where for each character x, $V_x(\alpha, \beta, i)$ is computed by replacing each wild card with character x. In summary,

Theorem 4.3.1. *The match-count problem can be solved in $O(m \log m)$ time even if an unbounded number of wild cards are allowed in either P or T.*

Later, after discussing suffix trees and common ancestors, we will present in Section 9.3 a different, more comparison-based approach to handling wild cards that appear in both strings.

4.4. Karp–Rabin fingerprint methods for exact match

The *Shift-And* method assumes that we can efficiently shift a vector of bits, and the generalized *Shift-And* method assumes that we can efficiently increment an integer by one. If we treat a (row) bit vector as an integer number then a left shift by one bit results in the doubling of the number (assuming no bits fall off the left end). So it is not much of an extension to assume, in addition to being able to increment an integer, that we can also efficiently multiply an integer by two. With that added primitive operation we can turn the exact match problem (again without mismatches) into an arithmetic problem. The first result will be a simple linear-time method that has a very small probability of making an error. That method will then be transformed into one that never makes an error, but whose running time is only expected to be linear. We will explain these results using a binary string P and a binary text T. That is, the alphabet is first assumed to be just $\{0, 1\}$. The extension to larger alphabets is immediate and will be left to the reader.

4.4.1. Arithmetic replaces comparisons

Definition For a text string T, let T_r^n denote the n-length substring of T starting at character r. Usually, n is known by context, and T_r^n will be replaced by T_r.

Definition For the binary pattern P, let

$$H(P) = \sum_{i=1}^{i=n} 2^{n-i} P(i).$$

Similarly, let

$$H(T_r) = \sum_{i=1}^{i=n} 2^{n-i} T(r+i-1).$$

That is, consider P to be an n-bit binary number. Similarly, consider T_r^n to be an n-bit binary number. For example, if $P = 0101$ then $n = 4$ and $H(P) = 2^3 \times 0 + 2^2 \times 1 + 2^1 \times 0 + 2^0 \times 1 = 5$; if $T = 101101010$, $n = 4$, and $r = 2$, then $H(T_r) = 6$.

Clearly, if there is an occurrence of P starting at position r of T then $H(P) = H(T_r)$. However, the converse is also true, so

Theorem 4.4.1. *There is an occurrence of P starting at position r of T if and only if $H(P) = H(T_r)$.*

The proof, which we leave to the reader, is an immediate consequence of the fact that every integer can be written in a unique way as the sum of positive powers of two.

Theorem 4.4.1 converts the exact match problem into a numerical problem, comparing the two numbers $H(P)$ and $H(T_r)$ rather than directly comparing characters. But unless the pattern is fairly small, the computation of $H(P)$ and $H(T_r)$ will not be efficient.[2] The problem is that the required powers of two used in the definition of $H(P)$ and $H(T_r)$ grow large too rapidly. (From the standpoint of complexity theory, the use of such large numbers violates the unit-time *random access machine* (*RAM*) model. In that model, the largest allowed numbers must be represented in $O[\log(n+m)]$ bits, but the number 2^n requires n bits. Thus the required numbers are exponentially too large.) Even worse, when the alphabet is not binary but say has t characters, then numbers as large as t^n are needed.

In 1987 R. Karp and M. Rabin [266] published a method (devised almost ten years earlier), called the *randomized fingerprint* method, that preserves the spirit of the above numerical approach, but that is extremely efficient as well, using numbers that satisfy the *RAM* model. It is a *randomized* method where the *only if* part of Theorem 4.4.1 continues to hold, but the *if* part does not. Instead, the *if* part will hold *with high probability*. This is explained in detail in the next section.

4.4.2. Fingerprints of P and T

The general idea is that, instead of working with numbers as large as $H(P)$ and $H(T_r)$, we will work with those numbers *reduced modulo* a relatively small integer p. The arithmetic will then be done on numbers requiring only a small number of bits, and so will be efficient. But the really attractive feature of this method is a proof that the probability of error can be made small if p is chosen randomly in a certain range. The following definitions and lemmas make this precise.

Definition For a positive integer p, $H_p(P)$ is defined as $H(P)$ mod p. That is $H_p(P)$ is the remainder of $H(P)$ after division by p. Similarly, $H_p(T_r)$ is defined as $H(T_r)$ mod p. The numbers $H_p(P)$ and $H_p(T_r)$ are called *fingerprints* of P and T_r.

Already, the utility of using fingerprints should be apparent. By reducing $H(P)$ and $H(T_r)$ modulo a number p, every fingerprint remains in the range 0 to $p-1$, so the size of a fingerprint does not violate the *RAM* model. But if $H(P)$ and $H(T_r)$ must be computed before they can be reduced modulo p, then we have the same problem of intermediate numbers that are too large. Fortunately, modular arithmetic allows one to reduce at any time (i.e., one can never reduce too much), so that the following generalization of Horner's rule holds:

Lemma 4.4.1. $H_p(P) = \{[\ldots(\{[P(1)\times 2 \bmod p + P(2)]\times 2 \bmod p + P(3)\}\times 2 \bmod p + P(4))\ldots] \bmod p + P(n)\} \bmod p$, *and no number ever exceeds* $2p$ *during the computation of* $H_p(P)$.

[2] One can more efficiently compute $H(T_{r+1})$ from $H(T_r)$ than by following the definition directly (and we will need that later on), but the time to do the updates is not the issue here.

For example, if $P = 101111$ and $p = 7$, then $H(P) = 47$ and $H_p(P) = 47 \bmod 7 = 5$. Moreover, this can be computed as follows:

$$\begin{aligned}
&1 \times 2 \bmod 7 + 0 = 2\\
&2 \times 2 \bmod 7 + 1 = 5\\
&5 \times 2 \bmod 7 + 1 = 4\\
&4 \times 2 \bmod 7 + 1 = 2\\
&2 \times 2 \bmod 7 + 1 = 5\\
&5 \bmod 7 = 5.
\end{aligned}$$

The point of Horner's rule is not only that the number of multiplications and additions required is linear, but that the intermediate numbers are always kept small.

Intermediate numbers are also kept small when computing $H_p(T_r)$ for any r, since that computation can be organized the way that $H_p(P)$ was. However, even greater efficiency is possible: For $r > 1$, $H_p(T_r)$ can be computed from $H_p(T_{r-1})$ with only a small *constant* number of operations. Since

$$H_p(T_r) = H(T_r) \bmod p$$

and

$$H(T_r) = 2 \times H(T_{r-1}) - 2^n T(r-1) + T(r+n-1),$$

it follows that

$$H_p(T_r) = [(2 \times H(T_{r-1}) \bmod p) - (2^n \bmod p) \times T(r-1) + T(r+n-1)] \bmod p.$$

Further,

$$2^n \bmod p = 2 \times (2^{n-1} \bmod p) \bmod p.$$

Therefore, each successive power of two taken mod p and each successive value $H_p(T_r)$ can be computed in constant time.

Prime moduli limit false matches

Clearly, if P occurs in T starting at position r then $H_p(P) = H_p(T_r)$, but now the converse does not hold for every p. That is, we cannot necessarily conclude that P occurs in T starting at r just because $H_p(P) = H_p(T_r)$.

Definition If $H_p(P) = H_p(T_r)$ but P does not occur in T starting at position r, then we say there is a *false match* between P and T at position r. If there is *some* position r such that there is a false match between P and T at r, then we say there is a false match between P and T.

The goal will be to choose a modulus p small enough that the arithmetic is kept efficient, yet large enough that the probability of a false match between P and T is kept small. The key comes from choosing p to be a *prime* number in the proper range and exploiting properties of prime numbers. We will state the needed properties of prime numbers without proof.

Definition For a positive integer u, $\pi(u)$ is the *number* of primes that are less than or equal to u.

The following theorem is a variant of the famous *prime number theorem*.

Theorem 4.4.2. $\frac{u}{\ln(u)} \leq \pi(u) \leq 1.26\frac{u}{\ln(u)}$, *where* $\ln(u)$ *is the base e logarithm of* u [383].

Lemma 4.4.2. *If $u \geq 29$, then the product of all the primes that are less than or equal to u is greater than 2^u* [383].

For example, for $u = 29$ the prime numbers less than or equal to 29 are 2, 5, 7, 11, 13, 17, 19, 23, and 29. Their product is 2,156,564,410 whereas 2^{29} is 536,870,912.

Corollary 4.4.1. If $u \geq 29$ and x is any number less than or equal to 2^u, then x has fewer than $\pi(u)$ (distinct) prime divisors.

PROOF Suppose x does have $k > \pi(u)$ distinct prime divisors $q_1, q_2, \ldots, q_k$. Then $2^u \geq x \geq q_1 q_2 \ldots q_k$ (the first inequality is from the statement of the corollary, and the second from the fact that some primes in the factorization of x may be repeated). But $q_1 q_2 \ldots q_k$ is at least as large as the product of the smallest k primes, which is greater than the product of the first $\pi(u)$ primes (by assumption that $k > \pi(u)$). However, the product of the primes less than or equal to u is greater than 2^u (by Lemma 4.4.2). So the assumption that $k > \pi(u)$ leads to the contradiction that $2^u > 2^u$, and the lemma is proved. □

The central theorem

Now we are ready for the central theorem of the Karp–Rabin approach.

Theorem 4.4.3. *Let P and T be any strings such that $nm \geq 29$, where n and m are the lengths of P and T, respectively. Let I be any positive integer. If p is a randomly chosen prime number less than or equal to I, then the probability of a false match between P and T is less than or equal to $\frac{\pi(nm)}{\pi(I)}$.*

PROOF Let R be the set of positions in T where P does *not* begin. That is, $s \in R$ if and only if P does not occur in T beginning at s. For each $s \in R$, $H(P) \neq H(T_s)$. Now consider the product $\Pi_{s \in R}(|H(P) - H(T_s)|)$. That product must be at most 2^{nm} since for any s, $H(P) - H(T_s) \leq 2^n$ (recall that we have assumed a binary alphabet). Applying Corollary 4.4.1, $\Pi_{s \in R}(|H(P) - H(T_s)|)$ has at most $\pi(nm)$ distinct prime divisors.

Now suppose a false match between P and T occurs at some position r of T. That means that $H(P) \bmod p = H(T_r) \bmod p$ and that p evenly divides $H(P) - H(T_r)$. Trivially then, p evenly divides $\Pi_{s \in R}(|H(P) - H(T_s)|)$, and so p is one of the prime divisors of that product. If p allows a false match to occur between P and T, then p must be one of a set of at most $\pi(nm)$ numbers. But p was chosen randomly from a set of $\pi(I)$ numbers, so the probability that p is a prime that allows a false match between P and T is at most $\frac{\pi(nm)}{\pi(I)}$. □

Notice that Theorem 4.4.3 holds for any choice of pattern P and text T such that $nm \geq 29$. The probability in the theorem is not taken over choices of P and T but rather over choices of prime p. Thus, this theorem does not make any (questionable) assumptions about P or T being random or generated by a Markov process, etc. It works for any P and T! Moreover, the theorem doesn't just bound the probability that a false match occurs at a fixed position r, it bounds the probability that there is even a single such position r in T. It is also notable that the analysis in the proof of the theorem feels "weak". That is, it only develops a very weak property of a prime p that allows a false match, namely being one of at most $\pi(nm)$ numbers that divide $\Pi_{s \in R}(|H(P) - H(T_s)|)$. This suggests that the true probability of a false match occurring between P and T is much less than the bound established in the theorem.

Theorem 4.4.3 leads to the following random fingerprint algorithm for finding all occurrences of P in T.

Random fingerprint algorithm

1. Choose a positive integer I (to be discussed in more detail below).
2. Randomly pick a prime number less than or equal to I, and compute $H_p(P)$. (Efficient randomized algorithms exist for finding random primes [331].)
3. For each position r in T, compute $H_p(T_r)$ and test to see if it equals $H_p(P)$. If the numbers are equal, then either declare a probable match or check explicitly that P occurs in T starting at that position r.

Given the fact that each $H_p(T_r)$ can be computed in constant time from $H_p(T_{r-1})$, the fingerprint algorithm runs in $O(m)$ time, excluding any time used to explicitly check a declared match. It may, however, be reasonable not to bother explicitly checking declared matches, depending on the probability of an error. We will return to the issue of checking later. For now, to fully analyze the probability of error, we have to answer the question of what I should be.

How to choose I

The utility of the fingerprint method depends on finding a good value for I. As I increases, the probability of a false match between P and T decreases, but the allowed size of p increases, increasing the effort needed to compute $H_p(P)$ and $H_p(T_r)$. Is there a good balance? There are several good ways to choose I depending on n and m. One choice is to take $I = nm^2$. With that choice the largest number used in the algorithm requires at most $4(\log n + \log m)$ bits, satisfying the *RAM* model requirement that the numbers be kept small as a function of the size of the input. But, what of the probability of a false match?

Corollary 4.4.2. When $I = nm^2$, the probability of a false match is at most $\frac{2.53}{m}$.

PROOF By Theorem 4.4.3 and the prime number theorem (Theorem 4.4.2), the probability of a false match is bounded by

$$\frac{\pi(nm)}{\pi(nm^2)} \leq 1.26\frac{nm}{nm^2}\frac{\ln(nm^2)}{\ln(nm)} = 1.26\frac{1}{m}\left[\frac{\ln(n)+2\ln(m)}{\ln(n)+\ln(m)}\right] \leq \frac{2.53}{m}. \qquad \square$$

A small example from [266] illustrates this bound. Take $n = 250$, $m = 4000$, and hence $I = 4 \times 10^9 < 2^{32}$. Then the probability of a false match is at most $\frac{2.53}{4000} < 10^{-3}$. Thus, with just a 32-bit fingerprint, for any P and T the probability that even a single one of the algorithm's declarations is wrong is bounded by 0.001.

Alternately, if $I = n^2m$ then the probability of a false match is $O(1/n)$, and since it takes $O(n)$ time to determine whether a match is false or real, the *expected* verification time would be constant. The result would be an $O(m)$ *expected* time method that never has a false match.

Extensions

If one prime is good, why not use several? Why not pick k primes $p_1, p_2, \ldots, p_k$ randomly and compute k fingerprints? For any position r, there can be an occurrence of P starting at r *only if* $H_{p_i}(P) = H_{p_i}(T_r)$ for *every* one of the k selected primes. We now define a false match between P and T to mean that there is an r such that P does not occur in T starting at r, but $H_{p_i}(P) = H_{p_i}(T_r)$ for *each* of the k primes. What now is the probability of a false match between P and T? One bound is fairly immediate and intuitive.

Theorem 4.4.4. *When k primes are chosen randomly between 1 and I and k fingerprints are used, the probability of a false match between P and T is at most $[\frac{\pi(nm)}{\pi(I)}]^k$.*

PROOF We saw in the proof of Theorem 4.4.3 that if p is a prime that allows $H_p(P) = H_p(T_r)$ at some position r where P does not occur, then p is in a set of at most $\pi(nm)$ integers. When k fingerprints are used, a false match can occur only if each of the k primes is in that set, and since the primes are chosen randomly (independently), the bound from Theorem 4.4.3 holds for each of the primes. So the probability that all the primes are in the set is bounded by $[\frac{\pi(nm)}{\pi(I)}]^k$, and the theorem is proved. □

As an example, if $k = 4$ and n, m, and I are as in the previous example, then the probability of a false match between P and T is at most by 10^{-12}. Thus, the probability of a false match is reduced dramatically, from 10^{-3} to 10^{-12}, while the computational effort of using four primes only increases by four times. For typical values of n and m, a small choice of k will assure that the probability of an error due to a false match is less than the probability of error due to a hardware malfunction.

Even lower limits on error

The analysis in the proof of Theorem 4.4.4 is again very weak, because it just multiplies the probability that each of the k primes allows a false match *somewhere* in T. However, for the algorithm to actually make an error at some specific position r, each of the primes must *simultaneously* allow a false match at the same r. This is an even less likely event. With this observation we can reduce the probability of a false match as follows:

Theorem 4.4.5. *When k primes are chosen randomly between 1 and I and k fingerprints are used, the probability of a false match between P and T is at most $m[\frac{\pi(n)}{\pi(I)}]^k$.*

PROOF Suppose that a false match occurs at some fixed position r. That means that each prime p_i must evenly divide $|H(P) - H(T_r)|$. Since $|H(P) - H(T_r)| \leq 2^n$, there are at most $\pi(n)$ primes that divide it. So each p_i was chosen randomly from a set of $\pi(I)$ primes and by chance is part of a subset of $\pi(n)$ primes. The probability of this happening at that fixed r is therefore $[\frac{\pi(n)}{\pi(I)}]^k$. Since there are m possible choices for r, the probability of a false match between P and T (i.e., the probability that there is such an r) is at most $m[\frac{\pi(n)}{\pi(I)}]^k$, and the theorem is proved. □

Assuming, as before, that $I = nm^2$, a little arithmetic (which we leave to the reader) shows

Corollary 4.4.3. When k primes are chosen randomly and used in the fingerprint algorithm, the probability of a false match between P and T is at most $(1.26)^k m^{-(2k-1)}(1 + 0.6 \ln m)^k$.

Applying this to the running example of $n = 250$, $m = 4000$, and $k = 4$ reduces the probability of a false match to at most 2×10^{-22}.

We mention one further refinement discussed in [266]. Returning to the case where only a single prime is used, suppose the algorithm explicitly checks that P occurs in T when $H_p(P) = H_p(T_r)$, and it finds that P does not occur there. Then one may be better off by picking a new prime to use for the continuation of the computation. This makes intuitive sense. Theorem 4.4.3 randomizes over the choice of primes and bounds the probability that a randomly picked prime will allow a false match anywhere in T. But once the prime has been shown to allow a false match, it is no longer random. It may well be a prime that

allows numerous false matches (a *demon seed*). Theorem 4.4.3 says nothing about how bad a particular prime can be. But by picking a new prime after each error is detected, we can apply Corollary 4.4.2 to each prime, establishing

Theorem 4.4.6. *If a new prime is randomly chosen after the detection of an error, then for any pattern and text the probability of t errors is at most $(\frac{2.53}{m})^t$.*

This probability falls so rapidly that one is effectively protected against a long series of errors on any particular problem instance. For additional probabilistic analysis of the Karp–Rabin method, see [182].

Checking for error in linear time

All the variants of the Karp–Rabin method presented above have the property that they find *all true* occurrences of P in T, but they may also find *false matches* – locations where P is declared to be in T, even though it is not there. If one checks for P at each declared location, this checking would seem to require $\Theta(nm)$ worst-case time, although the expected time can be made smaller. We present here an $O(m)$-time method, noted first by S. Muthukrishnan [336], that determines if any of the declared locations are false matches. That is, the method either verifies that the Karp–Rabin algorithm has found no false matches or it declares that there is at least one false match (but it may not be able to find all the false matches) in $O(m)$ time.

The method is related to Galil's extension of the Boyer–Moore algorithm (Section 3.2.2), but the reader need not have read that section. Consider a list $\mathcal{L}$ of (starting) locations in T where the Karp–Rabin algorithm declares P to be found. A *run* is a maximal interval of consecutive starting locations $l_1, l_2, \ldots, l_r$ in $\mathcal{L}$ such that every two successive numbers in the interval differ by at most $n/2$ (i.e., $l_{i+1} - l_i \leq n/2$). The method works on each run separately, so we first discuss how to check for false matches in a single run.

In a single run, the method explicitly checks for the occurrence of P at the first two positions in the run, l_1 and l_2. If P does not occur in both of those locations then the method has found a false match and stops. Otherwise, when P does occur at both l_1 and l_2, the method learns that P is semiperiodic with period $l_2 - l_1$ (see Lemma 3.2.3). We use d to refer to $l_2 - l_1$, and we show that d is the smallest period of P. If d is not the smallest period, then d must be a multiple of the smallest period, say d'. (This follows easily from the GCD Theorem, which is stated in Section 16.17.1.) (page 431). But that implies that there is an occurrence of P starting at position $l_1 + d' < d_2$, and since the Karp–Rabin method *never* misses any occurrence of P, that contradicts the choice of l_2 as the second occurrence of P in the interval between l_1 and l_r. So d must be the smallest period of P, and it follows that if there are no false matches in the run, then $l_{i+1} - l_i = d$ for each i in the run. Hence, as a first check, the method verifies that $l_{i+1} - l_i = d$ for each i; it declares a false match and stops if this check fails for some i. Otherwise, as in the Galil method, to check each location in $\mathcal{L}$, it suffices to *successively* check the last d characters in each declared occurrence of P against the last d characters of P. That is, for position l_i, the method checks the d characters of T starting at position $l_i + n - d$. If any of these successive checks finds a mismatch, then the method has found a false match in the run and stops. Otherwise, P does in fact occur starting at each declared location in the run.

For the time analysis, note first that no character of T is examined more than twice during a check of a single run. Moreover, since two runs are separated by at least $n/2$ positions and each run is at least n positions long, no character of T can be examined in

more than two consecutive runs. It follows that the total time for the method, over all runs, is $O(m)$.

With the ability to check for false matches in $O(m)$ time, the Karp–Rabin algorithm can be converted from a method with a small probability of error that runs in $O(m)$ worst-case time, to one that makes no error, but runs in $O(m)$ *expected* time (a conversion from a Monte Carlo algorithm to a Las Vegas algorithm). To achieve this, simply (re)run and (re)check the Karp–Rabin algorithm until no false matches are detected. We leave the details as an exercise.

4.4.3. Why fingerprints?

The Karp–Rabin fingerprint method runs in linear worst-case time, but with a nonzero (though extremely small) chance of error. Alternatively, it can be thought of as a method that never makes an error and whose expected running time is linear. In contrast, we have seen several methods that run in linear worst-case time and never make errors. So what is the point of studying the Karp–Rabin method?

There are three responses to this question. First, from a practical standpoint, the method is simple and can be extended to other problems, such as two-dimensional pattern matching with odd pattern shapes – a problem that is more difficult for other methods. Second, the method is accompanied by concrete proofs, establishing significant properties of the method's performance. Methods similar in spirit to fingerprints (or filters) predate the Karp–Rabin method, but, unlike the Karp–Rabin method, they generally lack any theoretical analysis. Little has been proven about their performance. But the main attraction is that the method is based on very different *ideas* than the linear-time methods that guarantee no error. Thus the method is included because a central goal of this book is to present a diverse collection of ideas used in a range of techniques, algorithms, and proofs.

4.5. Exercises

1. Evaluate empirically the *Shift-And* method against methods discussed earlier. Vary the sizes of P and T.
2. Extend the *agrep* method to solve the problem of finding an "occurrence" of a pattern P inside a text T, when a small number of insertions and deletions of characters, as well as mismatches, are allowed. That is, characters can be inserted into P and characters can be deleted from P.
3. Adapt *Shift-And* and *agrep* to handle a set of patterns. Can you do better than just handling each pattern in the set independently?
4. Prove the correctness of the *agrep* method.
5. Show how to efficiently handle wild cards (both in the pattern and the text) in the *Shift-And* approach. Do the same for *agrep*. Show that the efficiency of neither method is affected by the number of wild cards in the strings.
6. Extend the *Shift-And* method to efficiently handle regular expressions that do not use the Kleene closure. Do the same for *agrep*. Explain the utility of these extensions to collections of biosequence patterns such as those in PROSITE.
7. We mentioned in Exercise 32 of Chapter 3 that PROSITE patterns often specify a range for the number of times that a subpattern repeats. Ranges of this type can be easily handled by the $O(nm)$ regular expression pattern matching method of Section 3.6. Can such range