Introduction to String Matching

String and pattern matching problems are fundamental to any computer application involving text processing. A very basic but important string matching problem, variants of which arise in finding similar DNA or protein sequences, is as follows. Given a *text* $T[1,\ldots,n]$ (n characters) and a *pattern* $P[1,\ldots,m]$ (both of which are strings over the same alphabet), find all occurrences of P in T. We say that P *occurs in* T *with shift* s if $P[1,\ldots,m] = T[s+1,\ldots,s+m]$. A simple algorithm simply considers all possible shifts:

algorithm Simple-Pattern-Finding($P[1,\ldots,m], T[1,\ldots,n]$)
input: pattern P of length m and text T of length n
preconditions: $1 \leq m \leq n$
output: list of all numbers s, such that P occurs with shift s in T

for $s \leftarrow 0$ to $n-m$
{
if ($P[1,\ldots,m] == T[s+1,\ldots,s+m]$) { output s }
}

The total number of comparisons done by this algorithm is $(n-m+1)m = \Theta(nm)$. Any algorithm for the string matching problem must examine every symbol in T and P, and so requires $\Omega(n+m)$ time. We'd like to find an algorithm that can match this lower bound.

Exercises: Suppose for both of these problems that, at each possible shift (iteration of the **for** loop), characters are compared from left to right ONLY until a mismatch is found.

1. Will the algorithm still take $\Theta(mn)$ time?
2. If the pattern and text are chosen uniformly at random over an alphabet of size k, what is the *expected* time for the algorithm to finish?

The Knuth-Morris-Pratt (KMP) algorithm

We next describe a more efficient algorithm, published by Donald E. Knuth, James H. Morris and Vaughan R. Pratt, 1977 in: "Fast Pattern Matching in Strings." In SIAM Journal on Computing, 6(2): 323–350. To illustrate the ideas of the algorithm, we consider the following example:

$$T = \text{xyxxyxyxyyxyxyxyyxyxyxxy}$$

and

$$P = \text{xyxyyxyxyxx}$$

At a high level, the KMP algorithm is similar to the naive algorithm: it considers shifts in order from 1 to $n-m$, and determines if the pattern matches at that shift. The difference is that the KMP algorithm uses information gleaned from partial matches of the pattern and text to skip over shifts that are guaranteed not to result in a match.

Suppose that, starting with the pattern aligned underneath the text at the leftmost end, we repeatedly "slide" the pattern to the right and attempt to match it with the text. Let's look at some examples of how sliding can be done. The text and pattern are included in Figure 1, with numbering, to make it easier to follow.

1. Consider the situation when $P[1,\ldots,3]$ is successfully matched with $T[1,\ldots,3]$. We then find a mismatch: $P[4] \neq T[4]$. Based on our knowledge that $P[1,\ldots,3] = T[1,\ldots,3]$, *and ignoring symbols of the pattern and text after position 3*, what can we deduce about where a potential match might be? In this case, the algorithm slides the pattern 2 positions to the right so that $P[1]$ is lined up with $T[3]$. The next comparison is between $P[2]$ and $T[4]$.

2. Since $P[2] \neq T[4]$, the pattern slides to the right again, so that the next comparison is between $P[1]$ and $T[4]$.

3. At a later point, $P[1,\ldots,10]$ is matched with $T[6,\ldots,15]$. Then a mismatch is discovered: P[11] $\neq$ T[16]. Based on the fact that we know $T[6,\ldots,15] = P[1,\ldots,10]$ (and ignoring symbols of the pattern after position 10 and symbols of the text after position 15), we can tell that the first possible shift that might result in a match is 12. Therefore, we will slide the pattern right, and next ask whether $P[1,\ldots,11] = T[13,\ldots,23]$. Thus, the next comparisons done are $P[4] == T[16]$, $P[5] == T[17]$, $P[6] == T[18]$ and so on, as long as matches are found.

Sliding rule

We need to make precise exactly how to implement the sliding rule. The following notation is useful. Let $S = s_1 s_2 \ldots s_k$ be a string. Each string of the form $s_1 \ldots s_i, 1 \leq i \leq k$ is called a *prefix* of s. Also, we define the empty string (containing no symbols) to be a prefix of s. A prefix s' of s is a *proper* prefix if $s' \neq s$. Similarly, each string of the form $s_i \ldots s_k, 1 \leq i \leq k$ is called a *suffix* of s. Also, the empty string (containing no symbols) is a suffix of s. A suffix s' of s is a proper suffix if $s' \neq s$.

Suppose that $P[1,\ldots,q]$ is matched with the text $T[i-q+1,\ldots,i]$ and a mismatch then occurs: $P[q+1] \neq T[i+1]$. Then, slide the pattern right so that the longest possible proper prefix of $P[1,\ldots,q]$ that is also a suffix of $P[1,\ldots,q]$ is now aligned with the text, with the last symbol of this prefix aligned at $T[i]$. If $\pi(q)$ is the number such that $P[1,\ldots,\pi(q)]$ is

P:	x	y	x	y	y	x	y	x	y	x	x
q:	1	2	3	4	5	6	7	8	9	10	11
$\pi(q)$:	0	0	1	2	0					3	

Table 1: Table of π values for pattern P.

the longest proper prefix that is also a suffix of $P[1, \ldots, q]$, then the pattern slides so that $P[1, \ldots, \pi(q)]$ is aligned with $T[i - \pi(q) + 1, \ldots, i]$.

The KMP algorithm precomputes the values $\pi(q)$ and stores them in a table $\pi[1, \ldots, m]$. We will explain later how this is done. Some of the values $\pi(q)$ for our example are given in Table 1. Can you figure out what are the remaining values?

In summary, a "step" of the KMP algorithm makes progress in one of two ways. Before the step, suppose that $P[1, \ldots, q]$ is already matched with $T[i - q + 1, \ldots, i]$.

- If $P[q+1] = T[i+1]$, the length of the match is extended, unless $q+1 = m$, in which case we have found a complete match of the pattern in the text.
- If $P[q + 1] \neq T[i + 1]$, the pattern slides to the right.

In either case, progress is made. The algorithm repeats such steps of progress until the end of the text is reached. Pseudocode for the KMP algorithm is given in Algorithm 1.

Running Time

Each time through the loop, either we increase i or we slide the pattern right. Both of these events can occur at most n times, and so the repeat loop is executed at most $2n$ times. The cost of each iteration of the repeat loop is $O(1)$. Therefore, the running time is $O(n)$, assuming that the values $\pi(q)$ are already computed.

Computing the values $\pi(q)$

We now describe how to compute the table $\pi[1, \ldots, m]$. Note that $\pi(1)$ is always equal to 0. Suppose that we have computed $\pi[1, \ldots, i]$ and we want to compute $\pi(i + 1)$. Initially we know $P[1, \ldots, \pi(i)]$ is the longest proper prefix of $P[1, \ldots, i]$ that is also a suffix of $P[1, \ldots, i]$. Let $q = \pi(i)$. If $P[i + 1] = P[q + 1]$ then it must be that $\pi(i + 1) = q + 1$. Otherwise, we set $q = \pi(q)$ and repeat the test again. We continue until $q = 0$, at which point we just set $\pi(i + 1) = 0$. Some pseudocode is given in Algorithm 2; you should fill in the rest as an exercise. Algorithm 2 runs in linear time for much the same reason as does the KMP algorithm. Therefore, the whole KMP algorithm runs in time $O(n + m)$, which is much better than the simple quadratic time algorithm.

Exercise:

```
algorithm KMP(P[1,...,m], T[1,...,n])
    input:          pattern P of length m and text T of length n
    preconditions:  1 ≤ m ≤ n
    output:         list of all numbers s, such that P occurs with shift s in T

    q ← 0;
    i ← 0;
    while (i < n) /* P[1,...,q] == T[i − q + 1,...,i]
    {
      if (P[q + 1] == T[i + 1])
      {
        q ← q + 1;
        i ← i + 1;
        if (q == m)
        {
          output i − q;
          q ← π(q); /*slide the pattern to the right
        }
      }
      else /* a mismatch occurred
      {
        if (q == 0) { i ← i + 1 }
        else { q ← π(q) }
      }
    }
```

Algorithm 1: KMP algorithm.

1. Construct a pattern of length 10, over the alphabet $\{x, y\}$, such that the number of iterations of the while loop of Algorithm 2, when $i = 10$, is as large as possible.

2. Suppose that the pattern P and the text T are strings over an alphabet of size 2. In this case, if you know that $P[q+1] \neq T[i+1]$, it is possible to tell by looking only at the pattern (specifically at $P[q+1]$), what is the symbol of the text at position $i+1$. Such knowledge could be used (in some cases) to increase the amount by which the pattern slides, thus speeding up the algorithm. How might you change the algorithm to take advantage of this? How does this affect the amount of memory needed by the algorithm?

algorithm Compute-π-values($P[1, \ldots, m]$)
input: pattern P of length m
preconditions: $1 \leq m$
output: table $\pi[1, \ldots, m]$

$\pi(1) \leftarrow 0$;
for ($i \leftarrow 1$ to $m - 1$)
/* $\pi[1, \ldots, i]$ is already calculated; calculate $\pi[i + 1]$
{

}

Algorithm 2: Algorithm to compute the π values. Can you fill in the details?

Appendix: review of Big-Oh notation

Table 2 summarizes big-oh notation, which we use to describe the asymptotic running time of algorithms.

We say that $f(n)$ is:	Mean that $f(n)$ grows:	Write:	If:
little-oh of $g(n)$	more slowly than $g(n)$	$f(n) = o(g(n))$	$\lim_{n\to\infty} f(n)/g(n) = 0$
big-oh of $g(n)$	no faster than $g(n)$	$f(n) = O(g(n))$	there exist some c, $n_0 > 0$: for all $n > n_0, f(n) <= cg(n)$
theta of $g(n)$	about as fast as $g(n)$	$f(n) = \Theta(g(n))$	$f(n) = O(g(n))$ and $g(n) = O(f(n))$
approximately equal to $g(n)$	as fast as $g(n)$	$f(n) \approx g(n)$	$\lim_{n\to\infty} f(n)/g(n) = 1$
omega of $g(n)$	no slower than $g(n)$	$f(n) = \Omega(g(n))$	$g(n) = O(f(n))$

Table 2: Summary of big-oh notation.

T:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
	x	y	x	x	y	x	y	x	y	y	x	y	x	y	x	y	y	x	y	x	y	x	x	y

P:	1	2	3	4	5	6	7	8	9	10	11
	x	y	x	y	y	x	y	x	y	x	x

Figure 1: Text and pattern used in our examples, with characters numbered.