Philosophers gathered from far and near
To sit at his feet and hear and hear,
Though he never was heard
To utter a word
But "Abracadabra, abracadab,
Abracada, abracad,
Abraca, abrac, abra, ab!"
'Twas all he had,
'Twas all they wanted to hear, and each
Made copious notes of the mystical speech,
Which they published next –
A trickle of text
In the meadow of commentary.
Mighty big books were these,
In a number, as leaves of trees;
In learning, remarkably – very!

— Jamrach Holobom, quoted by Ambrose Bierce,
The Devil's Dictionary (1911)

Why are our days numbered and not, say, lettered?

— Woody Allen, "Notes from the Overfed", *The New Yorker* (March 16, 1968)

7 String Matching

7.1 Brute Force

The basic object that we consider in this lecture note is a *string*, which is really just an array. The elements of the array come from a set Σ called the *alphabet*; the elements themselves are called *characters*. Common examples are ASCII text, where each character is an seven-bit integer, strands of DNA, where the alphabet is the set of nucleotides $\{A, C, G, T\}$, or proteins, where the alphabet is the set of 22 amino acids.

The problem we want to solve is the following. Given two strings, a ***text*** $T[1..n]$ and a ***pattern*** $P[1..m]$, find the first *substring* of the text that is the same as the pattern. (It would be easy to extend our algorithms to find *all* matching substrings, but we will resist.) A substring is just a contiguous subarray. For any *shift* s, let T_s denote the substring $T[s..s+m-1]$. So more formally, we want to find the smallest shift s such that $T_s = P$, or report that there is no shift. For example, if the text is the string 'AMANAPLANACATACANALPANAMA'[1] and the pattern is 'CAN', then the output should be 15. If the pattern is 'SPAM', then the answer should be None. In most cases the pattern is much smaller than the text; to make this concrete, I'll assume that $m < n/2$.

[1]Dan Hoey (or rather, Dan Hoey's computer program) found the following 540-word palindrome in 1984. We have better online dictionaries now, so I'm sure you could do better.

> A man, a plan, a caret, a ban, a myriad, a sum, a lac, a liar, a hoop, a pint, a catalpa, a gas, an oil, a bird, a yell, a vat, a caw, a pax, a wag, a tax, a nay, a ram, a cap, a yam, a gay, a tsar, a wall, a car, a luger, a ward, a bin, a woman, a vassal, a wolf, a tuna, a nit, a pall, a fret, a watt, a bay, a daub, a tan, a cab, a datum, a gall, a hat, a fag, a zap, a say, a jaw, a lay, a wet, a gallop, a tug, a trot, a trap, a tram, a torr, a caper, a top, a tonk, a toll, a ball, a fair, a sax, a minim, a tenor, a bass, a passer, a capital, a rut, an amen, a ted, a cabal, a tang, a sun, an ass, a maw, a sag, a jam, a dam, a sub, a salt, an axon, a sail, an ad, a wadi, a radian, a room, a rood, a rip, a tad, a pariah, a revel, a reel, a reed, a pool, a plug, a pin, a peek, a parabola, a dog, a pat, a cud, a nu, a fan, a pal, a rum, a nod, an eta, a lag, an eel, a batik, a mug, a mot, a nap, a maxim, a mood, a leek, a grub, a gob, a gel, a drab, a citadel, a total, a cedar, a tap, a gag, a rat, a manor, a bar, a gal, a cola, a pap, a yaw, a tab, a raj, a gab, a nag, a pagan, a bag, a jar, a bat, a way, a papa, a local, a gar, a baron, a mat, a rag, a gap, a tar, a decal, a tot, a led, a tic, a bard, a leg, a bog, a burg, a keel, a doom, a mix, a map, an atom, a gum, a kit, a baleen, a gala, a ten, a don, a mural, a pan, a faun, a ducat, a pagoda, a lob, a rap, a keep, a nip, a gulp, a loop, a deer, a leer, a lever, a hair, a pad, a tapir, a door, a moor, an aid, a raid, a wad, an alias, an ox, an atlas, a bus, a madam, a jag, a saw, a mass, an anus, a gnat, a lab, a cadet, an em, a natural, a tip, a caress, a pass, a baronet, a minimax, a sari, a fall, a ballot, a knot, a pot, a rep, a carrot, a mart, a part, a tort, a gut, a poll, a gateway, a law, a jay, a sap, a zag, a fat, a hall, a gamut, a dab, a can, a tabu, a day, a batt, a waterfall, a patina, a nut, a flow, a lass, a van, a mow, a nib, a draw, a regular, a call, a war, a stay, a gam, a yap, a cam, a ray, an ax, a tag, a wax, a paw, a cat, a valley, a drib, a lion, a saga, a plat, a catnip, a pooh, a rail, a calamus, a dairyman, a bater, a canal—Panama!

Indeed, Peter Norvig *has* done better.

Here's the "obvious" brute force algorithm, but with one immediate improvement. The inner while loop compares the substring T_s with P. If the two strings are not equal, this loop stops at the first character mismatch.

```
ALMOSTBRUTEFORCE($T[1..n], P[1..m]$):
  for $s \leftarrow 1$ to $n-m+1$
      $equal \leftarrow$ TRUE
      $i \leftarrow 1$
      while $equal$ and $i \le m$
          if $T[s+i-1] \ne P[i]$
              $equal \leftarrow$ FALSE
          else
              $i \leftarrow i+1$
      if $equal$
          return $s$
  return NONE
```

In the worst case, the running time of this algorithm is $O((n-m)m) = O(nm)$, and we can actually achieve this running time by searching for the pattern AAA...AAAB with $m-1$ A's, in a text consisting of n A's.

In practice, though, breaking out of the inner loop at the first mismatch makes this algorithm quite practical. We can wave our hands at this by assuming that the text and pattern are both random. Then on average, we perform a constant number of comparisons at each position i, so the total expected number of comparisons is $O(n)$. Of course, neither English nor DNA is really random, so this is only a heuristic argument.

7.2 Strings as Numbers

For the moment, let's assume that the alphabet consists of the ten digits 0 through 9, so we can interpret any array of characters as either a string or a decimal number. In particular, let p be the numerical value of the pattern P, and for any shift s, let t_s be the numerical value of T_s:

$$p = \sum_{i=1}^{m} 10^{m-i} \cdot P[i] \qquad t_s = \sum_{i=1}^{m} 10^{m-i} \cdot T[s+i-1]$$

For example, if $T =$ 3141592653589793$\underline{2384}$626433832795028841971 and $m = 4$, then $t_{17} = 2384$.

Clearly we can rephrase our problem as follows: Find the smallest s, if any, such that $p = t_s$. We can compute p in $O(m)$ arithmetic operations, without having to explicitly compute powers of ten, using *Horner's rule*:

$$p = P[m] + 10\Big(P[m-1] + 10\big(P[m-2] + \cdots + 10\big(P[2] + 10 \cdot P[1]\big)\cdots\big)\Big)$$

We could also compute any t_s in $O(m)$ operations using Horner's rule, but this leads to essentially the same brute-force algorithm as before. But once we know t_s, we can actually compute t_{s+1} in constant time just by doing a little arithmetic — subtract off the most significant digit $T[s] \cdot 10^{m-1}$, shift everything up by one digit, and add the new least significant digit $T[r+m]$:

$$t_{s+1} = 10\big(t_s - 10^{m-1} \cdot T[s]\big) + T[s+m]$$

To make this fast, we need to precompute the constant 10^{m-1}. (And we know how to do that quickly, right?) So at least intuitively, it looks like we can solve the string matching problem in $O(n)$ worst-case time using the following algorithm:

```
NumberSearch($T[1..n], P[1..m]$):
    $\sigma \leftarrow 10^{m-1}$
    $p \leftarrow 0$
    $t_1 \leftarrow 0$
    for $i \leftarrow 1$ to $m$
        $p \leftarrow 10 \cdot p + P[i]$
        $t_1 \leftarrow 10 \cdot t_1 + T[i]$
    for $s \leftarrow 1$ to $n - m + 1$
        if $p = t_s$
            return $s$
        $t_{s+1} \leftarrow 10 \cdot \big(t_s - \sigma \cdot T[s]\big) + T[s+m]$
    return None
```

Unfortunately, the most we can say is that the number of *arithmetic operations* is $O(n)$. These operations act on numbers with up to m digits. Since we want to handle arbitrarily long patterns, we can't assume that each operation takes only constant time! In fact, if we want to avoid expensive multiplications in the second-to-last line, we should represent each number as a string of decimal digits, which brings us back to our original brute-force algorithm!

7.3 Karp-Rabin Fingerprinting

To make this algorithm efficient, we will make one simple change, proposed by Richard Karp and Michael Rabin in 1981:

$$\boxed{\text{Perform all arithmetic modulo some prime number } q.}$$

We choose q so that the value $10q$ fits into a standard integer variable, so that we don't need any fancy long-integer data types. The values $(p \bmod q)$ and $(t_s \bmod q)$ are called the *fingerprints* of P and T_s, respectively. We can now compute $(p \bmod q)$ and $(t_1 \bmod q)$ in $O(m)$ *time* using Horner's rule:

$$p \bmod q = P[m] + \big(\cdots + \big(10 \cdot \big(P[2] + \big(10 \cdot P[1] \bmod q\big) \bmod q\big) \bmod q\big) \cdots\big)\big) \bmod q.$$

Similarly, given $(t_s \bmod q)$, we can compute $(t_{s+1} \bmod q)$ in constant time as follows:

$$t_{s+1} \bmod q = \big(10 \cdot \big(t_s - \big(\big(10^{m-1} \bmod q\big) \cdot T[s] \bmod q\big) \bmod q\big) \bmod q\big) + T[s+m] \bmod q.$$

Again, we have to precompute the value $(10^{m-1} \bmod q)$ to make this fast.

If $(p \bmod q) \neq (t_s \bmod q)$, then certainly $P \neq T_s$. However, if $(p \bmod q) = (t_s \bmod q)$, we can't tell whether $P = T_s$ or not. All we know for sure is that p and t_s differ by some integer multiple of q. If $P \neq T_s$ in this case, we say there is a *false match* at shift s. To test for a false match, we simply do a brute-force string comparison. (In the algorithm below, $\tilde{p} = p \bmod q$ and $\tilde{t}_s = t_s \bmod q$.) The overall running time of the algorithm is $O(n + Fm)$, where F is the number of false matches.

Intuitively, we expect the fingerprints t_s to jump around between 0 and $q-1$ more or less at random, so the 'probability' of a false match 'ought' to be $1/q$. This intuition implies that $F = n/q$ "on average", which gives us an 'expected' running time of $O(n + nm/q)$. If we always choose $q \geq m$, this bound simplifies to $O(n)$.

But of course all this intuitive talk of probabilities is meaningless hand-waving, since we haven't actually done anything random yet! There are two simple methods to formalize this intuition.

7.3.1 Random Prime Numbers

The algorithm that Karp and Rabin actually proposed chooses the prime modulus q *randomly* from a sufficiently large range.

```
KarpRabin($T[1..n], P[1..m]$):
  $q \leftarrow$ a random prime number between 2 and $\lceil m^2 \lg m \rceil$
  $\sigma \leftarrow 10^{m-1} \bmod q$
  $\tilde{p} \leftarrow 0$
  $\tilde{t}_1 \leftarrow 0$
  for $i \leftarrow 1$ to $m$
      $\tilde{p} \leftarrow (10 \cdot \tilde{p} \bmod q) + P[i] \bmod q$
      $\tilde{t}_1 \leftarrow (10 \cdot \tilde{t}_1 \bmod q) + T[i] \bmod q$
  for $s \leftarrow 1$ to $n - m + 1$
      if $\tilde{p} = \tilde{t}_s$
          if $P = T_s$        ⟨⟨brute-force $O(m)$-time comparison⟩⟩
              return $s$
      $\tilde{t}_{s+1} \leftarrow \big(10 \cdot \big(\tilde{t}_s - \big(\sigma \cdot T[s] \bmod q\big) \bmod q\big) \bmod q\big) + T[s+m] \bmod q$
  return None
```

For any positive integer u, let $\pi(u)$ denote the number of prime numbers less than u. There are $\pi(m^2 \log m)$ possible values for q, each with the same probability of being chosen. Our analysis needs two results from number theory. I won't even try to prove the first one, but the second one is quite easy.

Lemma 1 (The Prime Number Theorem). $\pi(u) = \Theta(u/\log u)$.

Lemma 2. *Any integer x has at most $\lfloor \lg x \rfloor$ distinct prime divisors.*

Proof: If x has k distinct prime divisors, then $x \geq 2^k$, since every prime number is bigger than 1. □

Suppose there are no true matches, since a true match can only end the algorithm early, so $p \neq t_s$ for all s. There is a false match at shift s if and only if $\tilde{p} = \tilde{t}_s$, or equivalently, if q is one of the prime divisors of $|p - t_s|$. Because $p < 10^m$ and $t_s < 10^m$, we must have $|p - t_s| < 10^m$. Thus, Lemma 2 implies that $|p - t_s|$ has at most $O(m)$ prime divisors. We chose q randomly from a set of $\pi(m^2 \log m) = \Omega(m^2)$ prime numbers, so the probability of a false match at shift s is $O(1/m)$. Linearity of expectation now implies that the expected number of false matches is $O(n/m)$. We conclude that KarpRabin runs in $O(n + \mathrm{E}[F]m) = \boldsymbol{O(n)}$ ***expected time***.

Actually choosing a random prime number is not particularly easy; the best method known is to repeatedly generate a random integer and test whether it's prime. The Prime Number Theorem implies that we will find a prime number after $O(\log m)$ iterations. Testing whether a number x is prime by brute force requires roughly $O(\sqrt{x})$ divisions, each of which require $O(\log^2 x)$ time if we use standard long division. So the total time to choose q using this brute-force method is about $O(m \log^3 m)$. There are faster algorithms to test primality, but they are considerably more complex. In practice, it's enough to choose a random *probable* prime. Unfortunately, even describing what the phrase "probable prime" means is beyond the scope of this note.

7.3.2 Polynomial Hashing

A much simpler method relies on a classical string-hashing technique proposed by Lawrence Carter and Mark Wegman in the late 1970s. Instead of generating the prime modulus randomly, we generate *the radix of our number representation* randomly. Equivalently, we treat each string as the coefficient vector of a polynomial of degree $m-1$, and we evaluate that polynomial at some random number.

```
CARTERWEGMANKARPRABIN($T[1..n], P[1..m]$):
    $q \leftarrow$ an arbitrary prime number larger than $m^2$
    $b \leftarrow$ RANDOM$(q) - 1$          ⟨⟨uniform between 0 and $q-1$⟩⟩
    $\sigma \leftarrow b^{m-1} \bmod q$
    $\tilde{p} \leftarrow 0$
    $\tilde{t}_1 \leftarrow 0$
    for $i \leftarrow 1$ to $m$
        $\tilde{p} \leftarrow (b \cdot \tilde{p} \bmod q) + P[i] \bmod q$
        $\tilde{t}_1 \leftarrow (b \cdot \tilde{t}_1 \bmod q) + T[i] \bmod q$
    for $s \leftarrow 1$ to $n-m+1$
        if $\tilde{p} = \tilde{t}_s$
            if $P = T_s$          ⟨⟨brute-force $O(m)$-time comparison⟩⟩
                return $s$
        $\tilde{t}_{s+1} \leftarrow \big(b \cdot \big(\tilde{t}_s - \big(\sigma \cdot T[s] \bmod q\big) \bmod q\big) \bmod q\big) + T[s+m] \bmod q$
    return NONE
```

Fix an arbitrary prime number $q \geq m^2$, and choose b uniformly at random from the set $\{0, 1, \ldots, q-1\}$. We redefine the numerical values p and t_s using b in place of the alphabet size:

$$p(b) = \sum_{i=1}^{m} b^{m-i} \cdot P[i] \qquad t_s(b) = \sum_{i=1}^{m} b^{m-i} \cdot T[s+i-1],$$

Now define $\tilde{p}(b) = p(b) \bmod q$ and $\tilde{t}_s(b) = t_s(b) \bmod q$.

The function $f(b) = \tilde{p}(b) - \tilde{t}_s(b)$ is a polynomial of degree $m-1$ over the *variable* b. Because q is prime, the set $\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$ with addition and multiplication modulo q defines a *field*. A standard theorem of abstract algebra states that any polynomial with degree $m-1$ over a field has at most $m-1$ roots in that field. Thus, there are at most $m-1$ elements $b \in \mathbb{Z}_q$ such that $f(b) = 0$.

It follows that if $P \neq T_s$, the probability of a false match at shift s is $\Pr_b[\tilde{p}(b) = \tilde{t}_s(b)] \leq (m-1)/q < 1/m$. Linearity of expectation now implies that the expected number of false positives is $O(n/m)$, so the modified Rabin-Karp algorithm also runs in ***O(n) expected time***.

7.4 Redundant Comparisons (Dynamic Programming)

Let's go back to the character-by-character method for string matching. Suppose we are looking for the pattern ABRACADABRA in some longer text using the (almost) brute force algorithm described in the previous lecture. Suppose also that when $s = 11$, the substring comparison fails at the fifth position; the corresponding character in the text (just after the vertical line below) is not a C. At this point, our algorithm would increment s and start the substring comparison from scratch.

HOCUSPOCUSABRA|BRACADABRA. . .
ABRA|~~CADABRA~~
ABR|ACADABRA

If we look carefully at the text and the pattern, however, we should notice right away that there's no point in looking at $s = 12$. We already know that the next character is a B — after all, it matched $P[2]$ during the previous comparison — so why bother even looking there? Likewise, we already know that the next two shifts $s = 13$ and $s = 14$ will also fail, so why bother looking there?

HOCUSPOCUSABRA | BRACADABRA...
ABRA | ~~$\cancel{\text{C}}$ADABRA~~
~~$\cancel{\text{A}}$BR~~ | ~~ACADABRA~~
~~$\cancel{\text{A}}$B~~ | ~~RACADABRA~~
A | BRACADABRA

Finally, when we get to $s = 15$, we can't immediately rule out a match based on earlier comparisons. However, for precisely the same reason, we shouldn't start the substring comparison over from scratch — we already know that $T[15] = P[4] =$ A. Instead, we should start the substring comparison at the *second* character of the pattern, since we don't yet know whether or not it matches the corresponding text character.

If you play with this idea long enough, you'll notice that the character comparisons should always advance through the text. **Once we've found a match for a text character, we never need to do another comparison with that text character again.** In other words, we should be able to optimize the brute-force algorithm so that it always *advances* through the text.

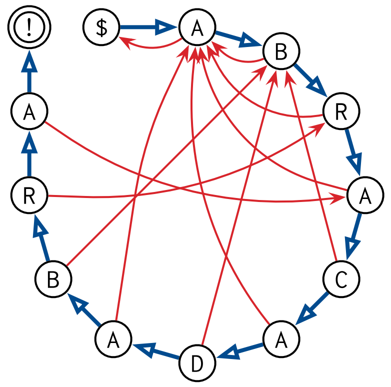
You'll also eventually notice a good rule for finding the next 'reasonable' shift s. A *prefix* of a string is a substring that includes the first character; a *suffix* is a substring that includes the last character. A prefix or suffix is *proper* if it is not the entire string. Suppose we have just discovered that $T[i] \neq P[j]$. **The next reasonable shift is the smallest value of s such that $T[s\,..\,i-1]$, which is a suffix of the previously-read text, is also a proper prefix of the pattern.**

in 1977, Donald Knuth, James Morris, and Vaughn Pratt published a string-matching algorithm that implements both of these ideas.[2]

7.5 Finite State Machines

We can interpret any string matching algorithm that always advance through the text as feeding the text through a special type of ***finite-state machine***. A finite state machine is a directed graph. Each node (or ***state***) in the string-matching machine is labeled with a character from the pattern, except for two special nodes labeled $\textcircled{\$}$ and $\textcircled{!}$. Each node has two outgoing edges, a ***success*** edge and a ***failure*** edge. The success edges define a path through the characters of the pattern in order, starting at $\textcircled{\$}$ and ending at $\textcircled{!}$. Failure edges always point to earlier characters in the pattern.

[2]The history of this algorithm is rather convoluted. Morris discovered an equivalent algorithm in 1969 while implementing a text editor, as a means to reduce buffering. Sadly, the other programmers didn't understand Morris's routine, so they "fixed" it, which of course meant it no longer worked. Independently, in early 1970, Knuth extracted another version of the algorithm from a result of Cook that the language of any two-way deterministic pushdown automaton can be recognized in linear time. "This was the first time in Knuth's experience that automata theory had taught him how to solve a real programming problem better than he could solve it before." Knuth showed his algorithm to his PhD student Pratt, who cast it into the form described here. Knuth and Pratt wrote a manuscript about their algorithm, but before publishing it, Pratt showed the algorithm to Morris, who recognized it as his own. Morris and Pratt published a technical report about the algorithm in 1970, giving Knuth credit but for some reason not including him as an author. A second technical report with all three authors was published in 1974, introducing (and fully analyzing) the optimized failure function described in Section 7.7. The journal version finally appeared in 1977.



A finite state machine for the string ABRACADABRA.
Thick arrows are the success edges; thin arrows are the failure edges.

We can use this finite state machine to search for the pattern as follows. At all times, we have a current text character $T[i]$ and a current node in the graph, which is usually labeled by some pattern character $P[j]$. We iterate the following rules:

- If $T[i] = P[j]$, or if the current label is ($), follow the success edge to the next node and increment i. (So there is no failure edge from the start node ($).)
- If $T[i] \neq P[j]$, follow the failure edge back to an earlier node, but do not change i.

For the moment, let's simply assume that the failure edges are defined correctly—we'll see how to do that later. If we ever reach the node labeled (!), then we've found an instance of the pattern in the text, and if we run out of text characters ($i > n$) before we reach (!), then there is no match.

The finite state machine is really just a (very!) convenient metaphor. In a real implementation, we would not construct the entire graph. The success edges always traverse the pattern characters in order, and each state has exactly one outgoing failure edge, so we only have to remember the targets of the failure edges. We can encode this ***failure function*** in an array $\mathit{fail}[1..n]$, where for each index j, the failure edge from node j leads to node $\mathit{fail}[j]$. Following a failure edge back to an earlier state corresponds exactly, in our earlier formulation, to shifting the pattern forward. The failure function $\mathit{fail}[j]$ tells us how far to shift after a character mismatch $T[i] \neq P[j]$.

$P[i]$	A	B	R	A	C	A	D	A	B	R	A
$\mathit{fail}[i]$	0	1	1	1	2	1	2	1	2	3	4

Failure function for the string ABRACADABRA
(Compare with the finite state machine above.)

Finally, here's the actual algorithm. The first line calls a subroutine that computes the failure function. For now, let's just assume that this subroutine works correctly; again, we'll see exactly how it works later.

```
KnuthMorrisPratt($T[1..n], P[1..m]$):
    $fail[1..m] \gets$ ComputeFailure($P$)
    $j \gets 1$
    for $i \gets 1$ to $n$
        while $j > 0$ and $T[i] \neq P[j]$
            $j \gets fail[j]$
        if $j = m$        ⟨⟨Found it!⟩⟩
            return $i - m + 1$
        $j \gets j + 1$
    return None
```

Before we discuss the failure function in detail, let's analyze the running time of KnuthMorrisPratt under the assumption that the subroutine ComputeFailure computes it correctly. After each character comparison, either the characters match, so we increase i and j by one, or the characters do not match, so we decrease j and leave i alone.

- We can increment i at most $n-1$ times before we either find the complete pattern or run out of text. So there are at most $n-1$ matching comparisons.
- The number of times we decrease j cannot exceed the number of times we increment j. So there are at most $n-1$ non-matching comparisons.

Thus, the total number of character comparisons performed by KnuthMorrisPratt in the worst case is at most $2n - n = O(n)$.

7.6 Computing the Failure Function

We can now rephrase our second intuitive rule about how to choose a reasonable shift after a character mismatch $T[i] \neq P[j]$:

$P[1..fail[j]-1]$ is the longest proper prefix of $P[1..j-1]$
that is also a suffix of $T[1..i-1]$.

Notice, however, that if we are comparing $T[i]$ against $P[j]$, then we must have already matched the first $j-1$ characters of the pattern. In other words, we already know that $P[1..j-1]$ is a suffix of $T[1..i-1]$. Thus, we can rephrase the prefix-suffix rule as follows:

$P[1..fail[j]-1]$ is the longest proper prefix of $P[1..j-1]$
that is also a suffix of $\boldsymbol{P[1..j-1]}$.

This is the definition of the Knuth-Morris-Pratt failure function $fail[j]$ for all $j > 1$. By convention we set $fail[1] = 0$; this tells the KMP algorithm that if the first pattern character doesn't match, it should just give up and try the next text character.

We could easily compute the failure function in $O(m^3)$ time by checking, for each j, whether every prefix of $P[1..j-1]$ is also a suffix of $P[1..j-1]$, but this is not the fastest method. The following algorithm essentially uses the KMP search algorithm to look for the pattern inside itself!

```
ComputeFailure($P[1..m]$):
    $j \leftarrow 0$
    for $i \leftarrow 1$ to $m$
        $fail[i] \leftarrow j$            $(*)$
        while $j > 0$ and $P[i] \neq P[j]$
            $j \leftarrow fail[j]$
        $j \leftarrow j + 1$
    return $fail[1..m]$
```

Here's an example of this algorithm in action. In each line, the current values of i and j are indicated by superscripts; \$ represents the beginning of the string. (You should imagine pointing at $P[j]$ with your left hand and pointing at $P[i]$ with your right hand, and moving your fingers according to the algorithm's directions.)

$j \leftarrow 0, i \leftarrow 1$	$\$^j$	$\mathbf{A}^i$	B	R	A	C	A	D	A	B	R	X	…
$fail[i] \leftarrow j$		**0**											…
$j \leftarrow j+1, i \leftarrow i+1$	\$	$\mathbf{A}^j$	$\mathbf{B}^i$	R	A	C	A	D	A	B	R	X	…
$fail[i] \leftarrow j$		0	**1**										…
$j \leftarrow fail[j]$	$\$^j$	A	$\mathbf{B}^i$	R	A	C	A	D	A	B	R	X	…
$j \leftarrow j+1, i \leftarrow i+1$	\$	$\mathbf{A}^j$	B	$\mathbf{R}^i$	A	C	A	D	A	B	R	X	…
$fail[i] \leftarrow j$		0	1	**1**									…
$j \leftarrow fail[j]$	$\$^j$	A	B	$\mathbf{R}^i$	A	C	A	D	A	B	R	X	…
$j \leftarrow j+1, i \leftarrow i+1$	\$	$\mathbf{A}^j$	B	R	$\mathbf{A}^i$	C	A	D	A	B	R	X	…
$fail[i] \leftarrow j$		0	1	1	**1**								…
$j \leftarrow j+1, i \leftarrow i+1$	\$	A	$\mathbf{B}^j$	R	A	$\mathbf{C}^i$	A	D	A	B	R	X	…
$fail[i] \leftarrow j$		0	1	1	1	**2**							…
$j \leftarrow fail[j]$	\$	$\mathbf{A}^j$	B	R	A	$\mathbf{C}^i$	A	D	A	B	R	X	…
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	$\mathbf{C}^i$	A	D	A	B	R	X	…
$j \leftarrow j+1, i \leftarrow i+1$	\$	$\mathbf{A}^j$	B	R	A	C	$\mathbf{A}^i$	D	A	B	R	X	…
$fail[i] \leftarrow j$		0	1	1	1	2	**1**						…
$j \leftarrow j+1, i \leftarrow i+1$	\$	A	$\mathbf{B}^j$	R	A	C	A	$\mathbf{D}^i$	A	B	R	X	…
$fail[i] \leftarrow j$		0	1	1	1	2	1	**2**					…
$j \leftarrow fail[j]$	\$	$\mathbf{A}^j$	B	R	A	C	A	$\mathbf{D}^i$	A	B	R	X	…
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	C	A	$\mathbf{D}^i$	A	B	R	X	…
$j \leftarrow j+1, i \leftarrow i+1$	\$	$\mathbf{A}^j$	B	R	A	C	A	D	$\mathbf{A}^i$	B	R	X	…
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	**1**				…
$j \leftarrow j+1, i \leftarrow i+1$	\$	A	$\mathbf{B}^j$	R	A	C	A	D	A	$\mathbf{B}^i$	R	X	…
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	**2**			…
$j \leftarrow j+1, i \leftarrow i+1$	\$	A	B	$\mathbf{R}^j$	A	C	A	D	A	B	$\mathbf{R}^i$	X	…
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	2	**3**		…
$j \leftarrow j+1, i \leftarrow i+1$	\$	A	B	R	$\mathbf{A}^j$	C	A	D	A	B	R	$\mathbf{X}^i$	…
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	2	3	**4**	…
$j \leftarrow fail[j]$	\$	$\mathbf{A}^j$	B	R	A	C	A	D	A	B	R	$\mathbf{X}^i$	…
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	C	A	D	A	B	R	$\mathbf{X}^i$	…

ComputeFailure in action. Do this yourself by hand!

Just as we did for KnuthMorrisPratt, we can analyze ComputeFailure by amortizing character mismatches against earlier character matches. Since there are at most m character matches, ComputeFailure runs in $O(m)$ time.

Let's prove (by induction, of course) that ComputeFailure correctly computes the failure function. The base case $\mathit{fail}[1] = 0$ is obvious. Assuming inductively that we correctly computed $\mathit{fail}[1]$ through $\mathit{fail}[i-1]$ in line $(*)$, we need to show that $\mathit{fail}[i]$ is also correct. Just after the ith iteration of line $(*)$, we have $j = \mathit{fail}[i]$, so $P[1\,..\,j-1]$ is the longest proper prefix of $P[1\,..\,i-1]$ that is also a suffix.

Let's define the iterated failure functions $\mathit{fail}^c[j]$ inductively as follows: $\mathit{fail}^0[j] = j$, and

$$\mathit{fail}^c[j] = \mathit{fail}[\mathit{fail}^{c-1}[j]] = \overbrace{\mathit{fail}[\mathit{fail}[\cdots[\mathit{fail}}^{c}[j]]\cdots]].$$

In particular, if $\mathit{fail}^{c-1}[j] = 0$, then $\mathit{fail}^c[j]$ is undefined. We can easily show by induction that every string of the form $P[1\,..\,\mathit{fail}^c[j]-1]$ is both a proper prefix and a proper suffix of $P[1\,..\,i-1]$, and in fact, these are the only such prefixes. Thus, the longest proper prefix/suffix of $P[1\,..\,i]$ must be the longest string of the form $P[1\,..\,\mathit{fail}^c[j]]$—the one with smallest c—such that $P[\mathit{fail}^c[j]] = P[i]$. This is exactly what the while loop in ComputeFailure computes; the $(c+1)$th iteration compares $P[\mathit{fail}^c[j]] = P[\mathit{fail}^{c+1}[i]]$ against $P[i]$. ComputeFailure is actually a *dynamic programming* implementation of the following recursive definition of $\mathit{fail}[i]$:

$$\mathit{fail}[i] = \begin{cases} 0 & \text{if } i = 0,\\ \max\limits_{c\ge 1}\left\{\mathit{fail}^c[i-1]+1 \;\middle|\; P[i-1] = P[\mathit{fail}^c[i-1]]\right\} & \text{otherwise.}\end{cases}$$

7.7 Optimizing the Failure Function

We can speed up KnuthMorrisPratt slightly by making one small change to the failure function. Recall that after comparing $T[i]$ against $P[j]$ and finding a mismatch, the algorithm compares $T[i]$ against $P[\mathit{fail}[j]]$. With the current definition, however, it is possible that $P[j]$ and $P[\mathit{fail}[j]]$ are actually the same character, in which case the next character comparison will automatically fail. So why do the comparison at all?

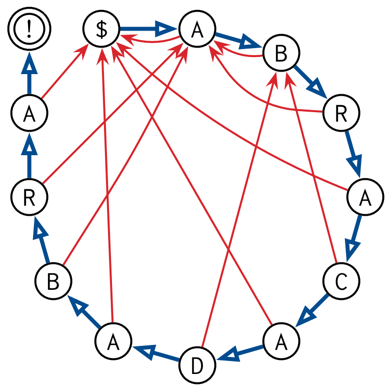
We can optimize the failure function by "short-circuiting" these redundant comparisons with some simple post-processing:

```
OptimizeFailure($P[1..m]$, $fail[1..m]$):
  for $i \leftarrow 2$ to $m$
      if $P[i] = P[fail[i]]$
          $fail[i] \leftarrow fail[fail[i]]$
```

We can also compute the optimized failure function directly by adding three new lines (in bold) to the ComputeFailure function.

```
ComputeOptFailure($P[1..m]$):
  $j \leftarrow 0$
  for $i \leftarrow 1$ to $m$
      if $P[i] = P[j]$
          $fail[i] \leftarrow fail[j]$
      else
          $fail[i] \leftarrow j$
      while $j > 0$ and $P[i] \ne P[j]$
          $j \leftarrow fail[j]$
      $j \leftarrow j + 1$
```

This optimization slows down the preprocessing slightly, but it may significantly decrease the number of comparisons at each text character. The worst-case running time is still $O(n)$; however, the constant is about half as big as for the unoptimized version, so this could be a significant improvement in practice. Several examples of this optimization are given on the next page.



$P[i]$	A	B	R	A	C	A	D	A	B	R	A
unoptimized *fail*$[i]$	0	1	1	1	2	1	2	1	2	3	4
optimized *fail*$[i]$	0	1	1	**0**	2	**0**	2	**0**	**1**	**1**	**1**

Optimized finite state machine and failure function for the string 'ABRADACABRA'

$P[i]$	A	N	A	N	A	B	A	N	A	N	A	N	A
unoptimized *fail*$[i]$	0	1	1	2	3	4	1	2	3	4	5	6	5
optimized *fail*$[i]$	0	1	0	1	0	4	0	1	0	1	0	6	0

$P[i]$	A	B	A	B	C	A	B	A	B	C	A	B	C
unoptimized *fail*$[i]$	0	1	1	2	3	1	2	3	4	5	6	7	8
optimized *fail*$[i]$	0	1	0	1	3	0	1	0	1	3	0	1	8

$P[i]$	A	B	B	A	B	B	A	B	A	B	B	A	B
unoptimized *fail*$[i]$	0	1	1	1	2	3	4	5	6	2	3	4	5
optimized *fail*$[i]$	0	1	1	0	1	1	0	1	6	1	1	0	1

$P[i]$	A	A	A	A	A	A	A	A	A	A	A	A	B
unoptimized *fail*$[i]$	0	1	2	3	4	5	6	7	8	9	10	11	12
optimized *fail*$[i]$	0	0	0	0	0	0	0	0	0	0	0	0	12

Failure functions for four more example strings.

★★★

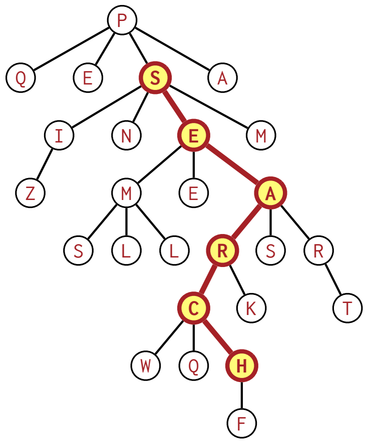
Manacher's palindrome-substring algorithm and the Aho-Corasick dictionary-matching algorithm build on the ideas developed by KMP. Boyer-Moore is far better in practice but harder to analyze, especially if we want $O(m+n)$ worst-case time. See also Gusfield's 1984 Z-algorithm, which is a variant of KMP preprocessing; for any fixed string S, Gusfield defines $Z[j]$ to be the length of the longest prefix of S that is also a prefix of $S[j\,..\,m]$.

Exercises

1. A *palindrome* is any string that is the same as its reversal, such as `X`, `ABBA`, or `REDIVIDER`. Describe and analyze an algorithm that computes the longest palindrome that is a (not necessarily proper) prefix of a given string $T[1..n]$. Your algorithm should run in $O(n)$ time (either expected or worst-case).

2. Describe and analyze an efficient algorithm to find strings in labeled rooted trees. Your input consists of a *pattern string* $P[1..m]$ and a rooted *text tree* T with n nodes, each labeled with a single character. Nodes in T can have any number of children. A path in T is called a *downward path* if every node on the path is a child (in T) of the previous node in the path. Your goal is to determine whether there is a downward path in T whose sequence of labels matches the string P.

   For example, the string `SEARCH` is the label of a downward path in the tree shown below, but the strings `HCRAES` and `SMEAR` is not.



3. Each of the following substring-matching problems can be solved in $O(n^2)$ time using dynamic programming.[3] For each problem, describe a faster algorithm; all of the necessary algorithmic tools are developed in this lecture note.[4]

   (a) Find the longest common substring of two given strings. For example, given the strings `ABRAHOCUSPOCUSCADABRA` and `HOCUSABRACADABRAPOCUS` as input, your algorithm should return the substring `CADABRA`.

   (b) Find the longest palindrome substring of a given string that is also a palindrome. For example, given the input string `PREDIVIDED`, your algorithm should `EDIVIDE`.

   (c) Given a string T, find the longest string w such that ww is a substring of T. For example, given the input string `BIPPITYBOPPITYBOO`, your algorithm should `PPITYBO`.

   (d) Find the longest substring that appears more than once in a given string. For example, given the input string `BIPPITYFLIPPITY`, your algorithm should `PPITY`, and given the input string `ABABABABA`, your algorithm should return the substring `ABABABA`.

[3]Make sure you know how to solve them that way!

[4]The *fastest* algorithms for many of these problems rely on *suffix trees* or *suffix arrays*. Unfortunately, a proper treatment of these data structures is beyond the scope of these notes. Maybe someday.

(e) Find the longest substring that appears both forward and backward in a given string *without overlapping*. For example, given the input string PR$\underline{\texttt{EDI}}$V$\underline{\texttt{IDE}}$D, your algorithm should return EDI, and given the input string $\underline{\texttt{ABAB}}\texttt{A}\underline{\texttt{BABA}}$, your algorithm should return the substring ABAB.

(f) Compute the number of times each prefix of a given string appears as a substring of that string. For example, the prefix BABABBA occurs three times in the string $\underline{\texttt{BABABBA}}\texttt{B}\underline{\texttt{BABAB}\underline{\texttt{BA}}}\underline{\texttt{BABBA}}\texttt{B}$.

4. A *cyclic shift* of a string w is any string obtained by moving a suffix of w to the beginning of the string, or equivalently, moving a prefix of w to the end of the string. For example, the strings ABRAABRACAD and ADABRAABRAC are cyclic shifts of the string ABRACADABRA.

   (a) Describe a fast algorithm to determine, given two strings A and B, whether A is a cyclic shift of B.

   (b) Describe a fast algorithm to determine, given two strings A and B, whether A is a substring of some cyclic shift of B.

   *(c) Describe a fast algorithm to determine, given two strings A and B, whether some cyclic shift of A is a substring of B.

5. A *fugue* (pronounced "fyoog") is a highly structured style of musical composition that was popular in the 17th and 18th centuries. A fugue begins with an initial melody, called the *subject*, that is repeated several times throughout the piece.

   Suppose we want to design an algorithm to detect the subject of a fugue. We will assume a *very* simple representation as an array $F[1..n]$ of integers, each representing a note in the fugue as the number of half-steps above or below middle C. (We are deliberately ignoring all other musical aspects of real-life fugues, like multiple voices, timing, rests, volume, and timbre.)

   (a) Describe an algorithm to find the length of the longest prefix of F that reappears later as a substring of F. The prefix and its later repetition must not overlap.

   (b) In many fugues, later occurrences of the subject are ***transposed***, meaning they are all shifted up or down by a common value. For example, the subject $(3,1,4,1,5,9,2)$ might be transposed transposed down two half-steps to $(1,-1,2,-1,3,7,0)$.

   Describe an algorithm to find the length of the longest prefix of F that reappears later, *possibly transposed*, as a substring of F. Again, the prefix and its later repetition must not overlap.

   For example, if the input array is

$$\underbracket{\overbracket{3,1,4,1},5,9,2},6,5,\overbracket{3,1,4,1},\!\!\underbracket{\;,-1,2,-1,3,7,0},1,4,2$$

   then your first algorithm should return 4, and your second algorithm should return 7.

6. Describe a modification of either KarpRabin or KnuthMorrisPratt where the pattern can contain any number of *wildcard* symbols $*$, each of which matches an arbitrary string. For example, the pattern ABR$*$CAD$*$BRA appears in the text SCH$\underline{\texttt{ABR}\textbf{AIN}\texttt{CADBRA}}$NCH; in this

case, the second * matches the empty string. Your algorithm should run in $O(m+n)$ time (either expected or worst-case), where m is the length of the pattern and n is the length of the text.

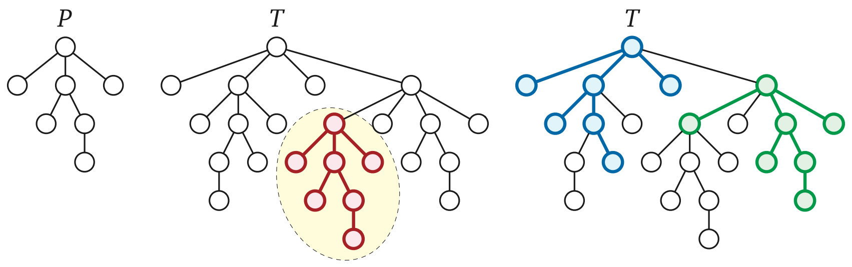
7. A *rooted ordered tree* is a rooted tree where every node has a (possibly empty) *sequence* of children. The order of these children matters. Two rooted ordered trees *match* if and only if their roots have the same number of children and, for each index i, the subtrees rooted at the ith children of both roots match (recursively). Any data stored in the nodes is ignored; we are only comparing the *shapes* of the trees.

   Suppose we are given two rooted ordered trees P and T, and we want to know whether P (the "pattern") occurs anywhere as a subtree of T (the "text"). There are two variants of this problem, depending on the precise definition of the word "subtree".

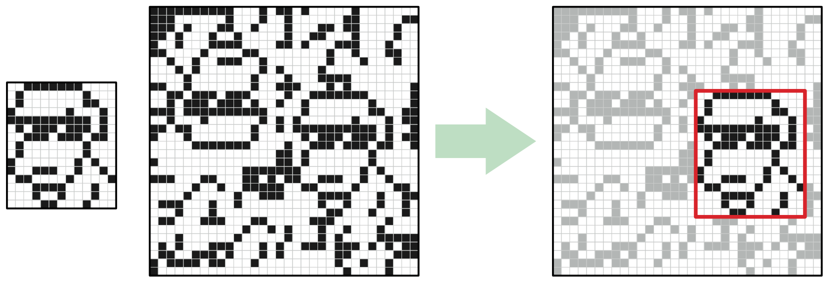
   (a) An *induced* subtree of T consists of some node and ***all*** its descendants in T, along with the edges of T between those vertices. Describe an algorithm to determine if P matches any induced subtree of T.

   (b) An *internal* subtree of T is any connected acyclic subgraph of T; the children of any node inherit their ordering from T. Every induced subtree is also an internal subtree, but not vice versa. Describe an algorithm to determine if P matches any internal subtree of T. *[Hint: See the previous problem.]*

   For example, the following figure shows a pattern tree P that matches exactly four internal subtrees of a text tree T, including exactly one induced subtree. (Only three of the matching internal subtrees are shown.)



8. (a) Describe an efficient algorithm to determine if a given $p \times q$ "pattern" bitmap P appears anywhere in a given $m \times n$ "text" bitmap T. The pattern may be shifted horizontally and/or vertically, but it may not be rotated or reflected.



*(b) Modify your algorithm from part (a) to return a list of *all* appearances of a given $p \times q$ "pattern" bitmap P in a given $m \times n$ "text" bitmap T. That is, your algorithm should compute all pairs (i, j) such that P matches the subarray $T[i+1 \,..\, i+p, j+1 \,..\, j+q]$.

*9. How important is the requirement that the fingerprint modulus q is prime in the original Karp-Rabin algorithm? Specifically, suppose q is chosen uniformly at random in the range $1 \,..\, n$. If $t_s \neq p$, what is the probability that $\tilde{t}_s = \tilde{p}$? What does this imply about the expected number of false matches? How large should N be to guarantee expected running time $O(m+n)$? *[Hint: This will require some additional number theory.]*

10. Describe a modification of KnuthMorrisPratt in which the pattern can contain any number of *wildcard* symbols ?, each of which matches an arbitrary single character. For example, the pattern ABR?CAD?BRA appears in the text SCHABR**U**CAD**I**BRANCH. Your algorithm should run in $O(m+qn)$ time, where m is the length of the pattern, n is the length of the text., and q is the number of ?s in the pattern.

*11. Describe another algorithm for the previous problem that runs in time $O(m+kn)$, where k is the number of runs of consecutive non-wildcard characters in the pattern. For example, the pattern ?FISH???B??IS????CUIT? has $k=4$ runs.

12. Describe a modification of KnuthMorrisPratt in which the pattern can contain any number of *wildcard* symbols =, each of which matches *the same* arbitrary single character. For example, the pattern =HOC=SPOC=S appears in the texts WH**U**HOC**U**SPOC**U**SOT and ABR**A**HOC**A**SPOC**A**SCADABRA, but *not* in the text FRI**S**HOC**U**SPOC**E**STIX. Your algorithm should run in $O(m+n)$ time, where m is the length of the pattern and n is the length of the text.

13. This problem considers the maximum length of a *failure chain* $j \to \mathit{fail}[j] \to \mathit{fail}[\mathit{fail}[j]] \to \mathit{fail}[\mathit{fail}[\mathit{fail}[j]]] \to \cdots \to 0$, or equivalently, the maximum number of iterations of the inner loop of KnuthMorrisPratt. This clearly depends on which failure function we use: unoptimized or optimized. Let m be an arbitrary positive integer.

    (a) Describe a pattern $A[1 \,..\, m]$ whose longest *unoptimized* failure chain has length m.

    (b) Describe a pattern $B[1 \,..\, m]$ whose longest *optimized* failure chain has length $\Theta(\log m)$.

    *(c) Describe a pattern $C[1 \,..\, m]$ *containing only two different characters*, whose longest optimized failure chain has length $\Theta(\log m)$.

    *(d) Prove that for any pattern of length m, the longest optimized failure chain has length at most $O(\log m)$. *[Hint: Prove that $m-1 > \mathit{fail}(m) + \mathit{fail}(\mathit{fail}(m)) - 2$]*