one class object. It is possible that there are specific tasks in which this does not work.

UPD: To speed up programs, you can quickly calculate the remainder of the divisions by modules $2^{31} - 1$ and $2^{61} - 1$. The main difficulty is multiplication. To understand the principle, look at [this post](#) by [dacin21](#) in **Larger modulo** and his [comment](#).

Mult mod $2^{61}-1$

Problem 1. Searching all occurrences of one string of length n in another string length m in $O(n + m)$ time

Problem 1 statement in English

Link on problem on acmp.ru.

[Solution and code](#)

Problem 2. Searching for the largest common substring of two strings of lengths n and m ($n \geq m$) in $O((n + m \cdot \log(n)) \cdot \log(m))$ and $O(n \cdot \log(m))$ time

[Link on problem on acm.timus.ru with length \$10^5\$.](#)

[Link on problem on spoj.com with length \$10^6\$.](#)

[Solution and code](#)

Problem 3. Finding the lexicographically minimal cyclic shift of a string of length n in $O(n \cdot \log(n))$ time

[Problem 3 statement in English](#)

[Link on problem 3](#)

[Solution and code](#)

Problem 4. Sorting of all cyclic shifts of a string of length n in lexicographic order in $O(n \cdot \log(n)^2)$ time

[Problem 4 statement in English](#)

[Link on problem 4](#)

[Note](#)[Solution and code](#)

Problem 5. Finding the number of sub-palindromes of a string of length n in $O(n \cdot \log(n))$ time

[Problem 5 statement in English](#)

[Link on problem 5 with length \$10^5\$](#)

[Link on problem 5 with length \$10^6\$](#)

[Solution and code](#)

Problem 6. The number of substrings of string of length n that are cyclic shifts of the another string length m in $O((n + m) \cdot \log(n))$ time

[Problem 6 statement in English](#)

[Link on problem 6](#)

[Solution and code](#)

Problem 7. The number of suffixes of a string of length n , the infinite extension of which coincides with the infinite extension of the given string for $O(n \cdot \log(n))$ (extension is a duplicate string an infinite number of times).

[Problem 7 statement in English](#)

[Link on problem 7](#)

[Solution and code](#)

Problem 8. Largest common prefix of two strings length n with swapping two chars in one of them in $O(n \cdot \log(n))$ time

[Link on problem](#)

You can check the editorial written by [a_kk](#) and [smokescreen](#) on this site for getting solution without hashes in $O(n)$ time.

[Solution and code](#)

That's all. I hope this post will help you to apply hashing and solve more difficult problems. I will be glad to any comments, corrections and suggestions from you. Share other problem and, possibly, your solutions, solutions without hashes. Thank you for reading this tutorial! There are many useful comments in russian under this blogpost, if you want, you can check them with google translate.

Links:

[Rolling Hash \(Rabin-Karp Algorithm\)](#)

rolling hashes, polynomial hash, hashes, tutorial, editorial, sorting, binary search, randomisation

+239

[dmkozyrev](#)

13 months ago

37



Comments (37)

[Write comment?](#)



a_kk

13 months ago, # |

+5

Awsome blog :D

→ [Reply](#)



dmkozyrev

13 months ago, # ^ |

+6

Thanks, I not added problem from your [contest](#), I'm sorry, I will add this

→ [Reply](#)



dmkozyrev

13 months ago, # ^ |

+6

Added your problem and my solution as a Problem 8

→ [Reply](#)

13 months ago, # ^ |

+3

feeling honoured(:p) after u mentioned our problem XD!!



a_kk

Although while making the problem I was sure a solution with binary search exists for this problem but the closest we can get to a binary search solution was this : [solution](#) which is surely wrong (:p).

Now after learning rolling hash, will try to implement your solution :).

→ [Reply](#)



dmkozyrev

13 months ago, # ^ | ← Rev. 6

+6

Please add a [anti-hash test](#) against single modulo 2^{64} for your problem. [Special generator for your problem](#). My old **accepted solution** with single hash **gets wrong answer** on this test.

Single hash answer = 130048 - WRONG ANSWER

Double hash answer = 2 - OK

UPD: added, now solution with 2^{64} gets wrong answer

And in this problem it is necessary to generate more (maybe random) tests with length **200000** and not-zero LCP, because single integer mod $\sim 10^9$ accepted. For example, in **problem 6** 46 tests, so solution with random single hash $\sim 10^9$ to this problem **gets wrong answer**.

UPD: I generated 100000 random tests with length

UPD. I generated 100000 random tests with length 200000 and no one collision for single modulo with random point.

→ [Reply](#)



a_kk

13 months ago, # [^](#) | [▲](#) 0 [▼](#)

Got your point..

Thanks a lot [dmkozyrev](#) :)

→ [Reply](#)

13 months ago, # | [▲](#) 0 [▼](#)



amulyagaur_111

Can you please explain the method used in problem 3 with some examples and pseudo-code? I'm not able to understand the method of Comparison by greater / less in $O(\log(n))$ time.

→ [Reply](#)

13 months ago, # [^](#) | [←](#) Rev. 6 [▲](#) 0 [▼](#)

Example for string `aaaaab`



dmkozyrev

I write substring as pair of position of start `i` and length of substring `len` because we can compare this pairs by equal in `O(1)` with polynomial hashes. I hope that this example can help for you to understand this technique

→ [Reply](#)



amulyagaur_111

13 months ago, # [^](#) | [▲](#) 0 [▼](#)

Thanks!! it helped.

→ [Reply](#)



amulyagaur_111

13 months ago, # | [▲](#) 0 [▼](#)

This question : QQ is same as problem 2. But here i'm getting TLE. [Submission](#). Any suggestions?

→ [Reply](#)

13 months ago, # [^](#) | [←](#) Rev. 2 [▲](#) 0 [▼](#)



dmkozyrev

Solution has $10^6 * \log_2(10^6)^2 \approx 4 * 10^8$ operations in worst case, no way gets accepted with `std::map`.

`std::map` is slowest way in C++ to find something in $O(\log(n))$

You need to write your own hashtable to getting $O(n * \log(n))$ solution with $O(1)$ time per search

→ [Reply](#)

4 months ago, # [^](#) | [▲](#) 0 [▼](#)

Actually it will depend on the range of elements of the array.



WaitingForTheEnd

(Assuming the number are uniformly distributed in the array).

Let say size of array is $N = 1e6$ and range of elements is 1 to $1e9$, then surely sort + binary_search is a better option.

But if $N = 1e6$ and range of elements is 1 to $1e4$ than `std::map` will beat sort + binary_search.

→ [Reply](#)



dmkozyrev

4 months ago, # [^](#) | [▲](#) +1 [▼](#)

I still think that sort + unique + binary search will beat `std::map`. Do you have any experiment?

→ [Reply](#)

4 months ago, # [^](#) | [▲](#) 0 [▼](#)

Oh! by adding unique I can accept it



WaitingForTheEnd

13 months ago, # ^ |

← Rev. 9 ▲ -10 ▼

Note that we are very close in time. We can solve this problem if we introduce new symbols of fixed width. Note that each character can take four different values. We can combine adjacent characters and encode them as one. For example, if we combine four characters, we will make $4 * 4 * 4 * 4 = 256$ different combinations. We will convert the string in this way and find the maximum length in this case. Let this be max1 . Then we convert back and look for $4 * \text{max1}$ to $4 * \text{max1} + 7$.



dmkozyrev

UPD: This improvement led to a solution in $O(n \log(n)^2)$ with smaller constant. **Accepted on SPOJ**, [code](#). On ideone.com $0.83s$ time (just uncomment gen in line 148 for getting it)

→ [Reply](#)

13 months ago, # |

▲ 0 ▼

Can you help me with this problem ?



coolreshab

Given a string s of length $\leq 10^5$ and $Q \leq 10^5$ queries, In each query you have to consider a sub-string from index L to R and print number of palindromic sub-strings of the query string.

I know the hash based approach for finding the number of palindromic sub-strings of a given string but I am not able to extend the logic for handling queries.

→ [Reply](#)

dmkozyrev

13 months ago, # ^ |

← Rev. 2 ▲ 0 ▼

I think that it can be solved with [palindromic tree](#). Maybe with hashes too, but I don't know how.

→ [Reply](#)

coolreshab

13 months ago, # ^ |

← Rev. 2 ▲ 0 ▼

Yes It can be solved using that also but I think there exist a solution which is hash based and make use of bit/ segment tree for handling range queries. It will be a great help if you can tell me that solution

→ [Reply](#)

13 months ago, # ^ |

▲ 0 ▼

I have an offline solution in $O(n * \log(n) + q * \log(n))$.



Nson

Pass i in decreasing order. Let k be the longest odd palindrome with center on i computed as problem 5. On a segment tree add 1 on every position on range $[i-k+1, i]$. Do the same for even length palindromes (center on i and $i+1$). Then we can answer a query $[L, R]$ such that $L = i$ by doing range sum query on the segment tree on range $[L, R]$.

→ [Reply](#)

dmkozyrev

13 months ago, # ^ |

▲ 0 ▼

Please tell in more detail how you will answer on query?

```
index: 0123456789
array: aabbababba - input string
odd: 1111252111 - max len of odd palindrome
with center i
even: 1020000200 - max len of even palindrome
with center {i, i+1}
```

total: 2333232111 - after applying all queries
of increment on segments

Query: $[l = 2, r = 3]$ - how we can **get** answer 3
with segment tree? **Sum** on range **is** 6.

→ [Reply](#)



Nson

13 months ago, # |

▲ +10 ▼

I think the complexity of problem 2 is $O((n + m * \log(n)) * \log(m))$

→ [Reply](#)



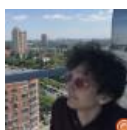
dmkozyrev

13 months ago, # ^ |

▲ 0 ▼

Changed, thanks

→ [Reply](#)



MStrechen

13 months ago, # ^ |

← Rev. 2 ▲ 0 ▼

I think you have to change it also in a russian version.

Actually, I believe that we could reach $O((n + m) \log m)$ asymptotics with hashtable (but not STL version, it's kinda slow). As far as I remember I had something like 3-4 times boost (maybe even more) with hashtable.

→ [Reply](#)



dmkozyrev

13 months ago, # ^ |

▲ 0 ▼

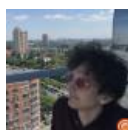
Can you, please, solve [this problem](#) with hashtable? I tried, but my hash table did not win against a accepted binary search

→ [Reply](#)

13 months ago, # ← Rev. 2

▲ 0 ▼

Well, I have two different variants of hashtables (with separate chaining and open adressing). Both got AC, but actually I needed to make some fixes because of size (first, I forgot about alphabet, then I forgot about size). But after this fixing separate chaining hashtable was good. Open adressing hashtable had some troubles with time before I changed the size of table up to $4 \cdot 10^6 + 37$. Finally I have 22ms (I hope it is ms) with open adressing hashtable and up to 15.97ms (with some experiments, my first result was 17.31ms) with separate chaining one.



MStrechen

I don't think that my hashtable is the fastest in the world, but here is my old-but-gold code, maybe you are interested:
<https://ideone.com/hxlv0>

UPD: I also have TL with binary search, so I think I can improve my code performance by changing the algorithm of string hashing.

→ [Reply](#)



dmkozyrev

13 months ago, # ^ | ▲ 0 ▼

I passed the solution with binary search only after I reduced the hidden constant, compressing

four characters into one. [More in](#)

your characters into one. More in
this comment

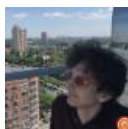
I think that time in seconds —
time of working on all test cases
summary. For example, my
binary search solution gets
19.19 time on SPOJ.

Your solution takes 0.8 seconds
on [ideone.com](#) on test `10^6`
len, this is very fast hashtable,
thanks! My solution with
hashtable on this test takes 2.5
seconds, with binary search 1.6
seconds, with compressing 4
chars to 1 and binary search
0.66 seconds.

→ [Reply](#)

13 months ago, # 0

Maybe your
implementation of
hashtable uses
something like
`vector<vector<...>`
`>` that can slow down
solution because of
memory
allocations/deallocations.



MStreichen

My implementation uses
something like linked lists
stored in continuous
section of memory
without changing the size
so it could allocate it
once and then just reuse
it.

→ [Reply](#)

13 months ago, # 0

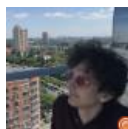


dmkozyrev

Thanks! I passed
a problem with
my open
addressed hash
table based on
`std::array`.

→ [Reply](#)

13 months ago, # 0



MStreichen

Now you
can change
the
asymptotics
of the
problem 2 :)

→ [Reply](#)



dmkozyrev

12 months ago, # |

← Rev. 2 0

Added rolling hash right-to-left and code for fast multiplication of remainders
modulo $2^{61} - 1$ with link to the author

Добавлен полиномиальный хэш справа-налево и код для быстрого
вычисления остатка по модулю $2^{61} - 1$ после умножения двух остатков со
ссылкой на автора

События на сайте

→ [Reply](#)

9 months ago, # ^ |

▲ 0 ▼

Hi @dmkozyrev



varunvats32

In the second solution for rolling hash. You have mentioned that on both sides we need to multiply by $\text{MaxPow} - i - \text{len} + 1$. Do you think we can skip even len also, and make it $\text{MaxPow} - i + 1$? Because anyhow if both the substrings are of same len, we can check the equality without len also.

→ [Reply](#)



dmkozyrev

9 months ago, # ^ |

▲ 0 ▼

Yes, your approach — fixing least power of base in hash, and it's working

Example

→ [Reply](#)

5 months ago, # |

← Rev. 2

▲ 0 ▼



_greymatter

".. we take a module of the order 10^{18} , then the probability of collision on one test is ≈ 0.001 . If the maximum tests are 100, the probability of collision in one of the tests is ≈ 0.1 , that is 10%."

If the probability of collision on one test is 0.001, isn't the probability of at least one collision in 100 tests = $1 - (0.999)^{100}$?

→ [Reply](#)

5 months ago, # ^ |

▲ 0 ▼

I used **Taylor Series**. When n is large, but p is small, we can just multiply p^n by n :

$$p^n \approx n \cdot p$$



dmkozyrev

$$1 - (1-p)^n \approx 1 - (1 - n \cdot p + O(p^2)) \approx n \cdot p + O(p^2)$$

Lets calculate original formula in wolfram:

$$1 - (1-0.001)^{100} \approx 0.095$$

From Taylor approximation we got:

$$n \cdot p + O(p^2) \approx 0.1$$

→ [Reply](#)



_greymatter

5 months ago, # ^ |

▲ 0 ▼

Got it, thanks!

→ [Reply](#)



seven_triple

12 days ago, # |

▲ 0 ▼

can some explain me the point of **advantages of rolling hashing**. both points that are given into advantages of rolling hashing.

→ [Reply](#)

12 days ago, # |

▲ 0 ▼



Ramprosad

How Can I Find The the Kth Lexicographical Minimum Substring of a given string S ??

Please help....

→ [Reply](#)

The only programming contests Web 2.0 platform

Server time: Aug/04/2019 16:29:33^{UTC+8} (g2).

Desktop version, switch to [mobile version](#).

[Privacy Policy](#).

Supported by



ITMO UNIVERSITY