String Hashing

Spencer Compton

CPSI 2018: Squires Lecture 1

1 Introduction

Many string problems boil down to comparing strings (either in terms of equality, or lexicographical ordering) very quickly. We can use String Hashing to compare two strings for equality and get the correct answer with high probability. This simple ability allows us to tackle the majority of string problems.

2 Polynomial Representation

When we see a number, for example 234, what does that really mean? We automatically interpret that to mean $2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$. Similarly, we automatically interpret 1011_2 to mean $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$. To generalize, for any base B number, where D_i represents the ith digit from right to left (e.g. D_2 represents the hundreds digit), we can represent it as a polynomial in the form of $D_0 \times B^0 + D_1 \times B^1 + D_2 \times B^2 + D_3 \times B^3 + ...$. If we represent a string as a base-27 number (where $A = 1$, $B = 2$, $C = 3$, and so on), it can also be viewed as a polynomial. For example, "SQRT" could be viewed as $19 \times 27^3 + 17 \times 27^2 + 18 \times 27^1 + 20 \times 27^0 = 386,876$. We would call this our polynomial hash of "SQRT".

Personally I like to make my hashes base 29 (because 29 is prime and 27 is not, so the remainder of the lecture will use $B = 29$ to denote the base).

If we want to design our polynomial hashes in a way that different strings have the same hash with low probability, why might we not want to map the characters such that $A = 0$, $B = 1$, $C = 2$, and so on?

3 Hashing Under Modulo

If we created a polynomial hash in the same manner for a string of length 10^6, for example, this could take $1,414,974$ digits to represent as a base 10 number.

Clearly this is impractical to deal with, and even for shorter strings (e.g. length 15) the hash would overflow a 64 bit integer. How do we deal with this? We take our hashes under some large modulo (e.g. $10^9 + 7$). By doing this, we accept a trade-off. While our hashing is no longer "perfect" (i.e. two different strings could "collide" and have the same hash), the numbers are much easier to deal with. If we take our hashes modulo some large prime, we can essentially treat this as if we are getting a deterministic random number in the range $[0, m-1]$ where m is our modulo. This means that the probability of two different strings having the same hash under modulo is quite small, or $\frac{1}{m}$.

4 Rolling Hash

We will use Rolling Hash to keep track of a strings hash as we modify the string. We will support the following operations:

- Add First: Add a character to the front of the string. E.g. "ARUP".addFirst("C") = "C" + "ARUP" = "CARUP". Let *len* denote the length of the string before we add a character to the front. The polynomial hash of "ARUP" is $1 \times B^3 + 18 \times B^2 + 21 \times B^1 + 16 \times B^0$. Thus, the first character in the string before the addition is being multiplied by B^{len-1}. If we add a character to the beginning, then the new first character will be multiplied by B^{len}. Since the rest of the string doesn't need to be shifted over, all we need to do to support this operation is add the new character multiplied by B^{len}. This can be seen as:

  $1 \times B^3 + 18 \times B^2 + 21 \times B^1 + 16 \times B^0$ (original polynomial before operation)

  $3 \times B^4 + 1 \times B^3 + 18 \times B^2 + 21 \times B^1 + 16 \times B^0$ (polynomial after adding new character)

- Add Last: Add a character to the end of the string. E.g. "ARUP".addLast("C") = "ARUP" + "C" = "ARUPC". The polynomial hash of "ARUP" is $1 \times B^3 + 18 \times B^2 + 21 \times B^1 + 16 \times B^0$. To add a character to the end, we will shift the letters left by multiplying the polynomial by B and then adding the character's mapping multiplied by B^0. This can be seen as:

  $1 \times B^3 + 18 \times B^2 + 21 \times B^1 + 16 \times B^0$ (original polynomial before operation)

  $1 \times B^4 + 18 \times B^3 + 21 \times B^2 + 16 \times B^1$ (polynomial after shifting left)

  $1 \times B^4 + 18 \times B^3 + 21 \times B^2 + 16 \times B^1 + 3 \times B^0$ (polynomial after adding new character).

- Pop First: Remove a chracter at the beginning of the string. E.g. "ARUP".pollFirst() = "RUP". Let *len* denote the length of the string before we remove a character from the front. The polynomial hash of "ARUP" is $1 \times B^3 + 18 \times B^2 + 21 \times B^1 + 16 \times B^0$. This, the first character in the string before the

removal is being multiplied by B^{len-1}. If we want to remove that character, we will simply subtract the character multiplied by B^{len-1}. Since the rest of the hash remains unchanged, this is all we need to do. This can be seen as:

$1 \times B^3 + 18 \times B^2 + 21 \times B^1 + 16 \times B^0$ (original polynomial before operation)

$18 \times B^2 + 21 \times B^1 + 16 \times B^0$ (polynomial after removing first character)

- Pop Last: Remove a character from the end of the string. E.g. "ARUP".popLast() = "ARU". The polynomial hash of "ARUP" is $1 \times B^3 + 18 \times B^2 + 21 \times B^1 + 16 \times B^0$. To remove a character from the end, we want to get rid of the polynomial's last term and then shift the remaining terms to the right one. For getting rid of the last term we can simply subtract the last letter multiplied by B^0. To shift to the right, we want to divide by B, but since we are only keeping track of the hash under modulo (not the actual value) it might not be divisible by B. For dividing under modulo, we need to instead multiply by the modular inverse of B, or B^{-1}. If m is prime, then we can use Fermat's Little Theorem to state $B^{-1} \equiv B^{m-2} (mod\ m)$. We can then use Fast Modular Exponentiation to precompute $B^{m-2} (mod\ m)$ in $O(\log m)$. Thus we will subtract out the last latter multiplied by B^0 and then multiply by B^{-1} to shift the remaining terms right. This can be seen as:

  $1 \times B^3 + 18 \times B^2 + 21 \times B^1 + 16 \times B^0$ (original polynomial before operation)

  $1 \times B^3 + 18 \times B^2 + 21 \times B^1$ (after subtracting the last character)

  $1 \times B^2 + 18 \times B^1 + 21 \times B^0$ (after multiplying by $B-1$ to shift)

5 Collisions

When should we worry about hash collisions? We want to make sure our program compares all strings correctly with high probability. Two identical strings will always have the same hash, and two non-identical strings will have the same hash with probability $\frac{1}{p}$. However, if we hash N non-identical strings, the probability of all hashes being different is not $(\frac{p-1}{p})^N$. This can be explained by the Birthday Paradox. Instead the probability is $\frac{p-1}{p} \times \frac{p-2}{p} \times ... \times \frac{p-N+1}{p}$. If $m \approx 10^9$ and $N \approx 3,000$, it would more than 150 test cases for there to be a 50% chance of at least one hash collision. If $m \approx 10^{18}$ and $N \approx 10^7$, it would take more than $13,000$ test cases for there to be a 50% chance of at least one hash collision. Since in most upper level problems $N >= 10^4$, it is much safer to use $m = 10^{18}$. However, if we tried implementing this in a straightforward way with 64 bit integers we have overflow issues (e.g. multiplying two values that are $\approx 10^{18}$. Instead, we can employ a technique called "double hashing". Double hashing states that if we maintain the hash value for a string under two

prime modulos, m_1 and m_2, it is equivalent to maintaining a hash under modulo $m_1 m_2$ (this can be shown through the Chinese Remainder Theorem).

6 Comparing Substrings

What if we are given a string of length N and want to quickly compute hashes for certain substrings. Although the previously mentioned Rolling Hash operations are $O(1)$ each, doing $O(N)$ of them to change from one substring to another would be expensive. Let's consider a string "ABCDEF". We will do this polynomial hash in a way where the first letter has a coefficient B^0 and the last letter has a coefficient B^{len-1}. The polynomial hash of "BACDEF" is $2\times B^0+1\times B^1+3\times B^2+4\times B^3+5\times B^4+6\times B^5$. Let's store a prefix sums array A where $A[0] = 0$, $A[1] = 2\times B^0$, $A[2] = 2\times B^0+1\times B^1$, and so on. If we wanted to know the hash of range $[i, j]$ we might be tempted to do $A[j+1] - A[i]$, however this is slightly off. E.g. for range $[2, 4]$, $A[5] - A[2] = 3 \times B^2 + 4 \times B^3 + 5 \times B^4$. For this to be correct, the first term would need a coefficient of B^0, thus we need to shift B by 2. We can do this by multiplying our hash value by B^{-2} (we will precompute all B^{-x} for all $1 <= x <= N$ beforehand). Thus, more generally, the hash for a range $[i, j]$ is $(A[j + 1] - A[i]) \times B^{-i}$.

7 Example Uses

- Given a string S and queries of two substrings A and B, how can we determine in $O(\log |S|)$ whether $A = B$, $A < B$, or $A > B$ with $O(|S|)$ preprocessing?
- (Generally accomplished with Knuth-Morris-Pratt) Given two strings S and T, count the number of occurrences of T in S in $O(|S| + |T|)$.
- (Generally accomplished with Suffix Arrays) Given a string S and queries of two substrings A and B, how can we determine in $O(\log |S|)$ the longest common prefix of A and B, with $O(|S|)$ preprocessing?
- (Generally accomplished with Suffix Arrays) Given two strings S and T, how can we determine theor longest common substring in $O((|S| + |T|) \times \log(|S| + |T|))$.
- (Generally accomplished with Manacher's Algorithm) Given a string S find the longest palindromic substring and count the number of palindromic substrings in $O(|S| \log |S|$.
- (Generally accomplished with Aho-Corasick) See USACO Gold February 2015: Censoring

8 Example Problems

USACO Silver February 2015: Censoring

USACO Gold February 2015: Censoring

ACM-ICPC Asia Jakarta Regional Contest 2013: Pasti Pas! (Problem F)

9 Additional Resources

https://www.youtube.com/watch?v=rA1ZevamGDc

https://codeforces.com/blog/entry/60445

http://courses.csail.mit.edu/6.006/spring11/rec/rec06.pdf

https://activities.tjhsst.edu/sct/lectures/1516/SCT_Hashing.pdf