Advanced Data Structures and Algorithms
Exercise Sheet 7

Stuart Golodetz

April 8, 2006

1. (a)

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$\pi(n)$	0	0	1	2	0	1	2	0	1	2	0	1	2	3	4	5	6	7	8

(b) From the lecture notes, we know that $\pi^*[q]$ consists of the lengths of all proper subsequences of P_q that are both prefixes and suffixes. Now there are exactly q proper prefixes of P_q, namely P_0, P_1, ..., P_{q-1}. Even if all of those prefixes are suffixes as well, there can't be more than q of them. So:

$$|\pi^*[q]| \leq q$$

To show that this bound is tight, consider the example $P_q = \underbrace{a \cdots a}_{q\ characters}$. Clearly $P_k \sqsubseteq_R P_q$ for all $0 \leq k < q$, so the size of the orbit of q is q in this instance.

(c) If we can find all substrings of $P \cdot T$ that end with P then we can work out where the pattern matches are. In particular, if we know that $P \sqsubseteq_R (P \cdot T)_q$, then we can deduce that there's a match at $q - m$.

Now, if we have the π function for the string $P \cdot T$, then we can also easily work out the corresponding π^* function (note that we don't have to calculate it explicitly, we can just work backwards through the values of the π function using the recursive definition of π^*). We know that if (for some q), $m \in \pi^*(q)$, then $(P \cdot T)_m \sqsubseteq_R (P \cdot T)_q$, i.e. $P \sqsubseteq_R (P \cdot T)_q$. Thus the occurrences of P in the text are given by the set $\{q - m \mid m \in \pi^*(q) \wedge q \geq 2m\}$. Note that the $q \geq 2m$ condition is there to prevent $q - m$ being less than m. (This would be important if we had, say, $P = aaa$ and $T = a$.)

This might all be somewhat clearer with an example. Consider what happens if we have $P = aba$ and $T = bababa$. Then $P \cdot T = abababab a$ and the π and π^* functions for it are:

n	1	2	3	4	5	6	7	8	9
$\pi(n)$	0	0	1	2	3	4	5	6	7
$\pi^*(n)$	{0}	{0}	{0,1}	{0,2}	{0,1,3}	{0,2,4}	{0,1,3,5}	{0,2,4,6}	{0,1,3,5,7}

The set of occurrences we're after is $\{q - 3 \mid 3 \in \pi^*(q) \wedge q \geq 6\} = \{4, 6\}$.

2. (a) If all the characters in P are distinct, then whenever we see (for some i and j) that $t_i = p_j$, we know for certain that if t_i is part of a match then it matches up with p_j. This gives rise to an efficient linear-time algorithm as follows:

```
DISTINCT-PATTERN-MATCHER(T,P)
  n <- length(T)
  m <- length(P)
  count <- 0
  for i = 0 to n - 1 do
    ▷ Invariant: T[i-count..i) = P[0..count) && count < m
    if T[i] = P[count]
    then count <- count + 1
         if count = m
         then print "Match at position " + (i + 1 - count)
              count <- 0
    else if count > 0
         then count <- 0
              i <- i - 1    ▷ Next time compare T[i] with P[0]
```

In English, all this does is walk along the letters in the text, trying to match them with the pattern. If we've matched m letters in a row, we've found a match with the pattern: we output the position of the match and reset the count in order to look for more matches. If the current character doesn't match, then either (a) we carry on without doing anything extra if we haven't matched any characters so far or (b) if we have matched characters, we reset the count and try matching the current character again with the start of the pattern on the next iteration.

(b) This is pretty much straightforward, as it turns out. If $T = t_0t_1...t_{n-1}$, then the rotation problem is equivalent to pattern matching P against $T' = t_0t_1...t_{n-1}t_0t_1...t_{n-2}$. For example, checking whether ABCD is a rotation of CDAB simply involves pattern matching ABCD against CDABCDA, since if it's a rotation of CDAB then it must match with one of CDAB, DABC, ABCD or BCDA (i.e. one of the four-letter contiguous substrings of CDABCDA). Now, the length of T' is $2n - 1$, so the pattern matching can be done in $O(2n - 1 + m) = O(2n - 1 + n) = O(n)$ steps. The only remaining thing to note is that it clearly doesn't take more than linear time to add the extra characters onto the end of T: in fact, we can avoid adding the extra characters on at all by thinking of T' as an abstract array (only T itself is actually in memory) and ensuring that accesses to T' are simply forwarded to T. This is trivial to do: we just add a layer of indirection so that $T'[k]$ actually refers to $T[k \bmod n]$. Thus 'generating' the text T' to match against is $O(1)$ in practice.

3. (a) The Rabin-Karp string-matching algorithm is based on two main ideas:

   i. If, instead of comparing the pattern (of length m) to each contiguous text block of length m, we were to compare hashes of the same (using some as yet unspecified hash function), then we might be able to rule out a lot of blocks fairly cheaply (comparing integers is much faster than comparing strings of length m). We know we can rule out a block if its hash and that of the pattern don't match, since if the block and the pattern were equal then their hashes would certainly match. We can't, on the other hand, conclude that if a block's hash and that of the pattern are the same, then the two are equal: it's quite possible for non-equal things to hash to the same value (the hash function isn't necessarily injective). What we can do, though, is check all such cases using the naive algorithm (i.e. if a block's hash equals the pattern's hash, we compare it with the pattern manually, otherwise we skip it). Provided we get relatively few potential matches (i.e. relatively few blocks whose hashes equal the pattern's), this should be much faster than the naive algorithm.

   ii. To make this (potential) improvement, we clearly need hashes of all the various blocks (as well as the pattern itself). This would appear to be a stumbling block, but observe that we can devise a hash function such that the hash of a block can be computed in constant time from the hash of its predecessor: the hash function in question just involves viewing the text as a number in base α (modulo some number p): for details, see the lecture notes! Now we can compute the hash of the first block in $O(m)$ time, where m is the length of the pattern, and we can compute the hash of the pattern in $O(m)$ time also. So we can compute all the hashes in $O(m + n)$ time.

   The algorithm itself basically runs through all potential shifts of the pattern against the text (i.e. 0 through $n - m$) and does the following:

   - Compare the hash of the current block against that of the pattern.
   - If it's not equal, compute the next hash and carry on to the next shift.
   - If it is equal, compare the block and the pattern character-by-character. If they're equal, output something.

(b) No, it's not a great idea. Consider that $x \bmod 2^{14}$ just gives us the 14 least significant bits of x. In other words, our hash of a (potentially long) piece of text depends on no more than two of its characters. Consider that if

$$x = x_0 + x_1\alpha + x_2\alpha^2 + ... + x_n\alpha^n = x_0 + 2^8 x_1 + 2^{16} x_2 + ... + 2^{8n} x_n$$

then

$$x \bmod 2^{14} = (x_0 + 2^8 x_1) \bmod 2^{14}$$

(c) No, it's not always reasonable to assume that. One point which could be made (though possibly not the one the question's looking for) is that, since using $p = 2^k$ for some k gives us the k least significant bits of $t_{s+1} \cdots t_{s+m}$, the assumption would only hold if the probability distribution on text segments made all low-order k-bit patterns equally likely. This is unlikely to be the case for, say, a regular piece of ASCII text, which will in all likelihood contain a preponderence of alphanumeric characters (consider this sentence, for example). Thus using any power of 2 as the modulus would tend to cause the assumption not to hold.

4. (a) We observe that (by the equation given for calculating h_{s+1} from h_s):

$$h_s \equiv \alpha h_{s-1} + cT[s] + T[s+m] \bmod p$$

So:

$$\alpha h_{s-1} \equiv h_s - cT[s] - T[s+m] \bmod p$$

Thus far, the value of p didn't make any difference. Now, however, we need to multiply both sides by $\alpha^{-1} \bmod p$ (i.e. the multiplicative inverse of α, modulo p), which only exists if $gcd(\alpha, p) = 1$. Assuming that this is the case, we derive that:

$$h_{s-1} \equiv \alpha^{-1} (h_s - cT[s] - T[s+m]) \bmod p$$

(b) Once we can calculate h_{s-1} from h_s in constant time, actually modifying the algorithm so that it can search in either direction is easy.

Assume we're maintaining the following in memory: T, P, n, m, α, c, $inv = \alpha^{-1}$, h_*, s (the current position) and h (the hash at the current position). We provide functions as follows:

```
MOVE-TO-START(start)
  while s $\neq$ start
    do if s > start then PREV-SHIFT
       else NEXT-SHIFT

NEXT-SHIFT
  h <- ($\alpha$h + cT[s+1] + T[s+m+1]) mod p
  s <- s + 1

PREV-SHIFT
  h <- inv * (h - cT[s] - T[s+m]) mod p
  s <- s - 1
```

```
RK-BACK(start)
  MOVE-TO-START(start)

  while s >= 0 do
    if h_* = h
    then if P[1..m] = T[s+1..s+m]
         then return s
    if s > 0 then PREV-SHIFT

  return -1

RK-FORWARD(start)
  MOVE-TO-START(start)

  while s <= n - m do
    if h_* = h
    then if P[1..m] = T[s+1..s+m]
         then return s
    if s < n - m then NEXT-SHIFT

  return -1
```

The user uses RK-BACK(start) to search backwards starting from the given start position and RK-FORWARD(start) to search forwards from the given start position. Both functions return the position of the first match found, or -1 if no match was found (in that particular direction from the given starting point).

(c) It's not necessary for p to be prime, only for it to be relatively prime to α. Any prime greater than α certainly satisfies that condition, of course.