String Matching

Pattern Matching

- Given a text string T[0..n-1] and a pattern P[0..m-1], find all occurrences of the pattern within the text.

- Example: T = 000010001010001 and P = 0001, the occurrences are:
  - first occurrence starts at T[1]
  - second occurrence starts at T[5]
  - third occurrence starts at T[11]

Naïve algorithm

```
for (s = 0; s <= n-m; s++)
    if P[0..m-1] equal to T[s..s+m-1]
        output s;
```

Example:

```
          T   0 0 0 0 1 0 0 0 1 0 1 0 0 0 1
s=0           0 0 0 1
                    ^mismatch

s=1             0 0 0 1
                      ^match

s=2               0 0 0 1
                      ^mismatch

s=3                 0 0 0 1
                      ^mismatch

s=4                   0 0 0 1
                      ^mismatch

s=5                     0 0 0 1
                              ^match
```

Worst-case running time = O(nm).

Rabin-Karp Algorithm

- Key idea:
  - think of the pattern P[0..m-1] as a key, transform (hash) it into an equivalent integer *p*
  - Similarly, we transform substrings in the text string T[] into integers
    - For s=0,1,…,n-m, transform T[s..s+m-1] to an equivalent integer t_s
  - The pattern occurs at position s if and only if $p=t_s$
- If we compute p and t_s quickly, then the pattern matching problem is reduced to comparing p with n-m+1 integers

Rabin-Karp Algorithm …

- How to compute p?

  $p = 2^{m-1} P[0] + 2^{m-2} P[1] + \ldots + 2 P[m-2] + P[m-1]$

- Using horner's rule

$$p = P[m-1] + 2*(P[m-2] + 2*(P[m-3] + \ldots 2*(P[1] + 2*P[0]) \ldots).$$

```
p = 0;
for (i = 0; i < m; i++)
      p = 2*p + P[i];
```

This takes O(m) time, assuming each arithmetic operation can be done in O(1) time.

Rabin-Karp Algorithm …

- Similarly, to compute the (n-m+1) integers t_s from the text string

```
for (s = 0; s <= n-m; s++) {
    t[s] = 0;
    for (i = 0; i < m; i++)
        t[s] = 2*t[s] + T[s+i];
}
```

- This takes O((n – m + 1) m) time, assuming that each arithmetic operation can be done in O(1) time.
- This is a bit time-consuming.

Rabin-Karp Algorithm

- A better method to compute the integers is:

```
t[0] = 0;
offset = 1;
for (i = 0; i < m; i++)
    offset = 2*offset;
for (i = 0; i < m; i++)
    t[0] = 2*t[0] + T[i];
for (s = 1; s <= n-m; s++)
    t[s] = 2*(t[s-1] - offset*T[s-1]) + T[s+m-1];
```

This takes O(n+m) time, assuming that each arithmetic operation can be done in O(1) time.

Problem

- The problem with the previous strategy is that when m is large, it is unreasonable to assume that each arithmetic operation can be done in O(1) time.
  - In fact, given a very long integer, we may not even be able to use the default integer type to represent it.
- Therefore, we will use modulo arithmetic. Let q be a prime number so that 2q can be stored in one computer word.
  - This makes sure that all computations can be done using single-precision arithmetic.

```
p = 0;
for (i = 0; i < m; i++)
    p = (2*p + P[i]) % q;


t[0] = 0;
offset = 1;
for (i = 0; i < m; i++)
    offset = 2*offset % q;
for (i = 0; i < m; i++)
    t[0] = (2*t[0] + T[i]) % q;
for (s = 1; s <= n-m; s++)
    t[s] = (2*( t[s-1] - offset*T[s-1]) + T[s+m-1]) % q;
```

- Once we use the modulo arithmetic, when $p=t_s$ for some s, we can no longer be sure that P[0 .. M-1] is equal to T[s .. S+ m -1]
- Therefore, after the equality test $p = t_s$, we should compare P[0..m-1] with T[s..s+m-1] character by character to ensure that we really have a match.
- So the worst-case running time becomes O(nm), but it avoids a lot of unnecessary string matchings in practice.