Hacking Hashing in Competitive Programming

Masaki Takeuchi

June 3, 2024

1 Introduction

During Competitive Programming contests, participants compete for the maximum number of problems solved and earliest time solved.

One of the most popular Competitive Programming websites, Codeforces features a "hacking" system where after you have submitted a solution that passes the tests provided by the problem-setter, you have the option to try to find a counter-test for a submission by another participant.

Although hashing is a very powerful tool for competitive programming, it's necessary to take precautions like introducing runtime randomness into a program to make it non-deterministic and thus more robust against hacking in a contest.

I researched the two most popular applications of hashing in Competitive Programming in C++: Hashtables and Polynomial String Hashing.

2 Defining Robustness

Due to the nature of hashing, for all hashes with more possible states than hash values, with an unlimited number of attempts and computing power, a collision can theoretically be found such that `M != M'` and `H(M) == H(M')`.

The Codeforces platform typically gives a participant +100 points for a successful hacking attempt against another competitor and -50 points for unsuccessful tries. In addition, the hacking period is typically around 12 hours after the end of a contest. These parameters give more reasonable bounds for the number of attempts and the computing power of an adversary.

A robust hashing solution would be one that cannot be hacked in a handful of attempts using an average computer within 12 hours with high probability.

3 Pseudo-Random Number Generators (PRG)

In order to make hashing solutions robust against hacking attempts, runtime randomness can be added using a pseudo-random number generator with a random seed.

3.1 `rand()` : BAD

The C-standard library function rand is legacy code that is not suitable for most applications today. The function returns a library-dependent value between 0 and RAND_MAX but the C-standard only guarantees that `RAND_MAX` is at least `32767` or $2^{15} - 1$.[2]

3.2 Mersenne Twister : GOOD

32-bit and 64-bit versions of the Mersenne Twister random-number generator are included in the C++ standard library as mt19937. The GCC compiler also includes SIMD-oriented fast Mersenne Twister sfmt19937.

Pros:

- Can be initialized with a single line due to being in the C++ standard library
- Easy to generate distributions by using with std::uniform_int_distribution

Cons:

- Uses an excessive 19937 bits of state and slow [6]

In the context of competitive programming where implementation speed is a large factor, std::mt19937 is still a very good PRG.
Example Usage:

```
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
#define uid(a, b) uniform_int_distribution<>(a, b)(rng)
```

The `uid` macro generates a value between `[a, b]`.

3.3 Xoshiro : GOOD

New PRGS such as Xoshiro (xor, shift, rotate) only utilize a handful of operations such as add, xor, shift, and rotate which each only take a single cpu instruction on modern hardware.

Pros:

- Very fast in comparison to std::mt19937
- Small state (256 bits for xoshiro256 64-bit generator)

Cons:

- Not in C++ standard library, requires additional code
- Requires additional code to implement fair mapping to a range compared to simply using std::uniform_int_distribution with Mersenne Twister [5]
- Easily reversible

Example Template Code

It is not worth using xoshiro or xoroshiro PRGS in competitive programming even if template code is set up beforehand since the speed difference is not significant for algorithm contests.

However in Heuristics contests such as Atcoder Heuristic Contest, the speed difference of the PRG allows for more iterations of non-deterministic algorithms like Simulated Annealing and thus can lead to a competitive advantage.

4 Seeding PRGs

4.1 Fixed seed : BAD

With a fixed seed, the sequence generated by a PRG is also fixed. The sequence can be generated once to find an adversarial testcase.

4.2 `time(nullptr)` : BAD

Like rand, time is part of the C standard library.

"The value returned generally represents the number of seconds since 00:00 hours, Jan 1, 1970 UTC (i.e., the current *unix timestamp*)" [3]

The return value is a 32-bit value that increments every second. This value is very predictable. By finding the fixed sequence for a time range, for example it is possible to generate 60 collision sequences for a time range of a minute and

combine the results to generate an adversarial testcase. As long as the test is ran on the server within the next minute, this will result in a successful hacking attempt.

4.3 `std::chrono::steady_clock` : GOOD

```
chrono::steady_clock::now().time_since_epoch().count()
```

This returns a 64-bit value with significantly higher precision in comparison to the previous ctime. Although the top bits of the number are very predictable like ctime, the lower bits are very unpredictable. Even calling the function above twice in two successive lines in a local program yields differing lower bits.

Since the lower bits are very volatile, combined with the avalanche effect in good PRGs, it is not practical to take advantage of this value when used as a seed.

5 Hashtables

5.1 How `std::unordered_map` works

The C++ standard library hashtable std::unordered_map works in the following way:[4]

1. std::hash is called on the key value.
2. The hashed value is distributed to one of the buckets by taking hash % bucket-count. The bucket size is always a prime number.
3. When different key values are mapped to the same bucket, std::unordered_map uses separate chaining to deal with collisions.
4. When the number of items in the data structure go over a set load factor, the bucket-count is increased.

5.2 Hacking

Several parts of this implementation make it vulnerable to hacking.
First, std::hash when called on integer values acts as the **identity function**, meaning the value is not actually hashed at all.
Second, the prime numbers used for the bucket-counts are fixed for each compiler version.
Third, the load factor by default is 1.0, meaning that the hashtable bucket-size does not increase until the number of elements equals the previous bucket-size.

By calling std::unordered_set::bucket_count() or std::unordered_map::bucket_count() while adding elements, it is possible to find the set bucket sizes for the specific compiler version.

```
int find_bucket_count(int N) {
    unordered_set<int64_t> st;
    int sz = -1;
    for (int i = 0; i < N; i++) {
        st.insert(i);
        if (st.bucket_count() != sz) {
            sz = st.bucket_count();
            cout << st.size() << ": " << sz << "\n";
        }
    }
    return sz;
}
```

Suppose a problem involves finding the number of duplicated values from a sequence of `n` integers. With few collisions, the expected running time using a hashtable is O(1) for each insertion and lookup operation to process `n` numbers.

```
void run(int N, int bucket_size) {
    unordered_map<int64_t, int64_t> mp;
    int64_t curr = 0;
    for (int i = 0; i < N; i++) {
        mp[curr]++;
        curr += bucket_size;
    }
    int64_t sum = 0;
    for (int i = 0; i < N; i++) {
        curr -= bucket_size;
        sum += mp[curr];
    }
    cout << sum << "\n";
}
```

Full example here

By instead adding in equivalent values modulo the last bucket-count, an adversary can force a collision chain of length `n`. Instead of each add and lookup operations taking about `O(1)` time, the time complexity could increase to up to `O(n)` for each operation and cause a "Time limit exceeded" verdict.

Using $n = 10^5$, the above code runs in around 10 seconds on a M1 Macbook, while a random set of numbers runs in well under a second.

5.3 Robust solution

One way to make Hashtables robust against the previous attack is to actually hash the input value. First created for the Java language, the Splitmix PRG takes in a state value and applies xorshift operations to create the next state.[7]

This can be added as a hash function to use with std::unordered_set or std::unordered_map.

However, since Splitmix can easily be reversed , runtime randomness must be additionally added to the funciton. Without runtime randomness, instead of adding multiples of the bucketcount, an adversary could add the inverse splitmix of the multiples of the bucketcount.

```
struct custom_hash {
    constexpr static uint64_t splitmix64(uint64_t x) {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }
    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM = chrono::steady_clock::now().time_since
        return splitmix64(x + FIXED_RANDOM);
    }
};
template<class K> using hashset = unordered_set<K, custom_hash>;
template<class K, class V> using hashmap = unordered_map<K, V, custom_hash>;
```

This template code allows the usage of "hashset" in an identical way to std::unordered_set [8]

6 Polynomial String Hashing

Suppose you want to hash a string `s` of length `n` with base `b`.

Let $hash(s) = s[0] \cdot b^0 + s[1] \cdot b^1 + s[2] \cdot b^2 + ... + s[n-1] \cdot b^{n-1}$

Polynomial string hashing uses the information of what character is at what position in order to create a hash value. Because b^{n-1} quickly becomes larger than what would fit in an integer, all the calculations are typically done modulo a large prime number.

Hashing allows for comparisons of long substrings by doing integer comparisons in `O(1)` time. In addition, the Rabin-Karp Rolling Hash technique makes it possible to modify a hash from one substring to a neighboring substring in `O(1)` time, to make it even more useful.

Although polynomial string hashing has a lot of applications to competitive programming, it is also very susceptible to collisions.

6.1 Birthday Attack

Suppose many random numbers are picked within the range of `[1, N]`. The birthday phenomenon describes that when the number of values chosen becomes much larger than $\sqrt{N}$, it is likely to see at least one repeated value.

When the polynomial base and modulo are fixed, an adversary can generate random strings and expect a repeated hash value after about $\sqrt{\texttt{mod}}$ strings tested since the hash will result in a value within $[0, \texttt{mod} - 1]$.

Note: In the test code, the string hashing is reversed as
$s[0] \cdot b^{n-1} + s[1] \cdot b^{n-2} ... + s[n-1]$ for ease of implementation.

6.2 Thue-Morse Sequence

The Thue-Morse sequence is "a binary sequence obtained by starting with 0 and successivel appending the Boolean complement of the sequence so far".[1]

This can be used to attack hashing solutions that let the result of polynomial hashing overflow using the unsigned integer bound.

Let the '0's of a Thue-Morse sequence of length of a power of 2 be replaced by 'A' and '1's with 'B'. This AB Thue-Morse string and its inverse (all 'A's swapped with 'B's and vice versa) hash to the same value when the base is odd and the length of the string becomes long enough.
Generating AB Thue-Morse strings `s`, `t` of length 2^q. Full Example Code

```
int n = 1 << q;
string s, t;
char ch[2] = { 'a', 'b' };
for (int i = 0; i < n; i++) {
    s.push_back(ch[__builtin_parity(i)]);
    t.push_back(ch[!__builtin_parity(i)]);
}
```

6.3 Other attacks against Fixed-Base and Fixed-Modulo hashes

Since the birthday attack takes about 2^{30} operations against a 2^{60} modulus, hashing the same string twice with two different bases or modulus, or double-hashing, can be an effective strategy against it. However, other attacks are able to quickly find collisions against 64-bit modulos and double-hashing.

A codeforces user recently created a website that runs a Lattice-Reduction attack locally within the browser. For example, this attack was able to generate a collision against a double hash with modulo $2^{61} - 1$ within 2 seconds, whereas it would be impractical to attempt to find a collision using a birthday attack.

6.4 Robust solution

Randomizing the base at runtime prevents offline hash collision attacks, as long as the modulus is not a power of two (in which case the Thue-Morse Sequence can be utilized).

Randomizing the base for the polynomial string hashing at runtime and using a 64-bit modulo is enough to make polynomial string hashing robust against hacking.

Template code

References

[1] May 2024. URL: https://en.wikipedia.org/wiki/Thue%E2%80%93Morse_sequence.

[2] URL: https://cplusplus.com/reference/cstdlib/RAND_MAX/.

[3] URL: https://cplusplus.com/reference/ctime/time/.

[4] Michal Forisek. *Internet Problem Solving Contest 2014 Solutions Problem H Hashsets*. 2014. URL: https://ipsc.ksp.sk/2014/real/solutions/booklet.pdf.

[5] Daniel Lemire. "Fast random integer generation in an interval". In: *ACM Transactions on Modeling and Computer Simulation* 29.1 (Jan. 2019), pp. 1–12. DOI: 10.1145/3230636.

[6] Vigna Sebastiano. In: *It is high time we let go of the Mersenne Twister* (Oct. 2019). DOI: arXiv:1910.06437.

[7] Guy L. Steele, Doug Lea, and Christine H. Flood. "Fast splittable pseudorandom number generators". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages amp; Applications* (Oct. 2014). DOI: 10.1145/2660193.2660195.

[8] Neal Wu. *Blowing up unordered$_m$ap, andhowtostopgettinghackedonit*. Oct. 2018. URL: https://codeforces.com/blog/entry/62393.