String Matching Algorithms

Suffix Array

A suffix array is a data structure used to store all the suffixes of a given string and is frequently used for searching for a certain pattern within a string. It is an array of all the possible suffixes of the given string sorted in lexigraphical order. For example, the suffix array for the string "bananas" would be:

`[‘ananas’, ‘anas’, ‘as’, ‘bananas’, ‘nanas’, ‘nas’, ‘s’]`

The lexigraphical ordering gives us the advantage of searching for any possible substring with binary search instead of linear searches, meaning we only have to perform $\Theta(\log(n))$ instead of $\Theta(n)$ comparisons, which gives us $\Theta(n\log(n))$ plus the time to construct the suffix array instead of $\Theta(n^2)$ where n is the length of the string in which we are searching and the substrings can be as large as the text itself. The advantage is especially apparently when we have to search for m separate patters as we only have to construct the suffix tree once but get to save on all the binary searches. If we were to search for m substrings, repeated linear search would have a total runtime of $\Theta(mn^2)$ whereas repeated binary search would be $\Theta(mn\log(n))$ plus the time it takes to construct the suffix array.

To build this suffix array, a naive approach would be to run a simple sort. This would take $\Theta(n\log(n))$ comparisons, where each comparison takes $\Theta(n)$, so the total runtime is $\Theta(n^2\log(n))$. This is very bad. As we would preferably want the search to take more time than the construction. We can take a short cut by using the fact that the suffixes of all the suffixes in the suffix array are also in the suffix array.

Specifically, consider the example of "bananas," initially, we just have the array in the order the suffixes are generated:

`[‘bananas’, ‘ananas’, ‘nanas’, ‘anas’, ‘nas’, ‘as’, ‘s’]`

Now we sort them only by first letter, this can be done with a bucket sort in $O(n)$ time, giving us:

`[‘ananas’, ‘anas’, ‘as’, ‘bananas’, ‘nanas’, ‘nas’, ‘s’]`

This is not completely sorted, we have two unsorted ranges, specifically those starting with `a` and those starting with `n`. But note to solve those ranges we look to everything after the first letter, but those substrings are already present in the suffix array in approximately sorted order. Specifically, for those starting with `a`, even though we don't yet know the order of `ananas` and `anas` as `nanas` and `nas` are within an undetermined range, we can see that those two definitely precede `as` as `s` is further in the approximately sorted array than those starting with `n`.

Now we have the suffixes sorted by the first two letters, we can safely compare the first four, and after that we can safely compare the first eight. This technique is called **prefix doubling**, and allows us to construct the array in $O(n + n\log(n)) = O(n\log(n))$ time, with n spent doing the first bucket sort, and at most $\log(n)$ sorts to clean up the uncertain ranges.

Rabin-Karp Algorithm

Another way to quickly match strings is via a hashing algorithm. The annoying thing about string matching is that checking for equivalence is $O(N)$ where N is the length of the string instead of $O(1)$ for most other types (such as ints). One solution to this is to turn a string into an integer via a technique known as **polynomial hashing**.

The basic idea of polynomial hashing is very similar to how we represent numbers. When we see the number 1314, for example, what it represents is actually a polynomial where each digit is a coefficient evaluated at the base, i.e. $P(x) = 1x^3 + 3x^2 + 1x^1 + 4x^0$ evaluated at 10. Similarly the binary string of "101101" is equivalent to the polynomial $P(x) = 1x^5 + 0x^4 + 1x^3 + 1x^2 + 0x^1 + 1x^0$ evaluated at 2.

A similar approach can be approached to strings. Suppose we take the entire ASCII character set, where each character actually represents a digit in a base 128 system (ASCII values range from 0 to 127), we can rewrite any string s of length n as its hash value $H(s)$:

$$H(s) = \sum_{i=0}^{n} s_i 128^{(n-i)}$$

Where s_i is the i^{th} character's ASCII value. Note also that 128 was an arbitrary value I chose because of the size of the ASCII character set, it can really be changed to one more than the amount of characters to represent (because each character can be thought of a digit), and in the general case we call this value the base, or B. This works for short strings but for long strings we run into issues of H growing too large for long s. To solve this we take the function mod P where P is a large prime. We make P prime such that no combination of s and the base system will result in a multiple of P and create a collision (when two strings end up having the hash value). This, of, course, will not eliminate all collisions, if we try to hash $P + 1$ strings for example, two would still end up having the same hash value (by the Pigeon-Hole principle).

So the general has function $H(s)$ is:

$$\boxed{H(s) = \sum_{i=0}^{n} s_i B^{(n-i)} \mod P}$$

Hash Rolling

Consider the following problem: suppose we want to find a pattern, p, in a string s. We can of course, find a hash for p, $H(p)$ and find a hash for each substring of s with length the same as that of p. However, if we recompute the hash each time, it would take $\Theta(nm)$

complexity where n is the length of s and m is the length of p (n hashes to compute, each has takes m operations). What we are forgetting is that the substring $s[i, i+n]$ is only 2 letters away from $s[i+1, i+n+1]$. This gives us an efficient algorithm known as hash rolling to compare all m length hashes of s in $O(n)$ time.

The trick is if we call $s[i, i+n]$, H_s, we can compute the next hash by taking away the leading term of H_s, which is $s_i B^{(m-1)}$, multiplying what's left over by B, then adding s_{i+n+1}, thereby constructing the hash for $s[i+1, i+n+1]$.

With this method, we can find all occurrences of the pattern in the string in $\Theta(n+m)$ time, which is significantly better than $\Theta(mn)$.

Double Hashing

However, as said before, there are hash collisions once we have more than P strings. To solve this problem, we can either use hashes to generate a list of possible candidates, then only perform regular string comparison on them, but this runs into problems when the substring occurs very often in the text. Take the example of trying to find "aaa" in "aaaaaaaaaaaaaaaa." In scenarios like this, the runtime is reduced down to $O(mn)$, which is Rabin-Karp algorithm's worst runtime.

But we don't necessarily have to resort to this. More often than not we use something called **rational gambling**, which is based on the concept that the probability of two hash functions colliding at the same time is extremely low. And since each functions are determined solely by the base and modulus (P), we can create two hash functions H_1 and H_2, employing two different B, P pairs, and assume that every time both hash functions return the same value actually contain the same string. Of course, this is a probabilistic approach that doesn't guarantee 100% accuracy, but it works for most (read as "all") practical cases.