dacin21's blog

On the mathematics behind rolling hashes and anti-hash tests

By dacin21, history, 13 months ago,

This blog assumes the reader is familiar with the basic concept of rolling hashes. There are some math-heavy parts, but one can get most of the ideas without understanding every detail.

The main focus of this blog is on how to choose the rolling-hash parameters to avoid getting hacked and on how to hack codes with poorly chosen parameters.

Designing hard-to-hack rolling hashes

Recap on rolling hashes and collisions

Recall that a rolling hash has two parameters (p, a) where p is the modulo and $0 \le a < p$ the base. (We'll see that p should be a big prime and a larger than the size $|\Sigma|$ of the alphabet.) The hash value of a string $S = s_0 \dots s_{n-1}$ is given by

$$h(S) := \left(\sum_{i=0}^{n-1} a^{n-1-i} s_i\right) \mod p$$

For now, lets consider the simple problem of: given two strings S, T of equal length, decide whether they're equal by comparing their hash values $h(S), h(T)$. Our algorithm declares S and T to be equal iff $h(S) = h(T)$. Most rolling hash solutions are built on multiple calls to this subproblem or rely on the correctness of such calls.

Let's call two strings S, T of equal length with $S \neq T$ and $h(S) = h(T)$ an **equal-length collision**. We want to avoid equal-length collisions, as they cause our algorithm to incorrectly assesses S and T as equal. (Note that our algorithms never incorrectly assesses strings a different.) For fixed parameters and reasonably small length, there are many more strings than possible hash values, so there always are equal-length collisions. Hence you might think that, for any rolling hash, there are inputs for which it is guaranteed to fail.

Luckily, randomization comes to the rescue. Our algorithm does not have to fix (p, a), it can randomly pick then according to some scheme instead. A scheme is **reliable** if we can prove that for arbitrary two string $S, T, S \neq T$ the scheme picks (p, a) such that $h(S) \neq h(T)$ with high probability. Note that the probability space only includes the random choices done inside the scheme; the input (S, T) is arbitrary, fixed and not necessarily random. (If you think of the input coming from a hack, then this means that no matter what the input is, our solution will not fail with high probability.)

I'll show you two reliable schemes. (Note that just because a scheme is reliable does not mean that your implementation is good. Some care has to be taken with the random number generator that is used.)

Randomizing base

This part is based on a blog by rng_58. His post covers a more general hashing problem and is worth checking out.

This scheme uses a fixed **prime** p (i.e. 10^9+7 or $4 \cdot 10^9+7$) and picks a uniformly at random from $\{0, \dots, p-1\}$. Let A be a random variable for the choice of a.

To prove that this scheme is good, consider two strings (S, T) of equal length and do some calculations

$$\begin{aligned} & h(S) = h(T) \\ \Leftrightarrow\quad & \left(\sum_{i=0}^{n-1} A^{n-1-i} S_i\right) \mod p = \left(\sum_{i=0}^{n-1} A^{n-1-i} T_i\right) \mod p \\ \Leftrightarrow\quad & \sum_{i=0}^{n-1} A^{n-1-i} S_i \equiv \sum_{i=0}^{n-1} A^{n-1-i} T_i \pmod p \\ \Leftrightarrow\quad & P(A) := \sum_{i=0}^{n-1} A^{n-1-i}(S_i - T_i) \equiv 0 \pmod p \end{aligned}$$

Note that the left-hand side, let's call it $P(A)$, is a polynomial of degree $\leq n - 1$ in A. P is non-zero as $S \neq T$. The calculations show that $h(S) = h(T)$ if and only if A is a root of $P(A)$.

As p is prime and we are doing computations $\mod p$, we are working in a field. Over a field, any polynomial of degree $\leq n - 1$ has at most $n - 1$ roots. Hence there are at most $n - 1$ choices of a that lead to $h(S) = h(T)$. Therefore

$$\Pr\left[h(S) = h(T)\right] = \Pr\left[P(A) = 0\right] \leq \frac{n-1}{p}$$

So for any two strings (S, T) of equal length, the probability that they form an equal-length collision is at most $\frac{n-1}{p}$. This is around 10^{-4} for $n = 10^5, p = 10^9 + 7$. Picking larger primes such as $2^{31} - 1$ or $4 \cdot 10^9 + 7$ can improve this a bit, but one needs more care with overflows.

Tightness of bound

For now, this part only applies to primes with smooth $p - 1$, so it doesn't work for $p = 10^9 + 7$ for example. It would be interesting to find a construction that is computable and works in the general case.

The bound $\frac{n-1}{p}$ for this scheme is actually tight if $n - 1 \mid p - 1$. Consider $S = \mathrm{ba}\ldots\mathrm{aa}$ and $T = \mathrm{aa}\ldots\mathrm{ab}$ with

$$P(A) = A^{n-1} - 1$$

As p is prime, $\frac{\mathbb{Z}}{p\mathbb{Z}}^\times$ is cyclic of order $p - 1$, hence there is a subgroup $G \subseteq \frac{\mathbb{Z}}{p\mathbb{Z}}^\times$ of order $n - 1$. Any $g \in G$ then satisfies $g^{n-1} = 1$, so $P(A)$ has $n - 1$ distinct roots.

Randomizing modulo

This scheme fixes a base $a \geq |\Sigma|$ and a bound $N > a$ and picks a **prime** p uniformly at random from $[N, 2N - 1]$.

To prove that this scheme is good, again, consider two strings (S, T) of equal length and do some calculations

$$\begin{aligned} & h(S) = h(T) \\ \Leftrightarrow\quad & \left(\sum_{i=0}^{n-1} a^{n-1-i} S_i\right) \mod p = \left(\sum_{i=0}^{n-1} a^{n-1-i} T_i\right) \mod p \\ \Leftrightarrow\quad & \sum_{i=0}^{n-1} a^{n-1-i} S_i \equiv \sum_{i=0}^{n-1} a^{n-1-i} T_i \pmod p \\ \Leftrightarrow\quad & X := \sum_{i=0}^{n-1} a^{n-1-i}(S_i - T_i) \equiv 0 \pmod p \end{aligned}$$

As $X \equiv 0 \pmod p$, $p \mid X$. As we chose a large enough, $X \neq 0$. Moreover $|X| < a^n$. An upper bound for the number of distinct prime divisors of X in $[N, 2N - 1]$ is given by $\log_N(|X|) = \frac{n \ln(a)}{\ln N}$. By the prime density theorem, there are around $\frac{N}{\ln N}$ primes in $[N, 2N - 1]$. Therefore

$$\Pr\left[h(S) = h(T)\right] = \Pr\left[p \mid X\right] \leq\sim \frac{n \ln(a)}{N}$$

Note that this bound is slightly worse than the one for randomizing the base. It is around $3 \cdot 10^{-4}$ for $n = 10^5, a = 26, N = 10^9$.

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

pllk → 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

algo3rythm → How to solve IOI 2019 Data Transfer Practice task

Musin → Checking lower constraints in networks with cycles

algo3rythm → IOI 2019 Practice Task CYCLE

PikMike → Educational Codeforces Round 68 Editorial

TozanSoutherpacks → YouTube channel: Any requests for algorithm (contest technique) lectures?

Detailed →

How to randomize properly

The following are good ways of initializing your random number generator.

- high precision time.

```
    chrono::duration_cast<chrono::nanoseconds>
(chrono::high_resolution_clock::now().time_since_epoch()).count();
    chrono::duration_cast<chrono::nanoseconds>
(chrono::steady_clock::now().time_since_epoch()).count();
```

Either of the two should be fine. (In theory, `high_resolution_clock` should be better, but it somehow has lower precision than `steady_clock` on codeforces??)

- processor cycle counter

```
    __builtin_ia32_rdtsc();
```

- some heap address converted to an integer

```
    (uintptr_t) make_unique<char>().get();
```

- processor randomness (needs either pragma or asm) (Thanks **halyavin** for suggesting this.)

```
    // pragma version
    #pragma GCC target ("rdrnd")
    uint32_t rdrand32(){
        uint32_t ret;
        assert(__builtin_ia32_rdrand32_step (&ret));
        return ret;
    }

    // asm version
    uint32_t rd() {
      uint32_t ret;
      asm volatile("rdrand %0" :"=a"(ret) ::"cc");
      return ret;
    }
```

If you use a C++11-style rng (you should), you can use a combination of the above

```
    seed_seq seq{
        (uint64_t) chrono::duration_cast<chrono::nanoseconds>
(chrono::high_resolution_clock::now().time_since_epoch()).count(),
        (uint64_t) __builtin_ia32_rdtsc(),
        (uint64_t) (uintptr_t) make_unique<char>().get()
    };
    mt19937 rng(seq);
    int base = uniform_int_distribution<int>(0, p-1)(rng);
```

Note that this does internally discard the upper 32 bits from the arguments and that this doesn't really matter, as the lower bits are harder to predict (especially in the first case with chrono.).

See the section on 'Abusing bad randomization' for some bad examples.

Extension to multiple hashes

We can use multiple hashes (Even with the same scheme and same fixed parameters) and the hashes are independent so long as the random samples are independent. If the single hashes each fail with probability at most $\alpha_1, \ldots, \alpha_k$, the probability that all hashes fail is at most $\prod_{i=1}^{k} \alpha_i$.

For example, if we use two hashes with $p = 10^9 + 7$ and randomized base, the probability of a collision is at most 10^{-8}; for four hashes it is at most 10^{-16}. Here the constants from slightly larger primes are more significant, for $p = 2^{31} - 1$ the probabilities are around $2.1 \cdot 10^{-9}$ and $4.7 \cdot 10^{-18}$.

Larger modulo

Using larger (i.e. 60 bit) primes would make collision less likely and not suffer from the accumulated factors of n in the error bounds. However, the computation of the rolling hash gets slower and more difficult, as there is no `__int128` on codeforces.

One exception to this is the Mersenne prime $p = 2^{61} - 1$; we can reduce $\bmod p$ by using bitshifts instead. (Thanks **dmkozyrev** for suggesting this.) The following code computes $a \cdot b \mod p$ without `__int128` and is only around 5 % slower than a 30 bit hash with modulo.
Code

A smaller factor can be gained by using unsigned types and $p = 4 \cdot 10^9 + 7$.

Note that $p = 2^{64}$ (overflow of unsigned long long) is **not prime** and can be **hacked** regardless of randomization (see below).

Extension to multiple comparisons

Usually, rolling hashes are used in more than a single comparison. If we rely on m comparison and the probability that a single comparison fails is p then the probability that any of the fail is at most $m \cdot p$ by a union bound. Note that when $m = 10^5$, we need at least two or three hashes for this to be small.

One has to be quite careful when estimating the number comparison we need to succeed. If we sort the hashes or put them into a set, we need to have pair-wise distinct hashes, so for n string a total of $\binom{n}{2}$ comparisons have to succeed. If $n = 3 \cdot 10^5$, $m \approx 4.5 \cdot 10^9$, so we need three or four hashes (or only two if we use $p = 2^{61} - 1$).

Extension to strings of different length

If we deal with strings of different length, we can avoid comparing them by storing the length along the hash. This is not necessarily however, if we assume that **no character hashes to** 0. In that case, we can simple imagine we prepend the shorter strings with null-bytes to get strings of equal length without changing the hash values. Then the theory above applies just fine. (If some character (i.e. 'a') hashes to 0, we might produce strings that look the same but aren't the same in the prepending process (i.e. 'a' and 'aa').)

Computing anti-hash tests

This section cover some technique that take advantage of common mistakes in rolling hash implementations and can mainly be used for hacking other solutions. Here's a table with a short summary of the methods.

Name	**Use case**	**Runtime**	**String length**	**Notes**
Thue-Morse	Hash with overflow	$\Theta(1)$	2^{10}	Works for all bases simultaneously.
Birthday	Small modulo	$\Theta(\sqrt{p}\log p)$	$\approx 2\log_{\lvert\Sigma\rvert}(p)$	Can find multiple collisions.
Tree	Large modulo	$\Theta(2^{\sqrt{2\lg p}})$	$2^{\sqrt{2\lg p}} + 1$	faster; longer strings
Multi-tree	Large modulo	$\approx \Theta\left((2^{\sqrt{2\lg_m p}} + \log_{\lvert\Sigma\rvert}(m)) \cdot m\log m\right)$	$\approx 2^{\sqrt{2\lg_m p}} + \log_{\lvert\Sigma\rvert}(m)$	slower; shorter strings

Name	Use case	Runtime	String length	Notes
Lattice reduction	Medium-large alphabet, Multiple hashes	$\approx \Theta\left(length^3\right)$	$\approx \sum_{i=0}^{n-1} \log_{\|\Sigma\|}(p_n)$	Great results for $\|\Sigma\| = 26$, good against multiple hashes. Bad on binary alphabet.
Composition	Multiple hashes	Sum of single runtimes	Product of single string lengths	Combines two attacks.

Single hashes

Thue–Morse sequence: Hashing with unsigned overflow ($p = 2^{64}$, q arbitrary)

One anti-hash test that works for *any* base is the Thue–Morse sequence, generated by the following code.

code

See this blog for a detailed discussion. Note that the bound on the linked blog can be improved slightly, as X^2 - 1 is always divisible by 8 for odd X. (So we can use $Q = 10$ instead of $Q = 11$.)

Birthday-attack: Hashing with 32-bit prime and fixed base ($p < 2^{32}$ fixed, q fixed)

Hashes with a single small prime can be attacked via the birthday paradox. Fix a length l, let $k = 1 + \sqrt{(2\ln 2)p}$ and pick k strings of length l uniformly at random. If l is not to small, the resulting hash values will approximately be uniformly distributed. By the birthday paradox, the probability that all of our picked strings hash to different values is

$$\prod_{i=0}^{k-1}\left(1-\frac{i}{p}\right) < \prod_{i=0}^{k-1}\left(e^{-\frac{i}{p}}\right) = e^{-\frac{k(k-1)}{2p}} < e^{-\ln 2} = \frac{1}{2}$$

Hence with probability $> \frac{1}{2}$ we found two strings hashing to the same value. By repeating this, we can find an equal-length collision with high probability in $\tilde{\Theta}\left(\sqrt{p}\right)$. In practice, the resulting strings can be quite small (length ≈ 6 for $p = 10^9 + 7$, not sure how to upper-bound this.).

More generally, we can compute m strings with equal hash value in $\tilde{O}\left(m \cdot p^{1-\frac{1}{m}}\right)$ using the same technique with $r = m \cdot p^{1-\frac{1}{m}}$.

Tree-attack: Hashing with larger prime and fixed base (p fixed, q fixed)

Thanks Kaban-5 and pavel.savchenkov for the link to some Russian comments describing this idea.

For large primes, the birthday-attack is to slow. Recall that for two strings (S, T) of equal length

$$\begin{aligned} & h(S) = h(T) \\ \Leftrightarrow \quad & \sum_{i=0}^{n-1} a^{n-1-i}(S_i - T_i) \equiv 0 \pmod{p} \\ \Leftrightarrow \quad & \sum_{i=0}^{n-1} a^{n-1-i}(\alpha_i) \equiv 0 \pmod{p} \\ \Leftarrow \quad & \sum_{i=0}^{n-1} \left(a^{n-1-i} \mod p\right) \cdot \alpha_i = 0 \end{aligned}$$

where $\alpha_i = S_i - T_i$ satisfies $-|\Sigma| \leq \alpha_i \leq |\Sigma|$. The tree-attack tries to find $\alpha_i \in \{-1, 0, 1\}$ such that

$$\sum_{i=0}^{n-1} (a^{n-1-i} \mod p) \cdot \alpha_i = 0$$

The attack maintains clusters C_1, ..., C_k of coefficients. The **sum** $S(C)$ of a cluster C is given by

$$S(C) = \sum_{i \in C} (a^{n-1-i} \mod p) \cdot \alpha_i$$

We can merge two clusters C_1 and C_2 to a cluster C_3 of sum $S(C_1) - S(C_2)$ by multiplying all the α_i from C_2 with -1 and joining the set of coefficients of C_1 and C_2. This operation can be implemented in constant time by storing the clusters as binary trees where each node stores its sum; the merge operation then adds a new node for C_3 with children C_1 and C_2 and sum $S(C_1) - S(C_2)$. To ensure that the $S(C_3) \geq 0$, swap C_1 and C_2 if necessary. The values of the α_i are not explicitly stored, but they can be recomputed in the end by traversing the tree.

Initially, we start with $n = 2^k$ and each $\alpha_i = 1$ in its own cluster. In a phase, we first sort the clusters by their sum and then merge adjacent pairs of clusters. If we encounter a cluster of sum 0 at any point, we finish by setting all α_j not in that cluster to 0. If we haven't finished after k phases, try again with a bigger value of k.

For which values of k can we expect this to work? If we assume that the sums are initially uniformly distributed in $\{0, \ldots, p-1\}$, the maximum sum should decrease by a factor $\sim 2^{k-i}$ in phase i. After k phases, the maximum sum is around $\frac{p}{2^{\binom{k}{2}}}$, so $k \approx \sqrt{2 \lg p} + 1$ works. This produces strings of length $n = 2^{\sqrt{2 \lg p} + 1}$ in $\Theta(n)$ time. (A more formal analysis can be found in the paper 'Solving Medium-Density Subset Sum Problems in Expected Polynomial Time', section 2.2. The problem and algorithms in the paper are slightly different, but the approach similar.)

Multi-tree-attack

While the tree-attacks runs really fast, the resulting strings can get a little long. ($n = 2048$ for $p = 2^{61} - 1$.) We can spend more runtime to search for a shorter collision by storing the smallest m sums we can get in each cluster. (The single-tree-attack just uses $m = 1$.) Merging two clusters can be done in $\Theta(m \log m)$ with a min-heap and a $2m$-pointer walk. In order to get to m strings ASAP, we allow all values $\alpha_i \in \{-|\Sigma|, \ldots, |\Sigma|\}$ and exclude the trivial case where all α_i are zero.

Analysing the expected value of k for this to work is quite difficult. Under the optimistic assumption that we reach m sums per node after $\log_{|\Sigma|}(m)$ steps, that the sums decrease as in the single tree attack and that we can expected a collision when they get smaller than m^2 by the birthday-paradox, we get $k = \sqrt{2 \frac{\lg p}{\lg m}} + \log_{|\Sigma|}(m)$. (A more realistic bound would be $k = \frac{\lg p}{\lg m} + \log_{|\Sigma|}(m)$, which might be gotten by accounting for the birthday-paradox in the bound proven in the paper 'On Random High Density Subset Sums', Theorem 3.1.)

In practice, we can use $m \approx 10^5$ to find a collision of length 128 for $|\Sigma| = 2$, $p = 2^{61} - 1$ in around 0.4 seconds.

Lattice-reduction attack: Single or multiple hashes over not-to-small alphabet

Thanks to **hellman_** for mentioning this, check out his write-up on this topic here. There's also a write-up by someone else here.

As in the tree attack, we're looking for $\alpha_i \in \{-|\Sigma|, \ldots, |\Sigma|\}$ such that

$$\sum_{i=0}^{n-1} (a^{n-1-i} \mod p) \cdot \alpha_i \equiv 0 \pmod p$$

The set

$$\left\{ (\alpha_0, \ldots, \alpha_{n-1}, \beta) \middle| \beta \equiv \sum_{i=0}^{n-1} (a^{n-1-i} \mod p) \cdot \alpha_i \right\}$$

forms a **lattice** (A free $\mathbb{Z}$-module embedded in a subspace of $\mathbb{R}^n$.) We're looking for an element in the lattice such that $\beta = 0$ and $|\alpha_i| \leq |\Sigma|$. We can penalize non-zero values of β by considering

$$\beta = 10^5 \left(\left(\sum_{i=0}^{n-1} (a^{n-1-i} \mod p) \cdot \alpha_i \right) \mod p \right)$$

instead, then we seek to minimize $\max(|\alpha_0|, \ldots, |\alpha_{n-1}|, |\beta|)$. Unfortunately, this optimization problem is quite hard, so we try to minimize

$$\alpha_0^2 + \ldots + \alpha_{n-1}^2 + \beta^2$$

instead. The resulting problem is still hard, possibly NP-complete, but there are some good approximation algorithms available.

Similar to a vector space, we can define a basis in a lattice. For our case, a basis is given by

$$\left\{ e_\beta + 10^5 (a^{n-1-i} \mod p)\, e_{\alpha_i} \middle| 0 \leq i < n \right\} \cup \{p \cdot 10^5 e_\beta\}$$

A lattice reduction algorithm takes this basis and transforms it (by invertible matrices with determinant ± 1) into another basis with approximately shortest vectors. Implementing them is quite hard (and suffers from precision errors or bignum slowdown), so I decided to use the builtin implementation in sage.

code

Sage offers two algorithms: `LLL` and `BKZ`, the former is faster but produces worse approximations, especially for longer strings. Analyzing them is difficult, so I experimented a bit by fixing $|\Sigma| = 26$, $p = 2^{61} - 1$ and fixing $a_1, \ldots, a_n$ randomly and searching for a short anti-hash test with both algorithms. The results turned out really well.

Experiment data

Note that this attack does not work well for small (i.e binary) alphabets when $n > 1$ and that the characters have to **hash to consecutive** values, so this has to be the first attack if used in a composition attack.

Composition-attack: Multiple hashes

Credit for this part goes to ifsmirnov, I found this technique in his jngen library.

Using two or more hashes is usually sufficient to protect from a direct birthday-attack. For two primes, there are $N = p_1 \cdot p_2$ possible hash values. The birthday-attack runs in $\tilde{\Theta}\left(\sqrt{N}\right)$, which is $\approx 10^{10}$ for primes around 10^9. Moreover, the memory usage is more than $\sqrt{(2 \ln 2) N} \cdot 8$ bytes (If you only store the hashes and the rng-seed), which is around 9.5 GB.

The key idea used to break multiple hashes is to break them one-by-one.

- First find an equal-length collision (by birthday-attack) for the first hash h_1, i.e. two strings $S, T, S \neq T$ of equal length with $h_1(S) = h_1(T)$. Note that strings of equal length built over the alphabet S, T (i.e. by concatenation of some copies of S with some copies of T and vice-versa) will now hash to the same value under h_1.
- Then use S and T as the alphabet when searching for an equal-length collision (by birthday-attack again) for the second hash h_2. The result will automatically be a collision for h_1 as well, as we used S, T as the alphabet.

This reduces the runtime $\tilde{\Theta}\left(\sqrt{p_1} + \sqrt{p_2}\right)$. Note that this also works for combinations of a 30-bit prime hash and a hash mod 2^{64} if we use the Thue–Morse sequence in place of the second birthday attack. Similarly, we can use tree- instead of birthday-attacks for larger modulos.

Another thing to note is that string length grows rapidly in the number of hashes. (Around $2 \log_{|\Sigma|}\left(\sqrt{(2\ln 2)p_1}\right) \cdot \log_2\left(\sqrt{(2\ln 2)p_2}\right) \cdots \log_2\left(\sqrt{(2\ln 2)p_k}\right)$, the alphabet size is reduced to 2 after the first birthday-attack. The first iteration has a factor of 2 in practice.) If we search for more than 2 strings with equal hash value in the intermediate steps, the alphabet size will be bigger, leading to shorter strings, but the runtime of the birthday-attacks gets slower ($\tilde{\Theta}\left(p^{\frac{2}{3}}\right)$ for 3 strings, for example.).

Abusing bad randomization

On codeforces, quite a lot of people randomize their hashes. (Un-)Fortunately, many of them do it an a suboptimal way. This section covers some of the ways people screw up their hash randomizations and ways to hack their code.

This section applies more generally to any type of randomized algorithm in an environment where other participants can hack your solutions.

Fixed seed

If the seed of the rng is fixed, it always produces the same sequence of random numbers. You can just run the code to see which numbers get randomly generated and then find an anti-hash test for those numbers.

Picking from a small pool of bases (`rand() % 100`)

Note that `rand() % 100` produced at most 100 distinct values ($0, \ldots, 99$). We can just find a separate anti-hash test for every one of them and then combine the tests into a single one. (The way your combine tests is problem-specific, but it works for most of the problems.)

More issues with `rand()`

On codeforces, `rand()` produces only 15-bit values, so at most 2^{15} different values. While it may take a while to run 2^{15} birthday-attacks (estimated 111 minutes for $p = 10^9 + 7$ using a single thread on my laptop), this can cause some big issues with some other randomized algorithms.

Edit: This type of hack might be feasible if we use multi-tree-attacks. For $p = 10^9 + 7, |\Sigma| = 26$, running 2^{15} multi-tree attacks with $m = 10^4$ takes around 2 minutes and produces an output of $5.2 \cdot 10^5$ characters. This is still slightly to large for most problems, but could be split up into multiple hacks in an open hacking phase, for example.

In C++11 you can use `mt19937` and `uniform_int_distribution` instead of `rand()`.

Low-precision time (`Time(NULL)`)

`Time(NULL)` only changes once per second. This can be exploited as follows

1. Pick a timespan ΔT.
2. Find an upper bound T for the time you'll need to generate your tests.
3. Figure out the current value T_0 of `Time(NULL)` via custom invocation.
4. For $t = 0, \ldots, (\Delta T) - 1$, replace `Time(NULL)` with $T_0 + T + t$ and generate an anti-test for this fixed seed.
5. Submit the hack at time $T_0 + T$.

If your hack gets executed within the next ΔT seconds, `Time(NULL)` will be a value for which you generated an anti-test, so the solution will fail.

Random device on MinGW (`std::random_device`)

Note that on codeforces specifically, `std::random_device` is deterministic and will produce the same sequence of numbers. Solutions using it can be hacked just like fixed seed solutions.

Notes

- If I made a mistake or a typo in a calculation, or something is unclear, please comment.
- If you have your own hash randomization scheme, way of seeding the rng or anti-hash algorithm that you want to discuss, feel free to comment on it below.

> - I was inspired to write this blog after the open hacking phase of round #494 (problem F). During (and after) the hacking phase I figured out how to hack many solution that I didn't know how to hack beforehand. I (un-)fortunately had to go to bed a few hours in (my timezone is UTC + 2), so quite a few hackable solutions passed.

rolling hashes, math, string

+371 dacin21 13 months ago 22

Comments (22)

Write comment?

13 months ago, # | +5

dacin21

*Auto comment: topic has been updated by **dacin21** (previous revision, new revision, compare).*

→ Reply

13 months ago, # | ← Rev. 6 +18

Kaban-5

Suppose that we use some fixed prime p (for example, $p = 10^9 + 7$, the reason to consider this number is that $p - 1 = 2q$, where $q = 5 \cdot 10^8 + 3$ is prime, has only one small prime factor: 2) and randomize the base. This clearly rules out your high collision probability example because it only can happen when $n = 3$ or $n \geq q + 1$, neither of this situation is close to the model case when $n \approx 10^5$. Can we find a bad case in this situation?

I have a proof that there are two strings that have same hash with probability at least $\frac{cn}{p}$, where $c \in (0, 1)$ is some constant. However, I don't know any way to construct such two strings in reasonable time. Does anybody know a way (possibly completely different)?

Proof, some math here

P. S. It is also possible to break single hash modulo 64-bit (link to the problem, it is in Russian though, Breaking hashing). I hope **Gassa** wouldn't mind explaining the solution, he will definitely do this better than me. In fact, I remember somebody claiming that they can break fixed hashes modulo number of order 10^{100}.

→ Reply

13 months ago, # ^ | +20

pavel.savchenkov

/blog/entry/17507?locale=ru#comment-223614

→ Reply

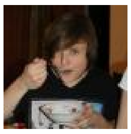
13 months ago, # ^ | 0

dacin21

Big thanks to both of you for linking me this, I'll add a subsection on this. (The attack can be improved by storing multiple values at each node; if we store 10^4 values per node, the resulting length drops to 64 for $p \approx 2^{31}$ and 256 for $p \approx 2^{61}$ and with 10^5 values it drops to 32 and 128 and runs in < 1 s.)

→ Reply

13 months ago, # | ← Rev. 6 +17


dmkozyrev

Thank you very much for your post!

By the way, today I came to the conclusion, that we can calculate double rolling hash without operations `%`, if we use two modules: prime `m1 = 2^31-1` and not prime `m2 = 2^64`. For hash by `m2` we just need to work in `unsigned long long`.

Let's find out how to take the remainder of the division by `m1`.

A simple analogy with the 10-base number system

Let `x` be from `1` to `(m1-1)^2` — result from multiplication of two

Let `x` be from `1` to `(m1-1)^2` - result from multiplication of two remainders. For getting remainder modulo `2^31-1` we need to set `x` a sum all digits in base `2^31` and repeat while `x >= 2^31-1`.

```
x = (x >> 31) + (x & 2147483647);
x = (x >> 31) + (x & 2147483647);
return x;
```

But it works for `(2^31-1) * k` numbers not as you expect. This method gives the result from `1` to `2^31-1` for all positive values and `0` only for `0`.

`x % (2^31-1)` is number from `[0..2^31-2]`

This method definitely gives number from `[1..2^31-1]`.

It works great for rolling hashes if we cut `0` and if we compare results from one type of method (works only with remainders `[1..2^31-1]` against `[0..2^31-2]`). Maybe it can speed up rolling hashes solutions.
→ Reply

dacin21

13 months ago, # ^ | +20

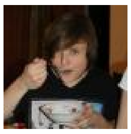
Thanks a lot for your comment.

First of all, **don't use** 2^{64} as your modulo! It can be hacked regardless of the base. (See the section 'Hashing with unsigned overflow ($p = 2^{64}$, q arbitrary)'). Combining it with another hash does not save you, see the section 'Multiple hashes', the Thue–Morse sequence can be used there as well.

Second of, your bit-trick technique can be used to map to $\{0, ..., 2^{31} - 2\}$ as well, just add 1 to x beforehand and subtract 1 from x afterwards. It is probably more useful if we use $p = 2^{61}$ - 1 (with some more bit-tricks), as the alternative with `__int128` does not exists on codeforces.

Badly tested code for that p

I'll add this (with $p = 2^{61}$ - 1) to the section about larger modulos once I get around to test and benchmark it.
→ Reply



dmkozyrev

13 months ago, # ^ | ← Rev. 2 +10

Thanks, please tell me how fast hacker can hack my solution if he knows:

- I'm using double polynomial hash with `p1 = 2^31-1` and `p2 = 2^64`
- I'm using random generation of odd q with uniform distribution from `(256, 2^31-1)` with `std::random_device`, `std::mt19937`, `std::uniform_int_distribution`
- For `2^64` exist anti-hash test regardless of the base `q`
- Time of running solution on server in seconds, for examplt `14:52:31`.

→ Reply

dacin21

13 months ago, # ^ | +10

At first I misread `std::random_device` as `high_resolution_clock` and it seemed quite hard. (The main issue is to break p_1 with less than $\sqrt{p_1} \log_{|\Sigma|}(p_1)$ characters.) But on codeforces, the former always returns `3499211612` as the first number. I can hack you in less than a minute as follows

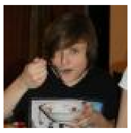
1. Copy/retype your random-generation code and run it in a custom invocation to get your q. (As noted above, it will always be the same.)
2. Run a birthday attack against p_1 (takes ≤ 1

2. Run a birthday attack against p_1 (takes ~ 1 second).
3. Build a Thue–Morse sequence using the strings I got from the birthday-attack instead of 'A' and 'B'.

→ Reply

13 months ago, # ^ |← Rev. 2 0

You need to combine `sqrt(p1) ~=~ 2^16` chars with `2^11` of Thue–Morse string, this is `2^27 = 134.217.728` chars, if I understand correctly, it is possible? I never seen input greater than `5 * 10^6` chars.


dmkozyrev

I asked it because I wrote tutorial for beginners last day and in this tutorial I used two modules: `2^64` , `10^9+123` and random point, generated with `std::mt19937` , `std::chrono::high_resolution_clock` and `std::uniform_int_distribution` and wrote that using `std::time(0)` or `std::random_device` in MinGW not safety, but now I understand that you need very big input to hack solution with fixed point and maybe don't need randomization?

→ Reply

13 months ago, # ^ +20

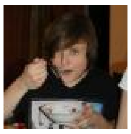
You're mixing apples and oranges. For the birthday attack, we need to generate $\approx \sqrt{p_1}$ **strings**. These strings can be quite short, in practice around 6 **characters** each. We then pick **two** strings that hash to same value and use them in the Thue-Morse sequence. So the total length would be $6 \cdot 2 \cdot 2^{10} = 12288$. ($2^{10}$ sufficies, 2^{11} was based of a non-tight bound from the linked post on hashes mod 2^{64}.)

dacin21

If you use 2^{64} and a prime around 10^9 with good randomization, I haven't figured out a good hack yet. (Note that this does not mean you can't be hacked, someone might find a better attack.) The best I thought of was building a generalized Thue-Morse sequence in base $\sqrt{2p}$, so that all pairs collide mod 2^{64} and there is a good change for a birthday-collision mod p. This would lead to massive input however (Estimated $2^{10} \cdot 6 \cdot 45'000 \approx 3 \cdot 10^8$), as it involves submitting an input with $\sqrt{2p}$ long strings. So I guess there's a good chance you won't be hacked, but I myself prefer something where I can proof that it's hard for me to get hacked and not rely on other not figuring out some new technique.

→ Reply

+15

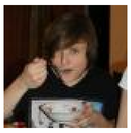

dmkozyrev

13 months ago, # ^ |

Now I understood, thanks

→ Reply

13 months ago, # ^ | ← Rev. 3 +5


dmkozyrev

I not understood why this method not works on this problem on generated test. Code on ideone.com or my solution is bad.

I generated test with length 5000:

```
s = 'v' * 2500 + '-' * 2500
t = '~' * 2500 + 'v' * 2500
```

True answer is 2500, but it gets 2499 or 2500 (Answer depends on launch with random point).

UPD: Sorry, bug in function diff, my bad:

a and b unsigned long long and it is not correct (compare with zero) in function sub:

```
return (a -= b) < 0 ? a + mod : a;
```

It should be like this:

```
return (a -= b) >= mod ? a + mod : a;
```

→ Reply

13 months ago, # | +10

halyavin

My randomization function:

```
uint32_t rd() {
  uint32_t res;
#ifdef __MINGW32__
  asm volatile("rdrand %0" :"=a"(res) ::"cc");
#else
  res = std::random_device()();
#endif
  return res;
}
```

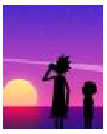
→ Reply

13 months ago, # ^ | 0

dacin21

Thanks, added it and a version that uses the corresponding built-in function instead of asm.

→ Reply


HanaElhami

13 months ago, # | ← Rev. 3 +1

Double hashing has never failed me. So I think you don't have to worry for collision in CE if you use double hashing. My primes for mod are 10^9+7 and 10^9+9 and I use two random primes like 737 or 3079 or 4001 as base.

→ Reply

13 months ago, # ^ | 0

dacin21

If the input is random so that we can assume the hash-values are uniformly distributed, you're fine.

But if you don't randomize your bases at runtime (and it sounds like you don't), someone might hack your solution. (And AFAIK it makes no difference whether the **base** is prime or not.)

→ Reply

khokharnikunj8

13 months ago, # ^ | +9

Now that you are no more random :D, Get ready to... :D

→ Reply

dacin21

13 months ago, # | 0

Auto comment: topic has been updated by ***dacin21*** *(previous revision, new revision, compare).*

→ Reply

hellman_

13 months ago, # | +43

On abusing multiple known modulos: you can **quickly** generate **short** hack-tests using lattice reduction methods (LLL). There was a related challenge at TokyoWesterns CTF (an infosec competition) and I made a writeup for that (or another writeup). There you have 8 32-bit modulos and the generated hack-test is less than 100 symbols. (The problem there is palindrome search but the method is generic).

→ Reply

dacin21

13 months ago, # ^ | 0

Thank, lattice reduction works great for multiple hashes if the alphabet size is not to small. I added a section on it with some modified code. For many 61-bit hashes, the LLL algorithm deteriorates, but the BKZ algorithm in sage works instead.

→ Reply

semi_rated

13 months ago, # | -10

Thanks for your awesome post !

Just another suggestion or request: it would be great if you would provide a sample full code combining all of the ideas at the end of the post.

→ Reply

iamabjain

7 months ago, # | +8

Awesome post. .simple and really informative..!!

→ Reply