19 String Matching (April 8)

19.1 Brute Force

The basic object that we're going to talk about for the next two lectures 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[1], 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 match. For example, if the text is the string "AMANAPLANACATACANALPANAMA"[2] and the pattern is "LANA", then the output would be 7. If the target were "SPAM", then the answer would be "none". In most cases the pattern is much smaller than the text; to make this concrete, I'll assume that $m < n/2$.

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 .. n]):
  for s ← 1 to n − m − 1
      equal ← true
      i ← 1
      while equal and i ≤ m
          if T[s + i − 1] ≠ P[i]
              equal ← false
          else
              i ← i + 1
      if equal, return s
  return "none"
```

[1] Yes, seven. Most computer systems use some sort of 8-bit character set, but there's no universally accepted standard. Java is technically supposed to use the Unicode character set, which has variable-length characters and therefore doesn't really fit into our framework. (Just think ... Someday you'll be able to write "¶ = ℵ[∞++]/℧;" in your Java code!)

[2] Dan Hoey (or rather, his 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!

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.

19.2 Strings as Numbers

For the rest of the lecture, let's assume that the alphabet consists of the numbers 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 $T = 31415926535897932384626433832795028841971$ 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*:

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

We could compute any t_s in $O(m)$ operations using the same idea, 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 it seems that we can solve the string matching problem in $O(n)$ worst-case time using the following algorithm:

```
NumberSearch(T[1..n], P[1..n]):
    σ ← 10^(m−1)
    p ← 0,  t_1 ← 0
    for i ← 1 to m
        p ← 10 · p + P[i],  t_1 ← 10 · t_1 + P[i]
    for s ← 1 to n − m + 1
        if p = t_s, return s
        t_{s+1} ← 10 · (t_s − σ · T[s]) + 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!

19.3 Karp-Rabin Fingerprinting

To make this algorithm efficient, we will make one simple change, discovered 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 "modulo q"

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

and similarly, given $(t_s \bmod q)$, we can compute $(t_{s+1} \bmod q)$ in constant time.

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

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.

```
KarpRabin(T[1..n], P[1..n]):
    choose a small prime q
    σ ← 10^(m−1) mod q
    p ← 0,  t_1 ← 0
    for i ← 1 to m
        p ← (10 · p mod q) + P[i] mod q
        t_1 ← (10 · t_1 mod q) + P[i] mod q

    for s ← 1 to n − m + 1
        if p = t_s
            if P = T_s      // brute-force O(m)-time comparison
                return s
        t_{s+1} ← (10 · (t_s − (σ · T[s] mod q) mod q) mod q) + T[s + m] mod q

    return "none"
```

The running time of this 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 the average", which gives us an "expected" running time of $O(n + nm/q)$. If we always choose $q \geq m$, this simplifies to $O(n)$. But of course all this intuitive talk of probabilities is just frantic meaningless handwaving, since we haven't actually done anything random yet.

19.4 Random Prime Number Facts

The real power of the Karp-Rabin algorithm is that by choosing the modulus q *randomly*, we can actually formalize this intuition![3] The first line of KARPRABIN should really read as follows:

$$\boxed{\text{Let } q \text{ be a random prime number less than } nm^2.}$$

For any positive integer u, let $\pi(u)$ denote the number of prime numbers less than u. There are $\pi(nm^2)$ possible values for q, each with the same probability of being chosen.

Our analysis needs two results from number theory, which I won't even try to prove.

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

Lemma 2. *Any integer $x > 2^{29}$ has at most $\pi(\lfloor \lg x \rfloor) = O(\log x/\log\log x)$ prime divisors.*

Let's assume that there are no true matches, so $p \neq t_s$ for all s. (That's the worst case for the algorithm anyway.) Let's define a strange variable X as follows:

$$X = \prod_{s=1}^{n-m+1} |p - t_s|.$$

Notice that by our assumption, X can't be zero.

Now suppose we have false match at shift s. Then $p \bmod q = t_s \bmod q$, so $p - t_s$ is an integer multiple of q, and this implies that X is also an integer multiple of q. In other words, if there is a false match, then q must one of the prime divisors of X.

Since $p < 10^m$ and $t_s < 10^m$, we must have $X < 10^{nm}$. Thus, by the second lemma, X has $O(nm/\log(nm))$ prime divisors. Since we chose q randomly from a set of $\pi(nm^2) = \Omega(nm^2/\log(nm^2))$ prime numbers, the probability that q divides X is at most

$$\frac{O(nm/\log(nm))}{\Omega(nm^2/\log(nm^2))} = O\left(\frac{nm\log(nm^2)}{nm^2\log(nm)}\right) = O\left(\frac{1}{m}\right).$$

We have just proven the following amazing fact.

$$\boxed{\text{The probability of getting a false match is } O(1/m).}$$

Recall that the running time of KARPRABIN is $O(n + mF)$, where F is the number of false matches. By using the *really* loose upper bound $E[F] \leq \Pr[F > 0] \cdot n$, we can conclude that the expected number of false matches is $O(n/m)$. Thus, the expected running time of the KARPRABIN algorithm is $O(n)$.

[3][CLR] completely misses the *point* of the Karp-Rabin algorithm by saying " ... it is difficult to formalize and prove such an assertion ... ". Nonsense!