Hashing

Kevin Geng

5 May 2016

1 Hash functions

1.1 Definition

We define a *hash function* as a function that accepts a string as input and returns an integer in the range $0...n-1$.

1.2 Properties

Any good hash function should:

- *Be deterministic.* The same string should always map to the same output. To ensure this, canonicalization of the input may be necessary.
- *Be uniform.* A hash function should map inputs as evenly as possible over its domain. This reduces the likelihood of collisions.
- *Have a defined range.* Hash functions should have a fixed output size, even for arbitrary input data. Limiting the output to an arbitrary size is often accomplished simply by using the modulus operation.
- *Be relatively fast.* Hash tables aren't fast unless hashing is fast! However, the speed of a hash function does depend on what other properties it may have.

1.3 Canonicalization

Canonicalization involves transforming data into a *canonical format* so that all equivalent inputs map to the same output. For example, if we wish to create a hash function that ignore cases, we may need to canonicalize an input string by lowercasing it (at least, if the string is ASCII-only).

1.4 Collisions

By the Pigeonhole Principle, hash functions will inevitably have collisions. However, the incidence of collisions can be greatly reduced by using multiple, independent hash functions.

2 Hash tables

As this topic is already covered in the AP Computer Science course at TJHSST, I won't discuss it here.

3 Polynomial hashing

3.1 A perfect hash

Perhaps the first question we should ask is, how can we represent a string as a number? By encoding a string with ASCII or UTF-8, we can look at a string as a sequence of bytes. Those bytes are themselves series of 8 bits, so we can simply concatenate those bits together into one long series of bits. Then, we can interpret that series of bits as a number.

Because this hash function will map each string to a unique integer (i.e. it is an injective function), it is a *perfect hash function*. This means that it will never have collisions (unless a fixed-size integer is used, in which case values will overflow). However, it's not particularly useful as a hash function, although it is used to encode strings for RSA.

3.2 Definition

A *polynomial hash* can be described by a polynomial in some base b with coefficients s_i, hence the name.

$$hash(s) = \sum_{i=1}^{n} p^i * s_i \mod M$$

Polynomial hashes have several nice properties that make them very useful in competitive programming. In particular, the hash of a string is directly related to the string's contents.

3.3 Choosing b and M

In order to avoid collisions, it's best to pick prime numbers for both b and M. The obvious reason for this is that if $b = 2$ and $M = 2^{32}$ due to integer overflow, then the hash will be independent of anything beyond the first 32 bytes of the string.

Another reason is that it is often necessary to find a modular inverse with respect to M, which is only guaranteed to exist if M is prime. In particular, by Fermat's little theorem, $b^{M-1} \equiv 1 \mod M$ if M is prime and b is not divisible by M.

But not all implementations pick prime numbers for both:

- The reference implementation of Java's `String.hashCode` uses $b = 31$, and because it doesn't use an explicit modulus, effectively $M = 2^{32}$.
- Sedgewick's implementation of Rabin-Karp uses $b = 256$, and a large prime as the modulus M.

3.4 Substrings

When using a polynomial hash, it's possible to quickly compute the hash of the concatenation of two strings, given their hashes:

$$hash(s_1 \oplus s_2) = hash(s_1) * p^{|s_2|} + hash(s_2)$$

Furthermore, by computing a table of hashes of every prefix, the hash of any substring can be computed in $O(1)$ time. The computation of that prefix hash table itself only requires linear time.

$$hash(s_{i,j}) = (hash(s_{0,j}) - hash(s_{0,i-1}))/p^i$$

3.5 String comparisons

By comparing hash values, string equality can be checked in constant time, although there is some probability of collision. Furthermore, if the prefix hashes of two strings a and b have been computed, then they can be compared by binary searching for the number of characters in the prefix that match.

3.6 Rabin-Karp

The Rabin-Karp algorithm requires a rolling hash in order to achieve its $O(m + n)$ search performance. Most commonly, this is accomplished with a polynomial hash.

4 Other applications

Note: This section is fairly useless for competitive programming.

4.1 Bloom filters

Bloom filters are a probabilistic data structure that can test whether an element exists in a set.

Bloom filters work in the same way that hash tables do, except that the actual values of the strings aren't stored. In other words, instead of a string array A, a Bloom filter consists of a bit array A, where $A[i]$ is true if the set contains a string s with a hash value of $h(s) = i$.

The significant advantage of Bloom filters is that they take much less space, as they don't require storing actual string values. However, this means that it's impossible to resolve collisions, resulting in false positives. As a result, they must be used carefully. The probability of a false positive can be reduced by creating multiple Bloom filters with different, independent hash functions.

A commonly cited application is Google's Safe Browsing service, which formerly used a Bloom filter to determine whether a given webpage was malicious.

4.2 Cryptography

Hash functions are also widely useful for cryptographic purposes. However, such hash functions must have additional properties in order to be useful against attackers:

- *Preimage resistance.* Given a known output h, it should be infeasible to find an input s that has that output; i.e., such that $h = hash(s)$.
- *Second preimage resistance.* Given an input s_1, it should be infeasible to find another input s_2 with the same hash; i.e. such that $hash(s_1) = hash(s_2)$.
- *Collision resistance.* It should be difficult to find any pair of s_1 and s_2 such that $hash(s_1) = hash(s_2)$.

As a result, cryptographic hash functions must be specifically designed for the purpose. The most well-known hash function is the SHA series (SHA-1, SHA-256, SHA-512). Such hashes are useful for a plethora of purposes in cryptography.

Unfortunately, simple hashes such as the polynomial hash satisfy none of the above properties. These properties are important to keep in mind when writing code that may be subject to adversarial input, even outside of cryptography. In 2011, researchers discovered that the majority of server-side programming languages were vulnerable to denial-of-service attacks on their hashing implementations. An attacker could trigger hash collisions in order to place a lot of data in one bucket, resulting in operations on the order of $O(n)$ rather than $O(1)$.

5 Problems

Stolen from Lalit Kundu's post on *String Hashing for Competitive Programming.*

1. You have a string S of length N. You are given Q queries of the form L_i, R_i $(N, Q \leq 105)$. For each query, determine whether the substring denoted by $S_{L_i}, S_{L_i+1}, ...S_{R_i}$ is a palindrome.
2. You have a string S of length N $(N \leq 105)$. Given an $M \leq N$, find the number of substrings of S that are palindromes and are of size M.
3. Given a string S, find the size of the substring of largest size which occurs at least twice in the string.

Something fun to try:

1. Estimate the number of distinct elements in a data stream that is too large to be stored in memory.

And, of course, the reason I wrote this lecture:

1. USACO 2016 US Open, Platinum Problem 2: Bull in a China Shop