String Matching

String matching problems occur throughout computing - everything from mundane (but essential) utilities such as find-and-replace operations in word processors to literary analysis (how many times does Shakespeare use the word "eyebrow") to genetic research to internet search engines ... they all rest on searching for one string of characters (a pattern) inside another string of characters (a target). The naive algorithm can take a long time, but we will see that we can improve on it quite a bit.

Notation: Let T be the target string, with length **n**. The positions in T are indexed with 1 .. n

Let P be the pattern string, with length **m**. The positions in P are indexed with 1 .. m

Shift i corresponds to writing P below T so that P[1] aligns with T[1 + i]. We say that there is a **match at shift i** if P[1] = T[1+i], P[2] = T[2+i], ... P[m] = T[m+i]

Task: Given T and P, find **all** values of i such that there is a match at shift i

Brute Force and Ignorance Algorithm

```
for i =  0 .. n - m :    # i is the shift
     match = True
     for j = 1 to m:
        if P[j] != T[i+j]:
          match = False
          break
     if match:
        print "Match found at shift ",i
```

This algorithm is obviously correct since it tests every possible shift. Its running time is clearly in O((n-m+1)*m), since each comparison of P with a substring of T takes O(m) time.

The worst-case is easily achieved : if T = “xxxxxx .... x” (ie every character in T is identical) and P = “xxxxy” then testing each shift will require m comparisons.

Rabin-Karp Algorithm

The Rabin-Karp algorithm is based on trying to eliminate the inner loop of the BFI algorithm, which takes O(m) time to test each shift. Rabin-Karp reduces this to O(1) to test each shift by using integer comparisons instead of string comparisons.

To do this, the search string P and each m-character substring of T must be converted to integers.

As an ongoing example domain for this unit we will look at DNA sequences ... all of which are composed of four basic building blocks commonly referred to by their initials: A, C, G, T. We will suppose that T is a large DNA sequence and P is a smaller one – we want to find all the occurrences of P in T.

In this domain it is very easy to convert P to an integer. We can assign different single digit integers to the four letters in our alphabet and just string them together. (When dealing with a larger character set we might choose to use the ASCII ordinal numbers of the characters.)

For example we can create a function that maps the letters onto digits like this:

x	f(x)
A	1
C	2
G	3
T	4

Converting P to an integer looks like this:

$$IP = f(P[1]) * 10^{m-1} + f(P[2]) * 10^{m-2} + ...f(P[m-1]) * 10 + f(P[m])$$

which is most efficiently computed using Horner's Rule:

```
def HR(P):
     IP = f(P[1])
     for j = 2 .. m:
          IP = IP*10 + f(P[j])
     return IP
```

Horner's Rule computes IP in O(m) time

Suppose T = “ACTTGGACTTATCTTGAG” and P = “CTTG”

Converting P to an integer using the method just shown gives IP = 2443. Clearly different values of P will give different values of IP.

For T, we need to convert each substring of length m into an integer. We can do that for each substring just before we see if it matches P

Our revised BFI algorithm would now look like this:

```
IP = HR(P)
for i =  0 .. n - m :    # i is the shift
     X = HR(T[i .. i+m])      # taking a slice of T a la Python
     if IP == X:
        print "Match found at shift ",i
```

This looks good – the inner loop is gone and since integer comparisons take constant time we seem to have acheived our goal. But looks can be deceiving! Each call to HR() takes O(m) time so we have really just replaced one O(m) loop with another and the complexity has not changed.

This is where Rabin and Karp start to show why they are famous! We can actually compute almost all the integers for substrings of T in O(1) time for each.

Here’s the idea. Consider the integer we get for the first shift (i = 0) in the example above: HR(“ACTT”) = 1244 . Now consider the integer for the second shift: HR(“CTTG”) = 2443. For the third shift it is HR(“TTGG”) = 4433. Each one is derived from the previous one by

- deleting the first digit
- shifting the remaining digits one column left
- adding the digit for the next letter in T

To delete the first digit, we just subtract that digit (which we get by converting the appropriate letter of T) multiplied by the appropriate power of 10 ... which a moment’s thought shows is 10^{m-1}

To shift the remainder one column left we just multiply it by 10

To add the digit for the next letter we just add it

These operations all take constant time ... so we can get the next integer we need in $O(1)$ time

To save time, we pre-compute 10^{m-1}. If we want to be super-efficient we can do this in $O(\log m)$ time using a divide and conquer algorithm, but because we’re only doing it once we can just use the obvious $O(m)$ method. I’ll call this value E.

So here's the algorithm to convert the integer for shift i to the integer for shift i+1:

```
# let X be the integer for shift i
X = (X-f(T[i+1])*E)*10 + f(T[i+m+1])
# X is now the integer for shift i+1
```

To do a quick check that all the index values in that expression are correct, let i = 0. In this case X represents the substring T[1 ... m] before the operation, and the substring T[2 ... m+1] after.

Plugging these changes into our algorithm we get

```
IP = HR(P)
E = 10^(m-1)
X = HR(T[1.. m])
for i =  0 .. n - m - 1: # i is the shift - notice we don't
                         # include the last shift here
     if IP == X:
        print "Match found at shift ",i
     # compute the integer for the next shift
     X = (X-f(T[i+1])*E)*10 + f(T[i+m+1])

if IP == X:
     print "Match found at shift ",n-m
```

(Point of interest – why don't we handle the test for the last shift inside the loop?)

The first three steps are all in $O(m)$. The loop executes $O(n-m)$ times and each iteration takes constant time. The last step takes $O(1)$ time. Assuming n is much larger than m (which is virtually always the case) these combine to give $O(n-m)$ time for the algorithm.

Mission Accomplished!

Except ... for ... one ... thing:

This algorithm's claim of fast running time is built around the idea that we can compare integers in constant time. That is actually true ONLY if the integers are $\leq$ whatever the maximum allowable integer happens to be in our computing environment. In Java for example, int variables must be $\leq 2,147,483,647$ and long integers must be $\leq 9,223,372,036,854,775,807$. So in Java there is no (native) way to represent a 20 digit integer, but if $m \geq 20$ then IP will have ≥ 20 digits. We can use a class such as BigInteger but this loses the constant time comparison operation.

This is where Rabin and Karp prove once again that they are pretty darn smart. They realized that we can side-step the excessively large integer problem by doing all the arithmetic mod q, where q is a prime number of suitable size (see below).

So our Horner's Rule method becomes

```
def HR(P):
     IP = f(P[1])
     for j = 2 .. m:
          IP = (IP*10 + f(P[j]))  mod q
     return IP
```

and our string matching algorithm becomes

```
IP = HR(P)
E = 10^(m-1) mod q
X = HR(T[1.. m])
for i =  0 .. n - m - 1: # i is the shift – notice we don't
                         # include the last shift here
     if IP == X:
        print "Match found at shift ",i
     # compute the integer for the next shift
     X = (((X-f(T[i+1])*E mod q)*10 ) mod q + f(T[i+m+1]) ) mod q

if IP == X:
     print "Match found at shift ",n-m
```

This keeps all the integers small enough that we can complete the arithmetic operations in O(1) time, but it introduces a new problem: when we compare IP to X , we may get equality even though the strings don't actually match. This is because when we do our arithmetic $\bmod q$ there are only q possible values. It is entirely possible that the integers for P and some substring of T will be congruent $\bmod q$ even if they are different – our algorithm will report this as a match. This means that whenever we get a potential match, we must check it in detail. The algorithm now looks like this:

```
IP = HR(P)
E = 10^(m-1) mod q
X = HR(T[1.. m])
for i =  0 .. n - m - 1: # i is the shift – notice we don’t
                         # include the last shift here
     if IP == X:
        match = True     # check for valid match
        for j = 1 .. m:
          if P[j] != T[i+j]:
               match = False
               break
        if match:
          print ”Match found at shift ”,i
     # compute the integer for the next shift
     X = (((X-f(T[i+1])*E mod q)*10 ) mod q + f(T[i+m+1]) ) mod q

if IP == X:
     match = True        # check for valid match
     for j = 1 .. m:
          if P[j] != T[n – m + j] :
               match = False
               break
     if match:
          print “Match found at shift “,n-m
```

You will perhaps be relieved to know that this is our final form of the Rabin-Karp algorithm – but we need to examine its running time.

To do this I’m going to introduce the term “might-match” to describe a shift where `IP == X` - ie a shift for which we have to do the character-by-character comparison to confirm or reject the match.

Some of the might-matches are valid matches, and the rest are false positives.

Clearly if every shift is a might-match, this algorithm has exactly the same complexity as the BFI algorithm. However we can argue that the number of might-matches is probably much smaller than this.

Let v be the number of valid matches - ie the number of times P actually occurs in T. Clearly the number of might-matches is $\geq v$.

With some amount of hand-waving about getting a uniform distribution of remainders when we divide random integers by q , we can claim that the probability of each X equalling IP is $\frac{1}{q}$. Since there are no more than n shifts, the expected number of false positives is $\leq \frac{n}{q}$.

Combining these, we can claim that the expected number of might-matches is $\leq v + \frac{n}{q}$

Since the loop executes O(n) times, and each might-match takes O(m) time to verify, we can see that the expected running time for the loop is in $O(n) + O(m * (v + \frac{n}{q}))$

Now if $q > m$ (which is not usually a challenge - m is the length of P, and even if P is thousands of characters long we have a good selection of primes to choose from), this reduces to $O(n) + O(m * v + n)$

Furthermore, it is not unreasonable to assume that v is likely to be small, possibly even in $O(1)$. If so, $m * v$ is in $O(m)$, and since $m \leq n$, the whole algorithm has running time in $O(n)$

Thus under some reasonable assumptions, Rabin-Karp has expected running time in $O(n)$