Solution to Homework 10

Problem 1 (CLRS 32.1-2). (1 point) Suppose that all characters in the pattern $P$ are different. Show how to accelerate Naive-String-Matcher to run in time $O(n)$ on an $n$-character text $T$.

Solution: A character mismatch $P[i] \neq T[s+i]$ for $i > 1$ indicates that characters in $P[1..i)$ and $T[s+1..s+i)$ matched successfully. As all characters in $P$ are distinct, this partial match means that only $P[1] = T[s+1]$ and thus none of $T(s+1..s+i)$ could match $P[1]$ and start a new potentially valid match. Taking advantage of this fact, our algorithm can skip to character $T[s+i]$ – the first character that can potentially match $P[1]$:

```plaintext
DISTINCT-CHARS-PATTERN-MATCHER(T, P)
1 n = T.length
2 m = P.length
3 s = 0
4 while s ≤ n - m
5   i = 1
6   while i ≤ m and P[i] = T[s + i]
7     i = i + 1
8   if i = m + 1
9      print "Pattern occurs with shift" s
10   s = max(s + 1, s + i - 1)
```

Problem 2 (CLRS 32.1-4). (2 points) Suppose we allow the pattern $P$ to contain occurrences of a gap character ♦ that can match an arbitrary string of characters (even one of zero length). For example, the pattern $ab♦ba♦c$ occurs in the text $cabccbacab$ as $cabccbacab$ and as $cabccbacab$.

Note that the gap character may occur an arbitrary number of times in the pattern but not at all in the text. Give a polynomial-time algorithm to determine whether such a pattern $P$ occurs in a given text $T$, and analyze the running time of your algorithm.

Solution: We start with a simpler problem of determining whether the entire $T$ matches $P$.

Let us define $match[i, j]$ to be true if $T_i$ matches $P_j$, and false otherwise. Then:

- $match[0, 0] = true$, as an empty text matches an empty pattern.
- $match[0, j] = match[0, j - 1]$ if $P[j] = ♦$ for $1 \leq j \leq m$, as an empty text matches ♦ as long as the previous characters match.
- If $P[j] = ♦$, we can either treat ♦ as an empty string and skip it: $match[i, j] = match[i, j - 1]$, or assume it matched the last character of $T_i$: $match[i, j] = match[i - 1, j]$.
• If $P[j] = T[i]$, the problem reduces to matching $T_{i-1}$ and $P_{j-1}$: $\text{match}[i, j] = \text{match}[i - 1, j - 1]$.

• $\text{match}[i, j] = \text{FALSE}$ in all other cases.

This allows us to formulate the following recursive definition:

$$
\text{match}[i, j] = \begin{cases} 
\text{TRUE} & \text{if } i = 0 \text{ and } j = 0, \\
\text{match}[0, j - 1] & \text{if } i = 0 \text{ and } P[j] = \diamond, \\
\text{match}[i, j - 1] \lor \text{match}[i - 1, j] & \text{if } P[j] = \diamond, \\
\text{match}[i - 1, j - 1] & \text{if } P[j] = T[i], \\
\text{FALSE} & \text{otherwise.}
\end{cases}
$$

And an associated dynamic programming algorithm:

MATCH-WITH-GAPS($T, P$)

1. $n = T\. \text{length}$
2. $m = P\. \text{length}$
3. let $\text{match}[0..n, 0..m]$ be a new array initialized to $\text{FALSE}$
4. $\text{match}[0, 0] = \text{TRUE}$
5. for $j = 1$ to $m$
6.   if $P[j] = \diamond$
7.     $\text{match}[i, j] = \text{match}[0, j - 1]$
8. for $i = 1$ to $n$
9.   for $j = 1$ to $m$
10.   if $P[j] = \diamond$
11.     $\text{match}[i, j] = \text{match}[i, j - 1] \lor \text{match}[i - 1, j]$
12.   elseif $P[j] = T[i]$
13.     $\text{match}[i, j] = \text{match}[i - 1, j - 1]$
14.   else
15.     $\text{match}[i, j] = \text{FALSE}$
16. return $\text{match}[n, m]$

The algorithm fills an $n \times m$ table, spending constant time on each cell, so the running time and space are both $\Theta(nm)$ (we note that faster algorithms are possible).

We can solve the original problem of finding $P$ anywhere in $T$ by calling MATCH-WITH-GAPS with $P' = \diamond P \diamond$.

Problem 3 (CLRS 32.2-1). (1 point) Working modulo $q = 11$, how many spurious hits does the Rabin-Karp matcher encounter in the text $T = 3141592653589793$ when looking for the pattern $P = 26$?

Solution: A spurious hit occurs when $t_s = p \mod q = 26 \mod 11 = 4$, but $s$ is not a valid shift. This happens three times for the given input: for $t_3 = 15$, $t_4 = 59$, and $t_5 = 92$. $t_6 = 26$ indicates a valid shift. (Values given before mod 11.)
Problem 4 (CLRS 32.3-1). (1 point) Construct the string-matching automaton for the pattern $P = aabab$ and illustrate its operation on the text string $T = aaababaababaababaab$.

Solution: Applying the DFA construction method from section 32.3 with $P = aabab$ and $\Sigma = \{a, b\}$ gives the following transition table:

<table>
<thead>
<tr>
<th>state</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

With the following sequence of state transitions for $T = aaababaababaababaab$:

- $a : 0 \rightarrow 1$
- $a : 1 \rightarrow 2$
- $a : 2 \rightarrow 2$
- $b : 2 \rightarrow 3$
- $a : 3 \rightarrow 4$
- $b : 4 \rightarrow 5$ (match)
- $a : 5 \rightarrow 1$
- $a : 1 \rightarrow 2$
- $b : 2 \rightarrow 3$
- $a : 3 \rightarrow 4$
- $a : 4 \rightarrow 2$
- $b : 2 \rightarrow 3$
- $a : 3 \rightarrow 4$
- $b : 4 \rightarrow 5$ (match)
- $a : 5 \rightarrow 1$
- $a : 1 \rightarrow 2$
- $b : 2 \rightarrow 3$

Problem 5 (CLRS 32.3-3). (1 point) We call a pattern $P$ nonoverlappable if $P_k \sqsubseteq P_q$ implies $k = 0$ or $k = q$. Describe the state-transition diagram of the string-matching automaton for a nonoverlappable pattern.

Solution: Recall that the state number indicates the number of successfully matched characters from $P$. On each transition, the state number can remain the same, be increased by 1 (successful match), decreased to a non-zero value (partial regress), or decreased to zero (complete regress). For a nonoverlappable pattern, remaining the same, being increased by 1, and being decreased to zero are valid options. Partial regress, however, is only possible to state 1 (in the case of two adjacent two pattern occurrences).
Problem 6 (CLRS 32.4-1). (1 point) Compute the prefix function \( \pi \) for the pattern \( ababbabababababb \).

Solution: Following the prefix function computing method from section 32.4 gives:

| \( i \) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| \( P[i] \) | a | b | a | b | b | a | b | b | a | b | b | a | b | b | a | b | b | a | b |
| \( \pi[i] \) | 0 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  |

Problem 7 (CLRS 32.4-7). (1 point) Give a linear-time algorithm to determine whether a text \( T \) is a cyclic rotation of another string \( T' \). For example, arc and car are cyclic rotations of each other.

Solution: A few options:

- Observe that \( T \) is a cyclic rotation of \( T' \) if and only if \( T \) is a substring of \( T'T' \), use a linear time pattern matching algorithm, such as KMP.
- Use Booth's \( O(n) \) algorithm to compute lexicographically minimal string rotations of \( T \) and \( T' \), compare for equality. See https://en.wikipedia.org/wiki/Lexicographically_minimal_string_rotation.
- Modify a linear time pattern matching algorithm, such as KMP, to "wrap around" when the end of \( T \) is reached, match \( T \) against \( T' \).

Problem 8. (3 points) The longest palindromic substring is a maximum-length contiguous sub-string of a given string that is a palindrome. For example, the longest palindromic substring of ultramarine is ramar.

Give an efficient algorithm to determine the longest palindromic substring of a given string. Explain the algorithm and illustrate its operation on the string evenness.

Solution: What follows is a linear time algorithm for finding the longest palindromic substring, known as Manacher’s algorithm. The pseudocode and explanation are based on https://en.wikipedia.org/wiki/Longest_palindromic_substring and http://articles.leetcode.com/longest-palindromic-substring-part-ii.

We begin by making the following observations:

- It is convenient to refer to palindromes in terms of their center and length, instead of start and end positions.
- Palindromes of even length are centered at the empty string between characters, and it is convenient to view such empty strings as characters in their own right. We use \# to represent these special characters.
- Let us define \( P \) as an array of palindrome lengths, where \( P[i] = k \) indicates the existence of a length \( k \) palindrome centered at position \( i \). The problem can now be reduced to computing \( P \), finding the maximum element, and reconstructing the actual palindrome.
• We expect values in $P$ to exhibit symmetry around a given $P[i]$, as $P[i] = k$ indicates that $[k/2]$ characters to the left of $i$, character at $i$, and $[k/2]$ characters to the right of $i$ form a palindrome. For example, having computed $P[0..5]$ for $S = ababa$:

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>#</td>
<td>a</td>
<td>#</td>
<td>b</td>
<td>#</td>
<td>a</td>
<td>#</td>
<td>b</td>
<td>#</td>
<td>a #</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


• To utilize the symmetry property correctly, we have to consider not only $P[i]$ around which we mirror the values, but also each $P[j]$ we are mirroring: if the palindrome centered at $P[j]$ extends past the palindrome centered at $P[i]$, we cannot rely on the symmetry property and have to expand palindrome character by character. See http://articles.leetcode.com/longest-palindromic-substring-part-ii for a good visual explanation.

Observations above lead to the following linear time algorithm for finding the longest palindromic substring:

**MANACHER($T$)**

1. $S = \text{INSERT-SENTINELS}(T)$
2. $n = S.length$
3. let $P[0..n]$ be a new array initialized to 0
4. // Current palindrome's center and the right boundary respectively.
5. $center = 0$
6. $right = 0$
7. for $i = 1$ to $n$
8.   $mirror = 2*center - i$
9.   if $right > i$
10.   // Can use the symmetry property.
11.     $P[i] = \text{min}(right - i, P[mirror])$
12.   // Attempt to expand current palindrome character by character.
13.     while $S[i + P[i] + 1] = S[i - P[i] - 1]$
14.         $P[i] = P[i] + 1$
15.     // Adjust center and right boundary if we went past current ones.
16.     $\text{newRight} = i + P[i]$
17.     if $\text{newRight} > right$
18.         $center = i$
19.         $right = \text{newRight}$
20. $\text{PRINT-LONGEST}(T, P)$

Executed for string evenness, the algorithm produces the following array of lengths:

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>#</td>
<td>e</td>
<td>#</td>
<td>v</td>
<td>#</td>
<td>e</td>
<td>#</td>
<td>n</td>
<td>#</td>
<td>n</td>
<td>#</td>
<td>e</td>
<td>#</td>
<td>s</td>
<td>#</td>
<td>s</td>
<td>#</td>
</tr>
<tr>
<td>P</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>