Rolling Hash (Rabin-Karp Algorithm)

Objective

If we have text string S and pattern string P, we want to determine whether or not P is found in S, i.e. P is a substring of S.

Notes on Strings

Strings are arrays of characters. Characters however can be interpreted as integers, with their exact values depending on what type of encoding is being used (e.g. ASCII, Unicode). This means we can treat strings as arrays of integers. Finding a way to convert an array of integers into a single integer allows us to hash strings with hash functions that expect numbers as input.

Since strings are arrays and not single elements, comparing two strings for equality is not as straightforward as comparing two integers for equality. To check to see if string A and string B are equal, we would have to iterate through all of A's elements and all of B's elements, making sure that $A[i] = B[i]$ for all i. This means that string comparison depends on the length of the strings. Comparing two n-length strings takes $O(n)$ time. Also, since hashing a string usually involves iterating through the string's elements, hashing a string of length n also takes $O(n)$ time.

Method

Say P has length L and S has length n. One way to search for P in S:

1. Hash P to get $h(P)$ **O(L)**
2. Iterate through all length L substrings of S, hashing those substrings and comparing to $h(P)$ **O(nL)**
3. If a substring hash value does match $h(P)$, do a string comparison on that substring and P, stopping if they do match and continuing if they do not. **O(L)**

This method takes $O(nL)$ time. We can improve on this runtime by using a **rolling hash**. In step 2. we looked at $O(n)$ substrings independently and took $O(L)$ to hash them all. These substrings however have a lot of overlap. For example, looking at length 5 substrings of "algorithms", the first two substrings are "algor" and "lgori". Wouldn't it be nice if we could take advantage of the fact that the two substrings share "lgor", which takes up most of each substring, to save some computation? It turns out we can with rolling hashes.

"Numerical" Example

Let's step back from strings for a second. Say we have P and S be two integer arrays:

$$P = [9, 0, 2, 1, 0] \tag{1}$$
$$S = [4, 8, 9, 0, 2, 1, 0, 7] \tag{2}$$

The length 5 substrings of S will be denoted as such:

$$S_0 = [4, 8, 9, 0, 2] \tag{3}$$
$$S_1 = [8, 9, 0, 2, 1] \tag{4}$$
$$S_2 = [9, 0, 2, 1, 0] \tag{5}$$
$$... \tag{6}$$

We want to see if P ever appears in S using the three steps in the method above. Our hash function will be:

$$h(k) = (k[0]10^4 + k[1]10^3 + k[2]10^2 + k[3]10^1 + k[4]10^0) \bmod m \tag{7}$$

Or in other words, we will take the length 5 array of integers and concatenate the integers into a 5 digit number, then take the number mod m. $h(P) = 90210 \bmod m$, $h(S_0) = 48902 \bmod m$, and $h(S_1) = 89021 \bmod m$. Note that with this hash function, we can use $h(S_0)$ to help calculate $h(S_1)$. We start with 48902, chop off the first digit to get 8902, multiply by 10 to get 89020, and then add the next digit to get 89021. More formally:

$$h(S_{i+1}) = [(h(S_i) - (10^5 * \text{ first digit of } S_i)) * 10 + \text{ next digit after } S_i] \bmod m \tag{8}$$

We can imagine a window sliding over all the substrings in S. Calculating the hash value of the next substring only inspects two elements: the element leaving the window and the element entering the window. This is a dramatic difference from before, where we calculated each substring's hash values independently and would have to look at L elements for each hash calculation. Finding the hash value of the next substring is now a $O(1)$ operation.

In this numerical example, we looked at single digit integers and set our base $b = 10$ so that we can interpret the arithmetic easier. To generalize for other base b and other substring length L, our hash function is

$$h(k) = (k[0]b^{L-1} + k[1]b^{L-2} + k[2]b^{L-3}...k[L-1]b^0) \bmod m \tag{9}$$

And calculating the next hash value is:

$$h(S_{i+1}) = b(h(S_i) - b^{L-1}S[i]) + S[i + L] \bmod m \tag{10}$$

Back to Strings

Since strings can be interpreted as an array of integers, we can apply the same method we used on numbers to the initial problem, improving the runtime. The algorithm steps are now:

1. Hash P to get $h(P)$ **O(L)**
2. Hash the first length L substring of S **O(L)**
3. Use the rolling hash method to calculate the subsequent $O(n)$ substrings in S, comparing the hash values to $h(P)$ **O(n)**
4. If a substring hash value does match $h(P)$, do a string comparison on that substring and P, stopping if they do match and continuing if they do not. **O(L)**

This speeds up the algorithm and as long as the total time spent doing string comparison is $O(n)$, then the whole algorithm is also $O(n)$. We can run into problems if we expect $O(n)$ collisions in our hash table, since then we spend $O(nL)$ in step 4. Thus we have to ensure that our table size is $O(n)$ so that we expect $O(1)$ total collisions and only have to go to step 4 $O(1)$ times. In this case, we will spend $O(L)$ time in step 4, which still keeps the whole running time at $O(n)$.

Common Substring Problem

The algorithm described above takes in a specific pattern P and looks for it in S. However, the problem we've dealt with in lecture is seeing if two long strings of length n, S and T, share a common substring of length L. This may seem like a harder problem but we can show that it too has a runtime of $O(n)$ using rolling hashes. We will have a similar strategy:

1. Hash the first length L substring of S **O(L)**
2. Use the rolling hash method to calculate the subsequent $O(n)$ substrings in S, adding each substring into a hash table **O(n)**
3. Hash the first length L substring of T **O(L)**
4. Use the rolling hash method to calculate the hash values subsequent $O(n)$ substrings in T. For each substring, check the hash table to see if there are any collisions with substrings from S. **O(n)**
5. If a substring of T does collide with a substring of S, do a string comparison on those substrings, stopping if they do match and continuing if they do not. **O(L)**

However, to keep the running time at $O(n)$, again we have to be careful with limiting the number of collisions we have in step 5 so that we don't have to call too many string comparisons. This time, if our table size if $O(n)$, we expect $O(1)$ substrings in each slot of the hash table so we expect $O(1)$ collisions for each substring of T. This results in a total of $O(n)$ string comparisons

which takes $O(nL)$ time, making string comparison the performance bottleneck now. We can increase table size and modify our hash function so that the hash table has $O(n^2)$ slots, leading to an expectation of $O(\frac{1}{n})$ collisions for each substring of T. This solves our problem and returns the total runtime to $O(n)$ but we may not necessarily have the resources to create a large table like that.

Instead, we will take advantage of string **signatures**. In addition to inserting the actual substring into the hash table, we will also associate each substring with another hash value, $h_s(k)$. Note that this hash value is different from the one we used to insert the substring into the hash table. The h_sk hash function actually maps strings to a range 0 to n^2 as opposed to 0 to n like $h(k)$. Now, when we have collisions inside the hash table, before we actually do the expensive string comparison operation, we first compare the signatures of the two strings. If the signatures of the two strings do not match, then we can skip the string comparison. For two substrings k_1 and k_2, only if $h(k_1) = h(k_2)$ and $h_s(k_1) = h_s(k_2)$ do we actually make the string comparison. For a well chosen $h_s(k)$ function, this will reduce the expected time spent doing string comparisons back to $O(n)$, keeping the common substring problem's runtime at $O(n)$.